Model-Pipe-Hardware: Method for Test Driven Agile Development in Embedded Software

ALBIN CASSIRER, ERIK HANE
Sammanfattning

I denna avhandling presenteras utveckling och utvärdering av en ny utvecklingsmetod för mjukvaruutveckling i inbyggda system. Långsam utvecklingshastighet är ett stort hinder för applicerandet av Test Driven Utveckling (eng. Test-Driven-Development, TDD) inom inbyggda system. Mer specifikt, uppstår flaskhalsar i TDD cykeln på grund av koduppladdningar och dataöverföringar mellan utvecklingsmiljö (host) och plattformen för det inbyggda systemet (target). Vidare användningen av ”mock”-objekt (abstraherar bort hårdvaruberoenden för att möjliggöra tester i hostmiljö) kostar mer därför att implementering och design av ”mock”-objekten förlänger utvecklingstiden.

Den föreslagna modellen, Model-Pipe-Hardware (MPH), adresserar detta problem genom att introducera strikta design regler vilket möjliggör tester i hostmiljö utan användning av mocks. MPH bygger på en lagerprincip, en så kallad ”trigger-event-loop” och en tillhörande hårdvaruarkitektur. Lagerprincipen möjliggör isolering mellan hårdvaru-beroende/oberoende kod medan trigger-event-loopen fungerar som en proxy mellan lagren.

Abstract
In this thesis, we present development and evaluation of a new test driven design method for embedded systems software development. The problem of development speed is one of major obstacles for transferring Test Driven Development (TDD) methodologies into the domain of embedded software development. More specifically, the TDD cycle is disrupted by time delays due to code uploads and transfer of data between the development “host” system and the “target” embedded platform. Furthermore, the use of “mock objects” (that abstract away hardware dependencies and enable host system testing techniques) is problematic since it creates overhead in terms of development time.

The proposed model, Model-Pipe-Hardware (MPH), addresses this problem by introducing a strict set of design rules that enable testing on the “host” without the need of the “mock objects”. MPH is based on a layer principle, “trigger-event-loop” and supporting “target” architecture. The layer principle provides isolation between hardware dependent/independent code. The trigger-event-loop is simply a proxy between the layers. Finally, the developed testing fixture enables testing of hardware dependent functions and is independent of the target architecture.

The MPH model is presented and qualitatively evaluated through an interview study and an industry seminar at the consulting company Sigma Technology in Stockholm. Furthermore, we implement tools required for MPH and apply the model in a small scale industry development project. We construct a system capable of monitoring and visualisation of status in software development projects. The combined results (from interviews and implementation) suggest that the MPH method has a great potential to decrease development time overheads for TDD in embedded software development. We also identify and present obstacles to adaptation of MPH. In particular, MPH could be problematic to implement in software development involving real-time dependencies, legacy code and a high degree of system complexity. We present mitigations for each one of these issues and suggest directions for further investigations of the obstacles as part of future work.
1 Introduction

This chapter gives a brief introduction to the context of this thesis, problem formulation, the purpose and the research questions. Assumptions, limitations and a summary of contributions of the research are presented as well.

Traditionally, software development was managed similar to more mature engineering disciplines such as mechanical and electrical engineering. Specifications were gathered from the customer, converted to design and then constructed into a finished software product. With time, engineers realized that software development is a far more creative process and much less predictable than the fields from which the development methods had been inherited[1]. The constantly changing specifications coupled with software development required more agile methods. Agile methods attempts to be light-but-sufficient and view process as secondary to the primary goal of implementation. In the 1980s, the agile movement gained significant traction, changing the industry completely. eXtreme Programming (XP), Scrum, Lean Software Development, etc. all share the values and beliefs documented in the “agile manifesto”[2]. The arguably most successful methodology from XP, Test-First Programming (TFP) or Test-Driven Development (TDD)[3] has quickly gained wide spread popularity. As the name implies, TDD developers write tests before production software. Newly written code is verified through the tests and the developer does not proceed until the software passes all tests, [3]. The developer strives to keep this cycle so short that each attempted problem is trivial enough to be solved at the first attempt and hence avoid debugging. The iteration is refereed to as a micro cycle and is repeated throughout the entire development process. The frequent testing makes TDD highly dependent on so called automated unit test, which is the practice automating the testing of the smallest pieces of testable code, a unit. Furthermore, closely coupled to TDD is the concept of Continuous Integration (CI), which requires developers to integrate committed software into its final environment and running all tests as often as possible, usually many times a day[4].

Embedded software to a large degree faces the same challenges as hardware independent software (i.e. computer programs)[4] but have not widely adopted agile methods. Cordemans[5] identify development speed, memory footprint, cross compilation issues and hardware dependencies as the primary reasons to why TDD is avoided in embedded software development.

The most common method for developing testable embedded software, independent of the target architecture, is to abstract away hardware dependencies. This requires the developer to write so called mocks to simulate hardware interactions[4]. Given the high dependency between hardware and software in embedded systems, implementing mocks becomes a significant part of the total development cost. Even if hardware is available, in order to completely test a component, manual interactions are often deemed necessary[4]. The manual interactions could include tasks as verifying that LEDs do indeed light up when expected. The manual tasks could be partially automated by automating the procedure which the human inspector follows[4]. Platforms capable of testing real interaction with the physical hardware can be used to automate tests of the software behavior in the physical world. These platforms are however, usually costly and designed for a single type of hardware.

One well-known approach for introducing TDD in embedded development, is the Model-Conductor-Hardware (MCH) design pattern method presented by Karlesky et al [6]. The aim of MCH is to isolate functional logic from hardware for testing purposes, and by that enable for TDD. This method is reviewed in section 2.11. In this thesis we propose a novel method, Model-Pipe-Hardware (MPH), for introduction of TDD, which is superior to the MCH. The MCH method serves as a frame of reference for the developed MPH and the
conducted interview study.

1.1 Problem Formulation
Hardware independent software development has benefited greatly by TDD and other agile methods. Automated unit tests are an essential part of TDD and requires hardware specific test setups or extensive mocking for implementation in embedded software. Furthermore, uploading code and test to the target introduces delays into the TDD micro cycle. The large overhead reduces the profitability of TDD for embedded software, blocking the adoption of agile methods[5].

1.2 Purpose
The purpose of this thesis is to reduce overhead for TDD in embedded software by eliminating the need for mocks in automated unit tests. More specifically, the thesis proposes and evaluates a set of design rules and a programmable testing platform which enables automated unit tests without the need of mocks.

1.3 Research Questions
To achieve the above purpose a set of research questions are stated and answered consequently. Firstly, the nature of how the implementation of mocks is used in embedded software development needs to be investigated. In particular, how mock implementation affects the development process in terms of overhead. Furthermore, the possibility of alternative methods must be investigated in terms of implementability. In this context, the following research questions are stated:

RQ 1: Would reduced mocking make TDD more attractive for embedded software development?
RQ 1.1: Is mocking needed for automated unit tests in embedded software?
RQ 1.2: Does mocking constitute a significant portion of the total development?
RQ 2: Can automated unit tests for embedded software be constructed without using mocks?
RQ 3: Can automated unit tests for hardware dependent functions be constructed without target architecture specific setups?

1.4 Assumptions
Based on the success of agile methods in hardware independent software development, the following is assumed

A 1: The positive effects of TDD in hardware independent software development translates to embedded software development.

1.5 Limitations
Due to lack of previous research on similar methods, the approach of the thesis is chosen to be exploratory. The intention is not to deliver the final verdict with respect to the proposed method but to evaluate its usefulness and identify where further research is required.

A substantial part of the conclusions are drawn from the data gathered through an interview study with industry practitioners from Sigma Technology, Stockholm. The limited
population and sample size poses a risk for bias and limits the generalizability.

Implementation of the proposed method only serves as a "proof of concept". For the proposed method to serve as a viable alternative to current methods, tools assisting the developers have to be constructed. Design of these tools are discussed but not fully implemented.

1.6 Contributions

- Literature study indicates that the primary blockers for using TDD in embedded development are: Development speed, Memory footprint, Cross compilation issues and Hardware dependencies.

- Proposed development method, Model-Pipe-Hardware (MPH), which uses strict design patterns, and a programmable target independent testing platform. This enables
  - Automated unit testing of hardware dependent code which reduces the need of manual testing
  - Isolated unit tests without mocks, hence, reducing overhead for TDD in embedded software development

- Evaluations of MPH in a interview study with developers from industry indicate that the proposed method could reduce overhead in TDD as compared to MCH, and can be implemented in industry projects under certain constraints. In particular, the usage of mocks in MCH drastically derates its performance compared to the proposed MPH according to the respondents.

- Verification of MPH on a small scale industry project shows that MPH is possible to implement, no mocks are required.

1.7 Applied Method

The overall research method of this thesis is composed of initial development and evaluation of the proposed agile development method for embedded software. This includes the use of several scientific methods. First, a literature study which is presented in Chapter 2 is used to create a context, present related work and identify gaps in current knowledge in the field. This is then used as the basis for the conceptualization and development of MPH which is presented in Chapter 3. Chapters 5 and 6 present evaluations of MPH with qualitative methods (interview and seminar) and experimental validation in form of project implementation. Finally, thesis findings are summarized and discussed in Chapter 7, along with future work. For a detailed description of the applied scientific methods see Chapter 4.
2 Literature Review

The literature review presents an overview of Extreme Programming (XP) followed by a discussion on traditional software development lifecycle models and software testing. Finally Test Driven Development (TDD) is presented together with the constraints of TDD in embedded software development, followed by presentation of the MCH method.

2.1 Extreme Programming (XP)

“The outcome, in the real world, of software system operation is inherently uncertain with the precise area of uncertainty also not knowable.”[7]

As stated by the uncertainty principle[7], software is uncertain in nature. Increased competition and playing an increasing importance in almost every industry today, the ability to produce high quality software is highly desired. Processes such as Personal Software Process (PSP)[8] attempts to serve as guidelines on how to create quality software. These processes have a strong focus on planning before implementation but do not excel in managing changes in requirements and design[9]. The dissatisfaction led to the development of agile development models[1, 9]. Agile methods attempt to be light-but-sufficient and view processes as secondary to the primary goal of implementation.

Extreme Programming (XP) was formulated by Kent Beck, Ward Cunningham and Ron Jeffries in the late 1980s[9]. XP received much attention due to the success of Chrysler Comprehensive Compensation (C3) system which was developed using XP[9]. The success of the new methods is largely attributed to its ability to manage changes in software during development.

Testing is the very core in XP as it utilizes test driven development (TDD)[3]. TDD has proven to be very effective as an agile practice. Proponents will argue that the success of XP is largely due to TDD[3]. The simplicity advocated by XP, “Do the simplest thing that could possibly work”, is achieved through TDD’s approach of implementing only the minimal required software. Test cases are written before the code for every function which forces developers to design with testability in mind. The tests and short development cycle give developers instant feedback on the implementation which allows for software changes without affecting existing functionality[4].

2.2 Traditional Software Development Lifecycle Models

The waterfall model, or linear sequential model, is a systematic and sequential approach to software development originally proposed by Winston Royce in 1970[10]. The original model have feedback loops and seven phases:

- System Requirements
- Software Requirements
- Analysis (of the system and software requirements)
- Program Design (requirements are translated to a representation of software)
- Code (software representation is implemented in a machine readable format)
- Test (verification of the correctness of implementation)
- Operation (usage of finished software product)
The model places heavy focus on the analysis phase and promotes extensive documentation with at least six different types of documents[11]. Although waterfall model is one of the oldest model still in use today, it has issues. Requirements must be determined at an early stage[10, 12], which introduces long incubation periods due to the sequential nature of the model.

Figure 1: Diagram from Royce’s original publication "Managing The Development of Large Software Systems"[10].

Numerous modifications to the original waterfall model has been suggested since its introduction. The B-model adds an evolutionary enhancement process to the operational phase[11]. The incremental model (also known as the iterative waterfall model) can be seen as a three dimensional representation of the original model where the z-axis contains a series of waterfall models allowing for feedback from the earlier integration and for more stakeholder inclusion[11]. The V-model, originally developed by NASA[11], is a variation of the waterfall model with emphasis on tight integration of testing in all phases of software development[13]. The model is visualized as a "V" with software development activities flowing downwards on the left hand side and corresponding testing activities on the right hand side going upwards. In contrast to the waterfall model, the V-model promotes tests to be designed in parallel (instead of after)[13]. Each phase of development has a corresponding testing phase to ensure that requirements and the implementation are verifiable in a SMART (Specific, Measurable, Achievable, Realistic, Time-framed)[14] manner[11]. Typically business requirements are tested with acceptance tests, design with integration tests, and code with unit tests.

2.3 Testing Methodologies

Homés[15] argues that testing is not a substitute for a weak development process. Instead tests should be considered as a method for asserting quality and proper functionality of software. In his classical Software engineering economics[16], Boehm showed that the cost of defects in software raises exponentially with the advancement of the software development cycle. Incremental testing can help developers detect problems at earlier stages and hence lower the cost of development.

Gelperin and Hetzel[17] summarize testing with four models: Demonstration, Destruction, Evaluation and Prevention. First stated in 1988, the prevention model attempts to prevent
defects. The unpredictable nature of software\cite{7} makes it impossible to predict and prevent all defects\cite{18}. The prevention model, therefore, has the secondary goal of detecting faults and defects\cite{17}. The prevention model and ANSI/IEEE standards for unit testing\cite{19} led to the formulation of the Systematic Testing and Evaluation Process (STEP)\cite{17}. STEP states that testing should be carried out in parallel with development which leads to significant increases in quality\cite{17}. Parallel testing was however not introduced with STEP as records of it dates back to 1974\cite{13}. The activity of designing a test, is in it self known to be one of the most effective defect preventers\cite{17}.

2.4 Dynamic and Static Testing

Testing can be divided into two sections, dynamic testing and static testing\cite{20, 21}. Dynamic tests execute the code using a set of test vectors and evaluate the results using specifications and requirements\cite{20}. A weakness of dynamic testing is that it can only catch bugs (common term used interchangeably with faults and defects) if it affects a test case. Static tests on the other hand analyse the code itself, and not its behavior at run time. The static tests are typically carried out using combinations of code inspections and code walkthroughs\cite{21}. Static tests are generally less expensive than dynamic tests and are therefore usually carried out before dynamic testing\cite{21}.

The static tests rely heavily on human domain expertise, and hence, are often done manually. Automatic tests can be a very useful as well and reduce manual effort. Wang Zhong-Mi et. al\cite{22} show that automatic static tests can discover between 30\% and 70 \% of defects caused by logical design and coding errors. Typical faults include dereferencing null pointers, memory management errors, inconsistent usage of global variables, possibly infinite loops and buffer overflow vulnerabilities.

2.5 Unit Testing

Testing is typically divided into four distinct types, unit/component tests, integration tests, system tests and acceptance tests\cite{23}. Each type has a different approach for detecting defects. Naik et. al\cite{21} states that there is no consensus on the exact definition of a unit. One definition of a unit is the smallest testable piece of code that can be compiled, linked, loaded and put in control of a driver or test harness\cite{13}. Unit tests verify that a unit behaves according to expectations\cite{21} derived from system and design requirements. Unit tests are most effective when performed by the developer of the unit\cite{21} because the developer has the best understanding of the units content\cite{23}. Unit tests are used to expose faults rather than to prove their absence\cite{21}. If independent (running a unit does not affect other units) and run frequently unit tests can be very effective.

Ideally, tests should verify every line of code, every branch, all possible paths of branches, correctness of every operation of normal and limiting values, abnormal terminations of loops and recursion, evaluate timing, synchronization and hardware dependencies\cite{23}. Testing all possible permutations and combinations of inputs is rarely practically possible\cite{17, 23}. The set of test inputs should, therefore, be selected carefully\cite{17}. Many developers rarely practice testing and are likely to avoid it all together when pressured by deadlines\cite{13}. In agile methods tests are either developed in parallel or before the code\cite{15} which requires a change in practice for many developers.
2.6 Other Types of Testing

In addition to unit tests, other types of testing include integration, system and acceptance testing[23]. Similar to unit testing, integration testing is performed by developers. Fox[24] defines integration testing as the process of verifying that aggregation of units behaves according to specification when integrated with each other. Fox argues that the process should be performed incrementally in either top-down or bottom-up fashion. The importance of regression tests is stressed as it can detect changes in behavior due to integration. Furthermore, it is important to execute the tests again if a defect is found and fixed in order to verify that the software changes did not cause defects in other system parts.

System tests is a series of tests intended to evaluate functionality of the complete product. System test may include other component than the software (hardware, people etc)[20]. The tests should verify that system components have been correctly integrated to create the functionality specified by the requirement[23].

Acceptance testing is performed by the customer, typically at their own site. The objective for the customer is to evaluate and understand the system in order to make a final decision whether the product should be accepted or rejected. The customer wants to verify that the behavior of the system conforms to specifications and requirement[20, 23].

2.7 Testability

Testability is the measurement of how easy it is to test a software[23]. Software with high testability is more likely to reveal defects during testing and can therefore be delivered with better quality than software with lower testability. A common measurement for testability is the number of test case required to test a piece of software (few required tests indicate high testability)[25]. The IEEE Standard Glossary of Software Engineering defines testability as:

“(1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met[26].
(2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met[26].”[26]

The definition can be paraphrased as the probability that defect software will fail during testing. Pressman[23] argue that testability can be divided into the following characteristics (paraphrased from the original work):

- **Operability** Designing the software with tests in mind will result in fewer bugs blocking the test execution which allows for a smoother process.
- **Observability** Making the internals of the software under test visible to the tests. Internals include system states, variable contents and source code.
- **Controllability** Better control over software allows for more automation and better tests. All possible outputs must be generatable from controllable inputs, all code executable through combinations of inputs and all software and hardware states must be accessible and controllable by the test engineer.
- **Decomposability** The software is built from independent modules which can be decomposed and tested in isolation from each other.
**Simplicity** Less complex software is easier and faster to test. The simplicity can be divided into functional simplicity (no more features than necessary), structural simplicity (modularized design to limit the propagation of faults) and code simplicity (coding standards are followed so code is easily to read and understand).

**Stability** Changes to software should be minimized and should not invalidate existing tests.

**Understandability** Knowledge about the internals of the software under test is vital for designing good tests.

Designing with these characteristics in mind, helps creating more testable software which results in higher quality. A general rule is to design software which “fails big”. Silent errors are harder to spot and, therefore, carry a greater risk of remaining in the final product. Test driven development is especially good at producing highly testable software because the process forces the developer to think about testing and testability at all stages of development[4].

### 2.8 Limitations of Testing

Testing is a very effective tool for detecting defects but complete testing is not possible[23]. Even with a finite number of inputs, testing all combinations of inputs and paths through the code is both theoretically and practically impossible[23, 15]. Testing, hence, can never assert the absence of faults and defects, only expose their presence[15]. The pesticide paradox states that tests lack efficiency when they are used over time. The loss of efficiency is due to the execution-paths being executed before and, hence, tests are unlikely to catch new regression faults (bugs or regression in the system caused by software changes or the adding of new code)[15]. Furthermore, the complexity barrier states:

“**software complexity (and therefore that of defects) grows to the limits of our ability to manage that complexity**”[13]

Since complete testing is not possible, the testing process should stop when the value of new tests reaches inconsequential levels[13]. These inconsequential level depends on various factors including business, safety and technical requirements.

### 2.9 Test Driven Development

*Test driven development* (TDD) is an agile methodology which aims to produce high quality software by managing changes in requirements and design effectively[13]. TDD has been sporadically used by developer for decades but was popularized with the success of XP. Proponents of TDD sometimes claim that it is the most successful component produced by the agile movement[3]. Although the name suggests otherwise, TDD is not a testing method[4]. It is a practice for driving the development process where it informs and guides analysis, design and programming decisions[27, 4].

TDD dictates that software development should be done incrementally by implementing a small number of automated unit tests prior to writing production software[28, 29, 1, 3, 5, 4]. The tests are produced by developers[4] as opposed to more conventional developing methods where testing is sometimes seen as a separate process, often carried out after software implementation[11].

The Agile Alliance defines TDD as:
"...the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code."[28]

The underlying principle is that not a single line of new code should be written if there are no failing tests[29]. The principle includes fixing “bugs” discovered in later stages of the development cycle. To fix a “bug” is nothing more than altering behavior (typically removing unwanted behavior) of the code. The process of fixing a “bug” is, therefore, no different than the process of introducing of new behavior or creating a new feature. The developer implements a test, which fails due to the existence of the ”bug”, thus exposing the bug. The bug is then fixed and the developers receive instant feedback as all the tests are passing.

In TDD, tasks are clearly separated and ordered, which allows the developer to focus on a single goal in each step. In Figure 2, the TDD development process is illustrated as a flowchart. In the first step the developer specify new functionality by writing a test, which exposes the absence of the feature[4]. Motivated by the failing test, the minimal code required to pass the test (without breaking existing tests) is implemented. When all tests pass, the functional code is refactored. Refactoring is the process of altering the internal structure of code and tests without changing their external behavior[4]. Refactoring is done to improve the overall structure, remove duplication and increase understandability of both code and tests[27]. It is important to note that tests and code are never allowed to be refactored simultaneously. Before focus is shifted from code to tests (or vice versa) the entire test suite must be executed without failures[5]. When quality of both tests and code is considered satisfying, the cycle starts over.
The cycle of TDD is called a *micro cycle* and is repeated throughout the entire development process[4]. The microcycle creates an incremental development process where small tests are added and run frequently, accordingly driving development, one new feature at a time.

TDD requires the use of automated unit tests (See section 2.5). Incrementally specifying functionality as automated unit tests eventually create a test suite which is used to detect regression faults. The developer can therefore assert that new code, functionality or refactoring, does not affect existing behavior[5]. Furthermore, maintaining a clean test suite should
receive high priority[4]. Clean and understandable tests become valuable not only for fault
detection, but, for documentation as well. Martin[30], argues that good a unit test follow
the rules of F.I.R.S.T. which were defined by Tim Ottinger and Jeff Langr in the Agile in a flash project[31]. These can be paraphrased as:

**Fast** Tests should be run fast. If the tests become slow the developer will be less inclined
to run them frequently. If tests are not run frequently, problems will be discovered later,
and hence, be more difficult and expensive to fix. Refactoring becomes a burden and
eventually the code will begin to deteriorate.

**Independent** Tests should not depend on each other. One test should never set up
the conditions for the next test. Tests must be executable independently in any order.
When tests depend on each other, one failing test can cause a cascade of downstream
failures, making diagnosis difficult and hiding downstream defects.

**Repeatable** Tests should be repeatable in any environment. The tests should be able to
run in the production environment, in the QA environment, and on your laptop while
riding home on the train without a network connection. If tests are not repeatable in
any environment, then there is always an excuse to why they are failing. Furthermore,
the developers will be unable to run the tests when the specific environment is not
available.

**Self-validating** Tests should have a boolean output, pass or fail. The developer should
not be required to read through a log file to determine whether the tests passed or failed.
If the tests are not self-validating, then failure becomes subjective and running the tests
could require long manual evaluations.

**Timely** The tests should to be written in a timely fashion. Unit tests should be written
just before the production code that makes them pass. If written after the production
code, then the production code might be hard to test as testability is easily ignored if it
increases complexity.

### 2.9.1 Test Doubles

Truly isolated units are rare and in many cases dependencies of the unit hinder automated
unit tests. The component might not be available at the time of development, it might
not return values required for a specific execution path or it might be difficult to execute
in an automated environment[32]. The issue exists for computer software development but
is especially evident for embedded software development where hardware dependencies are
very common and important.

In a test where the real component is unavailable or impractical for tests, a test double can
be used instead. In simple terms, a test double is a piece of software (or hardware) which is
inseparable to the real component from the perspective of the code under test (CUT). The
test double does not have to be an exact copy of the real component, it merely needs to
provide the same API as the real component (or, at least, the parts used by the CUT)[32].
Gerard Meszaros[32] splits test doubles into: test stub, test spy, mock object and fake object. Test stubs are used to verify indirect inputs; test spies and the mock object verify indirect outputs; and the fake object provides an alternative implementation.

Test stubs allow tests to control indirect inputs of the CUT. Controlling these indirect inputs of the CUT allows testing of execution paths which might otherwise be very difficult to reach (e.g. rare errors, special timings etc). Stubs are sometimes referred to as temporary implementations because they substitute the final components until it becomes available for the tester[32].

The test spy is a more advanced version of the test stub. Apart from being able to provide the CUT with indirect inputs, it can also capture indirect outputs of the CUT. The indirect outputs can then be used by the test to evaluate the CUT[32].

Just as test spies are an advanced version of the test stubs, mock objects are a more advanced version of the test spies. In addition to the functionality of a test spy, a mock object is able to perform verification of the indirect outputs produced by the CUT. The verification includes combinations of assertions of arguments ranged in calls, sequences of calls and timing of calls. Furthermore, mock objects are able to alter injected inputs depending on the registered outputs[33].

Finally, fake object is an alternative implementation for the component it imitates. In contrast to other types of test doubles, fake objects are neither configured nor hard coded to inject specific inputs to each test. Instead they hold an alternative, more lightweight, implementation of the real component. The overhead of developing a fake object is much higher than configuring mock objects and should, therefore, only be used when a large number of tests can utilize the fake object[32].

2.9.2 Unit Test Frameworks

An essential tool in test driven development is unit testing framework which helps automate unit tests execution[4]. Unit testing frameworks typically consist of two main components; a test library and a test runner. The library should provide a selection of assertions for types, exceptions, timing and memory leaks. The test runner is used to set up the test environment, call the unit tests, clear the environment (tear down) and finally to report the results to the developer. A key feature of a test runner is that failing tests do not halt the execution, instead it generates an informative error message for the report and continues the execution of remaining tests. Upon completion, a report containing binary results (pass
or fail) of each test is generated.[5]

2.9.3 Advantages and Difficulties for Test Driven Development

Many proponents of TDD claim that the method reduces (if not eliminate) the need for upfront design. Software is unpredictable in nature[7] and, hence, close to impossible to design correctly upfront. However, TDD is highly flexible which allows adaption to any process methodology, even if it includes upfront design. All processes have a gap between decision and feedback. Reducing this gap is often attributed as the biggest source of success of TDD.

Test-driven development is a fundamental practice of agile development and has been subject of numerous studies and evaluations[34, 35, 13]. Cordemans et al.[5] summarizes the strongest advantages with TDD as frequent testing, steady development, focus on current issue, focus on external behavior and testing not debugging. Cordemans et al.’s description of these aspects can be paraphrased as:

**Frequent Testing** An essential part of the TDD process is the frequent execution of the test suite, which provides the developer with fast feedback. If refactoring causes unwanted behaviour, or if regression faults are introduced with new features the test will fail. “Bugs” found in later stages of development are exposed with tests before being fixed which improves regression tests as the development progresses. Furthermore, designing for testability ensures that modules can be executed in isolation which increases reusability. Finally, the test execution supplies a continuous stream of feedback of the current project state, which makes progress tracking less ambiguous.

**Steady Development** TDD eliminates the unpredictable process of debugging hence produces a more steady development rate. All features are represented by one or more tests. A suite of passing tests corresponds to said features being successfully implemented. Hence development can tracked and progress can be assured.

**Focus on Current Issue** The TDD mantra ensures that focus is placed on a single issue at the time. One feature is developed at a time and the phases of specifying the feature, writing a test, implementing functional code and refactoring is clearly separated from the other phases.

**Focus on External Behaviours** TDD focuses on the interface and external behavior of software. By testing his or her own software, TDD forces the developer to think how the functionality will be presented to the external world.

**Testing not Debugging** TDD attempts to replace debugging with testing. Debugging often constitute the majority of development time, is unpredictable and usually relies on a more or less effective debugger (depending on the platform).

Cordemans et al[5] also identify the biggest weaknesses of TDD. The following list is paraphrasing of these findings:

**Overhead** TDD introduces overhead through increased work effort. TDD doubles the amount of code that has to be written for each feature (compared to code without tests). Furthermore, while TDD is very effective in producing library code (i.e functions that does not directly interact with the outside world), external dependencies becomes problematic. The solution is to abstract away external dependencies using test doubles. Design and implementation of the test doubles adds additional overhead.

**Test Coverage** In TDD, unit tests are implemented by the developer, which results in corner cases (typically complex test cases, typically requiring maximal settings in
multiple dimension) often being untested. Due to the minimalistic nature of TDD, developers often ignore these cases as it sometimes cause tests to overlap. Corner cases will be tested if they are a source of faults for the isolated unit. The problem arises when corner cases causes regression faults. The false sense of security makes the developer less likely to discover these “bug” until later stages of development. Furthermore, since tests are designed by the developer, only problems known to the programmer will be exposed. A large suite of unit tests does, therefore, not remove the need for integration and system tests.

2.10 Test Driven Development for Embedded Software

Software plays an increasingly important role in almost every industry today. An increasing part of innovations, especially in technical fields, are based on electronics and software. Take for example the automotive industry where software and electronics constitute a major source for advancement in everything from safety to passenger comfort and fuel consumption. The number of errors in the software has a direct impact on the quality of the product and should therefore be managed with extreme care. Unfortunately it has been proven virtually impossible to develop complex software ”first-time-right” [18]. Boehm’s law states that the cost for fixing software bugs increases exponentially as the project progress [16]. Frequent measures should therefore be performed so errors are detected as early as possible. Besides reviews and inspections, testing has proven to be extremely valuable.

Software “bugs” can have disastrous effects stretching from the company bottom line to brand perception and human safety. Embedded software is a specialty within the broader field of computer programming. While a high-level computer, generally, is detached from the physical world, embedded software is very tightly coupled with the real world. High-level computer program “bugs” can in most cases be fixed by distributing a software patch. An embedded software “bug” residing in, for example the fuel injection of an automobile, can lead to mass callback and extreme costs, or even worse, losses of human life. Agile methods have proven to reduce the number of “bugs” with one magnitude [36] and are becoming a common practise in high-level computer software development. Embedded software has not embraced the new paradigm to the same extent. Instead the industry still mostly relies on more traditional methods from more mature fields of engineering such as electrical and mechanical [4].

2.10.1 Embedded Constraints

Embedded software development is generally not performed in the same environment as the software will run. The system where the developer produces the software is referred to as the host (i.e. a PC) and the system where the software will run in production is referred to as the target (i.e. an MCU).

Testing embedded software is, by no means, an easy task as embedded software are often highly dependent on its environment. Real time constraints and hardware related performance are just a few of many constraints in embedded software [37, 18].

Cordemans et al [5] identify four main reason to why TDD is not used in embedded software, which is partly based on the work of Grenning [4]. These four are paraphrased below:

Development Speed TDD is performed in short cycles where the test suite is compiled and run frequently. When host and target are different systems (typically the case), compiling and uploading the code and tests to target introduces large delays into the micro cycle. Furthermore, the execution time on a slow target and transmission of test
data from the target to host adds further delays[4]. Large delays disturbs the rhythm of the TDD micro cycle which causes the developer to reduce the frequency of test executions by attempting to add more features per cycle. The complexity increases and thus bugs become common. Debugging introduces more delays, eventually completely destroying the TDD rhythm[5].

**Memory Footprint** The size of target memory footprint poses constraints on the embedded software development. Fitting production code alone on target can be challenging. Fitting production code, test suite and test harness is often impossible[5].

**Cross-compilation Issues** Testing on host instead of target mitigates the constraints of development speed and memory footprint. Unfortunately the method introduces something called cross-compilations issues[5]. The problem arise because target and host differ in processor architecture and/or build tool chain which can cause incompatibility problems[4].

**Hardware Dependencies** Embedded systems software typically have hardware dependencies which makes automation of tests problematic. To ensure deterministic test execution the dependencies must be controllable. Furthermore, hardware is not always available to the developer at the time of development (e.g. expensive, not yet developed etc)[5, 4].

### 2.10.2 Test-on-Target

Test-on-target is based on the *Embedded Test Driven Development* (*ETDD*) method presented by Smith et al.[38] and requires both code and tests to be loaded onto the program memory of the target. The test-on-target strategy enables test of behaviors which rarely can be accurately tested on a host system, such as memory management operations, real-time execution etc. Besides test cases, a test framework with on-target test functionality such as time-asserts, memory leak detection, runners and reporting (to host) must be able to fit alongside the code. According to Cordemans et al[5], the method should be used in combination with test-on-host to assert behavior on target architecture. However, if the complexity of mock development for certain hardware aspects is too high, all testing should be performed on target. Furthermore, in development of a product with legacy code, test-on-target might be the only approach capable of asserting that regression faults are not introduced.

The frequent uploading to target makes test-on-target too time consuming for TDD to be used for embedded software. Instead, test-on-target should extend a TDD cycle on host to detect cross-platform issues. Grenning[4] and Cordemans[5] define an *embedded TDD cycle* process with four different levels of testing. First, code is developed on host with a pure on-host TDD microcycle. When all test pass on host, the target compiler is invoked to detect any compile time errors. Target compiler error is resolved and automated unit test are executed on target. The final level includes manual system test or acceptance tests which is performed every few days. Furthermore, Grenning[4] suggest that code should be run on an evaluation board before it is ported to the target platform to avoid cross compilation issues.

### 2.10.3 Test-on-Host

The test-on-host strategy is based on the *Dual Targeting* approach suggested by Grenning[4]. According to Cordemans et al.[5], test-on-host allows the fastest possible feedback as both code and tests reside on host. In addition to eliminating delays from uploads, a test-on-host approach provides the tester with virtually limitless memory and processing power. Furthermore, the development and testing are independent from the availability of the target.
hardware. This enforces a modular design, which increases portability and reusability of the code and tests. Since all development and testing are done on a host machine, hardware calls and other behaviour of target hardware must be mocked. However, at some point the code has to be migrated to the target for verification.

The test-on-host strategy is implemented using (at least) two build configurations, one (for each) target build and one for the host system. In the host build hardware components are mocked to enable testing on host. However, cross-platform issues are common and are impossible to detect without a reference build or deployment model. Some issues can however be mitigated by running a cross compiler or using a development board before the actual target is available [5].

2.11 Model-Conductor-Hardware

An alternative for TDD in embedded systems is the MCH-pattern by Karlesky et al [6] [36] [5]. According to Karlesky et al. using design patterns is a well documented method for solving reoccurring problems within software development and that such an approach could be applied in the embedded field as well.

The aim of the MCH design pattern is to isolate functional logic from hardware for testing purposes. MCH is implemented by designing/dividing the software in three different functional members. The idea is to replace the hardware dependent code by mocks, thus, decoupling functional logic from the hardware. This enables isolated automated unit testing of hardware dependent and pure logic code. The MCH is based on three functional members: Model, Conductor and Hardware. They together constitute a triad (see Figure 4). We will describe them one by one, following the original work by Karelesky et al. [36], [6].

![MCH Triad Diagram](image_url)

Figure 4: MCH triad, with the functional members, Model, Conductor and Hardware. Adapted from [36]
2.11.1 Model
This layer includes pure logic functions e.g. control logic, states, and equations. It also contains a communication interface towards other parts of the the system outside the triad. The model is only connected to the Conductor within the triad, and it has no link to the Hardware member[6].

2.11.2 Conductor
The conductor member contains the main control logic of the triad. The conductor functions is a divider between the Model and the Hardware and conducts data between the two. It works by acting on triggers from the Model or Hardware which results in the following [36]:

- Setting state within the Model
- Querying the state contained by the Model
- Querying the state contained in the Hardware
- Moving data between Model and Hardware
- Initiating hardware functions

2.11.3 Hardware
The third member of the triad, the Hardware, represents a thin layer around the hardware (sometimes called hardware abstraction layer, HAL). This layers encapsulates the ports, registers and Interrupt Service Routines. Similar to the Model, the Hardware member is only connected to the Conductor and has no direct reference to the Model.

2.11.4 Testing with Model-Conductor-Hardware
The implementation of MCH is based on extensive use of mock objects, using test assertions against the information captured in the mocks. Each member of the triad is unit tested in isolation using mock representation of other system members. Since mocks are constructed for each member of the MCH triad, isolated testing possibilities are enabled. For example, a Hardware member would be tested using a mock representation of its corresponding Conductor (to which the Hardware has a “dependency”). In Figure 4, the triad members are illustrated with their internal dependencies and corresponding mock representations. Using the MCH approach, Test-on-Host and Test-on-Target strategies and combination of the two are enabled.

2.11.5 Comparing Model-Conductor-Hardware and Model-Pipe-Hardware
In the context of the thesis MCH is used as a reference method for the interview study and is, therefore, only described briefly¹. However, similarities and differences between MCH and the developed MPH method were essential for conducting the interview study. The main differences between the two methods are listed below.

- The MCH uses an approach where all functional members are mocked while MPH avoids mocking by using a reusable trigger function.
- In MPH the isolating middle layer is never explicitly unit tested, which present advantages for development, e.g., less tests needed. In MCH, this layer (the Conductor) is unit tested by using mock representations of the Hardware and Model layers.

¹For further reading and implementation examples of MCH, the authors recommend the related work of Karlesky et al [6],[36].
The MCH design process uses a top “down” approach where overall system behaviour is designed and tested before developing detailed functionality. In MPH, development of needed functionality is made in the beginning of the process and the overall system design is made at a later stage. See Appendix A for a comparison of design/testing processes.

In the text, Chapter 3 detailed description of the developed MPH method is presented. Later, Chapters 4.2 and 5 will account for the conducted interview study where MCH will be used as reference/baseline to gather a rich body of qualitative data on the new MPH method.
3  Model Pipe Hardware (MPH)

The theoretical framework chapter presents the main contribution of the thesis, the proposed method Model-Pipe-Hardware (or MPH). A motivation based on the literature review is presented followed by purpose, rules and test methods for each of the three layers (Model, Pipe and Hardware).

3.1  Motivation

The cost of “bugs” increases exponentially with the progress of the software development life cycle[16]. The embedded software industry still relies heavily on traditional development processes due to the usage waterfall model where tests are carried out after implementation[11]. Agile processes such as Test-driven development (TDD) integrates unit test into the development process, which increases the chance of detecting faults at an earlier stage of development. In TDD, unit tests are executed very frequently making the need for automation and speed essential[5].

Due to hardware limitations and dependencies, TDD in embedded software is coupled with multiple complications. Cordemans et al[5] identify upload time and test execution on target as major sources of delays when compared to host-only development. The delays slow down the TDD microcycle, causing developers to attempt larger development steps, which in turn, increases complexity and, hence, weakens the benefits of TDD. Furthermore, limited memory resources can make it impossible to run both production and test code on target simultaneously. Finally, automating unit tests for units directly interacting with hardware requires target hardware-specific testing platforms. The cost of these platform causes many developers to use manual testing procedures instead.

Issues related to development speed motivates the use of processes that minimize development and testing performed on the target hardware. Testing-on-host avoids delays from uploading code to target and reduces execution time for tests as host generally has more processing power. Embedded software does, however, have large dependencies on target hardware making mocks necessary for on host development and testing[4]. Project-specific mocks have no value by themselves, making implementation time for mocks a pure development overhead. Furthermore, in addition to on host testing, manual test procedures will be needed to verify hardware interactions (i.e lighting LEDs, measuring analog signals, etc )[4].

The proposed method attempts to make TDD a viable methodology for embedded software development by maximizing on host development and eliminating the need of mocks and manual testing.

3.2  Layer Structure

To minimize development on target hardware, isolation of hardware dependent functions is required. Abstracting hardware dependencies decouples software and hardware making test on host possible for parts of the code. A typical layer structure has at least three layers, a hardware abstraction layer (HAL), one or several middle layers and a layer holding the business logic. Layer structures do result in clearer delimitation between hardware dependent and hardware independent code allowing testing on host for hardware independent sections. However, data still has to flow between layers, which creates dependencies between adjacent layers. Adjacent layers, therefore, have to be substituted with mocks in order to fully isolate the unit.
The number of dependencies in layer structures tends to increase with the complexity of the product[39]. This behavior can be divided into two general cases (function A and B may or may not belong to the same layer):

1. A function A requires data from other layers, hence A calls a function B. B executes and returns data to A.
2. A function A receives data that is needed by other layers. A calls a function B with the data as arguments. The two sub cases:
   (a) A depend on the return value of B.
   (b) A does not depend on the return value of B.

A depends on B in case 1). A test vector for A, therefore, include both function argument (for A) and return value of B. For A to be testable, B must be controllable. A mock for B ensures testability of A. However, if B is called before A and its return value passed as argument to A, then no mock is required. The test vectors, hence, consist solely of function arguments to A. Listing 1 depicts the two approaches.

```python
def function_A(*args, **kwargs):
    """Function A that depends on B""
    # Call B to retrieve the data needed
    data_from_B = function_B(...)

    # Do stuff using function arguments passed to A and
    # data_from_B
    ....

def helper_of_A(*args, **kwargs):
    """Calls B and passes data to A so that A becomes independent of B"

    # Fetch data from B
    data_from_B = function_B(...)

    # Pass the data to A so it does not have to call B
    function_A_independent_of_B(data_from_B, *args, **kwargs)

def function_A_independent_of_B(data_from_B, *args, **kwargs):
    """Function A that does NOT depends on B"

    # Do stuff using function arguments passed to A
    # including data_from_B
    ....
```

Listing 1: In case 1. A fetches data by calling B in function_A. By moving the dependency to helper_of_A, A becomes independent of B.
For A to be testable in 2), a mock for B is required. In both sub cases, the arguments passed by A to B must be observable for A to be testable. In the sub case 2b) the mock merely has to register passed arguments as the execution path of A does not depend on return values from B.

For the sub case 2a), in addition to being observable, the return value of the mock of B must be controllable for A to be testable.

Figure 5: Passing data between layers creates dependencies which creates the need for mocks.

In 1). and 2b), the dependency between A and B can be avoided. Suppose that instead of A calling B directly, A calls a proxy function with the data intended for B. The proxy function then uses the data to call B such as A would. For an individual case, a proxy function does nothing more than moving the dependency from B to the proxy. However, if the proxy function is shared throughout the entire project, the dependencies are contained at a single integration point, which is this proxy function. A single mock of the proxy function hence enables isolated testing of A without mocking B.
Figure 6: Using a proxy function forces all dependencies to a single integration point

In 2a), A does still depend on the return value of B (see Listing 2). 2a is, however, nothing more than a combination of 1). and 2b). A in case 2a) can be split into two parts. The first part, the code preceding and the call of B, and the remaining part which depends on the return value of B. We see that the first part is 2b). The remaining part of A depends on the input values of A and the return value of B. This can be moved into a separate function which hence has either no dependencies or can be described by the discussed cases. By induction, dependencies for all A (of finite length), can be reduced to either 2b) or 1). Forbidding 2a) does, therefore, not reduce possible functionality. Hence, using a proxy function and by passing required data as argument, all dependencies can be integrated into a single function.
def function_A(*args, **kwargs):
    """General structure for a function A that passes data to B and depends on its output."
    """

    # Arguments passed to A is used to produce the data which should be passed to B
    data_needed_by_B = ....

    # Call B and save the return value
    important_return_value = function_B(data_needed_by_B)

    # The return value of B will determine how A will behave
    if some_condition(important_return_value):
        # Do something
    else:
        # Do something else

    # Potentially do something more

Listing 2: In sub case 2a) A depends on important_return_value. In sub case 2b) lines \( \geq 12 \) would not exist or be independent of important_return_value.

Based on this reasoning, Model-Pipe-Hardware (MPH) method is proposed. In MPH, software products are separated into apps. Each app represents an encapsulated module of the product. Similar to most models in Object-Oriented-Programming, deciding how to modularize a project is left to the discretion of the developer.

As the name suggests, MPH uses a three layer structure to separate the content of each individual app. Hardware dependent functions are located in the hardware (H-layer), business logic in the model (M-layer), while the pipe (P-layer) separates the two. H- and M-layer functions are not allowed to call other functions. Instead, all data must be provided as arguments. If the function wishes to pass on data, the proxy function is used. The proxy function routes data to function calls in the P-layer where H- and M-layer functions can be called with the data as argument.

MPH uses a trigger-event-loop as proxy function. The trigger-event-loop has two components, a trigger function and an event-loop. In short, a trigger-event-loop is a very primitive scheduler in which the trigger function is used to schedule jobs by passing an event-identifier and data. The two are combined and put into a queue. The event-loop instead monitors the queue, pops items when available, and translates the event-identifier to function calls using the data as argument.

Figure 7 depicts the general design of an MPH-system. The trigger function is used across all modules and receives event identifiers and data objects from H- and M-layer functions. The data is then passed to the event loop that translates event identifiers to zero, one or multiple callback functions in the P-layer. Data can, hence, be sent between apps without forming dependencies. Apps highlight the presence of data by triggering an event so that apps which desire the data can access. These “producing” and “receiving” apps need not to be aware of each other.

In Section 3.3, 3.4 and 3.5 the three layers are explained in detail.
3.3 Model

The *model* or *M-layer* consists of hardware independent functions and the state of the app. Isolating and abstracting of hardware dependent code allow hardware independent parts to be developed and unit tested on host. Furthermore, separating hardware and software facilitates more modular, portable and reusable software designs[4]. Being testable on host shortens the TDD cycle as upload time is removed[5]. It also allows software development to start before the hardware has been chosen, which is often the case in embedded software development[4].
To remain self contained, the M-layer has a persistent structure. The structure should hold information about the current state of the app. The structure `private`, meaning that it is accessible by all M-layer functions of the app but hidden from all other. If the content of the structure is needed by other functions the app should present an API for retrieving the data. No direct access should be allowed.

M-layer functions are, however, not allowed to utilize the API of other apps as it creates dependencies between apps, that needs to be mocked during testing. Instead, P-layer functions use these API to fetch the data and pass it as arguments to M-layer functions. Hence, the dependency is moved from the M-layer to the P-layer, which makes isolated unit tests of the M-layer possible without mocks.

### 3.3.1 Testing the M-layer

An M-layer function uses only function arguments and the content of the app structure for its execution. The functions have three possible ways of interacting with the system: *return value to caller* (1), *update state struct* (2) and *trigger event* (3).

The M-layer functions are not allowed to call other functions and thus do not depend on other functions. Unit tests therefore only need to monitor the return value, app’s structure and triggered calls for full testability. The structure can easily be made accessible to the tests. The trigger function however has to be replaced with a mock function. The trigger function is always the same meaning that the mock only need to be implemented a single time and then reused.

The M-layer does not depend on hardware and can therefore be tested on host. Consequently, the developer it not constrained by tools aimed toward embedded software testing. The developer can choose freely among the almost infinite number of existing testing framework and tools which otherwise would not be applicable.
3.4 Hardware

The Hardware-layer (H-layer) holds all target architecture dependent software. The H-layer intents to isolate and abstract away the hardware towards the rest of the code. The H-layer has three main types of functions: Initiation of hardware (configuring hardware functionality at start up), Interrupt handlers (also known as ISR, is a callback function in the firmware of a microcontroller triggered by the reception of an interrupt) and reading/sending signals from/to the environment.

The H-layer is isolated and independent of other apps as well as the P- and M-layers of the same app. However, since the H-layer interacts with the hardware (and, in extension, with the environment) this layer is not completely encapsulated. Figure 9 depicts the general structure of the H-layer. As Interrupt handlers often require global variables to perform their task. For example, a global array could hold a buffer of bytes waiting to be sent by an interrupt-driven communication software (i.e. USART, CAN, I2C etc.). The H-layer is, therefore, allowed to store one structure to provide interrupt handlers with the data needed. In addition, to the two input sources used by the M-layer (function arguments and private state struct), the H-layer uses data received through the hardware peripherals. In Appendix E, the H-layer of an app using USART is included. The program shows how hardware complexity can be encapsulated and abstracted away from the rest of the system without introducing any dependencies at all.

Figure 9: Overview of the H-layer. The blue numbers depicts all types of output and red numbers depicts how an H-layer function can be called.

3.4.1 Testing the H-layer

Automation of tests involving real electrical signals from (and to) the environment is a complex task. Setting up environment that enables this type of tests can be costly both in terms of equipment and time. Tests involving hardware are therefore often performed manually. However, in order to make TDD viable for embedded software development test automation is necessary. The constraints enforced by MPH limits how an H-layer function can be called and the type of output it produces. Figure 9 depicts how a H-layer function can be called (red numbers) and how it can interact with its environment (blue numbers). Table 1 explains meaning of the numbers.
Table 1: Possible ways to call an H-layer function and its types of outputs

<table>
<thead>
<tr>
<th>Possible ways to call an H-layer function (red numbers in Figure 9)</th>
<th>Possible outputs of an H-layer function (blue numbers in Figure 9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. From a P-layer function</td>
<td>1. Return value to caller</td>
</tr>
<tr>
<td>2. As an interrupt handler callback triggered internally (e.g. a clock)</td>
<td>2. Update struct</td>
</tr>
<tr>
<td>3. As an interrupt handler callback triggered from external stimuli (e.g. a USART, GPIO)</td>
<td>3. Send electrical signals to environment (e.g. USART, GPIO)</td>
</tr>
</tbody>
</table>

Due to dependency on target hardware, unit tests for H-layer functions are best executed on the target. Uploading software to the target slows down the development speed and should, hence, be minimized. By executing the tests from host environment and remotely calling functions on the target, the size and number of uploads are reduced by a factor two for ideal TDD development. This since, in ideal TDD development (see 2.9), modification of tests or production code is followed by modification of the other. Between each modification, the tests and software are compiled and executed. If test code is executed from target hardware, then all modifications require uploads. Those tests that are executed from the host environment only require uploads when production code has been changed, i.e. half of the times. Furthermore, the size of the upload is reduced because test cases are not uploaded decreasing upload time even more.

From Table 1 it is clear that automated and remotely (on target from host) executable unit tests for the H-layer requires a system capable of

1. Sending signals to the target
2. Reading signals from the target
3. Remotely call H-layer functions on the target and monitor its return value
4. Remotely monitor the triggered events
5. Remotely monitor and update the content of state struct

If 3) is fulfilled then fulfilling 4). and 5). is simple because helper functions can be implemented on the target and then called remotely. By replacing the trigger with a mock function, the event loop with a broker capable of communicating with the host and by using a programmable hardware setup to send/measure electrical signals, all requirements can be fulfilled. In 6.1.3 an implementation of this setup is presented.

3.5 Pipe

The Pipe (P-layer) lies between the H- and M-layers and should contain as little logic as possible. The purpose of the P-layer is to transport between H- and M-layer functions. It
should also fetch (by calling M-layer functions) and pass data needed but not accessible by M-layer functions. As mentioned previously, by decoupling the M-layer functions, unit testing becomes simpler as mocking is avoided. There will, however, still be a need for data produced by other functions. The P-functions therefore call other functions and pass the data as argument to the “main” function (in Figure 10, the main function could be either the H- or M-layer function next to the highlighted P-layer function). The behavior can be paraphrased as the P-layer “taking the hit” in terms of creating dependencies.

P-layer functions are never called directly. Instead they serve as callback functions. The callback functions are executed in the event-loop as a response to specific event identifiers. The events can origin from both H- and M-layer which both are completely unaware of the P-layer.

Figure 10: Overview of the P-layer. P-layer functions are allowed to call H- and M-layer functions in all apps and are always called as callbacks through the trigger-event-loop.

3.5.1 Testing the P-layer

Clearly, the proposed method results in P-layer functions with a high number of dependencies and, hence, would require extensive mocking for isolated unit tests. However, P-layer are not complex because its functions only fetch and pass data needed by M- and H-layer functions. If complexity arises in the P-layer, the complexity can be moved to helper functions in the M-layer. The helper function can, then, be unit-tested in isolation and called by the P-layer function. The complexity of the P-level can, therefore, be kept at a trivial level. Unit tests for the P-layer can, therefore be neglected since integration and system test will provide sufficient validation.

Due to collecting data for M- and H-layer functions the P-layer becomes highly dependent on other functions and, therefore, troublesome to test in isolation. On the other hand, Limiting the complexity of P-layer functions to trivial levels makes dedicated unit test unnecessary. Instead P-layer functions can be tested implicitly in integration and system tests, because, they are in fact functions for “system-level” integration.
4 Method

The research methods applied in this thesis are here presented; a literature study, combined qualitative methods (interview study and industry seminar) and an experimental validation.

The method applied in this thesis for answering the research questions employs different scientific tools including literature study, an interview study, an industry seminar and experimental validation. The qualitative nature of applied scientific method does not make any claims of statistical certainty. This scientific method builds a combined weight of evidences and well-triangulated conclusions that support hypotheses and propositions that are formed in the study process. This thesis should be regarded as a pre-study with the main result of a well grounded hypothesis of the usefulness of the MPH. This could be explored in future research of a more quantitative nature.

4.1 Literature study

A literature study should according to Brocke et al.[40] reconstruct the giant of accumulated knowledge in a specific domain. The literature study aims to uncover sources relevant for the study as well as preventing any reinvestigation of what is already known. The literature study in this thesis is based on the framework proposed by Brocke et al.[40].

Firstly, the scope of the literature review was defined. In this paper, the literature review was in large part a state of the art analysis. More specifically, the literature formed the basis for identifying current gaps of knowledge within the field in question, which also defined and limited the scope of the thesis. Furthermore, the literature study aided in creating the theoretical framework for the development of the proposed MPH method.

The next step in the literature study included identification of knowledge needed, i.e working definitions of key terms and crucial areas within the research field, for this purpose papers published by Srinivasan et al.[41] and by Cordemans et al.[5] were used. This also formed the basis for the next step which formulated keywords and the conduction of the literature search. The primary keywords used were: Test Driven Development (TDD), Unit Testing, Remote prototyping, Embedded software testing, Test Embedded Software, Qualitative methods in software development and In depth interviews. The search methods included both keywords and backwards searching.

4.2 Interview Study

The interview study aimed to gather a rich body of qualitative data that was used to evaluate and improve the proposed MPH method and provide answers to the research questions. The design of the interview study was, in a large extent according to Turner [42], The steps of this method are described below.

4.2.1 Preparation

The major part of the preparation process was to design the interview questions and the rest of the interview material.

Other important preparations included choosing location for the interviews, inviting and informing the respondents before each interview session. Each respondent was informed about the purpose and the format of the interview as well as the terms of confidentiality. This information was summarized in the invitation email for each interview, and then it was
more extensively explained in the introduction to each session. Some practical preparations included making ensuring that respondents the got access to the interview material and asking their permissions for recording the sessions.

4.2.2 Selecting Participants

The interviews were conducted with senior developers and managers at Sigma technology who are experienced within embedded software testing. The selection of participants was made using referral sampling [43], where one respondent was identified as a starting sample. This person fulfilled the key characteristics necessary for the interview design; having embedded testing experience. This person also had access to a network of other people with suitable characteristics. This network was then used to identify and nominate respondents for the next wave of data collection.

The total sample size of the interview study was limited to five respondents. According to Trotter [43] an ideal sample size in qualitative research can be reached using an interview to saturation method, which dictates that saturation occurs when no new concepts or themes emerge in subsequent interviews. In the case of this thesis, this point was partly reached after five interviews as no new major themes was emerging at this point. However, the small sample size puts some limitations of the generalizability of the results, and adds a risk of biased data, which is also stated as a limitation of our study (see section 1.5).

4.2.3 Designing interview questions

As stated, the interview study looked to gather qualitative data as a basis for answering the first group of research questions: Would reduced mocking make TDD more attractive for embedded software development? Is mocking needed for automated unit tests in embedded software? and Does mocking constitute a significant portion of the total development? Hence, the interview questions must enable identification of limitations and possible improvements of the MPH method. To be able to extract this type of qualitative data the interviews were in large part designed as comparison of two concepts. The respondent was to compare the developed MPH method with a state of the art method from the literature. The questions were divide into four categories: Subject Background, Method Comparison, Implementation and Detail questions. The questions were designed to be open-ended and neutral to avoid any influence on the respondent. The full list of questions and description of the state of the art method MCH can be found in Appendix C and A. The four categories as summarized below.

Respondent Background

The first group of questions focused on the respondents experience within embedded testing and opinions and experience with test driven development. Accordingly, the respondents did self-assessment on how the understand these topics. The objective was to create a context for the rest of the interview where these answers could give a hint on the quality of the collected data.

Method Comparison

In this part of the interview the respondents were introduced to the MPH and the similar Model-Conductor-Hardware (MCH) method (see Appendix A). These were presented in terms of design rules and application in a testing/development process. The respondents were then asked to compare these two methods, identify crucial differences and assess advantages and disadvantages of the MPH method. By using this comparing approach, MPH was put into an appropriate context and aided identification of possible advantages and disadvantages with the MPH method.
Implementation of MPH

The next part focused on the possible implementation of MPH. The respondent was here introduced to one of Sigma’s current development projects where they were developing a complete software platform for an elevator system. The respondent were given a brief description of the project and how the development team was currently working with testing. The respondent was then asked to elaborate if he/she believes that MPH could be implemented in this project.

Code Quality And Reusability

The interviews were concluded by asking questions of a more specific nature. Firstly, did the respondent believe that MPH could be used to increase the reusability of code? And secondly, did the respondents believe that MPH could increase code quality?

4.2.4 Analysing Data

After each interview, the recording was transcribed and data was analyzed, this based on an interview method suggested by Seaman[44]. Seaman’s method suggests the use of field memos where preposition are formed for further analysis during the study process. After each round of interviewing and analysis, there is more data collection to be done, which provides an opportunity to check the preposition that has been formed. By using this technique, adaptations and changes could be done to both the interview material and the interview process to optimize the gathering of the sought after data. As an example of this technique, the first interview led to formation of a certain preposition regarding the MPH-MCH method comparison questions. For the following interview, this preposition could be further investigated by presenting this preposition to the new respondent, who had the possibility of oppose or strengthening the statement. This, in turn, led to the new propositions formed and, in the end, to more exhaustive gathering of the qualitative data.

After implementations of the interview sessions, the set of transcribed data was codified and further analyzed. This was done with a four-steps method where the transcripts were analyzed and a vast set of highly specific themes were identified from the five interviews. These themes were then divided into groups of similar themes which were given a more general topic encompassing all the identified sub-themes. With the identified topics the transcripts was once again analyzed and the vast amount of specific themes were re-sorted to validate the method ensuring that all relevant data was included in this abstraction. The final step included analyzing the content of each of these general topics and identifying a set of specific statements (or themes) that could be presented as concluding results. This set of themes was formulated based on data found under one or several of the general topics, and, together, the set reflect the key opinions and statements found in the interview study. A schematic description of the analyzing process can be seen in Figure11.
4.3 Industry seminar

To further evaluate the developed MPH method and gather a richer body of qualitative data, a seminar was held with developers at Sigma Technology. This is a method for creating a triangulation of qualitative data that adds to the findings of the interview study. The seminar was outlined as a presentation followed by a group discussion. As which the interview study, the objective was to identify limitations and possible improvements of the MPH method.

According to Gillham [45], this triangulation approach is often used when combining regular interviews with group interviews. The seminar form that was used here can hardly be defined as a group interview, which usually includes a very structured outline and a moderated discussion. However, the seminar form used is similar to the type of unstructured group interviews which is very common in exploratory research where a group is presented with the topic that they then discuss freely. Using this analogy, the seminar aimed to add to the findings of the interview study, an an “unstructured group interview”.

4.3.1 Selecting Participants

Embedded developers at Sigma technology were identified as a suitable target group for the industry seminar since they are expected to be familiar with the context of the presented problem and can relate to the proposed MPH method. This choice is also motivated by the fact that this subject group could easily be accessed with the help of management at Sigma, which increased the number of participants compared to what otherwise would have been
possible.

4.3.2 Preparation
Preparations to the seminar included the choice of settings and refining the seminar material.

Since the participants are all developers at Sigma technology the seminar was set to take place at one of the companies regular personnel meetings. This also increased the number of participants.

The material for the seminar consisted of a presentation containing an introduction to the purpose of the seminar, the seminar questions, background leading to the problem statement, and, finally, a thorough presentation of the developed MPH method.

4.3.3 Designing seminar questions
The questions to be answered/discussed during the seminar were:

- Are you familiar with TDD?
- Do you practice TDD?
- Could you identify any gains with using MPH? Motivate your answer.
- Could you identify any costs with using MPH? Motivate your answer.
- Would you consider using the method in a real project?

These questions were presented in introduction part of the seminar and given to the participants in paper hand-outs where they could fill in their answers.

4.3.4 Gathering and analyzing the data
The gathering of data from the seminars were made from the hand-out-forms provided to the participants and through recording of the seminar session. The gathered data was then analyzed. Themes reflecting the the found topics were then identified.

4.4 Experimental Validation on Industry Project
According to Marvin[46], software development research has historically been lacking in validation through implementation. The lack of evidence causes the industry to be more reluctant to implement the findings in their operations. Therefore, to complement the interview study, MPH was experimentally validated through construction of a proof-of-concept prototype of tools required by MPH. The tools are, then, validated by usage in developing of a minor product requested by Sigma Technology.

The project was divided into three parts; selection of validation project, development of tools required, and implementation of validation project according to MPH using TDD and developed tools. Details for each of the project parts are described in the following sections.
4.4.1 Project Selection

Maximizing validation data quality while being executable with given time resources determined selection of the project. For the project, to provide meaningful data, a set of requirements had to be fulfilled. Other specifications would, if fulfilled, increase the richness and/or quality of acquired data. The identified aspects are summarized in Table 2. Time limitations, usage of I/O and the possibility of using TDD were identified as the only critical aspects and hence limited the project selection.

Table 2: Desired properties of validation project

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
<th>Cost</th>
<th>Critical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product has real usage for Sigma Technology</td>
<td>Closeness to real usage</td>
<td>Limited options</td>
<td>No</td>
</tr>
<tr>
<td>Independence to other systems</td>
<td>Development can be carried out solely by authors. No time spent understanding legacy code.</td>
<td>No data on how MPH work with legacy code</td>
<td>No</td>
</tr>
<tr>
<td>Implementable within time frame</td>
<td>Results guaranteed</td>
<td>Limited options</td>
<td>Yes</td>
</tr>
<tr>
<td>Tools implemented in C</td>
<td>Reusable on most processors</td>
<td>Slow development</td>
<td>No</td>
</tr>
<tr>
<td>Target capable of running Python</td>
<td>Author experience can be leveraged for fast development</td>
<td>Very limited selection of micro-controllers</td>
<td>No</td>
</tr>
<tr>
<td>Target using I/O</td>
<td>Crucial aspect of MPH motivation</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Developed using TDD</td>
<td>Crucial aspect of MPH motivation</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

Python is an open-source, interpreted, oriented programming language with dynamic semantics\[47\] originally developed by Guido van Rossum in 1980s\[48\]. It is packaged with a large collection of built-in high level packages which in combinations with dynamic typing and dynamic bindings makes it a great tool for Rapid Application Development[47]. Selecting Python over C means limited reusability of implemented tools as the number of available microprocessors capable of running Python is very limited. However, having large experience in Python development and due to the time limitations, Python was chosen for implementation.

Given limitations of Python and the aspects provided in Table 2, it was possible to identify a project fulfilling the remaining requirements with the exception of implementing the tools in C at Sigma Technology.

4.4.2 Development of Tools for MPH

A trigger-event-loop was necessary for the core functionality software using an MPH structure. Furthermore, isolated unit tests and in extension TDD requires additional tools to be in place before implementation can begin. Given the specifications of the validation project a set of requirements on the tools could be produced. Identified requirements are presented in the list below where 1 is necessary for core functionality, and 2 and 3 are required for unit testing of M- and H-layer functions respectively.

1. Trigger-event-loop
(a) Trigger events
(b) Queue events until consumer ready
(c) Consumer translating events to function calls

2. Unit testing framework for M-layer functions

3. A testing system capable of executing automated tests which
   (a) Measuring outgoing I/O from the unit under test
   (b) Calling functions on the unit under test

As mentioned in 4.4.1, the primary intended usage for MPH is systems implemented in C and a Python implementation can look very different. Design choices during the development of both tools and validation project were therefore controlled so the Python implementation should resemble a C implementation. A conversion to C would therefore be very straightforward and hence it is easy to see that findings from the validation process also hold true for a C implementation.

4.4.3 Validation Project

As part of the selection process (see 4.4.1) specifications of the requested product was determined. The requested product was described as

“... a system which monitors the status for all builds of the [...] project. If all tests in all builds are passing then the main status should be displayed as successful. If at least one test is failing then the main status should change and the failing build or builds should be visible.”

Through further discussion the desired behavior and information of related systems was condensed to the following list

1. The system should monitor the status of all builds of a specific project, given that:
   (a) There are five builds
   (b) The name of the builds are unique, static and known
   (c) When a build has been tested, the result is compiled and sent to an email address.
       The subject of the email contains:
       i. Name of the build
       ii. Test number of the build which increments with one every time the build is tested
       iii. Test results

2. Using GPIO the system should output:
   (a) The last received test result of each build
   (b) The main status of the project should be displayed as
       i. successful if all builds pass all tests
       ii. failed if at least one test fails

With the infrastructure in place, the implementation of the validation project was initiated. The code was developed according to the principles of MPH and TDD by one of the authors. A more detailed description is found in 6.1
5 Interview Study and Industry Seminar

This chapter presents the findings made by the qualitative research method: an interview study and an industry seminar. Due to the qualitative nature of these methods, results are here presented together with the analysis, since in this case they are hard to differentiate.

In parallel with the experimental validation, qualitative methods were used to answer the stated research questions and to evaluate the developed MPH. This is an essential part in the applied scientific method, where the objective is to form a combined weight of evidences and well-triangulated conclusions.

5.1 Implementation of Interviews

The interviews were conducted with five chosen respondents at Sigma Technology according to the method described in section 4.2. Out of the five interviews, two were conducted in person at the Sigma headquarters in Stockholm and three were conducted by conference phone calls. Both authors were present for all interviews except session number 4 where only one of the authors was present.

During the implementation phase some limitations of the initial interview method were identified and alterations were made. More specifically, the first interview was built on a rather minimal interview material containing scripted introduction, questions and unscripted descriptions of the MPH and MCH methods (see Appendix A). Furthermore, in the initial interview process interview material was not sent to the respondent beforehand, instead all the content was presented during the interview session. This relatively unstructured approach resulted in a interview flow where it was sometimes difficult to keep the dialogue with the respondent on track, which in turn affected the quality of the gathered data. This problem led to some improvement of the process and the interview material was fully scripted. This was a vast improvement of the process and increased the quality of the gathered data. Additional improvement was made for the 4th and 5th interviews were the respondents were sent an extensive interview material two days before their interviews appointments. This material contained the same information that was previously presented in the interview script. Consequently the respondents could be better prepared, which improved the quality of the gathered data even more. The material can be found in Appendix A.

The gathered data was analyzed and interpreted according to the method described in section 4.2. Due to the rich and diversified nature of the data set, the process of coding was complicated and many different methods were considered. Finally, the method section 4.2 was employed. The analysis of the transcribed interview produced a large set of specific themes, from which a small set of all-encompassing more general topics were formulated. After a second round of sorting and a final analysis of the formulated topics the final themes were formulated.

5.2 Results and Analysis of Interview Study

The open ended nature of the interview questions produced a rich body of qualitative data. The analysis of the transcripts led to the formation of seven general topics that encompassed all the identified specific themes. Together they constitute the overall reflection and opinion expressed by the respondents and they were used to identify a set of ten final themes which represent the concluding results of the interview study. The formed general topics were: Respondents Background and Views on TDD, Properties of the MPH Method, Comparison
Implementability of MPH and Code Quality/Reusability of MPH. In the following sections each of the topics are presented in detail and how they were used for deriving the final ten themes. On each topic the general body of often diversified data is presented as well as specific statements of some of the respondents. The individual respondents are labeled R1, R2, R3, R4 and R5 (corresponding to the order the interviews were made) to enable tractability of the respondents’ statements.

5.2.1 Respondents Background and views On TDD

The first topic corresponds to the initial part of the interview where questions were asked about the respondents’ background within embedded software testing, and experience and views on TDD. The objective was to create a context for the quality of the gathered data and the respondents’ points of view. More specifically respondents got to assess themselves on a scale from 1 to 5 on their skill level within given areas (with 5 being the highest level).

Overall, respondents R2, R3, R4 and R5 rated themselves 3 in terms of general experience within testing, and R1 rated him/herself as a 4. A typical statement here is exemplified by R2 who rated him/herself a 3 and said:

“I would say I am about a 3 [...] Since well, I have mainly worked with development and sometimes even neglect testing since I am a developer, on the other hand I have had some considerable working experience with it.”

On the subject of TDD, the respondents rated themselves between 2-4. R3 and R4 rated themselves as 2, R1 rated him/herself as a 3, and R2/R5 rated themselves as 4. An example statement on this matter made by R1 was:

“I have not worked with it practically, but I have worked a great amount with automation [...] It was more within the general scope of the testing practice in the company. [...] But not in a pure form (TDD)”

A part the interviews were designed to gather data on the respondents’ opinions on TDD and its application in embedded systems. This created further background context and supported findings in the literature study.

In general, the respondents were very positive towards TDD, and the benefits of the method were recognized across the interviews. Aside from this, the discussion on TDD led to understanding of possible drawbacks and challenges in introducing TDD. For example, respondent R4 stated that TDD is indeed a superb tool, provided that fast running test suites can be run in short intervals. The method gives a structure for testing and enforces modular code which provides great support during development. In many cases TDD also results in a better design of software as all code is testable.

R4 made the following positive statement about TDD:

“It becomes a great support with modular code or test driven method. Because it enforces that you write your code in a very modular way. This becomes obvious when you do test driven development since you automatically create a structure which enables testing. Furthermore, you remove unnecessary functionality which is a huge up side.”

Some challenges with TDD were also brought up across the interviews. The target platform was identified as a limiting factor since it can be associated with long build times and in many cases getting the results from tests can also cause delays. This will disrupt the development speed that is essential to the TDD cycle. Another challenge with TDD that was recognized was that it often requires a rather steep learning curve. On this matter R4 stated:
“You could put it like this, it is a threshold to get started with TDD [...] It is a bit like starting with the dessert first.”

Finally a potential drawback with TDD is when that test suite will become big and complex. Time must therefore be spent refactoring the test code in order to maintain it manageable. R4 commented on the issue as follows:

“A great drawback is when your test cases start to become a huge system. The result of this may be that you need to put a lot of work into keeping your code clean so it passes. Then the question is if you should do it at all.”

Overall, the gathered data suggested that the respondents were experienced with testing and are familiar with, but do not practice TDD. However, the general attitude towards TDD was positive and many of its advantages/disadvantages were recognized, such as increased modularity and testability, at the cost of a possible increase in time spent in maintaining the tests.

5.2.2 Properties of the MPH Method

A major part of the interview sessions focused on MPH, where the respondents discussed its properties, advantages and disadvantages. Generally the respondents held a positive attitude to MPH method and identified several benefits with the method. Among these were isolation of functions and reduction of mocks. This implies that MPH could potentially reduce overhead in TDD. This statement presupposes that mocking is a time consuming undertaking that disrupts the TDD cycle. Which was also confirmed by several of the respondents, who either had first hand experience with designing mocks or had worked in projects where mocking was involved.

However, it was also stressed by all of the respondents that in the presented concept stage (see Appendix A) it is very hard to determine if MPH could be implemented. Consequently, a majority of the respondents concluded that they wanted to see examples of implementation before they could make any definite evaluation. Some respondents went a step further and stated that the method also needed to be tested and validated against some appropriate metric. The topic of mocking and the need for implementation is well summarized by R1 who stated the following:

“From my experience mocking takes a lot of time, also in larger projects. So if you can reduce mocking that would be an advantage. If this [talking about MPH] does that, I cannot say. [...] This is something that has to be more carefully evaluated.”

A third important aspect that was commonly occurring in discussion about MPH was that the method itself seemed difficult to understand on the presented concept level. When the method was presented most of the respondents had questions about specific features of the method and there were also problems understanding some of the general aspects. For example, there was a confusion about what level of testing MPH is addressing.

A typical example here was the interview with R2, where there was a discussion about the P-layer and how you would avoid mocking it. It was explained to this person that by using a trigger-event loop, the trigger function is the only object that requires a mocking for testing purposes. This led into a discussion about how the trigger event loop is designed so that the trigger mock can be reused across projects. Furthermore, the respondent raised questions regarding how the apps communicate with each other, and the possibility of having pure M-layer apps. This was discussed in the form of an example elevator system. The result was that he/she agreed that this could indeed be possible by letting M-layer communicating
indirectly with each other through P-layers.

Overall, similar aspects were discussed in most of the interviews. Most of them focused on the trigger event loop and how the passing of arguments would work. This kind of data points out two things. Firstly, it is clear that it was difficult for the respondents to understand and be able to evaluate the technical aspects of MPH based on the information they were given. Secondly, the data gathered on the properties of MPH was useful for development and implementation of the method in the parallel validation project (see 6). However, once again, emphasis was put on that the method needs further development and implementation, as an example of this, respondent R5 stated:

“[Talking about MPH] I believe you are on the right way but I would like to see a concrete example of how it works.”

5.2.3 Comparison MPH/MCH

The interview study was in large part designed to be a comparison between MPH and the similar MCH method (see Appendix A). The aim with this approach was to create a frame of reference which would aid the respondents in understanding the MPH, and identifying possible weaknesses and strengths with the method. The data collected on this topic constituted a major part of the interview results and many of the final themes were identified here.

Mocking

When the respondents compared MPH to MCH the practice of mocking was one of the most commonly occurring topics, and it was identified as the biggest difference between MPH and MCH. A majority of the respondents stated that MPH would have definite advantage over MCH in terms of overhead. It was indicated that MPH could reduce overhead since it reduces mocking and consequently MPH would, in most, cases be superior to MCH. As an example of this R2 stated:

“I do not see any advantages with MCH [...] I can not see any reason for going into this mock-frenzy. You write a lot of code that you will need to maintain, without adding any value. Here you can start your testing at once (MPH).”

Application on different types of projects

Comparing MPG to MCH in general, yielded discussions about scenarios in which one of the method would perform better than the other. The data was diverse and different views were present.

Respondents R2 and R4 strongly advocated that MPH would be better in most cases since MCH seemed too complex and includes an excessive use of mocks. It was argued that MCH would require a great deal of engagement from software engineers, in terms of enforcing the method in a development team. It would also involve keeping track of a considerably more complex file structure. Furthermore, MPH would probably be faster to implement and could be used in early project stages where rapid prototyping is an important part of the development process. Finally, the method is more similar to the ones that are practiced in the industry. The motivation being that industry practitioners heavily rely on ad hoc solutions instead of trying to fit reality to a model. The respondent stated that they believed MPH could more easily be adapted to problem at hand when compared to MCH. This connects with another more general remark, made by many of the respondents; the use of standardized design patterns is not always an optimal approach in development projects. In many cases, such approach can create unnecessary complexity. R4 discussed this viewpoint in detail. During, the following statement was made:

“In some cases [...] Well, you want to have this standardized model as a starting point but considering how the system is to be designed this might not be the
optimal approach, which, as a result, can make the end result complex and not very logical as a solution. [...] and you get a more logical approach if you are able to make some adaption yourself. If you use a standard method, the implementation could end up hard to understand and not suitable to the system in question. [...] That, I would say is an advantage with your method (MPH), that it is more adaptable” [...] From that point of view it is clear that your model (MPH) gives a better understanding.”

Other data on the subject of applying MCH and MPH in different types of projects were much more balanced. And respondents recognized that both methods had their own benefits that could be employed in different types of projects. On one hand, respondents stated that MPH as a more lightweight method, would be more suitable for less complex projects where there is little or no legacy code. Ideally, it should be used in development projects where software is designed according to MPH from the beginning. On the other hand, many respondents expressed that MCH could be of better use in projects focused towards quality improvements or development on existing systems.

**The Test And Development Process**

Another comparison made by the respondents was on how MCH and MPH approach the design process. Diverse opinions were expressed. However, in general, MCH was perceived as a top down method and MPH was seen as bottom up approach.

Many of the respondents statements favored MPH:s way of starting to develop parts of the functionality even though the complete system designed is not exactly known. Respondent R1 summarized this line of thought with the following statement:

“If you are supposed to build this functionality in this system, you know that will need to be in place at some point but you are not exactly sure on how it should behave. [...] and you can start making your MPH triads² for the components you will need. Then you will get a lot of structure, to which you can later add logic into the M layers to use the triads in a way that you want. This way it is also easier to interface everything.”

Once again, in accordance with previously presented evidence, discussions regarding mocking indicated that elimination of mocks would result in a considerable reduction of overhead. Furthermore, the bottom up approach of MPH was seen as a more agile development approach. This was perceived very positively.

When discussing MCH, it was indicated that there was, indeed, an advantage with the top down approach since it allows for planning the software structure beforehand. However, it was also stressed that even if this is good in theory, it is often very hard to do, especially if the end functionality of the system is now entirely known.

It should also be noted that R3 had opinions that considerably diverged from the rest on these topics and argued that the top down approach of the MCH was preferable to MPH. He/she stated:

“Some kind of an overview would be good form the start [...] Yes, I would like to have overview on the whole system from the beginning”

He/she argued that the MPH bottom up approach would cause developers to focus more on the individual modules in the software, this risking to miss the big picture.

²A triad refers to the M-,P- and H-layer of an app
“You will get a good detail level but the whole system might not turn out as good. [...] Then it will be no one that overlooks the big picture so when changes are made to a some small part of the system it might result in destroying the overall functionality.”

This somewhat weakens earlier findings indicating that MPH is good for reducing overhead and that MPH is better in most types of projects. However, it should be added that this respondent had considerable problems understanding the presented concepts of MPH and MCH.

To summarize the discussions focused on comparing MPH and MCH methods, we conclude that MPH was perceived as superior to MCH in most types of development projects. This is because MPH is more adaptable and lightweight while MCH could be a useful method in a more complex development project. We also found that the use of these types of standardized design pattern methods are considered to not always be the optimal approach. On the contrary, it could, in some cases result in more complexity and thus overhead. Furthermore, by comparing MPH and MCH, it is, once again, confirmed that the reduction of mocking in MPH is a great advantage and would reduce overhead.

5.2.4 Implementability of MPH

A significant part of the interview session was focusing on the possibility of implementing MPH. Note that this data was gathered at a time preceding the industry project implementation (see section 6.1).

On this matter respondents specifically discussed whether MPH could be implemented on the presented Sigma elevator project. (See Appendix A). On this topic, once again, the issue of the current lack of validation and implementation of MPH was discussed. However, across the interviews, it was indicated that there was nothing explicitly hindering MPH from usage in the elevator project. Emphasis was put on the fact that in this case, a completely new software platform would be developed and MPH could be used from the very start. This was something that many of the respondents viewed as a prerequisite for the application of MPH. The elevator project was also perceived as a project with a relatively low complexity which would suit the MPH method well.

Detailed data was also gathered on more general aspects of implementing MPH. Respondents further strengthened the notion that MPH is implementable in industry projects by stating how the method in many respects is similar to some practices used by the respondents. R4 specifically discussed a development project where a test fixture was developed. The respondent stated that MPH would have been very useful in that particular case as it would have provided structure and assurance that all code was testable. To add to this line of thought, R2 stated the following:

“MPH would probably work in almost all types of project, no matter if you have access to the hardware or not.”

However, the data gathered on this topic also points to some potential problem areas in implementing MPH. First of all, legacy code and development on existing system was identified as possible constraining factors. A single module to be later added to the system could possibly be developed with MPH and then integrated to the rest of the system. However, the MPH approach would not be feasible if it included making any significant changes to the excising system. This would cost too much money and/or time and stakeholders would, in most cases, only see the associated risks with such an undertaking. Respondent R5 summarized with the following statement:
“If you have something that works, no one will say that we should do this [talking about MPH] just to make the code work better.”

The complexity of the system was another implementation aspect that was discussed by the respondents. Here, it was advocated that MPH would be easier to apply to less complex systems such as the previously discussed elevator project. Once again, emphasis was put on the fact that MPH should be included into the development process from the start if it is going to be used.

Another issue was related to the introduction of MPH into development teams and ensuring that the method is followed. It was stated that this is not a problem that applies only to MPH but to all types of design methods. This is connected to the previous findings regarding how the usage of standardized methods such as MPH always bring a trade off. R2 said the following on this topic:

“A general drawback with these kinds of methods, not specifically regarding MPH or MCH, is that if you do not define rules, so that method practice is automatically followed, they will fail. And, as always, the implementation in the form of adaptation and improvement work in the development teams is always difficult. [...] But this, of course, applies to all models.”

Note that the above does not weaken the arguments that MPH would be implementable. However, it is something to take into consideration when introducing the method in real projects. This aspect will have to be further investigated, which we delegate to further work.

Finally, strict real time requirements on a system were identified as a possible source of problems. The effect of using a trigger event loop on real time behaviour is not known. This is a critical aspect, since the trigger event loop is one of the core features in MPH. Therefore, how this will impact and possibly limit the implementability of the method must be further investigated.

5.2.5 Code Quality and Reusability of Code

Code quality and the matter of reusability of code are aspects that were part of the interview design and resulted in some observations that can be associated with the previous findings.

On one hand, a majority of respondents stated that the MPH impact on code quality cannot be assessed without more implementation. However, one possible indicator of higher code quality is that strict design rules of MPH would lead to well-defined interfaces and well-structured code. On the other hand, the rules could also lead to overly complex.

The matter of reusability was also hard for the respondents to evaluate due to the lack of presented implementation. Once again, however, the well-defined interfaces were pointed out as an advantage with MPH. It was indicated that, if the code is designed according to strict rules, it would probably help reusability. The trigger function, and certain types of apps for simple functionalities were specifically pointed out to be reusable. The M-layer was identified as a crucial component and that its design would determine if the apps are reusable or not. The conclusion was that M-layers in more trivial systems would be easier to reuse than the ones in more complex systems. R2 stated on this topic:

“Then it depend on what kind of M-layer you have, if you are making a scheduler that could be reused. Or, if you are making a small web server, it will also be reusable. [...] But, normally, if you are doing a business logic for an elevator you cannot reuse it for a business logic in a truck”
In general, there were some uncertainty of how MPH would impact reusability and it was not possible to derive a definite conclusion. Even, if some gathered data indicated that MPH could increase both code quality and code reusability, these aspects of MPH were hard to evaluate the presented concept state. Despite this, the respondents concluded that the major gains with a method such as MPH are on a more structural level; setting up environment; rules for how to design/test etc. Respondent R2 stated:

“But I believe it is more on the structure, to be able to quickly setup your environment for compiling, how the test should be run where the reports go etc. [...] And I believe that the real benefit here is the general one for TDD.”

This suggest that the greatest technical quality advantage of MPH would be in introducing TDD with its general advantages.

5.3 Summary of interview results

By analyzing findings presented in the above topics a set of ten identified themes that encompass the main outcome of the interview study were identified. These themes are presented below.

1. MPH seems to be a good approach to reduce overhead and enable TDD.
2. It can be confirmed that reducing mocking would significantly reduce overhead.
3. Trigger-event-loop could eliminate the need of mocks.
4. MPH could be implemented in industry development projects, possibly restricted by system complexity, real time requirements and legacy code.
5. MPH is more adaptable and lightweight than MCH. In general, MPH is perceived as a superior method.
6. MCH is applicable to more complex systems, and projects involving quality improvement and legacy code.
7. The use of standardized design patterns is always a trade off.
8. Understanding the MPH/MCH methods is difficult when presented on a concepts level.
9. Respondents were experienced with testing and are familiar but does not practice TDD.
10. Thus, MPH must be further developed and tested to be thoroughly evaluated.

We conclude that MPH seem to be a suitable approach for reducing development speed overhead associated with TDD. Based on the fact that MPH would reduce mocking which is now clearly confirmed as an overhead. MPH is, therefore, superior to the comparable MCH method, which relies heavily on mocking and was perceived as less flexible and a more complex method, in general. However, at this concept level, no definite conclusions regarding capabilities of MPH can be drawn. More implementations and evaluations are needed. Furthermore, the MPH methodology could likely be applied in industry projects, except projects with legacy code and hard real time requirements, which are possible limiting factors for MPH.
5.4 Implementation of Industry Seminar

To complement the qualitative data gathered in the interview study, an industry seminar was conducted with embedded developers at Sigma Technology, see 4.3. This industry seminar took place at one of Sigma Technologies personnel meeting and included 19 consultants working with embedded software development. In the seminar, participants were first introduced to the purpose and to the questions that would be discussed. This was followed by a presentation of the seminar material including problem statement and a detailed description of the proposed MPH method.

5.5 Results and Analysis of Industry Seminar

In total, 13 of the participants filled in the question forms that were provided in the beginning of the seminar. Together with the analysis of the seminar discussion these forms served as the basis for identifying a set of themes that formed the concluding results of the seminar. These themes reflect the design of the seminar questions, and they are here presented in three categories: Experience with TDD, Gains And Advantages Of MPH, Costs And Disadvantages of MPH and Other Remarks.

In order to gather some data about participants background within TDD, they were asked to fill in if they were familiar with TDD and if they practiced the method in their daily work when developing embedded software.

5.5.1 Experience With TDD

In total, 13 answers were received and they were as follows:

- 11 People stated they were familiar with the TDD method.
- 2 People stated that they usually practiced TDD when developing software.

This indicates that the developers at Sigma were familiar with TDD but did not practice it.

5.5.2 Gains and Advantages of MPH

The participants were asked to fill any positive feedback they had on MPH, and, even though the amount of data was very sparse, three themes could be identified. These were as follows:

- MPH could increase the test possibilities in embedded projects.
- MPH could be used in a real project if the product is designed according to MPH principles from the beginning. Otherwise, massive refactoring work would be required.
- Strictly defined dependencies used in MPH give similarity between software blocks independent of the individual developer.

The first theme regarding increased test possibilities is interpreted in the light of how MPH enables automated unit testing of hardware-dependent code, confirming the positive effect of using MPH. The second theme is crucial since it supports implementability of MPH but it also clearly restricts the application to projects where completely new systems are developed, that is projects not involving any legacy code. The third theme is also interesting since the aspect of similarity is a subject not discussed earlier. This could be connected to an aspect that was partly discussed in the interview study where respondents believed that the most substantial quality gain of MPH would be in the form of having an established practice of software development.
5.5.3 Costs and Disadvantages of MPH

Participants were also asked to fill any negative feedback that they had on MPH. This resulted in four identified themes which were the following:

- MPH could introduce a risk of hiding complexity in the P layer.
- Problems to integrate with real time dependencies.
- Interface between different software layers is not always clearly defined.
- Could be hard to integrate with larger systems with a high degree of complexity.

The first theme was identified from the hand out forms but it was also discussed during the seminar. The argument here was that MPH would introduce the possibility for developers to put complex functionality, which is hard to test, in the P-layer. Since the P-layer is not explicitly unit tested in the MPH development process, these functions would not be unit tested at all and would ”simplify” the development of the H- and M-layers, this sifting functionality to the P-layer. However, this would cause problems later in the design process when all layers are integrated and tested.

The second theme clearly states that, by using MPH and introducing trigger-event loop, problems could occur when dealing with hard real-time dependencies. The third argues that it could be hard to define interfaces between different layers, that is what should be in the M-layer and what should be in the H-layer. This is related to the problem pointed out in the first theme. Finally, it was stated that MPH could be troublesome to integrate with larger systems and other systems, in general, which have been designed with another methods. This connects with the second theme identified in section 5.5.2.

5.5.4 Other Remarks

The seminar discussion and the hand out forms also identified some other aspects. These were:

- Investigate how the P-layer grows with the size of the code base.
- The background and problem description should be more specific in terms of what problems the MPH method is aimed at.
- It should be clearly stated that MPH is used when develop “code not something you run in overnight test or in some other kind of interval testing. It is when you sit by the keyboard and you do not want to wait.”

The first theme identifies an interesting area for possible further research while the second and third are aimed at how the MPH method should be presented.

5.6 Combined Result of Qualitative Methods

Together, the interview study and the industry seminar establishes a notable body of qualitative data that can be used to evaluate the MPH method. When comparing results of evaluation of MCH and MPH, similar groups of resulting themes can be formed. These groups are presented in table 3 below.
Table 3: Groups of identified themes from interview study and industry seminar

<table>
<thead>
<tr>
<th>Group</th>
<th>Interviews Theme</th>
<th>Seminar Theme</th>
</tr>
</thead>
</table>
| 1     | • MPH seems to be a good approach for reduce overhead and enable TDD.  
      | • It can be confirmed that reducing mocking would significantly reduce overhead. | • MPH could increase the test possibilities |
| 2     | • MPH could be implemented in industry development projects, possibly restricted by system complexity, real time requirements and legacy code. | • Problems to integrate with real time dependencies.  
      | | • Could be hard to integrate into larger systems with a high degree of complexity. |
| 3     | • Respondents were experienced with testing and are familiar but did not practice TDD | • 11 out of 13 participants stated that they were familiar with TDD  
      | | • 2 out of 13 participants stated they practice TDD |
| 4     | • Understanding of the MPH/MCH methods is difficult when presented on a concepts level. | • The background and problem description should be more specific in terms of what kinds of problems the MPH method is aimed at.  
      | | • It should be clearly stated that MPH is used when developing code not something you run in overnight test or in some other kind of interval testing. It is when you sit by the keyboard and you do not want to wait |

As seen from the table, the combined results of the interviews and the seminar suggests that MPH would indeed reduce overhead and enable TDD in embedded software development. This confirms that the implementation of mocks in software development is a considerable cost. The data also points out that MPH would be implementable in most types of embedded software development with some possible restrictions in the form of: *Real Time Constraints, Implementation in Complex Systems Development, and Development Including Legacy Code.*
Finally, the study and the seminar indicate that MPH would be able to enhance the development and testing within embedded development. Furthermore, the results of the interview and the seminar strongly suggest that MPH must be further implemented and validated on a real industry project before anything explicitly conclusive can be argued about the possible gains and cost with the method. This leads to the next part of the thesis, Implementation and Validation of MPH on an Industry Project, that will be presented in the next chapter.
6 Implementation and Validation of MPH on an Industry Project

In this chapter, implementation and validation process of MPH is presented in detail. Results are highlighted with the corresponding analysis.

As previously discussed, an essential part of answering the research questions in this study is by the means of using experimental validation of the MPH method. The purpose is to investigate implementability of MPH and if it can enable TDD on an industrial development project. The applied selection method resulted in development of MPH with necessary tools and their validation; process of applying the MPH method; and a TDD approach to the chosen industry project. This chapter will present these topics followed by presentation of results and analyses of the experimental validation.

6.1 Implementation

Sigma Technology develops embedded software using agile methods such as TDD and Continuous Integration (CI). The very core of CI is building and testing software as frequent as possible. As strong believers in CI, Sigma uses the Bamboo by Atlassian[49], a continuous integration server, to monitor its repositories for newly checked code by its developers. When changes are detected, the software is built together with its test suite and run. Once completed, the result is communicated to the team via email. As part of Sigma’s dedication to keeping all builds free from failing tests, a device capable of monitoring and displaying the test results in an indiscreet fashion was requested. The requested system requires proper I/O, has very few dependencies to other systems and should be implementable with the available time resources. The system constitutes a good target for testing the MPH development methodology.

As earlier mentioned, implementation of MPH included development of the MPH-necessary tools and the application of MPH and TDD in this Sigma project. This section continues with a detailed description of the development of the MPH features, hardware and software environment, trigger event loop and automation of test on hardware dependent code. This is, then, followed by presentation of implementation details of MPH in the Sigma project.

6.1.1 Target Hardware and Software Environment

As discussed in 4.4.1, due to limitation in time resources, Python was selected for our implementation. Although there exist Python versions which run directly on specific hardware[50], Python generally requires an operating system intended for personal computers such as Windows, Linux or OS X. The Raspberry Pi[51] is a low cost, ARM-based[52], open source, credit card-sized single-board computer, which has become extremely widespread since its introduction. As the name suggests, a single-board computer is a fully functioning computer built on a single PCB. Raspbian[53] is a Linux distribution intended to run on the Raspberry Pi. The operating system is based on Debian Wheezy armhf[54] and comes packaged with Python. With its 26 GPIO pins, the Raspberry Pi fulfills all necessary requirements for the industry validation project and was therefore selected as the target hardware.

By using a Linux-based operating system capable of running Python, authors could leverage previous knowledge for fast prototyping. However, in order to simulate restrictions imposed by a less equipped operating system, artificial restrictions were placed upon the design. The implementation was, therefore, not allowed to rely on the Raspbian to schedule tasks but should instead simulate the behavior of an interrupt driven scheduler using an internal clock.
6.1.2 Trigger-event-loop

Before the requested product could be developed, infrastructure had to be designed and implemented. Being the most essential component of the MPH system, the trigger-event-loop was designed and implemented first. If implemented on hardware without an operating system, a trigger-event-loop could save triggered events in a global queue and let the main loop consume events when available. With little divergence from the principles of the C-implementation, a similar system can be implemented in Python using a message queue. In short, a message queue allows asynchronous messaging between applications. Basically, it allows applications called producers to place messages into the queue. Other applications, called consumers, can then retrieve messages when they have time. RabbitMQ[55] is an AMQP[56] message queue implementation with several maintained Python bindings. Using RabbitMQ and the Python binding pika[57], the trigger-event-loop could be implemented.

The trigger-event-loop has two main components, a trigger-function and a set of callback functions. Once an event has been triggered, it has to be passed on to the consumer. It is possible that one or more events are triggered before the callback of the last one has finished. Hence it is necessary that the system is able to queue triggered events. The trigger-event-loop was implemented using RabbitMQ as message queue. In Figure 12, a flowchart of the trigger-event-loop is displayed.

A call to the trigger function can be thought of as a piece of code which has come across "something interesting" and wants to tell the rest of the system of this discovery. A message is, therefore, put into the message queue to be consumed by the event-loop. The nature of the trigger is described by the event identifier but in many cases this identifier must be grouped with some data. Take, for example, an app managing a communication channel using USART. Whenever a new byte has been received, an interrupt is triggered and an H-layer function extracts the byte from the register. Incoming data is of interests for other
part of the system and, hence, an event should be triggered. However, if the event is not
grouped with the new byte then it has limited value to whoever is listening. Therefore, it is
not enough for a message to hold the identifier. It must also include relevant pieces of data.
As the listener receives a package, containing an identifier and data, the listener maps the
identifier to a specific callback function. The callback function is then executed, using the
grouped data as argument to the function call.

JavaScript Object-Notation or JSON[58] is an open standard that uses human-readable text
to transmit data objects in a key value structure. The format is language-independent. Since
decoders and encoders are available for almost all languages, it has become an universal lan-
guage used in a wide range of systems.

RabbitMQ represents messages as strings and Python is equipped with JSON encoder/de-
coder, by default. Hence, the format was suitable for our task. The trigger function takes
a string as the event identifier and a variable number of arguments for the data of interest.
The string identifier is inserted into a Python-object together with the list of data argu-
ments. The object, is then, encoded as a JSON-string and inserted into the message queue.

The main-loop of the system acts as a consumer of the trigger-event-loop. The behavior of
the loop is shown in the right part of Figure 12. If the message queue is non-empty, the
message in the first position is fetched and parsed using Python’s built in JSON-library. The
event identifier is used as the key in a hash table holding lists of callback functions. Using
the list of arguments, each callback function is called with the arguments. Once finished, the
loop starts over again with a new check for available messages.

6.1.3 Automating Tests of Hardware-Dependent Code

Testing platform had to be developed for unit tests of H-layer functions. An H-layer function
can either be called from the P-layer or as a interrupt callback and the possible outcomes
from the functions are sending electrical signals, returning data to the caller or triggering an event. The only hardware functionality used in this project was communication with GPIO-ports. Hence, a capable test platform must be able to:

1. Measure I/O from the target under test
2. Send I/O signals to the target under test
3. Remotely call functions on target and retrieve its return value
4. Monitor the message queue

The ROBOT Framework is a generic test automation framework for acceptance testing and
acceptance test-driven development (ATDD)[59]. The framework has a simple and verbose
syntax which creates self-explanatory test cases, understandable by both developers and non
technical-stakeholders. The framework has a set of built-in libraries and can be extended
with new libraries written in Python or Java. The modularity of ROBOT in combination
with the easy-to-understand syntax motivates its usage in embedded system development
where not everyone is knowledgeable in software development. ROBOT Framework was,
therefore, selected for the test automation of H-layer tests.

Raspberry Pi is equipped with 26 GPIO pins, easily accessible in Python using the open
source module RPi.GPIO [60]. Using RPi.GPIO a ROBOT framework library could be im-
plemented fulfilling requirements 1) and 2) from the above list of requirements on the test
platform. The library provides ROBOT with keywords allowing it to set pins as "input" or
"output", draw the pins "high" or "low" and activate "pull up" or "pull down" resistors
when available. Calls to the library are verified and exceptions are raised with descriptive messages if illegal calls are attempted. This means that, if a test is designed badly, for example activating a "pull up" resistor on a pin that is not equipped with one, this test will fail and "explain" why it failed. The developer, therefore, can be convinced that the test cases, in themselves, are working. Hence, the source of a failing test is limited to the software under test. A simple example fulfilling the mentioned requirements is displayed in Listing 3. The example is a ROBOT Framework program that report successful test results if and only if pin 7 and pin 9 on the Raspberry Pi have been connected to each other. The test in Listing 3 exhibits easiness of GPIO-dependent test construction.

```
*** Settings ***
Library GPIOLibrary
Suite Setup Setup Pins

*** Variables ***
${PIN A} 7
${PIN B} 8

*** Keywords ***
Setup Pins
  Set As Output ${PIN A}
  Set As Input ${PIN B}

*** Test Cases ***
Check High Pin A Draws Pin B High
  Set Pin As High ${PIN A}
  Check Pin Is High ${PIN B}

Check Low Pin A Pulls Pin B Low
  Set Pin As Low ${PIN A}
  Check Pin Is Low ${PIN B}
```

Listing 3: ROBOT Framework test for checking if two pins are connected

In the trigger-event-loop described in Section 6.1.2, as the consumer, retrieving messages from the queue and translating it into the function calls in the P-layer. When unit testing H-layer functions, this behavior is not desirable. The main-loop is, therefore, replaced by a broker capable of communicating with the Raspberry Pi running the ROBOT Framework test suite.

Replacing the main-loop with a broker makes it possible to fulfill both requirements 3) and 4) from the list of requirements on the test platform. Only the H-layer is tested and, hence, callbacks, otherwise produced by the main loop, must be disabled. Instead, messages must remain in the queue until Raspberry Pi tester requests their retrieval. Using Ethernet and the two Raspberry Pi's the units are connected to the local network and messages are sent between the two units over the local network using TCP network protocol[61]. A ROBOT Framework library provides an easy-to-use interface for the developer. The library encodes messages as JSON-strings containing a list of arguments and the name of the function which the tester wants that the target to call. When receiving a message, the broker parses the message using Python’s built in JSON-library and checks if a functions with the supplied
name exists. If it does, the function is called with received arguments. The return value is encoded as a JSON-string and sent back to the tester using the same TCP socket. If the function call fails to execute (raises an exception) or if the requested function does not exist, a JSON-encoded message containing the type of the error and the error-message itself are both sent back to the testing unit. The message is parsed by the ROBOT framework library and evaluated. A high level schematic of the system is displayed in Figure 13.

![Figure 13: Tester unit and the broker communication](image)

With the broker in place, monitoring the message queue is achieved by implementing a simple message queue "getter"-function for the target system. The function can, then, be called remotely and the result can be compared to the expected ones. When combined with the GPIO library, the platform can test H-layer functions completely. The two test cases in Listing 4 combine all four types to test if an external interrupt handler triggers a specific event when its pin is drawn "high", and if another pin is drawn "high" when **light_success_led** is called.

---

3 A socket is the unique identification to or from which data is transmitted through a network.[62]
Listing 4: ROBOT Framework test using all four types of interactions, (1) sending I/O, (2) sensing I/O, (3) remotely calling functions and (4) inspecting the message queue.

6.1.4 Validation Project

The implementation of the actual production system was initiated by identifying the main functionality and what data flow was required. Possible subsystems were identified and classified as separate apps. Within each app, main functionality was split between the H-, M- and P-layers. Incoming commands, which should trigger P-layer functions, were specified using descriptive names. Potential outgoing signals, which other apps may need, were identified and given descriptive names. The process was repeated for all identified apps.

Given the set of applications, outgoing and incoming signals were matched, creating directed edges between apps. The directed edges form a model of the flow between triggers and listeners. Outgoing signals, which could not be matched with an incoming signal, were removed. This concluded the system design for the requested product. The final system design is presented in Figure 14.

The development of the requested system was successfully carried out using MPH and TDD. The software was developed in Python on a Raspberry Pi 2 B+ and an overview of the software is shown in Figure 14. The directed edges correspond to triggered commands and listeners.
Figure 14: System overview of the industry product

The system, depicted in Figure 14, is divided into three apps, one managing the email account, one for scheduling and one for GPIO. The scheduler uses a clock to periodically check for scheduled tasks.

In a typical embedded system, interrupts of an internal clock would create the periodic behavior but clocks and interrupts does not exists in Python in the same way as it does in low-level languages. To simulate the interrupts triggered by an internal clock a dedicated thread is spawned tasked solely with executing the interrupt-callback periodically. The initiation of the timer, init_interrupt_timer, and the periodic callback, interrupt_timer_isr, are both located in the H-layer of "Scheduler". The interrupt-callback triggers an event, called CHECK_SCHEDULE, on each call. As seen in Figure Figure14, CHECK_SCHEDULE is caught by the P-layer of "Scheduler". CHECK_SCHEDULE is mapped to the callback trigger_scheduled which in turn calls get_scheduled in the M-layer of "Scheduler". get_scheduled is a simple scheduler which triggers events periodically. In this software, only one event is triggered; CHECK_EMAIL. As the app does not contain any I/O, all software is tested by using Python’s built in unit testing framework; unittest.

The "Email" app listen to CHECK_EMAIL, using check_email as callback function. call_email, first, call fetch_email in the H-layer. fetch_email downloads the email from an IMAP-server using build names and test number for searching, and returns the raw data to check_email. check_email pass the data to parse_email, which compares the data to previous calls. The new email data is saved locally for future checks. Finally two events are triggered; BUILD_STATE_UPDATED and PROJECT_STATE_UPDATED. Each identifier is grouped with data describing the states of the builds and project respectively. Just as the scheduler,
the email app was tested using Python’s unittest.

Triggered by BUILD_STATE_UPDATED and PROJECT_STATE_UPDATED, the "GPIO" app calls update_state as callback function. update_state passes the data, grouped with the identifier, to state_to_pins in the M-layer. As the name suggests, state_to_pins, maps state-data from builds and project, to pin states. The pin states are returned to update_state, which calls display_state with the returned values. display_state, located in the H-layer, displays the result by controlling GPIO-pins in order to turn LEDs on and off. As state_to_pins is located in the M-layer and, hence, hardware-independent, the function is tested using Python’s unittest. display_state, is located in the H-layer and interacts with the hardware. The function is tested using the remote call and the GPIO library developed for ROBOT Framework. Tests were written before production code according to TDD and implemented incrementally.

6.2 Results and Analysis of Industry Project

Overall the implementation proceeded as expected and both the tools and the validation project were successfully finalized. This section summarizes and analyzes the insights gained on the strengths and weaknesses of MPH. First, general results and analysis of both the implementation of the MPH testing platform and the validation project are presented. This is followed by more detailed analysis of specific aspects of interests that were found in the overall implementation process. These areas are treated under the Programming Language and Hardware Choice and Transferability, which comments on representativeness of the conducted implementation, and the Trigger-Event-Loop, which discusses some identified consequences of using a trigger-event-loop. Finally, identified weakness in MPH are discussed under the Identified Weaknesses in MPH.

6.2.1 Testing Platform

Our testing platform was designed and implemented using a Raspberry Pi, Python and ROBOT Framework. The platform can send and measure GPIO signals from all its 26 GPIO pins of the Raspberry Pi using the extremely readable ROBOT Framework (see Listing 3). Furthermore, it can remotely call H-layer functions on the target and monitor the message queue. Remote calls and monitoring the measure queue is achieved by replacing the main loop with a broker. The broker waits for incoming TCP connections, reads a JSON serialized message, parses the message, executes specified function with specified arguments, serializes the return value and sends it back through the same TCP socket.

6.2.1.1 Generalization of the Testing Platform

The testing platform can be reused across projects, and it is independent of which language the production software has been written in. Only the broker has to be rewritten in the production code language.

TCP was used in this particular implementation mainly because the Raspberry Pi has a built in Ethernet port. The principles is however not limited to TCP. Through some minor modifications of the remote call library allows virtually any communication channel to be used instead. The testing platform can therefore easily be modified to use whatever communication channel the system under test can afford to spare for the automated tests.
6.2.1.2 Using the Platform In Development

The purpose of the testing platform is to enable automated isolated unit tests of the hardware dependent functions with as little effort as possible. As seen in Listing 4, test cases are easily implemented and this undertaking can therefore be considered as successful.

In addition to automated unit tests of software, the setup can help the developer identify mistakes in the hardware and wiring. Given the low complexity of constructing tests the developer can feel comfortable that the tests are free from fault.

If the wiring of the system-under-test is complex, it can sometimes be beneficial to write a simple test suite, which will verify that the wiring is correct. This benefit was directly observed by the authors during development of the validation project. In one instance a test failed even though it used the same software as other successful tests. After further investigation the source of error was identified as a wrongly connected wire in the breadboard used during the development. In another instance, the tests were returning different results each run, even though neither software nor hardware was changed in between test runs. At first, the target was believed to be damaged. The fault was, however, not at the target but at the wire connecting the ground of the two systems. Without automated tests these types of errors can be hard to identify which sometimes lead to long sessions of debugging, slowing down the progression of the development.

6.2.2 Validation Project

The goal for the validation project was to validate the implemented tools and the design rules of MPH if used in TDD development. The project was successfully completed using TDD from beginning to end and abiding the rules of MPH. Although implemented in Python, the system did not diverge much in overall design compared to a C-implementation. Translating the method into C is trivial and identified benefits and drawbacks is considered to hold for a C implementation as well.

Isolation of functions and testability is the primary intended outcomes of MPH. The validation project was developed one app at the time, one function at the time, one test at the time. Isolation was easily achieved and TDD could be applied without any problem. Automated tests were implemented for all H- and M-layer functions in complete isolation making changes to the code easy. MPH, hence, provided a good setup for agile development and TDD for the validation project.

6.2.2.1 Limitations of Generalization

Although MPH can be considered as success for this particular project it is important to recognize the limitations of the project and how it affects the generalisability. The primary limitations identified were the following: Not very complex product, no legacy code, no real time dependency.

Due to time limitations, a not so complex product was selected for validations. It is therefore, difficult to predict how well MPH will work for larger, more complex systems. Increased isolation generally leads to reduction in complexity but if the level of isolation achieved during the validation project can be reached in more complex systems remains unanswered. Although nothing was identified that would suggest that the isolation properties will not translate to more complex systems, it is important to recognize that there is limited evidence to motivate such a generalization.
The second and third issues are not touched upon during the validation project. No conclusions can be made regarding the performance of MPH if it is used in the development of a product with an already existing code-base, or how it performs with real time constraints. The validation project was a completely new product with no legacy code and had no real time requirements, hence, no conclusions can be drawn regarding MPH's relation to legacy code and how well it functions in hard real-time systems.

6.2.3 Programming Language and Hardware Choice and Transferability

Motivated primarily by time limitations, Python was used for the software and Raspberry Pi's was used as hardware for both target and testing platform. As discussed in 6.1.1, Python is not a very common language in the realm of embedded systems and carries little resembles to the more common embedded programming language C.

The choice of the target hardware, a Raspberry Pi running Raspbian, is an non-typical environment for the the software execution. Raspbian is Debian-based and, hence, have access to the almost endless number of tools and programs using the built-in package manager. Furthermore, the Raspberry Pi has 512 Mb RAM, an amount which by most embedded developers would be considered as nearly endless.

Clearly, the implementation is not a typical embedded system and it is therefore questionable how well the findings translate to a more general case. First of all, a general case is hard to define in embedded systems. That being said, the C-language, real-time dependence, sparse processing power and memory are recurrent themes in the field, none of which, being part of the discussed implementation. However, by designing the Python implementation with the limitations of C in mind, the findings can act as a starting point for further discussion about issues which might arise in a more constrained system.

6.2.4 Trigger-Event-Loop

As described in Section 6.1.2, this particular implementation uses a queue for passing messages and the main loop as the consumer of the trigger-event-loop. The main loop runs continuously meaning that average time between trigger and consumption is minimized. In addition, being efficient in terms of delay, the strategy is straight forward to implement. At first sight this implementation might seem to be like the obvious choice for every system. However, by considering a few edge cases (see below) it quickly becomes evident that it is only suitable under certain conditions. These are discussed below.

6.2.4.1 Processor Usage and Real Time Dependency

Let us consider the case where a system using the described trigger-event-loop is at risk of receiving inputs in large bursts from the environment. Possible scenarios could, for example, be that multiple external modules reach low probability states simultaneously. In these states these external modules might send a higher than normal amount of data to the system in question. The incoming signals generate large quantities of interrupt callbacks which may or may not trigger events. The main loop has lower priority than interrupt calls meaning that if events are triggered in a faster pace than they are consumed, then the number of messages in the queue will increase over time. Eventually, the queue could run out of memory and depending on the implementation the system may ignore new messages, overwrite old messages or overflow the memory. The message queue and the main loop consumer are, therefore, not a viable option if it is critical that all triggered events are managed. This, since there is a risk that the processing power is at least temporary insufficient.
Clearly, there is no way of handling all interrupts as well as consuming all messages in the above example. As long as the scheduled load exceeds the available processing power the queue will continue to grow. If the behavior is only temporary, then increasing the message queue’s memory can mitigate the issue. If not, a more powerful processor is required. In either case the delay between trigger and consumption is subject to potentially large variation. This unpredictability may make the suggested implementation unsuitable for a real time dependent system.

The variability in delay can be reduced by moving the consumption from the main loop to a periodic interrupt on a system call. By configuring the priority of the interrupt in relation to other interrupt calls, consuming messages can be prioritized in favor of inputs considered less important. In short it becomes possible to disregard certain inputs and prioritize message consumption. Furthermore, by adjusting the frequency of the event loop calls, a response time processor usage trade-off can be controlled. Non continuous consumption means an increase in delay between the trigger and the consumption. A high frequency reduces the delay but increases processor usage and vice versa.

The choice of data structure used for messaging also offers room for improvement. A queue is very easy to implement but will lead to inefficient behavior if not all triggered events are equally important to execute with minimal time. Most systems with real time dependency perform tasks with non uniform time distribution. In these cases, a heap could offer less delay for time-critical tasks. More time critical events are simply given a higher priority when triggered and hence will be consumed first. This also enables the possibility to combine both of the two above described designs. The main-loop can run continuously with low priority, consuming events in absence of higher priority system calls. The interrupt driven event-loop can then be implemented to only handle high priority events. The frequency can then be adjusted to guarantee that important time-critical tasks are executed with acceptable response time even if the system experiences a temporary high load of low priority interrupt callbacks.

6.2.4.2 Overhead in Message Serialization and Parsing

The developed implementation uses JSON to serialize messages before passing them to the message queue. JSON is widely used and open source libraries for serialization and parsing can easily be found for virtually every programming language, C is no exception. JSON encodes a data object to a string which increases the reusability of code produced according to MPH. This is because the interface to trigger and P-layer callback functions becomes independent of the type of data passed, this is since strings are always used.

Serializing and parsing data surely introduces overhead in terms of processor usage and depending on the nature of the system it might matter. Messages are serialized once and parsed once for each callback function of the event in question. Neither process is inefficient but could become an issue if the frequency of triggered events is very high or the available processing power is sparse.

In addition to processing power, JSON serialization could introduce overhead in terms of memory usage. Using strings to represent numbers is not as efficient as pure binary. Furthermore, JSON uses a key-value format, which means that keys and separators will increase memory overhead. The overhead from JSON serialization will therefore grow with the size of the serialized object.

The alternative to JSON serialized messages would be to use a general purpose struct for all passed messages. This struct will most likely be project specific which reduces the reusability of the code. Furthermore, in order for a general struct to be memory efficient, the data
passed with the messages have to be quite similar in terms of size and data types. If all messages have the exact same format, say three 8-bit integer and a ten character string, then virtually no overhead is will be introduced. Let us now consider the case that a single 8-bit integer is triggered with a much higher frequency than the previously mentioned messages. Since the struct still have to be able to hold the larger messages, the smaller ones will be sent in the larger struct leading to an overhead of 96 bit per message in this particular example.

To summarize, JSON serialization introduces memory overhead, which grows with the size of the individual message. On the other hand, a general purpose struct does not have any overhead, provided all messages have the same format. However, if messages differ in size then all messages will be at least as big as the biggest. JSON serialization might therefore be more memory efficient if messages differ in size. Furthermore, using JSON makes the interface more general and, therefore, increases reusability of code.

6.3 Combining Results Of Qualitative Methods And Experimental Validation

By comparing the outcome of the qualitative methods and the experimental validation a considerable triangulation of result can be made. However, this is complicated to the diverse nature of the methods and only a two groups of comparable results can be formed. These groups are presented in Table 4 below.
Table 4: Groups of comparable results of the used research methods

| Group | Interviews 
Results | Seminar Results | Implementation Results |
|-------|---------------|-----------------|------------------------|
| 1     | • MPH seems to be a good approach to reduce overhead and enable TDD.  
• It can be confirmed that reducing mocking would significantly reduce overhead. | • MPH could increase the test possibilities | • MPH enabled TDD without the use of mocks. |
| 2     | • MPH could be implemented in industry development projects, possible restricted by system complexity, real time requirements and existence of legacy code. | • Problems to integrate with real time dependencies.  
• Could be hard to integrate into larger systems with a high degree of complexity. | • MPH was successfully implemented on small scale industry project.  
• Real-time dependencies were identified as a possible limitation. |

In the first group all of the methods strongly indicates that MPH could be used to improve the testing process and reduce the overhead that is inherent to TDD in embedded development. The interview study suggest that mocking creates substantial overhead in a development process and the experimental validation shows that MPH indeed can be implemented in real life development.

The second group identifies real-time dependencies as a possible weakness of the MPH method. All methods suggest that the use of a trigger-event-loop will have consequences for real time applications and this aspect is, therefore, crucial for further development of the MPH method. Furthermore, complex systems and the existence of legacy code were also identified as possible limiting factors. These could also be interesting topics for further development of MPH.
7 Summary, Conclusions and Future Work

This chapter presents the concluding findings and suggests possible future work.

7.1 Summary

The application of TDD in software development is heavily dependent on the ability to keep the incremental development cycle short and fast. According to Cordeman[5] this becomes a problem in embedded software since hardware dependencies create bottlenecks in the test process, which disrupts the TDD cycle and creates what he defines as the development speed constraints. These constraints are commonly addressed with mock implementations that abstract away hardware dependencies and enable development and testing independent of the target architecture.

The thesis was based on a combination of three scientific methods, a literature study, combined qualitative methods and experimental validation. The literature study forms the foundation of the conducted research, by reviewing of related work, formulating research questions and making a state-of-the-art analysis, which enabled conceptualization of the developed MPH method. The literature study is in large extent based on the related work by Cordeman et al.[5], which identify the inherent challenges of using TDD in embedded software development. Through this, the problem of development speed was identified by the authors to be an area of particular interest for further research. Based on this related work, the problem statement could be established and the research questions defined. This, in turn, resulted in the development of the MPH method.

7.2 Conclusions

The undertaken research in this thesis is of an exploratory nature and the results presented should be regarded as a starting point for further investigation. The combined qualitative methods and the industry implementation constitute the major contributions made in this thesis, through which answers to the research questions are provided. Furthermore, they provided a considerable weight of evidences suggesting that MPH is a valid method to effectively enable TDD in embedded software development.

Firstly, the results presented indicate that implementation of mocks is problematic since it adds to the cost of development, disrupts the TDD cycle and does not add any value to the development. Therefore, the authors propose a new development method, MPH, which enables automated unit testing in embedded software without the use of mocks. The method was evaluated through an interviews study, an industry seminar and experimental evaluated validation on an industry project.

The combined qualitative methods of the interview study and industry seminar were primarily used for investigating the first research question (RQ1: Would reduced mocking make TDD more attractive for embedded software development?). The qualitative methods were also used as an approach for acquiring data that can be used to evaluate and possibly improve the developed MPH concept. The concluding findings suggest that MPH would, indeed, reduce overhead and enable TDD in embedded software development. This is based on the confirmation that the implementation of mocks in software development is a considerable cost. The study also concluded with an indication that MPH would be able to enhance the development and testing processed within most types of embedded development projects. However, the results of the interviews and the seminar strongly suggested that MPH must be
further implemented and validated on real industry project before conclusion can be made clear about possible gains and drawback of the method.

Explicitly, the combined qualitative methods answers the first group of research questions. Firstly it were found that reduced mocking would make TDD more attractive for embedded software development (RQ1). Furthermore, mocking were confirmed to constitute a significant portion of the total development (RQ1.2: Does mocking constitute a significant portion of the total development?).

The qualitative methods also approaches research question RQ1.1 (Is mocking needed for automated unit tests in embedded software?) and RQ2 (Can automated unit tests for embedded software be constructed without using mocks?). Here, it was indicated that automated unit tests for embedded software does not require the use of mocks, and by using the MPH method automated unit test could be constructed without the application of mocks (RQ2).

The industry project implementation complements the qualitative methods and gives an experimental validation of MPH. The project included testing how the implemented tools and design rules hold up in a real development project. Although the scope of the implementation was rather limited, it clearly showed that MPH could be successfully applied in embedded software development. More specifically, it provided isolation of functions and overall testability of code, which enabled the application of TDD. Consequently, automated unit test could be constructed without the need of mocks in a setup that were independent of the specific target architecture. This results added further to the qualitative findings with regard to RQ1.1 and RQ2, and it also provided answer to RQ3; automated unit tests for hardware dependent functions can be constructed without target architecture specific setups.

The above stated research claims are subjected to limitations and weaknesses of the applied research method. The first limitation is the quality of the data acquired through the interviews and the seminar. This is a consequence of conducting these studies at a time where no real implementation could be demonstrated. Respondents therefore had to base their answers on concept models of MPH which made it difficult for the them to make any definite statements about the properties of the concept. To amend this limitation and improve the quality of the qualitative data gathering, considerable work was put in optimizing the interview process and preparing the industry seminar. As a consequence the time that could be used for the industry project implementation was substantially shortened. In the end this became a limiting factor since it affected the scope of the implementation, and hence it did not involve a very complex product, legacy code or requirements on real time dependency.

In this context, it is clear that applying the combined qualitative methods (interviews and a seminar) before implementation weakened the claims made by both these research methods. Possibly, a better approach would be to start with implementation in the form of a proof-of-concept, which would then be evaluated through qualitative measures. Another suggestion would be to use a more iterative approach where implementation and qualitative valuations would be interacted in a cyclic process. This approach could strengthen the conclusions presented.

7.3 Limitations

The results of the combined qualitative method and the experimental validation established several areas and limitations of special interest. In many cases, these topics; Real Time, Strictness of the MPH design rules, Hardware and Development time, serves as basis for some of the conclusions drawn in this thesis and as context for possible further research.
These topics are presented below.

7.3.1 Real Time
Real time requirements on the embedded system under development were identified as a probable limitation of MPH, since using a trigger event loop would introduce delays into a system. This issue was confirmed by the results of the interview study and the seminar where it was stated that MPH could be problematic when used to develop real time critical applications. In the case of the project validation there were no real time dependencies, but nonetheless observations on how real time issues could arise were made. These issues and possible solutions are discussed in section 6.2.4.1. Furthermore, this is an area that can be further investigated.

7.3.2 Strictness of the MPH design rules
The results of the interview study states that the use of standardized design methods such as the MPH is not always an optimal approach. In fact, it could sometimes lead to overly complex system designs and decrease readability of code. This was also confirmed in the industry implementation where it was indicated that the strict design rules of MPH would cause code repetition and reduced readability.

7.3.3 Hardware and Development Time
The analysis of the industry implementation conclude that MPH is not completely “lean” in terms of processing power and memory usage. However, if MPH could led to increased reusability of code combined with the time save with TDD, the overall developing time could be reduced. Hence, there is a trade off between using more expensive hardware and less time spent (for same quality, counting testing as part of quality). Developers must, therefore, decide where they should put the focus in development projects. Is it a product where million of copies will be manufactured, then using low-cost hardware will be of great consequence. The opposite is true for products sold in smaller series.

7.4 Future Work
7.4.1 Scientific Method
It is clear that the reasoning undertaken in this thesis implies that MPH would benefit the use of TDD in embedded software development. However, further investigation is needed to strengthen this claim. One suitable approach would be to apply other types of scientific tools that are of less qualitative nature than ones used. For example, surveys or controlled experiments could be possible methods for providing complementary data that could add statistical weight to the claims made. One such approach used was by George[13] to compare the use of TDD and traditional development methods. In these experiments the data samples were constituted by programming pairs, which were assigned to solve small programming problems within a given time frame. The pairs were randomly divided into two groups, a TDD group who were instructed to solve the problem using TDD and a second group who were instructed to use a traditional development method (in this case; Waterfall). The experiment was combined with the opinion survey study. By comparing and making statistical analysis of the results, well grounded claims were made. A similar approach would be effective in a quantitative validation of MPH where its performance could be compared to other methods, as well as to provide possible improvements of the method.
7.4.2 Further development and validation

Another area of interest is the experimental validation of MPH. The scope of the industry implementation presented in this thesis was very limited. Areas of particular interest such as considering real time constraints, legacy code and development of complex systems were not included. Naturally investigating these topics would be suitable for further experimental investigations. Such undertakings would include further development of the tool set that is used by MPH such as the Broker generation, creation of test dependencies etc.

One research area not treated at all in this thesis is the effect on organizational processes; How methods such as MPH should be implemented as practices in a development organizations. Consequently, topics of interest here are; introduction of development methods in development teams, using of the these methods in different kinds of embedded development projects, and enforcement and use of such methods. These topics connect to the issues of the strictness of design rules and generalization of MPH.

Finally, the the identified critical trade-off between hardware and development time should be further investigated. The possible gains of reusability and time saving inherent to TDD are an interesting topic for further research. For example it can be interesting to find correlation between hardware requirements and implementability in different types of embedded system development projects.

References


[40] Alexander; Niehaves Bjoern; Niehaves Bjorn; Reimer Kai; Plattfaut Ralf; Brocke, Jan vom; Simons and Anne Cleven. Reconstructing the giant: On the importance of rigour in documenting the literature search process. pages 2–6. AIS Electronic Library, 2009.


A  Interview material

(Note that the development/testing process figures can be found in the original swedish version)

Introduction
The topic of our thesis is testing of software for embedded systems. More specifically, it deals with the problems inherent to TDD in embedded development. The core of TDD is: new behaviour/functionality is introduced by first writing a test, when the test fail, code is written till that the new test and all previous ones passes. A prerequisite for this method to be practically implementable, is that all unit tests can be automated. This enables the running of test with short intervals while the production code is being developed.

The problem arises when designing unit tests for code that is dependent on hardware. Research has identified the main strategies for approaching this problem:

Test on Host
The first alternative is to develop and test all software on a host environment where hardware is mocked. TDD can then be implemented as usual with short microcycles. This approach serves well for pure logic functions, but behaviour on the final product platform can not be guaranteed.

Test on Target
On the other side of the spectrum the development and testing is done directly on the target platform. The advantage with this approach is reduced risk for unexpected behaviour in the final product environment and the possibility for automated testing of hardware dependent functions without the use of mocks. The disadvantage is that every TDD-cycle requires uploading of code to the target platform. Since a TDD-cycle should be kept very short, the uploading time constitutes a major delay, and therefore this approach is not suitable for larger projects.

Remote Prototyping
The third method; remote prototyping, uses a test environment where functions on host could communicate with hardware functions that are run on target. This approach combines the advantages of the two other strategies mentioned above: fast development/test on host + the real behaviour of the real hardware on target. This approach is, therefore, in theory the best strategy. However, it requires an initial investment in the form of a relatively complex development/testing environment.

MCH
On method that aims to isolate hardware dependencies and enable unit testing of code is the design pattern Model-Conductor-Hardware (MCH) Karlesky et al [38]. MCH is based on the isolation of hardware dependent software (H-layer) from pure logic (M-layer), with the use of a third dividing layer (C-layer). These three layers can then be tested in isolation by mocking the integrations points of the layers. The method forces the programmer to design testable code but at the same time introduces overhead in the form of mock design.
Purpose

Our thesis inquires the possibility of using a method similar to MCH, that with some modifications will eliminate the need for mocking and therefore reduce the above mentioned overhead. Furthermore, this method aims to enable testing of hardware dependent code on target while minimizing the number of uploads to the same.

Below, a summary of the design patterns MCH and the by us developed MPH is made. For each method the structure and rules of the pattern is presented. This is followed by a description of how theses would be used in a development process.
MCH
(Based on the work of Karlesky et al [38])

MCH summary
On the picture you see the previously mentioned MCH components:

- Hardware layer: H-layer
- Logic layer: M-layer
- Middle layer: C-layer

Together they make out a so-called triad. These components interact with each other using well-defined rules. The purpose with this is to isolate hardware dependent functions from logic functions.

H-layer
The hardware layer is a thin layer that deals with ports, registers and other hardware specific features. The H-layer reports state changes in the hardware to the C-layer. In- and outgoing signals are always handled by the H-layer. Note that the H-layer can only communicate with the C-layer and has no communication with the M-layer.

M-layer
The M-layer is the business logic of the triad. This layer includes pure logic functions, states, calculations and models. The M-layer also contains the triad's interface towards other parts of the software. This means, that all communication to other parts of the software within the component must pass through the M-layer. Also note, as with the H-layer, the M-layer can only communicate with the C-layer within the triad.

C-layer
The C-layer functions as isolator between the two layers and controls the flow of data between the H and M-layers. The C-layer is stateless.
Development and test process MCH
The development and test process for MCH is initiated with the creations of component tests the whole triad, this represents some functionality that is to be created. A typical test could be how the the system should answer to a certain input. The test code is the run, and results in a fail, since at this time there is no production code.

The next step is to identify which triads that will be needed for the desired functionality. A motor application for example, could contain a positions triad, a speed control triad and a communication triad etc.
When this is done, the main control loop of the functionality is created, which, will initiate the triad by calling the different C-layers. This loop is then tested by running test with mocked C-layers.

In the next step, the C-layers are created by developing these according to TDD. This is done on a host environment. Since the C-layers are dependent of M and H, these has to be mocked.

In step 4 of the process, the M and H layers are created with TDD. M is designed and tested similar to the C-layer (on host with a mocked C-layer).
The H-layer is more problematic since it contains hardware dependencies. This is typically solved through the use of simulation or different types of ad-hoc test where the production and test code is run on target. Note, the H-layer testing also requires mocking of the C-layer.

In the last step, all the code is compiled in a production version and uploaded to target. The component test are run with the aid of a hardware fixture. Often, debugging is required. But when all test pass a complete module has been implemented.

MCH can be summarized as top-down approach, where the major part of development and testing is done on Host. The method creates a huge amount of dependencies between functions which requires comprising mocking for unit testing.
MPH Summary
Model-Pipe-Hardware (MPH) is based on the same principles as MCH but is modified to reduce the need for mocks. In MPH the software is designed very similar to the principles of MCH, business logic in M, hardware dependent software in H and stateless layer (P) as a divider. In MPH, M and H layers are not allowed to call functions in the P-layer. Instead, M and H communicate which each other through a trigger-event-loop. M and H layer contains trigger calls while the P-layer only contains callback functions. This creates a structure where M and H layer are completely independent of other functions and, therefore, can be easily unit tested in isolation. Instead, the P-layer is dependent on large amount of functions and, therefore, becomes difficult to unit test. By reducing the complexity of the P-layer to trivial levels, the need for explicitly unit testing the P-layer is reduced. And instead, the P-layer is tested implicitly when conducting component test of the whole triad.

Development and testing process MPH
The MPH process is initiated with identification of the triads that will be required for the desired functionality. Skeletons for the mainloop is also generated.

Subsequently, the development of H and M-layer begins independently of each other. The M-layer is developed and tested on Host with TDD just as in MCH. However in MPH no mocks are needed with the exception of the trigger-function. But, this function is the same for all triads and reusable across projects, and consequently, only needs to be made once.

The development of the H-layer in MPH differentiate form MCH. We will develop a tool that is independent of which architecture the software is aimed at. This tool will enable easy and fast design of tests in ROBOT Framework; a high level language based on Python. In the test code the developer an specify what signals that should be sent to the hardware module from the enviroment, measure outgoing signals from the modules pins, call functions on the target and monitor the trigger function. On the target module the software for the H-layer is run together with a modified trigger function and broker that calls functions from the H-layer. By using this approach, the H-layer software can be automatically unit tested without the use of mocks.
In the next step the P-layer code is written together with component tests.

Finally, the software is compiled in a product version and uploaded to the target. Component tests are run with the aid of the ROBOT Framework and when all tests pass the desired functionality has been implemented.

To summarize, MPH is a bottom-up approach where almost all mocking is avoided. Furthermore, tests that are usually performed manually can with MPH be automated, e.g. the verification that a LED lights up when it is supposed to.
B Interview material (Original, in Swedish)

Introduktion
Vårt ex-jobb handlar om testning av mjukvara för inbyggda system, mer specifikt handlar det om att försöka minska den problematik som TDD metoder medför. Kärnan i TDD är att nytt betande introduceras genom att först beskriva detta i ett test, köra testet, skriva kod så att det nya och alla tidigare tester lyser grönt. För att denna metodik ska vara praktiskt möjlig måste alla unit tester vara automatiserade så de kan köras i korta intervall samtidigt som man utvecklar produktionskoden.

Problemets med det här är att göra unit tester för kod som beror på hårdvaran som produktionskoden ska köras på. Tidigare forskning har identifierat tre huvudstrategier för att hantera nämnd problematik:

Test på Host
Det första alternativet är att utveckla och testa all mjukvara i en hostmiljö där hårdvaruberoende mockas bort. TDD kan då tillämpas precis som vanligt och med korta TDD-cykler. Detta fungerar väldigt bra för renodlade logik-funktioner, men det är inte möjligt att garantera samma beteende på den slutgiltiga plattformen.

Test på Target
På motsatt sida av spektrummet utförs istället all utveckling och testning direkt på target. Fördelen är då minskad risk för oväntat beteende i produktionsmiljö samt möjligheter att automatisera tester av hårdvaruberoende kod utan att mocka bort hårdvaran. nackdelen är att varje TDD-cykel kräver uppladdning av kod till target. Då en TDD-cykel önskas hållas väldigt kort utgör tiden för uppladdning en väsentlig del och blir därför för större projekt inte praktiskt möjligt.

Remote prototyping
I den tredje metoden; Remote prototyping, skapas en testmiljö där funktioner på host kan kommunicera med hårdvarufunktioner som körs på target. Detta kombinerar fördelarna hos de tidigare strategierna: snabb utveckling/test på host + det verkliga beteende av hårdvaran på target. Detta är i teorin den mest effektiva strategin men kräver en initial investering i form av en relativt avancerad utvecklingsmiljö.

MCH
En metod som ämnar att isolera hårdvaruberoende och underlätta unit test av kod är användandet av designmönstret Model-Conductor-Hardware, eller MCH Karlesky et al [38]. MCH bygger på att isolera hårdvaruberoende mjukvara, H-lagret, från ren logik, M-lagret, med ett tredje lager C-lagret. De olika lagren kan då testas oberoende av de andra genom att mocka bort integrationspunkterna under test. Metodiken tvingar programmeraren att designa testbara och bra kod men introducerar samtidigt mycket overhead i och med design och konfigurering av mockar.
**Syfte**


Nedan görs en sammanfattning av designmönstrena MCH och det nyutvecklade MPH som vi har tagit fram. För varje mönster presenteras mönstrets struktur och regler, vilket sedan följs av en beskrivning av hur det används i en utvecklings/testprocess.
MCH
(Baserad på artikel av Karlesky et al [38])

MCH sammanfattning
På bilden ser du de tre tidigare nämda MCH 3 komponenterna:
- Hårdvarulagret: H-layer
- Logiklagret: M-layer
- Mellanlagret lagret: C-layer

H-lagret

M-lagret
**C-lagret**

C-lagret fungerar som en avgränsare mellan de två andra lagrena och styr flödet av data mellan H- och M-lagret. C-lagret är helt tillståndslöst.

**Utvecklings-/testprocess MCH**

Test- och utvecklings-processen för MCH inleds med skapandet av komponenttester för hela triaden, detta svarar mot någon form av funktionalitet som ska implementeras. Ett typiskt sådant beteende behandlar hur systemet skall svara på given input. Testet körs och resulterar i en fail då det i det här stadiet inte finns någon produktionskod.

Nästa steg är att identifiera vilka triader som kommer att behövas för att uppnå den önskade funktionaliteten. En motor applikation kan till exempel innehålla en triad för positionsbestämning, en för hastighetskontroll en för kommunikations osv.

När detta är gjort skapas main-loopen som initierar triaderna genom att anropa de olika C-lagren och denna testas genom att köra test mot mockade C-lager.

I nästa skapas C lagren för triaderna genom utveckling dessa enligt TDD principen. Detta görs helt i en hostmiljö. Eftersom C lagren är beroende av M och H så måste dessa beroende mockas.

I steg 4 i processen så skapas H och M lagren, även dessa i TDD cykler. M utvecklas och testas precis som C lagret helt på host och då med ett mockat C lager. H lagret är mer problematiskt då det har kontakt med hårdvaran. Det löser man typiskt genom simulering eller olika typer av ad hoc tester där man kör produktions- och testkod på target. Även H lagret kräver att man mockar beroendet till C lagret.

I det sista steget kompileras all kod i produktionsform och laddas upp på target. Komponenttesterna körs sedan med hjälp av en hårdvarufixtur. Oftast krävs här viss debugging men nätt all tester passar har man en färdig modul.

MCH kan sammanfattas som en top-down approch där den största delen av testningen och utvecklingen sker på Host. Metoden ger upphov en stor mängd beroenden mellan funktioner vilket kräver omfattande mockning vid unit-testning.
1. Create component tests
   → FAIL

2. Identify triads and create main loop
   → Test main loop by testing calls to Mock C layers
   → Write unit test C
   → Mock H/M layer
   → TDD cycle on the C layer
   → SUCCESS

3. Create C unit tests + Production code
   → Write unit test M
   → Mock C layer
   → TDD cycle on the M layer
   → SUCCESS

4. Create M/H unit test + Production code
   → Write unit test H
   → Mock C layer
   → TDD cycle on the H layer
   → SUCCESS

5. Complete all tests and production code

   On target
   - Compile in non-test mode
   - Upload on HW
   - System test with HW fixture
   → FAIL
   → SUCCESS
MPH

![Diagram of MPH model](image)

Fig. 1: Visar förhållandet mella M, P och H.

MPH sammanfattning

Utvecklings-/testprocess MPH
Processen inleds genom identifiering de triader som kommer att behövas. Skelett för dessa samt mainloopen genereras.


I nästa steg skrivs P-lagret och komponentens tester.

Till sist kompileras mjukvaran i produktionsform och laddas upp på target. Komponenten tester körs sedan med hjälp av ROBOT framework och när alla tester passar har man en färdig implementerad modul.

Sammanfattningsvis är MPH en bottom-up metod där man frångår i princip all mockning. Vidare automatiseras tester som i många andra fall sker manuellt, t ex verifiera att en LED tänds under specifika tillstånd.
1. Identify triads

2. Create M/H unit test+production code
   - Write unit test M
   - TDD cycle on the M layer
   - SUCCESS

3. Create P production code + Component test

4. Run component test using ROBOT and HW fixture from H-layer test

Only done first time
- Set up HW fixture
- Write unit test H in ROBOT
- Build for target (only production code)
- TDD cycle on H layer
- Run test (production code on target, test on host)
- SUCCESS

On target
- Compile: non test mode
- Upload on HW
- Component test with HW fixture and ROBOT
- Debug
- FAIL
- SUCCESS
C Interview Questions

Questions

1. Respondent Background
   a. How experienced are you with testing (Rate 1-5)?
   b. How would you rate your knowledge about TDD (Rate 1-5)?
   c. What are your views on applying TDD in embedded software development?

2. Comparing methods
   a. [We] Present the MCH pattern and test process.
      i. Design rules
      ii. Process
   b. [We] Present the new MPH pattern and test process.
      i. Design rules
      ii. Process
   c. [Discussion with the respondent] Discuss how the methods differ.
   d. Could you identify any advantages with the new method compared to MCH?
      i. Give an example on a real project where you believe that the new method would have aided the development process.
      ii. Gains/cost for the project?
   e. Could you identify any disadvantages with the new method compared to MCH?
      i. Give an example on a real project where you believe that the new method could not be implemented.
      ii. Gains/cost for the project?

3. Other questions
   a. Are you familiar with the sigma elevator project?
      i. Sigma is developing a complete control system for a platform elevator. This is a development of completely new system, since there is no legacy system that the development could be based on. In the project, mechanics for the drive system, elevators shaft, elevator platform and all the electronic system is designed. Sigma’s part is to develop the complete software.
      ii. When it come to testing, unit tests are written in unity for all code that is not hardware dependent. Almost no time has been spent on stubbing HW-functions. The test are built and run on a PC environment. Theses test are automated. Tests of HW-functions are made with simulated I/O signals (Labjack). However, currently this process is not automated.
   b. Do you believe that the new method could be used on the elevator project?
   c. Do you believe that the new method could increase the reusability of code?
   d. Do you believe that the new method could increase the code quality in a project?
Frågor

1. Personen bakgrund
   a. Hur är stor är din erfarenhet inom testing (Värdera 1-5)?
   b. Hur skulle du värdera dina kunskaper/erfarenhet inom TDD (Värdera 1-5)?
   c. Vad anser du om TDD i inbyggda system?

2. Jämförelse metoder
   a. [Vi] Presenterar MCH mönstret och test processen.
      i. Utvecklingsregler
      ii. Process
   b. [Vi] Presenterar det nya MPH mönstret och test processen.
      i. Utvecklingsregler
      ii. Process
   c. [Diskutera med personen] Diskutera hur de olika processerna/mönstren skiljer sig åt.
   d. Kan du identifiera några fördelar med den nya metoden jämfört med MCH?
      i. Ge ett exempel på ett riktigt projekt där du tror den nya metoden hade hjälp.
      ii. Kostnader/Vinster i projektet?
   e. Kan du identifiera några nackdelar med den nya metoden jämfört med MCH?
      i. Ge ett exempel på ett riktigt projekt där du tror den nya metoden inte hade fungerat.
      ii. Kostnader/Vinster i projektet?

3. Övriga frågor
   a. Känner du till Sigmas hissprojekt?
      i. Sigma ska utveckla ett komplett styrsystem till en plattforms hiss. Det ska göras från grunden då det inte finns något tidigare system som kan vidareutvecklas. I projekten så utvecklas all mekanik för drivsystem, hisschakt och hissplatfrom. All elektronik och all mjukvara. Sigmas roll är att vara mjukvaruavdelningen i projektet.
      b. Tror du att den nya metoden skulle kunna användas på hissprojektet?
      c. Tror du att den nya metoden kan öka återanvändbarhet av kod?
      d. Tror du att den nya metoden kan öka kodkvalitén i projekt?
Example of H-layer in app using USART

```c
#include <setjmp.h>
#include <stdio.h>

// Official drivers for STM32F4
#include "stm32f4xx.h"
#include "stm32f4xx_usart.h"
#include "stm32f4xx_gpio.h"
#include "stm32f4xx_rcc.h"
#include "misc.h"

#include <trigger_event_loop.h> // Project global trigger-function
#include "usart_h.h" // Corresponding h-file for this c-file

// Private (layer-global) struct used as outgoing buffer
typedef struct {
    char buffer[USART_OUTPUT_BUFFER_LEN];
    uint8_t insertIndex; // Pos to insert next char in buffer
    uint8_t sendingIndex; // Pos of next char to be sent
    /* If isSending true then ISR will send char in buffer if any.
       isSending is also used as mutex to avoid inserting new
       chars as buffer is being transmitted */
    uint8_t isSending;
} USART_HStateTypeDef;

USART_HStateTypeDef USART_H_State = {"", 0, 0, 0};

/*
* Activate USART hardware with interrupts triggers when
* bytes are received and when a byte has been sent
*/
void USART_Init(void) {
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB1PeriphClockCmd(ROBOT_USART_RCC, ENABLE);
    RCC_AHB1PeriphClockCmd(ROBOT_USART_PORT_RCC, ENABLE);

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Pin = (ROBOT_USART_Tx_Pin | ROBOT_USART_Rx_Pin);
    GPIO_Init(ROBOT_USART_PORT, &GPIO_InitStructure);

    GPIO_PinAFConfig(
```
ROBOT_USART_PORT,
ROBOT_USART_Tx_Pinsource,
ROBOT_GPIO_AF_USART
);
GPIO_PinAFConfig(
  ROBOT_USART_PORT,
  ROBOT_USART_Rx_Pinsource,
  ROBOT_GPIO_AF_USART
);

USART_StructInit(&USART_InitStructure);
USART_Init(ROBOT_USART, &USART_InitStructure);

USART_ITConfig(ROBOT_USART, USART_IT_RXNE, ENABLE);
USART_ITConfig(ROBOT_USART, USART_IT_TC, ENABLE);

NVIC_InitStructure.NVIC_IRQChannel = ROBOT_USART_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

USART_Cmd(ROBOT_USART, ENABLE);
}

/*
 * Attempts to insert character into outgoing buffer. 
 * Will not insert if buffer full or if buffer is 
 * currently being transmitted.
 * 
 * Return 1 if successful, 0 if not successful.
 */
uint8_t USART_QueueChar(char c) {
  if( !USART_H_State.isSending &&
      USART_H_State.insertIndex < USART_OUTPUT_BUFFER_LEN ){

    USART_H_State.buffer[USART_H_State.insertIndex++] = c;
    USART_H_State.buffer[USART_H_State.insertIndex] = '\0';

    return 1;
  }
  return 0;
}

/*
 * Queues int value as a string (base 10) using USART_QueueChar. 
 * 
 * Returns 1 if successful, 0 if not successful.
 */
uint8_t USART_QueueInt(int d) {
  char * msg;

  return 1;
}


sprintf(msg, "%d", d);
return USART_QueueString(msg);

/*
 * Attempts to insert array of characters into outgoing buffer.
 * Return 1 if all successful, 0 if not all successful.
 */
uint8_t USART_QueueString(char * msg) {
    uint8_t i = 0;
    uint8_t allQueued = 1;

    while( msg[i] != '\0' && allQueued ){
        allQueued &= USART_QueueChar(msg[i++]);
    }

    return allQueued;
}

/*
 * Initiates transmission of buffer.
 */
void USART_SendBuffer(void){
    // If it is already sending then just let it finish
    if( USART_H_State.isSending ) return;

    // Don’t do anything if there isn’t anything to send
    if( USART_H_State.insertIndex == 0 ) return;

    // Start the process by sending the first character
    USART_H_State.isSending = 1;
    USART_SendData( // Included function from drivers
        USART2,
        USART_H_State.buffer[USART_H_State.sendingIndex++]
    );
}

/*
 * Stops transmission and resets buffer.
 */
void USART_ResetBuffer(void) {
    uint8_t i = 0;

    USART_H_State.isSending = 0;
    USART_H_State.insertIndex = 0;
    USART_H_State.sendingIndex = 0;

    while( i < USART_OUTPUT_BUFFER_LEN ){
        USART_H_State.buffer[i++] = '\0';
    }
}
/* 
* Interrupt callback function of the peripheral used (USART2).
* 
* The interrupt is triggered when a byte has been received or
* sent.
*/

void USART2_IRQHandler(void) {
    /* If the USART2 receive interrupt flag set then fetch data 
       and trigger event with the data */
    if( USART_GetITStatus(USART2, USART_IT_RXNE) ){
        trigger(USART_CHAR_RECIEVED, USART_ReceiveData(USART2));
    }

    /* If a byte has been transmitted then the next byte should be 
       sent iff isSending is true and there are bytes left in the buffer. 
       If the isSending is true but the there are no bytes lefts to send 
       then the buffer is reset. */
    if( USART_GetITStatus(USART2, USART_IT_TC) ){
        USART_ClearFlag(USART2, USART_FLAG_TC);

        if( USART_H_State.isSending ){
            if( USART_H_State.sendingIndex < USART_H_State.insertIndex ){
                USART_SendData(
                    USART2,
                    USART_H_State.buffer[USART_H_State.sendingIndex++]
                );
            } else {
                USART_ResetBuffer();
                USART_H_State.isSending = 0;
            }
        }
    }
}