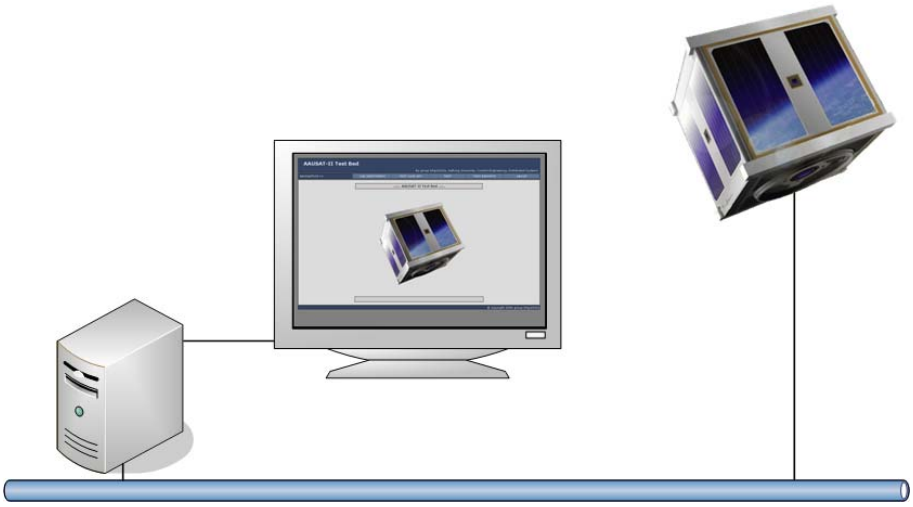
Developing a Test Bed for the AAUSAT-II

Master's Thesis
Distributed Application Engineering



Aalborg University
Institute of Electronic Systems
Department of Control Engineering
Group 04gr1032a
Spring 2004

Institute of Electronic Systems
Department of Control Engineering

TITLE:

Developing a Test Bed for the AAUSAT-II

Master's Thesis
Distributed Application Engineering

ABSTRACT:

This master's thesis deals with the development of a test bed for the AAUSAT-II satellite.

The AAUSAT-II consists of five subsystems connected by a CAN bus, to which the test bed is also connected. On top of the CAN bus protocol, a meta-protocol has been applied for internal satellite communication.

Based on an analysis of existing software testing techniques suitable for testing all phases of the V-model, the most suitable test strategy is chosen, and the necessary functionality is outlined. The test bed makes it possible to perform integration test of the satellite, and it supports unit testing of CAN bus related units. Furthermore, the test bed provides means for simulating non-finished subsystems.

The test bed is designed as three modules. A test bed engine module handles CAN bus communication, execution of planned tests, and subsystem simulation. The planned tests are managed using a web interface module. The CAN bus traffic is displayed by a CAN monitor module, which is also capable of sending user defined CAN frames.

The test bed is implemented on an RTAI patched Linux PC running MySQL and Apache servers. The PC is equipped with four PCI CAN cards. The software is implemented using C, PHP, JavaScript and C++.

Through testing it became evident, that the test bed is able to perform the required tasks. The test bed is functional, reliable, and usable for the intended purpose.

PROJECT PERIOD:
February 2nd, 2004 -
June 3rd, 2004

PROJECT GROUP:
04gr1032a

GROUP MEMBERS:
Morten Tofte Koch
Jesper Noer
Michael Pedersen

SUPERVISOR:
Jens Dalsgaard Nielsen

Copies printed: 9

Thesis pages: 146

Appendix pages: 46

Total number of pages: 192

Aalborg University ♦ Fredrik Bajers Vej 7 ♦ 9220 Aalborg Ø
tlf: 98 15 85 22 ♦ fax: 98 15 97 57

Resumé

Dette speciale omhandler udvikling af en testbænk til AAUSAT-II satellitten.

Satellitten består af fem delsystemer forbundet via en CAN bus, hvortil testbænken ligeledes tilsluttes. En meta-protokol til intern satellit kommunikation anvendes ovenpå CAN bus protokollen.

På baggrund af en analyse af software test teknikker, som kan anvendes i alle V-modellens faser, er den bedst egnede strategi valgt, og den nødvendige funktionalitet skitseret. Testbænken skal gøre det muligt at integrationsteste satellitten, samt yde støtte til enhedstest af enheder som berører CAN bus kommunikation. Endvidere skal testbænken kunne simulere ufærdige subsystemer.

Testbænken er designet i tre moduler. Et test bed engine modul varetager CAN kommunikation, udførelse af planlagte tests og simulering af subsystemer. Planlagte tests administreres via et web interface modul. CAN bus trafikken vises af et CAN monitor modul, som også kan sende brugerdefinerede CAN frames.

Testbænken er implementeret på en RTAI patchet Linux PC, som kører MySQL og Apache servere. PC'en er udstyret med fire PCI CAN kort. Softwaren er implementeret i C, PHP, JavaScript og C++.

Tests har vist at testbænken er i stand til at udføre de krævede opgaver. Testbænken er funktionel, pålidelig og brugbar til formålet.

Preface

This master's thesis is written as the final project documentation of the Distributed Application Engineering specialisation at the Department of Control Engineering, Aalborg University, during the spring semester 2004.

The project group has chosen to work with developing a test bed for the AAUSAT-II. The theme of the project is within the overall directions of the specialisation, given by the department.

The thesis caters for the projects supervisor, the examiners, future students of this specialisation, project groups involved in AAUSAT-II, and other people interested in the topic treated. Some knowledge of distributed systems and programming terms and technology is required, in order to gain full profit of the thesis' contents.

Morten Tofte Koch

Jesper Noer

Michael Pedersen

How to read this Thesis

Bibliographic references are given in square brackets with the last name of the author and year of publishing. These references are listed in the bibliography on page 132. The bibliography is sorted by the order of appearance. If no explicit author can be identified, the company or organisation name is used.

References within the thesis are given with both chapter, section and/or subsection number.

Figures and tables are identified by the chapter number, and an incrementing number within each chapter. The same number scheme is used for equations and formulas.

In text references to software code, such as variables, functions or commands in terminal sessions, are indicated by the `courier` font.

On page 176, a nomenclature, that explains the acronyms and terms used throughout this document, is placed.

The Bibliography and the nomenclature are printed on double size paper, and can be unfolded and visible while reading the thesis.

A CD-ROM, attached to this thesis, includes the source code, screen-shots of the user interfaces, test results, Doxygen documentation of the source code, and other additional material. Appendix I on page 179 shows the table of contents on the CD-ROM. References to files on the CD-ROM are indicated by the CD-ROM logo shown below. The CD-ROM has a self-executable menu, from which the contents can be browsed.

References on CD-ROM:
This is an example of a reference to the enclosed CD-ROM. For details on how to navigate the CD-ROM, see appendix I.

Contents

Introduction

Chapter 1

This chapter contains a description of the initiating problem, and the historical reasons that made the foundations of this thesis. Based on the initiating problem and the project scenario, the scope is considered, and the problem is defined.

Designing reliable and fault-tolerant systems is a challenging task which requires systematic models and structured methods throughout the entire design phase. This is a fact that no engineer or developer would think of calling in question.

When it comes to ensuring that such a system has actually been achieved, the procedure is often somewhat less structured. This is emphasised by the fact that experience has shown, that 50% of a software developers working time is spent on testing and debugging, whereas less than 5% of the educational time is dedicated to these tasks. [Beizer, 1990]

It is commonly understood, that the risk of hazards and malfunctions in a system rises proportional to the complexity of the system. This is indeed the case when building distributed systems, where information flows between multiple hosts, and terms like real-time demands, data consistency, and redundancy dominates the requirements, faced by the system developers.

The advantages of building distributed systems are numerous, such as the possibility of spreading load on multiple hosts, moving data processing closer to sample points and sensors/actuators, adding redundancy by letting multiple hosts be able to perform vital operations, and so on. On the other hand, these improvements come at a cost. When spreading load on multiple hosts, steps need to be taken to retain data consistency. Adding redundancy, whether it is in software or hardware, requires more or less complicated mechanisms to decide whether the original unit or the redundant unit should be used. These side effects makes it even more necessary to test thoroughly.

If the system being built is a part of a critical application, either in terms of safety or mission critical, extensive testing is needed to estimate statistical certainty of a successful mission. This is the case in space applications such as satellites. Once a satellite is put into orbit, the possibility of correcting errors is very limited. It may be possible to upload a new piece of application software to the satellite, but correcting errors in hardware is impossible, and correcting software errors in more vital software, such as operating system or low level software, is often not possible either.

All together, systems developed for space usage, as well as distributed systems in general, require a severe amount of testing, before reliable behaviour can be guaranteed. In September 1999, the Mars Climate Orbiter mission failed after successfully travelling 416 million miles in

41 weeks. It disappeared just as it was about to begin orbiting Mars. The fault should have been revealed by integration testing: Lockhead Martin Astronautics used acceleration data in English units (Pounds), while the Jet Propulsion Laboratory did its calculation in metric units (Newton's). NASA announced a $50.000 project to discover how this could have happened [Jorgensen, 2002]. - They should have read this thesis.

1.1 Initiating Problem

Aalborg University (AAU) started developing the AAUSAT-II satellite in 2003. AAUSAT-II is a pico satellite being developed by students under the specifications defined by the CubeSat concept from Stanford University. [SSDL, 2004] The development process is a spin-off from AAU CubeSat, the first student satellite launched by AAU. The development of this satellite started in September 2001. It was launched in June 2003, and officially announced end of operation in September 2003 [CUBESAT, 2003]. The mission was partly successful, since communication was established with the satellite, but the scientific purpose of taking pictures and sending them back to earth, did not succeed.

The reason why the scientific part of the mission failed is not known. Although some of the housekeeping data which was received from the satellite, indicate that the batteries did not work as well as expected, the cause can not be determined. The time spent on testing the satellite prior to launch was very limited, because the project was delayed, and because the process of assembling and disassembling the satellite to make integration tests etc. was very time consuming, due to a poor designed interface, or lack of experience.

However, in many ways this maiden voyage into space has been nothing less than a success. First of all the publicity given to the project by both local and national medias helped putting the space activities at Aalborg University on the map. A large number of students have become interested in studying and working with space activities, and by making the solutions used on the mission available to the public, many foreign universities are looking to be scientific partners or discuss solutions with Aalborg University. As a concrete spin-off, parts developed for the AAUSAT-II have already been sold to two other satellites.

So, as the space era at AAU continues with the AAUSAT-II, it is desirable to learn from the experience of the previous mission. Therefore it is desirable to introduce a more thorough testing of the satellite, to achieve a higher possibility of successful missions in the future, and to gain experience in building reliable and robust systems. This requires a more structured satellite design, so that a generic interface can be used to access all parts of the satellite during tests, and it requires a test bed to support the developers in performing the tests.

A satellite is a complex architecture consisting of many different parts, carrying out more or less important tasks. To simplify things, these parts are divided into subsystems, where each subsystem is handled by one student group at AAU. This way the project group only needs to focus on one part of the satellite, and the amount of work is easier fit into the boundaries of a semester. One consequence of this working scheme, is that one particular project group may only be working on the satellite for one semester, and they might not be available when the satellite is ready for testing perhaps a year later. Therefore it is vital to educate groups to think

of testing in the earliest possible state of the development process. By making it possible to perform tests during the entire development phase, the test cases needed to test a subsystem are also generated, so that the necessary information needed to test that particular part of the satellite is already present, once the entire satellite is ready for test. To make this possible, a highly adaptable test bed is needed.

The test bed has to make it possible for developers to perform tests, both when their subsystem is fully implemented, and when they just have a few lines of software, they wish to monitor. By simulating a subsystem on the test bed, it also has to be possible to define test cases, so that the subsystem designers also design the test cases needed to perform a thorough test of their subsystem.

When a product is ready for testing, the engineers are often stressed from a hectic design- and implementation phase, where extra working hours have been put into the product to make it finished on time. Therefore the use of the test bed has to be very intuitive, so that the developers are not stressed even more, by having to spend time on learning how to use and operate the test bed.

It is also important, that the test bed is made adaptable to changing requirements as the development of the satellite goes along. This can be achieved by keeping the design modularised, so that parts of the test bed can be changed without requiring complete re-coding of the entire test bed.

Such modifications and improvements are easiest to design and implement in a modularised system, where things are kept as simple as possible. By keeping things simple, the number of errors is also minimised, thereby leading to a more reliable test bed. Such a property is very desirable, since it is very important that a test bed does not lead to wrong conclusions in test cases.

1.2 Project Scenario

The test bed is needed to perform a stepwise integration test of the AAUSAT-II, as well as an acceptance test of this satellite. The AAUSAT-II is a cubic satellite with very small form factor and mass. The satellite is divided into subsystems, where each major functionality is handled by one subsystem, such as on-board computer, communication system, attitude determination and control system, and payload. The payload on the mission is a Gamma Ray Burst Detector delivered by the Danish Space Research Institute (DSRI). A description of the satellite subsystems and the internal communication concept (INSANE) is given in appendix A on page 137.

The subsystems are connected to a common Controller Area Network (CAN) bus, which is also the interface between the satellite and the test bed, as illustrated in figure 1.1.

Since the common medium used for communication in the satellite is the CAN bus, the test bed has to be able to monitor all communication on this medium. This communication must be logged, so that inspection can be made at a later time. The log must include a high resolution time-stamp, so the time of the received CAN frames is known.

It also has to be possible to send frames to the CAN bus, so that various events can be triggered from the test bed.

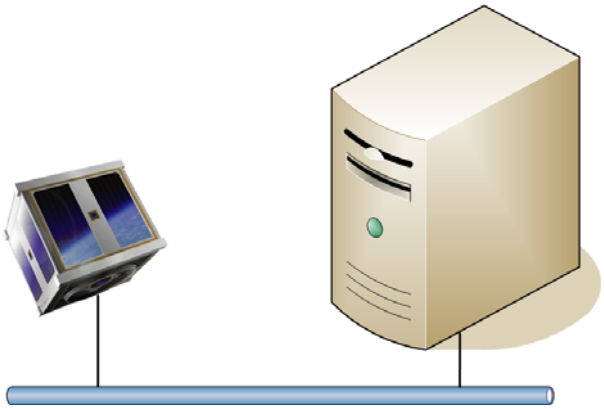


Figure 1.1: *The test bed connected to the AAUSAT-II through CAN bus.*

To determine whether a test case has been fulfilled or not, it must be possible to define a number of conditions when designing a test case. A test case refers to a specific CAN identifier, and it consists of a set of inputs and outputs, that is, the input to be given, and the expected output. But the success criterion does not only depend on the output, but perhaps also the time between the input and output. Therefore also timing requirements must be definable when designing test cases.

It is important that it is possible to run a test automatically, so that no manual actions are required when running an already defined test. This way the amount of labour needed to test, can be reduced. After running a test, the results should be presented to the tester in a logical and intuitive way, that emphasises which test cases have passed and which have failed. The test can then be replayed when errors have been corrected, if any.

1.3 Scope

Testing a satellite prior to putting it into orbit is not limited to testing software quality. Thorough testing of hardware is also necessary to see if the satellite works under high pressure, extreme temperatures, vacuum etc.. However, since these topics are well outside the topic of this thesis, only software testing is considered.

The AAUSAT-II contains many different subsystems, where the development has not been finished, or in some cases not even started. Therefore, complete documentation of every subsystem is not present. This means that test cases can not be fully specified. Consequently, this thesis does not provide a complete set of test cases for each subsystem. Instead an interface for future developers, and/or testers, to add test cases for each subsystem, is given.

1.4 Problem Definition

Based on the previous introduction and initiating problem, the purpose of this project is to develop a test bed for the AAUSAT-II. The test bed must be adaptable to changing specifications which may occur during the development phase of the satellite. Furthermore, the test bed

design and implementation must be documented and modularised, since further development by future students may become necessary.

The test bed must provide services as a CAN bus monitor and logger, a CAN bus event trigger and it must support facilities to run planned tests. A user friendly user interface must be developed to make the user interaction easy.

The implementation of the test bed must be reliable, robust, and fault-tolerant, so that errors in the test bed does not lead to wrong conclusions in test cases. This is achieved by keeping the design simple and modularised.

The use of the test bed must also be kept simple and intuitive, when testing, as well as when setting up test cases and tests.

Part I

Analysis

Part I contains an analysis of the functionality and test methods, to be included in the test bed design.
In chapter 2 the possible ways of performing unit-, integration-, and system testing are analysed according to the V-model, as well as alternative development models.
Chapter 3 contains an analysis of which type of tests can be performed using the test bed, and a description of the requirements the test bed must fulfill.
The analysis part defines the primary design criteria and requirements, on which the test bed design is based.

Test Analysis

CHAPTER 2

This chapter gives an overview of different testing methods. It starts with testing methods used for the structured program development, followed by a description of unit testing, structural testing, and finally integration tests.

The two main reasons for testing are: To make a judgement about quality or acceptability and to discover problems. [Jorgensen, 2002]

All software programmers make errors. Some errors are discovered in the debugging phase of the the software development, but not all. Therefore it is necessary to test. A fault is a result of an error. The faults can be found in the testing phase, and from these faults, the errors must be found in the code by debugging.

The faults can be more or less critical. Table 2.1 shows the faults classified by severity in a bank example.

	Severity:	Consequence:
1.	Mild	Misspelled word.
2.	Moderate	Misleading or redundant information.
3.	Annoying	Truncated names, bill for $0.00.
4.	Disturbing	Some transaction(s) not processed.
5.	Serious	Lose a transaction.
6.	Very serious	Incorrect transaction execution.
7.	Extreme	Frequent "very serious" faults.
8.	Intolerable	Database corruption.
9.	Catastrophic	System shutdown.
10.	Infectious	Shutdown that spreads to others.

Table 2.1: *Faults classified by severity. [Jorgensen, 2002]*

It is very important that satellite software has no serious or higher faults, since it might not be possible to contact it anymore, if the system crashes.

2.1 Test Cases

The essence of software testing is to determine a set of test cases for the piece of software to be tested. A test case contains information about the input and expected output of the software. Inputs and outputs are of two types:

- Inputs: Preconditions and actual input.
- Outputs: Post conditions and actual output.

When software is tested it is important to be sure that the actual input for the software is equal to the preconditions. It is also important that the actual output of the software is matched against the post conditions of the outputs. Only if the actual output equals the expected output, the test case passed successfully. Otherwise it failed and the software must be corrected.

2.2 The V-model

The V-model (also known as the waterfall model) is widely used when software is developed using structured program development. A discussion of other software developing methods is found in section 2.9. The V-model is shown in figure 2.1.

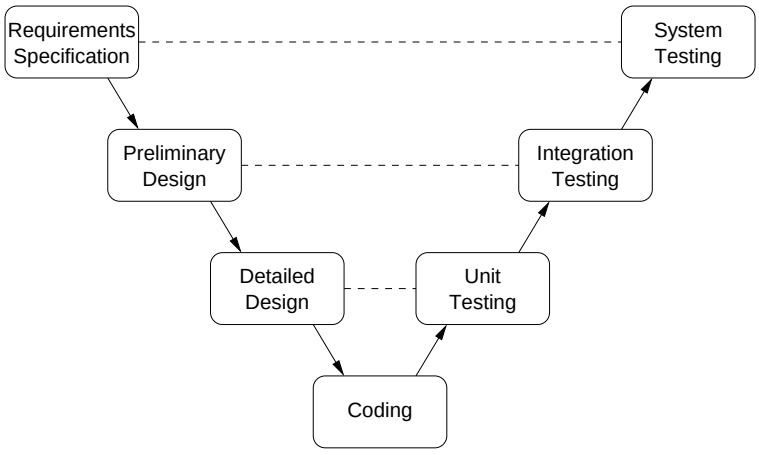


Figure 2.1: *The V-model used for structured software development.*

The V-model has three testing steps. Each step corresponds to a step in the design phase, visualised by the dashed lines in figure 2.1. Unit testing is detailed testing of all the units of the system. Integration testing is testing of several units interconnected. At system testing level the system is tested by a user of the system. That user could be the same as the user who came up with the requirements for the system, and has no understanding of the different units in the system.

Often test is something just to be done when all the code has been made. But that is not recommended since testing is very important. In each design phase, test specification must be made. This has many advantages. The overall time of testing is decreased, because it is easier to generate test cases if the tests are specified in the design phase. A tester that has not been involved in coding the software will have a better overview of the software, if the tester has been working with the specification of the system requirements.

The satellite test bed is primarily used for integration testing. It is expected that all the subsystems of the satellite are unit tested by the individual groups, before they are integration tested at the test bed. The test bed can be used in the unit testing phase, but only for testing the CAN communication capabilities of the subsystem.

Integration testing often uses some of the testing methods used in unit testing. Therefore unit testing methods are described in this thesis.

Three levels of testing are described in details in the following sections.

2.3 Unit Testing

Unit testing is the most straightforward of the three testing levels. Maybe because unit testing often is done by the persons who write the code.

[Jorgensen, 2002] defines a unit as:

- "A unit is the smallest software component that can be compiled and executed."
- "A unit is a software component that would never be assigned to more than one designer to develop."

A program can have many behaviours. The specification behaviour, the actual programmed behaviour, and the tested behaviour. This is shown as a Venn Diagram in figure 2.2.

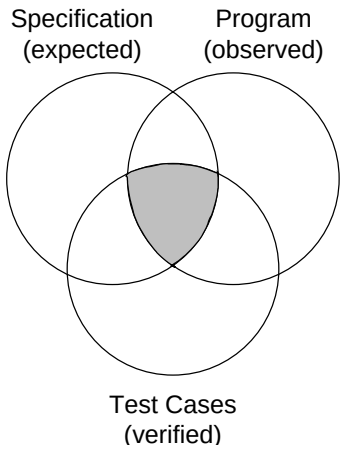


Figure 2.2: *Specified, implemented and tested behaviours. [Jorgensen, 2002]*

The specified behaviour is the expected behaviour of the program, given by the costumer. It can also be referred to as the requirement specification. The program behaviour is the actual implemented software, and the test case behaviour is the verified behaviours of the software. The key issue is to make the innermost shaded area as large as possible. For a programmer, the goal is to understand the requirement specification and implement it correct. For a tester it is to

know the requirement specification, but if the unit tester is the same person as the programmer, the unit tests might not test the specified behaviours, but instead the programmed behaviours.

For an optimal specified, programmed and tested system, these circles connects as one simple ring. Unfortunately this is seldom.

Many different testing methods can be applied at unit testing level. The ones discussed in this section are given by [Beizer, 1990] and [Jorgensen, 2002].

2.3.1 Functional vs. Structural Testing

Unit testing can be divided into functional and structural testing. Functional testing is testing of inputs and outputs. A system is given some inputs and the outputs are measured to see if they correspond to the expected outputs. Functional testing is often referred to as black-box testing, because the tester does not know what is going on in the system.

Structural testing is often referred to as white-box testing. This is because the tester has the complete insight in the software, and is able to go into the very details of every single variable and loop inside the software.

Figure 2.3 compares the Venn Diagrams of functional and structural testing.

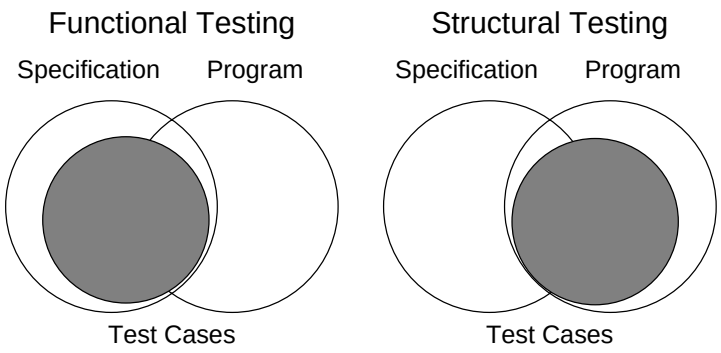


Figure 2.3: *Venn diagrams of functional and structural testing.*

As seen in figure 2.3, functional testing is testing on the specifications, and structural testing is testing of the implemented behaviours, which is illustrated by the shaded circle.

A tester has to decide whether functional or structural testing is to be used (of course both can be chosen). But determining which one is best, is difficult. If not all specified behaviours are implemented, structural testing is not able to test these. Conversely, implemented software that has not been specified, is not tested in functional testing.

As a rule of thumb, structural testing is most appropriate at unit level, while functional testing is most appropriate at integration and system level. Different functional and structural testing techniques are described in the following. Afterwards an example shows the differences between the different techniques.

2.4 Functional Testing

Three different functional testing techniques are described is this section. These are: Boundary value testing, equivalence class testing, and decision table-based testing.

2.4.1 Boundary Value Testing

The idea of boundary value testing, is to test the output of a function when the input is in the outermost extremes. Errors like a "<" that should be replaced by a "$\leq$" and inconsistency between the data types in the input and the data types in the program, are examples of errors found in boundary value testing. The examples used in the following have a function F with two input variables $F(x_1, x_2)$ where

$$\begin{aligned} a \leq \ & x_1 \ \leq b \\ c \leq \ & x_2 \ \leq d \end{aligned}$$

Basic Boundary Value Testing

The basic idea of boundary value testing is to use test cases with input variable values at their minimum, just above minimum, a nominal value, just below maximum, and at their maximum, denoted $< x_{min} >$, $< x_{min+} >$, $< x_{nom} >$, $< x_{max-} >$, and $< x_{max} >$. If there is more than one input variable for the function, the one variable must be held at its nominal value, while the other variables assume their extreme values. This way of testing is based on a single fault assumption, and is depicted for two variables in figure 2.4. The number of test cases increases exponential, according to the number of input variables to the function.

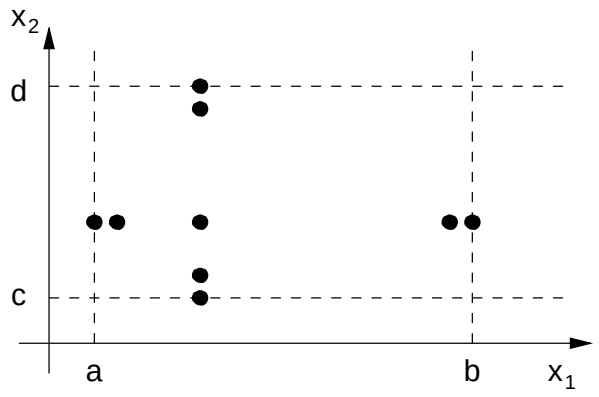


Figure 2.4: *Boundary value test cases for a function of two input variables, x_1 and x_2.*

Robustness Testing

Robustness testing is a simple extension of the basic boundary value testing. The test cases in robust testing are extended with the input variables $< x_{min-} >$ and $< x_{max+} >$. These test cases are added to figure 2.4 and shown in figure 2.5.

Worst Case Testing

Worst case testing investigates what happens if more than one variable reaches its outermost

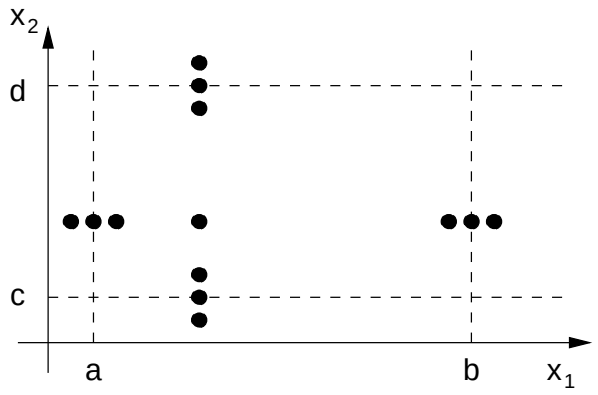


Figure 2.5: *Robustness test cases of a function of two input variables, x_1 and x_2.*

extreme values at the same time. Worst case testing is based on a multiple fault assumption, and thereby more thorough. This is seen by the fact that basic boundary value testing is a subset of worst case testing. The number of test cases also increases dramatically. In boundary testing, the number of test cases is $4 \cdot n + 1$ where n denotes the number of variables. In worst case testing the number of test cases is 5^n. This is visualised in figure 2.6.

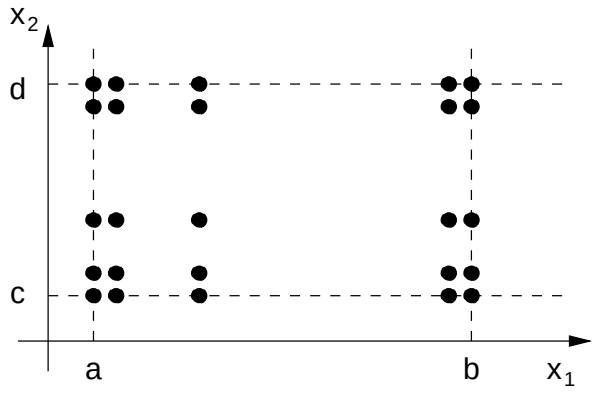


Figure 2.6: *Worst case test cases for a function of two input variables, x_1 and x_2.*

Robustness Worst Case Testing

Figure 2.7 shows the test cases for robust worst case testing. The number of test cases is now increased from $6 \cdot n + 1$ in robust testing to 7^n in robust worst case testing.

Many programmers perform a kind of ad-hoc testing of the software. The test cases are in general based on the programmers experience and best engineering judgement. No guidelines are used, and the quality of the test is very dependent of the programmers abilities as a tester.

Some input variables of a function can sometimes be more interesting to test than other. In the next section, methods to generate test cases for testing special input variable values are

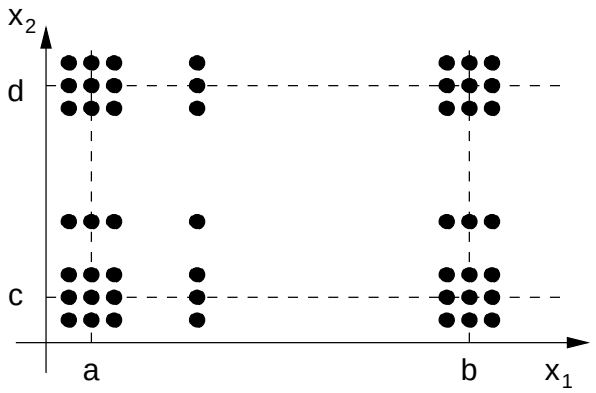


Figure 2.7: *Robustness worst case test cases for a function of two input variables, x_1 and x_2.*

described.

2.4.2 Equivalence Class Testing

The methods used for generating test cases so far, gives a lot of redundant testing, and the testing does not go deep into testing the normal input variable values. This brings a term called equivalence class testing in sight. Equivalence class testing does not fulfil this fully, but for some functions this method is much more valuable than boundary value testing.

The essence of equivalence class testing is to divide the input variable values of the function into classes. The classes must be structured as intervals of the input variable values. The intervals must be found during the design of the function, by a programmer that has insight in using the variable in the function.

Equivalence class testing can be divided into four terms, namely: weak normal, strong normal, weak robust, and strong robust equivalence class testing. Some of the terms may seem contradictory, so in the following sections a description is given.

The same notation as used in boundary value testing is used. The examples used in the following have the function F with two input variables $F(x_1, x_2)$ where

$$a \leq x_1 \leq d \text{ with intervals [a,b[, [b,c[, [c,d]}$$
$$e \leq x_2 \leq g \text{ with intervals [e,f[, [f,g]}$$

Weak Normal Equivalence Class Testing

Weak normal equivalence class testing is testing of at least one variable in each equivalence class (interval) in a test case (single fault assumption).

Figure 2.8 shows an example of the weak normal equivalence test cases for the function F.

Strong Normal Equivalence Class Testing

Strong normal equivalence class testing is based on the multiple fault assumption. As seen in figure 2.9 every combination of possible inputs is covered.

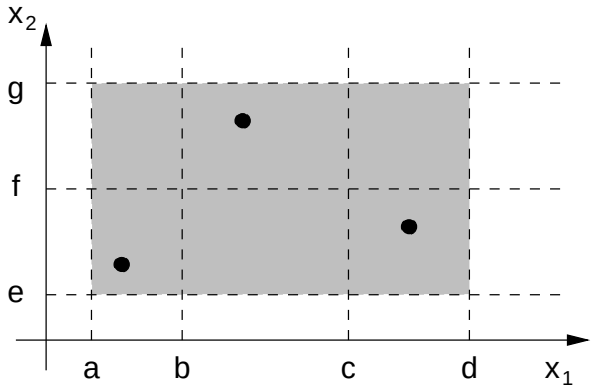


Figure 2.8: *Weak normal equivalence class test cases for a function of two input variables, x_1 and x_2.*

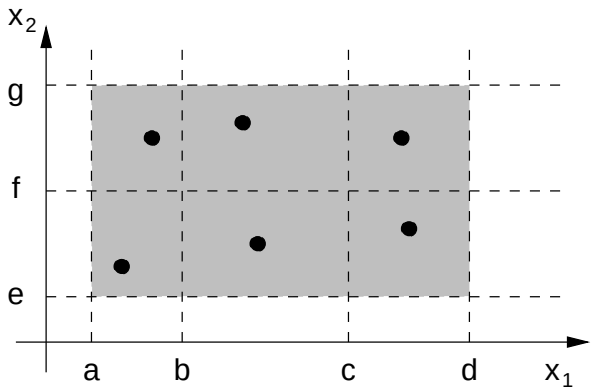


Figure 2.9: *Strong normal equivalence class test cases for a function of two input variables, x_1 and x_2.*

Weak Robust Equivalence Class Testing

Now the robust term comes into the picture again. As in boundary value testing this means that input values outside the range, are used in the test cases. Each valid input must have one value in each valid class (as in weak normal equivalence class testing). For valid inputs, a test case will have one invalid value and the remaining values will be valid. Figure 2.10 shows an example of a set of weak robust equivalence class testing.

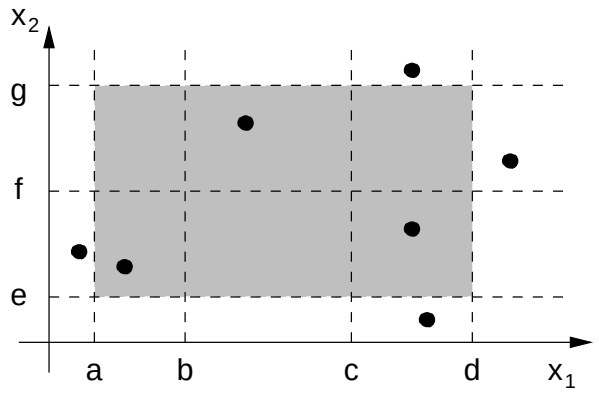


Figure 2.10: *Weak robust equivalence class test cases for a function of two input variables, x_1 and x_2.*

Strong Robust Equivalence Class Testing

The strong robust equivalence class testing contains robustness, and is based on the multiple fault assumption. Test cases are generated at all possible combinations of the input classes and shown in figure 2.11.

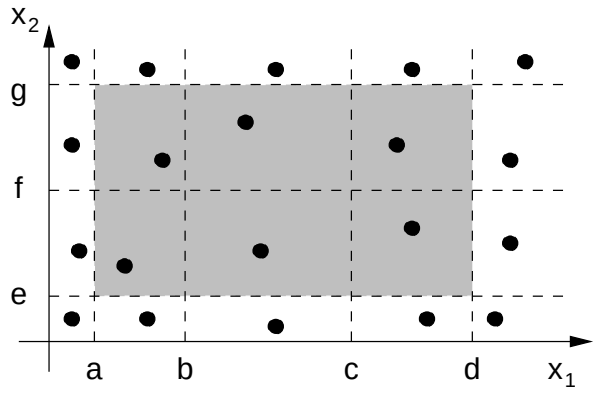


Figure 2.11: *Strong robust equivalence class test cases for a function of two input variables, x_1 and x_2.*

The weak forms of equivalence class testing are obviously not as comprehensive as the strong forms. Equivalence class testing is appropriate when a function is complex. In such a situation

the input must be divided into small or large classes. It takes somewhat more effort to find the right equivalence classes for complex functions, but when it is done, it is more powerful than boundary value testing.

2.4.3 Decision Table Based Testing

By going one step further in details of defining functional test cases, the decision table based testing technique can be discussed.

Decision tables are used to represent and analyse complex logical relationships. A decision table consist of four parts: The leftmost column is the stub portion, the right columns are the entry portion, the upper part is the condition portion, and the lower part is the action portion.

Table 2.2 shows an example of a decision table.

Stub:	Rule 1:	Rule 2:	Rule 3,4:	Rule 5:	Rule 6:	Rule 7,8:
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	-	T	F	-
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

Table 2.2: *Portions of a decision table. T = true, F = false and X = execution. [Jorgensen, 2002]*

Each column in the entry portion of the decision table is a rule. Rules indicate which action is taken for the conditional circumstances, given in the condition portion of the rule. Rule 1 in the table say, that if and only if the three conditions c1, c2 and c3 are true, action a1 and a2 executes.

To identify test cases in a decision table, the inputs are interpreted as conditions and the outputs are interpreted as actions. Since the conditions must be true or false, some conditions of the inputs must be made. This can be all logical expressions, or the input can be split into classes as in the equivalence class testing. The rules are interpreted as test cases. A relatively low number of test cases is assumed for this technique. This is a big advantage, especially if the number of input variables is more than two or three.

How to find the sufficient conditions is often an iterative process. A lot effort is required to find the sufficient condition, and some insight knowledge in the function is valuable when conditions are made. Therefore this technique can be called a grey-box testing technique.

One of the big advantages of decision table based testing, compared to the other functional testing techniques, is that the number of test cases is reduced to a minimum. An example of this is given in section 2.6.

The decision table based technique works well for functions where a lot of decisions needs to be taken. This can for instance be if the function has a prominent if-then-else logic or if there is a logical relationship between the input variables.

More information of the details in the decision table based technique is found in [Beizer, 1990, P.322-332] and [Jorgensen, 2002, P.111-122].

2.5 Structural Testing

As earlier mentioned, structural testing is white-box testing, which means that it is based on testing directly in the source code. In the following, a short description of the two most widely used structural testing techniques, namely path testing and data flow testing, is given. Only a short description of these techniques is given, because structural testing can not easily be applied to integration testing, which this test bed mostly deals with.

2.5.1 Path Testing

All structured programs have a program graph. A program graph is a directed graph in which nodes are statement fragments, and edges represent flow control. Let i and j be two nodes in a program graph. An edge going from i to j represents that the statement fragment corresponding to node j, can be executed immediately after the statement fragment corresponding to node i.

Graph examples of the most used structured programming constructs and loops, are shown in figure 2.12.

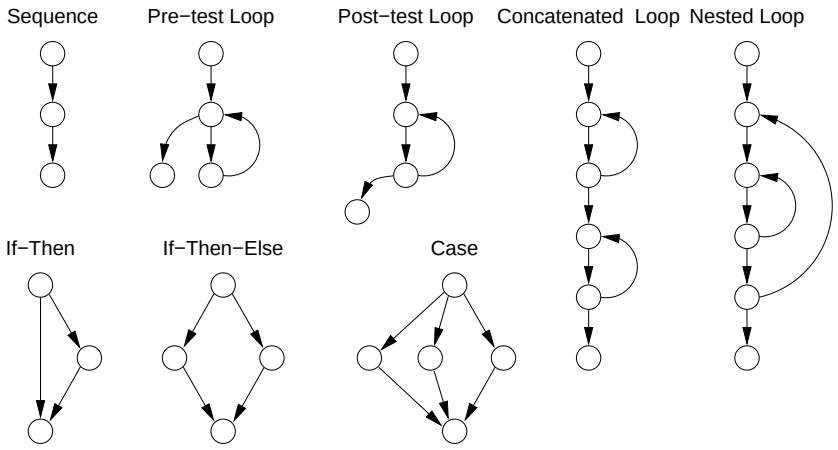


Figure 2.12: *Structured programming constructs and loops.*

A structured program is built of the building blocks in figure 2.12 and it is relatively easy to build a program graph from the program source code. If a program graph does not have any loops, it is a directed acyclic graph.

An example of a directed acyclic program graph is shown in figure 2.13.

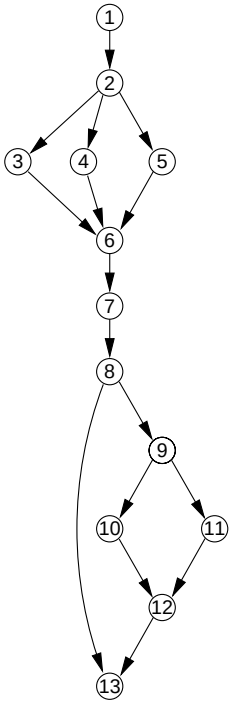


Figure 2.13: *An example of a directed acyclic program graph.*

There are many paths between the source and sink node of a typical routine. Every decision doubles the number of paths, and every loop multiplies the number of paths by the number of iterations of the loop.

The graph in figure 2.13 has nine different paths going from the first source node (node 1) to the last sink node (node 13). All nine paths must be tested before a complete test of the routine is verified. This is relatively easy for this graph because it contains no loops.

Three different path testing strategies can be discussed:

Path testing execute all possible paths through the routine. This is not possible for routines with loops. When path testing is applied, it is said to have achieved 100% path coverage. This is the strongest criterion in the path testing family.

Statement testing execute all statements in the routine at least once. It means that all nodes are covered, leading to say that 100% node coverage or 100% statement coverage is achieved. This strategy is the weakest criterion in the path testing family.

Branch testing execute enough tests to assure that every branch alternative is tested at least once under test. When enough tests are done to achieve this prescription, 100% link coverage or 100% branch coverage is achieved. This strategy is also a weak criterion in the path test family, but stronger than statement testing.

Many other strategies can be made. They are all stronger than the last two described and weaker than 100% path coverage.

If a graph contains loops, boundary value testing can be applied to the index of that loop to test if it is possible to exit the loop. The boundary value testing can be weak, robust, worst case, or robust worst case.

2.5.2 Data Flow Testing

Data flow testing is met by going one step deeper into structural testing. Here the details of each variable in the source code is described fully. This leads to finding bugs such as variables that are defined but never used, variables that are used but never defined, and variables that are defined twice before used.

Two main forms of data flow testing are described, namely define/use testing and slice based testing.

Define/Use Testing

The idea with define/use testing (du-testing), is to find errors like those mentioned above. A program P has a program graph G(P). G(P) has a single entry source node, a single exit sink node and no edges from a node to itself. The set of all paths in P is denoted PATHS(P). The graph in figure 2.13 is an example of a program graph.

A node n can be a defining node or a usage node of a variable v in G(P).

- n is a defining node if and only if v is defined at the statement fragment corresponding to node n. A defining node is written as DEF(v,n)
- n is a usage node if and only if v is used at the statement fragment of node n. A usage node is written as USE(v,n).
- A du-path is a path in PATHS(P) such that for some variable v, the define and usage nodes DEF(v,m) and USE(v,n) are the initial (m) and final (n) node of a path.
- A definition-clear path is a du-path in PATHS(P) with a variable v and initial and final nodes DEF(v,m) and USE(v,n) such that no other node in the path is a defining node of v. A definition-clear path is denoted as a dc-path.

Du-paths and dc-paths describe the flow of data across source statements, from nodes where a variable is defined, to nodes where a variable is used. Du-paths that are not definition-clear are potential trouble spots.

A few examples of du-paths are given below. Figure 2.14 is used to explain the paths p1-p5. DEF(var1,1) and USE(var1,2) gives a du-path p1:

$$\text{p1} \quad = \quad < 1, 2 >$$

p1 is definition-clear.

DEF(var2,5), USE(var2,6), and USE(var2,7) gives two du-paths (if the nodes are in a sequence).

$$\text{p2} \quad = \quad < 5, 6 >$$
$$\text{p3} \quad = \quad < 5, 6, 7 >$$

p2 and p3 are also definition-clear.

DEF(var3,8), DEF(var3,9), and USE(var3,10) also gives two du-paths (if the nodes are in a sequence).

$$\begin{aligned} \text{p4} &= < 8, 9, 10 > \\ \text{p5} &= < 9, 10 > \end{aligned}$$

p4 is not definition-clear while p5 is.

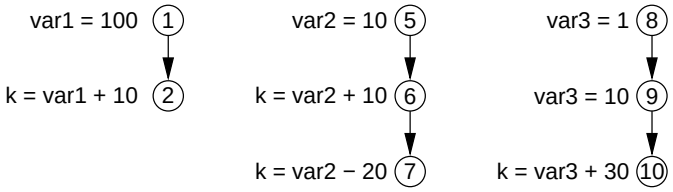


Figure 2.14: *Three parts of a programs graph. Used to explain the five paths, p1 to p5.*

Slice Based Testing

A program slice, is a set of program statements that contribute to, or affect the value of a variable, at some point in the program.

The idea of slices is to separate a program into components that have some useful meaning.

- A slice in a program P with a set of variables V and a statement n, is denoted S(V,n).
- S(V,n) is the set node numbers of all statement fragments in P, prior to n, that contribute to the values of variables in V, at statement fragment n.

The DEF and USE relationships presented in the du-path testing can be identified more specifically. Two forms of definition nodes can be identified:

- I-def, defined by input.
- A-def, defined by assignment.

The relationship usages pertains five forms of usage:

- P-use, used in a predicate.
- C-use, used in a computation.
- O-use, used for output.
- L-use, used for location.
- I-use, used for iteration in loop indices.

An example of slices:
Node 12 in a program contains an A-def of a variable var4, node 14 has a C-use of var4, and node 16 has an O-use of var4. The slices for var4 are:

$$\begin{aligned} \text{s1: S(var4,12)} &= \{12\} \\ \text{s2: S(var4,14)} &= \{12, 14\} \\ \text{s3: S(var4,16)} &= \{12, 14, 16\} \end{aligned}$$

All dependent nodes for each use is a part of the slice. From the slices, direct acyclic graphs can be made, where slices are nodes, and the edges represent the proper subset relationship. This is a way to have full control over every single variable in the program. Several programming environments support the use of slices when programming. This leads to simpler and better testing of the program.

For the GNU C-compiler, tools as `gcov` and `gprof` can be very useful. These are profiling tools, and are both able to make basic performance statistics of the software.

More information about structural testing can be found in [Jorgensen, 2002] and in [Beizer, 1990].

2.6 Comparing the Testing Techniques

[Jorgensen, 2002] gives an example which is good to compare the different techniques. An insurance program computes the car insurance premium based on two parameters, namely the policy holders age and driving record.

The program is tested with some of the functional testing techniques. By using path testing, it turns out that 11 different paths exist in the program. The number of test cases and the paths used in the functional tests are shown in table 2.3.

Functional Testing Technique:	Test Cases:	Paths Covered:
Boundary value	25	p1, p2, p7, p8, p9, p10
Worst case boundary value	273	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
Weak equivalence class	5	p2, p4, p6, p8, p9
Strong equivalence class	25	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
Decision table	10	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10

Table 2.3: *Path coverage of functional testing techniques.*

Table 2.3 shows that structural testing is more thorough than functional testing, since not all paths are covered in functional testing. It also shows that many test cases are needed in functional testing, so there is a lot of redundancy in the test cases.

Three functional testing techniques covers ten of eleven paths in the program, and of these the decision table based testing technique is the technique with fewest test cases, which makes this

method the most rational of the functional testing techniques. But decision table based testing requires a lot effort, so for the test bed the strong equivalence class testing might be the best applicable.

2.7 Integration Testing

All the subsystems of the satellite are unit tested by the developers of the individual subsystems. When the subsystems are interconnected, integration testing is performed to test for bugs that become visible when the subsystems communicate. This is where the test bed is needed most.

If all the subsystems are interconnected all at once, it is a big bang integration. This method is not advisable. It is not easy to keep the overview of the system, and a lot of gaps and redundancies will appear when the tests are made.

An integration tester needs to have an overview of the system and its subsystems to be tested. A decomposition tree is very useful to get such an overview. Figure 2.15 shows a decomposition tree with a system (1), three subsystems (2,3,4), and parts of the subsystems (5,6,7,8,9,10,11).

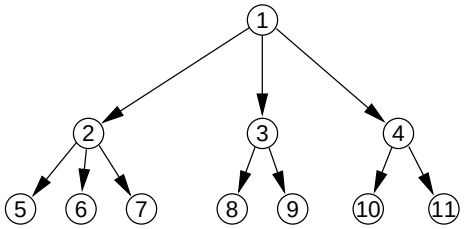


Figure 2.15: *A functional decomposition tree.*

Each of the subsystems, and/or parts of these, may communicate with other subsystems or parts of other subsystems. The rows in table 2.4 shows the nodes in the decomposition graph, and the x's denotes the calls to other nodes, for this example.

A call graph can be made of table 2.4. A call graph is a directed graph in which nodes correspond to program units and edges corresponds to program calls. The call graph for this example is shown in figure 2.16.

The decomposition tree as well as the call graph can be used to make the integration test. The decomposition tree can be used to make a top-down-, bottom-up- and sandwich integration. The call graph can be used to make a pair-wise integration and a neighborhood integration.

Top-Down Integration

The top-down integration begins with the main program in the decomposition tree (node 1 in figure 2.15). It uses a breadth-first traversal, which means that one level of the tree is integrated at a time. This is depicted in figure 2.17.

The lower level units may appear as a "stub", which is pieces of throw-away code, that simulate a called unit. The developing of stubs can be a big task to do for a tester, so it may be a

	2	3	4	5	6	7	8	9	10	11
1	x	x			x				x	
2					x				x	
3				x		x	x	x	x	
4										
5									x	x
6									x	
7										x
8										
9			x							x
10										
11										

Table 2.4: *The adjacency matrix for the decomposition tree.*

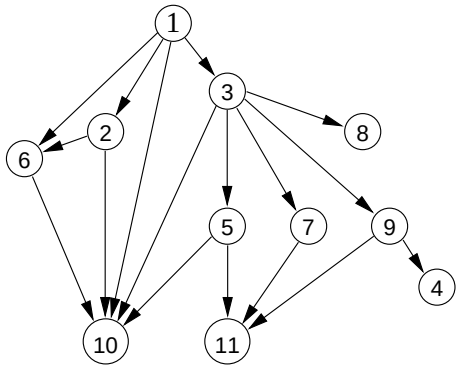


Figure 2.16: *Call graph for the example.*

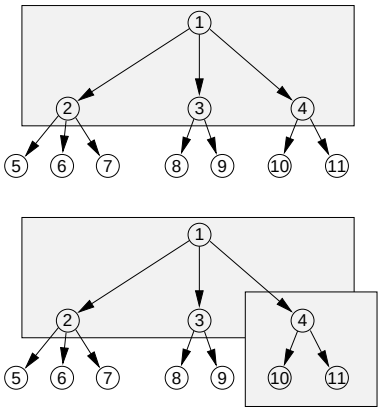


Figure 2.17: *Top-down integration.*

good idea to develop these in the design phase. Otherwise there are commercial tools to help generate stubs, and to do tests with them, exists.

Bottom-Up Integration

The bottom-up integration is the opposite integration. First the subtrees are integrated, one level at a time until the main node is reached. This is depicted in figure 2.18.

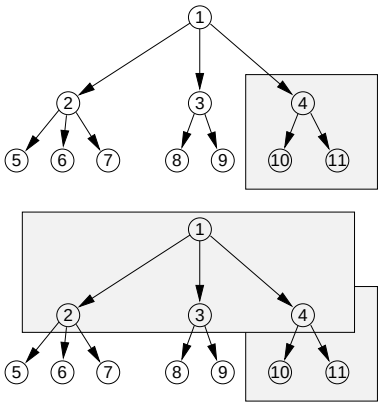


Figure 2.18: *Bottom-up integration.*

In bottom-up integration, drivers instead of stubs are used to simulate other units. A driver is more complicated than a stub, since it has do deal with many calls to other units. On the other hand not as many drivers for bottom-up integration as stubs for top-down integration is needed. In this example, 7 stubs are needed (node 5-11) and only three drivers (node 2-4).

Sandwich Integration

Sandwich integration is a combination of the top-down and the bottom-up integration. The integration is a kind of depth-first integration, which does not need much of stub or driver development. This is depicted in figure 2.19.

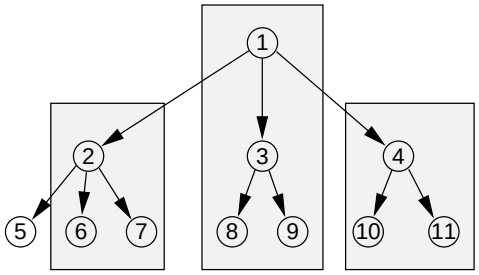


Figure 2.19: *Sandwich integration.*

Sandwich integration makes it very difficult to isolate faults in a system, which is a consequence of big bang integration. Sandwich integration is not advisable.

Pair-Wise Integration

The idea of pair-wise integration is to reduce the effort in developing stubs and drivers, and instead use the actual code. To do this, the call graph from figure 2.16 is used. For each edge in the call graph, there is one integration test. This normally gives just about the same number of test sessions as in the top-down or bottom-up integration. This is depicted in figure 2.20.

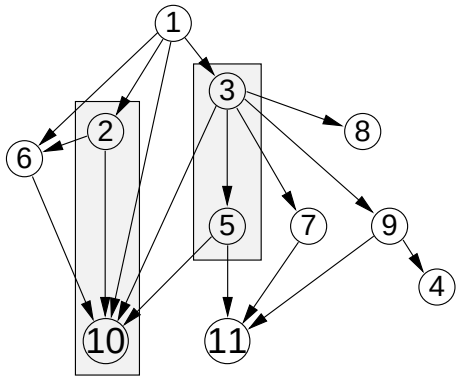


Figure 2.20: *Pair-wise integration.*

Neighborhood Integration

The neighborhood of a graph is the set of nodes that are one edge away from the given node. In figure 2.21 the neighborhood of node two and nine is shown.

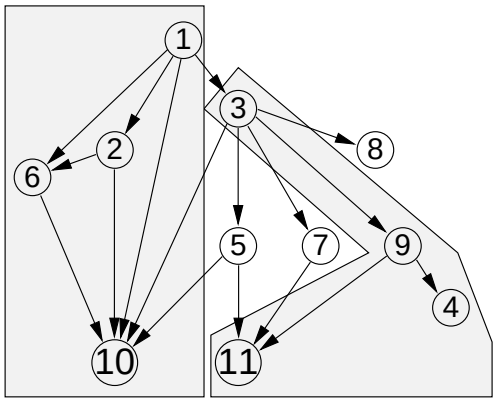


Figure 2.21: *Neighborhood integration.*

The neighborhood nodes corresponds to the stubs and drivers for the given node. This makes the use of stubs and drivers needless. Neighborhood integration also reduces the number of test sessions, but as in sandwich integration, it has problems with the fault isolation.

As unit testing could be divided into functional and structural testing, integration testing can be divided in similar terms. The integration methods described above, all corresponds to unit

functional testing. Path-based integration corresponds to unit structural testing. This will not be explained in this thesis, since it is not sufficient for a test bed used for black-box testing. Path-based integration testing theory is found in [Jorgensen, 2002, Page 216-229].

2.8 System Testing

System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system against the original requirement specification. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery. [Beizer, 1990, Page 22]

System testing is often done by independent testers in cooperation with a user of the system. No special techniques to do a system test is described in this thesis, because the test bed is used for integration test. When all subsystems are connected to the test bed, it can be used for system testing.

2.9 V-model alternatives

The V-model used in this thesis is a great model for software development. The V-model is not the only model which can be used for structural software development and testing. A number of alternatives exist, which is itemized below.

- The chaos model.
- The prototyping model.
- Extreme Programming (XP).
- The spiral model.
- The evolutionary model.

Two derivatives to the V-model, namely the spiral and the evolutionary development models, are examined in the following two sections.

2.9.1 Spiral Model

The methods of the spiral model are:

"For both the evolutionary and single-step approaches, software development shall follow an iterative spiral development process in which continually expanding software versions are based on learning from earlier development." [Boehm, 2001]

The spiral development model is a risk-driven process model generator, which has two main features:

- It uses a cyclic approach for incrementally growing the systems degree of definition and implementation, while decreasing its degree of risk of failures.

- The use of milestones to ensure that the developer reaches a satisfactory system solution within the time frame.

The interesting part of the spiral development model is the risk analysis, and that testing is included within each phase of the development. This implicitly implies that integration becomes more feasible, and by this testing can be minimised because major integration tests have already been carried out.

The spiral model is shown in figure 2.22, where the essential phases are represented.

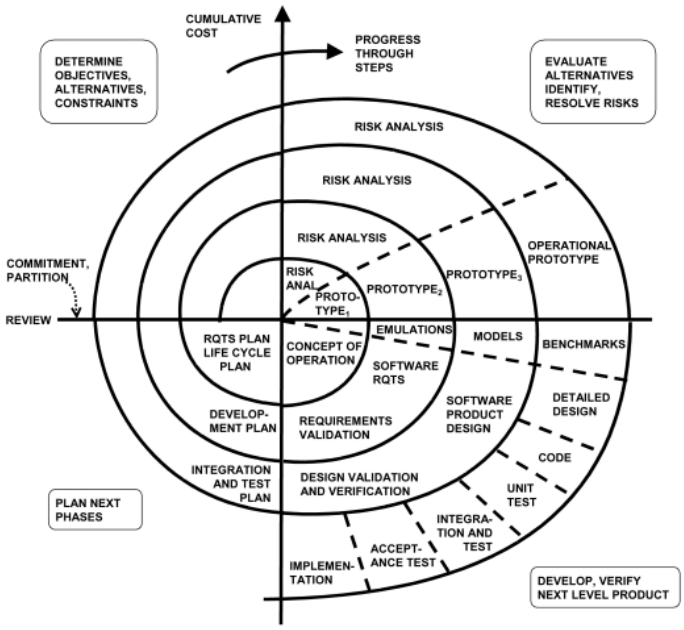


Figure 2.22: *The process of the spiral model. The development phase starts at the inner circle and continuous outwards, as the spiral depicts. [Boehm, 2001]*

By using the spiral model when developing a product, it becomes possible to change requirements within the entire development phases. As new requirements are added or changed to the project, they are analysed in relation to risk, requirements, design, validation, test, etc., before the process continues.

2.9.2 Evolutionary Development

The second alternative development model explained, is the evolutionary project management model.

The fundamental idea of the V-model, and the spiral model, are both to attempt to finish the developed project at a given time limit, even though some of the functionalities are not fully implemented. This is in proportion to the evolutionary model, which finishes the functionalities, but does not guarantee the complete project finishes.

In other words: The V-model and spiral model achieves 100% finished product with each functionality 80% done, within the time limit. While the evolutionary model achieves 80% finished product, but all the implemented functions are 100% done.

Each function is implemented one at a time. At the end of the integration, the function is tested and evaluated against the specifications before another function is implemented. Errors are corrected before the project can proceed. The flow of evolutionary development can be examined in figure 2.23 where each delivered function is qualified before the development can proceed to the next function.

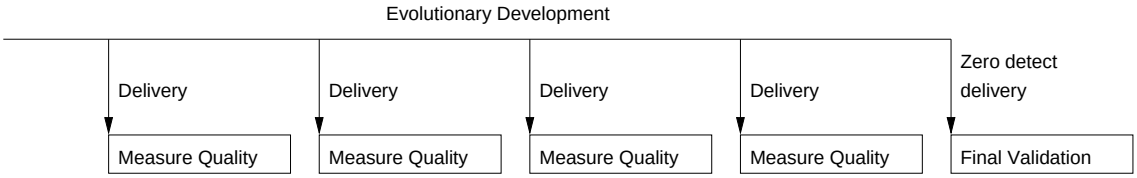


Figure 2.23: *Testing of early deliveries, helps the developer to get ready for zero-defect final delivery. [Malotaux, 2004]*

The differences in project development between V-model/spiral and the evolutionary development model, has influence on how the final system testing is carried out, and the types of bugs discovered by integration testing.

When developing using the V-model, testing is done when the final part is constructed. By this development model there is a potential chance that a lot of bugs first are discovered at this stage, because no continuously integration test has been performed. By using evolutionary development each function is tested in forehand and by this working 100% the integration should be painless. Only fundamental concept errors might be discovered. [Malotaux, 2004]

Test Bed Analysis

CHAPTER 3

This chapter contains an analysis of the functional requirements of the test bed. First an overview of how the errors found in a typical piece of software can be divided into categories, and how large percentage of the total number of errors each category contributes with. Based on this, a test strategy is chosen, and possible error scenarios are outlined. After outlining how test cases are defined and identifying the necessary interfaces, an analysis of the functional requirements is given. Based on the topics covered throughout the analysis, the design requirements are defined.

3.1 Test Bed Objectives

The development phase of a complex distributed system, such as the AAUSAT-II, is an iterative process. Occasionally, results gained and decisions made at different stages of the development process, causes the system designers to reconsider previously made decisions and change their designs or requirements. This may make it necessary for other subsystem developers to do the same thing to their respective subsystems. These situations can be supported and accommodated by applying a well established development model throughout the entire development phase. One such method is the V-model, which is briefly illustrated in section 2.2 on page 10 and described in more detail by [Madsen, 2000].

The fundamental issue of the V-model is, that in every step of the development, results of testing are used as feedback to the specification and design of that particular step and the steps above, as illustrated in figure 2.1 on page 10. One obvious advantage of using this method, is that developers are forced to consider testing throughout the entire development process. When a test bed needs to support this, the nature of the tests that can be performed needs to be flexible.

During early state development, the things to be tested are basic functionality such as CAN bus connection, acceptance filters etc.. As the development progresses, the complexity of the tests follow, and testing topics, such as compliance with agreed message formats and meta-protocols, become relevant. When the development of a subsystem has reached full functionality, it is desirable to determine the achieved robustness and reliability, by probing the subsystem with input variables of both valid and invalid values. Once these tests are successfully completed, the subsystems is said to have passed its unit tests.

The process of test cases becoming more sophisticated is iterated once more, as other subsystems receives unit test approval. These subsystems are then connected, and integration testing

is started. Again the tests start by simple schemes to test whether communication can be established between the subsystems, and evolve towards more complex cases, where the outputs are probed while stressing the inputs.

The third iteration level is taken once all subsystems have been integrated, and the system is ready for the final system testing, against the original requirement specification.

To support such an evolvement in test case complexity, the test bed has to provide means for running both simple connectivity tests and more complex cases of multiple inputs and multiple outputs. In terms of the V-model, the test bed needs to support both unit-, integration-, and system testing. Complete unit- and system testing can not be supported by the test bed, but the parts of these tests that concern testing the CAN communication can be done using the test bed.

When performing system testing, it is often a good idea to have separate persons responsible for the code writing and the testing. This is because programmers may not be aware of which failures they are committing when coding, and by this does not notice the pitfalls when testing. [Beizer, 1990]

The more complicated the system being developed is, the more times the process of design, implementation and testing may need to be iterated.

3.2 Categorisation of Bugs

A test bed is a helpful utility for detecting different types of failures. From the test beds report of a failure, the developers should obtain some log material to be used for debugging the errors causing the fault.

The cause of failures can be categorised into different groups. Table 3.1 shows the distribution of bugs on these groups in a sample software of 6,877,000 statements.

Category:	Number of bugs:	Percentage of bugs:
Requirement Bugs	1,317	8.1 %
Feature and Functionality Bugs	2,624	16.2 %
Structural Bugs	4,082	25.2 %
Data Bugs	3,638	22.4 %
Implementation and Coding Bugs	1,601	9.9 %
Integration Bugs	1,455	9.0 %
System Software Architecture Bugs	282	1.7 %
Test Definition and Execution Bugs	447	2.8 %
Other Bugs	763	4.7 %
Total:	**16,209**	**100 %**

Table 3.1: *Bug statistics from a sample size of 6,877,000 statements including comments. The average is 2.36 bugs per 1,000 statements. [Beizer, 1990, page 57]*

The groups of bugs presented in table 3.1, are described in the following sections.

3.2.1 Requirement Bugs

This group includes bugs caused by incorrect requirements, or misunderstanding the requirements due to improper documentation of these. Such bugs are often caused by misunderstandings between the people implementing, and the people writing the specification. The specification authors may leave out important issues because they take it for granted that the implementers already know it. Another pitfall in this category is specifications that change during design - or even worse - during implementation. The requirement bugs occur in the top most layer of the V-model on page 10. In general, requirement bugs are said to be the first to enter the system, and the last to leave.

3.2.2 Feature and Functionality Bugs

The bugs that arise from badly designed features, or feature interaction, constitute the group of feature and functionality bugs. This category includes bugs that arise from adding features and functionality which seem to "come for free" without much effort. However, experience has shown that this is almost never the case. The interaction between features cause even more concern. Consider, as an example, a telephone with call holding and call forwarding. Call holding makes it possible to put caller one on hold while answering caller two. Call forwarding makes it possible to forward a caller to another phone number. Separately, the testing of these two features is straightforward. But what happens with a third call, when one call is already on hold. And how about phone A forwarding a call to a phone number B, when B is already forwarding a call back to phone A. [Beizer, 1990] The feature and functionality bugs occur in all phases of the V-model.

3.2.3 Structural Bugs

The structural bug group includes bugs caused by path errors, such as unreachable code, improper nesting of loops etc.. These bugs are often somewhat simple to catch, and should be wiped out once a subsystem has passed its unit test. The mean for doing so, is structural testing - especially path testing. Another structural bug is caused by processing. Processing bugs include arithmetic bugs such as algebra, mathematical function evaluation as well as processing in general. One very common source to structural bugs is type conversion from one type to another. Structural bugs have their origin in the detailed design- and coding phases of the V-model, and should not be present after unit testing has been done.

3.2.4 Data Bugs

Data bugs contains all bugs related to the creation, initialisation and format of data objects. Typical examples are wrong indexing of structures and tables, non-initialised variables, or wrong data units. Shared memory is another concept which contributes heavily to data bugs. If a shared resource is not initialised correctly - or is not protected from multiple processes accessing it simultaneously - data bugs are unavoidable. This type of bugs mostly belong to the detailed design- and coding phases.

3.2.5 Implementation and Coding Bugs

Typing errors and syntax errors belong to the implementation and coding group. Syntax errors should be caught by the compiler or translator, as well as most initialisation bugs. One of the most found reasons for coding bugs is actually documentation bugs - or incorrect comments! There seems to be a tendency towards the lifetime of software becoming longer, and code being reused. Therefore programming labor will be dominated by maintenance. This makes incorrect comments a serious problem. If a comment states that a function returns "true" on a specific input, but the actual code correctly returns "false", then the maintenance programmer that needs to interface the function might do it wrong, because he/she takes for granted that the comment is correct.

3.2.6 Integration Bugs

Bugs related to internal and external interfaces are categorised in the integration group. The external interfaces are fixed, and must comply with some dictated protocol. The protocol itself can be wrong, or the implementation of the protocol can be wrong. Common problems with the external interfaces are invalid timing issues and misunderstanding external input and output formats. The internal interfaces are more flexible, because they can be negotiated. The errors found when using external interfaces, can also be found when using internal interfaces. Furthermore, protocol design bugs, and inadequate protection against corrupted data, seems to be widespread as the source of integration bugs. The cause of these bugs is often found in the preliminary design phase of the V-shaped development process, but often the bugs are not eliminated, until integration testing is started.

3.2.7 System Software Architecture Bugs

Bugs arising when using system calls provided by the operating system, or performance issues, exceptions and software architecture bugs are all included in the system software architecture category. One way of avoiding these problems is to use explicitly defined interface modules or macros for operating system calls. Bugs in software architecture can be difficult to find, because they often depend on the load. Hence the system has to be stressed to reveal them. Examples are to assume that no interrupts will occur, bypassing Mutex data locks, assuming memory was initialised etc.. System software architecture bugs arise in the requirement specification- and preliminary design phases, but can be revealed in all of the testing phases of the V-model on page 10.

3.2.8 Test Definition and Execution Bugs

Although testing is supposed to prevent errors from occurring, some errors are also introduced by tests. This is the case in all testing phases of the development process. Performing system- and integration testing requires sophisticated scenarios of preconditions, events and results connected together across interfaces, databases and applications. It can be very difficult to determine test bugs from real bugs. The test designer can misinterpret the specifications, just as well as the program designer can. This leads to incorrect test design. Other test bugs are execution bugs, documentation bugs and incomplete test cases. The most effective way to

reduce test bugs, is to replace manual testing by automatic testing. This can be done using a test bed.

3.3 Test Strategies

The basic test strategy is to send a CAN frame to the satellite and observe the output. This strategy is chosen, because the CAN bus interface is the only interface present in all subsystems.

All of the categories of table 3.1 are important, and should be considered when developing a satellite. The two most widely found groups, structural bugs and data bugs, cover almost half of the discovered bugs. These two categories of bugs should be abated during unit testing, and are not particular easy to support by a test bed.

The third most found group is the features and functionality bugs, holding a total of 16.2 percent of the bugs. The tracking of this type of bugs can be supported by a test bed. The fourth most found group is coding and implementation bugs, which can also be supported by a test bed to some extent.

The most obvious bugs to be eliminated, through the use of a test bed, is integration bugs. This group of bugs represent 9 percent of the total number of bugs found in the statistics of table 3.1. The main purpose of integration testing is to test all internal interfaces. In the AAUSAT-II case, the internal interface is the CAN bus and the INSANE communication agreements on top of this bus. Typically integration bugs does not count the majority of software bugs, but this type of bugs is much more complicated to get rid of, compared to other bugs. Integration bugs are mostly caught late in the system development phase, and may require several modifications in the fundamental software components and data structures, before the error is found and corrected. [Beizer, 1990]

However, for the test bed to be able to provide support in minimising bugs in all categories, and support unit- as well as integration testing, several test strategies must be available. Because the AAUSAT-II is not fully developed yet, complete knowledge of the software structure of each subsystem is not available. Therefore the main focus of the test bed is to support integration testing, because the interface connecting the subsystems has already been defined by the AAUSAT-II steering committee.

In section 2.3.1 structural- and functional testing methods are discussed. Since the satellite is not yet designed or implemented, it is not possible to perform structural integration testing on the satellite, because the software structure is unknown. Therefore functional testing must be applied.

The functional testing techniques boundary value testing, equivalence class testing, and decision table based testing are discussed in section 2.4 on page 13. The decision table based testing method is unfit for the test bed for three reasons. First of all it needs some insight into the code of all individual subsystems, secondly a lot of effort is needed to design each individual test case, and third, the test cases are most likely not able to be reused for other tests, which is desirable.

The main advantage, with decision table based testing, is that not many test cases are needed to run a complete test. This is a huge advantage if many input parameters are present in

the system. But since only one input parameter, the data in the CAN frame, is present in the satellite, the number of test cases to run a boundary value test or an equivalence class test is not alarming, because it does not increase exponentially. To obtain the best trade off between test coverage and the amount of variables to be specified manually, a combination of equivalence class testing and boundary value testing is used. This is done by performing boundary value testing on the interval boundaries. For simplicity reasons this combination of testing methods is referred to as equivalence class testing in the remaining parts of this thesis.

When performing unit tests, the user may need a more simple scheme of testing. Instead of performing complete equivalence class tests, it is desirable to be able to send just a single frame, and observe the output from the satellite. These single frame tests can be used to send simple "ping" frames, self-test frames etc..

Both the single frame testing and the equivalence class testing is intended for sort of well-structured and planned tests. To support the developers even further during product design, the possibility of doing direct and spontaneous communication on the CAN bus is desirable. This is done by using a CAN monitor, as mentioned in the problem definition in section 1.4 on page 4. This CAN monitor is an interface that can be started on the test bed, which provides means for writing to the CAN bus and gives a graphical representation of the ongoing communication on the CAN bus. This is analysed further in section 3.5.

3.3.1 Error Scenarios

When testing the AAUSAT-II to determine if sufficient quality, reliability and robustness have been achieved, several error scenarios should be considered. In the following, an analysis of these scenarios is given.

CAN Communication

Scenarios where communication fails because the transmitting subsystem and the receiver do not agree on the used frame format, may result in critical satellite errors. If the CAN bus controllers used in the subsystems are configured correct, these controllers should be able to handle communication errors automatically, but the test bed should still test this. It is important that the CAN bus controllers used by the test bed are not configured to ignore and automatically fix communication errors. If this is the case, communication errors do not appear in the test reports generated by the test bed.

Given that the identifier is correct, the data of the sent frame has to comply with what the recipient is expecting to receive. If this is not the case, and the recipient does not check the value before using it, unexpected behaviour may occur. This could be the case if a subsystem tries to set a parameter which exceed the expected boundary value of the receiving subsystem.

Deadlocks

Detection of deadlocks should be carried out when testing a subsystem. Unexpected behavior can happen if a subsystem is waiting for a specific frame from another subsystem, which may never come. This can cause a deadlock in the affected system, which makes it impossible to handle all other functionalities in the subsystem.

Livelocks

Another major concern the AAUSAT-II developers have to be aware of, is preventing that the

communication protocol is exposed to livelocks. A livelock could appear if e.g. a subsystem is requesting a specific data value from another subsystem, but the data is never received. If the receiver subsystem continues to request this unavailable data value, then a livelock has occurred. Such loops can have fatal consequences in communication systems, like flooding the communication channel up with useless data.

Sleep Mode

Another important problem is to investigate how subsystems control their sleep mode capabilities, if such operating modes are implemented. Every other subsystem should know when a satellite part has entered sleep mode, and know how to wake the part, so it is possible to establish communication. If a given subsystem periodically enters sleep mode, without notifying other parts of the satellite, deadlocks may occur. Making standardised sleep mode scheduling can be a solution to prevent some system faults, where other systems tries to communicate with a subsystem, that have entered sleep mode or been shut down.

Throughput

A kind of throughput analysis could be performed for each subsystem and compared to the specifications. The CAN bus does only have a certain amount of bandwidth available, and every subsystem should only use the bandwidth it has been assigned. Otherwise there is a risk that a subsystem is blocking the CAN bus for other devices to interact. Occupying the CAN bus too long can also affect the real-time perspective, that some subsystems may be expecting.

Performance

Testing the performance of each subsystem, gives an indication of whether it is possible to handle the expected requests it is designed for, or not. This is extremely important when data networks are used in control systems. As an example, if a sensor and an actuator is implemented as two separate subsystems, then the sensors are connected to the actuators through the CAN bus in another subsystem. If the performance of one system is lower than required, or the bus is loaded, the actuator may not receive a sensor input early enough.

Stress Test

The test bed must contain methods for stress testing the connected subsystems and the satellite. This can be carried out by sending a lot of frames to the given subsystem, and checking whether the system can handle and respond to all requests. The purpose is to monitor that the subsystems does not lock up or break down.

3.3.2 Integration Testing Method

In section 2.7 on page 24, a discussion about different integration methods is found. The discussed methods are: Top-down, bottom-up, sandwich, pair-wise and neighbourhood integration. The three first deals with functional integration testing and the two last deals with structural integration testing.

This test bed is developed before the subsystems of the satellite. That makes it impossible to do structural integration testing on the satellite subsystems, which excludes the pair-wise and neighbourhood integration methods. Of the last three methods sandwich integration is inappropriate, because of the difficulties of isolating the faults in the system.

Bottom-up integration seems the most appropriate for the development of the satellite. One driver is then needed from each subsystem, and the subsystems must perform a kind of integration test of the different parts of the subsystem. The test bed deals with the integration of all the subsystems.

Figure 3.1 shows the bottom-up integration of the satellite subsystems.

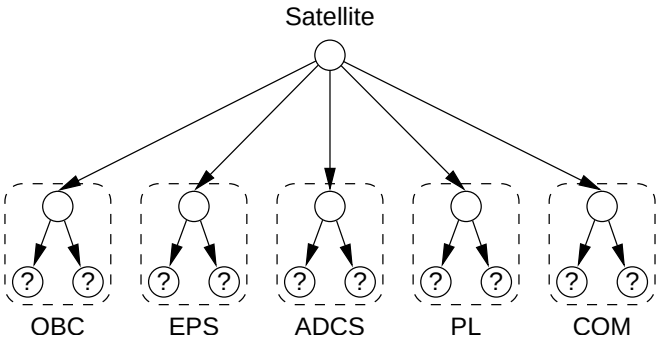


Figure 3.1: *Bottom-up integration of the subsystems in the satellite.*

The subsystems shown in figure 3.1 are described in appendix A.

3.4 Specification of Test Cases

In the most simple version, the execution of a test case consists of transmitting a single CAN frame and evaluating the reply. To be able to specify this, it is necessary that the test bed can maintain a list of CAN identifiers.

When running planned tests by using either equivalence class testing or single frame testing, the test cases have to be defined beforehand. A single frame test requires a single test case, but equivalence class testing requires a set of test cases, for every identifier that needs to be tested.

The reason for this structure is explained in the following.

3.4.1 Equivalence Class Test Case Sets

Equivalence class testing is used to test a subsystem or the entire satellite for many different values of an input, represented by an identifier. The principal behind equivalence class testing is explained in section 2.4.2 on page 15.

When defining a set of Equivalence Class Test Cases (ECTC), the user needs to specify how many intervals the input value is divided into, and the value of the borders of these intervals. If the user requests to divide the input value in N intervals, the user has to specify N+1 border values.

The rest of the input values needed to complete the set of inputs should be generated automatically by the test bed. Since the test method is an equivalence class test, a total of 4·N+3 frames need to be sent, for N chosen intervals. This is illustrated in figure 3.2 for two intervals, A and B.

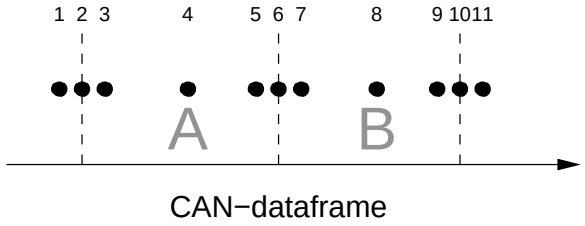


Figure 3.2: *Necessary CAN frames for a chosen number of intervals of two (N=2). Each dot represents a value of the input data, and the dashed vertical lines represent the borders of the intervals.*

It is also necessary to choose the identifier to use for the input frames.

As described in appendix A.4.3, the satellite uses the INSANE concept for internal communication. This concept includes a meta-protocol which is used on top of the CAN protocol. The use of this meta-protocol has a number of consequences the test bed must handle. Section 3.5.2 contains a description of some of these. One consequence is, that the first data byte of every CAN frame is reserved for this meta-protocol. Therefore it is necessary for the user to specify data inputs and outputs in terms of a single byte value, used by the meta-protocol, and a seven byte value available for data.

Specification of the expected output requires somewhat more effort from the user. For each of the 4·N+3 input frames, the expected output must be defined either as a don't care, or an identifier with a data value, or a data value interval. Furthermore, for each input frame, a maximum time before the output should be received, must be specified.

This gives the user the opportunity to test for a given number of intervals of an input parameter value, and to implicitly define which outputs are expected in each interval, and how fast the outputs are expected. Based on the settings of the expected output parameters, and the measured outputs, the test bed determines whether a test case has passed or failed.

3.4.2 Single Frame Test Cases

Single frame testing requires less effort from the user, but does not test the device under test as thorough as the equivalence class test.

When defining Single Frame Test Cases (SFTC) the input is specified in terms of an identifier and a data value. The expected output can be one or more CAN frames. Each expected output frame is specified in terms of an identifier, a data value or a data value interval, and the maximum time before the output should be received.

3.4.3 Planning Tests

A test is defined as a set of ECTC's, SFTC's or both, depending on whether the test consist of equivalence class testing, single frame testing, or a combination of these.

When the user schedules a test, this is done by selecting between the created ECTC's and SFTC's, thereby defining which test cases to include in the test. Furthermore, it is possible to define if the order of execution of these test cases is sequential or random. This makes it possible to test what happens if certain events occur in a random order - instead of the order that the designers may have expected.

It is also possible to define how long time the test bed has to wait between each frame transmission. By adjusting this time, the test bed can be used to stress test the satellite by increasing the CAN bus load, or to determine if a subsystem still responds after a long period of silence.

3.5 Interfaces

The test bed needs interfaces to communicate with the satellite, as well as the user running the tests. This section analyses the requirements set by the interfaces, and the requirements the interfaces need to comply with.

3.5.1 User Interface

For the user to operate the test bed, a user interface is necessary. This interface must make it possible for the user to perform:

- Configuration of the test bed.
- CAN bus monitoring.
- Review of test results.

As described in section 3.3, the test bed features both single frame testing and equivalence class testing.

A configuration interface must handle the creation of these tests, by providing means for creating CAN identifiers, test cases using these identifiers, and tests including sets of test cases.

The CAN monitor interface is used to monitor the CAN traffic during tests, and to spontaneously send frames on the CAN bus.

Once tests have completed, the test results can be reviewed by the user. A tool for presenting the result of the finished tests must be developed. The collected and analysed data might be displayed on a graphical user interface, for easier and more intuitive understanding of the result. Measurements and statistical information can be presented by graphs and histograms. Counters showing values of the number of transmitted and received frames, as well as frames containing errors, could also be displayed on the user interface.

3.5.2 Satellite Interface

The interface to the satellite is agreed by the AAUSAT-II steering committee to be the CAN bus. This interface is used to give an input to the devices under test, and to receive an output. Based on a comparison between the achieved output, and the expected output defined in the test specification, the test bed has to determine whether the test passed or not. In every test case the CAN bus is used to interface the satellite or subsystem to the test bed, as it is the only common data interface, all subsystems are equipped with.

INSANE Interface

The steering committee of the AAUSAT-II has also decided to use the INSANE concept as a meta-protocol on top of the CAN bus. INSANE, which is briefly described in appendix A.4.3 on page 140, has some impact on the performance of the CAN bus network inside the satellite, and also establish a few requirements that the test bed must meet.

The objectives stated prior to developing the INSANE concept, was to establish a very flexible protocol stack requiring a minimum of identifier space.

The INSANE protocol uses a single 11 bit CAN identifier which all CAN units must receive. This identifier is always present in the standard identifier field of a CAN frame, and the extended field is used for "unique identifier" which is used to determine which subsystem or application is transmitting the frame. Hence, the CAN format is fixed to be using extended identifiers. The CAN data field is used for data transfer, as well as INSANE protocol layer distinguishing and frame type. The first data byte, B0, is used to determine which INSANE protocol layer the frame belongs to. Depending on the value, this can be either Datalink, Network, Application, or reserved for future use. Depending on which layer is used, the data bytes B1-B4 are also used for "payload description".

All together the INSANE concept features a large overhead taking up much of the data bytes. Instead of using more of the half billion available identifiers, the INSANE architects have chosen to occupy more than half of the data transfer capacity with protocol overhead.

Along with the protocol stack, an INSANE API has been developed, providing the subsystem developers with a set of functions for implementing communication according to the INSANE concept. The API features functions such as `send()`, `receive()`, and `subscribe()`, but for some reason all function calls are blocking. In the description of the API it says:

> *Please note that* all *calls are* blocking. *However, only the* `send()` *and* `receive()` *calls can be expected to block for longer periods, since only these are directly dependent on network traffic.*

All together, the INSANE concept seems very complicated and overkill for a satellite such as the AAUSAT-II. In order to have a successful mission, the communication scheme should be kept as simple as possible. Therefore it seems more relevant, to rely on normal CAN bus communication, and use the identifiers to separate frame types etc.. In the current suggestions, the INSANE concept uses approximately five CAN identifiers per subsystem. Furthermore, specifying a timing issue as "blocking for longer periods" does not seem to be acceptable in a system where real-time performance may be necessary under some circumstances.

INSANE Specification on CD-ROM:
The available specification of the INSANE concept and API can be found on the enclosed CD-ROM.

3.6 Functionality Analysis

To fulfill the requirements of the problem definition in section 1.4 on page 4 and maximise the detection skills of the test bed in the bugs categorised in section 3.2, the functionality requirements need to be analysed. This is done in the following.

3.6.1 Testing Methods

As derived from the analysis of the test bed objectives in section 3.1, the test bed has to support both integration testing and to some extent unit testing. This is necessary to support the satellite development in all phases of the V-model.

To minimise the risk of introducing test bugs, it is important that the test bed makes it possible to run tests automatically without human interaction during the test. This is also advantageous, because the large series of equivalence class tests to be executed is a very tedious task to do manually.

3.6.2 CAN Monitor

The problem definition states that the test bed has to include a utility to send frames on the CAN bus as well as monitoring and logging the traffic on the CAN bus. Such a utility works like a network sniffer that captures every transmitted CAN frame, and stores the information with a time stamp for further treatment. The resolution of the time stamp has to be sufficient, to make it possible to determine the exact order of frames. The maximum transfer rate is 1 Mbit/s, which indicates that a time resolution of 1 microsecond is sufficient. This time stamp is used to evaluate whether the timing requirements defined in a test case have been met.

By using the CAN monitor, it must be possible to send both data- and remote frames with both standard- and extended identifiers. This feature is primarily intended to support unit testing in collaboration with the single frame testing feature.

3.6.3 Logging Test Data

While tests are running, whether it is spontaneous tests through the CAN monitor or automated tests, all test data must be logged. It is important that the logging is done using technology that makes it possible to process the data at a later time. This may become very important, if some situation requires in-deep analysis of what happens under certain circumstances, once the satellite is launched. The test data may also become valuable for comparing performance of models with the housekeeping data received, when the satellite is in mission.

3.6.4 Subsystem Simulation

To encourage subsystem developers to begin thinking of testing their subsystem as early in the development phase as possible, it must be possible to include a subsystem in the test bed before the subsystem is actually built. This can be done by letting the test bed software be able to simulate subsystems in drivers or stubs. This way the subsystem developers can test e.g. algorithms without having to implement CAN interfaces etc.. Each subsystem simulated in the

test bed must be equipped with its own CAN interface. When planning a test, the user must decide which subsystems should be simulated using the subsystem drivers.

The subsystem drivers must provide an application interface that allows the subsystems to send frames to the CAN bus, configure the acceptance filter of the CAN bus interface etc.. The drivers must be notified, when data for the subsystem is received.

3.6.5 Modular Structure

In the introduction it is emphasised that it is important that the test bed is made adaptable to changing requirements. This is necessary, because the entire satellite has not been designed yet. The best way to ensure adaptability, is to make a modularised design structure, so that adaptions can be adjusted only in the necessary modules, without having to redesign the entire test bed. This also makes it possible to use the test bed as a general test tool for CAN bus based systems in the future.

Very often, when dealing with university projects such as the AAUSAT-II, the time available for testing is limited. Therefore it would be practical if it is possible to work with entering test cases, identifiers etc. at the same time a test executed. This increases the efficiency, and the number of tests that can be executed. To make the system capable of handling this simultaneous use, also requires a modularised structure.

To minimise the risk of introducing test bugs, the design and implementation must be kept as simple as possible. Especially the handling of logged data from the CAN bus must be kept at a minimum, so that erroneous results and conclusions are not taken based on incorrect conversions and type casts done in the test bed.

3.6.6 Analysing Results

The test bed needs to be able to automatically determine whether a test case has passed or failed its requirements. When a series of tests has been made, a report must be available to the user. The report should include the list of the executed test cases, and the results of each test case. The traffic logged during the test, should also be shown in the report, with the frames that causes test cases to pass emphasised.

3.6.7 Expandability

By making the test bed design modularised, it becomes possible to expand the functionality with new features. One such feature could be to exploit the results of test cases to handle software bugs. In such a system, a failed test case could be connected to a specific bug, and the responsible developer could be notified. Once the bug is solved, the developer marks the bug state as fixed, and the test bed could re-run the test that found the bug to see if the bug appears to be solved.

Another feature that could be used to expand the functionality of the test bed is to make it possible to simulate subsystems directly in a tool like Simulink. This could make it possible for the subsystem developers to design their subsystem as blocks in Simulink, and have Simulink generate the driver code to test it with the test bed. The obtained driver code could then

form the basis of the final subsystem implementation. Such a system would be an advantage for subsystems working with mathematical models and heavy calculations, such as the ADCS subsystem.

A process monitor in the test bed could be used to determine which subsystems are running, and what kind of functions or algorithms are running in the subsystem drivers. The process monitor could keep track of which processes are associated to which subsystem and store a time stamp of the process execution. This type of monitor is most relevant for debugging of the software implemented subsystems, but may put too much complexity on the test bed design.

The test bed has to be able to analyse recorded data and presenting the relevant data for the user, in terms of whether a test succeeded or not. This data analysis could be expanded to monitoring all two-way communication between subsystems. This would make it possible to trace a specific event, and investigate whether the transferred CAN frames are transferred as expected, and the event is handled by the subsystems as it should be. Such an analysis of data could also calculate the actual network utilisation, so that the throughput from the subsystems can be estimated and evaluated. Performance parameters could be extracted from the data available, and allow the user to assess whether the given subsystem uses to much bandwidth compared to the specified amount.

3.7 Design Criteria

The analysis done in the previous sections have revealed a number of features and functionality which could be useful to have in the test bed. However, there is a trade-off between adding functionalities, and maintaining simplicity, robustness, and reliability.

Reliability and robustness in the test bed might be underrated, compared to the necessity of reliability and robustness in the satellite itself. However, the latter can not be guaranteed, unless the test bed itself is reliable and robust. Therefore reliability and robustness are very important criteria when designing the test bed. One way to obtain this, is to keep solutions simple, and only add the necessary functionalities. Introducing a modularised design, with sharply defined interfaces between important and less important parts of the test bed is another method.

Once a reliable and robust test bed has been achieved, the biggest threat against a successful test phase, is that the test bed is not used at all. To prevent this, it is very important that getting started with the test bed and becoming acquainted with it is intuitive. Since the AAUSAT-II project consists of many involved people cooperating on building a complex distributed system, within the boundaries of a sharp launch deadline, the probability of people becoming stressed in the final phase of development is high. When people are in a hurry finalising the satellite, they will not use a test bed where they need to read several hundred pages of manuals just to perform a simple ping test. Therefore it is extremely important that the user quickly realises the potential of the test bed by having a break-through experience. This can be obtained through the CAN monitor, where the users immediately can see that CAN communication works. Usability is also a very important factor when designing the web interface.

Because the satellite is not fully designed nor implemented, the test bed must be designed very adaptable. By doing so, the test bed can be adapted to changing specifications that may occur.

This is achieved as another benefit of a modular design, because the adapting then only require that some of the modules are redesigned.

A modularised structure also makes it possible for future groups of students to add functionality to the test bed. However, this requires that the design and implementation of the test bed is documented thoroughly, so that the interfaces between the modules of the test bed can be clearly identified, and used by future developers.

These considerations leads to the following design criteria:

- Reliability and Robustness - through a minimal amount of features and simple design.
- Adaptable - to ensure that future requirements can be met.
- Modularity - to ensure that future development is possible and robustness is obtained.
- Intuitivity and Simplicity - in user interfaces and daily use.

3.8 Design Requirements

Based on the analysis of requirements in section 3.1 to 3.6.7, the desirable functionality of the test bed is outlined. By evaluating these requirements against the design criteria in section 3.7, the final requirements to be used in the design on the test bed can be chosen. This is done in the following.

The purpose of providing the AAUSAT-II developers with a generic tool for testing in all phases of the V-shaped development process, yields that the following test types must be possible:

- Single frame testing.
- Equivalence class testing.
- Spontaneous manual testing (through the CAN monitor).

For the first two of these test types, the test bed has to evaluate whether a test case succeeded or not, based on the users specification of:

- Input identifier.
- Input data value or interval.
- Expected output identifier.
- Expected output data value or interval.
- Maximum output delay.

When defining a test as a set of test cases, it must be possible to specify:

- Which test case sets to run.

- In which order to run the test case sets.
- Which subsystems drivers to include in test.
- The time interval between transmitting frames.

The logging of traffic must be provided with a time stamp on each CAN frame with sufficient resolution to uniquely identify that particular frame in a log.

Furthermore, to support testing of subsystems at a very early development stage, it must be possible to include subsystems in tests at the following levels:

- Physically connected.
- Simulated in test bed software, using drivers.

The necessity of obtaining a robust and reliable test bed that can be further improved by future students, and is able to adapt to changing requirements that may occur as the AAUSAT-II evolves towards launch, the test bed design has to be:

- Reliable.
- Modularised.
- Kept simple.

The criteria of an intuitive system that is easily operated requires graphical user interfaces providing the following functionalities:

- Configuration of identifiers, test case sets, and tests.
- CAN monitoring (send frames manually and monitor traffic).
- Reviewing test results.

The items presented in this section form the requirements needed to design the test bed.

Part II

Design

Part II contains the software and hardware design of the test bed to be used for testing the AAUSAT-II.
Based on the functional requirements and conditions analysed in the previous part, the necessary services to be delivered by the test bed are designed.
The test bed design is separated into three modules. The interfaces between these modules are defined, and each module is designed separately.

Test Bed Design

CHAPTER 4

This chapter contains the conceptual design of the test bed. The analysis in chapter 3 emphasises the necessity of a modular structure. This structure is outlined in the conceptual design, and the interfaces between the modules are designed. The design of each module is carried out in the following chapters.

4.1 Conceptual Design

The test bed analysis defines the design requirements, on which the conceptual design is based. The basic functionality of the test bed is to send and receive data on the CAN bus, to log the traffic, and process this traffic to determine if specified requirements have been met.

Due to reasons mentioned in the test bed analysis in chapter 3, one of the most important design requirements is modularity. When observing the test bed functionality, a number of module candidates can be identified.

An obvious module candidate, is the software needed to handle the subsystem drivers and the interfacing with the CAN bus. This module has more critical timing issues than the user interfaces, because this module has actual impact on wether a test is performed correctly or not. Because the module includes communication on the CAN bus, the programming language used to implement this module needs to be a language that includes support for hardware near programming, such as CAN device drivers.

Depending on how much the test bed is used, the test results generate a rather large amount of data. In order to be able to re-analyse the results of a test at a later time, the test bed must store this test data in a re-usable format. The test bed also needs to handle lists of CAN identifiers, test case sets, and tests. This extensive need of data storage and data handling is best achieved through the use of a database.

Another module candidate can be obtained by gathering the user interfaces in one module. This reduces the possibility of errors and performance issues with graphical user interfaces interfering with the performance of other modules. Such structure is known from Unix-based systems such as Linux and BSD, which are all known to be stable. However, the user interfaces of the test bed have different performance requirements with regards to timing.

The CAN monitor needs to send frames on the CAN bus on requests from the user, and therefore needs a direct and immediate access to the CAN bus. The user interface used for configuring the test bed and reviewing test results does not have such timing requirements. Therefore it

seems better to separate the user interface with timing requirements, and the user interface without critical timing requirements in two separate modules.

The test bed analysis stated that it is desirable that the test bed can be used by several people simultaneously, meaning that while a test is running, other developers can be defining test case sets or identifiers. To do this, the configuration interface needs to provide remote access to users. This can be done by implementing the interface for configuration and reviewing results as a web based interface. This interface needs to be closely connected with the database, hence these two can be collaborated into one module.

The CAN monitor does not need to be accessible remotely, because it is intended for spontaneous use, when a test engineer is sitting at the test bed and working on some part of the satellite system. It is not expedient to implement this module as a web interface, due to the timing issues described previously. Therefore the CAN monitor must be developed as a graphical user interface available only when working locally on the test bed computer.

When investigating the purpose of the test bed, the results of the analysis, and the observations done in this section, three obvious candidates for modules arise:

- Test Bed Engine (TBE).
  Handles the CAN bus communication and runs the timing-dependent part of the test bed, where performance is critical.
- Web Interface Unit (WIU).
  Manages input from users in terms of configuration of tests, test case sets and identifiers. The module includes a database, and does not communicate directly on the CAN bus.
- CAN Monitor Unit (CMU).
  Handles spontaneous testing by monitoring the CAN bus traffic, and makes it possible for the user to send frames on the CAN bus. The actual CAN bus communication happens through the test bed engine module.

This conceptual design is illustrated in figure 4.1.

4.2 System Architecture

The introduction in chapter 1 mentions a standard PC, equipped with a number of CAN controllers, as the hardware architecture on which the test bed could be implemented. Choosing this platform with standard off-the-shelf components, gives a cheap, stable, and powerful processing platform. Compared to the time needed to develop hardware and software from scratch, the use of a standard PC gives a shorter development time.

The software architecture is based on the operating system Linux, because it contains tools for each of the needed modules. It has good support for hardware near programming, and many possibilities available for database servers, web servers etc.. Furthermore, excellent development tools are available, the full documentation is available, and it is open source and available free of charge.

In the following sections, the architecture of each of the modules defined in section 4.1, is designed.

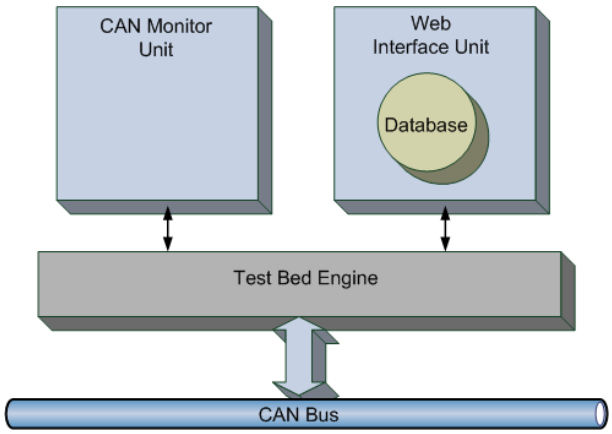


Figure 4.1: *The conceptual design of the test bed.*

4.2.1 Test Bed Engine

The main task of the Test Bed Engine module is to handle communication on the CAN bus. The analysis in chapter 3 states, that it must be possible to simulate satellite subsystems in the test bed, and that each simulated subsystem must have a dedicated CAN interface. This gives a total of 5 CAN ports for subsystems and one for the TBE and CMU. Each CAN PCI card contains 2 CAN ports, corresponding to a total of 4 cards needed for the test bed.

Appendix C contains a description of the card and a hardware test, which is executed to ensure that the cards are compatible with the test bed computer, and that the software for the cards work. Due to reasons described in this appendix, it is necessary that no subsystem share PCI card with the TBE.

The TBE is designed in standard C, because this is the programming language used by the API software delivered with the CAN cards.

When designing a test bed based on 4 CAN cards, then parallel processing becomes necessary. This is achieved by applying POSIX threads.

The TBE module also needs to communicate with the database in the WIU. Therefore an API for this is also needed.

4.2.2 Web Interface Unit

The WIU requires a web server and a database server running on the test bed. When running Linux, the most used web server is Apache. The most used database server that is available for free is MySQL. PHP is a programming language that is used for bringing dynamic contents to web pages. PHP is available as a module for Apache, and has extensive support for MySQL connectivity. Both Apache, MySQL, and PHP are available free of charge, and open source licensed. There is also a MySQL API available for the C programming language, which means that the TBE module can also interface the database.

4.2.3 CAN Monitor Unit

The CMU is used to monitor the bus traffic and send frames to the CAN bus. As stated in the analysis, this module must provide a graphical user interface, that is intuitive to use, and can provide the user with the "break through" experience of seeing communication work.

The graphical user interface is based on Trolltech's Qt, because this platform contains a lot of predefined graphical objects and classes. Another advantage is that Qt is extensively supported and very commonly used for both smaller and larger software projects, such as the KDE desktop environment. Qt is licensed under GPL, meaning that it is available for free for non-commercial use, and that the source code of projects using it, also needs to be open source.

4.2.4 Software Architecture

Based on the chosen architecture of each module, the software structure of the test bed is designed. This is illustrated in figure 4.2.

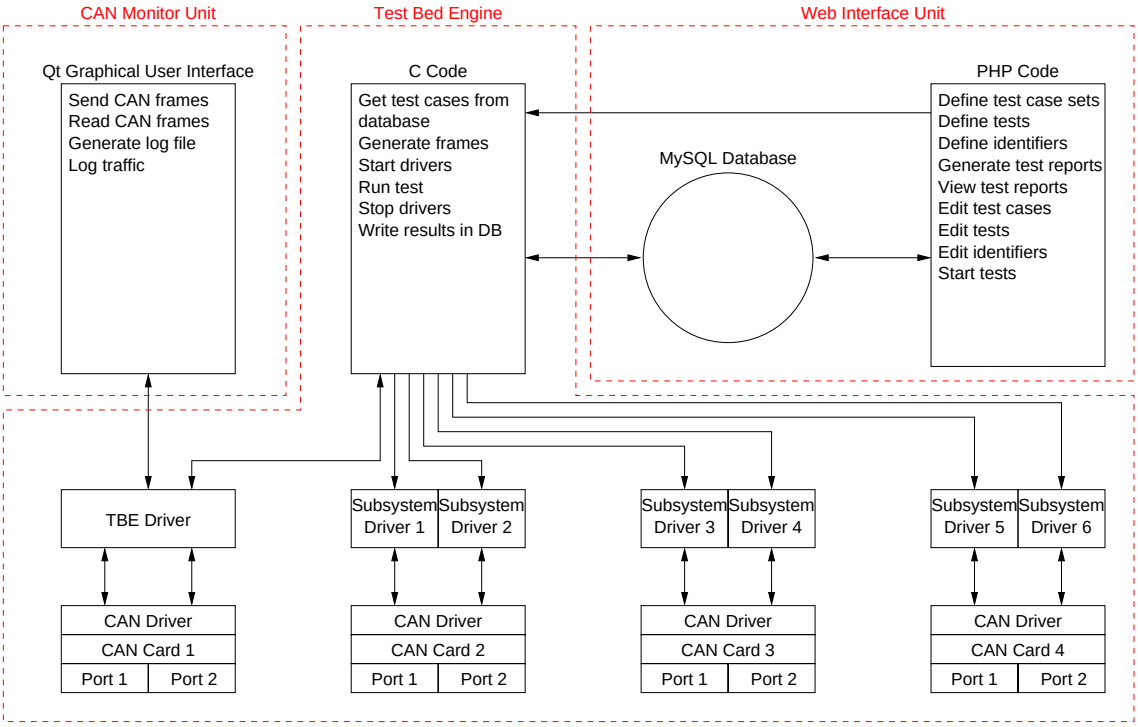


Figure 4.2: *The software structure of the test bed.*

The TBE Driver provides CAN bus access to the CMU and TBE.

4.3 Interfaces

As illustrated in figure 4.2, the test bed contains a number of interfaces, to ensure communication between the test bed modules and the external world. The following two sections defines the internal- and external interfaces, that form the basis for the design of the modules.

4.3.1 External Interfaces

The test bed contains two types of external interfaces for CAN bus communication and the web interface. The CAN bus interface is defined by the CAN 2.0b standard, which is described in appendix B.

The CAN bus interface contains four PCI cards. A high data rate may be present on the CAN bus during tests. If every CAN card is configured to poll the CAN bus for data, then the CPU is using many resources on polling the cards. This is the case whether or not data is present on the CAN bus, and the CPU utilisation is considered waste. Instead of polling, the CPU is noticed by an interrupt handler.

Furthermore, a filter in the CAN cards can be setup to ignore particular identifiers, thereby limiting the number of interrupts needed to be serviced by the test bed CPU.

For ensuring that hardware based subsystems are connected correctly to the test bed, a CAN-connector box is designed.

The wiring of the box consists of six serial connected DSUB9 hardware terminals, terminated at each end by a 120 Ω resistor. At the opposite end of the bus, eight DSUB9 plugs are mounted to connect to the PCI CAN cards. The CAN-connector box ensures that the required CAN termination is correct.

The electrical characteristics on the AAUSAT-II has defined that the voltage of CAN_L - CAN_H are 0 - 3.3 V. This range is supported by the chosen PCI cards, because these are equipped with the PCA82C251 transceiver from Philips. This transceiver is able to operate at this differential voltage.

The interface used for the web interface is an Ethernet connection. This connection is maintained by the operating system, and the Apache web server is configured to listen at port 80.

4.3.2 Internal Interfaces

Internal interfaces are used to connect the modules of the test bed together, and to connect parts of a module where the infra structure implies, that an agreed interface is needed. The test bed contains the following internal interfaces:

- Between CMU and TBE.
- Between WIU and TBE.
- Between TBE and Database.
- The Subsystem Drivers.
- The TBE Driver.

The three topmost of these interfaces, can be identified as the arrows connecting two dashed areas of figure 4.2. The rest of the interfaces are present for infra structural reasons. All five interfaces are defined in the following.

Interface between CAN Monitor and Test Bed Engine

The software on each side of this interface is built in two different programming languages, and executed in different processes. The TBE is based on C, and the CAN monitor is based on C++, because of the Qt graphical user interface. Therefore, the interface needs to be supported by these programming languages, and be able to interface through different processes.

The Linux operating system, gives a number of different methods for inter process communication, discussed in the following items:

File Polling The simplest way of exchanging data between processes is to write data to a file, and then poll this file for reading changes. This method is considered slow, when relative high data rates are expected between interfaces.

IP sockets Using IP sockets for process communication gives the possibility to move the CMU away from the TBE machine. But using IP sockets introduces a relative large overhead, used for routing and data encapsulation. However, this separation would increase the necessity of strict rules and methods for ensuring data integrity. It is considered unnecessary to implement the CMU apart from the engine.

Shared memory Shared memory uses a defined region of the memory to move data between two processes. This type of IPC is a very fast method to share data. But using shared memory, the same type of programming language is preferred at each side of the communicating processes, to keep this data and memory consistent.

Message Queues The chosen IPC method used for connection between CMU and TBE is the IPC message queue, known from the Unix system V. IPC is described in details in appendix F. The communication with the CAN bus passes through the TBE driver.

Interface between Web Interface Unit and Test Bed Engine

Planned tests, such as single frame- or equivalence class tests, are started from the WIU, but the actual testing is done by the TBE. Like the interface between CMU and TBE, the interface between WIU and TBE is also running in different system processes.

The communication between WIU and the TBE only consist of parsing a single value, which is used for identifying which test to run at the TBE. Because this interface is not time critical, it is considered overkill to setup an IPC message queue.

Instead, the interface is achieved by letting the WIU generate a file on the local file system. This file contains the test id of the test to be executed. The TBE probes this file, and starts testing, when the file exists. When the test is complete, and the results are available in the database, the TBE deletes the file again.

The file is also used to ensure that two tests are not run at the same time.

Interface between Test Bed Engine and Database

When the TBE is requested to start a test from the WIU, the TBE connects to the database, and receives the test case sets of that particular test.

Based on the test case set, the TBE generates the necessary CAN frames and executes the test. When the test is finished, all the traffic on the CAN bus during the test is transferred to the database.

The communication with the database is done using the C MySQL API. The format of the data being exchanged is described in section 4.4.

Subsystem Driver Interface

The subsystems to be simulated by using the test bed software, are each provided with an interface. This interface is an API placed on top of the CAN card API, that makes the subsystems capable of using the CAN bus. The subsystem driver is started by the main application of the TBE, and it must provide an interrupt handler. That is executed upon retrieval of a CAN frame, as well as an initialisation routine being executed when the test starts.

Furthermore, a thread in which the primary subsystem routines can be included is needed. This thread is running in parallel with all other threads running in the TBE, hence subsystem developers can implement busy sleeping etc. without degrading the performance of other threads or subsystems dramatically. The subsystem also provides a routine for writing to the CAN bus and the possibility of configuring acceptance filter.

TBE Driver Interface

The TBE driver is somewhat similar to the subsystem drivers, except that it needs to provide CAN bus access to two modules, namely the CMU and the TBE itself, as illustrated in figure 4.2.

This fact implies, that steps need to be taken, to prevent the two modules of accessing the drivers functions simultaneously, when sending data on the CAN bus. When data is received by the TBE driver, the data is forwarded to both TBE and CMU.

4.4 Data Structures

The test bed operates on a number of central data structures used across the previously defined interfaces. These data structures hold the information to be exchanged from one module to another. The structures are defined in the following sections.

4.4.1 Incoming CAN Frame

The data structure of a CAN frame, received by a CAN bus adapter, is handed to either a subsystem driver or the TBE driver. This data structure is mapped directly from the API of the CAN cards to ensure compatibility. The CAN card API supports different types of CAN bus cards, and therefore not all elements in the API structure are necessary with the CAN cards used for the test bed. The relevant parameters of the structure are explained in table 4.1.

The FrameType parameter is used to determine the type of the received CAN frame. The possible values of FrameType are described in appendix C.

Name:	Type:	Size:	Description:
Identifier	Unsigned long	4 bytes	CAN identifier.
DataLength	Integer	4 bytes	Length of data.
CanData[8]	Unsigned char	8 · 1 byte	Data bytes received.
TimeStamp	Unsigned long long	8 bytes	Time stamp with resolution of 1 μs.
FrameType	Integer	4 bytes	The frame type.

Table 4.1: *Parameters in the data structure for incoming CAN frames.*

Name:	Type:	Size:	Description:	
Identifier	Unsigned long	4 bytes	CAN identifier.	
DataLenght	Integer	4 bytes	Length of data.	
CanData[8]	Unsigned char	8 · 1 byte	Data bytes to be sent.	
Extended	Integer	4 bytes	Extended flag:	1 = Ext. identifier.
				0 = Std. identifier.
Remote	Integer	4 bytes	Remote flag:	1 = Remote frame.
				0 = Data frame.

Table 4.2: *Parameters in the data structure for outgoing CAN frames.*

4.4.2 Outgoing CAN Frame

The data structure of a CAN frame to be sent on the CAN bus, being either a frame entered manually by a user on the CMU, or an outgoing frame from the TBE, is shown in table 4.2.

The Extended and Remote parameters are used to determine whether an extended or standard identifier is used, and whether the frame is a data frame or a remote transmission request frame.

4.4.3 Test Case Definition

The test case definitions are stored in the database and retrieved by the TBE before running a test. However, the TBE does not need as much information about a test case as the WIU does. The TBE only needs information about which frames to send and how often. This is a consequence of the WIU being responsible for determining whether a test case failed or succeeded. In order to do this, the WIU needs to know when the test frame is sent, hence this parameter also needs to be a part of the test case definition.

The data structure being exchanged between the TBE and the WIU is described in table 4.3.

Besides exchanging the structure given in table 4.3, a MaxTime and six subsystem variables are transferred.

The MaxTime variable is an integer, that contains information of how long time the TBE has to wait before stopping the test. The MaxTime is expressed in milliseconds, and is the maximum time to wait for the subsystems to reply on a test frame.

The subsystem variables indicates which subsystems are enabled when running the test. The interface makes it possible to enable or disable up to six subsystems. Each subsystem variable uses an integer to present the enable or disable status.

Name:	Type:	Size:	Description:
TestCaseId	Integer	4 bytes	The unique ID of the test.
Identifier	Unsigned long	4 bytes	CAN identifier.
DataLenght	Integer	4 bytes	Length of data.
CanData[8]	Unsigned char	8 · 1 byte	Data bytes to be sent.
TimeStamp	Unsigned long long	8 bytes	Time stamp of transmission time.

Table 4.3: *Parameters exchanged between WIU and TBE to define a test frame.*

The definition of a test case in terms of database variables is somewhat more complex, and is explained in chapter 6 on page 73 and in appendix D.

4.4.4 Test Results Data

The data returned to the database by the TBE when a test is done, consists of all the traffic received by the TBE drivers during the test period, and an update of the test definition data, listed in table 4.3. The traffic being logged, holds the format of incoming CAN frames, defined by the structure in table 4.1.

When a test is done, the TBE must update the database with the time stamp for when the frames are transmitted. This is used by the WIU to find the data on the CAN bus during the test. This is explained further in chapter 6.

The data used by the WIU to generate reports, contains more detailed information. A description of this is found in chapter 6 on page 73 and in appendix D.

4.4.5 Data Formats

To ensure consistency between user inputs and the actual data values used in the test bed, the input/output formats need to be the same on the WIU and the CMU. The data values that need to be specified by, or presented to the user, are given in table 4.4

Name:	Format:	Description:
CAN identifier	Decimal	CAN identifier.
CAN data	Hexadecimal	CAN data values.
Time stamp	Decimal	Time stamp.

Table 4.4: *Input and output formats of data values.*

As described previously, the AAUSAT-II is based on the INSANE concept. This implies that the first data byte of every CAN frame is used for identification purposes in the Meta-protocol defined by INSANE. Therefore CAN data is presented as two values, B0 and B1-B7 when displayed to, or requested from the user.

4.5 Timing

The timing requirements of the test bed are not specified, since the current AAUSAT-II design does not include exact specifications on e.g. how fast subsystems must be able to transmit

frames after each other. But since the test bed is used to sharply determine whether a subsystem passes tests, where one of the criteria is a time limit which a reply CAN frame must be received, it is important that the timing behavior of the test bed can be controlled as exactly as possible.

One issue where exact timing is required, is when scheduling the transmission of CAN frames from the test bed. The test bed needs to sleep for a while, in between the transmission of two consecutive frames, in order to match the specified time interval between transmissions.

Appendix G contains an experiment of measuring the time offset when implementing sleep periods using a standard Linux sleep function compared to a Linux patched with Real Time Application Interface (RTAI). The conclusion of the test, is that the offset is smaller and more constant using RTAI than using plain Linux. Therefore the test bed is designed to be executed on a Linux patched with RTAI, and use RTAI's functions for implementing sleep mode. RTAI is used in user-space, running soft real-time.

4.5.1 Generating Time stamp

As previously described, the test bed needs to attach time stamps to incoming CAN frames and transmitted test frames. The CAN cards contain a register that provides a 32 bit time stamp with a resolution of 1 μs. The timer generating this time stamp is restarted every time the CAN controller is started or reset. However, because the time stamp is needed to uniquely determine when a frame has been sent or received, this time stamp is not enough.

Instead, Linux and GNU C libraries can provide a time stamp in terms of a timeval structure containing the elapsed time in seconds and the rest of the elapsed time in microseconds since midnight of January 1, 1970. By combining these to numbers in a 64 bit value, a unique time stamp is obtained.

Test Bed Engine Design

CHAPTER 5

This chapter contains the design of the test bed engine module, derived from the conceptual design of the previous chapter. The usage scenarios are described, and the software structure is designed. From the description of the interfaces, the routines needed to handle CAN cards and drivers are designed, along with the routines needed to communicate with the database.

5.1 Usage Scenarios

The purpose of the test bed engine is to provide subsystem drivers for simulating subsystems, to provide CAN bus access to the CAN monitor module, and to execute the planned tests on request from the web interface module. As illustrated in figure 4.2 on page 52, the CAN monitor and the test bed engine share a common CAN interface through the TBE driver.

The description of the overall program flow is divided into two scenarios, depending on what the test bed is used for. These two scenarios are:

- Running planned tests.
- Running CAN monitor.

The planned tests consist of single frame- and interval tests configured through the WIU.

Both scenarios can be used simultaneously, if for instance a tester wants to monitor the CAN bus traffic while a planned test is running. In order to use any of the scenarios, the TBE must be running. The TBE uses CAN card one for communication needed by the CAN monitor and the TBE itself. The other cards are used for subsystems. When the test bed is started, it needs to initialise the CAN cards and be ready to perform the following tasks:

- Receive frames on the CAN bus, and send them to the CAN monitor through the IPC interface.
- Receive frames from the CAN monitor through IPC, and transmit them on the CAN bus.
- Receive requests to run tests from the web interface, execute the test, and place the test data in the database.

The test bed engine must always be ready to perform these task types, and no task of one type may prevent another task type of being executed. This implies that the processes for handling these three tasks must be running in three parallel threads. Furthermore, a process is needed to handle requests from the user, while the TBE is running. This is done by pressing chars on the keyboard, hence a fourth thread is needed for scanning the keyboard periodically. This is illustrated in the flow chart of the start up process shown in figure 5.1.

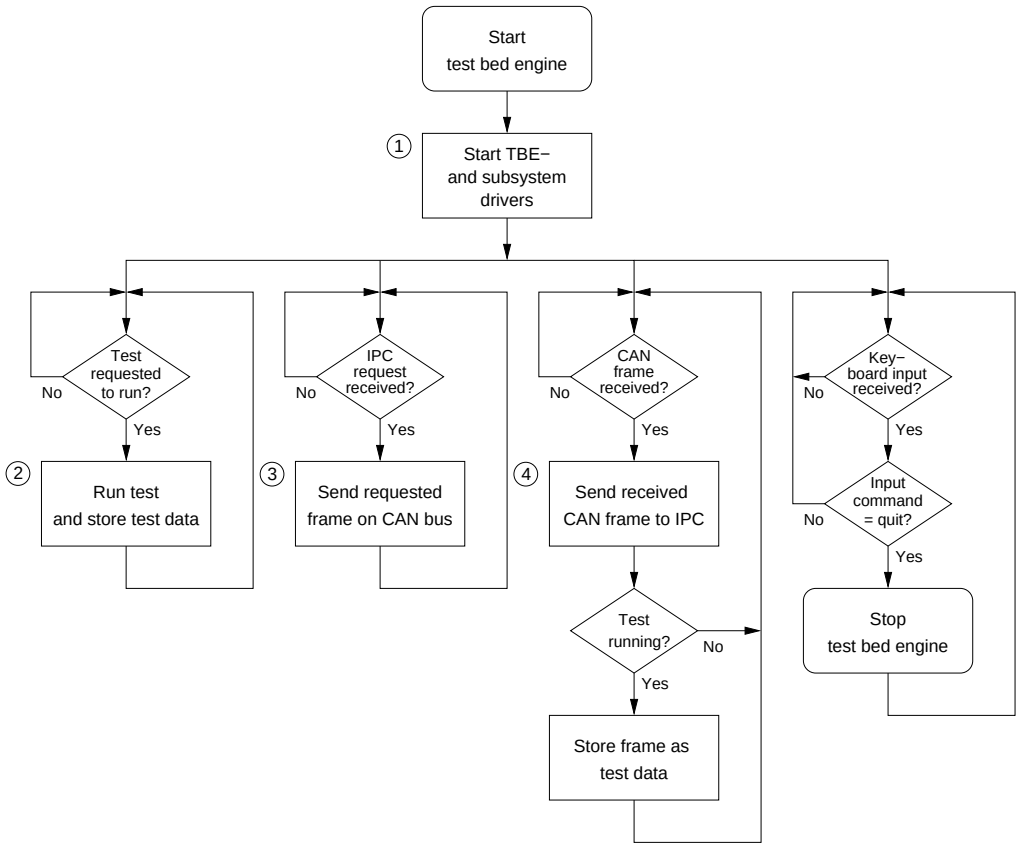


Figure 5.1: *The start up process executed when the TBE is started. The boxes marked with encircled numbers are explained in these sections:* ① *section 5.1.1,* ③ *: section 5.3.3, and*

After starting the CAN cards, the initialisation continues in the four parallel processes. The leftmost process checks on the file that is used to exchange test requests with the web interface, as described in section 4.3.2. The next process handles incoming requests from the CAN monitor. These requests are received through the IPC interface as described in section 4.3.2. The third process in figure 5.1 handles incoming CAN frames. This thread is invoked by interrupt when frames are received on card one. Details on how interrupts are handled are given in section 5.3.2. The fourth process scans the keyboard, and shuts down the test bed engine when a quit command is received. The boxes marked with encircled numbers in figure 5.1, are explained in more detail in this chapter.

The general program flow of the TBE, when used in the two scenarios, running planned tests and running CAN monitor, is explained in the following sections.

5.1.1 Running Planned Tests

When running a planned test, the TBE needs to transmit the specified frames with specified interval, and with the specified subsystems active. It is important that only the specified subsystems participate in the test, because otherwise the test result can not be trusted. This can be done by stopping the CAN cards that are not needed entirely.

However, the initialisation process of each card takes more time than e.g. checking a flag. Having the TBE shut down cards during planned tests, and reinitialising them when the test is done, may also cause instability if for instance the CMU tries to send frames to a card, that has not been initialised completely yet. Instead a flag is used to indicate whether a subsystem should respond to an incoming frame or not. The routine for decoding incoming frames needs to check this flag, before signalling a subsystem. If the subsystem is not needed during test, then the subsystem is not notified, and the frame is discarded. This process is illustrated in figure 5.6, and explained as a part of the interrupt handling in section 5.3.2.

The overall program flow for running tests is illustrated by the flow chart in figure 5.2.

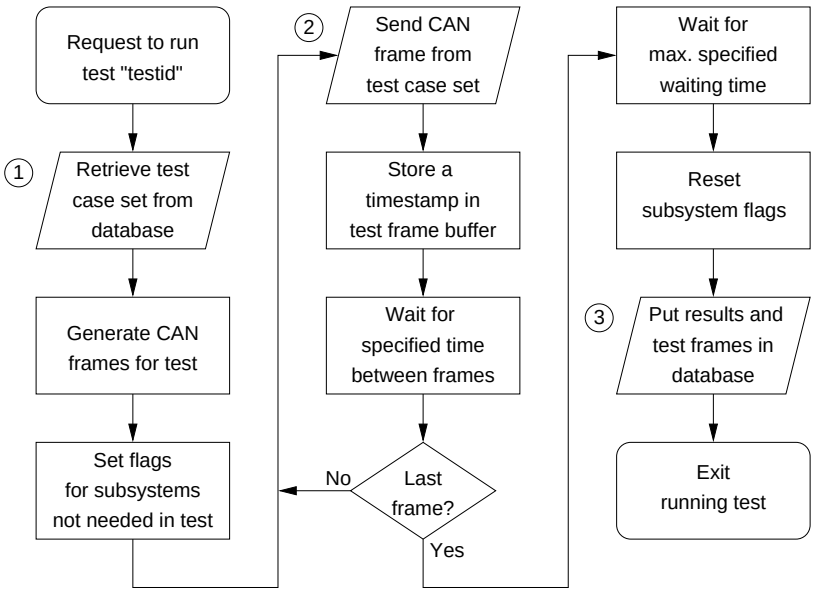


Figure 5.2: *Flowchart of the test bed engine software when running a test. The boxes marked with encircled numbers are explained in these sections: ① : section 5.3. tion 5.5.1, ②*

The testing is started when the box labelled ②
the web interface places a test id in the file used as interface between the web interface and the test bed engine. The TBE uses this test id to retrieve information about the requested test from the database. The contents of the information retrieved, is explained in section 4.4.3.

The necessary CAN frames are then generated and stored in a buffer. This is done prior to starting the actual test, to prevent database communication from slowing down the execution of the test, thereby making it impossible to transmit CAN frames with the specified interval.

Instead, all the processing of the frame is done prior to starting the test, so that it is only necessary to pop a frame of a buffer between the transmission of two consecutive CAN frames,

thereby maximising the frequency with which frames can be sent. After the frame generation, the subsystems flags not needed during the test, are set.

The test bed engine then sends frames with the specified interval, until the last frame is sent. After being transmitted, the frame is stored in a buffer along with a time stamp generated just before the frame is transmitted. This way, the frames sent by the test bed are popped off a buffer, transmitted, and stored in another buffer along with a time stamp.

This time stamp is needed, when the test bed needs to determine, whether an expected output from a test, has been received within the specified time of the transmission of the input. After transmitting the last frame, a pause is made for a period corresponding to the maximum specified time between a test frame and the expected reply. This is done to ensure, that the test is completed, before resetting the subsystem flags that were set before the test, and storing the test data in the database. The format of the test data is explained in section 4.4.4. Once this data is stored, the test bed engine is ready to run a new test.

5.1.2 Running CAN Monitor

As mentioned in the conceptual design of the test bed, the communication between the TBE and the CMU is handled by using IPC.

The part of the IPC exchange done by the TBE, follows the scheme illustrated by the two middle processes in figure 5.1.

When a frame is received on card one, the CAN frame is placed in an IPC buffer, and a message is sent to the CAN monitor. When a transmit request is received through IPC, a CAN frame corresponding to the request is generated and transmitted on the CAN bus.

As previously described, a separate thread is needed for receiving IPC requests, because the test bed engine needs to respond to IPC requests at all times. The transmission of IPC messages is only needed when a CAN frame is received, hence this can be handled by the routine handling incoming frames. Appendix F contains a description of the functions used for IPC under Linux.

5.2 Software Structure

Based on the conceptual design, and the description of usage scenarios, the software structure is designed. The organisation of this structure is explained in the following.

The software structure of the TBE module is layered and modularised. Figure 5.3 shows how these layers and modules are connected to other parts of the test bed engine.

The four PCI CAN cards each carrying two CAN ports are shown at the bottom of this figure. All four cards are controlled by a common CAN device driver, provided by the card vendor, Softing Computing, but corrected in this thesis, as described in appendix C.

On top of the device driver, a Softing CAN Library provide a number of functions needed to e.g. send data to the CAN cards, or retrieve an incoming CAN frame. However, this library does not support multiple instances of the same card in one computer. Therefore the library is modified, so it can be used in the test bed with four identical cards simultaneously. An explanation of

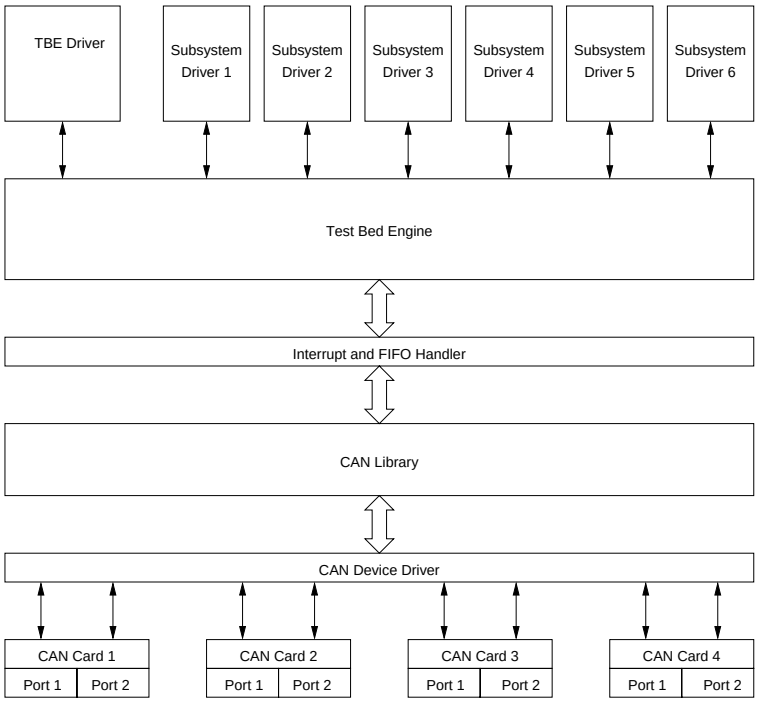


Figure 5.3: *The software structure of the test bed engine.*

how this is done is given in appendix C. Applying this modified library makes it possible to distinguish between the PCI cards.

The Interrupt and FIFO Handler layer is also common for the four CAN cards. Through the Linux kernel, each PCI card is provided with an interrupt level that is associated with the interrupt service routine in the device driver. When initialising a CAN card, this interrupt service routine is associated with a thread in the TBE, that handles incoming frames from that particular card. This means that when a CAN frame is received on one of the PCI cards, an interrupt is generated. The interrupt service routine signals the interrupt thread of that particular card in the test bed engine. Details on how the interrupt handling is done is given in section 5.3.2.

The Test Bed Engine layer of figure 5.3 is the central part of the test bed. It handles the overall software flow as illustrated in figure 5.2. This layer is responsible for starting and stopping tests, initialising drivers and forwarding data between the CMU, the TBE, and the WIU.

The top most layer of the TBE module, is the drivers that are used to interface simulated subsystems, and the other parts of the test bed software structure, which is illustrated in figure 4.2 on page 52. The reception of frames is handled by a separate thread for each CAN port. This is necessary because the processing that may be implemented in a subsystem driver must not delay the frame handling on other ports.

5.3 Managing the CAN Controllers

Before running planned tests, or using the CAN monitor, all four CAN cards must be up and running. When frames need to be sent or received through the card, a number of routines for

doing this are necessary.

The following sections describe how the CAN controllers are managed in different situations, such as initialisation, during interrupt handling, and when sending data.

5.3.1 Initialising CAN Cards

Before using a CAN interface, the card need to be initialised. This is done when the TBE is started. The initialisation routine is shown in figure 5.4.

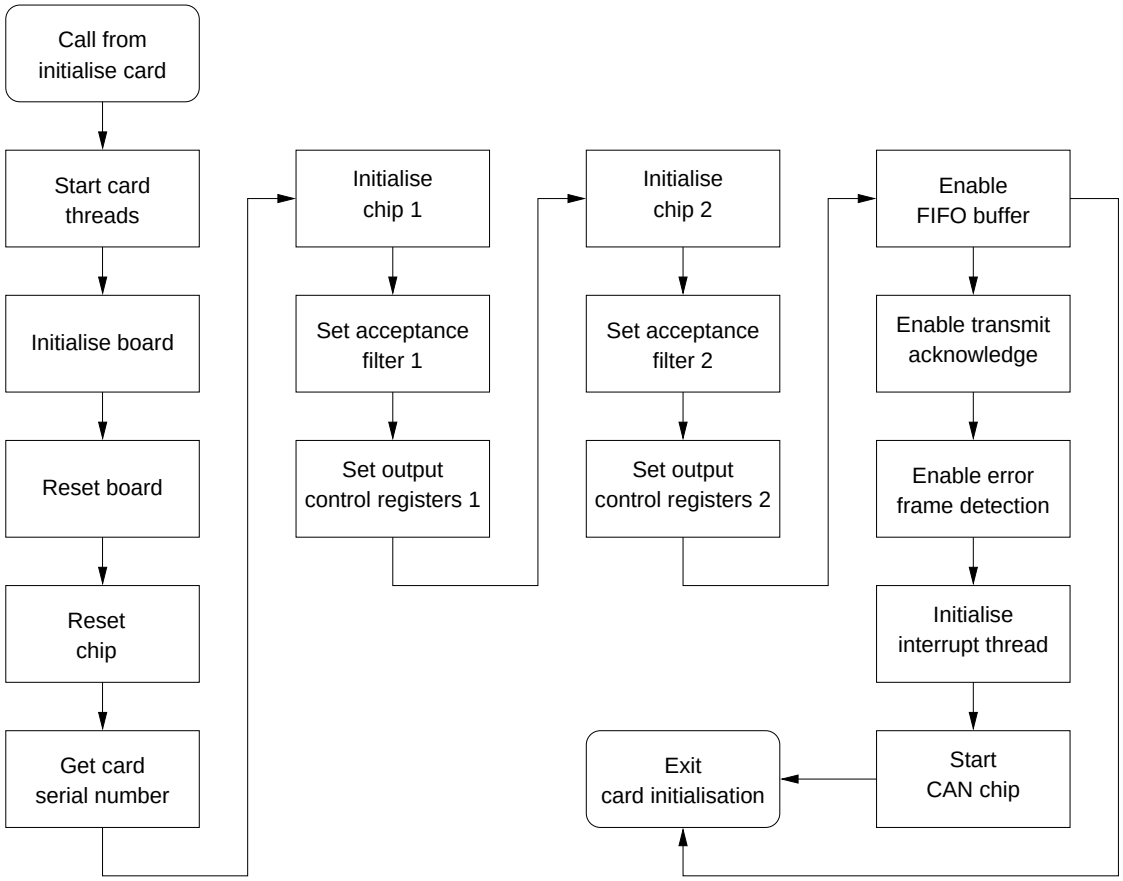


Figure 5.4: *Flowchart of the initialisation of a CAN card.*

The initialisation routine is called with the card number to be started as argument, and iterated once for each card.

First the threads for handling incoming frames on each port are started. When these threads are running, the initialisation routine continues the actual card initialisation.

The initialisation method follows the suggested procedure given by the user manual found in [Softing, 2004]. The cards are setup in FIFO mode with interrupts. This means that frames are stored in a FIFO buffer, and the application is notified about incoming frames, through interrupts.

The initialisation routine activates the ports, and configures the card for acknowledging transmissions and notifying the application of error frames. The last things done during initialisation

is to start the threads needed to handle incoming interrupts and put the CAN chip in operational mode. A complete description of the initialisation routine, and the operation of the card is given in appendix C.

5.3.2 Interrupt Handling

As described in section 5.2, an interrupt is signalled to an interrupt handler thread in the test bed engine when a frame is received on a CAN interface. There is one thread for each CAN card, and these threads reside on the Interrupt and FIFO Handler layer of figure 5.3.

This interrupt thread then uses a FIFO handler to retrieve the incoming frame from the FIFO buffer in the CAN card, places the frame in FIFO buffers in the test bed engine, and eventually signals either a subsystem driver or the TBE driver, depending on which CAN card and port that received the frame. Only ports that are needed while a planned test is running are signalled. Otherwise the frame is discarded.

Figure 5.5 shows the process of handling an interrupt on port one of card one. The encircled numbers corresponds to the events described below.

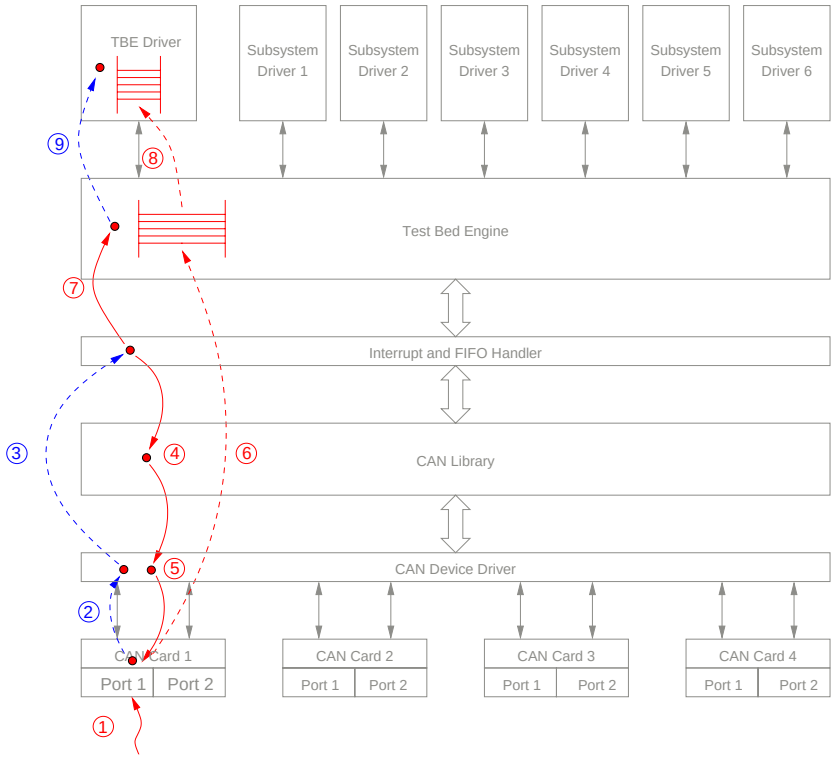


Figure 5.5: *The process of handling an interrupt. The blue dashed arrows represent semaphores or signals, the solid red arrows represents function calls, and the dashed red arrows represent data transfer. The encircled numbers corresponds to the steps explained in the text.*

① An interrupt is generated on p

② The interrupt is signalled to the device

③ The interrupt thread receives the interr

④ The interrupt and FIFO handler uses a card.

⑤ This library function communicates wit

⑥ The received frame is stored in a queue

⑦ The interrupt and FIFO handler calls a the frame and determine which card and port the frame is received by.

⑧ The frame is transferred to a FIFO buff the frame.

⑨ The port thread handling incoming fra from the FIFO.

Figure 5.6 shows a simplified flow chart of interrupt handling from the interrupt and FIFO handler layer and upwards. The simplification consists of omitting the process of storing frames in FIFO buffers, which is not included in the flow chart to limit the size of the chart.

When an interrupt signal is received, the parameters of the CAN frame causing the interrupt are read in order to determine the port number. Based on these numbers, the port threads are signalled if the port flags indicate that the port is allowed to participate. The port threads corresponds to the threads in the TBE- or subsystem drivers responsible of receiving frames.

5.3.3 Sending Frames

The routine for sending frames on the CAN bus is common for both subsystem drivers and the TBE driver. The routine is given the port number, card number, and the CAN frame to be sent as arguments. The flowchart of the routine is shown in figure 5.7.

The CAN library contains separate functions for sending data and remote frames, on port one or two respectively. Therefore, the routine for sending frames need to decode whether port one or two is to be used, and the type of frame to be sent. If the RTR bit is set, the frame is a Remote Transmission Request frame. Else the frame is a data frame.

The routine sends a default frame length of eight bytes. The INSANE meta-protocol used by the AAUSAT-II does not define whether fixed or variable frame length should be used. The advantage of using fixed frame length is that timing calculations are simplified, because all frames can be estimated to have the same length and duration. Although this is not entirely the case, when bit stuffing occurs, the test bed is designed to send fixed frame lengths. Furthermore, with the data field overhead applied by the INSANE concept, it seems that most of the data bytes are needed anyway.

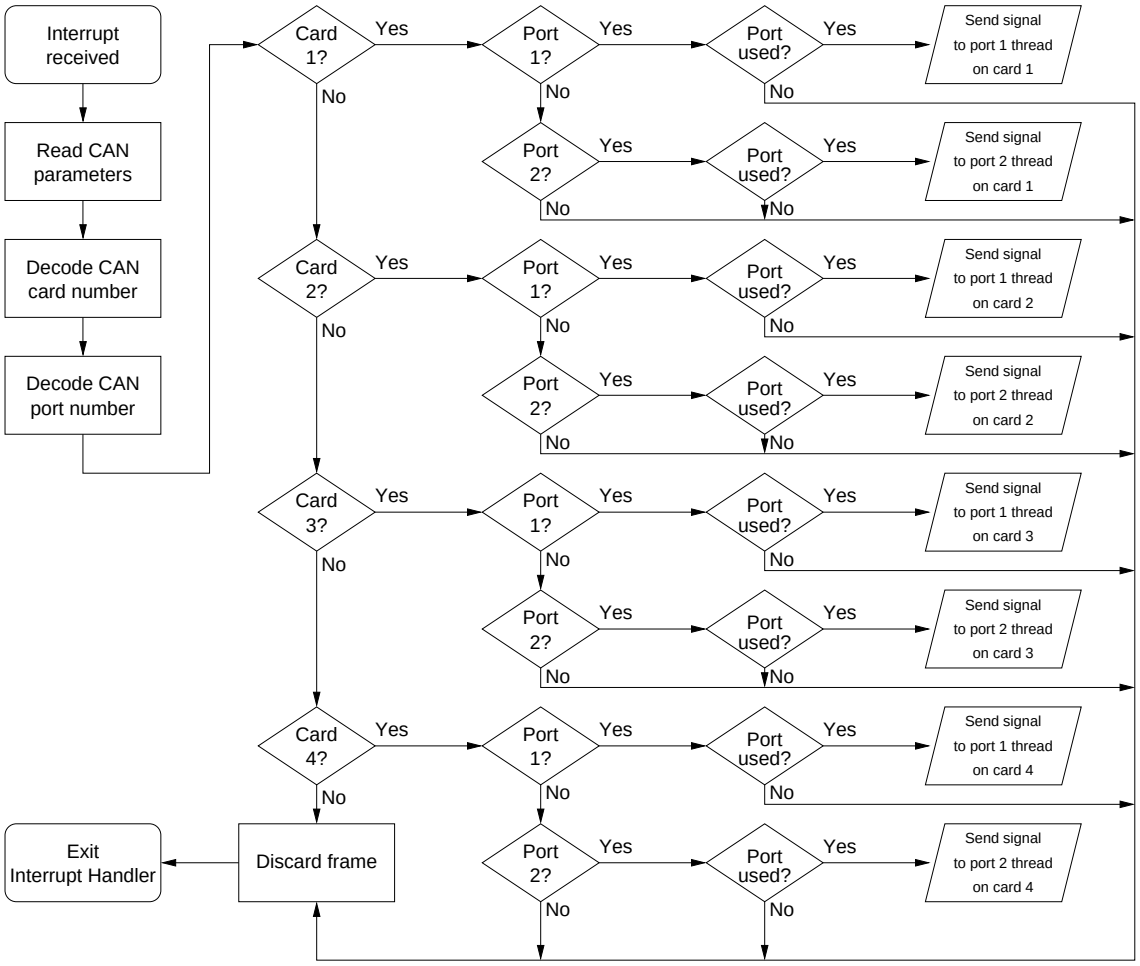


Figure 5.6: *Flowchart of the interrupt handling process in the test bed engine.*

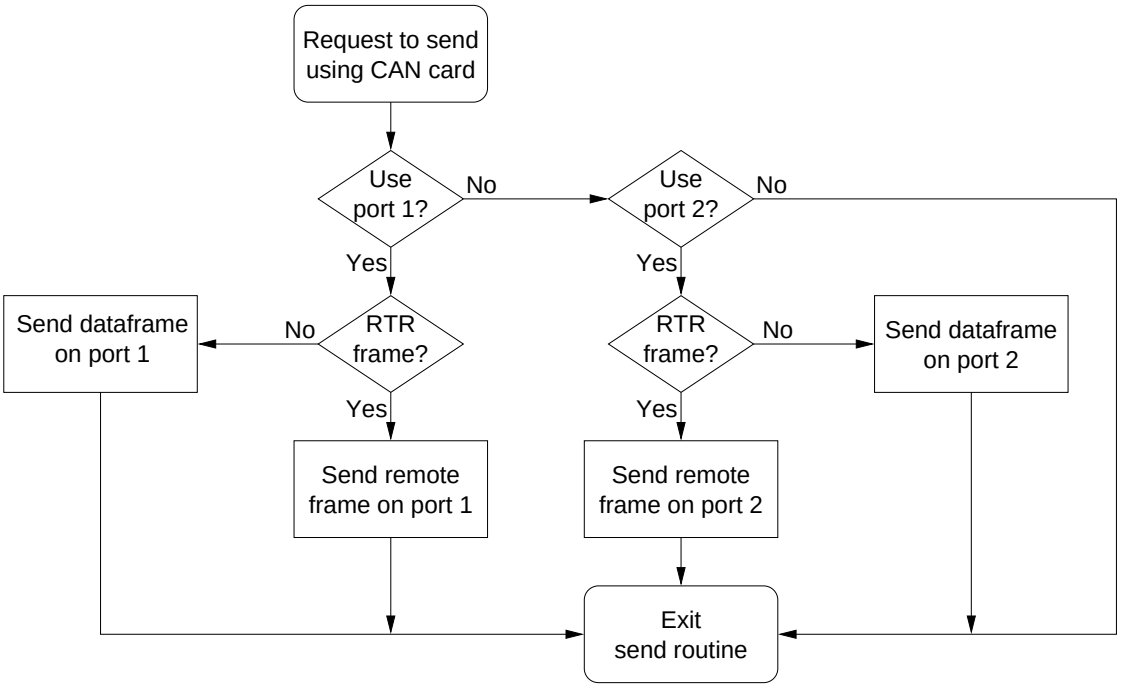


Figure 5.7: *Flowchart of the routine for sending CAN frames.*

5.4 Drivers

The test bed engine includes software drivers for simulating subsystems and the TBE driver for running the tests and the CAN monitor. These drivers are described shortly in the following.

5.4.1 Subsystem Drivers

The routine for initialising CAN cards, shown in figure 5.4, includes a box for starting card threads for handling incoming frames. These threads are a parts of the subsystem drivers.

The functionality provided by the subsystem drivers is a thread which is signalled on incoming frames, a routine executed when initialising the driver, and the common routine for sending frames, as described in section 5.3.3.

When a subsystem thread is requested to start, the initialising routine sets the acceptance filter settings for the corresponding CAN port and creates the receive thread. The receive thread is signalled on incoming frames, by the interrupt handling routine of figure 5.6.

5.4.2 TBE Driver

The structure of the TBE Driver, is identical to the subsystem drivers, and the initialisation is identical as well. However, the functionality to be executed when frames are received in the subsystem drivers, is provided by the subsystem developers, the TBE driver needs to execute the code for running planned tests and CAN monitor.

The code for running the CMU is always executed, but the code used for running planned tests, is only executed when a test is executed.

5.5 Database Communication

The TBE communicates with the database when exchanging test definitions and test data of the structures specified in section 4.4.3 and 4.4.4. The routines for carrying out these tasks are designed in this section.

5.5.1 Retrieving Test Definitions

The test bed engine needs to retrieve test definitions from the database. This is done by a routine which is called from the box labelled ①
the interface file between the WIU and the TBE is present, indicating that a test has been requested to run. The test id is written in the file, and based on this, the test bed engine connects to the database and retrieves the definition of the test. The contents of the test definition is given in section 4.4.3, and the process of retrieving it is illustrated in figure 5.8.

Based on the test definitions, the necessary CAN frames are generated and stored in a buffer. After transmission they are moved to another buffer along with a time stamp taken at the time of transmission. This way, each transmitted test frame has a time stamp showing when the frame is transmitted.

When the routine in figure 5.8 finishes, the routine in figure 5.2 on page 61 continues to execute until the box labelled ③ is r
the following.

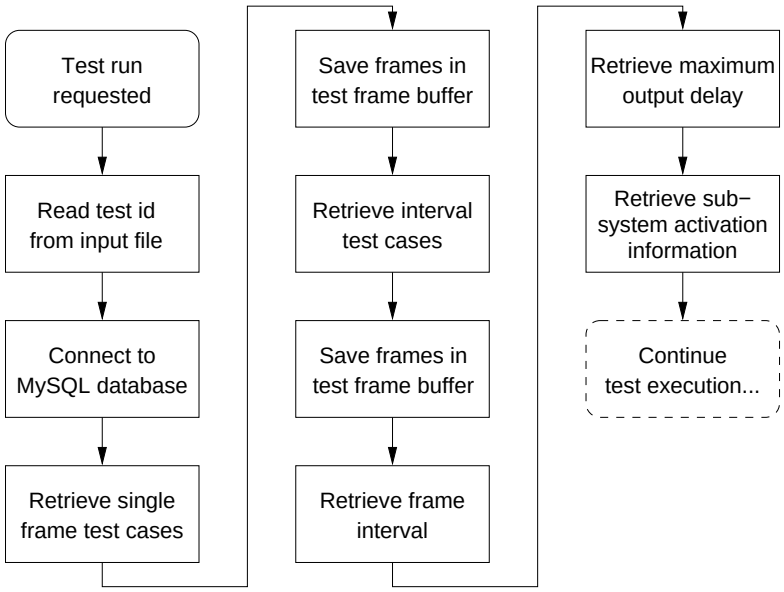


Figure 5.8: *Flowchart of the routine for retrieving test definitions from the MySQL database.*

5.5.2 Uploading Test Data

When the test is finished, a routine starts to upload results to the database. The transmission time stamp of each test frame is the first thing to be uploaded to the database. This time stamp is used for determining if an output of a test frame is received within the specified period of time. The routine for uploading test data is shown in figure 5.9.

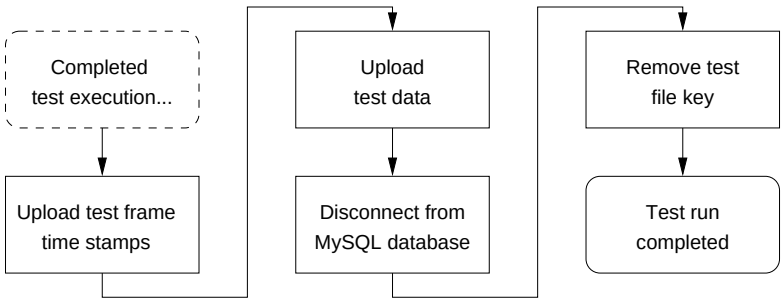


Figure 5.9: *Flowchart of the routine for storing test data in the MySQL database.*

After uploading the transmission time stamps, all the CAN traffic that appeared during the test is uploaded in the format specified in section 4.4.4. When this process is finished, the database connection is closed, and the interface file is deleted to indicate that the test bed engine is ready to execute a new test.

5.6 Error Handling

Error handling is an important issue in the TBE, because this module is the lowest software level of the test bed. Often the focus of the user is on either the WIU or the CMU, because these modules contain the graphical interfaces, targeted at providing intuitive access to the test bed. This means that notifying the user of errors in the TBE should be done through a log file, which can be inspected by the user.

For this purpose, an error handling functions is designed. This function takes a description of the error, the name of the function in which the error occurred, and a severity parameter as argument. The severity is used to determine whether the TBE should shut down or continue operation.

Another issue is how to handle error frames on the CAN bus. When a sufficient number of error frames occur on the CAN bus, the CAN controllers change their operational state, according to the description found in section B.2.3 on page 146. This means that a CAN adapter will change to "bus off" state, when the number of errors is high enough. The analysis done in section 3.3.1 states, that such errors should not be fixed automatically.

If the test bed automatically restarted a CAN controller in "bus off" mode, the user might assume that everything is fine. This can not be allowed, because the test bed must not be responsible for wrong test conclusions under any circumstances. Therefore it is decided that CAN adapters are not automatically reset, once they have entered "bus off" mode. Instead the user needs to manually restart the TBE.

Web Interface Unit Design

CHAPTER 6

In figure 4.2 the web interface unit and the database is shown. This chapter contains the design of the web interface unit, and in the end the database design is given.

As mentioned in the analysis, the user interface to configure tests and test cases is built in a web environment. The web page is built in PHP/HTML, and the test and test case data is stored in a MySQL database. PHP is a widely-used general-purpose server side scripting language, that is intended for web page development. PHP also has a very good defined interface to MySQL.

The WIU has a high amount of interaction. Many input values are entered, and many test results are analysed from the WIU. For systems with a lot of user interaction, the main key word is usability. If the test bed is difficult to use, it will not be used. Usability must be in mind of the developer from the start of the design phase, therefore the usability features are described in this chapter.

The WIU deals with four concepts, namely *CAN identifiers*, *test case sets*, *tests*, and *test reports*. The CAN identifiers are simply the CAN identifiers to be sent, and they must be typed in before they can be used in a test. A test case set is a set of test cases, which can be split up into two groups, namely single frame test cases and interval test cases. These groups are explained below. A test contains one or more test case sets, which makes it possible to combine a number of different test case sets in one single test. A test report is a report containing the test results.

These four concepts are available from the main menu on the test bed, and they are designed in the next four sections.

6.1 CAN Identifiers

When the "CAN Identifiers" link is chosen from the main menu, an overview of all the CAN identifiers appear. A CAN identifier has four attributes: The CAN identifier itself given in decimal format, a name, the authors AAU user name, and the date and time of creation. These are all displayed in the overview of the CAN identifiers.

The flowchart of the CAN identifier part is shown in figure 6.1.

From the CAN identifier overview there are four options: Order the CAN identifiers by one of the attributes, insert a new CAN identifier, delete a CAN identifier, and view/edit a CAN identifier. If the delete option is chosen, a confirm box appears to ensure that an identifier is not deleted by accident. If insert a new CAN identifier or view/edit is chosen, three attributes

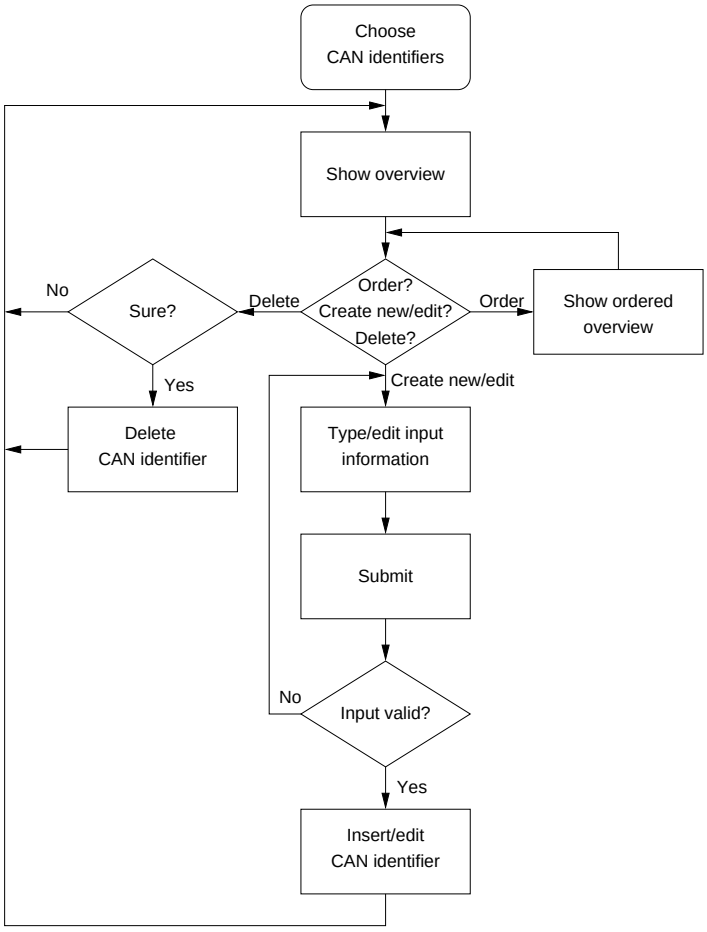


Figure 6.1: *Flowchart for the CAN identifiers.*

are possible to key in or change, namely the name, the CAN identifier and the author. The creation date and time is given automatically by MySQL.

The edit possibility is only meant to be used if an error is found in an identifier. It is undesirable to edit the identifiers from time to time, because another person may expect the CAN identifier to be something else. Therefore it is recommended only to edit CAN identifiers made by one self.

6.1.1 Input Validation

It is important to check whether or not the user inputs are valid. The following list of validations are made on the CAN identifier user input:

- No input elements must be empty.
- There is a limit of 30 chars for the identifier name.
- The CAN identifier field must only contain [0-9] digits.
- The CAN identifier must not exceed 536,870,912 (29 bits).

If this is not fulfilled, the user is not able to submit the form.

6.2 Test Case Sets

When " Test Case Set" is chosen from the main menu, an overview of all the test case sets appear. A lot of attributes are connected to a test case set, so only a few are chosen to be visible in the overview. These are name, author, type, creation date and time, and last modified date and time. Except the type attribute, these are all straight forward. Type denotes whether the test case set is a single frame or an interval set.

The flowchart of the test case set part is shown in figure 6.2.

From the overview of the test case sets, the user has the same possibilities as with the CAN identifiers, namely order by an attribute, insert, delete and view/edit a test case set. If insert a new test case set is chosen, the user must enter a name for the test case set, a short description, and the users AAU user name. After that the user must decide whether the test case set is a single frame test case set or an interval test case set.

6.2.1 Single Frame Test

The single frame test is a test where only one specified frame is sent. The frame is specified with a CAN identifier from the CAN identifier list, the first byte (B0) of the data field, and the last seven bytes of the data field (B1-B7).

The user specifies the number of expected output CAN bus frames for this test case, and for each expected output a number of attributes must be specified. These are the CAN identifier, B0, an interval of B1-B7, and the maximum waiting time, given in milliseconds. The data fields

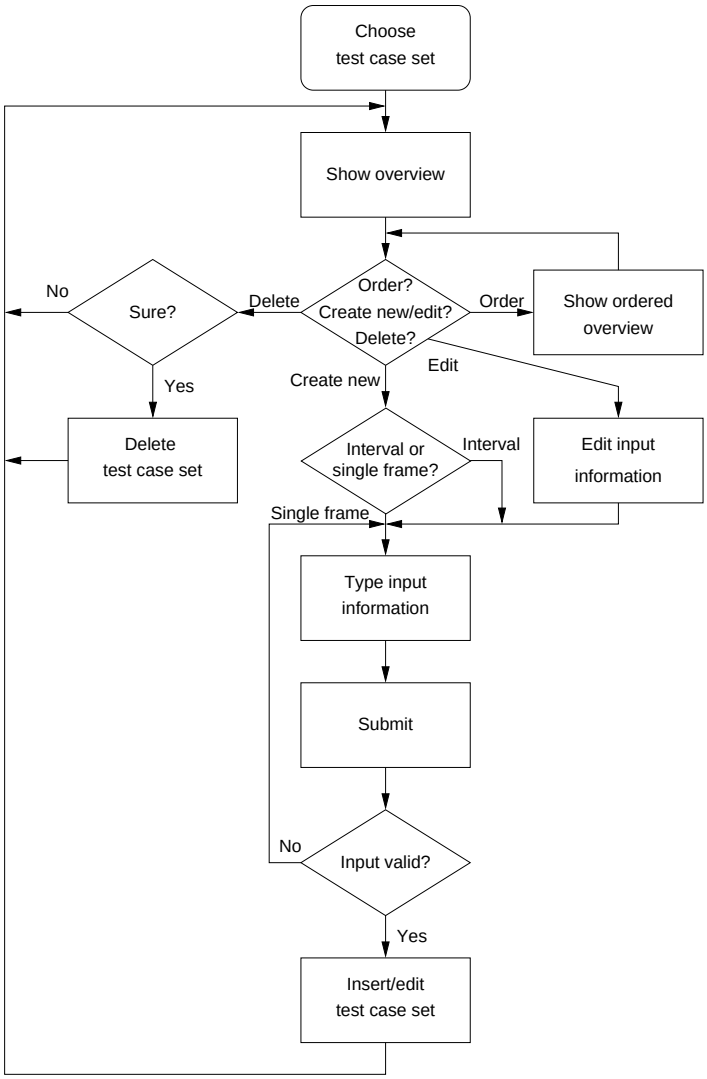


Figure 6.2: *Flowchart for the test case set.*

are given in hexadecimal format, and the maximum waiting time is given in decimal format. The maximum number of expected outputs must be set to 100, which seems reasonable.

If there is more than one expected output, some of the expected outputs could be identical. For instance the same CAN identifier is expected for all outputs. Therefore it is useful to have a feature to copy values from a copy field to all fields in the expected output area. CAN identifier, B0, B1-B7, and maximum time values can be copied individually, or all together.

6.2.2 Interval Test

The interval test is performed by a combination of boundary value testing and equivalence class testing, as described in the analysis in section 3.3 on page 35. Figure 3.2 on page 39, is repeated in figure 6.3 for convenience, and it shows a robust worst case boundary value test on two intervals in a CAN bus data frame (B1-B7).

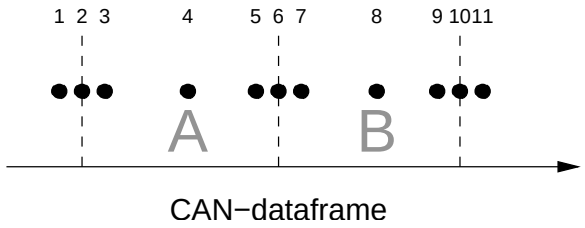


Figure 6.3: *Test cases for an interval of the data in the CAN frame.*

When the user enters the number of desired intervals, the input and expected output frames must be specified. The input frames all have the same CAN identifier and B0. For B1-B7 the border values must be entered. The number of borders is the number of intervals + 1.

A reasonable maximum number of intervals must be set. It is not expected that the users create tests on more than 20 borders, which gives a number of intervals of 19.

To specify the expected outputs is somewhat more complicated. The example in figure 6.3 has 11 test cases. The number of test cases is $4 \cdot \text{number of intervals} + 3$.

As within single frame testing, the CAN identifier, B0, B1-B7 interval and maximum time is specified. The first and last test cases are special cases, because the user might not expect an output for these, because they are out of range. But it is important to send these frames, because it must be verified that the subsystem can handle invalid data.

Whether or not the output is expected for the first and/or the last test case, must be denoted by radio-buttons. No other output information is needed for these, and the fields to enter the expected outputs disappear.

As mentioned, the expected outputs for the test cases are depicted in intervals for B1-B7. It gives the ability to group some of the expected outputs of the test cases, because they can be in the same interval. When expected output test cases are grouped, the user do not need to fill in as many fields, and it might save time. In figure 6.3 test case 3, 4, and 5 can be grouped, as

well as test case 7, 8, and 9. The number of user defined expected output intervals is at least 5 and maximum 7 for this example.

As within single frame testing, a copy line is needed in case the same output is expected from many test cases.

The test case attributes are described in the database design, in section 6.5.

6.2.3 Input Validation

The following validations are made on the test case set user input:

- No input elements must be empty.
- There is a limit of 30 chars for the test case set name.
- The maximum value for B0 (both input and expected output) is 0xFF (one byte).
- Only [0-9,a-F] digits are allowed in the border, B0, and B1-B7.
- Only [0-9] digits are allowed in the maximum time fields.
- The maximum value for byte B1-B7 is 0xFFFFFFFFFFFFF (7 bytes).
- The maximum waiting time for the frames is 20,000 ms (20 s).

If this is not fulfilled, the user is not able to submit the form.

6.3 Tests

When " Test" is chosen from the main menu, an overview of all the tests appears. A lot of attributes are connected to tests, so only a few is chosen to be visible in the overview. These are name, author, creation date and time, last modified date and time, and the number of test case sets connected to the test.

The flowchart of the test part is shown in figure 6.4.

From the overview of the tests, the user has the same possibilities as in the previous, namely order by an attribute, insert, delete and view/edit a test. If insert a new test is chosen, the user must enter the following:

- A name for the test.
- A short description.
- The users AAU user name.
- The frame interval time in milliseconds.
- The frame order (sequential or random).

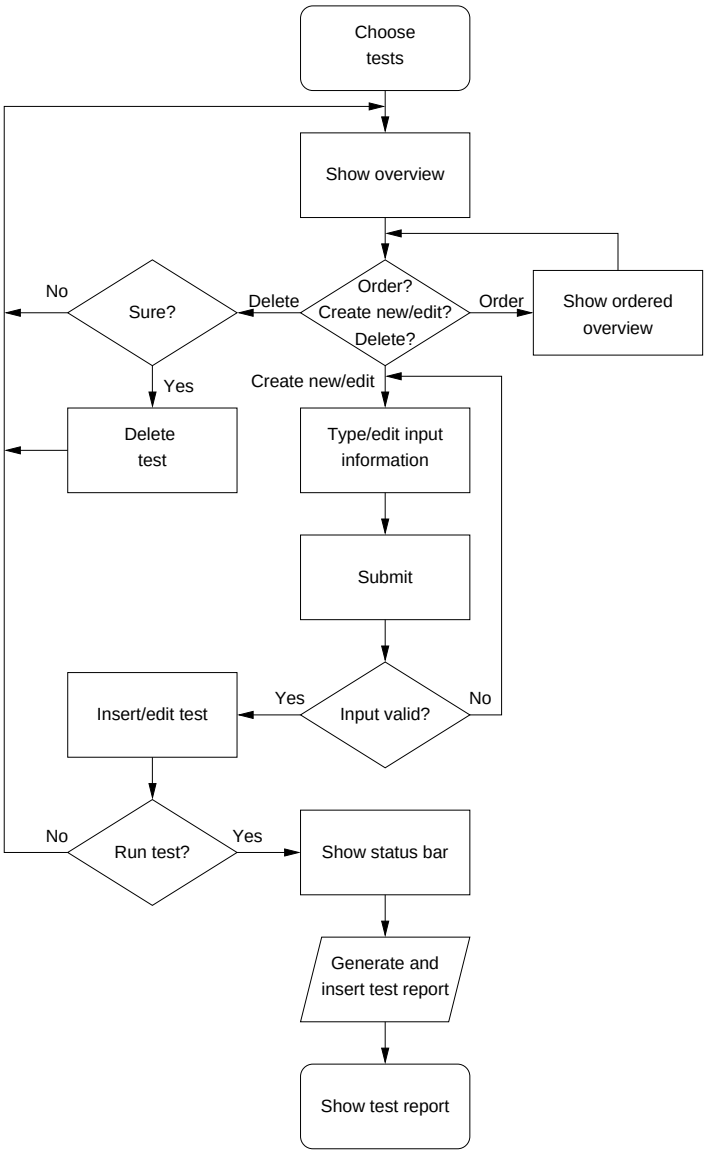


Figure 6.4: *Flowchart for the test.*

- The subsystem drivers to be activated during the test.
- The test case sets to belong to the test.

A list of all the test case sets are shown, and the user can add as many as desired, but minimum one. The test case sets are shown by their name, and it is possible to view and edit the test case set by clicking on it. When the test case set is modified, the test appear again.

When the test is defined, the user can choose to save it, or to save it and run the test. Only if another test is not running, the user is able to start the test. It must not be possible to start two tests simultaneously, because it is confusing to have traffic from two tests at the same time on the CAN bus.

During the test, a status bar is shown, and the test finish date and time is shown. Equation 6.1 shows the accurate testing time.

$$\text{frame interval time} \cdot \text{number of frames} + \text{the maximum waiting time} \tag{6.1}$$

The maximum waiting time is the largest waiting time given by the user when the expected outputs are defined for the test cases.

When a test finishes, the test report is generated and displayed for the user.

6.3.1 Input Validation

The following validations are made on the test user input:

- No input elements must be empty.
- There is a limit of 30 chars for the test name.
- The frame interval time must be in the interval of 1 ms and 10,000 ms.
- The frame interval time must only contain [0-9] digits.

If this is not fulfilled, the user is not able to submit the form.

6.4 Test Reports

When "Test Reports" is chosen from the main menu, an overview of all the test reports appear. A lot of attributes are connected to test reports, so only a few are chosen to be visible in the overview. These are test name, test author, run time, number of frames and the result.

The flowchart of the test report part is shown in figure 6.5.

From the test report overview, the user has three options, namely order by an attribute, view a test report and delete a test report. They are all straight forward, though it is important to note that the test reports must not change, if something changes in the test for the test report. It is important that the test report shows exactly the settings the test is run with.

When a test finishes, the test report is made. This is done by comparing all the CAN bus data during the test with the user defined expected outputs. The elements to compare are:

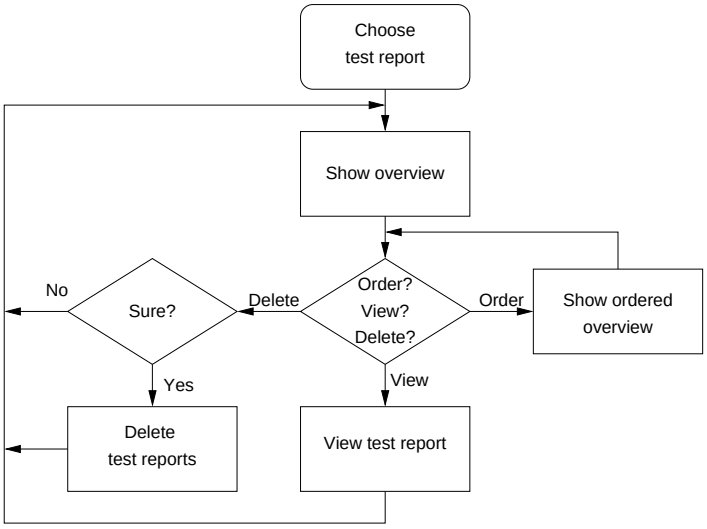


Figure 6.5: *Flowchart for the test report.*

- Is the CAN bus time stamp between the frame sending time and the frame sending time + the defined maximum time?
- Is the frame type correct (data frame, error frame, request frame)?
- Is the specified CAN identifier identical to the actual?
- Is the specified B0 byte identical to the actual?
- Is the last seven bytes of the CAN data frame in between the specified interval?

It is important that one frame on the CAN bus only can be used to validate one test case. This can cause a problem with the don't care frames, because they can reserve a CAN data frame to validate a don't care frame, but the CAN data frame maybe should have been used to validate another test case. Therefore the comparing of don't care frames must not be compared before all other test cases are compared.

The test passes successfully if and only if all test cases in the test passed. The report must show all the input frames, and it must be very clear if a test case passed or not. If a test case passes, it must be possible to see the CAN data frame that has validated the test case. All traffic on the CAN bus during the test must be shown in the test report as well.

A test report is saved for all tests, and possible to view at a later time. All the test report data must be saved in two comma separated files, one for the specified frames, and one for all the CAN bus traffic. It is possible to download the files from the test report.

6.5 Database Design

All input and output data is saved in a relational MySQL database.

Eight tables are used in the database, to store information about planned tests, test reports, and the CAN bus traffic.

The entity-relationship diagram (ERD) design is shown in figure 6.6. Appendix D contains a description of the fields as well as their data types.

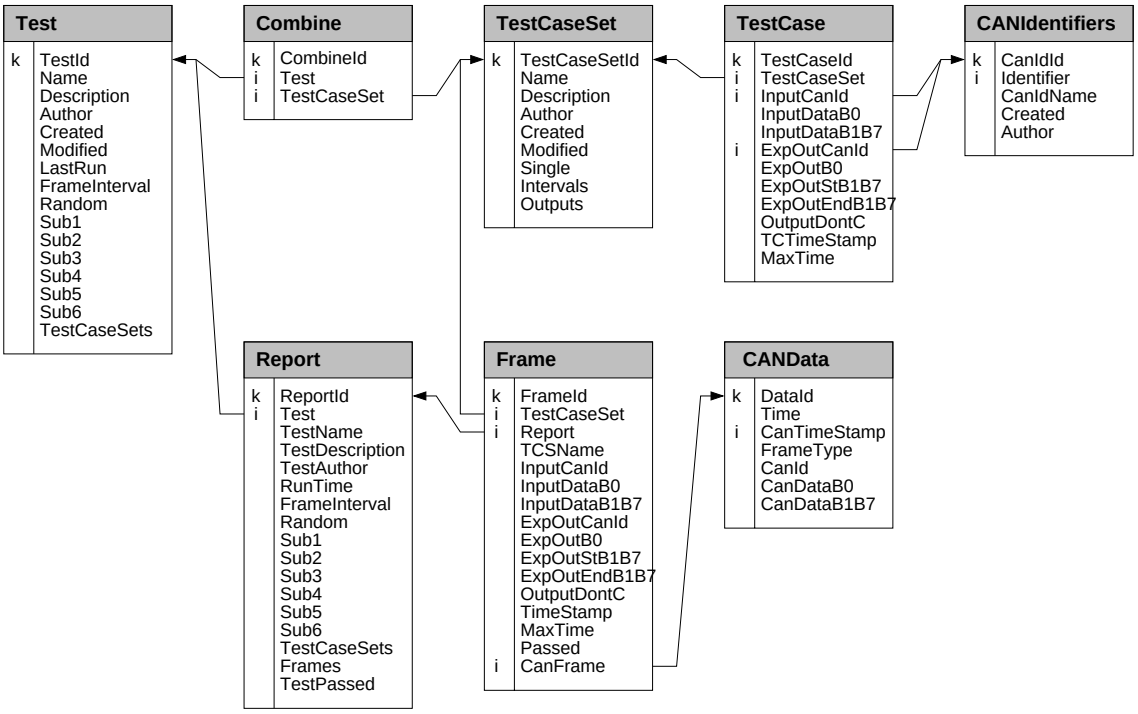


Figure 6.6: *Database design. k denotes a primary key, and an i denotes an index.*

The *Test* table in the leftmost corner contains all the information about the user defined tests. The first field in the table is the TestId, which is an auto-incremented primary key for the Test table. The other fields in the Test table, all contains information typed in by the user. Table 6.1 shows a description of the Test table.

The test case sets are stored in two tables, namely the *TestCaseSet* and the *TestCase* tables. The TestCaseSet table contains the general information of each test case set, such as name, description, author, etc.. It also contains information about the type of test case set, whether it is a single frame or an interval test case set.

The TestCase table contains the frames of the test case sets. Each row in the table contains the frame to be sent on the CAN bus, and the expected output for that frame. If the test case set is a single frame test case set, the number of frames for that test case set is the number of expected output frames. In these frames, the input data and the sending time stamp is equal, but the expected output data differs.

The TestCase.TestCaseSet field connects the TestCase table with the TestCaseSet table, by having the id of the test case set in the TestCaseSet.TestCaseSetId field.

Test			
k	TestId	uns. int	Unique id and primary key for Test table.
	Name	varchar	The name of the test.
	Description	text	A short description of the test.
	Author	varchar	The AAU user name of the creator of the test.
	Created	datetime	Date and time for test creation.
	Modified	datetime	Date and time for last test modification.
	LastRun	datetime	Date and time for last run of the test.
	FrameInterval	int	The interval time of frame sending.
	Random	tinyint	Denotes whether frames are sent random or sequential.
	Sub1	tinyint	Denotes whether or not the subsystem driver 1 is used.
	Sub2	tinyint	Denotes whether or not the subsystem driver 2 is used.
	Sub3	tinyint	Denotes whether or not the subsystem driver 3 is used.
	Sub4	tinyint	Denotes whether or not the subsystem driver 4 is used.
	Sub5	tinyint	Denotes whether or not the subsystem driver 5 is used.
	Sub6	tinyint	Denotes whether or not the subsystem driver 6 is used.
	TestCaseSets	int	The number of test case sets.

Table 6.1: *A field description of the Test table.*

The CAN identifiers are stored in the *CANIdentifiers* table. Relations are made to this table from the TestCase table, where it uses the CAN identifiers to describe the input- and the expected output identifier.

One of the requirements for the WIU, is that many test case sets must be included in a test. It means that the Test and the TestCaseSet tables must have a many-to-many relationship. This is done by adding a table containing the id's of the tests and the test case sets to be combined. This is done by the *Combine* table.

The *Report* and *Frame* tables are used to store the auto generated test reports. The Report table copies the settings for the test from the Test table. It is done because the user must be able to see the exact settings for the test, when it is run. A field contains information about whether the test passed or failed. A test only passes if all test cases in the test passes.

The Frame table contains all the frames of the performed test. Both the input and the expected output is copied from the TestCase table, when the test report is generated. It also contains a field to tell if the test case passed or failed. If the test case passed, the id of the CAN frame to validate the test case set, must be set in the last field of the Frame table, namely the CanFrame field.

The *CANData* table contains all the traffic that appear on the CAN bus during the tests started from the WIU. The data is inserted by the test bed engine when a test finishes. The CAN bus information saved, is the date and time, the 64 bit unique time stamp in μs, the frame type, the CAN identifier, and the CAN bus data fields divided into B0 and B1-B7.

A description of the data type of the fields is given in appendix D.

6.5.1 Report Generation

When a test finishes, the PHP report generator starts generating a test report. When a test report is generated, PHP needs the test cases connected to the test. It connects to the database, and by the SQL-query given below, it gets all the information about the sent frames. The TCTimeStamp field in the TestCase table is updated by the test bed engine when a test is finished.

```
SELECT TestCaseId, HEX(InputDataB0) AS InputDataB0,
       HEX(InputDataB1B7) AS InputDataB1B7, ExpOutCanId,
       HEX(ExpOutB0) AS ExpOutB0, HEX(ExpOutStB1B7) AS ExpOutStB1B7,
       HEX(ExpOutEndB1B7) AS ExpOutEndB1B7, OutputDontC, TCTimeStamp,
       MaxTime, testcaseset.TestCaseSetId, testcaseset.Name AS
       TCSName, canidentifiers.Identifier AS inputcanid
FROM testcase
LEFT JOIN canidentifiers ON testcase.InputCanId = canidentifiers.CanIdId
LEFT JOIN testcaseset ON testcase.TestCaseSet = testcaseset.TestCaseSetId
LEFT JOIN combine ON testcaseset.TestCaseSetId = combine.TestCaseSet
LEFT JOIN test ON combine.Test = test.TestId
WHERE TestId = {TestId}
ORDER BY TCTimeStamp ASC
```

To verify a test case, the expected output frames for the test case must be compared with the CAN bus data during the test. This is uploaded by the TBE, and received in PHP from the CANData table by the following SQL-query:

```
SELECT DataId, Time, CanTimeStamp, FrameType, CanId,
       HEX(CanDataB0) AS CanDataB0, HEX(CanDataB1B7) AS CanDataB1B7
FROM  candata
WHERE CanTimeStamp BETWEEN {time of first frame} AND {time of last frame}
ORDER BY CanTimeStamp ASC
```

Only if the expected frame appears on the CAN bus during the test, the test case passes, and only if all test cases passes, the test passes.

6.5.2 Indexes

When many tests are defined and run, some of the tables contain many rows. Therefore the need of indexes arises in the TestBed database. An index is used by the MySQL database to speed up searching. An index is applied to fields in the database, which are often used in SELECT, INSERT, UPDATE, and DELETE SQL-queries.

Especially in the CANData table, an index is very useful, because this table easily becomes very large, since it contains all the traffic on the CAN bus during all tests, and the data is never deleted.

When data is received from the CANData table, the CanTimeStamp field is used in the SELECT SQL-query, hence an index is applied to that field. The other indexes are shown in figure 6.6.

6.6 Interface to Test Bed Engine

As described in section 4.3 on page 52, the interface between WIU and TBE is decided to be a file, that contains the id of the test to be run. The file is created by PHP, immediately after the test is started. When the test finishes, the TBE deletes the file, and a new test can be run. If the file exists, it is not possible to start a new test from the WIU.

CAN Monitor Unit Design

CHAPTER 7

This chapter covers the Object Oriented Analysis and Design (OOAD) of the CMU. The OOAD deals with the unit that logs CAN frames and makes it possible for the user to send user defined CAN frames on the CAN bus. Only parts of the object oriented methods are applied, because the CMU only covers a minor part of the complete test bed system, as discussed in section 4.1.

7.1 The Object Oriented Method

Methods used for the OOAD are adopted from templates presented in object oriented analysis and design literature [Mathiasen et al., 2000]. Figure 7.1 gives an overview of the document structure, which is separated into a OOA and OOD section in this document.

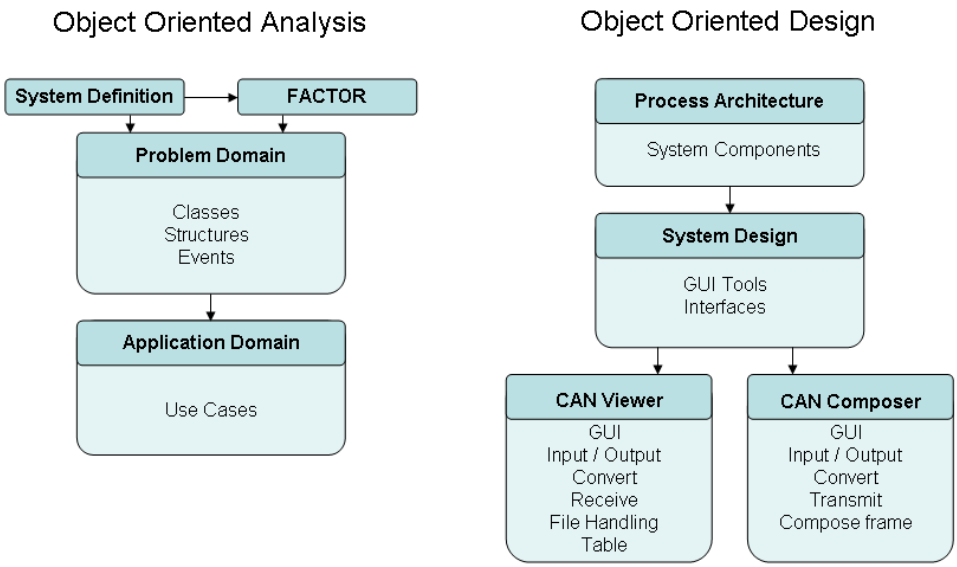


Figure 7.1: *The structure of the object oriented analysis and design.*

In figure 7.1, the OOA starts by defining a system definition which form the basis of the CMU. This basis is used in the analysis, design, and implementation. A short and precise summary of the system definition is given in section 7.2.2, using the method called the FACTOR criterion. This criterion emphasise important topics to be treated during the OOAD.

From the system definition, and the FACTOR criterion, the treated problem domain is realised. The problem domain is used to limit the domain, in which the analysed project is intended to be used. This is carried out by defining classes, structures, and events, after which the treated system is designed.

The OOA finishes by defining the application domain. This defines use cases and functionality for how the system should behave and how the user uses the system.

Figure 7.1 also shows the structure of the OOD. It starts by defining the process architecture. This is where the designed system is running and which components are necessary to run the system.

When the process architecture is in place, the actual system design is started. This implies investigation of which development tools and interfaces are needed to design the components, that are discovered during the problem domain/process architecture.

The object oriented design covers the structure of the designed components, for the developed system. The CMU contains two main classes, namely a CAN viewer and a CAN composer. The subcomponents of these two classes are described.

7.2 Object Oriented Analysis

The OOA section contains a system definition, that introduce the problem which is solved during the OOD. The system definition is summarised by the FACTOR criterion. The system definition identify possible classes. At last the application domain is set up, in which functions and program flow is analysed.

7.2.1 System Definition

The system definition is a clarification of the CMU specifications given in the test bed analysis, in section 3.6.2 on page 42.

The CMU has two purposes. One is to log all CAN frames, when the CMU is put into capturing mode, and the other is to send CAN frames. These tasks are carried out through a user interface. This user interface has to present all the functionality of the CMU, and be easy to navigate for the user.

The CMU needs one CAN port for both sending customised CAN frames and capturing every transferred CAN frame on the CAN bus. The CMU does not directly interact with the CAN port, but uses application interfaces provided from the TBE. This is illustrated in figure 4.2 in the test bed design chapter.

The CMU also has to be capable of sending frames from any of the test bed CAN ports. Because it is expected, that this functionality can be used for testing the simulated subsystems.

The process of capturing CAN frames is carried out by the TBE, and the frames are delivered to the CMU through an interface.

Before the CAN frames are presented to the user, it is necessary to analyse and process the content of the CAN frame. This processing is carried out by using a packet handler. The packet

handler checks the type of the captured frame. The processing makes sure that every received frame type can be displayed on a Graphical User Interface (GUI).

The packet handler can change the CAN frame format into hexadecimal, decimal, or to a textual description, that can be displayed on the GUI. The CAN frame is displayed in a table on the GUI.

The CMU interface can handle log files. The representation of the transferred frames, can be stored by using a file handling mechanism. The file handler can for open an already saved log file for further inspection. The log files are in csv format.

The CMU provides an interface, which is used for generating and sending frames to the CAN bus. The user can compose normal and extended frames by the user interface, and type in parameters such as CAN identifier, data frame (B1-B7), and B0. The parameters are validated, so erroneous CAN frames are avoided. The user is warned when incorrect parameters are typed.

When the frame is sent, the user gets feedback information in terms of a successful frame transfer, and a presentation of the sent CAN frame.

The functionality which is seen on the user interface for the CMU, is summarised in the items below.

- Starting and stopping frame capture.
- Storing and opening log files in csv format.
- Presenting CAN frames in a table on the screen.
- Changing format of captured frames between decimal, hexadecimal, and textural description.
- Sending composed frames.
- Selecting CAN parameters for composed frames.
- Typing in CAN parameters for composed frames.
- Validation of composed frames.
- Feedback to the user.

A graphical presentation of all functionality of the CMU is important. This is done by showing intuitive buttons and text for navigating the CMU.

7.2.2 FACTOR

From the system definition, the FACTOR criterion can be set up. Applying the FACTOR criterion to the system definition, makes it easy to distinguish, between more important system topics and less important. The FACTOR criterion is shown in table 7.1.

Criterion:	Description:
Functionality:	Captures every transferred frame on the CAN bus, and presents the result on a GUI. Interact with the user to compose a CAN frame that can be sent on the CAN bus. Make log files that can be stored on disk.
Application Domain:	The CMU is the part of the test bed, which presents CAN frames graphically.
Conditions:	The system is designed specific for the test bed application, and only works when the TBE is present.
Technology:	The CMU is implemented on a standard PC. The implementation is based on free GPL (GNU General Public License) software. These terms demands that the source code of the CMU is open. By this, it is possible for other persons to make further improvements of the test bed software.
Object System:	CAN frames, frame composer, packet handler, file handler, and GUI.
Responsibility:	The CMU is responsible for presenting CAN frames on a display. It makes the user capable of producing CAN frames, that are frames on the CAN bus. All captured transmissions on the CAN bus can be stored for further treatment.

Table 7.1: *The FACTOR criterion for the analysed CMU.*

7.2.3 Problem Domain

The problem domain deals with definition of system boundaries. The CMU is divided into smaller tasks, which are easier to analyse and design, compared to designing the complete system at once.

The first task is to define a set of classes to make up the system. Secondly the coherence between the classes is found. Finally, the problem domain deals with the dynamics of the analysed system, in terms of what type of events that are carried out using the CMU.

Classes

From the system definition in section 7.2.1, it is possible to extract suitable classes. These classes are derived from the context of the system definition, in which significant nouns are used to present the classes.

Extracted classes are shown in table 7.2, with a description of the class purpose.

System class:	Class description:
GUI:	The GUI class contains all parameters accessible on the CMU. It is the interface towards the user, which is used for controlling the CMU. This is in terms of controlling when to start and stop the packet handler, how to compose frames, and when to store or load an already captured CAN transmission.
Packet handler:	When the packet handler is started it is responsible for getting CAN frames from the TBE. The CAN frames are then transformed to a suitable format, which the GUI is able to display.
Frame composer:	The frame composer class is responsible for creating and transmitting frames on the CAN bus, by the use of the TBE. The frame composer provides a range of predefined options for the user to create specific types of frames.
CAN frame:	A CAN frame object is created each time the packet handler notify the arrival of a new CAN frame on the CAN bus. A CAN frame is also created by the frame composer.
File handler:	The file handler creates csv files of the GUI contents in means of all captured CAN frames. When loading a stored csv file, the GUI is updated with its original contents, from when the csv file is created. The csv file is stored on disk.

Table 7.2: *Identified classes for the CMU.*

7.2.4 Structure

As the main classes of the CMU are located, the structure of the system indicates how the classes are related to each other. This is shown in the class diagram in figure 7.2.

Figure 7.2 shows that the file handler, packet handler, and the CAN composer are aggregated to the GUI. The CAN composer is also associated to a CAN frame. As can be seen in the figure, each CAN frame is associated to the packet handler.

Events

State diagrams are drawn for each class, where input and output is shown for the event. States within each class are shown as looped arrows. The state diagrams are shown in figure 7.3.

A description of the events is given in table 7.2.

7.2.5 Application Domain

The application domain determines who can use the system, and how to handle the system. These two demands are both seen from the users point of view, and from the systems point of

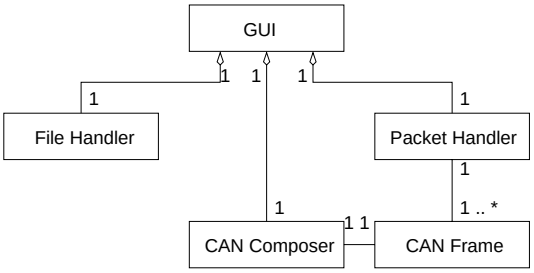


Figure 7.2: *Class diagram of the CMU.*

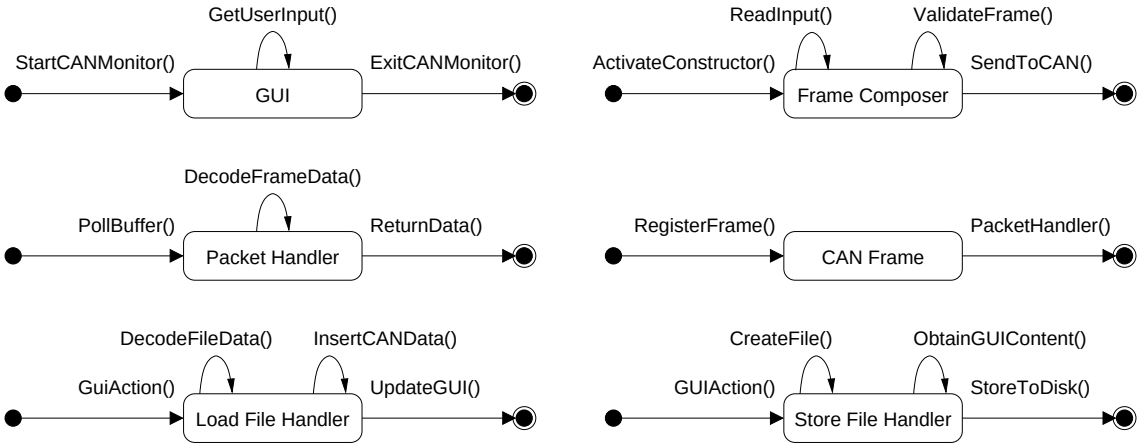


Figure 7.3: *State diagrams of the CMU.*

view. When the actors are clarified, the use cases for each actor are identified. Then functions for the use cases are given.

Use Cases

From the system definition, in section 7.2.1 on page 88, two main actors are identified. These actors are shown in table 7.3.

Actor:	Goal:
System operator:	Generate CAN frame.
	Load and store log files.
CAN frame:	Get recorded.

Table 7.3: *The actors and their goals.*

The type of functionality the actors can control, are defined as use cases. The use cases are listed in table 7.4.

Use case:	Description:
Generate CAN frame:	The operator composes frames using drop down menus, where predefined parameters are given. The operator can type in parameters as well.
Load and store log files:	The system operator can store or load captured CAN frames. The log file contains information of the captured frames. These are: frame number, time stamp, CAN identifier, frame type, B0, the CAN data, and the data length.
Get recorded:	Every CAN frame on the CAN bus is recorded when the packet handler is running. The frame type and the frame capturing time, are parameters which the packet handler sends to the GUI.

Table 7.4: *Description of the use cases functionalities.*

The program flow of the system which generates frames, is shown in figure 7.4. This system is the test bed frame composer.

When the frame composer is initialised, a connection to the TBE is established. The user has to enter valid frame parameters or select from a list, in order to be able to send the frame. The composed frame is validated and afterwards, it is sent using the established connection to the TBE. The TBE converts the composed frame into a suitable format, and transmit it on the CAN bus.

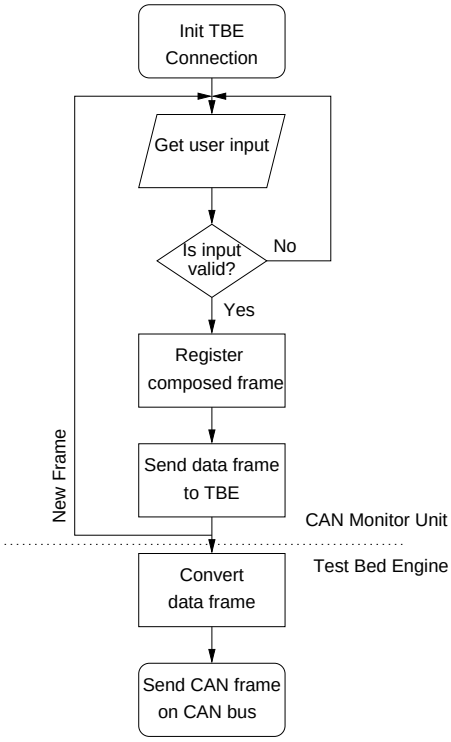


Figure 7.4: *Flow diagram of the CAN frame composer.*

Figure 7.5 shows the flow diagram of how CAN frames are recorded and how to load and store CAN frames. This is carried out by the CAN frame viewer system.

Like the frame composer, the CAN frame viewer establishes a connection to the TBE. If the user wants to record frames sent on the CAN bus, this is selected by enabling a capture button. If the button is not enabled, received frames are discarded and not shown on the display of the frame viewer. The recorded frame is decoded into a readable format, and displayed in a table on the GUI. When the capturing is not enabled, it is possible to store recorded frames, delete recorded frames, or load a frame transmission. Together, the frame composer and the CAN frame viewer constitutes the CMU.

7.3 Object Oriented Design

The OOD deals with defining and designing components for the CMU, in accordance with the chosen system architecture. Interfaces towards the TBE and towards the user are investigated. The GUI itself is designed on basis of the chosen software structure and the needed system functions.

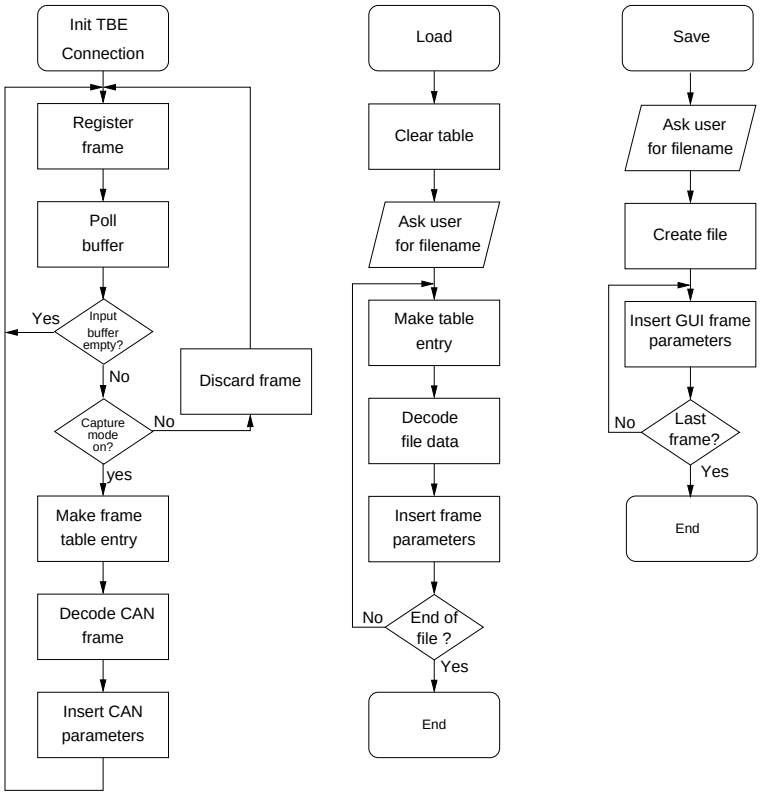


Figure 7.5: *Flow diagram of the CAN frame viewer.*

7.3.1 Process Architecture

The CMU is running on the same platform as the TBE and the WIU. The CMU consist of a number of classes and functions. Both the frame sending module and the frame capturing module has to run at the same time.

Real-time behavior is expected at some level, meaning that the display refresh rate should be fast enough to display incoming frames as they appear.

System Components

The developed system is a combination of two types of components, namely interfaces and functions. The interfaces in the CMU primarily consist of dialog boxes wherein interaction towards the user is carried out, but also interfaces for data communication to the TBE. Function components are used for calculations and transformation of CAN frames, on behalf of the users choice or demands.

Figure 7.6 shows the systems components and how they are connected across subcomponents. A short description of the design perspective of the subcomponents is given in table 7.5.

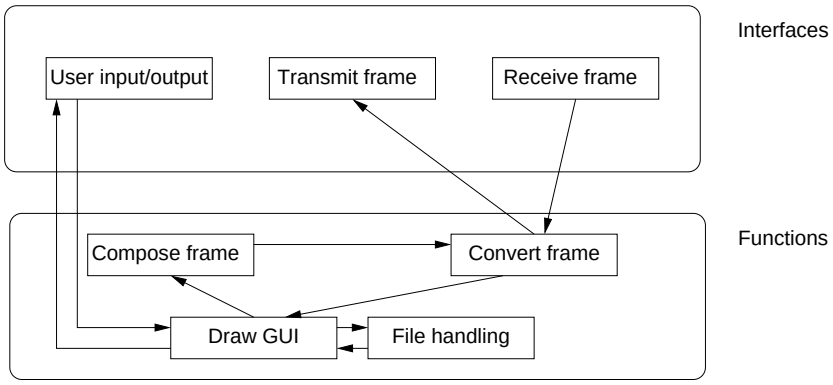


Figure 7.6: *Components for the CMU.*

Subcomponent:	Component description:
Transmit frame:	Uses an established communication channel towards the TBE to transmit the composed CAN frames.
Receive frame:	Receives CAN frames from the TBE.
User input/output:	Present data on the screen and register user input.
Compose frame:	Gather information from the user to compose a specific CAN frame.
Convert frame:	Converts human readable CAN frames into B1-B7 data CAN frames. As well as converting B1-B7 data CAN frames into an understandable format to be displayed on the GUI.
File Handling:	Loading and storing frames, for later inspection.
Draw GUI:	Maintains the graphically user interface, and records user inputs.

Table 7.5: *List of subcomponents in the CMU*

7.3.2 System Design

To realise the CMU, by using the components defined in table 7.5, certain tools and interfaces are required. These tools and interfaces are described in the following two sections.

GUI Tool

It is chosen to develop the CMU using the GUI library Qt from Trolltech. This choice is made, because Qt contains many well defined GUI classes. Every interaction towards the user requires graphical interfaces, such as input and output boxes, which the Qt library provides.

A discussion on how to develop programs using Qt, is given in appendix E.

The CMU is developed in C++ and Qt, by using the graphical editor Qt Designer. This editor implies, that auto generated source code is included from Qt Designer in the CMU. The included

code can not be omitted when compiling the CMU. Programmed components are negotiated with the Qt Designer, before it can be included in the auto generated source code.

The Qt Designer architecture has great impact of how to implement components, and thereby how to design the components for the CMU.

Interfaces

Interfaces towards the user are carried out by using the GUI components from Qt, as discussed previously. Interfaces to the TBE are more sophisticated, than parsing variables between programmed functions. This interface is discussed in the test bed design, section 4.3.2 on page 54, and the chosen interface is IPC message queues.

The data transferred between the CMU and the TBE are data structures described in section 4.4. To transfer data between the CMU and TBE, both systems has to establish a connection to the same message queue. This is carried out by using a unique identifier to identify the queue and the queue elements.

Implementation of the CMU is separated from the TBE. This is done to prevent performance- or stability issues, running the graphical environment, from spreading to the TBE.

7.3.3 Component Design

The CMU provides two types of service. Namely receiving CAN frames, and sending CAN frames. The CMU GUI implementation is divided up into two systems that maintains the mentioned services.

It is desirable that the structure of the test bed becomes as modular as possible, because then it is possible for future students to understand and continue the work on the test bed.

To minimise the risk of software failures, inputs from the user are validated. When the user types a CAN identifier, this value is in decimal between 0 and 536,870,912 for extended CAN frames, and between 0 and 2,048 for standard frames. The B0 byte has to be in hexadecimal between 0 and FF. The B1-B7 data is hexadecimal between 0 and FFFFFFFFFFFFFF. To protect the test bed, the CAN frame can not be sent if one or more of the CAN parameters are wrong.

The separation into frame composer and frame viewer programs, ensures that each program manage its own screen refreshing when needed. It also guarantee that composed frames are sent to the hardware CAN bus and grabbed by the TBE before the frame is displayed on the CAN frame viewer.

Due to the structure of source code generated from Qt Designer, it is not appropriate to develop classes that Qt already provides. From the class diagram in figure 7.2 on page 92, five classes presented. Instead of designing these five classes, the functionality of the mentioned classes is moved to the auto generated header file .ui.h, as discussed in appendix E.

The two developed programs for the CMU uses a number of subcomponents, shown in figure 7.6 on the facing page. In section 7.3.4 and 7.3.5, the programs are designed.

7.3.4 CAN Viewer

The functionality of the CAN viewer is described in the following section.

GUI

The GUI design focus primarily on usability, and where to put the GUI content on the users screen.

- A table presents the transferred CAN frames. The use of a table makes it possible to put CAN parameters in cells that are aligned under each other. For each new CAN frame the CAN viewer presents, a new table row is inserted.
- A button is used for the user to start and stop the CAN frame capturing. Buttons for loading and storing table content are presented, as well as a button for clearing the table.
- A number of radio buttons are used by the user to convert the received frames into the desired data type.
- For aesthetic and usability reasons, buttons which are related are grouped together.

Figure 7.7 shows the conceptual design of the CAN Viewer.

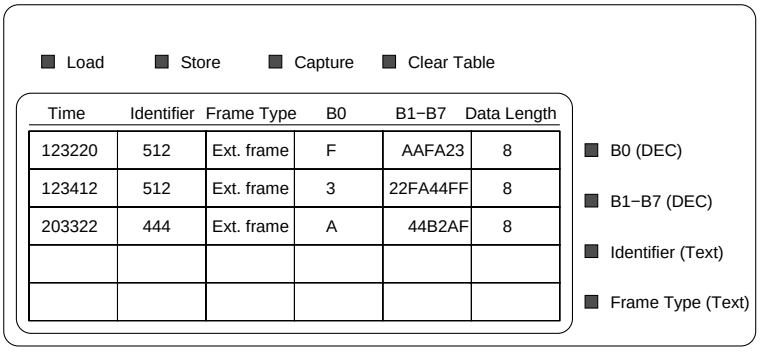


Figure 7.7: *CAN viewer, based on the conceptual GUI design.*

Input/Output

The input/output component handles data that are presented to the user, and takes arguments from the user.

- When buttons are pressed, it is an input for the CAN viewer, from the user.
- The output from the CAN viewer is displaying CAN frames in the table.
- Loading and storing CAN frames is carried out on files. The data format for storing CAN frames is .csv format.

Convert

Because the user should not be confused about raw date streams from the CAN bus, the data is converted into a readable format.

- When data is received by the CAN viewer, the data contents are converted into the data format selected from radio buttons, and shown on the GUI. CAN data in the AAUSAT-II consist of 7 data bytes (B1-B7). These are converted into a string.
- Frame types from the CAN card are converted to readable text messages and displayed on the GUI, if the text format is selected.

Receive

Reception of data from the CAN bus is delivered from the TBE. The receive component takes CAN frames from the interface between TBE and CMU.

- Every CAN frame that is received by the TBE, is put into the IPC message queue one at a time. Each frame is packed into an item of a CAN frame data structure. To remove an item from the queue, the IPC message queue has to be read.
- Not all CAN frames are interesting to receive for the user. Only when the CAN viewer is put into capturing mode the CAN frames should be displayed. Unwanted frames are discarded immediately from the message queue. This is carried out by polling the IPC message queue at a high frequency. The frequency is controlled by a timer event.

File handling

A file handling component makes it possible to archive CAN frame transmissions on disk. The data is stored as a comma separated file, with a .csv extension.

- When storing a data transmission by using the file handler, the content from the generated table is put into a file. The user has to specify the file name and location to put the file.
- Loading an already stored CAN table file, overwrites the present content of the table. The user is warned about the situation, and is able to cancel the operation.

Table

In order to obtain a nice overview of all transferred frames, they are put into a table.

- Each time a CAN frame is captured by the CAN viewer, the frame is put into a new created row in the table. By this no unused table rows are shown.
- When a new set of frames has to be captured, the old frames can either be deleted, by removing all rows in the table, or the new frames can be put into the end of the previously captured frames.
- Before cleaning the table contents the user is warned that the content is not stored.
- The items in the table are: Time stamp, Identifier, Frame type, B0, B1-B7 Data, and Data length.

7.3.5 CAN Composer

The functionality of the CAN composer and the design of the GUI is given in the following sections.

GUI

The GUI of the CAN composer contains a number of GUI components, presented in the items below.

- The content of the GUI in the CAN composer contains a button for sending the composed frame, and a number of drop down menus.
- For easy frame composing, predefined CAN frame parameters are present in the drop down menus where parameters also can be typed in. The menus contains: identifier, B0, B1-B7 data, transmit card, and port.
- Two radio buttons are displayed. These can be set for changing between standard or extended CAN frame, and be set if the composed frame is a remote CAN frame.
- Feedback messages for the user is displayed in a output text field.
- If the typed in frame parameters are incorrect, a warning box is shown to the user about what is wrong.

Figure 7.8 shows the conceptual design of the CAN composer program.

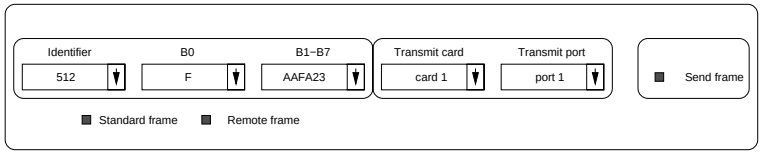


Figure 7.8: *CAN composer, based on the conceptual GUI design.*

Input/Output

The input/output component ensure that the user can interact with the functions of the GUI.

- Inputs for the CAN composer is the selected or typed in CAN parameters. The user can choose between standard or extended identifiers and whether it is a remote frame or data frame.
- If the typed in frame parameters are incorrect, the content of the composed frame is not used for input to the system.

- For debugging purpose, the user can choose from which CAN card and port the TBE sends the composed frame.
- The CAN composer gives feedback to the user, when a frame is correctly composed and sent. If the frame is incorrect, the user is also notified.

Convert

The converting component converts the constructed frame into a format that can be sent to the TBE.

- When typing a string of data for transmitting, it is converted into 7 bytes of B1-B7 data packets plus the B0 byte, to fit the 8 byte data structure of a CAN frame.
- CAN identifiers and B0 parameters which are presented in readable text, are converted into the CAN data format.

Transmit

- Before the verified CAN frame is transmitted, the CAN frame is packed into an item of a data structure.
- The item is put into the transmitting IPC message queue. It is up to the TBE to decide when to poll the transmitting queue and transmit the frame.

Compose frame

This component is the last action on the user composed frame before it is sent to the TBE.

- The composed frame is put together of CAN identifier, B0, and B1-B7 data. Options for choosing extended, standard, and remote frame are presented.
- All user selected and typed in CAN frame parameters has to pass the validation, before the CAN frame is finally composed.

Both user interfaces contains dialogue boxes that pop up and warns the user, if incorrect parameters are typed in, or buttons are inactive.

Part III

Implementation

Part III contains the implementation of the test bed software, designed in the previous part. The overall file structure of the implemented test bed is given, as well as an overview of the threads used in the test bed engine.
In the end, screen-shots of the graphical user interfaces is shown.

Test Bed Implementation

CHAPTER 8

This chapter contains a description of how the test bed designed in the past chapters is implemented. A section is used to describe the implementation of each of the three test bed modules.

The documentation of how the designed test bed is implemented, does not go into technical details on how each individual loop or expression is implemented. Instead the overall structure of the implemented software is described, along with the relevant versions of the tools used.

The tool Doxygen (http://www.doxygen.org) is used to generate browsable documentation of the source code. The tool parses the source code, and generates lists of functions, variables, and so on. These lists can be navigated using a web browser, and the source code of each function can be displayed with syntax highlighting. This is done for all three modules, since Doxygen supports both C, PHP, and C++ syntax.

Doxygen Documentation on CD-ROM:
The enclosed CD-ROM contains browsable versions of the source code for TBE, WIU, and CMU.

8.1 Test Bed Engine

The test bed engine (TBE) is the software module responsible for CAN bus communication, and interfacing the CMU and the database. The TBE is linked together with the modified CAN library, described in appendix C. Furthermore, the TBE communicates through the corrected CAN driver, which is also described in appendix C.

The TBE is designed to have four parallel threads, each operating on one CAN card through the common CAN library. However, the vendor does not guarantee that this software library is thread proof, meaning that hazards does not arise when accessed by multiple threads from the same program. Furthermore, inspection of the library, and device driver source code, indicates that such considerations have not been taken into account. Therefore it is decided to implement the test bed engine in two versions. One version using four threads and another version where the four threads are converted to four separate programs.

Appendix C contains a description of the tests made on the these two versions, to determine whether performance or reliability differences could be measured, when running "light threads" compared to "heavy threads". The test does not reveal or proof any difference, but hardening the

Softing software for multi-threaded use and hazard avoidance is still considered an issue, that could be subject to further research. This could be combined with a project oriented towards developing a hard real-time driver for the CAN card.

Based on the results of the test, and the fact that the usability of the test bed is improved when using the multi-threaded version, it is decided to continue implementation using the original TBE design from chapter 5.

8.1.1 Software Versions

The TBE is compiled by the GNU Compiler Collection (gcc) version 2.96[1] and executed on the test bed platform running Mandrake Linux 9.2 on top of kernel 2.4.22, patched with RTAI version 24.1.11. The database API is obtained by using libmysql version 4.0.15.[2]

8.1.2 File Organisation

The file organisation closely follows the layer structure of figure 5.3 on page 63, with files for each layer, and a few extra files for infra-structural reasons. The TBE includes the following files:

- Engine.c
- Fifo.c
- IrqHandler.c
- Mysql.c
- Settings.h
- Canlib/Canlib.c
- Card1/Port1.c
- Card1/Port2.c
- Card2..4/Port1.c
- Card2..4/Port2.c

The contents of these files are described in the following.

Engine.c

This file contains the main routine of the test bed engine, and the routines for scanning keyboard, initialising CAN cards, starting tests, and decoding incoming CAN frames.

Engine.c also contains an error-handling function. This function is used to write errors to a log file, "testbed.log" when it is called. It takes three arguments, namely the name of the function where the error occurs, a message to be written to the log file, and a value to tell whether the engine must stop when the error occurs, or continue operating.

[1] Obtained from the *gcc version 2.96 20000731 (Mandrake Linux 9.2 2.96-0.83mdk)* package.
[2] Obtained from the *libmysql12-devel-4.0.15-1mdk.i586.rpm* package.

Fifo.c

Fifo.c contains the routines for initialising the FIFOs on the CAN cards, and reading frames from the FIFO, when interrupts are received and handled. Furthermore, Fifo.c contains a routine that passes incoming commands, received through the keyboard, to CAN card one. This makes it possible to use the functionality provided by the CAN API on card one. This functionality includes transmission of frames, resetting card FIFOs and other debug facilities, as described in appendix C.

IrqHandler.c

This file contains the threads for interrupt handling. A common initialisation thread and one interrupt thread for each card.

Mysql.c

All functions using MySQL communication are written in this file. This includes functions for retrieving test definitions, running the test, and storing the results in the database.

Settings.h

Settings.h is a common header file included by all C-files. The header contains all global definitions, prototypes and the inclusions of standard library headers.

Canlib/Canlib.c

The library for communicating with the CAN cards is implemented in Canlib.c. This file contains the entire modified CAN API, as described in appendix C on page 151. The library is placed in a separate subdirectory.

Card1/Port1.c

The software attached to reception of frames on a given card number, is placed in a subdirectory named CardX, where X denotes the card number from 1 to 4. Card1/Port1.c contains the software operating the test bed itself. This includes the routines for buffering incoming frames for the CMU and for later upload to the database. The transmission of manually entered frames by the CMU, is also handled from this file, although the actual transmission can be done on any of the active PCI cards.

Card1/Port2.c

Port two on card one is not used in the present implementation, due to the time stamp issue mentioned in section 4.2.1. Therefore this file does not contain actual functionality, but the structure follows the other PortX.c files so that this port can be used for future extensions of the test bed.

Card2..4/Port1.c

The rest of the CAN cards also have a Port1.c file attached. This file is used for implementing the subsystems to be simulated on port one of a given CAN card. The implementation includes an initialisation routine executed by the test bed engine on startup. Another routine is used to shut down the subsystem, when the test bed is shut down. However, the most important routine for the user, is a thread which is invoked on each CAN frame retrieval, unless the system is excluded from participating in an ongoing planned test.

The CAN acceptance filter can be set in all ports, and they are also set in these files.

Card2..4/Port2.c

The Port2.c files are identical to the Port1.c files, except they operate on port two of the CAN cards.

8.1.3 Threads

The implementation of the test bed requires multiple processes running in parallel. Table 8.1 contains a list of all threads running in the test bed.

Thread Name:	Description:	File:
Main*	Main routine of the test bed engine	Engine.c
kbd_thread	Scans keyboard input.	Engine.c
MonitorThread	Receives incoming IPC requests from the CMU.	Card1/Port1.c
Interrupt_pthread1	Handles interrupts on card one	IrqHandler.c
Interrupt_pthread2	Handles interrupts on card two	IrqHandler.c
Interrupt_pthread3	Handles interrupts on card three	IrqHandler.c
Interrupt_pthread4	Handles interrupts on card four	IrqHandler.c
C1P1Thread	Receives signal on incoming frames on port one of card one, and retrieves frame.	Card1/Port1.c
C1P2Thread	Receives signal on incoming frames on port two of card one, and retrieves frame.	Card1/Port2.c
C2P1Thread	Receives signal on incoming frames on port one of card two, and retrieves frame.	Card2/Port1.c
C2P2Thread	Receives signal on incoming frames on port two of card two, and retrieves frame.	Card2/Port2.c
C3P1Thread	Receives signal on incoming frames on port one of card three, and retrieves frame.	Card3/Port1.c
C3P2Thread	Receives signal on incoming frames on port two of card three, and retrieves frame.	Card3/Port2.c
C4P1Thread	Receives signal on incoming frames on port one of card four, and retrieves frame.	Card4/Port1.c
C4P2Thread	Receives signal on incoming frames on port two of card four, and retrieves frame.	Card4/Port2.c

Table 8.1: *Parallel threads running in the test bed engine.*

* Main is not an actual thread, but the main routine creates the other threads, and continues operation of scanning the interface file for starting tests.

8.1.4 Buffers

Incoming frames are stored in a number of queues while interrupt signals are sent back and forth. Since these buffers are accessible from two sides, they are all protected by Mutex locks. Table 8.2 contains a list of the queues and their respective Mutex locks.

Queue Name:	Description:	Mutex Variable:
inQueue[1]	Stores frames between FIFO handler and frame decoder	C1InQueueLock
inQueue[2]	Stores frames between FIFO handler and frame decoder	C2InQueueLock
inQueue[3]	Stores frames between FIFO handler and frame decoder	C3InQueueLock
inQueue[4]	Stores frames between FIFO handler and frame decoder	C4InQueueLock
portQueue1[1]	Stores frames between frame decoder and port thread	C1P1QueueLock
portQueue2[1]	Stores frames between frame decoder and port thread	C1P2QueueLock
portQueue1[2]	Stores frames between frame decoder and port thread	C2P1QueueLock
portQueue2[2]	Stores frames between frame decoder and port thread	C2P2QueueLock
portQueue1[3]	Stores frames between frame decoder and port thread	C3P1QueueLock
portQueue2[3]	Stores frames between frame decoder and port thread	C3P2QueueLock
portQueue1[4]	Stores frames between frame decoder and port thread	C4P1QueueLock
portQueue2[4]	Stores frames between frame decoder and port thread	C4P2QueueLock
testQueue	Stores data frames received during tests	TestQueueLock
testcases	Stores test frames to be sent during tests	TestCaseQueueLock
sentTestCases	Stores test frames already sent during the ongoing test	SentTestCaseQueueLock

Table 8.2: *Buffers present in the test bed engine implementation, and their Mutex locks. The numbers in brackets corresponds to the card number.*

To prevent Mutex calls causing system software architecture bugs, as described in section 3.2.7, a common function for locking and unlocking the correct Mutex locks is made.

TBE Files on CD-ROM:
The enclosed CD-ROM contains the source files of the TBE.

8.2 Web Interface Unit

This section present the web interface unit implementation. First the server software versions are described, followed by a description of the file organisation.

8.2.1 Software Versions

The test bed web-server is an Apache-AdvancedExtranetServer version 2.0.47 for Mandrake Linux/6mdk installation. The PHP module is version 4.3.3, extended with gmp version 3 library. The gmp library is GNU **M**ultiple **P**recision Arithmetic Library, and it is needed to be able to operate with integers of arbitrary length.

The database server is MySQL Version 12.21 Distribution 4.0.15, for Mandrake Linux GNU (i586). The "testbed" database is created with standard settings and permissions.

All user input validation is done client-side by JavaScript. All the JavaScript functions are stored in a javascript.js file, which is included in index.php.

8.2.2 File Organisation

In the web interface unit design, it is given that a main menu for the test bed interface contains the four concepts, namely: "CAN identifiers", "test case set", "test", and "test reports". Additionally the menu has an "about" option, to show a short user manual for the test bed.

All navigation is done by the use of HTML query-strings. The index.php file takes the "show" query-string and redirects to the specified file. If no file with the given name exists, the first page (a AAUSAT-II picture) is shown. At the top of the index file a "top.php" file is included to show the top of the page including the horizontal menu bar. At the bottom of the index file a similar "bottom.php" file is included to show the bottom of the page.

A file containing a number of functions ("functions.php") is included in the index file. The functions are pieces of code which are often used. These are calls to the database, redirect, error handling, show header and footer boxes, test passed/failed figures, common used form elements, and a text array containing confirmation text, to be used in the top of pages.

The five elements in the main menu are attached to one file each. These are:

- canid.php
- testcaseset.php
- test.php
- report.php
- about.php

The files are split into a number of parts, and they use the query-string to decide what part to enter when loaded. The parts are described in the following sections.

canid.php

The CAN identifier page is split into four sections, namely: Show a CAN identifier, insert a new, delete one, and show an overview of them all. It is controlled by the following query-strings: canid, insert, delete, and order.

The canid query-string contains the id of the CAN identifier to show or edit. The insert query-string either contains the text "new" or "edit". If it contains "new" a new identifier is to be set into the database, and if it contains "edit" an existing identifier is edited. The delete query-string contains the id of the CAN identifier to be deleted.

If no query-string exists, an overview of all CAN identifiers is shown and ordered by the date and time of creation. If the order query-string exists the overview is ordered by the text in the query-string.

testcaseset.php

The test case set page is also split into four sections. They are: Show/edit a test case set, insert a new, delete one, and show an overview. It is controlled by the following query-string: testcaseset, insert, delete, and order. They all work as explained in canid.php.

If a test case set is deleted, both the test case set, the test cases for the test case set, and the id's in the *combine* table in the database are deleted. (See figure 6.6 or appendix D.)

The user input for byte 1-7 is given in one field, which makes the range of that field go from 0x00 to 0xFFFFFFFFFFFFFF. Plain PHP is not able to handle such a number with an acceptable precision, since an integer has a maximum of 32 bits. This is where the gmp module for PHP is required. It is able to handle integers with a precision of 64 bits.

The gmp module converts the hexadecimal number to a decimal number, and makes the calculations on the decimal representation. But it is not able to convert the decimal representation to hexadecimal representation for 64 bit numbers. Therefore high numbers are saved in the database as integers in decimal format and converted to hexadecimal numbers, when presented to the user.

test.php

The test page is also split into four sections. They are: Show/edit a test, insert a new, delete one, and show an overview. It is controlled by the test, insert, delete, and order query-strings. If a test is deleted, both the test and the test id's in the *combine* table are deleted.

report.php

The report page is also split into four sections. They are: Show, insert, delete and show overview. It is controlled by the report, insert, delete, and order query-strings.

The report query-string contains the id of the test report to show. It is not possible to edit a test report, because once it is made, it becomes static. The insert query-string contains the test identifier. It is used to determine, to what test, the CAN bus data is compared, in order to auto-generate the test report.

If a test report is deleted, both the test report and the frames for the report are deleted. Also the .csv files for the deleted test report is deleted. The traffic on the CAN bus during the test is not deleted. The show query-string works as described above.

Two comma separated files containing the frames and all CAN bus traffic are generated when a test report is shown. They are saved in the files/ folder and named "frames_report{reportid}.csv" and "candata_report{reportid}.csv". A link to the two files is placed on the page where the reports are shown. If a user accidently deletes a file, it is regenerated next time the test is shown.

about.php

The about page contains a short user manual of how to use the test bed, and the name and email-address of the authors.

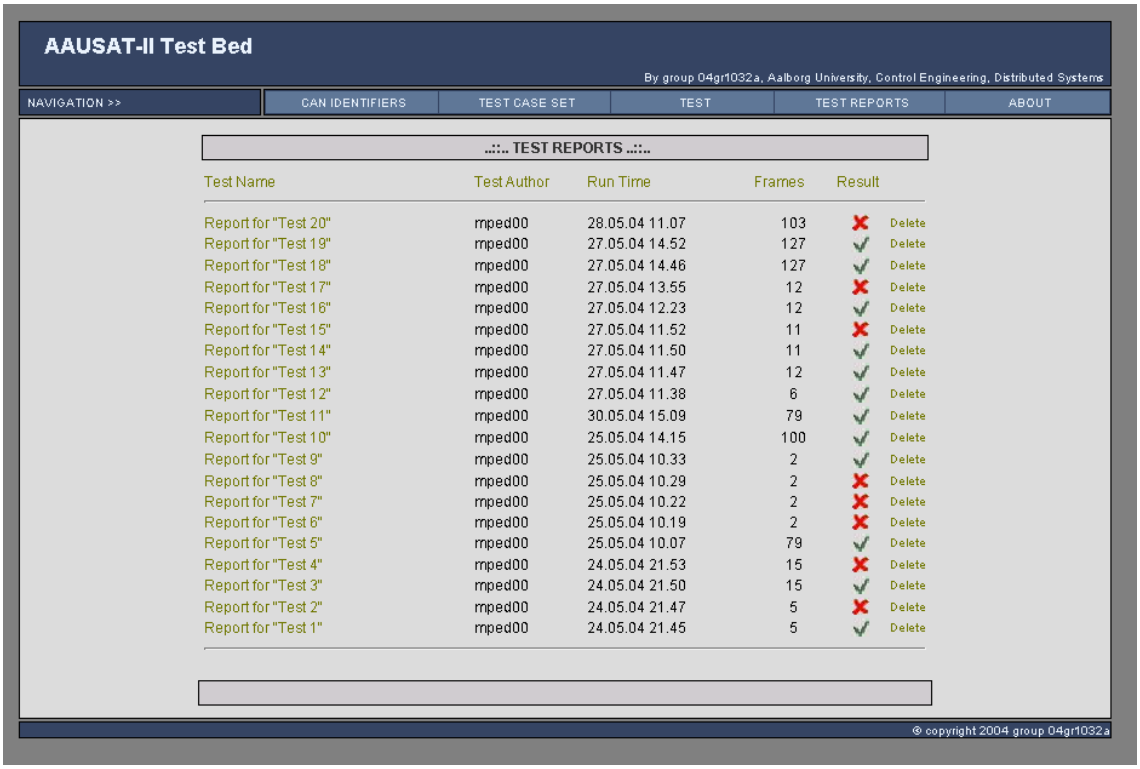


Figure 8.1: *Screenshot of the report overview of the web interface unit.*

Figure 8.1 shows a screen-shot of the report overview of the web interface unit. The shown reports are the test reports used to test the test bed. This is explained in chapter 9.

WIU Files on CD-ROM:
The enclosed CD-ROM contains the source files of the WIU.

8.3 CAN Monitor Unit

This section covers the implementation of the frame composer and the frame viewer, which together makes up the CAN monitor unit.

The implementation is carried out on the test bed hardware, by using the Linux operating system. The CAN monitor is developed using Qt Designer version 3.3.2, and compiled by g++ version 3.3.1.

Schedule the Message Queue

One of the important tasks for the CAN monitor, is to schedule when to poll the IPC message queue for new frames. If frames are not removed immediately from the IPC queue, they will

pile up in the queue, and that makes the captured frames useless at the time they are polled. Different methods for scheduling such events exist. In Qt programs, it is not recommended to use standard POSIX threads due to software portability issues. Instead Qt provides two classes, namely QThread and QTimer.

QThread works as POSIX threads, and is able to prioritise when to run each thread. The QTimer can be used to count down a timer, and when it reaches zero, it runs a software routine.

The precision of QThread and QTimer is expected to be the same, depending on the underlying operating system.

It is chosen to use the QTimer because it contains the signal/slot mechanism as explained in appendix E. When the QTimer times out, it emits a signal to poll the IPC message queue.

Real-time Consideration

The screen of the CAN monitor unit is automatically updated when an event happens to the frame composer or frame viewer. This is for instance when a CAN frame is captured, or when a button is pushed.

Any real-time behaviours are not implemented for the two GUI's. This is due to the fact, that the only time demanding issue is when to update the screen, which is considered less critical.

CAN frames arriving at the CMU have a time stamp, provided by the TBE. Frames stored in the IPC message queue, can not be deleted by other programs than the TBE and the CMU. There is no strict real-time requirements in handling the queued frames. There is also no real-time requirements on the time from transmitting a frame in the GUI until it is present on the CAN bus.

File Organisation

Due to the fact that both frame viewer and frame composer are developed by the Qt Designer toolkit, the organisation of the file structure for both systems is the same. In the following a short description on the content of the files is given. The files named "form.", must be replaced by "viewer." and "compose." for the frame viewer and the frame composer respectively.

Further information about the file structure is given in appendix E.

main.cpp

Creates and destructs the GUI. Includes the form.h where the content of the GUI class is located.

form.ui

XML description of the GUI designed by the Qt Designer toolkit. This file is compiled into form.cpp and form.h.

form.cpp

The implementation file of the GUI. Every Qt GUI object is constructed and destructed within this file. Because the file is auto generated from the form.ui, modifications on the file are overwritten when recompiling the GUI project. Objects that are implemented for the CAN composer and the CAN viewer by using Qt Designer, are also defined in this file.

form.h

Header file for the implemented GUI objects.

form.ui.h

This file contains the user developed functionality. GUI objects are connected to functions which remain in the form.ui.h. Within this file, Qt signal/slot connections are used for running specific functions e.g. to create rows in the monitor table and validate input parameters from the user. Functions and objects that are implemented in this file, are designed in section 7.3.4 on page 97 and in section 7.3.5 on page 101.

Qt Classes

Besides the GUI components mentioned in the object oriented design section 7.3 on page 94, a number of Qt classes are used. These classes are listed in table 8.3, along with a short description. Further information of the Qt classes can be found at Trolltech's webpage [Trolltech, 2004].

Qt class:	Functionality:
QWidget	Creates the GUI main window.
QTable	Establish the table for the CAN viewer and uses member functions for adding and deleting rows, columns.
QTimer	Schedules when to poll the IPC message queue.
QButton	Creates push buttons.
QGroupBox	Makes a group box frame with a title around GUI elements.
QLabel	Provides text on the window.
QComboBox	Gives a drop down menu.
QRadioButton	Makes a radio button with text description.
QMessageBox	Small pop-up widget with boolean input buttons, e.g. Yes/No messages.

Table 8.3: *Classes inherit from the Qt library.*

Figure 8.2 shows a screen shot of the implemented CAN frame composer and figure 8.3 of the CAN frame viewer, both in action on the test bed.

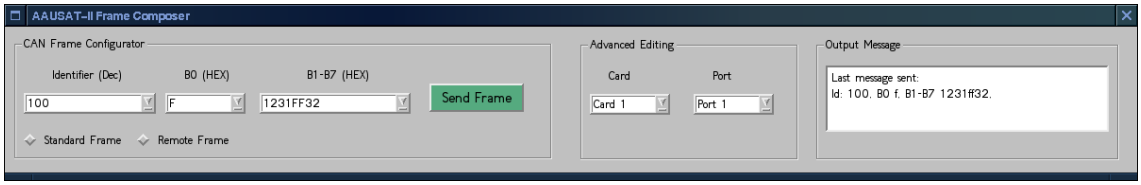
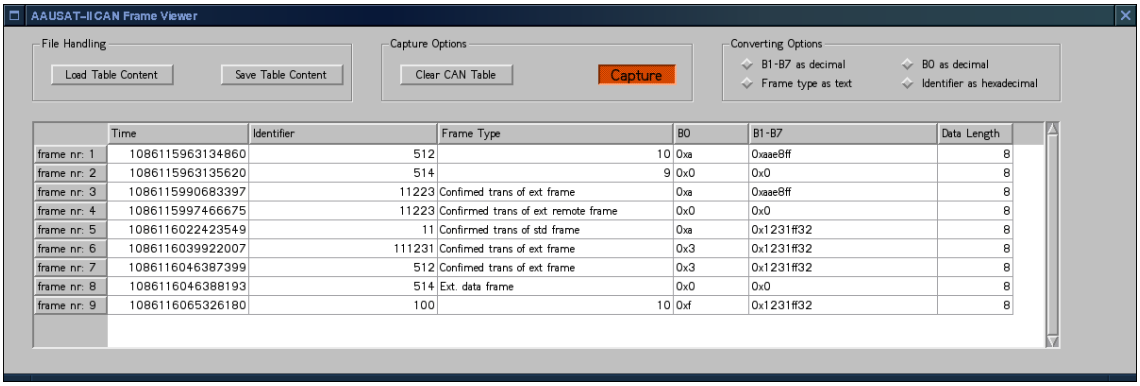


Figure 8.2: *The CAN frame Composer of the CAN monitor unit.*



	Time	Identifier	Frame Type	B0	B1-B7	Data Length
frame nr: 1	1086115963134860	512	10	0xa	0xaae8ff	8
frame nr: 2	1086115963135620	514	9	0x0	0x0	8
frame nr: 3	1086115990683397	11223	Confirmed trans of ext frame	0xa	0xaae8ff	8
frame nr: 4	1086115997466675	11223	Confirmed trans of ext remote frame	0x0	0x0	8
frame nr: 5	1086116022423549	11	Confirmed trans of std frame	0xa	0x1231ff32	8
frame nr: 6	1086116039922007	111231	Confirmed trans of ext frame	0x3	0x1231ff32	8
frame nr: 7	1086116046387399	512	Confirmed trans of ext frame	0x3	0x1231ff32	8
frame nr: 8	1086116046388193	514	Ext. data frame	0x0	0x0	8
frame nr: 9	1086116065326180	100	10	0xf	0x1231ff32	8

Figure 8.3: *The CAN frame viewer of the CAN monitor unit.*

CMU Files on CD-ROM:
The enclosed CD-ROM contains the source files of the CMU.

Part IV

Test & Conclusion

Part IV contains the tests performed on the implemented system and the conclusion. The system is tested against the requirements stated in the end of the analysis in section 3.8 on page 45. Based on the test results, and the knowledge gained throughout this project period, a conclusion is written. The conclusion summarises, and comments the obtained results, and puts the project into perspective. As a final remark, future improvements, or research within the project area, are suggested.

Test

CHAPTER 9

The purpose of this chapter is to verify that the test bed fulfils the given requirements. From the test results it is possible to conclude, whether the implemented system behaves as expected from the designed system.

The strategy for testing the test bed, is to use the V-model described in section 2.2 on page 10. Unit tests are made on the three modules of the test bed individually as shown in figure 9.1.

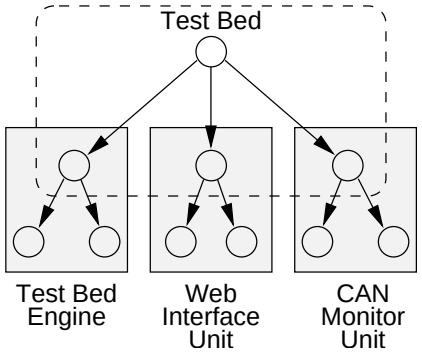


Figure 9.1: *Test strategy for the test bed. Unit tests are made on the three modules individually.*

On relatively small software systems, such as this test bed, the integration test and the system test can advantageously be combined. When the unit test is performed on the three modules, the combined integration and system test is performed on the complete test bed (the dashed line in figure 9.1). The principle of bottom-up integration test is used, known from the test analysis in section 2.7 on page 24.

The test bed integration and system test is referred to as system test throughout this chapter.

9.1 Unit Tests

As mentioned, the three modules are unit tested individually. Path testing is performed on all modules, but also other tests are performed to verify the functionality. In the following, a description of the performed tests is given.

9.1.1 Test Bed Engine

The test bed engine is developed by making use of the modified CAN library attached to the corrected CAN drivers from Softing GmbH. Additional test software is attached to the driver, which is used to test the TBE. The subsystems to be included and compiled with the TBE is tested in the system test, because they interacts with other modules of the test bed. The TBE does not require any interaction from the user, when testing is in progress.

To verify that the CAN frames actually is send on the CAN bus, an oscilloscope is connected to the CAN bus. The frames all appeared on the CAN bus as expected.

9.1.2 Web Interface Unit

The WIU is tested by the use of functional testing techniques. Boundary value testing is performed on all user input and database variables. All specified functionalities are implemented, and many of these can be tested by visualisation.

The usability features such as delete confirmation, the user input copy fields, and status bar is implemented by client side JavaScripts. The JavaScript functions are tested by use of a JavaScript console in the Mozilla web-browser.

9.1.3 CAN Monitor Unit

Boundary value testing is also performed on the user inputs from the CAN monitor unit. The monitor is tested by sending CAN frames from the frame composer, and receiving them on the CAN frame viewer. The results are matched against each other to verify that the CAN monitor works an intended.

Table 9.1 shows the test results from the unit tests.

Module:	Test description:	Result:
TBE	- Unit test of the test bed engine.	✓
	- Initialisation of CAN cards.	✓
	- Transmission of CAN frames.	✓
	- Reception of CAN frames.	✓
WIU	- Possible to create and edit CAN identifiers, test case sets and tests.	✓
	- Automatic generation of test reports.	✓
	- Generation of .csv files from the test reports.	✓
CMU	- Sending specified standard, extended, and remote frames.	✓
	- Specify the card and port to send from.	✓
	- Capture CAN frames and specify the format (HEX or Dec).	✓
	- Save and load .csv files of CAN bus frames.	✓

Table 9.1: *Unit test results for the test bed.*

9.2 Integration and System Tests

The system test is performed by running a series of tests, where input and expected output of each test case is precisely known. Each functionality of the test bed is tested using a specific test case. The output of the performed test is evaluated in accordance to expected behaviour. If the intended test results are met, the system test is successful.

A small description of the 20 test cases is listed below, and table 9.2 shows the test results. The tests are all performed from the web interface, and all traffic during tests are logged on the CAN monitor. The traffic logged by the CAN monitor unit frame viewer, and in the web interface unit must be the same.

Test Reports on CD-ROM:
All test reports from the tests are included on the enclosed CD-ROM.

Test 1 - Single frame test
Test 1 tests if it is possible to send a single frame from the web interface with a given identifier and data. A driver in subsystem one sends five frames to the CAN bus, when it receives a specified frame. Successful if test passes.

Test 2 - Single frame test with faults
Test 2 tests if the web interface makes false reports. The same frame as in test 1 is sent, and the same driver is in subsystem one. The expected outputs all differs a little from the driver output, so this test is successful if all test cases fail.

Test 3 - Interval test
Test 3 tests if it is possible to make an interval test. An interval test case is specified, and the expected results from subsystem one are the same as its input. The test is successful if the test passes.

Test 4 - Interval test with faults
Test 4 is similar to test 2, but this time with interval test. Subsystem one send back data that differs a little from the expected, and the test is successful if all test cases fails.

Test 5 - Several frames test
Test 5 tests the stability of the test bed when many frames are sent. The maximum number of intervals (19) on an interval test is chosen, and the same subsystem as in test three is used. The test is successful if the test passes.

Test 6 - Subsystem test one
Test 6 to test 9 tests the subsystem enabling and disabling. Two subsystem drivers must send one frame when they receive a given frame. In test 6 both systems are disabled, and the test is successful if none of the subsystems answer.

Test 7 - Subsystem test two
In test 7 one subsystem is enabled while the other is disabled. The test is successful if only the not deactivated subsystem sends back a frame.

Test 8 - Subsystem test three
In test 8 the other subsystem is enabled while the first is disabled. The test is successful if only the not deactivated subsystem sends back a frame.

Test 9 - Subsystem test four
In test 9 Both subsystems are enabled. The test is successful if both subsystems answer with a frame.

Test 10 - Timing test one
The time from a frame is transmitted on the CAN bus, until a subsystem is given an interrupt signal and sends back a frame is measured to approximately 400 μs. It may seem like a long time, but the computer also has a lot operations to do. When a CAN bus frame is received, there will be eight interrupts to be serviced. This is 50 μs for each interrupt, which seems reasonable.

Test 10 tests the timing of the test bed. A subsystem is set to send 100 frames to the CAN bus when a specified frame is received. All 100 frames are expected to be sent to the CAN bus within 40 ms. The test is successful if the test passes.

Test 11 - Timing test two
Test 11 also tests the timing of the test bed. An interval test is specified, and a subsystem is set to send back a frame with the same data. The frame-interval is set to 1 ms, and the test is successful if the test passes, and the frame interval is 1 ms.

Test 12 - Multiple subsystems test one
Test 12 tests if all subsystems are able to receive and send frames to the CAN bus. All subsystems are set to send back a frame when a specified frame is received. They must send identical identifier and different data to the bus. Successful if the test passes.

Test 13 - Multiple subsystems test two
Test 13 tests the subsystem intercommunication. One frame is sent from the test bed, the first subsystem triggers on the frame, and sends a new frame. Subsystem two triggers on that frame and sends a new frame, which subsystem three triggers on, and so on, until all six subsystems has sent frames. All subsystems also sends another frame to the bus, so the test is successful if 12 specified frames appear on the CAN bus.

Test 14 - Subsystem calculations test one
Test 14 tests if the subsystems are able to make calculations on the received data. A number of interval test frames are send to the CAN bus, and a subsystem triggers on the identifier. The subsystems doubles the received data and send out the result. Successful if the test passes.

Test 15 - Subsystem calculations test two
Test 15 is identical to test 14, but a failure is initialised in the subsystem. The data-type used to make the calculations on the received data is wrong, and it must send invalid data. Successful if only some test cases passes.

Test 16 - Acceptance filter test one
Test 16 tests if the acceptance filters in the subsystems works. A single frame is sent, a subsystem triggers on the identifier and sends six frames with different identifiers to the CAN bus. Another subsystem triggers on all identifiers and sends back a frame for all received frames. The acceptance filter in the subsystems are set to accept all identifiers, and to send back two frames each. Successful if 12 frames are received from the CAN bus.

Test 17 - Acceptance filter test two
Test 17 is similar to test 16, but the acceptance filter in subsystem two is set to allow only uneven identifiers. Therefore only 9 of the 12 expected frames are expected on the CAN bus. Successful if 9 frames are received.

Test 18 - Multiple test case sets in sequential order
Test 18 and test 19 tests if the test bed is able to send frames from multiple test case sets in sequential and random order. The frames in test 18 are sent in sequential order, and it uses the test case sets for test 3, test 10, and test 13.

Because of the high amount of data on the CAN bus, the expected output time for the specified frames in the test is increased. Test 18 is successful if same test result as in the three mentioned tests appears.

Test 19 - Multiple test case sets in random order
Test 19 is similar to test 18, but this time the frames are sent in random order. Successful if same test result as in the three mentioned tests appears.

Test 20 - Accuracy test
Test 20 tests the accuracy of the time between the frames sent from the test bed. 100 frames are sent to the bus from the web interface unit, and the CAN bus traffic is logged in the CAN monitor module for further analysis.

Test:	Test description:	Result:
Test 1	Single frame test	✓
Test 2	Single frame test with faults	✓
Test 3	Interval test	✓
Test 4	Interval test with faults	✓
Test 5	Several frames test	✓
Test 6	Subsystem test one	✓
Test 7	Subsystem test two	✓
Test 8	Subsystem test three	✓
Test 9	Subsystem test four	✓
Test 10	Timing test one	✓
Test 11	Timing test two	✓*
Test 12	Multiple subsystems test one	✓**
Test 13	Multiple subsystems test two	✓
Test 14	Subsystem calculations test one	✓
Test 15	Subsystem calculations test two	✓
Test 16	Acceptance filter test one	✓
Test 17	Acceptance filter test two	✓
Test 18	Multiple test case sets in sequential order	✓
Test 19	Multiple test case sets in random order	✓
Test 20	Accuracy test	✓***

Table 9.2: *System test results for test bed engine.*

* Test 11 shows that the test bed is not able to send frames with 1 ms interval with the given RTAI mode (periodic). The frames are send immediately after each other. Though test 20 tells that the test bed is able to send frames with 10 ms interval very precise. The problem only persist for frames below 10 ms. If the mode is changed to RTAI oneshot mode, the test bed is able to send frames with 1 ms, so the test is made in oneshot mode. The two modes are compared in section 9.2.1.

** In the test results from test 12 it is seen that a high number of error frames are sent by the cards. It is done because they loose their arbitration (read about the CAN standard in appendix B on page 143). If test 12 is performed multiple times, the cards will enter "bus off" state because their error counter exceeds 255.

*** Statistics needs to be done on test 20 to verify the test. This is done below.

The subsystems can also be triggered by the CAN monitor frame composer. The CAN monitor frame viewer verifies that the same data appears on the CAN bus, no matter of the test is started by the WIU or the CMU.

9.2.1 Timing Test

In test 20, 100 frames are sent to the CAN bus. In the first test, the RTAI module is set in periodic mode, and secondly the RTAI is set in oneshot mode. A histogram of the test results is shown in figure 9.2 and 9.3, and table 9.3 compares the test results.

System:	Mean $\bar{x}$:	Minimum:	Maximum:	STD:	Sample size:
Periodic mode:	9.9998 ms	9.8550 ms	10.1560 ms	0.0019 ms	100
Oneshot mode :	10.0348 ms	9.8921 ms	10.1730 ms	0.0017 ms	100

Table 9.3: *Statistical indicators derived from the sampled data.*

The histograms show that most of the frames are sent very accurate with 10 ms interval in periodic mode. The oneshot mode is not as accurate as the periodic mode, the interval mean time is 35 μs above the desired. It means that the oneshot mode introduces a small delay for each frame. This might be annoying if very large tests with many frames are sent.

The periodic mode is more precise, but it is not able to handle frame interval times less than 10 ms. This is not acceptable, and therefore the test bed is running in oneshot mode. If hard real time is introduced to the test bed computer, the interval times will be more precise. But hard real time might seam overkill to implement in the test bed computer, since the timing requirements are not that strict.

The test result show that the deviation of the frames sent from the test bed is acceptable for the test bed.

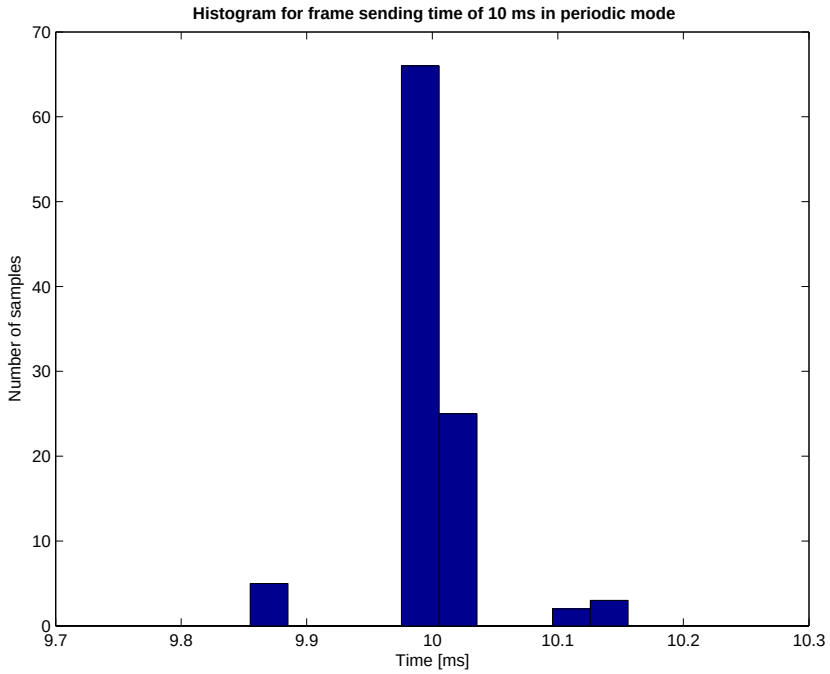


Figure 9.2: *Histogram of test results of test 20 in periodic mode.*

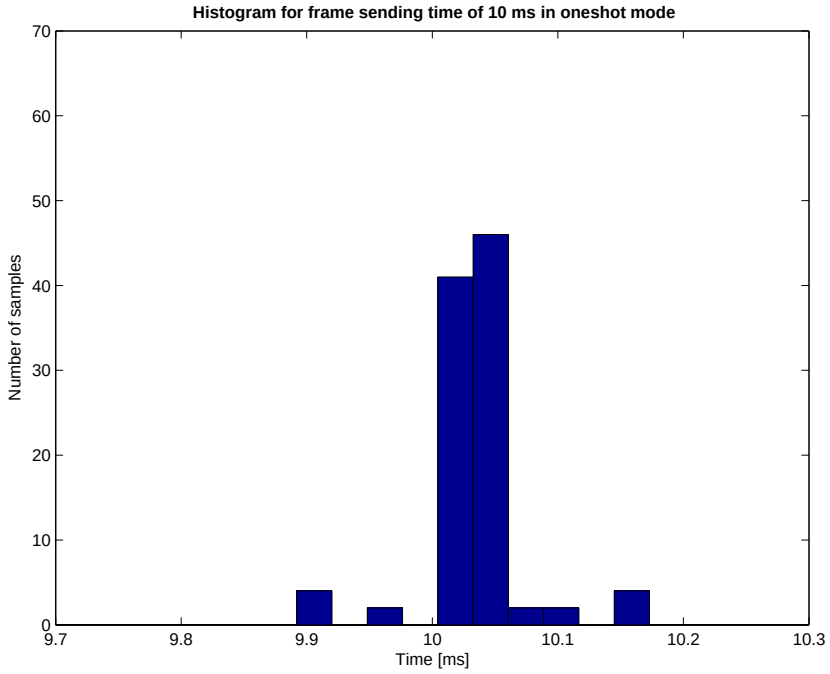


Figure 9.3: *Histogram of test results of test 20 in oneshot mode.*

Conclusion

CHAPTER 10

10.1 Summary

Within the specialisation of Distributed Application Engineering at Aalborg University, this master's thesis documents the development of a test bed for the AAUSAT-II pico-satellite.

AAUSAT-II is based on the CubeSat programme from Stanford University, featuring a ten by ten centimeter cube, equipped with subsystems for attitude determination and control, onboard computer etc.. AAUSAT-II is negotiating for launch in November 2005. The internal satellite communication system features a concept called INSANE, which is a protocol overhead based on CAN bus technology. The concept restricts the use of identifiers, and reserves some of the data capacity in CAN frames for protocol layer- and service type identification.

To guarantee reliable and robust behaviour of complex distributed systems, such as satellites, an extensive amount of testing is necessary. To successfully achieve a reliable product, the developers must consider testing throughout the entire development process. This can be supported by applying reputable and well established methods, such as the V-model, to the development process.

However, extensive testing is a time consuming and monotonous task, once the tests are defined. The nature of the V-model entails that multiple iterations of the process may be necessary, hence the tests have to be executed many times. If this is done manually, the risk of introducing test errors is large, and the probability of not testing thoroughly enough is large. This can be avoided by developing a test bed for automating the test execution.

It is important that a test bed supports testing at as many development levels as possible. When developing smaller subsystems, unit testing must be completed before combining them to larger units. Then integration testing is necessary, and prior to delivering a finished system, the system testing must pass. All of these testing phases are supported by the test bed.

10.1.1 Analysis

Based on an analysis of a number of test methods suitable for software validation at different system levels, functional testing, in terms of a combination of robust boundary value- and equivalence class testing, is chosen. Functional testing methods are the most suitable for systems where the testers, or in this case the test bed developers, do not have complete insight in the software structure of the devices under test. Equivalence class testing has a good balance between the robustness of the test, and the amount of data that needs to be specified by the user.

A statistical distribution of software errors is analysed. The discovered errors are categorised in error types. The proportion of each type, compared to the total number of errors, and their relation to the development- and test phases of the V-model, is described.

To ensure extensive support for all test phases, three test modes are found necessary. The test bed needs to support planned and structured tests, in terms of both equivalence class tests and single frame tests, and spontaneous tests consisting of manually entered CAN frames. The spontaneous tests are handled by a graphical user interface, where the user can log the ongoing CAN traffic and send frames directly on the CAN bus.

The planned equivalence class tests are defined by the user and stored in the test bed. The user specifies input data, expected output data and a time interval, during which the output must be received for the test to pass. Besides testing towards subsystems connected physically to the CAN bus, the test bed needs to be able to simulate subsystems.

Based on a discussion of the satellite interfaces and error scenarios, as well as an analysis of the desired functionality, a set of design criteria and requirements are outlined.

10.1.2 Design

With reliability, robustness, modularity, intuitivity, and simplicity as design criteria, the test bed design is outlined. The robustness and reliability is necessary to ensure that reliable results are obtained from tests. The modularity is desirable, because a test bed is a system which could be expanded and adapted to changing specifications, and because a modularised structure makes it possible to improve parts of the system, without needing to redesign the entire test bed. The simplicity and intuitivity is required for the test bed to be successful in supporting the satellite developers in obtaining a high possibility of mission success. If the test bed is too difficult to use, it is not used at all.

The test bed design is divided into these three modules:

- Test Bed Engine (TBE).
- Web Interface (WIU).
- CAN Monitor (CMU).

The test bed engine manages communication on the CAN bus through four Softing PCI CAN adapters, each equipped with two CAN ports. For infra structural reasons, this makes it possible to simulate six subsystems in the test bed. The web interface module is used to configure the planned tests, and to determine if a test passed or not. When a planned test is executed, the test bed engine logs all data occurring on the CAN bus during the test. When the test is finished, the data is stored in a database. The web interface then processes the logged data, and compares this to the test specification. A report is then generated, to display which test cases that passed or failed, and what traffic occurred during the test. The CAN monitor module handles the spontaneous CAN communication and traffic logging. Users can make CAN frames to be sent on the CAN bus, and the ongoing CAN traffic is presented to the user.

10.1.3 Implementation

The test bed is implemented on a 2.4 GHz Pentium 4 based computer, running Mandrake Linux with an RTAI patched kernel. An Apache web server with PHP support is installed along with a MySQL database server. The GMP library is used to make PHP capable of handling 64 bit integers.

Test Bed Engine

The TBE is implemented in the C programming language. The timing behaviour is controlled using soft real-time in user space, provided by RTAI's LXRT. To maintain the execution of the six simulated subsystems, and the TBE itself, POSIX threads are applied to obtain parallel processing on a uniprocessor system. Internal data buffers are protected by Mutex locks to ensure data consistency.

The device driver provided by Softing GmbH is corrected, thereby making it possible to use interrupts for signalling CAN events. Incoming CAN frames are stored in onboard FIFO buffers, until interrupt requests are served. The enclosed CAN library from Softing GmbH is optimised, thereby making it possible to operate four identical CAN cards in the same computer.

Web Interface Unit

The WIU includes a large relational database implemented in MySQL. The database manages tables of CAN identifiers, test case sets, tests, and test reports. Functions for creating, modifying, and deleting entries in these tables are implemented in PHP. JavaScript is used to perform form validation and usability features, such as status bar, timers, and confirmation dialogues.

Planned tests are started, when the WIU writes a desired test id in an interface file. This file is read by the TBE, and the necessary test case sets are retrieved from the database. Then the requested test is executed. When a test finishes, the test data is stored in the database, and the WIU generates the test report. Tests and test data can be exported to a file, as comma separated values, for further processing.

CAN Monitor Unit

The CMU is implemented in C++, based on class files from Qt, for the graphical user interface. The graphical user interface is developed by using the Qt Designer tool.

The communication between the TBE and the CMU is handled by Linux IPC message queues. One queue is used to send CAN frames from the TBE to the CMU, and another queue is used to send transmission requests from the CMU to the TBE. The CMU takes care of input validation, before sending transmission requests to the TBE.

CAN traffic logged by the CMU can be exported to a file, as comma separated values, for further processing. Such a log file can be imported by the CMU for inspection.

10.1.4 Test

Each of the three test bed modules are unit tested, before integrating the test bed. The unit testing is done partly using path testing, and partly using external tools, such as Softings test software and an oscilloscope.

When integrating the test bed, an experiment is made, to determine whether the Softing library and CAN driver are thread proof. This is done, because the software provided by Softing does not state to be thread proof, when program flow operates the CAN library. When the TBE operates on the CAN cards, this is done through functions in the CAN library. This library

contains some global variables and status indicators. To determine whether this causes hazard problems when operating the library from multiple threads within the same program, the TBE is implemented both as a multithreaded program using "light threads", and as four individual programs, all running one "heavy thread".

The test does not reveal any performance- or stability issues, but the topic is still considered interesting, and may be subject to further studies in future projects.

The test bed is integration tested by executing 20 test cases. Each of the test cases verify a certain functionality in the test bed. The tests are carried out by inserting test code as simulated subsystems. The test code is designed to perform a certain task on a given input. This input is configured as a test case, by using the WIU. Then the test is executed, and the result is shown on the WIU. At the same time, the test traffic is logged by the CMU. Then the input is sent from the CMU, thereby re-running the test. The test passed when the result generated by the WIU is as expected, and the test traffic displayed by the WIU corresponds to the traffic shown twice on the CMU.

The result of the tests confirm that the designed functionality work as intended, and that the test bed is performing as expected. Furthermore, the test bed is stable and reliable, and capable of handling error frames.

The test bed timing is tested with two different timer modes; periodic and oneshot mode. These modes determine, how RTAI runs its schedule timer. The timers are used to control the sleeping in between frame transmission, when a certain pause between frames is defined in the test case. The results show, that the test bed is most accurate when running periodic mode, but in this case pauses less than 10 ms can not be met. When running oneshot mode, the pause time can be down to 1 ms, but with an extra delay on each frame transmission of 35 μs.

Based on this result, it is decided to run the test bed in oneshot mode. One way of obtaining more accurate timing, is to port the test bed to hard real-time. However, this would require that the device driver and CAN library is ported to support hard real-time. This is considered a time consuming task, which is not possible within the time restrictions of this thesis.

10.2 Future Development

The modularised structure of the test bed makes it interesting to consider some of the topics briefly touched on by this thesis.

One such topic is the hardening of the Softing code. Both the device driver and the CAN library could be subject to a thorough analysis of hazard behavior, when heavily stressed. Such an analysis could focus on rewriting both software items, with real-time performance in mind.

In terms of the communication scheme of the AAUSAT-II, it seems appropriate to investigate the INSANE concept further. The concept includes a large protocol overhead, taken up approximately half the data transfer capacity of each CAN frame for protocol overhead, whereas much less than one per thousand available identifiers are used. This scheme seems open to improvements.

To convert the test bed into a general-purpose tool, for systems based on CAN bus, only one change need to be applied. The INSANE concept defines that the first data byte in the CAN frame is handled separately from the other data bytes. This restriction has been adapted by the test bed, for compatibility reasons. This means that CAN frames are displayed as two values by the test bed. The first data byte is one value, and the remaining seven bytes is the other.

The test bed can also be ported to other communication systems than the CAN bus. If for instance the CAN cards are replaced by Ethernet cards, the test bed could be ported to a test facility for testing network parameters.

10.3 Project Evaluation

Through the process of developing the test bed, it has become evident, that by using reputable development models, such as the V-model, complex systems including multiple programming languages, communication technologies, and interfaces, can be integrated successfully.

During the development, experience has been gained in disciplines such as working towards a testable result. The necessity of considering tests throughout the entire development process has been emphasised, and the advantages of doing so are numerous.

The importance of specifying both internal- and external interfaces in detail, has been realised. By doing so, a highly modularised test bed has been developed. This entails that future adaptions to changing specifications can be done without redesigning the complete test bed.

Altogether, the project group is satisfied with the knowledge gained, and the results obtained, through the process of elaborating this master's thesis.

Bibliography

[AAUSAT-II, 2004] AAUSAT-II Project (2004), Aalborg University, http://www.aausatii.auc.dk.

[Beizer, 1990] Beizer, B. (1990), *Software Testing Techniques*, International Thomson Computer Press.

[Boehm, 2001] Boehm, B. (2001), The spiral model as a tool for evolutionary acquisition, http://www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.

[Bosch, 1991] Bosch, R. (1991), Can specification - version 2.0, Robert Bosch GmbH.

[Bovet and Cesati, 2001] Bovet, D. P. and Cesati, M. (2001), *Understanding Linux Kernel*, O'Reilly.

[CUBESAT, 2003] CUBESAT (2003), Aalborg University CubeSat Project, http://www.cubesat.auc.dk.

[Softing, 2004] Softing GmbH, S. (2004), CAN-AC2-PCI user manual v. 4.03.

[Jorgensen, 2002] Jorgensen, P. C. (2002), *Software Testing - A Craftman's Approach*, CRC Press.

[Lawrenz, 1997] Lawrenz, W. (1997), *CAN System Engineering - from Theory to Practical Applications*, Springer.

[Madsen, 2000] Madsen et al. (2000), *Håndbog i Struktureret Programudvikling*, Ingeniøren|Bøger.

[Malotaux, 2004] Malotaux, N. (2004), Evolutionary project management methods, http://www.malotaux.nl/nrm/pdf/MxEvo.pdf.

[Mathiasen et al., 2000] Mathiasen, L., Munk-Madsen, A., Nielsen, P. A., and Stage, J. (2000), *Objekt Orienteret Analyse & Design*, Marko ApS.

[SSDL, 2004] SSDL, Stanford university - space system development laboratory, http://ssdl.stanford.edu/.

[Tanenbaum, 2003] Tanenbaum, A. S. (2003), *Computer Networks, 4rd. edition*, Prentice Hall.

[Trolltech, 2004] Trolltech (2004), Qt's main classes, http://doc.trolltech.com/3.2/.

[03gr731, 2003] Group 731, Aalborg University (2004), Designing, prototyping and testing a flexible on board computer platform for pico-satellites.

Part V

Appendices

About AAUSAT-II

APPENDIX A

This appendix describes in short the AAUSAT-II satellite and the CubeSat concept. It is inspired by the AAUSAT-II home-page; http://www.aausatii.auc.dk/

A.1 A Student Satellite

AAUSAT-II is a student satellite project at the University of Aalborg, Denmark, which was initiated in the Summer of 2003. The satellite project is a joint venture of the following institutes:

- Institute of Electronic Systems.
- Institute of Mechanical Engineering.
- Institute of Computer Science.
- Institute of Energy Technology.

The project is giving the students a unique chance to experience a real engineering project with real engineering problems.

A.2 The CubeSat concept

The CubeSat concept is developed by Stanford University, and has been chosen as framework for both AAU satellites. The reasons are a fast development phase (within two years), and an inexpensive launch possibility. Both reasons fit ideally to the project based education form used at Aalborg University.

CubeSat is a standardised platform for small orbital experiments. The weight of the entire satellite is less than one kg. The 10-centimeter cubes are designed to house small experiments that otherwise would be cost-prohibitive to flight validate. Universities, along with industrial interests, are able to bring their own CubeSat into orbit using a standardised deployment system developed by California Polytechnic State University.

The deployment system is known as P-POD (Poly Pico-satellite Orbital Deployer), which mounts to various launch vehicles. This way a particular CubeSat team is not concerned with the issues related to the launch.

A.3 The Satellite Structure

The AAU satellite is a CubeSat, thus its measurements have to fulfill the requirements set up by Stanford University and California Polytechnic Institute.

To achieve this, light materials are used for the structure of the satellite. Its design is based on a frame of aluminium, with sides made of carbon fibres.

High requirements have been set regarding the structure of the satellite and its integrity, as it has to withstand high temperature variations (+80 and -40 C), vibrations and shocks, radiation, and the vacuum in space.

However, the most vital task when designing the structure is to keep the weight limited, and to be able to fit all necessary subsystems into the structure.

A.4 AAUSAT-II Description

This section contains a description of the AAUSAT-II satellite and its subsystems. It is based on the design of the on-board computer and the internal communication protocol design made by group 03gr731 and 03gr722 at Aalborg University during fall 2003.

A.4.1 Mission

AAUSAT-II is equipped with a miniature Gamma Ray Burst Detector (GRBD) supported by the Danish Space Research Institute (DSRI).

The scientific mission objectives are threefold: [AAUSAT-II, 2004]

GRBD In Orbit Performance: The technological objective of the mission is to study the performance of this novel detector (GRBD) in space environment. Of particular interest is the knowledge of the background rates of the detector in space and the evolution of radiation damage effects during the mission. The performance is monitored by the detectors response to a (weak) X-ray line.

GRB detection: Although the detector is small, it is able to detect the largest gamma ray burst (GRB) of the universe, with one expected GRB/month. The on-board software constantly monitors the GRBD data flow for presence of GRBs. GRB data is transmitted to ground with high (< 1 s) time resolution.

X-ray Solar flare detection: It is planned to have the detector pointed at the sun with the purpose of Solar (X-ray) flare detection. This requires that AAUSAT-II is equipped with attitude control with a 5° pointing accuracy.

A.4.2 Subsystems

AAUSAT-II consist of several subsystems. Each subsystem is designed and implemented by one project group. These subsystems are all connected to a master bus. The CAN bus is chosen to be the master bus for AAUSAT-II. This is done by the former groups on the AAUSAT-II project. [03gr731, 2003].

Figure A.1 shows the computer architecture of the satellite as designed by former groups.

A small description of the subsystems for AAUSAT-II follows:

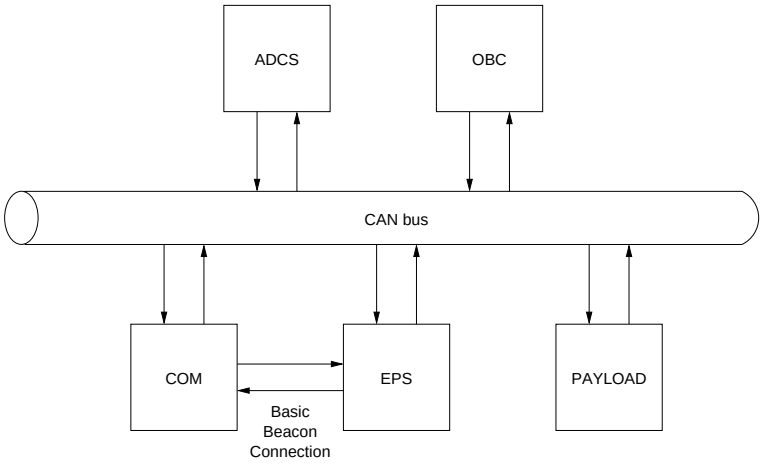


Figure A.1: *The computer architecture of AAUSAT-II.*

OBC - On-Board Computer

The on-board computer is the main computer of the satellite. Several threads are running on the main computer. One of them is a supervisor, which makes decisions of when and what to happen in the satellite. Another is the flight planner, which controls when the other subsystems in the satellite has to communicate with another subsystem or the earth. The OBC contains memory to store housekeeping data and measurement data from the payload.

ADCS - Attitude Determination and Control System

As earlier mentioned, an attitude control system is needed to rotate the satellite to a desired position. The ADCS uses sensors to determine the attitude of the satellite, and actuators to rotate the satellite. The OBC can receive the satellite position from the ADCS via the CAN bus.

COM - Communication System

The communication system is responsible for earth communication. Housekeeping data and payload data are sent from the OBC to the COM subsystem on the CAN bus and the COM subsystem enables a two-way communication with the ground station.

A basic beacon connection exist between the COM and EPS subsystems. This connection is only used if it is impossible to communicate with the satellite as intended. I'm alive signals is sent to earth, by use of this connection.

EPS - Electronic Power Supply

The electronic power supply unit distributes power to all subsystems in the satellite. The EPS must also be able to communicate with other subsystems, which is done by using the CAN bus. The power system knows how much power is available, and this might influence the decisions taken by the flight planer on the on-board computer.

PL - PayLoad

The payload subsystem is responsible for the control of the scientific equipment on-board AAUSAT-II. It must start the equipment, store the measurements, and send the measurements to the OBC via the CAN bus.

A.4.3 Internal Communication Concept

Almost all of the subsystems have their own micro controller connected to the CAN bus. This may seem that it is waste of power and space in the satellite, because all software could be put in the same micro controller. But this makes very high requirements of the pieces of software in that micro controller. Since AAUSAT-II is a university project, there is more than one criteria of success. If a two way communication can be established with the satellite from the ground station, the mission is a success.

Therefore it is good to have independent subsystems in the satellite. If for instance the ADCS does not work, the other subsystems can still bring valuable information to the ground.

If this satellite was to be built by the industry, the subsystems probably would be implemented in fewer micro processors than is the case for AAUSAT-II. The relatively high number of micro controllers lead to the specification of an internal communication protocol.

The INSANE Concept

Figure A.2 shows the INternal Satellite Area NEtwork (INSANE) concept. The network allows subsystems to communicate via a common protocol no matter how the subsystems connect to the network. They can be physically connected to the CAN bus or be running in threads in the OBC.

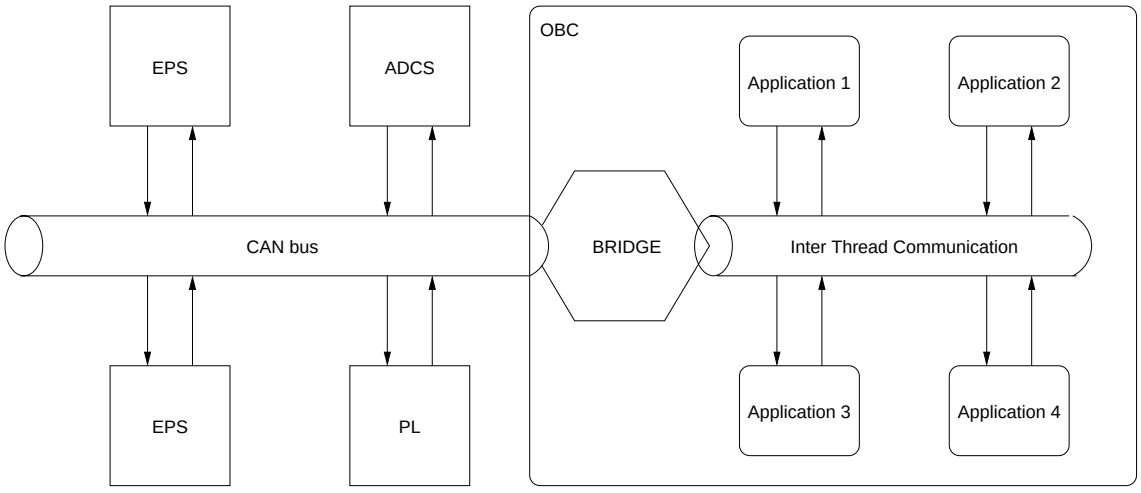


Figure A.2: *The INternal Satellite Area NEtwork (INSANE) concept.*

A bridge acts as a router between the CAN bus and the inter thread communication. Each frame type has its own identifier. All applications on the OBC and on the subsystems must

subscribe to the identifier frame they want to receive. This subscription frame is sent to the bridge, and the bridge stores the subscriptions in a large table.

Protocol Description

Figure A.3 shows the INSANE protocol stack.

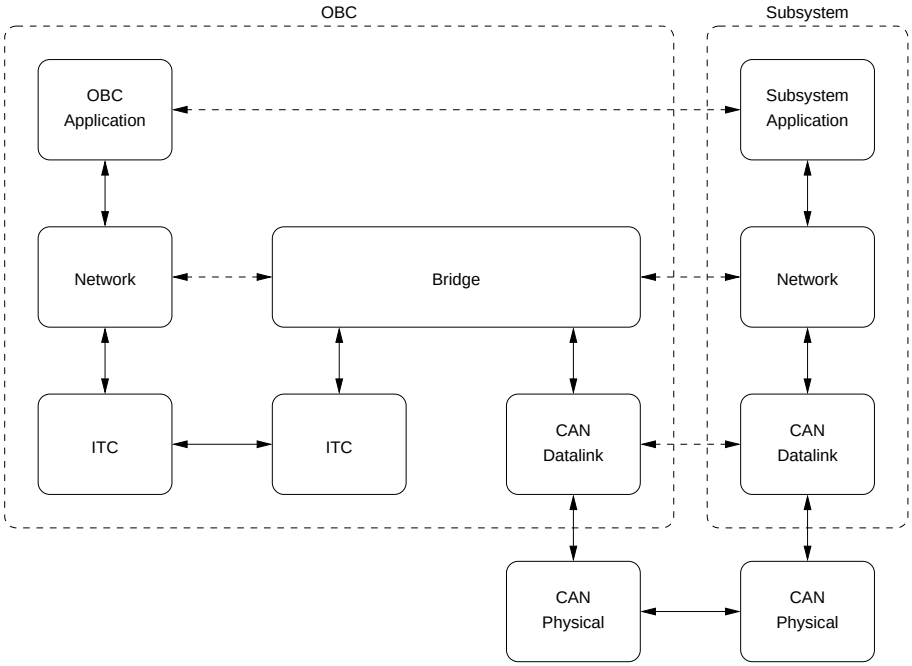


Figure A.3: *The INSANE protocol stack.*

The top level of the protocol stack is the application layer where the individual subsystem programs are placed. The general concept of the protocol is to make it invisible for the application whether it sends messages to a thread running on the OBC or to an application running on another subsystem.

The bridge listens to all the frames on the CAN bus and on the ITC (Inter Thread Communication) layer, and sends on the frame to the ITC layer/CAN bus layer if some threads/subsystems has subscribed to the message identifier.

Controller Area Network

APPENDIX B

This appendix contains a basic explanation of the CAN bus technology, history and applications. The frames for data transmission, transmission request, and error handling are described, and the communication scheme is outlined. The contents of this appendix is based on [Lawrenz, 1997] and [Bosch, 1991].

In the early 1980s Robert Bosch GmbH developed the CAN bus, and through collaboration with Intel the first silicium CAN controller, the 82556, was ready for market in 1989. Soon after other semiconductor manufacturers followed.

The CAN bus is developed as a so called "Autobus" protocol, with the purpose of providing the possibility of building complex computer systems in cars - with a very simple network structure, so that the necessary wiring and cabling is cut to a minimum. Since it's origin, the CAN bus has also become widely used within industrial control design, primarily due to high performance, low cost, real-time capabilities, and multiple vendors available.

B.1 Physical Characteristics

The CAN bus is a twisted pair cable with a terminal resistance of 120 Ω at each end, as illustrated in figure B.1.

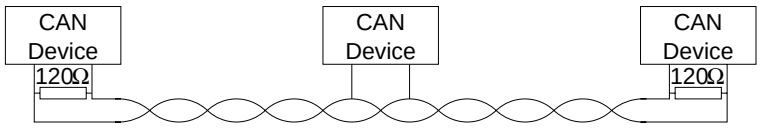


Figure B.1: *A CAN bus is a twisted pair cable with each end terminated by a 120 Ω resistor.*

The typical bus interface is a DSUB9 connected with pin connections as illustrated in figure B.2.

The signal levels 1 or 0 are determined by the electrical potential between CAN_H and CAN_L. Bits of level 0 are dominant to recessive bits of level 1. Each frame in a CAN bus system has a unique identifier, which is used to prioritise and identify packets.

B.2 CAN Bus Frames

The CAN bus uses different frames for data transfer, remote transmission requests, error signalling etc. These frames are described in the following.

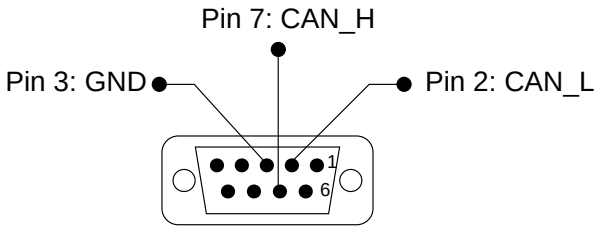


Figure B.2: *Pin connections of a DSUB9 CAN bus interface.*

B.2.1 Data Frame

The CAN bus protocol is available in two different versions:

- CAN 2.0A:
  The original CAN specification, occasionally referred to as "Standard CAN" or "Basic CAN", provides an 11 bit identifier.

- CAN 2.0B:
  This version contains the original CAN specification and an extended CAN specification providing an 11 bit identifier, as well as an optional 18 bit extension, giving a total identifier of 29 bits.

The two data frames of CAN 2.0B are are illustrated in figure B.3 and the fields are explained in table B.1 and B.2.

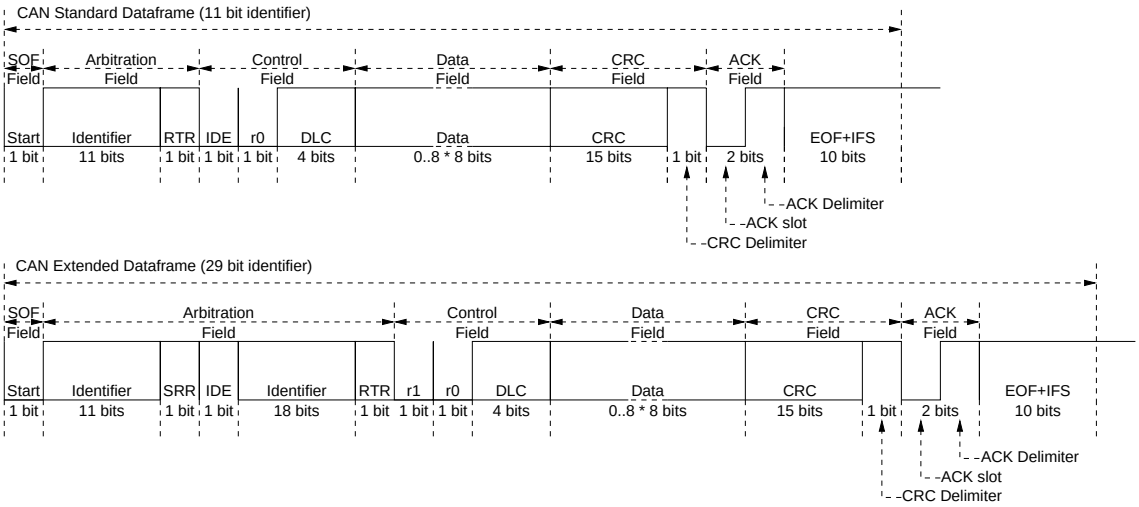


Figure B.3: *The data frames in standard and extended edition.*

The AAUSAT-II steering committee has decided to use the extended identifier. This is a requirement to use the INSANE meta-protocol.

Name:	Size:	Description:
Start	1 dominant bit	Marks the start of a frame.
Identifier	11 bit	The identifier/priority/address of a frame.
RTR	1 bit	Used to request a particular transmission from a remote transmitter.
IDE	1 bit	The IDentifier Extension is used to distinguish between standard- and extended identifier. 0 = standard, 1 = extended.
r0	1 bit	Reserved.
DLC	4 bits	The data length code for the data field.
Data	0-64 bits	The data field.
CRC	16 bits	CRC checksum used for error detection.
ACK	2 bits	An acknowledge bit stating that one or more nodes has received the frame.
EOF	7 bits	The EOF marks the end of a frame.
IFS	n bits	Inter frame spacing, under normal circumstances 3 recessive bits.

Table B.1: *Explanation of fields in the standard CAN bus data frame.*

Name:	Size:	Description:
Start	1 dominant bit	Marks the start of a frame.
Identifier	11 bit	The first part of the identifier/priority/address of a frame.
SRR	1 bit	Placeholder bit for the RTR bit used in the standard frame, but unused.
IDE	1 bit	The IDentifier Extension is used to determine between standard- and extended identifier. 0 = standard, 1 = extended.
Identifier	18 bit	The second part of the identifier/priority/address of a frame.
RTR	1 bit	Used to request a particular transmission from a remote transmitter.
r0	1 bit	Reserved.
r1	1 bit	Reserved.
DLC	4 bits	The data length code for the data field.
Data	0-64 bits	The data field.
CRC	16 bits	CRC checksum used for error detection.
ACK	2 bits	An acknowledge bit stating that one or more nodes has received the frame.
EOF	7 bits	The EOF marks the end of a frame.
IFS	n bits	Inter frame spacing, under normal circumstances 3 recessive bits.

Table B.2: *Explanation of fields in the extended CAN bus data frame.*

B.2.2 Remote Frame

The remote frame is identical to the data frame, except that the data field has length zero and the value of the Remote Transmission Request (RTR) bit. When the frame is a remote frame, the value of RTR is recessive. When a remote transmission is requested, a remote frame is sent, and then a data frame with the same identifier is sent as a reply.

B.2.3 Error Frame

If a host detects an error on the CAN bus, it sends an error frame to notify other hosts of the problem. These hosts then immediately discard the data received in the erroneous frame so far, in order to maintain consistency in the bus data.

The error frame contains a superposition of error flags from different nodes, and an error delimiter, as illustrated in figure B.4.

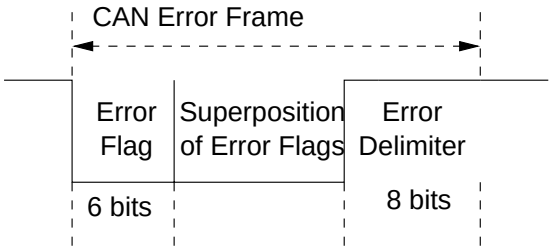


Figure B.4: *The CAN error frame.*

The error flag can have two different values:

- Active Error: Six dominant bits.
- Passive Error: Six recessive bits.

The different states depend on values of the error counters present in all CAN controllers. A Receive Error Counter (REC) and Transmit Error Counter (TEC) counts errors, and the error state is determined according to figure B.5.

B.2.4 Overload Frame

The overload frame, illustrated in figure B.6, is similar to the error frame. Overload happens when internal conditions of a receiving host requires the next data frame to be delayed (buffering reasons) or when special bit patterns occur in the interframe space.

B.2.5 Interframe Space

Data frames and remote frames are separated from preceding frames using interframe spacing. This frame is illustrated in figure B.7, and consist of three recessive intermission bits and eight recessive suspend transmission bits. The bus idle period can have arbitrary length, until another host requests bus access.

Further details on overload handling and interframe spacing is given in [Bosch, 1991].

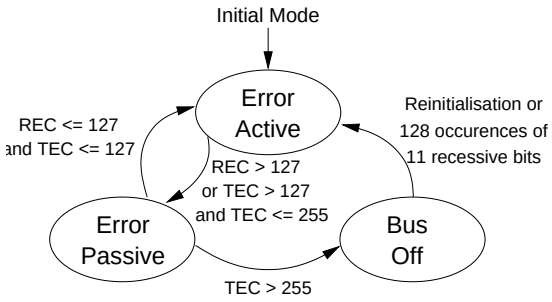


Figure B.5: *CAN error states.[Lawrenz, 1997]*

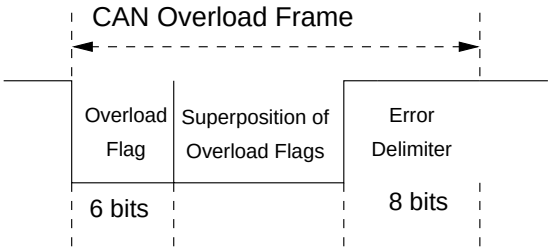


Figure B.6: *The CAN overload frame.*

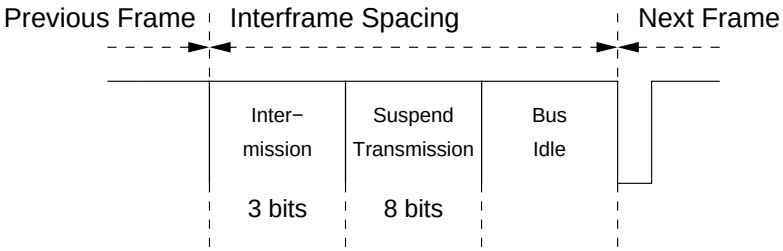


Figure B.7: *The CAN interframe space.*

B.3 Communication Scheme

CAN bus uses special mechanisms for coding, arbitration and error detection. These are described shortly in the following.

B.3.1 Coding

The frame elements SOF Field, Arbitration Field, Control Field, Data Field, and CRC (see figure B.3) are coded using bit stuffing. Whenever a transmitting node detects five consecutive bits of identical value in the outgoing bit stream, a complementary bit is stuffed into the actual transmitted bit stream.

The remaining elements CRC Delimiter, ACK Field, and EOF are of fixed form, and does not use bit stuffing.

The bit stream in a frame is coded according to the Non-Return-to-Zero (NRZ) method. This means that during the total bit time the generated bit level is either dominant or recessive. [Bosch, 1991]

B.3.2 Arbitration

The mechanism for accessing the bus, is a nondestructive, bitwise arbitration scheme, meaning that the winner of arbitration, the one with the highest priority, does not have to restart transmission once arbitration is won.

The CAN bus uses CSMA/CD+AMP technology - Carrier Sence Multiple Access with Collision Detection and Arbitration on Message Priority. This is illustrated when two devices desires to use the bus for transmission simultaneously. Both units wait for the bus to become idle (Carrier Sence), and then both sends a start bit (Multiple Access). Both units transmit simultaneously, until the two signals are not identical. In this case, the unit sending a dominant bit (0) continues to send, and the other device aborts its transmission, because it realizes that it's sent signal is not the same, as the signal received (Collision Detection). The aborting host then immediately switches to receive mode, because the frame being transmitted could require processing from this host (Arbitration on Message Priority).

If this happens outside the start bit, the arbitration field, and the ACK slot, it is treated as a bit error, and the error management routine is started.

B.3.3 Error Detection

The error frame is sent when one of the following errors are detected: [Lawrenz, 1997]

- Bit Error:
  When a bit transmitted by a node differs from the bit received by the same node.
- Bit Stuffing Error:
  When six consecutive bits are identical in a frame field where bit stuffing should have been applied.

- CRC Error:
  When the calculated check sum does not match the transmitted value.
- Form Error:
  When a fixed-form field contains one or more illegal bits.
- Acknowledgement Error:
  When ACK does not contain a dominant bit.

The Hamming Distance of the CRC checksum code is 6. This means that it is possible to detect up to 6 single bit errors distributed across a frame, or so-called burst errors up to a length of 15 bits. [Tanenbaum, 2003, page 194-198] and [Lawrenz, 1997].

When an error has occurred, the following steps take place:

1. Error is detected.
2. An error frame is transmitted.
3. The frame is discarded by every host.
4. The error count is increased in every host.
5. The original frame is retransmitted.

These steps are handled automatically by the CAN controller.

Interfacing the Softing CAN Controller

APPENDIX C

This chapter contains a description of the Softing CAN-AC2-PCI adapter used in this thesis. The card and the included software is described, and a number of modifications of the included software are made, to match the requirements of the test bed. A number of hardware tests are made during the test bed development, to ensure compatibility prior to fully implementing the test bed.

C.1 Card Description

The Softing CAN-AC2-PCI is used as CAN bus interface for the test bed. This card is chosen because two cards were already available at the Control Department of Aalborg University, and because the use of PCI cards instead of ISA cards makes it possible to use more modern computers with faster CPUs and bus speed. Two additional cards were bought using financial allocation from the ESN Study Board. These cards were delivered two weeks before the deadline for handing in this thesis.

The communication between card and PC happens through a Dual-Port RAM (DPRAM) that connects the PC to the onboard firmware.

The board is supplied with loadable onboard firmware, and a driver library including a small test application. The library can be linked with the application software, thereby providing CAN bus access to the application. The CAN connection is implemented by two separate CAN channels, each controlled by an SJA1000 controller from Philips Semiconductor, according to CAN 2.0B. The controller has support for both 11 and 29 bit identifiers. The TX/RX signals from the controllers are converted by two 82C251 transceiver circuits. The CAN controllers and the transceivers are separated by optocouplers to ensure galvanic separation.

C.1.1 Operational Modes

The CAN-AC2-PCI card can be used in three different operating modes:

- FIFO Mode.
- Dynamic Object Buffer Mode.
- Static Object Buffer Mode.

The "Static Object Buffer Mode" can only be used for 11-bit identifiers, and is therefore not considered. The "Dynamic Object Buffer Mode" can not be implemented using interrupts, hence the application software must poll the card to check if frames have been received. This behaviour is not desirable, because the test bed needs to operate on several CAN cards simultaneously, and therefore using CPU resources for constantly polling a CAN card is undesirable, and may lead to unsatisfactory performance.

Instead "FIFO mode" is used. This mode features interrupts on frame reception, and in- and outgoing frames are places in FIFO buffers on the CAN card. The FIFO mode structure is shown on figure C.1.

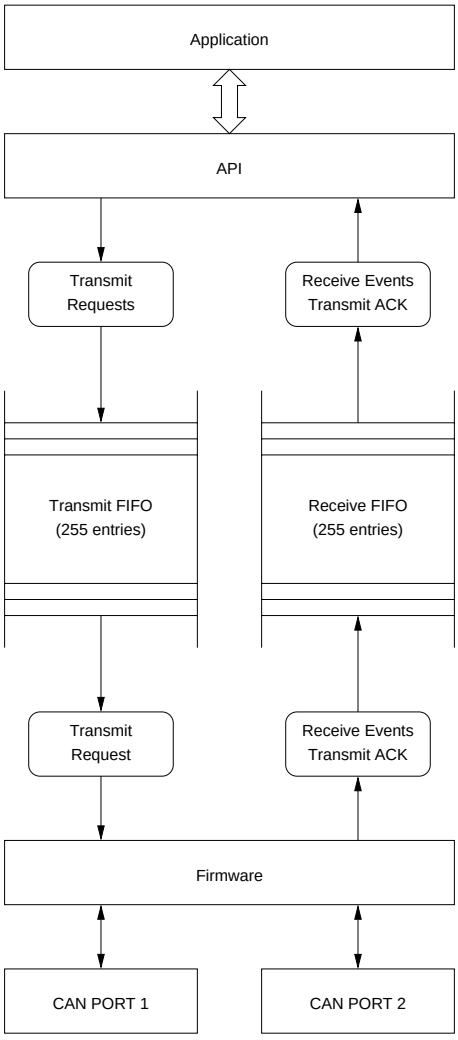


Figure C.1: *The FIFO Mode structure. [Softing, 2004]*

The "Transmit FIFO" handles all transmit requests from the application sent by `CANPC_send_data()` or `CANPC_send_data2()` depending on the port number. The application can obtain incoming messages, bus events, and transmit acknowledges through the "Receive FIFO" by calling `CANPC_read_ac()`. When a CAN frame is received, this function is used to retrieve the frame from the CAN card and decode parameters such as the port on which the frame is

received, the data bytes in the frame etc.. When using interrupts, the `CANPC_read_ac()` has to be implemented in the interrupt thread to monitor the bus.

To operate the CAN cards in FIFO mode, each card has to be initialised by following a certain pattern. First the CAN chips are set in reset mode, and then parameters such as bit timing etc. are set. Then the FIFO mode is enabled before selecting to use transmission acknowledge and error frame detection or not. Then the interrupt event is configured. Then the CAN chips are started. The entire initialisation process is shown in figure C.2.

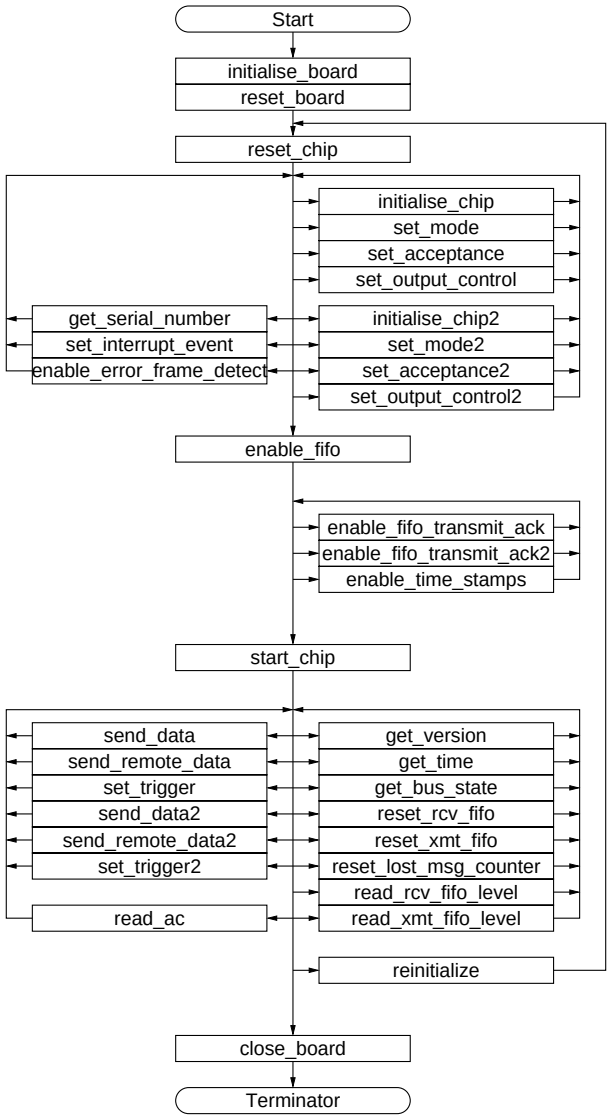


Figure C.2: *The flow chart for initialising FIFO Mode. [Softing, 2004] To reduce the size of the flowchart, the string "CANPC_" have been omitted in each box.*

The boxes below the "start_chip" are the functions provided by the CAN card API. These commands are available through keyboard inputs when running the driver and the test software. The menu for choosing the functions can be seen in figure C.3.

```
                COMMUNICATION KEYS

        CAN 1                       CAN2
t: transmit data can1          z: transmit data can2
r: transmit remote can1        f: transmit remote can2

    p: toggle identifier type standard/extended


                ADMINISTRATION KEYS

l: get time                    k: reset fifos
G: get fifo levels             L: reset lost msg counter
B: get bus state can1          N: get bus state can2
h: help menu                   d: disable/enable CANPC_read()
i: reinitialize                c: clear screen

                               q: quit
```

Figure C.3: *Screen shot of the menu for operating the CAN API during tests.*

C.2 CAN Card API

As described in section C.1.1, `CANPC_read_ac()` is used to retrieve incoming frames from the CAN card. This function is a part of the CAN card API or library provided by Softing, and returns a struct of data about the incoming CAN frame, and the card status, as well as a return code describing which type of event that caused the interrupt. Table C.1 shows the possible return codes from the function.

The data structure returned by the `CANPC_read_ac()` is the `param_struct`. This structure contains a number of parameters that describe the event that caused the execution of the `CANPC_read_ac()` function. It should be noted, that the parameters are only valid for certain return codes of the `CANPC_read_ac()` function. Table C.2 contains the parameters that are relevant when running in "FIFO Mode". Column two shows the return code for which the parameter is valid. The return codes are explained in table C.1.

The parameter `Bus_state` is used to determine the error state of the CAN bus. It can obtain the values 0, 1, or 2 corresponding to the states "Error Active", "Error Passive" and "Bus Off". These terms are described in section B.2.3.

The general program flow, when executing a function provided by the CAN API library, is that the function, which is some variant of a `CANPC_xxx()` function, is translated into an `ioctl_candriver()` command, which is then translated into a normal `ioctl` call operating on the actual device through the device driver. The device driver distinguishes between the individual cards by evaluating the file descriptor given as argument to the `ioctl` call. However, the CAN API library does not support operation on multiple cards by default. To do this, all `CANPC_xxx()` needs to be ported to taking a CardNr as argument. In `ioctl_candriver()` this number needs to be translated into the correct file descriptor, which is then used for the `ioctl` call. This modification of the CAN API is implemented, and used to control a total of four PCI CAN cards.

Return Code:	Description:
0	No new event.
1	Standard data frame received.
2	Standard remote frame received.
3	Transmission of a standard data frame is confirmed.
4	Overrun of the remote transmit FIFO. (Not used in FIFO mode).
5	Change of bus status.
8	Transmission of a standard remote frame is confirmed.
9	Extended data frame received.
10	Transmission of an extended data frame is confirmed.
11	Transmission of an extended remote frame is confirmed.
12	Extended remote frame received.
13,14	Not valid. Only useful with CANcard API.
15	Error frame detected.
-1	Function not successful.
-3	Error accessing DPRAM.
-4	Timeout firmware communication.
-99	Board not initialised: INIPC_intialize_board() is not yet called or a INIPC_close_board() is done.

Table C.1: *Function return codes of `CANPC_read_ac()`. [Softing, 2004]*

Parameter:	Valid for return code:	Description:
`Ident`	1,2,3,8,9,10,11,12	Identifier CAN frame.
`DataLength`	1,3,9,10	Number of data bytes in sent or received CAN frame.
`RCV_fifo_lost_msg`	1,2,8,9,11,12	Number of lost frames in receive FIFO.
`RCV_data`	1,9	Data bytes of the received CAN frame.
`Bus_state`	5	Change of bus state.
`Can`	1,2,3,4,5,8,9,10,11,12,15	The CAN channel where the return code event occurred.
`Time`	1,2,3,8,9,10,11,12	Time stamp of the event with a resolution of 1 μs.

Table C.2: *Relevant parameters in the `param_struct` when running in "FIFO Mode". [Softing, 2004]*

C.3 Hardware Tests

During the development of the test bed, a number of tests were made on the CAN cards. This section documents the purpose, procedure, and results of these tests, as well as the conclusions drawn from the tests.

C.3.1 Driver Test

The standard CAN card package provides the PCI card, a users guide and a floppy disk with a Windows driver. The Linux driver can be downloaded from the vendors homepage. To ensure compatibility with the chosen test bed software architecture, the driver is downloaded and tested before the design is started. The provided Linux driver is loaded, and the test software started. The driver probes the card automatically, and then prompts the user to select operational mode according to the descriptions in section C.1.1. When initialised, the card responds to the commands shown on figure C.3.

When selecting "Static Object Buffer Mode" or "Dynamic Object Buffer Mode" every thing works fine. However, when using "FIFO Mode" the driver only works when the test program is configured to poll for incoming frames, and not use interrupt. An extensive amount debugging shows, that interrupt is not enabled correctly by the driver. The reason for this is found to be an error in the device driver from Softing. The driver includes a function called `CanEnableIrq`, where the lines from 246 and onwards contains the following:

```
if (!bDoEnable)
  {                              /* start device */
    ulPortData |= ENAB_IRQ_PC;
  }
else
  {                              /* halt device */
    ulPortData &= ~ENAB_IRQ_PC;
  }
```

Figure C.4: *The source code lines 246-253 of* `can_pci.c`.

With `bDoEnable` defined as `0 = disable interrupt, 1 = enable interrupt`. This code does not operate as intended, because the if statement checks on `!bDoEnable`. The exclamation mark causes the error, hence this is removed and the test is re-run. After this correction the test software works in "FIFO Mode" with interrupt enabled. Softing has been informed of the error, and has confirmed that they have made the error, and corrected it in their future releases.

The structure of the device driver code is rather complex, because it is a driver made by porting a CAN-ACX-104 windows driver to Linux. The driver supports a number of different card types on platforms such as PCI, ISA, PCMCIA, USB and PC104. Furthermore, the software is written to support Dos-, Win32-, and Linux based operating systems. Altogether this leads to an unnecessary complex driver, that should be rewritten if used in critical implementations. These facts are, probably unintentionally, confirmed by Softing, through the header greeting the user is met by when viewing the program source:

```
The source code is written to be usable independent of the
applied interface type.

Good luck running it !
```

Figure C.5: *The greeting users are presented for when viewing the source code.*

C.3.2 Multiple Cards Test

When receiving the two new CAN cards, approximately two weeks before handing in this thesis, a compatibility test is made. The purpose of the test is to determine, whether the CAN driver and library supports the use of multiple cards in one machine.

The first test is to run the implemented test bed software with all four cards inserted. This software contains four threads each accessing one card through the common CAN library. A simple test is generated, where all simulated subsystems are configured to send a unique identifier upon reception of a certain identifier. This identifier is then transmitted on the CAN bus, and the replies from the subsystems are registered. The test shows that all interfaces are working as expected.

The next test is to configure the subsystems to send identical identifiers and identical data bytes when receiving a certain identifier. This test also succeeds as expected.

The third test is to configure the subsystems to send identical identifiers, but unique data bytes when receiving a certain identifier. Running this test twice causes one or two CAN cards to stop responding and generating interrupts. This can be caused by different reasons, and is therefore investigated further by evaluating the bus status of the CAN bus and the CAN library's abilities to handle multithreaded processes.

Bus Status Testing

When running the third test, a lot of error frames are detected. This phaenomenon implies that the error counters are increasing their values, which may lead to a change of error state. To monitor the error state, the test bed is configured to evaluate the `Bus_state` value every time the `CANPC_read_ac()` function returns five, to indicate a change of bus state. The actual value of the error counters is not available to the application.

Re-running the test shows that the bus status of the cards that stop responding changes to "Bus Off" during the test. This shows that the fact that the cards stop responding and generating interrupts is a healthy sign, that indicates that the CAN cards operate as intended and according to the CAN 2.0B standard. The reason for the error frames is found, when a bit difference between two frames transmitted at the same time, causes one transmitter to back off. If this happens outside the start bit, the arbitration field, and the ACK slot, it is treated as a bit error. This process is described in section B.3.2 and B.3.3. This fact explains why error frames are not generated, when the subsystems are configured for sending their replys with unique identifiers. This does not cause error frames because the identifier is a part of the arbitration field, as explained in section B.2. On the other hand, the third test does cause error frames, because the bit differences occur in the data field of a frame.

Thread Proof Test

Another issue that could cause errors or malfunctions is that the test bed design uses multiple threads for accessing the device drivers through the CAN library. The test bed software structure can ensure that this is reliable while communication goes on in the upper layers, and Linux' way of handling the device drivers can assure that hazard issues does not occur at the lowest layer. But in the intermediate layers, this responsibility belongs to the CAN library and Softings device drivers.

Softing does not provide any documentation on whether their software is thread proof or not. Thread proof software is software that remains stable, when accessed by multiple software threads that do not belong to separate programs. Even though the test software provided by Softing actually does use threads, but only one thread, it does not state whether this procedure can be taken as stable and reliable. Therefore a test is made, to determine whether the library and driver software behaves different, when accessed by multiple threads from one program, and when accessed by multiple programs. To perform this test, the test bed software is re-implemented in such a way that each card is handled by a separate program. The three tests described in section C.3.2 is then repeated, and the output is monitored. The tests does not proof any difference in performance or reliability, when running the test bed either as multiple "heavy" threads or multiple "light" threads. However, when studying the source code of the library and particularly the driver, it does not seem like hazard avoidance and thread proofing has been given much thought. This is an area that could be interesting to investigate further, but is omitted due to the time available for this thesis.

C.4 Port Usage

A number of functions in the CAN card API operates card wise. This means that certain operations can not be performed on one port of a card, and not the other. This is the case with the `CANPC_read_ac()` function. This function retrieves the frame and returns a value that indicates what type of frame or event that occurred. By evaluating this output, the transmission of e.g. an extended data frame on the given card number can be determined, but it is not possible to determine on which port the transmission occurred. As an example, consider that the test bed engine, which logs the traffic, is running on port one of card one, and subsystem one is running on port two of card one. A second subsystem, subsystem two, is running on port one of card two. In this situation, the return code of `CANPC_read_ac()` seen by the test bed engine is 9 when subsystem two is transmitting an extended data frame, but 10 when subsystem one is transmitting the same type of frame. An extended frame transmission from the test bed engine itself would also appear as a return code 10. This means that the test bed can not distinguish between frames sent by the test bed engine on port one, and frames sent by a subsystem on port two.

To determine if a given test has passed or failed, a log containing all traffic on the CAN bus during the test must be evaluated. The evaluation checks whether the specified input frames are followed by the expected output frame within a given interval. But in order to do this correct, then the frames sent by the test bed engine during the test, should not be used in the

verification, meaning that test input frame number one must not be verified by another test input frame sent some time later. Therefore the test bed engine needs to be able to distinguish between frames sent from test bed engine and frames sent from subsystems. This implies that a subsystem can not be sharing CAN card with the test bed engine.

The CAN monitor and the test bed engine could share CAN card, but use separate ports. However, this is not a good solution, because running these to modules on separate CAN ports would mean that the time stamps would not be identical on the WIU and the CMU. Therefore these two modules need to do their CAN communication through the same port.

The solution is to reserve card one for the test bed engine only. This means that port two of card one is not used.

Database Design

APPENDIX D

This appendix contains a description of the database design used in this thesis. All database tables are shown, as well as the data type and a description of each field in the TestBed database.

The database design is repeated in figure D.1 for convenience.

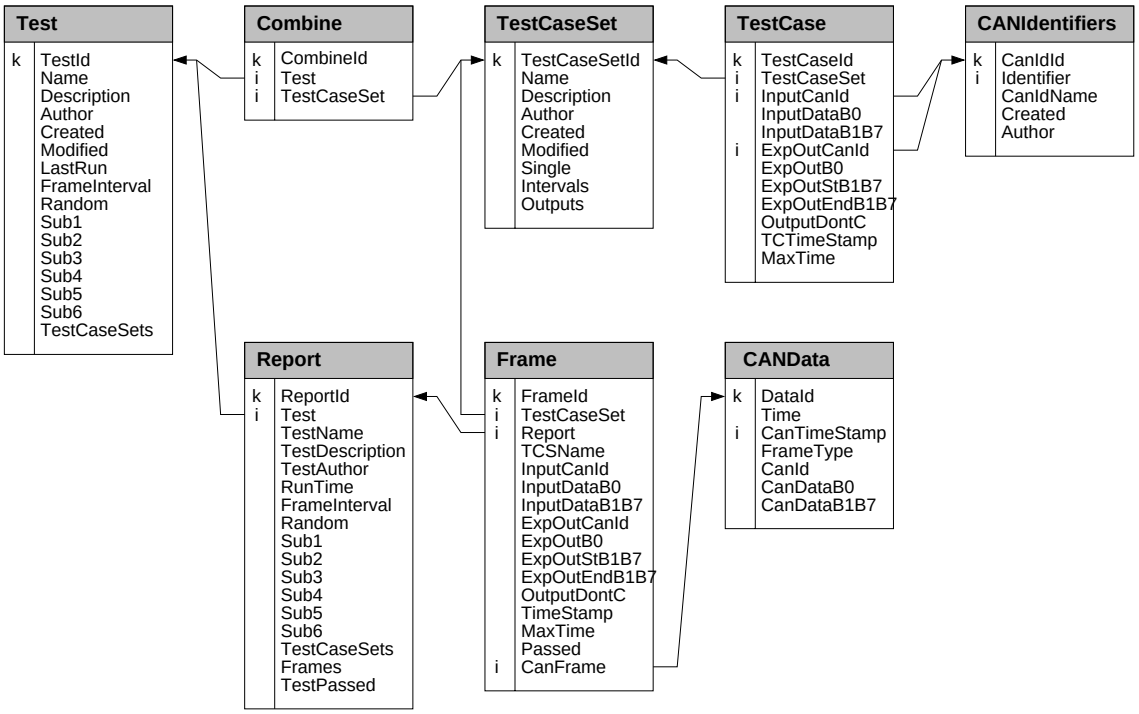


Figure D.1: *The TestBed database design. k denotes a primary key, and i denotes an index.*

The tables are described with their data type and a description in the following. A "k" denotes a primary key, and an "i" denotes an index.

Database structure on CD-ROM:
A phpMyAdmin-2.5.6 dump of the TestBed database structure, is found on the enclosed CD-ROM.

Test			
k	TestId	uns. int	Unique id and primary key for Test table.
	Name	varchar	The name of the test.
	Description	text	A short description of the test.
	Author	varchar	The AAU user name of the creator of the test.
	Created	datetime	Date and time for test creation.
	Modified	datetime	Date and time for last test modification.
	LastRun	datetime	Date and time for last run of the test.
	FrameInterval	int	The interval time of frame sending.
	Random	tinyint	Denotes whether frames are sent random or sequential.
	Sub1	tinyint	Denotes whether or not the subsystem driver 1 is used.
	Sub2	tinyint	Denotes whether or not the subsystem driver 2 is used.
	Sub3	tinyint	Denotes whether or not the subsystem driver 3 is used.
	Sub4	tinyint	Denotes whether or not the subsystem driver 4 is used.
	Sub5	tinyint	Denotes whether or not the subsystem driver 5 is used.
	Sub6	tinyint	Denotes whether or not the subsystem driver 6 is used.
	TestCaseSets	int	The number of test case sets.

Table D.1: *Test table.*

Combine			
k	CombineId	uns. int	Unique id and primary key for Combine table.
i	Test	uns. int	Refer to TestId in Test table.
i	TestCaseSet	uns. int	Refer to TestCaseSetId in TestCaseSet table.

Table D.2: *Combine table.*

TestCaseSet			
k	TestCaseSetId	uns. int	Unique id and primary key for TestCaseSet table.
	Name	varchar	The name of the test case set.
	Description	text	A short description of the test case set.
	Author	varchar	The AAU user name of the creator of the test case set.
	Created	datetime	Date and time for test case set creation.
	Modified	datetime	Date and time for last test case set modification.
	Single	tinyint	Denotes whether the test case set contains a single frame.
	Intervals	int	The number of intervals used for this test case set.
	Outputs	int	The number of outputs used for this test case set.

Table D.3: *TestCaseSet table.*

TestCase			
k	TestCaseId	uns. int	Unique id and primary key for TestCase table.
i	TestCaseSet	uns. int	Refer to TestCaseSetId in TestCaseSet table.
i	InputCanId	uns. int	The input CAN frame identifier.
	InputDataB0	int	The B0 byte in the CAN frame to be send.
	InputDataB1B7	bigint	The B1-B7 bytes in the CAN frame to be send.
i	ExpOutCanId	uns. int	The expected output CAN frame identifier.
	ExpOutB0	int	The expected output B0 byte.
	ExpOutStartB1B7	bigint	The start interval of the expected output in B1-B7.
	ExpOutEndB1B7	bigint	The end interval of the expected output in B1-B7.
	OutputDontC	tinyint	Denotes whether an output is a don't care or not.
	TCTimeStamp	uns. bigint	64 bit time stamp of the time of frame sending.
	MaxTime	int	The maximum time before an output is expected.

Table D.4: *TestCase table.*

CANIdentifiers			
k	CanIdId	uns. int	Unique id and primary key for CANIdentifier table.
i	Identifier	uns. int	The CAN identifier.
	CanIdName	varchar	Name of the CAN identifier.
	Created	datetime	Date and time for CAN identifier creation.
	Author	varchar	The AAU user name of the creator of the CAN identifier.

Table D.5: *CANIdentifier table.*

Report			
k	ReportId	uns. int	Unique id and primary key for Test table.
i	Test	uns. int	Refer to the test from which the report is created.
	TestName	varchar	The name of the test.
	TestDescription	text	The short description of the test.
	TestAuthor	varchar	The AAU user name of the creator of the test.
	RunTime	datetime	Date and time for the report creation.
	FrameInterval	int	The interval time of frame sending.
	Random	tinyint	Denotes whether frames are send random or sequential.
	Sub1	tinyint	Denotes whether or not the subsystem driver 1 is used.
	Sub2	tinyint	Denotes whether or not the subsystem driver 2 is used.
	Sub3	tinyint	Denotes whether or not the subsystem driver 3 is used.
	Sub4	tinyint	Denotes whether or not the subsystem driver 4 is used.
	Sub5	tinyint	Denotes whether or not the subsystem driver 5 is used.
	Sub6	tinyint	Denotes whether or not the subsystem driver 6 is used.
	TestCaseSets	int	The number of test case sets.
	Frames	uns. int	The number of frames in the test.
	TestPassed	tinyint	Denotes whether the test passed or not.

Table D.6: *Report table.*

Frame			
k	FrameId	uns. int	Unique id and primary key for Frame table.
i	Report	uns. int	Refer to ReportId in Report table.
i	TestCaseSet	uns. int	Refer to TestCaseSetid in TestCaseSet table.
	TCSName	varchar	The name of the test case set.
	InputCanId	uns. int	The input CAN frame identifier.
	InputDataB0	int	The B0 byte in the CAN frame to be send.
	InputDataB1B7	bigint	The B1-B7 bytes in the CAN frame to be send.
	ExpOutCanId	uns. int	The expected output CAN frame identifier.
	ExpOutB0	int	The expected output B0 byte.
	ExpOutStartB1B7	bigint	The start interval of the expected output.
	ExpOutEndB1B7	bigint	The end interval of the expected output.
	OutputDontC	tinyint	Denotes whether an output matters or not.
	TimeStamp	uns. bigint	64 bit time stamp of the time of sending.
	MaxTime	int	The maximum time before an output is expected.
	Passed	tinyint	Denotes whether the test case passed or not.
	CanFrame	uns. int	Refer to the CAN data frame to validate the test case.

Table D.7: *Frame table.*

CANData			
k	DataId	uns. int	Unique id and primary key for CANData table.
	Time	datetime	Data and time for received frame.
	CanTimeStamp	uns. bigint	64 bit time stamp.
	FrameType	int	Determines the type of the frame.
i	CanId	uns. int	The CAN frame identifier.
	CanDataB0	int	The first data byte of the CAN frame.
	CanDataB1B7	bigint	The last seven data bytes of the CAN frame.

Table D.8: *CANData table.*

Trolltech Qt & Qt Designer

Appendix E

This appendix explains how to develop graphical user interfaces, using programming tools provided from Trolltech. The tools discussed in the following are the C++ library Qt, and the graphical design tool Qt Designer. Qt provides some special programming methods for connecting C++ classes, which is explained as well. The Qt tool uses GNU g++ for compiling the object oriented source code. This appendix is based on [Trolltech, 2004].

E.1 Introduction to Qt

Qt is a multi platform toolkit for development of applications with a GUI using C++. This means that the same source code can be compiled and work in the same way for operating systems as Linux, HP-UX, Mac OS X, Solaris, Microsoft Windows, etc.. Qt offers a range of object oriented classes, that provides functions, that are presented graphically on the users screen.

E.1.1 Inter-Object Communication

In almost every GUI project, buttons, scroll lists, menu bars, or tables, has to communicate when an object is selected or pushed.

Typical methods to achieve this kind of communication are by using callbacks. A callback is a pointer to a function that is called when the processing function wants to notify the callback function. By this, the processing function passes the callback pointer to the function when an event has happened.

Qt introduces another method for object communication. This method is a signal/slot mechanism which is sketched in figure E.1.

The signal/slot mechanism is a central feature of Qt, and is considered the most significant strength of Qt compared to other toolkits.

To communicate between two objects in Qt, the signal/slot method is used: A signal is emitted when a particular event occurs to an object e.g. a button is pressed. The receiving function has a slot which receives the signal. Through the signal/slot mechanism, parameters can be passed.

To establish communication between two objects the `connect()` function is used. As figure E.1 shows, when Object1 wants to call Object2, a signal from Object1 has to connect to one of the slots of Object2. This is carried out by `connect(Object1, signal1, Object2, slot1)`.

The signal/slot mechanism is inherited from the Qt QObject class, which both the sending and the receiving object has to be a subclass of.

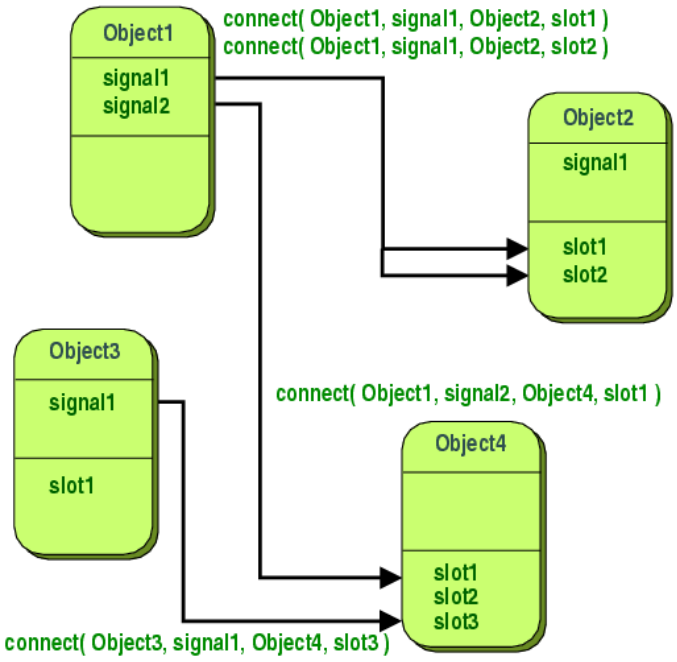


Figure E.1: *Connection between Qt's signal and slots. [Trolltech, 2004]*

E.1.2 Qt Designer

Qt Designer is a visual form designer for editing files in the .ui format. The .ui files contains XML descriptions of graphical GUI elements. The .ui file has to be compiled by a uic (user interface compiler) into C++ source code, for further treatment.

From the .ui file two C++ files are generated, namely a header file form.h and an implementation file form.cpp. These files contains all functionality and the graphical interface generated from Qt Designer. A GUI program is created and destroyed by an QApplication object which is started in the main.cpp file. The main includes form.h which will show the GUI designed from the XML .ui file.

To create more complex GUI programs, additional functionality can be added to the the GUI implementation by the developer. These types of implementation are functions and signal/slot connections. Because the form.h and form.cpp are generated from the .ui file, it is not appropriate to implement additional functionality into these files. This is because the developed functionality will be overwritten when the uic recompiles the developed system.

To overcome this problem, Qt Designer reads and writes another file, namely a form.ui.h. The .ui.h file is an ordinary C++ source file that contains implementations of custom slots and functions. The file gets included from the generated form implementation file form.cpp.

No matter what type of code that is implemented in the .ui.h file, Qt Designer has to be synchronised with the signal/slot implementation. Otherwise it is not possible to emit signals to slots that are implemented in the .ui generated files. To accomplish this task, Qt Designer adds stubs to the .ui.h file, for each C++ function, signal and slot the user includes in a project.

An overview of generated Qt Designer files, and their relations are given in figure E.2.

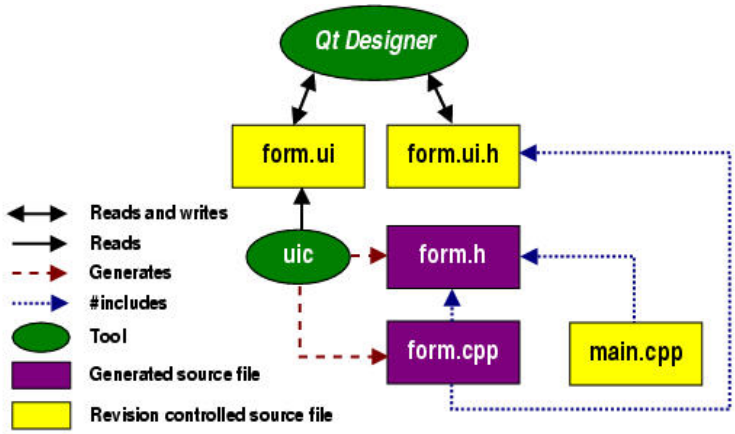


Figure E.2: *Overview of files generated by Qt Designer. [Trolltech, 2004]*

One disadvantage of using Qt Designer is, that it is not possible to directly merge software developed by traditional object oriented methods, such as the OOAD presented by [Mathiasen et al., 2000]. This is the case, because every custom implemented function or slot, is being forced to become a subclass of the form.cpp file, which is not a custom developed file.

Despite this detail, many positive things are also achieved by using Qt Designer.

Qt Designer offers a graphical user interface, where drag and drop methods are used when designing and developing custom user interfaces. Qt Designer includes an advanced source code editor. The editor is able to help the designer, by giving a list of known functions and methods to each created object that is associated to a standard Qt class. This makes it easier to find useful information about functions that are included in the used Qt classes.

Qt Designer is an intuitive design tool, it is a freeware program. The uic compiler and the Qt classes are both free to use under the terms of GPL for development of non commercial products.

Linux IPC Message Queues

APPENDIX F

The following appendix deals with information on how to transfer data between processes in the Linux operating system, using Inter Process Communication (IPC). This appendix is based on [Bovet and Cesati, 2001].

F.1 Introduction to IPC

The IPC structure was created in the early UNIX days, and has since been adopted to most UNIX systems and UNIX variants as Linux, Mac OS X, etc. which supports AT&T's System V.

IPC is a shared resource that make calls for messages, semaphores, and shared memory. The IPC resource is created in memory and remains persistent until the process, which has created the IPC, releases the IPC resource. Each created IPC resource is unique, and generated by the kernel from a 32 bit identifier. To access the shared IPC resource by other processes, the IPC has to be identified by a an equivalent unique 32 bit identifer, which is generated by a user defined key. When two processes wants to communicate through an IPC resource, both has to refer to the same unique IPC identifier.

F.2 IPC Message Queue

This project uses IPC message queues for parsing data between the TBE and the CMU. To establish the unique IPC identifier, the system function `msgget` is used. For the kernel to return a generated message identifier, the arguments are applied like shown in this function call: `msgget(keyval,IPC_CREAT | 0660)`, where the `keyval` are the user defined key. `IPC_CREAT` is a flag that tells the kernel to create a new unique identifier on basis of the `keyval`, with the privilege masked by `0660`. The `msgget` returns the 32 bit identifier, which is used when processes sends and receive messages over the IPC message queue.

Each message generated by a process is sent to the IPC message queue where it stays until another process reads the message. The message queue is constructed in means of a linked list. New messages are put in at the end of the linked list, but can be pulled out at any order.

The message is composed of a fixed sized header, and the data content can be of a variable size. The data can be presented as an array or a structure.

In order to send the message to the queue, the function `msgsnd()` is called. This takes arguments for the given message identifier, a pointer to the data message, the length of the

message, and a message flag `msgflg`, which indicates what to do if the message somehow can not be sent.

The data structure has to include a field for `long mtype` which tells the receiving process in which order the data message has to be read.

The receiving process has to invoke the `msgrcv()` function to receive an element from the queue. Parameters that are passed to the receiving function are the IPC identifier to the queue, a pointer to where the `msgrcv()` should store the received message, including the length of the message. A value which specifies what message should be retreived. This value has to be the same as indicated in `msgsnd()` as a `long mtype`. The `msgrcv()` also has a message flag `msgflag`, which is used to control the receiving function: whether the `msgsnd` should block until it has received a message, poll the queue and return nothing if no message are presented or receive whatever message might be in the queue regardless the size is incorrect.

Test Bed Timing

Appendix G

This appendix contains an experiment with the purpose of determining whether standard Linux or Linux with RTAI support should be used to control the timing behaviour of the test bed. The experiment is explained and the results are presented. Based on these results the method to be used by the test bed is chosen.

G.1 Test Bed Timing Requirements

The timing requirements of the test bed are not easily determined, since the AAUSAT-II documentation does not provide any requirements, regarding the timing on the internal satellite communication network, that can be inherited to the test bed. However, in order to obtain reliable measurements, where conclusions are based on time stamps taken at the appearance of certain conditions, some certainty of the timing behaviour is desirable.

Using standard Linux kernels does not provide any hard real-time guarantees of certain processes always keeping their specified deadlines. To obtain this, a real-time add-on such as RTAI (www.rtai.org) should be applied. RTAI patches the Linux kernel with, among others, an Interrupt Dispatcher, meaning that RTAI takes over the interrupt handling by using a "Real-Time Hardware Abstraction Layer" (RTHAL) that runs the standard Linux process as an idle task. However, to obtain extensive hard real-time performance, this would imply that the device drivers of the CAN adapters should be ported to RTAI. This is a rather time consuming task, and considered less beneficial compared to the results gained when considering the purpose of the test bed.

Instead of converting the entire test bed to a hard real-time platform, RTAI provides possibility of implementing soft real-time in Linux userspace. This can be obtained by patching a Linux kernel with RTAI, or more specific RTHAL support and compiling RTAI with a subsystem called LXRT. LXRT is RTAI's option for providing soft real-time performance with a less risky and faster development process than conventional RTAI.[1] The result is said to be precise average timing performance, compared to plain Linux.

When running tests, the most important timing requirement that can be affected by the performance differences between a userspace soft real-time system and soft real-time in plain Linux, is whether the timing requirements of the pauses between the frame transmissions are met.

[1] Hard real-time is also possible using LXRT, but this would also require that the CAN drivers are rewritten, since standard Linux system calls such as ioctl are not allowed.

Other factors such as the latency of servicing interrupts on the CAN cards is not considered to be affected, unless a hard real-time kernel is applied along with a set of ported CAN drivers.

In order to determine whether RTAI's LXRT should be preferred over plain Linux, for controlling the timing between test bed frame transmissions, an experiments is made.

G.2 Timing Experiment

The typical way of pausing a process in plain Linux is to use `usleep()` to suspend the process for a number of microseconds. LXRT's counterpart of this function is `rt_sleep()`.[2]

The experiment consists of measuring the actual time spent while suspending a task for one second, using both `usleep()` and `rt_sleep()`. The elapsed time is measured using `gettimeofday()` which returns a `timeval` structure containing a 32 bit variable of the time in seconds since epoch, and another 32 bit value of the microseconds remaining. By sampling these values before and after the sleep commands, and subtracting the results, the time slept can be measured. The principal behind the code used to perform the test is shown in figure G.1.

```
/* Measure sleep time using RTAI LXRT */
gettimeofday(&now,NULL);
rt_sleep(One second);
gettimeofday(&now2,NULL);

/* Calculate time slept */

/* Measure sleep time using plain Linux */
gettimeofday(&now,NULL);
usleep(One second);
gettimeofday(&now2,NULL);

/* Calculate time slept */
```

Figure G.1: *Pseudocode of the routine for measuring the accuracy of the sleep functions.*

This process is iterated 100 times, and the results are stored in two data files. These files are processed using MATLAB and the curve of figure G.2 is generated.

Figure G.2 shows that the RTAI LXRT method is more accurate than the the plain Linux method. This behaviour is expected, and it is likely that this result will be more obvious the higher the system load on the CPU gets. Table G.1 shows a selection of statistical indicators derived from the sampled data.

[2] The function `rt_busysleep()` also has similarities, but this process suspends ALL running processes, and is therefore quite impractical to use.

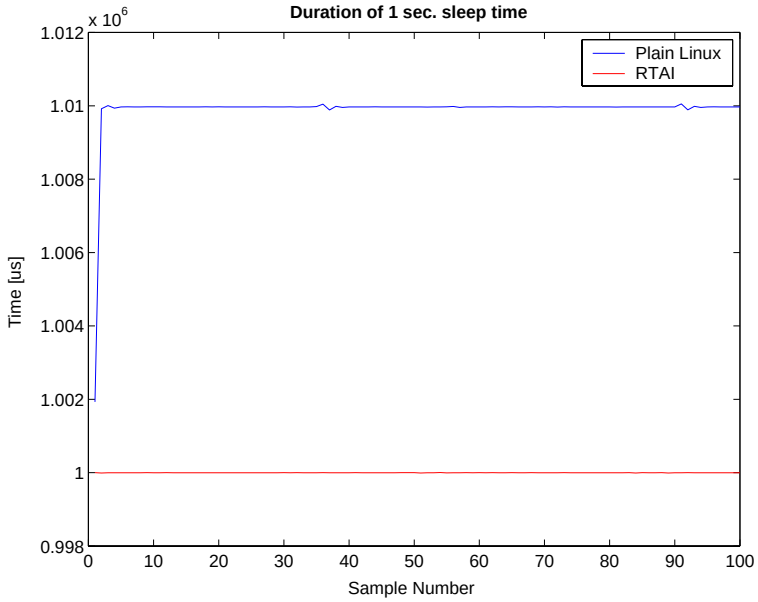


Figure G.2: *Measurement of the actual time spent in sleep mode when a sleep of one second is implemented using standard Linux functions and RTAI functions.*

System:	Mean $\bar{x}$:	Minimum:	Maximum:	STD:	Sample size:
RTAI LXRT:	1000.00 ms	999.99 ms	1000.00 ms	1.80E-3 ms	100
Linux:	1009.90 ms	1001.93 ms	1010.05 ms	0.80 ms	100

Table G.1: *Statistical indicators derived from the sampled data.*

G.3 Conclusion

The experiment shows that a more accurate timing is obtained by letting RTAI LXRT handle timing. The standard deviation of LXRT is much less than using plain Linux. Furthermore, the possibility of using some of RTAI's functionality may become beneficial to the subsystem developers during tests with subsystems simulated in the test bed. It is therefore decided to use RTAI LXRT for handling the critical timing of the test bed implementation.

Test Files on CD-ROM:
The C-program, MATLAB script and data files of the test can be found on the enclosed CD-ROM.

Nomenclature

The nomenclature is sorted alphabetically for symbols and letters respectively.

AAU	: Aalborg University.
ACK	: Acknowledge bit stating correct reception of CAN bus frame.
ADCS	: Attitude Determination and Control System.
CAN	: Control Area Network.
CMU	: Can Monitor Unit - the module used to monitor the traffic on the CAN bus and send spontaneous frames.
COM	: Communication System.
CRC	: CRC checksum used for error detection of CAN bus frame.
CSMA/CD+AMP	: Carrier Sence Multiple Access with Collision Detection and Arbitration on Message Priority - the scheme used by CAN bus .
DLC	: Data Length Code - the length of a CAN bus data field.
DSRI	: Danish Space Research Institute.
ECTC	: Equivalence Class Test Case - a set of frames used to perform equivalence class testing over a given number of intervals.
EOF	: End Of Frame - marks the end of a CAN bus frame.
EPS	: Electronic Power Supply.
ERD	: Entity-Relationship Diagram.
GRB	: Gamma Ray Burst.
GRBD	: Gamma Ray Burst Detector.
IDE	: IDentifier Extension - used to determine between standard- and extended CAN bus identifier.
IFS	: Inter Frame Spacing - the space between CAN bus frames.
INSANE	: INternal Satellite Area NEtwork.
IPC	: Inter Process Communication.

ITC	: Inter Thread Communication.
OBC	: On-Board Computer.
PL	: PayLoad.
RTR	: Remote Transmission Request - request a transmission from a remote CAN bus host.
SFTC	: Single Frame Test Cases - a frames used to perform tests involving a single input frame and a single output frame.
SRR	: Substitute Remote Request Bit - replaces the RTR bit from the standard CAN bus frame, when using extended CAN bus frame.
TBE	: Test Bed Engine - the C module that handles communication with CAN cards.
WIU	: Web Interface Unit - the module that allows the user to configure the test bed.

CD-ROM

Appendix I

Throughout the thesis, a number of references are given to the enclosed CD-ROM. The CD menu is build in HTML, by using the design of the web interface unit on the test bed. The CD contains an autorun facility, so when it is put into a computer, the standard browser opens, and the HTML menu is shown.

If the autorun feature is disabled on the computer, or it does not open the menu page, please open the /index.html file from a browser.

From the menu at the first page, all stuff added to the CD is available. Below a list of the contents is found.

I.1 CD-ROM Contents

The contents of the CD-ROM are:

- The INSANE specifications.
- The test bed source code.
- Doxygen source code documentation files.
- The database structure.
- The test reports.
- Timing test files.
- This master's thesis.
- Screen shots of the CAN monitor.
- A small demo of the web interface unit