View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HSN Open Archive

NTNU – Trondheim
Norwegian University of
Science and Technology

Software quality assurance on a Student Satellite

The NTNU Test Satellite

Torleif Ajer Thomassen

Master of Science in Informatics
Submission date: June 2013
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

The NTNU Test Satellite(NUTS) is set to launch in 2014. Its primary mission is to take pictures of atmospheric gravity waves from orbit using an infrared camera. The satellite will consist of five primary modules and a backplane. Every module will be software controlled to a smaller or greater degree. Space is a hostile environment and once the satellite is launched it will be difficult or even impossible to make changes to the software. This thesis aims to develop and evaluate methods on how to help ensure that the satellite's software is sufficiently robust for it mission. The most important observation to make about the NUTS project is that up until very recently, all the work being done was done by students working on their master thesis. This presents several problems. Firstly, a master thesis does not naturally fit into a project of this type. A thesis is quite often grounded in theoretical work, software produced is often proof of concept and may lack the desired robustness. Secondly, a student only works on their thesis for ten to six months before they graduate and move on. What they leave behind is mainly their paper and some proof of concept code. These two issues makes producing code viable for the mission a slow and arduous task.
In 2012 the NUTS project group added a volunteer student branch. Their task is to help the project in various ways, including software development. The project also has an "Experts in Teams" village that do projects more or less related to NUTS. While this will make it easier to produce software, the project still lacks a well defined approach to software development. Questions that arise are for instance: Where do we start? What are the different software modules we need? What are the requirements for this software module? How far along are we with this software module?
My thesis is not about project management as a whole. But rather a subset of tools and methods that can be helpful during the software development lifecycle.

Abstract

The NTNU student test satellite project aims to launch a satellite into low earth orbit in 2014. Its goal is to make observations of atmospheric gravity waves with an infrared camera. The satellite is a interdisciplinary project where students from several departments have come together to build it. So far the work on the satellite software has not been on the forefront of this project and my thesis aimed aid in the development process.

I split my thesis work into two areas of focus. The first area was to find methods to help developing software through planning and documentation. As such I created a *Software Design Document*(SDD) *template* based on an IEEE standard. I focused on making the template fit the NUTS project group. I wanted a template that could be applicable on any level of abstraction meaning that a SDD template can be used on high level software descriptions as well as very specific low level ones. This became a question of finding the common denominators for all software systems regardless of size and complexity. The final SDD template works in such a way that developers can document the whole system by splitting it up into separate SDD files that are logically linked together. It also makes it a lot easier to pass knowledge down from one developer to another.

The second area of focus was on creating a *Software Support Service* that would run seamlessly in the background and give developers valuable reports on the software. This service would work by analysing the source code at given time intervals and apply different tools to the source code. For the first version of this service I planned on implementing two tools: A code documentation system and a static code analysis tool. For code documentation I ended up choosing Doxygen and for static code analysis I ended up with Clang Static Analyser(CSA). Integrating Doxygen into my service proved to be a fairly easy task and it worked well with the NUTS source code. However getting CSA to work with the source code proved to be a much larger and more complex issue than I had anticipated. In the end I was forced to abandon the work on integrating it into my service. I however do still believe that a static analysis tool for the service would be of great help to the project and dedicated parts of this thesis report to elaborate on the subject. I did not fully reach my goals with this service, however it does run and is capable of generating documentation of the NUTS source code.

Sammendrag

NTNU sitt student satellitt program, eller NUTS, sikter på å sende en satellitt inn i lav bane rundt jorden innen 2014. Målet med satellitten er å observere atmosfæriske gravitasjonsbølger med et infrarødt kamera. Satellitten er et interdisiplinært prosjekt hvor studenter fra flere institutter er med å bygge denne satellitten. Software utvikling på satellitten har så langt ikke kommet langt og min masteroppgave handler om å finne måter å gjøre det enklere å utvikle software for satellitten.

Jeg delte arbeidet mitt inn i to biter. Den første biten var å finne metoder for å hjelpe softwareutviklingen ved hjelp av planlegging og dokumentering. Jeg begynte derfor å jobbe å utvikle en mal for *Software Design Documents*(SDD) som skulle gjøre det mulig å beskrive software moduler på forskjellige abstraksjonsnivå. Denne malen var basert på en langt mer generisk IEEE standard som jeg brukte som basis. Altså at malen skulle kunne brukes til å beskrive høynivå software moduler såvel som spesifikke implementasjoner av eksempelvis metoder eller klasser. Dette ble et spørsmål om å finne fellesnevnere for all type software uansett størrelse og kompleksitet. SDD malen jeg har laget fungerer på en slik måte at utviklere kan lett dokumentere hele systemet ved splitte systemet inn i flere SDD-er som er logisk koblet sammen. Den gjør det også enklere for utviklere å videreføre kunnskap til den neste generasjonen av utviklere.

Den andre biten av oppgaven min fokuserte på å bygge en *software support*-tjeneste som ville jobbe i bakgrunnen og hjelpe utviklere ved å gi verdifulle rapporter basert på kildekoden. Tjenesten ville fungere ved å analysere koden ved gitte tidspunkt ved å bruke forskjellige verktøy integrert i tjenesten. Som en start valgte jeg å fokusere på to verktøy: Et verktøy som auto-dokumenterte kildekoden og et verktøy som gjorde statiske analyser av koden. Verktøyene jeg endte opp å velge var Doxygen for auto-dokumentering og Clang static analyser (CSA) for statisk analyse. Jeg klarte å integrere Doxygen inn i tjenesten min ganske greit og den fungerte godt sammen med kildekoden for NUTS. Men å få CSA til å fungere på NUTS kildekoden viste seg å være en langt større og mer kompleks utfordring enn jeg hadde forutsett. Tilslutt ble jeg tvunget til å avslutte arbeidet mitt på dette for å kunne fokusere på andre områder. Jeg valgte likevel å dedikere en del av denne rapporten på statisk analyse og CSA fordi jeg tror det være et veldig nyttig verktøy for software utvikling på NUTS. Jeg klarte da altså ikke å fult nå de målene jeg hadde sett meg ut for denne tjenesten, men til tross for dette så er tjenesten operativ og kan generere kildekodedokumentasjon for NUTS prosjektet.

Table of Contents

Problem description **1**

Abstract **i**

Sammendrag **i**

Table of Contents **iii**

List of Tables **v**

List of Figures **vii**

Abbreviations **viii**

1 Introduction **1**
1.1 Software quality 1
1.2 Thesis aims 1
1.3 Previous work 2
1.4 Thesis outline 2

2 Background **3**
2.1 The NTNU Test Satellite 3
2.1.1 History 5
2.1.2 The NUTS Mission 5
2.1.3 Mechanical structure 6
2.2 NUTS Hardware 6
2.2.1 Backplane 7
2.2.2 On board computer/controller (OBC) 8
2.2.3 Radio 8
2.2.4 Attitude Determination & Control System (ADCS) 9
2.2.5 Power distribution and EPS 9
2.2.6 Payload 9

2.2.7 Ground station and communication 10

3 Software development **11**
3.1 Software development and quality assurance 11
3.1.1 Software quality 11
3.2 NUTS Software and development 12
3.2.1 Challenges 12
3.2.2 Source code management 13
3.2.3 The C language 13

4 Software Design Descriptions **15**
4.1 Rationale 15
4.2 An introduction to SDDs 16
4.2.1 Terminology 16
4.2.2 How SDDs work 17
4.3 SDD in the software life cycle 20
4.4 The NUTS SDD template proposal 20
4.4.1 NUTS SDD template content 22
4.5 Using the NUTS SDD template 23
4.5.1 Tools and design languages 23
4.5.2 Guidelines for using the template 24
4.6 NUTS SDD proof of concept 24
4.7 NUTS OBC SDD 25
4.8 Discussion and Results 26

5 Software Support Service **29**
5.1 Rationale 29
5.2 Concept 29
5.3 Code documentation 30
5.3.1 Example commands 31
5.3.2 Complete code example 31
5.3.3 Conclusion 35
5.4 Static code analysis 35
5.4.1 Theory 35
5.4.2 Choice of analyser 37
5.4.3 Testing the Clang static analyser 37
5.4.4 Getting CSA to work with NUTS 39
5.4.5 Conclusion 39
5.5 Assembly and deployment 39
5.5.1 First iteration: C daemon 40
5.5.2 Second iteration: C daemon with a Python script 40
5.5.3 Third iteration: Python script with Cron 40
5.5.4 Deployment 41
5.6 Discussion and Results 41

6 Summary and Conclusion **43**
6.1 Suggestions for future work 43
6.2 Conclusion 44

Bibliography **45**

Appendix **47**

List of Tables

List of Figures

2.1 3D model of NUTS courtesy of Christian Nomme 4
2.2 Prototype for one of the NCUBE satellites 5
2.3 Latest prototype for the NUTS chassis 6
2.4 NUTS backplane 8
2.5 NUTS On board computer 9

4.1 The general overview of an SDD - image courtesy of IEEE 18
4.2 The general overview of design elements - image courtesy of IEEE 18
4.3 The different levels of abstraction for SDDs 21

5.1 Part 1 of the output for the C file 32
5.2 Dependency graph for Main.c in the NUTS OBC 33
5.3 Part 2 of the output for the C file 34
5.4 A simple AST 36
5.5 Result of scan-build 38

6.1 Information flow diagram 57
6.2 Flow chart for NUTS RSS listener 58
6.3 UML diagram for NUTS RSS listener 59
6.4 UML class diagram for NUTS RSS parser 60
6.5 UML class diagram for NUTS RSS logger 62

Abbreviations

NAROM: Norwegian Centre for Space-related Education
ANSAT: Norwegian Student Satellite Program
IEEE: The Institute of Electrical and Electronics Engineers
SOLID: Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion
OBC: On-board controller or On-board computer
Development hell: A term used when a project has become ”trapped” and is not moving forward as expected.
Toolchain: A set of tools used to create something. For instance a compiler and a linker.
RSS: Really Simple Syndication
UML: Unified modelling language
FreeRTOS: A real time operating system
EPS: Electrical Power Supply system
ADCS: Attitude and Determination & Control system
GIT: A source code management system
GCC: GNU Compiler Collection
LLVM: Low Level Virtual Machine(former name)
SVN: A source code management system
CSP: Cubesat Space Protocol
SRAM: Static Random access Memory
EPROM: Erasable programmable read only memory

Chapter 1

Introduction

1.1 Software quality

Software quality is a somewhat nebulous term. In fact, if you ask five different software developers you may get just as many different answers. If you ask an application developer she may tell you that quality lies in code that adheres to the SOLID principles [1]. If you ask someone working on weather simulations, quality may be linked to how effective the algorithms are. And an embedded developer will perhaps tell you that energy efficiency is an integral part of software quality.
I will not attempt to define software quality here, in a perfect world we want all of the above (We will not always get there, but software developers should always strive towards it). What we however can discuss is what sort of qualities are most important to us, a list of priorities if you will. In the case of NUTS we know the target platform is a satellite. A satellite inhabits a naturally hazardous environment where radiation can wreak havoc on the system, the satellite also has a finite power budget and patching software can be very hard. We can therefore make the argument that it is more important for the software to be reliable and not crash than it adhering to specific design principles or being clean(though arguments can be made that these are closely linked).

1.2 Thesis aims

This thesis endeavours to find tools (in this case tools can mean actual tools or simply methods) to make it easier to achieve higher quality software, specifically for the NUTS project. It is not a complete guide or roadmap to managing a software project, but rather a select few tools the NUTS project group may find useful to incorporate into the software lifecycle. I have chosen to look at two such tools:

- **Software Design Descriptions:** Software Design Descriptions/Documents (SDD for short) for NUTS are derived from the IEEE standard for SDDs [2]. I worked on

adjusting them for the NUTS project group to make them accessible, unambiguous and fit at any level of abstraction.

- **Software development support service:** I worked on creating a small, autonomous service that could work in the background to support such tasks as generate documentation from code and static code analysis.

My primary concern when working on these tools was making sure that they would fit the NUTS project group (more on this later).

1.3 Previous work

There has been no work done on this particular subject in the NUTS project. This meant that I had to go outside the project to find all my resources. It also means that there could be considerations I have missed that would have been more apparent if there had been previous work done on this subject.

1.4 Thesis outline

Chapter 2 introduces the NUTS project and software development. I will briefly discuss the satellite and its major components. I will also discuss software development and software development specifically for the NUTS project.

In Chapter 3 I will first discuss Software development and quality assurance in general and how it relates to my thesis. Then introduce the software and development from a NUTS perspective.

In Chapter 4 I will introduce Software Design Descriptions. I'll show how the IEEE SDD standard works. I will also discuss the steps I have taken to make the SDD standard a better fit for an organization like the NUTS project groups.

Chapter 5 will introduce the Software support service. I will discuss the various applications this service can have like creating documentation and static code checking.

Chapter 6 I will discuss the results I found and make some conclusions. I will also discuss the road ahead and make suggestions for further work in this field.

Chapter 2

Background

In this chapter I will give an introduction to the NTNU Test Satellite. I will give the reader some background for the motivation of the project and the how it will be realized. This chapter will be about the NUTS mission and the hardware. In the next chapter I will talk about software development in general and software development for NUTS.

2.1 The NTNU Test Satellite

The NUTS project is a multidisciplinary project spanning across several departments at NTNU. It aims to educate students in the creation of satellites from their respective discipline. Further, the mission is to create a satellite based on the CubeSat standard. The CubeSat standard was developed by California Polytechnic State University and aimed to make it easier for universities in particular to create and launch satellites. A CubeSat is defined as a 10 cm cube(often referred to as a 1U), though it can be expanded in depth to three times this. The NTNU Test Satellite is a *double* Cubesat or a 2U to accommodate for its payload.

The satellite's main payload is an IR camera to observe *atmospheric gravity waves* from orbit. These gravity waves are created by air blowing over the surface of the earth, particularly mountains. The satellite will also feature a wireless communication bus to test its viability on satellites.

The goal is to launch the satellite sometime in 2014. As of today there are still some uncertainties tied to the project. Some of the components, like the IR camera has not been properly defined. As such, the reader should keep in mind that some of the information presented here is subject to change.

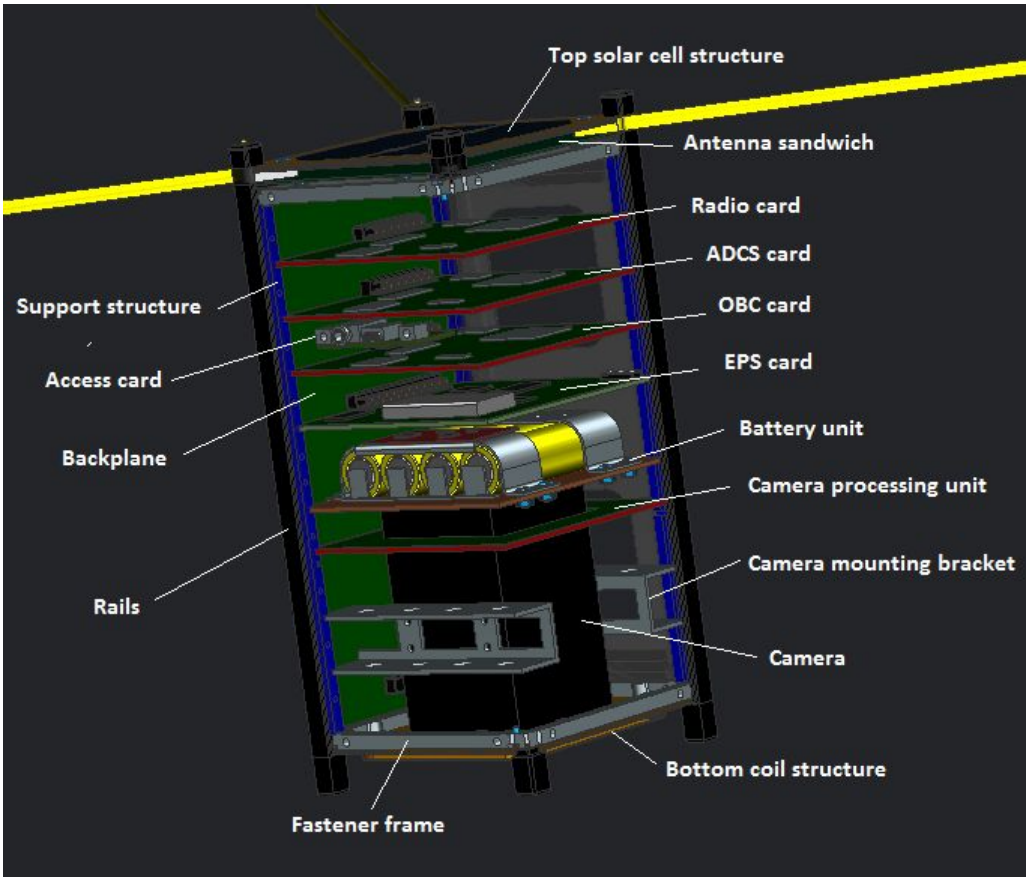

Figure 2.1: 3D model of NUTS courtesy of Christian Nomme

2.1.1 History

NTNU has been involved in two Cubesat projects prior to NUTS. NCUBE1 and NCUBE2, these were both single cubesats. NCUBE2 was launched in 2005 from Plesetsk in Russia, however no one has been able to make contact with it. NCUBE1 was launched in 2006, but something went wrong during the second stage of launch and the satellite was lost.

Figure 2.2: Prototype for one of the NCUBE satellites

In 2006 a small group of students started working on a specification for a new satellite. And based on this work began on the NUTS project in 2010. NUTS along with HINCube(Narvik University College) and CubeStar(University of Oslo) are a part of ANSAT, a student satellite program initiated by NAROM. ANSAT aims to further cooperation between educational institutions and Norwegian industry. So far 24 students have worked on a master thesis related to the NUTS project in previous years. And this semester 10 students are working on their thesis in relation with the NUTS project along with several volunteer students.

2.1.2 The NUTS Mission

NUTS will be launched into low earth orbit(LEO) at approximately 600 km above earth's surface. Low earth orbit is generally defined as being between 2000 km and 160 km

above the earth's surface. This is believed to be the optimal altitude for the mission. Also, objects in LEO will at some point fall to the ground and thus not become "space junk". The satellite will be placed in a sun-synchronous polar orbit. This means it will pass over or almost over both poles and it will pass over at the same time day after day. The satellite will make about 15 passes per day.

The satellite will be launched into orbit by "piggybacking" on the launch of a larger satellite. This is common for cubesat project as it cuts the cost of launch considerably.

2.1.3 Mechanical structure

The NUTS 2U cubesat satellite measures 10x10x20 cm^3 and has a maximum weight of 2.66 kg. Unlike most cubesat projects the NUTS satellite incorporates composite materials to a much greater degree. Carbon fiber has some very useful properties for satellites, firstly it weighs less than metals and it is also quite stiff. The lower weight means that more of the satellite's weight budget can be used on other parts of the satellite. Since this has not been done before, a lot of research has gone into testing the properties of composite materials and how well they apply to satellites.

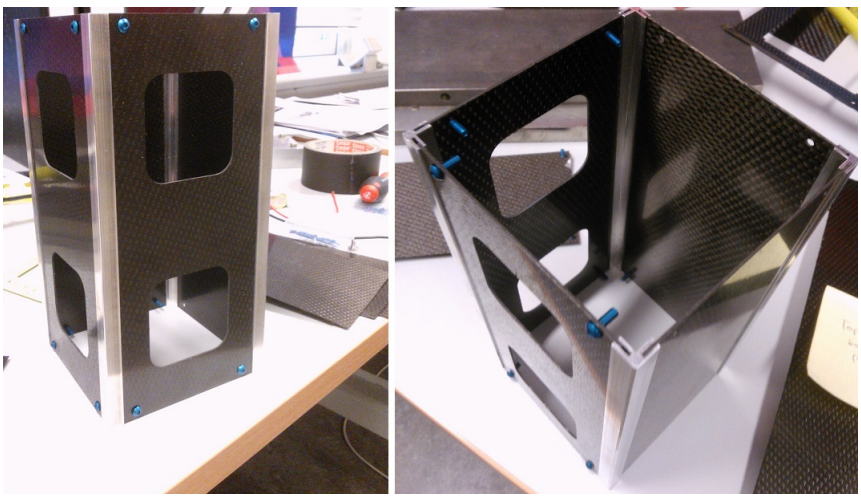

Figure 2.3: Latest prototype for the NUTS chassis

2.2 NUTS Hardware

The hardware for NUTS is a product of several different students working on the project over the years. While there exists CubeSat "kits" commercially available on the market today, one of the goals of the NUTS project is to make the satellite from scratch. This allows the project to have students from a wide range of disciplines involved. One of the design goals has been to use commercially available hardware as opposed to space grade

hardware. This will only be a cursory introduction, as the hardware was not a primary concern for me during this thesis.

2.2.1 Backplane

Unlike many other cubesats, NUTS is designed with a modular backplane. This backplane was was developed by Dewald De Bruyn [3] and Marius Volstad [4]. Its main responsibility is the distribution of power and data between the modules. It is built completely with hardware, not relying on software to control it. The feature set was originally proposed as the following:

- 8 module connectors: two for backplane master modules, one for EPS and 5 general purpose slave module connectors.
- Short-circuit protected 3:3 V and 5 V supplies for each module.
- Power supply protection should include a latch-up recovery mechanism.
- Dual redundant power supply buses for both 3:3 V and 5 V supplies.
- Voltage and current monitoring for both supply voltages for each module.
- Power switches to disable power to individual modules.
- Multi-master capable communications bus with bus isolation for individual modules.
- Debug lines to allow master modules to update program memory of other modules (JTAG/SWD).
- Backplane power and bus isolation switches must be controllable from two master modules.
- Default reset-state of backplane must be to enable all modules.
- Integrated watchdog circuit to revert to reset-state.
- The backplane electronics must be protected from short-circuits by the same mechanisms as system modules.

The communication bus is an I^2C(Inter integrated Circuit) communication bus. The I^2C standard is a widely adopted two-wire serial bus scheme in embedded systems. It relies on a master-slave system, where one master controls all the dataflow. Over the I^2C bus, the satellite will use the CubeSat Space Protocol(CSP) communication protocol for ease of use.

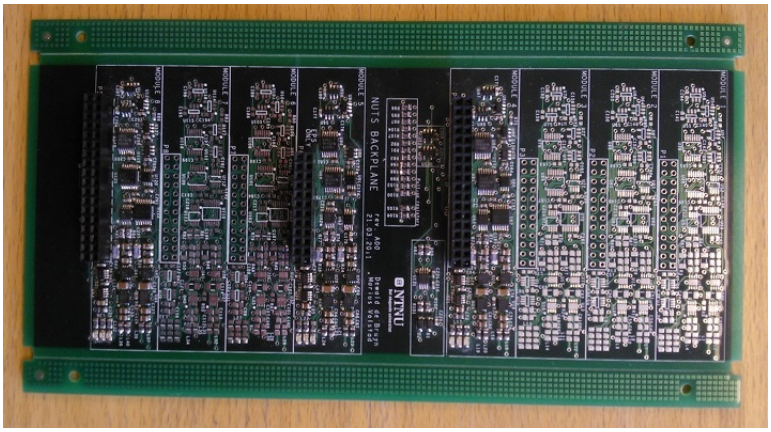

Figure 2.4: NUTS backplane

2.2.2 On board computer/controller (OBC)

The OBC, colloquially known as the ”brain of the satellite” is the primary controlling unit for the satellite. It is responsible for supervising the satellite’s health and issue commands to the other modules. Since it is one of the master modules it is also granted power to shut down or reset other modules. Its primary hardware components:

- **Atmel AVR32UC3A3256:** A 32bit microcontroller designed for low power consumption.
- **16Mb SRAM:** This memory is there to compliment the storage of the microcontroller.
- **16Gb NAND flash:** This memory serves as the primary storage for the satellite. Payload and housekeeping data will be stored here.
- **4Mb EPROM:** This is a One-time programmable EPROM. Since satellites are subjected to radiation there is a chance of data corruption. While EPROM is not radiation hardened, it does provide a more resilient storage.

2.2.3 Radio

The Radio is the second master module in the satellite. Because of that, its hardware is in many ways similar to that of the OBC. If the OBC fails, the radio will attempt to restart the OBC, failing that the radio will take over for OBC. The Radio does however not have a EPROM unit or any external persistent storage. Because of that, if the Radio becomes the permanent master of the satellite it will run at a diminished capacity. As it is a radio it also features two radio transceivers, one 145 MHz transceiver for downlink and one 437 MHz transceiver for uplink from analog devices. The downlink rate is 9600 bps.

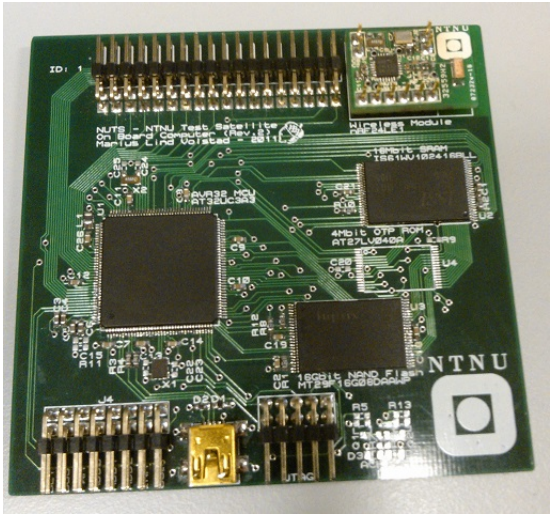

Figure 2.5: NUTS On board computer

2.2.4 Attitude Determination & Control System (ADCS)

The ADCS is tasked with de-tumbling the satellite after launch and controlling the satellite's attitude in relation to Earth. It uses magnetic torques inside the satellite to achieve this. To get a sense of bearing it uses different sensors to such as a gyro, magnetometer and solar panels will act as a sun sensor.

2.2.5 Power distribution and EPS

Power is distributed through the backplane using dual 3:3V and 5V busses working in active redundancy. The electric power system(EPS) module is tasked with charging the batteries and distributing power to the backplane and into the modules. Power is recharged using solar panels around the satellite.

2.2.6 Payload

The primary payload of the NUTS satellite is an infrared camera used to take images of gravity waves in the OH airglow layer(roughly between the Meosphere at the Thermospere in the athmosphere). At the time of writing it has not been decided which camera the satellite will use. The camera has to have characteristics which are not common for this type of camera. It has to be small enough to fit in the satellite, but also be robust enough to handle the strain of space.

The secondary payload on NUTS is the inclusion of internal wireless transceivers. This will be a secondary communication bus for the satellite. The goal with this payload is to research the possibility of using wireless communication inside satellites. The obvious advantage of this is that there is no need for extra wiring, which releases real estate for

other hardware. The wireless transceiver chosen is the nRF24LE1 from Nordic Semiconductors which has a transmission rate of 2Mbps. This also shows the the other advantage of a wireless bus: since the I^2C bus on the backplane will have a limit of 400Kbps, the wireless bus can offload some of the heavy lifting from the I^2C bus. Which is especially useful with the infrared camera in mind.

2.2.7 Ground station and communication

The ground station is based on common ham radio components. It uses a IC9000 radio, Yaesu 5500 rotor and crossed Yagi antennas. Using ham radio components allows utilization of more ground stations. This is important since the satellite can only communicate with a ground station when it is in line of sight.

Chapter 3

Software development

In this chapter I will first discuss software development on a general basis, then I will present software development from the NUTS perspective.

3.1 Software development and quality assurance

Many books can be written about software development(and many have!). So instead of going broadly into every aspect of software development I will focus on the parts of software development that directly touch on the subject I have chosen to be the prime focus for my thesis, quality assurance. Furthermore I will explain how it relates to the subjects I briefly introduced in the first chapter: Design documents and the Software Support Service.

3.1.1 Software quality

In Chapter 1 I mentioned the notion of software quality and how it can be looked at as a bit ambiguous. However moving forward I will focus on the type of quality that matters the most for the NUTS project, namely reliability. It may be hard to pick one favourite virtue of software quality, as I noted in the introduction a software developer will often strive for software quality across the board. However when we consider the potential result of an error that is not handled properly, on a satellite at 600 kilometres above the Earth's surface the choice becomes a lot easier. So even though my choices for research areas are coloured by the needs of the NUTS project, they should be applicable on any project where reliability is very important.

Documentation is a common component to most software projects. In most industry projects it is even a requirement to follow some standard of documentation. The benefits of having a well documented code base is that knowledge can be passed from one developer to the next a lot easier than without documentation. This is where the Software Design Document really shines as it forces the designer defend the existence of the design subject.

If the designer cannot defend that existence then it may very well mean that subject has no place in the software system.

The Software Support Service I created fills another role in software development. By ”always working” it can offload jobs that normally would fall to the developer to do. Leaving him free to focus on design rather than menial tasks such as running tests or auto-documenting software. Some Source code management systems offer this capability by running unit tests and functional tests every time the users commit code to the repository.

3.2 NUTS Software and development

Software development for the NUTS project is still very much in the early stages. In fact things have not changed a lot since I started working on this thesis and now near the end of my thesis. The work that has been done and the work currently under way has been on very specific areas. No one has yet made any definitive plans to approach the satellite software as one entity. As such there is a lot of knowledge about specific areas of the satellite software, but almost nothing on how it all fits together.

Beyond firmware, the first software development for the satellite was done in 2011 by Dan Erik Holmstrøm [8]. He worked on a bus protocol for the I^2C bus. The goal was to create a fair arbitration system for the bus. Unlike many other systems that use I^2C, the NUTS I^2C bus has potentially two masters.

The NUTS software that does exist is primarily a version of FreeRTOS that works on the OBC and a shell service that makes it possible to communicate with the OBC from a computer. NUTS also uses Atmel Software Framework for micro controller specific code.

3.2.1 Challenges

As I have already eluded to, there has been very little work done on the NUTS software from a project management point of view. There is no well defined plan for what should be worked on or any state or goal the software should reach. The reason for this as I see it is that since most of the work done on the project is in essence a byproduct of students working on their thesis, little work is done on moving the software development as a whole forward. Students work on their own separate problems, but no one is looking at the bigger issues. When we combine this with the fact that *most* of the student work has been on hardware issues the end result is what we have right now. And this will continue to be the case until someone makes a concious effort to work with software on a management level. The project leader Roger Birkeland has made an effort to put software on the agenda, however he also has to oversee every other aspect of the NUTS project. I believe that only a group effort will push the software side of the NUTS project out of ”development hell” as it is called. My thesis does not seek to solve this issue on itself, what I have aimed to do with my thesis is make this push a little easier by providing tools that can help to get software development to a state where meaningful progress can be made.

3.2.2 Source code management

Until recently the source code was stored on a SVN repository hosted on NTNU. However it has later been moved to the Bitbucket service(www.bitbucket.org). This service offered free hosting for academic projects and utilizes GIT for repositories which is a source code management systems a lot of students are familiar with.

3.2.3 The C language

The C language is used almost exclusively in the NUTS project. It is an immensely popular language in embedded environments since it offers the user a lot of flexibility. It looks and behaves like a high level programming language, but offers the freedom of an assembly level language. Furthermore the language is quite compact and tools for compiling and working with C have proven reliable over many years. This has made C the language of choice for many working on system programming(low level programming, for instance the Linux operating system is largely written in C and assembly).

However because of these things there are also a lot of pitfalls when it comes to C in terms of reliability. C does not force the user to implement any steps to ensure that the program does not do anything that might crash or destabilize the system. I will address some of the most pertinent problems with the language here to give the reader a little insight into the inherent issues that comes with using a C in a safety-critical system.

- C allows the programmer to point to any address in the memory using pointers. This offers a great deal of power for the user. However since pointers can be modified with arithmetic operations there is also the possibility for pointers to address the wrong memory.
- C is not a strongly typed language, or rather the inclusion of pointers make it possible to circumvent type checking done at compile-time.
- C requires the user to manage memory themselves. Unlike languages like Java or C# where the garbage collector makes certain that allocated memory is freed up once it is no longer used. In the C language the programmer herself must free up any memory that is not intended to be used any more.
- C does not check bounds on any set of allocated memory(strings or arrays). This means the user must be sure to always let the program know just how big an array or string really is.

These are just some of the more prevalent issues with C. For a more in-depth view of the issues of C the reader should consult chapter 11 of the NASA Software Safety Guidebook [5] which gives a good introduction to the issues connected to programming in C.

Chapter 4

Software Design Descriptions

Software Design Descriptions or *Documents* (I will use these two interchangeably) are documents used to describe software or systems. In this case I will focus on the IEEE standard 1016-2009 [2] as a base for developing an SDD template tailored for NUTS. Firstly I will go into detail about what an SDD really is and how it fits into software development. Further I will present the SDD version which I have proposed for the NUTS project and then give an example of how it has been used in the NUTS project. Lastly I will discuss the results of using the SDD.

I'd like to preface this chapter with the following: The reader will soon learn that the SDD standard is quite abstract, and may even seem convoluted. This is by design, the standard seeks to be as open-ended as possible. What I have done is to narrow down the standard into something that fits for NUTS and is more tangible for someone who has no experience with SDDs.

4.1 Rationale

Before I go into detail on software design descriptions I would like to discuss why I chose to focus on SDDs. In a project where reliability is such an important part of the software the developers need proper documentation. There needs to be a rationale behind every decision made. There should be a clear and concise answer to why every line of code exists, since every line of code could potentially introduce a devastating bug. To be able to accommodate this need there needs to be a system of documentation. When I first started to investigate documentation as possible field of research I already had this in my mind. And the only question I had was how to make it easy to use. While anecdotes make for poor proof; I have yet to meet a developer that enjoys documenting software and as such a large part of my work on SDDs was to make them as easy and painless to use as possible.

4.2 An introduction to SDDs

IEEE describes SDD as a representation or model of a software system. It should contain concise information on how the system works to help in planning, analysis and implementation. SDDs should be used as a tool to partition the system into manageable pieces which can be worked on individually without having to take into account the whole software stack.

An SDD can be seen as a collection of *design views* whom each in turn is governed by one *design viewpoint*. Design viewpoints are based on different *design concerns* which can be addressed using various tools. I will explain these terms and how they fit together.

4.2.1 Terminology

From the IEEE Std 1016-2009 [2] I have chosen the most pertinent ideas which make up an SDD.

Design element

Any item occurring in a design view. They can be classified as one of the following types: entity, relationship, constraint or attribute.

Design entity

A design entity is a component, module, process or similar that is distinct from other such elements. They are usually the result of a one or more of the entire software system's requirements.

Design attribute

A design attribute is a specific characteristic of a design entity, relationship or constraint. One example would be a explanation of a entity's function in the software system or for instance other entities this entity relies upon to function.

Design relationship

Describing connections between different design entities. For example the correspondence between a sensor module and the housekeeping process.

Design constraint

A rule imposed on the design of the software. For instance it has to comply to a certain standard.

Design concern

A specific area of interest with respect to the software design.

Design rationale

The reasoning behind the creation of the software and the design choices made.

Design Concern

A specific area of interest with respect to the software.

Design view

A collection of one or more design elements to address a set of design concerns from a specific design viewpoint.

Design viewpoint

The specification of elements and conventions available for constructing and using a design view. *Table 1 – Summary of design viewpoints* in IEEE Std 1016-2009 [2] should give a good idea of what viewpoints an IEEE compliant SDD should contain when applicable.

4.2.2 How SDDs work

As mentioned SDDs are used to describe or conceptualize a subject. What this subject is, the standard does not really concern itself with. The subject may be almost any form of software, however it is generally software of such a size or complexity that it merits an in-depth analysis. The standard does not really put any limits on when someone should use SDDs. But it is often thought that they should be used in the planning and early development phase, they may also be used as a tool to properly document a system that is already in a production state. Whatever the reason, this is achieved by addressing design concerns. Design concerns are areas of interest in the design of the software and may include for instance performance or functionality (in the case of NUTS and this thesis perhaps most of all reliability). To address these concerns we use design viewpoints which governs design views. These viewpoints introduces design elements which come in four different variants: design entities, relationships, attributes and constraints. These are used to build a design view.

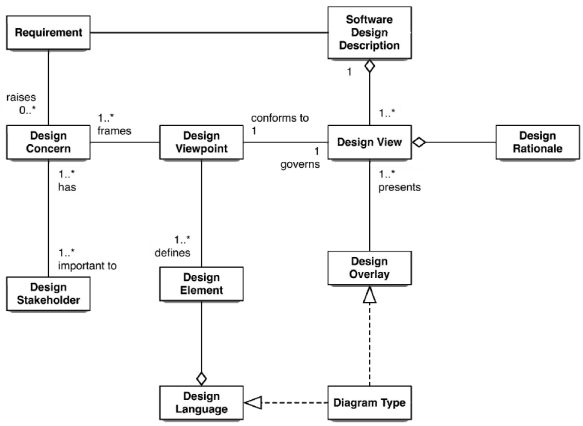

Figure 4.1: The general overview of an SDD - image courtesy of IEEE

Figure 4.1 shows the relationship between the different building blocks of an SDD. We see how a viewpoint defines a set of design elements. These are all presented by a design view.

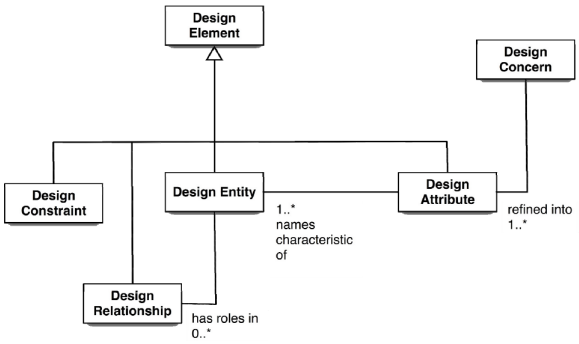

Figure 4.2: The general overview of design elements - image courtesy of IEEE

Figure 4.2 shows the anatomy of design elements. Design attributes here are specific characteristics of any design entity which in turn can be components or functions or similar elements. The design entities can be any type of module or component in a software system. One could perhaps say that these entities are the most tangible pieces in an SDD because they can be directly mapped to actual methods, modules or components in the software.

As already mentioned a SDD is a collection of design views governed by different design

concerns. The point of these views is to separate the concerns in such a manner that a team can work using a top-down approach. Starting with simple black box designs and going down to specific implementations.

Such an SDD would look something like this(roughly as seen in the IEEE Std 1016-2009[2] Annex C):

Frontpiece

- *Date of issue and status*
- *Issuing organization*
- *Authorship*
- *Change history*

Introduction

- *Purpose*
- *Scope*
- *Context*
- *Summary*

Body

- *Identified stakeholders and design concerns*
- *Design viewpoint 1*
- *Design view 1*
- ...
- *Design viewpoint n*
- *Design view n*
- *Design rationale*

4.3 SDD in the software life cycle

SDDs can be used in various stages of the software life cycle. It can be introduced as a tool to better understand an existing software system or it can be used to plan and implement a software system. It is often used in conjunction with a *software requirements specification*. It also serves as a tool for testing software to see that the implementation does what is was specified to do.

4.4 The NUTS SDD template proposal

When I first started working on a design document template for the NUTS project I had certain goals which I wanted to reach. Chief among them was that one template should work on all levels of abstraction. I wanted a familiar document form that could be used for giving a complete system overview as well as describing a single subsystem. I created a list of goals I hoped to achieve when designing the template.

NUTS SDD template design goals

- The template should be a *one size fits all* type document.
- The template content should be instantly recognizable by the user.
- The template should be easy to use and not require the software designer to read up on any software design methodology beyond an introduction to the NUTS SDD template.
- The SDD template should be lightweight enough that anyone working on the software is not bogged down by formalities or bureaucracy.
- Introduce a language that is unambiguous.

Like previously mentioned the IEEE standard is clearly designed to be very open on how to use it. I wanted to narrow it down to a more precise template, and what allowed me to do that was that I had information about the project. While the IEEE standard allows for use on any project, I could do away with a lot of its content since the NUTS project software is very specific. As the reader will see the original IEEE standard and the template I have created are very different, but I think it is important to see the source of inspiration that lead me to creating this SDD template.

The first thing I did was to merge together *views*, *viewpoints* and *concerns*, and calling them viewpoints. I wanted the software designer to look at the template as if he or she was answering logical questions to the existence of this particular software system. I concluded that there where four very important questions(or viewpoints) you could ask about a software system: Firstly, what is this and why does it exist? I think this probably the most important question to ask when creating software. Especially when keeping in mind that complexity in software is often a source of bugs. So the first question a software designer should ask himself is, can I justify the existence of this software system? Also,

for someone who is looking in from the outside, they need to understand why this software is needed. So the first question the designer has to answer is ***Context***.

The second question is what are the pieces that make up the software system. Does it have subsystems or not? To make this template work on all levels of abstraction, we have to answer this question. While a localized subsystem may not have many components, a document describing the entire NUTS software ecosystem may have dozens of subsystems that come together to make the entire system work. So this really is a question about what are the key components to make this particular software system work. This is a question I refer to as ***Composition***.

The third question I wished the designer to answer is how the system implemented? On a microlevel this means answering for instance what algorithms or methods are in use in the system. On a macrolevel the question can become a question of what language is used or what important design decisions resonate through the entire software stack. This is the question of ***Implementation***.

The last question to answer is what does this software system depend on? On almost any level of abstraction software depends on other software. This may be third party software, or other software developed for the system (preferably with a reference to the SDD of that other software). This question should also answer what services that this software system offers. Therefore I call this question ***Dependencies and interfaces***.

These four questions is the core of my SDD template. These should be applicable on any level of abstraction. To expand on this particular notion, when I say this I'm thinking of how the template can be used to describe the system from a bird's view for those who want to gain a general understanding of the system as well as those who want to look at very specific implementations within the NUTS software stack.

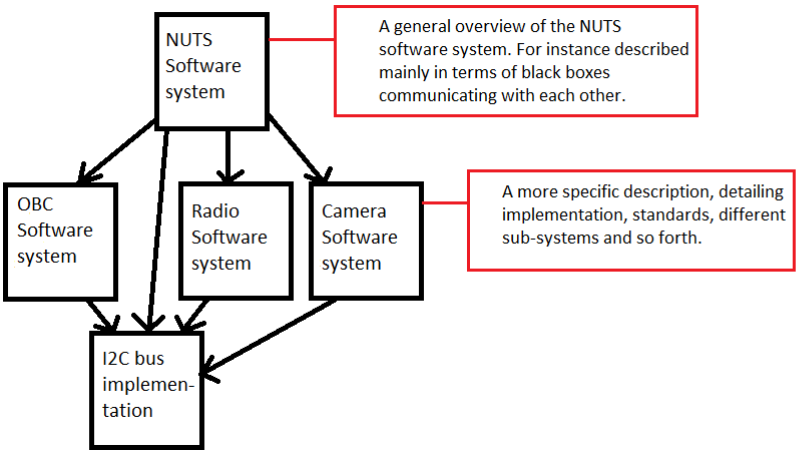

Figure 4.3: The different levels of abstraction for SDDs

Figure 4.3 shows what exactly I was aiming for in that regard. These blocks all represent SDDs, but at different levels. So that for someone who just wants to know what the different building blocks of the NUTS software stack is they can review the top document.

For someone who wants to understand the I^2C implementation they will look at the bottom level SDD. There is no hard limit as to how specific or how high level these SDDs can be created as, which was one of the characteristics from the IEEE standard I wanted to keep.

4.4.1 NUTS SDD template content

Project Name/ID

A unique name and/or ID to set it apart from other SDDs. This ID should be such that there is a identifiable relationship between SDDs where this is applicable. An example form would be "Project name" - "ID number" - "A very short description or name that is instantly recognized".

Creation Date

Date the document was created.

Author(s)

Authors current and previous.

Change history

Changes done to the SDD. A stated change should include a rationale when applicable.

Context viewpoint

This viewpoint should describe the what and why. It should contain a rationale for its existence and what services this software system provides for the overall software stack(when applicable).

Composition viewpoint

Thisv viewpoint should contain a breakdown of all the major components and how they interact with each other internally(when applicable).

Implementation viewpoint

How this system is implemented when applicable.

Dependencies and interfaces viewpoint

This section should contain a list of any systems this system depends on (preferably with a reference to the SDD of that system). If this system provides services which other systems wish to use then it should describe its interface and how to use it.

Glossary

A quick explanation of terms which are not readily apparent.

References

Other SDDs, links to code documentation, or other relevant information pertaining to this software system.

Appendix

How-to guides, examples and so forth.

4.5 Using the NUTS SDD template

The NUTS SDD, like the IEEE standard is intended to be used anywhere in the software lifecycle. However, it is obvious that the earlier in the lifecycle the SDDs are used the more can be gained from them. Since post launch of the satellite they will more or less serve as a documentation for any one who are building the next generation of satellites. Further more, the SDDs are supposed to be living documents. What that means is that the first draft of the SDD does not have to be the definitive versions. As in everything else, plans change in software development. The idea was that the documents would grow and become a good representation of what the software is supposed to be. The more the document is in line with the developer's vision for a given software module the easier it becomes to compare it to the actual software and make changes when appropriate.

4.5.1 Tools and design languages

Just as the IEEE standard, I did not put a lot of weight on how the software designer chooses to describe their software. While the IEEE standard does mention using UML components to represent the software it does not dictate exactly how to go about this. I wanted to continue that trend as I felt that to specify exactly how to describe the software went against the spirit of the template. Meaning that in terms of my design goals, the template should be very easy to use. So to avoid having the designers conform to arbitrary standards that they would have to invest a lot of time into learning I wanted to them to use the tools they were familiar with. At the level that the NUTS project is I can expect the software designers to have a certain level of understanding when it comes to system design. They have a understanding of graphical design languages such as UML and to

impose rules on how to describe every element in the software stack did not make sense.

4.5.2 Guidelines for using the template

The only guideline I offered was that the designer should rely on visual design languages as opposed to written text. To write a textual description of complex software and to ensure that it makes sense I felt would take too much time. As the old adage goes: ”A picture is worth a thousand words”. Secondly I felt that the SDDs would be better if they were short and to the point, that it would be better to use two SDDs smaller SDDs instead of one bigger. The reason I felt this was a good choice was that this keeps with the original idea that should work on any level of abstraction and if the designer wants to really specify a certain aspect of the software he can do that by creating a more precise SDD for that aspect.

4.6 NUTS SDD proof of concept

After I had created my rough draft for a template I started working on a proof on concept SDD(or in this case: three SDDs). To avoid any complications I made a example that is not directly related to NUTS. This way it could serve as an example even if the plans for the NUTS software changed. The original premise was as following:

> *The example will be a simple Windows Service that checks an RSS feed for news about NUTS and stores these news in a file.*

This is a very simple example that in reality probably would not need an SDD at the level of detail I created. However as an example it works. The SDDs for this example can all be found in Appendix D. I will go through them step by step, but a reader should familiarize himself with these three to fully understand what is going on.

As one would expect the SDDs start out with a ProjectName/ID. When looking at the first SDD it says *NUTS RSS listener - 01 - ”Background Service”*, it starts out with the project name, then an ID number (01) and finally a colloquial name for the software module this SDD describes. When looking at the next two SDDs they are formated almost the same way with the exception of one thing. There is an extra ID number so the **Downloader/Parser**(for simplicity I will refer to this one as the Parser from now on) has the ID *01 - 01* and the **Logger** has the ID *01 - 02*. This makes it easy to navigate the SDDs since if you know which SDD is the parent to the SDD you are currently looking at.

The *Creation Date* and *Author(s)* are as one would expect, and if more than one person has created or worked on this SDD then all names should be stated. This is in case there are segments in the SDD that is clear and the original author might need to explain his or her reasoning. *Change history* is as one would expect a complete log on what changes has been made to the SDD with the date of those changes. It should be clear from the change log what changes has to been made and why they were made. In the case of this proof of concept the only change made is to the primary SDD (NUTS RSS listener - 01 - ”Background Service”) and it simply states that a change was made to clarify an issue.

The *Context* viewpoint in all three SDDs give a clear picture of what they represent. The first one is a general introduction to the software and its purpose while the two subordinate SDDs explain their role in the greater scheme of things. On the *Composition* viewpoint the primary SDD identifies the two key components of the service, the Parser and Logger (which are the ones that have their own SDDs). These can be clearly identified from the IEEE standard as *design entities* that have their own distinct features. However these features are not discussed here as this is the primary or ”top level” SDD and for someone reading this SDD it may not be of interest to know the exact specifications of these entities. The Composition viewpoint of the subordinate SDDs however describe with detail what classes are in used and how they interact.

In the *Implementation* viewpoint I rely largely on images images to tell the reader what is going on. As previously mentioned images and UML diagram are far more effective at explaining the relationship between software modules than simply a textual description [9]. As one would perhaps expect by now the primary SDD has a more abstract approach to describing the implementation, such as stating what programming language the service will be developed in, but it also explains the relationship between the Windows service, the Parser and Logger. The two subordinate SDDs go into detail on their respective fields. The last viewpoint, the *Dependencies and interfaces* states what external software they are dependant on to work or if they offer any services to other modules. In this case the Background service is dependant on data from Downloader/Parser and to store the data it needs the Logger. These two respectively state the interfaces they offer in their Dependencies and interfaces viewpoint.

The last point in each SDD is the *References*. Here we can see that the respective SDDs reference either external sources of information or reference other SDDs. So for instance the primary SDD references its two subordinate SDDs and they in turn reference the primary SDD. In this example none of the SDDs needed either glossary or appendix so I had them cut from the SDDs.

4.7 NUTS OBC SDD

Once I felt I had created a viable model for how the SDD template should work and made a proof of concept SDD it was time to try it out on the NUTS project. While I was working on the template the NUTS project had started to employ a wiki system to replace the old system of using Google docs for all their project documentation. A wiki system is very much in line with how I envisioned the SDDs because I emphasised on splitting the documentation up into smaller and more manageable pieces. However to support that a user needs to be able to quickly navigate between these documents. A wiki system works very well for this purpose as it is easy to set up links to different wiki articles. So each SDD would get one article page, with the contents described in my template.

To work out the first SDD for the NUTS project I sat down with others working on software related topics for the NUTS project and discussed the current state of the NUTS software. At the time there was little direct work being done on the NUTS software, most of the work was proof of concept or exploratory work. Also there was still a lot of uncertainty regarding the final version of several components in the satellite. The hardware module we knew the most about was the OBC, so it became the natural center of discus-

sion. During the meeting we created a list of what features the OBC should have and I later processed the list to create the first draft for the OBC SDD. This SDD can be seen in Appendix E.

Just as in my proof of concept SDD the Context viewpoint covers the question of what and why. Giving a clear answer to the purpose of the OBC. The ”meat” of this SDD is in the Composition viewpoint, here I have broken down the various tasks alluded to in the Context viewpoint into distinct components (the *design entities*). However some of these are still somewhat vague at this point in development, for instance the first component *”Housekeeping / logging”* does not specify in any detail what kind of sensors are being used. These are issues which are still unclear at this point.

The Implementation viewpoint is pretty sparse beyond stating that the OBC software will run on FreeRTOS and the rationale behind this decision. Furthermore it states that even details on how FreeRTOS will run is yet to be determined. The last noteworthy point in this SDD is the References section, just as in the proof of concept the components are referenced with their own SDDs. These SDDs are not created yet, but in the wiki system it is very easy to start working on them right away. The last thing I created was a ID-number reference chart(Appendix F) that stipulates how the ID numbers should work within the SDD hierarchy. For instance the ADCS and its subcomponents' SDDs should have the ID number 1.3 and 1.3.* respectively. This way the user can know where a given SDD belongs.

Overall the SDD for the OBC came out more or less as I had expected. Since there is still very little that is certain about the NUTS software stack it became pretty much a cursory glance at the top level of the software stack. However for students who are not familiar with the project, or even what the OBC really is it does serve as a stepping stone for getting into the project.

4.8 Discussion and Results

When I started work on a template for software documentation (at the time I had not discovered the IEEE standard) I was uncertain as of what level of *control* I wanted to assert over the users. One of my first ideas was to create software to support the creation of the documentation. For instance one idea I had to link these documents directly to the code documentation created by my software support service (described in chapter 4). However I came to a few conclusions while doing this work, the first one being that this documentation would only work if the users made the effort to make it work. For that to happen, it has to be easy, especially considering that this project is being worked on by students who are either working on their thesis or volunteers. While a project in the software industry could impose requirements on the people working on it, it is a lot harder to make that happen when the workforce is largely volunteer based. Secondly, since software development for NUTS still is in its infant stage I could not predict needs that may arise later in development. Because of this I had to make the NUTS SDD template easy to use and open ended.

In terms of results it is hard to gauge just how well this template will perform. One of the main issues is that there is very little work being done on the actual satellite software. This meant for me that I had to rely on doing proof of concept work and a preliminary draft for

the OBC. Had there been more work being done on the satellite software I could have had users create SDDs and conduct interviews on how they felt the SDDs worked for them. Furthermore I could not find much research related to this specific topic, so I could not compare it to other work. Aside from a some encouraging comments from my colleagues I feel that the true worth of this endeavour will not reveal it self until the SDD templates are put into use during development of actual satellite software. Non the less I do think that having worked out a way to properly document software for the NUTS project can help the project down the road. It should at the very least make it easier for new students to gain an understanding of what the students before them had planned for the software.

Chapter 5

Software Support Service

The Software Support Service(I will usually refer to it simply as *the service*) was conceptualized as a service that would run in the background to aid the developers. In this chapter I will first go through the concept of the service, then present two of the tools which I wanted to integrate into the service and finally will discuss the actual assembly and use of the service. This service did not turn out as I had hoped as I could only integrate one of the two tools I had originally planned. While I do believe that both tools can be integrated, however due to time constraints I could only get one of the two working as I will explain further in this chapter.

5.1 Rationale

As in Chapter 4 I wish to open up with an explanation as to why I chose this particular topic for my thesis. In this project there are quite a few different individuals involved, there are both volunteer students and students working on their thesis. Now keeping that in mind, my immediate concern was that these people would be using different toolchains and have different levels of experience developing software. So what if I could create a tool which the developers themselves did not have to use, but generated useful information for them? I wanted to explore this premise as it seemed to me like there was an opportunity here for me to lessen the workload for the developers despite what platforms they used.

5.2 Concept

The primary mission of the service was to lessen the workload of the developers. My basic idea was an autonomous service that could do several different tasks. It would produce valuable reports and documentation no matter what tools and methods the individual developers used. Also, it would run on a server and at specified intervals it would perform these tasks. I decided I would focus on two such tasks, namely auto-generating code commentary and static code analysis.

Instead of developing my own software for doing these tasks I decided that my service would rather call upon already developed tools. So my first task was to find tools that could do this job for me. Early on I decided that I would pursue free or open-source software. My primary reasoning was two fold: Reduce the cost of operating the service and be able to adjust the software to suit the needs of service. With that I also elected to develop the service primarily for the Linux operating system.

With these things in mind I created a small list of requirements for the service.

- The Service shall run autonomously without any user interference.
- The Service shall be able to download the latest version of NUTS software from the NUTS repository.
- The Service shall process data from the NUTS repository and upload reports to a specified area atleast once a day.
- The Service infrastructure shall support ”plugging” in new tasks with a minimum of work.
- The Service shall run on a Linux operating system.
- The Service shall employ free or open-source third party software.

The service works in essence as a four step process: Clone the target repository, apply the different tools to the source code, publish the reports on a server accessible by the NUTS project team and finally delete all reports and source code stored locally. Instead of keeping a version of the source code locally and just updating it I elected to always clone, process and delete. The main reason for this is that it is a lot simpler to handle. With a local copy that needs to be updated there is the chance that conflicts in the code will occur and the system will not always be able to auto-resolve these.

5.3 Code documentation

When finding a tool that could help create documentation for the code the choice came down to Doxygen. It was during my investigation of static code checking that I first encountered Doxygen. After reading up on it I found it garnered favourable reviews for its customizability, but what sold me was that the Atmel Software Framework(ASF) used it in their documentation. Since ASF is an integral part of the NUTS software stack it became a natural choice for me. DoxyGen is also open-source, fulfilling one of my requirements for the service.

Doxygen is supports multiple languages, amongst them C. It can also export its reports into a variety of formats. For NUTS it was decided to use html since it can be hosted from one of the project servers and is almost universally accessible. As previously mentioned, the output Doxygen produces can be highly customized. The specific user preferences can be saved into a file for later use, as such I created a *baseline* file for the NUTS project. This file will be used by the service every time it documents the code.

To make use of Doxygen's code documentation capabilities it requires a special set of commands. This command set is quite rich, but for the NUTS project's uses a smaller subset is more than ample. I'll show some example code of how these commands look and explain their effects.

5.3.1 Example commands

The simplest form of comment is (note the double asterisk at the first line), this is called a detailed description. This will be attached to the closest method under the command.

```
/**
*  .. Some text here ..
*/
```

By adding \brief you can describe the method or file with one line:

```
/**
* \brief Describe your method or file in this one line
* ... A more detailed description ...
*/
```

If you want to comment the whole file add the following command:

```
\**
* \file yourfile.c
*  ..different commands..
*/
```

These makes up the basic building blocks for commenting code using Doxygen. However as mentioned Doxygen supports a lot more comment keyword like \todo and \bug which can be helpful during development.

5.3.2 Complete code example

Here is a more extensive example of a C file.

```
/** \file Testfile.c
* \brief This is a brief Description
* \details This is a more detailed description
*/
#include <stdio.h>
#include "someheader.h"
/**
* Detailed description of this fine looking main method
*/
int main()
{
int i;
// Some normal comments
for(i = 0; i < 100; i++)
{
```

```
16      //Loop stuff.
17  }
18
19  return 0;
20  }
21
22  /** \brief A brief description
23  *\todo Not implemented
24  * \bug Might eat your lunch
25  */
26  int bar(int foo)
27  {
28    //Comment
29    return 0;
30  }
```

Doxygen processes code on a folder basis, meaning the user instructs it to search through a folder and subfolders(the user can specify this themselves). The output for this C file is becomes the following.

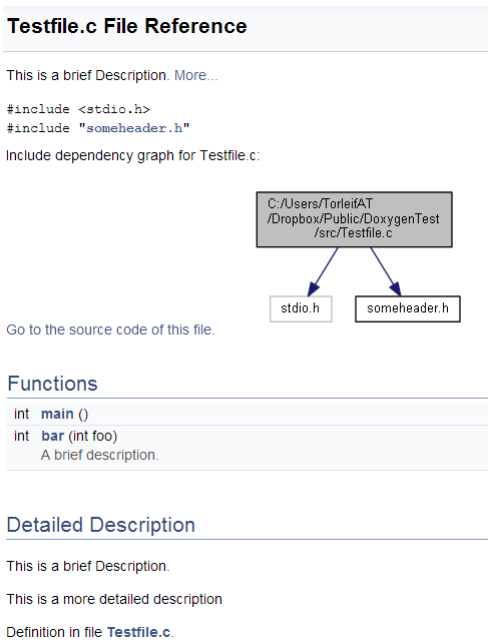

Figure 5.1: Part 1 of the output for the C file

In figure 5.1 we see Doxygen has done several things for us. Firstly it lists up which included header files this file relies on. Secondly we that Doxygen has listed the functions in the file. The second function has a \brief description at line 22 which coincides with the description in the function listing. We also have a Detailed Description area which coincides with line 1 to 4. Doxygen also shows a dependency graph, which can be very

useful during development and debugging.

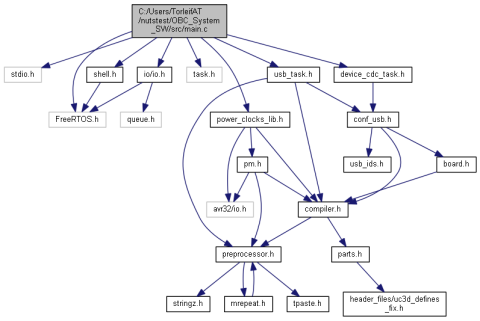

Figure 5.2: Dependency graph for Main.c in the NUTS OBC

Figure 5.2 shows is the dependency graph for the main C file for the NUTS OBC project. It shows all the header files it depends on as well as what these header files depend on.

Function Documentation

int bar (int foo)

A brief description.

Todo:
Not implemented

Bug:
Might eat your lunch

Definition at line **28** of file **Testfile.c**.

```
{
    //Comment
    return 0;

}
```

int main ( )

Detailed description of this fine looking main method

Definition at line **12** of file **Testfile.c**.

```
{
    int i;
    // Some normal comments
    for(i = 0; i < 100; i++)
    {
        //Loop stuff.
    }

    return 0;
}
```

Figure 5.3: Part 2 of the output for the C file

In figure 5.3 we see the documentation for the two functions in the C file. Note that the *Definition at line 12/28 of file Testfile.c* is slightly off from the C file above, this is because the original file had some empty lines. We see the use of both the \todo and \bug in the *bar* function. These are also presented in their own lists for easy access.

5.3.3 Conclusion

As a tool for documentation Doxygen covers the needs of the NUTS project quite well. It works on multiple operating systems, such as Linux and can be easily invoked from command line. This last point make it very useful since my service relies on invoking tools from the *command processor* as I will explain later. It also worked very well on the NUTS software repository. Overall my conclusion is that Doxygen is a good fit for both the NUTS project and the Software support service.

5.4 Static code analysis

Static code analysis is the process of analysing code without fully compiling(I say *without fully compiling* because there are certain steps which this analysis share with a compiler) or running the code. Static analysis have a number of uses such as detecting anomalies in the code or enforce a certain programming standard. The primary different between static and dynamic analysis is that dynamic analysis is performed during runtime. As opposed to static analysis which is performed without running the code. Dynamic analysis is performed primarily to find and debug errors. Since it is done during runtime it will only look at one execution path. The static analysis covers every possible execution path based on the Abstract syntax tree (AST).

5.4.1 Theory

Like mentioned, static code analysis is related to compiling. In fact, most compilers do employ static analysis to a degree. When a compiler processes code it is done through several stages. First a *lexer* splits the code into symbols it that can be recognized. For instance a line like:

```
int foo = bar + 4;
```

The lexer will take this stream of signs and split them into individual tokens. Once this is done, the lexer will identify what *kind* of tokens they are. In this case there is a reserved word *int*, two identifiers *foo* and *bar*, a plus sign and a constant. At this point the *parser* takes over. The parser knows the *grammar* of the target language. The difference between the two is clear when the considering the following line of code:

```
int foo bar + 4;
```

This will not raise any problems with the lexer, however the parser will identify this as an error. Simply put the lexer *recognizes* words while the parser *understands* their context. It will identify errors in the code and make sure the code is grammatically sound.

What happens during the parser's process is that it creates an *abstract syntax tree*(AST for short). ASTs can be regarded as the purest form of the source code(at least from a human perspective), because only the pertinent information has been stored. The information is stored in a tree structure where each node corresponds to a token. How these are created do vary some from compiler to compiler, but as a general rule this is how they look. For instance the example code above could have an AST like the following:

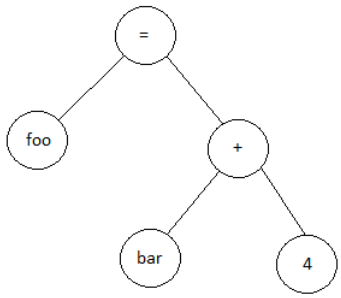

Figure 5.4: A simple AST

Up to this point there is no difference between static analysis and simply compiling the source code. What static analysis does however, is taking things one step further. A normal compiler will create an executable so long as the source code is grammatically correct. What static analysis does is look at what *might* go wrong and what can be considered "dead code". Consider the following piece of code:

```
int foo(int i) {
  int * bar = malloc(i * sizeof(int));
  return = bar[0];
}
```

Without looking at the larger picture, what could go wrong with this piece of code? There is no verification what *i* is, it could for all we know be a negative number. Since we don't know anything about the rest of the code, this example is obviously contrived. However errors even at this level of banality do occur. As shown in this article about PVS-Studio, a static analyser for Visual Studio [6](Beware: The article itself very biased towards the product, but the errors shown are very real).

Static analysis employs a wide variety of techniques to generate their report. And the effectiveness of these tools vary greatly. I will not go into detail about it because it requires a lot of theory that is outside the scope of this thesis. For a closer look at static analysis the reader should consider reading this paper by Anders Møller and Michael I. Schwartzbach on static analysis [7].

False positives

False positives are as one would expect the analyser flagging a segment of code as a potential bug when in reality the bug does not exist. This usually happens if the analyser takes a

path that it cannot actually take or it is not taking into account other circumstances. These happen every now and then and it will be up to developers to identify them.

5.4.2 Choice of analyser

When it came to choosing an analyser I had to take into that I wanted a tool that the members of the NUTS software team themselves didn't have to use. A lot of the tools available rely on the end user installing it and using it in their own tool chain. I also needed a tool that was easily invoked with the command processor. The choice fell on Clang static analyser(CSA for short), this analyser is a part of the LLVM compiler package In fact Clang itself is a compiler, and the static analyser is built on top of it. I chose this primarily for two reasons, it is open source and LLVM/Clang is an ongoing project and therefore kept up to date. This analyser works by injecting itself into the compilation process and uses the data created by the compiler to perform the analysis. The biggest hurdle with choosing CSA was that I had to compile the entire LLVM project, a task which can take hours.

5.4.3 Testing the Clang static analyser

I decided to test the CSA on an old assignment from one of my previous courses. This program uses *pthreads* to calculate the integral

$$\int_{-1}^{1} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\, dx$$

using Simpson's method. The program works by invoking it in the command line and the user can specify the number of threads they want to use to integrate over the area. If the user does not specify a number the program will default to 2 threads. The program works using two loops, one loop divides the area to integrate over the specified number of threads and the second loop joins the result of each thread into a final result. I chose this program because it is fairly small and contained thus making it easier to explain what the CSA found. The whole of the file can be found in appendix A. Invoking CSA can be done in different ways depending on what you want to analyse. The general format for invoking CSA is:

```
scan-build [scan-build options] <command> [command options]
```

Scan-build is usually used to scan an entire project using the project's makefile as a guideline. However since my program did not have a makefile I instructed scan-build to only scan one specific file.

```
scan-build gcc -c -std=c99 ps5_pthr.c
```

Here I dictate which compiler I will be using (GNU Compiler collection, gcc) as well as which compiler flags I want. The anomaly it found was a uninitialized variable which was used in a function call.

```
33  int main(int argc, char* argv[])
34  {
35          double final_result;
            1 Variable 'final_result' declared without an initial value
36          long thread;
37          if(argc < 2)
            2 Assuming 'argc' is >= 2
            3 Taking false branch
38          {
39                  printf("No specified number of threads, defaulting to 2 threads\n");
40                  thread_count = 2;
41          }
42          else
43                  thread count = strtol(argv[1], NULL, 10);
44
54
55
56          for(thread = 0; thread < thread_count; thread ++)
                    4 Assuming 'thread' is >= 'thread_count'
            5 Loop condition is false. Execution continues on line 67
57          {
64          }
65
66          //Join 'em
67          for(thread =0; thread < thread_count; thread++)
            6 Loop condition is false. Execution continues on line 71
68          {
70          }
71          free(thread_handles);
72
73          printf("the result is %.6f\n",final_result);
            7 Function call argument is an uninitialized value
74          return 0;
75  }
```

Figure 5.5: Result of scan-build

Here we see seven labels injected into the code. The actual error reported is the last label which states: ”Function call argument is an uninitialized value”. At line 73 we see the function call *printf* is using the variable . On each label I have marked small arrows with a red ring. These arrows allow the user to navigate back and forth through the code to find the origin of the reported anomaly. If we go from label 1 we see that at line 35 that *final_ result* is never given an initial value. If we then move onto the next label and assume the program does get a specified input then the variable *thread_ count* (which is a global variable) is assigned the value from *argv[1]* on line 43. This is where the CSA gets the idea that this might be a problem. Because what happens if the input is something unexpected? For instance what happens if the value from *argv[1]* is a negative number? If we continue to label 4 we can see that the variable *thread* is set to 0, and if *thread_ count* is a negative number then the loop will never be entered. Label 5 and 6 merely states this

and we are back at label 7. If events go as CSA here suggests than *final_ result* will indeed be uninitialized at the function call on line 73. Now in this case it does not matter, because the C language will always default a statically stored variable to either zero or null(for pointers). And since this is a number, all that will happen is that the result of the program will be zero.

5.4.4 Getting CSA to work with NUTS

As the CSA was showing promise on paper it turned out that making it work with the existing NUTS software was a lot harder than I had anticipated. Software for the satellite is mostly developed using Atmel Studio 5 or in some cases 6. Atmel Studio is an IDE exclusive for the Windows platform that uses Microsoft's Visual Studio as its base. However Atmel Studio still uses GCC, or at least a ported version of GCC for Windows along with a makefile. This makefile is however built for a Windows environment which that it has to be converted to work in a Linux environment. And this has to be done every time the project is updated. This in itself is not a huge problem, writing a script that makes the makefile work in Linux is achievable, but very time consuming. What proved to be the larger issue was getting CSA to work with the AVR32 GNU toolchain as well as the Atmel Software Framework(ASF) that the NUTS project uses in the Linux environment. I was never able to fully uncover why the scan-build and the toolchain would not work together, but one of the larger issues was getting scan-build to find all the dependencies the satellite software had to the ASF. In the end I abandoned the work in favour of working on the primary service and Doxygen.

5.4.5 Conclusion

As a tool, a static analyser is immensely valuable for a project such as this. Static analysis can uncover corner issues which would normally be very hard to spot, but when the stakes are as high as they are they become a real concern [10]. I did not achieve what I had set out to do, that is to integrate it into the service. It is possible that with more time I would have found a way to successfully run CSA on the NUTS software, but I decided I would rather focus on areas. And I think that for further work into software quality assurance for NUTS that static analysis should be considered.

5.5 Assembly and deployment

The Software support service has gone through several iterations during my work. I will describe the different stages of development here. During the development I went through two different programming languages and two very different approaches to the problem. It is a point of interest that the final iteration of the service is in many ways the simplest as I will explain. For the full source code of the final service script see Appendix B, Appendix C covers the C daemon source code.

5.5.1 First iteration: C daemon

My initial idea for the realization of the service was to create a Unix daemon using the C language. A daemon is just a name for a computer program that runs as a background service. Once deployed it works autonomously with little or no user control. A daemon is usually initiated by an executable that creates a child process, this child process continues to run once the parent process has been terminated. In this case I used the *fork()* function to create a child process.

Once the child process was created I could program it to wait for a specific time of day to execute its tasks. I achieved this by using the *system()* function. This function invokes the *command processor* which is how the operating system invokes commands. For instance the function call *system("Python script.py");* would invoke the python runtime and run the script.py file. This was how I planned on adding new tasks.

At this point I was having stability issues with my service. It either would either fail to create a child process or it would stop working. While these were issues I could eventually iron out I came to another realization: once a task had completed I had to handle the data it produced somehow. This involved manipulating folders and strings of text. The C language is not very well equipped for these sorts of tasks and I found that it would be easier to use a language with better string manipulation support that also supported rapid prototyping. At this point I decided to outsource the service's logic to a python script and keep the code for the daemon as bare bones as possible.

5.5.2 Second iteration: C daemon with a Python script

With a small bare bone Unix daemon written in C I then started working on a Python script that would serve the logic of the service. An added bonus to this solution was that I could replace the script without restarting the daemon. I primarily focused my efforts on implementing the Doxygen software into the service, as the static analysis tool did not work out as I had expected. At the time of writing this, the NUTS project has one repository for its satellite software. However I wished to add support for documenting multiple repositories. To achieve this the Python script reads from a list stored in a text file which specifies the address of each repository as well as which specific folder contains the actual software. The script also read the *baseline* file I created for Doxygen so that the report would always have the same look. Furthermore I created functionality to log every operation done by the service. It now stores all the output from each operation in text files that users can access to try and resolve any issues that might occur.

5.5.3 Third iteration: Python script with Cron

After having developed a working version of the python script that implemented Doxygen I was still having issues with the stability of the C daemon. At this point I became aware of the possibility of using *Cron* to run my script. Cron is a time-based job scheduler(a system daemon in fact) found in most Unix systems, it can execute commands or scripts at specified times. This is done by editing the *crontab* (Cron table) file. It can dictate when to run a script down to the minute. The added bonus was that Cron has been around for a long time and has few stability issues. Since my C daemon at this point only had one

job which was to execute a script at a certain time each day I decided that I might as well outsource this job to Cron. Each line in the crontab corresponds to a scheduled job and has the following form:

```
<minute> <hour> <day> <month> <weekday> <command>
```

My crontab entry for the service was as following:

```
0 6 * * * python <script path>/script.py
```

This translates to the script being run every weekday, for every month and every day at 6 am in the morning(That seemed like the best time to run the script as very few people would be working on the repository at that hour).

5.5.4 Deployment

My service was set up on a virtual machine that ran XFCE version of Linux Mint 13 ”Maya”. For executing the python script I ran Python 2.7.3 and the Doxygen version was 1.7.6.1. The source control system I employed was GIT 1.7.9.5. As GIT was the system used for the NUTS software repository. The choice of operating system was one of convenience as the job the service does is not overly complicated or require any special considerations. During this phase the result from Doxygen was uploaded onto my IDI web area as that was the most convenient way publishing the report as long as I was just testing it.

5.6 Discussion and Results

Overall the results I got from developing this service were not what I had hoped. While Doxygen works as planned, the lack of a static analyser makes the service only partially valuable. What has paid off however was approaching this service with the mindset of using already existing software as much as possible, and rather focus on connecting these together. I believe that future work on a service such as this should focus on working with well proven software and rather work on how to connect multiple tools together and produce reports that are easy for those working on the NUTS software to use.

Chapter 6

Summary and Conclusion

During my thesis I explored various methods of supporting the development of NUTS satellite software. I have presented one methodology for documenting software with the software design description template for NUTS. As well as created a service that would serve as a supporting role in software development.

The NUTS Software design description template is a guideline on how to document software. It is designed to support multiple levels of abstraction and focuses on the core mechanics of the software. So far the only SDD produced using this template has been for the OBC. The reason for this is that so far there has been very little active work on the NUTS software stack beyond exploratory work and proof of concept software. It will be up to the NUTS project group to employ this template as they see fit.

The Software support service I developed during my thesis work has mainly focused on two key areas. The first one being auto documentation of code and the second one being static code analysis. I attempted to integrate tools for both of these methods. The auto documentation tool was successfully integrated in my service and works as intended. The static code analysis bit however does not work but I have done some proof of concept work to display its capabilities and uses in software development. Going forward the NUTS project group can easily integrate the current software support service into their software development as the service requires very little beyond a stable Linux environment and access to the source code repositories hosting the NUTS software.

6.1 Suggestions for future work

I have some recommendations for anyone who wishes to continue research in this vein. There is definitely room for expansion and improvement of the software support service I created. If someone were to get the static code analysis working it would be a tremendous boon to the software development effort being made on the satellite. Further an easy way

of getting access to all the reports generated on the service would be good as of now there is no common interface for the service output. The users have to know where to look.

The NUTS SDD template stands I feel stands well on it self. However work into creating a more holistic project management for the software development could potentially be both interesting from an academic standpoint as well as very beneficial for the NUTS project.

6.2 Conclusion

In conclusion I would say that my thesis did not reach the point I had hoped it would. While I do have some results that are promising and can help the NUTS project I feel that too much time was wasted in researching different methodologies and tools for my software support service that ended up not working as well as I had hoped. It became a problem of cherry picking different topics that may or may not lead to something beneficial.

On a personal note I would not have chosen the same thesis work two times. I felt that I had a hard time creating something tangible that was worth pursuing. Perhaps if I had chosen a more specific topic on the NUTS software stack I would have been more in my element.

Bibliography

[1] Pablo's SOLID Software development http://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf

[2] IEEE Recommended Practice for Software Design Descriptions http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5167253 Dewald De Bruyn og evt Marius Voldstad

[3] Dewald De Bruyn: Power Distribution and Conditioning for a Small Student Satellite Module, 2011 http://daim.idi.ntnu.no/masteroppgaver/005/5933/masteroppgave.pdf

[4] Marius Lind Volstad: Internal Data Bus of a Small Student Satellite, 2011 http://daim.idi.ntnu.no/masteroppgaver/006/6403/masteroppgave.pdf

[5] NASA Software Safety Guidebook http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf

[6] PVS-Studio advertisement - static analysis of C/C++ code http://www.viva64.com/en/a/0077/#ID0ELTNQ

[7] Anders Møller and Michael I. Schwartzbach: Static Program Analysis, 2012 http://cs.au.dk/~mis/static.pdf

[8] Dan Erik Holmstrøm: The Internal Data Bus in a Student Satellite http://daim.idi.ntnu.no/masteroppgaver/007/7456/masteroppgave.pdf

[9] Kušek, Dešić and Gvozdanović: UML Based Object-oriented Development: Experience with Inexperienced Developers, 2001 http://bib.irb.hr/datoteka/85049.055_C28.pdf

[10] Baca, Carlsson and Lundber: Evaluating the Cost Reduction of Static Code Analysis for Software Security, 2008 http://dl.acm.org/citation.cfm?id=1375707&bnc=1

Appendix

Appendix A: Clang Static Analyser Test Code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#define N  25000000

//Some precalculated values
double pi;
double invsqrt2pi;
//Global variables!
int thread_count;
pthread_mutex_t mutex1;


struct arguments
{
  double start;
  double end;
  double *final_res;
  int num_pieces;
};

//Proto-types//
void createArguments(struct arguments *args, double *final_res, double
    start, double end, int pieces);
void* integral(void *arg);
double f(double x);
int divideWork(int thread_count);
double simpson(double x1,double x2,double x3);



int main(int argc, char* argv[])
{
  double final_result;
  long thread;
  if(argc < 2)
  {
    printf("No specified number of threads, defaulting to 2 threads\n");
    thread_count = 2;
  }
  else
```

```
    thread_count = strtol(argv[1], NULL, 10);

  pthread_mutex_init(&mutex1, NULL);
  pthread_t* thread_handles;
  thread_handles = malloc(thread_count*sizeof(thread_handles));

  pi=2*acos(0);
  invsqrt2pi=1/sqrt(2*pi);

  int thread_work_part = divideWork(thread_count);
  double start = -1;
  double sub_parts = (double)2/thread_count; // Total length from -1 to 1 is 2.

  for(thread = 0; thread < thread_count; thread ++)
  {
    double start_pos = (start + (sub_parts*(int)thread));
    struct arguments *arg;
    arg = malloc(sizeof(*arg));
    //Note: We pass along a pointer to the final_result here
    createArguments(arg ,&final_result,  start_pos, start_pos + sub_parts, thread_work_part);
    pthread_create(&thread_handles[thread], NULL, integral, (void*)arg);
  }

  //Join 'em
  for(thread =0; thread < thread_count; thread++)
  {
    pthread_join(thread_handles[thread], NULL);
  }
  free(thread_handles);

  printf("the result is %.6f\n",final_result);
  return 0;
}

// Determines how many of N each thread should do.
int divideWork(int thread_count)
{
  if(N % thread_count != 0)
  {
    printf("Thread count not divisible by number of tasks! aborting\n");
    exit(-1);
  }
  else
    return N / thread_count;
}

//Bakes in the relevant data for each thread.
void createArguments(struct arguments *args, double *final_res, double start, double end, int pieces)
{
  args->start = start;
  args->end = end;
  args->final_res = final_res;
  args->num_pieces = pieces;
}
```

```

double f(double x)
{
  return invsqrt2pi*exp(-0.5*x*x);
}

//return area for the given x-range
double simpson(double x1,double x2,double x3)
{
  return (x2-x1)*(f(x1)+4*f(x2)+f(x3))/3;
}


//calculate the integral using simpson's method.
//start,end: endpoints of interval to integrate over
//num_pieces: number of trapezoids the interval is divided into
void* integral(void *arg)
{
  struct arguments* args = (struct arguments*)arg;
  double start = args -> start;
  double end = args -> end;
  int num_pieces = args -> num_pieces;
  double *final_result = args -> final_res;
  double res = 0.0;
  double h = (end-start)/num_pieces;

  for(int i=0;i<num_pieces;i+=2)
  {
    res += simpson(start+i*h, start+(i+1)*h, start+(i+2)*h);
  }

  pthread_mutex_lock(&mutex1);
  *final_result += res;
  pthread_mutex_unlock(&mutex1);

  return NULL;
}
```

Appendix B: The Software support service source code

```
import os
import subprocess
import datetime

#CONSTANTS
TIMESTAMP = datetime.datetime.utcnow()
NUTFACTORY_PATH = os.path.dirname(os.path.realpath(__file__))+"/"
MAIN_LOGFILENAME = "MAIN_OPERATIONS_LOG_"
DOXYGEN_REPORT_FILENAME = "DOXYGEN_OUTPUT_FOR_"
GIT_REPORT_FILENAME = "GIT_OUTPUT_FOR_"
SCP_REPORT_FILENAME = "SCP_OUTPUT_FOR_"
DOXYGEN_BASELINE_FILE = "NUTS-doxygen-baseline"
GITREPOLIST = "gitlist.txt"
CONFIG_FOLDER = "/config/"
SCP_TARGET = "torleift@login.idi.ntnu.no:./public_html/NUTS/"
#ENUMS
LOG_TYPE_MAIN_OP = 1
LOG_TYPE_DOXY_OP = 2
LOG_TYPE_GIT_OP = 3
LOG_TYPE_SCP_OP = 4

def current_timestamp():
    return datetime.datetime.utcnow()

def current_timestamp_string():
    return current_timestamp().strftime('%Y-%m-%d_%H-%M-%S')

def operation_log(log, logtype):
    #Check if the main log directory exsists
    logdir = NUTFACTORY_PATH+"/log/"
    if logtype == LOG_TYPE_MAIN_OP:
        logfilename_withpath = logdir+MAIN_LOGFILENAME+TIMESTAMP.strftime('%Y-%m-%d_%H-%M-%S')
    elif logtype == LOG_TYPE_DOXY_OP:
        logfilename_withpath = logdir+DOXYGEN_REPORT_FILENAME+TIMESTAMP.strftime('%Y-%m-%d_%H-%M-%S')
    elif logtype == LOG_TYPE_GIT_OP:
        logfilename_withpath = logdir+GIT_REPORT_FILENAME+TIMESTAMP.strftime('%Y-%m-%d_%H-%M-%S')
    elif logtype == LOG_TYPE_SCP_OP:
        logfilename_withpath = logdir+SCP_REPORT_FILENAME+TIMESTAMP.strftime('%Y-%m-%d_%H-%M-%S')
    else:
        logfilename_withpath = logdir+MAIN_LOGFILENAME+TIMESTAMP.strftime('%Y-%m-%d_%H-%M-%S')
    if os.path.isdir(logdir):
        if os.path.isfile(logfilename_withpath):
            f = open(logfilename_withpath, 'a')
            f.write("\n"+log+"\n")
            f.close()
        else:
            f = open(logfilename_withpath, "w+")
            f.write("\n"+log+"\n")
            f.close()
    else:
```

```
        createLogDir = subprocess.Popen('mkdir '+ logdir, shell=True)
        createLogDir.wait()
        f = open(logfilename_withpath, "w+")
        f.write("\n"+log+"\n")
        f.close()


def check_gitlist():
    f = open(NUTFACTORY_PATH+"/"+CONFIG_FOLDER+"/"+GITREPOLIST)
    lines = f.readlines()
    f.close()
    repos = {}
    for line in lines:
        if line[0] != '#':
            repo = line.split()
            if(len(repo) == 1):
                repos[repo[0]] = ""
            elif(len(repo) == 2):
                repos[repo[0]] = repo[1]
    return repos


def scp_result(foldername):
    scp_cmd = "scp -rp " + foldername + " " + SCP_TARGET
    process = subprocess.Popen(scp_cmd, shell=True, stdin = subprocess.
        PIPE, stdout = subprocess.PIPE, stderr=subprocess.PIPE,)
    process.wait()
    output = process.communicate()[0]
    operation_log(output, LOG_TYPE_SCP_OP)


def doxygen_process(foldername, sourcefolder):
    doxyreportfoldername = foldername+"DoxyReport"
    #Baseline file for Doxygen config
    doxygen_baseline_path = NUTFACTORY_PATH+CONFIG_FOLDER+
        DOXYGEN_BASELINE_FILE
    baseline = open(doxygen_baseline_path, "r+")
    lines = baseline.readlines()
    baseline.close()
    #Create a repo specific doxygen config file
    doxyfileName = foldername+"DoxyGenFile"
    doxyfilePath = NUTFACTORY_PATH+"/"+foldername+"/"+sourcefolder+"/"+
        doxyfileName
    doxyfile = open(doxyfilePath, "w+")
    for line in lines:
        if "PROJECT_NAME" in line:
            line = "PROJECT_NAME     = " + foldername+ "\n"
        if "OUTPUT_DIRECTORY" in line:
            line = "OUTPUT_DIRECTORY     = " + NUTFACTORY_PATH +"/"+
                doxyreportfoldername+"/    \n"
        if "INPUT" in line:
            line = "INPUT    = " + NUTFACTORY_PATH + "/"+foldername+"/"+
                sourcefolder+"/    \n"
  if "PROJECT_LOGO" in line:
      line = "PROJECT LOGO = " +  NUTFACTORY_PATH + CONFIG_FOLDER + "logo.
          png \n"
        doxyfile.write(line)
```

```
    doxyfile.close()
    remove_directory(doxyreportfoldername)

    make_directory(NUTFACTORY_PATH+"/"+doxyreportfoldername)
    process = subprocess.Popen('doxygen ' + doxyfilePath,
                               shell=True,
                               stdin=subprocess.PIPE,
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE,)

    output = process.communicate()[0]
    operation_log(output, LOG_TYPE_DOXY_OP)
    scp_result(NUTFACTORY_PATH+"/"+doxyreportfoldername)


def remove_directory(folderPath):
    process = subprocess.Popen('rm -rf ' + folderPath,
                           shell=True,
                           stdin=subprocess.PIPE,
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,)
    string = "\n Deleting folder: "+folderPath +" at " +
        current_timestamp_string() +"\n"
    output = process.communicate()[0]

    if len(output) > 0:
        operation_log(string + "\t message from system: " + output,
            LOG_TYPE_MAIN_OP)
    else:
        operation_log(string, LOG_TYPE_MAIN_OP)


def make_directory(folderPath):
    process = subprocess.Popen('mkdir ' + folderPath,
                           shell=True,
                           stdin=subprocess.PIPE,
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,)
    string = "\n Creating folder: "+folderPath +" at " +
        current_timestamp_string() +"\n"
    output = process.communicate()[0]

    if len(output) > 0:
        operation_log(string + "\t message from system: " + output,
            LOG_TYPE_MAIN_OP)
    else:
        operation_log(string, LOG_TYPE_MAIN_OP)


def git_download():
    repos = check_gitlist()
    operation_log("Starting to download repos at "+
        current_timestamp_string(), LOG_TYPE_MAIN_OP)
    for key, value in repos.items():
        temp = key.split('/')
        foldername = temp[len(temp)-1]
        foldername = foldername[:-4]
```

```
154         make_directory(NUTFACTORY_PATH+foldername)
155         git_cmd = "git clone "+key + " " + NUTFACTORY_PATH + "/" +
                foldername
156         process = subprocess.Popen(git_cmd, shell=True,
157                                    stdin=subprocess.PIPE,
158                                   stdout=subprocess.PIPE,
159                                    stderr=subprocess.PIPE,)
160         process.wait()
161         proc_output = process.communicate()[0]
162         operation_log(proc_output, LOG_TYPE_GIT_OP)
163         operation_log("Attempting to generate Doxygen report for "+
                foldername+" at "+ current_timestamp_string(),
                LOG_TYPE_MAIN_OP)
164         doxygen_process(foldername, value)
165         remove_directory(NUTFACTORY_PATH+foldername)
166
167 git_download()
168 exit()
```

Appendix C: The Software support C daemon source

```
1
2  #include <getopt.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <syslog.h>
6  #include <unistd.h>
7
8  #define DEFAULT_INTERVAL 3600*24
9  #define DEFAULT_LOGFLAG 0
10
11 /* main */
12 int
13 main(int argc, char **argv)
14 {
15         static int      ch, interval, logflag;
16         pid_t   pid, sid;
17
18         interval = DEFAULT_INTERVAL;
19         logflag  = DEFAULT_LOGFLAG;
20
21
22         while ((ch = getopt(argc, argv, "lp:")) != -1) {
23                 switch (ch) {
24                 case 'l':
25                         logflag = 1;
26                         break;
27                 case 'p':
28                         interval = atoi(optarg);
29                         break;
30                 }
31         }
32
33         pid = fork();
34
35         if (pid < 0) {
36                 exit(EXIT_FAILURE);
37         } else if (pid > 0) {
38                 exit(EXIT_SUCCESS);
39         }
40
41         umask(0);
42
43         sid = setsid();
44
45         if (sid < 0) {
46                 exit(EXIT_FAILURE);
47         }
48
49         if ((chdir("/")) < 0) {
50                 exit(EXIT_FAILURE);
51         }
52
53         if (logflag == 1)
54                 syslog (LOG_NOTICE, " started by User %d", getuid ());
55
```

```
56          while (1) {
57                  system("python ~/nutfactory_daemon/githandler.py");
58                  sleep(interval);
59          }
60
61          exit(EXIT_SUCCESS);
62  }
```

Appendix D: NUTS SDD example

NUTS RSS listener

This document is an example of the use of the NUTS SDD template. The example below is a simple Windows Service that checks an RSS feed for news about NUTS and stores such news in a file. The example contains three SDDs, one main SDD and two subordinate SDDs. This example is a bit contrived and there are several bad design choices made within them, however it still serves as a fairly good example of how the SDD template can be used.

Project name/ID

NUTS RSS listener - 01 - ”Background Service”

Creation Date

01/03/2013

Author(s)

Torleif Ajer Thomassen

Change History

04/03/2013 - Clarified when the RSS Listener should check for news updates (06:00), rather than just at a given interval of 24 hours

Context

The NUTS RSS listener is built as a Windows Service. Its objective is to listen to a given RSS feed and pick out news that contain the keyword ”NUTS” or ”NTNU Student test satellite” in it. When it finds news containing these keywords it will log these in a text file. The service will do this at 06:00 each day.

Composition

The service has two major components: the *parser* and the *logger*. The parser is in charge of downloading the RSS feed from the specified address (which is hardcoded for convenience...) and the logger will log any instance of news containing the keywords.

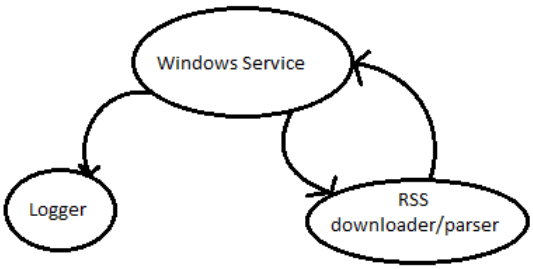

Figure 6.1: Information flow diagram

Implementation

The Windows Service is built using .NET and C# . The basic flow of the system can be described as this:

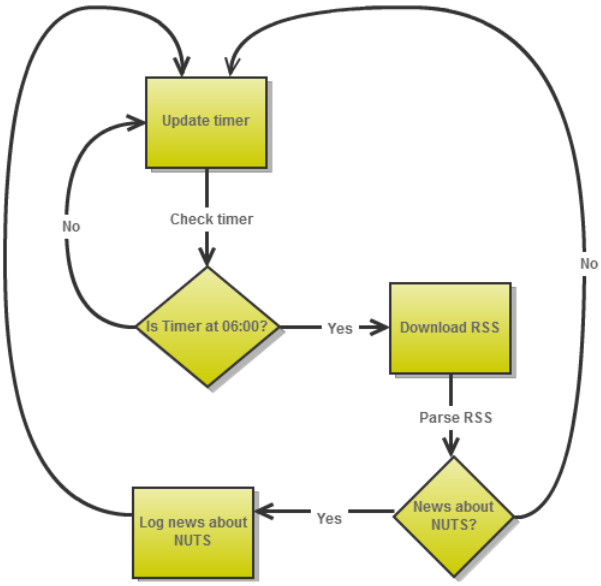

Figure 6.2: Flow chart for NUTS RSS listener

The Windows service creates a TimerCallback[3] delegate that is attached to the .NET Timer object. This delegate is a method in a StatusChecker class. When the timer is raised the delegate checks for RSS updates.

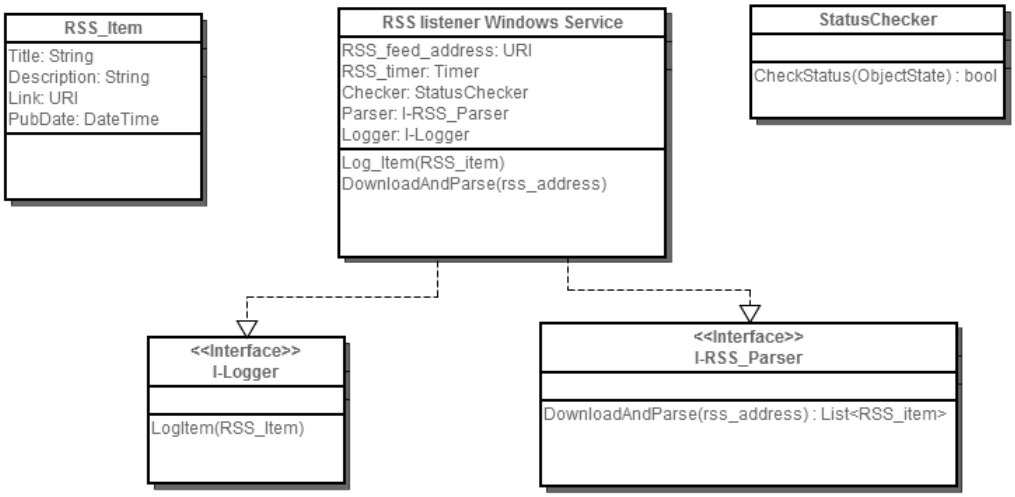

Figure 6.3: UML diagram for NUTS RSS listener

RSS_ item

The RSS_ item class is a simple data storing class used to relay information about a news publication internally.

Dependencies and interfaces

The RSS listener implements two interfaces. One for downloading and parsing RSS and one for Logging. These interfaces are further explained in documents 01-01 “RSS downloader/parser”[1] and 01 - 02 - “Logger”[2]

References

1. NUTS RSS listener - 01 - 01 - “RSS downloader/parser”
2. NUTS RSS listener - 01 - 02 - “Logger”
3. TimerCallBack http://msdn.microsoft.com/en-us/library/system.threading.timercallback.aspx

Project name/ID

NUTS RSS listener - 01 - 01 - ”Downloader/Parser”

Creation Date

01/03/2013

Author(s)

Torleif Ajer Thomassen

Change History

-

Context

The RSS download/parser components is in charge of actually checking for news about NUTS.

Composition

This subsystem contains two components: An interface that provides a method for parsing and downloading, and a class that implements this interface.

Implementation

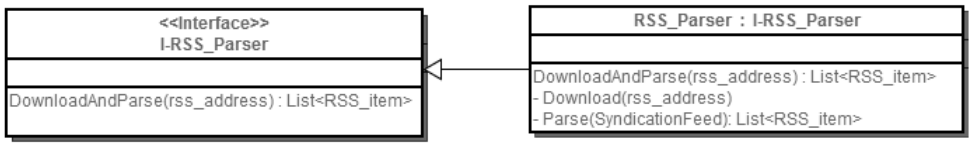

Figure 6.4: UML class diagram for NUTS RSS parser

The feed is downloaded by passing the URL of the RSS feed to the XmlReader.Create method. The implementation uses a *SyndicationFeed* object from the System.ServiceModel.Syndication namespace, the object is instantiated by means of the static method SyndicationFeed.Load which takes a XmlReader object. The feed is then searched for keywords matching ”NUTS” or ”NTNU Student test satellite”.

Dependencies and interfaces

This subsystem exposes the interface *I-RSS_ Parser* , which should be used instead of instantiating the class directly. This serves to keep the system decoupled.

References

1. NUTS RSS listener - 01 - ”Background Service”
2.SyndicationFeed http://msdn.microsoft.com/en
-us/library/system.servicemodel.syndication.syndicationfeed.aspx

Project name/ID

NUTS RSS listener - 01 - 02 - ”Logger”

Creation Date

04/03/2013

Author(s)

Torleif Ajer Thomassen

Change History

-

Context

This subsystem is in charge of logging any news pertaining to NUTS into a text file stored on the host system.

Composition

There are two components to this subsystem, a interface that exposes a method for storing news items to disk and an implementation of this interface.

Implementation

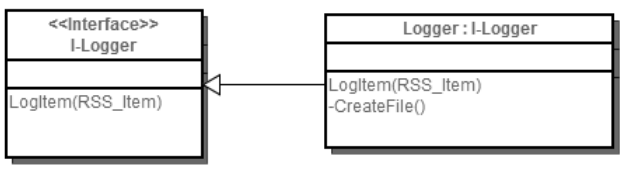

Figure 6.5: UML class diagram for NUTS RSS logger

The logger uses a System.Io.StreamWriter to write a given RSS item to a text file on disk. The CreateFile method is invoked if there are currently no file on disk.

Dependencies and interfaces

This subsystem exposes the interface I-Logger , which should be used instead of instantiating the class directly. This serves to keep the system decoupled.

References

1. NUTS RSS listener - 01 - ”Background Service”

Appendix E: NUTS OBC SDD

This was the first draft for a SDD for the OBC software.

Project name/ID

NUTS 1.1 “OBC”

Creation Date

20/04/2013

Author(s)

Torleif Ajer Thomassen

Change History

-

Context

The OBC software stack runs the hardware component colloquially known as the brain of the satellite. Its prime directive is to handle the housekeeping and all the day to day operations of the satellite. One alternative to the OBC solution is to let the Radio module do all of these tasks, however due to hardware constraints the OBC can have more memory, also the added redundancy will help achieve a more resilient satellite.

Composition

The OBC software stack can be logically broken into a set of services or tasks that run largely autonomously.

Component list

This list is a preliminary list for the services/tasks the OBC offers. Some of these tasks may become relegated to libraries at some point.

- **Housekeeping / logging:** One of the OBC’s tasks is to keep track of and log sensor data from various sensors on the satellite.
- **Payload processing:** While the payload is still not entirely determined, there may be need for a task that handles data from the payload.
- **Module supervisor / Power management:** The OBC needs to be able to turn other modules off and on if they have stopped responding.

- **File system:** Preliminary discussions leads us to believe that the file system should act as a task. Other tasks may queue up files they wish to save and once the file system task can run.
- **Wireless:**
- **Heartbeat monitor:** The OBC must regularly check if the radio is working. In the event that the radio has stopped giving off a heartbeat the OBC has to try and restart the radio module.
- **Intertask communication:**
- **I2C bus:** Intermodule communication

Implementation

The OBC software stack runs on FreeRTOS. This RTOS is chosen for its light weight, wide use and favourable licensing. FreeRTOS can be likened to a thread library where each subsystem runs like a task. The frequency of the task executions depend on the scheduling scheme chosen for FreeRTOS.

- The task scheduling scheme for FreeRTOS has yet to be decided
- The version of FreeRTOS to be used is undecided.
- The memory allocation scheme has not been decided yet.

Dependencies and interfaces

-

References

1. NUTS 1.1.1 ”Housekeeping”
2. NUTS 1.1.2 ”File System”
3. NUTS 1.1.3 ”Wireless”
4. NUTS 1.1.4 ”Payload processing”
5. NUTS 1.1.5 ”Heartbeat monitor”
6. NUTS 1.1.6 ”Intertask communication”
7. NUTS 1.1.7 ”I2C bus”
8. NUTS 1.1.8 ”Module supervisor / Power management”
9. FreeRTOS : http://www.freertos.org/

Glossary

OBC: Onboard computer
RTOS: Real time operating system

Appendix

-

Appendix F: NUTS SDD ID-number reference

1.* NUTS Satellite
2.* NUTS Ground station

1.1.* NUTS OBC
1.2.* NUTS Radio
1.3.* NUTS ADCS
1.4.* NUTS EPS
1.5.* NUTS Payload
1.6.* Internal Wireless Bus