NTNU
Det skapende universitet

Databussen i en studentsatellitt

Dan Erik Holmstrøm

Master i datateknikk

Innlevert:	Juni 2012
Hovedveileder:	Gunnar Tufte, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

The Internal Data Bus in a Student Satellite

by

Dan Erik Holmstrøm

June 2012

Master of Science in Computer Science

NTNU
Norwegian University of Science and Technology

Abstract

NUTS is a 2U CubeSat, scheduled for launch in 2014. NUTS is being developed at NTNU, and students take part in both the design and the construction of the satellite. Miniature satellites adhering to the Cubesat specification are often composed of separate modules. A data bus is common method for inter-module communication. But being a shared medium, traffic on data buses susceptible to disturbances from failing modules. NUTS uses I^2C as the bus type. The lack of centralised arbitration poses several challenges with respect fair use of bus time, fault resistance and error recovery.

This paper describes the process of developing a bus protocol for NUTS. The main goals of the NUTS bus protocol is to provide some extent of fairness between the modules, dictate a lower bound on data throughput given certain assumptions, as well as providing a useful abstraction to ease the implementation of higher-level logic. Several support functions are needed to make this possible. This includes defining a system architecture and develop a basic set of drivers required to communicate over an I^2C bus.

Each module participating in testing of the bus driver implementation, have USB interface connectors. This interface will be used to control the module or modules under test. If any latencies are to have guaranteed bounds, the latency in interrupt processing must be both predictable and constrained. The interrupt processing latency is measured to find out if this can be an issue.

To ease both testing and debugging, a USB interface was used to control the modules under testing. The existing USB stack was extended to provide a separate communications channel, making it possible to provide terminal services and data transmission features at the same time. A design for the NUTS bus protocol is also proposed. To aid the implementation of this protocol, an I^2C master driver has been developed. A design for an I^2C slave driver has also been suggested

We find that guaranteeing fair access to a I^2C bus requires special considerations in both transmitter and the receiver. Fairness is an issue that must be respected at system-level. Variations in the interrupt processing latency when using FreeRTOS suggests that there may be room for improvement.

Contents

List of Figures

List of Tables

Acronyms

ADCS	Attitude Determination and Control System. viii, 12–15, 17
AES	Advanced Encryption Standard. viii, 12
AIS	Automatic Identification System. viii
ANSAT	Norwegian Student Satellite Program. viii
Cal Poly	California Polytechnic State University. viii
CDHS	Command and Data Handling. viii
CDMS	Command and Data Monitoring System. viii
COTS	Commercial off-the-shelf. viii, 4, 11, 29
CSP	Cubesat Space Protocol. viii, 9, 13, 29, 30
EEPROM	electrically erasable programmable read-only memory. viii
EPS	Electrical Power Supply system. viii, 4, 12–14, 16, 24
GFSK	Gaussian Frequency-Shift Keying. viii, 11
GPS	Global Positioning System. viii, 11
GSE	Ground Support Equipment. viii, 13, 15
HiN	Narvik University College (Høgskolen i Narvik). viii
HMAC	Hash-based Message Authentication Code. viii, 16
ICE	In-Circuit Emulator. viii

ISA Industry Standard Architecture. viii, 4, 5

LEO Low Earth Orbit. viii, 7

MMU Memory Management Unit. viii

MPU Memory Protection Unit. viii

NAROM Nasjonalt senter for romrelatert opplæring. viii, 9

NUTS NTNU Test Satellite. viii, 3, 6–9, 11–13, 15, 16, 23, 24, 26, 29

OBC On-board Computing. vii, viii, 8, 9, 12–14, 23, 28, 29, 37

OBDH On-board Data Handling. viii, 13, 14

OS Operating System. viii, 13, 30

OTP One-Time Programmable. viii

P-POD Poly-PicoSatellite Orbital Deployer. viii, 3, 4

PCB Printed Circuit Board. viii, 4, 13, 24

SPI Serial Peripheral Interface. viii, 13, 30

SRAM Static Random Access Memory. viii, 9, 29

TMR Triple Modular Redundancy. viii

TT&C Telemtry, Tracking and Command. viii, 12, 13

UHF Ultra-high Frequency. viii, 11

UiO University in Oslo. viii, 11

USB Universal Serial Bus. viii, 17

VHF Very High Frequency. viii, 16

Part I

Overview and planning

Project overview 1

Introduction

Before diving into specifics, it is beneficial to understand the framework conditions for both the satellite project and the specific problems addressed in this thesis. The purpose of this chapter is to do just that, in addition to give a short summary of the status of the NTNU Test Satellite (NUTS) project.

The NUTS project is a project whose purpose is to build, design and launch a 2U CubeSat into Low Earth Orbit (LEO)-orbit. It is expected to launch by 2014, with a engineering model completed in June/July 2012. NUTS is part of the Norwegian Student Satellite Project, also known as ANSAT. The ANSAT programme was launched in 2006 [21] as a collaborative effort between educational institutions and related industries in Norway. Its ultimate goal is to build three CubeSats, and launch them into space by 2014. These three satellites are NUTS, CUBESTAR and Hincube. This involves at least three educational institutions: Narvik University College, University of Oslo and NTNU.

The primary focus will be on miniaturised satellites, more specifically picosatellites adhering to the CubeSat specification. Introductorily the CubeSat specification is described, including the availability of Commercial off-the-shelf (COTS) hardware that adhere to this specification. Following that, is a brief introduction to the background for this project and the main features of the NUTS project. This includes an overview over its other Norwegian peers. At last, the main topic of this thesis is described, followed by a very short description of each chapter in this report.

1.1 CubeSat specification

Developed by California Polytechnic State University and Stanford University, the specification provides a set of mechanical, electrical, operational and other requirements [6] for small satellites. This includes specifications for the orbital deployment mechanism, access ports that enable last-minute testing and procedures for testing and integration. CubeSats comes in standard sizes or ”units”, ranging from 1U up to 3U. A 1U CubeSat measures $10x10x10cm$ and has a maximum mass of $1kg$[18]. The outer dimensions of a 1U CubeSat is shown in figure 1.1.

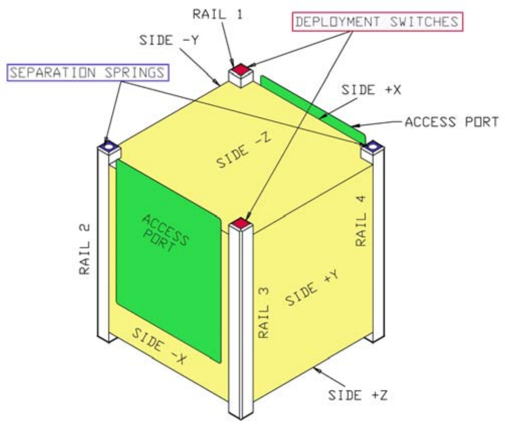


Figure 1.1: The CubeSat specification illustrated for a 1U CubeSat

Source: Nugent et al. [18]

CubeSats are launched into space by piggybacking on a rocket. When the rocket has reached the target altitude, it is deployed from a mechanism called a Poly-PicoSatellite Orbital Deployer (P-POD) like the one shown in figure 1.2. A CubeSat has separation springs that helps in separating the CubeSats after deployment from the P-POD. The CubeSats are completely powered off as long as the deployment switches are depressed. These switches are seen in figure 1.1.

The standardised dimensions and mechanical features of compliant CubeSats has led to the development and availability of COTS components [12][5][15]. It is even possible to buy complete ”CubeSat-kits”, where the customer only have to add payload, an Electrical Power Supply system (EPS), radio transceiver and antenna in order to have a satellite ready for deployment [7] — or solutions where almost all nec-

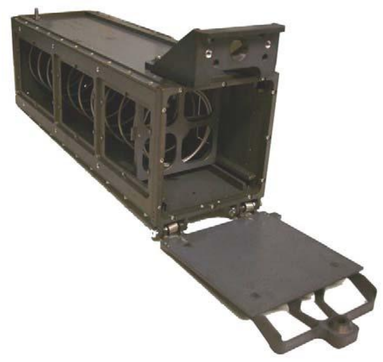


Figure 1.2: The CubeSat P-Pod Mk3. Holds up to 3U CubeSats in place inside the launch vehicle.

Source: Nugent et al. [18]

essary subsystems are implemented on a single Printed Circuit Board (PCB)[23].

CubeSat module connections

The CubeSat design specification focuses on structural properties and elements related to launch safety [6, p. 11], while everything else is left out. This is enough for hardware vendors to provide certain components like solar panels and battery packs, but the requirements in the specification do not dictate mechanical or electrical properties of modules PCB. Instead, the use of the CubeSat Kit Bus connector with a PC/104-compliant form-factor has become widespread for COTS-modules for CubeSats.

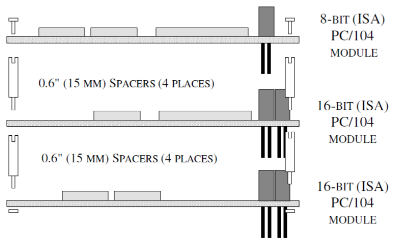

8-BIT (ISA) PC/104 MODULE
0.6" (15 MM) SPACERS (4 PLACES)
16-BIT (ISA) PC/104 MODULE
0.6" (15 MM) SPACERS (4 PLACES)
16-BIT (ISA) PC/104 MODULE

Figure 1.3: PC/104 are stacked on top of each other. This is possible through the use of stack-through connectors.

Source: **PC/104 Specification** *[20, p. 4]*

The PC/104 standard provide a more rugged and compact version of the PC/AT bus. It is based on the IEEE P996.1 draft specification [20, p. iii], of what has become known as the Industry Standard Architecture (ISA) bus standard [1]. Thus, PC/104 defines both a form factor and a data bus. The data bus type is ISA with some significant changes [20, p. 3]:

- Each ”expansion card” has a physical size of 90 by 96 mm.
- Reduced bus drive for most signals to 4mA.
- The cards on the bs are stacked with pin- and socket-connectors, as shown in figure 1.3. Eliminates the need for a backplane.
- Each card has standoffs in each corner, fastening them to neighbouring cards. This improves mechanical stability for a stack of cards.

While there is no need for a backplane, it is common to have one of the cards in a stack act like a mainboard. This mainboard would contain a CPU that would act like a peripheral controller — as well as a bus arbiter — if needed.

The PC/104 standard has become popular in CubeSats. This may be attributed to the CubeSat Kit-product. Specifically, the CubeSat Kit Bus connector is compatible with PC/104 [8, p. 17], meaning it conforms to the non-optional parts of the mechanical and electrical specifications in the PC/104-standard. Only a subset of the pins are implemented on the CubeSat Kit-motherboard; for power supply, only $5V$ and GND are provided. In addition, the CubeSat Kit does not use ISA as the data bus type. Instead, the pins are used for I^2C, SPI and user-defined purposes (general I/O). The PC/104 form factor with the CubeSat Kit Bus is by far the most commonly used for COTS parts. If a CubeSat project is to use any COTS-modules, it is difficult if not impossible to avoid CubeSat Kit and CubeSat Kit-like products.

1.2 Project mandate

The purpose of this project is to utilise the internal data bus of the NUTS satellite. The work in this thesis is a continuation of work done by the

[1] While literature on ISA refers to IEEE P996, the specification was never completed. This means that there may be minor variations in interpretation of signals and timings between different PC chipsets

author during the TDT4501 - Computer Science, Specialization Project last semester. There, the purpose was to develop identify overall software requirements and investigate means for inter-module and satellite-ground-communication.

The work done last semester has paved the way for developing this concept further, and to include the internal data bus. This means that any software developed has to both implement some kind of bus protocol, and be compatible with the existing software on the NUTS satellite.

1.3 Project background

The ANSAT programme was not the first of its kind in Norway. The predecessors to the three satellites in the ANSAT programme, is NCUBE-1 and NCUBE-2. With the goal of stimulating cooperation between Norwegian educational institution, their main mission was to communicate with amateur ground stations and deploy AIS (Automatic Identification System)-receivers[2] into space[11]. The AIS-receivers would have been used to track ships and reindeers equipped with AIS-transponders.

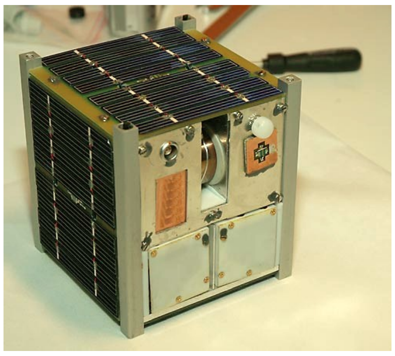


Figure 1.4: The NCUBE-2 student satellite

Source: **nCube (satellite)** ***[17]***

NCUBE-1 and NCUBE-2 were like each other. However both missions were failures in the sense that they never succeded in being sucessfully launched into orbit. NCUBE-1 was destroyed when its carrier vehicle – a Dnepr rocket – failed during launch[4]. NCUBE-2 was launched into orbit, but it was not possible to establish contact with the satellite.

[2] Automated tracking system, mainly used for tracking ships at sea. The tracked vessel uses a VHF transponder to send information about its identity, position, cargo, speed etc.

1.4 NTNU Test Satellite

Unlike most CubeSat projects, NUTS has a backplane for connecting module cards. For research purposes, it is interesting to try something different. But the rationale behind the having a backplane is the added flexibility it gives: In addition to having a satellite platform that can accommodate a variety of payloads, the use of a custom design enables improvements in the data and power buses [3]. The most important improvement is the ability to isolate modules from the rest of the system. This includes removing the module from the bus and disabling the module, effectively cutting it completely off the backplane. Also, had the traditional stacking approach been used, the logic in the NUTS backplane would have to be integrated into each module card. This would make it more difficult to implement some of the safety features stated in the requirements for the NUTS backplane [10]: That a single failing module should not be able to bring the whole system down. The caveat is that the use of a "non-standard"-backplane with custom connectors, makes it difficult to use any third-party modules if need arise.

The NUTS payload is an infrared camera. This camera will be used to observe a phenomenon called gravity waves. When air interacts with the earth terrain and in weather, then some of the air propagates into an OH-layer in the middle of the atmosphere. This layer emits shortwave infrared radiation, and perturbations from rising air flows makes it possible to observe wave patterns in the intensity of this radiation. This is called gravity waves.

Camera paylaods are common in Cubesats. What makes NUTS different, is the use of an infrared camera payload. The observation of gravity waves is made possible by using a InGaAs-sensor (Indium Gallium Arsenide).

1.5 Problem description

The NUTS backplane is the main mean of inter-module communication in the satellite. Being a shared medium by definition, any data bus is susceptible to interference patterns introduced in the case where multiple bus users want to utilise the bus at the same time. Being a relatively low-bandwidth data bus without any advanced arbitration scheme, I^2C-buses are particularly sensitive to these problems.

The main topic of this thesis is to investigate possibilities to guarantee bus fairness and possibilities with respect to fault tolerance on the

NUTS data bus. The goal is to develop a bus protocol and implement this in a portable software library.

One of the main issues encountered when developing software, is related to testing for correctness. In order to facilitate testing, the modules used during testing will use USB connectors to enable direct control of them.

Part of the assignment is to design the software architecture for the two master modules in the system: The radio module and the OBC-module. This will form the foundation of the whole system as a single entity.

1.6 Stakeholders

The project is sponsored by Nasjonalt senter for romrelatert opplæring (NAROM), the NTNU Department of Electronics and Telecommunications, in addition to the NTNU Department of Engineering Cybernetics. As the financing institutions, they have a vested interest in the success of the project. However, they are not directly involved in the decisions taken as part of the work on this thesis. The project leader, Roger Birkeland, is involved throughout the whole project. This makes him the ultimate decision-maker for all project work. Also, the students working on the projct are stakeholders. However at this point the work of each individual student is not tied to the rest of the individuals doing work on the NUTS-project.

1.7 Related work

I^2C is a popular bus type in CubeSats, but most projects rely on lax or on-existing latency-requirements and sheer bandwidth. In these situations, high utilisation of the data bus and bus fairness is not a problem in practice. Therefore, it is hard to find similar CubeSat projects where the issues addressed in this thesis, have been solved or evaluated.

1.8 Previous work

Hardware-wise, the OBC-module, the radio module and the backplane are more or less completed. FreeRTOS has been chosen as the operating system on the OBC-module, and a communications library called Cubesat Space Protocol (CSP) has been ported to Windows Vista, and verified to work on the OBC-module.

The designer of the OBC-module developed various drivers as part of testing the OBC module [30, p. 35]. This includes, but may not be exclusive to:

- Configuring the static memory controller for using external Static Random Access Memory (SRAM).
- Partial implementation of programming with JTAG by bit-banging designated JTAG-pins.
- Configuration of the flash memory controller.

This code has not been modified to work with FreeRTOS. It remains to be seen how much of this can be used further on in the project.

Background 2

Introduction

This chapter begins by introducing the two other satellites in the ANSAT programme, with comparisons with the NUTS-project where it is possible. Following this, comes a detailed overview over the current project status for NUTS. After that the NUTS backplane and OBC module is described in detail. The current state of the existing software used in the NUTS-project is also included.

Wrapping up the chapter, is a detailed discussion of I^2 and the Universal Serial Bus (USB) standard. The focus in these sections, is on details that are relevant to the material that follows the main part in this report.

2.1 CubeSats in the ANSAT programme

This section presents Cubestar and HiNCube. They share some features with NUTS, but are also quite different from other aspects.

CubeSTAR

Being developed at University in Oslo (UiO), Cubestar is a 2U cubesat due for launch in 2013 into low polar orbit. Its payload serves as a technology demonstration of a new probe: A multiple-Needle-Langmuir Probe (m-NLP). The instrument is able to measure the density of electrons in the upper earth atmosphere. This can be leveraged to get a better understanding of ”space weather” and predict how this will affect orbiting radio transceivers, such as Global Positioning System (GPS).

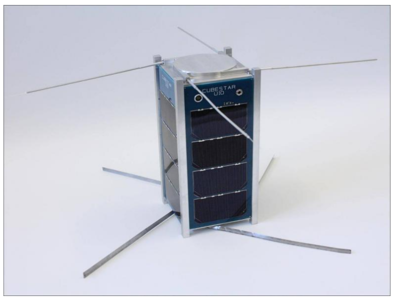


Figure 2.1: CubeSTAR

Source: **CubeSTAR** *[9]*

For communication, the CubeSTAR is fitted with a Ultra-high Frequency (UHF)-transceiver. It uses Gaussian Frequency-Shift Keying (GFSK)-modulation and the data link with a baudrate of 9600. AX.25 is used as the data framing protocol [13].

CubeSTAR does not use a COTS OS. Instead, they rely on developing their own systems software. Internal commands have set Hamming distance between them. When a module receives a command, it can then calculate the minimum Hamming distance between the received command, and the set of known commands. This scheme enables both error detection and correction.

To ensure system availability, the current plans are to use a combination heartbeat timers in the OBC- and Telemtry, Tracking and Command (TT&C)-module. If either fails to respond to the other within a reasonable timeframe, a module reset is initiated.

For computer hardware aboard the satellite, the main microcontrollers are Atmel ATmega128s. The ATmega128-series are 8-bit microcontrollers with limited processing capabilities. However, nothing aboard the CubeSTAR requires significant computational power. The most demanding computing application onboard, may be the encryption of downlink packets. But the microcontroller suppoorts instructions for performing encryption and decryption using Advanced Encryption Standard (AES).

When processing payload data, system integrity may be ensured by implementing triple modular redundancy.

Like NUTS, hardware-wise, the CubeSTAR is similar in that it has a backplane with separate modules for EPS , OBC, TT&C, payload and Attitude Determination and Control System (ADCS). I^2C is used as the internal data bus. But for added redundancy and throughput, it has two data buses.

2.2 HiNCube

HiNCube is a 1U Cubesat that is nearing completion. Its payload is simply a camera with no particular purpose than taking pictures and transmitting them to the ground station[19]. Part of the payload module is also logic for receiving telemetry data from eight thermal sensors situated in each corner of the Cubesat.

HiNCube has a ”traditional” Cubesat-design based on the PC/104 form-factor [1], with stacked module PCBs with a combination of modules boards developed and created by third parties, in addition to custom designed modules. In the case of HiNCube, the On-board Data Handling (OBDH)-module has been bought from Pumpkin Inc. The COMM- and EPS-modules provided by GomSpace. The software used on these modules is also developed and provided by GomSpace.

With the exception of the COMM and EPS-modules, the HiNCube systems software is custom designed. The internal communication protocol is CSP, with the CSP implementation library ported to the custom Operating System (OS) used on some of the modules. All modules expose their functionality through CSP services on the internal data bus. The COMM-module uses CCSDS Space data link protocol for the data link between ground and space.

During normal operation, the motherboard/OBDH-module takes the role as a command proxy, acting on the behalf of the Ground Support Equipment (GSE). This means that the radio forwards any commands from the ground segment, further to the OBDH.

The internal data bus uses I^2C in multi-master mode, but Serial Peripheral Interface (SPI) was used in earlier designs[14].

2.3 Cubesat satellite subsystems

A typical satellite has various subsystems. This is done on the basis of a functional decomposition into manageable units with separated responsibilities. The exact tasks that are assigned to each subsystem is highly specific to the mission, but in NUTS the subsystems are defined to be in one of the following categories:

- TT&C
- ADCS
- EPS

- OBC/OBDH
- Payload

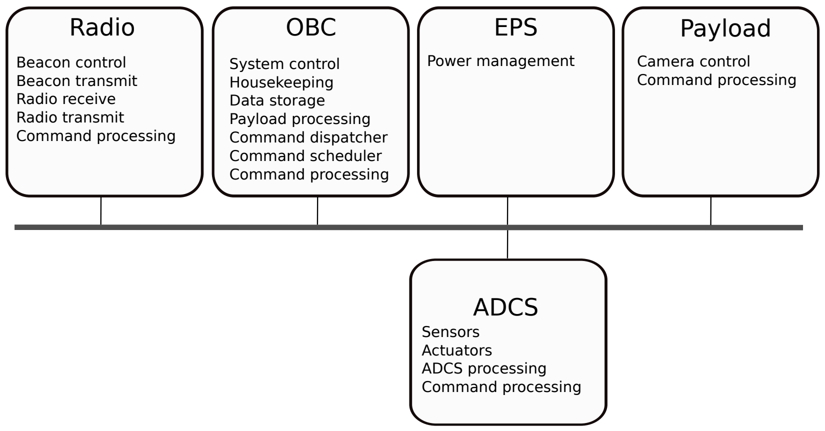


Figure 2.2: The NUTS subsystem with their delegated responsibilities

The TT&C-system is responsible for ground-satellite-communication. This involves receiving and responding to commands from the ground. Without the TT&C system, it is impossible to control the satellite.

The ADCS system provides the means to measure and change the orientation and altitude of the spacecraft. By using a model of the dynamics of the spacecraft, the ADCS-system can use actuators to counteract and interact with external forces from gravity and the earth magnetic field. Using the magnetic field of the earth for orientation is commonly used in small satellites, where the moment of intertia is small and there is no room for a propulsion system using compressed gas.

Providing power for the vehicle is the responsibility of the EPS system. Without a functioning power system that can provide reliable and stable power supply, the satellite is not of much use. The main power source for satellites orbiting the Earth is the combination of batteries and solar panels. During Sun eclipse the satellite is running on battery power, and the power budget may be more limited. The efficiency of both batteries and solar panels decrease over time, and the mission will essentially end when either of them fails. But the EPS-system may aid in increasing the life span, by preventing the batteries from overcharging and overheating.

Usually a satellite needs a computer for doing various computational tasks. This is typically related to processing of data from the payload-system, storage of data in non-volatile memory for later retrieval, periodic tasks such as logging of system status and general system health supervision. This responsibility lies with the OBC- or OBDH-system.

When all these assignments are put together, the end result is the complete system shown in figure 2.2.

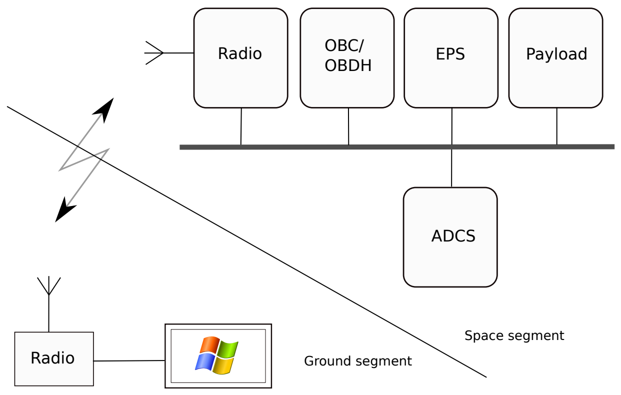


Figure 2.3: The defined subsystems in NUTS in the space segment, and the ground station in the ground segment

In addition to the satellite systems, there is a ground segment on earth. The ground segment contains the equipment necesarry to manage the mission, and process telemetry and payload data received from the satellite. An illustration of the whole system is shown in 2.3. The ground segment consists of at least a computer terminal and a radio connected to an antenna. For the purpose of understanding the system from a high level, this is as much details that are needed. But the satellite software does shape the software used on the ground station, because it is the ground station that must be capable of understanding data sent from the satellite and it must be able to issue commands that are understandable to the satellite software. However, this is not a topic in this report.

2.4 NUTS status and project timeline

Currently, there is about a dozen students working on NUTS as part of their master thesis. In addition, several students are working on the project as part of the course TFE4850 Experts in team. Starting at the end of April, there is also a group of volunteer students that will begin work on different aspects of the NUTS project. This includes three people that are assigned to work solely with software development, both GSE-software and software for embedded systems.

The NUTS backplane is more or less completed. However the design may be revised by removing superfluous board connectors and circuitry to enable the use of a larger-sized battery pack [16]. At the time of writing, there is also a minor issue with the per-module current limiter-logic. When this situation arises, the current limiters will continuously trip and reset with the result that the module is essentially left without power.

The ADCS system uses solar sensors, gyroscopes and magnetometers to determine the orientation of the satellite. The satellite will have solar panel on five sides, and intensity measured at each solar panel is used for sensing the relative direction to the sun. A new method for altitude estimation has been developed, tested and found to be reliable and accurate when compared to the well-known and commonly used "Extended Kalman Filtering". The method is a variant of QUEST (Quaternion Estimator), called EQUEST (Extended QUEST).

NUTS will use magnetorquers to change the orientation of the satellite. Magnetorquers are electromagnetic coils. By controlling the electric current through the coils, they are able to interact with the magnetic field of the Earth. The control algorithms used for stabilization and detumbling [1] are yet to be decided. But various controllers have been tested and developed for prototyping.

The main mechanical structure is responsible for holding the satellite components together. During launch, the satellite is exposed to major lateral forces, as well as excessive vibration. While the frame plays a minor role when the satellite is in orbit, it may provide some degree of protection from cosmic radiation. The thermal properties are also important, because heat dissipation can only happen through heat radiation.

The most common material used for the main structure is aluminium. The material of choice for the primary structure in NUTS is a composite material, composed of a carbon fiber fabric and an epoxy resin. While carbon fiber has been used in previous Cubesat missions, using an all-composite frame is not that common. Carbon fiber has several attractive properties such as good stiffness and strength, low weight and very low thermal expansion when compared to aluminium. A test frame has been manufactured. The ongoing work on the satellite structure, is development and manufacturing of a structure that attaches the satellite

[1] Detumbling is the process of reducing the angular velocity of the satellite after orbital insertion

components with the frame. There is also some remaining work related to testing, production and approval.

A security analysis has been performed [29]. The major threat was found to be the case where a malicious third party would be able to control the satellite, while payload confidentiality was not an issue. The encrption algorithm and the packet authentication mechanism in a communications library called CSP was also briefly evaluated. CSP uses SHA-1 for packet authentication using message authentication codes. This hashing algorithm has several weaknesses. Currently a new Hash-based Message Authentication Code (HMAC)[2] algorithm called BMW is being tested for use on the existing NUTS computer hardware. One of the key elements is benchmarking the algorithm when used on the actual hardware. There is also some issues related to how private keys are to be distributed, as well as the generation of suitable sequency numbers used to prevent replay attacks.

The EPS system is also being designed and tested. The EPS is comprised of the module board itself, in addition to a battery pack and solar panels. It provides the backplane with 3.3V and 5V power buses. For added reliability, the power buses are duplicated. This is to prevent mechanical failure in the backplane, from rendering all the modules powerless. The EPS module is expected to be finished by the end of june 2012.

While the payload is going to be an InGaAs camera, a suitable camera is yet to be found. Because of the limited downlink, some processing will also have to be done aboard the satellite. This includes compression and post-processing of the images taken by the camera. Because of the relatively long exposure times required to take good pictures, motion blur is also going to be an issue. These matters are currently being worked on.

The antenna system consists of a radio module with two Very High Frequency (VHF) transceivers. The modulation scheme is going to be FSK (Frequency Shift Keying) with carrier frequencies at 146MHz and 437MHz. Because the ADCS system may fail, the aim is to achieve close to isotropic radiation pattern for the antennas. The current design uses two crossed dipole atennas located in zenith and nadir direction, one for each of the frequencies. The satellite is also going to be equipped with a beacon transmitting morse code. The purpose of having a morse code

[2] Message authentication codes are used to verify the integrity and authenticity of messages sent from the satellite.

transmitter, is that it should always be present and active. Even if all the software on the satellite fails, the morse code beacon would still work.

2.5 NUTS system overview

This section gives a thorough description of the elements that are of direct relevance to the development of a bus protocol for NUTS. The key elements are the master modules and the backplane itself. The two master modules are able to supplement each other and enforce the bus protocol. The backplane provides the means to make this possible.

We begin by describing the NUTS backplane, by giving a overview over the features followed by an exhaustive description of the semantics of all of the control signals. After that, the OBC and the radio module is described. The OBC module is the main computer in the satellite. At last, the software that is currently used or implemented for NUTS is briefly touched upon.

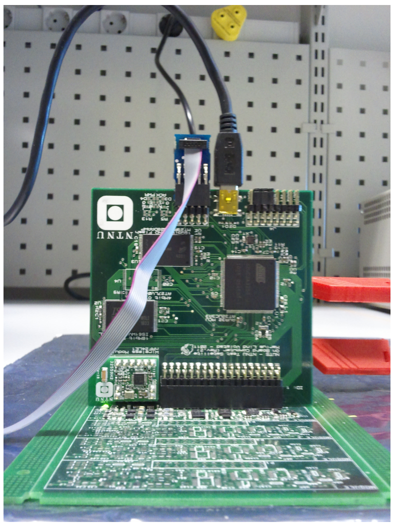


Figure 2.4: OBC setup for development. It sits in one of the master slots on the backplane.

NUTS backplane

The NUTS backplane is responsible for distributing power and provide access to a data bus. The backplane is the central PCB, into which modules are connected. The connectors on the backplane also gives each module additional mechanical stability.

Overview

Unlike a traditional motherboard in a desktop or laptop computer, the NUTS backplane does not contain any peripheral hardware or any electronics not related to generic control of backplane modules. Instead, the backplane basically provides a set of slots, power buses, a data bus and some control logic without the use of complex controllers. One of the design goals was that the backplane should operate without the use of software.

The data bus type is I2C. From a electronics design standpoint, the key advantages of using I2C is that no central arbitration logic is required and that only two bi-directional bus lines are needed for communication. The rest of the logic is implemented in each chip that wants to use the I2C bus. Support for I2C is also widespread.

The backplane has three types of connectors for different types of modules: Slaves, two masters and an EPS connector. The EPS socket must be different, because the EPS module supplies power to the power lines in the backplane, instead of drawing power from it. Master connectors has extra lines to control the backplane. This enables features that are not available from slave modules.

Each module is provided with dual 3.3V and 5V power lines. There are per-module power switches to disable the power supply to individual modules. In addition, each module has current monitors. The INA219 current monitor has an I2C interface, and they are connected to the backplane data bus as long as the module has not been isolated from the internal bus. This makes it possible to monitor both the voltage and current provided to each module.

Each module connector has bus isolation buffers/repeaters. This does not only make it possible for each side of the buffer to operate at different voltages: The NXP PCA9517 bus repeater used on the backplane has an input to enable or disable it. This input is controllable by backplane master modules. The bus isolation mechanism may be used by backplane masters to prevent other modules from erroneously occupying the bus for prolonged perionds of time. The use of bus repeaters also reduce bus capacitance.

Each power line supplied through each module connector, is protected by current-limiting switches. The current backplane design limits the current to each module to about 300mA [10, p.41]. If the load current to a module exceeds that, the current limiting switch will latch off for a brief period of time before turns on again. If the reason for the high current draw was a single event latchup or any transient event, the switch will not turn off again. A backplane master may see if the current limiting switches on either power line through one on the control lines made available to backplane masters.

I2C is a multi-master capable bus. The same semantics applies in the NUTS backplane. This means that any regardless of whether the module is a backplane master or not, it is still able to initiate communication to any other module connected to the backplane. The I2C bus supports fast mode, giving a theoretical throughput of 400kbit/s.

The state of the power switches and bus isolation buffers are stored in flip-flops. When the backplane is reset, the state of all power switches and bus repeaters defaults to *on*. The backplane has a reset monitor with a watchdog reset. The indended purpose of the backplane watchdog is twofold: a) During backplane power-up, the watchdog resets all modules to the default state, and b) If both master modules become disabled for some reason, the watchdog timer triggers (and reset all bus and power switches to ON). While there are reset lines going to all modules, a watchdog trigger does not assert this signal to the backplane modules.

Figure 2.5 shows a block diagram over the elements that constitute a slave module slot on the backplane. The master modules have access to an extra set of lines, making them able to control the logic for a specific module. Each module is also supplied with two power lines at 3.3V and 5V. Inside the power module dedicated to the connector, there is a circuit that does OR-ing of the power lines. This circuit could fail, but it would not affect the rest of the system. The power module also contains current limiters that serves as an extra safety mechanism if the module short-circuits the power lines.

The bus repeaters are located within the logic module shown in figure 2.5. Functionally, they act as bi-directional buffers that repeat the signal seen on one side, to the other side of the buffer.

The control logic in the backplane is controllable from either of the two master module slots. There is no arbitration on these lines. Instead, they use open-drain logic to eliminate the risk of damaging high currents if two masters try to use the control lines at the same time.

There is a risk that memory elements in the backplane may change state because of single event effects. A possible work-around is to have a master continuously set the correct state for each module.

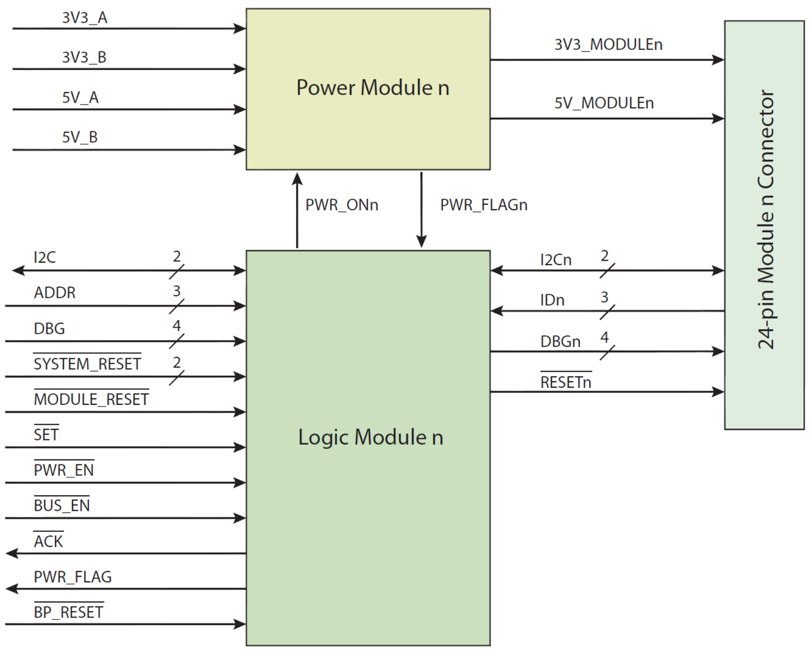


Figure 2.5: Backplane logic module

Source: [10]

Master module control signals

Module logic is controlled by a master by setting three ADDR lines (each module has three ID lines). If the module is present, the $\overline{ACK}$ signal is asserted. The master can then assert $\overline{PWR\_EN}$ and/or $\overline{BUS\_EN}$ and store the result into latches by pulling $\overline{SET}$ low. When a module is selected by having its ID match the current settings on the ADDR lines, four JTAG/SWD lines are connected through a bidirectional buffer. Also, when the module is selected they can be reset by pulling $\overline{MODULE\_RESET}$ low. The $\overline{PWR\_FLAG}$ signal can be read when a module is addressed, indicating whether there is an over-current condition at the addressed module.

Some signals control does not control a specific module. These signals are $\overline{BP\_RESET}$, $\overline{SYSTEM\_RESET1}$ and $\overline{SYSTEM\_RESET2}$.

There is no arbitration for the backplane control logic. To ensure that a master has exclusive access to the backplane, it can assert the address lines and then compare them with the expected values. If they match, then the master can proceed.

In summary, a backplane master has access to the following signals when addressing a module through the ADDR lines:

- $\overline{ACK}$: Indicates that the addressed module is present (or not)
- $\overline{PWR\_EN}$: Used to switch the module power switch on or off.
- $\overline{BUS\_EN}$: Used to enable or disable the module bus repater.
- $\overline{PWR\_FLAG}$: If asserted, it indicates that the current limiters have tripped.
- $\overline{MODULE\_RESET}$: Reset the module.
- $\overline{SET}$: Used to latch the module settings asserted on the other control lines.
- DBG_TMS, DBG_TDO, DBG_TCK, DBG_TDI: JTAG lines used to program the module.

When trying to force a badly behaving slave off the data bus by using the BUS_EN# signal, one must take som precautions. The datasheet for PCA9517 states that the enable pin must not change during I2C bus operations. Thus a badly behaved slave should be held in reset before disabling the bus repeater. The signals needed to achieve this is shown in figure 2.6.

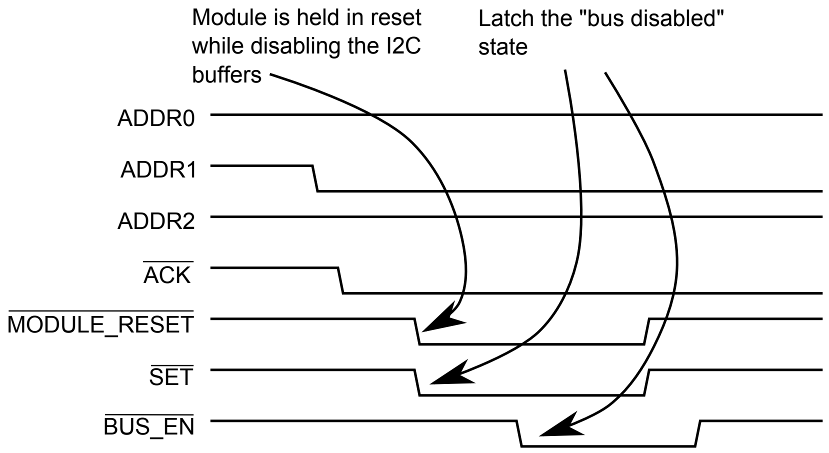

Module is held in reset while disabling the I2C buffers
Latch the "bus disabled" state
ADDR0
ADDR1
ADDR2
$\overline{ACK}$
$\overline{MODULE\_RESET}$
$\overline{SET}$
$\overline{BUS\_EN}$

Figure 2.6: Timing diagram for isolating a module from the bus

A master module is also able to program other modules. This procedure is encapsulated in the timing diagram shown in figure 2.7. This shows that the master receive a confirmation of the presence of the slave

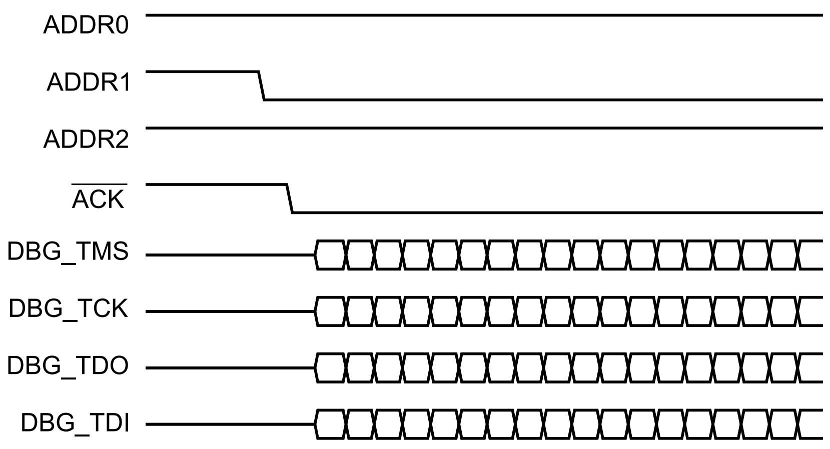
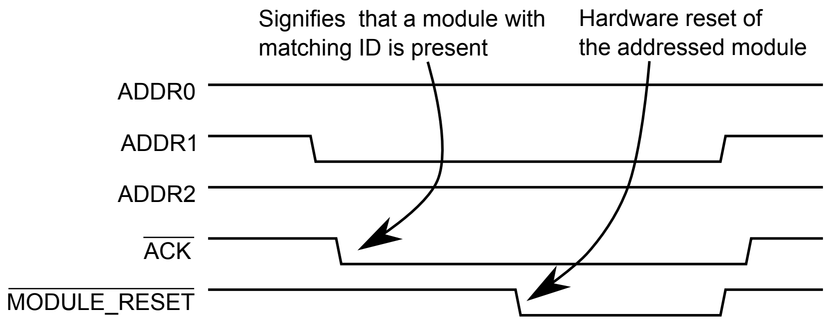


Figure 2.7: Timing diagram for programming a module

with address 101. The JTAG lines are routed to this module, and programming can commence.

A module can also be reset. The reset line is routed into the addressed module. In a properly designed slave module, pulling the reset line low should perform a hardware reset. The signals asserted by a master module that is to reset slave with address 101 is shown in figure 2.8.


Signifies that a module with matching ID is present
Hardware reset of the addressed module
ADDR0
ADDR1
ADDR2
$\overline{\text{ACK}}$
$\overline{\text{MODULE\_RESET}}$
Figure 2.8: Timing diagram for resetting a module

A slave can also be completely disabled. This disabled state is stored in the logic module shown in figure 2.5. The corresponding timing diagram is shown in figure 2.9.

The following signals are not specific to any addressed module:

- $\overline{BP\_RESET}$: Reset the backplane
- $\overline{SYSTEM\_RESET1}$ and $\overline{SYSTEM\_RESET2}$: System reset. This resets all modules except the initiating master module.

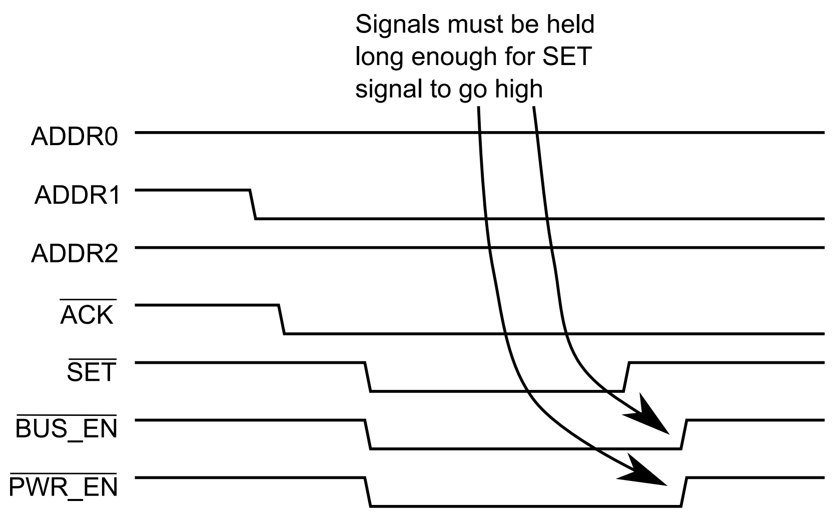

Signals must be held long enough for SET signal to go high
ADDR0
ADDR1
ADDR2
$\overline{\text{ACK}}$
$\overline{\text{SET}}$
$\overline{\text{BUS\_EN}}$
$\overline{\text{PWR\_EN}}$

Figure 2.9: Timing diagram for disabling a module

These lines are always available, and no kind of arbitration mechanism is necessary for them to work properly.

NUTS OBC

The OBC module is the main computer in the satellite. It is responsible for the supervision of system health, logging of flight data and the control of other modules through commands sent over the internal data bus. The OBC module is also the only module that has extra external memory.

The different memory types are shown in figure 2.10. The module also has a 4Mb OTP (One-time Programmable) EPROM (Eraseable Programmable Read Only Memory). While it is not radiation hardened, the OTP memory is likely to be more resilient memory corruption because of radiation. This makes it suitable for storing system software.

The CPU on the OBC module is an Atmel AVR32UC3A3256. It is a modern 32-bit microcontroller that targets cost-sensitive, low power applications with need for processing power. The CPU core does not have a MMU (memory management unit), so hardware support for memory protection is limited. However it does have a MPU (memory protection unit). The MPU makes it possible to set protection bits on memory segments, although there is no support for paging.

The 16Gb NAND flash is used for persistent storage of various data. However, the software for realising this has not been developed. It is clear that some means of storing and retrieving information from the

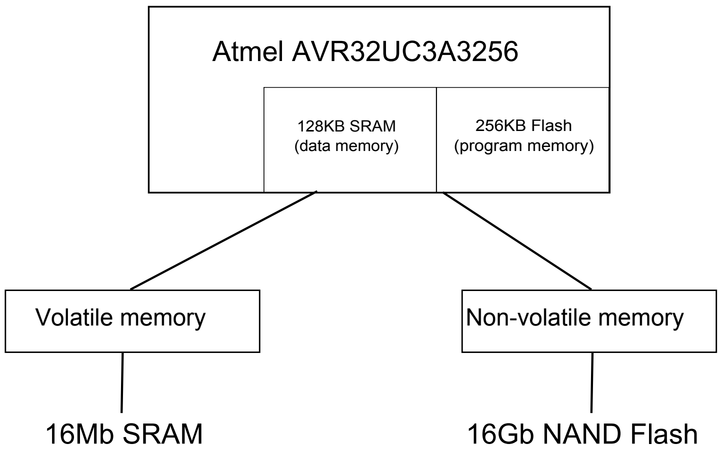


Figure 2.10: Hierarchic view of the different types of memories available on the OBC module.

flash memory is needed. What is not clear, is what kind of file system satisfies the need to guarantee the integrity of the stored data.

To supplement the meager amount of RAM integrated into the MCU package, the OBC has 16Mb of SRAM. SRAM is expensive and more DRAM (Dynamic RAM) can be had for the same price and same physical size. However DRAM is more complex to interface with, requiring periodic refreshes, higher access latency than SRAM and more complex addressing to the the multiplexing of row and column addresses on the same address lines. The access latency to SRAM is also better.

The general physical structure of the OBC module is shown in 2.11. What is missing from the diagram, is a UHF(Ultra High Frequency)-transceiver meant to be used in a wireless bus experiment. The state of this wireless bus is not known, and the transceivers have never been tested on the OBC module itself.

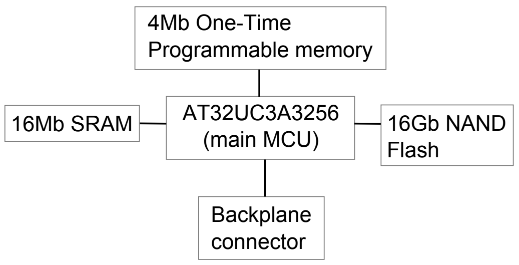


Figure 2.11: Block view of the OBC module

The OBC module uses FreeRTOS as the operating system. To ease communication internally in the satellite, and between the satellite and the ground station, it uses a communications/networking library called CSP.

During normal operation, the OBC module shall perform mission critical tasks. In addition, it will act on behalf of the ground station. Any commands sent from the ground is forwarded to the OBC module. The OBC is also responsible for executing tasks that are scheduled for later execution.

NUTS Radio

The radio module is similar to the OBC module hardware-wise. Unlike the OBC, it does not have any One-Time Programmable (OTP)-memory or external flash memory for persistent storage. Currently, no software has been tested on the radio module and the state of the hardware remains unknown. It is very likely that the basic software on the OBC module, including the OS will work without any complications on the radio module.

Existing software for NUTS

In order to be able to fulfill the needs of the project, the ground station software is going to be a combination of COTS and/or freely available software and custom developed software. The ground support equipment software for NUTS is currently being developed. At this point, the first steps is to write a software requirements specification and determine how to best make use of the available man-power.

FreeRTOS is a GPL licensed operating systems that targets microcontrollers. It is a real-time operating system, meaning that the CPU scheduler satifies real-time requirements. It supports various degrees of multitasking, ranging from a preemptive sheduler and co-operative multitasking to co-routines. The latter is suited for environments where memory is *extremely* constrained.

FreeRTOS could be considered to be an operating system kernel, with various utility functions. There is no driver model, support for file-systems or any other inherent hardware support except for the processors that it is ported to. But it still gives a foundation for a hardware abstraction. Also, using a multitasking OS scales better than using a simple control loop when the specific computational processes are not known beforehand. And even then, the OS may prove to be the best alternative.

FreeRTOS is not the only operating system that targets microcontrollers and embedded systems. There are several examples of RTOS, such as VxWorks, Windows CE, RT-Linux, ECOS, QNX and Integrity. The main reason that FreeRTOS was chosen was because it was already ported to the AVR32UC3.

Developing a library for general internal communication, and communication between the satellite and the ground would be time-consuming. Fortunately, there was a suitable library available under a free license: CSP. The library still has to be augmented to fit into the specific scenario, but it saves development time to use software that already implements some of the features that are needed.

2.6 USB standard

The USB (Universal Serial Bus) standard is an industry standard that describes an serial bus interface for extending a host computer with peripheral devices. The motivation for developing such a standard was to replace a mixture of different serial and parallell connectors, and various interfaces such as SCSI, RS-232 and Centronics with a single unified interface. The main goal of the USB standard, is to have a single, versatile specification that can be used by different vendors to design and build hardware devices that are compatible with each other.

The USB specification defines not only a communication protocol, but also programming interfaces required by devices and USB host systems, electrical and mechanical properties and the bus protocol[27]. USB 2.0 defines three fundamental transfer rates called low, full and high speed at 480Mbps, 12Mbps and 1.5Mbps respectively. USB 3.0 adds a new transfer type named SuperSpeed. SuperSpeed mode enables transmission rates up to 5Gbps[24].

USB devices are divided into classes, after their capabilities. There are classes for mass storage devices, human interface devices (HID) and communication devices (CDC). They are identified by special class codes, and they aid the host in finding a driver. Often, a single driver can support devices from different vendors.

The most recent revision of the standard is USB 3.0. This replaces previous revisions, but remains backwards compatible with USB 2.0. It uses some of the same concepts as the previous revisions of the standard, but adds a second bus to enable higher speed. The dual-bus nature also makes it a bit different, and therefore any discussion of USB 3.0-specific features are omitted.

USB is a cable bus where peripheral devices are connected in a tiered star topology. The center of each star is a hub. A hub is connected to another USB device, which is either a function or an another hub. When devices are connected to hubs, they add a new tier. Up to five hubs may be connected to each other. At the top tier is the root hub. The root hub is integrated into a host system, and the USB interface to the host computer is referred to as a host controller.

USB is a polled bus where the host controller initiates all transfers. The only exception to this rule is that devices are allowed to notify the host when they wake up from power-saving modes. Normally, the host sends a special SOF (Start Of Frame) packet or applies a keep alive signal at regular intervals. If no such event take place for 3ms, a device is effectively commanded to suspend itself.

These keep-alive signals establish a common time frame that is used as basis for transactions to an endpoint. For a low- or full-speed bus the time base is 1ms, and for a high-speed bus there is a 125μs time base. These are called frames and microframes respectively. As an example, the transfer types isochronous and interrupt are given access to the bus every Nth frame/microframe, where N is an attribute of the endpoint. The polling interval for the endpoint is expressed in frames instead of an absolute time period.

When a device is connected to a host, one of the first steps is that the host applies a reset condition. When a device is reset, it returns to a unconfigured state. This is a way for the host to put the device in a known state, from which further configuration can happen.

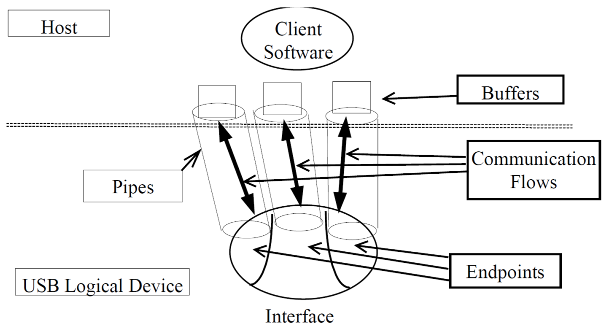


Figure 2.12: Data flow between the endpoints in an interface to buffers on the USB host

Source: **USB 2.0 specification**

Communication between a host and a device happens through pipes.

As shown in figure 2.12, a pipe is a logical connection between software on the host and an endpoint on a device. An endpoint is either a source or a consumer of data, and it is uniquely identified by the device address, a 4-bit endpoint address and a direction that is either IN or OUT. The direction refers to the flow of data with respect to the host, meaning that an IN endpoint always sends data upstream, towards the host.

Client software on the host sends and receives data through pipes. A pipe has several parameters associated with them:

- Allocated bandwidth and bus access
- Characteristics from the endpoints associated with the pipe (direction and maximum packet payload sizes)
- Transfer type, which is either isochronous, bulk, interrupt or control
- A type, which is either stream or message

Data sent on stream pipes are one-way and have no USB-defined format. Message pipes allow data to be transferred in both directions, and the data must have a USB-defined structure. These pipes are always used for control transfers. The default pipe consists of the two endpoints with address zero. It is always available after a device has received a bus reset. The default pipe is used by the host software to identify the device, get configuration requirements from the device and finally configure the device.

Interrupt transfers have a guaranteed latency, and are typically used to notify the host that some non-periodic event has happened on the device. Because the device can not notify the host itself, the device must wait until the next polling interval to notify the host of the occurence (or non-occurence) of an event. For instance, for IN transactions a interrupt endpoint may notify the host that it has buffered a character and that the host may retrieve this by doing a bulk transfer to a different endpoint.

Control transfers are typically used to send commands to a device, and query for device status. These kind of transfers always happen over message pipes. Control transfers are used for the initial configuration of the device over the default control pipe, but after configuration control transfers can be used on either the default control pipe or other pipes for device- or device class-specific control. Being carried through message pipes, means that the payload has a USB defined structure. Chapter 9 of the USB specification[27] defines standard, class-specific or vendor-specific requests that can be used to change the state of the device. Being carried through message pipes, means that the payload has a USB defined structure. Chapter 9 of the USB specification[27] defines standard, class-specific or vendor-specific requests that can be used to change the state of the device.

USB devices has a number of descriptors. Descriptors are simply data structures with a defined format. They are used to describe what a device can do and what it requires from the host. Their contents form the basis for what kind of commands a host can send to a device. The descriptors are organised in an hierarchy of descriptors. This hierarchy is shown in 2.13.

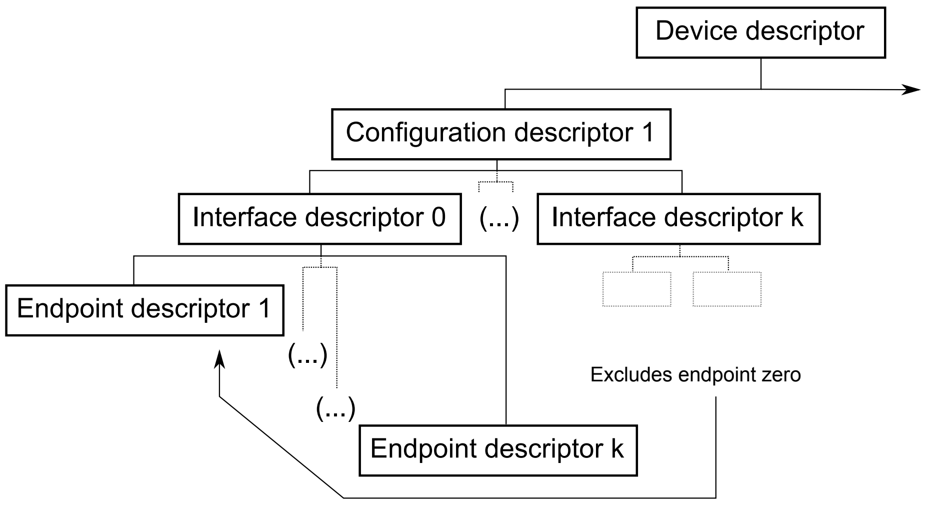


Figure 2.13: USB standard descriptors. Class-specific and special descriptors are omitted from the figure.

Device descriptors are retrieved from the device through requests over control pipes. Standard-defined requests to get descriptors are GET_CONFIGURATION and GET_DESCRIPTOR. These commands must be supported by a USB device, and even class-specific or vendor-specific descriptors are retrieved throgh these requests.

Descriptors that are common and present in all USB devices are:

- A single device descriptor
- One or more configuration descriptors
- One or more interface descriptors
- Zero or more endpoint descriptors

The device descriptor provides the host with fundamental information, including the supported USB revision, number of configurations, device class and subclass and maximum packet length for endpoints zero. Device class may be omitted from the configuration descriptor. In this case, the device class and subclass is specified in the interface descriptors. This could be the case for a device that provides two UART-to-USB bridges. Only one configuration may be active at the same time, but there is no such limitation with interfaces. While all interfaces in a configuration are active at the same time, they may be "bound" to different drivers on the host. A device that has multiple independent interfaces, the device is called a "composite device".

A configuration has attributes that describes if the device is bus powered or not, the maximum power consumption and the number of interfaces. When the host requests a configuration descriptor, it will read the whole subtree below and including the selected configuration descriptor, that is all related interface and endpoint descriptors are also returned.

Interface descriptors are groupings of endpoints, where the endpoints together provide a specific feature in the device. Notably, the two default control endpoints are *not* included in any interface descriptor. Their attributes are fully described by the device descriptor and the fact that they are always used for control transfers.

When connected to a USB host, a USB device undergoes multiple state changes. The general configuration steps for a USB device could be:

1. Initiate device reset. Device responds to commands sent to address 0. Because only one device is reset at the time, and on the same bus, address 0 is guaranteed to be unique.

2. Host finds the maximum packet size for the endpoint 0, through the "get descriptor" command.

3. Host sends a "set address" command, and gives the device a bus-unique address.

4. Host uses GET_DESCRIPTOR, GET_CONFIGURATION to find a suitable driver. The driver configures the device with a SET_CONFIGURATION request.

5. Device-specific requests can be sent now, for further configuration.

These steps are shown as a state diagram in figure 2.14. The process of identifying and assigning an address to a device, is called bus enumeration in the USB specification.

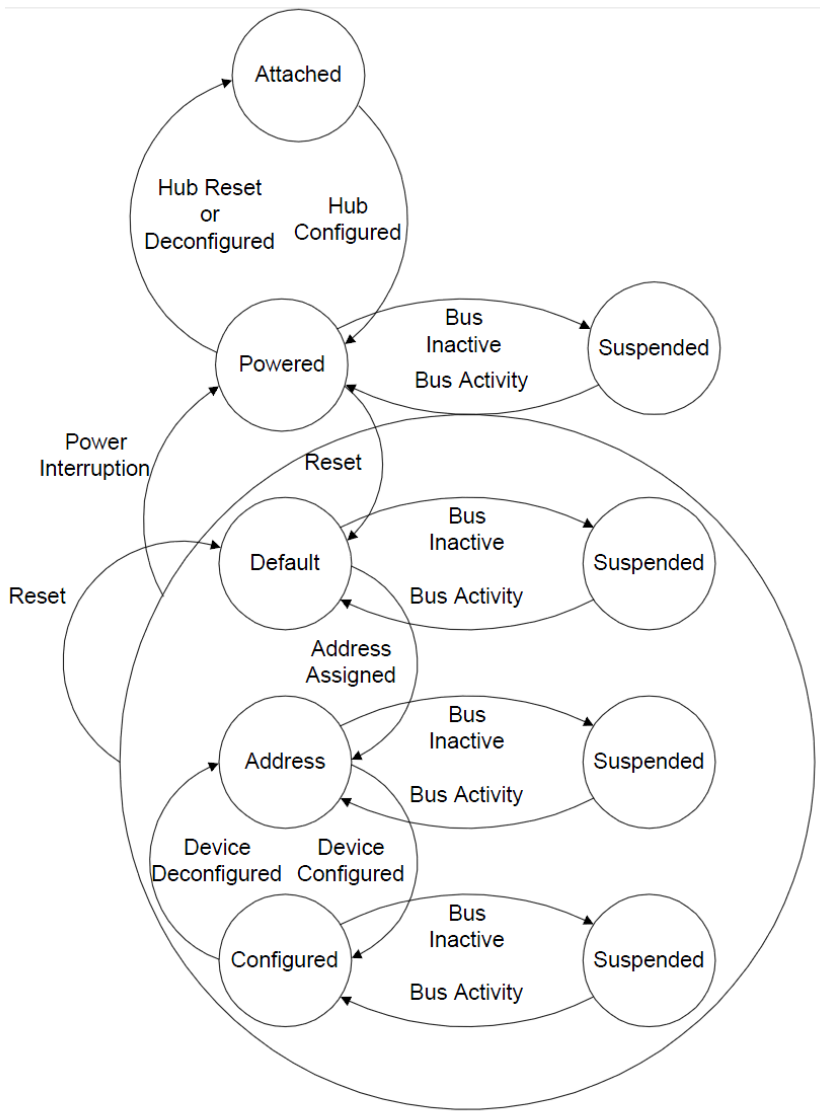


Figure 2.14: State diagram for a USB device

Source: **USB 2.0 specification**

2.7 USB Communications Device Class (CDC)

The CDC specification is a generic USB device class that defines an architecture that is suitable for typical communications devices. Revision 1.2 of the CDC specification focuses on telecommunication and networking devices. The goal of the specification is not to add new communications protocols, but simply to enable a host to identify what existing protocols to use[25].

The CDC 1.2 specification actually covers three classes:

- Communications Device Class
- Communications Interface Class
- Data Interface Class

The device class itself serves to identify the device to the host. The communications interface class defines an interface used for device management. This consists of a management element and optionally a notification element. The management element uses the default control pipe to manage the communications device and its interfaces. The optional notification element of the communucations interface is used to send interface- and endpoint-events to the host. This is typically an interrupt pipe.

All USB CDC devices have at least one interface using the Communications Interface Class. Interfaces implementing this class could be regarded as ″master interfaces″, because they manage zero or more interfaces that together implement a complete function. The association between the ″master interface″ and the interfaces it accepts management requests on behalf on, is described in union functional descriptors.

The communications device presents data to the USB host in a form defined by another class than Communications Device Class. The communications interface is *not* used for actual data transfer. This task is delegated to one or more additional interfaces. If no other device class is suitable for data transfer, the Data Interface Class can be used to define an interface used to transfer data from and to the host.

In addition to the classes described in the CDC specification, there are subclasses for specific types of devices. These subclasses define communications function. The CDC specification defines a wide variety of *models*. A model describes a type of device by specifying requirements with respect to interfaces, endpoints and requests that a device implementing a model must support.

2.8 Inter-module communication and the I^2C bus standard

A common factor between all modular satellite systems, is that they need some means for internal communication. Using a wired bus is the traditional solution to this.

For satellites, I^2C has become a de-facto standard. However SPI is also used. Attractive features with serial buses, are that they have simpler design, use fewer lines and are not as susceptible to clock skew as parallel buses. Using few lines also means that the parasitic capacitance may become less of a problem with higher clock frequencies.

The disadvantage of serial buses are mainly throughput. This can be mitigated by using higher transfer rates, but other problems may arise when using high frequencies.

Inter-Integrated Circuit (I^2C) was developed by Phillips. It is a simple two-wire serial bus suitable for a wide variety of uses in electronics. One of the main advantages with using I^2C is that there is in many cases not necessary to design a bus interface, as support for I^2C is likely to the present in the chip used in a particular design. Many microcontrollers have support for this in hardware, and even if the hardware does not support I^2C, the relative low complexity of the bus makes it feasible to implement it in software. Only using two wires for the bus, also makes it easier to trace on a PCB. Supported transmission rates are 100kbps, 400kbps, 1Mbps and 3.4Mbps. These transfer rates are called Standard-mode, Fast-mode, Fast-mode Plus and High-speed mode respectively[22].

Devices participating in a transaction on an I^2C bus can assume one of two roles: Master or slave. All transfers are initiated,and eventually terminated, by the master. The master and slave role may change, and multiple masters may be connected to the same bus. Devices that are using the bus are either reading or writing data. This means that a module can take one of four possible roles in a bus transaction:

- Master transmitter or receiver
- Slave transmitter or receiver

The two wires used on the bus are called SDA and SCL. The SDA line is used to carry data bits, while the SCL line is used for the clock signal. The bus lines are open-drain design and devices manipulates the bus lines by performing a wired-AND function. That is, devices are only able to pull the bus lines towards ground. This design requires the use of pull-up resistors, to pull the bus lines high when no module is driving them. Figure 2.15 shows a typical design, with two modules and pull-up resistors on the SDA and SCL lines. The maximum wire capacitance, C_w is 400μF. If the capacitance is high, an alternative is to use smaller pull-up resistors or split the bus into segments with bus repeaters.

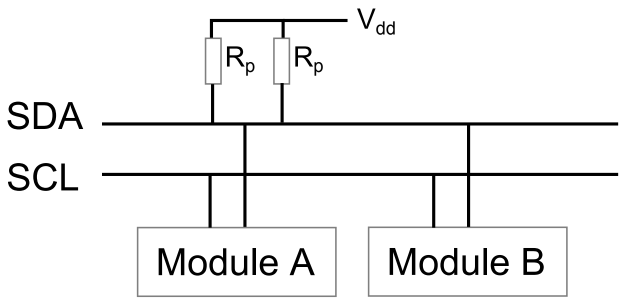
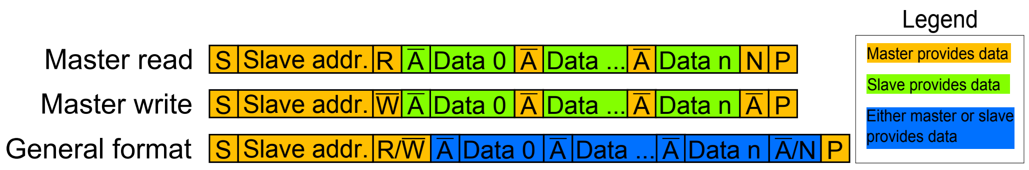


Figure 2.15: Devices connected to an I^2C bus

Each device connected to the bus has an address, and it is not uncommon for a device to respond to several addresses. The address length is either 7 or 10 bits. Some addresses are reserved. For instance, address 0000 000 is the general call address. It is used to address all devices on the bus. Sending the word 0x6h to the general call address is the same as requesting all modules to perform a software reset. Address 1111 0XXX (X is don't care) is reserved for 10-bit addressing. In total, eight addresses are reserved for special purposes.

Bus transactions happens nine bits at the time: The first eight bits are data bits (first bit sent is the most significant). They are followed by a acknowledge/not acknowledge bit (ACK/NAK). Uses 8-bit words: A bus transaction consists of a multiple of 8bit words. The exception to this is the start bit precedeing a new bus transaction, and a stop bit following a transaction. The format of bus transactions are shown in figure 2.16.



Legend
Master provides data
Slave provides data
Either master or slave provides data
Master read
S | Slave addr. | R | $\overline{A}$ | Data 0 | $\overline{A}$ | Data ... | $\overline{A}$ | Data n | N | P
Master write
S | Slave addr. | $\overline{W}$ | $\overline{A}$ | Data 0 | $\overline{A}$ | Data ... | $\overline{A}$ | Data n | $\overline{A}$ | P
General format
S | Slave addr. | R/$\overline{W}$ | $\overline{A}$ | Data 0 | $\overline{A}$ | Data ... | $\overline{A}$ | Data n | $\overline{A}$/N | P

Figure 2.16: The general format of I^2C transactions

Each transaction begins with a start condition (S) and ends with a stop condition (P). In between the start and the stop condition, transactions happens in multiples of nine bits. A start condition can happen during a transaction, provided it was a multiple of nine bit since last start condition. In that case, it is called a repeated start condition (Sr).

The start condition is signaled then SDA is pulled low while SCL is high. The stop condition is raised when SDA transitions from low to high, and SCL is already a logic high. The bus is busy in between the start and the stop condition.

A transaction starts by the master sending a start condition, followed by the slave address and a direction bit. If the direction bit is low, the master want to write to the addressed slave. Otherwise, it is a master read from slave. If a slave recognises its own address on the bus, it pulls the SDA line low during the clock following the direction bit. With the exception of the start and stop condition, data must be kept stable when SCL is high. Figure 2.17 shows the different stages of an I^2C bus transaction.

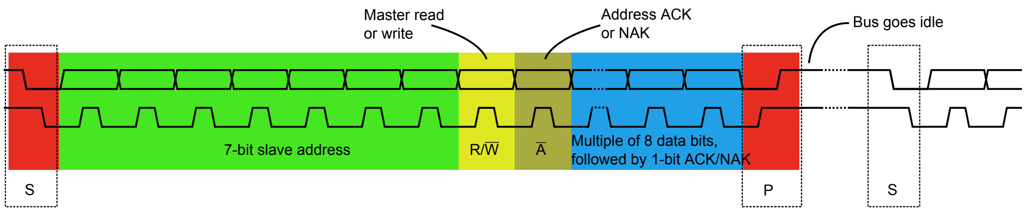


Figure 2.17: The general format of an I2C transaction. Logically separate parts have different coloring.

The timing diagram for minimal master read and master write transfers are shown in figures 2.19 and 2.18. The clock is always generated by the master, but a slave is allowed to stretch the low clock. Clock stretching happens if the slave needs more time to process or retrieve data.

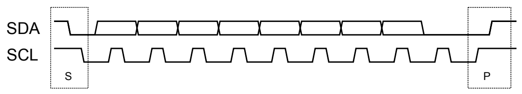


Figure 2.18: Minimum master write to slave module. Master writes 0 bytes.

The master read transfer shown in figure 2.19 illustrated the fact that during a master read, the slave have to provide at least one byte of data. As shown in figure 2.16, the slave provides the data after the addressing stage with the direction bit set to high (”read”).

A NAK may take on different meanings. During a master write, a NAK means that the slave is unable to accept more data. During a master read, a NAK indicates to the slave that this was the last byte to be sent to the master, and that a stop or repeated start condition should follow after. If a NAK is received during the addressing stage, it means that no device has recognised the address put on the bus.

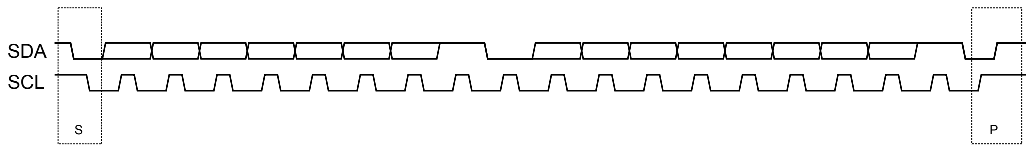


Figure 2.19: Minimum master read from slave. Master reads a single byte.

A realistic use-case for combined starts is shown in figure 2.20. Here the master starts the transaction by writing to the slave. It then sends a repeated start, followed by a new addressing stage. After that, the master reads data from the slave. This is a sensible communication pattern where the slave acts like a piece of memory. During the write stage, the master tells the slave what address to read from. The read stage retrieves data from memory. It also fits well with a command/response-pattern.

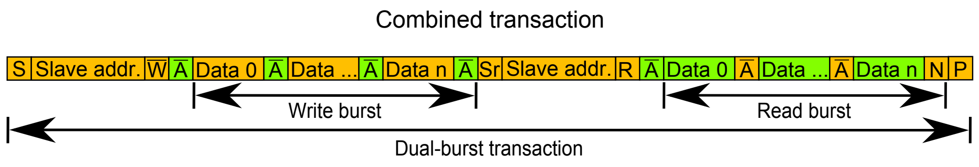


Figure 2.20: The format of a combined I^2C transaction

There is no centralised arbitration mechanism with I^2C. In a multi-master environment, it may happen that two masters start transmitting data at exactly the same time. The primary consideration, is that a device is only able to pull a line low. With I^2C arbitration happens if a master tries to send a logic high, but senses a logic low. The master module detecting this difference loses arbitration and must wait for the bus to become idle. In theory, it is possible for I^2C masters to start sending the same data at exactly the same time without any arbitration happening. If this occurs, the master devices will all perceive the transaction as being successful.

When master is reading data from slave, it is not possible for the slave to signal that it doesn't have more data. Moreover, the master does not identify itself to the slave. This means that it is not possible for a slave to discriminate between masters, unless this is implemented in software.

I^2C also has other disadvantages. First, it is not a very fast bus. The maximum supported data rates are lower than other serial buses. In addition, it is not very hard to lock up the bus. Programming errors can easily lead to stuck SCL lines, or modules can perceive the bus as being in the wrong state. This can happen if a master module has seen a start condition on the bus, but fails to see the stop bit. When that happens, the bus logic in the master must assume that the bus is busy. There is no restrictions on the amount of clock stretching, and the bus design itself gives very little help in finding the offending device.

Part II

NUTS support functions

USB standard I/O 3

Introduction

To aid in debugging and development, the OBC module is fitted with a mini-B receptable USB connector. This allows a computer used for development, to communicate directly with the module – without going through the backplane.

This chapter describes how this feature was implemented, and the challenges involved doing so. The necessary background is provided, followed by a brief description of the solution

The microcontroller used on the OBC module has support for USB, but it also has a UART (Universal Asynchronous Receiver/Transmitter) module. An alternative would be to communicate with the UART throgh a typical D-Sub DB9 connector. But using USB for this kind of communication brings some advantages over the traditional approach: Size of the connector and flexibility, software-wise.

Adding a DB9-connector could quickly get in conflict with the frame. This is especially the case if the connector adds more surface area to the module. Even if it would be possible to overcome this potential problem, USB still has an edge. USB is very flexible and is well suited for any type of data transfers, both low- and high-speed. Currently, the main concern is transmitting binary data in addition to having a simple terminal interface. Having both of these, without developing custom software on the development computer is not trivial using only a single DB9-connector. In addition, using USB is more future-proof: The OBC module has NAND flash memory. In the future, it is not unlikely that a file system is used to manage the flash memory. And in that case, one could reap the benefits of existing USB drivers for mass storage devices in addition to having better data rates than using a simple UART.

3.1 Motivation

The USB-connector on the OBC shall provide a debugging interface. This debugging interface shall serve two functions:

1. It shall provide a serial terminal interface for interactive debugging and testing of software

2. It shall provide a independent serial interface for general transfer of binary data

To reduce complexity and take advantage of existing software, it is imperative that the solution:

- Uses existing drivers in the OS when possible

- Does not require the implementation of extra software, to multiplex terminal input/output with other binary data

- Works with FreeRTOS – the OS used on the OBC module

The goal is to make the OBC module appear as a virtual COM-port to the host. To avoid the need to multiplex unrelated data on the same port, the OBC module must provide *two* virtual COM-ports to the host. The former is already working. Atmel provides a working USB-stack, with device firmware to enable a host computer to communicate with the OBC over a virtual COM port. The current solution make the OBC appear as a CDC-class device. Generic drivers for CDC-class USB-to-serial devices is shipped with windows, in the form of usbser.sys. This means that no custom drivers are necessary on the host computer. However, Windows must be told to match the usbser.sys driver with the device through the use of an Inf-file. The Inf-file tells the operating system what driver to use for a specific combination of a USB vendor ID, product ID and a management interface ID.

The goal is to make the OBC present itself as two virtual COM-ports, by using as much as the existing code as possible.

3.2 Current CDC-class device implementation

The USB device code provided by Atmel, implements a CDC device that follows the Abstract Control Model. The Abstract Control model is suitable virtual COM-port devices. When used for this purpose, a Communication Class Interface and a Data Class Interface is required[26, p.

8]. The data class interface uses two endpoints for data transfer: Bulk IN and OUT.

When the host requests the configuration configuration, all related descriptors are returned. In this case, the descriptors that are returned are:

1. Configuration descriptor with two interfaces
2. Communication Class Interface descriptor
3. Header Functional descriptor
4. Call management functional descriptor
5. Abstract control management functional descriptor
6. Union functional descriptor with 0 as the master interface, and 1 as the first slave interface
7. The notification endpoint descriptor (Interrupt IN)
8. Data Class Interface descriptor
9. Bulk OUT endpoint descriptor
10. Bulk IN endpoint descriptor

The architecture of the USB stack on the OBC is shown in figure 3.1. The UART lib provides an Application Programming Interface (API) that is similar to what a typical UART would have. This is acheived by reading and writing from and to endpoint buffers. The CDC task is responsible for calling UART lib at a configurable time interval. Queuese are used to store incoming and outgoing characters.

3.3 Solution overview

Extra descriptors have to be added to provide a device with two virtual COM ports. Interface Association Descriptors (IAD) are needed to "bind" interfaces associated with the same function, together. Without these descriptors, Windows would not be able to bind the interfaces associated with the two virtual COM ports, with the same driver[28]. IAD descriptors are not part of the official USB 2.0 standard. Instead

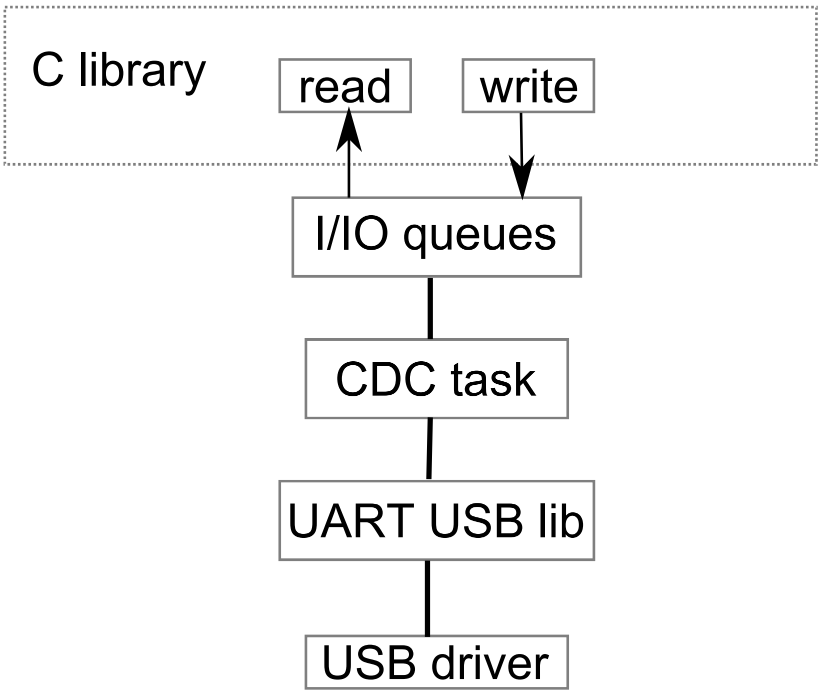


Figure 3.1: The USB stack architecture on the OBC

they are defined in an USB Engineering Change Notice. After the necessary modifications were applied, the following descriptors are returned when the host requests the configuration descriptor:

Buffer space must be allocated for the endpoints must be allocated when the set configuration request is received from the USB host.

1. Configuration descriptor with four interfaces
2. IAD descriptor with 0 as the first interface, and 2 as the number of interfaces
3. Communication Class Interface descriptor
4. Header Functional descriptor
5. Call management functional descriptor
6. Abstract control management functional descriptor
7. Union functional descriptor with 0 as the master interface, and 1 as the first slave interface
8. The notification endpoint descriptor (Interrupt IN)

9. Data Class Interface descriptor
10. Bulk OUT endpoint descriptor
11. Bulk IN endpoint descriptor
12. IAD descriptor with 2 as the first interface, and 2 as the number of interfaces
13. Communication Class Interface descriptor
14. Header Functional descriptor
15. Call management functional descriptor
16. Abstract control management functional descriptor
17. Union functional descriptor with 2 as the master interface, and 3 as the first slave interface
18. The notification endpoint descriptor (Interrupt IN)
19. Data Class Interface descriptor
20. Bulk OUT endpoint descriptor
21. Bulk IN endpoint descriptor

The same inf-file as before was used, except that the extra master interface was added as a declaration in the file. The UART lib, the CDC task and the I/O library was also changed to make it possible to use both of the data interfaces at the same time.

Endpoint buffers are allocated when the USB host sets a configuration. This is done in the function `void usb_user_endpoint_init(uint8_t conf_idx)`. Configuration of the three extra endpoints were added to this function.

System architecture 4

Introduction

This chapter describes the system architecture. The architecture is realised through the implementation of the bus protocol and the software used on each module in the satellite. The topic of this chapter, is to give a overview over the communication patterns within the system, because they are the most relevant to the design of the NUTS bus protocol.

4.1 Architecture overview

The architecture defined does not attempt to be all-encompassing. The crucial elements to the bus protocol, are the communication pattern between the modules. The architecture tries to take into account requirements that have been identified before the start of the project described in this thesis. These requirements can be found in section 8.

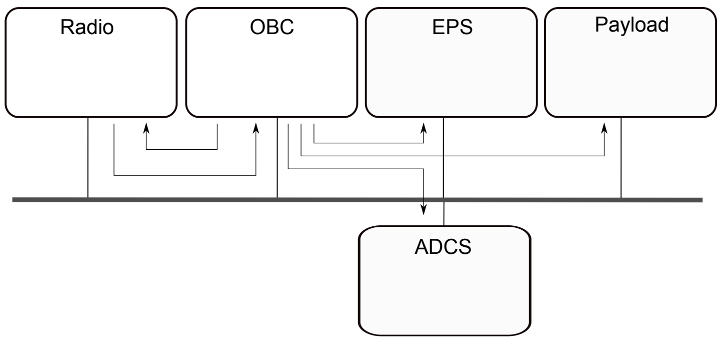


Figure 4.1: System architecture

The architecture restricts how modules are allowed to communicate with each other. These restrictions are capture in figure 4.1. During normal operation, the OBC module act like a proxy for commands originating from the ground segment. All subsystems can perform internal tasks without using the data bus. Under normal circumstances, only the OBC is allowed to issue commands to other modules. The only exception is the radio module. This module is allowed to send commands to the OBC. The commands originate from the ground segment.

The OBC monitors the health of the system, by periodically querying each subsystem for status. Failing modules can be reset or disabled through the control lines in the backplane. The absence of any status query commands at the radio is an indication that the OBC has failed. In that event, the radio is responsible for resetting or disabling the OBC module. If this fails, the radio must assume the role what the OBC had.

Part III

NUTS bus protocol

NUTS bus protocol 5

Introduction

This chapter introduces the proposed design of the NUTS bus protocol. First the context of the NUTS bus protocol is provided. This is to give the reader an understanding of why a bus protocol. The background behind the proposed design is rooted in other protocols for media access. The section following next, describes typical media access control protocols. A brief discussion of typical mechanisms gives a general overview over how this is solved in contemporary systems.

Lastly the proposed NUTS bus protocol is described. Part of this, is also a discussion about how the protocol can be implemented, as well as any potential challenges when doing so.

5.1 Context

NUTS has several subsystems realised on separate, physical module cards. These modules communicate with each other an I^2C bus. The I^2C-standard does specify a bus protocol, but if the purpose is to guarantee fairness, it is not sufficient.

Assume a data bus that supports a transfer rate of W bps. If only one module uses the bus, it is reasonable to expect it to be able to use the bandwidth available. If N modules wants to use the bus at the same time, an optimal and "fair" bus protocol would enable the modules to get a throughput of W/N bps, excluding any overhead.

5.2 Media access methods and fair data bus access

When developing a bus protocol, it is necessary to have an understanding of the environment in where the protocol shall be used. The key points about the data traffic on the internal data bus are:

- Small packets
- Short, transient bursts of data transfer, followed by longer idle periods
- Not inherently real-time
- Want configuration flexibility and operational robustness

Some decisions, tradeoffs and desired features shape the bus protocol. They are the drivers behind the design of the protocol. The key drivers in the NUTS bus protocol are:

- Low overhead from packaging and routing
- Reduce bandwidth used by arbitration
- Shuld provide satisfactory use of the available bandwidth in both worst-case and light traffic-scenarios
- Deterministic latency
- Support for prioritization, or some method to pre-empt ongoing transactions
- Robustness
- Must be possible to implement in software

There are various protocols that may be suitable to use as a starting point for the NUTS bus protocol. What follows is a quick overview over various methods to control access to a medium, and these methods provide relevant inspiration. The methods under consideration are:

- Polling
- TDMA. Time Division Multiple Access
- Binary countdown

Polling

The main attractive feature of polling is its simplicity. In a polling based system, a central master is given the responsibility of starting data transfers. It does this by ”asking” each module in turn, if they have any data to send. A strict implementation would require all data to be sent through the central master, and peer-to-peer communication would not be allowed. In the case of NUTS, this is not a major limitation because that is how the system should work during normal operation. But the dependency on a central master *is* a major issue. The central master is a single point of failure, and it is clear that using a polling-based solution calls for the need to have a failover mechanism.

For NUTS, the OBC would be the central master. It would periodically query each module for data. This would also work when sending data from the ground station to the OBC module. But it would require the radio to buffer incoming data until the next polling interval. Using polling is also not very efficient, because bus resources are wasted when modules have no data to send or receive. The main idea of using polling is still quite reasonable for a system such as NUTS. The main challenge would be handling the change of master roles if the OBC module should fail.

TDMA

The principle of operation with TDMA, is that modules are assigned time slots. When a module has data to send, it would be required to wait until its next time slot. The time slot itself is simply a time interval where the module is allowed to access the bus. This is illustrated in figure 5.1.

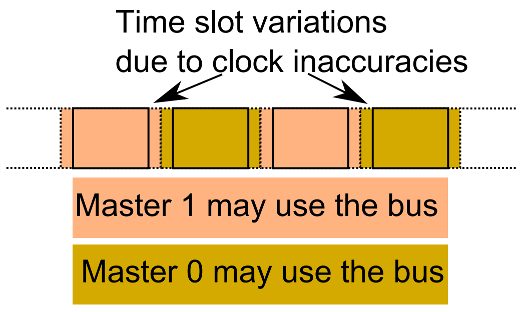


Figure 5.1: I2C with TDMA-like frames

Using TDMA could give better bus utilization than polling-based methods. But it is also the case, the in a low traffic sitation, a lot of the time slots will be wasted. The result would be extremely poor bus utilization. Most of the time, only the OBC module is going to need to use the bus. But when receiving commands from the ground station, the radio will have forward this command to the OBC. But in all other situations, the time slot belonging to the radio would go unused. The time slot could have different sizes, but that would not work well during periods where both the radio and the obc have substantial amounts of data to send.

TDMA would also require support for breaking transactions up into smaller pieces. This will require supporting software on both I^2C slave and masters. The time slice wuld also have to be quite long, to reduce overhead for bus transfers. But this would again be in conflict with the desire to divide the bus resources evenly when the radio also has to use the bus.

A implementation using TDMA would require some means of establishing global time. That would require an authoritative time source that would also be a single point of failure.

Binary countdown

Binary countdown is a method to give data transfers a priority. A possible implementation using a general transmission medium, could start transfers with a number. By "counting ones", it would be possible to establish priority in the case of bus conflicts. With binary countdown all modules would wait for an idle channel. If a module is the sole user of the bus at the current point in time, it is allowed to proceed as usual. But ff two modules start transmission at the same time, the message priority would determine who would win arbitration.

Using binary countdown would require data bus signals to be able to override each other. With I^2C, the module that sends the "lowest" signal would win arbitration. Unfortunately the first bits in an data transfer is the slave address. The principle of using binary countdown is therefore not very realistic in practice.

Summary

Neither of the alternatives are trivial to implement without violating the I^2C-standard. Their use would require a software implementation of a I2C-like protocol, through bit-banging. This would be very time-consuming to both implement and test.

But in the end, the system can rely on raw bandwidth and non-existing latency requirements. This grants some leniency with respect to the design of the bus protocol.

5.3 NUTS bus protocol

The proposed solution is a combination of polling and TDMA. The I^2C protocol can not be changed, but it is possible to dictate fairness within the confines of the system architecture decribed in . The relevant framework conditions are:

- Only the two backplane masters are allowed to initiate bus transfers. This means the OBC and the radio module.
- During normal operation, only the OBC is allowed to address slave modules.
- The OBC sends "heartbeat" packets periodicaly to the other modules, including the radio module.
- The absence of OBC-to-radio-requests is an indication that the OBC has hung.

The main idea is to introduce mandatory wait cycles during long bus transactions. The wait cycles is an opening for the other module to take the bus, as is illustrated in figure 5.2. The bus is idle between the stop condition (P) and the next start condition (S).

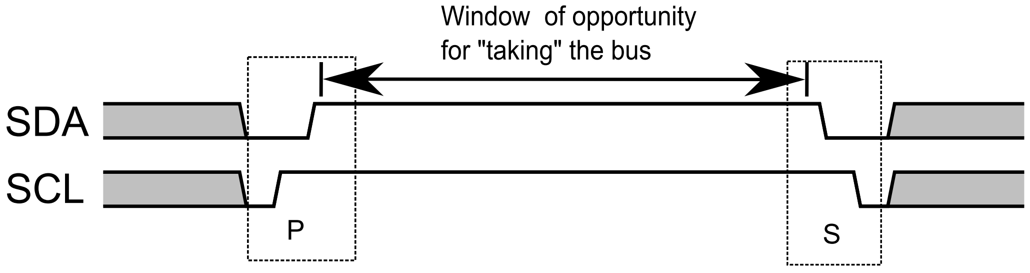


Figure 5.2: The I2C bus can be taken when idle

As if shown in figure 5.3, a bus transfer consists of up to $T_{\max}$ cycles. This is followed by up to w wait cycles. The minimum number of cycles is about ten. This can be seen in the I^2C timing diagram shown in figure 2.18 in section 2.8. Ten cycles is the approximate length of the shortest bus transfer possible. T_{buf} is the minimum number of cycles between a stop signal and a start signal. This is dicated by the I2C standard. In practice, T_{buf} is so small that it can be taken to be zero.

The bus allocation is illustrated in figure 5.4. The grey areas denote wait stages.

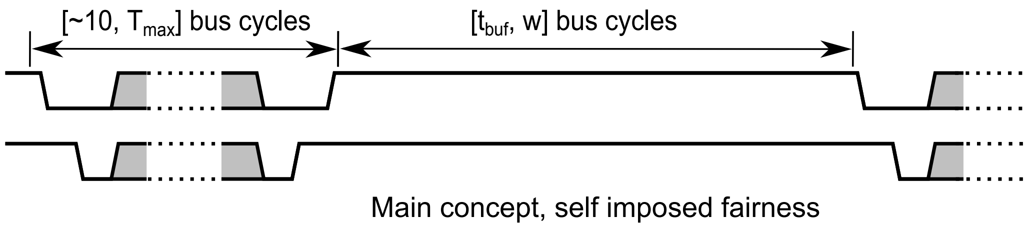


Figure 5.3: Main concept of "self-imposed fairness"

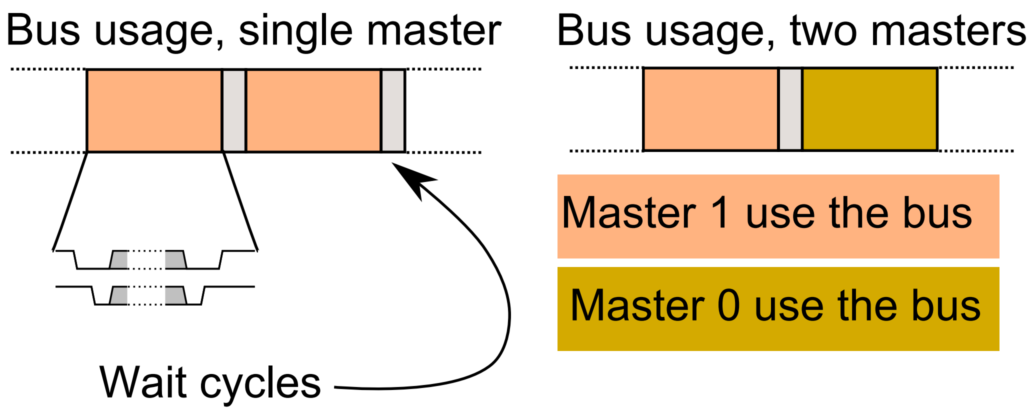


Figure 5.4: Bus activity with mandatory wait cycles

During the wait stage, the module that owned the bus immediately prior to the transaction cycles, has to wait at least w cycles before attempting to take the bus. This scheme has the following properties:

- The delay cycles gives other modules a chance to take the bus
- The maximum latency is bounded by the sum of the burst length and the duration of a single wait stage
- Fixed overhead, albeit a bit high

The expression in equation 5.1 states the throughput given n wait cycles and k bytes to be either sent or received. This assumes a single addressing phase.

$$f(n,k) = \frac{f_{\text{bus\_hz}}}{2 + 8k + n} \tag{5.1}$$

5.4 Assumptions

The use of this method to provide fair access to the bus assumes some knowledge of the system. Notably, the following assumptions have to be made:

- Have to be able to know who is responsible for holding the bus.
- The radio must have a set maximum latency for addressing the OBC during long OBC-to-slave bus transactions.
- The radio is restricted to communicating with only the OBC module normal operation.
- It must be possible to resume a bus transaction, because long transactions will have to be interrupted by wait cycles.
- Clock stretching is not allowed, neither for master nor for slave.

There are two I^2C masters in NUTS. When a module is holding the bus busy, it must be either the other master or a slave that this master communicates with.

Because bus transfers may be broken into smaller pieces, the software on both the I^2C slave and master must support resuming of transfers. Some types of I^2C devices may not be able to resume a transfers. A typical example would be NAND flash with an I^2C interface. Writes have to be done in entire blocks. If several blocks are to be written, it is not uncommon that the master has to wait 100ms per block write. This means that the maximum allowed duration of a bus transfer, has to exceed that of the minimum duration of all transfers to all i2c slaves on the internal data bus. There are no such memories in the present design of NUTS. If this is changed in the future, they memory interface has to be hidden behind a microcontroller used for buffering and assembly/reassembly of data.

Deciding the amount of wait cycles

The number of wait cycles, w, is a compromise between the minimum I^2C transfer length, the target maximum latency for getting access to the bus, and the amount of protocol overhead.

Implementing the wait stages is likely to require scheduling events in the future. This requires accurate timers. In addition, the I^2C hardware is likely to depend on interrupts. Given that this is the case, the variation in interrupt processing can be make or break the NUTS bus protocol.

Test cases

How the bus protocol should work in practice, is best conveyed through example cases.

Case 1

Scenario: The OBC master does read/write from/to slave. The requires a multiple of $T_{\max}$ bus cycles.
Expected outcome: The OBC does transactions in intervals $T_{\max}$ cycles, followed by w wait cycles.

Case 2

Scenario: The same as Case 1. But the radio module also wants to use the bus to communicate with the OBC module.
Expected outcome: The OBC and the radio module does transactions in intervals $T_{\max}$ cycles, followed by $[T_{\text{buf}}, w]$ wait cycles.

Case 3

Scenario: The radio wants to use the bus, but the OBC is using the bus longer than expected.
Expected outcome: Radio resets the and tries to use the bus again. The OBC is responsible for resuming the transaction, but because a backplane reset occurred during an OBC read/write, the OBC must wait. The Radio successfully takes the bus, and communicates with the OBC.
Notes: The backplane reset is not very likely to resolve the situation, but the alternative would be to do a system reset.

Case 4

Scenario: The same as Case 3. The backplane reset fails to resolve the situation.
Expected outcome: There are several possibilities:

- The OBC already knows about the situation, but has not taken any action yet.
- The OBC has not noticed because it was not using the bus.
- The OBC is the culprit. Either due to software or hardware error.

The radio should proceed with disabling the least important modules. his happens until everything except the OBC. When only the OBC and the radio module are left, the radio must then reset the OBC module.

Case 5

Scenario: The OBC reset from Case 4 resolves the sitation. The radio finishes its transaction with the OBC. The other modules are still disabled.
Expected outcome: When a slave fails to respond to respond to heartbeat commands, the OBC re-enables them.

Case 6

Scenario: The OBC reset does *not* resolve the situation.
Expected outcome: The radio reactives the other modules, except the OBC. This is left disabled. The radio must assume the role the OBC had as a system status logger. Manual intervention is required to exit from this state.

Case 7

Scenario: A module is responsible for holding the bus longer than allowed. This is detected by the OBC during an OBC-to-Radio-transaction, or OBC-to-other-module-transaction.
Expected outcome: Try to reset the backplane. If this does not work, the least important modules are disabled in order until the offender is found.

Case 8

Scenario: The OBC finds that the Radio is responsible.
Expected outcome: The OBC resets the radio. If this fails to resolve the situation, the OBC waits up to two passes, periodically testing the bus state. If the situation is not solved within two passes, the OBC assumes that either an software or hardware error on the radio has occured. The OBC starts reprogramming the radio, to see if this resolves the situation.
Rationale: If the bus is no longer working, it should still be possible to solve the situation from the ground. The possible recovery procedure at this point, is to disable all modules from the radio. A module reset of the OBC or any other module cannot be expected to work because the culprit is in the backplane. This assumes that the reset logic still works.

This is a very serious situation. At this point the missions should be considered to be a at EOL (End-Of-life). All modules will now have to operate autonomusly. The OBC is disabled at this point. Any software or hardware error in the radio at this point means the definitive end of the mission.

The OBC is disabled because it no longer makes sense to have it active. The only hope is that the radio is still working. And if that is the case, the OBC must to be allowed to interfere by repogramming the radio module with a possibly faulty firmware image. It is also quite possible that the ADCS system is still operating.

Implementational considerations

Because bus transfers are split into pieces, both bus masters and slave must be able to resume bus transfers. This may be challenging, because an I^2C master does not identify itself to the slave. A identification mechanism must be implemented in software.

Splitting bus transfers may also be complicated. It would not be possible to split a transfer containing a repeated start, without changing the semantics of the transfer. Instead a set of rules must the developed, for what format transfers should have. This is clearly an application level concern, and no bus abstraction library would be able to provide a satisfactory all-around solution.

The first step is to develop a I^2C driver that enabled applications to use the bus using high level function calls. But it would still be up to each application to not violate the bus protocol.

Summary

The advantages with this solution to the NUTS bus protocol are:

- Requires no extra hardware. The protocol is implemented in software.
- Does not require us to go outside the I^2C standard.
- No single point of failure. Each master module is responsible for following the protocol, and both master modules are able to respond to protocol violations.
- Deterministic latency
- Relatively low implementational complexity

The disadvantages are:

- Lower bus utilisation than without any arbitration.
- Protocol violations are only going to be detected when the other master wants to use the bus.
- Have to find a upper bound on the response time of each master.
- It may be difficult to support resuming of transfers.

Interrupt latency 6

Introduction

The software responds to hardware events through interrupt service routines. The underlying event may happen at a certain frequency, but both the hardware and the software introduce delays. These delays makes it impossible to respond instantaneously to these events.

This chapter describes how the delay introduces by software, can be measured. In an experiment, hardware timers are used to generate interrupt at various frequencies. At the end of each period, a hardware interrupt is raised. The delay between the timer where the interrupt was raised, and to the point where the software can respond is measured both when using FreeRTOS v7.0.0. As a reference, the delay is also measured when not using an operating system.

6.1 Motivation

When the purpose is to build a system with that provides "fair" access to the data bus, a combination of hardware and software makes this kind of behaviour possible to implement. What kind of involvment is required from either is also highly dependent on the chosen hardware and the software. Because I2C does not have a central arbitration mechanism, each I2C master has to behave in a fair manner. This means giving other I2C masters the chance to use the bus. This almost certainly is going to involve the use of software.

The events of interest in the context of fair access to the data bus are:

- A module has held the bus for too long, and it must relinquish control to give other module a fair chance to use it.

- A module wants to use the bus, and the hardware senses that the bus is idle. An interrupt is raised, and the module must try to take control over the bus.

- A module has more data to send, but must has already continously used the bus as long as it is allowed to do. A ”wake-up” event must be scheduled, so that the module can take the bus yet again if no other module takes the bus in the meantime.

- A module wants to use the bus, and the bus has just become idle. The module should take the bus as fast as possible.

When measuring delays introduced by software, the hardware where a specific event originates from is not of much interest. Hardware timers are convenient to set up to generate ”timer events”. The delay between a timer event and the point where a piece of useful program code is able to do something about it, is unlikely to be different for different hardware events.

The purpose of measuring interrupt latency is not so much tied to the context of using this with either a hardware or software implementation of I2C, as it is to determine exactly how much we can rely on the predictability of events processed by interrupt service routines. At this point, it is not possible to say anything meaningful about the software that is going to run on each module in the satellite. For this reason, the experiment does not try to simulate any kind of ”workload” in the form of extra tasks and drivers.

6.2 Test environment

The experiment uses the same microcontroller as is used on the OBC-module. While little is known about the software at launch-time, using the same hardware should should give results that are possible to reproduce on the OBC-module. The operating system and compiler toolchain is also the same.

The hardware used for this experiment is:

- UC3-A3 Xplained evaluation board with an AVR32UC3A3256 microcontroller

- AVR JTAGICE mkII programmer and debugger

The software enironment is the combination of compiler toolchain, operating system and drivers provided by Atmel. The drivers are a part of AVR Software Framework (ASF).

- FreeRTOS v7.0.0
- Atmel AVR Studio 5 (Version: 5.0.1163)
- AVR32 GCC version 4.4.3 with GNU toolchain 3.2.3_261
- AVR Software Framework ASF-2.5.1-17860.53

For this experiment, the code is compiled with the O1-option. No other optimization flags were changed from the defaults. The relevant drivers from ASF are:

- Timer/Counter driver
- Interrupt controller driver

The drivers are likely to affect the results. The timer/counter driver because it is ultimately thius code that alters the hardware registers of the timer/counter module. Interrupt processing times are also affected by the interrupt controller driver. To test what kind of impact the interrupt controller driver implementation has on the results, the experiment is also run without using an operating system.

6.3 Experiment description

The experiment uses two hardware timers. The timers are located within the same peripheral timer/counter-module, because that enables the possibility of using a synchronised start of the timer/counters. Without being able to start the timers at *exactly* the same time, small errors are introduced in the measurement. This is due to drift between the timers because of the instruction execution latency between starting the timers separately.

One of the timers are used as a counter, while the other timer is configured to generate interrupts at a specific frequency. In a situation where there is no jitter, the expected time (in cycles) between each interrupt is given by $\frac{f_{CPU}}{f_{interrupts}}$. Deviations from this number is regarded as jitter.

The actual number of cycles between each interrupt, k, is bounded by:

$$\frac{f_{CPU}}{f_{interrupts}} \leq k < \frac{2 \cdot f_{CPU}}{f_{interrupts}}$$

If k was lower than the lower bound, it would mean that the interrupt controller could somehow predict the future and generate the interrupt ahead of time. The upper bound is there because exceeding this, would mean that interrupts would be lost simply because the time taken to process a single interrupt is longer than the time to the next interrupt. In this case, the system simply cannot accomodate the current interrupt frequency, and any measurements are meaningless. This also means that the time executing the Interrupt Service Routine (ISR) must be lower than the time between each interrupt.

6.4 Hardware considerations

The UC3 has several timers that are appropriate for this kind of experiment. The Timer/Counters on the UC3, uses 16-bit counters. Because the timers use the peripheral clock B signal, the resolution is limited for higher frequencies. This stems from the fact that the peripheral clock B (PBA) frequency is ultimately derived from the CPU frequency. In practice, this is not a major issue. The jitter benchmark is only interesting for high interrupt rates. Moreover, the PBA clock source may be prescaled at the Timer/Counter. In any case, the limited size of the 16-bit counter register and a high input clock to the timer is only a limitation if longer time durations are to be measured. That is time intervals spanning multiple seconds, minutes, hours and days.

Each Timer/Counter has three independent channels, and they can be individually configured to tick with the same frequency as one of either five internal clock sources or three external clock sources. The clock source to each channel can also be individually enabled and disabled. The center of each channel is a 16-bit ”Counter Value” register, in addition to three registers: RA-RC.

In addition, each channel has two separate input/outputs, TIOA and TIOB. The TIOA signal of a channel may also be clock input to *other* channels in the same Timer/Counter module. Also, both the TIOA and TIOB signals may be outputs or trigger inputs, depending on whether the channel operates in waveform or capture mode.

Each Timer/Counter channel can operate in two modes: Waveform mode and capture mode. Capture mode allows a TC channel to perform measurement on input signals, while waveform mode is used for wave generation. For measuring the interrupt latency, both the counter and the interrupt channel is set up in waveform mode. In waveform mode, the TIOA and TIOB signals may be configured to be outputs. They may drive external pins if the I/O-controller is configured to let the Timer/Counter-module do this. This functionality may be useful for debugging purposes, to ensure that the timer "ticks" at the expected rate.

Channel triggers resets the counter and starts the clock, if it is not already started. In addition to external triggers on there are other methods to trigger a channel:

- Software: Programmatically "starts" a channel.
- RC Compare: The channel triggers when the Counter Value register equals the RC register, when channel is in waveform mode.
- SYNC trigger: Same effect as a software trigger, except that this triggers *all* channels within the Timer/Counter module.

6.5 Hardware setup

The benchmark uses two channels on a Timer/Counter module. The channels are set to operate in waveform mode. In waveform mode, the RA, RB and RC-registers are used as compare registers against the channel Counter Value Register.

For the experiment, the overall hardware configration was:

- Use Timer/Counter 0
- Use channel 1 as a counter channel
- Use channel 2 as the "interrupt" channel
- Both channels are configured in "up" mode, and trigger on RC compare
- Channel 1 is configured to generate interrupt on Counter Value overflow
- Channel 2 is configured to generate interrupt on RC compare

- TIOB is configured as output on channel 2, and set to toggle on RC compare
- `TIMER_CLOCK3` is set as the clock source for both channel 1 and channel 2
- The PBA clock is set to the same as the CPU clock, and the CPU clock is set to 12MHz

In up mode the counter iterates up to the value in the RC-register. When they are equal, the counter starts again at the beginning. TIOB on channel 2 is configured to toggle when this happens. The result is that the corresponding I/O pin is toggled at half the interrupt frequency. This way, the timer ticks can be observed on an oscilloscope. The expected waveform compared to the counter register and the RC register is shown in figure 6.1. The period of the TIOB waveform would be exactly twice as long as the period of the waveform "traced" by the counter value register.

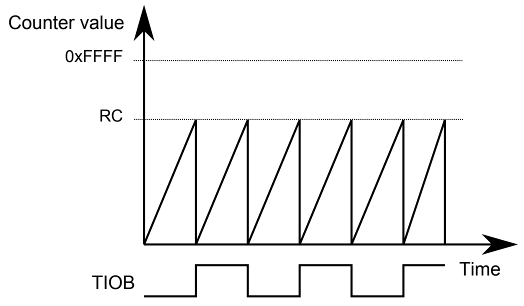


Figure 6.1: The TIOB pin toggles on RC compares. The compare event resets the RC register, and toggles the TIOB pin.

`TIMER_CLOCK3` is connected to the Peripheral B clock, divided by 8. Available dividers are 2, 8, 32, 128. Ideally the implementation could choose a clock source based upon the requested interrupt frequency, but this functionality was not implemented for simplicity reasons. However `TIMER_CLOCK3` is a good compromise because with a Peripheral B clock of 12MHz, it is possible to generate interrupts at a wide and set of high frequencies.

Channel 2 is used as the interrupt channel, because the TIOB output on this channel on TC0 is available on a pin header on the UC3-A3 Xplained evaluation board. Channel 1 was chosen as the counter channel because FreeRTOS may be configured to use TC0 with channel 0 in waveform/RC-compare mode to generate timer ticks. Thus, the same Timer/Counter module could be used to run the benchmark, without affecting the timer channel used by the operating system. However, the implementation was changed to use SYNC triggering to synchronise the counter and interrupt channel, after it was found to affect the results. There is no way to do a selective SYNC trigger, so all channels are affected. This means that the benchmark must have exclusive access to the Timer/Counter module used in the experiment.

To make this possible, the FreeRTOS source code was changed to use Timer/Counter 1, when the OS is configured to use a Timer/Counter. In any case, this is a configurable parameter when initializing the benchmark.

The benchmark may be configured to use the AVR32 Count register to calculate the number of cpu clock cycles spent in the timer interrupt ISR. However, by default FreeRTOS uses AVR32 Count/Compare interrupts for the periodic timer ticks. If this feature is to be used with FreeRTOS, care should be taken that the AVR32 Count register is not used by the operating system. Instead, FreeRTOS should use a Timer/Counter channel when this benchmark feature is used.

The counter channel is set to interrupt on counter value overflow. This is to catch serious errors during debugging, because this ISR should never be executed. The implementation of the ISR that is responsible for storing the result is shown in listing 6.1

Listing 6.1: The ISR implementation for the interrupt channel

```
__attribute__((interrupt("none")))
static void timer_interrupt(void) {
#if jitterSTORE_ISR_EXECUTION_CYCLES == 1
        Set_system_register(AVR32_COUNT, 0);
#endif
        uint16_t time;
        uint32_t jitter;
        time = Rd_bitfield(timers_tc->channel[counter_channel].cv, AVR32_TC_CV_MASK);

        jitter = Abs((time<<3) - jitterEXPECTED_TIME_DIFF);
        jitter_samples[jitter_samples_idx++] = jitter;
        jitter_samples_idx &= jitterSAMPLE_COUNT-1;

        if( time > max_time ) {
                max_time = time;

                if( jitter > max_jitter ) {
                        max_jitter = jitter;
                }
        }

        tc_read_sr(timers_tc, interrupt_channel);
        reset_channels();
#if jitterSTORE_ISR_EXECUTION_CYCLES == 1
        interrupt_exec_time = Get_system_register(AVR32_COUNT);
#endif
}
```

The implementation is kept as short as possible. The value of the counter channel bypasses the driver to save time between the start of the ISR and the ”latching” of the counter value. The routine stores the `jitterSAMPLE_COUNT` last results in an array, to calculate an estimated average at the end of the experiment.

The jitter experiment is run from an FreeRTOS task, or in an ordinary function when no operating system is used. The code for both of them is shown in listing 6.2

Listing 6.2: The code that calls the jitter test code. The first function is used with FreeRTOS, while the second function is sued when not testing within FreeRTOS

```
static void jitter_benchmark_task(void *pvParams) {
        for(size_t i = 0; i < params_count; i++) {
                vTaskDelay(TASK_DELAY_S(5));
                jitter_init(params[i]);
                jitter_start();
                vTaskDelay(TASK_DELAY_S(benchmark_time_sec));
                jitter_stop();
                jitter_results[i] = jitter_get_metrics();
        }
        barrier();
        asm volatile("nop"); /* Breakpoint for fetching results */
        vTaskDelete(NULL);
}

static void jitter_benchmark_notask(void) {
        bool irq_enabled = Is_global_interrupt_enabled();
        Enable_global_interrupt();

        for(size_t i = 0; i < params_count; i++) {
                jitter_init(params[i]);
                jitter_start();
                cpu_delay_ms(benchmark_time_sec * 1000, APPLI_CPU_SPEED);
                jitter_stop();
                jitter_results[i] = jitter_get_metrics();
        }

        barrier();
        asm volatile("nop"); /* Breakpoint for fetching results */

        if( !irq_enabled )
                Disable_global_interrupt();
}
```

The benchmarks are run for 20 seconds. This is more than enough time to gather enough sample points when not using an operating system. When using FreeRTOS, the results may be affected by ”phases” of high jitter. This is difficult to predict when a lot of tasks are running. To reduce the effect of this, the only tasks that are running during the experiment is the jitter benchmark task itself. In addition to the timer interrupts in the jitter benchmark, there are periodic ”tick”-interrupts from the operating system. These interrupts lead to the execution of a critical section, albeit short. This was not possible to eliminate. Unfortunately, the relative timing of these interrupts are likely to affect the results when benchmarking with FreeRTOS.

6.6 Testing the implementation

The Timer/Counter was configured to toggle an external pin. The output on this pin was observed by connecting an oscilloscope probe to the matching pin on a pin header on the evaluation board. This hardware setup is shown in figure 6.2.

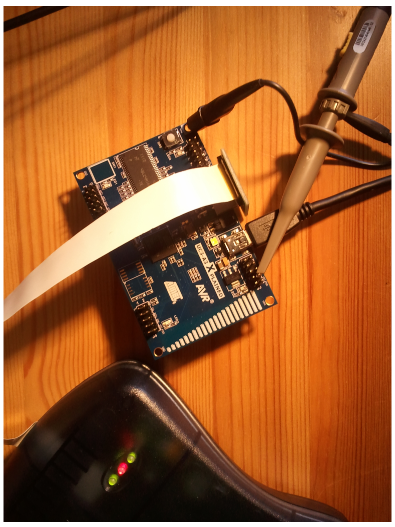


Figure 6.2: The setup used to test the experiment implementation

Observing the effects of the jitter benchmark code, without affecting the results is not trivial. During testing, an AVR JTAGICE mkII debugger/programmer was used. By using a debugger, one can assert that appropriate functions are called. The timer clock is frozen while the debugger has stopped program execution. This means that timer increments can be observed by single stepping through the code and observing the hardware registers with the debugger.

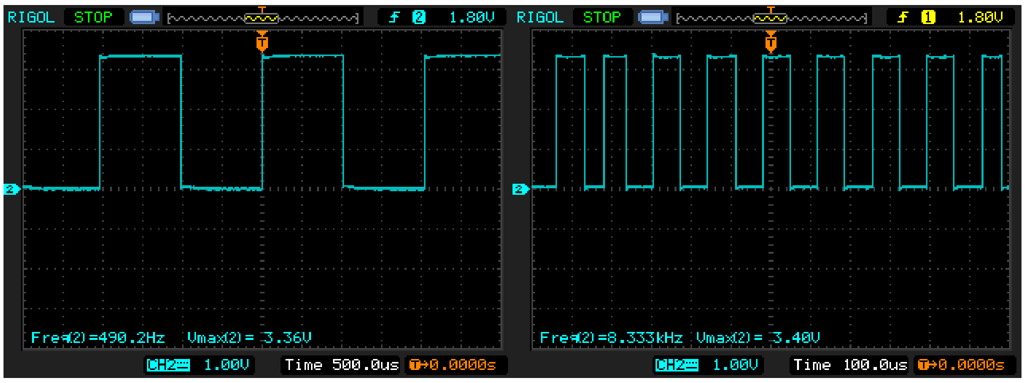


Figure 6.3: The waveform created by toggling TIOB at a target rate of 1kHz (left) and 20kHz (right)

The code was developed and run on an UC3-A3 Xplained evaluation board, and the benchmarking code was configured to trigger the TIOB pin on each RC compare. The TIOB pin on Timer/Channel 0 is pin PX19 on the AVR32-UC3A3. On the evaluation board, this pin was available as pin 8 on header J3. Using a probe, the toggling could be observed with an oscilloscope. An example with an interrupt frequencies of 1kHz and 20kHz is shown in figure 6.3. The measurments taken shows that at a low interrupt rate of 1kHz, the signal on the TIOB pin agrees with what could be expected. But at a interrupt rate of 20kHz, there is a quite substantial discrepancy between the expected result of 10kHz and the actual result of about 8.3kHz. This could be due to the fact that the timing requirements with respect to the CPU frequency, are more strict at high interrupt rates. The interrupt rate is set by the code snippet shown in 6.3

Listing 6.3: The code that configures the RC register with the requested interrupt rate

```
static void set_timer_channel_rc(void) {
        unsigned short rc_value;
        rc_value = (jitterTIMER_CLOCK_HZ + (int_freq_hz << 2))
```

```
                        /(int_freq_hz <<3);
        if( tc_write_rc(timers_tc, interrupt_channel, rc_value)
                == TC_INVALID_ARGUMENT ) {
                __builtin_breakpoint();
        }
        actual_int_freq_hz = (jitterTIMER_CLOCK_HZ >> 3)/rc_value;
}
```

The requested interrupt rate is rounded to a number that is possible with the current prescaler. But there was no difference between the actual and the requested interrupt rate in any of the test cases. The actual cause of the deviation from the expected pin toggling rate was not found.

6.7 The test cases

The purpose of the experiment is to get an indication of how long it takes between a hardware interrupt, and to the time where the system is able to process the event. The test cases are run in two different contexts:

- With FreeRTOS, from within a task. The only other interrupt source is the periodic timer "tick"-interrupts from the OS.
- Without an operating system and with no other interrupt source than the timer interrupts from the jitter test code.

The CPU core frequency is 12MHz, but the results are expressed in clock cycles. For that reason, the result themselves should not be affected by increasing or lowering the clock frequency, as long as the number of cycles between each interrupt is higher than the number of clock cycles it takes to set up the system stack, calling the interrupt routine and returning from it. The interrupt frequencies used in the test are not directly related to any specific interrupt pattern when using other peripheral modules on the UC3 microcontroller. This is deliberate in this case, because at this point there is very little existing software for the OBC that could be used as a meaningful reference point. Instead the interrupt frequencies are chosen on the merit that the time between each interrupt, can be expressed as various lengths of I2C transactions on the internal data bus in the satellite. This assumes a CPU clock frequency of 12MHz, although the test results themselves are not tied to any particular use case.

The microcontroller supports four interrupt levels. In addition to testing various interrupt frequencies, interrupt level zero and interrupt level three will be compared. These are the interrupt levels with the lowest and highest priority. The test parameters are as following:

- Interrupt rate at 1kHz, with interrupt level zero and three.
- Interrupt rate at 5kHz, with interrupt level zero and three.
- Interrupt rate at 10kHz, with interrupt level zero and three.
- Interrupt rate at 15kHz, with interrupt level zero and three.
- Interrupt rate at 20kHz, with interrupt level zero and three.

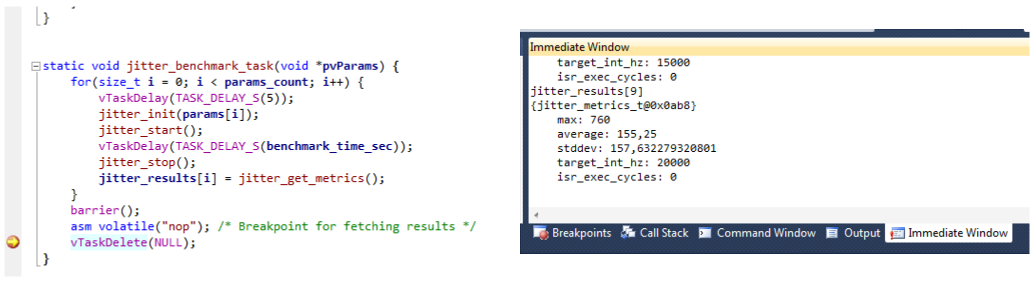
Assuming a CPU clock frequency $f_{cpu} = 12MHz$, a series of comparisons can be made as shown in table 6.1. At 12MHz the cycle time is about 83.3ns. At the maximum supported CPU clock frequency of 66MHz, the cycle time is about 15.15ns.

Table 6.1: Clock cycles between interrupts, related to I2C bus cycles

Interrupt frequency	CPU cycles	Bus cycles (100kHz)	Bus cycles (400kHz)
1kHz	12000	100	400
5kHz	2400	20	80
10kHz	1200	10	40
15kHz	800	6	26
20kHz	600	5	20

The results in table 6.1 can be compared to the duration of a bus transaction. Assume that a start and a stop bit together takes two bit times on an I2C bus, that the address stage takes nine bit times, and each byte sent or received to or from a I2C slave also takes nine bit times. In that case, it is useful to use 20 bus cycles as a baseline for a minimal but ″useful″ bus transaction (2 for start and stop, 18 for address and a single data byte). For a I2C bus operating in standard mode, the time between each interrupt is already at the minimal transaction time limit at 5kHz.

6.8 Retrieving the results



```
}

static void jitter_benchmark_task(void *pvParams) {
    for(size_t i = 0; i < params_count; i++) {
        vTaskDelay(TASK_DELAY_S(5));
        jitter_init(params[i]);
        jitter_start();
        vTaskDelay(TASK_DELAY_S(benchmark_time_sec));
        jitter_stop();
        jitter_results[i] = jitter_get_metrics();
    }
    barrier();
    asm volatile("nop"); /* Breakpoint for fetching results */
    vTaskDelete(NULL);
}
```

```
Immediate Window
    target_int_hz: 15000
    isr_exec_cycles: 0
jitter_results[9]
{jitter_metrics_t@0x0ab8}
    max: 760
    average: 155,25
    stddev: 157,632279320801
    target_int_hz: 20000
    isr_exec_cycles: 0
```

Figure 6.4: A breakpoint is set at some point after the results are stored. The tests results are retrieved through the immediate window

There are few alternatives for getting the test results from the micro-controller. One option would be to send the results over an USB cable. This requires a USB stack. While this is available, it could affect the test results both when using FreeRTOS and without FreeRTOS. This would also require extra code for formatting the data. For simplicity, the jitter benchmark code stores the results in an globally accessible array. By setting a breakpoint at some point after the test is completed, it is possible to read out the results from the immediate window. The immediate window in AVR Studio 5 allows the user to execute C statements when the program is paused by the debugger. This procedure of pausing at a breakpoint and reading the results is shown in figure 6.4

6.9 Results

If any guarantees are to be made with respect to the interrupt processing latency, only the maximum jitter is of any interest. But there are no inherent real-time requirements in the system. Having a high delay in the worst case may be acceptable. Also, designing for the worst case may also be overly pessimistic. The worst-case jitter observed in the tests are shown in figure 6.5, for both interrupt level zero and interrupt level three. Judging from the results, it would seem that the interrupt level has a definite impact on the interrupt latency. To see if this is the case, it is useful to compare the average jitter, as is shown in figure 6.6. The same figure also shows the result when not using an operating system.

From figure 6.6, it is clear that the operating system has an impact on the interrupt latency. However the results also show that the error

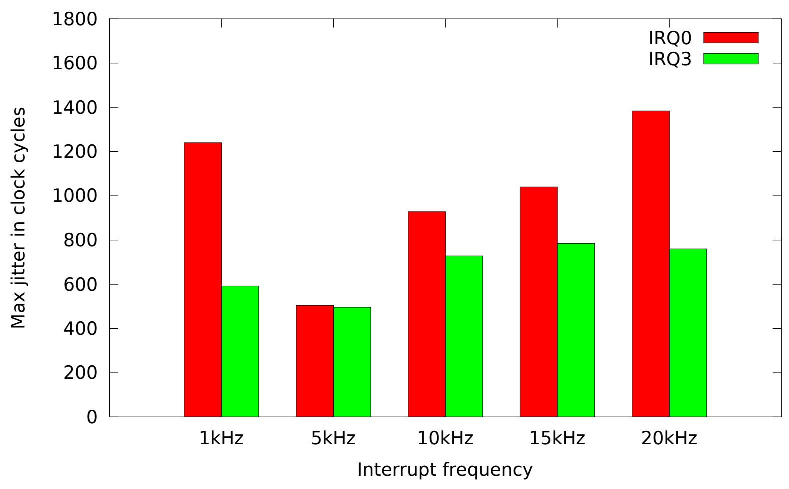


Figure 6.5: The maximum measured jitter when using FreeRTOS

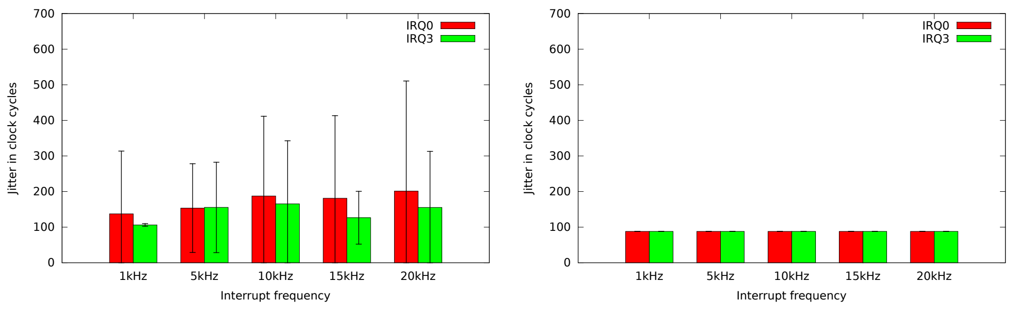


Figure 6.6: The jitter results with estimated averages and standard deviation with FreeRTOS (left) and without FreeRTOS (right)

margins are huge relative to the average latency. Still, if there is no strict requirement that the interrupt latency is bounded by a certain amount of cycles, the results shows that it is better to design for the average case.

6.10 Discussion

When not using an OS, the interrupt latency is always the same. This is as expected, but because the same interrupt controller driver is used both with and without FreeRTOS, the ”no-OS” results gives best-case results. When not using an OS the interrupt latency is always 88 cycles. The worst case result dictated by the architecture is 17 cycles, and the best latency possible is twelve cycles[2]. The relative slowdown seen in the results are due to the implementation of the interrupt controller driver. When an interrupt occurs, the CPU has to go through at least two jump tables until the actual ISR is called. This comes in addition to the worst-case latency described by the technical reference manual.

The standard deviations and the averages are estimated from the last 32 interrupts. This is similar to a Finite Impulse Response (FIR) filter, because old history is quickly forgotten. This is especially true when measuring at high interrupt frequencies. There were a couple of alternatives for calculating the average:

- Doing a ″rolling″ avereage.
- Use a longer sample history buffer.
- Do multiple test runs and add the results together. The weights are the same, so the result would have been correct.

The UC3 has ho hardware support for floating point calculations. This means that even simple single-precision calculations take many clock cycles. This would lead to the risk that execution of the ISR would take too long. As an alternative, using integers and round the result at each sample point. This was found to be too biased to old history. The average would simply ″stick″ to the same value, unless an outlier were encountered.

Using a longer sample buffer history was also an alternative. But this introduced some overflow errors with the current implementation. This could likely have been solved by running the tests several times and average the results from different test runs. This would introduce some rounding errors, but it would likely be possible to increase the history. The key point is that the history must be long enough to cover any high latency phases. Due to limited time, this solution was not tried.

A completely different source of error is the time between the beginning of the ISR, and the time where the value of the counter register is read. But this only takes a few cycles, is on a completely different scale than the test results shown in figure 6.6 and figure 6.5.

6.11 Implementational notes

The UC3 supports nesting of interrupts. This means that an interrupt with a higher priority may interrupt the servicing of a lower-prioritised interrupt. If the ISR of a low-priority interrupt is executing, and a high-priority interrupt occurs, control is transferred to the new ISR. Some ports of FreeRTOS support interrupt nesting. In this case, there are two configurable parameters: `configKERNEL_INTERRUPT_PRIORITY` and `configMAX_SYSCALL_INTERRUPT_PRIORITY`. The former sets the interrupt priority used by the tick interrupt, while the latter sets the highest interrupt priority from which interrupt-safe API functions can be called from. This affects what kind of interrupts that can be "put on hold" because of critical sections created by code like shown in listing 6.4. The current version of the UC3 port of FreeRTOS does not take advantage of this. Instead, critical sections ultimately end up disabling interrupts globally.

Listing 6.4: FreeRTOS critical sections

```
taskENTER_CRITICAL();
// Atomic operations
taskEXIT_CRITICAL();
```

If the UC3 port could take full advantage of the interrupt nesting feature, it would be possible to write ISRs that could not be interrupted or deferred. On the other hand, those ISRs would not be able to use the FreeRTOS API. This is quite unrealistic for interrupt routines that must communicate with FreeRTOS task, and that is not a very unrealistic requirement.

To see if the results when using FreeRTOS could be improved, some minute changes were done to the FreeRTOS source code. All critical sections in the UC3 port of FreeRTOS port eventually calls the `portDISABLE_INTERRUPTS()` macro. For instance, `taskENTER_CRITICAL()` simply keeps track of an interrupt nesting variable and then calls `portDISABLE_INTERRUPTS()`.

In a quick attempt at improving the results when using FreeRTOS,the call to `portDISABLE_INTERRUPTS()` was substituted with a call to `DISABLE_INT_LEVEL(0)`. This way, critical sections would have no way to interfering with the execution of the ISR used in the experiment when the latter is executed under interrupt level three. This did seem to improve the maximum interrupt latency to be slightly above the maximum latency when not using an OS. This was not investigated further, because the results are purely academic: Using this method, it would not be allowed to call functions from the FreeRTOS API. However, one could reasonably assume that the reason that there was *still* a difference, was at least partly due to the timer tick interrupts from the operating system.

6.12 Conclusion

The test results show that the worst case latency is up to around 1500 clock cycles. This means that it is difficult to schedule events with an accuracy much better than a complete, but short transaction on the internal data bus. This could be a serious problem for a software implementation of I2C, but in practice it is not a problem when using the I2C module on the UC3. Still, it does mean that even if the hardware is able to monitor the bus for I2C start- and stop events, it may be difficult to use the bus effectively. For improved throughput, DMA (Direct Memory Access) should be used for both tranmission and reception of data.

In any case, the results indicate that it is likely to be difficult to predict exactly how long a bus transfer is going to take. Even if the UC3 has support for I2C in hardware, it does need software intervention during the bus transaction. And the latency for responding to these events is in the order of several bus transactions in the worst case.

I^2C driver implementation 7

Introduction

This chapter describes the implementation of an I^2C master driver for FreeRTOS, and the AVR32UC3A3256 microcontroller. This driver is intended to be the basis for implementing higher-order logic required to support resuming transfers over the bus, and breaking long transfers into smaller parts. This chapter also outlines the design of an packet-based I^2C slave driver.

7.1 I^2C master driver

The I^2C master driver is based on the concepts of messages. FreeRTOS tasks wanting to use the bus, calls wrapper functions in the driver. The operations that are supported are master read from slave, master write to slave, master write followed by read and master read followed by write. The driver functions that do bus operations creates a single "i2c message" per function invokation. This message is posted to an internal queue of i2c messages.

A separate I^2C driver task is responsible for "consuming" items in the message queue. The items are posted to an ISR, where the actual bus transfer happens. The i2c message buffer is a static array. In addition, there is a queue of pointers. The queue of pointers have room for two items fewer than the static message buffer. The driver task consumes items from the queue. The fact that the queue has the given size relative to the static message buffer, ensures that the ISR can be in the process of sending a message at the same time that someone is adding an item to the message buffer. This is illustrated in figure 7.1.

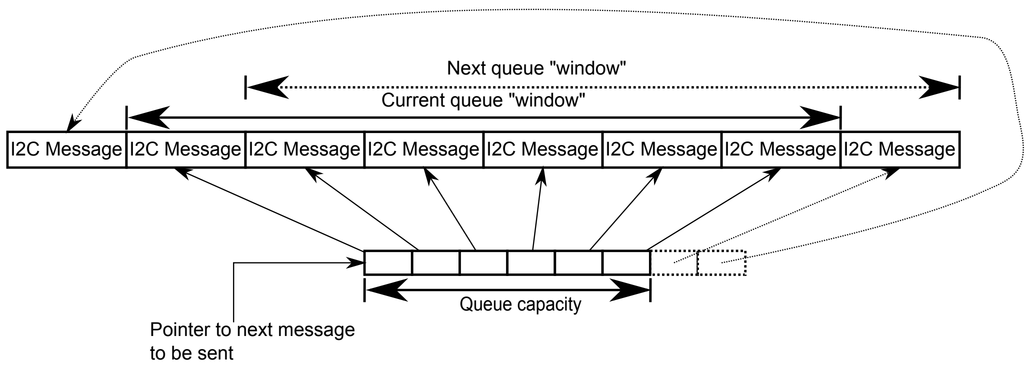


Figure 7.1: I2C master driver, principle of operation

I2C messages contain a series of 1 or more "bursts". A burst is a one-way transfer with no effective length limit. A burst is again broken down into commands. Commands are units that the hardware is able to understand. This relationship is shown in figure 7.2.

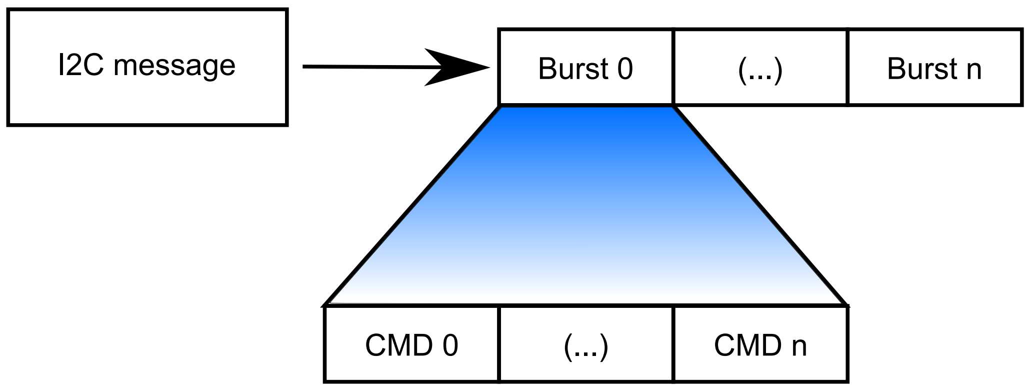


Figure 7.2: The I2C master driver breaks I2C messages into commands, that is something the hardware understands.

Testing the I^2C master driver

The i2c message to command translation are tested separately from the driver itself. This is because this functionality is self-contained and not dependent on the driver itself. This makes it possible to write unit tests for the translation-functions. An example procedure for doing this is shown in listing 7.

Listing 7.1: I^2C driver API for FreeRTOS tasks

```c
void i2c_transaction_test_single_cmd_single_burst_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[2];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0xDEADBEEF;
        message.burst[0].bufsz = 255;
        message.burst[0].direction = I2C_BURST_MASTER_READ;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[1]);

        assert_cmd_bytes_equals(command[0], 255);

        assert_cmd_saddr_equals(command[0], 0xAA);

        assert_cmd_start_set(command[0], true);

        assert_cmd_stop_set(command[0], true);

        assert_cmd_valid_set(command[0], true);

        assert_cmd_read_set(command[0], true);

        assert_cmd_rxack_set(command[0], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0xDEADBEEF);
}
```

The tests for the I^2C driver calls for the use of an actual slave module. In thos case, an XMEGA A1-Xplained evaluation board was used for testing while an UC3-A3 Xplained evaluation board contained the master under test. The test circuit is shown in figure 7.3. There are serial resistors at the pins on the master module. Using an oscilloscope, it is then possible to see what module is using the bus. The whole test setup is seen in figure 7.4.

To make the XMEGA-A1 Xplained evaluation board a suitable platform for testing, the following software had to be implemented:

- A command-line interface to enable interaction with the I^2C slave module on the microcontroller.

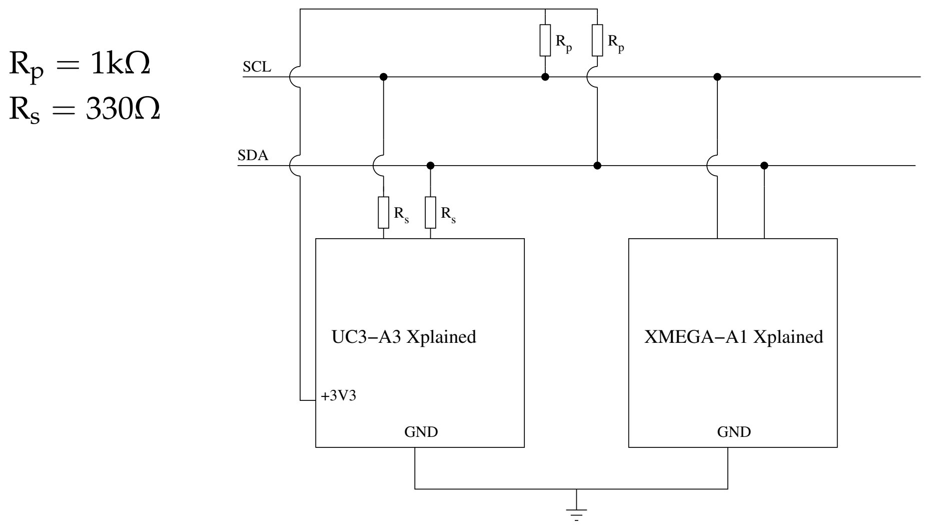


Figure 7.3: Circuit used to test the I2C master and slave driver

- Logic for I^2C slave operation. Used drivers from ASF, but they had to be modified to be acceptable for these tests.
- USART-driver. The driver is interrupt-driven, and uses ring buffers to send and receive data from and to "application code".

An example of the command line interface is shown in figure 7.5.

When the XMEGA A1-Xplained operates in slave mode, it has two modes of operation:

- Sink mode
- Normal mode

The distinguishing feature between them, is that in sink mode, the slave never NAKs anything received from the master. In normal mode, the slave NAK bytes if they lead to a buffer overflow. In sink mode, a buffer overflow is simple the same as overwriting the oldest data stored.

The integration tests for the I^2C driver assumes that data sent to the slave is also returned when the master tries to read from the slave.

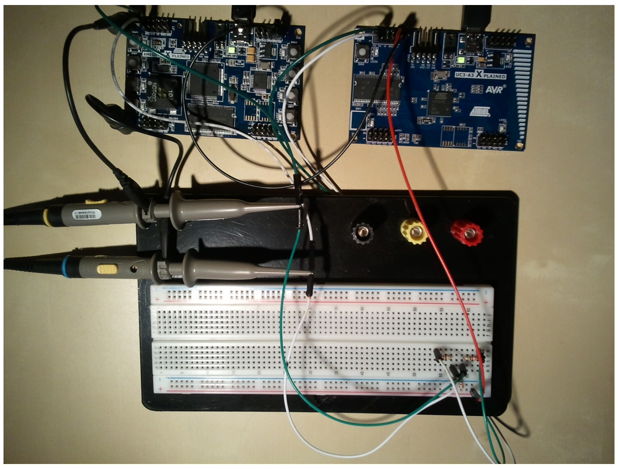
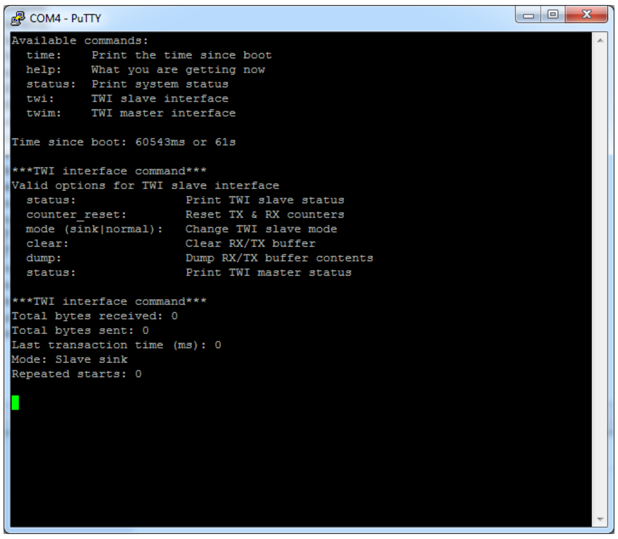


Figure 7.4: Setup for testing I2C driver



```
Available commands:
  time:    Print the time since boot
  help:    What you are getting now
  status:  Print system status
  twi:     TWI slave interface
  twim:    TWI master interface

Time since boot: 60543ms or 61s

***TWI interface command***
Valid options for TWI slave interface
  status:                Print TWI slave status
  counter_reset:         Reset TX & RX counters
  mode (sink|normal):    Change TWI slave mode
  clear:                 Clear RX/TX buffer
  dump:                  Dump RX/TX buffer contents
  status:                Print TWI master status

***TWI interface command***
Total bytes received: 0
Total bytes sent: 0
Last transaction time (ms): 0
Mode: Slave sink
Repeated starts: 0
```

Figure 7.5: The I^2C slave interface is accessible from a serial terminal program

Comparing bytes sent with bytes received, and checking bus conditions are the key element in the integration tests.

Limitations with the implementation

Observation with an oscilloscope reveals that the master does clock stretching when receiving data from slave. This indicates the the ISR takes too long to execute. Also, the driver does not utilise DMA.

The driver supports 7-bit addressing. It has not been designed for 10-

bit addresses, but it should not require major changes to enable support for this.

7.2 I^2C slave driver

Like the master driver, the slave driver is based on the concept of a queue of pointers backed by a static buffer. The driver reads received data into a packet buffer. When a complete packet has been received, it may be retrieved by a FreeRTOS task. This is shown in figure 7.6

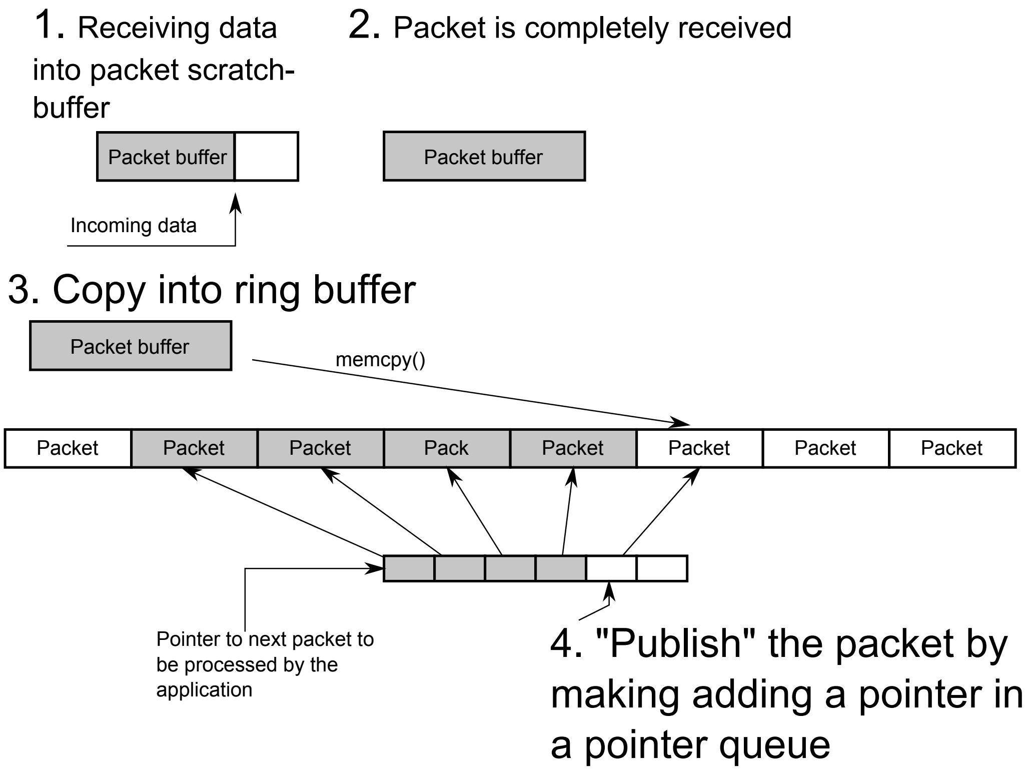

1. Receiving data into packet scratch-buffer
2. Packet is completely received
Packet buffer
Packet buffer
Incoming data
3. Copy into ring buffer
Packet buffer
memcpy()
Packet
Packet
Packet
Pack
Packet
Packet
Packet
Packet
Pointer to next packet to be processed by the application
4. "Publish" the packet by making adding a pointer in a pointer queue

Figure 7.6: I2C Slave driver architecture

Testing

The slave driver has not been tested very thoroughly. The XMEGA A1-Xplained was used as a master module. The command-line interface allows a user to fill transmit buffers with an arithmetic series or with random data.

To test the slave driver, the transmit buffer at the XMEGA was set to contain a number series. upon reception the contents of the received data could be verified by visual inspection.

The driver has not been tested in corner cases, such as loss of arbitration during slave transmitter mode, or bus errors. Tests with repeated starts have not been tested.

Part IV

Conclusion and evaluation

Conclusion and evaluation 8

The implementation and design of the software for NUTS is still in the beginning. This is particularily true if the software for the ground station is also included in the equation.

Testing and debugging in an embedded environment is challenging. In the course of this project, most of the time were spent writing tests, test code and utilities. One of the key elements in improving debugging techniques, is having a terminal interface. The Atmel USB stack was modified to support both a terminal and a data payload port. Testing of the I^2C drivers also required an elaborate framework outside the module under test. The result was the development of an I^2C master driver and a draft implementation if a slave driver.

Testing of the I^2C bus was aided by the use of a separate evaluation board. The code developed for the XMEGA A1 proved to be very useful for testing bus communication.

The main topic of this report is the NUTS bus protocol. A design for this protocol has been design. However it turns out that the constraints set by the protocol is not possible or sensible to implement at a hardware driver level. The issue of fairness must be addressed at the system level.

The work completed throughout this project gives a solid foundation for building better software abstractions. The I2C slave driver must be further developed in the future. It turns out that there are a lot of design decisions that will have to be taken, before it is possible to come up with a definite design for the slave logic.

The communications library used in this project, CSP, must be integrated with the I^2C drivers and the USB drivers. Module commands will also have to be implemented. It is also clear that an utility library for the backplane is needed.

References

[1] Espen Hugaas Andersen, ed. *HiNCube Integration and Assembly Document*. 1.5. Jan. 21, 2009.

[2] *AVR32UC Technical Reference Manual*. 32002F. Atmel. Mar. 12, 2010.

[3] Roger Birkeland. *NUTS-1 Mission Statement*. Tech. rep. NTNU, June 14, 2011.

[4] Stephen Clark. *Russian rocket fails. 18 satellites destroyed*. July 26, 2006. URL: `http://www.spaceflightnow.com/news/n0607/26dnepr/` (visited on 03/02/2012).

[5] *Clyde Space*. 2012. URL: `http://www.clyde-space.com/` (visited on 02/29/2012).

[6] *CubeSat Design Specification Rev. 12*. California Polytechnic State University, 2009.

[7] *CubeSat Kit*. 2012. URL: `http://www.cubesatkit.com/index.html` (visited on 02/29/2012).

[8] *CubeSat Kit Motherboard (MB) Data Sheet. Single Board Computer Motherboard for Harsh Environments*. Pumpkin, Inc., Sept. 2009.

[9] *CubeSTAR. A Space Space Weather Satellite Built by Students*. 2012. (Visited on 03/12/2012).

[10] Dewald de Bruyn. "Power Distribution and Conditioning for a Small Student Satellite". Design of the NUTS Backplane & EPS Module. MA thesis. NTNU, 2011.

[11] Egil Eide and Jørgen Ilstad. "NCUBE-1, the first Norwegian CUBESAT student satellite". In: *16th ESA Symposium on European Rocket and Balloon Programmes and Related Research, 2 - 5 June 2003, Sankt Gallen, Switzerland*. Ed. by Barbara Warmbein. European Space Agency, 2003.

[12] *GomSpace*. 2012. URL: http://gomspace.com/ (visited on 02/29/2012).

[13] Markus Alexander Grønstad. *Implementation of a communication protocol for CubeSTAR*. 2010.

[14] *HiNCube*. 2012. URL: http://hincube.com (visited on 04/02/2012).

[15] *ISIS - Innovative Solutions In Space*. 2012. URL: http://www.isispace.nl (visited on 04/03/2012).

[16] Lars Erik Jacobsen. *Power System of the NTNU Test Satellite. Backplane Study and Design of the EPS*. Technical Report. NTNU, Dec. 20, 2011.

[17] *nCube (satellite)*. 2012. URL: http://en.wikipedia.org/wiki/Ncube_satellite (visited on 03/05/2012).

[18] Ryan Nugent et al. *The CubeSat: The Picosatellite Standard for Research and Education*. Tech. rep. California Polytechnic State University, 2008.

[19] Espen Oland, Torbjørn Houge, and Frank Vedal. "Norwegian Student Satellite Program – HiNCube". In: *22nd Annual AIAA/USU Conference on Small Satellites*. 2008.

[20] *PC/104 Specification*. 2003.

[21] Andøya Rocket Range. *ANSAT*. 2012. URL: http://www.rocketrange.no/?page_id=254 (visited on 02/25/2012).

[22] *The I2C-Bus Specification*. 2.1. NXP Semiconductors. June 2007.

[23] *Tyvak Nano-Satellite Systems LLC*. 2012. URL: http://tyvak.com/default.html (visited on 02/29/2012).

[24] *Universal Serial Bus 3.0 specification*. 3.0. Hewlett-Packard, Intel, Microsoft, NEC, ST-NXP Wireless, and Texas Instruments. Nov. 2008.

[25] *Universal Serial Bus Class Definitions For Communications Devices*. 1.2. Nov. 2007.

[26] *Universal Serial Bus Communications Class Subclass Specification for PSTN devices*. 1.2. Feb. 2007.

[27] *Universal Serial Bus Revision 2.0 specification*. 2.0. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Apr. 2000. URL: http://www.usb.org/developers/docs/.

[28] *USB Interface Association Descriptor*. Microsoft. May 21, 2012. URL: http://msdn.microsoft.com/en-us/library/windows/hardware/ff540054%28v=vs.85%29.aspx (visited on 06/02/2012).

[29] Vilius Visockas. *Access control and securing of the NUTS uplink*. Technical Report. NTNU, Dec. 2011.

[30] Marius Voldstad. “Internal Data Bus of a Small Student Satellite”. MA thesis. NTNU, 2011.

Appendix

Configuring the OBC AVR32 UC3 project

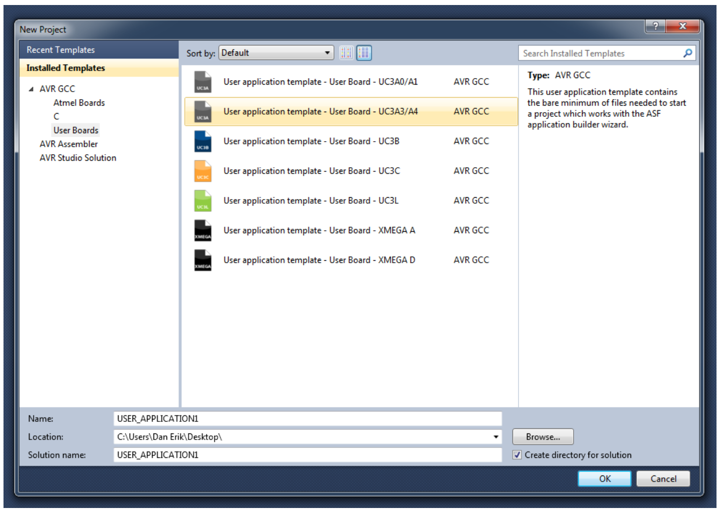
This section describes how the OBC module was initially set up, to enable development for both the OBC module itself, as well as with the UC3-A3 Xplained evaluation kit.

Setup procedure

The OBC project was initially created by using AVR Studio 5 by performing the following the steps described. Ssome steps omitted, because they are not relevant to this discussion. Starting from the beginning:

1. Create a new project. Choose "User application template - User Board - UC3A3/A4" as the template. This is shown in figure 1.
2. Create a new example project in AVR Studio 5, and store in a different location. The example project chosen was "USB Device CDC Example (from ASF V1) - EVK1100 - AT32UC3A0512". This should look similar to what is shown in figure 2.
3. Copy the empty board initialization stubs from step one into the example project. The example project is now the basis for the OBC firmware
4. Remove irrelevant parts, and code that is specific to EVK1100. Such as the use of a joystick, the use of the USART, USB to USART-bridging and the board initialization code. The "User application template"-project can now be removed.

It was found to be easier to use this example project as a starting point, instead of copying code from the example project into an empty

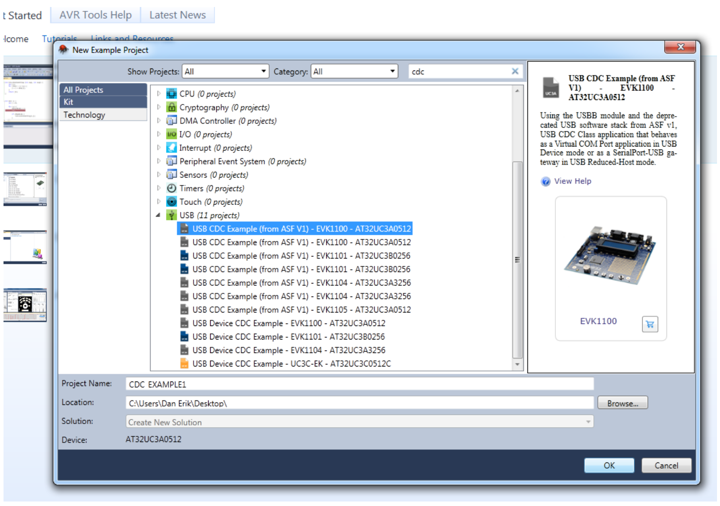


User application template - User Board - UC3A0/A1 AVR GCC
User application template - User Board - UC3A3/A4 AVR GCC
User application template - User Board - UC3B AVR GCC
User application template - User Board - UC3C AVR GCC
User application template - User Board - UC3L AVR GCC
User application template - User Board - XMEGA A AVR GCC
User application template - User Board - XMEGA D AVR GCC

Type: AVR GCC
This user application template contains the bare minimum of files needed to start a project which works with the ASF application builder wizard.

Name: USER_APPLICATION1
Location: C:\Users\Dan Erik\Desktop\
Solution name: USER_APPLICATION1
[✓] Create directory for solution

Figure 1: Creating an empty UC3A3 project



Show Projects: All Category: All cdc

CPU (0 projects)
Cryptography (0 projects)
DMA Controller (0 projects)
I/O (0 projects)
Interrupt (0 projects)
Peripheral Event System (0 projects)
Sensors (0 projects)
Timers (0 projects)
Touch (0 projects)
USB (11 projects)
USB CDC Example (from ASF V1) - EVK1100 - AT32UC3A0512
USB CDC Example (from ASF V1) - EVK1100 - AT32UC3A0512
USB CDC Example (from ASF V1) - EVK1101 - AT32UC3B0256
USB CDC Example (from ASF V1) - EVK1101 - AT32UC3B0256
USB CDC Example (from ASF V1) - EVK1104 - AT32UC3A3256
USB CDC Example (from ASF V1) - EVK1104 - AT32UC3A3256
USB CDC Example (from ASF V1) - EVK1105 - AT32UC3A0512
USB Device CDC Example - EVK1100 - AT32UC3A0512
USB Device CDC Example - EVK1101 - AT32UC3B0256
USB Device CDC Example - EVK1104 - AT32UC3A3256
USB Device CDC Example - UC3C-EK - AT32UC3C0512C

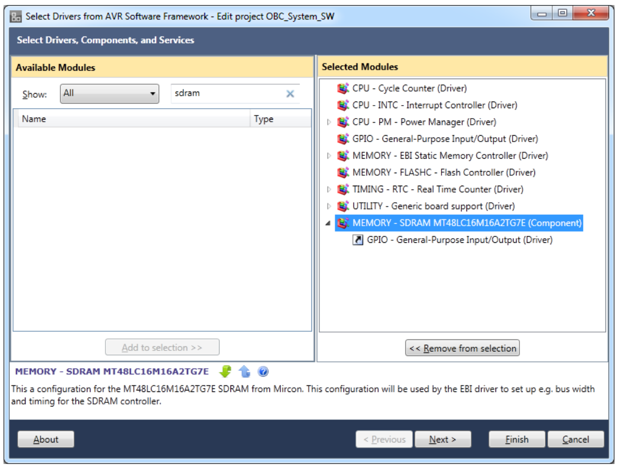
USB CDC Example (from ASF V1) - EVK1100 - AT32UC3A0512
Using the USBB module and the deprecated USB software stack from ASF v1, USB CDC Class application that behaves as a Virtual COM Port application in USB Device mode or as a SerialPort-USB gateway in USB Reduced-Host mode.

Project Name: CDC EXAMPLE1
Location: C:\Users\Dan Erik\Desktop\
Solution: Create New Solution
Device: AT32UC3A0512

Figure 2: Adding the USB CDC example projects

"user board"-project. Or start from scratch, using the an empty "user board" project, the source code from the FreeRTOS website and copying the source code for the USB stack from the example project mentioned above, into the newly created project.

After preparing the project as described above, the SDRAM controller driver was added from ASF. This is done by choosing "Project" and then clocking "Select drivers from ASF..." from within the open project in AVR Studio 5. This should bring up the dialog shown in 3, and from here it should be possible to add the driver for the memory controller.


Select Drivers from AVR Software Framework - Edit project OBC_System_SW

Select Drivers, Components, and Services

Available Modules

Show: All   sdram

Selected Modules

CPU - Cycle Counter (Driver)
CPU - INTC - Interrupt Controller (Driver)
CPU - PM - Power Manager (Driver)
GPIO - General-Purpose Input/Output (Driver)
MEMORY - EBI Static Memory Controller (Driver)
MEMORY - FLASHC - Flash Controller (Driver)
TIMING - RTC - Real Time Counter (Driver)
UTILITY - Generic board support (Driver)
MEMORY - SDRAM MT48LC16M16A2TG7E (Component)
    GPIO - General-Purpose Input/Output (Driver)

MEMORY - SDRAM MT48LC16M16A2TG7E

This a configuration for the MT48LC16M16A2TG7E SDRAM from Mircon. This configuration will be used by the EBI driver to set up e.g. bus width and timing for the SDRAM controller.

Figure 3: Creating an empty UC3A3 project

The SDRAM controller driver also includes specifications for a DRAM chip used on the Atmel EVK1100 evaluation board. After adding the SDRAM controller driver, the sdramc headers and code were wrapped behind conditional compilation based on the value of the preprocessor define ”BOARD”. This was to to prevent compilation errors as well as to not include the code when running on the ”real” OBC module.

The header containing the specifications for the DRAM chip used on UC3-A3 Xplained were placed in `src/config/mt48lc4m16a2p7e.h`. Note that they are somewhat different than those for EVK1100, or even the UC3-A3 Axplained memory example project!

The board-specific code from the UC3 A3 Xplained example project, and the user board source files, must be wrapped behind macros similar to what is shown in listing 1.

Listing 1: Conditional compilation by using preprocessor directives

```
#include ”board.h”
#if BOARD == USER_BOARD
...
#endif

or

#include ”board.h”
#if BOARD == UC3_A3_XPLAINED
...
#endif
```

Board initialization is done in `int _init_startup(void)`. This is defined in `src/thirdparty/freertos/source/portable/gcc/avr32_uc3/port.c`. Preprocessor macros make sure that only one `board_init()` is actually compiled, and calling this function results in behavior that is dependent on whether the code is compiled for the OBC or the UC3 A3 Evaluation Kit. The board initalization code sets up various peripherals, so it is important that `board_init()` is called *after* initializing the power manager and the clocks are set up and enabled.

The value of the `BOARD`-define is set as a compile time constant. This can be changed from project properties.

Requirements

This section presents the system- and software requirements, with a focus on the space segment. The requirements specifications are the first of its kind in the NUTS-project. While no "complete" requirements specification exists at this point, it is given that the software and drivers used on the OBC module must work within the confines of the existing requirements.

System requirements

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

Table 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

Table 1: (continued...)

ID	DESCRIPTION
S-6	Both ground station and satellite must be able to detect data corruption or complete loss of transmitted data

Functional requirements

The system requirements in table 1 are further refined into more specific, functional requirements as shown in 2:

Table 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

Table 2: (continued...)

ID	Description	Priority (H/M/L)
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
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 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.

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 describes the desirable properties of the system. Four main quality attributes has been identified: 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

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

Table 4: (continued...)

ID	Description	Priority (H/M/L)
NF-T3	The OBC and radio shall be able to store a program execution trace to non-volatile memory	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

Conclusion

The requirements specification is a work in progress, and it is clear that a more detailed document is needed. This is especially true for a project spanning over several years, such as the NUTS project.

I^2C master driver code listings

Listing 2: I^2C driver API for FreeRTOS tasks

```
/*
 * i2c.h
 *
 * Created: 04.06.2012 22:04:49
 *  Author: Dan Erik
 */


#ifndef I2C_H_
#define I2C_H_
#include "FreeRTOS.h"
#include "semphr.h"
#include <stddef.h>
#include "status_codes.h"

#define I2C_BAUDRATE                 100000

/* The TWI module that should be used as I2C master */
#define I2C_MODULE                               (AVR32_TWIM0)

/* IRQ number used for registering TWI interrupts with the INTC */
#define I2C_MODULE_IRQ               (AVR32_TWIM0_IRQ)

/*
```

```
 * The pins used for SDA and SCL, with their
 * GPIO function number (due to pin multiplexing)
 */
#define I2C_TWD_PIN                          TWIMS0_TWD_PIN
#define I2C_TWD_FUNCTION             TWIMS0_TWD_FUNCTION
#define I2C_TWCK_PIN                 TWIMS0_TWCK_PIN
#define I2C_TWCK_FUNCTION            TWIMS0_TWCK_FUNCTION

/* A number denoting the module whose clock should be enabled in the PM */
#define I2C_PM_CLOCK                 AVR32_TWIM0_CLK_PBA

/* The maximum number of bursts in an i2c message */
#define I2C_MAX_BURSTS               2

typedef
enum i2c_message_status {
        I2C_SUCCESS, I2C_ARBITRATION_LOST, I2C_RECV_ANAK, I2C_RECV_DNAK, I2C_AB
} i2c_message_status_t;

/* Burst  direction determines if it is reading or writing from/to a slave */
typedef
enum i2c_burst_dir {
        I2C_BURST_MASTER_READ, I2C_BURST_MASTER_WRITE
} i2c_burst_dir_t;

/*
 * Representation of a burst within a i2c message
 * A burst is simply a single-direction transfer, and
 * each burst is separated by a repeated start
 */
typedef
struct i2c_burst {
        i2c_burst_dir_t direction;      /* Determines if this burst is a read
        uint8_t *buffer;                        /* Send or receive buffer */
        size_t bufsz;                           /* Size of the send/receive bu
} i2c_burst_t;

typedef
struct i2c_message_result {
        portTickType time_started;       /* The time in ticks, when the messag
        portTickType time_completed; /* The time in ticks, when the message wa
        i2c_message_status_t status; /* Status of the transaction after it is
} i2c_message_result_t;

typedef
struct i2c_message {
        xSemaphoreHandle completion_semaphore;
        i2c_burst_t burst[I2C_MAX_BURSTS];
        uint32_t burst_count;
        uint8_t slave_address;
        i2c_message_result_t *result;
} i2c_message_t;

#define i2c_queue_read_n(buffer, bufsz, saddr, completion_handle) \
        i2c_queue_read(buffer, bufsz, saddr, completion_handle, NULL)
#define i2c_queue_write_n(buffer, bufsz, saddr, completion_handle) \
        i2c_queue_write(buffer, bufsz, saddr, completion_handle, NULL)
#define i2c_queue_write_read_n(tx_buffer, tx_bufsz, rx_buffer, rx_bufsz, saddr, compl
        i2c_queue_write_read(tx_buffer, tx_bufsz, rx_buffer, rx_bufsz, saddr, complet
#define i2c_queue_read_write_n(tx_buffer, tx_bufsz, rx_buffer, rx_bufsz, saddr, compl
        i2c_queue_read_write(tx_buffer, tx_bufsz, rx_buffer, rx_bufsz, saddr, complet

void i2c_queue_read(
        uint8_t *buffer,
        size_t bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result);

void i2c_queue_write(
        uint8_t *buffer,
        size_t bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result);

void i2c_queue_write_read(
        uint8_t *tx_buffer,
        size_t tx_bufsz,
        uint8_t *rx_buffer,
        size_t rx_bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result);

void i2c_queue_read_write(
        uint8_t *tx_buffer,
        size_t tx_bufsz,
        uint8_t *rx_buffer,
        size_t rx_bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result);

const char *i2c_message_status_str(i2c_message_status_t status);

/*
 * Starts the i2c master task. This function
 * WARNING: This function *MUST* be called prior to any other functions
 *                        in the driver!
 */
void i2c_task_init(void);

/* Returns true if the TWIM module does not use the bus now */
uint8_t i2c_is_idle(void);

/* Force stop the current transaction by sending a STOP at the next byte */
void i2c_force_stop(void);

/* Return a pointer to a string representation of a i2c message status */
const char *i2c_message_status_str(i2c_message_status_t status);

#endif /* I2C_H_ */
```

Listing 3: I^2C driver API for FreeRTOS tasks

```
/*
 * i2c.c
 *
 * Created: 04.06.2012 22:04:35
 *  Author: Dan Erik
 */
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "queue.h"

#include <stdint.h>
#include "board.h"
#include "conf_board.h"
#include "cycle_counter.h"
#include "gpio.h"
#include "pm.h"
#include "i2c.h"
#include "i2c_transaction.h"
#include "status_codes.h"
#include <atmel/twim.h>
#include "gpio.h"

/* The number of elements in the static message buffer */
#define MESSAGE_QUEUE_LENGTH 8

/* Contains the representations of i2c transactions */
static i2c_message_t messages[MESSAGE_QUEUE_LENGTH];
/* Pointer to the next element to be written in messages */
static uint32_t messages_idx = (uint32_t) 0;
/* Used to serialize access to the messages ring buffer */
```

```
xSemaphoreHandle messages_mutex = NULL;
/* Queue with pointers into the messages buffer */
static xQueueHandle message_queue = NULL;

/* Semaphore used to signal that the device is ready for a new i2c_message */
static xSemaphoreHandle device_ready = NULL;

/* Presents an active i2c transaction. Used for internal bookkeeping */
static i2c_transaction_t transaction;

/* Represents the currently active command, during an ongoing transaction */
static volatile i2c_cmd_spec_t current_cmd;

/* The next command, following the current command */
static volatile i2c_cmd_spec_t next_cmd;

/* String descriptions used in message results */
static const char *message_status_str[] = {
        "Success",
        "Arbitration lost",
        "Received address NAK",
        "Received data NAK",
        "Aborted"
};

/* The i2c task processes incoming i2c_messages and dispatches to the hw */
static void i2c_task(void *pvParameters);

static void hardware_init(void);
static void enable_muxed_pins(void);
static void init_master(void);
static void reset_disable_twim(void);
static i2c_message_status_t flag_to_status(unsigned long twim_status);
static void clear_interrupts(void);

static void queue_single_burst(
        uint8_t *buffer,
        size_t bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_burst_dir_t dir,
        i2c_message_result_t *result);

static void queue_bursts(
        i2c_burst_t *bursts,
        size_t burst_count,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result);
```

```c

uint8_t i2c_is_idle(void) {
        uint32_t status = AVR32_TWIM0.sr;
        if ((status & AVR32_TWIM_SR_IDLE_MASK)) {
                return 1;
        } else {
                return 0;
        }
}

static i2c_message_status_t flag_to_status(unsigned long twim_status) {
        i2c_message_status_t status = I2C_SUCCESS;
        if( twim_status & AVR32_TWIM_SR_ANAK_MASK ) {
                status = I2C_RECV_ANAK;
        } else if( status & AVR32_TWIM_SR_DNAK_MASK ) {
                status = I2C_RECV_DNAK;
        } else if( status & AVR32_TWIM_SR_ARBLST_MASK ) {
                status = I2C_ARBITRATION_LOST;
        } else if( status & AVR32_TWIM_SR_IDLE_MASK ) {
                status = I2C_ABORTED;
        }
        return status;
}

__attribute__((__noinline__))
static portBASE_TYPE twim_isr_nonnaked( void )
{
        portBASE_TYPE       xHigherPriorityTaskWoken = pdFALSE;
        uint8_t transaction_done = 0;
        unsigned long status = AVR32_TWIM0.sr;
        static size_t buffer_idx = 0;

        /* Check for any error conditions. Nothing further down, changes the r
        if( status & (AVR32_TWIM_SR_ANAK_MASK|AVR32_TWIM_SR_DNAK_MASK|AVR32_TW
                I2C_MODULE.NCMDR.valid = 0;
                I2C_MODULE.CMDR.valid = 0;
                transaction_done = 1;
        }

        /* Handle TX/RX within this command */
        if( buffer_idx < current_cmd.cmd_bytes ) {
                if( status & AVR32_TWIM_SR_RXRDY_MASK ) {
                        current_cmd.buffer[buffer_idx++] = I2C_MODULE.rhr;
                } else if( !current_cmd.read && (status & AVR32_TWIM_SR_TXRDY_M
                        I2C_MODULE.thr = current_cmd.buffer[buffer_idx++];
                }
        }

        /* Issue the next command, or mark the transaction as done */
        if( status & AVR32_TWIM_SR_CCOMP_MASK ) {
                if( next_cmd.cmd == 0UL ) {
                        transaction_done = 1;
                }

                if(next_cmd.read) {
                        I2C_MODULE.idr = AVR32_TWIM_IDR_TXRDY_MASK;
                        I2C_MODULE.ier = AVR32_TWIM_IER_RXRDY_MASK;
                } else {
                        I2C_MODULE.idr = AVR32_TWIM_IDR_RXRDY_MASK;
                        I2C_MODULE.ier = AVR32_TWIM_IER_TXRDY_MASK;
                }

                current_cmd = next_cmd;
                I2C_MODULE.cmdr = current_cmd.cmd;
                I2C_MODULE.scr = AVR32_TWIM_SCR_CCOMP_MASK;
                buffer_idx = 0;
        } else if( (next_cmd.cmd != 0UL) && (status & AVR32_TWIM_SR_CRDY_MASK) ) {
                next_cmd = i2c_transaction_next_cmd(&transaction);
                I2C_MODULE.ncmdr = next_cmd.cmd;
        } else if( status & AVR32_TWIM_SR_IDLE_MASK ) {
                transaction_done = 1;
        }

        if( transaction_done ) {
                if( transaction.message->completion_semaphore != NULL ) {
                        xSemaphoreGiveFromISR(transaction.message->completion_semaphor
                }

                if( transaction.message->result != NULL ) {
                        transaction.message->result->status = flag_to_status(status);
                        transaction.message->result->time_completed = xTaskGetTickCou
                }

                buffer_idx = 0;
                clear_interrupts();
                xSemaphoreGiveFromISR(device_ready, &xHigherPriorityTaskWoken);
        }

        return ( xHigherPriorityTaskWoken );
}

__attribute__((__naked__))
static void twim_isr( void )
{
        portENTER_SWITCHING_ISR();
        twim_isr_nonnaked();
        portEXIT_SWITCHING_ISR();
}
```

```

void i2c_task_init(void) {
        message_queue = xQueueCreate(MESSAGE_QUEUE_LENGTH-2, sizeof(struct i2c_

        vSemaphoreCreateBinary(device_ready);
        messages_mutex = xSemaphoreCreateMutex();

        hardware_init();

        xTaskCreate(i2c_task,
              ((const signed portCHAR *)"I2C␣Master␣task"),
              1024,
              NULL,
              tskIDLE_PRIORITY+1,
              NULL);
}

void i2c_queue_write_read(
        uint8_t *tx_buffer,
        size_t tx_bufsz,
        uint8_t *rx_buffer,
        size_t rx_bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result)
{
        i2c_burst_t tx_burst = {
                .buffer = tx_buffer, .bufsz = tx_bufsz, .direction = I2C_BURST_
        };
        i2c_burst_t rx_burst = {
                .buffer = rx_buffer, .bufsz = rx_bufsz, .direction = I2C_BURST_
        };
        i2c_burst_t bursts[] = {tx_burst, rx_burst};
        queue_bursts(bursts, 2, saddr, completion_handle, result);
}

static void queue_bursts(
        i2c_burst_t *bursts,
        size_t burst_count,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result)
{
        i2c_message_t *message;

        while( xSemaphoreTake(messages_mutex, portMAX_DELAY) != pdTRUE );

        message = &messages[messages_idx];
        messages_idx = (messages_idx+1) & (MESSAGE_QUEUE_LENGTH-1);

        message->completion_semaphore = completion_handle;
        for(size_t i = 0; i < burst_count; i++) {
                message->burst[i] = bursts[i];
        }
        message->burst_count = burst_count;
        message->slave_address = saddr;
        message->result = result;

        xQueueSend(message_queue, &message, portMAX_DELAY);

        xSemaphoreGive(messages_mutex);
}

void i2c_queue_read_write(
        uint8_t *tx_buffer,
        size_t tx_bufsz,
        uint8_t *rx_buffer,
        size_t rx_bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result)
{
        i2c_burst_t tx_burst = {
                .buffer = tx_buffer, .bufsz = tx_bufsz, .direction = I2C_BURST_MASTER_
        };
        i2c_burst_t rx_burst = {
                .buffer = rx_buffer, .bufsz = rx_bufsz, .direction = I2C_BURST_MASTER_
        };
        i2c_burst_t bursts[] = {rx_burst, tx_burst};
        queue_bursts(bursts, 2, saddr, completion_handle, result);
}

void i2c_queue_read(
        uint8_t *buffer,
        size_t bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result)
{
        queue_single_burst(
                buffer, bufsz, saddr, completion_handle, I2C_BURST_MASTER_READ, resul
        );
}

void i2c_queue_write(
        uint8_t *buffer,
        size_t bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_message_result_t *result)
{
        queue_single_burst(
                buffer, bufsz, saddr, completion_handle, I2C_BURST_MASTER_WRIT
        );
}

static void queue_single_burst(
        uint8_t *buffer,
        size_t bufsz,
        uint8_t saddr,
        xSemaphoreHandle completion_handle,
        i2c_burst_dir_t dir,
        i2c_message_result_t *result)
{
        i2c_message_t *message;
        i2c_burst_t burst = {
                .buffer = buffer, .bufsz = bufsz, .direction = dir
        };

        while( xSemaphoreTake(messages_mutex, portMAX_DELAY) != pdTRUE );

        message = &messages[messages_idx];
        messages_idx = (messages_idx+1) & (MESSAGE_QUEUE_LENGTH-1);

        message->completion_semaphore = completion_handle;
        message->burst[0] = burst;
        message->burst_count = 1;
        message->slave_address = saddr;
        message->result = result;

        xQueueSend(message_queue, &message, portMAX_DELAY);

        xSemaphoreGive(messages_mutex);
}

static void hardware_init(void) {
        enable_muxed_pins();
        init_master();
}

static void enable_muxed_pins(void) {
        const gpio_map_t twim_gpio_map = {
                {I2C_TWD_PIN, I2C_TWD_FUNCTION},
                {I2C_TWCK_PIN, I2C_TWCK_FUNCTION}
        };
        gpio_enable_module(twim_gpio_map,
                sizeof(twim_gpio_map)/sizeof(twim_gpio_map[0]))
        );
}

static void init_master(void) {
        pm_enable_module(&AVR32_PM, I2C_PM_CLOCK);

        portENTER_CRITICAL();

        I2C_MODULE.idr = ~0UL; /* Disable TWI interrupts */

        /* Enable the TWIM and reset it, in order to remove stale data */
        I2C_MODULE.cr = AVR32_TWIM_CR_MEN_MASK;
        I2C_MODULE.cr = AVR32_TWIM_CR_SWRST_MASK;

        /* Clear the status reg for old status flags, just to be sure */
        I2C_MODULE.scr = ~0UL;

        /*
         * Re-enable the TWIM again. This is to make sure that the state machine
         * assumes that the bus is idle (which may, or may not be true)
         */
        I2C_MODULE.cr = AVR32_TWIM_CR_MDIS_MASK;
        I2C_MODULE.cr = AVR32_TWIM_CR_MEN_MASK;

        INTC_register_interrupt(twim_isr, I2C_MODULE_IRQ, AVR32_INTC_INTLEVEL_INT1);

        portEXIT_CRITICAL();

        twim_set_speed(&I2C_MODULE, I2C_BAUDRATE, APPLI_PBA_SPEED);
}

static void reset_disable_twim(void) {
        I2C_MODULE.cr = AVR32_TWIM_CR_MEN_MASK;
        I2C_MODULE.cr = AVR32_TWIM_CR_SWRST_MASK;
        I2C_MODULE.cr = AVR32_TWIM_CR_MDIS_MASK;
}

static void clear_interrupts(void) {
        // Clear the interrupt flags
        I2C_MODULE.idr = ~0UL;
        // Clear the status flags
        I2C_MODULE.scr = ~0UL;
}

static void enable_twim(void) {
        I2C_MODULE.cr = AVR32_TWIM_CR_MEN_MASK;
}

static void i2c_task(void *pvParameters) {
        i2c_message_t *current_message;
        while(1) {
                while( xQueueReceive(message_queue, &current_message, portMAX_D

                while( xSemaphoreTake(device_ready, portMAX_DELAY) != pdTRUE )

                if( current_message->result != NULL ) {
                        current_message->result->time_started = xTaskGetTickCo
                }

                i2c_transaction_init(&transaction, current_message);

                taskENTER_CRITICAL();
                reset_disable_twim();
                clear_interrupts();

                current_cmd = i2c_transaction_next_cmd(&transaction);
                next_cmd = i2c_transaction_next_cmd(&transaction);
                I2C_MODULE.cmdr = current_cmd.cmd;
                I2C_MODULE.ncmdr = next_cmd.cmd;
                I2C_MODULE.ier = AVR32_TWIM_IER_CCOMP_MASK
                                                | AVR32_TWIM_IER_CRDY_MASK
                                                | AVR32_TWIM_IER_ANAK_MASK
                                                | AVR32_TWIM_IER_DNAK
                                                | AVR32_TWIM_IER_IDLE_MASK
                                                | ((current_cmd.read ? 0 : 1) <
                                                | ((current_cmd.read ? 1 : 0) <
                                                | AVR32_TWIM_IER_ARBLST_MASK;
                enable_twim();
                taskEXIT_CRITICAL();

        }
        vTaskDelete(NULL);
}

/*
 * Try to force the TWI module to send a stop signal.
 * This is an alternative to resetting the TWI, because other masters
 * should notice that the stop signal is sent, meaning that the bus
 * is available for others to use
 */
void i2c_force_stop(void) {

        if( !(I2C_MODULE.sr & AVR32_TWIM_SR_IDLE_MASK) ) {
                I2C_MODULE.cr = AVR32_TWIM_CR_STOP_MASK;
        }
}

const char *i2c_message_status_str(i2c_message_status_t status) {
```

```c
        return message_status_str[(int)status];
}
```

I^2C transaction listings

Listing 4: I^2C driver API for FreeRTOS tasks

```c
/*
 * i2c_transaction.h
 *
 * Created: 10.06.2012 04:47:24
 *  Author: Dan Erik
 */


#ifndef I2C_TRANSACTION_H_
#define I2C_TRANSACTION_H_

#include <stddef.h>
#include <stdint.h>
#include "compiler.h"
#include "i2c.h"


typedef unsigned long twi_cmd_t;

typedef
struct i2c_cmd_spec {
        twi_cmd_t cmd;
        uint8_t *buffer;
        uint8_t cmd_bytes;
        uint8_t read;
} i2c_cmd_spec_t;

typedef
struct i2c_transaction {
        uint32_t burst_idx; /* The current burst number*/
        uint32_t burst_count; /* "Const" for the whole transaction */
        size_t burst_buf_idx; /* The offset into the burst_buffer into which data from
        uint8_t *burst_buffer; /* "Const" within burst */
        size_t burst_bufsz; /* "Const" within burst */
        i2c_burst_t *burst;
        uint8_t saddr; /* "Const" for whole transaction*/
        i2c_cmd_spec_t cmd;
        i2c_message_t *message; /* "Const" for whole transaction*/
        uint8_t last_cmd_sz;
        uint8_t burst_read;
} i2c_transaction_t;
```

```c


void i2c_transaction_init(i2c_transaction_t *transaction, i2c_message_t *messa

i2c_cmd_spec_t i2c_transaction_next_cmd(i2c_transaction_t *transaction);



#endif /* I2C_TRANSACTION_H_ */
```

Listing 5: I^2C driver API for FreeRTOS tasks

```c
/*
 * i2c_transaction.c
 *
 * Created: 10.06.2012 04:47:12
 *  Author: Dan Erik
 */

#include <stdint.h>
#include "compiler.h"
#include "i2c.h"
#include "i2c_transaction.h"

static void transaction_update_burst_variables(i2c_transaction_t *transaction)
static bool transaction_update_burst_offset(i2c_transaction_t *transaction, si

void i2c_transaction_init(i2c_transaction_t *transaction, i2c_message_t *messa
        transaction->burst_idx = 0;
        transaction->burst_count = message->burst_count;
        transaction->burst_buf_idx = 0;
        transaction->burst = message->burst;
        transaction->burst_buffer = transaction->burst[0].buffer;
        transaction->saddr = message->slave_address;
        transaction->burst_bufsz =      transaction->burst[0].bufsz;
        transaction->last_cmd_sz = 0;
        transaction->burst_read = transaction->burst[0].direction == I2C_BURST_
        transaction->message = message;
}

i2c_cmd_spec_t i2c_transaction_next_cmd(i2c_transaction_t *transaction) {
        i2c_cmd_spec_t ret = {.cmd = 0UL, .buffer = NULL, .cmd_bytes = 0, .rea
        if( transaction_update_burst_offset(transaction, transaction->last_cmd
                return ret;

        size_t remaining_burst_bytes = transaction->burst_bufsz - transaction-
        uint8_t cmd_bytes = min(UINT8_MAX, remaining_burst_bytes);
        uint8_t read_cmd = transaction->burst_read;
        uint8_t first_burst_cmd = transaction->burst_buf_idx == 0;
        uint8_t last_burst = transaction->burst_idx == (transaction->burst_count -1);
        uint8_t last_burst_cmd = (remaining_burst_bytes-cmd_bytes) == 0;
        uint8_t last_cmd =  last_burst && last_burst_cmd;

        twi_cmd_t cmd = (transaction->saddr << AVR32_TWIM_CMDR_SADR_OFFSET)
                        | (cmd_bytes << AVR32_TWIM_CMDR_NBYTES_OFFSET)
                        | (AVR32_TWIM_CMDR_VALID_MASK)
                        | ((first_burst_cmd ? 1 : 0) << AVR32_TWIM_CMDR_START_OFFSET)
                        | ((last_burst_cmd ? 0 : 1) << AVR32_TWIM_CMDR_ACKLAST_OFFSET
                        | ((last_cmd ? 1 : 0) << AVR32_TWIM_CMDR_STOP_OFFSET)
                        | ((read_cmd ? 1 : 0) << AVR32_TWIM_CMDR_READ_OFFSET);

        transaction->last_cmd_sz = cmd_bytes;

        ret.cmd = cmd;
        ret.cmd_bytes = cmd_bytes;
        ret.read = read_cmd;
        ret.buffer = transaction->burst_buffer + transaction->burst_buf_idx;

        return ret;
}

static bool transaction_update_burst_offset(i2c_transaction_t *transaction, size_t of
        bool past_end = false;

        transaction->burst_buf_idx += offset;

        if( transaction->burst_buf_idx >= transaction->burst_bufsz ) {
                transaction->burst_buf_idx = 0;

                past_end = ++transaction->burst_idx >= transaction->burst_count;

                if(!past_end) {
                        transaction_update_burst_variables(transaction);
                }
        }
        return past_end;
}


static void transaction_update_burst_variables(i2c_transaction_t *transaction) {
        uint8_t i = transaction->burst_idx;
        transaction->burst_buffer = transaction->burst[i].buffer;
        transaction->burst_read = transaction->burst[i].direction == I2C_BURST_MASTER_
        transaction->burst_bufsz =      transaction->burst[i].bufsz;
}
```

I^2C transaction tests

Listing 6: I^2C driver API for FreeRTOS tasks

```c
/*
 * i2c_transaction_test.h
 *
 * Created: 10.06.2012 04:54:19
 *  Author: Dan Erik
 */


#ifndef I2C_TRANSACTION_TEST_H_
#define I2C_TRANSACTION_TEST_H_

void i2c_transaction_test_run_all(void);

void i2c_transaction_test_single_cmd_single_burst_tx(void);

void i2c_transaction_test_single_cmd_single_burst_rx(void);

void i2c_transaction_test_double_cmd_single_burst_rx(void);

void i2c_transaction_test_double_cmd_single_burst_tx(void);

void i2c_transaction_test_triple_cmd_single_burst_tx(void);

void i2c_transaction_test_triple_cmd_single_burst_rx(void);

void i2c_transaction_test_triple_cmd_double_burst_rx(void);

void i2c_transaction_test_triple_cmd_double_burst_tx(void);

void i2c_transaction_test_quad_cmd_double_burst_rx_tx(void);

void i2c_transaction_test_quad_cmd_double_burst_tx_rx(void);

void i2c_transaction_test_quad_cmd_double_burst_tx_tx(void);

void i2c_transaction_test_quad_cmd_double_burst_rx_rx(void);

#endif /* I2C_TRANSACTION_TEST_H_ */
```

Listing 7: I^2C driver API for FreeRTOS tasks

```c
/*
 * i2c_transaction_test.c
 *
 * Created: 10.06.2012 04:54:12
 *  Author: Dan Erik
 */

```

```
#include <stdint.h>
#include <avr32/io.h>
#include "compiler.h"
#include "i2c.h"
#include "i2c_transaction.h"
#include "i2c_transaction_test.h"

static void assert_cmd_empty(i2c_cmd_spec_t cmd);
static void assert_cmd_start_set(i2c_cmd_spec_t cmd, bool start);
static void assert_cmd_stop_set(i2c_cmd_spec_t cmd, bool stop);
static void assert_cmd_valid_set(i2c_cmd_spec_t cmd, bool valid);
static void assert_cmd_read_set(i2c_cmd_spec_t cmd, bool read);
static void assert_cmd_rxack_set(i2c_cmd_spec_t cmd, bool acklast);
static void assert_cmd_saddr_equals(i2c_cmd_spec_t cmd, uint8_t saddr);
static void assert_cmd_bytes_equals(i2c_cmd_spec_t cmd, uint8_t nbytes);
static void assert_cmd_buffer_equals(i2c_cmd_spec_t cmd, uint8_t *ptr);


void i2c_transaction_test_single_cmd_single_burst_tx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[2];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0xDEADBEEF;
        message.burst[0].bufsz = 255;
        message.burst[0].direction = I2C_BURST_MASTER_WRITE;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[1]);

        assert_cmd_bytes_equals(command[0], 255);

        assert_cmd_saddr_equals(command[0], 0xAA);

        assert_cmd_start_set(command[0], true);

        assert_cmd_stop_set(command[0], true);

        assert_cmd_valid_set(command[0], true);

        assert_cmd_read_set(command[0], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0xDEADBEEF);
}
```

```

void i2c_transaction_test_single_cmd_single_burst_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[2];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0xDEADBEEF;
        message.burst[0].bufsz = 255;
        message.burst[0].direction = I2C_BURST_MASTER_READ;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[1]);

        assert_cmd_bytes_equals(command[0], 255);

        assert_cmd_saddr_equals(command[0], 0xAA);

        assert_cmd_start_set(command[0], true);

        assert_cmd_stop_set(command[0], true);

        assert_cmd_valid_set(command[0], true);

        assert_cmd_read_set(command[0], true);

        assert_cmd_rxack_set(command[0], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0xDEADBEEF);
}

void i2c_transaction_test_double_cmd_single_burst_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[3];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 256;
        message.burst[0].direction = I2C_BURST_MASTER_READ;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[2]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 1);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);

        assert_cmd_read_set(command[0], true);
        assert_cmd_read_set(command[1], true);

        assert_cmd_rxack_set(command[0], true);
        assert_cmd_rxack_set(command[1], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
}

void i2c_transaction_test_double_cmd_single_burst_tx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[3];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 256;
        message.burst[0].direction = I2C_BURST_MASTER_WRITE;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[2]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 1);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);

        assert_cmd_read_set(command[0], false);
        assert_cmd_read_set(command[1], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
}

void i2c_transaction_test_triple_cmd_single_burst_tx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[4];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 512;
        message.burst[0].direction = I2C_BURST_MASTER_WRITE;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[3]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 255);
        assert_cmd_bytes_equals(command[2], 2);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);

        assert_cmd_read_set(command[0], false);
        assert_cmd_read_set(command[1], false);
        assert_cmd_read_set(command[2], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)(0x20 + 255 + 255));
}

void i2c_transaction_test_triple_cmd_single_burst_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[4];

        message.burst_count = 1;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 512;
        message.burst[0].direction = I2C_BURST_MASTER_READ;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[3]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 255);
        assert_cmd_bytes_equals(command[2], 2);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);

        assert_cmd_read_set(command[0], true);
        assert_cmd_read_set(command[1], true);
        assert_cmd_read_set(command[2], true);

        assert_cmd_rxack_set(command[0], true);
        assert_cmd_rxack_set(command[1], true);
        assert_cmd_rxack_set(command[2], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)(0x20 + 255 + 255));
}

void i2c_transaction_test_triple_cmd_double_burst_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[4];

        message.burst_count = 2;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 510;
        message.burst[0].direction = I2C_BURST_MASTER_READ;
        message.burst[1].buffer = (uint8_t*)0x2000;
        message.burst[1].bufsz = 255;
        message.burst[1].direction = I2C_BURST_MASTER_READ;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[3]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 255);
        assert_cmd_bytes_equals(command[2], 255);

        assert_cmd_saddr_equals(command[0], 0xAA);
```

```c
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], true);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);

        assert_cmd_read_set(command[0], true);
        assert_cmd_read_set(command[1], true);
        assert_cmd_read_set(command[2], true);

        assert_cmd_rxack_set(command[0], true);
        assert_cmd_rxack_set(command[1], false);
        assert_cmd_rxack_set(command[2], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)0x2000);
}

void i2c_transaction_test_triple_cmd_double_burst_tx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[4];

        message.burst_count = 2;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 510;
        message.burst[0].direction = I2C_BURST_MASTER_WRITE;
        message.burst[1].buffer = (uint8_t*)0x2000;
        message.burst[1].bufsz = 255;
        message.burst[1].direction = I2C_BURST_MASTER_WRITE;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[3]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 255);
        assert_cmd_bytes_equals(command[2], 255);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], true);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);

        assert_cmd_read_set(command[0], false);
        assert_cmd_read_set(command[1], false);
        assert_cmd_read_set(command[2], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)0x2000);
}

void i2c_transaction_test_quad_cmd_double_burst_rx_tx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[5];

        message.burst_count = 2;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 510;
        message.burst[0].direction = I2C_BURST_MASTER_READ;
        message.burst[1].buffer = (uint8_t*)0x2000;
        message.burst[1].bufsz = 510;
        message.burst[1].direction = I2C_BURST_MASTER_WRITE;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);
        command[4] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[4]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 255);
        assert_cmd_bytes_equals(command[2], 255);
        assert_cmd_bytes_equals(command[3], 255);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);
        assert_cmd_saddr_equals(command[3], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], true);
        assert_cmd_start_set(command[3], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], false);
        assert_cmd_stop_set(command[3], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);
        assert_cmd_valid_set(command[3], true);

        assert_cmd_read_set(command[0], true);
        assert_cmd_read_set(command[1], true);
        assert_cmd_read_set(command[2], false);
        assert_cmd_read_set(command[3], false);

        assert_cmd_rxack_set(command[0], true);
        assert_cmd_rxack_set(command[1], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)0x2000);
        assert_cmd_buffer_equals(command[3], (uint8_t*)(0x2000 + 255));
}

void i2c_transaction_test_quad_cmd_double_burst_tx_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[5];

        message.burst_count = 2;
```

```
    message.slave_address = 0xAA;
    message.burst[0].buffer = (uint8_t*)0x20;
    message.burst[0].bufsz = 510;
    message.burst[0].direction = I2C_BURST_MASTER_WRITE;
    message.burst[1].buffer = (uint8_t*)0x2000;
    message.burst[1].bufsz = 510;
    message.burst[1].direction = I2C_BURST_MASTER_READ;

    i2c_transaction_init(&transaction, &message);
    command[0] = i2c_transaction_next_cmd(&transaction);
    command[1] = i2c_transaction_next_cmd(&transaction);
    command[2] = i2c_transaction_next_cmd(&transaction);
    command[3] = i2c_transaction_next_cmd(&transaction);
    command[4] = i2c_transaction_next_cmd(&transaction);

    assert_cmd_empty(command[4]);

    assert_cmd_bytes_equals(command[0], 255);
    assert_cmd_bytes_equals(command[1], 255);
    assert_cmd_bytes_equals(command[2], 255);
    assert_cmd_bytes_equals(command[3], 255);

    assert_cmd_saddr_equals(command[0], 0xAA);
    assert_cmd_saddr_equals(command[1], 0xAA);
    assert_cmd_saddr_equals(command[2], 0xAA);
    assert_cmd_saddr_equals(command[3], 0xAA);

    assert_cmd_start_set(command[0], true);
    assert_cmd_start_set(command[1], false);
    assert_cmd_start_set(command[2], true);
    assert_cmd_start_set(command[3], false);

    assert_cmd_stop_set(command[0], false);
    assert_cmd_stop_set(command[1], false);
    assert_cmd_stop_set(command[2], false);
    assert_cmd_stop_set(command[3], true);

    assert_cmd_valid_set(command[0], true);
    assert_cmd_valid_set(command[1], true);
    assert_cmd_valid_set(command[2], true);
    assert_cmd_valid_set(command[3], true);

    assert_cmd_read_set(command[0], false);
    assert_cmd_read_set(command[1], false);
    assert_cmd_read_set(command[2], true);
    assert_cmd_read_set(command[3], true);

    assert_cmd_rxack_set(command[2], true);
    assert_cmd_rxack_set(command[3], false);
```

```

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)0x2000);
        assert_cmd_buffer_equals(command[3], (uint8_t*)(0x2000 + 255));
}

void i2c_transaction_test_quad_cmd_double_burst_tx_tx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[5];

        message.burst_count = 2;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 260;
        message.burst[0].direction = I2C_BURST_MASTER_WRITE;
        message.burst[1].buffer = (uint8_t*)0x2000;
        message.burst[1].bufsz = 260;
        message.burst[1].direction = I2C_BURST_MASTER_WRITE;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);
        command[4] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[4]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 5);
        assert_cmd_bytes_equals(command[2], 255);
        assert_cmd_bytes_equals(command[3], 5);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);
        assert_cmd_saddr_equals(command[3], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], true);
        assert_cmd_start_set(command[3], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], false);
        assert_cmd_stop_set(command[3], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);
        assert_cmd_valid_set(command[3], true);

        assert_cmd_read_set(command[0], false);
        assert_cmd_read_set(command[1], false);
        assert_cmd_read_set(command[2], false);
        assert_cmd_read_set(command[3], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)0x2000);
        assert_cmd_buffer_equals(command[3], (uint8_t*)(0x2000 + 255));
}

void i2c_transaction_test_quad_cmd_double_burst_rx_rx(void) {
        i2c_message_t message;
        i2c_transaction_t transaction;
        i2c_cmd_spec_t command[5];

        message.burst_count = 2;
        message.slave_address = 0xAA;
        message.burst[0].buffer = (uint8_t*)0x20;
        message.burst[0].bufsz = 260;
        message.burst[0].direction = I2C_BURST_MASTER_READ;
        message.burst[1].buffer = (uint8_t*)0x2000;
        message.burst[1].bufsz = 260;
        message.burst[1].direction = I2C_BURST_MASTER_READ;

        i2c_transaction_init(&transaction, &message);
        command[0] = i2c_transaction_next_cmd(&transaction);
        command[1] = i2c_transaction_next_cmd(&transaction);
        command[2] = i2c_transaction_next_cmd(&transaction);
        command[3] = i2c_transaction_next_cmd(&transaction);
        command[4] = i2c_transaction_next_cmd(&transaction);

        assert_cmd_empty(command[4]);

        assert_cmd_bytes_equals(command[0], 255);
        assert_cmd_bytes_equals(command[1], 5);
        assert_cmd_bytes_equals(command[2], 255);
        assert_cmd_bytes_equals(command[3], 5);

        assert_cmd_saddr_equals(command[0], 0xAA);
        assert_cmd_saddr_equals(command[1], 0xAA);
        assert_cmd_saddr_equals(command[2], 0xAA);
        assert_cmd_saddr_equals(command[3], 0xAA);

        assert_cmd_start_set(command[0], true);
        assert_cmd_start_set(command[1], false);
        assert_cmd_start_set(command[2], true);
        assert_cmd_start_set(command[3], false);

        assert_cmd_stop_set(command[0], false);
        assert_cmd_stop_set(command[1], false);
        assert_cmd_stop_set(command[2], false);
        assert_cmd_stop_set(command[3], true);

        assert_cmd_valid_set(command[0], true);
        assert_cmd_valid_set(command[1], true);
        assert_cmd_valid_set(command[2], true);
        assert_cmd_valid_set(command[3], true);

        assert_cmd_read_set(command[0], true);
        assert_cmd_read_set(command[1], true);
        assert_cmd_read_set(command[2], true);
        assert_cmd_read_set(command[3], true);

        assert_cmd_rxack_set(command[0], true);
        assert_cmd_rxack_set(command[1], false);
        assert_cmd_rxack_set(command[2], true);
        assert_cmd_rxack_set(command[3], false);

        assert_cmd_buffer_equals(command[0], (uint8_t*)0x20);
        assert_cmd_buffer_equals(command[1], (uint8_t*)(0x20 + 255));
        assert_cmd_buffer_equals(command[2], (uint8_t*)0x2000);
        assert_cmd_buffer_equals(command[3], (uint8_t*)(0x2000 + 255));
}

void i2c_transaction_test_run_all(void) {
        irqflags_t flags = cpu_irq_save();
        i2c_transaction_test_single_cmd_single_burst_tx();
        i2c_transaction_test_single_cmd_single_burst_rx();
        i2c_transaction_test_double_cmd_single_burst_rx();
        i2c_transaction_test_double_cmd_single_burst_tx();
        i2c_transaction_test_triple_cmd_single_burst_tx();
        i2c_transaction_test_triple_cmd_single_burst_rx();
        i2c_transaction_test_triple_cmd_double_burst_rx();
        i2c_transaction_test_triple_cmd_double_burst_tx();
        i2c_transaction_test_quad_cmd_double_burst_rx_tx();
        i2c_transaction_test_quad_cmd_double_burst_tx_rx();
        i2c_transaction_test_quad_cmd_double_burst_tx_tx();
        i2c_transaction_test_quad_cmd_double_burst_rx_rx();
        cpu_irq_restore(flags);
}

```

```c
static void assert_cmd_empty(i2c_cmd_spec_t cmd) {
        if( cmd.cmd != 0UL )
                __builtin_breakpoint();
}

static void assert_cmd_start_set(i2c_cmd_spec_t cmd, bool start) {
        bool start_set = (cmd.cmd & AVR32_TWIM_CMDR_START_MASK) != 0;
        if( start_set != start )
                __builtin_breakpoint();
}

static void assert_cmd_stop_set(i2c_cmd_spec_t cmd, bool stop) {
        bool stop_set = (cmd.cmd & AVR32_TWIM_CMDR_STOP_MASK) != 0;
        if( stop_set != stop )
                __builtin_breakpoint();
}

static void assert_cmd_valid_set(i2c_cmd_spec_t cmd, bool valid) {
        bool valid_set = (cmd.cmd & AVR32_TWIM_CMDR_VALID_MASK) != 0;
        if( valid_set != valid )
                __builtin_breakpoint();
}

static void assert_cmd_read_set(i2c_cmd_spec_t cmd, bool read) {
        bool read_set = (cmd.cmd & AVR32_TWIM_CMDR_READ_MASK) != 0;

        if( read_set != read )
                __builtin_breakpoint();
}

static void assert_cmd_rxack_set(i2c_cmd_spec_t cmd, bool acklast) {
        bool acklast_set = (cmd.cmd & AVR32_TWIM_CMDR_ACKLAST_MASK) != 0;

        if( acklast_set != acklast )
                __builtin_breakpoint();
}

static void assert_cmd_saddr_equals(i2c_cmd_spec_t cmd, uint8_t saddr) {
        if( ((cmd.cmd & AVR32_TWIM_CMDR_SADR_MASK) >> AVR32_TWIM_CMDR_SADR_OFF
                __builtin_breakpoint();
}

static void assert_cmd_bytes_equals(i2c_cmd_spec_t cmd, uint8_t nbytes) {
        if( cmd.cmd_bytes != nbytes )
                __builtin_breakpoint();

        if( ((cmd.cmd & AVR32_TWIM_CMDR_NBYTES_MASK) >> AVR32_TWIM_CMDR_NBYTES
                __builtin_breakpoint();
}
```

```

static void assert_cmd_buffer_equals(i2c_cmd_spec_t cmd, uint8_t *ptr) {
        if( cmd.buffer != ptr )
                __builtin_breakpoint();
}
```