Juho Lepistö

Embedded Software Testing Methods

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Electronics
Thesis
6th March 2012

Author	Juho Lepistö
Title	Embedded Software Testing Methods
Number of Pages	53 pages + 10 appendices
Date	2nd November 2011
Degree	Bachelor of Engineering
Degree Programme	Electronics Engineering
Specialisation option	
Instructors	Pasi Lauronen, M.Sc Janne Mäntykoski, Lecturer

This thesis was carried out at Efore Group Product Development department, Espoo. The aim of the thesis was to develop and tailor embedded software testing process and methods and develop customised testing tool.

Traditional software testing methods were studied to familiarise oneself with the basic concepts of software testing. Several software testing methods were studied to map options for exploiting existing methods in developing software testing method for low-level embedded software environment.

The customised testing method was built around Test Maturity Model integration TMMi model to ensure integration of the software testing practices to the existing software development process and pave the way for continuous improvement of software testing. A spiral model was selected to be the main process control structure due to its iterative and test intense nature. The concrete testing practice incorporated into the customised method was based on agile Test Driven Development TDD method to shorten defect lifecycles and set emphasis on developer oriented software testing. From these elements a tailored software testing method was formed.

In order to perform software testing in early stages of project, a hardware software testing platform was designed. The platform was able to simulate analogue, digital and PWM signals to enable testing in simulated target environment before actual hardware is available. The testing platform could be fully automated by test scripts. The platform gathered test log from inputs and formatted the data for further processing and documentation.

The platform prototype was tested on one actual project and proven to be functional and suitable for effective early-stage testing.

Keywords	software testing, embedded systems, Efore

Metropolia

Tiivistelmä

Tekijä Otsikko Sivumäärä Aika	Juho Lepistö Embedded Software Testing Methods 53 sivua + 10 liitettä 2. marraskuuta 2011
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Elektroniikka
Suuntautumisvaihtoehto	
Ohjaajat	Pasi Lauronen, DI Janne Mäntykoski, lehtori

Insinöörityö tehtiin Efore Oyj:n tuotekehitysosastolle Espooseen. Insinöörityön tavoitteena oli kehittää ja räätälöidä prosessi ja menetelmät sulautettujen järjestelmien software-testaukseen sekä kehittää työkalu testauksen suorittamiseksi.

Työssä tutkittiin perinteisiä software-testauksen menetelmiä ja testauskonsepteja. Lisäksi tutkittiin useita software-testausmetodeja ja kartoitettiin mahdollisuutta olemassa olevien metodien hyödyntämiseen tilaajayrityksen sulautettujen järjestelmien matalan tason software testauksen kehittämisessä.

Räätälöity testausmetodi rakennettiin Test Maturity Model integration TMMi –mallin ympärille, jotta voitiin varmistaa testauskäytänteiden integroituminen softwaren kehitysprosessiin ja mahdollistaa testausmetodien jatkuva kehittäminen. Pääasialliseksi prosessin ohjaus- ja kontrollimetodiksi valittiin spiraalimalli iteratiivisen ja testausorientoituneen luonteensa vuoksi. Kehitetyssä metodissa konkreettinen testauksen toteutus pohjautuu joustavaan Test Driven Development TDD –metodiin, jotta vikojen elinikä voitiin minimoida ja painottaa testaus kehittäjälähtöiseksi. Näistä elementeistä koottiin räätälöity testausmenetelmä Eforen käyttöön.

Jotta software-testaus olisi ollut mahdollista aloittaa varhaisessa vaiheessa, työssä kehitettiin software-testausalusta. Alusta pystyi tuottamaan analogisia ja digitaalisia signaaleja sekä PWM-pulssisignaaleja. Alustalla voitiin simuloidan lopullista laiteympäristöä ja testaus voitiin suorittaa kohdeprosessorissa ennen varsinaisen laitteiston valmistumista. Testausalustan lähdöt voitiin automatisoida täysin skripteillä. Alusta keräsi sisääntuloista dataa ja muotoili kerätyn datan pohjalta lokitiedostoja jatkokäsittelyä ja dokumentointia varten.

Alustan prototyyppiä testattiin meneillään olleen projektin yhteydessä ja testausalustan konsepti todettiin toimivaksi. Laitteisto soveltui tehokkaaseen varhaisessa vaiheessa tapahtuvaan softwaren testaukseen.

Avainsanat	Software-testaus, sulautetut järjestelmät, Efore

Glossary

ADC	Analogue-to-digital converter
BFU	Battery Fuse Unit
CAN	Controller Area Network
CMMI	Capability Maturity Model Integration
DAC	Digital-to-analogue converter
DAQ	Data Acquisition
DUT	Device Under Test
HW	Hardware
I/O	Input/output
LITO	Lifecycle, Techniques, Infrastructure, Organisation
PDM	Product Data Manager
PSP	Personal Software Process
RAM	Random Access Memory
ROM	Read-Only Memory
SPI	Serial Peripheral Interface
SW	Software
TDD	Test-Driven Development
TEmb	Embedded system testing method
TMMi	Test Maturity Model integration
UUT	Unit Under Test

Contents

1 Introduction

Embedded systems have found their way to all areas of electronics, including power electronics where computer control has started to replace analogue circuitry in modern industry-level power supply units. Clients are demanding increasingly sophisticated and more complex control and monitoring features that have created requirement for including microprocessors and embedded systems on power electronics products, thus software has eventually become important part of product development process in power electronics. This has created demand for extensive software testing, verification and validation methods to ensure the operational reliability of the products.

This thesis is made for Efore Group Product Development. Efore Group was founded in Finland in 1975 and in present day Efore is an international technology company focused on custom designed highly embedded power products that require high performance, reliability and efficiency. In product development Efore focus on minimizing the energy consumption by improving energy efficiency, thus contributing to environmental friendliness.

Often comprehensive software testing of embedded systems can be performed only in the later stages of the project when the first prototype hardware is available. This approach has several drawbacks. First, it makes the first prototype prone to software defects and discovering the defects are emphasized on the later stages of product development lifecycle. Secondly, in-depth testing of software is difficult, or even impossible, since the first true opportunity to run the software is in the first prototype round. This can slow down the software development process and hinder quality control measures since the defects cannot be rooted out as early as possible. Thirdly, inadequate software testing methods increase risk of software defects passing through the tests and eventually manifesting themselves in the production or, in the worst case, in the field.

The aim of this thesis is to develop early-phase embedded software testing methods to suit the needs of Efore Product Development department, create a solution for testing of multiple types of 8-bit and 16-bit microcontrollers and I/O configurations, design

hardware implementation of testing environment and test finally the system in practice with a real product. The goal in testing platform design is that it could be used right from the low-level unit testing all the way to testing complete code, since it would be ideal that software testing in hardware environment could be started way before the first prototype in emulated hardware environment. This would move the emphasis of defect discovering and fixing to the early stages of the lifecycle thus improving quality control and defect fixing lead-time.

2 Embedded Systems

2.1 Overview

Term “embedded system” is rather vague. Generally it depicts a computer system that is dedicated to one or few specific tasks offering minimal amount of flexibility. Embedded systems often deal with real-time computing in which the system must react to real-time stimuli. Embedded systems vary from simple interface applications to massive control and monitoring systems. A Venn diagram shown on figure 1 clarifies how embedded systems settle in contrast to other computing systems.

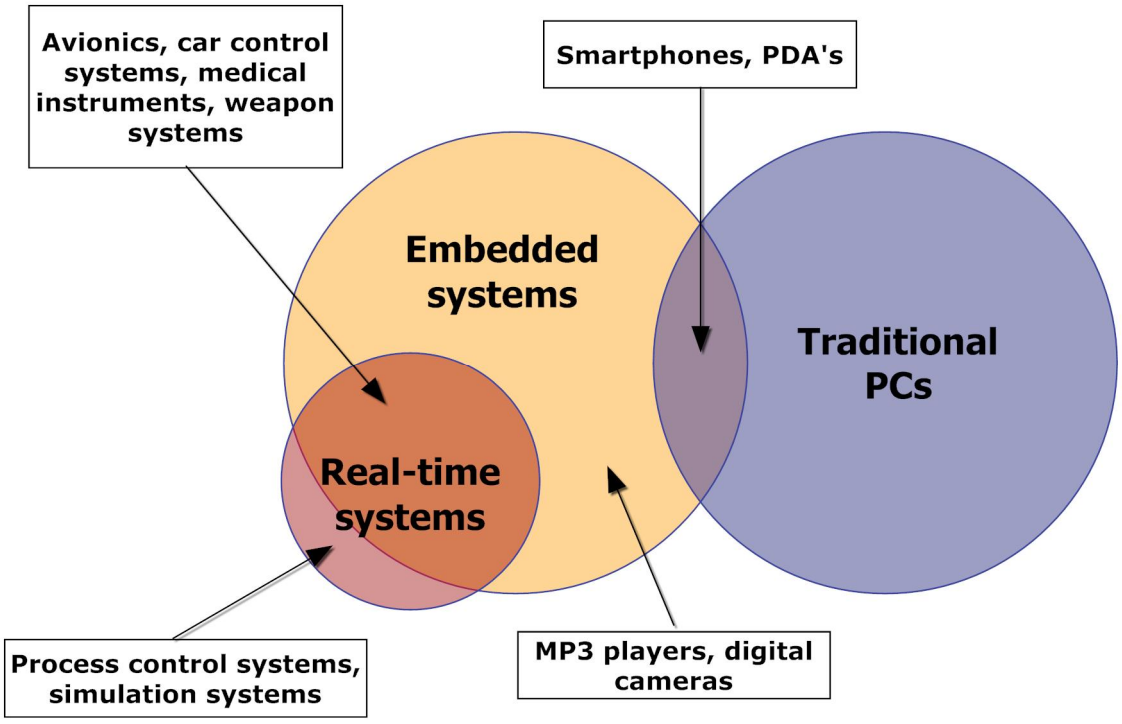


Figure 1. Embedded systems in contrast to other computing systems.

An embedded system can be, for example, an MP3 player, an ECG machine, a microwave oven, a cell phone, a missile tracking system or a telecommunications satellite. The rallying point of all these applications is that they interact with real physical world controlling some application specific hardware that is built-in on the system as shown on figure 2. (1, p. 5)

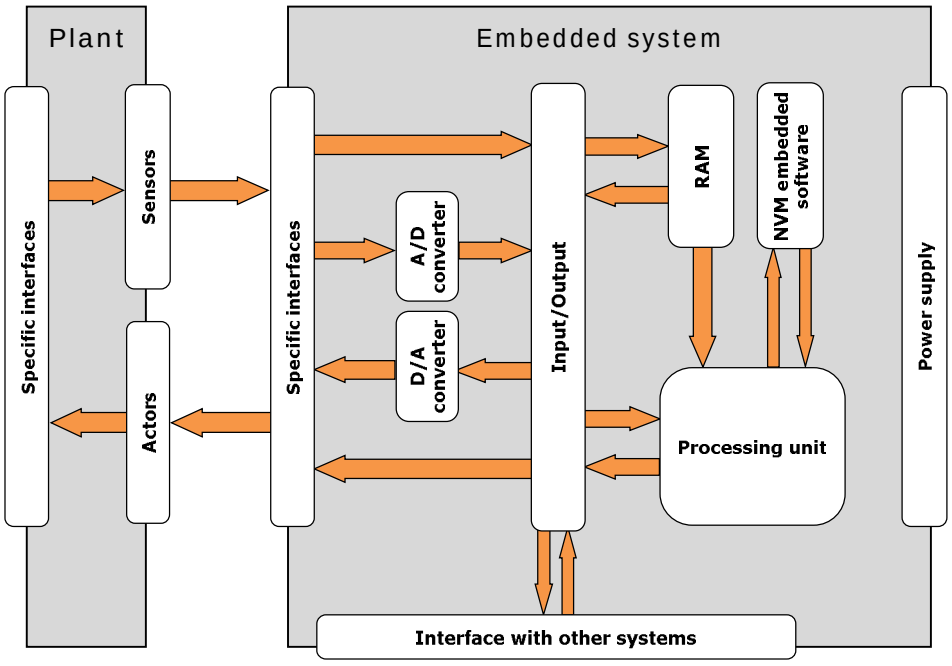


Figure 2. Generic scheme of an embedded system (1, p. 5)

The processing unit is typically a microcontroller or digital signal processor (DSP). Greater scale systems can contain several processing units. The system interacts with the real world in real-time by receiving signals through sensors and sending outputs to actors that manipulate environment. The environment, including the sensors and actors, is often referred to as the plant. Embedded systems include special application specific built-in peripheral devices such as A/D and D/A converters, instrumentation amplifiers, Flash memory circuits, et cetera. The software on embedded system is stored in any kind of non-volatile memory, often ROM or Flash, but the software can also be downloaded via network or a satellite upon start-up. (1, p. 5)

Due to the strict framework in which embedded systems are designed to operate in the size, performance and reliability of the system can be easily optimized in both hardware and software domains. Embedded systems are also easy to mass produce to reduce costs. These are great advantages and they have enabled the invasion of embedded systems into our everyday lives. Embedded systems have become so pervasive that they perform bulk of computation today (2, p. 1).

2.2 Embedded Software

Unlike conventional computer systems that are designed to be as flexible as possible – they can run wide variety of programs and applications and can be extended with additional hardware add-ons – embedded systems are compact computer systems dedicated to operate in certain strictly specified framework running single firmware that is not customizable by the user, offering a little or none flexibility. In addition, the resources, for example the amount of memory and processing power, on embedded systems are often very limited. These differences affect the way software (SW) code is written for embedded systems contra software code for conventional computer systems.

Due to the special hardware (HW) environment the embedded software is written for, one main characteristic of embedded software is that it relies on direct manipulation of the processors registers that control the peripheral functions and I/O of the system. It is usual for the embedded system not to have any kind of standard function library and everything must be written from scratch.

The lack of standard libraries is often due to the fact that in embedded systems the code should be compact and as efficient as possible, since poorly optimized code has heavy impact on the performance on a system where hardware resources are scarce. In the worst case poor optimisation can lead to the software binary not fitting in the system memory at all. The emphasis on optimisation and efficiency is great also because the system is often required to react immediately to real-time stimuli. Thus, in some time-critical embedded systems parts of the code, or even all the code, is written in assembly to ensure that the processing power is used at maximum efficiency.

On embedded systems reliability is also a key issue, since the software is often required to run without problems non-stop for long periods of time. The code efficiency and reliability can be matters of life and death for the whole project. Fortunately embedded software is compiled for a particular processor and particular hardware, which makes code optimisation easier and increases reliability since conflicts between hardware and software can be avoided.

3 Software Testing

3.1 Overview

Increasing share of software in systems has increased the importance of software testing in the product development process and it has become a major factor in quality control measures. In practice it is impossible to create defect-free code (1, p. 3), which inevitably creates a requirement for effective testing process.

Software testing methods and techniques have been developed from the 1970s with major breakthroughs in the 1980s by software testing pioneer Barry Boehm who introduced the spiral model for software testing and studied the economic effects of software testing. During the 1990s software testing had become the basic process of software development companies and nowadays nearly 40 per cent of development costs are spent on software testing and defect removal (3, p. 7). Value of rigorous testing is clearly understood in the software industry. Even so neglected software testing is one of the most common reasons for project not meeting its deadline. In some cases defected software can even lead to project being aborted.

The economic importance of software testing is significant, which drives interest and emphasis in software testing. According to a report on the Economic Impacts of Inadequate Infrastructure for Software Testing published in 2002 by Research Triangle Institute and the National Institute of Standards and Technology (USA) the annual cost incurred as insufficient software testing amounts to $59.5 billion in USA alone (2, p. 184). Failure to test and validate software properly can lead to major financial losses, since the longer the defect lifecycle is, the more expensive it is to fix, as figure 3 illustrates (4).

If software testing and validation is insufficient and a defect injected in the requirements is found not till operational testing, the cost to fix the defect is enormous. The costs can get intolerable if the defect is found in the field, especially if the defect requires the product to be recalled from customers. Thus, it seems intuitive that a successful testing pushes defect lifecycle as short as possible. This requires that the testing process is included in the development process right from the beginning.

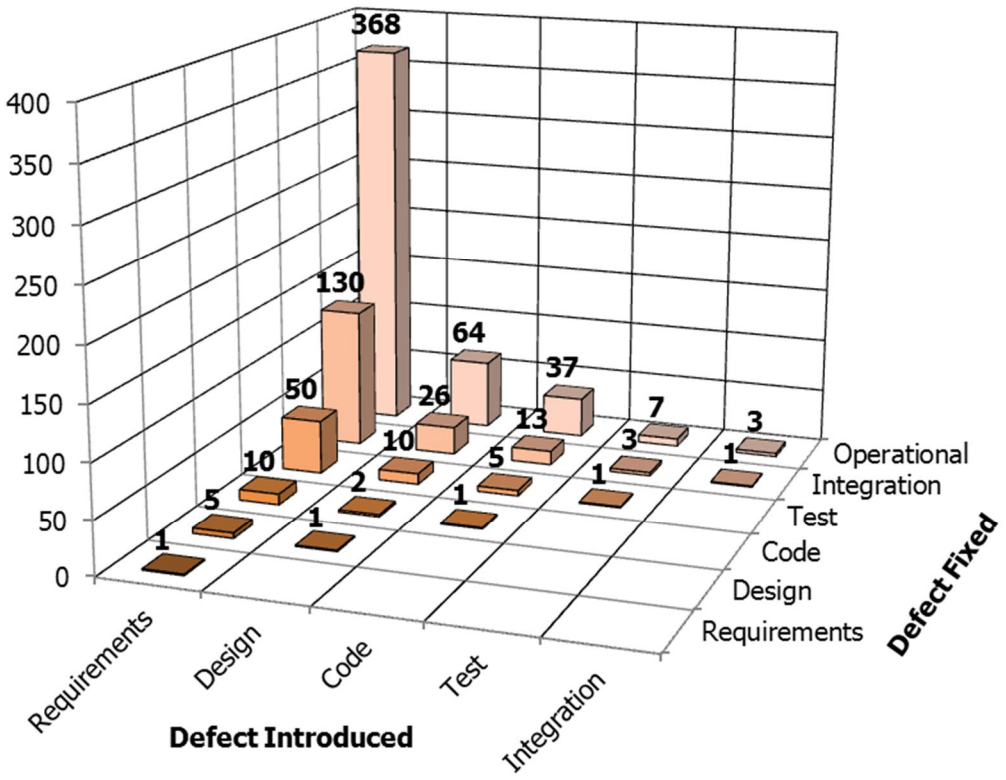


Figure 3. Relative cost to fix defects in relation of phase the defect is introduced to phase the defect is fixed.

Studies have shown that software has high defect densities: 13 major errors per 1000 lines of code on average (2, p. 184). In this light the reliability of the system is highly dependent on efficient and rigorous software testing and validation. It does not directly improve the quality of the system per se, but it does it indirectly by offering valuable data of weak spots of the system and helps to focus resources on critical areas. One should bear in mind that all defects cannot be found and there is never enough time and resources to test everything (1, pp. 3-4). Thus software testing, like any other testing process, shall be planned carefully and preparing a test strategy is essential for efficient testing since excessive testing can lead to major financial losses. The strategy should pinpoint the balance of testing costs and defects found depending on the criteria of maximum defects allowed.

3.2 General concepts

3.2.1 Software testing principles

Often software testing is considered to be a process to demonstrate that the program functions correctly, but this definition is erroneous. More appropriate interpretation is that testing is a process to find as many defects as possible. The difference may sound irrelevant semantics, but it is essential in comprehending the philosophy of testing and setting the right kind of goal. If the goal would be in demonstrating that the software works correctly, then the whole process is in danger to steer towards avoiding the defects rather than deliberately making the test fail. Testing could be described as a destructive sadistic process. (5, pp. 5-6)

There are few basic principles in software testing that greatly affect the outcome and the efficiency of the testing process. The most important principles are:

- A test case must contain a definition of the expected output and results.
- Each test result shall be thoroughly inspected.
- Test cases must be written for invalid and unexpected input conditions, as well as for input conditions that are valid and expected.
- Examine a program to see if it does not do what it should do and if it does what it should not do.
- Test cases should be stored and be repeatable.
- Plan a testing effort in assumption that defects will be found.
- The probability of existence of defects is proportional to the number of defects already found. (5, p. 15)

These principles are intuitive guidelines for test planning and help getting a grasp on the nature of testing process and its goals.

3.2.2 Dynamic Testing

Dynamic testing is the most common type of testing and it is often misinterpreted that testing as a whole is dynamic testing. There are two general types of dynamic testing: black box testing and white box testing. The difference between these two types is the

focus and scope of testing. A test plan is built on a combination of these two approaches.

In black box testing the internal behaviour of UUT (Unit Under Test) is not concerned. Test data is fed to UUT and the received output data is compared to expected data. In this approach the goal is to find circumstances in which the program does not behave according to its specification. In theory an exhaustive input testing that finds all the errors requires to test with all possible inputs in the operational area. This easily grows the number of test cases in practice to infinity. Thus, it is essential in black box testing to design the test cases in a way that the number of test cases is on acceptable level and the yield is as big as possible. (5, pp. 9-10)

In white box testing the test data and test cases are structured from the internal structure of the program. The emphasis on white box testing is often on coverage testing. It aims to test that every statement in the program executes at least once or every logical branch gets values true and false at least once during the test run. In theory white box testing suffers the same kind of infinity problem as black box testing; exhaustive path testing, in which every branching combination is tested, leads quickly to practically infinite number of test cases. Thus, in practice compromises must be done. (5, p. 11)

There is also a combination of white box and black box testing that is called grey box testing. It is black box type method that, in addition to the module specifications, has some information from the actual code of the UUT as a basis for test case designing.

3.2.3 Static Testing

Static software testing is usually inspections and reviews of the code or a document. In static testing the software is not actually used at all, but visually inspected – In other words: read. Static testing is proven to be extremely effective type of testing both in means of cost-effectiveness and defect spotting effectiveness. Due to these facts, it is advised to include static testing in every test plan and test level. (5, pp. 21-23)

The most common form of static testing is code reviews in which the code is analysed by a group of people, often by peers. Suitable number of participants is three to four persons of which one is the author of the code, one is the moderator, one is programmer peer and one is a test specialist. The moderator schedules the review, makes the practical arrangements, records the defects and ensures that the errors are corrected. The rest of the group is focused solely on the review. Usually the participants get acquainted with the code beforehand and the review conference is held where the participants meet for the actual review. The duration of the review should not exceed 120 minutes to ensure effectiveness. (5, pp. 24,26)

The code is inspected to find basic coding mishaps, but it is important to inspect the structure and evaluate the algorithm decisions since most of these types of defects are invisible to traditional dynamic computer-based testing methods. The aim of static testing is to find defects and potential hazards, but not to fix them. The advantage of code review is that the exact locations of the defects that are found are known. This makes fixing process quick and a mass of defects can be fixed in one sitting.

The code review progresses so that the programmer narrates the logic of the program step by step. The other participants raise questions and try to determine if defects exist in a collective effort. After the narration the program is analysed with respect to a checklist of common defects. See Appendix 1 for a revised list of defects. The moderator ensures that the discussion proceeds along productive lines and the focus is on finding defects and not fixing them. After the review the programmer is given a list of the defects found. (5, p. 25)

In addition to code, other documents can be reviewed in the same manner as code is reviewed. It is well advised to review software specifications, test plans, et cetera, to root out defects from the design documents. Specification documents should be reviewed with care, since defects that are injected into the specifications are extremely hazardous and can jeopardise the whole project when the developed system does not meet the customer's requirements.

3.2.4 Levels of Testing

Software testing can be divided roughly to four levels. They are, from low-level to high-level: unit testing, integration testing, system testing and acceptance testing. Different levels are performed by various testers and teams at various phases in the project timeline, so that the levels help organizing the testing process. Test levels define who is performing the testing and when. It structures the testing process by incremental principles from small isolated parts tested at the lower levels to larger components or subsystems being tested at the higher levels. A good distinction between low-level and high-level testing is the position of the testing in the product development lifecycle. (1, p. 34)

Unit testing is the lowest level of software testing process in which individual units of software is tested. A unit is the smallest testable part of software, which is usually a single function. This level of testing is normally carried out by developers in parallel with development process as a part of test driven development practice. In unit testing a white box testing method is often used due to the small size and simplicity of the code under test. The aim is that majority of defects are discovered at this level, because discovering and fixing the defects at this stage is easy and fast compared to fixing defects discovered at later development stages. Unit testing also increases resistance to defects caused by changes in the code, since the test scripts can be run every time a change is made to assure the unit works as intended after the changes. Due to the sheer volume of defects possible to discover and fix with unit testing, it can be regarded as the most important level of testing. (6)

Integration testing is a level of software testing process where the units are combined and tested as functional groups. At this level the emphasis is on testing the interface and interaction between the units. Testing method can be black box testing, white box testing or grey box testing. The choice of method depends on the type of the units and on complexity of the group under test. Integration testing is carried out by the developers or, in some cases, an independent software testing team. (6)

System testing is a higher level of the software testing process where a complete integrated system is tested. On embedded systems this phase is usually carried out on prototypes where the functionality of the embedded software in the destination plat-

form is tested in black box method. System testing aims to reveal defects in which the software does not meet the requirements set for the product and defected interaction between the embedded system and the plant. System testing is usually carried out by independent testing team but on small projects this phase can be carried out by the developers. (6)

Acceptance testing is the final stage of testing where the final product is put under test and the design is accepted for release. The aim is to find the defects in which the product does not fulfil its original requirements and purpose for the customer. Acceptance testing is usually purely ad hoc and the number of defects discovered is minimal, preferably none. In some cases acceptance testing can be performed by the customer. (6)

3.3 Testing Methods

3.3.1 Black Box Testing Methods

As mentioned earlier in context of black box testing, an exhaustive testing of all input values leads to practically infinite number of test cases. The test cases must be scaled down and this is when equivalence partitioning and boundary-value analysis techniques are applied. These techniques are very intuitive way to scale down the test cases.

In *equivalence partitioning* the input values are divided into two or more partitions depending on the function of UUT. The partition classes are value ranges that are either valid or invalid equivalence classes. In other words, every value on the class range can be safely assumed to be either valid or invalid input. For example, if the UUT is a function that requires a voltage as a floating point input V in range of 0.0 to 5.0 volts, the values of V can be divided into three equivalence classes: valid class $0.0 \leq V \leq 5.0$ and invalid classes $V < 0.0$ and $V > 5.0$.

With *boundary-value analysis* a test cases can be constructed from the equivalence partitions. In this technique the test cases explore the boundary conditions of the equivalence classes. For example on the case of the voltage V, the boundary condition

test cases could be 0.0, 5.0, -0.001 and 5.001. With these test cases probable comparison errors can be detected. In some cases it can be profitable to analyse the output boundaries and try to produce a test case that causes the output to go beyond its specified boundaries. In reality boundary-value analysis can be challenging task due to its heuristic nature and it requires creativity and specialisation towards the problem. (5, pp. 59-60)

In addition to these equivalence classes, a *defect guessing* technique can be used to add test cases that are not covered by the classes. These test cases can be inputs in invalid data type, void inputs or other defect prone inputs. Effective defect guessing requires experience and natural adeptness. (5, p. 88)

3.3.2 White Box Testing Methods

Logic coverage testing is a white box testing method and it is the most common type of coverage test. As mentioned in chapter 3.2.2, complete logic path testing is impossible due the practically infinite number of test cases, so the test must be scaled down but still endeavour to meet the required level of coverage. Finding the optimal test case set is a challenging task.

Statement coverage testing is a test in which every statement of the UUT is executed. Such a test is easily designed but it is very weak type of test since it overlooks possible defects in branch decision statements. It can be said that statement coverage testing as such is so weak that it is practically useless. (5, pp. 44-45)

Stronger logic coverage test is *decision coverage* test. In this test type the test cases are written so that every branch decision has a true and false outcome at least once during the test run. Branch decisions include `switch-case`, `while` and `if-else` structures. This also very often covers statement coverage, but it is advisable that statement coverage is ensured. Decision coverage is stronger than statement coverage but still is rather weak since some conditions can be skipped to fulfil complete decision coverage. (5, pp. 45-46)

Condition coverage is the next stronger criterion. In condition coverage test cases are written so that every condition in a decision takes on all possible outcomes at least once. In addition, to benefit from this criterion statement coverage must be added on top of the condition coverage. The decision coverage and condition coverage can be combined to achieve acceptable logic coverage. (5, pp. 46-47, 49)

The choice of criterion depends on the structure of the UUT. For programs that contain only one condition per decision, decision coverage test is sufficient. If the program contains decisions that contain more than one condition, it is advisable to apply condition coverage. (5, p. 52)

3.3.3 Non-Incremental and Incremental Unit Testing

Testing the whole system at once with an acceptable coverage can be a devious and often impossible task. Therefore it is more effective to test the system in small units, hence the term unit testing. This way the test cases become easier to design and acceptable test coverage can be managed more efficiently.

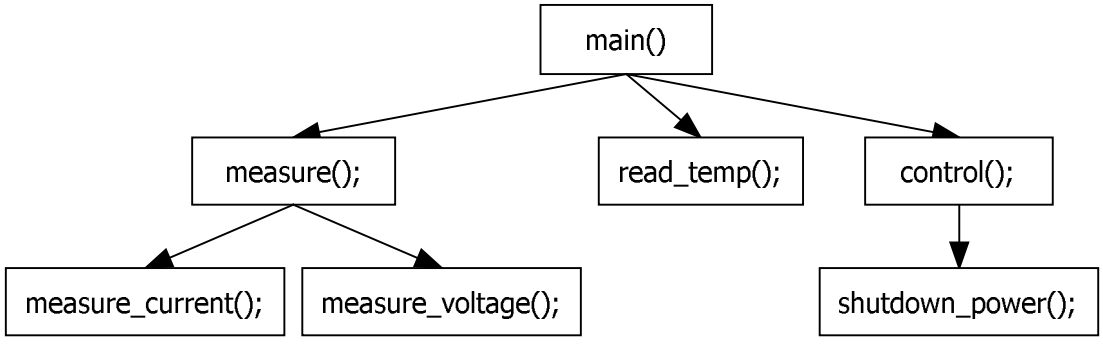


Figure 4. Example program structure

One way to perform unit testing is to test every unit independently and combine the tested modules to form the complete program. In this method when the functions of the simple example program in Figure 4 are tested, every function needs a driver and in some cases one or more stubs. A driver is a program that passes the test case input arguments to the module. It simulates the higher level module. A stub is a program that simulates the effects of the lower level modules. When every function is tested the

functions are combined and final integration tests are performed. This method is called *non-incremental* big bang method.

There are several obvious problems in this approach. Since every function requires driver and stubs to be written, testing requires lots of additional work – The example program in Figure 4 requires six stubs and six drivers, which can result in hundreds of lines of additional test code. Stubs can get complex, since they have to simulate sections of the code. Because of this writing a stub is not always a trivial task. Also defects caused by mismatching interfaces or incorrect assumptions among modules are overlooked; not to mention possible defects in written stubs causing incorrect test results. Also, when the modules are combined in a big bang manner, pinpointing the location of defects revealed in that phase can be difficult, since the defect can be anywhere. (5, pp. 107-110)

The only real advantage in non-incremental testing is that it allows parallel testing of units, which can be effective in massive software projects. This is rarely the case in small scale embedded systems. The problems of non-incremental method can be solved by incremental testing, which can be considered superior to non-incremental testing. In incremental testing one module is tested and then a new module is combined to the tested module gradually building and testing the complete program. The integration testing can be considered to be performed in parallel with the unit testing. There are two incremental testing methods: top-down and bottom-up.

In *top-down method* the testing is started from the highest module: main() in the example on Figure 4. This method requires only stubs to be written. The stubs are gradually replaced by the actual functions as the testing progresses. In this method the test cases are fed to the modules via the stubs. This may require multiple versions of stubs to be developed. To ease the testing process, it is advisable to test I/O modules first. The I/O module often reduces the required code and functionality of stubs. The most complex and defect prone modules are advisable to be tested as soon as possible, since pinpointing the defects in the early stages is somewhat easier. (5, pp. 110-114)

The advantage of top-down method is that an early working skeletal version is gradually formed and it serves as evidence that the overall design is sound. This makes test-

ing the software as a whole system on the destination platform possible earlier. A serious shortcoming of top-down method is the problem of test case feeds. Since there are no drivers, the test case injection and result gathering can be a challenging, or even impossible, task. Also the distance between the point of test case injection and the actual UUT can become unnecessarily long, thus making test case designing difficult. These features can lead to insufficient testing. (5, pp. 114-116)

The bottom-up method addresses these problems. In this method the testing is started from the lowest modules and the drivers are gradually replaced by the functions as the testing progresses. A unit can be tested if, and only if, all of its lower functions are tested, e.g. measure() on the example program can be tested only after measure_current() and measure_voltage() have been tested and fixed. Bottom-up method can be considered to be an opposite of top-down method; the advantages of bottom-up are the disadvantages of top-down and vice versa. The greatest advantage in bottom-up method is, no doubt, that the test cases can be implemented with ease in the drivers contrary to in the stubs in the top-down method. Also, multiple versions of the same driver are not needed, thus reducing the work required for performing the tests. The problem in bottom-up is that the working complete program, or the skeletal version, is achieved only after the last module. (5, pp. 116-117)

There is no absolute truth which method is better: top-down or bottom-up. It depends on the structure and function of the software. Also sheer luck is a factor, since bottom-up method is disadvantageous if major defects are manifested in the higher level modules and vice versa. This is especially troublesome if the defect is injected in a design phase and the fix requires redesign. Naturally top-down and bottom-up methods can be combined. (5, pp. 118-119)

3.3.4 System Testing

The system testing is not a process of testing the functions of the complete system – That would be redundant, since the functions are tested in the integration testing level. System testing is a process to compare the system to its original objectives, e.g. the requirements set by the customer. The test cases are not developed on the grounds of detailed software specifications, but on the grounds of user documentation. There are

several categories in system testing. Naturally, not all categories apply to all systems. (5, pp. 130-131)

The system testing categories possibly affecting Efore products are as follow:

- *Facility testing* aims to determine whether each facility mentioned in the user documentation or customer requirements is actually implemented.
- In *volume testing* the system is subjected to heavy volumes of data – For example the system I/O is fed with continuous control data for a long period of time.
- *Stress testing*. It should not be confused with volume testing, since in stress testing the system is subjected to heavy maximum loads for a short period of time. Stress test can also test a situation in which the maximums are temporarily exceeded.
- *Usability testing* is a rather broad category, since it includes all user interface questions, e.g. is the data input syntax consistent and are the possible error messages and indicators logical.
- *Performance testing* is also an intuitive category. This category addresses the question whether the system contains timing or response time defects.
- In *storage testing* the memory features of the system are tested. The aim is to show that the memory related features are not met.
- *Compatibility/conversion testing* is a category that affects products that are designed to replace an existing obsolete product. This testing aims to find compatibility defects between the tested product and the obsolete product.
- *Reliability testing* is a self-explanatory category. The aim is to show that the product does not meet the reliability requirements. This is especially important feature in Efore products, since software crash is a completely unacceptable situation. Testing for long uptime requirements is impossible, and therefore special techniques must be used in order to determine the reliability issues.
- In *recovery testing* the system is introduced to unexpected temporary error states, such as power failures, hardware failures and data errors. The aim is to show that the system does not recover from these failures.
- *Serviceability testing* is required if the system software is designed to be updated on the field or it has other serviceability or maintainability characteristics. (5, pp. 132-142)

It is advisable that system testing is performed by an independent testing team to ensure unbiased testing, since psychological ties of the development team can subconsciously stand in the way of rigorous system testing. After all the aim of system testing is to break it. (5, p. 143)

3.4 Testing of Embedded Software

Basic rules of software testing also apply to embedded software. There are, however, certain special requirements and additional factors that have to be taken into account when testing embedded software. As mentioned in chapter 3.2.4, there are four levels of testing that apply to software. In embedded systems the second level, integration testing, can be divided into two segments: software integration testing and software/hardware integration testing (1, p. 35). In the latter, interfaces and interaction of the software components and hardware domain are tested. This can include testing the interaction between software and built-in peripheral devices or the plant.

One special characteristic in embedded software development is that the actual environment, in which the software is run, is usually developed in parallel with the software. This causes trouble for testing because comprehensive testing cannot be performed due to the lack of hardware environment until the later part of the project lifecycle. This can be solved by a hardware testing platform on which the tests can be carried out in simulated environment before the actual environment is available.

3.4.1 TEmb Method Overview

Broekman and Notenboom introduce TEmb method, which helps to tailor testing scheme for a particular embedded system. In TEmb method the test approach for any embedded system can be divided in two parts: the generic elements applicable to any structured approach and application specific measures. The generic part consists of for instance planning the test project, applying standardized techniques, organizing test teams, dedicated test environments, reporting, et cetera. They are parts of four cornerstones of structured testing: lifecycle, infrastructure, techniques and organization, referred to as LITO. (1, p. 7)

This generic test approach is not concrete and needs to be supplemented with details such as what design techniques will be applied and which tools will be used. These choices depend on the characteristics of the system under test. These choices are made in the initial state of the project when the overall test approach is planned. This process in TEmb method is called the mechanism. (1, p. 7)

3.4.2 TEmb Generic

The generic part of TEmb can be divided to LITO, as stated before. These LITO parts are the backbone of the testing method and they contain everything that is required for any project. These four LITO characteristics, lifecycle, techniques, infrastructure and organisation are shared characteristics and every one of these cornerstones must be equally covered. The lifecycle is the central piece that binds the whole process together. When all the cornerstones are supported, the testing system becomes structured and thus manageable. (1, pp. 10-11)

The lifecycle contains the actual plan in time domain defining what shall be done and when. It can be divided into five phases, which are planning & control, preparation, specification, execution and, finally, completion. As illustrated on figure 5, the planning & control is the guiding process that holds up the rest of the lifecycle phases in order and control.

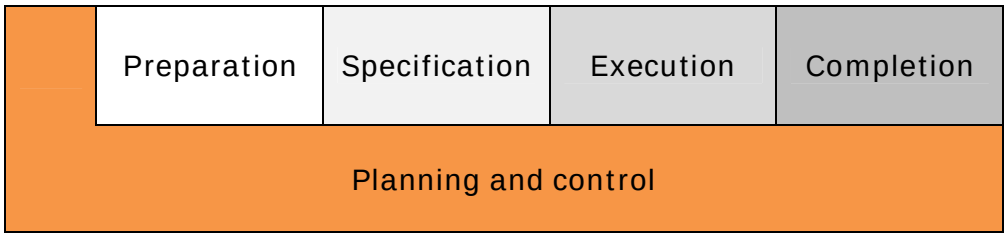


Figure 5. Lifecycle Model

The main function of the lifecycle process is to speed up and organise the workflow of the testing process. When the execution phase is reached, everything concerning specification and preparation shall be completed. For example, when testing (execution) begins, the test cases (specification) and testing techniques (preparation) shall be

completed. Basically it considers things that must be done and arranges them efficiently for streamlined execution.

Techniques cornerstone is a supporting process that defines the techniques used not only in the testing itself, but also techniques used in controlling the process, tracking the process and evaluating the results. It is a collection of old and new techniques that are evaluated and chosen for use. The techniques can be, for example, strategy development, test design, safety analysis, test automation and checklist techniques.

The infrastructure consists of all facilities required to perform the testing. These facilities are test environment, tools, automation and office environment. The test environment consists of the actual environment where the testing takes place. This can be a prototype, a hardware emulation environment, a computer simulation or a production unit. Depending on the nature of the platform the testing takes place, the testing task may require different types of measurement equipment and if the tested unit needs external simulated signals and impulses, the generation of these must also be included in the infrastructure. The test environment also contains test databases for archiving results, test cases and simulation runs for required repeatability.

Organisation is the group of people performing the testing. It contains the organisation structure, roles, staff & training and management & control procedures.

3.4.3 TEmb Mechanism and Specific Measures

For a particular embedded system the TEmb generic must be topped up with system specific measures which add definitions to the LITO matrix. The aim is to define the application specific characteristics that dictate the specific measures. One tool to categorise the system under test is to use a checklist of different features: safety critical, technical-scientific algorithms, autonomous, one-shot, mixed signal, hardware restrictions, state-based, hard real-time, control and extreme environments. The system can fulfil one or several of these features which then guide the test planning process. (1, p. 16)

An embedded system is *safety critical* if a failure can cause serious physical damage or serious risk to health or even lead to death, although some systems can be considered partly safety critical if they can cause serious hazard indirectly. In safety critical systems risk analysis is extremely important and rigorous techniques are required to analyse and guarantee the reliability. (1, p. 16)

Some applications have to perform complex *scientific algorithms*. These require vast amounts of processing and often heavy data traffic. These types of systems are often control applications that control vehicles, missiles or robots. The large and the most complex part of their behaviour is practically invisible from the outside the system. Therefore the testing should be focused on early white-box level testing and less is required for black-box oriented system testing and acceptance testing. (1, p. 16)

Autonomous systems are meant to operate after start-up without human intervention or interaction. This kind of system can be a traffic signalling systems and some weapon systems. Such systems are designed to work continuously and react to certain stimuli independently without intervention for indefinite period of time. This means that manual testing of this type of systems is difficult or even impossible. A specific test environment and specific test tools are required to execute the test cases and analyse the results. (1, p. 17)

Unique *one-shot* systems are released only once and cannot be maintained. They are bespoke systems that are meant to be built correctly in one shot, hence the name one-shot system. These types of systems are usually satellites. For unique systems there are no long-term goals in testing due to the first and only release. For this kind of system the testing scheme must be reconsidered in areas of maintenance, reuse, et cetera. (1, p. 17)

Mixed signal systems are systems that contain, in addition to binary signals, analogue signals. Inputs and outputs do not have exact values but have certain tolerances that define flexible boundaries for accepted test output. The boundaries often contain a grey area where the tester is required to make a decision if the test output is accepted or rejected. These types of systems are often tested manually because the test results are not always implicit. (1, p. 17)

The limitations of hardware resources set certain restrictions on the software, like memory usage and power consumption. Also, most of the hardware timing issues are solved in software. These issues require specialised and technical testing methods. (1, p. 17)

State-based behaviour is described in terms of transitions from a certain state to another triggered by certain events. The response depends not only on the input, but also on the history of previous events and states. On a state-based system identical inputs are not always accepted and, if accepted, may produce completely different outputs. Testing this kind of system requires careful test case planning and test automation. (1, p. 18)

The definition of *hard real-time* system is that the exact moment that stimulus occurs influences the system behaviour. When testing this type of systems, the test cases must contain detailed information about the timing of inputs and outputs. In addition, the test cases are often sequence dependent, which require some level of automation. (1, p. 18)

A control system interacts with the environment according to continuous feedback system. The system output affects the environment and the effects are returned to the system which affects the control accordingly. Therefore the behaviour of the system cannot be described independently, since the system is tightly interlinked with events in the environment. These kinds of systems require accurate simulation of the environment behaviour. (1, p. 18)

If the system is exposed to *extreme environment*, such as extreme heat, cold, mechanical strain, chemicals, radiation, et cetera, it should be considered to take the conditions into account in testing the embedded system. Environment dependent testing is more hardware oriented since the embedded software per se is not environment dependent. (1, p. 18)

All these characteristics help to plan special test cases for the system under test. The characteristics are also a good tool for risk analysis. The selected measures can be

divided to lifecycle, infrastructure, techniques and organisation, similarly to the TEmb generic methods. The relationships between system characteristics and the required test cases are analysed and they can be presented in a so-called LITO-matrix. An example matrix is presented in Table 1. This matrix helps to define testing approach.

Table 1. Example of LITO matrix

System characteristic	Lifecycle	Infrastructure	Techniques	Organisation
Mixed signals		Mixed signal analyser Signal generator	Analyse mixed signal output	Laboratory personnel
State-based behaviour		State modelling and testing tools	State transition testing Statistical usage testing Rare event testing	
Control system	HW and SW in the loop	Simulation of feedback-loop	Statistical usage testing Rare event testing Feedback control testing	Hardware engineering

The LITO matrix is not software exclusive, but neither are embedded systems. The matrix contains measures that overlap with general system and hardware testing. Therefore is advisable to establish the testing approach in collaboration with the testing team and hardware designers. These hardware overlaps are the very core of problem in embedded software testing. With TEmb method these issues can be addressed when planning the software testing.

4 Implementation of Software Testing

The current Efore Software Development process is illustrated in appendix 2. The process is waterfall based and it contains an iterated loop to finally refine the software from the first prototype round to final release version. The actual software testing methods are static reviews between the process phases and the final verification testing in the SW Verification phase. The process lacks strictly defined low level testing directions and software testing policy needs improvement. Every designer needs to develop their own methods to test and debug the code, because there is no low-level software testing plan. Testing is considered solely to be an invisible part of the coding process and therefore it is easily neglected and poorly documented.

The implementation of rigorous software testing in Efore is challenging. There are some characteristics in Efore software development process that make it difficult to apply conventional software testing methods and models to Efore Product Development. The greatest challenge is that the projects are software-wise almost always one-man projects from the beginning to end. This means that there is no separate software testing team. All the testing is done in later stages in form of acceptance testing and no separate software testing team exists. This is problematic because most of developed software testing methods postulate that the process contains independent software testing team and the emphasis is set on that team.

The size of the software development team and lack of independent software testing team means that it is unavoidable that the software testing responsibility will be on the developers. There are, however, a few methods that can be applied to this kind of scenario. Due to the special requirements, instead of selecting a single existing method leading to unsatisfying compromise, desired result can be achieved with a combination of several methods and models.

The tailored method for Efore consists of three segments, which cover all the requirements for effective testing and quality control. The segments are Test Maturity Model integration TMMi covering organising and evaluating the testing process, spiral process model covering process control and Test Driven Development (TDD) covering the testing itself during coding process. With these methods a firm basis for effective software

testing can be established and developed further when the software development team grows.

4.1 Test Maturity Model integration TMMi

The Test Maturity Model integration, shortly TMMi, is developed by an independent non-profit TMMi foundation dedicated to improving software testing in the software industry. The foundation was founded by an international group of leading testing and quality experts aiming to standardise software testing processes and offering open domain model for the industry to improve software quality. (7)

The two main influences of TMMi model are TMM framework developed by the Illinois Institute of Technology and the Capability Maturity Model Integration (CMMI), which is often regarded as the industry standard for software processes. The relation to CMMI ease up integration of TMMi to the vast number of companies that already have CMMI based process structure, and the TMM giving the test focused approach to the quality control and quality assurance questions. The other influences in TMMi are Gelperin and Hetzel's Evolution of Testing Models, Beizer's testing model and International software testing standard IEEE 829. (3, pp. 7-8)

The need for improved process model arises from two problems. First of all, the semi-standard process model CMMI gives only little attention to testing. The TMMi is a more detailed model for software process improvement including more broad view on software testing and emphasis on the importance of testing, quality control and quality assurance. Secondly, CMMI model does not define clear fixed levels or stages to proceed through during the improvement process. This hinders the evaluation of the current level or stage the company is on. TMMi model is structured clearly in fixed levels in order to aid planning of development strategy and eventually evaluate the progress. All in all, TMMi is developed to support evaluation and improvement of testing process, making possible to establish progressively more effective and efficient test process. (3, p. 7)

One of the reasons TMMi was chosen is, that the scope of TMMi model is in software and system engineering. This makes TMMi extremely suitable for embedded software

design because the model accommodates processes outside the pure software domain, which is essential in embedded system design where the separation of software from the complete system itself is virtually impossible. The second fact that supports TMMi is that it covers all levels of software testing from low-level test planning to high-level project validation. It addresses all the cornerstones of testing in the LITO model included in TEmb method. Thirdly, the clearly defined stages offer great guideline for the development of Efore software testing process and its further improvement.

4.1.1 TMMi Maturity Levels

TMMi contains five maturity stages through which the testing process evolves and the improvement can be assessed. On the lowest level, level 1, testing is chaotic, ad hoc based and unmanaged and on the highest level, level 5, testing is managed, defined, measured and highly optimised. Every stage contains basis for the next stage, thus the prerequisites for development are ensured when the testing process development plan follows the model carefully. This way the improvement steps can be followed systematically and development cannot get stuck or end up unbalanced. (3, p. 10)

Level 1 – Initial

The first level is the initial step where the testing is undefined process and is often considered solely as debugging. The supporting organisation is weak or often nonexistent. The test cases are developed in ad hoc manner after the coding process itself is completed. The objective on this level is to ensure the software runs without major failures. The testing also lacks resources, tools and testing experience. One notable characteristic of level 1 is that project deadlines are often overrun and the products do not tend to be released on time. (3, p. 11)

Level 2 – Managed

At the second TMMi level the testing process is managed and disciplined. Testing itself is separated from debugging and the good existing practices are endured. On this level a general test strategy is established to guide the testing improvement towards level 3. Test plans are developed, which are further defined by means of product risk assess-

ment as instructed in TEmb method. Testing is monitored and controlled to avoid neglects. On level 2 the testing is already on four levels, from unit testing to acceptance testing; although testing may begin late in the development lifecycle after the coding phase. Every testing level has separate defined testing objectives in the general testing strategy. The objective of testing is to verify that the product fulfils the product specification. Defects and quality problems still occur. Since the testing is started too late, defect lifecycles are long and some defects even propagate from the initial design phase. There are no formal review programs to address the problems. (3, p. 11)

Level 3 – Defined

At level 3 testing is continuous parallel process integrated to the project lifecycle. Test planning is done during the requirements phase and is documented in a master test plan. The standard test processes acquired during level 2 are aggregated and the set improved further. A separate testing organisation exists and testing training is given. Software testing is perceived as a profession. The test process improvement strategy is institutionalised. The importance of reviews is understood and formal review program is implemented, which addresses reviewing across the project lifecycle. At this level test designs and techniques are expanded beyond functionality testing to include reliability. Testing is now more rigorous. Also on level 2 test procedures, process descriptions and standards were defined separately for each instance, for example particular projects, whereas on level 3 a general default template exists and this template is tailored to suit different instances, thus creating more consistent testing practice. (3, pp. 11-12)

Level 4 – Measured

Previous levels have created capable infrastructure for rigorous testing and further testing improvements. When the infrastructure is in its place, the process can be measured and evaluated encouraging further accomplishments. At this level testing is more of an evaluation process than productive process attached to development lifecycle. An organisation-wide test measurement program will be established to evaluate the quality and productivity of testing and monitor improvements. Measures are used as basis for decision making and they direct predictions relating to test performance

and costs. The measurement program also allows evaluation of better product quality by defining quality needs, attributes and metrics. This way product quality can be understood in quantitative terms instead of quality being an abstract goal. This makes possible to set concrete objectives regarding quality. Reviews and inspections are regarded as part of the test process and are used early in the lifecycle to control quality. Peer reviews are transformed into a quality measurement technique, in which the data gathered from the reviews is used to optimise the test approach integrating the static peer reviewing into the dynamic testing process. (3, p. 12)

Level 5 – Optimised

At level 5 the process can be continually improved by the means of quantitative and statistical understanding of the testing process. The performance is enhanced by technological improvements and the testing process is fine-tuned. The testing organisation that was founded on level 3 now has specialised training and the responsibility of the group has grown. The defect prevention program is established to support quality assurance process. The test process is statistically managed and it is characterised by sampling-based measurements. A procedure is established to identify enhancements in the process, as well as to select and evaluate new testing technologies for possible future use. The testing process is now extremely sophisticated and continuously improving process. (3, pp. 12-13)

4.1.2 TMMi Structure

As mentioned before in chapter 4.1.1, the TMMi model consists of five different maturity stages that indicate the degree of the current testing process quality and that can be used as a guideline for measures required for the next stage. Every stage contains three kinds of component groups: required, expected and informative components.

Required components describe the base core of requirements the testing must fulfil. They are specific and generic goals and the goal satisfaction is used to measure if the process is mature for that TMMi level. *Expected components* describe what will typically be implemented to achieve typical component. Expected components guide the process as they contain specific and generic practices. These practices, or their alterna-

tives, must be present in the planned and implemented process for the goals to be considered satisfied. *Informative components* provide details and help solving how to approach the required and expected components.

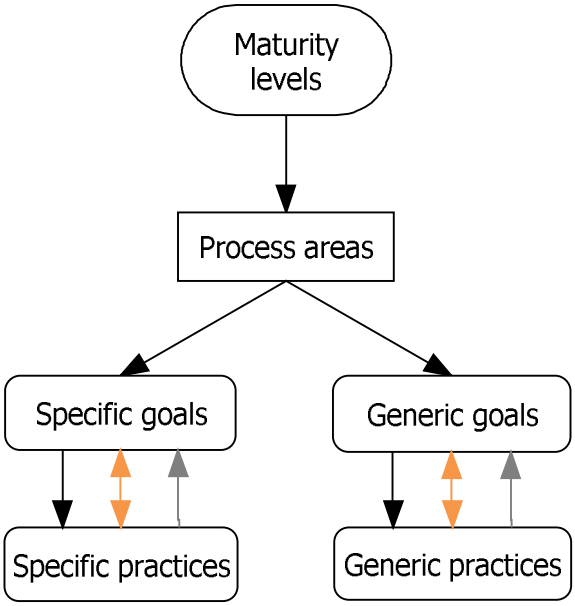


Figure 6. TMMi model structure

Every maturity level from level 2 to higher levels contains several process areas that indicate the characteristics of that level. In other words, the very requirements the testing process must fulfil in order to be considered to fulfil the maturity level in question. As shown on figure 6, each process area contains the two types of goals and the practices linked to them. The component groups bind goals and practices together. The black arrow illustrates required components that define the practices needed, grey arrow illustrates expected components that contribute to the goals and the orange double-arrow illustrates informative components that define the relation of the goals and practices in detail.

The TMMi document contains detailed descriptions of each maturity level. In that document the maturity levels are defined by several types of components that bundle up in the three component groups. *A purpose statement* is an informative component that describes the generic purpose of a process area. *Introductory notes* is an informative component that describes major concepts behind a process. *Scope* identifies test practices that are addressed by a process. *A specific goal* is a required component that

describes the unique characteristics of a process. *A generic goal* is a required component that describes the characteristics that must be present to institutionalise a process. *A specific practice* is an expected component which is a description of an activity that is important for achieving the associated goal. It describes the activities expected to result from the goal. *A typical work product* is an informative component that lists sample outputs from a specific practice. *A sub-practice* is an informative component that provides guidance for implementation of a specific practice. *A generic practice* is an expected component that describes an important activity for achieving the associated generic goal. *Generic practices elaboration* is an informative component that guides how the generic practice should be applied. (3, pp. 15-16)

4.1.3 TMMi Level 2 – General

Level 2 is the first TMMi level that actually has requirements to fulfil. It is the first step from the level 1 chaos towards more organised and efficient testing principles and processes. The process areas at TMMi level 2 are

- Test Policy and Strategy
- Test Planning
- Test Monitoring and Control
- Test Design and Execution
- Test Environment

Level 2 is the initial step Efore has to take in improving its software testing process. Therefore it is justified to inspect the level 2 and its requirements and process areas in detail.

4.1.4 TMMi Level 2 – Test Policy and Strategy

The purpose of this process area is to develop and establish a general software test policy in which the test levels are unambiguously defined and test performance indicators are introduced. The test policy defines overall test objectives, goals and strategic views creating a common view on software testing between all stakeholders. The policy should contain objectives for test process improvement and these objectives should

subsequently be translated into test performance indicators. The test policy and these indicators provide a clear direction for test process improvement. (3, p. 24)

A test strategy will be defined from the test policy. The strategy contains generic test requirements and it addresses the generic product risks and a process to mitigate those risks. Preparation of the test strategy starts by evaluating the risks addressed in the TEmb method. The test strategy is the foundation for project's test plan and activities. The strategy describes the test levels that are to be applied and the objectives, responsibilities, main tasks and entry/exit criteria for each of these levels. (3, p. 24)

The test policy and strategy shall be established on basis of the current TMMi level: in this case, level 2. The policy and strategy modification is required as the test process evolves and moves up the TMMi levels. The detailed description of the Test Policy and Strategy process area can be found on Test Maturity Model integration TMMi v.3.1 document on pages 24-31. (3, p. 24)

4.1.5 TMMi level 2 – Test Planning

The purpose of Test Planning is to define a test approach based on risk analysis and the test strategy and create a plan for performing and managing the software testing. The plan is established according to the details of the product, the project organisation, the requirements and the development process. The risks define what will be tested, to what degree, how and when. This information is used to estimate the costs of testing. The product risks, test approach and estimates are defined in co-operation with all the stakeholders. The plan further defines the provided test deliverables, the resources that are needed and infrastructural aspects. If the plan does not comply with the test strategy in some areas, it should explain the reason for these non-compliances. (3, p. 32)

The test plan document is developed and accepted by the stakeholders. The plan is the basis for performing and controlling the testing in the project. The test plan can be revised if needed. The detailed description of the Test Planning process area can be found on Test Maturity Model integration TMMi v.3.1 document on pages 32-46. (3, p. 32)

4.1.6 TMMi level 2 – Test Monitoring and Control

The purpose of this process area is to provide methods for monitoring product quality and progress of testing so that actions can be taken when test progress deviates from plan or product quality deviates from expectations. Monitoring is based on data gathering from the testing process, reviewing the data for validity and calculating the progress and product quality measures. The data can be for example from test logs and test incident reports. The test summary is written from the monitoring process on periodic event-drive basis to provide information on testing progress and product quality, with emphasis on the product quality. (3, p. 47)

When deviations are revealed in the monitoring, appropriate corrective actions should be performed. These actions may require revising the original plan. If the corrective actions affect the original plan, all stakeholders must agree on the actions. (3, p. 47)

The essential part of this process area is test project risk management, which is performed to identify and solve major problems that compromise the test plan. It is also important to identify problems beyond the responsibility of testing, for example changes in product specification, delays in product development and budget issues. (3, p. 47)

4.1.7 TMMi Level 2 – Test Design and Execution

This process area aims to improve test process capability during design and execution phases by establishing design specifications, using test design techniques, performing a structured test execution process and managing test incidents to closure. Test design techniques are used to select test conditions and design test cases according to requirements and design specifications. (3, p. 58)

Test conditions and test cases are documented in a test specification. A test case consists of the description of the input values, execution preconditions, expected results and execution post conditions. Test cases are later translated to test procedures. A test

procedure contains the specific test actions and checks in an executable sequence, including specific test data required for the execution. (3, p. 58)

The test design and execution activities follow the test approach, which is defined in the test plan. The test approach determines the test design techniques applied. During the execution, defects that are found are reported, logged using a defect management system and delivered to the stakeholders. A basic incident classification scheme is established and procedure is created to manage defect lifecycle process and ensure each defect to closure. (3, p. 58)

4.1.8 TMMi Level 2 – Test Environment

The purpose of this process area is to establish and maintain testing environment in which it is possible to execute tests in a manageable and repeatable way. A managed and controlled test environment is essential for successful and effective testing. The environment is needed to obtain test results under conditions close to the real-life simulation. The test environment also contains test data to guarantee test reproducibility. The test environment is specified early in the project, preferably in conjunction with product specification. The specification is reviewed to ensure correct representation of the real-life operational environment. (3, p. 69)

Test environment management is responsible for availability of the testing environment for the testing stakeholders. Test environment management manages access to the test environment, manages test data, provides and executes configuration management and provides technical support during test execution. (3, p. 69)

Test Environment process area also addresses requirements of generic test data and the creation and management of the test data. The specific test data is defined in test design and analysis phase, but the generic test data is, if defined as a separate activity, included in Test Environment process area. Generic test data is reused and provides overall background data to perform the system functions. It often consists of master data and some initial content for primary data with possible timing requirements. (3, p. 69)

4.2 Spiral Model

It is essential for successful and effective software testing that the testing process is controlled and structured as pointed out in chapter 3.4. This way the testing process can be evaluated and, most importantly, executed with ease. By defining testing process strictly, testing level overlaps and general neglects can be avoided and the efficiency and coverage of the testing process is optimised.

The spiral model fits very well to general Efore Product Development process and it gives good support to testing and quality control measures due to its iterative nature. The model is cyclic in nature, which means every development level follows the same phases. The original spiral model consists of six phases:

1. Determine objectives, alternatives and constraints
2. Identify and resolve risks
3. Evaluate alternatives
4. Develop and test the current level
5. Plan the next level
6. Decide on the approach for the next level

The main idea behind the spiral model is that everything is not defined in detail at the very beginning of the project and testing is included in every cycle. In Efore Software Development lifecycle every phase can be considered to consist of one or more cycles. This makes the process more structured, yet flexible, and controlled throughout the project lifecycle. In addition the model requires extensive testing in every cycle, making static testing the mandatory minimum.

For Efore the spiral model can be simplified and modified to include the phases of TEmb lifecycle introduced in the chapter 3.4.2:

- Preparation and specification (includes phases 1.—3.)
- Execution (phase 4.)
- Completion (includes phases 5.—6.)

The beginning of the cycle starts by preparation and specification. In this phase the means to achieve the goal set for the cycle are evaluated and defined. The deadlines for the cycle are defined.

In the second phase the required operations are carried out and the result is tested. The testing consists of at least of static testing. This means that every cycle deliverables are reviewed, including the test plan and software specifications. The project contains several reviews along the whole project lifecycle. Due to the sheer number of reviews, only major revisions should be reviewed by formal review process described in chapter 3.2.3.

The minor revision reviews are carried out by means of peer reviews. For every project a review software engineer peer is appointed to support the main software engineer. The peer participates in all reviews of the project. The minor reviews can be informal, but static testing deliverables must always be generated. When the peer reviewer is one and same person during the whole project, the time required for reviews is minimised as the reviewer is already familiar with the project and the functionality of the code. One cycle can contain several informal reviews.

In the last phase the decision is made whether to advance to the next phase in the project lifecycle or reiterate the previous phase. If the project is not yet finished, goals for the next iteration cycle are defined. The plan is composed according to the project plan and test plan.

4.3 Test Driven Development

The general problem with embedded system development is that hardware is developed concurrently with the software. The hardware may go through several iterations causing need for changes in the software and the hardware is usually available late in the project. This often leads to a situation where software testing is pushed to the first prototype round where it ends up being endless bout of debugging and regressive testing. This is also the current situation at Efore. This process method wastes time and resources. TDD addresses this problem and, in addition, TDD thinking suits Efore Software Development process excellently due to its developer oriented nature.

TDD is agile software development method. The main characteristic of TDD is that unit tests are written before the actual code. The benefit in this reverse thinking is that it forces the developer to plan the unit and understand the specification completely before coding. This way code is more optimised and defect injections from poor ad hoc planning are prevented. When every unit is tested and planned individually, the problems in integration phase are also minimised. The unit and integration tests are performed frequently during the coding phase, thus ensuring rigorous and comprehensive testing. (5, pp. 178-179)

Other benefits of TDD are that it offers very high test coverage and leads to modular design (8). In other words, TDD results in high quality basic modules that help establishing high quality system that requires less high-level testing, thus having positive influence on total project time (1, pp. 45-46). These modules can be safely reused because they are confirmed to be working, thanks to existing documented and repeatable unit tests. This can be used to reduce redundant coding of recurrent functions in later projects.

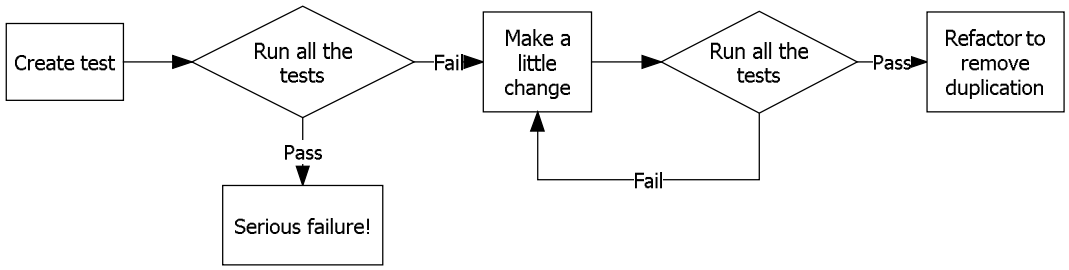


Figure 7. The TDD cycle

The TDD cycle consists of five phases which are illustrated in figure 7. The cycle starts by creating the test for the unit that is developed. Then *all* the tests, including the newly created test and all the previous tests, are run. A test pass is considered a serious failure since the newly created test cannot pass because there is no coding done yet for the test to pass. It can be said that the test is initially created to fail. The third phase is that a small change is made and the tests are run again. Finally, when the tests pass, the code is refactored. (8)

The TDD cycle is designed to be as short as possible. The aim is to develop the code in as small fractions as possible. It is inevitable that TDD slows down the coding process, but the amount of time saved in later testing phases, debugging and code fixing is unsurpassed, not to mention the increase in quality and reliability. The fast paced cycle of TDD also ensures that defect lifecycle is short since most of the defects are fixed almost immediately after injection. This is as close as ideal situation as one can get.

During the development phase, the lifecycle of TDD is less structured than it is for independent test team. The general test plan is established and the test cases are developed on basis of the test plan. Because test cases are designed before the UUT, the unit test cases are exceptionally designed using black box methods only. The testing is controlled so that every TDD cycle produces a set of deliverables: the test software & scripts and a test document that includes the test cases and the test results. If changes are done in a module that has already been tested, all the tests shall be performed again and new test deliverables shall be established. (1, pp. 50-53)

The same lifecycle strategies apply for TDD development that apply to incremental unit testing. The code can be developed in top-down and/or bottom-up method where the TDD cycle is integrated into the original incremental method. The operations of embedded systems rely heavily on the real-time events in the plant and the embedded environment itself. Therefore it is advisable to favour bottom-up method for TDD, since writing stubs simulating the real-time events is difficult and even redundant when the complexity of stubs approach the complexity of the module they are simulating. Nevertheless, the structure and functions of the system are the main features that dictate the best TDD approach for the project.

There are some drawbacks on the TDD method. One problematic feature is the requirement of complete dedication from the developer. As mentioned in chapter 3.2, writing comprehensive test cases require certain “destruction oriented” mind-set that differs greatly from the mind-set of a software developer. Adapting to the dualistic feel of the process requires time and patience from the developer and during this adaptation period productivity can be temporarily reduced. In addition, some software structures can be challenging to write in module oriented method, which can make writing test cases difficult or even impossible.

4.4 Tailored Software Testing for Efore

The most essential action that has to be taken in order to implement software testing to Efore is to create the base frame for the software testing process. This is accomplished by following the TMMi method directives. Without working infrastructure and controlled process, all software testing improvement efforts are futile. Therefore the first initial step is to implement software test policy and software test strategy according to TMMi Level 2. The guidelines for those are discussed in chapter 4.1 and in further detail in TMMi Version 3.1 document available from The TMMi Foundation.

The TMMi Level 2 introduces software test planning to Efore Software Development process. The test plan further defines software test monitoring and control measures, which include frequent static reviews during the software development process and documenting the testing process.

The whole process is structured to spiral model method where the Software Development lifecycle is divided into iterative cycles. These cycles are used to control the SW development process and add flexibility into the lifecycle. The cycles include frequent reviews from which review deliverables are generated. These deliverables are used to evaluate the progress of the project.

Test Driven Development and peer reviews are introduced as general practices in the coding cycle and the deliverables are created according to the test plan. The low level testing is developer oriented and the final verification and acceptance testing is performed by an external tester.

Methods and emphasis of test case development is guided by test design and execution directives of TMMi. All the tests are performed in the test environment defined by TMMi. The test environment not only contains the actual hardware required for testing, but also the database for the test deliverables and other testing related documents.

With these measures, rigorous software testing process is introduced to Software Development process and the TMMi maturity level can be raised from initial Level 1 to structured Level 2. This allows further continuous improvement of software testing and software quality control.

The testing intense project lifecycle requires changes to the current project lifecycle model (appendix 2). The proposed new project lifecycles are presented in the appendix 3.

5 Testing Platform

5.1 Overview

In order to extensively test embedded systems and the peripheral functions, the target environment must be simulated. The simulation can be performed in software or hardware.

For this project a hardware simulation was chosen due the diverse number of different microcontrollers and need for mixed signals. With hardware testing platform it will be possible to simulate the real target environment and perform unit and integration tests in the target processor. This allows rigorous testing of the software before the first hardware prototype and even makes possible to perform software/hardware integration tests.

To simulate the plant, the platform must be able to generate analogue voltages up to +5V, receive analogue signals, handle up to 50 digital Input/Outputs, generate PWM signals and communicate via SPI, I2C and UART protocols.

Functions of the platform are controlled by a separate microprocessor, which is scripted by a host computer via USB. The platform can generate test signals and receive analogue and digital signals as a result. In addition, the platform can communicate with the processor board via UART (Universal Asynchronous Receiver Transmitter), I2C or SPI (Serial Peripheral Interface) bus. The outputs can be scripted in order to fully automatize the testing process and signal generation. This way the tests are always identically repeatable and the results can be gathered automatically.

5.2 Hardware Implementation

The testing platform consists of a motherboard which contains basic features required for the testing, such as

- mbed NXP LPC1768 for data gathering and control functions
- 64-bit digital programmable Input/Output

- total of 12 channel Digital-to-Analogue converters (DAC) for test signal generation
- six Analogue-to-Digital converters (ADC)
- five Pulse Width Modulation (PWM) outputs
- connector for a fan
- voltage regulator and other operating voltage related circuitry
- eight indicator LEDs for digital signals
- connectors for a processor board
- test pins for measurement equipment connections
- battery for real-time clock backup

The platform is shown on figure 8. The core of the testing platform is mbed NXP LPC1768 rapid prototyping miniature development board. The NXP LPC1768 core is 32-bit Cortex-M3 ARM running at 96 MHz clock with 512kB Flash and 64kB RAM. The board offers USB connection for PC. The LPC1768 contains integrated peripherals such as SPI, UART, CAN and I2C buses, ADC and DAC converters and PWM outputs. (9)

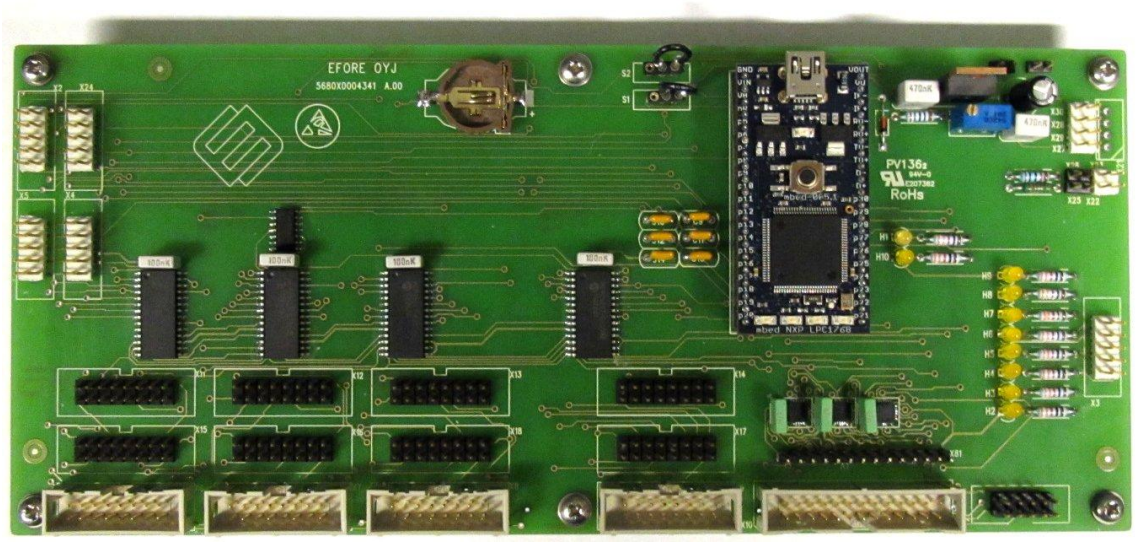


Figure 8. Software testing platform

Due to the restricted number of I/O pins on the mbed the digital I/O is implemented with Microchip Technology Inc. 16-bit I/O expansion chips MCP23S17. The platform contains total of four MCP23S17s offering the total of 64 bit digital I/O. The chips are

controlled via fast 10 MHz SPI bus. The chips contain interrupt on change functionality which is used to read the inputs when a change in inputs occurs. (10)

The Digital-to-Analogue functionality is implemented with Maxim Integrated Products' four channel 12-bit DAC chips MAX5135 (11). The motherboard contains three MAX5135s. The DACs are connected to the same 10 MHz SPI bus as the I/O expansions. For Analogue-to-Digital conversions mbed NXP LPC1768's internal 12-bit ADCs are used.

The motherboard block diagram can be found in appendix 4. All the connectors for the processor board are standard male IDC connectors. All outputs are divided in separate connectors by type, except digital I/O ports which are further divided in separate 16-pin connectors with two ports each. Most of the ports also contain test pins for external measurement equipment connections. The board contains a battery backup for mbed real-time clock.

The processor boards contain the actual target processor and connectors for the processor I/O, programming connector and other processor and application specific hardware.

5.3 Software Implementation

The software is written in C/C++ and compiled with Code Sourcery G++ Lite 2011.03-42. The software offers automated output control by scripting and a manual mode for controlling the outputs directly from console. It generates test logs automatically for further processing and documentation.

5.3.1 Automatic Mode

The script format is designed to be easily generated with Excel. The log file format is also designed to be effortlessly pasted to Excel and displayed in graphical format. Every I/O function can be scripted and the inputs are logged.

Script files can be commented in C style by marking the comment lines with double-dash. Every output is scripted in individual script blocks separated by a blank line. The block can contain one to 200 lines of script, depending on the type of script. The scripts can be divided into two categories: signal scripts and configuration scripts. Intuitively, control scripts are used to generate test signals for the target board and configuration scripts are used to setup the peripheral functions, e.g. communication ports and digital I/O data directions. Every block starts with a definition line that defines what is scripted. The order of script blocks is not restricted, but for clarity it is advisable to write scripts in pairs, e.g. data direction script is followed by related I/O signal scripts. An example script is shown in appendix 5.

The first script block on the example is a single line configuration block for digital I/O port B data directions. The format for data direction definition is `DDN` where N represents the alphabetical symbol of the digital I/O port. The definition is followed by tab and 8-bit data direction value in hexadecimal. Logical 1 presents an input and logical 0 presents an output. Every pin is an output as a default.

The data direction block is followed by port B signal script block. Port script is defined by `POutN` where N represents the alphabetical symbol of the digital I/O port. The script data starts from the next line. The data is structured in two columns. The first column contains time values in seconds and the second column contains the port value in hexadecimal. The columns are separated with tab. This structure is identical to all signal scripts.

Digital I/O pins can also be scripted individually. Individual scripting overrides port scripts for the individually scripted pins. The format is `DOutN` where the N represents the pin number from 0 to 63.

The next script block contains a definition of DAC reference voltage that is used in the platform hardware. It is followed by a signal script for analogue out 3. Voltage script is defined by `VOutN` where N represents the DAC channel number. The values for analogue outputs are scripted in volts. The transitions in voltage scripts are not instantaneous. The script points can be considered to be points of piecewise defined continuous linear function. The voltage run of the example script is illustrated in figure 9.

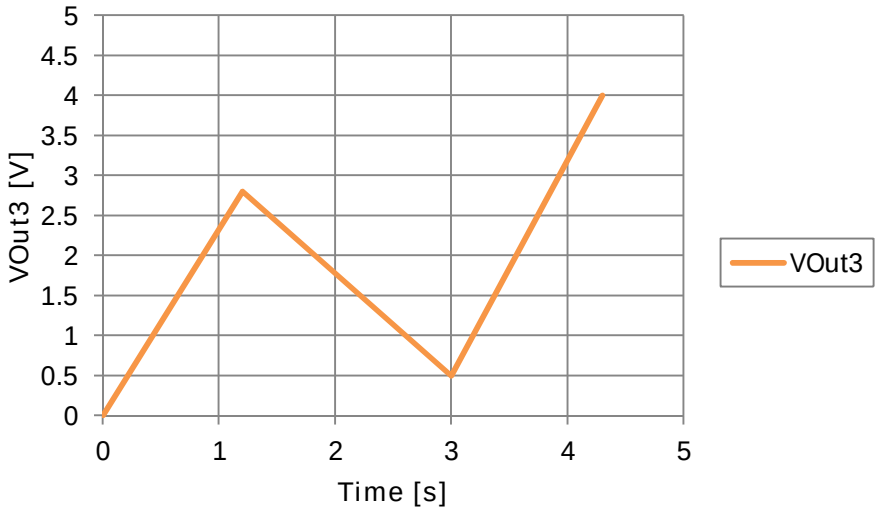


Figure 9. Voltage generated in VOut3 by the example script

The next block is a single line configuration script of PWM frequency. The block is defined by `PWMF`. The definition is followed by tab and frequency in Hertz. PWM frequency is global and same for all PWM outputs.

The PWM signal script block is defined by `PWMN` where N represents the number of PWM channel. The data on the PWM is the duration of high state in percentage contrast to the whole cycle duration. Therefore 0 generates constant low, 1 generates constant high and 0.5 generates even 50% square. The PWM pulse width is swept linearly in the same way as voltage.

The script file is read to and ran from RAM. Memory is dynamically allocated to ensure efficient memory usage. In theory, maximum simulation run duration is over 10 days.

Every automated test run generates two log files that contain the test results. The first log file is simple easy-to-read log that contains all the events that happened in the inputs during the test run. The second log file contains preformatted data where the event data can be copy pasted into Excel spread sheet for graphing. The detailed descriptions of automatic script commands are found on appendix 6.

5.3.2 Manual Control Mode

Every output can be controlled manually from the console by entering in manual control mode. The commands are similar to the automatic script commands. The same commands work in manual mode, but also shortened versions are available. The commands are non-case-sensitive. While in the command mode, all other manual control commands are disabled.

The logic behind the commands is identical to the script commands in automatic mode described in previous chapter 5.3.1. Unlike the multi lined structure of the scripts, manual control commands are single line.

Analogue outputs are controlled by `VN x`. The voltage change in manual mode is immediate. Digital outputs can be controlled either pinwise with `DN x` commands or by portwise with `PN x` commands. The port values are inputted as hexadecimal. PWM is controlled by `PWMN x`. The PWM frequency command is identical to the command in automatic mode.

While in the manual mode, the user can enter in Comm mode which contains all the communications functionality. The communications functions are in experimental stage since there has not been a chance to test and debug them. Thus, the functionality has not been confirmed on actual products.

The Comm mode can be entered with `Comm` command. The communications port must be configured in order to use Comm mode functions. This is done with `Configure` command. Every communications port type needs different configuration values. The commands are:

- Configure UART <baud_rate> <bit_length> <parity> <stop_bits>
- Configure SPI <baud_rate> <bit_length> <polarity_mode>
- Configure I2C <baud_rate> <address> <slave/master>

Baud rates, message bit lengths and number of stop bits are intuitive. The parity on UART is defined by an integer from 0 to 4, where 0 = None, 1 = Odd, 2 = Even, 3 =

Forced 1 and 4 = Forced 0. The polarity mode in SPI configuration is also an integer from 0 to 3. The polarity functionality is described in table 2.

In the I2C configuration address is the slave address of the testing platform in hexadecimal. It can be left to 0 if the device acts as a master. The slave/master bit selects the type of communication where 0 = master and 1 = slave.

Table 2. SPI polarity mode descriptions (12)

Polarity mode	When first data bit is driven	Other data bits are driven	When data is sampled
0	Prior to first SCK rising edge	SCK falling edge	SCK rising edge
1	First SCK rising edge	SCK rising edge	SCK falling edge
2	Prior to first SCK falling edge	SCK rising edge	SCK falling edge
3	First SCK falling edge	SCK falling edge	SCK rising edge

To send data to the DUT TX command is used. For SPI communications the command is followed by maximum of 127 values in hexadecimal. Length of each value is between two to four hexadecimals, depending on the data length set in the configuration.

For I2C and UART there are two transmitting options. TXC command transmits a string of characters and TXH sends raw numeric data in hexadecimals. The limit is 127 values; two hexadecimals per value.

Data received via UART is displayed immediately. When the platform acts as a master for I2C or SPI bus, the receiving command must be used. For SPI the command is simply RX x and for I2C the commands are RXC x for character data and RXH x for numeric data. The x represents the number of values read from the DUT.

While in manual mode, changes in digital inputs or ADC limit alarm triggers an interrupt, and the state of the outputs is displayed immediately on change. Manual mode is ideal for quick tests and communicating with the unit via Comm mode, but it should not be used for actual tests due the lack of repeatability. The detailed descriptions of manual mode commands are found on appendix 7.

6 Testing in Practice

The concept and functionality of the testing platform was tested on a set of test cases of an on-going project.

6.1 The Processor Board

The first step was to build the processor board for the target processor. The board was built on a breadboard, as displayed in figure 10. The analogue outputs, digital I/O and communication port were wired to separate pin header connectors. The processor board contained also a test pins for PWM output, PWM input and power connections.

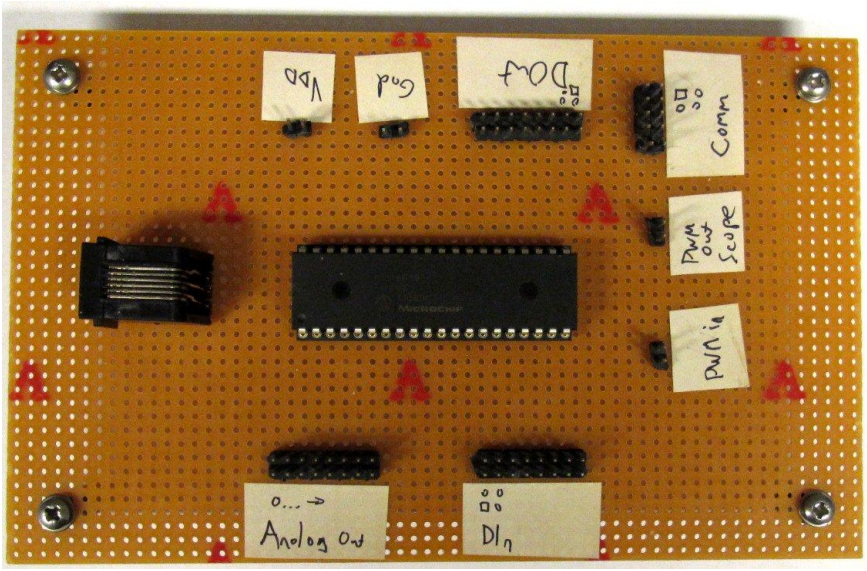


Figure 10. Processor board for the UUT

In addition, the board included RJ11 connector for programmer/debugger. This connector was solely used for programming the blank target MCU with the binary image.

6.2 Example Test Case

One of the tests performed concerned a hysteresis of operation modes depending on the current flow through the UUT. The current is an analogue signal from 0V to +5V. The Unit is defaulted to state A at the power on. The unit should enter state B when

the current drops to -0.5A or below and return back to the state A when the current is 0.5A or greater.

In addition to monitoring current, the unit measures two voltages. To avoid simulating abnormal behaviour, the voltages are also scripted so that the voltage potential corresponds with the hysteresis trigger current.

The example script can be found in appendix 8. The script is divided into three script groups: Configuration scripts, dynamic scripts and static or irrelevant scripts. This makes it easier to read the script and separate signals that are essential for the current test from the signals that are irrelevant for the result.

The first group is configuration group where the basic configuration is made. The reference voltage for DACs is set to +5V by Vref command. Next, the input and output ports are defined. The Processor board is connected to ports A, C and D. Without data direction definition the port defaults to all outputs, but every port is defined for clarity. The unit also monitors fan speed via pulse capture. This is emulated in the platform with PWM output, which frequency is set to 10Hz.

The next group is dynamic script group. This group contains all signals that are mentioned in the test case and/or that change during the test run. As mentioned before, the test contains three main signal inputs: current and two voltages. The actual voltage and current values that the simulated signals represent in the HW environment are commented in the script to make the script more readable. In addition to the analogue input signals the processor is reset at the beginning of the test to ensure the software is in the right mode for the test.

The last group is for static or irrelevant scripts. These are signals that are not mentioned in the test script. These signals can be e.g. external control signals that are used to switch the software in special mode or static analogue signals, like temperature sensor readings.

6.2.1 Expected Result

At the beginning the unit enters a self-test mode due reset. A LED signal connected to platform input Port A pin 5 shall first flicker in two phases and then go off to signal that the self-test is complete. When the current drops below -0.5A or more, the LED should light up and stay lit until the current exceeds 0.5A when the LED should go off.

6.2.2 Measured Results

The test was run successfully and the log files were generated. A shortened version of the generated log is in appendix 9. The log is shortened by 300 lines from the middle, but the essential part is included. The visualisation of the test signals and digital input, Port A, is included in the appendix 10. The essential graphs for the test are illustrated in Figure 10.

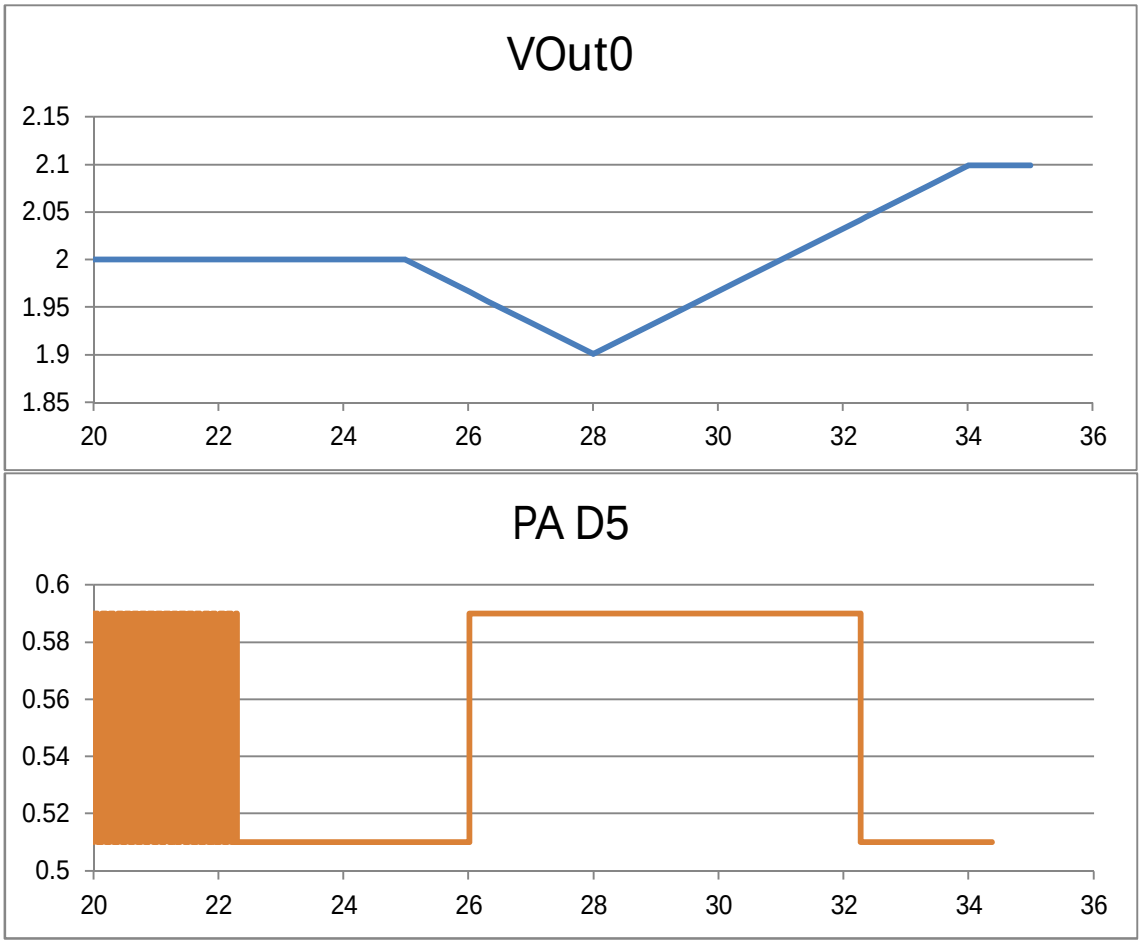


Figure 11. Essential test graphs

The transition points of the led can be deciphered from the graphs. The exact time of each transition can be acquired by hovering mouse cursor over the transition points in Excel. The transition points, 26.0184 seconds and 32.2708 seconds, are marked green in the test log in appendix 10. There the exact voltage values for the current, 1.966V and 2.042V, can be read.

According to the SW-HW interface specification of the unit the ADC voltage value for -0.5A is 1.975V and for 0.5A 2.025V. Therefore, according to the measured results, the hysteresis is greater than -0.5A to 0.5A and the test is a pass.

7 Conclusions

There were two goals for this thesis: to develop and tailor embedded software testing methods for Efore SW development team and develop tools to carry out the testing. The most interesting finding was that software testing of low level embedded systems is surprisingly young art of engineering and it is still in rather immature state. Many resources even stated embedded software testing being mostly neglected in electronics industry. This was a very surprising and somewhat concerning finding considering that embedded systems have been around for over two decades and in modern electronics the software is the heart of the whole product.

There are certain difficulties in testing embedded software that makes it a bit more challenging than conventional software testing. The most critical problem is the tight dependency on the hardware environment that is developed concurrently with the software and that is often required to perform reliable software testing. Sometimes it is even impossible to test the software without custom tools, which easily makes the idea of focusing on testing in late post proto stages very tempting. This problem was taken into account in the thesis by developing the required hardware tools for the testing in addition to the testing methods and processes.

7.1 The Method

There were certain characteristics in Efore SW process that made integrating rigorous software testing to the process challenging. The main challenges were caused by the nature of the SW development projects. Instead of one or two big software unities there were multiple smaller one-developer software projects. This feature made traditional methods that relied on independent testing team somewhat inefficient because to familiarise oneself with every project and keep track of every project's progress can get difficult and needlessly time-consuming. Due to these reasons, an external test engineer may even hinder the development process on such small scale and short lifecycle projects.

Because of this, a developer oriented approach was chosen. The testing is integrated in the development process which makes the testing efficient because the tester is already familiar with the design and requirements. This somewhat increases the dura-

tion of the coding process, but takes the losses back in increased quality and drastically decreased defect lifecycle.

To aid the implementation of the new testing methods the TMMi process was chosen as vehicle for the implementation process. In addition to introducing rigorous testing to the software process, the TMMi offers a roadmap for further developing and expanding the testing process. This is an essential feature since the amount and importance of software in power products will definitely continue to increase in the future. The power electronics industry is showing signs that in the future more and more of the expensive analogue control circuits are being replaced with sophisticated digital control algorithms. This will make successful and adaptive software quality assurance essential.

7.2 The Testing Platform

The testing platform that was designed as part of the thesis proved to be suitable for pre-proto testing and the motherboard & processor board concept proved to be functional, although some technical problems were encountered during the testing.

The memory requirements of a DAQ (data acquisition) application turned out to be too much for the LPC1768. The first revisions of the software were done on mbed online compiler. In the later stages the software was modified to be compiled offline on Sourcery G++ Lite compiler. It resulted in increased memory problems due to poor optimisation and general performance of the compiler. The image size was increased from 38kB to 179kB. With tweaking the image size could be reduced to 88kB. After the porting to Code Sourcery compiler, the behaviour of standard library functions became highly unpredictable, causing random crashes during file writes very likely due to conflicts between heap and stack. The only way to avoid crashes during test runs was to perform hard reset by depowering the platform before every test run. Fixing these defects require either better processor and/or additional external RAM. Either way, a new hardware revision is required in addition to changes in the code.

References

1. Broekman, Bart and Notenboom, Edwin. Testing Embedded Software. Harlow : Pearson Education Ltd., 2003.
2. Roychoudhury, Abhik. Embedded Systems and Software Validation. Burlington, MA : Elsevier Inc., 2009.
3. TMMi Foundation. Test Maturity Model integration (TMMi) Version 3.1. [Adobe Acrobat PDF] Dublin : TMMi Foundation, 2010.
4. Eliminating Embedded Software Defects Prior to Integration Test. Bennett, Ted and Wennberg, Paul. 12, s.l. : Crosstalk, 2005.
5. Myers, Glenford. The Art of Software Testing, 2nd Edition. New Jersey : John Wiley & Sons, 2004. ISBN 0-471-46912-2.
6. Software Testing Fundamentals. [Online] 2011. [Cited: 6 June 2011.] http://softwaretestingfundamentals.com/.
7. About the Foundation. TMMi® Foundation. [Online] 11. March 11. [Cited: 29. June 2011.] http://www.tmmifoundation.org.
8. Grenning, James. Test Driven Development for Embedded Software. San Jose : s.n., 2008.
9. mbed NXP LPC1768. mbed Web site. [Online] mbed, 25. September 2010. [Cited: 11. August 2011.] http://mbed.org/handbook/mbed-NXP-LPC1768.
10. Microchip Technologies Inc. 16-Bit I/O Expander with Serial Interface. [Datasheet] 20. April 2005.
11. Maxim Integrated Products. MAX5134. [Datasheet] Sunnyvale : s.n., 2008.
12. NXP Semiconductors. LPC17xx Datasheet. [Adobe Acrobat document] 2010.

Defect and problem checklist for code reviews

Data Reference Defects

1. Referenced variable is unset or uninitialized
2. Array referenced out of bounds
3. Array referenced by non-integer
4. Pointer points to an unreferenced memory
5. Value of a variable has a type or attribute other than what the compiler expects
6. Pointer points to a different data type than is expected
7. A data structure is referenced in multiple functions, but it is not defined identically
8. Reference to array is off by one

Data Declaration Defects

1. Variable is not explicitly declared
2. Declarative initialisation of an array is wrong
3. Variable assigned to wrong data type
4. Variables have almost identical names thus increasing a risk of confusion

Computation Defects

1. Computations have non-arithmetic data types
2. Computation is performed between different data types without type-casting
3. Computation is performed between the same data types but with different data lengths
4. Computation result exceeds the range of the result variable
5. Computation result is in the range of the result variable, but an overflow or underflow is possible during the computation
6. In division computation nominator can be zero
7. Variable value goes beyond meaningful range (e.g. probability is negative)

8. On expressions containing more than one operator assumption about the order of operator execution is incorrect
9. Invalid use of integer arithmetic (e.g. in expression using integer division)

Comparison Defects

1. Compared variables have different data types
2. Mixed-mode comparison with different data lengths
3. Comparison operator is incorrect
4. Boolean expression does not state what it should state
5. Operands of Boolean operation are not Boolean; comparison and Boolean operators are erroneously mixed
6. On expressions containing more than one Boolean operator assumption about the order of evaluation and execution is incorrect

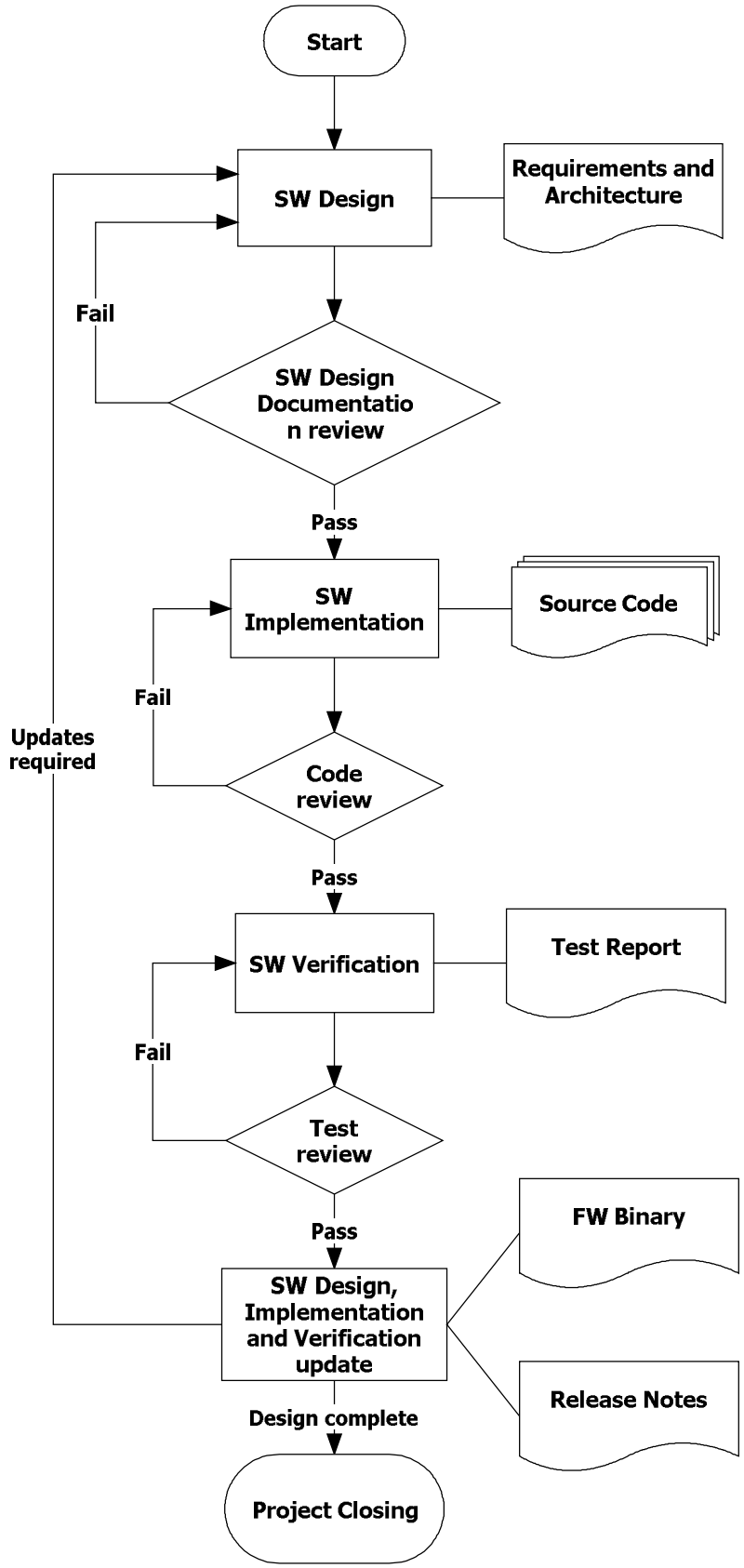
Control Flow Defects

1. A loop will erroneously never terminate
2. Function will never terminate
3. A condition or loop will never execute due the conditions
4. An iteration loop has off-by-one error
5. Loop or condition decision is non-exhaustive (e.g. expected values are 1, 2 and 3 and if the value is not 1 or 2 the program assumes it is 3)

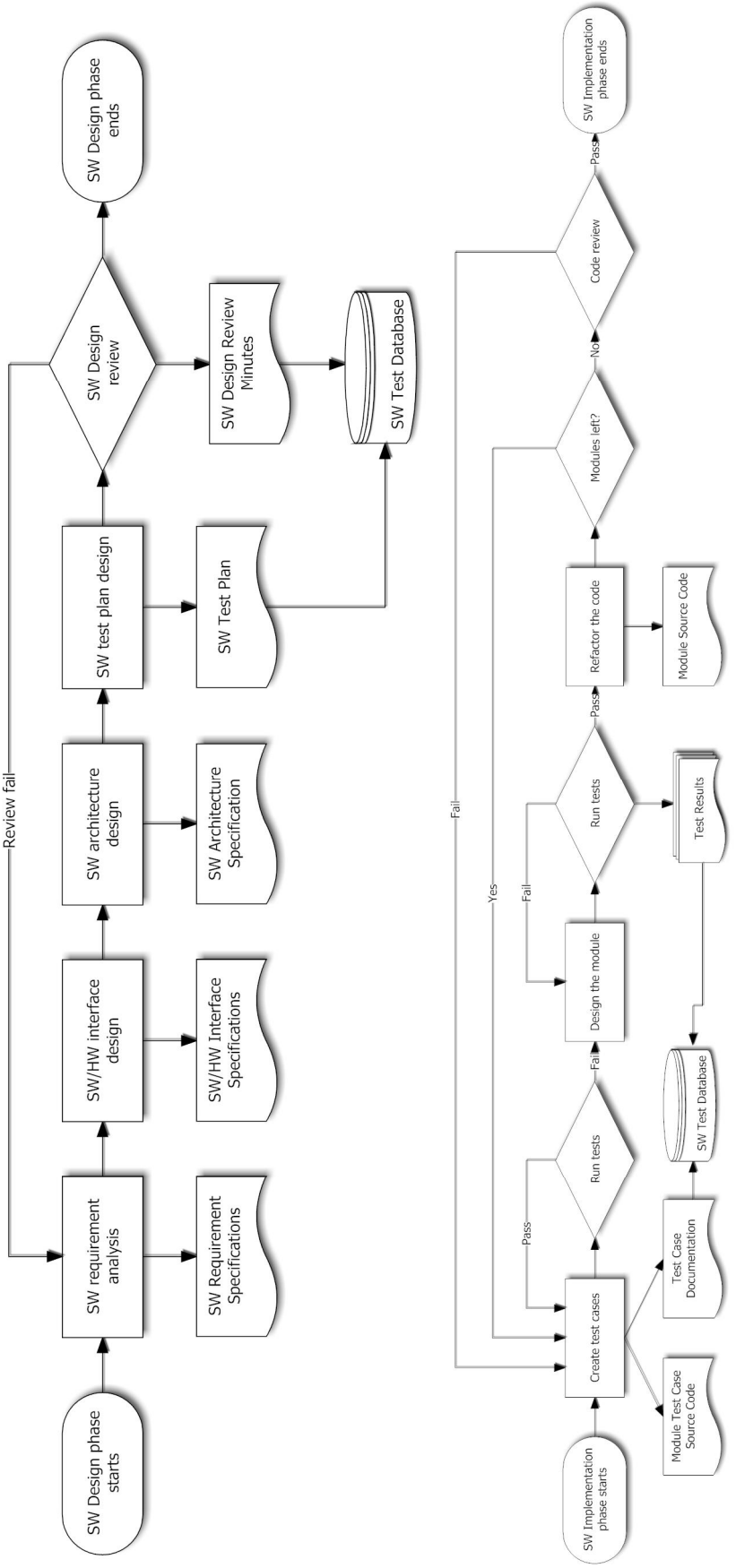
Interface Defects

1. Parameters sent to a function are in wrong order
2. Units system of parameters does not match (e.g. voltage is sent to function in volts but function expects millivolts)
3. A function has multiple entry points and it references a variable that is not initialised in all entry points
4. A function alters a parameter that is intended to be only an input value

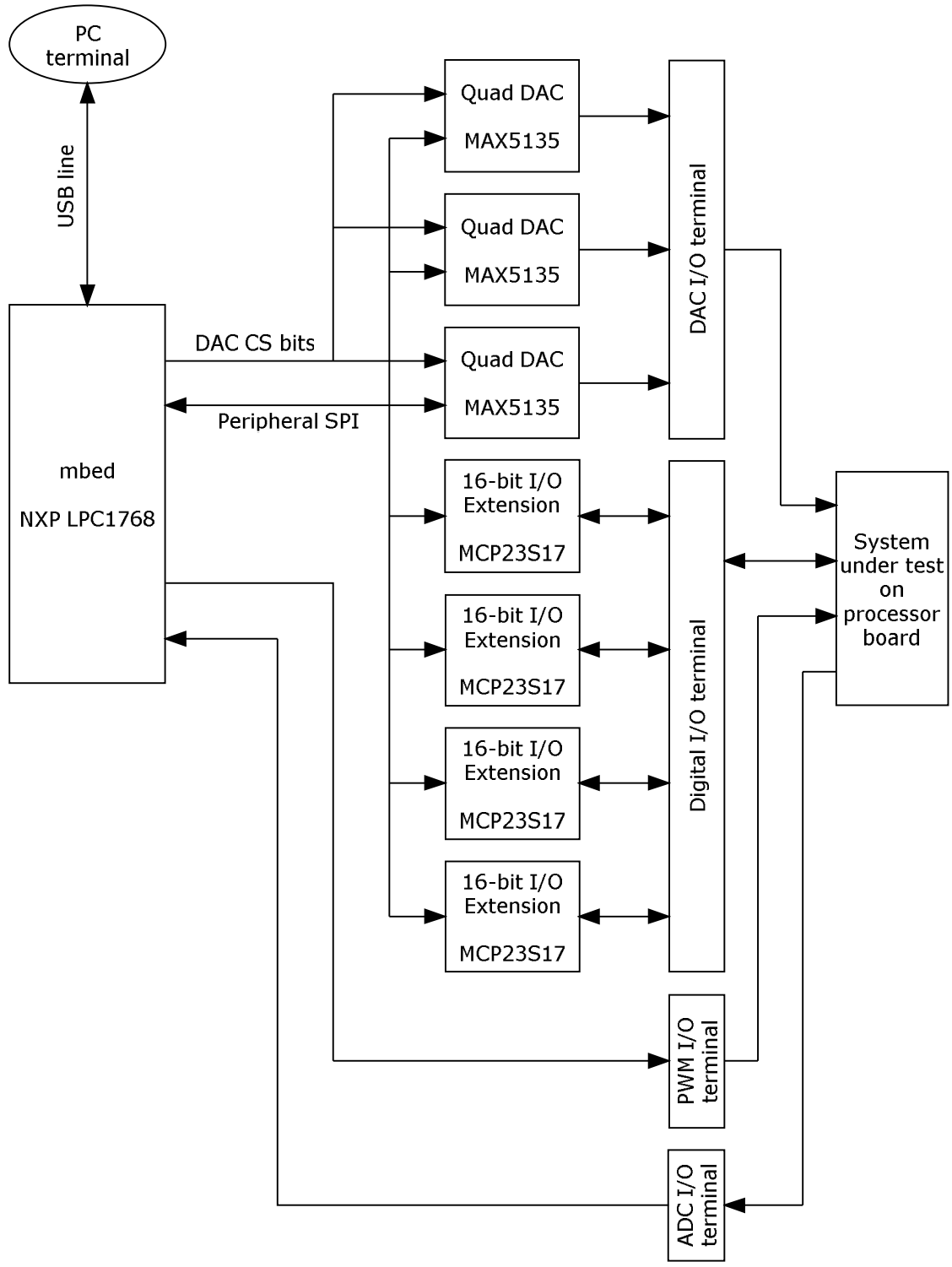
Software development process



Proposed software development process



Software testing platform block diagram



Test script demonstration

```
// An example script file “example”
// 2011-08-01

// Script for Port B data direction
DDB       0xA0

// Script for digital port B in hex
POutB
0         02
0.2       32
2.8       10
3.5       2F

// Script for digital pin 40
DOut40
1         0
1.5       1
1.7       0
2.2       1

// Reference voltage is +5V
Vref      5

// Script for voltage 3
VOut3
0         0
1.2       2.8
3         0.5
4.3       4

// Script for PWM frequency
PWMF      1000

// Script for PWM output 1
PWM1
0         0.5
1.5       0.8
3         0.2
```

Automatic script syntax

Analog output script	
Syntax	Description
VOut<channel> <time_0> <value_0> <time_1> <value_1> … … <time_n-1> <value_n-1> <time_n> <value_n>	Channel is an integer between 0-11. Time steps must be in chronological order. Values are floating point numbers between 0 and 1, which represent the percentage of the reference voltage of the DAC. The value sweeps linearly between the script points.
Analogue output reference voltage	
Syntax	Description
Vref <value>	The value is reference voltage of the DAC on the hardware.
Digital port data direction script	
Syntax	Description
DD<port> <value>	Port is a letter between A-H. Value is a 8-bit hexadecimals between 00 and FF representing the I/O configuration where logical 1 is an input and 0 an output.
Digital pin output script	
Syntax	Description
DOut<pin> <time_0> <state_0> <time_1> <state_1> … … <time_n-1> <state_n-1> <time_n> <state_n>	Pin is an integer between 0-63. Time steps must be in chronological order. States are either 0 or 1.
Digital port output script	
Syntax	Description
POut<port> <time_0> <value_0> <time_1> <value_1> … … <time_n-1> <value_n-1> <time_n> <value_n>	Port is a letter between A-H. Time steps must be in chronological order. Values are 8-bit hexadecimals between 00 and FF.
PWM frequency script	
Syntax	Description
PWMF <frequency>	Frequency is an integer in hertz from 1 to 48000000.
PWM output script	
Syntax	Description
PWM<channel> <time_0> <value_0> <time_1> <value_1> … … <time_n-1> <value_n-1> <time_n> <value_n>	Channel is an integer between 0-4. Time steps must be in chronological order. Values are floating point numbers between 0 and 1, which represent the percentage of the high pulse duration to the whole cycle duration. The value sweeps linearly between the script points.
Voltage input alarm script	
Syntax	Description
VIn<channel> <level>	Channel is between 0-5. Level is a floating point number between 0 and 1 representing the alarm level in percentage of 3.3 Volts that triggers alarm.

Manual command syntax

Analog output command	
Syntax	Description
`V<channel> <value>`	Channel is an integer between 0-11. Values is a floating point number between 0 and 1, which represent the percentage of the reference voltage of the DAC. The voltage is set immediately.
Digital port data direction command	
Syntax	Description
`DD<port> <value>`	Port is a letter between A-H. Value is a 8-bit hexadecimals between 00 and FF representing the I/O configuration where logical 1 is an input and 0 an output.
Digital pin output command	
Syntax	Description
`D<pin> <state>`	Pin is an integer between 0-63. State is either 0 or 1.
Digital port output command	
Syntax	Description
`P<port> <value>`	Port is a letter between A-H. Value is a 8-bit hexadecimal between 00 and FF.
PWM frequency command	
Syntax	Description
`PWMF <frequency>`	Frequency is an integer in hertz from 1 to 48000000.
PWM output command	
Syntax	Description
`PWM<channel> <value>`	Channel is an integer between 0-4. Value is a floating point number between 0 and 1, which represents the percentage of the high pulse duration to the whole cycle duration. The pulse width is set immediately.
Voltage input alarm script	
Syntax	Description
`VIn<channel> <level>`	Channel is between 0-5. Level is a floating point number between 0 and 1 representing the alarm level in percentage of 3.3 Volts that triggers alarm.
Communications mode command	
Syntax	Description
`Comm`	This command enters the communications mode.
UART configuration command	
Syntax	
`Configure UART <baud_rate> <data_length> <parity> <stop_bits>`	
Description	
Baud rate is an integer between 1 and 1000000. Data length defines the number of bits in one data packet. Parity is an integer between 0-4 that defines the parity: 0 = None, 1 = Odd, 2 = Even, 3 = Forced 1 and 4 = Forced 0	

SPI configuration command	
Syntax	
`Configure SPI <baud_rate> <data_length> <polarity_mode>`	
Description	
Baud rate is an integer between 1 and 20000000. Data length defines the number of bits in one data packet. Polarity mode defines when data is transferred. This depends on the Clock Polarity (CPOL) and Clock Phase (CPHA): Polarity mode = 0 CPOL = 0, CPHA = 0 Polarity mode = 1 CPOL = 0, CPHA = 1 Polarity mode = 2 CPOL = 1, CPHA = 0 Polarity mode = 3 CPOL = 1, CPHA = 1	
I2C configuration command	
Syntax	
`Configure I2C <baud_rate> <slave_address> <slave/master>`	
Description	
I2C supports standard baud rates of 100000 and 400000. Slave address is used to determine the address of the platform when used in slave mode. Slave/master bit activates slave mode: 0 = master, 1 = slave.	
UART & I2C transmit hexadecimal data command	
Syntax	Description
`TX <data_string>`	Data string can be up to 127 values long, two hexadecimals per value.
I2C receive data	
Syntax	Description
`RX <data_values>`	Data values define the number of characters read from the I2C bus.
SPI transmit data command	
Syntax	Description
`TX <data_string>`	Data string can be up to 127 values long, two to four hexadecimals per value depending on the data length configuration.
SPI receive data command	
Syntax	Description
`RX <data_values>`	Data values define the number of values read from the SPI bus.

```
//*****************************************************************************
//
//  COPYRIGHT (C) 2011 EFORE OYJ, EFORE PLC
//
//*****************************************************************************
//
//  Project     : X
//  Description : X
//  Revision    : X
//
//  Test      : X
//  Author      : Juho Lepistö
//  Date      : 2011-10-19
//
//*****************************************************************************



//*****************************************************************************
//  Configuration Scripts
//*****************************************************************************


// Reference to 5 V
Vref          5

// Data directions
DDA           FF

DDC           01

DDD           00

// FAN RPM
PWMF          10


//*****************************************************************************
//  Dynamic Scripts
//*****************************************************************************


// Reset active low
// Reset MCU and start after one second
DOut26
0             0             // Reset
1             1
35            0             // Reset

// Voltage A
VOut2
0             2.889         // -52V
27            2.889
28            3             // -54V

// Voltage B
VOut3
0             3             // -54V
27            3
28            2.889         // -52V

// Current
VOut0
0             2             // 0 A
25            2
28            1.901         // -2 A
34            2.099         // 2A

// Current, large scope
VOut1
0             2             // 0 A
```

```
//*****************************************************************************
//  Static or irrelevant scripts
//*****************************************************************************


// ********** Digital outputs ********** //


// Control signal 1, active low
DOut17
0            1

// Address bit 1
DOut18
0            0

// Address bit 2
DOut19
0            0

// Address bit 3
DOut20
0            0

// Address bit 4
DOut21
0            0

// Alarm out 1
DOut22
0            0

// Alarm out 2
DOut23
0            0

// Control signal 2
DOut24
0            0

// Control signal 3
DOut25
0            0

// Control signal 4
DOut27
0            1


// ********** Analogue outputs ********** //

// Temperature A
// Static 27 celcius
VOut4
0            3

// Temperature B
// Static 27 celcius
VOut5
0            3

// 2.5V reference
// Static 2.5 V
VOut6
0            2.5

// Fan PW
PWM0
0            0.2
```

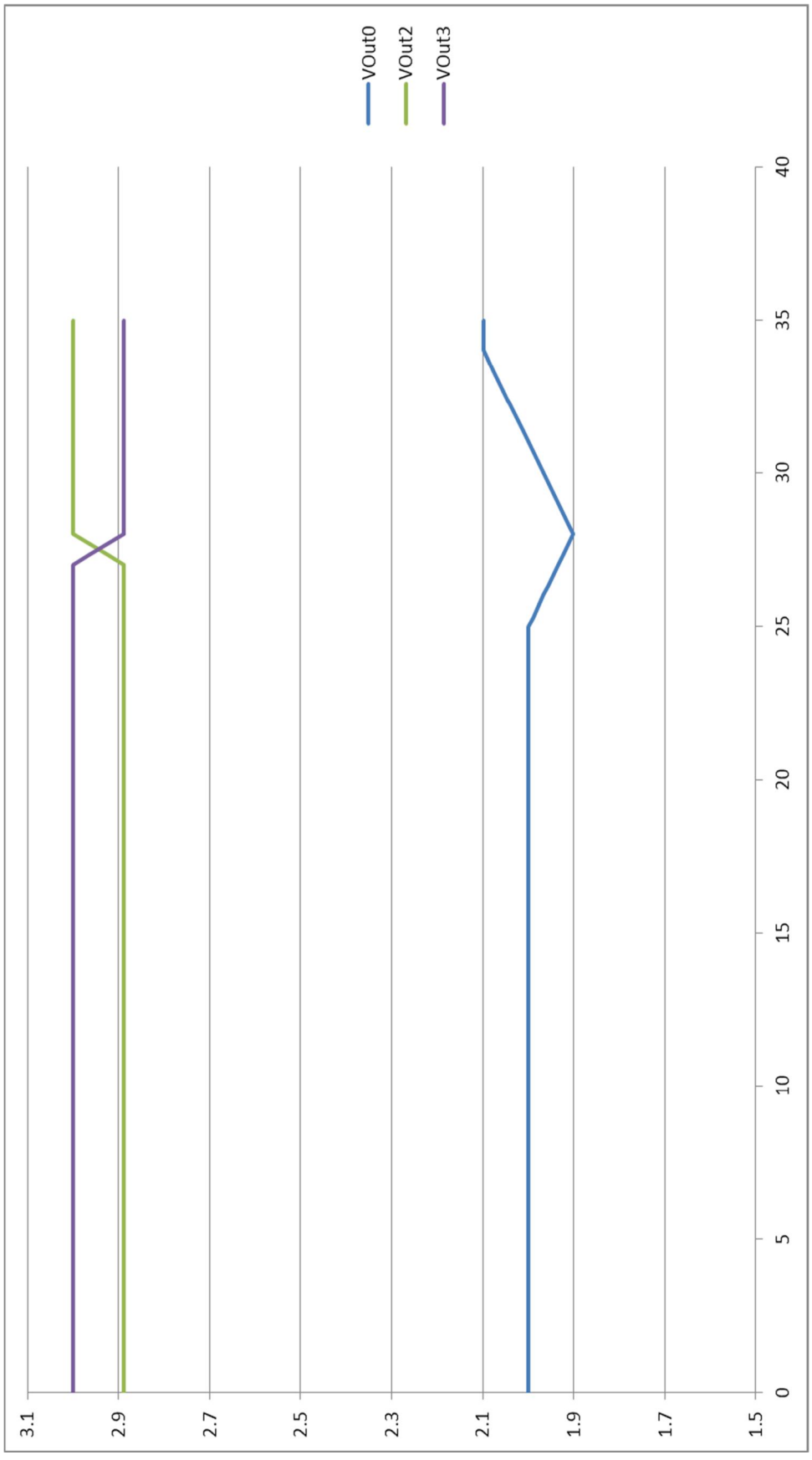
Example test log

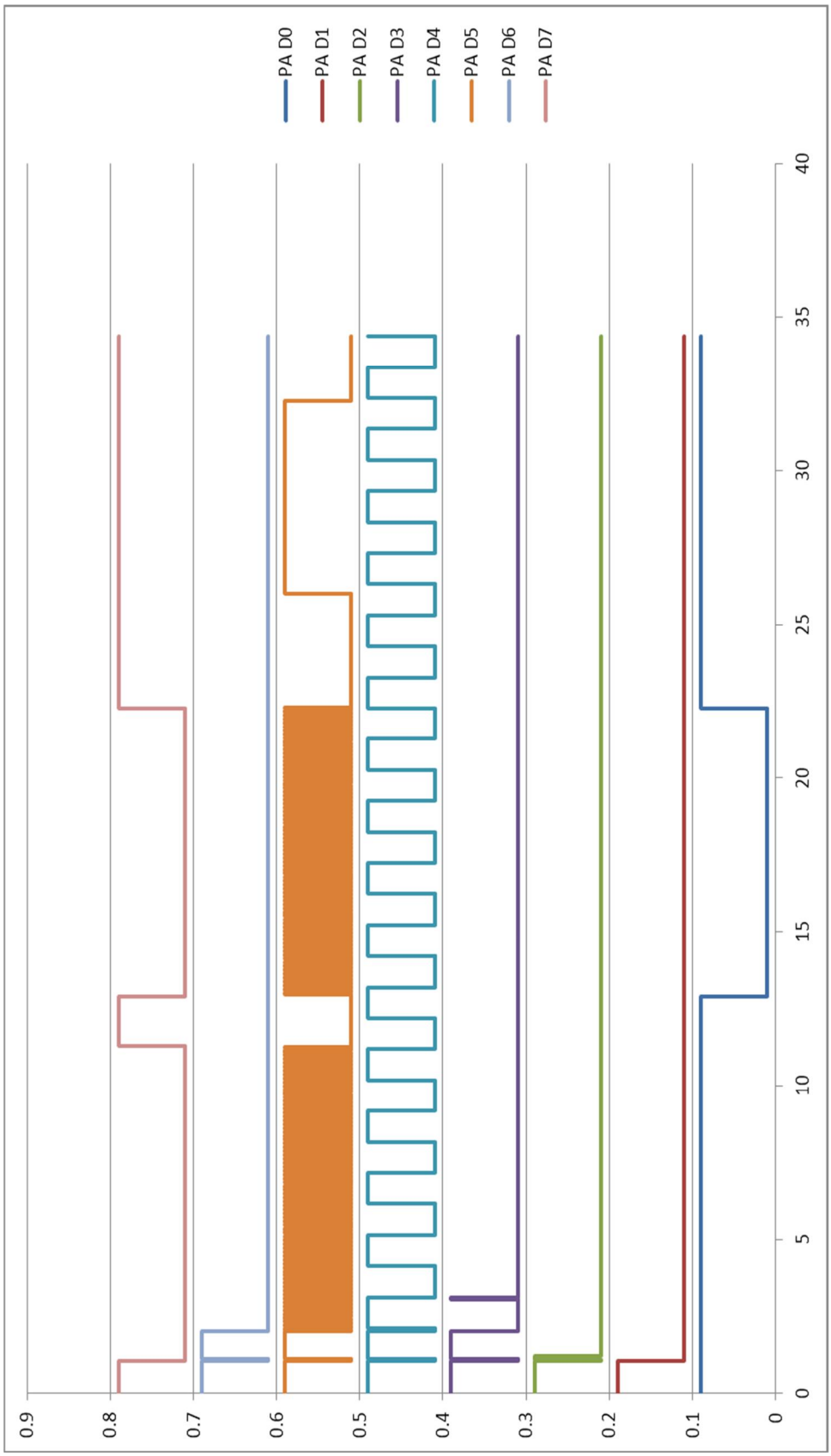
Test run log file /local/example.log

Date: 2011-10-20 13:07;39

Time	VOut0	VOut1	VOut2	VOut3	VOut4	VOut5	VOut6	PWM0	PA	PC	PD
0	2	2	2.889	3	3	3	2.5	0.2	0xFF	0x03	0x08
1	2	2	2.889	3	3	3	2.5	0.2	0xFF	0x03	0x0C
1.0642	2	2	2.889	3	3	3	2.5	0.2	0x01	0x02	0x0C
1.0938	2	2	2.889	3	3	3	2.5	0.2	0x05	0x02	0x0C
1.1026	2	2	2.889	3	3	3	2.5	0.2	0x7D	0x02	0x0C
1.2044	2	2	2.889	3	3	3	2.5	0.2	0x79	0x02	0x0C
2.0098	2	2	2.889	3	3	3	2.5	0.2	0x01	0x02	0x0C
2.0398	2	2	2.889	3	3	3	2.5	0.2	0x21	0x02	0x0C
2.0996	2	2	2.889	3	3	3	2.5	0.2	0x01	0x02	0x0C
2.1114	2	2	2.889	3	3	3	2.5	0.2	0x11	0x02	0x0C
2.1614	2	2	2.889	3	3	3	2.5	0.2	0x31	0x02	0x0C
2.2322	2	2	2.889	3	3	3	2.5	0.2	0x11	0x02	0x0C
...	...	...	...	...	...	...	...	...	...	...	...
22.298	2	2	2.889	3	3	3	2.5	0.2	0x91	0x02	0x0C
23.2864	2	2	2.889	3	3	3	2.5	0.2	0x81	0x02	0x0C
24.2948	2	2	2.889	3	3	3	2.5	0.2	0x91	0x02	0x0C
25	2	2	2.889	3	3	3	2.5	0.2	0x91	0x02	0x0C
25.3032	1.99	2	2.889	3	3	3	2.5	0.2	0x81	0x02	0x0C
26.0184	1.966	2	2.889	3	3	3	2.5	0.2	0xA1	0x02	0x0C
26.3118	1.957	2	2.889	3	3	3	2.5	0.2	0xB1	0x02	0x0C
27	1.934	2	2.889	3	3	3	2.5	0.2	0xB1	0x02	0x0C
27.3202	1.923	2	2.925	2.965	3	3	2.5	0.2	0xA1	0x02	0x0C
28	1.901	2	3	2.889	3	3	2.5	0.2	0xA1	0x02	0x0C
28.3286	1.912	2	3	2.889	3	3	2.5	0.2	0xB1	0x02	0x0C
29.3372	1.945	2	3	2.889	3	3	2.5	0.2	0xA1	0x02	0x0C
30.3456	1.978	2	3	2.889	3	3	2.5	0.2	0xB1	0x02	0x0C
31.354	2.012	2	3	2.889	3	3	2.5	0.2	0xA1	0x02	0x0C
32.2708	2.042	2	3	2.889	3	3	2.5	0.2	0x81	0x02	0x0C
32.3624	2.045	2	3	2.889	3	3	2.5	0.2	0x91	0x02	0x0C
33.371	2.078	2	3	2.889	3	3	2.5	0.2	0x81	0x02	0x0C
34	2.099	2	3	2.889	3	3	2.5	0.2	0x81	0x02	0x0C
34.3794	2.099	2	3	2.889	3	3	2.5	0.2	0x91	0x02	0x0C
35	2.099	2	3	2.889	3	3	2.5	0.2	0x91	0x02	0x08

Example test log graphs




PA D0
PA D1
PA D2
PA D3
PA D4
PA D5
PA D6
PA D7
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25
30
35
40