Software Framework for Reconfigurable Distributed System on AAUSAT3
Title:
Software Framework for
Reconfigurable Distributed System on AAUSAT3

Theme:
Networks and Distributed Processing

Project period:
7th Semester, Fall 2008

Project group:
08gr724@space.aau.dk

Group members:
Jesper Bønding
Kasper Fulgsang Jensen
Marc Pessans-Goyheneix
Morten Burchard Tychsen
Kasper Vinther

Supervisor:
Jens Frederik Dalsgaard Nielsen

Copies: 8

Pages: 112

Finished 16/12/2008

Synopsis:
A distributed system has been proposed for the student satellite AAUSAT3, in order to improve redundancy, reliability and to ease application development and debugging. A suitable software framework is needed for the processing units to ensure cooperation between applications, and to set the ground for reconfiguration of software in space.

This collection of worksheets presents the work on such a software framework, which involved selection and test of kernel, development of a protocol stack and software update possibilities through a bootloader system. The focus has been put on robustness, reconfigurability, testability and ease of application development.

Tests have shown that the chosen processing unit for AAUSAT3 is on the boundary of what is needed. A small test program used almost 85% of the available RAM, without allocating space for receive buffers, leaving little freedom for applications. There are still some robustness issues in the protocol that needs to be addressed, but an acceptable goodput has been achieved, and flexibility and ease of application development has improved. Furthermore the software framework is highly generic and therefore portable within the AVR family. Implementing updatability through a bootloader system has not been finished, but estimates indicate that software update times can be considerably reduced, compared to AAUSAT-II.

The contents of this report are freely available, but publication (with specification of source) may only be done after arrangement with the authors.
Preface

This report is a collection of the worksheets supplementing the paper and together they document the project, conducted at Aalborg University in the autumn 2008, with the title Software Framework for Reconfigurable Distributed System on AAUSAT3.

Each chapter starts with a figure indicating the progress and it serves as a means of giving an overview of the project. Below is an example from the project aims chapter. After defining the software framework in the third part, the project splits into three branches.

Sources are indicated with the first three letters of the author’s last name and year followed by page numbers, if referring to a book e.g. [Law97, p.38]. Internet sites are not marked with page reference. Figures, formulas and tables are numbered by chapter and location, e.g. the second figure in chapter three is numbered 3.2. Functions are written as main() and the same font is used for file names and variables.

The worksheet report, paper, poster, abstract, test files and c-code with doxygen are placed on the group homepage http://kom.aau.dk/group/08gr724/. Module test files are only referenced by name in the chapters, but can be found on the group homepage.

Abbreviations used in the project are presented after this preface.

The authors would like to thank associate professor Jens Frederik Dalsgaard Nielsen for supervising and the AAUSAT3 engineering group for revision of design.

Aalborg University, December 16th, 2008

Jesper Bønding
Marc Pessans-Goyheneix
Kasper Fuglsang Jensen
Morten Tychsen
Kasper Vinther
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAUSAT</td>
<td>Aalborg University SATellite</td>
</tr>
<tr>
<td>ACK</td>
<td>ACKnowledgement</td>
</tr>
<tr>
<td>ADCS</td>
<td>Attitude Determination and Control System</td>
</tr>
<tr>
<td>AEG</td>
<td>AAUSAT Engineering Group</td>
</tr>
<tr>
<td>AIS</td>
<td>Automatic Identification System</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>OBC</td>
<td>On-Board Computer</td>
</tr>
<tr>
<td>COM</td>
<td>COMmunication</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CSP</td>
<td>CAN Space Protocol</td>
</tr>
<tr>
<td>EPS</td>
<td>Electrical Power Supply</td>
</tr>
<tr>
<td>FLAK</td>
<td>FLAK: Lightweight AAUSAT Kernel</td>
</tr>
<tr>
<td>FP</td>
<td>Flight Plan</td>
</tr>
<tr>
<td>GND</td>
<td>GrouND</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>IDE</td>
<td>IDentifier Extension</td>
</tr>
<tr>
<td>IFS</td>
<td>Inter Frame Space</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>LOG</td>
<td>LOGging</td>
</tr>
<tr>
<td>RDANH</td>
<td>Royal Danish Administration of Navigation and Hydrography</td>
</tr>
<tr>
<td>REC</td>
<td>Receive Error Counter</td>
</tr>
<tr>
<td>RTC</td>
<td>Real Time Clock</td>
</tr>
<tr>
<td>RTR</td>
<td>Remote Transmission Request</td>
</tr>
<tr>
<td>SWIS</td>
<td>SoftWare Image Server</td>
</tr>
<tr>
<td>TEC</td>
<td>Transmit Error Counter</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver and Transmitter</td>
</tr>
</tbody>
</table>

*Table 1: List of the abbreviations used in the project.*
## Contents

1 Introduction 1
   1.1 The AAUSAT Project 1
   1.2 AAUSAT3 Mission Objective 1

2 The Structure of AAUSAT-II and AAUSAT3 3
   2.1 Structure of AAUSAT-II 3
   2.2 Structure of AAUSAT3 4
   2.3 Comparison of the Structure of AAUSAT-II and AAUSAT3 5
   2.4 Status of the Distributed System on AAUSAT3 6

3 Analysis of Communication Infrastructure 9
   3.1 CAN Bus 9
   3.2 CAN Communication on AAUSAT3 14
   3.3 Evaluation of the CAN Communication on AAUSAT3 and Ideas for Improvements 22

4 Project Aims 25
   4.1 Problems 25
   4.2 Objectives 26
   4.3 Project Aims 26
   4.4 Limitations 27

5 Software framework 29
   5.1 Wanted Software Framework 29
   5.2 Needed Software and Goals for The Project 30
   5.3 Functionality and Requirements 31
   5.4 Division of System Level Design 31

6 Kernel Selection 33
   6.1 Possible Kernels for AAUSAT3 34
   6.2 FreeRTOS for AT90CAN128 35
## CONTENTS

6.3 Test of FreeRTOS ......................................................... 38  
6.4 SRAM Memory Management ............................................. 44  
6.5 Conclusion on Kernel Selection ...................................... 46  
6.6 User guide for FreeRTOS .............................................. 47  

7 Protocol Stack for AAUSAT3  
7.1 Protocol Stack Analysis ................................................. 53  
7.2 Functionality and Requirements ...................................... 55  
7.3 Modifications of CSP and CAN Driver ............................... 57  
7.4 Design of Service Layer .............................................. 63  
7.5 Conclusion ............................................................... 68  
7.6 CSP and Service Layer User Guide ................................ 68  

8 Bootloader & Software Image Server (SWIS)  
8.1 Introduction ............................................................. 75  
8.2 Requirements .......................................................... 76  
8.3 General Design ........................................................ 77  
8.4 Communication ....................................................... 77  
8.5 Bootloader Client ...................................................... 80  
8.6 Software Image Server (SWIS) ...................................... 85  
8.7 Transferring Images from Computer to Satellite .................. 88  
8.8 Estimate of Time Needed to Transfer and Flash Software to Client .............................................. 89  

9 Performance tests  
9.1 RAM Usage ............................................................ 91  
9.2 CSP Goodput, Overhead and Traffic Utilization .................... 94  
9.3 Load on MCU with Continuous CSP Communication .............. 98  
9.4 Stress Test .............................................................. 102  
9.5 Bootloader Startup Delay ............................................. 104  

10 Discussion and future work  
10.1 Robustness ........................................................... 109  
10.2 Reconfigurability ....................................................... 110  
10.3 Testability ............................................................. 110  
10.4 Ease of Development ................................................ 110  

References ................................................................. 112
1.1 The AAUSAT Project

AAUSAT is a series of educational satellite projects with the purpose of introducing students to the process of developing a satellite. Since development of a satellite is a larger project than normal semester projects, several groups participate during a period of a few years. The work is supervised by university staff and weekly meetings are held, to address everything regarding system level design.

The AAUSAT satellites are all based on the CubeSat standard which entails a maximum size of 10x10x10 cm and a maximum mass of 1 kg. This standard is used because of the limited time available and the budget.

The AAUSAT project has currently developed two satellites, the AAU-CubeSat and AAUSAT-II. The third satellite, AAUSAT3, currently in development, and will be based on the lessons learned from the previous satellites.

1.2 AAUSAT3 Mission Objective

The primary mission objective of the satellite is given by The Royal Danish Administration of Navigation and Hydrography (RDANH) who wish to examine the possibility of having an Automatic Identification System (AIS) receiver on board a satellite capable of receiving AIS signals from ships around Greenland. Such a satellite will have a large area coverage, thus eliminating the need for expensive ground stations around the rough nature there. The AIS system is already operational with many ground stations around the world and it has made the seas around Denmark safer and more effective, by the introduction of ship traffic surveillance and navigation statistics [The08].

Among the most promising proposals for a secondary mission objectives for AAUSAT3 is mounting of a camera in order to take pictures of the Earth or a Global Positioning System (GPS) receiver that works in
low earth orbit and fits within the space and power constrains of a CubeSat. Another objective could also involve observation of Aurora Borealis and examination of how this interferes with the AIS transponder signals.
The Structure of AAUSAT-II and AAUSAT3

This worksheet provides a description of the internal structure and subsystems used on AAUSAT-II and on AAUSAT3. Based on this description, the differences between them is analyzed and it is described why a distributed system has been chosen for AAUSAT3, in favor of the centralized structure of AAUSAT-II.

Finally it contains an overview of the distributed system of AAUSAT3, a description of the single components and a short status for each component.

2.1 Structure of AAUSAT-II

The logical structure of AAUSAT-II is centralized around the On Board Computer (OBC). Each subsystem is running on separate hardware with limited computational power and is connected to the common CAN bus. Central data handling is carried out on the OBC which has a thread running for each subsystem and coordinates everything centrally. The structure of AAUSAT-II is shown in Figure 2.1.

This logical structure means that a message from one subsystem to another, have to go through the OBC where it will be handed over to the thread of the receiving subsystem and sent out on the bus again. The reason for this structure is that the processing power in the subsystems is very limited and as such, some calculations have to be carried out on the OBC. Coordination between subsystems is also carried out on the OBC, which means that much communication have to go through it in any case.

This centralized structure provides one type of interface between all subsystems and it is quite structured. A centralized structure means that subsystem hardware can be simpler and less powerful. When heavy computation is centralized across subsystems, it can be implemented more efficient, leading to lower power consumption of the overall system.

The downsides are that it introduces single points of failure in the OBC and related hardware, and makes the system complex to test and debug. It also means that the software for each subsystem is located partially on the subsystem hardware, and partially on the OBC hardware. Software developers have to
take care of two different hardware platforms. The developed software is difficult to debug, because each subsystem needs the OBC.

### 2.2 Structure of AAUSAT3

The structure of AAUSAT3 is determined partly based on the lessons learned from AAUSAT-II. AAUSAT3 is also composed of a number of subsystems, but no central coordination is carried out, so the satellite is essentially a distributed system which can be seen in Figure 2.2. The idea is to create a satellite that is capable of functioning even if some parts of it are not working. The subsystems will be based on the same hardware and software platform to achieve a high degree of modularity, which enables flexible development and easy reconfigurability before and after launch.

The subsystems on AAUSAT3 can be divided into three groups. The first one contains four subsystems which are essential for the satellite to be working and support the mission:

- **Electrical Power Supply (EPS)**
  Delivers electrical power to all the other subsystems and monitors theirs status. If a subsystem draws too much power, gets overheated or stops responding, it is shut down or rebooted.

- **Communications (COM)**
  Takes care of communication with Earth over UHF radio link. This includes basic beacons as well as more advanced communication.
• **Flight Planner (FP)**
  
The Flight Planner should be able to control the satellite when it is not in contact with a ground station. This basically means issuing commands to different subsystems at a specified time or when a given event occurs.

• **Log system (LOG)**
  
For determining satellite behavior and status when it is not in contact with a ground station, the data need to be recorded and saved in a log.

The second group contains the subsystems that are needed to fulfill the primary mission of the satellite, namely to gather AIS data:

• **Automatic Identification System receiver (AIS)**
  
The main payload of the satellite which is supposed to test the possibility of receiving AIS signals in space.

• **Attitude Determination and Control System (ADCS)**
  
It is desired to control the attitude of the satellite since it improves signal conditions for the COM and AIS subsystems. It is also entirely necessary if a camera has to point in the correct direction.

A number of further payloads have been proposed, which can be added to the project if time, power etc. allows it. This list includes a secondary, faster COM, a camera (CAM), and GPS receiver (GPS). A proposal has also been made to have a COM subsystem operating on the VHF band. A further functionality would be to enable the satellite to work as a relay for e.g. voice or to relay the received AIS signals directly.

Ideally, the subsystems on AAUSAT3 can be developed individually and then at a late stage in the project, it can be determined which subsystems can share the same hardware.

### 2.3 Comparison of the Structure of AAUSAT-II and AAUSAT3

This subsection analyses and compares the centralized structure on AAUSAT-II versus the distributed structure on AAUSAT3. In this way it can be argued why a distributed system has been chosen for AAUSAT3 and what the consequences of this choice are. A number of differences can be listed as it is done in Table 2.1.

<table>
<thead>
<tr>
<th>Subsystem development</th>
<th>AAUSAT-II</th>
<th>AAUSAT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging</td>
<td>Dependent on OBC</td>
<td>Independent</td>
</tr>
<tr>
<td>Reconfigurability</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Add new subsystem</td>
<td>Complex</td>
<td>Easy</td>
</tr>
<tr>
<td>Power usage</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

*Table 2.1: Comparison of AAUSAT-II and AAUSAT3 structures.*

From the comparison table, it can be seen that a distributed system has a large number of advantages over a centralized system. These advantages however, is dependent on having a suitable platform for developing the subsystems so that they are capable of working together.

Thus the main disadvantage of the distributed system is that some general work has to be carried out before the advantages can become clear. When finished, it is likely to present a less optimized system than a centralized solution since the flexibility increases overhead in the system.
It is however considered that the advantages clearly outweigh the disadvantages of the distributed systems. As the analysis showed, it is important to have a good basic platform system, which is the topic of the next section.

2.4 Status of the Distributed System on AAUSAT3

As mentioned before, a good platform have to be made in order to achieve the goal of easily being able to develop subsystems for the AAUSAT3. This platform should make it possible for developers to focus on theirs subsystems functionality and less on hardware, communication etc.

A number of key points have been identified and these are listed below with a description of the current status.

2.4.1 Common hardware platform

If the subsystems are meant to run on the same type of hardware, it is very beneficial to have a development platform available. In spring 2008, group 414 developed such a platform [MPB+08]. It is working satisfactory and provides a good basis for development and test.

2.4.2 Kernel

It is essential to have a kernel running on the subsystems, since software can then be more modular and it is easy to have more tasks running on one hardware platform at one time. In spring 2008, group 450 developed FLAK (FLAK: Lightweight AAUSAT Kernel) [BRLPH08], which was supposed to be used on the subsystems.

It has turned out, that there is a need for a preemptive kernel, a feature which is not supported by FLAK. It will considerably increase the complexity of FLAK and will take some time to implement. It has to be determined, if FLAK is still the right choice or if some other kernel should be employed.

2.4.3 Reconfigurability

A cornerstone in the distributed structure used on AAUSAT3 is the possibility to move software to other hardware platforms. This requires a great deal of flexibility from the software platform, especially if reconfiguration have to take place after the satellite have been launched.

Some thought has been put into this area, but no practical results have been obtained yet.

2.4.4 Test procedures

Because all subsystems are running on the same basis, it should be possible to carry out some generic tests for each subsystem to confirm basic functionality.

No work have been done within this area.
2.4.5 Communication Infrastructure

Communication between subsystems is essential, so an effective communication infrastructure have to be developed. In autumn 2007, group 722 made a proposal for the CSP (CAN Space Protocol) to be used on AAUSAT3 [RJdCCA07]. CSP uses the CAN protocol and provides a simpler interface for the user application.

Much work has already gone into designing CSP. To obtain knowledge about this central part of the distributed system and analyze what still needs to be done to have a fully functional communication infrastructure, an analysis of CSP is made in the following.
An analysis of the internal communication infrastructure is needed, in order to understand the code, that has already been written by other groups and to point out where the code lacks. This includes a description of the chosen field bus (CAN) and a more thorough walk through of the CAN Space Protocol (CSP) with a user guide. Other forms of possible communication links, like a direct link between the VHF receiver and the UHF transceiver, will not be analyzed in this chapter, since it is still in a suggestive phase.

There will not be looked into other possibilities than CAN bus for many reasons. The first is that the CAN controller is implemented in a wide variety of MCUs, with error handling and fault confinement done at hardware level and the CAN protocol standard has been on the market for over 20 years [sol08]. The second is that its efficient, it works and its a bit late in the development process, meaning there has been put a lot of work into making CSP for the CAN bus and it would also require a change of hardware. CAN bus was furthermore used on AAUSAT2 with success.

### 3.1 CAN Bus

The purpose of this section is to give a brief introduction of the essential parts of the Controller Area Network bus (CAN bus) in order to have a basic understanding of the advantages/disadvantages of using the CAN bus. This section will furthermore contain a description of CAN framework architecture in order to gain knowledge in how to use the CAN specification. This section is based on information from the following source [Law97].

#### 3.1.1 CAN Bus Introduction

CAN is a serial network protocol and a bus standard for communicating between multiple devices without having a dedicated host. The CAN protocol is also called autobus protocol which is designed for sending
small messages over short distances between sensors and actuators.

The bandwidth of the CAN bus is limited to maximum bit rate of 1 Mb/s as long as the bus length is under 40 m. If the length of the bus increases the maximum bit rate decreases [Law97, p.38] (see following equation).

\[
\text{Bus length} \times \text{Bit rate} \leq 40 \text{ m} \times 1 \text{ Mb/s} \quad (3.1)
\]

Equation 3.1 shows that there is a linear correlation between the bit rate and the bus length. As an example, the maximum bus length with a bit rate of 500 kb/s is 80 m. This is however of no concern in AAUSAT3 where the length of the cable will not exceed 40 m.

Every device connected to the CAN bus will always be listening on the bus. Even if a device is transmitting data onto the bus, it simultaneously listen, thereby monitoring its own output. In order to avoid message collision, devices connected to the CAN bus is not allowed to transmit data onto the bus, when the bus is not idle. When the CAN bus is idle, every device is allowed to transmit data onto the bus without notification.

In order to avoid two devices to transmit different data at the same time, the CAN protocol utilize prioritizing of the data. Every packets, sent on the CAN bus, includes an identifier which indicates the meaning of the given packet. This identifier can also be used to prioritize packets (a more thorough description of the prioritizing of data will follow later in this section). The device with the lowest prioritized data will stop transmitting immediately.

Every data is always transmitted to all devices connected to the CAN bus and thereby its up to the individual device to filter which data there is relevant for the given device. The filtration is, like the prioritizing, based on the identifier and is done by the application.

The CAN protocol implements five error detection methods from which tree is at message level (frame check, acknowledgment check and Cyclic Redundancy Check (CRC)) and the last two is at bit level (bit monitoring and bit stuffing). The different methods is described here:

- **Frame check:**
  There are certain points of the CAN frame that has a predefined bit value (more about this will follow later). If the receiver detects an invalid bit value at those points, a form error is signaled.

- **Acknowledgment check:**
  In the last part of the CAN frame it is possible for the receiver node to mark a successful transmission by changing a bit value (more about this will follow later). If the transmitter node does not detect this change in the bit value, a acknowledgment error is signaled.

- **Cyclic redundancy check:**
  The CAN frame contains a 15 bit CRC code calculated by the transmitter. If the CRC code, calculated by the receiver CAN on base of the received frame, is different from the sent CRC code, a CRC error is signaled.

- **Bit monitoring:**
  When the transmitter are transmitting data onto the CAN bus it simultaneously monitors the signal on the bus. If the transmitted and the read signal is diverting from each other, a bit error is signaled.

- **Bit stuffing:**
  Bit stuffing is included in different parts of the CAN frame. For every five consecutive bits the transmitter will include an extra bit of opposite polarity. Therefore if a receiver detects six consecutive bits, a stuff error is signaled.
If an error is detected by any of the devices connected to the CAN bus it transmits an error frame. The message will then be discarded by all the other devices, the error counter is incremented on all the devices and the transmitter retransmit the message.

In addition to the error detection methods the CAN specification also includes an error confinement method which protect the bus from a permanent failure in a part of the bus, in one of the devices connected to the bus or due to, a long lasting external disturbances etc.

The error confinement system uses CAN’s error counters included in each CAN controller. The specific counters is the Receive Error Counter (REC) and the Transmit Error Counter (TEC). Based on the value of one or both counters, each of the devices connected to the bus can be in three different states:

- **Active error:**
  A device will be in the active error state when it is working properly. This means that it can transmit and receive data via the CAN bus. If the device detects an error it transmits an error flag.

- **Passive error:**
  A device will be in the passive error state when either of the error count registers of the device exceeds 127. In this state the device is still able to transmit and receive data via the CAN bus, but if the device detects an error it is restricted in relation to how it signals an error.

- **Bus off:**
  A device will be in the bus off state when the TEC register of the device exceeds 255. In this state the device will no longer be able to transmit or receive data via the CAN bus. In this state the device can only become active on the bus again if the user request it to return to active error state.

The state diagram can be seen in figure 3.1

![State Diagram](image)

**Figure 3.1:** State diagram for a device connected to the CAN bus.

The counters are incremented with different weight dependent on the type of error detected by either transmitter or receiver. The counters are decremented by one when the device successfully transmit or receive a message frame.

### 3.1.2 CAN Bus Frame Architecture

The CAN protocol includes several types of frames (data frame, remote frame, error frame, overload frame and inter frame). Furthermore the data frame and the remote frame can be either a standard frame
or an extended frame, depending on which CAN specification that has been used for the application. CAN 2.0A employs the standard frame (standard CAN), while CAN2.0B employs the extended frame (extended CAN). The architecture of each type of frame is described here:

**Data frame/Remote frame:**

The architecture of both the standard data frame and the extended data frame is seen on figure 3.2.

![Figure 3.2: Architecture of the CAN frame with standard identifier (top) and extended identifier (bottom).](image)

The first field of the data frame is the start of frame (SOF) which is a single dominant bit (a bit value of 0) used to synchronize all devices connected to the CAN bus.

The second field is the arbitration field. This field contains either an 11 bit identifier or a 29 bit identifier, dependent on if it is a standard data frame or an extended data frame, plus a single remote transmissions request bit (RTR). Furthermore the extended data frame contains a single identifier extension bit (IDE) and a single substitute remote request bit (SRR). The identifier is used by the application to filter relevant messages and by the CAN bus for message prioritizing. The prioritizing is done by bitwise arbitration where a dominant bit indicates a higher priority (0b001 has higher priority than 0b010).

The RTR bit indicates if the data frame in fact is a remote frame. The remote frame (standard and extended) has the same structure as the data frame (standard and extended), but the remote frame does not contain a data field even if the data code in the control field indicates somethings else. The purpose of the remote frame is to make it possible, for a device connected to the CAN bus, to request a data frame with a given identifier. The identifier transmitted in the remote frame is the wanted identifier. A dominant RTR bit indicates a data frame and a recessive (a bit value of 1) RTR bit indicates a remote frame.

The IDE bit indicates if the data frame/remote frame is a standard or an extended frame. A dominant IDE bit indicates a standard frame and a recessive bit indicates an extended frame.

The SSR bit is only used in an extended data frame/remote frame and substitutes the RTR bit in the standard frame. The SSR bit is always recessive.

The third field is the control field. This field contains the IDE bit, a single reserved bit (r0) and 4 data length code bits in the standard frame. In the Extended frame the IDE bit is replaced with a second reserved bit (r1). The reserved bits (r0 and r1) is always sent dominant but receivers accepts both dominant and recessive bits in all combinations.
The data length code (dlc) indicates the length of the following data field in bytes. The valid values is between 0b0000 (0 bytes) and 0b1000 (8 bytes).

The forth field is the data field. This field contains the application data and can be between 0 and 64 bit (8 byte) long.

The fifth field is the CRC field. This field contains 15 bits, used for the CRC checksum of the preceding bits, and a single CRC delimiter bit which is always sent recessive.

The sixth field is the acknowledgment field. This field contains a single acknowledgment slot bit and a single acknowledgment delimiter bit which both is sent recessive. If a receiver detects a correct message (a correct CRC checksum) it superscribe a dominant bit i the acknowledgment slot. If the transmitter detects a dominant bit in the acknowledgment slot the message is transmitted correctly.

The last field of the data frame is the end of frame (EOF) which is seven consecutive recessive bits which marks the end of the data frame.

**Inter frame:**

The inter frame, also called the inter frame space (IFS), consists of three consecutive recessive bits plus a variable bus idle time.

The three IFS bits is the minimum space required between each data frame in order to let the CAN controller transfer messages from the bus handler to the message buffer. The variable bus idle time is due to the requirement of the bus must be idle in order for any device to start transmission of data onto the bus.

**Error frame:**

The architecture of the error frame is illustrated on figure 3.3.

![Architecture of the error frame.](image)

Figure 3.3: Architecture of the error frame.

When a receiver detects an error it transmits either an active error flag or a passive error flag depending on which state the receiver is in.

If the receiver is in the active error state it transmits 6 consecutive dominant bits thereby violating the bit stuffing rule. This is detected by all the receivers which also transmits an error flag (active or passive depending on which state they are in). The actually monitored signal on the CAN bus is thereby a superposition of the different error flags. As a consequence of this the length of the error flag field is between 6 and 12 bits long.

If the receiver is in the passive error state it transmits 6 consecutive recessive bits. This means that it has no effect on the activity on the CAN bus. In order to terminate an error frame correctly the error passive receiver must detect 6 consecutive bits of same polarity.

The last field of the error frame is the error delimiter. This consists of 8 consecutive recessive bits and marks the end of an error frame.

**Overload frame:**

The overload frame is used be the CAN controller if it, due to internal delays, is not able to process the received message. The overload frame request the transmitter to repeat the message, thereby delaying the next transmission. The architecture of the overload frame is illustrated on figure 3.4.
CHAPTER 3. ANALYSIS OF COMMUNICATION INFRASTRUCTURE

Figure 3.4: Architecture of the overload frame.

The overload frame is identical to the active error frame as it appears from figure 3.4. The difference is that the overload frame starts at the last bit of the EOF or in the IFS.

As presented in this section the CAN bus has no message collision and gives and error robust network communication, with message prioritizing and automatic retransmission of messages. The drawbacks are the relative small packages (max 8 bytes) which gives a large overhead, and a maximum bit rate of 1 Mb/s. Since the focus when choosing a field bus for the satellite was low power usage, high speed and a low number of lines, the CAN bus was an ideal candidate out of five candidates in consideration.

The next section takes a look at how the CAN bus has been implemented for communication on AAUSAT3.

3.2 CAN Communication on AAUSAT3

In the fall semester of 2007 a 7. semester group (07gr722) made a new protocol for AAUSAT3 called CAN Space Protocol (CSP) [RJdCCA07].

AAUSAT2 has a centralized system with an OBC running all the treads, but for AAUSAT3 a distributed system was proposed and therefore a new and improved satellite communication infrastructure was needed. The aim was to make the protocol more simple, robust and easy to use. The new developed protocol stack includes a fast interrupt based driver and a connection oriented transport protocol and tests show that the distributed system and the new protocol gives the designers a more free and flexible development environment. It also improves CPU usage.

This section will look into CSP and the protocol stack in general and describe its architecture and how it should be used. At the end some suggestions for improvement will be presented. It is important to remember that the description of CSP was only valid, before any changes done in this project where performed.

3.2.1 The Protocol Stack

In accordance with the OSI and TCP/IP model, the communication infrastructure on AAUSAT3 can be divided into layers. Figure 3.5 shows the different layers also referred to as the protocol stack. Since there is only one bus and a fixed number of nodes, the network layer has been omitted. Also the session and presentation layers from the OSI model has been merged together to form the service layer.

The different layers will now be described and from the bottom up we have 1:

- **Physical layer:**
  This layer consists of the actual CAN bus hardware, described in section 3.1, including the CAN transceiver and the CAN controller inside the AVR MCU. The layer takes care of transmission of raw bit streams and forms the physical interface between the different hardware platforms.

1The data link layer is described a bit more in detail, since it is not described elsewhere, like the other layers.
CHAPTER 3. ANALYSIS OF COMMUNICATION INFRASTRUCTURE

Figure 3.5: Layers in the communication infrastructure on AAUSAT3. The illustration shows two nodes on a CANBUS.

- **Data link layer:**
  This layer consists of a driver that runs on the AVR MCU. It takes care of the lowest message addressing and implements the error handling described in section 3.1. It also builds the frame. The CAN frame described in section 3.1 has a standard (Base ID) or an extended (Base ID plus Extended ID) identifier. Figure 3.6 shows the two possible frames that the data link layer can make.

Base ID contains the priority of the message (3 bits), the source address (4 bits) and the destination (4 bits), which is 11 bits in total. This gives 8 priorities and 16 node addresses, see table 3.1. The priority is placed first in the identifier, which means that lower priority messages will be masked out by higher priority messages on hardware level. The source and destination address field should function as a filter on hardware level, so messages not sent to the particular node will not disturb the MCU unnecessarily, but at the moment it is done in software.

Extended ID is for the overlying protocol to keep track of the connection dependent values. These are destination port (3 bits), source port (3 bits), frame type (4 bits) and message length (8 bits). Frame types will be described in subsection 3.2.2.

Masking of frames (filtering based on the identifier) should be done by hardware and there are 15 message objects (MOBs) that can handle this.
When a frame is received by a node it is first read into a temporary buffer. Then it is matched against the different MOBs (MOB 0 first because it has highest priority). When the identifier of the frame matches the mask on a MOB, the data are read into the MOB. If there is no match the garbage collector MOB (MOB 14) will grab the frame, which can be used to see if there is a problem receiving messages. It is possible to set MOBs up for either transmitting or receiving and figure 3.7 shows the MOB setup.

Figure 3.7: Configuration of MOBs.

There is a need for multiple RX and TX MOBs, because the data is sent over CAN faster than they can be read and a requirement is, that there is no loss of frames.

The driver in the data link layer provides a set of functions to handle the MOBs and it is interrupt based, so it does a callback to the transport layer, when data is send or received. When receiving a new message, the callback will make sure that the data from the MOB are read in and will not be overwritten and it is set up for the handler in the transport layer.

For debugging a `can\_print\_status()` function has been made, that sends out statistics like errors and sent/received messages for the CAN communication on the serial port.

**Transport layer:**

Since it is possible to have multiple connections between the same subsystems at the same time, this layer makes a handler to each connection. The transport layer also opens and closes ports for
incoming or outgoing messages and reacts on the different frame types. It is also updating the RX and TX state and can function differently depending on the wanted reliability, the size of the data and the latency demand. If needed, this layer also fragments data, if there is more than 8 bytes to be send.

The transport layer consists of the CSP. The section 3.2.2, describes how the interface functions can be used to communicate between subsystems.

- **Service layer:**
  Intentionally this layer should be a combination of the network API and a set of middleware applications, but it is not implemented yet.

- **Application:**
  The subsystem application software should run on top of the protocol stack, where the service layer provides services to the application.

### 3.2.2 Description of CSP

The protocol consist of the following files:

- **canlib.h and canlib.c:**
  This is a modified version of the Atmel CAN driver.

- **libcsp.h and libcsp.c:**
  This is where all the CSP functions are placed.

- **csp_aausat3.h:**
  This file contains defines for nodes, priorities, packet types and frames.

- **board.h:**
  This file contains define for CAN baudrate.

The protocol is port and connection oriented. A subsystem can have 8 ports open and in order to receive a message a port must be opened in INCOMING state, with the `csp_open_port()` function and when sending a message the `csp_connect()` function is used and it changes the port state to OUTGOING. The different port states are shown on figure 3.8.

![State diagram for a port. In the code there is a PORT_ in front of the different states.](image)

When a frame for either an INCOMMING or OUTGOING port is recognized a port-connection list is searched for matching connections. If the frame did not belong to an already existing connection a new connection is made with allocation of memory. There are different frame types that the connection responds to and the frame type is defined in the extended identifier. The five frame types are:

- **BEGIN:** Indication of the first frame in a message.
• MORE: This is sent in the following frames to continue the message. The length of the data to be sent is also defined in the extended identifier, so the receiver knows when the last frame has come.

• ACK: When all the frames in a message have been received, the receiving side sends an ACK frame as a reply to the sender.

• ERROR: Sequence error, restarts transmission.

• END: close connection.

Figure 3.9 shows the sequence diagram for a typical communication between two nodes, where node 2 sends a message to node 1.

The sequence is described below with the real function call in parentheses.

• Listen() (function: `csp_port_open()`): Firstly the receiving node (node 1) must be set up to listen on a given port.

• Connect() (function: `csp_connect()`): Then another node (node 2) wants to send a message to node 1 and connects to the port that node 1 is listening on. This also makes a connection handler.

• Send() (function: `csp_zsend()`): Then node 2 sends the message to node 1. The first frame is of type BEGIN and this frame also sets up the connection on the receiving size and tells it how many frames the message consists of. In this example the message is larger than 8 bytes, so the next frame is of type MORE, but smaller than 17 bytes so only one MORE frame is sent.

• Callback/poll: When node 1 has received both frames it changes state to RXDONE and makes a callback to the application, telling it there is data available in the RX buffer. It is also possible to poll the ports, to see if any of them has data ready in the RX buffer (function: `csp_select()`).
• **Read()** (function: `csp_zread()``): Then the application reads the RX buffer.

• **Done()** (function: `csp_zdone()``): When the buffer is read in it runs done, which will send an ACK frame to node 2 and change the state to RXREAD.

• **Poll**; Node 2 is in state TXSENT and is polling and waiting to get an ACK frame and when it does, it changes state to TXDONE.

• **Close()** (function: `csp_conn_close()``): Then it closes the connection and the port and sends an END frame to node 1, which closes the connection on the receiving side, but not the port. Another option would be to send another message (function: `csp_zsend()``).

Figure 3.10 shows the different states a connection can be in.

![State diagram for a connection](image)

**Figure 3.10:** State diagram for a connection. In the code there is a SOCKET_ in front of the different states, except for receiving and sending, where the state are just RX and TX. Standard socket terminology has been used in the figure and the actual function calls is written in the text.

If a BEGIN frame is received on a port, a connection in state CLOSED will be changed to state RX and if the function `csp_zsend()` is called a connection in state OPEN will change state to TX. A connection can be closed when in TXDONE or ERROR state, with the `csp_conn_close()` function, which will also send an END frame to the receiver and terminate the connection there.

The error handling is described in the next subsection.

To understand how to implement communication between subsystems on the CAN bus, some example code is presented next. Place the following includes in a header file:

```c
#include "board.h" // General defines (CAN and UART baud rate)
#include "init.h" // Init of MCU, UART and interrupt
#include "flak.h" // Used for kernel
#include "csp_aausat3.h" // Defines for CSP (nodes, prio, frames)
#include "can_lib.h" // Init of CAN and CAN functions (this is Atmel's driver)
#include "rtc_drv.h" // Handles timer clocks and timeout functionality
#include "libcsp.h" // CSP functions
#include <stdio.h> // printf etc.
#include <stdlib.h> // Type conversion, mem allocation and process control (exit)
#include <avr/io.h> // makes it possible to use I/O pins
```

The main function can look like this:
```c
int main(void) {
    /* Start up board */
    mcu_init();
    interrupt_init();
    rtc_init();

    /* Initialize CAN */
    csp_init(NODE_R3);

    /* CSP event handlers: Receive packets on port 1 */
    csp_port_open(1,1,&recv_packet);

    /* Initialize kernel */
    flak_init();

    /* Create task: send out message each 5000 ms, with priority 1 */
    flak_create_task(&packet_sender, 5000, 1);

    return 0;
}
```

First initialization is performed. Line 9 sets the hardware address of the board to NODE_R3, which is defined in `csp_aausat3.h`. Each hardware board should have a unique address. On line 12 the board is set up to listen on port 1 with possibility of 1 connection. If a message is received a callback to the function `recv_packet()` is performed. Then a task with priority 1 is created which sends out a message each 5 seconds and finally the scheduler is started. A typical implementation of `recv_packet()` is shown here:

```c
static volatile U16 buffer = 0;
static volatile U8 len = 0;

void recv_packet(conn_t * conn) {
    /* Save data in buffer */
    U8 *p;
    len = csp_zread(conn, &p, 0);
    if (len==0){
        printf("Receive error");
    } else{
        buffer=((U16*) p);
    }

    /* Send ACK to transmitter */
    csp_zdone(conn);
}
```

The `csp_zread()` function makes a pointer to the data received on the connection, with 0 timeout. If the connection is not in state RXDONE it will return without a pointer to data and with timeout it can wait a certain amount of time. The function returns the length of the received data and if it is zero an error has occurred, otherwise the data is copied to the buffer variable. `csp_zdone()` sends an ACK frame back to the transmitter.

The `packet_sender()` function looks like this:

```c
void packet_sender() {
    U16 buffer=0xFFFF;

    /* Send out a packet */
    static conn_t* conn = NULL;
    if (conn == NULL)
        conn = csp_connect(PRIO_NORM, NODE_R4, 1);

    U8 *p = ((U8*) &counter_led);
```
The function `csp_connect()` tells the data link layer that a connection to NODE_R4 is needed on port 1 and returns a handler to the connection. `csp_zsend()` then sends buffer to NODE_R4 with data length 2 and timeout of 5 ms (if no ACK is received within 5 ms, it will timeout and change connection state to ERROR). `csp_conn_close()` sends an END frame to NODE_R4 and closes the connection and the port.

In order to summarize, the functions that interfaces the transport layer are presented here:

The `csp_zread()` function takes a port and looks for connections in RXDONE state and returns a pointer to the connection, which means that it polls for data. It can be used instead of the callback function, but the port must still be open.

The `csp_transaction()` function implements a complete transaction, where a client requests a service from a server. It could be used if for example a subsystem needs a value from another subsystem. Then `csp_transaction()` will send request and wait for answer.

### 3.2.3 Error handling

On physical and data link layer the error handling is done as described in section 3.1. The error handling here are bit monitoring (are the bits on the bus the same as i send?), bit stuffing (did I receive too many consecutive bits?), frame check (is the received frame architecture correct?), acknowledgment check (did the receiver acknowledge my frame?) and cyclic redundancy check (Is my received data corrupt?).

The error handling done in the transport layer will now be presented.

- **Missing ACK frame:** The `csp_zsend()` function polls until an ACK frame was received. If this does not happen within the specified timeout, the `csp_zsend()` function is terminated and the TXSENT state is changed to ERROR. It does however not close the connection, so if this happens too many times, the TX MOBs will run out, unless the connections are properly closed with `csp_conn_close()`.

- **Sequence error:** If the received frame is out of order the state will change to ERROR and an ERROR frame is sent to the transmitter. The transmitter will then stop sending and stay in the state it was in.

- **Receiver overflow error:** If a MORE frame is received and the RX buffer is full, the RX state is changed to ERROR and an ERROR frame is sent back and the transmitter stops sending.

- **Receiver out of memory:** If there is not enough space to allocate memory for the buffer, the receiver will send and END frame to the transmitter who closes the connection.
• **Out of memory when port open**: If there is not enough free memory to allocate pointers to connections when a port is opened, `csp_port_open()` returns -1.

• **Received standard frame**: A return is made since the CSP RX handler can not handle standard frames. It is simply ignoring the frame.

• **Destination not my address**: If a message is received to an address that is not my own, ignore it.

• **Out of connections**: If the frame did not belong to an open connection and there is no more free connections on a port, the frame will be ignored.

• **Out of TX MOBs**: Fail sending.

• **Out of source ports**: If the `csp_connect()` function can not find a closed port it returns NULL.

• **Received message for a closed port**: Frame ignored.

• **Received ERROR message for a conn in OPEN state**: An END frame is sent back and both port and connection is closed.

• **Received ERROR message in RX state**: Frame ignored.

• **Receive timeout**: If a timeout is specified for the `csp_zread()` function, then it will change the connection state to ERROR if it did not succeed to read data within the timeout limit.

There is also implemented some checking of input to functions and if input does not fit, the function returns. This form of error handling is not in the list, because that will make a very long list.

The next section evaluates the CAN communication and the implemented error handling and some ideas for improvements will be presented.

### 3.3 Evaluation of the CAN Communication on AAUSAT3 and Ideas for Improvements

As it is now, the CAN communication is working, but there is still some work that needs to be done. The list below states what needs to be implemented in order to comply with the requirements and it also gives some ideas for improvement.

• **Must implement (in accordance with requirements):**
  
  – There are currently implemented two checks to see if the receiver is out of memory. One check is made when trying to open a new port where the function returns -1 and the other check is made when trying to establish a connection, where the receiver sends an END frame to the transmitter. In these situations the subsystem should be rebooted or memory should be freed, otherwise its hard to say how well the subsystem will function.
  
  – At the moment there is no hardware filter implemented, meaning every frame send over CAN will interrupt all the subsystems on the bus and requirement §6.4.1 states that it must be possible to mask out messages on a destination address.

• **Ideas for improvement:**


The communication could be improved by implementing a better sequence number error handling. Right now the whole message will be dumped if one frame is missing and that is problematic if its a long message, but the problem is not that big since sequence errors are very seldom.

The RX handler can react on 6 different frame types. It is possible to have 16 different frame types, so some of them could be used to indicate what type of error occurred, which could result in better error handling.

It might be an idea to do something more graceful instead of having the garbage collector MOB reset the CAN when it gets a frame.

If the transmitter does not get an ACK within the specified timeout, it will go to error state, but it could be an idea to try and resend the message.

Rewrite some of the functions in CSP so they work with an array of connections, instead of having multiple for loops to run trough the all ports for the connections. This will make the code more simple.

If a device is responsible for a lot of errors it will go into bus off mode, but nothing is done if this happens. It would be nice if this could be handled in a nicely manner, so the device could be recovered and go into error active mode again.

It should be possible to place more than one application on the same hardware and still have unique addresses for each application.

Implement the possibility of sending frames with no ACK, to use when a sender does not care about if the data is received or not.

Develop the service layer in the protocol stack. This layer should consist of common services and APIs that will give the application developers a common API to work with.

Basically the existing code should be refined. A service layer should also be developed, in order to make common services and APIs a part of the infrastructure, thus making it easier to develop new application software.
The intent of the following is to give the reader insight in the problem statement, objectives and possible solutions for the work on AAUSAT3. This evolves to a list of aims for the project, and the limitations concerning the tests, which should be performed on the end solutions, in order to achieve the stated aims.

4.1 Problems

The problems which will be considered in the future work evolves from the known problems of AAUSAT-II and limitations in the already developed subsystems and protocols.

On AAUSAT-II some problems with a faulty CAN-transceiver causes the system to restart every few hours. This problem can be helped by uploading new software to the satellite. The current firmware only allows uploading of whole software images, which are quite large, and can not be done in a single pass. The software image is this large because AAUSAT-II only has one main processor and a few small PIC-type processors. Thus the main processor handles most of the satellite functionality. Due to this architecture the AAUSAT-II has a single point of failure, being the main processor.

On AAUSAT3 several subsystems are already developed. The subsystems communicate through a CAN bus interface. A CAN Space Protocol (CSP) has been developed. CSP handles communication calls between subsystems, but has no handling of internal communication between two applications running on the same MCU. Furthermore it does not filter frames on hardware level based on destination address.

The operating system and software development for AAUSAT3 makes programming applications comprehensive and difficult. A lot of knowledge of the underlying protocols, libraries and function calls are needed in order to program workable applications.
4.2 Objectives

The goals of the current development is divided into two main focuses. The first area of focus is based on the requirements from the E-study board at Aalborg University, and the second area of focus is the adaptation of these requirement to fit the problems stated in the above.

The study board states different requirements for the students to achieve. These requirements concern both the technical work and the process of producing this. The first technical objective is to demonstrate insight at application level in relevant theories, methods and techniques used for distribution, storage and processing of data in a distributed system. The second objective is to demonstrate insight in real-time, performance, safety and robustness aspects. The reason for the statement of these goals by the study board are that distributed information processing is an essential aspect of modern communication and control systems, which forms the basis for efficient utilization of hardware and software at all levels. These levels range from individual feedback loops, sensors or processors to large-scale Internet-based business systems. Achieving insight into real-time and distributed processing is an integral part of the student’s general engineering qualifications.

These objectives are interpreted and fitted to the problems stated above. This ends up in more specific technical goals proposed by the participants of the project group.

The main goal for this project is to ensure the AAUSAT3 mission objective. The main objective for the AAUSAT3 is to have a working CubeSat, capable of receiving AIS signals from ships round Greenland. There are also some secondary mission objectives, which involves a camera payload, multiple downlinks and especially distribution of the on-board computing. The AAUSAT3 is a student satellite, so students develop all the satellite software and hardware systems. This means that the different software applications will be developed by many different people, loosing overview and uniformity of the applications. Therefore a goal is to make sure that applications are easy to develop and easy to work with through some uniform framework.

In previous CubeSat missions the need to update software has proved very important. A software update can improve software already on the satellite or implement new functionality. AAUSAT3 is a highly distributed system, which makes multiple, and different, software updates a possibility. The goal of this project is then also to implement the software updatability. Implementing software updates, will make the whole satellite software platform scalable, if every software can be updated or moved to a different hardware platform.

4.3 Project Aims

A software framework with a good software platform should be made in order to achieve the goal of easily being able to develop application software for AAUSAT3. The platform should make it possible for developers to focus on application software and less on hardware, communication etc. and also allow subsystem applications to be executed on multiple or the same satellite hardware module.

A scheme for updating software in space and the mentioned software platform will make the foundation for the software framework. This foundation will make the distributed system on AAUSAT3 more robust, reconfigurable, testable and easy to develop.

The aims are as follows:

- Create a software platform.
- Make in-space software updates possible.
4.4 Limitations

There are some limitations to performing the test on the developed systems in accordance with the project aims. The primary limitation is in the overall test of the satellite, since it is scheduled for launch several years after the end of this project. Thus it is not possible to test the systems in space conditions.

There are also limitations to testing the software framework, since the requirements for it, is that it will be used and ease programming and testing. A test for this will take a semester as a minimum, for accurate results. The reason for this is that a new project group will have to use the framework in their work on the satellite, which takes a semester.
This chapter describes the wanted software framework, what kind of software is needed in this framework, and where the focus and goal for this project is. It also lists the functionalities/requirements.

5.1 Wanted Software Framework

In chapter 4 problems and goals for this project is presented. These problems and goals lead to the project aims stating that a software framework will make the satellite more robust, reconfigurable, testable and easy to develop.

This software framework should consist of a bootloader, a software platform containing kernel, communication, services and drivers and on top, the application software. Figure 5.1 illustrates the suggested software framework for AAUSAT3.

The distributed system on the satellite will consist of multiple MCUs. In the flash of each MCU a bootloader and a software platform will be placed. The software platform should consist of a kernel, communication protocol with services and drivers for external flash, Serial Peripheral Interfaces (SPI), etc.

The idea is to reuse the software platform on every MCU, creating a standard interface between applications and hardware, thus making the system more robust and testable. This will also make applications easier to develop. If the applications are seen as a house, then the software platform would be the foundation supporting this house.

If the software platform is the foundation then the bootloader could be seen as the building contractor, responsible for putting applications wherever wanted, thus the bootloader will make it possible to move applications from one MCU to another, if the application does not depend on specific hardware, and make it possible to update software if needed. The figure suggests that application 1 and 2 are placed on the same MCU and application 3 on another MCU, but Application 2 could ideally also be put together with application 3.
In the sense of modularity/reconfigurability the ideal case would be a system, with applications totally independent of specific hardware and no cross dependence to other applications. This however leads to increased software size and might not be possible with the constraints on the satellite, so a balance must be obtained. An application like COM has to interface with the antenna and can therefore not be moved, but should still rely on the same software platform as the other applications.

5.2 Needed Software and Goals for The Project

The proposed bootloader has not yet been developed. This work also includes a scheme for handling different software images and a fall back mechanism for security if a new booted image does not work.

There has already been developed a communications protocol called CSP, which uses a CAN driver. There are some functionalities that are missing in the communication like hardware filtering of messages and a list of possible upgrades are presented in section 3.3. On top of this a proposed service layer that takes care of trivial tasks needs to be developed.

The currently used kernel for the satellite is FLAK, but this kernel does not support preemption of tasks and it has to be decided if this kernel should be upgraded or another kernel should be used.

The basic parts of the software platform, e.g. drivers has been developed.

During the development of the software platform user guides should be made, to make it easier to develop the applications that interfaces with the software platform. This can also include a collection of test advice to use when testing the applications, since applications rely on the software platform.

The overall goal of this project is to make a working version of the software platform in order to test the project aims. This leads to the following goals or winning point for the project.

Figure 5.1: Principal sketch of the proposed software framework for AAUSAT3. The mechanical structure and hardware boards are simplified drawings, since these have not yet been developed.
• Software upgradability through CAN bootloader - The ability to dynamically upgrade software over CAN.

• Application execution flexibility/movability - The ability to move applications from one MCU to another without difficulties.

• Reusable software platform - By making the software general, it can be used for other satellites or other projects. One image of the software platform can be reused for all MCUs of the same type.

• Simpler programming of applications - By implementing a service layer that takes care of trivial tasks, it becomes easier to develop applications.

• Upgraded CSP - implementation of missing functionalities.

• Robust error handling - On a system level in general and in CSP error handling is needed to be sure the system are able to handle a broad spectrum of failures.

• User guide that people uses - A user guide that is easy to use, so that people will actually use it.

5.3 Functionality and Requirements

All requirements for the software framework can be traced back to the overall satellite requirement §5 stating that the satellite must be designed as a modular system consisting of subsystems connected via an internal communication system. This requirement, discussions made in the AAUSAT3 system Engineering Group (AEG) and the discussion in section 5.1 and 5.2 leads to the requirements in table 5.1.

| Req. ID | Change Date | Parameter Title | Requirement | Parent | Child | Author | Ref.
|---------|-------------|-----------------|-------------|--------|-------|--------|-------
| §5.1    | 2008-10-31  | Hardware        | The satellite shall contain multiple MCUs | §5     | N/A   | 08gr724 | AEG   |
| §5.2    | 2008-10-31  | Applications    | Functionality shall be divided into different software applications | §5     | N/A   | 08gr724 | AEG   |
| §5.3    | 2008-10-31  | Independence    | An application must be independent of other applications in the sense that if these are turned off it still runs | §5     | N/A   | 08gr724 | Sec: 5.1 |
| §5.4    | 2008-10-31  | Services        | There shall be a service layer that delivers services to the applications | §5     | N/A   | 08gr724 | Sec: 5.1 |
| §5.5    | 2008-10-31  | Communication   | Applications must communicate through CAN field bus | §5     | N/A   | 08gr724 | AEG   |
| §5.6    | 2008-10-31  | Reconfigurability | It must be possible to move applications between hardware platforms | §5     | N/A   | 08gr724 | Sec: 5.1 |
| §5.7    | 2008-10-31  | Coexistence     | Applications shall be able to coexist on the same hardware platform | §5     | N/A   | 08gr724 | Sec: 5.1 |
| §5.8    | 2008-10-31  | Updatability    | It must be possible to update software from earth | §5     | N/A   | 08gr724 | Sec: 5.1 |

Table 5.1: Requirements for the software framework. Reference to AEG means that the requirement has been made by the AAUSAT3 system Engineering Group.

5.4 Division of System Level Design

The software framework consist of three levels or parts, see figure 5.1. From the bottom up there is the bootloader, the software platform and the applications. This projects main focus is the bottom two parts of the whole software framework and therefore the development is divided into the following branches.
• **Kernel Selection:** This branch deals with the selection of a suitable kernel that supports preemption of tasks.

• **Protocol Stack:** The work here consists of the development of a service layer and an upgrade of CSP.

• **CAN Bootloader and Software Server:** This branch deals with development of a CAN bootloader and a server maintaining different software images.

The three branches together with the software that has already been written for the software platform, will make it possible to write a guide for development of applications, thus completing the work with the software framework.
For AAUSAT3, a kernel and scheduler is needed to allow software applications to be written as a set of independent tasks. The kernel is the part of the operating system that is responsible for task management, intertask communication and synchronization. The scheduler is a part of the kernel that is responsible for deciding which task to be executed and switching between tasks.

A kernel for AAUSAT3 called FLAK (Lightweight AAUSAT Kernel) has been developed. However this kernel does not support preemption of tasks. There are functions in the written communications protocol for AAUSAT3, that needs a preemptive kernel in order to work properly and the possibility to preempt a task will also make it easier and more flexible to develop application software. It would also be possible to utilize more of the CPU, since a non-preemptive kernel in worst case can have a task set with a utilizing approaching 0, that are still not possible to schedule. The backside of the preemptive kernel is that it uses more memory for the stack, since it needs a stack for each task. The time used for context switching also increases, because it changes task with a fixed period, where all registers needs to be pushed and pulled. The preemptive option will also require more comprehensive testing of applications, since the flow of the software is not that straight forward. However it has been decided that a preemptive kernel is needed.

The requirements for the kernel are specified in table 6.1. The parent of these requirements are §5.7 (Applications shall be able to coexist on the same hardware platform), except for requirement §5.7.2, which is inherited from the hardware choice.

<table>
<thead>
<tr>
<th>Req. ID</th>
<th>Change Date</th>
<th>Parameter Title</th>
<th>Requirement</th>
<th>Parent</th>
<th>Child</th>
<th>Author</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>§5.7.1</td>
<td>2008-10-31</td>
<td>Kernel</td>
<td>There must be a kernel</td>
<td>§5.7</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.7.2</td>
<td>2008-10-31</td>
<td>Hardware</td>
<td>The kernel must work on the AVR family</td>
<td>N/A</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.7.3</td>
<td>2008-10-31</td>
<td>Preemption</td>
<td>The kernel must support preemption of tasks</td>
<td>§5.7</td>
<td>N/A</td>
<td>08gr724</td>
<td>Cha: 6</td>
</tr>
<tr>
<td>§5.7.4</td>
<td>2008-10-31</td>
<td>Tasks</td>
<td>It must be possible to use more than one task for one application</td>
<td>§5.7</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
</tbody>
</table>

*Table 6.1: Requirements for the kernel.*
This chapter presents some possible kernels for AAUSAT3 with a comparison. On the basis of this and some testing, a kernel for AAUSAT3 is selected and a user guide is presented, in order to ease application development.

### 6.1 Possible Kernels for AAUSAT3

If preemptive features were added to FLAK it would be highly beneficial, since it will make the kernel easy understandable and is optimized for the specific purpose of controlling tasks on board a satellite. The problem is that it would probably take an entire semester to upgrade the kernel and there is not a group working specific on that. The kernel is needed now, in order to test it properly before launch and to have something to work with when developing the applications. Therefore it has been decided to look for a free kernel which has the needed functionalities.

The amount of available kernels is extensive and it will take a long time to look at them all, but at an AAUSAT3 meeting, FreeRTOS and AVRX has been mentioned as possible candidates, so a closer look at these will be presented.

#### 6.1.1 FreeRTOS

FreeRTOS is a free Real Time Operating System for embedded devices, developed by Richard Barry and the FreeRTOS team [Bar08].

It supports a lot of architectures, especially Atmel AVR (both 8 and 32) and ARM architectures (ARM 7, ARM 9, ARM Cortex-M3). There are many ported versions of FreeRTOS available with demos, which means it can easily be ported to both AT90CAN128 (the chosen AVR-8 MCU) and an AVR-32 MCU.

The system is reliable, free for use and FreeRTOS programmers are continually developing it and it has more than 6000 downloads per month.

The extensive documentation and the fact that FreeRTOS is predominantly written in C, makes it easy to modify and use.

Debugging can be done with AVR Studio and the JTAGICEmkII interface. FreeRTOS also provides a trace utility where it is possible to examine the different tasks execution time at runtime.

#### 6.1.2 AVRX

AVRX (short for AVR Executive) is a real time multitasking kernel written for the Atmel AVR series of MCUs. It is developed by Larry Barello [Bar07] and was last updated september 2005.

The kernel is written in assembler and only takes up about 700 words, which makes it very small. The data structure is compatible with C, so extensions can be written in C. However it is poorly documented thus making it harder to modify.

Debugging can (like with FreeRTOS) be done with AVR Studio and the JTAGICEmkII interface.

The fact that it is developed by one single person, the work on it seems to have stopped (last updated 2005) and the fact that it is written in assembler with poor documentation, makes it a bad choice as kernel for AAUSAT3, despite its very small size. AVRX is therefore disregarded as a possible kernel.

A comparison between FLAK and FreeRTOS is presented in the next subsection.
6.1.3 Comparison of the kernels

Table 6.2 shows some of the features and parameters supported by FLAK and FreeRTOS.

<table>
<thead>
<tr>
<th>Feature/Parameters</th>
<th>FLAK</th>
<th>FreeRTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relatively Easy to modify and port</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Non preemptive scheduling</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Preemptive scheduling</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Message passing/transfering primitives</td>
<td>CSP</td>
<td>Queues or CSP</td>
</tr>
<tr>
<td>Semaphores and mutex</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Documentation and forum support</td>
<td>Yes (08gr450)</td>
<td>Yes (<a href="http://www.FreeRTOS.org">www.FreeRTOS.org</a>)</td>
</tr>
<tr>
<td>Debug facilities</td>
<td>Yes (I/O pins)</td>
<td>Yes (trace)</td>
</tr>
</tbody>
</table>

Table 6.2: Features and parameters supported by FLAK and FreeRTOS. Trace is a debug facility for FreeRTOS to examine execution times for tasks.

The Preemptive feature and the extensive documentation and use makes FreeRTOS a very interesting kernel. However memory usage (SRAM and flash) and context switching and interrupt handling times still needs to be investigated. In order to test FreeRTOS it is first ported to AT90CAN128. This will also illustrate how easy it is to port.

6.2 FreeRTOS for AT90CAN128

This section contains a small description of the files included in the FreeRTOS kernel and a description of some minor known modifications in the files which is needed in order to make it work on the AT90CAN128 MCU.

6.2.1 Description of FreeRTOS kernel specific files

The FreeRTOS main functionalities is contained in tree source files (task.c, queue.c and list.c) unless the application utilize co-routines which functionalities is contained in a fourth file (croutine.c). The API for these four files is located in five header files which is listed here with a small description of the overall functionality:

- **task.h**: Functions for creating and controlling tasks.
- **queue.h**: Functions for creating and controlling queues.
- **list.h**: Functions for using lists. These functions is primarily used by the scheduler but can also be used by applications. A list can only store pointers to list items.
- **croutine.h**: Functions for creating and controlling co-routines. Due to the increased complexity of programming applications and the increased kernel memory size (almost 1 kiB), the functionalities of co-routines has been excluded. However it can with ease be implemented later if proved useful for applications.
- **semphr.h**: Creating and controlling semaphores and mutexes.

FreeRTOS has been designed to support multiple types of MCUs. In order to setup FreeRTOS to use a specific MCU, it is necessary to include an extra source file (port.c). The API for this file is contained in five header files which is is listed here with a small description of the overall functionality:
• freeRTOSconfig.h: Contains general defines which can be used to setup FreeRTOS constraints and include or exclude FreeRTOS functionalities dependent of the given MCU and application (see the configuration in section 6.2.2).

• freeRTOS.h: Extension of the freeRTOSconfig.h file which contains extra defines which is independent of the MCU and application.

• portmacro.h: MCU specific definitions used to configure FreeRTOS correctly for the given hardware and compiler.

• portable.h: Is primarily used to include the correct portmacro.h file based on the given MCU.

• projdefs.h: True, false and error definitions used by the FreeRTOS kernel.

Furthermore the FreeRTOS kernel can be implemented with three different memory management strategies separated into three different source files (heap_1.c, heap_2.c and heap_3.c).

• heap_1.c: Static memory allocation. Memory can not be freed once it has been allocated.

• heap_2.c: Dynamic memory allocation. Memory can be freed after it has been allocated.

• heap_3.c: A thread save implementation of malloc() and free() provided by libc.

### 6.2.2 Configuration of FreeRTOS

The following code is the chosen configuration of FreeRTOS for the AAUSAT3:

```c
65 #define configUSE_PREEMPTION 1
66 #define configUSE_IDLE_HOOK 0
67 #define configUSE_TICK_HOOK 0
68 #define configCPU_CLOCK_HZ ((unsigned portLONG) 8000000)
69 #define configTICK_RATE_HZ ((portTickType) 1000)
70 #define configMAX_PRIORITIES ((unsigned portBASE_TYPE) 4) // possible task priorities (high number uses more RAM)
71 #define configMINIMAL_STACK_SIZE ((unsigned portSHORT) 85) // The size of the stack used by the idle task.
72 #define stack_size ((unsigned portSHORT) 200) // The size of the stack used by tasks.
```

The stack size is the amount of SRAM allocated for a given task used in the application. The chosen stack size (200 bytes) is based on the results of test 8 (see section 6.3.9).
6.2.3 Modifying FreeRTOS to AT90CAN128

The FreeRTOS kernel supports multiple MCUs, including some 8 bit AVR MCUs. The AVR AT90CAN128 MCU however is not supported by FreeRTOS and thereby has to be modified. This has been done by using MCU dependent files (see section 6.2.1) from the 8 bit AVR ATmega323 MCU, which is supported by FreeRTOS.

In order to support AT90CAN128 in FreeRTOS, the following has been modified:

1. **Force FreeRTOS to include portmacro.h:**
   
   **Description:**
   
   FreeRTOS must include the `portmacro.h` in order to work.
   
   **Modification:**
   
   Add following line in `portable.h`:
   
   ```
   #include "portmacro.h"
   ```

2. **Use correct timer to generate a tick interrupt:**
   
   **Description:**
   
   FreeRTOS uses the timer TIMSK to generate a tick interrupt, which can not be used on AT90CAN128.
   
   In order to work, the used timer must be changed to TIMSK1.
   
   **Modification:**
   
   Replace the following lines in the source file port.c:
   
   ```
   ucLowByte = TIMSK;
   ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
   TIMSK = ucLowByte;
   ```
   
   With:
   
   ```
   ucLowByte = TIMSK1;
   ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
   TIMSK1 = ucLowByte;
   ```

3. **Set the correct bit to enable timer interrupt:**
   
   **Description:**
   
   Before the chosen timer can be used to generate tick interrupts, it is necessary to enable the timer interrupt by setting the timer register. This is not set correct by FreeRTOS.
   
   **Modification:**
   
   Replace the following lines in `port.c`:
   
   ```
   #define portCOMPARE_MATCH_A_INTERRUPT_ENABLE ((unsigned portCHAR) 0x10)
   ```
   
   With:
   
   ```
   #define portCOMPARE_MATCH_A_INTERRUPT_ENABLE ((unsigned portCHAR) 0x02)
   ```
Furthermore the following has been modified in order to get CSP to use the tick interrupts generated by FreeRTOS instead of using its own generated by the RTC driver:

4. Define rtc_tics as a global variable:

Description:
The tick counter (xTickCount) used in FreeRTOS is not defined globally and CSP can therefore not use xTickCount without calling a function.

Modification:
Add the following line in task.h:

```c
extern volatile unsigned long rtc_tics;
```

5. Copy the value of xTickCount to rtc_tics:

Description:
The global variable rtc_tics must always be the same as xTickCount.

Modification:
Add the following line in task.c:

```c
rtc_tics = xTickCount;
```

This synchronizes xTickCount and rtc_tics when the scheduler is initialized.

The following line must also be added in order to update rtc_tics when xTickCount is updated:

```c
rtc_tics = xTickCount;
```

### 6.3 Test of FreeRTOS

This section contains a short description of the test design, the general test setup and is then split up into subsections for each performed test. Instead of making one big test of FreeRTOS, 10 small test has been made, testing one feature at a time. This will hopefully make it easier to understand the test c-files and the idea was that the code from these files could be used as documentation for a user guide.

As mentioned earlier in this chapter, memory usage (SRAM and flash), context switching and interrupt handling times needs to be investigated in context with the AVR-8, since it has a limited amount of SRAM and computational power available. How to use the different API functions in FreeRTOS will also be investigated in the test.

#### 6.3.1 General test setup

Table 6.3 contains a list of equipment used for the test.

The list of files below are the c-files used for the different tests. Each c-file has a corresponding h-file. The files are documented with doxygen [JV08] and placed on the project web site.

- `test1_roundrobin.c`
- `test2_taskyield.c`
- `test3_mutex.c`
Equipment | AAU nr. | Type                          
------------|--------|------------------------------
2 x development boards | - | MCU: AT90CAN128 (made by 08gr414)  
Hameg power supply | 64341 | HM7042-5                       
USB to serial port cable | - | -                             
Pc with hyperterminal | - | Zepto 6625WD with Windows Vista  
Programmer | 77079 | AVRATJTAGICE-MKII             
Oscilloscope | 61603 | Agilent 54621A                
Probe | - | -                             

Table 6.3: Equipment list.

- test4_createtask.c  
- test7_queues.c  
- test7_queuesisr.c  
- test8_twotasksrr.c  
- test8_stackheapsize.c

The general approach in each test is as follows (the first two only needs to be done once):

- Connect the power supply to the development boards (3.3 V).
- Install WinAVR (AVR-GCC).
- Connect the AVR JTAG device to your pc and the hardware board and install the driver for the JTAG device, which is included in winAVR.
- Open the makefile and type in the name of the source file (for example test1_roundrobin).
- Open Hyperterminal or similar and open a connection via the USB to serial port cable, which should be connected between the pc and the UART connector on the development board. The baud rate is 38400.
- Run the following command in the command prompt: make clean all program
- Observe what happens (dependent on the specific test). An oscilloscope can be used to measure voltage on an I/O-pin for time measurement. Under time measurement all printf_P() calls are removed.

6.3.2 Test 1: Round Robin Check

This tests purpose is to check if the scheduler uses round robin, when tasks have the same fixed priority. It also illustrates a simple setup with two tasks running.

Two tasks are created both having idle priority plus one and with a period of 20 seconds. The scheduler uses time slices with 1 ms duration, which is the same as the RTC tick. In both tasks there is a while(1) loop making sure the task never ends. Within the while loop in task 1 an I/O-pin is set low and within the while loop in task 2, the same I/O-pin is set high. Despite the endless while loop in both tasks the scheduler never crashes, because the tasks are never put into the task queue more than once at startup. The 20 second period is just to pass a value to the function and it must be greater than 1 ms.

Figure 6.1 shows an oscilloscope picture of the voltage on the I/O pin.
6.3.3 Test 2: Priority Check and Task Yield

The purpose of this test is to check if the prioritizing of tasks works and to investigate if the task yields when it ends. Task yield means that if a task takes 0.2 ms to execute, the next task in the scheduler queue will start executing after the 0.2 ms (plus context switching time) and not after 1 ms, which is a full time slice.

The test file has 3 tasks with priority 1, 2, 3 above idle task priority. They all have a period of 1 ms and task 1 (highest priority) sets an I/O-pin low, task 2 sets the I/O-pin high and finally task 3 (lowest priority) sets the pin low again.

With an oscilloscope put on the I/O-pin it’s possible to measure the time between each task and all 3 tasks are executed a long time before 1 ms has passed. This indicates that the task yields after execution. If it did not task 2 and 3 would not run, since task 1 would get all the CPU time.

6.3.4 Test 3: Mutex and Priority Inheritance

Protection of data can be done with mutexes. If e.g. a task wants to update a global variable, it protects the update with a mutex. If it then gets preempted by a higher priority task which tries to take the mutex, the program will deadlock. In FreeRTOS this is avoided by priority inheritance, where the task that originally took the mutex inherits the priority of the task that tries to take it, which makes it possible for the first task to complete the code within the mutex and release it. When this is done the priorities are switched back again.

The purpose of this test is to check if the mutex system works and that the scheduler does not deadlock.

The test file has a quick task running with a period of 1 second and a slow tasks with a period of 4 seconds. They both take the mutex, but within the mutex of the slow task is a delay for 1.2 seconds, which will trigger a preemption within the mutex. A message is printed out on UART when a task starts and ends and when they give the mutex away. Below is what is printed on UART.

---

**Figure 6.1:** Output on I/O-pin during test 1. Low is when task 1 runs and high is when task 2 runs.

The tasks preempts each other as expected with a period of 1 ms (± a couple of µs), which indicates the use of round robin time slicing.
CHAPTER 6. KERNEL SELECTION

Quick task begin
Quick task gives mutex away
Quick task ended
Slow task begin
Quick task begin
Slow task gives mutex away
Quick task gives mutex away
Quick task ended
Slow task ended
Quick task begin
Quick task gives mutex away
Quick task ended

First the quick task runs its code and then slow task begins, but is preempted by the quick task. When quick task needs the mutex, slow task inherits the priority and gives the mutex away and then quick task can finish afterwards. Then the slow task finally ends. This shows that mutex with priority inheritance works.

6.3.5 Test 4: Create/Delete task runtime

The purpose of this test is to see how tasks can be created and deleted again within another task when the program is running. This functionality can be used if a large package needs to be send via the CAN bus. Then a low priority task could be created to send the data and likewise a receiving task can be created as well.

In the test one task with a period of 5 seconds first creates a task with a period of 200 ms and after a delay of 500 ms it deletes it again. Then it creates another task which deletes itself after completion. Everything done is sent to UART and the following is printed:

Start of task 1
Creating task 2
Start of task 2
End of task 2
Start of task 2
End of task 2
Start of task 2
End of task 2
Start of task 2
End of task 2
Deleting task 2
Task 2 deleted
Creating task 3
Start of task 3
End of task 3
End of task 1

An I/O-pin and an oscilloscope shows that it takes about 308 $\mu$s to create task 2 and 31 $\mu$s to delete it again.
6.3.6 Test 5: Interrupt Service Routine Handling

The purpose of this test is to examine how long it takes to execute an ISR and to determine if the ISR uses the tasks stack and if other interrupts are disabled during an interrupt.

All interrupts are assigned individual enable bits which must be written logic one together with the global interrupt enable bit in the status register in order to enable the interrupt. The AVR has an interrupt vector table where each interrupt has a priority. When an interrupt occurs, the global interrupt enable I-bit is cleared and all interrupts are disabled. The user software can write one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a return from interrupt instruction is executed [Atm08, p.15].

Since nested interrupts could mean long computation time for a low priority interrupt and an increased stack size, global interrupts should always be disabled when handling an interrupt. If other interrupts occur, while handling the first interrupt, they will set a flag and be serviced in order of priority, if the flag is not cleared before that.

Inside the interrupt it first pushes all the registers to the stack and when returning pops the registers back. The interrupt execution response time is minimum four clock cycles, where it pushes the program counter to stack and jumps to the ISR. The return from an ISR takes four clock cycles [Atm08, p.17].

In the lss-file for the tests, the number of instructions in an ISR can be found. If all registers needs to be pushed an popped it takes about 32-2 instructions and the push and pop instruction take 2 clock cycles [Atm08, p.416]. The CPU can both fetch one instruction and execute another at the same time, making it possible to do an instruction per clock cycle [Atm08, p.14], where one clock cycle at 8 MHz is equal to 125 ns. This means that an interrupt (with no code inside) roughly takes 72 instructions equal to 9 µs.

The 32 registers that are pushed onto the stack also takes up 64 bytes of SRAM.

6.3.7 Test 6: Context Switching Time

The purpose of this test is to examine how long it takes to switch between two tasks. This test uses the test file from test 2.

In the test file an I/O-pin is set low at the end of task 1 and set high again at the start of task 2. The oscilloscope shows a context switching time of 79.2 µs (50 µs/div, -width quickmess).

FLAK has a context switching time of 7.25 µs (same setup as with FreeRTOS, 5 µs/div, -width quickmess).

Comparing the two, FreeRTOS has a pretty high context switching time. Its hard to avoid a larger computation time with a preemptive kernel, code optimization might help this, as 79.2 µs is about than 633 instructions.

6.3.8 Test 7: Message Passing with Queues

The purpose of this test is to see how message queues can be used and to test how long it takes, to put a message in the queue and how long it takes to take the message of the queue again.

There are two test files (test7_queues.c and test7_queuesisr.c). test7_queues.c has a task that sends messages to another task with a message queue. The queue can hold 10 messages of 2 bytes each and the following is printed to UART:

messages waiting in queue: 0
messages waiting in queue: 2
Received value: 22
messages waiting in queue: 1
Received value: 22
messages waiting in queue: 0

Task 2 sends the value 22 two times to the queue and task 1 receives one item from the queue each time it runs.

test7_queuesisr.c has one task receiving messages from the queue and an ISR that is triggered by CAN which puts a message on the queue. Another board sends out CAN frames. This just illustrates how queues can be used from an ISR.

The time it takes to use queues are illustrated in table 6.4. All messages are 2 bytes long.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Timeout</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>xQueueSendToBack()</td>
<td>31.8 µs</td>
<td>=0</td>
<td>20 µs/div</td>
</tr>
<tr>
<td>xQueueSendToBack()</td>
<td>61.5 µs</td>
<td>&gt;0</td>
<td>20 µs/div</td>
</tr>
<tr>
<td>xQueueSendToBackFromISR()</td>
<td>18.2 µs</td>
<td>NA</td>
<td>20 µs/div</td>
</tr>
<tr>
<td>xQueueReceive()</td>
<td>31.7 µs</td>
<td>=0</td>
<td>20 µs/div</td>
</tr>
<tr>
<td>xQueueReceive()</td>
<td>61.0 µs</td>
<td>&gt;0</td>
<td>20 µs/div</td>
</tr>
</tbody>
</table>

Table 6.4: Functions and their computation time. The used messages size is 2 byte and the time is measured with an I/O-pin and oscilloscope. The timeout specifies how long the functions waits and if a timeout is specified the function will suspend and let other tasks run.

All functions starting with FromISR are light weight API functions, thus taking less time. If used outside an ISR, special care must be taken.

6.3.9 Test 8: Stack and Total Heap Size

The used memory management scheme (heap_2.c) will experience fragmentation of data if the same stack size is not used for every task. This is because the scheme uses a best fit algorithm that allows previously allocated blocks to be freed. However it does not combine adjacent free blocks into a single large block. If a task is created upon initialization and never deleted a random stack size within reasonable limits can be used.

The purpose of this test is to find a suitable stack and total heap size.

The test file test8_twotasksrr.c has two tasks which preempts each other and uses uxTaskGetStackHighWaterMark(). The stack of each task is filled with 0xa5 bytes upon creation which allows the high water mark to be viewed. uxTaskGetStackHighWaterMark() returns the lowest amount of free space on the stack in the time the task has been running. Task 2 prints the numbers to UART.

With a stack size of 200 it returns 145 for task 1 and 94 for task 2. Which means a basic setup with preemption of tasks and tick interrupts takes 55 bytes of RAM for stack and 106 bytes when printing to UART is added.

A stack size of 200 seems reasonable, in order to avoid stack overflow, but extreme care has to be taken from the software developers side.

Another important measure is total heap size. This number indicates the total amount of available SRAM for the kernel. The test program needs at least 730 bytes in order to avoid overflow. So in this case 400 is used for the two stacks and 330 is used by the kernel itself.
CHAPTER 6. KERNEL SELECTION

If a total of three applications can be placed on the same hardware and they all use three tasks, this amounts to a total heap size of about $9 \times 200 + 330 = 2130$ so a total heap size of 2200 has been set. More tasks could be added but then it would get close to the total of 4 kB of SRAM available.

Another test program (test8_stackheapsize.c) has 4 tasks running, and includes preemption, queues, UART, CAN messages (sent from other board) and mutex in order to stress the kernel a bit. This test shows 23 bytes as the lowest amount of free stack available to a task after running for 1 hour, indication 200 bytes for stack as reasonable, unless an application turns out to have special needs.

6.3.10 Test 9: Kernel Memory Usage

The purpose of this test is to see how much memory FreeRTOS uses and compare it to how much FLAK uses.

Table 6.5 contains used memory for different source files. Optimization for size has been used.

<table>
<thead>
<tr>
<th>File Name</th>
<th>text [bytes]</th>
<th>data [bytes]</th>
<th>bss [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>flak.o (FLAK)</td>
<td>1116</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>heap_2.o (FreeRTOS)</td>
<td>362</td>
<td>0</td>
<td>2213</td>
</tr>
<tr>
<td>list.o (FreeRTOS)</td>
<td>308</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>port.o (FreeRTOS)</td>
<td>628</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>queue.o (FreeRTOS)</td>
<td>1726</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tasks.o (FreeRTOS)</td>
<td>3140</td>
<td>5</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 6.5: Amount of memory used by different source files. Text is bytes used in flash, .data is bytes used in SRAM and .bss is bytes used for static variables initialized to zero by the programmer.

FLAK uses 1116 bytes of flash, 7 byte of SRAM.

FreeRTOS uses 6164 bytes of flash, 2316 bytes of SRAM.

The high amount of used SRAM in FreeRTOS are due to total heap size, where 2200 bytes are allocated (two tasks with kernel uses about 730 bytes). This number can be altered if necessary.

6.3.11 Test 10: Test Program Memory Usage

The purpose of this test is to see how much memory the program from test 8 uses (test8_stackheapsize.c).

With everything included the program takes up 19114 bytes in flash (14.6%) and 3515 in SRAM (85.8%). This is pretty close to maximum SRAM available, but a lot of it is allocated for unused stacks (4 out of 9 stacks used).

6.4 SRAM Memory Management

After working with the kernel for a while, it has been decided to use heap_3.c for memory management instead of heap_2.c. heap_3.c just makes sure to suspend all tasks and resume them again when using malloc() and free(). Another difference is that with heap_3.c, total heap size is not defined. Instead stacks, queues, semaphores and everything that is dynamically allocated with malloc() will be placed after the .bss section. The differences in the memory management are shown in figure 6.2 [AVR08].
The stack pointer (SP) grows from highest memory address to lower addresses [Atm08, s.14] and the SRAM addresses of the AT90CAN128 MCU are from 0x0100 (256) to 0x10FF (4351) [Atm08, s.18]. The reason not to use heap_2.c (top picture) is because total heap size just splits the memory not used for .data and .bss into two parts. One for stacks, queues and semaphores created by FreeRTOS and one section used by malloc().

malloc() does not know where to stop allocating, but this can be passed to the linker with the following line [AVR08]:

```
-WL--defsym=__heap_end=0x8010ff
```

If malloc() exceeds this boundary (address 0x10FF) it will return NULL and a software reset with the watch dog timer can be performed, since a crash is very likely to occur if the pointer is used (the pointer points outside internal SRAM).

It is not desirable to use malloc() runtime, since its non deterministic, non reentrant and can cause the system to crash if to much memory is allocated. The memory can also be fragmented if different amounts of memory is continuously being allocated and freed. The free() function can however merge adjacent freed blocks into larger blocks. If possible malloc() should only be used during initialization and free() should never be used.

A small test program with two tasks which allocates memory gives the following result with heap_3.c (see also test file memory_test.c):

- pointer1: 4344 //address of local variable during init (before tasks are created)
- pointer2: 977 //address returned by malloc(1) during init (before tasks are created)
- pointer3: 1197 //address of local variable created in task 1 (stack size 200)
- pointer4: 1198 //address of second local variable created in task 1
- pointer5: 1431 //address of local variable created in task 2 (stack size 200)
- pointer6: 1432 //address of second local variable created in task 2
- pointer7: 1580 //address of pointer returned by malloc(1) in task 2
- pointer8: 1584 //address of pointer returned by xQueueCreate() in task 2
- pointer5: 1431 //same as before
- pointer6: 1432 //same as before
- pointer7: 1621 //...
- pointer8: 1625

...
Since `free()` is never used the SRAM is eventually used up and `malloc()` returns NULL. The test program verifies figure 6.2.

### 6.5 Conclusion on Kernel Selection

It has been stated that a preemptive kernel is needed for the distributed system on AAUSAT3. The already developed kernel (FLAK) does not support preemption of tasks and since it will probably take a semester to upgrade it, two other possibilities has been investigated. The first is AVRX and the second is FreeRTOS. FreeRTOS is considered the best choice among these two, because it is easier to port and modify, is better documented and more widely used.

A more elaborate test of FreeRTOS shows that it can be used with an 8-bit AVR and that is has some good build in functionalities like message passing with queues and data protection with mutexes.

Some of the downsides of FreeRTOS are longer context switching times and higher memory usage. Especially usage of SRAM gets close to the limit, due to the high amount of memory allocated for stacks.
The program developer must be aware of the danger of stack overflow and take it into consideration when designing software.

As a conclusion, FreeRTOS will be used in order to meet the requirement stating that we need a preemptive kernel for the distributed system on AAUSAT3, see requirements in table 6.1.

A user guide for FreeRTOS will be presented in the next section.

6.6 User guide for FreeRTOS

The idea behind this user guide for FreeRTOS is to make it easier for software developer to write their programs and to make sure to pass on important knowledge gained during the FreeRTOS testing. This user guide is also placed on trac so people working with AAUSAT3 can find it easily.

The user guide is divided into general information, task handling, mutexes and queues, respectively.

6.6.1 General Information

- **heap_end**: The end address of internal SRAM should be passed to the linker with the following line (malloc() returns NULL when passing this address):

  ```
  -Wl,--defsym=__heap_end=0x8010ff
  ```

- **malloc() and free()**: free() should in general not be used and if malloc() is used be sure to check if it returns NULL and if so do not use the pointer.

- **Used memory**: The compiler indicates the amount of SRAM used by the program, but this amount does not include queues and semaphores which are created under initialization, nor memory allocated for stacks.

- **Stack size**: Is set to 200 bytes, but is application dependent and hard to calculate. It depends on the function call depth, number of local variables allocated, number of parameters in function calls made, interrupt stack requirements, etc.

- **ISR**: ISR uses the stack and can push up to 60 bytes of registers on the currently used stack. The interrupts are currently not nested, but it is possible to enable it on the MCU.

- **Stack or heap overflow**: Program crash is likely to be due to stack or heap overflow. Use uxTaskGetStackHighWaterMark() to check the amount of free stack during runtime.

- **Use of queues**: Queues should be used for message passing between tasks, between ISR and task and not between applications, since applications can be moved around. Be aware that creation of a queue both takes SRAM for the queue structure and and the queue storage area.

- **Use of mutex**: Mutex gives the possibility of protecting data and should generally be used when updating global variables.

6.6.2 Main Function

The main function of each application must always create at least one task before starting the FreeRTOS scheduler. It is then both possible to create all the tasks before starting the scheduler or using the one created task to dynamically create more tasks within the scheduler. A combination is also possible.
In order to create a task simply use \texttt{xTaskCreate()}.  
\begin{verbatim}
xTaskCreate(task1, (const signed portCHAR *) "Task 1", stack_size, NULL,  
TASK1_PRIORITY, &task1_handle);
\end{verbatim}

The above shown creates a task using code defined in \texttt{task1()}. The task is given the name Task 1, which is optionally. The maximum length of the name is defined in the FreeRTOS configuration file. This task only uses the minimal task stack size and do not parse any user defined parameters. The task has \texttt{TASK1_PRIORITY} and is given a handle called \texttt{task1\_handle}.

It is important to remember that \texttt{stack\_size} is the maximum stack size for each task (200 bytes). If more stack is required by a task just increase the stack size locally for the given task.

Also remember to define the \texttt{task1\_handle} as an \texttt{xTaskHandle}. This should be done in the application header file.

\begin{verbatim}
xTaskHandle task1\_handle;
\end{verbatim}

The scheduler can be started by using \texttt{vTaskStartScheduler()}.  
\begin{verbatim}
vTaskStartScheduler();
\end{verbatim}

When the scheduler is started it will never return.

\section*{6.6.3 Structure of Tasks}

A task can be structured in many different ways but has one basic constrain. It must never end.

Below four different basic structures of a task is given. These are useful in many scenarios.

The simplest structure of a task is to put it in a while loop.

\begin{verbatim}
void task1(void *pvParameters){  
  while(1){  
    /* Place you code here */  
  }  
}
\end{verbatim}

This task will always run unless it is interrupted by a higher priority task, a task with the same priority or an ISR. Keep in mind that a task with this structure properly should have a low priority.

If a task should only run once without a periodical time interval, the following structure could be used.

\begin{verbatim}
void task2(void *pvParameters){  
  while(1){  
    /* Place you code here */  
    vTaskSuspend(NULL); // Suspend until woken by vTaskResume(task2\_handle)
  }  
}
\end{verbatim}

\texttt{vTaskSuspend()} puts a task in the suspend queue until the task is resumed by another task calling \texttt{vTaskResume()}. Note that a task also can suspend other tasks than itself. It is also possible to resume a task from an ISR by using \texttt{vTaskResumeFromISR()}. 

If a task should run with a periodical time interval, the following structure could be used.

\begin{verbatim}
void task3(void *pvParameters){  
  portTickType last_start_time;
  const portTickType ticks = 4000; // 1 KHz tick -> 4 s delay
\end{verbatim}

48
CHAPTER 6. KERNEL SELECTION

```c
last_start_time = xTaskGetTickCount(); // Last time where task was blocked
while(1){
    — Place you code here —
    vTaskDelayUntil(&last_start_time, ticks); // Delay for 4 s
}
```

By using `vTaskDelayUntil()` last in the while loop, the task is suspended for a defined time interval. `vTaskDelayUntil()` updates last start time itself, thereby it is only necessary to define last start once before entering the while loop.

Note that instead of using `xTaskGetTickCount()`, the global variable `rct_tics` can be used to get the current tick count.

Another structure of a task, which should only run once and then not be used for a long period of time, can be.

```c
void task4(void *pvParameters){
    while(1){
        — Place you code here —
        vTaskDelete(NULL); // Deletes itself
    }
};
```

This is almost equal to the second mentioned structure where the task was suspended instead of deleted. Be careful when an application is dynamically creating and deleting tasks. If the stack size differs between the different tasks it is likely that memory fragmentation will occur.

Also note that if user parameters is parsed into a task, the application itself has to free the memory used by the parameters using `vPortFree()` as this is not done by `vTaskDelete()`.

If the application has no problem with the amount of tasks used, it is recommended to use `vTaskSuspend()` instead of `vTaskDelete()`. This recommendations is based on the amount of time it takes to create a task vs. the time used to move the task back from the suspend queue.

### 6.6.4 Data Protection with Mutex

The FreeRTOS kernel includes mutex to protect critical data from being accessed by two tasks at the same time. Furthermore FreeRTOS includes priority inheritance in order to minimize the risk of deadlocks.

A mutex can be created by using `xSemaphoreCreateMutex()`.

```c
data_mutex=xSemaphoreCreateMutex();
```

This will create a mutex called `data_mutex`. This should be defined, in the application header, as a `xSemaphoreHandle`.

```c
xSemaphoreHandle data_mutex;
```

In the following, three basic methods of using the mutex is presented.

The simplest method is to try to obtain the mutex and just continue, without obtaining critical data.

```c
if ( xSemaphoreTake(data_mutex, 0) != pdFALSE){
    — The mutex is obtained —
    xSemaphoreGive(data_mutex); // Free mutex
```
The mutex should always be freed as soon as possible. If a mutex is obtained for a long time, the risk of another task missing its deadline is increased.

If a mutex is already obtained by another task, it is possible to specify a timeout to wait before continuing without obtaining the mutex.

If the mutex is already obtained, the calling task (task 2) will be put in the suspend queue until it times out or the mutex is freed by the other task (task 1). If task 2 has a higher priority than task 1, this will inherit the priority of task 2 until the mutex is freed or the call from task 2 times out.

A calling task can also have an indefinite timeout using \texttt{portMAX\_DELAY}.

Note that critical data can still be accessed outside the mutex. Thereby it is the job of the application designer to ensure that critical data is always accessed through a mutex.

\section{Message Passing with Queues}

In order to parse a message between two tasks or an ISR and a task it is possible to use queues.

To create a queue simply use \texttt{xQueueCreate().}

This will create a queue with the name \texttt{queue1} which can contain 10 unsigned integers. Again remember to define the queue in the application header file.

In the following it is shown how to send a message to the queue and how to get a message from the queue.

In order to send a message to the queue \texttt{xQueueSendToBack()} can be used.
The above code uses queue1 to send a message using a pointer (pointcounter). The task will only try to put the message onto the queue once. If the queue is full it will continue without putting the message onto the queue.

Furthermore it is possible to specify a timeout when using xQueueSendToBack().

If the queue is full the task is blocked until it times out or space becomes available in the queue. If the task should block indefinitely until space becomes available in the queue, use portMAX_DELAY instead of the timeout.

Instead of using xQueueSendToBack() it is also possible to use xQueueSendToFront().

In order to receive a message from the queue, xQueueReceive() can be used.

uxQueueMessagesWaiting() returns the number of messages in the queue. If it returns a nonzero number xQueueReceive() will make one attempt to receive the data.

Like in xQueueSendToBack() it is possible to define a timeout in xQueueReceive() and to make the task wait indefinitely until the message is received from the queue using portMAX_DELAY.

Note that if the application is sending or receiving from an ISR, then xQueueSendToBack(), xQueueSendToBack() and xQueueReceive() functions must be replaced with xQueueSendToBackFromISR(), xQueueSendToBackFromISR() and xQueueReceiveFromISR() respectively.
This chapter describes the development of a protocol stack for AAUSAT3. A protocol called CAN Space Protocol (CSP), which communicates over CAN via a CAN driver, has already been developed. These two layers are modified to work on the FreeRTOS kernel, to implement hardware filtering and to implement a loopback functionality. Needed services are also implemented and gathered in a so called service layer.

The chapter starts with an analysis of the structure of the protocol stack and need-to-have/nice-to-have functionalities. This leads to a limitation stating what is done in the scope of the project and a requirements specification.

### 7.1 Protocol Stack Analysis

The wanted structure of the protocol stack is illustrated with two nodes on figure 7.1.

This structure has been chosen because the lowest three levels already deliver some well-defined API functions which the applications can use directly, see chapter 3. The idea is that multiple applications can coexist on top of the lowest three levels. The service layer will take care of functionalities that will be provided by all nodes. Thus creating common software which can be reused and be a part of the software platform. Another idea would be to encapsulate the function calls in CSP to create an even easier to use API, but this would reduce the flexibility and give another software layer, meaning more computation time and flash usage, which is why this will not be done.

A list of possible modifications to CSP and the CAN driver, including ideas for the service layer is presented below. Some of them has already been mentioned in chapter 3 but is presented again to provide a complete list. The list is also prioritized with P1 corresponding to most important.

**P1 Addressing:** There are currently 16 node addresses and 8 ports available for addressing. An addressing scheme must be implemented, which might require more ports.
Figure 7.1: Layers in the proposed protocol stack for AAUSAT3. The illustration shows two nodes on a CAN bus.

P1 **Hardware address filter**: At the moment there is no hardware filter implemented, meaning every frame send over CAN will interrupt all the nodes on the bus. A broadcast address can be added if needed.

P1 **Loopback**: A decision must be made if all communication should go through the CAN bus or if it should be allowed to pass messages between applications placed on the same MCU internally e.g. with queues. The chosen scheme should also be implemented.

P1 **Stack and heap overflow**: To increase robustness it would be beneficial to check up on stack and heap overflow errors, if possible, and do something about it.

P1 **Application control (service layer)**: A good functionality would be to stop and start single applications from other applications. This could also include a heartbeat functionality, where an application requests the state of another application.

P1 **Malloc/free**: These functions are currently in use but can cause errors, especially when allocating memory, since its not guaranteed that memory is available. When the whole software system gets closer to completion, it might be better to statically allocate memory for the needed connections. This will also make the system more deterministic, since the randomness in the time used for allocating and freeing memory will be removed.

P2 **Connection-less communication**: Implement the possibility of sending frames without making a connection. This should be a stripped down version of normal communication or an entire new protocol which can be implemented next to the CSP protocol.

P2 **Error state for CAN**: If a device is responsible for a lot of errors it will go into bus off mode, but nothing is done if this happens. It would be beneficial if this could be handled in a better manner, so the device could be recovered and go into error active mode again.

P2 **Remote frame type**: Implement remote frame type if needed.

P2 **Connection list**: Rewrite some of the functions in CSP so they work with an array of connections, instead of having multiple for loops to run through the all ports for the connections. This will make the code more simple.

P3 **CSP ping (service layer)**: A ping service can be implemented as debug tool, which makes it possible, in space, to check if CSP is responding. This is useful if the application control or the scheduler does not seem to work, if this ping service does not use tasks or queues.
P3 **Time synchronization (service layer):** With a real time clock with backup power it is possible to maintain a global time on an MCU for synchronization. A service can be added to update the global time, on a given MCU, received from a single application which is connected to a real time clock.

P3 **Printf using CAN:** An idea is to use the current printf_P() function that prints error messages to UART, and change it to use CAN. This will make it easy to send errors and state changes to LOG. This function should be a part of the software platform library.

P3 **CSP control (service layer):** An idea would be to control CSP/CAN, so it can be put in e.g. listen only mode.

P4 **Garbage collector MOB:** It might be an idea to do something more graceful instead of having the garbage collector MOB reset the CAN when it gets a frame.

P4 **Frame types:** It is possible to have more frame types, so some of them could be used to indicate what type of error occurred, which could result in better error handling.

P4 **Sequence error handling:** Right now the whole message will be dumped if one frame is missing and that is problematic if it is a long message. The problem is however not that comprehensive, since sequence errors are very seldom.

Modifications to CSP and the CAN driver are prioritized high because it is important that it works well before services are implemented, since they will use CAN communication.

Since a usable kernel for AAUSAT3 needs to be found and tested and a bootloader with image server also needs to be developed by the project group, everything in the list above cannot be developed. Therefore a limitation is made picking out the most important parts. The list above is put on the AAUSAT3 Trac website so issues which is not addressed in this project can be taken up at a later stage.

The main focus for this project is the addressing issues. This includes implementing a hardware filter, deciding how to address applications and to implement loopback functionality.

The project will further more address issues regarding modification of the CSP protocol in order to make it work with the FreeRTOS kernel and development of a service layer, which must implement CSP ping, application control, application heartbeat and time synchronization services. In connection with this stack and heap overflow handling will be examined.

Removing `malloc()`/`free()` and implementing connection list will not be a part of the work within this project, since this work is easier when the requirements set by the applications are known. CSP control is not on the list either, because application control can be used to stop a node from sending by stopping the applications. Printf using CAN should be implemented at some point, but its not that important in this phase of the satellite development.

In the next section all the functionalities and requirements will be presented.

### 7.2 Functionality and Requirements

From the previous section a list of requirements has been derived. These and previous requirements, decided by the AAUSAT3 Engineering Group (AEG), for the CSP protocol and the CAN driver is summarized in the following the three tables.

Table 7.1 states the requirements for the service layer.
Table 7.2 gives the requirements for the CSP protocol and communication through CAN.
Table 7.3 gives the requirements for the CAN driver.
### Table 7.1: Requirements for the service layer.

<table>
<thead>
<tr>
<th>Req. ID</th>
<th>Change Date</th>
<th>Parameter Title</th>
<th>Requirement</th>
<th>Parent</th>
<th>Child</th>
<th>Author</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>§5.4.1</td>
<td>2008-11-10</td>
<td>Heartbeat</td>
<td>The service layer must update the status of an application and send the status of an given application if it is requested</td>
<td>§5.4</td>
<td>N/A</td>
<td>08gr724</td>
<td>Sec 7.1</td>
</tr>
<tr>
<td>§5.4.2</td>
<td>2008-11-10</td>
<td>Global time</td>
<td>The service layer must be able to receive and update a global time</td>
<td>§5.4</td>
<td>N/A</td>
<td>08gr724</td>
<td>Sec 7.1</td>
</tr>
<tr>
<td>§5.4.3</td>
<td>2008-11-10</td>
<td>Application control</td>
<td>It must be possible to suspend and resume an application from another application</td>
<td>§5.4</td>
<td>N/A</td>
<td>08gr724</td>
<td>Sec 7.1</td>
</tr>
<tr>
<td>§5.5.1</td>
<td>2008-11-10</td>
<td>CSP ping</td>
<td>CSP must be able to reply to a ping without using the kernel</td>
<td>§5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td>Sec 7.1</td>
</tr>
</tbody>
</table>

### Table 7.2: Requirements for the CSP protocol and communication through CAN. Reference to AEG means that the requirement has been made by the AAUSAT3 Engineering Group.

<table>
<thead>
<tr>
<th>Req. ID</th>
<th>Change Date</th>
<th>Parameter Title</th>
<th>Requirement</th>
<th>Parent</th>
<th>Child</th>
<th>Author</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>§5.5.2</td>
<td>2008-10-31</td>
<td>Hardware</td>
<td>CSP must work on Atmel AVR (AT90CAN128) &amp; Atmel ARM7 (AT90SAM7A1) “TBD”</td>
<td>§5.5</td>
<td>§5.5.2.1</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.3</td>
<td>2008-10-31</td>
<td>Addressing</td>
<td>An application must be able to address all other applications including the ground station</td>
<td>§5.5</td>
<td>§5.5.3.1</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.4</td>
<td>2008-11-10</td>
<td>Monitoring</td>
<td>It must be possible to log all communication between applications</td>
<td>§5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.5</td>
<td>2008-10-31</td>
<td>Streaming</td>
<td>The communication protocol must be able to fragment and de-fragment streams of data</td>
<td>§5.5</td>
<td>§5.5.5.1</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.6</td>
<td>2008-10-31</td>
<td>Data handling</td>
<td>Data is either fully received on the bus, or dropped by CSP</td>
<td>§5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.7</td>
<td>2008-10-31</td>
<td>Connection</td>
<td>The protocol must support multiple connections between applications</td>
<td>§5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.8</td>
<td>2008-10-31</td>
<td>Unique addresses</td>
<td>Each application service must have an unique address</td>
<td>§5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
</tbody>
</table>

### Table 7.3: Requirements for the CAN driver. Reference to AEG means that the requirement has been made by the AAUSAT3 Engineering Group.

<table>
<thead>
<tr>
<th>Req. ID</th>
<th>Change Date</th>
<th>Parameter Title</th>
<th>Requirement</th>
<th>Parent</th>
<th>Child</th>
<th>Author</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>§5.5.1.1</td>
<td>2008-11-10</td>
<td>Hardware</td>
<td>The driver must make use of the CAN hardware on the Atmel AVR and the Atmel ARM7 MCU “TBD”</td>
<td>§5.5.2</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.3.1</td>
<td>2008-11-10</td>
<td>Message priority</td>
<td>It must be possible to give a message higher priority than another on the bus, independently of the destination or sender address</td>
<td>§5.5.3</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.3.2</td>
<td>2008-11-10</td>
<td>Unique addresses</td>
<td>Every MCU must have one or more unique addresses</td>
<td>§5.5.3</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.4.1</td>
<td>2008-11-10</td>
<td>Message masking</td>
<td>It must be possible to mask out messages based on a destination addresses</td>
<td>§5.5.3</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
<tr>
<td>§5.5.5.1</td>
<td>2008-11-10</td>
<td>Streaming</td>
<td>It must be possible to send streams of data</td>
<td>§5.5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td>AEG</td>
</tr>
</tbody>
</table>
7.3 Modifications of CSP and CAN Driver

In order to meet the above mentioned requirements, it is necessary to modify both the CSP protocol and the CAN driver. The CAN driver must be modified to include hardware addressing, while CSP must be modified to work on the FreeRTOS kernel and to introduce a loopback functionality, in order to make it possible for two applications, placed on the same MCU, to communicate with each other.

7.3.1 Hardware Filter and Ports

In order for an application to communicate with other applications, in a distributed system consisting of multiple MCUs, it is essential that each MCU is separated from the others by an unique id. Otherwise the applications running on the different MCUs do not know which MCU to address in order to communicate with another given application.

The communication between the MCUs is based on the CSP, which has 16 unique CAN addresses. By using the CAN addresses to separate each MCU it is possible to filter messages for every MCU using mobs. These match each identifier up against a predetermined identifier which is designed by the software developer. Furthermore it is possible to define which specific bits in the identifier the MOBs should use to filter on (masking bits). Table 7.4 gives an example on how the hardware filtering works.

<table>
<thead>
<tr>
<th>Received identifier</th>
<th>Predetermined identifier</th>
<th>Masking bits</th>
<th>MOB choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010 1001 0011</td>
<td>0x000 1001 0000</td>
<td>0x000 1111 0000</td>
<td>Accepted</td>
</tr>
<tr>
<td>0x010 1000 0011</td>
<td>0x000 1001 0000</td>
<td>0x000 1111 0000</td>
<td>Rejected</td>
</tr>
<tr>
<td>0x010 1000 0011</td>
<td>0x000 1001 0000</td>
<td>0x000 1110 0000</td>
<td>Accepted</td>
</tr>
</tbody>
</table>

Table 7.4: Example of how to use the hardware filtering functionality in CAN. Only the first 11 bits of the CSP identifier has been used to illustrate the hardware filter functionality. Spaces marks the fields in the CSP identifier (priority, destination address, source address).

Table 7.4 shows that the hardware filter uses the four destination address bits to filter messages. In the fist case all of the four destination bits of the received identifier is identical to the predetermined identifier and accordingly the message is accepted. In the second case, the last bit in the destination address field is not identical to the predetermined identifier and as a result the message is rejected. In the last case, only three of the destination address bits is used to filter messages. This means that the received message, mentioned before, will be accepted.

By using fewer bits to filter on, it is possible to allow more messages for a given MCU, which thereby doubles the number of available ports for the MCU. The list of available addresses is given in table 7.5. The reason for placing COM 1 and COM 2 in the upper half of the list is that messages originally intended for COM 1 easily could be changed to use COM 2, and vice versa, by changing the number of masking bits.

The CSP protocol is port oriented meaning that if an application have to use a function (service) of another application, it has to access a port providing the given service. By introducing a service layer and the wanted services, four ports must be reserved on each MCU (see section 7.4). By also introducing a bootloader, an additional port must be reserved on each MCU for flashing images (see section 8.4).

The previous version of the CSP protocol was designed to support 8 ports using three bits in the identifier. This means that, after implementing service layer and bootloader functionality, only three ports is left for application services and outgoing connections.

In order to increase the number of ports available in the CSP protocol, the first byte of the data field could be used to specify the destination of the message. This would correspond to having 128 (four extra bits
for the destination port field and four extra bits for the source port field) available ports on each MCU, but with the expense of only having a data size of 7 bytes in each CAN frame.

In order to maintain full data size for a CAN frame, the number of bits used in the frame type field has been reduced to 3 bits and the message length has been reduced by 3 bits. This induces that 4 bits can be distribute between the destination port field and the source port field, thereby increasing the number of available ports to 32. The structure of the identifier is illustrated on figure 7.2.

![Figure 7.2: Structure of CSP identifier supporting 32 ports.](image)

Due to the decreased number of bits, the message length field has been changed to represent the frame number instead of representing the byte number. This reduces the max package size from 255 bytes to 248 bytes (31 frames). Furthermore the number of frame types is reduced to eight.

By changing the message length field to represent the frame number it is necessary to send the smallest data frame first if the packet length is not divisible with eight. By sending the smallest data frame first it is possible use both the message length field (CSP) and data length field (CAN) to get a resolution of bytes instead of frames for the package size.

The list of available ports and the use of each port is summarized in table 7.6.

<table>
<thead>
<tr>
<th>Port number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bootloader command</td>
</tr>
<tr>
<td>1</td>
<td>CSP ping</td>
</tr>
<tr>
<td>2</td>
<td>Application control</td>
</tr>
<tr>
<td>3</td>
<td>Application heartbeat</td>
</tr>
<tr>
<td>4</td>
<td>Time synchronisation</td>
</tr>
<tr>
<td>5-7</td>
<td>Reserved service ports</td>
</tr>
<tr>
<td>8-23</td>
<td>Application ports</td>
</tr>
<tr>
<td>24-31</td>
<td>Outgoing ports</td>
</tr>
</tbody>
</table>

Table 7.6: List of available ports and the use of each port.

Both the hardware filter and changes made in order to increase number of ports to 32, have been tested by changing the destination address in the sent messages and observe if is was received or not. The number of masking bits has also been lower to see if it was possible to have multiple addresses on one MCU. The test has shown that the hardware filter is working and that 32 ports is supported in CSP.
7.3.2 Callback Handling

By introducing a preemptive kernel it is important that most of the code is running in a task, in order to minimize time spent in interrupts. CSP is interrupt based and must accordingly be executed as fast as possible, which generally is not an issue, but must be considered in the timeout and callback handling. Consequently the application developer should design the callback function to parse the returned connection pointer to a queue, which can be handled by a task and then return from the ISR. Preferably all callback should be put on the same queue, thus only one service task is needed to handle service request.

7.3.3 Connection Pool

In the original implementation of the CSP, each port was associated with a connection. This configuration is however not applicable for a large number of ports due to the amount of SRAM utilized to each connection. Therefore a connection pool has been implemented, with eight statically allocated connection structs. When a connection is established, one of the unused connection structs from the pool is taken. The advantage of the connection pool is that the amount of SRAM used is considerably less. The disadvantage is the possibility of expending all the connections, causing package loss.

To prevent package loss, the connection pool could be implemented as a linked list, where extra connection structs could be added dynamically. This should however be avoided if possible in order to maintain robustness of the software. Hence two extra connections have been statically allocated for use of highest priority packages, only to reduce the risk of losing high priority packages.

7.3.4 Timeout Handling

When using the functions `csp_zsend()` and `csp_zread()` a timeout can be specified.

When specifying a timeout for `csp_zsend()` the idea is that the function waits for an ACK from the receiver, indicating that the data was well received and has been handled. If `csp_zsend()` is used again before the data is read from the receive buffer, this data will be overwritten, which was probably not the intent. The timeout tells how long the function should wait before it generates an error, if no ACK was received. If only a single package needs to be send, there is however no need to wait, unless the sender wants to know if the transaction went well. The timeout for `csp_zread()` indicates how long the function should wait for data to become handled and acknowledged.

Originally the timeout handling was written for FLAK, which is a non-preemptive kernel. Here the functions polled on a state until the timeout expired. This implementation is not suited for a preemptive system. The functions should suspend during the waiting time defined by the timeout, and resume when an action occurs, like receipt of an ACK frame or a timeout expire. This will let other tasks run during the waiting time.

Timeout functionality is implemented with binary semaphores provided by FreeRTOS. When `csp_zsend()` has sent the first frame, it first takes a semaphore and then tries to take it again with a timeout. This will suspend the task running `csp_zsend()` until the timeout expires or until an ACK frame is received, giving the semaphore and making it possible for `csp_zsend()` to finally take it. The same is done for `csp_zread()`, except that the semaphore is given when the rxstate changes to RX_DONE.

Each connection struct has a tx and a rx semaphore created during initialization of CSP. When exiting `csp_zsend()` and `csp_zread()` the semaphores are always given away, to make sure they are ready for next call.
The original END frame has been removed. On receipt of an END frame, csp_conn_close() was called within an ISR. This introduced the risk of closing a connection struct while functions are still using it. Instead csp_conn_close() must be called between receipt of packages on the server side. This does however not mean that the connection is actually closed. If the connection should be closed on the server side, the client should send a BEGIN frame, and if the connection should not be closed a BEGIN_ALIVE frame is sent. The parameter keep_alive parsed to csp_zsend() determines if a BEGIN or BEGIN_ALIVE frame is sent.

Figure 7.3 shows a sequence diagram for communication between a client and a server. Here the client requests a service and gets a reply back from the server. The connection is not closed during the first transaction, but after the second transaction, which is controlled by csp_zsend().

The test files client_timeout_test.c and server_timeout_test.c has been used to test the implementation of the timeout handling. When setting a low timeout for csp_zsend() and csp_zread() they both time out and change state to SOCKET_ERROR, and when time out is increased they do not time out. The files also shows that csp_conn_close() works properly.

7.3.5 Communication Between Applications Placed on the Same MCU

Applications communicate with each other through CSP. In a normal scenario a message will be passed from CSP down to the CAN driver and onto the CAN hardware, through the CAN wire and up through a similar protocol stack on the receiving node/MCU. This however becomes a problem when two applications are placed on the same MCU. The current CAN driver cannot send and receive at the same time. This problem could potentially be solved by making a virtual link between the applications inside the MCU or by modifying the driver. This raises another question: Should all communication go trough the CAN bus, or should it be allowed to pass messages inside the MCU, that is hidden from the CAN bus?

The reason for putting all communication on the CAN bus is to be able to sniff the network for traffic, which can be used when debugging. Another reason would be to maintain timings between applications. If communication suddenly becomes slower or faster because an application is moved from one MCU to another, it could cause problems and potentially a lot of tests has to be redone. This however is not a problem in space, since applications will not be moved around in space, so a comprehensive test can be made before flight when all applications are finished.

The reason not to put all communication on the CAN bus, is that it puts more traffic on the bus, which can delay the communication between other applications and communication between two applications placed together can be made faster. Network sniffing is not possible in space, unless everything is send to LOG or send directly to COM and GND. If a lot of communication is going on inside an MCU, it might be better just to log state changes and error messages. This is the interesting information to have when debugging and everything else would be a waste of memory and space link.

If the message should always be put on CAN, it has to be sent to another node, or else the frames will not be hardware ACK’ed and eventually the CAN hardware will go into bus off mode. This other node could be where the LOG application is placed, but then another problem arises, if LOG is also placed on the same MCU. Another way would be to have at least two MCUs turned on all the time. This way it is possible to hardware ACK each other, but it would use more power. The other MCU could be one that does not run code that often, or an MCU that almost always has to be turned on like COM.

Based on the above and discussions made in AAUSAT3 system engineering group, it is decided that CSP should not send internal communication on the CAN bus. This is mainly because we can not sniff the network in space and the fact that at least two MCUs must be turned on in order to communicate. This means that testing of applications should involve both a scenario where internal communication is used and a scenario where CAN is used, in order to test timings.
Figure 7.3: Sequence diagram showing communication between a client and a server. The number after conn symbolizes the number of the connection struct in the pool.
In order to keep the software reconfigurable, the function calls in CSP used for communication should still be used, even though the destination address is internal. The application itself does not know where other applications are placed. It just uses the regular function calls and CSP takes care of the rest.

Figure 7.3 shows an example of communication between a client and a server during external communication. The corresponding sequence diagram for internal communication is shown on figure 7.4. The same function calls are used in both cases.

The `csp_connect()` call checks if the destination address is equal to its own address. If so, a guard is set, indicating that this is internal communication. This flag is used by `csp_zsend()`, which creates the server part of the connection (conn 2 on figure 7.4). To make sure that `csp_zsend()` only creates a server connection once, another guard is set. After creating the connection, it makes a callback to the receive ISR, which, in this case, puts a pointer to conn 2 on queue 2. After reading the pointer from the queue, `csp_zread()` is called. Data is copied directly between client and server, without the use of buffers, to decrease the amount of used SRAM. This means that data should not be changed during the timeout period for `csp_zsend()`. When the server has finished using the data, or the data has been
copied over, it calls csp_zdone(). csp_zdone() gives the semaphore csp_zsend() is waiting to take. To make it possible for csp_zdone() to give the client tx semaphore, the server connection struct contains a pointer to the client connection struct. The server then sends a reply back by putting a pointer to the reply data into the client connection struct, and then it changes its’ rxstate to RX_DONE and gives the semaphore csp_zread() is trying to take. When the client has finished using the data, or the data has been copied over, it also calls csp_zdone(). Since the connection is not set to close, the csp_conn_close() call on the server side returns without doing anything. It is all repeated in the second transaction, but this time csp_zsend() on the client side told the server side that it was allowed to close the connection.

The test files cli_serv_loopback_test.c, tests the implementation of the internal communication and it is working as intended. In the test, a client in one task requests data from a server in another task. The priority between tasks are changed between 1-1, 1-2 and 2-1 to test synchronization.

7.4 Design of Service Layer

According to the requirements specified in section 7.2, the service layer must contain a heartbeat functionality (req. id §5.4.1). This function must be able to get the status of all applications on the given MCU and send these to the source of the heartbeat request. The service layer must also implement an application control functionality (req. id §5.4.3). This functionality must make it possible to start or stop an application from another application. Further more the service layer must provide a time synchronization functionality (req. id §5.4.2) in order to update the global time of all applications and a functionality to ping the CSP protocol (req. id §5.5.1) in order to state whether a MCU has stopped working or if the application has deadlocked itself.

The design and implementation of the stated functionalities and the general structure of the service layer is described separately in the following. Small examples from the code is include in the description. The total code for the service layer is located in service_task.c and service_task.h which can be found on the project web page.

7.4.1 General Structure of Service Layer

Creating one service layer task for all the services in the service layer will save SRAM, compared to the alternative where each service has its own task. Likewise will one big queue for all the service layer ports be in favor of multiple queues due to the increased amount of SRAM used to create and handle multiple queues.

This means that csp_port_open() should be run on port 1 to 4 under initialization and they should all use the same callback function. The callback function for the service layer has a switch case where it either answers back immediately, if a message is received on port 1 (because CSP ping must not use tasks or queues), updates global time if a message is received on port 4 (to minimize time synchronization drift) or puts a pointer to the connection struct on the queue if a message is received on port 2-3.

The code for the switch case is shown here:

```
switch (Csp_get_edport(conn->idin)){
    case CSP_PING: // CSP Ping
        csp_zdone(conn);
        csp_conn_close(conn);
        break;
    case APP_CONTROL: // Application Control
        break;
    case APP_HEARTBEAT: // Application Heartbeat
        /* Put address of conn on the service queue */
```
if ( xQueueSendToBackFromISR ( service_queue , &conn , &xHigherPriorityTaskWoken )
! = pdPASS ) {
    printf_P ( PSTR( "Failed to put message on queue \r\n" ) );
}
break : 

  case TIME_SYNC : // Timesync :
    length = csp_zread ( conn , &receive_p , 0);
    if ( length == 0 ) {
        printf_P ( PSTR( "Time Sync receive error\r\n" ) );
    } else {
        global_time = *((unsigned long*) receive_p);
    } /* Free Zero-copy data again and send ACK to transmitter */
    csp_zdone( conn );
    csp_conn_close( conn );
    break ;
  default : 
    printf_P ( PSTR ( "Error – This is not a valid service task \r\n" ) );
}

The service layer task makes a blocking read from the queue and when something is received on the
queue it also runs a switch case on the port, which indicates what service is requested.

The task can run as a while(1) loop because of the blocking read, which allows other tasks to run when
there is nothing in the queue. This way the task can run with a high priority, thus taking precedence over
other tasks since the services are more important than whatever the application is doing. It should still be
possible to run applications functionalities with a higher priority than the service layer, but this should
however not contain application services.

The general structure of the service layer has been tested implicitly by testing each service contained in
the service layer. These test has shown that it is possible to address each port (service) and that pointers,
to the connection structure, is only put at the queue when port 2 and 3 is addressed. Furthermore it is still
possible to run other tasks besides the service task, indicating that the service task is only active when
it has a service request. It is thereby concluded that the general structure of the service layer is working
properly.

### 7.4.2 CSP Ping

The idea behind CSP ping is to be able to check if CSP is responding. If for example some applications
has gone into deadlock, or if a stack overflow made the scheduler crash, CSP ping will still respond
since it is run inside a ISR and does not use queues or tasks. Application heartbeat tells the state of the
application (a more detailed description of the application heartbeat will follow later in this section) and
CSP ping tells if the node is responding at all. This way its possible to narrow down the reason for a
system failure.

It has been decided that the CSP ping service is to be accessed on port 1 and can be used by EPS to check
if a reboot is needed or by GND for debugging. This means that 2 connections should be allocated on
this port, since there is only two nodes that potentially will use the service. The connection number can
be increased if necessary.

When the CSP ping service is accessed it will only respond with an acknowledge frame. In order to
access the CSP ping service, a message with no information must be send to port 1.

The CSP ping service has been tested on a single test board which utilizes internal communication. This
test has shown that the CSP ping functionality works for applications that has deadlocked itself, but do
not test the CSP ping functionality in cases of stack overflow. The test service_layer_test.c is
located on the project web site.
7.4.3 Application Control

The purpose of application control is to be able to start and stop applications from another application, see requirement §5.4.2 in section 7.2. This means that when an MCU is powered on it first goes into initialization where it creates all the tasks for the different applications, among other things. These tasks will be suspended right away when the scheduler starts and can only be started with an application control request. This request tells the tasks of an application to resume in order to start the application. This way it is possible to control which applications are running on an MCU. If the MCU only contains one application, a power up would mean the same as this application should be running, and the application control would be obsolete, but a situation might occur where it is favorable to initialize the application, so it is ready to run. Figure 7.5 presents the state diagram for an application.

![State diagram for an application. The MCU can be powered off in all the states and will initialize on power on. The transition condition and generated output are separated with /.

On power on, the MCU initializes and as a last thing it suspends the task. When the service layer receives a command to resume the task, it starts up the application tasks and put the application in state RUNNING. This state change is sent to LOG. The application tasks is stopped again when a task suspend request is received. The code for suspending a task can be seen here:

```c
if (cmd_state==2) {
  /* Set the tasks of the application to idle -> suspend tasks */
  for (U8 i=0;i<8;i++) {
    if (app_nr[req_nr].app_handles[i]!=0) {
      /* Save task handle */
      app_nr[req_nr].app_status=request;
      printf_P(PSTR("Changing state, new status is: App1: %d APP2: %d APP3: %d \r\n"), app_nr[0].app_status, app_nr[1].app_status, app_nr[2].app_status);
      vTaskSuspend(app_nr[req_nr].app_handles[i]);
    }
    else {
      break;
    }
  }
}
```

If the requested command is to suspend an application, it will suspend all tasks connected to the given application. This induces that tasks must be connected to an application when these is initialized. This is done by calling the `connect_app_control()` function in the start of each application. As an example `connect_app_control(1,&nightrider_handle)` will connect the "nightrider" task to application one. It is currently possible to connect up to eight tasks to an application.

If a task resume or suspend request comes when the tasks are already running or idle, the request will be ignored.
A heartbeat request can come during all the four states (the description of the heartbeat will follow later in this section), but a response can not be send in state ERROR and this indicates that an error has occurred. When a heap overflow occurs the state is changed to ERROR and a log message is sent. Afterwards an automatic reset of the MCU is performed. If a dynamic memory allocation fails, the MCU will perform a software reset using the watchdog timer.

During initialization all interrupts will be disabled and the heartbeat request will not be answered. This is also the case in state ERROR where an ISR might prevent an automatic reset from happening and its not certain if the heartbeat request can be handled at all due to the error.

The applications control service is located on port 2. Since it is only FP and GND that commands applications to start and stop, two connections will be preallocated. Other applications can also use application control, but then it might be an idea to increase the number of possible connections.

In order to resume or suspend a task, two bytes of data must be sent to port 2. The first byte must contain the number of the wanted application and can be of values from one to three. The second byte must contain the wanted state of the application which can be either RUNNING or IDLE.

The following code is an example of a frame used to suspend the AIS application connected to application one.

```c
U8 cmd_frame[] = {1, AIS_IDLE};
```

The application control service has been tested together with the CSP ping service in service_layer_test.c. The test has show that is it possible to suspend and resume a task using the service layer. Furthermore the software reset has been tested by allocating too much SRAM using malloc() and then observe if the MCU was restarted. The test has shown that the software reset is working.

### 7.4.4 Application Heartbeat

The application heartbeat is a service provided by all nodes, where it sends the states of the applications located on the node, when requested from an other application.

A byte is used for each state which means that the heartbeat response will be one frame with between 1-3 states (maybe more for more applications). To distinguish between the state for each application, all states for all applications each has a define in aausat3.h. This gives 45 states if there is 15 applications.

The response frame should be ready to send all the time and therefore the application states are put in a global array with a size equal to the number of applications on the MCU. Each time a state is changed for an application, the correct byte in the global array is changed immediately. The initialization function must decide where to put each state if there is more than one application on the MCU.

According to the state diagram for an application, see figure 7.5, heartbeat requests must be ignored unless in state IDLE or RUNNING.

How long an application has been running (uptime) could also be sent. This is however obsolete since this can be checked in the log file and there does not seem to be any need for this feature in space. It would also require more than one byte for each application unless hours are counted, but hours does not give that much information.

The application heartbeat service is located on port 3. Eight connections should be allocated, since all applications could potentially request this service at the same time.
In order to access the application heartbeat service, at least one byte of information must be sent. This byte of information is not used by the application heartbeat service, but must be included in order to pass the message true CSP. This means that the content of the byte it is unimportant.

The frame received for the application heartbeat service is of the following structure:

```
U8 app_status_frame_HB[] = {app_nr[0].app_status, app_nr[1].app_status, app_nr[2].app_status};
```

The application heartbeat service has been tested together with the CSP ping and application control services. The test has shown that the application heartbeat service is working.

### 7.4.5 Time Synchronization

A global time for the satellite can be used to time stamp log messages, AIS signals, pictures etc. Furthermore, several systems on different MCUs might need to have the time available so a synchronization mechanism have to be devised. On AAUSAT-II, the time is maintained by the OBC, but it is reset when the OBC is reset, which happens every few hours at the moment because of an error.

Therefore it has been decided for AAUSAT3 to provide a Real Time Clock (RTC) which is not affected by system reboot or loss of power by attaching an external RTC chip with backup power supply to EPS. EPS will then be able to act as a time server since it should be turned on as the first thing and shut down last in case of power failure and thus is always available to other MCUs when they are turned on. A survey has been done to evaluate possible RTC devices and the results have been posted to the AAUSAT3 Trac web site. It has not been determined exactly which applications require global time and how precise, but it is believed that a precision on the order of seconds is adequate.

All MCUs need to maintain their own time, which is done by FreeRTOS in the `rtc_tics` variable each millisecond during the ISR which preempts tasks. This ISR is triggered by overflow on Timer1, which is fed by a prescaled version of the 8 MHz system clock. On EPS, `rtc_tics` needs to be synchronized with the external RTC on startup and with a given interval. EPS can then provide synchronization to the other applications.

As mentioned, a service should take care of synchronizing the global time on each MCU. There are two possibilities: Either should each node request time at a regular basis or EPS should send out the time at a regular basis. The first solution requires an extra task on each node and the second solution only requires one task on the EPS node, which is why this solution is chosen.

With a frequency tolerance of about ±100 ppm for a crystal oscillator, the worst case time difference will be 0.72 s in one hour. This figure is just an estimate since it is temperature dependent and a final choice in oscillator has not been made. By having EPS transmit a time sync each hour, time can thus be maintained within one second among MCUs.

It can furthermore not be guaranteed that all nodes will be synchronized precisely, since the network is non-deterministic, but time synchronization messages should have very high priority in the network, to keep the differences as low as possible. The fact that synchronization can only be done with a certain precision needs to be kept in mind when developing applications. During initialization of an MCU, it must send a time synchronization request to EPS, otherwise it might take an hour before a synchronization is made, which will mess with the time stamping and timing.

The time synchronization service is located on port 4 on all MCUs and since only EPS talks to it, only one connection needs to be preallocated. To utilize the service, 4 bytes containing the synchronization time, must be sent to port 4. A time server application also has to be created for EPS, which can be used by MCUs to request a time sync when they are started up.
The time synchronization service has been tested together with the CSP ping, application control and application heartbeat services by sending a known time to the time synchronization service and then check the global time. The test has been performed for both external communication and using loopback. The test verified that the global time was updated.

### 7.5 Conclusion

Multiple issues have been addressed in order to develop a protocol stack for AAUSAT3. The list of addressed issues is summarized in table 7.7.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implement hardware filter</td>
<td>✓</td>
</tr>
<tr>
<td>Implement an addressing scheme</td>
<td>✓</td>
</tr>
<tr>
<td>Modifying CSP to work with the FreeRTOS kernel</td>
<td>✓</td>
</tr>
<tr>
<td>Implementing loopback functionality</td>
<td>✓</td>
</tr>
<tr>
<td>Implement CSP ping (service layer)</td>
<td>✓</td>
</tr>
<tr>
<td>Implement application control (service layer)</td>
<td>✓</td>
</tr>
<tr>
<td>Implement application heartbeat (service layer)</td>
<td>✓</td>
</tr>
<tr>
<td>Implement time synchronization (service layer)</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Table 7.7: List of issues addressed in order to develop a protocol stack for AAUSAT3. The ✓ marks that the software related to the issue is working.*

As appears from table 7.7 all issues have been addressed and is working accordingly to the requirements.

Four tests have been designed in order to test the performance of the protocol stack and to clarify whether the protocol stack is suitable for AAUSAT3. These concerns the memory consumption, the load of the MCU for both transmitting and receiving messages, the maximum achievable goodput for CSP as well as the robustness of the implemented software. The method used for testing and the results of the tests is placed in chapter 9.

### 7.6 CSP and Service Layer User Guide

The idea behind this user guide is to make it easier for software developers to write their programs. This user guide is also placed on the AAUSAT3 Trac web site so people working with AAUSAT3 can find it easily.

First general information is given, followed by a description of the API functions. Examples of use is presented at the end.

#### 7.6.1 General Information

- There are 32 available ports. Port 0 is reserved for the bootloader, port 1-7 is reserved for the service layer task, port 8-23 is reserved for services provided by applications and port 24-31 is reserved for outgoing connections. If more ports are needed for application services on a node, it must mask on 3 destination address bits instead of 4, thus doubling up the address space.

- Avoid creating a receiving queue and service, providing task for each service, since queues and tasks uses a lot of SRAM. Instead use one task per application taking care of providing services, and then do a switch case on the port number to distinguish between requests.
• Leave some SRAM space for receive buffers, since they are currently malloc’ed runtime.

• Be careful to close connections when not used. The amount of connections is limited to 10, where 2 is reserved for highest priority messages.

• `csp_conn_close()` must be called between receipt of packages on the server side, to ensure that the used guards are updated correctly. This does however not mean that the connection is actually closed. If the connection should be closed on the server side, the client should send a BEGIN frame, and if the connection should not be closed a BEGIN_ALIVE frame is send. The parameter `keep_alive` parsed to `csp_zsend()` determines if a BEGIN or BEGIN_ALIVE frame is sent. It has been implemented this way to increase robustness, since unpredictable errors can occur if a connection is close while still using it in other API calls.

• When specifying a timeout, give at least 5 ms when sending 1 byte, and more if sending larger packages.

### 7.6.2 CSP API

- **void csp_init(unsigned char my_node_address, unsigned char my_mask_nr)**: This function is used to initialize CSP. It specifies the node address, creates semaphores for timeout handling and calls `can_init()`. The node defines are located in `csp_aausat3.h`. `my_mask_nr` is used to indicate how large an address space the node needs, or how many bits of the destination address the hardware filter should mask on. If set to 4, 32 ports is available and if set to 3, the available ports are doubled at the cost of a destination address.

- **S8 csp_port_open(U8 port, U8 max_connections, void (*callback)(conn_t*))**: Readies a port for incoming connections. `max_connections` specify the maximum number of simultaneous connections on the given port. A pointer to a callback function is also parsed to the function. The callback function is an ISR putting a pointer to the established connection on the receiving queue. `csp_port_open()` returns the port number or -1 if it is already in use.

- **conn_t* csp_select(U8 port)**: Checks all the connections for data availability on a given port. It returns a pointer to a connection in RX_DONE or NULL if no connection has data available on the port. The function can be used to poll for data.

- **conn_t* csp_connect(U8 prio, U8 dest, U8 port)**: This function is used to create a connection, if data needs to be send from one application to another. It searches the port list for a free port that can be used as outgoing port and then searches for a free connection struct in the connection pool and initializes this struct. The parameters `prio` (message priority), `dest` (destination address) and `port` (destination port) are used to initialize the struct. The function returns a handler to the connection or NULL if unsuccessful in creating a connection.

- **void csp_conn_close(conn_t* conn)**: Closes a given connection by freeing the receive buffer, if larger than 8 bytes, resetting the states and closing the associated port, if it is an outgoing port. A check is made on the server side to see if the connection is actually supposed to be closed or not, which is determined by the `keep_alive` parameter passed to `csp_zsend()` on the client side.

- **U8 csp_zread(conn_t* conn, U8** ** zdata, U16 timeout)**: Writes a pointer to the received data in the `zdata` argument. If a timeout of 0 is used, it checks once to see if data is ready and if a timeout > 0 is used, it does a blocking wait on a semaphore until data is ready or the timeout expires. The function returns the length of the received data.
• U8 csp_zsend(conn_t* conn, U8* data, U8 len, U16 timeout, U8 keep_alive): This function sends the data, the pointer data is pointing at, having the length len. If a timeout of 0 is used, it sends the data and returns immediately and if a timeout > 0 is used, it does a blocking wait on a semaphore until the data has been ACKed by the receiver or the timeout expires. The function returns the txstate or 0 if it could not send the data. The parameter keep_alive must be set to either DONT_CLOSE_CONN or CLOSE_CONN, depending on what csp_conn_close() should do on the server side.

• void csp_zdone(conn_t* conn): This function is called after csp_zread() and it indicates that the user has read all the data in the zero-copy buffer and that it is safe to free or overwrite the buffer again. During external communication it sends an ACK frame back, which releases the semaphore csp_zsend() is waiting to take. During internal communication it just releases the semaphore.

• void csp_send_cmp(conn_t* conn, U8 type): Internal helper user function to transmit empty frames of a given type. Normally used for ACKs and ERROR frames. Input parameters are a handler to the connection struct and frame type to send.

7.6.3 Service Layer API

• void service_layer_init(U8 app1_idle, U8 app2_idle, U8 app3_idle): This function creates the service_queue, open up port connections and creates the service layer task. It also sets the state of the applications placed on the MCU to IDLE. If only one application e.g. AIS is located on the MCU the following is parsed to the function AIS_IDLE, 0, 0.

• void connect_app_control(U8 my_app_nr, xTaskHandle * my_task_handle): This function makes a connection between the given task and an application number between 1-3, in order to be able to suspend and resume tasks from the application control service. It must be run just before the while(1) loop in the task and it suspends the task. my_app_nr is determined in the main function by the order of the parameters passed to service_layer_init().

• void software_reset(void): This function enables the watchdog timer and then runs a while(1) loop forcing a software reset.

7.6.4 Main Function Example

The following functions should be called in the main() function of the node.

```c
/* Initialize CAN */
csp_init(NODE_AIS, 4);

/* Initialize service layer */
service_layer_init(AIS_IDLE, 0, 0);

/* Create a queue capable of containing 4 connection struct pointers. */
queueAIS = xQueueCreate(4, sizeof(conn_t *));

/* CSP event handlers:Recv_packet on port 10 */
csp_port_open(10, 4, &recv_packet);
csp_init(NODE_AIS, 4) initialize CAN and sets the hardware filter to only accept frames for AIS.
service_layer_init(AIS_IDLE, 0, 0) starts up the service layer and tells it that the AIS application, is the only application on the node out of three possible. The functions are more explicitly explained in subsection 7.6.3.
```

70
The `csp_port_open(10, 4, &recv_packet)` call opens port 10 for incoming packages and associates the callback function `recv_packet()` with this port, which puts a pointer to a connection on the created queue (see line 8), when a package is received. The port can have a maximum of four connections simultaneously and the queue is also limited to four waiting messages.

### 7.6.5 Structure of a Service Providing Task

If e.g. the AIS application provides three services, AIS_gather on port 10, AIS_decode on port 11 and AIS_send on port 12 one service task could be made providing these services. The code could look like this:

```c
app_AIS.h():

/* Task priority */
#define SERV_AIS_PRIO ( tskIDLE_PRIORITY + 1 )

/* Handles */
xTaskHandle AIS_service_task_handle;
QueueHandle queueAIS;

/* Function prototypes */
void recv_packet_AIS(conn_t * conn);
void AIS_service_task(void * pvParameters);

main():
int main(void){
... /* Initialize CAN */
csp_init(NODE_AIS, 4);

/* Initialize service layer */
service_layer_init(AIS_IDLE, 0, 0);

/* Create a queue capable of containing 8 connection struct pointers. */
queueAIS = xQueueCreate(8, sizeof(conn_t));

/* CSP event handlers: Recv_packet on port 10, 11 and 12. Only one connection can be made on these ports at a time */
csp_port_open(10, 1, &recv_packet_AIS);
csp_port_open(11, 1, &recv_packet_AIS);
csp_port_open(12, 1, &recv_packet_AIS);

/* Create the service providing task */
xTaskCreate(AIS_service_task, (const signed portCHAR *) "ServAIS", stack_size, NULL, SERV_AIS_PRIO, &AIS_service_task_handle);
...}

recv_packet_AIS():
void recv_packet_AIS(conn_t * conn) {
  signed portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
  /* Put address of conn on the AIS queue */
  if ( xQueueSendToBackFromISR(queueAIS, &conn, &xHigherPriorityTaskWoken) != pdPASS ){
    printf_P(PSTR("Failed to put message on queue \r\n"));
  }
}

AIS_service_task():
```
void AIS_service_task(void *pvParameters) {
    conn_t *conn_rx;
    // Make this task available to the service layer application control
    connect_app_control(1, &AIS_service_task_handle);
    while(1) {
        // Do a blocking read for a message on the queue
        xQueueReceive(queueAIS, &conn_rx, portMAX_DELAY);

        switch(Csp_get_eport(conn_rx->idin)) {
        case 10: // AIS\_gather
            //Do something
            break;
        case 11: // AIS\_decode
            //Do something
            break;
        case 12: // AIS\_send
            //Do something
            break;
        default:
            //Do something
            break;
        }
    }
}

7.6.6 Client/Server Example

/* Client code */
U8 request = 5;
U8 *p = &request;
conn_t* conn = NULL;

/* Make a connection */
conn = csp_connect(PRIIO_NORM, NODE_AIS, 12);

/* Send a request and wait for reply */
csp_zsend(conn, p, sizeof(request), 4, CLOSE_CONN);
if(csp_zread(conn, &p, 4) != 0){
    printf_P(PSTR("Reply received: %s \n\r"), p);
} else{
    printf_P(PSTR("No reply\n\r"));
}

csp_zdone(conn);
csp_conn_close(conn);

/* Server code */
void server_task(void *pvParameters) {
    while(1) {
        // Blocking read on queue
        conn_t *conn = NULL;
        if(xQueueReceive(queue_server, &conn, (portTickType) portMAX_DELAY)) {
            // Read request
            U8 *p;
            U8 request = 0;
            if(csp_zread(conn, &p, 0) != 0){
                request = *p;
                printf_P(PSTR("Request received: %u \n\r"), request);
                csp_zdone(conn);
            } else{
                printf_P(PSTR("Request failed\n\r"));
            }
Send reply back and close connection

static char data[] = "12345678123456781234567812345678";
if (request == 5) {
    U8*p = ((U8*) &data);
    csp_zsend(conn, p, sizeof(data), 5, CLOSE_CONN);
}
csp_conn_close(conn);

If the reply is larger than 248 bytes (max package size), csp_zsend() should be placed in a loop sending 248 bytes at a time. csp_zread() should then also run in a loop, and a sequence scheme should be provided, making it possible to resend lost packages.

7.6.7 Use of Services Provided By the Service Layer

CSP ping:

conn_t * conn;
U8 *cmd_p=NULL;
conn = csp_connect(PRIOR_NORM, NODE_AIS, CSP_PING);
/* Send a ping to node AIS and wait 10 ms for response */
if (csp_zsend(conn_tx, cmd_p, 0, 10, CLOSE_CONN) == SOCKET_ERROR) {
    //Do something since the node did not respond in time
}
csp_conn_close(conn_tx);

Application Control:

conn_t * conn;
U8 cmd_app_nr=1;
U8 cmd_stat=AIS_RUNNING;
U8 cmd_frame[]={cmd_app_nr, cmd_stat};
U8 *cmd_p=(U8*) &cmd_frame;
/* Connect to AIS and send a high prio message telling it to start running */
conn = csp_connect(PRIOR_CRITICAL, NODE_AIS, APP_CONTROL);
csp_zsend(conn, cmd_p, sizeof(cmd_frame), 10, CLOSE_CONN);
csp_conn_close(conn);

Application Heartbeat:

conn_t * conn;
U8 *request_p;
/* Connect to AIS and request status of applications */
csp_zsend(conn, cmd_p, 0, 10, CLOSE_CONN);
csp_zread(conn, &request_p, 10);
printf_P(PSTR("Status of applications → App1: %d App2: %d App3: %d \r\n"), *request_p, *(request_p+1), *(request_p+2));
csp_zdone(conn);
csp_conn_close(conn);

Time synchronization:

conn_t * conn;
U32 global_time=60000;
U8 *time_p=(U8*) &global_time;
/* Connect to AIS and send a time sync */
csp_zsend(conn, time_p, sizeof(global_time), 10, CLOSE_CONN);
csp_zdone(conn);
csp_conn_close(conn);
8

Bootloader & Software Image Server (SWIS)

This chapter contains documentation about the design of a bootloader for the MCU’s used on AAUSAT3. The goal of the bootloader will be to allow in-space software updates of the MCUs on an application level.

8.1 Introduction

A bootloader is a piece of program code that is executed before the main program on a given platform. It can be used to determine which software that are executed next. In some cases, the bootloader will also allow software updates, or booting over a network. The reason for wanting a bootloader on the AAUSAT3 stems from the distributed system architecture which allows multiple software applications to run on different or the same hardware platform. It is important to be able to change software on the satellite to add new features or fix errors and when the software is distributed on several MCUs it is needed to be able to update all of them.

The bootloader system consists of to functional parts: The actual bootloader running on each MCU and a software image server (SWIS), running on a single MCU. The bootloader is responsible for loading new software images onto the hardware it is on. The bootloader receives software images from the SWIS, through the internal CAN bus of the satellite. The server is responsible for receiving software images from earth, via the space link, and store these locally on the satellite. It acts as a kind of buffer or proxy server.

To be able to reduce the size of the images to be uploaded, it has been decided that the individual applications should be individually interchangeable. Usually changes to software are not very extensive, maybe just changing one value or adding a few line of code. In these cases, there is no need to update all the software on a given MCU. The concept of updating only the changed application can later be extended to updating e.g. individual functions or even just uploading a file containing the binary differenced between the old an the new software.
When signal is given to update software on a given MCU, the SWIS stores the command and updates the software the next time the MCU starts up. A block diagram of the bootloader-image server interaction can be seen in figure 8.1. The bootloader and software image server will as specified earlier communicate through the onboard CAN bus, meaning that the bootloader must have a reachable and functional CAN bus interface. Upon hardware reset, the bootloader will ask the SWIS through this interface if new software updates are available. The software image server will have a system of knowing which software is on each hardware, and compare this to new software uploads from earth. If new software is available, the actual boot process will begin.

### 8.2 Requirements

In order to have design guidelines, and meet the overall mission objectives and required functionalities, several requirements are made for the bootloader and software image server. These requirements come from several upper level requirements, stated as parent in the table. Requirements for the bootloader and software image server can be found in table 8.1.

<table>
<thead>
<tr>
<th>Req. ID</th>
<th>Change Date</th>
<th>Requirement</th>
<th>Parent</th>
<th>Child</th>
<th>Author</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>§5.8.1</td>
<td>20008-11-06</td>
<td>It shall be possible to exchange all satellite software, except the bootloader, while the satellite is operational</td>
<td>§5.8</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.8.2</td>
<td>20008-11-06</td>
<td>It shall be possible to upload software changes and store them on the satellite</td>
<td>§5.8</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.8.3</td>
<td>20008-11-06</td>
<td>It shall be possible to store several versions of the same software on the satellite</td>
<td>§5.8</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.5.9</td>
<td>20008-11-06</td>
<td>The bootloader shall communicate through the on board CAN bus without receiving false data</td>
<td>§5.5</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.3.1</td>
<td>20008-11-06</td>
<td>The bootloader must be able to recover after an unintended loss of power</td>
<td>§5.3</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.3.2</td>
<td>20008-11-06</td>
<td>A fallback mechanism shall be provided in case of software failure</td>
<td>§5.3</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.2.1</td>
<td>20008-11-06</td>
<td>It shall be possible to update only part of the software on one HW platform</td>
<td>§5.2</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.1.1</td>
<td>20008-11-06</td>
<td>A bootloader shall be running on each HW platform</td>
<td>§5.1</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
<tr>
<td>§5.1.2</td>
<td>20008-11-06</td>
<td>The data size for a software change shall be minimized</td>
<td>§5.1</td>
<td>N/A</td>
<td>08gr724</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Requirements for bootloader and software image server. All requirements are children of the ones given in 6.1.
8.3 General Design

The bootloader consists of two parts as described in section 8.1. In order to develop these two parts independently of each other, some general design decisions have to be made first. This is primarily related to the communication between the server (SWIS) and the client (bootloader) and is more elaborated in section 8.4.

If some failures is detected in a newly uploaded software image, it is necessary to be able to fall back to an earlier version of the software in order not to cause deadlocks or other unwanted behavior. The triggers for fallback will be determined later and only the functionality will be treated here. For the fallback mechanism to work, earlier versions of software images have to be available on the satellite. They could either be stored on the server or on the client.

Each subsystem has 120 kiB of flash available for program code, which must be shared between the kernel, service layer, drivers and applications. This should leave space enough the applications, but problems can arise if several versions of each part of the software have to be stored. It furthermore requires more logic in the bootloader and when starting individual applications. Therefore all versions of the different software images are stored on the server and if fallback is required, the server will simply reprogram the required MCU.

To make sure that the content of the software images are intact at all times, a CRC checksum is kept for the images. It has been chosen to use the common CRC-16 since this is also available in the AVR libc as an optimized assembler routine as well as the equivalent C code, which makes it easy to calculate on other platforms as well, e.g. on a computer when generating the image.

8.4 Communication

This section is about developing a protocol for the communication between the server and the client. This need to be defined before the detailed design of the two entities can be performed. The protocol have to use CSP, but in a simple form, since the client does not have the space for a full CSP implementation. The server is a standard application, so it uses the full CSP library.

8.4.1 Ports and Addresses

On each MCU there is a bootloader, which starts as soon as it is powered on. To prevent problems if some application starts sending data to a subsystem during boot, port 0 is reserved for bootloader. In this way, the bootloader can check the port number to be sure that the incoming frames are actually bootloader frames.

On the server, a fixed port is assigned to the bootloader application. Together with the address of the server, this should be defined and hard coded into the bootloader client. The server need also to know which applications is located on which MCU, but this can be defined from GND. The ports and addresses are yet to be defined for the satellite as a whole, but for the prototype, the SWIS is located on node 4, port 8.

8.4.2 Communication Protocol

For simplicity of implementing CSP on the bootloader, only single-frame packets are used, which limits the length to 8 bytes per sent packet. The first byte of the data field in each CAN frame is used to indicate
the command for that frame, being one of the types listed in Table 8.2. This leaves 7 bytes for either command parameters or data in each frame. Each command frame is acknowledged by the server or client using the standard CSP ACK frame type.

<table>
<thead>
<tr>
<th>#</th>
<th>From</th>
<th>Name</th>
<th>Data</th>
<th>Data length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Client</td>
<td>BL_CLIENT_BOOT</td>
<td>None</td>
<td>0 byte</td>
</tr>
<tr>
<td>1</td>
<td>Server</td>
<td>BL_SET_CLIENT_STARTAPP</td>
<td>None</td>
<td>0 byte</td>
</tr>
<tr>
<td>2</td>
<td>Server</td>
<td>BL_SET_CLIENT_FLASH</td>
<td>Start address and image size</td>
<td>4 bytes</td>
</tr>
<tr>
<td>3</td>
<td>Server</td>
<td>BL_DATA</td>
<td>Binary data</td>
<td>7 bytes</td>
</tr>
<tr>
<td>4</td>
<td>Server</td>
<td>BL_DATA_END</td>
<td>None</td>
<td>0 bytes</td>
</tr>
<tr>
<td>5</td>
<td>Server</td>
<td>BL_CALC_CHECKSUM</td>
<td>Start address, length</td>
<td>4 bytes</td>
</tr>
<tr>
<td>6</td>
<td>Client</td>
<td>BL_CHECKSUM_RESULT</td>
<td>Checksum calculation result</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

Table 8.2: List of valid commands in the bootloader protocol.

### 8.4.3 Boot Sequence

Generally three things can happen when the bootloader client starts up and contacts the SWIS. Either one or more images has to be flashed, or (more common) nothing has to be done. The third case covers when SWIS is not available because it is busy or powered off. This section outlines the communication of a boot sequence for the two first cases. In the last case, the bootloader will simply time out and just start the applications.

Figure 8.2 shows the sequence of communication going on between the client and the SWIS in the two mentioned cases. If the SWIS does not find any updated software image for the client, it simply sends BL_SET_CLIENT_STARTAPP to make the client start the applications right away.

If a software update is found, the following sequence is used:

1. Server sends BL_SET_CLIENT_FLASH along with the start address and size of the image.
2. Server sends the binary data as type BL_DATA.
3. Server sends BL_DATA_END to indicate that no more binary data is available. This is used to make sure that SWIS and client are synchronized in case something went wrong during a potentially long data transfer.
4. Server asks the client to compute a checksum using the BL_CALC_CHECKSUM command.
5. Client answers with the calculated checksum in an BL_CHECKSUM_RESULT frame.
6. Depending on whether the flashing were successful and if more applications have to be flashed, the server can send a new BL_SET_CLIENT_STARTAPP or a BL_SET_CLIENT_FLASH.

This sequence can also be seen in figure 8.2 giving a illustrative overview of both the start application and flashing of new data communication procedures.

The actual writing of data to flash takes some time. If the SWIS keeps sending data the client in this interval, the receiver could get flooded and data lost. To prevent this, it has been decided to use the ACK frames as simple flow control. The client should not send an ACK frame for a data packet before it is ready to receive the next packet, either after storing the received packet in a buffer of having written it to flash. In this way, the SWIS will simply wait for the ACK as long as the timeout is set to a reasonable number.
Figure 8.2: Sequence diagram of both start application and flash new data communications.
8.5 Bootloader Client

The bootloader is a self-contained piece of software, capable of writing in a MCU’s internal flash. This means that the bootloader software is independent of operating systems and other implemented functions. The bootloader is located in the top 8 kiB of the flash, from where it is possible to write to the remaining 120 kiB of flash memory. Thus the bootloader must be self-contained, because it becomes more complicated if it should access function code outside its own memory scope. As the bootloader must be self-contained, it can easily take up some space in the MCU’s internal flash, if made to complex. This sets some restraints in the design of the bootloader as it can not implement different common satellite functionalities i.e. CSP, due to the size of these functionalities exceeding the scope of 8 kiB.

8.5.1 Design

As several applications are running on the same MCU, the different applications need to be divided in the flash, for the ability to only update one at a specific time. Thus the assignment of finite address scopes must be implemented. The different applications run as tasks in FreeRTOS, the OS of the MCU. This cross functionality includes the need to also allocate specific address scopes for the OS and common service functions, such as CSP, service layer and others. An illustration of these memory scope divisions can be seen in figure 8.3, illustrating the "application stack".

![Application Stack Illustration](image)

Figure 8.3: Memory stack and address sub scopes on individual hardware platforms.

During the boot sequence, the bootloader enters several different states, depending on the commands sent to it form the Software Image Server (SWIS), or its internal progress and processes. These states, change implicit runtime. Several different instances can trigger a state change, both intentional instances, and unintentional, triggering the bootloader to go in an error state. The state transition diagram for the bootloader is seen in figure 8.4. The BL_OFF state is not actually for the bootloader, but a state indicating that the application beneath the bootloader is running. In order for the bootloader to load new software, the hardware system must be reset. Upon this reset the MCU enters the bootloader section in the flash, thus executing the bootloader program.

Upon the reset and start of the bootloader program, the systems enter the boot_init state. This state initializes the bootloader to normal operation. The initialization process includes setting up CAN communications services, and contacting the SWIS. Depending on the answer from the SWIS, the bootloader can enter two states. It can fall back to the running state, initializing the applications, or the image transfer state, initiating the actual software image transfer sequence. If there is no communication with the SWIS, the bootloader assumes, that it is off-line, for some reason. If there is no contact to the server, the bootloader will automatically start the applications beneath.
In the image transfer state, several actions such as receiving data, writing data in the MCU’s internal flash are performed. When the received data has all been flashed, the SWIS will ask the bootloader to perform a Cyclic Redundancy Check. The result of this CRC will be transmitted to the SWIS, if this CRC differs from the CRC-calculation, done in the SWIS, the image server will force the bootloader to enter the error state. To exit the error state, the SWIS retries the transmission of data, for flashing, assuming, the error can then be corrected. After a number of inaccurate flashings, the SWIS will assume the specific address scope to be faulty, and relocate the update software. When a boot procedure is through, the SWIS tells the bootloader to reset the MCU hardware. This causes the hardware system to enter the bootloader once again, only this time, the SWIS, will tell the bootloader to start the applications, if there is no new data to be transferred/flashed to the system.

As the above states, the bootloader will always be entered when a MCU is reset. This means, that it will take some time, to start the actual application, which was expected. The time delay comes from the need of communication between the bootloader and the SWIS, and negotiation of flash/non-flash procedures. This depends on many factors, so it can be difficult to estimate and will be measured once the bootloader is implemented, as done in section 9.5. The time taken to transfer and flash a software image is estimated in section 8.7.

8.5.2 General Implementation

During implementation of the bootloader, one must keep in mind the size restraints of the bootloader software. The 8 kIB restraint does not allow the bootloader to use the CSP protocol, as it itself has a compiled size of more than 5 kIB. Thus another communication scheme must be developed, whilst maintaining communications with other software using the CSP protocol. The programming of new software to the MCU, or In System Programming (ISP), revolves around an ISP-driver, delivered by the MCU manufacturer, Atmel. This ISP allows programs to write in the MCU onboard flash with relatively few function calls.

The implemented software is split into two parts, the bootloader.c file, which sets up the MCU and initializes the CAN peripheral, and the bl_can_isp_protocol.c, which handles the communication, and reacts on the data format, received by the SWIS. The bl_can_isp_protocol.c is essentially a switch structure, reacting on the frame type, specified in section 8.4.2.
8.5.3 Implementation of Communication

Setting up communications in the bootloader is an important task, due to the fact that the bootloader can not get new software if it can not communicate with the SWIS. It has been decided to put all the complex computing and decision-taking in the SWIS, leaving the bootloader as a simple receive-and-react program. The difficulty in ensuring communication is, that the SWIS must use the CSP protocol in order to communicate with the other applications onboard the satellite, and the bootloader can not include CSP due to the before mentioned size restraints. Thus the bootloader is configured to receive and send CSP-style CAN frames on the network. This poses some difficulties, as the CSP protocol is a connection oriented protocol, and therefore needs and ACK on every frame, from the receiver. Thus a CSP style ACK frame has been constructed, and is sent after each received CAN frame.

8.5.3.1 Transmitting

Setting up the CAN peripheral is done in the bootloader.c file, this set-up involves allocating two buffers, one for the data to be transmitted (bl_tx_buffer), and one for received data (bl_rx_buffer). Also two message structs are created, called bl_rx_msg and bl_tx_msg, using the st_cmd_t struct, for accessing the CAN interface. In order to send a data request to the SWIS, the data field of the CAN frame is put into the bl_tx_buffer, by taking the pt_data element on the bl_tx_msg, and assigning it to the address of the bl_tx_buffer. In order to construct the entire CAN frame correctly, the whole st_cmd_t struct must be filled and send, using the bl_send_frame() function, as seen below.

```c
1 bl_tx_msg.ctrl.id_e=1;
2 bl_tx_msg.id.ext = BL_ID;
3 bl_tx_msg.dlc = 1;
4 bl_tx_buffer[0]=0x00;
5 bl_send_frame();
```

The bl_send_frame() function uses the can_cmd() function, which prepares and transmits the CAN frames. It also utilizes the can_get_status() function. This function checks, if the above function has been performed or not. Both of the above mentioned functions are a part of the can_lib.c, provided by Atmel. The BL_ID is the entire identifier for the CAN frame, containing source and destination ports and addresses, window length and frame type. If an ACK frame should be sent, the BL_ID is bitwise OR’ed with the BL_ID_ACK, changing the CSP frame type from BEGIN_ALIVE to ACK. The above states makes sure, that a CAN frame transmitted from the bootloader will go through and communicate with a CSP embedded application.

8.5.3.2 Receiving

Rescieving the correct data is crucial for the bootloader. If false data is received at the wrong time, this might be written in the onboard flash, maybe causing the flashed software to not function correctly and maybe crash completely. In order to prevent this from happening, a “filter” or mask is applied to the incoming data. This mask is essentially a hardware filter, filtering on the MOBs of the CAN interface. Masks can be applied to the entire CAN identifier, or only to specific bits in this. As the bootloader, will only communicate with the SWIS, a mask is applied to the source address and source port of the identifier. The mask is set, as seen in the following code-example:

```c
1 U32 bl_mask;
2 bl_mask=0b11110000111111<<12;//mask applied to
3 U32 bl_id;;
4 bl_id= 0b00100000000000<<12;
5 Can_set_ext_id (bl_id);
6 Can_set_ext_msk (bl_mask);                                 //Set ID of bootloader
7 //Set masking on received identifier
```
In order to mask on the identifier, the filter must know the ID to compare with. This ID does not have to be the entire ID, but just the values, within the masked area. The macros `Can_set_ext_id()` and `Can_set_ext_msk()` are provided by the Atmel CAN driver. As seen in the above, the ID and mask are bit-shifted 12 times to the left, making the used partial identifier fit within the MCU identifier register setup.

The received data are put into a receive buffer named `can_isp_rx_buffer` by taking the `pt_data` data pointer of the `st_cmd_t` struct, and assigning it to the address of the `rx_buffer`. To ensure a full read of received data, the `can_cmd()` and `can_get_status()` functions are used again, ensuring no program progress until the buffer is filled. In this buffer, the data field of the CAN frame is stored. The first byte of the data field, is the frame type identifier, showing which command from the switch case should be performed. These will be described in the following subsections.

### 8.5.4 Flash Programming

The most important feature of the bootloader is the ability to program new data into the onboard flash. This allows new applications to be placed on the MCU. The flash programming uses ISP functions and drivers supplied by Atmel. There are four specific scopes of the flash programming sequence, first an address scope definition, then the actual data transfer and finally the data end, and CRC calculation. All functionalities in the bootloader is, as before mentioned, controlled by the SWIS. The following will concentrate on the data types `BL_SET_CLIENT_FLASH`, `BL_DATA`, `BL_DATA_END` and `BL_CALC_CHECKSUM`, describing the flash programming states.

#### 8.5.4.1 BL_START_FLASH

The flash start state is triggered by receiving a CAN frame, with the first byte in the data field specifying the `BL_SET_CLIENT_FLASH` frame type. The next four bytes specify the start and end of the address scope, needed to be programmed with new data, respectively. These data are use by the ISP library provided by Atmel, in the `isp_address_range()` function, specifying the address scope of the incoming new data. After specifying the address scope, a pointer is saved, defining the start address, for use in the block programming of the data. A flag defining the status of the ISP programming is also set, to indicate receiving data for programming the flash. The following code example shows the above mentioned:

```c
1 isp_address_range ( (((U16)can_isp_rx_buffer[1])<<8) | (((U16)can_isp_rx_buffer[2]) ) ,
2 (((U16)can_isp_rx_buffer[3])<<8) | (((U16)can_isp_rx_buffer[4]) ) ) ;
3 loc_buf_index = 0;
4 isp_prog_on_going = TRUE;
5 // Save pointer and size for block programming using
6 isp_start_address_copy = isp_start_address;
7 isp_number_of_bytes_copy = isp_number_of_bytes;
```

After performing the above mentioned preliminary steps of the flashing of new data, an CSP-style ACK frame is sent to the SWIS, in order to acknowledge the received command and to inform the server that the bootloader is ready to receive the data for flash programming.

#### 8.5.4.2 BL_DATA

This command is significant for the bootloader, allowing it to program new data into the internal flash, thus creating the ability to update or exchange software on the MCU. In order not to complicate the simple CSP protocol in this prototype, it has been chosen to send 7 bytes at a time for programming, since this will fit within one CSP frame. The block programming procedure is supported in Atmel’s
ISP library, making the implementation of this easy. The function writing the data in flash is called
isp_prog_block(), taking only two inputs, the data buffer and an index indication the wanted ad-

dress for the data.

In order to keep track of the amount of data to be flashed, within a CAN frame, the DLC identifier is
used. As the first byte of the data field is used as a frame type identifier, the maximum amount of data
transmitted in a CAN frame, between the bootloader and SWIS is seven bytes. As seen in the example
code below, a loop is run, to copy DLC number of bytes to the buffer, with one byte offset due to the
frame type identifier being within the data field range. The local buffer and index are then passed to the
isp_prog_block() function, writing the data into the flash.

```c
for (i = 0; i < can_isp_rx_msg.dlc; i++) {
    local_buffer[loc_buf_index] = can_isp_rx_buffer[i] + 0x01;
    loc_buf_index++;
    u8_temp = 0x00;
}
if ((loc_buf_index > ((U16)(LOCAL_BUFFER_SIZE − 0x07))) ||
    (loc_buf_index == isp_number_of_bytes)) {
    u8_temp = isp_prog_block(local_buffer, loc_buf_index);
    loc_buf_index = 0;
}
```

Since the SWIS uses CSP as communication protocol every data frame must be ACKed, in order to keep
the connection oriented transmissions working without errors. Thus an ACK frame is transmitted every
time a block has been programmed. This also gives the SWIS information on whether the programming
went well or crashed, as if it was erroneous the ACK frame will not be transmitted.

8.5.4.3 BL_DATA_END

The BL_DATA_END, is a single frame command telling the bootloader not to expect more data to be
programmed to the flash. Before sending an ACK frame, to the SWIS, telling it to initiate the CRC
calculations and check-up, the ISP start address pointer and number of bytes to be programmed, are
reset, in order to be able to receive new data.

8.5.4.4 BL_CALC_CRC

In order to ensure, that the correct data has been flashed to the MCU, the SWIS will request a calculation
of a CRC for the data. This takes place when the SWIS has received an ACK for the DATA\_END
command. The bootloader will calculate a CRC for the received data, and sends the result to the SWIS.
The SWIS will then compare the result from the bootloader with its own stored value. Depending on the
result of this, the SWIS will either start the application if comparison is correct, or retransmit the data
and run through the entire boot sequence once again. The CRC calculation function has not yet been
implemented, but as a proof of concept dummy-CRC values are sent to the SWIS.

8.5.5 Implementation of Application Startup

In the Atmel ISP library, application start from within the bootloader areas is made easy. When the
bootloader receives a frame telling it to start the application, because there is no new data to be flashed,
the bootloader enters the OFF state, jumping the program pointer to the start address of the applications
running on the MCU.

```c
isp_jump_to(APP_START_ADDR);
```
8.5.6 Delimitations and Future Work

Due to time constraints in the project period, some features of the bootloader are still under development. The bootloader does not fit into the bootloader section of 8 kiB in the MCU’s flash, due to debugging software currently implemented. Also CRC-calculations are not done yet, but an algorithm is available in the AVR libc, so implementation should be trivial. Only when the bootloader is located in the right section in the flash, can one be certain, that the actual flashing of the data is done. As proof of concept, it has been shown that a limited program can communicate and interact with a more sophisticated piece of software, using CSP, without the small software needing a CSP protocol implemented. When working with limited size software, one must always put only the very basic functionality in the limited software and make e.g. the server accordingly advanced.

8.6 Software Image Server (SWIS)

The design philosophy of the bootloader is to put as much logic and intelligence away from the bootloader client. In this way, it can be more simple, use up less space and be less error prone. This is important since its size is limited to 8 kiB and it cannot be changed in space if an error is found.

A consequence of this choice is that the image server has to take care of some of these decisions, e.g. what to do if an error occurs during boot or flashing. The main task of the server will be to administer flashing of the MCU’s, based on a number of rules and behavior descriptions provided along with the software images.

8.6.1 Overall Design Decisions

Because it is desirable to be able to reprogram all applications on the satellite, it would be beneficial to locate the image server on a separate MCU. Physical space is not the biggest problem on the satellite, so adding another MCU would not be a big task, especially not because it does not need to be turned on always. Placing the SWIS on a separate MCU however has the implication that it will not be able to update itself, but this could possibly be done another way. Whether the SWIS will have its own hardware or will be located together with other applications, can be decided at a later point.

In general, the SWIS consists of two parts: One part taking care of communicating with clients, called the “flasher” and a part taking care of receiving data from GND, called the “receiver”. They are each implemented as one task in FreeRTOS. Only the receiver task need to be started, since it takes care of initializing and starts the flasher task afterwards.

8.6.1.1 Data Structures

The SWIS needs to store a number of images for the clients. Their sizes can easily add up, so they are stored in flash. For this prototype, they are stored in the external flash on the development boards. Although this will probably not be the final solution, it demonstrates the concept and it should not be a problem to use another kind of flash. A file system has not been implemented, but this maybe has to be done, in case the SWIS has to share hardware with other applications.

For the SWIS, there are 3 main data structures, listed below. They are all allocated statically, since this makes memory allocation simple, avoids fragmentation and if SWIS is the only application running on an MCU, memory should not be a problem. All three data structures are allocated twice, both in RAM and EEPROM. EEPROM is used as non-volatile memory and every time a change has been made to a
data structure, the changed part is rewritten to ensure backup if power is lost. Writing to EEPROM is
very slow, at about 8.5 ms/byte, so writing should be limited to the least possible. EEPROM is specified
to handle 100,000 writes, whereas flash are specified to handle 10,000. It could later be considered if
permanent storage should be moved from EEPROM to flash if the slow write speed poses a problem.

In general, default values for variables are -1 or 0, depending of what is appropriate. When id is set to
-1, it indicates that the given entry is empty. The three data structures are:

- List of images available.
- List of relations between images and client nodes.
- Temporary buffer for image uploads.

Generally, an image has been given a unique id from GND, from 1 to 32767, which is used to identify
that given image. The id would not be reused. Each node is identified by its node id, which is equal to the
CSP hardware address of the MCU, which can be seen from the initial request made by the bootloader
client.

The first of these data structures is implemented as an array of structures, as shown below.

```c
typedef struct {
  S16 img_id;
  U16 crc;
  S32 size;
  S32 client_location;
  U8 *server_location;
} img_info_t;

img_info_t images[MAX_NO_IMAGES];
```

Meta data for the image is stored in a struct of this type. For the prototype, an array containing 10 of
these structs has been declared. The final number of images has to be considered when it is known how
many image will actually be in use on the satellite. Each image uses 14 bytes in this configuration, but
this will most likely increase to 16 if they are to be stored on an external flash with high capacity.

Listing the relationship between images and nodes are given in another table. Doing the relationships
in another table enables one image to be flashed to more than one MCU while only having one copy on
SWIS and thus saving space and transfer time from GND. Shown below is the implementation of each
row in the array. Besides the relationship, it is also recorded whether or not the image has actually been
flashed to the client. Having this information ensures that the image is not flashed every time an MCU
starts up. This information accounts for 4 bytes per images per node.

```c
typedef struct {
  S16 img_id;
  S8 node_id;
  Bool flashed;
} img_status_t;

img_status_t img_node_status[MAX_NO_STATUS];
```

When uploading a new image from GND, the meta data is stored in the temporary image buffer while the
data itself is saved directly to flash on its’ final location. After all parts of the image has been uploaded,
meta data is copied to the table of images and the temp buffer is cleared. Below is shown the struct,
which simply contains the usual meta information and an array of bytes. This array is used to store the
status for the individual parts of the software image.

```c
typedef struct {
  img_info_t info;
  U8 pktmap[PACKET_MAP_LENGTH];
```
8.6.1.2 Flasher part

Generally, the typical scenario of booting an MCU and flashing an image to it is described in Section 8.4.3. To handle this procedure, the flasher task is an implementation of the state machine shown in Figure 8.5.

As defined earlier, the flasher task listens for incoming connections on port 8. It allows only one connection at the time, since it simplifies the design of the state machine that it only needs to take care of one client at a time. It could happen that two MCUs would be powered on at the same time, but in this case only one would be accepted. The other client would time out when not receiving an ACK and just start the application.

Upon starting the flasher task, an infinite loop is entered, since the SWIS is not supposed to be shut down. The content of the loop is a switch structure where the switch statement is the current state. Table 8.3 shows a list of the states.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BL_STATE_IDLE</td>
<td>Waiting for request from client</td>
</tr>
<tr>
<td>1</td>
<td>BL_STATE_REQUEST</td>
<td>Handling request, i.e. check if any images needs to be flashed</td>
</tr>
<tr>
<td>2</td>
<td>BL_STATE_PROGRAMMING</td>
<td>Sending an image to the client</td>
</tr>
<tr>
<td>3</td>
<td>BL_STATE_DATA_END</td>
<td>Sending data end to indicate programming end</td>
</tr>
<tr>
<td>4</td>
<td>BL_STATE_REQ_CHECKSUM</td>
<td>Request checksum for flashed image</td>
</tr>
<tr>
<td>5</td>
<td>BL_STATE_TREAT_CHECKSUM</td>
<td>Receive and compare checksum</td>
</tr>
</tbody>
</table>

Figure 8.5: State transition diagram for the flasher part of the SWIS.

As defined earlier, the flasher task listens for incoming connections on port 8. It allows only one connection at the time, since it simplifies the design of the state machine that it only needs to take care of one client at a time. It could happen that two MCUs would be powered on at the same time, but in this case only one would be accepted. The other client would time out when not receiving an ACK and just start the application.

Upon starting the flasher task, an infinite loop is entered, since the SWIS is not supposed to be shut down. The content of the loop is a switch structure where the switch statement is the current state. Table 8.3 shows a list of the states.

The content of most states is pretty straightforward and will as such not be described further here. When an error has been detected, e.g. an invalid response or lack of acknowledgment on packets sent, the default action is to set the flasher to idle state.

During the request state, the flasher task first looks in the img_node_status array to see if the current client is missing any software updates. If this is the case, it looks up the image in the images array and
sends out the meta information to the client. During programming, the data itself is copied in smaller chunks, according to the need of the client.

### 8.6.1.3 Receiver Part

Before software images can be flashed onto an MCU, they have to be transferred from ground to the satellite. For this purpose the second part of the SWIS, called receiver, is designed. It is supposed to show the possibility of transferring an image to the server from GND. The protocol developed is however only seen as a proof of concept since more attention has to be given to how to optimally transfer data over the space link. The state diagram for the receiver part can be seen in Figure 8.6.

*Figure 8.6: State transition diagram for the receiver part of the SWIS.*

Upon startup, the task takes care of initializing the SWIS, which includes the external flash, CSP and mutex’es along with loading data stored in EEPROM. In the test program developed, a dummy image is also generated, since it is currently not possible to transfer an image from a PC. After initializing, the flasher task is started from within the receiver.

The structure of the receiver part is the same as the flasher part, an infinite loop, implementing the state machine. In the idle mode, the receiver task is waiting to receive data. Upon reception, the state machine goes into the command handling state and acts depending on the received command. A list of possible commands are shown in table 8.4.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BL_I_DELETE_IMG</td>
<td>Image id</td>
<td>Delete information about a given image</td>
</tr>
<tr>
<td>1</td>
<td>BL_I_START_UPLOAD</td>
<td>Image id, CRC, size, client location, server location</td>
<td>Save info about image to temporary buffer</td>
</tr>
<tr>
<td>2</td>
<td>BL_I_CANCEL_UPLOAD</td>
<td>None</td>
<td>Cancel upload of image, i.e. empty temporary buffer</td>
</tr>
<tr>
<td>3</td>
<td>BL_I_GET_UPLOAD_STATUS</td>
<td>None</td>
<td>Request status for current image upload</td>
</tr>
<tr>
<td>4</td>
<td>BL_I_GET_LIST</td>
<td>None</td>
<td>Request list of images currently on the satellite</td>
</tr>
<tr>
<td>5</td>
<td>BL_I_CHECK_MEM</td>
<td>None</td>
<td>Check all stored images for CRC errors and return the result</td>
</tr>
<tr>
<td>5</td>
<td>BL_I_CHANGE_IMG</td>
<td>Image id, node id and new status</td>
<td>Mark a given image for flashing on a given node</td>
</tr>
</tbody>
</table>

*Table 8.4: List of states in the SWIS flasher task.*

### 8.7 Transferring Images from Computer to Satellite

A key requirement for the bootloader to work in reality is the ability to transfer software images from computer/GND to SWIS. They will eventually be uploaded over the space link, so a suitable function for this has to be implemented in the MCC. For testing purposes, it will however be beneficial to transfer them from a computer directly to CAN using a USB to CAN dongle. Transferring of a binary file from computer to SWIS could be done using the dongle and a special program, being able to read the files generated by the linker.
CHAPTER 8. BOOTLOADER & SOFTWARE IMAGE SERVER (SWIS)

Being able to update only single applications on MCUs requires that single software images can be produced. When an update to an application is made, its internal references might change because some code takes up less or more space. For this reason, applications cannot be allowed to depend on each other, if they are to be updated independently.

All applications will depend on some functions in the AAUSAT3 library, namely CSP, FreeRTOS and drivers. Applications will also depend on various files from AVR libc, such as stdlib. The main idea is to supply these files in one or a few static images, because the are unlikely to change.

8.8 Estimate of Time Needed to Transfer and Flash Software to Client

In this section, a rough estimate for the time needed to transfer a software image from SWIS to bootloader client is made. It considers a software image with a size of 20 kiB.

The process consists of three parts: A prologue of communication between SWIS and client, transfer of image, and an epilogue with checksum calculation and starting the kernel.

The prologue consists of negotiating whether a software flashing is needed and sending meta data about this update. This means two packets between server and client and ACK for these packets plus some time for internal processing. The same applies for the part concerning ending of transmission. These are believed to be on the order of a few milliseconds and can be neglected when compared to the time taken to do the actual flashing, which is shown in the following.

The most time-consuming part of the flashing process is transferring data from the SWIS and saving this to flash. First of all, according to the data sheet [Atm08], flashing one page of 256 bytes takes between 3.7 ms and 4.5 ms (assuming that the bootloader client flashes an entire page). Thus, the worst case time for the 20 kiB image is

\[
t_{\text{flash}} = \frac{20480 \text{ bytes}}{256 \text{ bytes/page}} \cdot 4.5 \text{ ms/page} = 360 \text{ ms}
\]  

Assuming that data cannot be transferred during flashing, these have to be transferred also. To transfer 256 bytes, two packets of 128 bytes length is needed. According to 9.2.5, the goodput of CSP is 133 kbit/s at that packet size. Thus, the time need for transfer can be estimated as

\[
t_{\text{transfer}} = \frac{20480 \text{ bytes}}{133000 \text{ bit/s}} = 1232 \text{ ms}
\]

The time taken for transferring and flashing the example image is therefore 360 ms + 1232 ms = 1592 ms. Minor variations due to prologue and epilogue, busy CAN bus etc. are not considered since they are believed to be much smaller than this number.

This rough estimate has shown that it takes between 1 and 2 seconds to flash 20.480 bytes and that this time is mainly dependent on the size of the image. An image half this size will thereby take about half the time to flash.
This chapter describes which tests that has been performed in order to test the different aspects of the performance for the software framework and the bootloader.

Five tests has been performed from which four of them covers the performance of the software framework including memory usage for a minimalistic test program, goodput of both external and internal communication as well as traffic utilization for external communication using CSP, the load of the MCU for both the transmitting and receiving board doing external communication and a stress test of CSP. The last test covers the performance of the bootloader by testing the startup time for both bootloader server and client.

The test files used for all the tests in this chapter is available on http://kom.aau.dk/group/08gr724/

### 9.1 RAM Usage

#### 9.1.1 Measurements and Expected Results

The purpose of this test is to determine memory consumption for different parts of the used software. This can be used for optimization, if a part use up a lot of RAM. The following is examined:

- SRAM used for one queue (with space for 1 byte).
- SRAM used for one semaphore.
- SRAM used for one connection struct.
- SRAM used in total for CSP.
- SRAM used for the software library according to compiler.
• SRAM used for a simple test program running two tasks on the same MCU which are com- municating with each other through CSP.

• SRAM used by the scheduler in the same test program.

The expected result is that its possible to have 3 applications on one MCU without running out of SRAM. Each application is expected to consist of two tasks as a minimum, which gives 8 tasks when the service layer and idle task is included. With a stack size of 200 bytes and idle stack size of 85 bytes this gives 1485 bytes of SRAM allocated for tasks. This leaves 2611 bytes for initialized and uninitialized global variables, dynamic allocation of memory for CSP receive buffers, UART buffer, queues and semaphores.

9.1.2 Test Method

The size of queues and semaphores can be determined by e.g. creating two queues and taking the difference in address of these two. The difference can then be printed to UART.

The size of a connection struct can be determined by the use of \texttt{sizeof(conn\_t)}.

The amount of RAM used for CSP can be determined by looking at the .map file of the mentioned test program and add up everything related to CSP.

The test program with two tasks should be a client in one task requesting a service from a server task who responds back. This minimalistic program will make it possible to give an estimate of how much SRAM a full system uses. By allocating one byte with malloc while running the program and printing the returned address its possible to calculate the total amount of SRAM used if no receive buffer is allocated. It is also possible to calculate the amount of SRAM the scheduler uses by subtracting all queues, semaphores, buffers and global variables.

9.1.3 List of Instruments

Table 9.1 gives the list of instruments used for the test.

<table>
<thead>
<tr>
<th>Instrument</th>
<th>AAU no.</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>n/a</td>
<td>Acer TravelMate 6292</td>
</tr>
<tr>
<td>JTAGICE-mkII</td>
<td>77081</td>
<td>Atmel’s on-chip debugging tool</td>
</tr>
<tr>
<td>Test board</td>
<td>n/a</td>
<td>[MPB’08, ]</td>
</tr>
<tr>
<td>Power supply</td>
<td>64341</td>
<td>Hameg instruments, triple power supply</td>
</tr>
</tbody>
</table>

\textit{Table 9.1: List of instruments used for the test.}

9.1.4 Test Procedure

1. Connect one test boards to the power supply (3.3 V).

2. Connect the JTAGICE-mkII device to a USB port on the computer and to the JTAG socket on the board.

3. Connect a serial cable to the computer and the board.

4. Open HyperTerminal (in windows) or a corresponding program.

5. Set up serial communication on the computer (38400 bit/s, 8 data bits, no parity bit, 1 stopbit, no flow control).
6. Go to the following directory `../memoryusagetest/boards/test_program`.

7. Open makefile and set TARGET to `memory_usage_test`.

8. Open a console and set working directory to the before mentioned.

9. Type the following command `make clean program` and press enter.

10. Observe the results in HyperTerminal.

### 9.1.5 Results of Measurement

Table 9.2 gives the total amount of used for the minimalistic test program. The program functionalities has been divided into 8 subgroups in order to accentuate the amount of used SRAM for different parts of the program.

<table>
<thead>
<tr>
<th>Part of program</th>
<th>Size [bytes]</th>
<th>Size [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection struct</td>
<td>312</td>
<td>7.62</td>
</tr>
<tr>
<td>UART</td>
<td>540</td>
<td>13.19</td>
</tr>
<tr>
<td>Queues</td>
<td>123</td>
<td>3.00</td>
</tr>
<tr>
<td>Semaphores</td>
<td>792</td>
<td>19.34</td>
</tr>
<tr>
<td>Tasks</td>
<td>599</td>
<td>14.63</td>
</tr>
<tr>
<td>Scheduler</td>
<td>236</td>
<td>5.76</td>
</tr>
<tr>
<td>Other</td>
<td>819</td>
<td>20.00</td>
</tr>
</tbody>
</table>

*Table 9.2: Amount of total SRAM used for different parts of the minimalistic test program in bytes and the corresponding percentage of used SRAM. Other represents the remaining parts of the test program and is possible to identify in the mapping file for the test program (memory_usage_test.map).*

Figure 9.1 illustrates the percentage of used SRAM by the minimalistic test program.

*Figure 9.1: Percentage of total SRAM used for different functionalities by a minimalistic test program.*

The total amount of used SRAM for the minimalistic test program add up to approximately 83 %. About 42 % of of these is used by freeRTOS to handle tasks and semaphores created in CSP.
9.2 CSP Goodput, Overhead and Traffic Utilization

9.2.1 Measurements and Expected Results

The purpose of this test is to measure the maximum achievable goodput of CSP when a baud rate of 500 kbit/s is chosen and only two nodes are communicating on the network. The goodput is measured while ACKing all packages and is measured in bits of transferred data per second, not including overhead, where overhead is the header and trailer of data frames and ACK frames. An error rate of zero is assumed and is checked by printing can_status, which is a global struct keeping track of errors. The goodput when doing internal communication is also measured.

The traffic utilization on the network is then measured and is calculated according to equation 9.1. \( t_{frame} \) is time used by one frame on the network, \( t_{ifs} \) is the minimum time between two frames according to the CAN standard, see chapter 3, and the sum of these over time used for one iteration in the test program, will give the utilization.

\[
\text{traffic utilization [\%]} = \frac{\sum (t_{frame} + t_{ifs})[s]}{\text{time used for one iteration [s]}} \cdot 100\% 
\] (9.1)

This calculation is only of interest for communication which goes through CAN.

Goodput is measured when sending 1, 8, 9, 16, 17, 32, 33, 64, 65, 128, 129 and 248 byte packages. 1 and 248 bytes are chosen because they are minimum and maximum package size accordingly. The reason why the other chosen values are 8, 9 and 16, 17 and so on is because this is the boundary between sending only full frames or sending a last frame with only one byte.

The traffic utilization is measured for packet size of 1, 64 and 248 bytes as well as 8 and 9 bytes. The three values has been chosen in order to get some random samples of the possible packet sizes, the last two values is chosen because this, as mentioned before, is the boundary between sending only full frames or sending a last frame with only one byte.

Amount of overhead compared to actual data, in percentages, is also determined in this test.

There is no expected results for this test, but the overhead percentage can never be better than 50% due to the architecture of the extended ID frame, see figure 3.6 in chapter 3.

9.2.2 Test Method

Goodput can be measured by having one node sending packages to another node. When the sending node receives an ACK it immediately sends another package and over a period of one second the total amount data received is calculated, by printing can_status which also contains number of transmitted and received frames. When measuring goodput during internal communication a counter is kept to count number of received packages.

Times needed for calculating traffic utilization is measured on the CAN wire with an oscilloscope.

The amount of overhead can be calculated by adding the header for all the data frames and the header for the ACK frame for one package.

9.2.3 List of Instruments

Table 9.3 gives the list of instruments used for the test.
<table>
<thead>
<tr>
<th>Instrument</th>
<th>AAU no.</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>n/a</td>
<td>Acer TravelMate 6292</td>
</tr>
<tr>
<td>JTAGICE-mkII</td>
<td>77081</td>
<td>Atmel’s on-chip debugging tool</td>
</tr>
<tr>
<td>Test board</td>
<td>n/a</td>
<td>[MPB^308, ]</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>57339</td>
<td>Agilent 54621D, mixed signal oscilloscope</td>
</tr>
<tr>
<td>Power supply</td>
<td>64341</td>
<td>Hameg instruments, triple power supply</td>
</tr>
</tbody>
</table>

Table 9.3: List of instruments used for the test.

9.2.4 Test Procedure

1. Connect two test boards to the power supply (3.3 V).
2. Connect the JTAGICE-mkII device to a USB port on the computer and to the JTAG socket one of the boards (server board).
3. Connect a serial cable to the computer and the other board (client board).
4. Connect the two boards by a CAN wire.
5. Connect oscilloscope to I/O pin 0 on the client board and one of the two CAN wires.
6. Open HyperTerminal (in windows) or a corresponding program.
7. Set up serial communication on the computer (38400 bit/s, 8 data bits, no parity bit, 1 stopbit, no flow control).
8. Go to the following directory ../csp_goodputtest/boards/test_program.
9. Open makefile and set TARGET to csp_goodput_test_server.
10. Open a console and set working directory to the before mentioned.
11. Type the following command make clean program and press enter.
12. Connect the JTAGICE-mkII device to the client board.
13. Change the makefile target to csp_goodput_test_client.
14. Open csp_goodput_test_client.c data_size (line 78) to 1.
15. Repeat point 11.
16. Wait for one second and observe data printed in HyperTerminal.
17. Press the reset button on the test board and wait for new results to be printed in HyperTerminal.
18. Repeat point 17 until ten results has been been obtained.
19. Repeat point 14 to 18 with data_size set to 8, 9, 16, 17, 32, 33, 64, 65, 128, 129 and 248.
20. Repeat point 14 and 11 with data_size set to 1.
21. Capture a still picture of the output on I/O pin 0 and CAN wire on the oscilloscope.
22. Measure the time between, when the application starts sending a packet and the data frame is visible on the can wire, two data frames, a data frame and an acknowledgment frame and an acknowledgment frame and the start of sending a new packet.
23. Repeat point point 14, 11 21 and 22 with data_size set to 8, 9, 64 and 248.

24. Disconnect the CAN wire from the client board.

25. Change the makefile target to `csp_goodput_test_internal`.

26. Repeat point 14 to 19 while disregarding point 18.

### 9.2.5 Results of Measurement

The results of the goodput test for both internal and external communication, overhead and traffic utilization, for a given packet size, is summarized in table 9.4. The noted goodput values for external communication is the mean of ten samples.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.8</td>
<td>5.2</td>
<td>94.4</td>
<td>20.4</td>
</tr>
<tr>
<td>8</td>
<td>124.9</td>
<td>37.3</td>
<td>67.7</td>
<td>20.8</td>
</tr>
<tr>
<td>9</td>
<td>140.2</td>
<td>36.0</td>
<td>73.6</td>
<td>24.1</td>
</tr>
<tr>
<td>16</td>
<td>245.6</td>
<td>61.9</td>
<td>61.1</td>
<td>n/a</td>
</tr>
<tr>
<td>17</td>
<td>260.4</td>
<td>56.7</td>
<td>66.3</td>
<td>n/a</td>
</tr>
<tr>
<td>32</td>
<td>475.9</td>
<td>85.4</td>
<td>56.7</td>
<td>n/a</td>
</tr>
<tr>
<td>33</td>
<td>489.7</td>
<td>88.0</td>
<td>60.4</td>
<td>n/a</td>
</tr>
<tr>
<td>64</td>
<td>895.5</td>
<td>115.2</td>
<td>54.1</td>
<td>49.1</td>
</tr>
<tr>
<td>65</td>
<td>908.4</td>
<td>107.9</td>
<td>56.3</td>
<td>n/a</td>
</tr>
<tr>
<td>128</td>
<td>1602.6</td>
<td>133.5</td>
<td>52.7</td>
<td>n/a</td>
</tr>
<tr>
<td>129</td>
<td>1613.0</td>
<td>129.0</td>
<td>53.9</td>
<td>n/a</td>
</tr>
<tr>
<td>248</td>
<td>2593.1</td>
<td>147.3</td>
<td>51.9</td>
<td>60.7</td>
</tr>
</tbody>
</table>

*Table 9.4: CSP goodput, overhead and traffic utilization. Goodput is noted as the mean of ten samples. Traffic utilization has only been measured for 5 package sizes.*

The values for traffic utilization is calculated from the measured time between, when the application starts sending a packet and the data frame is visible on the can wire, two data frames, a data frame and an acknowledgment frame as well as an acknowledgment frame and the start of sending a new packet (see table 9.5 and 9.6).

Figure 9.2 illustrates the goodput for internal communication with a given packet size.

*Figure 9.2: The goodput of internal communication for a given packet size.*
## CHAPTER 9. PERFORMANCE TESTS

Table 9.5: Measured time between, an application starts sending a packet to a data frame is visible on the can wire (Begin-Frame), two data frames (Frame-Frame), a data frame and an acknowledgment frame (Frame-Ack) as well as an acknowledgment frame and the start of sending a new packet (Ack-Begin). Time is measured in µs. The first column indicates the sent packet size in bytes (part 1).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>648</td>
<td>1098</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>400</td>
<td>346</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>634</td>
<td>1118</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>366</td>
<td>382</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>366</td>
<td>387</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>875</td>
<td>743</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>N/A</td>
<td>1148</td>
<td>475</td>
</tr>
<tr>
<td>1</td>
<td>90</td>
<td>N/A</td>
<td>835</td>
<td>780</td>
</tr>
<tr>
<td>8</td>
<td>90.60</td>
<td>N/A</td>
<td>442</td>
<td>1175</td>
</tr>
<tr>
<td>8</td>
<td>90.40</td>
<td>N/A</td>
<td>393</td>
<td>380</td>
</tr>
<tr>
<td>9</td>
<td>60.40</td>
<td>118</td>
<td>1013</td>
<td>352</td>
</tr>
<tr>
<td>9</td>
<td>60.40</td>
<td>118</td>
<td>393</td>
<td>978</td>
</tr>
<tr>
<td>9</td>
<td>61.40</td>
<td>116</td>
<td>388</td>
<td>978</td>
</tr>
<tr>
<td>9</td>
<td>61.40</td>
<td>116</td>
<td>615</td>
<td>750</td>
</tr>
<tr>
<td>9</td>
<td>61.40</td>
<td>116</td>
<td>470</td>
<td>892</td>
</tr>
<tr>
<td>64</td>
<td>90.60</td>
<td>108</td>
<td>464</td>
<td>497</td>
</tr>
<tr>
<td>64</td>
<td>90.60</td>
<td>108</td>
<td>538</td>
<td>380</td>
</tr>
<tr>
<td>64</td>
<td>90.20</td>
<td>108</td>
<td>1088</td>
<td>873</td>
</tr>
<tr>
<td>64</td>
<td>91.40</td>
<td>108</td>
<td>840</td>
<td>1120</td>
</tr>
<tr>
<td>64</td>
<td>91.80</td>
<td>108</td>
<td>530</td>
<td>430</td>
</tr>
<tr>
<td>248</td>
<td>90.40</td>
<td>108</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>248</td>
<td>90.40</td>
<td>108</td>
<td>764</td>
<td>434</td>
</tr>
</tbody>
</table>

Table 9.6: Measured time between, an application starts sending a packet to a data frame is visible on the can wire (Begin-Frame), two data frames (Frame-Frame), a data frame and an acknowledgment frame (Frame-Ack) as well as an acknowledgment frame and the start of sending a new packet (Ack-Begin). Time is measured in µs. The first column indicates the sent packet size in bytes (part 2).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>248</td>
<td>91.00</td>
<td>108</td>
<td>628</td>
<td>570</td>
</tr>
<tr>
<td>248</td>
<td>91.20</td>
<td>108</td>
<td>1135</td>
<td>1075</td>
</tr>
<tr>
<td>248</td>
<td>91.20</td>
<td>108</td>
<td>1098</td>
<td>1113</td>
</tr>
</tbody>
</table>

Figure 9.3 illustrates the goodput for external communication with a given packet size.

Figure 9.3 illustrates the traffic utilization for external communication with a given packet size.

The results show that it is possible to transfer data externally with a goodput of up to 147.3 kbit/s and internally with speeds up to 2593.1 kbit/s when using a packet size of 248 bytes. This means that internal communication is 17.6 times faster than external communication.

The traffic utilization of the CAN wire is approximately 60 % when a packet size of 248 bytes is used.

### 9.2.6 Uncertainty of Measurements

The time measured between, when the application starts sending a packet to a data frame is visible on the can wire (Begin-Frame), two data frames (Frame-Frame), a data frame and an acknowledgment frame
Figure 9.3: The mean goodput of external communication for a given packet size.

Figure 9.4: The traffic utilization of external communication for a given packet size.

(Frame-Ack) as well as an acknowledgment frame and the start of sending a new packet (Ack-Begin) has only been measured for one iteration and then assumed to be the same for all other iterations.

Further more the before mentioned times was measured from a dominant bit to another dominant bit. This means that the acknowledgment delimiter, end of frame and inter frame space bits has all been assumed to have a length of $2 \mu s$ in accordance with a baud rate of 500 kB/s. The hardware insertion of stuff bits have not been taken into account, whilst performing the tests, as the ensurement of not having, or having the same amount of, stuff bits is quite comprehensive, and can not be controlled upon implementation.

9.3 Load on MCU with Continuous CSP Communication

9.3.1 Measurements and Expected Results

The purpose of this test is to measure the load on the MCU whilst sending data from one application to another as fast as possible.
The load is measured both on the sending node (client) and the receiving node (server) during external communication. For internal communication this test is not relevant due to the MCU will be fully loaded doing continuous transmissions and receiving.

The measurements are performed when sending 1, 8, 9, 16, 17, 32, 33, 64, 65, 128, 129 and 248 byte packages. 1 and 248 bytes are chosen because they are minimum package size and maximum package size accordingly. The reason why the other chosen numbers are 8, 9 and 16, 17 and so on is because this is the boundary between sending only full frames or sending a last frame with only one byte.

The expected result is that the load on the MCU for sending or receiving data is around the value, found in the test made by group 07gr722, which was 30% or preferably lower, meaning most of the time should be spent by the CAN controller feeding bits to the network.

**9.3.2 Test Method**

The load on the MCU can be measured by investigating the time where the operating system (FreeRTOS) is running its idle task. This task will only run when no other tasks, functions or ISR’s are running. The time spent can be determined by setting and clearing I/O pins and taking measurements with an oscilloscope. By summarizing the time spent in the idle task, after one iteration in the test program, it is possible to calculate the load in percentages, denoted $\Gamma$, of the MCU. Equation 9.2 gives the load as a function of time spent in the idle task (sum of $t_{idle}$) [Sch07, s.5].

$$\Gamma[\%] = \left(1 - \frac{\sum t_{idle}[s]}{\text{time used for one iteration [s]}}\right) \cdot 100\%$$ (9.2)

In order to measure the idle time an I/O pin is set every time the idle task is entered and cleared every time a context switch or an interrupt is made.

**9.3.3 List of Instruments**

Table 9.7 gives the list of instruments used for the test.

<table>
<thead>
<tr>
<th>Instrument</th>
<th>AAU no.</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>n/a</td>
<td>Acer TravelMate 6292</td>
</tr>
<tr>
<td>JTAGICE-mkII</td>
<td>77081</td>
<td>Atmel’s on-chip debugging tool</td>
</tr>
<tr>
<td>Test board</td>
<td>n/a</td>
<td>[MPB+08, ]</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>57339</td>
<td>Agilent 54621D, mixed signal oscilloscope</td>
</tr>
<tr>
<td>Power supply</td>
<td>64341</td>
<td>Hameg instruments, triple power supply</td>
</tr>
</tbody>
</table>

*Table 9.7: List of instruments used for the test.*

**9.3.4 Test Procedure**

1. Connect two test boards to the power supply (3.3 V).
2. Connect the JTAGICE-mkII device to a USB port on the computer and to the JTAG socket one of the boards (server board).
3. Connect a serial cable to the computer and the other board (client board).
4. Connect the two boards by a CAN wire.

5. Connect oscilloscope to I/O pin 0 on the client board and I/O pin 8 on both test boards.

6. Go to the following directory: `../csp_mcu_loadtest/boards/test_program/`

7. Open makefile and set TARGET to `csp_mcu_load_test_server`.

8. Open a console and set working directory to the before mentioned.

9. Type the following command: `make clean program` and press enter.

10. Connect the JTAGICE-mkII device to the client board.

11. Open `csp_mcu_load_test_client.c` and set `data_size` (line 73) to 1.

12. Change the makefile target to `csp_mcu_load_test_client`.

13. Repeat point 9 and observe the output on the oscilloscope.

14. The time to send a frame is measured on I/O pin 0. Observe the output for one minute to see if the time is stable or if it varies between a minimum time and a maximum time.

15. Press the run/stop button on the oscilloscope to capture a still picture of the event with the minimum transmit time.

16. Save the still picture of the idle time measured on I/O pin 8 for both test boards in a CSV file.

17. Unless the transmit time is stable repeat point 15 and 16 to capture a still picture of the max. time.

18. Repeat point 13 to 17 with data_size set to 8, 9, 16, 17, 32, 33, 64, 65, 128, 129 and 248.

19. Go to the following directory: `../csp_mcu_loadtest/results/`

20. Open the `csp_mcu_load_test.m` file in Matlab.

21. Set printfigures to 0 and change the directory under the csvread function to the directory of the obtained data.

22. Enter the correct name of the CSV file in the csvread function and set `Tsent` to the corresponding transmit time.

23. Run the script to calculate the load of both server and client.

24. Repeat point 22 and 23 for all CSV files.

### 9.3.5 Results of Measurement

The obtained results including the calculated loads of server and client is summarized in table 9.8. Figure 9.5 illustrates the maximum measured load on both server and client for a given packet size. Figure 9.6 illustrates the minimum measured load on both server and client for a given packet size. The results indicates that the MCU load has increased up to a mean of 44.06 % for the server and 41.90 % for the client. This corresponds to an increase of respectively roughly 14 % and roughly 12 % compared to the results of group 07gr722 which was 30 %.

The increase is properly partial due to the implementation of the FreeRTOS kernel which have a large context switching time compared to the FLAK kernel. The context switching time has been measured to 79.2 µs which corresponds to roughly 8 % of the CPU time (see section 6.3).
### 9.3.6 Uncertainty of Measurements

The I/O pin measuring the idle time is not cleared before the ISR routine is entered. Thereby the time to due the actual interrupt is not taking into account. In section 6.3 the interrupt time (with no code inside) has been measured to 8.5 µs.
CHAPTER 9. PERFORMANCE TESTS

9.4 Stress Test

9.4.1 Measurements and Expected Results

The purpose of this test is to see how well CSP and FreeRTOS works for a longer period of time with continuous communication going both ways between a client and server. This is both tested with internal and external communication.

The measures used on both nodes during external communication are received frames, send frames, ACK errors, form errors, stuffing errors, bit errors, CRC errors, receiver overflows, transmitter overflows, transmitter timeouts and receiver timeouts which are all members of the can_status_t struct. A predefined message is sent each time and the receiving node checks if the correct message is received.

The measures used during internal communication are transmitter timeouts, receiver timeouts, received packages, send packages and message errors (if data send is not equal to data received).

The expected result is close to zero errors during a period of one hours, no memory leak and zero unclosed connections.

9.4.2 Test Method

During the test with internal communication and the test with external communication. The client makes a connection, sends a message to the server which responds back with the same message. After 10 request-replies the connection is closed and a new connection is made. This is run within an infinite loop.

The reason for connecting and closing continuously is to repeatedly use malloc() and free() for rxbuffer. This also tests the connect/close part of the software and not only send and receive.

The used message is 0123456789 which requires two frames in order to have both BEGIN and MORE frames.
9.4.3 List of Instruments

Table 9.9 gives the list of instruments used for the test.

<table>
<thead>
<tr>
<th>Instrument</th>
<th>AAU no.</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>n/a</td>
<td>Acer TravelMate 6292</td>
</tr>
<tr>
<td>JTAGMKII-ice</td>
<td>77081</td>
<td>Atmel’s on-chip debugging tool</td>
</tr>
<tr>
<td>Test board</td>
<td>n/a</td>
<td>[MPB ’08, ]</td>
</tr>
<tr>
<td>Power supply</td>
<td>64341</td>
<td>Hameg instruments, triple power supply</td>
</tr>
</tbody>
</table>

*Table 9.9: List of instruments used for the test.*

9.4.4 Test Procedure

1. Connect two test boards to the power supply (3.3 V).
2. Connect the JTAGMKII-ice device to a USB port on the computer and to the JTAG socket one of the boards (server board).
3. Connect a serial cable to the computer and the other board (client board).
4. Connect the two boards by a CAN wire.
5. Open HyperTerminal (in windows) or a corresponding program.
6. Set up serial communication on the computer (38400 bit/s, 8 data bits, no parity bit, 1 stopbit, no flow control).
7. Go to the following directory `../stresstest/bords/stresstest/`.
8. Open makefile and set TARGET to `server_stress_test`.
9. Open a console and set working directory to the before mentioned.
10. Type the following command `make clean program` and press enter.
11. Connect the JTAGMKII-ice device to the client board.
12. Open `client_stress_test.c` and make sure that the `vTaskDelay()` function (line 85) has 2 as the input argument.
13. Change the makefile target to `client_stress_test`.
15. Wait for one hour and read off the results in HyperTerminal.
16. In line `client_stress_test.c` change the input argument for the `vTaskDelay()` function (line 85) to 5.
17. Repeat point 10 and 15.
18. Disconnect the CAN wire from the client board.
19. Change the makefile target to `cli_serv_stress_test`.
20. Repeat point 10 and 15.
9.4.5 Results of Measurement

Table 9.10 gives the amount of transactions and errors during one hour for both external and internal communication. One transaction is equal to 3 sent and 3 received frames.

<table>
<thead>
<tr>
<th></th>
<th>Ext. comm. (2 ms delay)</th>
<th>Ext. comm. (5 ms delay)</th>
<th>Int. comm. (2 ms delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK errors</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Bit errors</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>CRC errors</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Form errors</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Stuffing errors</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Receiver overflows</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Trans. overflows</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>Trans. timeouts</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Receiver timeouts</td>
<td>1416</td>
<td>1459</td>
<td>0</td>
</tr>
<tr>
<td>Transactions</td>
<td>1259323</td>
<td>1132970</td>
<td>2999999</td>
</tr>
<tr>
<td>Message errors</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.10: Number of transactions and detected errors during one hour. Ext. and Int. comm. means external and internal communication and the brackets indicates the delay between close and connect calls on the client side.

These results shows that there was no detected errors for communicating between two applications located on the same test board. When communicating between two test boards, with a timeout of 2 ms between close and connect calls, 1416 receiver timeouts was detected which corresponds to an error rate of 0.11 %. With a delay of 5 ms the error rate is increased to 0.13 %.

9.5 Bootloader Startup Delay

9.5.1 Measurements and Expected Results

The purpose of this test is to show how much extra time it will take to boot a given hardware platform when the bootloader is present, compared to when it is not.

The fact that the bootloader has to start up, connect to the SWIS and request information about what to do requires some additional time before the kernel and applications can be started. It is interesting to investigate this performance penalty, to evaluate whether or not it will have an impact on the overall performance. The extra time will be measured and compared with the total normal boot time. This delay can also be used to determine a timeout for the bootloader client in case no contact is established with the SWIS.

Casual observations about the time taken for each of the operations during booting are also made to evaluate where to put an optimization effort, should this be needed.

9.5.2 Test Method

Testing timing is most efficiently done by asserting and negating output pins found on the hardware. This eases the measurements of signals at a very low computational cost. During these tests all debug output, using the UART, are disabled since `printf()` is a computationally heavy function.

Because the metric actually wanted is the extra time needed for the bootloader, also the normal startup time needs to be measured. This leads to the following list of timings to be measured:
• Time from reset to entering main for bootloader client
• Time from entering main to launch of kernel on bootloader client
• Time from reset to entering main for arbitrary subsystem (SWIS chosen here for convenience)
• Time from entering main to tasks actually running on SWIS

9.5.3 List of Instruments

Table 9.11 gives the list of instruments used for the test.

<table>
<thead>
<tr>
<th>Instrument</th>
<th>AAU no.</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>n/a</td>
<td>Dell Inspiron 510m</td>
</tr>
<tr>
<td>JTAGMKII-ice</td>
<td>77081</td>
<td>Atmel’s on-chip debugging tool</td>
</tr>
<tr>
<td>Test board</td>
<td>n/a</td>
<td>[MPB+08, ]</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>57339</td>
<td>Agilent 54621D, mixed signal oscilloscope</td>
</tr>
<tr>
<td>Power supply</td>
<td>64341</td>
<td>Hameg instruments, triple power supply</td>
</tr>
</tbody>
</table>

*Table 9.11: List of instruments used for the test.*

9.5.4 Test Procedure

1. Connect two test boards to the power supply (3.3 V).
2. Connect the JTAGMKII-ice device to a USB port on the computer and to the JTAG socket one of the boards (server board).
3. Connect a serial cable to the computer and the other board (client board).
4. Connect the two boards by a CAN wire.
5. Connect oscilloscope probes to pin 0, 1 and 2 of the output port and the reset pin of the MCU.
6. Go to `..\swistest\boards\swistest`.
7. Open a console and set working directory to the before mentioned.
8. Program the SWIS board using the following command line `make clean program`.
9. Press the reset button and freeze the result on the oscilloscope.
10. Record the time taken from reset to rise of the first and second pin.
11. Connect oscilloscope probes to pin 0 and 1 of the output port on the client and reset pin of the MCU.
13. Press down reset on both the server and the client at the same time.
14. Release reset first on the server and after 1 sec. on the client.
15. Observe the result on the oscilloscope and freeze it.
16. Record the time taken from reset on the client until rise of the first and second pin.
CHAPTER 9. PERFORMANCE TESTS

9.5.5 Results of Measurement

Table 9.12 gives the startup time of different operations for the bootloader server.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Startup time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0</td>
</tr>
<tr>
<td>Entering main (pin 0)</td>
<td>1660</td>
</tr>
<tr>
<td>Started receiver task (pin 1)</td>
<td>680</td>
</tr>
<tr>
<td>Started and initialized flasher task (pin 2)</td>
<td>4720</td>
</tr>
</tbody>
</table>

*Table 9.12: Measurements of startup time for the bootloader server.

Table 9.13 gives the startup time of different operations for the bootloader client.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Startup time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0</td>
</tr>
<tr>
<td>Entering bootloader (pin 0)</td>
<td>850</td>
</tr>
<tr>
<td>Starting kernel (pin 1)</td>
<td>1580</td>
</tr>
</tbody>
</table>

*Table 9.13: Measurements of the startup time for the bootloader client.

The above states a total startup time of 7060 µs for the bootloader server and 2430 µs for the bootloader client.

It can be seen that the time from reset to entering the main function is quite different on the SWIS and the bootloader client. This is likely due to the fact that many more things have to be initialized in the SWIS before main. This issue is not investigated further here.

The time it takes for the bootloader client to contact the server, get a response and be ready to execute the kernel is 1580 µs. Compared to the general startup time for the SWIS which is used as an example, this number is small. It should not have any practical impact during normal use of the satellite. A timeout has to be specified for the bootloader client in case no response is received from the SWIS. With the knowledge of the normal response time this timeout can be chosen.

9.5.5.1 Casual Results

Along with the formal results, a number of informal results have been obtained to show more in detail how the time is spend during bootloading. It shows that most of the time can be attributed to the fact that information have to be transferred over the CAN-bus, handled by CSP and sent back. The actual time needed to process a request on the SWIS have been measured to 130 µs, but this would depend on the number of images available on the server and other things. It can be used to show that little or no performance gain would arise from optimizing server software.

9.5.6 Uncertainty of Measurements

The test consists of simple timing measurements using an oscilloscope, which in it self has good precision. Some uncertainty arises from the conditions of the test. Times from reset to entering the main function is affected by the used version of libraries, and the specifics of the code since variables is initialized before main is called. The time needed to transfer data over the CAN-bus is non-deterministic since it would depend on other traffic. In cases with very high load, it could be possible that transferring of data would even exceed a timeout.
These uncertainties are however considered non-critical since the purpose of the test is only to do a rough estimate of the time overhead introduced by the bootloader process.
This project has been concerned with improving the software framework for AAUSAT3, thereby making it more robust, reconfigurable, and testable. The aim is to make it easier to develop applications for AAUSAT3. Several improvements have been proposed to achieve these goals.

10.1 Robustness

As AAUSAT3 is a non-serviceable autonomous system, robustness is a key issue. During the project many bigger and minor improvements have been proposed, while some issues still need attention. During all tests, the software framework showed to be very stable and behaving as expected. During CSP stress test, receive timeouts was encountered. The reason for these timeouts should be further investigated.

Introducing FreeRTOS gained a lot of wanted and needed functionality, mainly a preemptive kernel, queues and semaphores. The added functionality also introduced new possibilities for errors, both in FreeRTOS itself or as a result of its use. Errors in FreeRTOS or in the use hereof will be difficult to detect so developers need be well informed about the usage.

Further work can be done to validate robustness. It should be validated that global variables are accessed atomicly where it is needed so as not to have undefined behavior as a result of updating these during a read. A more formal model of the CSP state machine can be developed for validation and verification, e.g. using UPPAAL.

In the proposed version of CSP, dynamic memory allocation is still used for receive buffers. Due to the non-deterministic behavior of dynamic memory allocation, changes should be made to avoid this. It would furthermore eliminate the concept of memory leaks and undefined operation due to memory fragmentation or lack of RAM.
10.2 Reconfigurability

Moving of applications was eased by making a loopback function in CSP. This enabled applications residing on the same hardware to communicate over CSP whether or not they were located on the same MCU. This feature significantly improved the reconfigurability of AAUSAT3 and ease of development as no special care has to be taken when programming.

A bootloader client and server was designed and partly implemented. It allows application images to be uploaded to the satellite and then transferred to the individual MCUs. The final version will improve robustness and reconfigurability by being able to update software with new functionality or correct errors. Robustness must however be assured by using a good fallback mechanism in case new faulty software results in errors on the satellite.

A choice was made not to prioritize support for moving of applications once the satellite is in space, since this would complicate the design. The proposed design, however, reduces software upload time to the satellite by reducing the image size, which is a big gain in comparison with AAUSAT-II. It was also shown that the bootloader would not slow down the startup process significantly.

10.3 Testability

Part of the concept of having a common software framework for applications is to be able to reuse libraries and kernel on different MCUs. By re-using software, testing only needs to be done once and thus can be more rigorous.

A system with deterministic behavior would greatly improve testability by being able to reproduce tests. However, goodput during internal communication is a lot faster than external, which has to be considered when testing synchronization and timing. This could be solved by using the physical CAN bus during loopback, and thus also easing debug by sniffing traffic. The downside is that it requires turning on at least two MCUs, which is not desirable.

Memory handling is non-deterministic at the moment due to the use of malloc() and free() for receive buffers. These functions cannot be assumed to behave the same each time they are run and in case of errors or low memory, hard detectable errors can occur. Therefore they should not be used and receive buffers should be statically allocated.

10.4 Ease of Development

The success of a software framework is largely dependent on the documentation available. User guides was developed for FreeRTOS and CSP to ease the startup process for new developers. An improvement to user manuals could be a single, consolidated source of information. At the moment information is distributed among a number of sources, each having different formats and authors.

During this project, it has become clear that the chosen MCU puts some constraints on applications running. If applications need to send or receive data at high rate and process these at the same time, the processing power may not be sufficient. Generally the amount of RAM available, 4 kiB, is very limiting. The library, communication and kernel uses up a large portion of the RAM available, leaving little to applications. This can pose a serious problem if a given application has a need for a large amount of RAM.
In CSP throughput tests, it has been shown that CPU load for CSP has increased somewhat compared to the original version of CSP [RJdCCA07]. This was expected because extra functionality has been added to CSP as well as FreeRTOS using about 8% of the CPU time for context switching.

One possible solution is to supply the MCU with external RAM, which is easy to do, but this solution takes up more power and space. Another solution could be to slim down some of the memory-consuming parts, but it will not be possible to gain very much from this without impacting performance and ease of development. A third possibility is to select another and more powerful MCU with more RAM. This will also help on the fact that FreeRTOS uses a lot of the CPU time for context switching. If the third option is not chosen, it might be an idea to consider if the computational overhead incurred by preemption is worth it.

The current software framework is usable for application development and because it is highly generic, it can easily be ported to other MCUs within the AVR family. A long term perspective for the AAUSAT3 software framework is a proposal to make it available as open source software under GPL. It would then be possible to make use of the work in other projects, both for satellites and distributed systems in general. This is however still a future vision as it requires clean interfaces, documentation etc.
Bibliography


