Software and software architecture for a student satellite

by

Dan Erik Holmstrøm

December 21, 2012

TDT4501 - Computer Science,
Specialisation Project

NTNU
Norwegian University of
Science and Technology

Abstract

Modern satellites use computer software to do on-board data handling and other tasks that are elemental to their successful operation. Design and development of such software poses several challenges due to factors such as the physical environment, reliability, security as well as cost and power constraints.

This paper addresses the software system of the NTNU test satellite. The satellite is a 2U CubeSat where the goal is to build a modular platform that is not specific to any particular payload. We discuss the requirements for an on-board computer module and highlight possible challenges and limitations with the current system design. The modules of the satellite communicate with each other over an I^2C bus. This means that inter-module communication may be hampered by failure in anything connected to the bus. Therefore, we discuss possible methods to mitigate the effects of this kind of interference.

When investigating how to satisfy system requirements, it is possible to benefit from existing software. We find that a communications library called cubesat space protocol can be used for inter-module communication, and communication between ground and space segment. In addition, we find that FreeRTOS is a suitable operating system for the on-board computer module.

Some changes had to be made to make cubesat space protocol and FreeRTOS suitable for use in this project. The necessary changes and challenges are described, including porting the cubesat space protocol library to Microsoft Windows.

In conclusion, we find that a lot of time can be saved by not developing all the software from scratch. However, we realise that there is a lot of software features and challenges that will have to be solved in the future.

Contents

List of Figures

List of Tables

List of abbreviations

ADCS	Attitude Determination and Control System
Cal Poly	California Polytechnic State University
CDHS	Command and Data Handling
CDMS	Command and Data Monitoring System
COTS	Commercial-off-the-shelf
CSP	Cubesat space protocol
EPS	Electrical Power Supply system
ICE	In-Circuit Emulator
MMU	Memory Management Unit
MPU	Memory Protection Unit
NUTS	NTNU Test Satellite
OBC	On-board Computer
OBDH	On-board Data Handling
OTP	One-Time Programmable

Preface

This paper is one of the deliverables in the course TDT4501 Specialisation Project, given by IDI at the Norwegian University of Science and Technology, NTNU, in the fall of 2011.

Initially, the assignment described the need to manage the main data bus in the satellite. However, as the project evolved it was clear that the focus had to be changed as well as the scope of the project. This was the first semester in the NUTS project, where an assigment put any focus on how software could be used to make the satellite function as a whole. Because of this, the development of the system software for the NUTS satellite was – and still is – in its infancy.

Given the limited resources available, it means that there is still a lot to do. And my effort is only a tiny step towards completion. But this also means that I am in a better position to focus on exactly the tasks that I find interesting.

Before implementing anything, one has to to decide what to make. The formalisation of this, is a list of requirements. Thus I had to create a software requirements specification. This kind of documentation is a work in progress-document, and is bound to change in the future. However, what is not possible to avoid, is communication between the different subsystems as well as the satellite and the ground. Implementation wise, this is also where I have put most of the effort this semester. I hope that people will find that at least some of the decisions taken when solving the assignment, were sound. And that they are willing to continue where I left off.

I would also like to express my appreciation to the people who have helped me realise this project. This includes my supervisor, Gunnar Tufte, and the NUTS project leader, Roger Birkeland, for helpful input and encouragment. Last, but not least, I would like to thank those who had to endure long workshop sessions with me.

Part I

Project overview and planning

Project description 1

Introduction

This chapter presents the project and describes how this specific assignment fits into the NTNU test satellite project as a whole. At last, this project is compared with similar CubeSat projects.

1.1 Project mandate

The purpose of the project is to specify the software requirements for a student satellite, and see how inter-module and satellite-ground communication could be implemented. Part of the investigative processs is to decide whether to use COTS software, design or implementation issues and see if software developed in the earlier stages of the project could be used.

While the ultimate goal of the NUTS mission, is to put the payload into orbit, it is the satellite platform that will make it usable. And for this to happen, a lot of elements have to play well together. This report describes the beginnings of creating the computer software in this system and the process of investigating possible implementations using freely available software.

The name of the specific project addressed in this report is ”Software and software architecture for a student satellite”. When using the term ”project” throughout this report, it can mean both the project directly related to this report or the NUTS project as a whole, depending on the context.

1.2 Project background

The Norwegian Student Satellite Project, ANSAT, was launched in 2006 [2] by a collaborative effort between NAROM and Andøya Rocket Range [17]. The goal of the ANSAT project is to launch three student satellites by the end of 2014. The common denominator with all of the three satellites, is that students shall play a significant role in realising the goal of putting a functioning satellite into earth orbit. The NTNU test satellite, NUTS, is one of them.

The work on NUTS started in September 2007. It is a 2U CubeSat, meaning it has twice the volume of a "1U" CubeSat. A CubeSat has a volume of 1 litre and measures $10x10x10cm$ [23] and a mass up to $1.33kg$. One of the design goals is to create a generic and modular satellite platform [9]. At the time of this writing, the payload is an IR camera. The camera will be used to observe gravity waves in the earth atmosphere. In addition, the radio and the on-board computer module are fitted with wireless transceivers that could be used to test the feasibility of using a wireless bus in space.

The CubeSat standard was created at California Polytechnic State University. Its purpose is to provide a standard for picosatellites. An established standard helps to reduce development time and cost. A lot of firms provide equipment adhering to the CubeSat standard, and many universities and educational institutions choose to buy complete modules from these vendors.

When creating CubeSats, it is common to use COTS hardware. However, this would limit the possibilities for students designing the electronics in the NUTS satellite. NUTS has a custom designed backplane, and this means that it is not possible to buy complete modules and trivially make use of them.

1.3 Stakeholders

The stakeholders in the NUTS project are the people and organisations that are affected, affects or take an interest in the project. Both NAROM and Andøya Rocket Range (ARR) are stakeholders, because they both created the ANSAT program. The other stakeholders are the NUTS project leader, Roger Birkeland, as well as the students that are working on the project.

NAROM, the department of electronics and telecommunications as well as the department of engineering cybernetics finance the project.

1.4 Resources

This semester there are 14 students working on the NUTS project [19]. This is the first semester where software is considered as a distinct entity, and at this point there is a single person working on this.

1.5 Scope

The abstraction level of the topics in this report are necessarily high. The main goal is to identify the software requirements for the NUTS space segment software. This has a higher priority than producing a working implementation at this point.

This report focuses on the OBC, the On-Board Computer — but in the context of the whole system.

1.6 Previous work (related to software)

Hardware design-wise, only the on-board computer module, the radio module and the backplane are more or less completed. There may be new revisions of each of the those two modules if any problems are found, but no significant hardware changes are expected.

The designer of the OBC-module developed a couple of drivers to demonstrate it, as well as demonstrate that the module is able to read sensor data from the NUTS backplane.

1.7 Project status

This semester there is ongoing work on the attitude determination and control system (ADCS), antenna design for both the ground station and the satellite, the external power supply system, the wireless bus and the payload module. The specific hardware configuration for the payload is yet to be determined.

Currently, only the radio module and the OBC module are fitted with $2.4GHz$ wireless transceivers [24]. Testing of the wireless bus is in the works this semester, and work on the camera payload module just started this semester.

There are also proceeding tasks of a more analytical nature: Mission analysis and design of uplink security.

Currently there is no software that facilitates inter-module communication. That will be a topic for discussion in this report.

1.8 NUTS status

The satellite will consist of multiple modules or sub-systems. The modules in NUTS are:

- **OBDH/OBC**: Does on-board data handling. The idea is that this module can provide computing power for at least the payload module, as well as being able to handle requests from the ground segment, if a request originating from the ground segment is destined for it.
- **EPS**: Provides power to the various systems in the satellite.
- **ADCS**: Stabilises the satellite and provides various means of determining the spatial orientation.
- **Radio/COMM**: Contains radio transceivers as well as a radio transmitter for a beacon signal. This module processes in- and outgoing requests from and to the ground segment during the mission.
- **Payload**: An infrared camera. Details of this module are yet to be determined.

The modules are realised as circuit boards that fit into slots on a backplane. The NUTS backplane can accommodate any module that adheres to the backplane specifications. The original backplane design has eight slots [14], and is responsible for distributing power to all of the modules.

1.9 Assignment

The task is to start work on creating specifications for the systems software on the satellite. This includes a general interface for communication between the satellite and the ground station.

Specific issues related to that are:

- What, if any, Commercial Off-The-Self (COTS) software shall be used.
- Is it possible to guarantee fairness over the internal data bus?

- Is it possible to use the same, or parts of the same, software on multiple modules?
- What kind of mechanism or means is used for communication between each subsystem, and between the ground and space segment?

1.10 Similar projects

With the assignment in mind, it is useful to compare this with other student satellite projects where the necessary information is disclosed. The level of detail for each project will vary with the amount of available information.

CubeSTAR

The CubeSTAR uses I^2C as the bus type for the internal data bus. It has two separate data buses, where the idea is that the second bus can be used either for redundancy or in case the ”main” bus has hung. The bus is used in multi-master mode, where both the communications and the OBC module may act as bus masters.

For fault detection and increased system availability, ping/echo-commands are used. The module will be reset if it fails to respond to a ping/status request. The system exploits the minimum Hamming distance between commands to provide error detection. Error prevention may also be possible, by finding the minimum distance from the received command to a set of possible commands.

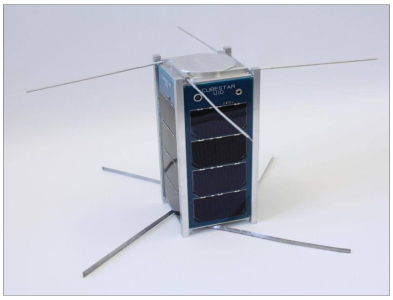


Figure 1.1: CubeSTAR
SOURCE: `http://cubestar.no/web_images/picture1.jpg`

The payload module also uses the main data bus. This means that the system has to rely on short bus transactions if erroneous resets are to be avoided.

No COTS operating system is used in any of the subsystems.

SwissCube

SwissCube was launched on September 23, 2009 [6][1]. As of today, the satellite continues to be operational![5]. Over time, the main data bus has become unstable for long frames [16]. At some point, the data bus also got stuck, but it was possible to reset the system by draining the batteries.

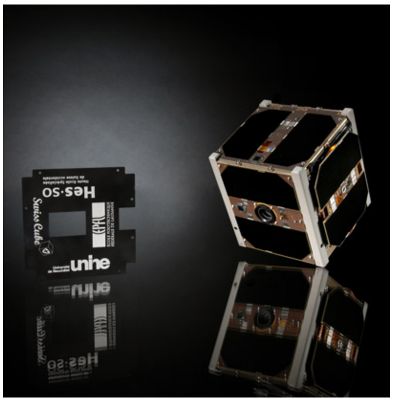


Figure 1.2: SwissCube
SOURCE: http://swisscube.epfl.ch/images/whatis.jpg

In the SwissCube project several bus types were evaluated [21] (p. 24). In the end I^2C was found to be the most suitable alternative, and this was the bus type that was used. The I^2C bus in SwissCube operates with a lower clocked ”standard mode”, as opposed to the higher clocked ”fast mode” in NUTS.

Several operating systems were evaluated [21] (p. 13), but no ”best” alternative was found. In the end, the SwissCube did not use a COTS operating system [18]. The evaluated operating systems are not ported to the main microcontrollers in any of the NUTS subsystems.

The subsystems in SwissCube are [22] (p. 37):

- COMM
- EPS
- ADCS
- Beacon
- CDMS

The NUTS OBC is most similar to the CDMS in the SwissCube. However, while NUTS has a separate backplane, the CDMS board in SwissCube serves as a ”connector hub”. A separate microcontroller is used on the CDMS to interface with the I^2C bus, because the main microcontroller (an Atmel ARM AT91M55800A) in the CDMS has no hardware support for I^2C [12] (p. 8). The same microcontroller is used as an I^2C bridge in each subsystem. For persistent storage, a combination of EEPROM and NAND flash memory was used [13] (p. 11).

The main tasks of the CDMS are [13] (p. 11):

- Execute scheduled tasks.
- Serve as a data storage for payload and subsystem statuses. The CDMS polls for housekeeping data at regular intervals.
- Execute control algorithms for the ADCS.

Ther NUTS OBC has similar responsibilities. But one of the most significant differences are that the main processor on the CDMS does processing for the ADCS system. On NUTS the ADCS system will have to be able to do processing itself and operate autonomously, but be able to respond to requests from the other modules on the NUTS backplane.

While NUTS is not like SwissCube, there it shares enough similarities that a lot of the same software design decisions taken for the SwissCube, also applies to NUTS.

AUSAT

The AUSAT uses a PC/104 compliant frame and circuit boards. The design is based on the concept of a centralised microcontroller that manages each subsystem [10]. The design philosophy is completely different from that of the NUTS project. AUSAT is a CubeSat that serves a very specific purpose. While the PC/104 standard gives physical modularity, the AUSAT is not a modular system in the same sense as NUTS.

The EPS module was COTS hardware. All the other electronics was grouped into PC/104 compliant circuit boards. In the end, the physical boards were[10]:

- EPS
- Communication
- CDHS
- ADCS

The Command and Data Handling system is the closest analog to the OBC in NUTS. Unlike NUTS, the central microcontroller on the CDHS has full knowledge of each subsystem to the extent that the hardware permits. They opted not to use an operating system. Instead the central ”controller program” is based around the concept of a state machine that is able to perform a single task at the time.

The bottom line, is that the design of AUSAT makes it hard to transfer the same software concepts to the NUTS project: The hardware design is different and the design philosophy for the satellite is also completely orthogonal.

Planning 2

Introduction

This chapter presents the NUTS project as it is this semester, the team composition and what kind of methods and aids the participants use during project work. Lastly the actual timeline of the assignment described in this report, is presented.

2.1 Team composition

This semester, the NUTS project has a team of 14 students. Their background is either from telematics, cybernetics, electronics, computer science or mechanics. This excludes technical staff.

The following departments are involved in the NUTS project:

Department of Engineering Design and Materials (IPM)

Department of Physics (PHYS)

Department of Electronics and Telecommunications (IET)

Department of Engineering Cybernetics (ITK)

Department of Computer and Information Science (IDI)

Department of Telematics (ITEM)

2.2 Method of work

The team has weekly meetings. Having regular and relative short meetings, makes it easier to know what the other team members are doing.

This also makes it easier to know who to ask if any problems or questions arise. Extra meetings were arranged when necessary.

However the members do not work closely together. The main objective for the student is to produce a report resulting from practical work and background studies.

2.3 Tools

Several general software tools is used to aid in project work.

- **Subversion** Centralised VCS. This is used mostly for code, but some team members use it for documentation as well.
- **Git** Distributed VCS. Some team members prefer to use this.
- **Yammer** Social network service for projects. The team uses this to exchange information and progress updates.
- **Google Docs** This is used to quickly share documentation.

Other software is used for specific tasks. Details of this can be found in the appropriate chapters. The NUTS project also has a Subversion repository. This is the official repository, and all team members who wants to share code with others have have to use this. But when working individually, a feasible solution is to merge data from an external repository into the official one.

2.4 Project phases

There were no distinct project phases. This is a natural consequence of not knowing the specific assignment until the middle of the semester, some uncertainties regarding the project itself and that there was a single person working on the software for the space segment.

The project duration was from August 29th to December 21st. In that time span, the work on the assignment can be broken down into the following stages:

Phase	Activites	Week no.
Project start	Briefing and administrative tasks	35
Workshops	Meetings with electronics group Setting up development environment NUTS kickoff	36–37
Planning	Requirements Commands draft Background study of bus types	38
Prestudy	Communication libraries Getting to know libcsp OS alternatives Comparisons with other CubeSats NUTS backplane & OBC	39–42
Implementation	Porting libcsp Analysing OBC code Learning FreeRTOS Deploying FreeRTOS to OBC	41–49
Report writing	Work on the report	50–51

Note that the table depicts the actual time spent. There was no way to plan ahead the prestudy and implementation phase because the nature and extent of them were completely unkown.

System description and environment 3

Introduction

This chapter gives a detailed insight into the parts of the NUTS project that are relevant to this report, or to future works related to software.

3.1 System overview

Figure 3.1: System topology

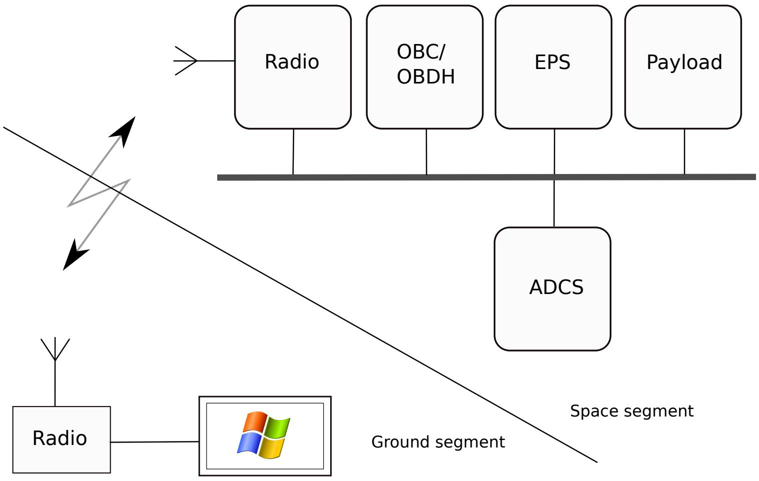


Figure 3.1 shows an overview over the whole system. The funtionality of the NUTS satellite is split into physical modules that are connected to a common data bus. In addition both the radio and OBC-modules

have radio transceivers meant for local data transfer. This functionality is not tested as of yet, but using this feature as a wireless bus may be used as a backup bus, used for bulk data transfers or used in the event that the main data bus fails.

3.2 NUTS subsystems

What follows is a brief description of each subsystem in NUTS.

ADCS

The satellite has to be able to change its orientation and reduce or control its angular velocity. The ADCS system is responsible for doing this. All processing is completed within the ADCS module. However the ADCS system shall be able to respond to high-level requests from either the OBC or the radio.

OBC

The OBC is responsible for running the flight management software aboard NUTS. Its primary task is to process data from the payload module, store system status information, execute scheduled commands and retrieve stored data upon request from the ground segment.

The OBC is one of two masters of the back plane. In this context, it means that the OBC has access to extra pins on the backplane. These pins makes it possible for the OBC to perform a partial or a full system reset

Radio

The radio module has two RF transceivers plus a radio transmitter for a beacon signal. The beacon signal is controlled by a dedicated microcontroller.

Frame encapsulation and decapsulation for radio transmission or receival is done on the radio module. The packet framing has not been decided yet, but it is common to use AX.25 framing in CubeSats.

A design goal, is that the radio module shall be able to operate as a feature reduced OBC in the event that the OBC module fails. However, the radio module lacks some hardware features that are present on the OBC module, so it does not have access to the same amount of program and data memory. In addition, no OTP memory is present on the radio module.

EPS

The EPS module handles the task of providing regulated power to the rest of the system and also charges the batteries from solar cells. This modules provides vital power telemetry data to the rest of the system.

Payload

The payload consists of an IR camera. This module is currently in the design phase, so the exact specifications are not clear. But its main task is to respond to commands from the OBC module, and send image data when requested to do so.

3.3 Hardware specifications

The project is still evolving, and not all hardware specification decisions are final. What follows is a brief discussion of the hardware features that are unlikely to undergo substantial changes in the future. The EPS and payload modules are omitted because exact hardware specifications are not known at this point.

Backplane

The backplane has eight connectors. The slot configuration is as follows [14]:

Module 4: OBC

Module 5: EPS

Module 8: Radio

Module connector 1–3 and 6–7 are generic connectors. They can be used for other subsystems or the payload module. The module connectors used by the OBC and the Radio have special access to the backplane: They connectors have extra pins for resetting the system, power on or off a module, reset a module and program a module. Extra address pins makes it possible to target a specific module. The signals for system reset are different for module connector five and eight. This is to prevent a module from resetting itself.

This means that the OBC and the radio are masters of the backplane. They are able to isolate other modules from the backplane, including the other backplane master.

The backplane is also responsible for distributing power throughout the system. To do this, the backplane provides duplicate $3.3V$ and $5V$ rails with over-current protection to each module.

For CubeSats, it is not unusual to follow the form factor and data bus-defining standard, PC/104 [11]. The NUTS backplane does not follow this, but has a different form factor and bus type. The bus type is I^2C, operating in "fast mode". This gives a theoretical throughput of $400kbps$ [20].

Radio

- UHF and VHF transceivers
- Main MCU: Atmel AVR32UC3A3256
- Beacon MCU: Atmel ATmega328P
- Nordic Semiconductor nRF24LE1D 2.4GHz RF transceiver chip (SoC)

The main MCU is connected to the main transceivers used for uplink and downlink communication with the ground station The beacon MCU is connected to the beacon transmitter. Testing the radio module is beyond the scope of this project.

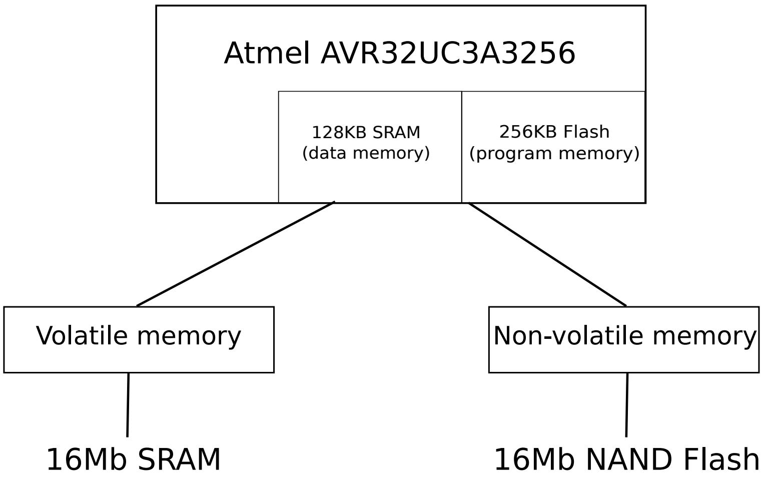
OBC

- Atmel AVR32UC3A3256
- Nordic Semiconductor nRF24LE1D 2.4GHz RF transceiver chip (SoC) connected to main CPU via a SPI interface. The transceiver chip has two external clock sources: A 32KHz crystal, used for protocol syncronisation, and a 16MHz crystal for inernal clocks[24].
- Micron MT29F16G08DAAWP 16Gb NAND flash
- Atmel AT27LV040A 4Mb EPROM, OTP

- Integrated Silicon Solutions IS61WV102416BLL 16Mb SRAM

The external memories are accessed through the external bus interface (EBI) of the AVR32UC3A3256 microcontroller. With this configuration, the static memory controller and NAND flash controller is used to interface with them. The memory topology is as depicted in 3.3.

Figure 3.2: OBC memory topology



Aftern configuration, we have the memory map shown in 3.3

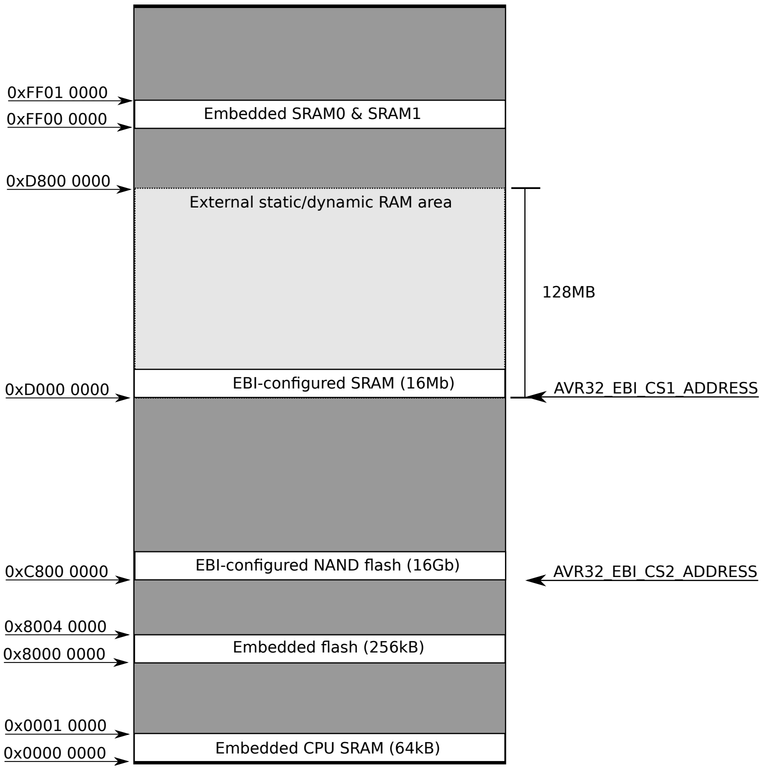
From the memory map, we see that the amount of integrated memory is limited. We will see that the current OBC program is very close to reaching the limits of the integrated program memory.

Both the radio and the OBC have a debug interface for on-ground testing and diagnostics, as seen in figure 3.4. Neither the radio or the OBC has a USART-to-USB-bridge. This means that the software on these modules has to implement a USB stack. The reasons for *not* having this, is because a simpler module design was desired. This is a tradeoff between software and hardware complexity. And the former loses out on this.

AVR32UC hardware features

The AVR32UC3 is based on the AVR32A micro-architecture [8]. Unlike the AVR32B architecture, it lacks certain features like register shadowing in interrupts contexts and dedicated registers for storing return values and return addresses.

Figure 3.3: OBC memory map



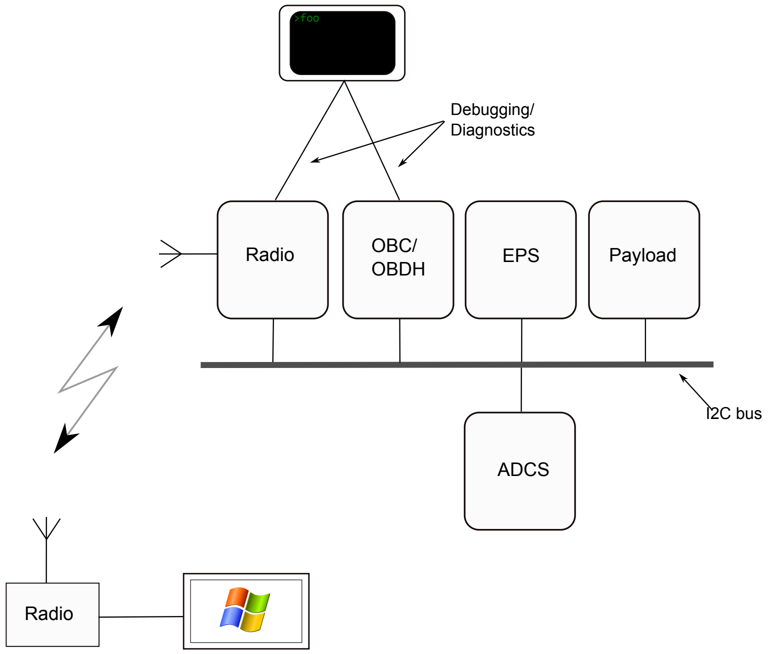
While the ISA is not binary compatible with the 8-bit AVRs, it still shares many similarities: The instruction set is that of a load-store architecture, with dedicated instructions for loading and storing data to memory. The instructions are also variable length, with either a size of 16-bit or 32-bit.

While the architecture lacks backwards compatibility with the AVR8-architecture, it is compatible with previous revisions of the same architecture. In this project, source compatibility is not a concern, since we have access to the source code to all programs that are to be executed on board the satellite. However, when using frameworks, the intended architecture may still impact compatibility. For instance, if a library is designed with a 32-bit architecture in mind, it may not be trivial to compile it into a binary for an 8-bit architecture.

Other notable hardware features are

- 15 general-purpose registers
- 32-bit stack pointer, program counter, link register

Figure 3.4: System overview with debug terminals. Both the radio and the OBC are have a USB interface connector



- Various DSP-instructions
- An MPU for protecting memory address ranges
- A flat memory model, but with separate program and data memory
- Dedicated flash controller for interfacing with external memories
- Privileged and unprivileged modes

When in unprivileged mode, certain rules apply:

- All memory accesses are checked for violations
- The high half word of the status register is not available. This part of the register makes it possible to enable and disable interrupts, mask interrupts etc.
- System registers are placed outside the virtual address space, and has to be accessed using the privileged instructions mfsr, mtsr. Examples of system registers are the registers that controls the MPU,

configuration registers for the processor, and the EVBA register. The latter stores the base address of the exceptions routines table.

The AVR32UC3 lacks a MMU, but the MPU is able to protect a limited number of address ranges. The AVR32UC3 MPU can split the address space into eight regions, where each reagion can be further divided into 16 subregions. This means that the main microcontrollers og the radio and OBC has some hardware features that can be leveraged to prevent runaway programs from causing trouble. Unfortunately, as described later, there is no support for this in the AVR32 port of FreeRTOS.

3.4 Software environment

Here we describe the existing software for both the ground and the space segment.

Space segment

There is some code for testing the OBC features, as part of designing the module. The program demonstrated:

- Controlling the backplane using I^2C
- Writing and reading to/from NAND flash
- Writing and reading to/from SRAM
- USB communication using Atmel Software Framework Drivers
- Testing of JTAG-programming by bit-banging the JTAG-pins on the module connector.

This program was only meant for testing, and significant changes are needed if the code is to be used in a different context. However it is useful as an example, and is a good basis for writing drivers for the implicated hardware.

Ground segment

- **Operating system**: Windows XP
- Hamradio Deluxe
- MixW

- WXTrack

In addition a custom .NET4-program for communicating with the OBC has been written. The program uses Windows Presentation Foundation, so it will not run in Mono — a portable implementation of Microsoft .NET framework.

Summary

Currently the software on the ground station dictates that Windows is used. This means that any third party library that does not work in Windows, has to be either discarded or ported.

Requirements

4

Introduction

This chapter presents the system- and software requirements, with a focus on the space segment. We then discuss desirable architectural attributes and tactics that will help us achieve them.

4.1 System requirements

The system requirements describes the main features of the system, without being to specific. They are the basis for requirements that are more specific. In NUTS, we have the following system requirements:

Table 4.1: NUTS system requirements

ID	Description
S-1	Satellite must process commands from the ground station
S-2	Satellite must send a beacon signal
S-3	Satellite must be able to send housekeeping data
S-4	Satellite must be able to send payload data
S-5	Ground station must be able to send commands to satellite
S-6	Both ground station and satellite must be able to detect data corruption or complete loss of transmitted data

4.2 Functional requirements

The system requirements in table 4.1 are further refined into more specific, functional requirements as shown in 4.2:

Table 4.2: NUTS functional requirements

ID	Description	Priority (H/M/L)
F-1	The satellite beacon must continuously transmit beacon signal	H
F-2	It must be possible to change the beacon signal pattern after launch	M/H
F-3	The satellite must execute a one-time initialisation sequence on first boot up	H
F-4	The satellite must accept a ”detumble” command. The command must be accepted by the ADCS system	M/H
F-5	The ADCS system must accept other commands from both the ground segment, the OBC or the radio	M/H
F-6	The satellite must accept and store commands sent from the ground segment	M/G
F-7	The satellite must be able to create and store commands programmatically	M
F-8	The satellite must execute housekeeping tasks periodically	M
F-9	The satellite must store the results of running the housekeeping tasks	H
F-10	The satellite must be able to send the result of housekeeping task runs, upon request from the ground station	H
F-11	The satellite must initiate a single housekeeping task run upon request from the ground station	H
F-12	The radio and OBC module must be able to run an arbitrary program	L/M

Table 4.2: (continued...)

ID	Description	Priority (H/M/L)
F-13	The satellite must transmit payload data when ground station requests it to do so	L
F-14	It must be possible to initiate a full or partial satellite system reset from the ground station	M/H
F-15	It must be able to set the current time in the satellite, from the ground station	L/M

Comments to table 4.2

F-1 The radio module has a separate microcontroller that controls the beacon signal.
F-3 The initialisation sequence includes unfolding the atennas, after som period of time. This program must *not* be executed more than once.
F-4 The power requirements for a detumbling operation are high. This action must only be performed when the system is requested to.
F-5 All necessary control algorithms runs within the ADCS system. However this item indicates that the ADCS system must accept high level-commands like ”point towards earth”.
F-8 Housekeeping tasks include logging telemetry data like current draw and actual voltage on the $3.3V$ and $5V$ rails for each module, andvalues from temperature sensors. **F-10** This is a more specific requirement than F-6. **F-12** This is a function that is useful to a developer, and the flight software must make it easy to do this.

The requirements are given priorities ranging from low to high. Intuitively, it would seem better to start implementing the higher priority items first. However, often lower prioritised items are prerequisites for those of a higher priority, or they take more time to implement.

4.3 Non-functional requirements

Non-functional requirements describe attributes or properties of the system. This includes business requirements, system requirements, quality requirements and other requirements. We omit listing business requirements, because we have been unable to identify them yet. [1]

Quality requirements

The quality requirements described the desirable properties of the system. We have identified four main quality attributes: Modifiability, testability, security and reliability.

Modifiability

ID	Description	Priority (H/M/L)
NF-M1	It shall be possible to compile and run the same programs on the OBC and the radio, unless they are dependent upon specific hardware drivers.	H
NF-M2	It must be possible to change the output device for diagnostics prinout without affecting more than a single file in the source code.	H

Testability

ID	Description	Priority (H/M/L)
NF-T1	It shall be possible to send a command via the debug interface on each module, without changing the command format	H
NF-T2	Commands sent to the debug interface shall be processed the same way as commands received via the data bus	H
NF-T3	The OBC and radio shall be able to store a program execution trace to non-volatile memory	L

[1] The project schedule is not known yet. Neither is the resources allocated to software.

Table 4.4: (continued...)

ID	Description	Priority (H/M/L)
NF-T4	It shall be possible to retrieve contents from non-volatile memory on the OBC, from the debug interface	L

Reliability

ID	Description	Priority (H/M/L)
NF-R1	Only uncorrupted commands shall be executed	H
NF-R2	A failing program must not affect the core functionality of the system	H
NF-R3	Execution of less-important tasks shall not affect the timelinss of higher-prioritised tasks	M
NF-R4	A frozen system program shall not render the satellite useless	H

Security

ID	Description	Priority (H/M/L)
NF-S1	Only commands sent from our ground station shall be executed	H
NF-S2	Data transmitted from the satellite to the ground station must not be encrypted	H

System requirements

ID	Description	Priority (H/M/L)
NF-O1	The ground station software must run on Windows Vista, or a more recent version of Windows	H
NF-O2	The operating system on the radio and OBC must both work on an Atmel AVR32UC3A3256 microcontroller	H

4.4 Conclusion

We have seen the various requirements for the NUTS project. The requirements focus on the satellite, but the ground segment is included when believed to be relevant. The requirements are work in progress, and the priorities are bound to change in the future as the project evolves.

Software architecture 5

Introduction

This chapter introduces the software architecture.

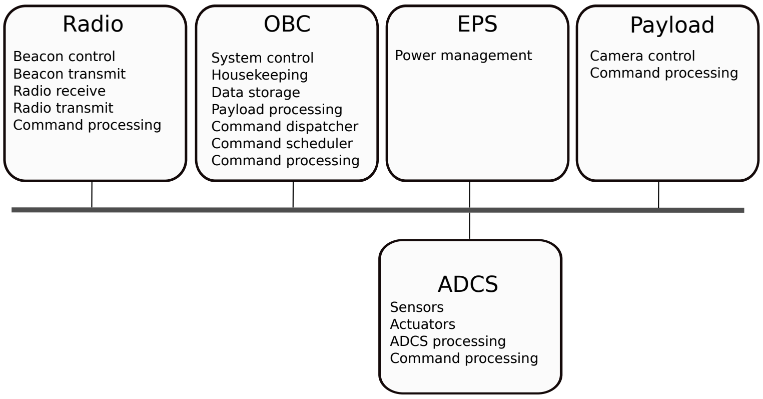
5.1 Software

The development environment of choice is AVR Studio 5.0 for the radio- and OBC-software. This development tool uses the AVR GNU Toolchain, and all libraries have to work with this.

5.2 Functionality mapping

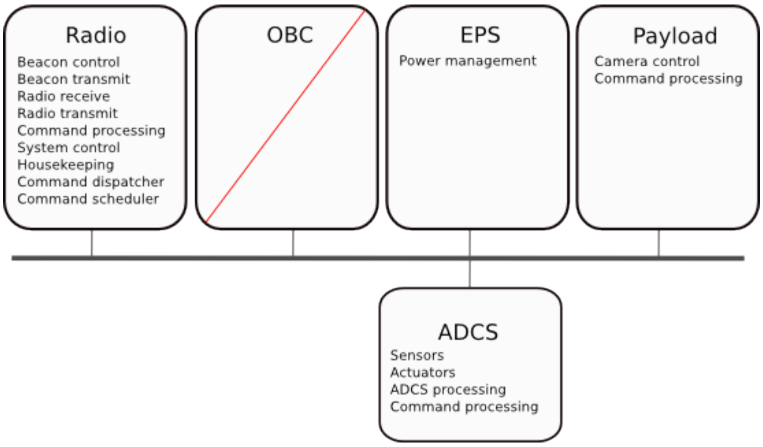
The functionality identified in chapter 4 is mapped into modules in the system. This can be seen in figure 5.1.

Figure 5.1: Nominal mode, module functionality



The radio is a single point of failure. However, we have two backplane masters. If the OBC fails, the system will have to operate in ”reduced” mode as seen in figure 5.2. The radio doesn’t have the same hardware features as the OBC, so it will not be able to substitute the OBC completely.

Figure 5.2: Reduced mode, module functionality



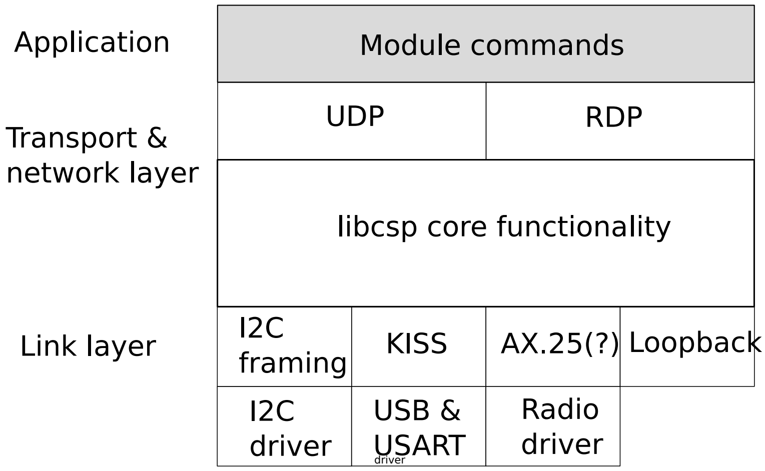
5.3 Communications architecture

When using libcsp, we end up with a layered architecture, as seen in figure 5.3

The libcsp core functionality contains basic functions in the CSP library. These functions include – but are not limited to – buffer management, service handlers, socket management, port management and packet routing. Above this layer, we have the two protocols which would belong in the transport layer in the OSI model. UDP is implemented in CSP, and at least parts of RDP is implemented. On top of the transport protocols, we find the module commands layer. It is in this layer we will do module-specific processing and it is here we have the business logic.

Below the libcsp core functionality layer, we find medium specific code. CSP is not dependent on what is used to transport data, but it needs code to interface with it. On the bottom layer, we find the hardware drivers. CSP has to interface with this layer to be able to push data around. This means that in order to transport CSP packets over the internal data bus in NUTS, we will have to implement an I^2C interface

Figure 5.3: Architecture



in CSP, and talk to an hardware driver. This driver must work with FreeRTOS.

5.4 Quality attributes

At this point, it is hard to verify that the architecture (or the implementation) achieves the desired quality attributes. To test this, one could make quality attribute scenarios. But it must be possible to test the system against these attribute scenarios for them to have any value.

But we can draw some general conclusions for each attribute.

Security

CSP supports both packet encryption and authentication using HMAC. Both methods are alone sufficient for ensuring that no malicious third party sends command to the satellite, without the radio or the subsystem knowing that the command is to be discarded. Of course, this is assuming the implementation does not contain any flaws.

Modifiability

Using an operating system like FreeRTOS, has the same effect as implementing a HAL — an Hardware Abstraction Layer. This makes typical ”user”-programs less specific to the hardware, because the operating system provides methods for synchronisation, intertask-communication etc.

Reliability

At this point, the architecture itself does not imply extra reliability. CSP does checksumming of packets, but tasks running within FreeRTOS are not restricted to what kind of memory addresses they are allowed to write to. Currently there is no FreeRTOS port for AVRUC3, that supports the MPU.

Testability

The code, as it is now, is not very testable.

5.5 Fairness on the data bus

I^2C is a multimaster-bus, and the module that sends the "lowest" byte is always going to win arbitration. This means that we will have to impose extra restrictions if it should be possible to guarantee fairness or any particular quality of service. In most CubeSat projects this has not been a problem in practice, because on could rely on loose timing constraints and sheer bandwidth. NUTS has strict no timing constraints, but the I^2C bus is not made for shuffling large amounts of data around.

We do not yet know how much data the payload mdule is going to transfer over the bus, and this makes it a theoretical issue. There are no other module in the satellite that has to shuffle around potentially huge amounts of data.

To reduce the noise on the bus, it has been decided that only the radio and the OBC are allowed to assume the roles as masters on the bus. This means that they are the only modules that are allowed to initiate transactions on the internal data bus. No other module is allowed to start a bus transaction on its own initiative. The result is that the system has to rely on either periodically polling each module for status information, or "know" that we have some data to receive.

It would be possible to impose some kind of bus arbitration scheme. This will not be discussed in this report, and it would be better to put a definitive limit to the size of a single picture before deciding whether this is going to be necessary in practice.

5.6 Conclusion

We have seen how using COTS software can help achieve some qualities. But there is still a lot to be desired. Methods for achieving reliability have to be researched, and a policy for writing testable code has to be developed.

Testing 6

Introduction

This chapter describes how the implementation was tested. We first describe how cubesat space protocol was tested on a development system representative for the ground station. We then describe how basic functionality in FreeRTOS was tested on the OBC. Last, we describe how cubesat space protocol was compiled and linked into the OBC firmware this firmware was tested and verified not to overflow or exceed the amount of available embedded memory on the main MCU.

6.1 Scope

No test plan has been developed, because it is too early in the software development. The typical documentation for testing software will follow the IEEE Standard 829-2008 [15], and adopted to the project in question. This will have to be done in the future, when it is possible to test the requirements described in chapter 4. Instead we choose to describe how the software evolved, and how it was tested during development to make sure that the desired features were present and working. We only describe the parts that are relevant to the implementation described in this report.

6.2 Testing cubesat space protocol

This section describes how cubesat space protocol was tested in a envitronment similar to that of the ground station. Here, the test system meet the minimum system requirements as described in the system requirements section 4.3.

Software

- **Operating system**: Windows 7
- **IDE/text editor**: vim 7.3
- **Compiler/Toolchain**: MinGW with GCC 4.5.2
- **Other**: cubesat space protocol

CSP in loopback mode

When running CSP in loopback mode, the server and client are two threads within the same process. To verify that the windows-port was working, a simple program was written. All it does is to use various services integrated in CSP as well as send a packet with arbitrary payload. The former tests that the ″service handler″ within CSP is working, and the latter tests that it works for an arbitrary packet.

The code for the test program can be found in the appendix, in section A.3 or in an archive file enclosed with this report.

```
Dan Erik@Flux /C/Repos/Studsat/Software/ground segment/GSE/CSP_Loopback/branches/_danerik
$ make
g++ -g -pedantic -D__STDC_LIMIT_MACROS -Iinclude -Ilib/libcsp/include   -c -o src/main.o s
rc/main.cpp
In file included from include/server.h:5:0,
                 from src/main.cpp:8:
lib/libcsp/include/csp/csp.h:635:34: warning: anonymous variadic macros were introduced in
 C99
lib/libcsp/include/csp/csp.h:141:19: warning: comma at end of enumerator list
lib/libcsp/include/csp/csp.h:203:2: warning: ISO C++ prohibits anonymous structs
lib/libcsp/include/csp/csp.h:263:17: warning: ISO C++ forbids zero-size array 'data'
lib/libcsp/include/csp/csp.h:264:20: warning: ISO C++ forbids zero-size array 'data16'
lib/libcsp/include/csp/csp.h:265:20: warning: ISO C++ forbids zero-size array 'data32'
lib/libcsp/include/csp/csp.h:620:16: warning: comma at end of enumerator list
mv src/main.o obj/
g++ -Llib/libcsp/lib   obj/main.o obj/event.o obj/mutex.o obj/thread.o obj/screenlock.o ob
j/client.o obj/server.o  -lcsp -lstdc++  -o obj/main
mv obj/main bin/main

Dan Erik@Flux /C/Repos/Studsat/Software/ground segment/GSE/CSP_Loopback/branches/_danerik
$ bin/main.exe
Level 0: value 1
Level 1: value 0
Level 2: value 0
Level 3: value 1
Level 4: value 1
Level 5: value 1
Level 6: value 1
```

```
1 csp_port.c:90 Binding socket 00542F78 to port 16
?[0mServer listening
Client trying to connect to server...
?[0;33m1 csp_buffer.c:115 BUFFER: Using element 1 at 00543108
?[0mPS node 1: ?[0;32m1 csp_io.c:188 Sending packet size 1 from 1 to 1 port 2 via interfac
e LOOP
?[0m?[0;32m1 csp_route.c:259 Router input: P 0x02, S 0x01, D 0x01, Dp 0x02, Sp 0x15, F 0x0
0
?[0m?[0;32m1 csp_io.c:188 Sending packet size 37 from 1 to 1 port 21 via interface LOOP
?[0m?[0;33m1 csp_buffer.c:147 BUFFER: Free element 1
?[0;32m1 csp_route.c:259 Router input: P 0x02, S 0x01, D 0x01, Dp 0x15, Sp 0x02, F 0x00
?[0m?[0mPS Length 37
Tasklist in not available on windows
?[0;33m1 csp_buffer.c:147 BUFFER: Free element 1
?[0m?[0;33m1 csp_buffer.c:115 BUFFER: Using element 2 at 00543248
?[0m?[0;32m1 csp_io.c:188 Sending packet size 0 from 1 to 1 port 3 via interface LOOP
?[0m?[0;32m1 csp_route.c:259 Router input: P 0x02, S 0x01, D 0x01, Dp 0x03, Sp 0x16, F 0x0
0
?[0m?[0;32m1 csp_io.c:188 Sending packet size 4 from 1 to 1 port 22 via interface LOOP
?[0m?[0;33m1 csp_buffer.c:147 BUFFER: Free element 2
?[0m?[0;32m1 csp_route.c:259 Router input: P 0x02, S 0x01, D 0x01, Dp 0x16, Sp 0x03, F 0x0
0
?[0m?[0;33m1 csp_buffer.c:147 BUFFER: Free element 2
?[0mFree Memory at node 1 is 1834471424 bytes
?[0;33m1 csp_buffer.c:115 BUFFER: Using element 3 at 00543388
?[0m?[0;32m1 csp_io.c:188 Sending packet size 0 from 1 to 1 port 5 via interface LOOP
?[0m?[0;32m1 csp_route.c:259 Router input: P 0x02, S 0x01, D 0x01, Dp 0x05, Sp 0x17, F 0x0
0
?[0m?[0;32m1 csp_io.c:188 Sending packet size 4 from 1 to 1 port 23 via interface LOOP
?[0m?[0;33m1 csp_buffer.c:147 BUFFER: Free element 4
?[0mUptime of node 1 is 155234 s
?[0;33m1 csp_buffer.c:115 BUFFER: Using element 5 at 00543608
?[0m?[0;32m1 csp_io.c:188 Sending packet size 0 from 1 to 1 port 8 via interface LOOP
?[0m?[0;32m1 csp_route.c:259 Router input: P 0x02, S 0x01, D 0x01, Dp 0x08, Sp 0x19, F 0x0
0
?[0mMessage from client: Hello world!
?[0;33m1 csp_buffer.c:147 BUFFER: Free element 5
?[0m
```

The tested version of CSP use ANSI escape sequences, and they are not interpreted by the terminal.

CSP in USART/KISS mode

In this mode, packet data is encapsulated in KISS frames and sent via an USART interface. The test program runs both a server and a client thread, as when testing CSP in loopback mode. The difference is that the packet is sent to a different interface. An Atmel XMEGA A-1 Xplained was used as the "OBC" in this example. The program on the A-1 does nothing but "bounce" any received characters back to the sender. This tests that CSP is able to use the USART interface, and that KISS encapsulation and decapsulation works.

The test output is:

```
Dan Erik@Flux /C/Repos
```

```
$ gcc -Wall -pedantic -std=gnu99 -Lcsp_deploy/lib -Icsp_deploy/include kiss.c -lcsp -o ki
ss
In file included from kiss.c:5:0:
csp_deploy/include/csp/csp.h:143:3: warning: ISO C doesn't support unnamed structs/unions
csp_deploy/include/csp/csp.h:196:11: warning: ISO C forbids zero-size array 'data'
csp_deploy/include/csp/csp.h:197:12: warning: ISO C forbids zero-size array 'data16'
csp_deploy/include/csp/csp.h:198:12: warning: ISO C forbids zero-size array 'data32'
csp_deploy/include/csp/csp.h:199:3: warning: ISO C doesn't support unnamed structs/unions
Dan Erik@Flux /C/Repos
$ ./kiss.exe
Level 4: value 1
Level 2: value 1
[01] usart_windows.c:93 Port: COM4, Baudrate: 9600, Data bits: 8, Stop bits: 0, Parity: None
[00 0040F074] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[01 0040F098] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[02 0040F0BC] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[03 0040F0E0] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[04 0040F104] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[05 0040F128] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[06 0040F14C] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[07 0040F170] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[08 0040F194] S:0, 0 -> 0, 0 -> 0, sock: 00000000
[09 0040F1B8] S:0, 0 -> 0, 0 -> 0, sock: 00000000
Node  Interface  Address
   1  KISS       1
   *  LOOP       255
LOOP    tx: 00000 rx: 00000 txe: 00000 rxe: 00000
                drop: 00000 autherr: 00000 frame: 00000
                txb: 0 (0.0B) rxb: 0 (0.0B)

KISS    tx: 00000 rx: 00000 txe: 00000 rxe: 00000
                drop: 00000 autherr: 00000 frame: 00000
                txb: 0 (0.0B) rxb: 0 (0.0B)

[01] csp_port.c:90 Binding socket 0040F098 to port 32
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 1 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x01, Sp 0x25, F 0x00
[01] csp_service_handler.c:94 SERVICE: Ping received
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 37 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x25, Sp 0x01, F 0x00
Ping with payload of 50 bytes, took 1092 ms
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 1 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x01, Sp 0x26, F 0x00
[01] csp_service_handler.c:94 SERVICE: Ping received
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 38 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x26, Sp 0x01, F 0x00
Ping with payload of 100 bytes, took 998 ms
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 1 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x01, Sp 0x27, F 0x00
[01] csp_service_handler.c:94 SERVICE: Ping received
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 39 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x27, Sp 0x01, F 0x00
Ping with payload of 150 bytes, took 1311 ms
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 1 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x01, Sp 0x28, F 0x00
[01] csp_service_handler.c:94 SERVICE: Ping received
[01] csp_io.c:234 Sending packet size 100 from 1 to 1 port 40 via interface KISS
[01] csp_route.c:254 Router input: P 0x02, S 0x01, D 0x01, Dp 0x28, Sp 0x01, F 0x00
Ping with payload of 200 bytes, took 999 ms
```

The output is different because this uses a more recent version of CSP, where ANSI escape sequences are not printed when the library is compiled for Windows.

The code for the A1 and relevant code snippets for this test can also be found in the appendix, in sections A.5 and A.4.

6.3 Testing the OBC

There were some technical difficulties when testing CSP on the OBC. To test CSP on the OBC, the goal was to make it work in loopback mode, while printing the result to a host computer over USB.

To do this, libcsp was compiled for FreeRTOS and our project with the command from within the libcsp root folder:

```
JOBS=1 CFLAGS="-mpart=uc3a3256␣-DBOARD=USER_BOARD␣-
    DFREERTOS_USED" ./waf distclean configure --toolchain=avr32-
     --prefix=../obc_deploy --with-os=freertos --with-freertos
    =../obc/OBC_System_SW/src/asf/thirdparty/freertos/source/ --
    includes=../obc/OBC_System_SW/src,../obc/OBC_System_SW,../
    obc/OBC_System_SW/src/config,../obc/OBC_System_SW/src/asf/
    thirdparty/freertos/source/portable/gcc/avr32_uc3,../obc/
    OBC_System_SW/src/asf/thirdparty/freertos/source,..obc/asf/
    thirdparty/newlib_addons/libcs/include,../obc/OBC_System_SW/
    src/asf/common/boards,../obc/OBC_System_SW/src/asf/common/
    boards/user_board,../obc/OBC_System_SW/src/asf/common/
    services/fifo,../obc/OBC_System_SW/src/asf/common/utils/,../
    obc/OBC_System_SW/src/asf/commonutils/interrupt,../obc/
    OBC_System_SW/src/asf/avr32/utils,../obc/OBC_System_SW/src/
    asf/avr32/utils/preprocessor,../obc/OBC_System_SW/src/asf/
    avr32/drivers/pm,../obc/OBC_System_SW/src/asf/avr32/drivers/
    intc build install
```

While both the server and the client were able to run, the server program quickly ran out of buffers. The test result was that both the server and the client were running and printing their output to the host computer. However, there was a bug that prevented it from functioning correctly.

The files under `../csp_deploy/{include,lib}` were copied into the AVR Studio project for the OBC. The linker was configured to link with the library, and the CSP include folder was added to the include paths in the project properties.

To verify that the code would work with the internal OBC memory, the `avr32-nm` was used to generate a list of symbols with adresses. Extracts from the output can be seen below:

```
avr32-nm -s -n obc_firmware.elf
         w _Jv_RegisterClasses
         w cpu_reset
         w cpu_set_reset_cause
         w csp_route_input_hook
00000008 d __CTOR_LIST__
00000008 B _data
0000000c d __CTOR_END__
00000010 d __DTOR_LIST__
00000014 D __DTOR_END__
00000018 d __JCR_END__
00000018 d __JCR_LIST__
0000001c D _GLOBAL_OFFSET_TABLE_
(...)
00000b3c B stdio_usart_base
00000b40 B prvScreenMutex
00000b44 B b_rx_new
00000b45 B b_tx_new
00000b48 B line_coding
00000b50 B my_address
00000b54 B interfaces
00000b58 B routes
00000c60 B routes_lock
00000c64 B handle_router
00000c68 B errno
00000c70 A __heap_start__
00000c70 A _end
00001000 A __stack_size__
00001000 A _stack_size
0000f000 A __heap_end__
0000f000 B _stack
00010000 B _estack
80000000 T _trampoline
80002000 t program_start
80002004 T _init
80002020 T _stext
(...)
ffffffff A __heap_size__
ffffffff A __max_heap_size__
```

We see that the heap begins at the end of the `.data` segment, and ends at the beginning of the stack. The heap size is `0xF000-0x0C70`, which is about 56kB. The same method can be used to find the size of the stack. The `.text` segment begins at `0x8000 0000`. The standard linker script puts a special "trampoline" routine. Normally, this routine does nothing but transfer control to the user code. We also see a special symbol called `_init`. This symbol denotes a special initialisation function. This function is defined by FreeRTOS, and it does early hardware initialisation. This includes configuring the interrupt controller, setting the stack register to point at _stack and configuring the exception vectors.

A linker script tells the linker how to lay out the final, linked binary. By looking at the OBC memory map in table 3.3, we see that it is very important that data and code are put on the correct load memory address.

If external memories are to be used, the linker script has to be modified to put the heap in external RAM, and the initialisation code has to initialize the memory controller *before* anything is allocated from the heap!

6.4 Conclusion

While the OBC code is not functioning properly, the Windows port of CSP does. We also have working USART support in CSP, which means that we have a very solid basis for developing ground station software for NUTS. FreeRTOS is also working on the OBC, so it is easy to create new tasks.

The vast majority of the features implied by the requirements specification are still lacking, so a more formal test could not be performed at this point.

Part II

NUTS software

Ground segment software 7

Introduction

This chapter describes the existing software for the ground segment. Software that is not directly relevant to the assignment will not be described. The rest of the chapter is about the porting of Cubesat Space Protocol for Windows. Testing of the implementation is detailed in chapter 6

7.1 Previous work

Currently, there is a single ground segment program for the OBC. This program uses an USART to communicate with the OBC firmware, and gives a graphical presentation of the results. This program works with the firmware developed as part of the task of designing the OBC module. The plan is that this program will evolve into a fuilly featured ground station program for controlling NUTS after launch. The program is written in C# and makes use of the .NET 4-framework, so we have to keep this in mind when choosing third party libraries.

The use of this program, and other software, basically dictates that we use Windows on the ground station.

7.2 Porting Cubesat Space Protocol

Prior to this project, libcsp was working on GNU/Linux and FreeRTOS. Looking at the features and testing the library in loopback mode in GNU/Linux, lead to the conclusion that trying to port it to Windows, was well worth the effort. This section describes the process of porting core libcsp to Windows, and creating a USART interface for it. Source code listings can be found in A.2.

Porting libcsp to Windows

The code of libcsp is quite modular. Almost all of the platform specific code is located in subfolders under `src/arch/` in the root folder of the project. The first step in developing a port, was to have a look at the ”POSIX” port. This was an obvious choice, because the author was already familiar with parts of the API dictated by the POSIX standard.

Now there was two ways of solving this: The first is to try to find an Windows-implementation of the POSIX API, and the other is to use native API functions. Both strategies were tried in parallell.

Because the author was not familiar with the Windows API, the obvious first step was to map the POSIX port into a set of ”How do I...”:

- Create a thread
- Get the amount of available heap memory
- Create a concurrent queue using the Windows API
- Get the current time
- Allocate memory? What is the difference between GlobalAlloc and LocalAlloc? Can I use malloc?

Each of those items were explored separately. In addition a suitable compiler had to be found; libcsp is littered with compiler-specific directives. The conclusion is that it would not be possible to use Microsoft’s own C compiler. Viable alternatives was to use MinGW or MinGW-w64 when developing a native port. The other option was to use GCC from within Cygwin.

In the end, a native port of libcsp was developed. There are mainly three reasons for this:

1. The POSIX implementation in Cygwin was lacking at least a function to get the time from a monotonically increasing clock source.
2. The resulting binary would have been dependent on a cygwin.dll
3. The upstream project developers did not want to use Cygwin

The native port of libcsp was first compiled using MinGW. We found that the Windows API headers part of the MinGW environment were lacking support for ”condition variables”. Support for this was introduced with Windows Vista, and apparently the MinGW-developers haven’t noticed.

Before investigating further, a another development environment was tried: MinGW-w64. This is more than a compiler: It is also a combination of runtime libraries and headers. To be on the safe side, we tried to install a completely new development toolchain for MinGW-w64. Because we could not find any working binaries, the whole toolchain plus compiler had to be compiled. Details on how that was done is described in the appendix, in section A.1.

Unfortunately, this turned out to be a waste of time. It did work, but was not necessary to make use of condition variables. Instead, it was enough to create appropriate declarations for the implicated functions. These declaration can be found in the file `src/arch/windows/windows_glue.h`, in the Cubesat Space Protocol port.

Merging Cubesat Space Protocol

We did make an effort at merging the port with the upstream project. This is an advantage for both parties. For GomSpace, it is nice to have support for another platform. This increases the value of their product. For us it is also an advantage: It is easier to keep track of the changes in the Cubesat Space Protocol library, and stay updated.

Because of the collaboration with the upstream developers, it was feasible to use our own Subversion repository when working with Cubesat Space Protocol. Instead, Git was found to be a better alternative.

Result

libcsp has been ported to Windows, and the changes have been merged with the upstream project. However it is still not possible to use the libcsp headers with MSVC. This is not a significant limitation at this point. But we will have to figure out how to make use of this library from a program written in C#.

Space segment software 8

Introduction

This chapter described the current state of the NUTS space segment software.

8.1 Previous work

A test program has been developed for the OBC. This test program takes commands via a USART-like interface where the OBC acts like a USB CDC devic, implementing the ”Abstract Control Model”. The program is able to control the backplane, and includes code snippets for writing to external memories.

8.2 OBC software

In this section we describe the software and the toolchain for the OBC.

Operating system

Because Cubesat Space Protocol is ported to FreeRTOS, and a FreeRTOS port existsis for the MCU on the OBC, the choice of OS on the OBC is quite clear. But compiling the FreeRTOS port found on `http://www.freertos.org/` would not work with the current version of the Atmel GNU toolchain. Instead, it was found that it was easier to start with an example project targeting one of Atmel's evaluation board. More details on how this example project was customised to our needs, can be found in the appendix in section A.7

C library

AVR 32-bit GNU Toolchain uses Newlib as the C standard library. The Newlib project on the official project website at `http://sourceware.org/newlib`, does not support AVR 32s. Instead Atmel, have created ther own set of patches to Newlib. These patches targets version 1.16.0 of Newlib [7].

In addition, Atmel have produced their own AVR32-port for GCC. These patches have not been merged with the upstream project, are part of the default installation of the AVR GNU toolchain.

Memory requirements

The combination of FreeRTOS, cubesat space protocol and Newlib resulted in quite high memory use. In FreeRTOS, the task context is stored on the top of the stack for each task and the stack for each task is allocated from the heap. This means that the task stacks and ”ordinary” dynamically allocated memory compete fo the same memory area.

The task stack has to be large enough for the most ”stack-consuming” function used by each task. In one case, we had problems with a stack overflow. It turned out that the `printf` function in Newlib was the culprit.

Result

FreeRTOS is working on the OBC, but there are some problems with using CSP in loopback mode. This is the first step in using CSP on the OBC, so this problem has to be solved before proceeding with building more elaborate services on top of the CSP library.

We have also seen that the memory requirements of the OBC firmware are quite high. A more long-term solution is to move the heap to external SRAM. But on the other hand, Atmel recommends that the stack is kept in embedded data memory because the access time is lower. This warrants more investigation.

Part III

Conclusion and evaluation

Conclusion and evaluation 9

Introduction

This chapter summarises this project. First we summarise the results of the project work, and then we evaluate the results and how externalities have affected the outcome.

9.1 Conclusion

Here we summarise the results of the project, and the decisions taken.

Overall system

Cubesat space protocol is used both on the OBC and the ground station. libcsp has not been tested for interoperability with the existing software on the ground station.

FreeRTOS is the chosen operating system on the OBC. In addition to libcsp, we use a USB stack implementing the device part of the USB CDC device class. The OBC still lacks a lot of drivers. The drivers developed during the design of the OBC have not been integrated into the current OBC firmware.

Testing

It is difficult to test embedded systems. The strategy for testing the Cubesat space protocol when porting this to windows, have been to try to isolate as much of the library as possible, and start working with the most trivial parts first.

Testing the OBC firmware was difficult at best. For instance, a stack overflow caused a lot of headache during development. The only visible behavior, was that the OBC ended up in an seemingly arbitrary exception handling routine. It was not possible to deduce *why* or *what* caused this to happen. In the end, the problem was a stack overflow when calling `printf`. The lessons learned here, is that concurrency makes both testing and debugging difficult.

Unit-testing has not been considered in this project. It would be interesting to research what kind of options we have. But it is important to note that unit-testing would most likely not reveal the problems enountered during testing of FreeRTOS and Cubesat Space Protocol on the OBC.

Summary

Cubesat space protocol, libcsp, was found to be a capable communications library. By using libcsp, and using our own services on top of that, we can take advantage of the fact that libcsp is a tried and testet product. Another option would be to develop a solution from scratch. This has been done in many other CubeSat projects. But we would still need at least the following features:

- Packet routing
- Packet checksumming
- Packet authentication or encryption
- Services for getting system status
- Be relative hardware independent

Each of those features themselves, would probably not take a tremendous effort to implement ourselves. Especially because we only have to implement exactly what we need. But we would still have to spend a lot of time designing, implementing and testing the system.

FreeRTOS was found to be the only viable OS alternative. There are other operating systems, but unless we are willing to do the porting ourselves, there is no other cost-effective option. libcsp is also ported to FreeRTOS.

It would be possible to implement operating system-like features ourselves. And while that would certainly be an interesting task, i would also just as certain be extremely time consuming.

9.2 Evaluation

Here I summarise the experiences from the project, and evaluate the work done in the project.

NUTS project

One of the main problems with developing software in this project, is to get an idea of what we are trying to do. The project is a typical example of bottom-up design. When it is not clear exactly what the satellite is supposed to do, this kind of strategy makes it difficult to develop software. While the project work described in this report mainly focuses on a single module, namely the OBC, the software still has to be design for the whole system. This means that developing software for the OBC, involves a huge integration problem.

For a software developer, it would be easier to do it in the opposite direction:

1. Decide what the system should do
2. Create a requirements specification for this
3. Design an architecture
4. Design a system architecture
5. Decide what kind of hardware is necessary to implement the architecture, with the required properties

Risks

It seems that one of the problems in this project is sharing of knowledge. Because this is a last-year project for most students, we are always being drained for knowledge.

Also, the hardware is a variation point. As it is now, we depend on the fact that both the radio and the OBC are using the same processor. Depending on what kind of hardware is used in the other modules, it may be challenging to use cubesat space protocol throughout the system. It may even be necessary to either port FreeRTOS to another architecture or develop a minimal operating system.

Resources devoted to software design

Having a single persion work on software is way too little

Issues

There are a couple of issues with the current implementation:

- libcsp must be integrated with the existing ground station software
- We are very close to reaching the limits of the internal OBC memory
- Input from the OBC code is not working as expected (`scanf`)
- The current `write`-implementation depends on FreeRTOS queues. Maybe it should go through a gateway task, instead of communicating (in)directly with the device_cdc_task.

Future work

Software-wise, the current implementation is sorely lacking features and testing. Suggestions for future work are:

1. Improve the I/O-facilities.
2. Develop a test plan for the software
3. Create integration tests
4. Develop services for peristent storage on the OBC, possibly using a file system
5. Write hardware drivers
6. Test the OBC firmware on the radio module

References

[1] Swisscube launch schedule, 2009.

[2] Ansat project. www.ANSAT.no, 2011. Last read: 17 December 2011.

[3] Built-in functions specific to particular target machines. http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Target-Builtins.html#Target-Builtins, 2011. Last read: December 17 2011.

[4] Other built-in functions provided by gcc. http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Other-Builtins.html#Other-Builtins, 2011. Last read: December 17 2011.

[5] Swiss cube — the first swiss satellite. http://swisscube.epfl.ch/, 2011. Last read: 17 December 2011.

[6] Swisscube — gunter's space page. http://space.skyrocket.de/doc_sdat/swisscube.htm, 2011. Last read: 17 december 2011.

[7] Atmel. *AVR 32-bit GNU Toolchain: Release 3.3.0.275*.

[8] Atmel. *AVR32UC Technical Reference Manual*, 2010.

[9] Roger Birkeland. Nuts-1 mission statement. Technical report, NTNU, 2011.

[10] Callum Chartier, Michael Mackay, Drew Ravalico, Sonja Russell, and Andrew Wallis. Ausat final report. Master's thesis, The University of Adelaide, 2010.

[11] PC/104 Consortium. Pc/104 specifications. http://www.pc104.org/specifications.php, 2011. Last read: 17 December 2011.

[12] Benoit Cosandier, Florian George, and Ted Choueiri. Swisscube flight software architecture. Master's thesis, EPFL, 2007.

[13] David Crettaz. Phase c: Cdms control & data management system. Master's thesis, HES-SO, 2007.

[14] Dewald de Bruyn. Power distribution and conditioning for a small student satellite - design of the nuts backplane & eps module. Master's thesis, NTNU, 2011.

[15] IEEE Standard for Software and System Test Documentation 829-2008. *IEEE Standard for Software and System Test Documentation 829-2008*, 2008.

[16] Anton Ivanov Muriel Noca. SwissCube status may 2 2010, 2010. Last read: December 17 2011.

[17] NAROM. Nasjonalt studentsatellittprogram, ANSAT, 2011. Last read: 17 December 2011.

[18] Muriel Nocaa, Fabien Jordan, Nicolas Steiner, Ted Choueiri, Florian George, Guillaume Roethlisberger, Noemy Scheidegger, Herve Peter-Contesse, Maurice Borgeaud, Renato Krpoun, and Herbert Shea. Lessons learned from the first swiss pico-satellite: Swisscube. In *Small Satellite Conference*, 2009.

[19] NTNU. NUTS - NTNU Test Satellite. A Norwegian CubeSat Project - Current activities, 2011. Last read: 17 December 2011.

[20] Philips. *UM10204 I2C bus specification and user manual*, 2007.

[21] Tapparel Pierre-Andre. Cdms. Master's thesis, HEVs, 2007.

[22] Guillaume Roethlisberger. Swisscube structural design and flight system configuration. Master's thesis, EPFL, 2007.

[23] California Polytechnic State University. Cubesat design specification, rev. 12, 2009.

[24] Marius Voldstad. Internal data bus of a small student satellite. Master's thesis, NTNU, 2011.

Appendix A

A.1 Installing and configuring MinGW

Introduction

This section describes how to set up MinGW for compiling programs in Windows.

Installing MinGW

An installer for MSYS and MinGW can be obtained by navigating to the project page on `http://sourceforge.net/projects/mingw/files/` and downloading the latest version of mingw-get-inst.

mingw-get-inst is a graphical installation program that installs both MinGW and MSYS. This installer will actually download mingw-get, and it is this program that will actually perform the installation. After installation, mingw-get can be used to download both updates and new programs available in the package repository.

When installing using mingw-get-inst, remember that

- Choose to use the ”pre-packaged repository catalogue”
- The installation path should not contain any whitespaces. The default of using `C:\MinGW` is good.

Setting up MinGW

Move the home directory

As default, MSYS will create a home directory derived from the Windows user name. If this name does *not* contain a whitespace, this section can be skipped.

First, open the MinGW-shell. Next, open /etc/profile within an text editor. At around line 30, the home directory is created if it doesn't already exist. If you are running MinGW shell, the home directory does exist, unless we rename it. So change the text from this:

```
# Set up USER's home directory
if [ -z "$HOME" ]; then
 HOME="/home/$LOGNAME"
fi
```

into something like this:

```
LOGNAME=${LOGNAME// /_}
# Set up USER's home directory
if [ -z "$HOME" ]; then
  HOME="/home/$LOGNAME"
fi
```

Save the file and exit the text editor, and navigate to the home directory by executing:

```
cd ~ && cd .. && mv My\ User\ Name My_User_Name
```

Open a new MinGW shell, to verify that everything is working. Because we changed the format of the LOGNAME variable, and renamed the home directory to follow the same formatting, we should end up in the same directory. The only difference should be the name of the home directory.

Create a programs folder within the home folder (and set up bash)

We are going to create a folder hierarchy within the home folder, similar to /usr/local. Within this hierarchy, we can install things like subversion – without affecting the ″global″ environment.

First, we need to to make `/etc/profile` ″source″ per-user settings. At the end of `/etc/profile`, add the logic for doing that. So the last line would look something like this:

```
if [ -e ~/.bashrc ]; then
  . ~/.bashrc
fi
```

Next, create the file ‘.bashrc‘ in the home directory. In this file, add the following statements:

```
alias g=’grep -r --exclude="*\.svn*" --color=always’
alias vi=vim
export PATH=$PATH:~/local/bin
```

This will also add useful aliases for vi and grep. The alias for grep is useful, because it makes grep use colorised output and makes it exclude subversion ”control” directories.

Next, create the folder that we just added to path:

```
mkdir -pv ~/local/bin
```

Installing Subversion

It is useful to be able to use subversion within the MSYS environment. A viable option is to use Subversion for Windows. The most recent version as of this writing, is 1.16.17. I prefer to simply download an archive file - as opposed to using an installer. Both would work, but when using the installer, the location of the binaries will have to be added to the path.

Assuming we are going to download and extract Subversion 1.6.17 (`http://sourceforge.net/projects/win32svn/files/1.6.17/svn-win32-1.6.17.zip/download`) from within MSYS:

```
    cd ~
    mingw-get install msys-wget msys-unzip
    wget \
 http://sourceforge.net/projects/win32svn/files/1.6.17/svn-win32-1.6.17.zip/downloa
    unzip svn-win32-1.6.17.zip
    cd svn-win32-1.6.17
    mv bin/* ~/local/bin
```

Installing dependencies for compiling GCC

GCC has several build-time dependencies, and some of them are not part of a ”standard” installation of MinGW. I found the following libraries to be missing:

- Mpc
- MPFR

- GMP

They can be installed by executing: `mingw-get install mpc mpfr gmp`

Notes about MinGW, MinGW-w32 & MinGW-w64

MinGW was formerly known as mingw32. The project was renamed to MinGW, to remove any impressions that it was limited to 32 bit archiectures only. It provides a set of runtime headers, but does not contain a C runtime library. The C runtime library is provided by Microsoft, and distributed with Windows. More specifically, MinGW does not provide a CRT because GCC does not provide it. However, GCC comes with some basic runtime support routines for things like exception handling (but then you must link with -lgcc instead of the often implicit -lc). In addition it comes with an implementation of STL – libstdc++. On a typical Linux system, the C library is glibc.

MinGW-w64 began as a spinoff project from mingw32, providing support for creating 64-bit binaries. It provides both headers and a runtime library, in addition to supporting both x86-32 and x86-64. When compiling for x86-32, the toolchain binaries are prefixed with mingw-w32. MinGW-w64 also provides a C interface to msvcrt.dll, and more up-to-date headers than MinGW. The C interface allows GCC to build for Windows.

What's a cross compiler?

In the context of compilers, the concepts about a "target", "host" and the "build machine" are common. In short:

- build: The machine you are building on
- host: The machine that you are building for
- target: The machine that you will produce code for. E.g. GCC will produce code for x86-64

If all of these three are equal, we build a native toolchain. If build and host are the same, but target is different, you are building a cross toolchain.

Get the sources of GCC and MinGW-w64

I've chosen to use the 4.6 branch of GCC. There is no particular reason for this. Using the source from trunk would probably also work, but I don't know much about the development process to say anything about the stability of the "HEAD" of the 4.6 branch or trunk.

```
    mkdir ~/repos
    cd repos
    svn co \
svn://http://gcc.gnu.org/svn/gcc/branches/gcc-4_6-branch gcc46x
```

For MinGW-w64, you can fetch the code either from the subversion repository, or download a snapshot from `http://sourceforge.net/projects/mingw-w64/files/`. If you choose to do the latter download the latest "headers and CRT source" into the `/repos` folder.

Compile and install binutils

Download and extract binutils to /repos. Then:

```
    mkdir -pv ~/repos/build/binutils
    cd ~/repos/build/binutils
    ../../binutils-2.22.51/configure --prefix=/usr/local \
--target=i686-w64-mingw32 --disable-multilib
    make
    make install
```

The build directory can be removed after installing. Also, the bin folder will have to be added to path, in order for it to be detected during configuration of gcc.

It is important to specify the target. Otherwise the "guessed" target would be i686-pc-mingw32, and linking would fail when compiling with make.

Compile and install the headers for MinGW-w64

Here, we assume that the sources are located in `/repos/mingw-w64-2.0`. We are going to build the sources in a separate directory:

```
    mkdir -pv ~/repos/build/mingw-w64-headers
    cd ~/repos/build/mingw-w64-headers
    ../../mingw-w64-v2.0/mingw-w64-headers/configure \
--prefix=/usr/local --host=i686-w64-mingw32 --disable-multilib
    make install
```

Compiling GCC core

Again, we are going to configure and compile the sources from outside the project folder

```
    mkdir -pv ~/repos/build/gcc
    cd ~/repos/build/gcc
    ../../gcc-4.6/configure --prefix=/usr/local --target=i686-w64-mingw32
    make all-gcc
    make install-gcc
```

These steps will build and install GCC core. That is, only the C-compiler.

Compiling the C runtime library

```
    mkdir -pv ~/repos/build/mingw-w64-crt
    cd ~/repos/build/mingw-w64-crt
    ../../mingw-w64-v2.0/mingw-w64-crt/configure \
--prefix=/usr/local --target=i686-w64-mingw32
    make
    make install
```

Compiling and installing the rest of GCC

```
    cd ~/repos/build/gcc
    make
    make install
```

The C runtime library is necessary for using condition variables from the Windows API. But in order to build the C runtime library, we also have to install both mingw-w64 headers and GCC. Otherwise the configure-script for the mingw-64 crt will fail because it is missing a header provided by GCC core.

A.2 Cubesat space protocol, windows port

Introduction

This section contains the source code for the most relevant files in the windows port, developed as part of this project. CSP have most of the architecture-specific files in the folder src/arch/windows. Headers in src/src include the headers for the architecture that is the build target. In addition various small changes had to be made. When creating a thread, a pointer to the thread function has to be passed to the threading API. However, the calling conventions are different for each platform. This had to be changed in src/csp_route.c. This files contains the definitions for the CSP router task, plus additional helper functions. Also, the service handler had to be changed: The service handler has to be able to process a "get available memory"-request. This is platform dependent.

After merging with the upstream project, the libcsp developers have made some restructuring changes to their source code. This means that the code listed here have evolved over time, and it is not unlikely that the code structure will change again. The original upstream project can be found at https://github.com/danerik/libcsp while the git repo used during porting is at the following URL https://github.com/danerik/libcsp

There are two main examples of code restructuring done since the merge with upstream libcsp: An abstraction for threads (that addresses the issue with the router task), and the service handler. I have taken the liberty to list the windows-specific service handler, because this file contains the code that was originally part of the src/src/csp_service_handler.c file.

main

```
#include <stdio.h>
#include <process.h>
#include <Windows.h>
#undef interface
#include <csp/csp.h>
#include <csp/interfaces/csp_if_kiss.h>
#include <csp/drivers/usart.h>

#define PORT 10
#define MY_ADDRESS 1
```

```
#define SERVER_TIDX 0
#define CLIENT_TIDX 1
#define USART_HANDLE 0

unsigned WINAPI serverTask(void *params);
unsigned WINAPI clientTask(void *params);

static HANDLE threads[2];

int main(int argc, char* argv[]) {
    csp_debug_toggle_level(CSP_PACKET);
    csp_debug_toggle_level(CSP_INFO);

    struct usart_conf settings;
    settings.device = argc != 2 ? "COM4" : argv[1];
    settings.baudrate = CBR_9600;
    settings.databits = 8;
    settings.paritysetting = NOPARITY;
    settings.stopbits = ONESTOPBIT;
    settings.checkparity = FALSE;

    csp_buffer_init(10, 300);
    csp_init(MY_ADDRESS);

    usart_init(&settings);

    csp_kiss_init(usart_putstr, usart_insert);
        usart_set_callback(csp_kiss_rx);

    csp_route_set(MY_ADDRESS, &csp_if_kiss, CSP_NODE_MAC);
    csp_route_start_task(0, 0);

    csp_conn_print_table();
    csp_route_print_table();
    csp_route_print_interfaces();

    threads[SERVER_TIDX] = (HANDLE)_beginthreadex(NULL, 0, &
       serverTask, NULL, 0, NULL);
    threads[CLIENT_TIDX] = (HANDLE)_beginthreadex(NULL, 0, &
       clientTask, NULL, 0, NULL);

    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    return 0;
}


unsigned WINAPI serverTask(void *params) {
    int running = 1;
    csp_socket_t *socket = csp_socket(CSP_SO_NONE);
    csp_conn_t *conn;
    csp_packet_t *packet;
    csp_packet_t *response;

    response = csp_buffer_get(sizeof(csp_packet_t) + 2);
    if( response == NULL ) {
        fprintf(stderr, "Could not allocate memory for response
           packet!\n");
        return 1;
    }
    response->data[0] = 'O';
    response->data[1] = 'K';
    response->length = 2;


    csp_bind(socket, CSP_ANY);
    csp_listen(socket, 5);

    while(running) {
        if( (conn = csp_accept(socket, 10000)) == NULL ) {
            continue;
        }

        while( (packet = csp_read(conn, 100)) != NULL ) {
            switch( csp_conn_dport(conn) ) {
                case PORT:
                    if( packet->data[0] == 'q' )
                        running = 0;
                    csp_buffer_free(packet);
                    csp_send(conn, response, 1000);
                    break;
                default:
                    csp_service_handler(conn, packet);
                    break;
            }
        }

        csp_close(conn);
    }

    csp_buffer_free(response);

    return 0;
}

unsigned WINAPI clientTask(void *params) {
    char outbuf = 'q';
    char inbuf[3] = {0};
    int pingResult;

    for(int i = 50; i <= 200; i+= 50) {
        pingResult = csp_ping(MY_ADDRESS, 1000, 100, CSP_O_NONE
            );
        printf("Ping with payload of %d bytes, took %d ms\n", i
            , pingResult);
        Sleep(1000);
    }
    csp_ps(MY_ADDRESS, 1000);
    Sleep(1000);
    csp_memfree(MY_ADDRESS, 1000);
    Sleep(1000);
    csp_buf_free(MY_ADDRESS, 1000);
    Sleep(1000);
    csp_uptime(MY_ADDRESS, 1000);

    Sleep(1000);
    csp_transaction(0, MY_ADDRESS, PORT, 1000, &outbuf, 1,
       inbuf, 2);
    printf("Quit response from server: %s\n", inbuf);

    return 0;
}
```

A.3 Cubesat space protocol, ground segment test code

Introduction

This section contains code listings fort the most relevant parts of the ground segment tests for libcsp. Note that the listed code relies on a more elaborate program that wraps the Windows API behind classes for events and threads.

main

The ScreenLock is a singleton instance. However, we cannot rely on thread safe static initialisation, so the main thread acquires and releases the thread once before starting the server and client.

```
#include <iostream>
#include <process.h>
#include <Windows.h>
#include "thread.h"
#include "mutex.h"
```

```
#include "event.h"
#include <client.h>
#include <server.h>
#include <csp/csp.h>
#include <common.h>
#include <screenlock.h>
#define DEBUG
using namespace std;

Mutex screenMutex;
Event readyEvent;

void configureDebugging();

void initScreenMutexSingleton();

int main() {
    initScreenMutexSingleton();
    // Set up the csp buffer the same way, as it would've been
       with static alloc
    csp_buffer_init(CSP_BUFFER_COUNT, CSP_BUFFER_SIZE);
    csp_init(ADDRESS);
    csp_route_start_task(4096, 1);

#ifdef DEBUG
    configureDebugging();
#endif

    Client c(readyEvent);
    Server s(readyEvent);
    try {
        c.Resume();
        s.Resume();
        c.Join();
        s.Join();
    } catch(exception e) {
        cout << e.what() << endl;
    }

    return 0;
}

void initScreenMutexSingleton() {
    ScreenLock::Instance().Acquire();
    ScreenLock::Instance().Release();
}

void configureDebugging() {
    csp_debug_toggle_level(CSP_INFO);
    csp_debug_toggle_level(CSP_ERROR);
    csp_debug_toggle_level(CSP_WARN);
    csp_debug_toggle_level(CSP_BUFFER);
    csp_debug_toggle_level(CSP_PACKET);
    csp_debug_toggle_level(CSP_PROTOCOL);
    csp_debug_toggle_level(CSP_LOCK);
}
```

server

```
#ifndef GUARD_SERVER_H
#define GUARD_SERVER_H
#include <signalablethread.h>
#include <event.h>
#include <csp/csp.h>
#include <stdint.h>
#include <string>

class Server : public SignalableThread {
    public:
        Server(Event& evt) : SignalableThread(evt) {}
    protected:
        void Work();
    private:
        csp_socket_t* initSocket() const;
        void printString(const uint8_t *str) const;
        void processRequest(csp_conn_t *conn, csp_packet_t *
            packet) const;
        char getRequestCode(csp_packet_t* packet) const;
        static std::string m_help;
};

#endif

#include <server.h>
#include <screenlock.h>
#include <common.h>
#include <iostream>
#include <event.h>
#include <csp/csp.h>
#include <stdint.h>
#include <stddef.h>
#include <stdexcept>

using namespace std;

std::string Server::m_help = "Help␣string␣from␣server";
```

```
csp_socket_t* Server::initSocket() const {
    static size_t backlog = 10;
    static uint32_t options = 0;

    csp_socket_t *socket = csp_socket(options);
    csp_bind(socket, CSP_ANY);
    csp_listen(socket, backlog);

    return socket;
}

void Server::printString(const uint8_t *dataptr) const {
    const char *str = (const char*) dataptr;
    Print("Message␣from␣client:␣" << str);
}

void Server::processRequest(csp_conn_t *conn, csp_packet_t *
   packet) const {
    const char code = getRequestCode(packet);

    if(code == 'h')
        Print(m_help);
}

char Server::getRequestCode(csp_packet_t* packet) const {
    if( packet->length < 1 ) {
        throw runtime_error("Request␣packet␣was␣empty!");
    }
    char code = (char) packet->data[0];
    return code;
}


void Server::Work() {
    csp_socket_t *socket = initSocket();
    m_signal->Signal();

    csp_conn_t *connection;
    csp_packet_t *packet;

    Print("Server␣listening");
    while(true) {
        connection = csp_accept(socket, CSP_MAX_DELAY);
        if(connection == NULL) continue; // Timeout

        while( (packet = csp_read(connection, 1001)) != NULL )
            {
            switch(csp_conn_dport(connection)) {
                case PAYLOAD_PORT:
                    printString(packet->data);
                    break;
                case CONTROL_PORT:
                    processRequest(connection, packet);
                    break;
                default:
                    csp_service_handler(connection, packet);
                    break;
            }
            csp_buffer_free(packet);
            packet = NULL;
            csp_close(connection);
        }
    }
}
```

client

```
#ifndef GUARD_CLIENT_H
#define GUARD_CLIENT_H

#include <signalablethread.h>
#include <stdint.h>
#include <common.h>

class Client : public SignalableThread {
    public:
        Client(Event& evt) :
            SignalableThread(evt),
            m_server_addr(ADDRESS),
            m_timeout(600000) {}
    protected:
        void Work();
    private:
        void ping();
        void ps();
        void freemem();
        void freebuffers();
        void uptime();
        uint8_t m_server_addr;
        unsigned int m_timeout;
};

#endif

#include <iostream>
#include <client.h>
#include <mutex.h>
#include <screenlock.h>
```

```
#include <event.h>
#include <csp/csp.h>
#include <stdexcept>
#include <common.h>
#include <cstring>
#include <Windows.h>

using namespace std;

void Client::Work() {
    m_signal->Wait();
    Print("Client␣trying␣to␣connect␣to␣server...");

    Sleep(1000);
    ps();
    Sleep(1000);
    freemem();
    Sleep(1000);
    freebuffers();
    Sleep(1000);
    uptime();
    Sleep(1000);

    const char *message = "Hello␣world!";
    csp_packet_t *packet = (csp_packet_t*) csp_buffer_get(
        sizeof(csp_packet_t) + strlen(message) + 1);


    if(packet == NULL) {
        throw runtime_error("Failed␣to␣allocate␣memory␣for␣
            packet!");
    }

    csp_conn_t *connection = csp_connect(CSP_PRIO_NORM, ADDRESS
        , PAYLOAD_PORT, 1000, CSP_O_NONE);

    if(connection == NULL) {
        csp_buffer_free(packet);
        throw runtime_error("Connection␣failed!");
    }

    strcpy((char*)packet->data, message);

    if(!csp_send(connection, packet, 1000)) {
        csp_buffer_free(packet);
        throw runtime_error("Send␣failed!");
    }
    csp_close(connection);
}
```

```
void Client::ping() {
    int pingResult = csp_ping(m_server_addr, m_timeout, 100,
       CSP_O_NONE);
    Print("Ping␣result:␣" << pingResult);
}

void Client::ps() {
    csp_ps(m_server_addr, m_timeout);
}

void Client::freemem() {
    csp_memfree(m_server_addr, m_timeout);
}

void Client::freebuffers() {
    csp_buf_free(m_server_addr, m_timeout);
}

void Client::uptime() {
    csp_uptime(m_server_addr, m_timeout);
}
```

A.4 Cubesat space protocol, KISS test

Introduction

This section contains the source code for the test of the KISS functionality in CSP.

The code was compiled with:

```
./waf distclean configure --prefix=../csp_deploy --enable-if-kiss \
 --with-driver-usart=windows --with-os=windows --enable-examples build insta
```

csp_malloc.c

Listing A.1: src/arch/windows/csp_malloc.c

```
#include "../csp_malloc.h"
#include <stdlib.h>

void * csp_malloc(size_t size) {
        return malloc(size);
}

void csp_free(void * ptr) {
        free(ptr);
}
```

csp_queue.c

Listing A.2: src/arch/windows/csp_queue.c

```
#include <stdint.h>
#include <csp/csp.h>
#include "../csp_queue.h"
#include "windows_queue.h"

csp_queue_handle_t csp_queue_create(int length, size_t
   item_size) {
        return windows_queue_create(length, item_size);
}

void csp_queue_remove(csp_queue_handle_t queue) {
        windows_queue_delete(queue);
}

int csp_queue_enqueue(csp_queue_handle_t handle, void *value,
   uint32_t timeout) {
        return windows_queue_enqueue(handle, value, timeout);
}

int csp_queue_enqueue_isr(csp_queue_handle_t handle, void *
   value, CSP_BASE_TYPE * task_woken) {
        if( task_woken != NULL )
                *task_woken = 0;
        return windows_queue_enqueue(handle, value, 0);
}

int csp_queue_dequeue(csp_queue_handle_t handle, void *buf,
   uint32_t timeout) {
        return windows_queue_dequeue(handle, buf, timeout);
}

int csp_queue_dequeue_isr(csp_queue_handle_t handle, void * buf
   , CSP_BASE_TYPE * task_woken) {
        if( task_woken != NULL )
                *task_woken = 0;
        return windows_queue_dequeue(handle, buf, 0);
}

int csp_queue_size(csp_queue_handle_t handle) {
        return windows_queue_items(handle);
}

int csp_queue_size_isr(csp_queue_handle_t handle) {
        return windows_queue_items(handle);
}
```

csp_semaphore.c

Listing A.3: src/arch/windows/csp_semaphore.c

```
#include <Windows.h>
#include <csp/csp.h>
#include "../csp_semaphore.h"

int csp_mutex_create(csp_mutex_t * mutex) {
    HANDLE mutexHandle = CreateMutex(NULL, FALSE, FALSE);
    if( mutexHandle == NULL ) {
        return CSP_MUTEX_ERROR;
    }
    *mutex = mutexHandle;
    return CSP_MUTEX_OK;
}

int csp_mutex_remove(csp_mutex_t * mutex) {
    if( !CloseHandle(*mutex) ) {
        return CSP_MUTEX_ERROR;
    }
    return CSP_MUTEX_OK;
}

int csp_mutex_lock(csp_mutex_t * mutex, uint32_t timeout) {
    if(WaitForSingleObject(*mutex, timeout) == WAIT_OBJECT_0) {
            return CSP_MUTEX_OK;
    }
    return CSP_MUTEX_ERROR;

}

int csp_mutex_unlock(csp_mutex_t * mutex) {
    if( !ReleaseMutex(*mutex) ) {
        return CSP_MUTEX_ERROR;
    }
    return CSP_MUTEX_OK;
}

int csp_bin_sem_create(csp_bin_sem_handle_t * sem) {
    HANDLE semHandle = CreateSemaphore(NULL, 1, 1, NULL);
    if( semHandle == NULL ) {
        return CSP_SEMAPHORE_ERROR;
    }
    *sem = semHandle;
    return CSP_SEMAPHORE_OK;
}

int csp_bin_sem_remove(csp_bin_sem_handle_t * sem) {
     if( !CloseHandle(*sem) ) {
          return CSP_SEMAPHORE_ERROR;
     }
     return CSP_SEMAPHORE_OK;
}

int csp_bin_sem_wait(csp_bin_sem_handle_t * sem, uint32_t
    timeout) {
     if( WaitForSingleObject(*sem, timeout) == WAIT_OBJECT_0 ) {
               return CSP_SEMAPHORE_OK;
     }
     return CSP_SEMAPHORE_ERROR;

}

int csp_bin_sem_post(csp_bin_sem_handle_t * sem) {
     if( !ReleaseSemaphore(*sem, 1, NULL) ) {
          return CSP_SEMAPHORE_ERROR;
     }
     return CSP_SEMAPHORE_OK;
}

int csp_bin_sem_post_isr(csp_bin_sem_handle_t * sem,
    CSP_BASE_TYPE * task_woken) {
     if( task_woken != NULL ) {
          *task_woken = 0;
     }
     return csp_bin_sem_post(sem);
}
```

csp_system.c

This code was moved from its original location in `src/csp_service_handler.c`.

Listing A.4: src/arch/windows/csp_system.c

```
/*
Cubesat Space Protocol - A small network-layer protocol
    designed for Cubesats
Copyright (C) 2011 Gomspace ApS (http://www.gomspace.com)
Copyright (C) 2011 AAUSAT3 Project (http://aausat3.space.aau.dk
    )

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
```

```
version 2.1 of the License, or (at your option) any later
    version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General
    Public
License along with this library; if not, write to the Free
    Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301  USA
*/

#include <stdint.h>
#include <string.h>
#include <Windows.h>

#include <csp/csp.h>
#include <csp/csp_error.h>

#include "../csp_system.h"

int csp_sys_tasklist(char * out) {
        strcpy(out, "Tasklist␣not␣available␣on␣Windows");
        return CSP_ERR_NONE;
}

uint32_t csp_sys_memfree(void) {
        MEMORYSTATUSEX statex;
        statex.dwLength = sizeof(statex);
        GlobalMemoryStatusEx(&statex);
        DWORDLONG freePhysicalMem = statex.ullAvailPhys;
        size_t total = (size_t) freePhysicalMem;
        return (uint32_t)total;
}

int csp_sys_reboot(void) {
        /* TODO: Fix reboot on Windows */
        csp_log_error("Failed␣to␣reboot\r\n");

        return CSP_ERR_INVAL;
}

void csp_sys_set_color(csp_color_t color) {
        /* TODO: Add Windows color output here */
}
```

csp_time.c

Listing A.5: src/arch/windows/csp_time.c

```
#include <Windows.h>
#include <stdint.h>
#include "../csp_time.h"

uint32_t csp_get_ms(void) {
        return (uint32_t)GetTickCount();
}

uint32_t csp_get_ms_isr(void) {
        return csp_get_ms();
}

uint32_t csp_get_s(void) {
        uint32_t time_ms = csp_get_ms();
        return time_ms/1000;
}

uint32_t csp_get_s_isr(void) {
        return csp_get_s();
}
```

windows_glue.h

This files contains definintions for condition variables. Without this "glue-header", it would not be possible to build libcsp under the current version of MinGW.

Listing A.6: src/arch/windows/windows_glue.h

```
#ifndef WINDOWS_GLUE_H
#define WINDOWS_GLUE_H

#include <Windows.h>
#undef interface

#if (_WIN32_WINNT >= 0x0600)

#define RTL_CONDITION_VARIABLE_INIT 0
#define RTL_CONDITION_VARIABLE_LOCKMODE_SHARED 1
#define CONDITION_VARIABLE_INIT RTL_CONDITION_VARIABLE_INIT
#define CONDITION_VARIABLE_LOCKMODE_SHARED
    RTL_CONDITION_VARIABLE_LOCKMODE_SHARED
```

```
typedef PVOID RTL_CONDITION_VARIABLE;
typedef RTL_CONDITION_VARIABLE CONDITION_VARIABLE, *
    PCONDITION_VARIABLE;

WINBASEAPI VOID WINAPI InitializeConditionVariable(
    PCONDITION_VARIABLE ConditionVariable);
WINBASEAPI WINBOOL WINAPI SleepConditionVariableCS(
    PCONDITION_VARIABLE ConditionVariable, PCRITICAL_SECTION
    CriticalSection, DWORD dwMilliseconds);
WINBASEAPI VOID WINAPI WakeAllConditionVariable(
    PCONDITION_VARIABLE ConditionVariable);
WINBASEAPI VOID WINAPI WakeConditionVariable(
    PCONDITION_VARIABLE ConditionVariable);

#endif // _WIN#"_WINNT
#endif
```

windows_queue

Listing A.7: src/arch/windows/windows_queue.h

```
#ifndef _WINDOWS_QUEUE_H_
#define _WINDOWS_QUEUE_H_

#ifdef __cplusplus
extern "C" {
#endif

#include <Windows.h>
#include "windows_glue.h"
#undef interface

#include "../csp_queue.h"

#define WINDOWS_QUEUE_ERROR CSP_QUEUE_ERROR
#define WINDOWS_QUEUE_EMPTY CSP_QUEUE_ERROR
#define WINDOWS_QUEUE_FULL CSP_QUEUE_ERROR
#define WINDOWS_QUEUE_OK CSP_QUEUE_OK

typedef struct windows_queue_s {
    void * buffer;
    int size;
    int item_size;
    int items;
    int head_idx;
    CRITICAL_SECTION mutex;
    CONDITION_VARIABLE cond_full;
    CONDITION_VARIABLE cond_empty;
} windows_queue_t;

windows_queue_t * windows_queue_create(int length, size_t
    item_size);
void windows_queue_delete(windows_queue_t * q);
int windows_queue_enqueue(windows_queue_t * queue, void * value
    , int timeout);
int windows_queue_dequeue(windows_queue_t * queue, void * buf,
    int timeout);
int windows_queue_items(windows_queue_t * queue);

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif // _WINDOWS_QUEUE_H_
```

Listing A.8: src/arch/windows/windows_queue.c

```
#include "windows_queue.h"
#include "windows_glue.h"
#include <Windows.h>

static int queueFull(windows_queue_t * queue) {
        return queue->items == queue->size;
}

static int queueEmpty(windows_queue_t * queue) {
        return queue->items == 0;
}

windows_queue_t * windows_queue_create(int length, size_t
    item_size) {
        windows_queue_t *queue = (windows_queue_t*)malloc(
            sizeof(windows_queue_t));
        if(queue == NULL)
                goto queue_malloc_failed;

        queue->buffer = malloc(length*item_size);
        if(queue->buffer == NULL)
                goto buffer_malloc_failed;

        queue->size = length;
        queue->item_size = item_size;
        queue->items = 0;
        queue->head_idx = 0;

        InitializeCriticalSection(&(queue->mutex));
        InitializeConditionVariable(&(queue->cond_full));
        InitializeConditionVariable(&(queue->cond_empty));
        goto queue_init_success;

buffer_malloc_failed:
        free(queue);
        queue = NULL;
queue_malloc_failed:
queue_init_success:
        return queue;
}

void windows_queue_delete(windows_queue_t * q) {
        if(q==NULL) return;
        DeleteCriticalSection(&(q->mutex));
        free(q->buffer);
        free(q);
}

int windows_queue_enqueue(windows_queue_t * queue, void * value
    , int timeout) {
        int offset;
        EnterCriticalSection(&(queue->mutex));
        while(queueFull(queue)) {
                int ret = SleepConditionVariableCS(&(queue->
                    cond_full), &(queue->mutex), timeout);
                if( !ret ) {
                        LeaveCriticalSection(&(queue->mutex));
                        return ret == WAIT_TIMEOUT ?
                            WINDOWS_QUEUE_FULL :
                            WINDOWS_QUEUE_ERROR;
                }
        }
        offset = ((queue->head_idx+queue->items) % queue->size)
            * queue->item_size;
        memcpy((unsigned char*)queue->buffer + offset, value,
            queue->item_size);
        queue->items++;

        LeaveCriticalSection(&(queue->mutex));
        WakeAllConditionVariable(&(queue->cond_empty));
        return WINDOWS_QUEUE_OK;
}

int windows_queue_dequeue(windows_queue_t * queue, void * buf,
    int timeout) {
        EnterCriticalSection(&(queue->mutex));
        while(queueEmpty(queue)) {
                int ret = SleepConditionVariableCS(&(queue->
                    cond_empty), &(queue->mutex), timeout);
                if( !ret ) {
                        LeaveCriticalSection(&(queue->mutex));
                        return ret == WAIT_TIMEOUT ?
                            WINDOWS_QUEUE_EMPTY :
                            WINDOWS_QUEUE_ERROR;
                }
        }
        memcpy(buf, (unsigned char*)queue->buffer+(queue->
            head_idx%queue->size*queue->item_size), queue->
            item_size);
        queue->items--;
        queue->head_idx = (queue->head_idx + 1) % queue->size;

        LeaveCriticalSection(&(queue->mutex));
        WakeAllConditionVariable(&(queue->cond_full));
        return WINDOWS_QUEUE_OK;
}

int windows_queue_items(windows_queue_t * queue) {
        int items;
        EnterCriticalSection(&(queue->mutex));
        items = queue->items;
        LeaveCriticalSection(&(queue->mutex));

        return items;
}
```

usart_windows

This file contains the USART driver for the windows port of CSP.

Listing A.9: src/drivers/usart/usart_windows.c

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>

#include <csp/csp.h>
#include <csp/drivers/usart.h>

static HANDLE portHandle = INVALID_HANDLE_VALUE;
static HANDLE rxThread = INVALID_HANDLE_VALUE;
static CRITICAL_SECTION txSection;
static LONG isListening = 0;
static usart_callback_t usart_callback = NULL;

static void prvSendData(char *buf, int bufsz);
static int prvTryOpenPort(const char* intf);
static int prvTryConfigurePort(const struct usart_conf*);
static int prvTrySetPortTimeouts(void);
static const char* prvParityToStr(BYTE paritySetting);

#ifdef CSP_DEBUG
static void prvPrintError(void) {
    char *messageBuffer = NULL;
    DWORD errorCode = GetLastError();
    DWORD formatMessageRet;
    formatMessageRet = FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        errorCode,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (char*)&messageBuffer,
        0,
        NULL);

    if( !formatMessageRet ) {
        csp_log_error("FormatMessage␣error,␣code:␣%lu\n",
           GetLastError());
        return;
    }
    csp_log_error("%s\n", messageBuffer);
    LocalFree(messageBuffer);
}
#endif

#ifdef CSP_DEBUG
#define printError() prvPrintError()
#else
#define printError() do {} while(0)
#endif

static int prvTryOpenPort(const char *intf) {
    portHandle = CreateFileA(
        intf,
        GENERIC_READ|GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        0,
        NULL);

    if( portHandle == INVALID_HANDLE_VALUE ) {
        DWORD errorCode = GetLastError();
        if( errorCode == ERROR_FILE_NOT_FOUND ) {
            csp_log_error("Could␣not␣open␣serial␣port,␣because␣
                it␣didn't␣exist!\n");
        }
        else
            csp_log_error("Failure␣opening␣serial␣port!␣Code:␣%
                lu", errorCode);
        return 1;
    }
    return 0;
}

static int prvTryConfigurePort(const struct usart_conf * conf)
    {
    DCB portSettings = {0};
    portSettings.DCBlength = sizeof(DCB);
    if(!GetCommState(portHandle, &portSettings) ) {
        csp_log_error("Could␣not␣get␣default␣settings␣for␣open␣
            COM␣port!␣Code:␣%lu\n", GetLastError());
        return -1;
    }
    portSettings.BaudRate = conf->baudrate;
    portSettings.Parity = conf->paritysetting;
    portSettings.StopBits = conf->stopbits;
    portSettings.fParity = conf->checkparity;
    portSettings.fBinary = TRUE;
    portSettings.ByteSize = conf->databits;
    if( !SetCommState(portHandle, &portSettings) ) {
        csp_log_error("Error␣when␣setting␣COM␣port␣settings!␣
            Code:%lu\n", GetLastError());
        return 1;
    }

    GetCommState(portHandle, &portSettings);

    csp_log_info("Port:␣%s,␣Baudrate:␣%lu,␣Data␣bits:␣%d,␣Stop␣
        bits:␣%d,␣Parity:␣%s\r\n",
            conf->device, conf->baudrate, conf->databits, conf
                ->stopbits, prvParityToStr(conf->paritysetting))
                ;
    return 0;
}

static const char* prvParityToStr(BYTE paritySetting) {
    static const char *parityStr[] = {
        "None",
        "Odd",
        "Even",
        "N/A"
    };
    char const *resultStr = NULL;

    switch(paritySetting) {
        case NOPARITY:
            resultStr = parityStr[0];
            break;
        case ODDPARITY:
            resultStr = parityStr[1];
            break;
        case EVENPARITY:
            resultStr = parityStr[2];
            break;
        default:
            resultStr = parityStr[3];
    };
    return resultStr;
}

static int prvTrySetPortTimeouts(void) {
    COMMTIMEOUTS timeouts = {0};

    if( !GetCommTimeouts(portHandle, &timeouts) ) {
        csp_log_error("Error␣gettings␣current␣timeout␣settings\
           n");
        return 1;
    }

    timeouts.ReadIntervalTimeout = 5;
    timeouts.ReadTotalTimeoutMultiplier = 1;
    timeouts.ReadTotalTimeoutConstant = 5;
    timeouts.WriteTotalTimeoutMultiplier = 1;
    timeouts.WriteTotalTimeoutConstant = 5;

    if(!SetCommTimeouts(portHandle, &timeouts)) {
        csp_log_error("Error␣setting␣timeouts!");
        return 1;
    }

    return 0;
}

unsigned WINAPI prvRxTask(void* params) {
    DWORD bytesRead;
    DWORD eventStatus;
    uint8_t recvBuffer[24];
    SetCommMask(portHandle, EV_RXCHAR);

    while(isListening) {
        WaitCommEvent(portHandle, &eventStatus, NULL);
        if( !(eventStatus & EV_RXCHAR) ) {
            continue;
        }
        if( !ReadFile(portHandle, recvBuffer, 24, &bytesRead,
           NULL)) {
            csp_log_warn("Error receiving data! Code: %lu\n",
               GetLastError());
            continue;
        }
        if( usart_callback != NULL )
            usart_callback(recvBuffer, (size_t)bytesRead, NULL)
               ;
    }
    return 0;
}

static void prvSendData(char *buf, int bufsz) {
    DWORD bytesTotal = 0;
    DWORD bytesActual;
    if( !WriteFile(portHandle, buf, bufsz-bytesTotal, &
       bytesActual, NULL) ) {
        csp_log_error("Could not write data. Code: %lu\n",
           GetLastError());
        return;
    }
    if( !FlushFileBuffers(portHandle) ) {
        csp_log_warn("Could not flush write buffer. Code: %lu\n
           ", GetLastError());
    }
}

void usart_shutdown(void) {
    InterlockedExchange(&isListening, 0);
    CloseHandle(portHandle);
    portHandle = INVALID_HANDLE_VALUE;
    if( rxThread != INVALID_HANDLE_VALUE ) {
        WaitForSingleObject(rxThread, INFINITE);
        rxThread = INVALID_HANDLE_VALUE;
    }
    DeleteCriticalSection(&txSection);
}

void usart_listen(void) {
    InterlockedExchange(&isListening, 1);
    rxThread = (HANDLE)_beginthreadex(NULL, 0, &prvRxTask, NULL
       , 0, NULL);
}

void usart_putstr(char* buf, int bufsz) {
    EnterCriticalSection(&txSection);
    prvSendData(buf, bufsz);
    LeaveCriticalSection(&txSection);
}

void usart_insert(char c, void *pxTaskWoken) {
    /* redirect debug output to stdout */
    printf("%c", c);
}

void usart_set_callback(usart_callback_t callback) {
    usart_callback = callback;
}

void usart_init(struct usart_conf * conf) {
    if( prvTryOpenPort(conf->device) ) {
        printError();
        return;
    }

    if( prvTryConfigurePort(conf) ) {
        printError();
        return;
    }

    if( prvTrySetPortTimeouts() ) {
        printError();
        return;
    }

    InitializeCriticalSection(&txSection);

    /* Start receiver thread */
    usart_listen();
}
```

A.5 XMEGA A1 Code

Introduction

This section contains the code for the XMEGA A1 Code.. This evaluation kit was used as a "character bouncer", when testing and porting the USART-driver and KISS-encapsulation and -decapsulation in CSP.

main

Listing A.10: xmega/main.c

```
/**
 * \file
 *
 * \brief Empty user application template
 *
 */

/*
 * Include header files for all drivers that have been imported
      from
 * AVR Software Framework (ASF).
 */
#include <asf.h>

// Define the Usart used in task
#define USART USARTC0
#define USARTPORT PORTC
#define LEDPORT PORTE
#define SWITCHPORTL PORTD
#define SWITCHPORTLMASK 0x3F
#define SWITCHPORTH PORTR
#define SWITCHPORTHMASK 0x3

void SetupUsart( void )
{

    // Place a jumper to connect Pin3 and Pin2
    // PD3 (TXD0) as output
    USARTPORT.DIRSET    = PIN3_bm; // (TXD0) as output.

    // PD2 (RXD0) as input
    USARTPORT.DIRCLR    = PIN2_bm; // (RXD0) as input.
    // USARTD0; 8 Data bits, No Parity, 1 Stop bit
    USART.CTRLC = (uint8_t) USART_CHSIZE_8BIT_gc |
        USART_PMODE_DISABLED_gc | false;

    // Enable both TX on PORTC and RX on PORTD
    USART.CTRLB |= USART_TXEN_bm | USART_RXEN_bm;

    // Target: Internal RC 2MHz (default), 9600baud
    USART.BAUDCTRLA = 12;
}

volatile uint8_t recvcounter = 0;

uint8_t recvByte() {
        while( (USART.STATUS & USART_RXCIF_bm) == 0 );
        uint8_t data = USART.DATA;
        USART.STATUS |= USART_RXCIF_bm;
        return data;
}

void sendByte(uint8_t data) {
        while( (USART.STATUS & USART_DREIF_bm) == 0);
        USART.DATA = data;
}

void initLeds(void);
void initButtons(void);
void updateLeds(void);

int main (void)
{
        board_init();
        initLeds();
        initButtons();

        SetupUsart();

        PMIC.CTRL |= PMIC_LOLVLEN_bm;
        sei();

        while(1) {
                uint8_t data = recvByte();
                cli();
                recvcounter = (recvcounter + 1) & 255;
                updateLeds();
                sei();
                sendByte(data);
        }

}

void updateLeds(void) {
        LEDPORT.OUT = recvcounter;
}

void initLeds(void) {
        LEDPORT.DIR = 0xFF;
        PORTCFG.MPCMASK = 0xFF;
        LEDPORT.PIN0CTRL |= PORT_INVEN_bm;
        LEDPORT.OUT = 0x00;
}

ISR(PORTD_INT0_vect) {
        if( SWITCHPORTL.IN & PIN0_bm ) {
                recvcounter = 0;
                updateLeds();
        }
}

ISR(PORTR_INT0_vect) {
}

void initButtons(void) {
        SWITCHPORTL.DIRCLR = SWITCHPORTLMASK;
        PORTCFG.MPCMASK = SWITCHPORTLMASK;
        SWITCHPORTL.PIN0CTRL |= (SWITCHPORTL.PIN0CTRL & ~(
            PORT_OPC_gm|PORT_ISC_gm)) | PORT_INVEN_bm |
            PORT_OPC_PULLUP_gc | PORT_ISC_RISING_gc;
        SWITCHPORTL.INT0MASK |= SWITCHPORTLMASK;
        SWITCHPORTL.INTCTRL = (SWITCHPORTL.INTCTRL & ~
            PORT_INT0LVL_gm) | PORT_INT0LVL_LO_gc;


        SWITCHPORTH.DIRCLR = SWITCHPORTHMASK;
        PORTCFG.MPCMASK = SWITCHPORTHMASK;
        SWITCHPORTH.PIN0CTRL = (SWITCHPORTH.PIN0CTRL & ~(
            PORT_OPC_gm|PORT_ISC_gm)) | PORT_INVEN_bm |
            PORT_OPC_PULLUP_gc | PORT_ISC_RISING_gc;
        SWITCHPORTH.INT0MASK |= SWITCHPORTHMASK;
        SWITCHPORTH.INTCTRL = (SWITCHPORTH.INTCTRL & ~
            PORT_INT0LVL_gm) | PORT_INT0LVL_LO_gc;
}
```

A.6 OBC Code

Introduction

This section contains the OBC code. The FreeRTOS port is taken from an example project for an evaluation board that uses the same microcontroller as the OBC and the radio.

The example project provided a USB stack, implemented both for a host and a device. The source code was modified so that it would work in device mode only, and the USB CDC task was changed into something appropriate for the OBC.

In addition to that, a couple of small I/O-functions were added. They handle writing output to USB.

The list of changed file should be (more or less):

- src/main.c
- src/device_cdc_task.c

- src/config/conf_board.h
- src/config/conf_io.h
- src/config/FreeRTOSConfig.h
- src/lib/*
- Compiled CSP was put in src/thirdparty/libcsp
- src/common/boards/*

In this section, we only list the most relevant parts. Consult the enclosed source code for the full listings.

main

This file contains the definitions for the CSP server and client. The code for this is basically a verbatim copy of the loopback example bundlet with the CSP library.. This is the code in the clientTask and serverTask functions.

Listing A.11: src/main.c

```
#include <stdio.h>
#include <unistd.h>
#include "compiler.h"
#include "board.h"
#include "intc.h"
#include "power_clocks_lib.h"
#include "main.h"
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "conf_board.h"
#include "conf_usb.h"
#include "usb_task.h"
#include "device_cdc_task.h"
#include <io/io.h>
#include <csp/csp.h>

#define MY_ADDRESS 1
#define MY_PORT 10

#define USE_CSP 1

pcl_freq_param_t pcl_freq_param =
{
  .cpu_f         = APPLI_CPU_SPEED,
  .pba_f         = APPLI_PBA_SPEED,
  .osc0_f        = FOSC0,
  .osc0_startup  = OSC0_STARTUP
};

static xSemaphoreHandle prvScreenMutex;

static void printTask(void* pvParameters);
static void readTask(void* pvParameters);
static void diagTask(void *pvParameters);

#ifdef USE_CSP
static void prvStartCsp(void);
void serverTask(void* pvParams);
void clientTask(void* pvParams);
#endif

void printDiag(char cmd);

/*! \brief Main function. Execution starts here.
 *
 * \retval 42 Fatal error.
 */
int main(void)
{
        prvScreenMutex = xSemaphoreCreateMutex();

        xTaskCreate(&printTask, (const signed portCHAR *)"Print
            ␣task", 128, NULL, tskIDLE_PRIORITY, NULL);
        xTaskCreate(&diagTask, (const signed portCHAR *)"Diag␣
            task", configMINIMAL_STACK_SIZE+128, NULL,
            tskIDLE_PRIORITY, NULL);

        // Configure system clocks.
        if (pcl_configure_clocks(&pcl_freq_param) != PASS)
                return 42;

        // Initialize USB clock.
        pcl_configure_usb_clock();

        // Initialize USB task
        usb_task_init();

        io_init();
        // Initialize device CDC USB task
        device_cdc_task_init();
#ifdef USE_CSP
        prvStartCsp();
```

```
#endif
        // Start OS scheduler
        vTaskStartScheduler();
        portDBG_TRACE("FreeRTOS␣returned.");
        return 42;
}

void diagTask(void *pvParameters) {
        int input;

        portTickType xLastWakeTime;
        xLastWakeTime = xTaskGetTickCount();

        while(1) {
                input = getchar();
                xSemaphoreTake(prvScreenMutex, portMAX_DELAY);
                printf("Got␣%c\n", input);
                xSemaphoreGive(prvScreenMutex);
                vTaskDelayUntil(&xLastWakeTime, 500);
        }

        while(1) {
                input = getchar();
                if( input != 'd' ) {
                        printf("Got␣%c\n", input);
                        continue;
                }
                xSemaphoreTake(prvScreenMutex, portMAX_DELAY);
                printDiag(input);
                xSemaphoreGive(prvScreenMutex);
        }
}

void printDiag(char cmd) {
        extern uint8_t __executable_start;
        unsigned portBASE_TYPE tasksCount =
            uxTaskGetNumberOfTasks();
        signed char *buffer = NULL; //pvPortMalloc(tasksCount *
            (40+10));
        printf("\nExecutable␣data␣starts␣at␣%p\n", &
            __executable_start);
        printf("Number␣of␣tasks␣is␣%lu\n", tasksCount);

        if( buffer != NULL ) {
                vTaskList(buffer);
                printf("%s\n", buffer);
                vPortFree(buffer);
        }
}
```

```
#ifdef USE_CSP
void prvStartCsp(void) {
        csp_buffer_init(2, 300);

        /* Init CSP with address MY_ADDRESS */
        csp_init(MY_ADDRESS);

        /* Start router task with 500 word stack, OS task
           priority 1 */
        csp_route_start_task(500, tskIDLE_PRIORITY);
        //
        xTaskCreate(&serverTask, (const signed portCHAR *)"
           Server␣task", 256, NULL, tskIDLE_PRIORITY, NULL);
        xSemaphoreTake(prvScreenMutex, portMAX_DELAY);
        printf("Created␣server␣task!\n");
        xSemaphoreGive(prvScreenMutex);
        xTaskCreate(&clientTask, (const signed portCHAR *)"
           Client␣task", 4096, NULL, tskIDLE_PRIORITY, NULL);
}

void serverTask(void* pvParams) {
        /* Create socket without any socket options */
        csp_socket_t *sock = csp_socket(CSP_SO_NONE);

        /* Bind all ports to socket */
        csp_bind(sock, CSP_ANY);

        /* Create 10 connections backlog queue */
        csp_listen(sock, 10);

        /* Pointer to current connection and packet */
        csp_conn_t *conn;
        csp_packet_t *packet;

        /* Process incoming connections */
        while (1) {

                /* Wait for connection, 10000 ms timeout */
                if ((conn = csp_accept(sock, 10000)) == NULL) {
                        printf("Timeout!");
                        continue;
                }

                /* Read packets. Timout is 100 ms */
                while ((packet = csp_read(conn, 100)) != NULL)
                    {
                        switch (csp_conn_dport(conn)) {
                        case MY_PORT:
```

```
                /* Process packet here */

        printf("Packet␣received␣on␣MY_PORT:␣%s\r\n", (
          char *) packet->data);


                csp_buffer_free(packet);

            default:
                /* Let the service handler
                  reply pings, buffer use, etc
                  . */
                csp_service_handler(conn,
                  packet);
                break;
            }
        }

        /* Close current connection, and handle next */
        csp_close(conn);

    }
}

void clientTask(void* pvParams) {
    csp_packet_t * packet;
    csp_conn_t * conn;
    while (1) {

        /**
         * Try ping
         */

        int result = csp_ping(MY_ADDRESS, 100, 100,
          CSP_O_NONE);
        printf("Ping␣result␣%d␣[ms]\r\n", result);



        /**
         * Try data packet to server
         */

        /* Get packet buffer for data */
        packet = csp_buffer_get(100);
        if (packet == NULL) {
            /* Could not get buffer element */
        printf("Failed␣to␣get␣buffer␣element\n");
            return;
```

```
                }

                /* Connect to host HOST, port PORT with regular
                    UDP-like protocol and 1000 ms timeout */
                conn = csp_connect(CSP_PRIO_NORM, MY_ADDRESS,
                   MY_PORT, 1000, CSP_O_NONE);
                if (conn == NULL) {
                        /* Connect failed */
                printf("Connection␣failed\n");

                        /* Remember to free packet buffer */
                        csp_buffer_free(packet);
                        return;
                }

                /* Copy dummy data to packet */
                char *msg = "Hello␣World";
                strcpy((char *) packet->data, msg);

                /* Set packet length */
                packet->length = strlen(msg);

                /* Send packet */
                if (!csp_send(conn, packet, 1000)) {
                        /* Send failed */
                printf("Send␣failed\n");

                        csp_buffer_free(packet);
                }

                /* Close connection */
                csp_close(conn);

        }

}
#endif

void printTask(void* pvParameters) {
        portTickType xLastWakeTime;
        xLastWakeTime = xTaskGetTickCount();
        int counter = 0;

        while(1) {
                xSemaphoreTake(prvScreenMutex, portMAX_DELAY);
                printf("Hello,␣world!␣%d\r\n", counter++);
                xSemaphoreGive(prvScreenMutex);
                vTaskDelayUntil(&xLastWakeTime, 10000);
        }
```

```
}

void readTask(void* pvParameters) {
        portTickType xLastWakeTime;
        xLastWakeTime = xTaskGetTickCount();

        while(1) {
                xSemaphoreTake(prvScreenMutex, portMAX_DELAY);
                printf("Hello,␣other␣world!\r\n");
                xSemaphoreGive(prvScreenMutex);
                vTaskDelayUntil(&xLastWakeTime, 900);
        }
}
```

device_cdc_task

This file contains a FreeRTOS task. This task handles input and output specific to the USB CDC device class (but not the actual USB request handling). This snippet contains an ugly hack that basically waits until ten seconds have passed before consuming data from the output buffer. This makes it possible for the host computer to connect to the OBC and still be able to read the first messages that was printed.

Listing A.12: src/device_cdcs_task.c

```
#include "FreeRTOS.h"
#include "conf_usb.h"
#include "usb_standard_request.h"
#include "device_cdc_task.h"
#include "uart_usb_lib.h"
#include "task.h"
#include <io/io.h>

static volatile uint16_t  sof_cnt;

static void prvFlushData(void);
static void prvReadInput(void);
static void prvWriteOutput(void);

void device_cdc_task_init()
{
  sof_cnt = 0 ;
  uart_usb_init();

  xTaskCreate(device_cdc_task,
              configTSK_USB_DCDC_NAME,
              configTSK_USB_DCDC_STACK_SIZE,
              NULL,
              configTSK_USB_DCDC_PRIORITY,
              NULL);
}

static portTickType ticksSinceEnumerated = 0;


void device_cdc_task(void *pvParameters)
{
  portTickType xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();

  while (true)
  {
    vTaskDelayUntil(&xLastWakeTime, configTSK_USB_DCDC_PERIOD);

    if (!Is_device_enumerated()) continue;

        if( ticksSinceEnumerated == 0 ) {
                ticksSinceEnumerated = xTaskGetTickCount();
        } else {
                portTickType diff = xTaskGetTickCount() -
                    ticksSinceEnumerated;
                if( (diff/portTICK_RATE_MS) > 10000 ) {
                        prvFlushData();
                        prvWriteOutput();
        prvReadInput();

                }
        }


  }
}

static void prvFlushData(void) {
        if( sof_cnt>=NB_MS_BEFORE_FLUSH )  //Flush buffer in
            Timeout
    {
      sof_cnt=0;
      uart_usb_flush();
    }
}

static void prvWriteOutput(void) {
        static portCHAR queueValue;

        if(writeQueueHandle == NULL) {
                return;
            }

            while( xQueueReceive( writeQueueHandle, &queueValue, (
                portTickType ) 0 ) == pdTRUE )
        {
            uart_usb_putchar(queueValue);
        }
}

static void prvReadInput(void) {
            static uint8_t c;

            if( readQueueHandle == NULL ) {
                return;
            }

            if (uart_usb_test_hit())                    // Something
                received from the USB ?
        {
                c = uart_usb_getchar();

                if( xQueueSendToBack(readQueueHandle, (void*) &
                    c, 0) != pdTRUE ) {
                        return;
                }
        }
}

//!
//! @brief usb_sof_action
//!
//! This function increments the sof_cnt counter each time
//! the USB Start-of-Frame interrupt subroutine is executed (1
   ms).
//! Useful to manage time delays
//!
void usb_sof_action(void)
{
  sof_cnt++;
}
```

write

The C library calls the write function when writing data to a file descriptor. In this case, we only support the file descriptors for standard output and standard error.

Listing A.13: src/lib/io/read.c

```
/*
 * read.c
 *
 * Created: 05.12.2011 11:17:18
 *  Author: Dan Erik
 */
#include <FreeRTOS.h>
#include <queue.h>
#include <unistd.h>
#include <io/io.h>
#include <config/conf_io.h>

int _read (int file, char * ptr, int len);

int _read (int file, char * ptr, int len)
{
        int nChars = 0;

        if (file != STDIN_FILENO)
        return -1;

        if( readQueueHandle == NULL )
                return -1;

  for (; len > 0; --len)
  {
          portCHAR c;
          if( xQueueReceive( readQueueHandle, &c, (portTickType
              ) IO_READ_TIMEOUT ) != pdTRUE )
                break;

    *ptr++ = c;
    ++nChars;
  }

  return nChars;
}
```

A.7 FreeRTOS on the OBC

Introduction

This section describes the AVR Studio 5 project setup for the OBC.

Starting from an example project

Create new example project from AVR Studio 5. The example chosen was "USB CDC Example (from ASF v1) - EVK1104 - AT32UC3A3256".

In the example project, a USB interface is set up as either host, device or both. When in device mode, both the USART and the USB interface is used. Traffic sent to one of them, is forwarded to the other.

The example is designed for the evaluation board called EVK1104. This won't work directly with the OBC design, but is a good basis for writing the logic in a USB CDC device. In addition, the example can use FreeRTOS. And it is this use-case, that is most interesting. In addition, the example project comes bundled with drivers for the USB interface. By using them we don't have to:

- Write a USB stack as implemented on a USB device
- Implement USB CDC Device class specific logic
- Write USB drivers for FreeRTOS

Now, do the following:

- Remove src/host_cdc_task.c & src/host_cdc_task.h. These files implement host specific logic for USB CDC Device class.
- Remove src/asf/avr32/drivers/intc/exception.S. FreeRTOS defines its own exception vector, and defining them twice is going to become a problem when linking the binary.
- Remove src/asf/avr32/boards & src/asf/avr32/components. These folders contain initialisation code and defines for EVK1104, as well as a joystick driver. These folders are also removed from the include directories
- Change -DBOARD=EVK1104 to -DBOARD=USER_BOARD and add -DFREERTOS_USED in project settings.

Then adjust the rest of the project according to preference.