Tampere University

Mikko Nieminen

SYSTEM TESTING A COMPLEX RADIO FREQUENCY EMBEDDED DEVICE

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Examiners: Emeritus Professor Hannu Koivisto
and University Instructor Mikko Salmenperä
January 2023

ABSTRACT

Mikko Nieminen: System Testing a Complex Radio Frequency Embedded Device
Master of Science Thesis
Tampere University
Degree Programme in Automation Engineering
January 2023

In this thesis, a case study of a testing setup is created for a complex radio frequency embedded device. Testing and software development is examined on a general level, followed by examining embedded radio frequency system testing and test automation.

The complex radio frequency embedded device under test in this thesis is a radio device, which receives and processes radio frequency signals. The testing setup is required to automatically verify the correct operation of the device. This requires testing the device with radio frequency inputs. The device consists of multiple internal components, which are working together to handle the signal inputs. The testing setup is developed targeting comprehensive testing of the device.

Testing in this thesis is examined on the software testing level, which is extended to cover embedded radio frequency system testing realm. A special emphasis is placed on testing radio frequency devices, and the embedded nature of the system.

For test automation, testing methods required for the setup are presented. Two test automation framework candidates are presented and examined.

The testing setup is created as a case study, using the device under test, a signal generator and a computer running the chosen testing framework. The setup is built based on set requirements with emphasis on accuracy, open nature of the software, and current and future usability. The completed case study serves as a base for future development, revealing and solving problems, which may occur in the future development of the setup.

In the final chapters, observations are noted on the challenges in creating such testing setup. Notable challenges are the limitations of commercial signal generators, interfaces between different devices, and balancing between the accuracy and the repeatability of the tests.

Next steps for future development are presented. This includes improvements such as integration to continuous integration pipeline to automate the testing further, and production testing as the next testing level for the setup.

Keywords: radio frequency, embedded system, system testing, testing, test automation, test automation framework, testing framework

TIIVISTELMÄ

Mikko Nieminen: Monimutkaisen sulautetun radiotaajuuslaitteen järjestelmätestaus
Diplomityö
Tampereen yliopisto
Automaatiotekniikan DI-tutkinto-ohjelma
Tammikuu 2023

Tässä työssä luodaan tapaustutkimuksena testausjärjestelmä monimutkaiselle sulautetulle radiotaajuuslaitteelle. Testausta ja ohjelmistokehitystä tarkastellaan yleisellä tasolla, jonka jälkeen tarkastellaan sulautettujen järjestelmien testausta ja testiautomaatiota.

Tässä työssä monimutkainen sulautettu radiotaajuusjärjestelmälaite on radiolaite, joka vastaanottaa ja prosessoi radiosignaaleja. Kehitettävän testausjärjestelmän on kyettävä automaattisesti todentamaan laitteen oikea toiminta. Tämä vaatii laitteen testausta radiotaajuussyötteillä. Laite koostuu useista sisäisistä komponenteista, jotka toimivat yhdessä signaalisyötteiden käsittelemiseksi. Testausjärjestelmän toiminnallisena tavoitteena on laitteen kattava testaus.

Testausta tarkastellaan tässä työssä ohjelmistotestauksen tasolla, joka laajennetaan kattamaan sulautettujen radiotaajuuslaitteiden testaaminen. Erityisesti painotetaan radiotaajuuslaitteiden testaamista ja sulautetun järjestelmän ominaisuuksia.

Testiautomaation osalta esitellään testausjärjestelmältä vaadittuja testausmenetelmiä. Kaksi mahdollista testiautomaatioviitekehystä esitellään ja tutkitaan.

Testauskokoonpano tuotetaan tapaustutkimuksena käyttäen testattavaa laitetta, signaaligeneraattoria sekä tietokonetta, jolla ajetaan valittua testausviitekehystä. Kokoonpano on rakennettu asetettujen vaatimusten mukaisesti, painotuksena tarkkuus, ohjelmiston avoimuus ja nykyinen sekä tuleva käytettävyys. Valmis tapaustutkimus toimii pohjana tulevalle kehitystyölle, paljastaen ja ratkaisten ongelmia, joita voi esiintyä kokoonpanon tulevassa kehitystyössä.

Viimeisissä kappaleissa esitetään huomioita haasteista, joita esiintyy työssä tehtävän testausjärjestelmän kehitystyössä. Erityisiä haasteita ovat kaupallisten signaaligeneraattoreiden rajoitteet, laitteiden väliset rajapinnat sekä tasapainottelu testaustarkkuuden ja -toistettavuuden välillä.

Jatkokehityskohteiksi esitetään testausjärjestelmään kehitettäviä seuraavia ominaisuuksia. Tällaisia ovat esimerkiksi sisällyttäminen jatkuvaan integraatioon, jotta testausta voidaan automatisoida pidemmälle, sekä tuotantotestaus seuraavana testauksen tasona testausjärjestelmälle.

Avainsanat: radiotaajuus, sulautettu järjestelmä, järjestelmätestaus, testaus, testiautomaatio, testiautomaatioviitekehys, testausviitekehys

PREFACE

I would like to thank Saab Finland for the opportunity of being able to complete my thesis for them. I am especially grateful for my thesis supervisors at the company for their time and expertise in guiding me with my thesis, helping me finish it on such a tight schedule.

Tampere, 13 January 2023

Mikko Nieminen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application programming interface
ASF	Atomic system function
CI	Continuous integration
DUT	Device under test
HTTP	Hypertext transfer protocol
JTAG	Joint Test Action Group
OSI	Open Systems Interconnection
RPA	Robotic process automation
SCPI	Standard Commands for Programmable Instruments
SSH	Secure Shell
TAP	Test access port
USB	Universal Serial Bus
WSL	Windows Subsystem for Linux

1. INTRODUCTION

This thesis is created for Saab Finland Oy, for their need of an automated testing setup for a complex radio frequency embedded device. The device forms a complex system with multiple internal components, which is operated by software running on the device. The device receives radio frequency signals as inputs, which it must process. After processing a signal, the device generates an appropriate output. A simplified view on the functionality of the device is presented below in Figure 1.

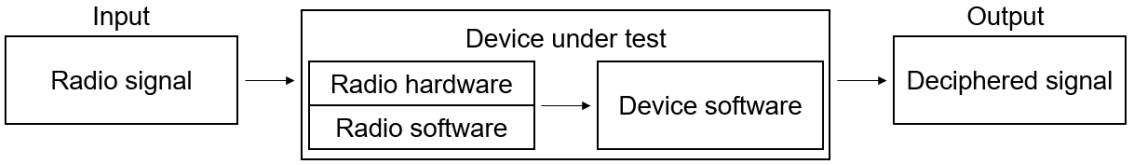


Figure 1. *Simplified view of the functionality of the device under test.*

Receiving and processing radio frequency signals increases the complexity of the testing setup. To control the tests, the setup requires a software framework that must be selected from possible candidates. The device and setup have specific requirements and there is no simple solution to fulfil these needs.

Research methods used to solve the testing problem are literature research and a case study. Literature is used to search for different testing and development methods, and to identify problems in testing systems such as the device under test (DUT) of this thesis. A case study is done by creating the testing setup, and observing the problems during the development.

Previous works on embedded system testing do not exactly match or resolve the problem at hand. Creating a sufficient testing setup consists of solving different problems, such as software choices, and the difficulties in testing embedded systems. Additional problems are created by the radio frequency property of the device under test.

1.1 Thesis goal

The goal of this thesis is to create a test automation setup for a radio frequency embedded device. Automated testing has many benefits in development, such as resource ef-

ficiency, and discovery of faults caused by new modifications in previously tested features. The setup is desired for use in continuous integration with rapid development methods.

This thesis is to answer two research questions. The first question is how to implement an automated testing setup for a device, which requires complex measuring equipment. For automated testing, the complex measuring equipment must be controllable by software and the equipment must be at disposal. For automation, the setup requires a feasible test automation framework. Solving of these problems are addressed in this thesis.

The second question is what decisions contribute to the creation of a modular test automation setup. The setup created in this thesis serves as the base for further improvements, targeted at testing the same device under test or different devices. Challenges regarding this question are solved by making suitable development choices, when solving the first question.

The setup created in this thesis will form a base for a testing setup, to be expanded upon in future iterations. The setup must solve the challenges of creating a usable testing setup, with a special emphasis on controlling the device under test, and creating required test inputs in the form of radio frequency signals.

There currently exists no comprehensive system testing setup for testing the device under test. Hardware tests are semi-automated with various scripts, with no overlying testing framework. Verifying software functionality on the device under test is done manually, by testing desired features after software revisions.

The automated testing setup is built from a starting point with some work done beforehand. There exists some library implementations for controlling the device and the equipment. The interfaces of the device under test and the test equipment are known and implemented. With this as the starting point, a suitable test automation framework must be chosen. Using this framework the device controls must be implemented. The setup is evaluated by creating simple test cases that utilize all the necessary devices.

1.2 Thesis structure

In this thesis, development and testing are examined on a general level from a software development perspective. Development is examined through different life cycle processes and testing is examined at different development levels.

Challenges in testing an embedded radio frequency system are examined. A complex device such as the device under test of this thesis has many challenges when creating

the testing setup. Controlling multiple devices is examined, as well as methods for communicating data to and from an embedded device, and the testing equipment.

Test automation is examined by presenting manual and automated testing and their characteristic features. Possible testing frameworks suitable for automated testing are presented, and their features are examined.

The case study of creating an automated testing environment is presented. A setup is created to test the complex radio frequency embedded device. The setup consists of the device under test, testing equipment and the software needed to run the tests.

The results of the case study are analysed and possible challenges are presented. Finally, next steps are proposed to further develop the testing setup, and research questions are answered.

2. SOFTWARE DEVELOPMENT AND TESTING

According to Jorgensen (2014), the main reasons for testing are to find problems and to assess quality. People make mistakes and particularly software systems are susceptible to these mistakes. (Jorgensen 2014, ch. 1)

To understand the methods and the process of testing, some definitions must be clarified. When a software is subjected to testing, the software is run with the frame of a tests case. The test case defines the inputs and the following expected outputs. When the tested software does not work as intended, there is an incident. An incident is the result of a failure, which in turn is caused by a fault. At the lowest level, a fault is caused by an error. This error may be, for example, a bug in the source code or an error in the requirements. (Jorgensen 2014, ch. 1.1)

Jorgensen (2014) states that the value of test cases is clear. He states that the test cases and the software source code are both equally valuable, with tests cases requiring their own focus. (Jorgensen 2014, ch. 1.2)

Test cases can be divided to specification-based and code-based tests. These are also known as black box, and white or clear box testing, respectively. Difference between these two is the knowledge of the system under test. In specification-based testing, only the specification of the software is known – we know what the software should do. In code-based testing, the tests are created based on the true implementation, which is the source code. (Jorgensen 2014, ch. 1.4)

Both testing methods have their strengths. Specification-based testing reveals errors when the specification is not implemented correctly, but it does not test the true functionality of the program like code-based tests do (Jorgensen 2014, ch. 1.4).

All programs can be considered as functions having inputs and outputs, where known inputs should produce corresponding outputs (Jorgensen 2014, ch. 3.2). Even if some system is actually more complex than a single function in a programming language, the system can be reduced to act like one. The device under test of this thesis receives radio frequency signals as inputs and produces output depending on the input. Even though the system consists of multiple different components and software working together, it can be examined as a function having an input, and an output that is dependent on the input.

In regard of test case creation, in this thesis the most focus is on specification-based testing. In many cases, there is no knowledge of the internal functionality of the function

under test. In some cases, there is the possibility to examine the program code. In a way, testing of the device under test could be considered both black box and clear box testing. Most testing is done against a black box, but if desired, there is sometimes the possibility to peek inside the box.

In this chapter, testing and development is viewed from the perspective of a program, software or system. It is important to note that all testing principles work with each implementation, regardless of the setup under observation.

2.1 Software development life cycle

Development models are important from the testing point of view. Testing is an integral part of the development process, being done at a specific part of the process. When testing happens is denoted by the development model. In this chapter, software development models are examined. The life cycles presented by them can be extended to other systems, where development is done over time.

Software development life cycle models are used to develop software. They demonstrate procedures for processes such as specification, development, validation and evolution of the software. General models for software development are a plan-driven waterfall model, incremental development model, and a model for integration and configuration. The waterfall and incremental development models are used when developing software from scratch, whereas integration and configuration uses already made components. (Sommerville 2016, pp. 44–47)

The waterfall software development model is a traditional model, which is has been used as the base for newer models (Jorgensen 2014, ch. 11). Agile development methods are new methods, created as a response to the requirements of fast software development (Myers et al. 2012, p. 175). Waterfall and agile development models are examined in the following chapters.

2.1.1 Waterfall software development

One plan-driven software development model is the waterfall model, where development happens in discrete phases. The phases start from the planning and end up in the development and validation of the software. Following the model, software development is planned beforehand, and executed according to the plan. The waterfall model is reliable and straightforward, and is often used in safety-critical applications for this reason. (Sommerville 2016, pp. 45–47)

Sommerville (2016) presents the waterfall model with five discrete steps. The first step is defining and analysing the requirements, where the requirements are decided in coordination with the users. Second step is designing the system and software, where architectural decisions are made. Third step is implementation and unit testing, where programming starts. Fourth step is integration and system testing, followed by the delivery of the system. The fifth and final step is operation and maintenance of the software, which is typically the longest phase in the model. In this step, the system is used and improvements are made, and discovered errors are corrected. (Sommerville 2016, pp. 47–48)

Jorgensen (2014) presents the waterfall model similarly but with more granularity. The model splits design to preliminary and detailed design, which are followed by coding. After coding there are discrete testing phases for unit, integration and system testing. (Jorgensen 2014, ch. 11.1)

Combining both view creates a linear development model. A modified view based on the models presented by Sommerville (2016) and Jorgensen (2014) is presented below in Figure 2.

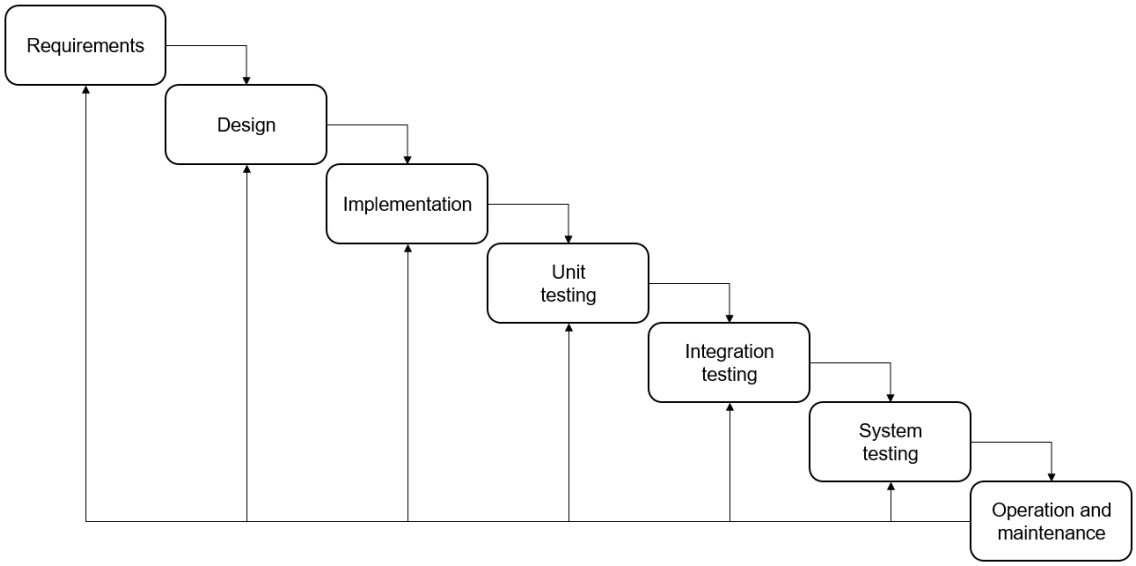


Figure 2. *Waterfall software development model (Based on Jorgensen 2014, ch. 11.1; Sommerville 2016, pp. 47–48).*

The waterfall software development model is linear and strongly organized, where everything is planned before the development starts. Waterfall development model is suitable for well-understood systems and as a frame in larger software systems. Additional suitable systems are the aforementioned safety-critical systems, which require detailed analysis of the system plans, and embedded systems where there are interfaces between hardware and software. (Sommerville 2016, pp. 45–49)

It is notable that, following the plan-driven waterfall software development method, testing happens as the final steps of the development process. Many steps have been completed before the testing of the software begins.

2.1.2 Agile software development

Agile software development is a rapid software development method, which was created to answer the needs of today's world's software development. Fast development is expected and the software requirements may change over the course of the development process. Software development based on a precise specification takes a long time and cannot adapt during the development, resulting in a software delivery, which may not fit the current needs. (Sommerville 2016, p. 73)

An agile development process develops the product in cyclic increments. Only important properties are defined, and based on them the development process begins. Each increment adds to the previous version and the end-users can propose new features and modifications. During the development process, the documentation is kept minimal and possibly created automatically, and tools such as automated testing are used. (Sommerville 2016, pp. 73–74)

Agile software development life cycle is a loop, unlike a plan-driven approach. An example of a life cycle model following agile methods is presented below in Figure 3.

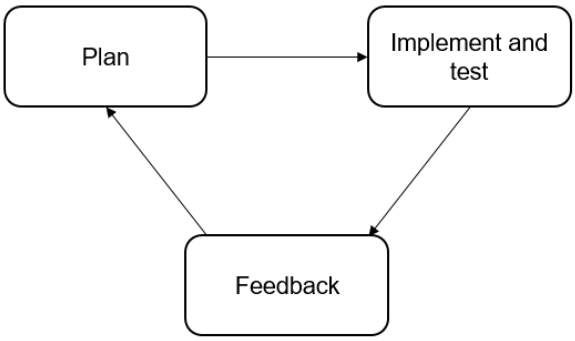


Figure 3. *Agile software development life cycle. (Based on Sommerville 2016, pp. 73–74)*

Agile methods are suitable for development process, where the end-product may have changing requirements over the development period. They have been especially successful for products of small and medium size and when developing products where the customer is involved in the development. Using agile methods with larger projects requires integrating the agile methods to a plan-driven approach, such as the waterfall model. (Sommerville 2016, pp. 47, 75–76, 91)

Weaknesses of agile methods are the lack of straightforward development process and its by-products. Because there is no clear requirement specification, customers may be reluctant to order software, which is to be developed with agile methods. The requirements may be specified and added during the development, and if this is not documented well, the final system may be difficult to maintain in the future. Other challenge of agile development is the development of large systems, because of their additional complexity. Large systems are often formed of smaller systems that are developed by different teams. This makes it difficult for the developers to have a complete view of the system. (Sommerville 2016, pp. 89–90, 93–94)

One popular agile development technique is test-first development. In this technique, the tests are written before any program code. This reduces problems when implementing the functionality's behaviour and interface afterwards, as the tests already define them. An additional advantage is removing the need to create tests after creating the program. Automated testing goes hand in hand with test-first development, making it possible to run the resulting large amount of tests each time when adding new functionality. (Sommerville 2016, pp. 81–83)

Test-first development is a form of test-driven development. In test-driven development the tests are the specification of the program, as they are created before the implementation (Jorgensen 2014, ch. 11.3.2). Creating the tests before program code makes pinpointing faults easy (Jorgensen 2014, ch. 11.3.2).

2.2 Testing levels

Typical levels of testing are unit testing, integration testing and system testing. The abstraction level increases starting from unit testing, which is the lowest testing level. At this level, program code is often tested. As the level of abstraction increases, the testing typically shifts away from the program code, to testing against the preliminary design and the requirements specification. At the final level, the integrated units are merged together and tested as a system. (Jorgensen 2014, chs. 1.6, 11.1)

In the following chapters, different testing levels are examined. These levels are unit testing, integration testing, system testing and release testing. The lowest three testing levels are focused on development, whereas the last presented level, release testing, is done before releasing a product.

2.3 Unit testing

Unit testing, as the name implies, is testing a single unit of a program. However according to Jorgensen (2014), the definition of a unit is not straightforward. He states that a unit can be as small as a single function, or some amount of code written in a given amount of hours. Jorgensen (2014) defines a unit as a part of software by one or two people who have designed, programmed and tested it. (Jorgensen 2014, Part II: Unit Testing)

Myers et al. (2012) likewise define unit testing as testing parts of a program, such as sub-programs or classes. They state that the benefit of this is fault isolation, as a failing unit test locates the fault to the unit being tested. (Myers et al. 2012, p. 85)

Jorgensen's (2014) definition above also sets limits to the size of a unit. Unit testing focuses on testing small parts of a software, the parts often being single functions of the program code. Jorgensen (2014) defines that even whole programs can be viewed as functions, as programs also have inputs and outputs (Jorgensen 2014, ch. 5). Methods to unit test functions can therefore be extended to test even larger bodies of software.

There are different ways to test these functions. Jorgensen (2014) presents input domain testing as testing method, in which a function is tested with different boundary values. He calls these tests normal, robust, worst-case and robust worst-case boundary value testing. Additional unit testing methods Jorgensen (2014) presents are equivalence class testing, decision table-based testing, path testing and data flow testing. (Jorgensen 2014, chs. 5–9)

Myers et al. (2012) present the approach to designing unit test cases as white box oriented (Myers et al. 2012, p. 86). Testing based on the unit's source code is harder on higher testing levels, and therefore it has focus on this level (Myers et al. 2012, p. 86). Myers et al. (2012) present unit testing methods such as boundary value analysis, equivalence partitioning, cause-effect graphing and logic coverage (Myers et al. 2012, p. 83). Some of these will be examined in the following subchapters, and many of them bear resemblance to methods presented by Jorgensen (2014).

2.3.1 Boundary value testing

Boundary value testing is performed by analysing boundary conditions, which appear around the minimum and maximum values of an input. Test cases analysing boundary values reveal faults better than test cases that do not account for them. Hence, boundary value analysis is a highly effective design method for test cases. (Myers et al. 2012, pp. 55–56, 61)

In normal boundary value testing the input values range from the minimum expected input values to the maximum values, which the function should handle. Between these values, the function is tested with a nominal value, and values just below or above the maximum and minimum values, respectively. Testing for boundary values is valuable, as errors often happen around the smallest and largest values of inputs. (Jorgensen 2014, ch. 5.1)

Robust boundary value testing is used to test the function for values smaller or bigger than the expected minimum and maximum values of the function. This is done to test the behaviour of the function, when it is given values outside of the allowed range. Depending on the application, the exceeded values may cause unfavourable behaviour or result in an error. The desired behaviour must be decided. (Jorgensen 2014, ch. 5.2)

Worst-case boundary value testing tests a function with multiple input values at the extreme. This is useful when the input values interact with each other. Worst-case testing also requires more effort, as test cases must to be made for all the different combinations. Even more work is robust worst-case testing, where also the values exceeding the minimum and maximum values are involved as inputs. (Jorgensen 2014, ch. 5.3)

In addition to testing a function with input values near the extremes, Jorgensen (2014) suggests a testing method called special value testing (Jorgensen 2014, ch. 5.4). Special value testing relies on the tester's knowledge to test with values that can be seen as problematic, and these tests often have good results at revealing faults (Jorgensen 2014, ch. 5.4). Myers et al. (2012) suggest a similar method, where the tester tries to come up with boundary conditions by being smart (Myers et al. 2012, p. 56).

A limitation of boundary value testing is the required input data format. As boundary value analysis relies on the numerical minimum, maximum and nominal values, it cannot be used to test unquantifiable values that cannot be placed in any numerical order. (Jorgensen 2014, ch. 5.1.2)

2.3.2 Other unit testing methods

Other unit testing methods Jorgensen (2014) mentions are equivalence class testing, decision table testing, path testing, and data flow testing. From these, path testing and data flow testing are based on the source code of the program being tested. (Jorgensen 2014, chs. 6–9)

Equivalence class testing is done by creating equivalence classes based on the input values of the function, so that each class defines specific limitations on the input values.

In decision table testing conditions and rules are listed to a decision table and their outcome is marked as a true, false or “don’t care”. Below these are listed actions, which are marked either happening or not happening, based on the conditions. Decision tables can be used to attain good coverage, but they do not scale upwards too well, as the amount of rules start to increase exponentially. (Jorgensen 2014, chs. 6.2, 6.4, 7, 7.1, 7.7)

Path testing follows the flow of the function inside the source code and different comparisons and loops are taken into account. Depending on the wanted coverage, each spot and path where the program branches are also tested. Data flow testing follows the variables inside a function, aiming to find errors in actions when the variables are defined or used. (Jorgensen 2014, chs. 8–9)

Myers et al. (2012) also present equivalence class testing as a method for discovering test cases (Myers et al. 2012, pp. 49–55). A decision table testing method is presented by using cause-effect graphing to create a decision table (Myers et al. 2012, pp. 61–80).

2.3.3 Unit testing considerations

There is a balance between specification-based and code-based testing, which needs to be found. Path-based testing is one extreme of code-based testing, where important information among the program code has been lost. On the other hand, some specification-based methods produce unnecessary tests or miss important ones. (Jorgensen 2014, ch. 8.5)

When creating the testing setup of this thesis, useful testing methods must be considered. The setup must support the testing methods that are most likely to be used. From the presented methods, boundary value testing will be useful for generating test cases. Electrical components have unique characteristics, which determine boundaries in their working conditions. Some components work only in a given temperature range, or a radio frequency receiver may handle only a specific frequency range. Many of the errors happen in the minimum and maximum ends of the values’ scales, which can be tested with boundary value testing.

2.4 Integration testing

Following the waterfall software development framework, integration testing aims to put together units that have been tested at the unit testing level (Jorgensen 2014, ch. 11.1.1). The exact point when integration testing happens is dependent on the software development framework (Jorgensen 2014, chs. 11.2–11.4). Myers et al. (2012) include integration testing to unit testing, stating that integration testing is not often a discrete testing

step (Myers et al. 2012, p. 118). Regardless, integration testing is part of integrating software modules together.

Jorgensen (2014) presents three different integration techniques, which involve testing the software in the process (Jorgensen 2014, ch. 13). These are decomposition-based, call-graph based and path-based integration methods (Jorgensen 2014, ch. 13). Myers et al. (2012) also present integration techniques, notably top-down and down-up integration methods (Myers et al. 2012, pp. 96–109). These are similar to the decomposition-based methods presented by Jorgensen (2014).

In decomposition-based integration method, units are integrated together either from top level downwards, from down to top, or multiple units at the same time (Jorgensen 2014, ch. 13.1). A stack of units can be integrated as a sandwich, or all units can be integrated at the same time with a method called “big bang” (Jorgensen 2014, ch. 13.1). Integration in a fixed direction is called incremental testing and integrating everything at the same time is called nonincremental testing (Myers et al. 2012, pp. 98–99). Incremental testing has benefits regarding fault isolation and bugs, when compared to nonincremental testing (Myers et al. 2012, pp. 99–100). Reasons for this are presented below. The integration order is based on the system’s structural relationship between units, and is usually based on the source code (Jorgensen 2014, ch. 13.1).

Integrating top-down starts with the highest level function. The lower level functions are created as stubs, which respond with a valid response. If integration is done down-up, the higher level functions are instead replaced with drivers. One by one, these stubs or drivers are replaced with the real units. If an issue arises, the problem can be located to the latest integrated unit. Big bang integration on the other hand integrates everything at the same time, making fault isolation difficult. (Jorgensen 2014, chs. 13.1.1–13.1.2, 13.1.4; Myers et al. 2012, pp. 98–100, 102–103, 106–108)

Combining the down-up and top-down methods creates a sandwich integration method, which integrates a stack of units that work together. This reduces the need for stubs and drivers, but it also makes pinpointing faults harder. (Jorgensen 2014, ch. 13.1.3)

Call-graph based integration integrates units relative to each other, based on which units call which. Call-graph based integration can also be done with stubs and drivers, replacing them with the respective units after testing. Other methods are pairwise and neighbourhood integrations. (Jorgensen 2014, ch. 13.2)

Path-based integration follows the behaviour of the software instead of the structural interfaces between units. This helps when transitioning to system testing. Path-based integration testing is laborious, as the paths between modules must be found. On the

other hand, testing with path-based integration is comprehensive. (Jorgensen 2014, chs. 13.3–13.5)

This thesis does not directly focus on integration testing. Because integration testing is one level below system testing, it is important to know how the integration has been performed. As the faults carry upwards from the lower levels, system testing will only reveal errors that must be fixed at the integration or unit level.

In the device under test of this thesis, integration can be seen as creating a seemingly working device. All the components and software elements inside the device are integrated together and tested, and testing the complete device itself will be system testing. Testing the system as a whole may still reveal erroneous interaction between the modules inside. This adds to the fact that integration testing in itself it not always clear, and neither is the step when integration testing is done.

2.5 System testing

Myers et al. (2012) present system testing as method, where the system is compared to the original objectives, to prove that the system does not meet these objectives. Rather than testing the functionality of the system against an external specification, system testing is testing against the objectives. Designing test cases based on the objectives is hard, and results in system testing being the most difficult of the different testing processes. Myers et al. (2012) state system testing process also being the most misunderstood. (Myers et al. 2012, pp. 116, 119–121)

According to Jorgensen's (2014) definition, system testing shifts from finding faults, to testing correct behaviour. Testing a system requires managing the system in smaller pieces, which are called threads. A thread is not precisely defined, but it is often a sequence of inputs, events or instructions. At system level, a thread is a sequence of pieces called ASFs, Atomic System Functions. (Jorgensen 2014, chs. 14, 14.1)

In the following subchapters, system testing is examined from the perspective of threads and atomic system functions. The final subchapter examines system testing from a non-functional viewpoint, presenting testing methods of more practical nature.

2.5.1 Atomic system functions

Atomic system functions are the smallest items in system testing, and at the same time, they are the largest items in integration testing. One ASF can be tested at a time, and together multiple ASFs create a thread. The size of a single ASF matters, so that the system tests are not too meticulous. Jorgensen (2014) present ASFs and threads with

an example of using an ATM. Entering a PIN to the ATM is a sufficient ASF. On the other hand, entering a single digit of the PIN is too specific, and not fit for an ASF. A thread would be the sequence of completing a transaction at the ATM. (Jorgensen 2014, chs. 14, 14.1)

2.5.2 Threads

As stated in the previous chapter, a thread is a sequence of ASFs. In the system under test of this thesis, a thread could consist of powering up the device, setting up required peripheral equipment, conducting a measurement and shutting down. Each step inside the thread forms a single atomic system function.

A system thread begins with a source ASF and ends with a sink ASF. In some systems, the ASFs between these two cannot be tested alone at the system level. The system needs to proceed in a set sequence, with preceding ASFs setting up the system for the next ones. (Jorgensen 2014, ch. 14.1.2)

Similarly, our system needs to start in a sequence to be testable. No measurements can be taken if the device under test is turned off or the testing equipment is not set up.

To use threads for testing, they need to be discovered from the system. Model-based threads can be found by creating a finite state machine model of the system. The state machine has transitions, and these transitions should match actual port input events. Likewise, the actions of transitions should match port output events. Jorgensen (2014) presents an example, in which these events would be entering a PIN digit to an ATM, and the screen updating accordingly. A thread can then be found by following the sequence of events. (Jorgensen 2014, ch. 14.3)

Another way to discover threads is by the use of use cases. Use cases portray what the system does, behaviourally. Use cases' advantage is their practicality, as all parties involved in the system's design and implementation can understand them. (Jorgensen 2014, ch. 14.4)

Final method Jorgensen (2014) presents to discover threads are Event-Driven Petri Nets. Petri Net analysis has many merits, such as discovering interactions between use cases and creating reverse use cases. Event-Driven Petri Nets can be derived from use cases with some manual work. Event-Driven Petri Nets can also be created by converting a finite state machine to a Petri Net, which can then be extended to an Event-Driven Petri Net. (Jorgensen 2014, ch. 14.4)

2.5.3 Requirements specification

Requirements specification goes hand in hand with system testing, as it can be used for discovering threads. Threads can be found by examining the constructs from requirements specification. These constructs are data, actions, devices, events and threads. For example, data in a system may be created by a thread. (Jorgensen 2014, chs. 14, 14.2)

Data are the information the system uses and creates, such as variables and files. Actions are viewed having inputs and outputs, and thus form the base for both specification- and code-based testing. (Jorgensen 2014, ch. 14.2)

The actions' inputs and outputs can be data, or events from port devices. Devices, or port devices as they are called, exist in every system. They are the physical interface between the system and the outside world, and they are devices such as buttons and displays. (Jorgensen 2014, ch. 14.2)

Events are the interface between what happens in the physical world via the port devices, and what happens on the logical side of the system. Whereas integration testing focuses on the logical side of events, system testing focuses on the reason the event, for example a button press, happened. The last construct, threads, are not used as often as the other constructs presented above. Finding them is left for the tester, by analysing the interactions between the other constructs. (Jorgensen 2014, ch. 14.2)

2.5.4 Test size and coverage

At system testing level, it is important to have suitably sized tests. In the use case paradigm, a single use case is how the system is used in one session. For Jorgensen's (2014) ATM example, this would be withdrawing money from start to finish. This is a long use case, but it is possible to slice it to smaller pieces. The pieces that make up the long use case are short use cases, such as inputting a banking card, and entering the PIN. It is important to note that the pre- and post-conditions of a short use case must be known, so they can be chained together to form longer use cases. (Jorgensen 2014, ch. 14.5)

Shorter use cases makes testing the system simpler. Instead of having multiple variations of the long use cases, the short use cases inside the long use cases can be tested solely. (Jorgensen 2014, ch. 14.5)

Regarding test coverage, many tests does not mean that the test coverage is excellent. For programmers, testing is not as favourable process as programming. This may result in insufficient tests with gaps in testing the program's logic. With rapid development prac-

tices, complex implementations featuring many components are hard to test as the program is developed. The tests must be reviewed and testing must continue after the initial development process. (Sommerville 2016, p. 83)

Test coverage metrics can be increased by combining different methods. Using only one method does not provide good results. Behavioural models – such as finite state machines and decision tables – based on the system never completely reflect the behaviour of the system, and creating a true model based on code is not feasible. (Jorgensen 2014, ch. 14.7)

2.5.5 Operational profiles

When using a system, most time is spent in a fraction of threads. Most threads are left unused or are used rarely. When considering reliability of a system, the locations of faults is important. Faults in threads that are rarely visited are not as critical as faults in commonly used threads. Considering this, choosing threads for testing can be done with a method called operational profiles. (Jorgensen 2014, ch. 14.8)

Operational profiles describe how often a given thread is run. This can be discovered by determining the transition probabilities from short use case to another. If all the transitions in a given path have high probabilities, the whole path has a high probability of being run, whereas a single low probability transition in a path may lower the overall probability of the whole path. The probabilities of transitions can be estimated in different ways. This can be done based on provided, observed, or experience-based data. (Jorgensen 2014, ch. 14.8)

Operational profiles can be extended to risk-based testing, where risk is determined by the probability of a use case, and the cost of a failure happening in the use case (Jorgensen 2014, ch. 14.8.2). Cost of the failure is given for each use case depending on the severity of the possible failure (Schaefer 2005, cited in Jorgensen 2014, ch. 14.8.2). Risk-based testing aims to find possibly costly failures, which might exist in rarely executed threads (Jorgensen 2014, ch. 14.8.2). Because the risk is determined partly by the use case probability, common use cases with a low cost of failure might have the highest risk factor (Jorgensen 2014, ch. 14.8.2).

2.5.6 Nonfunctional testing

Instead of testing a system based on different models, it is beneficial to conduct tests that reveal functionality in the real environment, and measure the system's real performance. This is called nonfunctional testing. One form of nonfunctional testing is called

stress testing, which can be done with methods such as compression and replication. (Jorgensen 2014, ch. 14.9.1)

Stress testing is testing a program at its highest load, which occurs only for a short time. It is suitable for programs that work continuously over a period, such as process controls, or programs with variable load. If a system is designed to handle a given amount of interactions at the same time, it is tested with this amount. (Myers et al. 2012, pp. 123–124)

Compression testing method is testing with a subset of a full system. If a full system would require one hundred devices that perform ten actions each, combining up to a total of one thousand actions, a compression test would test the same functionality with only ten devices. When testing the load capacity of a system, compression testing allows simulating a smaller load to generate comparable results. (Jorgensen 2014, ch. 14.9.1.1)

Considering the system under test of this thesis, a compression testing method could be utilized regarding a feature, where the device is required to handle a certain amount of inputs in a given time. If the specification requires a certain amount of inputs over a long period, this could be compressed to a more manageable timeframe.

Replication testing method involves testing a system in similar conditions, as the true operational environment of the system. By Jorgensen's (2014, ch. 14.9.1.2) example, a system may be required to sustain a parachute drop and still be functional. As testing a system by dropping it with a parachute would be difficult and expensive, the parachute drop can be substituted with a lower drop, with final velocity similar to the parachute-braked drop. (Jorgensen 2014, ch. 14.9.1.2)

Replication testing is a common testing method for our use case. Some inputs are difficult to create, and they are therefore substituted with laboratory devices, which simulate the real environment.

Another testing method for input is volume testing. Instead of testing the highest load, testing is done with large amount of data as an input. A program is specified with objectives to handle a certain amount of data, and volume testing tests aims to prove that the program is not capable of handling this. (Myers et al. 2012, p. 123)

Instead of stress testing, volume testing can be used to test continuous load or handing a single, large input. Size of the input may cause unexpected outcome in handling it, even if it would not be resource intensive similarly to stress testing.

Nonfunctional testing can also be approached from a mathematical viewpoint, with analyses such as reliability models and simulation. Reliability models calculate the chances

and time for a system component to fail and the repair time of the system, based on measured or estimated values. Reliability models can be applied to both physical and software systems, but both systems present reliability in a different way. Physical system components tend to fail quickly at the beginning of their lifespan, or by deteriorating as they reach the end of their lifespan. Software systems act differently, as there is no deterioration in a well-tested software. Operational profiles can be used to search for faults, but even so, testing can never find all the faults in a software. (Jorgensen 2014, ch. 14.9.2)

The device under test in this thesis is a physical device with software. This forms a layer between testing physical systems and software, and in the case of testing, brings out the worst of both worlds. As the physical side may be faulty or deteriorate, it is hard to determine the source of a fault. The fault may be caused by either incomplete software or a fault in the hardware. As many hardware faults happen at the beginning of the lifespan, these can be tested out during development. However, as the physical components age, faults originating from deterioration are hard to pinpoint.

2.6 Release testing

When a system is headed out of development, it must be tested accordingly. This testing is called release testing. The release is often for end users, but may also be for product management or other teams. Release testing no longer tries explicitly to find bugs in the system. Instead, the testing focuses on the specification and ensures the system works accordingly. (Sommerville 2016, p. 245)

Release testing should not be done by system developers. Usually the testing is done as a black box, relying only on the specification and observing the inputs and outputs. Release testing can be performed with methods such as requirements-based testing, scenario testing and performance testing. (Sommerville 2016, pp. 245–248)

Requirement-based testing is testing the system against set requirements, as the name implies. A single requirement may be a verbal use case, which must be pieced to smaller tests for thorough testing. The requirement should be written in a way that a test could be created to test the specific requirement. (Sommerville 2016, pp. 245–246)

Scenario testing involves creating a longer use scenario, which may comprise of multiple requirements. As well as strictly following the scenario path, it is also possible to make mistakes on purpose to test error handling. (Sommerville 2016, pp. 246–247)

Performance testing is done to ensure the system handles the load placed upon it. To reflect real use cases, operational profiles can be used to determine how heavily different

transactions are used in the system. Tests can then be created with most weight on the most used transactions, which will reveal the system's operational performance. To reveal defects, the system should be tested beyond its design limits. (Sommerville 2016, p. 248)

3. EMBEDDED RADIO FREQUENCY SYSTEM TESTING

Testing embedded devices is harder than plain software testing. Software testing can be done inside of a computer, and all the software components work together via software interfaces. An embedded device is a mix of software and hardware, where a new dimension, the hardware, is added.

Additional challenge is created by the requirements of testing a radio frequency device. This requires specialized hardware, such as signal generators and antennas. In this chapter, testing an embedded radio frequency device is examined at both the overall level, and in the regard of the device under test of this thesis. The device under test of this thesis is examined more closely in chapter 5.

3.1 Challenges

Several reasons make embedded system testing challenging. Many of these stem from the fact that physical components are added to the system, and that there are multiple devices working together, forming the system.

In addition to having a complex embedded system with different interfaces, the system also needs test inputs. If the embedded system interfaces with physical world, these inputs must be created there. This may require testing in different environments and using other devices for test input generation.

3.1.1 Physical components

Unlike software systems, physical systems age, as discussed in chapter 2.5.6. This poses a challenge when a functioning system is released to the customer. Initially the system may work, but over time, it starts to fail (Jorgensen 2014, ch. 14.9.2). These kinds of faults can only be found by physically harsh testing, and by simply letting the system age. Age-related testing is often not possible in a rapid development environment.

A physical device will be used in a physical environment, and it may require unconventional testing methods, as discussed in chapter 2.5.6. Unlike software, a physical device may be exposed to harsh outside conditions, such as extreme temperatures. There may be a need to sustain wear, such as exposure to different physical conditions for prolonged time. Physical components also create a challenge at the hardware and software layer, as hardware must be integrated with the software.

3.1.2 Multiple systems

Embedded system can be considered a system of systems, where multiple systems are working together. The subsystems of the complete system may be developed by different teams, adding to the challenge. An example of the challenges posed are design and architectural problems, such as those present in a software system of systems. Choices must be made when starting to integrate the systems together. One of these choices is agreeing on the interfaces between systems. (Sommerville 2016, pp. 94, 593–595)

Additionally, a system relying on multiple systems working together forms a distributed system. A distributed software system may consists of multiple computers with different hardware. The differences in the hardware and connection attributes, such as speed and performance, increase the complexity of a system. (Sommerville 2016, pp. 491–492)

Having multiple systems working together inside the system under test poses challenges when pinpointing faults. A fault in a subsystem needs to be located inside the whole system.

3.1.3 Different interfaces

When integrating a system of systems, interfaces are used to operate the systems together. Often the systems may only be accessible through their own user interfaces or some other application programming interface (API). In these situations, a software broker is needed between the different interfaces. (Sommerville 2016, pp. 595–596)

The device under test in this thesis is a system that consists of multiple systems. Different interfaces are needed for communicating with different port level devices, which were presented in chapter 2.5. In automated testing, it should be possible to perform the tests automatically. Therefore, different interfaces must be integrated to the testing environment to remove the need to manually decipher test input responses.

The possibly differing interfaces create a challenge for the tester. There may be a common interface the tester can use to interact with the device as a whole. If testing is preformed on separate components, multiple interfaces may need to be used at the same time. In this thesis, interaction with the device under test is done with a single interface, using different protocols.

3.1.4 Radio frequency equipment

Testing radio frequency devices requires specialized hardware. This includes but is not limited to signal generators, antennas and antenna measurement ranges. There is also

the added difficulty of using and sourcing this hardware, as the hardware may be complex or not readily available.

For acceptable results in radio frequency testing, a sufficient distance is required between the transmitter and the receiver. Depending on the application, this distance may be hard to implement.

For precise results, radio frequency performance may need to be tested in a special testing chamber, such as an anechoic chamber. Anechoic chamber is a type of antenna measurement range (Rodriguez 2019, p. 19). Different types of antenna measurement ranges are used to test different radio frequency properties of devices (Rodriguez 2019, p. 19).

The need for uncommon testing equipment makes scaling the testing harder. Software testing can be scaled by expending more resources on the testing computer system, which is relatively easy. Increasing the testing environments for a complex radio frequency device is harder. Nonfunctional testing methods such as compression testing discussed in chapter 2.5.6 must be used. If more complex test signals are needed, the required testing equipment is also more complex. This may reduce the available testing equipment even further.

Added difficulty of testing radio frequency equipment is the presence of noise and other signals. External signals can be mitigated with the use of a shielded enclosure (Rodriguez 2019, p. 79).

3.2 Control interfaces

As an embedded device may consist of multiple systems working together, the systems require interfaces to communicate with each other. From testing point of view, an interface is required to make inputs to the system and to verify desired output. Depending on the given device under test, the communication interface and protocol may be high or low level. A connection with the device must be made, often at as low as at the hardware level. Depending on the device under test and the possible devices used for test input generation, different interfaces may need to be used.

A software-connected interface is important when considering automated testing. Programmatically inputting test parameters and reading the results is required for effective automated tests. Manually tuning and reading different devices takes a lot of time, and should therefore be automated to the highest degree.

On the Open Systems Interconnection (OSI) model, there are seven levels of communication layers. The higher the level, the higher the abstraction. Each user on a given level interacts with peers using only the layer below their own. An exception is the lowest layer, physical, which is expected to handle the communication over physical media. (ISO/IEC 7498-1:1994(E) 1996, pp. 8–9, 28)

A high level software system works on the higher levels of the OSI model and is unaware of how the communication is performed on the lower levels. A software system most likely is not concerned how the data are physically transferred from one point to another. A low level embedded system on the other hand must implement communication down to the lowest, physical, level.

Control interfaces are additionally a challenge when designing the embedded device. Automated testing requires interfaces, and these must be implemented to the device. The same interfaces may be used for the normal operation of the device, or only for development purposes. One example of a testing-only interface is the use of a Test Access Port (TAP) on an integrated circuit, as presented in the standard by Joint Test Action Group (JTAG) (IEEE Std 1149.1-2013 2013, pp. viii, 1).

3.2.1 Device under test control

When developing distributed systems, Coulouris et al. (2011, cited in Sommerville 2016) list openness as a benefit of the systems. According to them, systems are often designed with standard Internet protocols, which helps with integration to other systems even if different vendors provide the systems. (Coulouris et al. 2011, cited in Sommerville 2016, p. 491)

Standardized protocols are used in communication technology. Communication with a higher level embedded device can be done over network connection, by interfacing with a specific protocol. Examples of such protocols are Telnet (Postel & Reynolds 1983), Secure Shell (SSH) (Ylönen & Lonvick 2006), Hypertext Transfer Protocol (HTTP) (Fielding et al. 2022), MQTT (Banks et al. 2019) and WebSocket (Fette & Melnikov 2011).

There are also common interfaces for low level communication. A common method of communication is wired communication over serial interface. One popular interface is RS-232, which was once especially popular in older computers before being replaced by Universal Serial Bus (USB). Nowadays RS-232 is still used when there is a need to communicate with embedded systems. Microcontrollers specifically tend to communicate with serial interfaces such as I^2C and SPI. Even though serial ports have been replaced in personal computing with interfaces such as USB, they are still prevalent in embedded

systems. Serial communication has many advantages for use in embedded systems, such as simplicity, affordability and availability. (Axelson 2007, pp. xiii, 1–3, 44)

3.2.2 Laboratory device control

Laboratory devices in the scope of this thesis are signal generators, which are needed to generate test input signals. Other laboratory devices may also be used with the setup, but in this thesis, only signal generators are interfaced with.

A common method for interfacing with laboratory devices is the Standard Commands for Programmable Instruments (SCPI). As the name states, SCPI standardizes communication among test instruments with agreed messaging format. The standardized messages are same for different manufacturers and they can be communicated over different interfaces. Using SCPI-supported instruments makes it easier to communicate with different instruments, using the same massaging format and content. (SCPI Consortium 1999, Volume 1: Syntax and Style, p. ii, chs. 1.3, 1.5)

An example of an SCPI command is setting a signal generator to a specific frequency. The command for setting a device to output a continuous wave at the frequency of 2 GHz is `FREQuency:CW 2000000` (SCPI Consortium 1999, Volume 1: Syntax and Style, ch. 5.1).

Devices not conforming to SCPI must be controlled individually. The devices may have their own interface with specified messaging format. In some cases, custom testing equipment may be used. In these situations, it is up to the design of the custom equipment to create an interface for the device, which is usable from testing point of view. Challenges of custom testing equipment are discussed in chapter 6.1.2.

3.3 Test input and output

Testing embedded systems has an additional challenge in creating test inputs and deciphering the output. In software unit testing, a block of code can be tested with a software-defined input. The output is likewise read in the same software. Creating automated tests for an embedded system aims for similar functionality. The inputs should be software-definable, so that when creating tests, the physical side of the system would not be a concern. When the test case is executed, the test input is created in the physical world.

An embedded system, such as the device under test of this thesis, requires inputs from outside of the device. The input is a change in the physical world, to which the device must react. This change must be physically generated, and it should be as close as possible to the real input the device is supposed to react to. An example of such input is

a radio frequency signal generated with a signal generator, as is the case with the device under test of this thesis.

Challenge of reading the output for test automation purposes depends on the versatility of the embedded device. A high level system may react to a physical stimulus in a similar manner as a software system would to a software stimulus, bringing the result up to the software level. Different interfaces down to the hardware level hide the functionality, and only a digital result is visible to device's software. Similar to a software unit test, the test output is generated on the software side, and can be read and deciphered equally simply.

A low level embedded system may not react to the stimulus in an easily decipherable way. A really simple system may only output an analogue signal changing level. Such systems are hard to automate for testing, as they need special devices to bring these outputs to software level. Devices, which must be interfaced with, can have a specific communication interface as presented in chapter 3.2.1. Devices such as USB to serial converters can be used to interface with these devices (Axelson 2007, p. 5).

Output of any test stimulus must be validated to confirm correct behaviour of the device under test. Test inputs are created with known values and correct output from the device demonstrates correct behaviour.

As discussed in chapters 3.1.3 and 3.2.1, it may be challenging to get output from the device under test. Automated testing requires an interface to the device that can be used to decipher the result in software. Manually reading output from the device's own display after each test is extremely slow. Hence, output validation should be automated as well as possible.

4. TEST AUTOMATION

Test execution can be automated to a degree, depending on the system that is tested. This automation can be done with the help of testing frameworks. Automation of the testing framework execution can be done with the use of continuous integration methodology, CI for short.

Two testing methods benefitting from automated testing are data driven testing and regression testing. Both are laborious testing methods, and automating them makes testing with them more efficient.

Testing can be both manual and automated. Manual testing relies on the tester, whereas automated tests are executed automatically by another program. Automated tests are faster, but on the other hand more limited than manual tests. It is not possible to test everything automatically, for example to determine if a there are side effects which have not been accounted for in the automated tests. (Sommerville 2016, p. 231)

4.1 Manual testing

Manual testing involves the tester entering input data and comparing the output to what is expected. Manual testing is advantageous if performing the test may produce unexpected results or the output is hard to decipher automatically, such as output in a form of graphical user interface. (Sommerville 2016, p. 231)

Manual testing is required in acceptance testing, which is the final testing phase before handing over a developed system. Part of the testing process is testing the system as an end-user. Regardless if the development follows a traditional or an agile path, thorough acceptance testing requires using the system to complete everyday tasks in the customer's environment. Replicating the true environment involving all the different variables and users is difficult. (Sommerville 2016, pp. 249–251)

4.2 Automated testing

Automated testing is running the test cases automatically. Automated testing allows saving and re-running tests as needed (Sommerville 2016, p. 83). The upsides are clear – test are run on their own and test results are generated with no work needed. As observed in chapter 2.1.2, automated testing is relevant in agile software development methods.

Automated test execution is necessary especially when development is done by creating the tests first before any code is written (Jorgensen 2014, ch. 11.3.2; Sommerville 2016, p. 82). An example of these kinds of methods is Test-driven development (Jorgensen 2014, ch. 11.3.2). Test-first development being another example (Sommerville 2016, p. 82). A suitable framework helps with this task (Sommerville 2016, p. 82). Frameworks for testing are examined more closely in the chapter 4.3.

Regression testing is a testing method helped notably by automated testing. Manually running again old test cases requires a lot of effort and time, which promotes choosing only a subset of tests to run. This may cause important tests to be skipped. Running the tests automatically requires less time and resources. (Sommerville 2016, p. 244)

4.2.1 Data driven testing

Ideally, a system should work as expected with any different inputs and produce correct output. In reality, the system may not work as intended with a given input. Data driven testing is used to test a system with different inputs. Data driven testing is useful in testing physical components, given that they have specific characteristics. They may only work as specified with a range of input values, and their function must be verified.

In data driven testing, test data are created using the specification, with focus only on results with the given data, and not on the internal implementation (Myers et al. 2012, pp. 8–9). This method is similar to black box testing presented in chapter 2. Black box testing can also be called data driven testing, coincidently (Myers et al. 2012, p. 8).

Data driven testing is used to find when the behaviour is not in line with the specification. Inversely, if no faults are found, we know that the system works as specified. (Myers et al. 2012, p. 9).

So-called exhaustive input testing can be done to ensure that there are no errors. Testing can be done with all possible valid inputs, resulting in an enormous amount of test cases. Even after creating all these test cases, not all errors are found. To find all errors, also invalid inputs must be tested. In reality, testing with this level of granularity is not possible, and some compromises must be made based on observations and assumptions of behaviour. Test cases must be chosen in a manner that produces the most errors, and possible duplicate test cases testing same behaviour must be cut. (Myers et al. 2012, pp. 9–10)

4.2.2 Regression testing

When one part of a program is modified, regression tests are run to ensure changes have not regressed inside it. When compared to original code, fixes and modifications in the same code are more prone to errors. Fixing one error in a program may correct just that, but unexpected side effects may cause errors in other parts of the program. (Myers et al. 2012, pp. 134, 171)

In practice, regression testing is saving test cases and running them again after making changes. Manual testing can be running the program manually and figuring out the test cases each time the program needs to be tested, which takes a lot of time. Work needs to be done to create the test cases again each time, and the time is often saved by recreating and running only a small subset of the previous tests. Saving and re-running the test cases reveals possible faults in areas of the program that have already been tested before, which would not be tested otherwise. (Myers et al. 2012, p. 16)

4.3 Testing framework

In software development, an application framework is a collection of artifacts such as classes and components, which can be used in similar software (Schmidt et al. 2004, cited in Sommerville 2016, p. 443). For programming languages, the frameworks are language-specific (Sommerville 2016, p. 444). In addition to these application frameworks, there are also specific testing frameworks.

Testing frameworks provide the same functionality as application frameworks, but for testing. Most programming languages have testing frameworks, which are specific for that language (Jorgensen 2014, ch. 19.2). These testing frameworks integrate with the programming language and are used with similar or same programming syntax. When using a testing framework, the tester specifies inputs and expected outputs for the software, which the framework runs and validates (Jorgensen 2014, ch. 19.2).

In the following chapters two testing frameworks are explored. These frameworks were chosen as predetermined candidates for the automated testing setup. The following chapters examine their properties, which would make them suitable for use as the test automation framework for the setup of this thesis.

4.3.1 Robot Framework

Robot Framework is an open source test automation framework based on the Python programming language. It describes itself as suitable for acceptance testing, behaviour driven development and robotic process automation. (Robot Framework 2022a, ch. 1.1)

Tests in Robot Framework are written in Robot Framework's own syntax. Within the framework it is possible to use functions, which Robot Framework calls keywords. Many keywords are provided by the framework's libraries, and it is possible to create own keywords. Robot Framework syntax supports variables in the form of scalars, lists and dictionaries, and execution flow can be controlled with if-statements, loops and exception handling. (Robot Framework 2022a, chs. 1.1.1, 2.6.1, 2.7, 2.9)

In Robot Framework, tests are contained in test suites. The test suite is a text file written in Robot Framework's syntax, or a directory containing these files. A test suite file contains the tests cases. As the test cases execute, keywords are called which perform the required functionality of the test case. Both the test cases and test suites can have setups and teardowns, which can be supplied as a keyword. In the suite's case, the setup keyword is run before any test cases in the suite are executed. Likewise, the teardown is run after all the tests are executed. For test case level setup and teardown, the functionality is identical but the execution happens before and after the test case. (Robot Framework 2022a, chs. 1.1.1, 2.2.1, 2.2.6, 2.4.1)

A Robot Framework test suite file is a .robot-file. The file contains the test cases, and in this case, the supplementary keywords. The file is presented below in Program 1.

```
*** Settings ***
Documentation   Documentation of the suite is written here.
Suite Setup     Setup Keyword
Suite Teardown  Teardown Keyword


*** Variables ***
${variable_1}   = ${None}
${variable_2}   = ${None}


*** Test Cases ***
Test Variables 1 and 2 are inequal
    Should Be Equal     ${variable_1}   ${variable_2}


Test Variables 1 and 2 are equal
    Should Not Be Equal     ${variable_1}   ${variable_2}


*** Keywords ***
Setup Keyword
    Set Suite Variable  ${variable_1}   1
    Set Suite Variable  ${variable_2}   2


Teardown Keyword
    Log     Teardown is performed here.
```

Program 1. *Robot Framework test suite, contents of a .robot file.*

Running Robot Framework tests creates output to the console, showing a brief overview of how the tests succeeded. Output is presented below in Figure 4.

```
$ robot --consolewidth 70 test_suite_file.robot
==============================================================================
Test Suite File :: Documentation of the suite is written here.
==============================================================================
Test Variables 1 and 2 are inequal                                    | FAIL |
1 != 2
------------------------------------------------------------------------------
Test Variables 1 and 2 are equal                                      | PASS |
------------------------------------------------------------------------------
Test Suite File :: Documentation of the suite is written h...         | FAIL |
2 tests, 1 passed, 1 failed
==============================================================================
Output:   /tests/output.xml
Log:      /tests/log.html
Report:   /tests/report.html
```

Figure 4. *Output of Robot Framework. Output width narrowed to fit.*

Robot Framework's functionality can be extended further by creating libraries (Robot Framework 2022a, ch. 4.1). The libraries are created in Python (Robot Framework 2022a, ch. 4.1.1). Python itself supports creating libraries in C and C++ (Python 2022). Other programming languages can be used by using Python as a wrapper for them (Robot Framework 2022a, ch. 4.1.1).

Executing tests with Robot Framework produces multiple output files containing the test execution log and the results. By default, three files are created: HTML-files of the test report and the test execution log, and an XML-file containing the results in a machine-readable format. (Robot Framework 2022a, ch. 3.6)

The report and log files can be inspected in a web browser. The report file presents a brief summary of all tests. The report file created by running Program 1 is presented below in Figure 5.

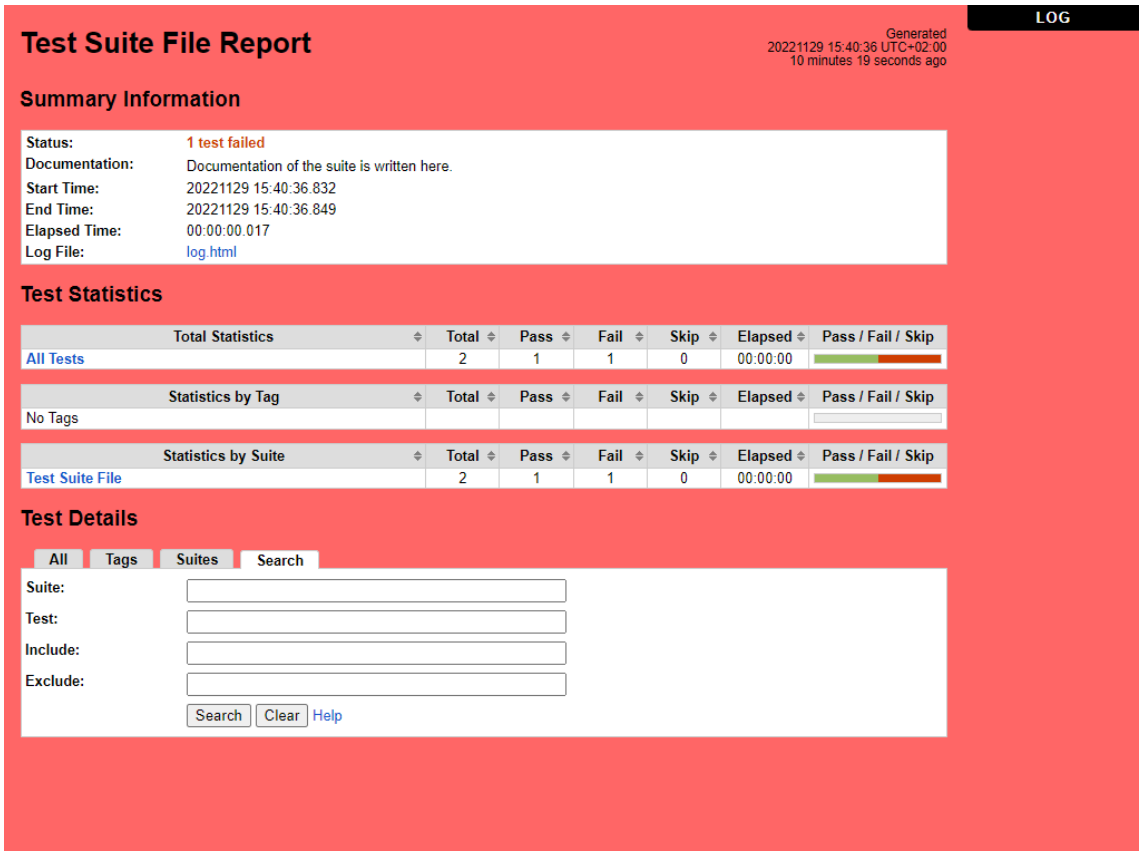
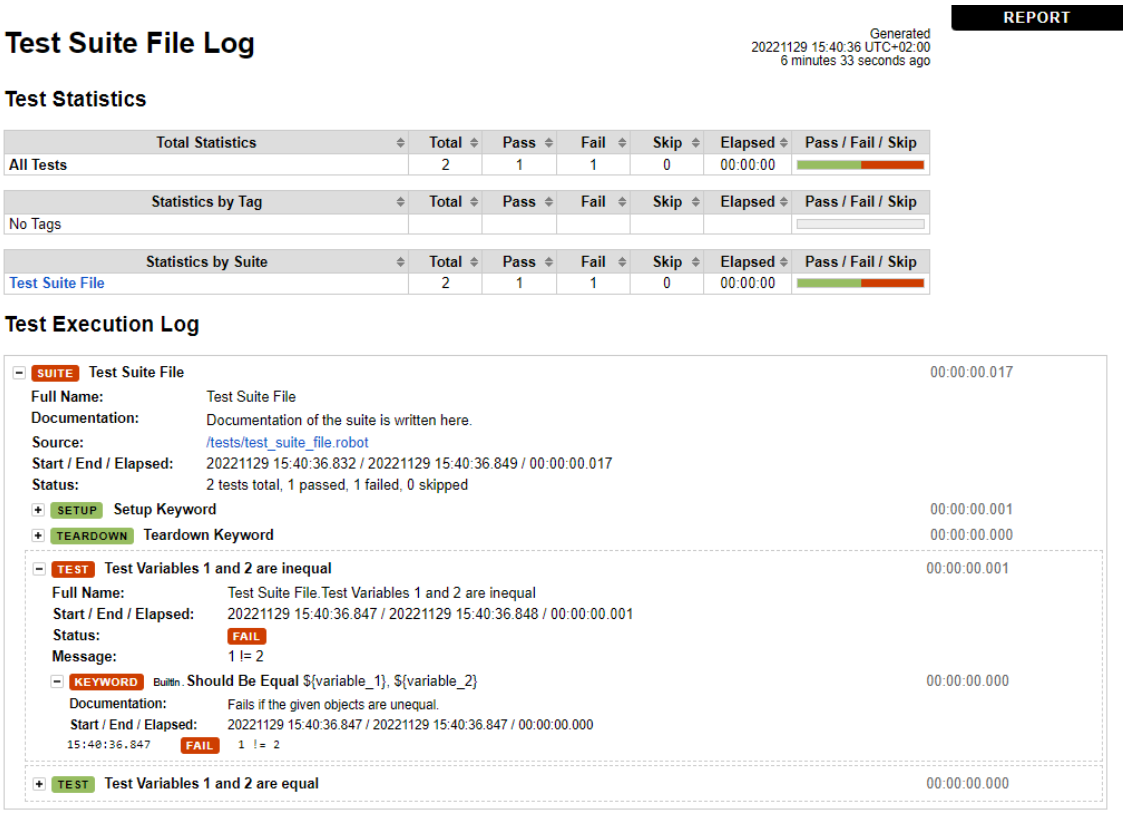


Figure 5. *Robot Framework test report.*

The log file provides a more detailed summary of the test execution. The log file is presented below in Figure 6.



Test Suite File Log

REPORT

Generated
20221129 15:40:36 UTC+02:00
6 minutes 33 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	2	1	1	0	00:00:00	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Test Suite File	2	1	1	0	00:00:00	

Test Execution Log

SUITE Test Suite File 00:00:00.017
Full Name: Test Suite File
Documentation: Documentation of the suite is written here.
Source: /tests/test_suite_file.robot
Start / End / Elapsed: 20221129 15:40:36.832 / 20221129 15:40:36.849 / 00:00:00.017
Status: 2 tests total, 1 passed, 1 failed, 0 skipped
SETUP Setup Keyword 00:00:00.001
TEARDOWN Teardown Keyword 00:00:00.000
TEST Test Variables 1 and 2 are inequal 00:00:00.001
Full Name: Test Suite File.Test Variables 1 and 2 are inequal
Start / End / Elapsed: 20221129 15:40:36.847 / 20221129 15:40:36.848 / 00:00:00.001
Status: FAIL
Message: 1 != 2
KEYWORD BuiltIn. Should Be Equal ${variable_1}, ${variable_2} 00:00:00.000
Documentation: Fails if the given objects are unequal.
Start / End / Elapsed: 20221129 15:40:36.847 / 20221129 15:40:36.847 / 00:00:00.000
15:40:36.847 FAIL 1 != 2
TEST Test Variables 1 and 2 are equal 00:00:00.000

Figure 6. *Robot Framework test log.*

The report file contains a simplified overview of the tests run, showing passed, failed and skipped tests. The log file provides the results in a more meticulous format and it is possible to inspect test execution down to the keyword level. (Robot Framework 2022a, ch. 3.6)

Examining the log file, it can be noted that the different keywords act as atomic system functions as defined in chapter 2.5.1. The path created by the keywords form a thread. As tests in Robot Framework are written by chaining keywords, developing the test cases follows the paradigm of discovering threads in a system.

Data driven testing in Robot Framework is achieved by using templates. A keyword is specified as a template for a test case. The test case is then supplied with different test data, which is all passed though the template keyword. The test template keyword is usually a custom keyword, which contains the rest of the test functionality. (Robot Framework 2022a, chs. 2.2.7–2.2.8)

Robot Framework is a versatile test automation tool with both built-in tools and the possibility to extend functionality with Python. Using Robot Framework requires learning a new syntax language, however the most complex logic can be written in Python. Out of the box, reports and logs generated by Robot Framework are clear and comprehensive.

Robot Framework is clearly aimed at testing large entities. No tools are provided for granular testing of small units.

In embedded system testing, an important feature is the ability to communicate with external devices. Robot Framework provides some libraries for this, such as an SSH library (Robot Framework 2022b). Additional connectivity can be achieved with the help of Python, with the help of Python libraries such as the serial communication library PySerial (Liechti 2022).

Robot Framework has been used in multiple previous works. Juurinen (2020) presents using Robot Framework to perform hardware tests on Valmet DNA, a process automation system (Juurinen 2020, pp. 17, 28–29). Turunen (2018) uses Robot Framework for testing an unmanned aerial vehicle (Turunen 2018, pp. 1, 24). Nopanen (2021) uses Robot Framework to test a monitoring application for Valmet DNA (Nopanen 2021, pp. 28–29, 32, 38).

Robot Framework has a consortium called Robot Framework Foundation, which companies can join to help further the development of Robot Framework (Robot Framework 2022c). 49 members are listed, many of them explicitly stating that they are using Robot Framework (Robot Framework 2022c). Robot Framework is a popular testing framework in technical theses, and in internal use of companies. Scientific articles on the use of Robot Framework are not as readily available.

4.3.2 Pytest

Pytest is a testing framework written in Python, for testing Python programs. Tests written with Pytest are also written in Python. Pytest can be used to write simple tests with assertion statements. In addition to this, Pytest advertises that it can scale, and testing applications and libraries is possible. (Krekel et al. 2022a, p. 285; Krekel et al. 2022b)

Pytest tests are written in Python files, where the file name either start with the string `test_` or ends in the string `_test`. Inside the files, each function prefixed with `test` is run. Each class prefixed with `Test` is considered a tests class, and functions inside the class prefixed with `test` are run. (Krekel et al. 2022a, pp. 3–6, 289)

An example of a Pytest test file is presented below in Program 2. The file contains a test class with test cases inside.

```
import pytest

class Test:

    @pytest.fixture(scope="class")
    def variable_1(self):
        return 1

    @pytest.fixture(scope="class")
    def variable_2(self):
        return 2

    def test_variables_1_and_2_are_inequal(self, variable_1, variable_2):
        assert variable_1 != variable_2

    def test_variables_1_and_2_are_equal(self, variable_1, variable_2):
        assert variable_1 == variable_2
```

Program 2. *Contents of Pytest test file test_ex.py.*

The test are executed from command line, and the test results are displayed as written output. Each test file name is displayed, followed by a character denoting whether the test passed, failed, was skipped or produced an error. (Krekel et al. 2022a, pp. 3–4; Krekel et al. 2022c)

Output of Pytest can be configured for increased verbosity, so that faults can be pin-pointed more easily. By default, if an assertion fails, only the failed assertion is shown together with the differing values. Increased verbosity allows inspecting the variables with their full values. If a failure is caused by an exception, Pytest will print some trace-back of the Python stack trace, up to the full length if desired. It is possible to create machine-readable test logs to be parsed by auxiliary services. (Krekel et al. 2022c)

Relevant output of Program 2 test execution is presented below in Figure 7.

```
$ pytest -v --no-header
========================== test session starts ==========================
collected 2 items

test_ex.py::Test::test_variables_1_and_2_are_inequal PASSED         [ 50%]
test_ex.py::Test::test_variables_1_and_2_are_equal FAILED           [100%]

================================ FAILURES ================================
___________________ Test.test_variables_1_and_2_are_equal ___________________

self = <test_ex.Test object at 0x1234567890abcdef>, variable_1 = 1
variable_2 = 2

    def test_variables_1_and_2_are_equal(self, variable_1, variable_2):
>       assert variable_1 == variable_2
E       assert 1 == 2

test_ex.py:17: AssertionError
======================== short test summary info ========================
FAILED test_ex.py::Test::test_variables_1_and_2_are_equal - assert 1 == 2
====================== 1 failed, 1 passed in 0.01s ======================
```

Figure 7. *Example of relevant Pytest output with increased verbosity.*

Pytest supports the use of fixtures. Fixtures are used to create base data to be used with test cases, so that each test is run using the same values. To save resources, fixtures can be scoped to different levels of test hierarchy. Lowest level is function-scope, where each test function requesting the fixture gets new data. At the highest scope, session, the fixture persists until all tests are run. This is beneficial if, for example, the base data requires creating a network connection. By sharing the fixture within the scope, each test does not need to open a new connection each time. (Krekel et al. 2022a, pp. 23–24, 286)

Data driven testing is possible by parametrizing the test functions or fixtures. Parametrizing a fixture causes all the tests depending on the fixture to be run multiple times, once for each parameter. Likewise, parametrizing a test case causes the test to be run with each parameter. (Krekel et al. 2022a, pp. 36, 48–49)

Setup and teardown methods in Pytest are possible with the fixture system, or with explicit setup and teardown functions. Using the fixture method is recommended. With fixtures, the setup resides inside the fixture. After the setup completes, the desired object is yielded to the test depending on the fixture. As the fixture reaches the end of its scope, rest of the fixture is executed as the teardown. Another method is declaring an explicit finalizer function, which is executed similarly in the end of the scope. (Krekel et al. 2022a, pp. 25–29, 108–110)

Pytest has been used for test automation in previous works. Karhula (2022) uses Pytest to verify software functionality of a vehicle measurement portal (Karhula 2022, pp. 1, 39). Lammi-Mihaljov (2020) uses Pytest to aid in the development work of a web server,

where Pytest is used for unit testing (Lammi-Mihaljov 2020, pp. 1, 29). As with Robot Framework, scientific articles regarding Pytest are hard to find.

Pytest is clearly aimed for the unit testing and integration testing level. As the underlying programming language is Python, in theory everything that is possible in Python is possible in Pytest. Out of the box, the reporting is brief and becomes hard to examine as the test cases increase. Despite this, Pytest can also be used for higher levels of testing, as proven by Karhula (2022, p. 34).

4.4 Continuous integration

Continuous integration is part of system building process. System building is combining the information, such as source code, libraries and data to a working system. The process involves building and testing the software on the developer's machine or at a remote machine. Continuous integration is a method where the builds and tests are done frequently with small changes in the source code. (Sommerville 2016, pp. 740–743)

In the continuous integration process, automated tests should be run on the developer's machine before committing changes to the version control system. Afterwards, the system is built on an external build server, which also runs the automated tests. (Sommerville 2016, pp. 742–743)

Problems arise when the target platform differs from the one used for development. The target platform may be, for example, an embedded system. In this situation, it is not possible to test the system on the developer's machine or on the external build server. (Sommerville 2016, pp. 742–743)

Sommerville (2016) states that it is often impossible to use continuous integration, when the execution platform differs from the development platform (Sommerville 2016, p. 743). In these situations a daily build system should be utilized. In this system, changes are committed daily and tested by a dedicated testing team. The testing team reports found faults, and the developers fix them in following versions. (Sommerville 2016, pp. 743–744)

It is clear that a daily build system is slower than a continuous integration system, where builds are tested automatically. Even though continuous integration is difficult to perform on embedded systems, there likely is benefit in pursuing such setup.

5. CASE STUDY OF AUTOMATED TESTING SETUP

The case study is creating a testing setup for the development and testing needs of a complex radio frequency embedded system device. The device is used to receive and process radio frequency signals. Thorough testing of the device requires using external testing equipment, such as signal generators. In the scope of the setup created in this study, testing is done to verify correct operation of software on the device, and of all the internal components that make up the complete system.

In the scope of this thesis, a pilot testing setup is created. In future, it should be possible to integrate the setup to continuous integration practices, which must be taken into account in the development of the setup.

The testing setup of this thesis is developed using agile development methods presented in chapter 2.1.2. End-user considerations are taken into account during iterations to ensure correct features have been implemented. The development starts with small proofs of concept, advancing towards a more complex setup. Parts of the testing setup are developed individually at first, following more closely a traditional waterfall software development method. Afterwards, these have been tested and integrated together to form a complete system.

5.1 Requirements for testing

The main requirement for the tests is ensuring correct operation of the device under test. Each feature of the device must be tested, and it should be possible to run the tests automatically and repeatedly. Additional requirements exist regarding the future of the testing setup. Requirements are presented in the following chapters.

5.1.1 Accuracy and repeatability

For developing the device under test with agile development methods, testing the device must be automated. Testing a complex embedded system is not straightforward, as has been presented in the previous chapters. As the inputs are created with an external device, testing the test system must control multiple devices at the same time to create inputs and verify outputs. For automated testing, everything must be integrated to the automated testing setup.

Accurate and repeatable tests are a key requirement of the testing solution. This requires an automated solution. Automated running of the tests removes the human factor for errors, and makes running the tests easier. Tedious and data driven tests can be run automatically faster than they could be run manually, as the tests may require changing a single input variable slightly, and ensuring the output stays correct.

For example, the device under test may be tested with a given frequency, generated with a signal generator. A range of frequencies is tested by changing the frequency on the signal generator and reading the output from the device. Modifying the frequency manually is possible, but becomes tedious and error-prone as the amount of different test frequencies increases.

As discussed in chapter 4.2, regression testing benefits from automated testing. As the same tests can be run each time, errors from changes are revealed in all code that are covered by the tests. Automated and identical tests also ensure that the tests are run accurately, as there is no human factor involved. They are not reliant on the tester manually inputting test values.

Automation is achieved by selecting an automated testing framework and using it to construct the test cases and operate the device under test and test instruments. Repeatability is the result of writing the tests as machine-interpretable code, which ensures that the tests are repeated identically each time.

5.1.2 Software choices

Two candidates were identified for a test automation framework, Pytest and Robot Framework. These were presented in chapter 4.3.

When choosing the test automation software, weight is placed on the openness of the choice. Both Pytest and Robot Framework fulfil this requirement. An open source solution is more flexible to use and maintain, as there are no licensing fees and the software is freely available. Open source software is also modifiable. In the case development of the software halts, it can still be used and even developed further in-house.

The testing framework must support the testing methods used for testing the device under test. Two notable methods are data driven testing and regression testing, which were presented in chapter 4.2. The framework must be suitable for controlling the device under test and the external test equipment, such as signal generators.

Additional requirement are the ease of use and the clarity of test reports. This allows developers of different backgrounds to run the tests and understand the output.

5.1.3 Future usability

The testing setup will be developed further, and it should have support for it starting from the design and testing phase. The setup must promote architectural choices that allow reusing of modules. The setup must be scalable to different testing levels, such as release testing. A benefit is, if the setup can be used on different platforms. Being possible to use the testing setup for other devices than the device under test of this thesis, is also favourable.

5.2 Testing setup

Testing in the scope of this thesis done using the device under test, an external signal generator and a testing computer. Communication with both devices is handled through the testing computer over a network connection. Image of the setup is presented below in Figure 8.

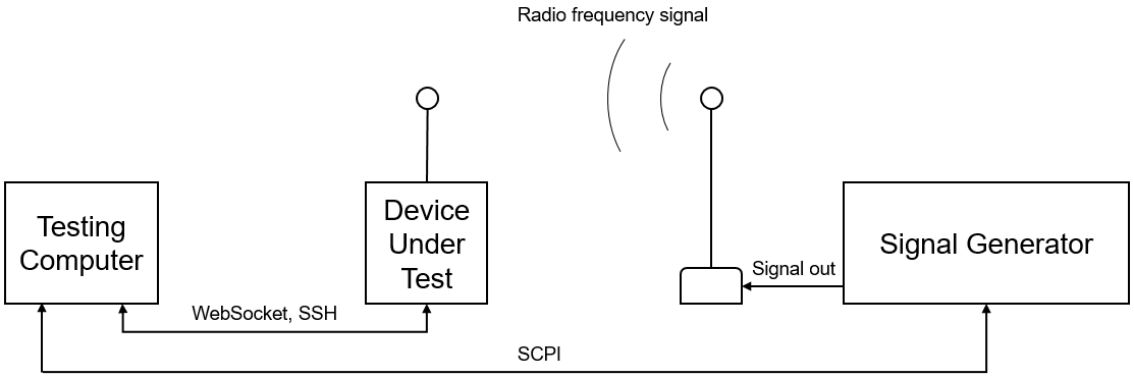


Figure 8. *Testing computer, device under test and signal generator.*

Complete testing setup features the use of antennas for radio frequency transmission. This creates a setup that most closely mimics a real world environment, and allows testing as many features as possible.

5.2.1 Testing setup challenges

Wireless transmission has challenges when it comes to testing. A wireless signal is easily distorted, suppressed or reflected because of the surrounding environment. When creating the testing setup for this thesis, all of the above phenomena were observed. The wireless testing configuration is presented below in Figure 9.

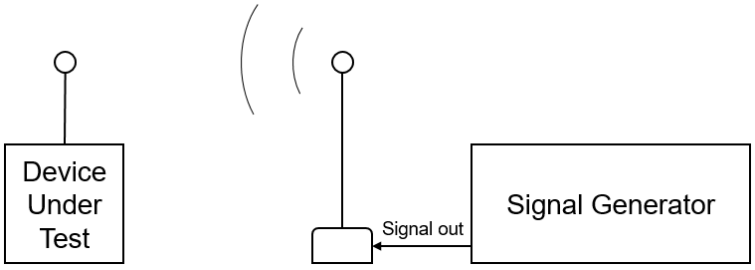


Figure 9. *Wireless testing configuration.*

Testing wirelessly with antennas does not guarantee equal test results on subsequent test runs. Even when no modifications have been made to any part of the setup, static background noise causes interference with the signals, resulting in inaccurate measurements. Getting different test results on equal setups is not an optimal situation. If the results do not stay the same when no modifications are made, it is impossible to decipher whether an abnormal test result is the result of a modification of the system.

To counteract signal interference, the tests can be performed in a special environment, as presented in chapter 3.1.4. Another possibility is wiring the signal generators directly to the device under test, which bypasses the problems with wireless signals. A wired connection alternative is presented below in Figure 10.

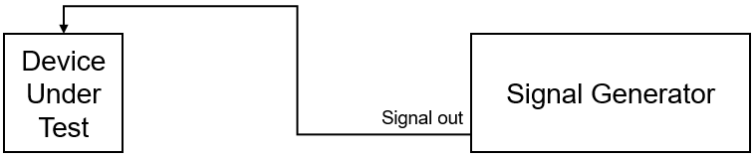


Figure 10. *Wired testing configuration.*

A wired configuration is simpler than performing the tests in a specialized environment. This is a compromise regarding the testing of the whole device. When the signal is guaranteed to be interference free, the software and hardware stack of the system requiring the signal input can be tested. When no antennas are used, the stack of the device handling the incoming radio frequency signal is not used. Simplified comparison is presented below in Figure 11.

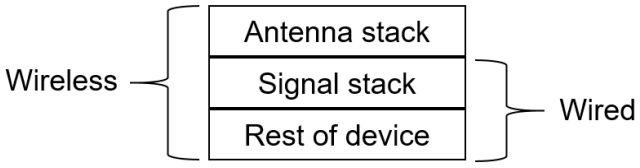


Figure 11. *Test coverage of different testing configurations.*

For automated tests, wired testing configuration was found to be better. When there is need to test the whole device, for example in release testing, a wireless configuration can and must be used.

Additional problem in the testing setup is the required laboratory devices. The setup does not scale well, as possibly complex laboratory equipment is required to test all functionality. The equipment may not be always available, or there exists only a limited amount of equipment. For these reasons, considerations on using a custom signal generator are presented in chapter 6.1.2.

5.2.2 Device under test

The device under test is controlled from the testing computer. The communication is done over network connection with WebSocket and SSH protocols, which provides a high level communication interface to the device.

The device has internal components, which can be controlled using the same interface. These components cannot be directly interfaced over the network connection, and instead the device, using internal interfaces, handles the communication with them. However, from testing point of view, the higher level interface is flexible and practical for creating automated tests with a testing framework.

The device under test receives radio frequency signals as inputs and creates an output, which is read from the device. It must be tested both for execution of this task, and for the functionality of all auxiliary devices related to this task. The testing path contains both the hardware receiving the signal and the software deciphering it. It is presented below in Figure 12.

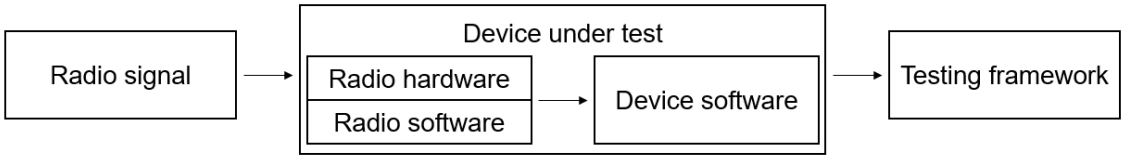


Figure 12. *Device testing path.*

The signal-receiving stack has both hardware and software that must be tested to ensure correct operation of the complete device. After receiving a signal, the device processes it and generates an output, which is transmitted to the testing framework.

5.2.3 Test data signal generator

Testing framework controls the test data signal generator over network, using SCPI messages. The signal generator produces the radio signal used as an input. Communication path leading to the radio signal is presented below in Figure 13.

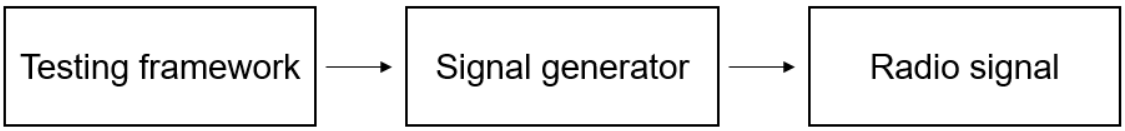


Figure 13. *Test signal creation path.*

The testing framework communicates the desired output to the signal generator. The signal generator produces the radio signal, which is transmitted to the device under test. The signal may be transmitted either wirelessly or by a wired connection.

5.2.4 Test input parameter selection

In chapter 2, the reasons for testing were discussed. Errors in the device under test's functionality can be discovered by using inputs for which the correct outputs are known. The inputs should be chosen for the highest possible coverage.

An example test case in this thesis is verifying the frequency of a received signal. This displays the different features of radio frequency and physical device testing, as the test result is affected by all the presented challenges. The test case is also suitable for a laborious data driven testing example, which can be automated.

Considering the testing is done with radio frequency signals, the scale of any frequency spectrum is practically infinite. It is not possible to test the device with each frequency there exists. Instead, testing is done with boundary value analysis, as discussed in chapter 2.3.1, using data driven testing methods, as discussed in chapter 4.2.1.

Physical components inside the device have different characteristics. The components are designed for a given frequency spectrum, and their functionality deteriorates at the extremes of this spectrum. Therefore, testing the frequency spectrum is a perfect example of a boundary value analysis.

Even though unit testing methods are used, the device under test is being system tested. The device works as a complete system, and complete functionality is required.

Receiving signals is tested with a nominal value inside the characteristics and with the smallest and biggest values at the ends of the characteristics. The system is also tested with values near the minimum and maximum values, just below or above. Robust testing is done by testing the values outside of these values. It is expected that the system will not work as expected, but the correct behaviour must be ensured. Test input value points within the these limits are presented in Figure 14.

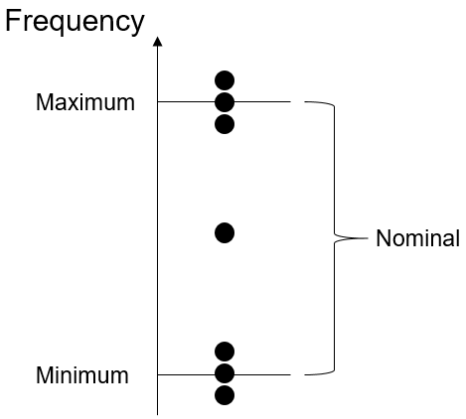


Figure 14. *Test input value points within characteristic frequency scale.*

Similar method can be extended for other parts of the system and their characteristics. Instead of the signal frequency, the limiting factors may be signal amplitude, processing power of a component or a bandwidth limit between system components.

Different input values for data driven test cases are provided by test data files. The testing framework reads these files, and uses the values to configure the signal generator and the device under test accordingly.

5.3 Software environment

Software environment consists of test automation framework running on a testing computer. The software environment of the testing environment in following chapters is presented. This includes the software stack used, as well as the architectural decisions when developing the tests. The decisions regarding software environment, such as choosing the test automation framework, affect the future development and the usability of the setup.

5.3.1 Test automation framework

Robot Framework was chosen as the test automation framework. Pytest was equally considered because of the open nature of the framework, and because of the flexibility of Python. However, Pytest was not as versed in system testing with complex setups and teardowns. Robot Framework also has powerful logs, which can be followed to the point where a test failed for any reason. Pytest can be used in a similar manner, but pinpointing the failure on the default output was harder. Robot Framework creates comprehensive results of each test. The results are easy to understand, even with no understanding of Robot Framework.

Additional reasons for choosing Robot Framework include the versatility in data driven testing. Testing with different data is possible in Pytest, but it is easier in Robot Framework. The use of Pytest's parametrization for data driven testing is not suitable for complex setups and long testing chains. In Robot Framework, tests can be more easily and clearly pieced to setups and teardowns, with data driven testing performed between them.

Following are additional reasons that concreted the choice of Robot Framework. Pytest also supports some of the same features, but the advantages presented above led to choosing Robot Framework over Pytest.

Robot Framework is compatible with Python. It is possible to create complex methods in Robot Framework itself, but it is also possible to drop down to Python and create the method there, instead. As Python itself is already a flexible language, this makes Robot Framework even more flexible. Anything that can be done in Python can be called from Robot Framework as an external library.

Robot Framework is platform agnostic. The framework can be used on Windows and on Linux environments. Linux environment support has the added upside of making containerization of the test environment feasible. This is possible with tools such as Docker (Docker 2022). Containerization makes it possible to distribute the testing environment to multiple platforms and devices without the need of device specific setup.

The software stack in the test automation setup consists of a Windows host with Python 3 installed. Robot Framework version 5.0.1 is installed as a Python package. If wanted, Robot Framework can be run on both Windows and Linux hosts. Running on Linux is possible natively or though virtualization, using technologies such as Windows Subsystem for Linux (WSL) or Docker.

For communication with different signal generators, a custom-built Python package was used. A library for Robot Framework was created using this package. Communication with the device under test is similarly achieved by the use of Robot Framework libraries. Required libraries are either provided by Robot Framework or custom-made.

5.3.2 Testing framework architecture

As discussed in chapter 4.3.1, the tests are created with test files, supported by keywords and libraries. A test file is understood by Robot Framework as a test suite, which contains multiple test cases. As the tests are divided to different files, architectural decisions must be made. A structured testing project is easier to maintain and improve.

To configure different testing environments, a file containing test variables is used. The libraries provide additional functionality and interfaces to communicate with required devices. The variables are common variables, and also provide test data. An example of the framework architecture is presented below in Figure 15.

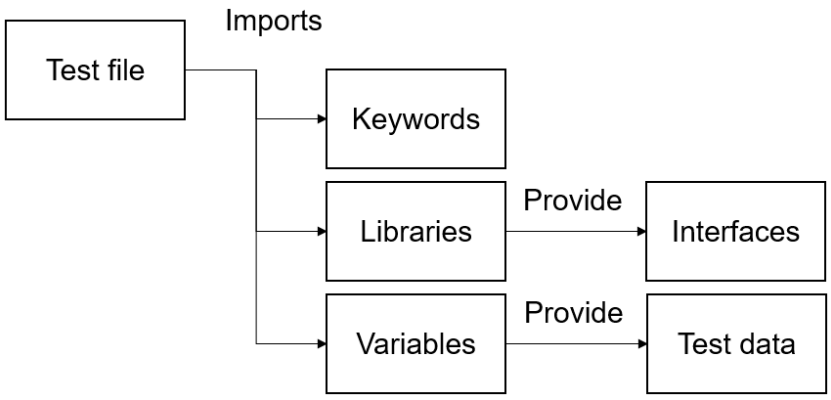


Figure 15. *Testing framework architecture.*

The files containing keywords are shared keywords, or keywords specific to certain test suites. Libraries are modules, which translate functionality of Python programs to be usable as keywords in Robot Framework.

Variables are global, suite-specific and environment-specific. Global variables do not change between test suites, and contain values such as network addresses to different devices. Suite-specific variables are used to test the functionality the suite is targeted at, which can be a specific functionality or component. Environment-specific variables are used for different testing environments. Testing the device with wireless signals and testing the device with hard-wired signal lines create two different environments, as an example. The desired environment is chosen by passing a variable to Robot Framework when starting test execution, denoting either a wired or a wireless testing configuration.

Test data may contain, for example, a list of frequencies that should be tested. They may also contain data for other tests.

In Robot Framework, test cases consist of multiple keywords being executed one after another. If a test case fails, the failure is pinpointed to the keyword which failed. This promotes piecing the test case to as many keywords as possible, which should also be as informative as possible.

5.3.3 Common and project specific libraries

Some of the libraries can be shared between different testing environments. This is beneficial when testing devices other than the device under test of this thesis. Most notable of these libraries are the one providing interfaces to laboratory devices. The underlying

Python code is packaged as a Python package, to be used with Python programs. This package can be used in a library created for Robot Framework. The Robot Framework library acts as a wrapper for the Python package functions.

In addition to sharing libraries, Robot Framework's keyword files can also be shared. Commonly used keywords should be constructed in a way that they can be used with different projects. Such keywords are ambiguous and not device-specific. Even though each different project has unique testing needs, shared keywords could be methods used to calculate or test similar parameters, which must be tested between different devices. For example, a similar radio frequency device may have identical or similar components, which can be tested in the same manner.

Libraries used to communicate with the device under test are specific to each project. Different devices have different communication interfaces, and the interfaces may use different messaging protocols. Using a common library for every project is not possible, unless the different projects use an identical interface. Challenges regarding testing interfaces are discussed more in chapter 6.2.2.

5.4 Test flow and execution

Executing a test starts from the test case. The test case is interpreted by Robot Framework, which starts the test execution. Necessary keywords are imported to setup the framework to be ready for testing. The keywords call for libraries, which provide interfaces to the device under test and the laboratory equipment.

Testing the device under test consists of initializing it, and setting it to the desired state to listen for radio frequency signals. After the device is ready, external signal generator is used to create a signal. When the signal is transmitted, the framework waits for a response from the device under test. The device deciphers the results from the signal and communicates the result to the framework. Finally, the framework compares the result from the device to the known correct result. An example sequence executing a test case is presented below in Figure 16.

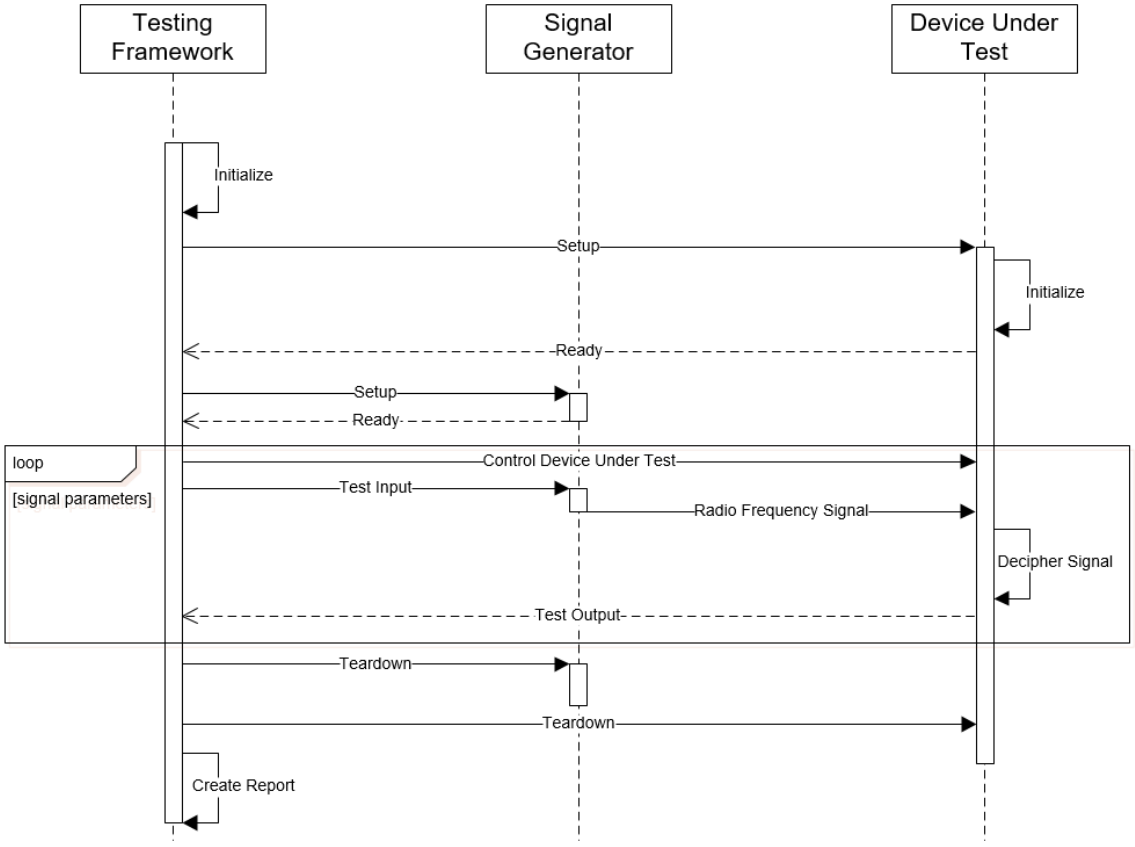


Figure 16. *Test execution flow.*

A testing thread, which were presented in chapter 2.5.2, can be followed in the test execution flow. Above test case presents the thread to test the device under test with different signal parameters, where each discrete step in the test execution flow presents an atomic system function. Atomic system functions were presented in chapter 2.5.1. With different test cases, different threads are tested. Signal parameters in the loop of the test execution flow present different variable values, used for data driven testing.

The test report can be examined for the whole execution path of the test. In the case of a successful test, there is little need to examine the whole report. If a test case fails, the report can be used to pinpoint the reason. Output of the testing framework was examined more thoroughly in chapter 4.3.1.

An example of a test suite for frequency testing is presented below in Program 3. The suite with its related files are presented as a simplified pseudo example, which mimics how the tests are performed with the true setup.

```
*** Settings ***
Documentation   Tests for verifying correct frequency readings.
Resource        frequency.resource
Variables       config_${ENVIRONMENT}.py
Suite Setup     ${ENVIRONMENT_SETUP}
Suite Teardown  ${ENVIRONMENT_TEARDOWN}


*** Variables ***
@{FREQUENCIES} =    ${None}


*** Test Cases ***
Test Frequencies
    [Template]  Test Frequency
    FOR     ${frequency}    IN  @{FREQUENCIES}
        ${frequency}
    END
```

Program 3. *Contents of test_frequency.robot file.*

Creation of a test case starts by identifying the feature that needs to be tested. Execution of the test is pieced together by different actions, which must be performed to complete the test. These actions are the atomic system functions, which will be formed into keywords. Piecing together the keywords produces the whole test sequence.

Data driven testing is achieved with the use of a template test case. The test suite is supported by a resource file, imported in the Resource-section. The file contains the necessary keywords for test execution. It is presented below in Program 4.

```
*** Settings ***
Documentation   Keywords for frequency reading testing.
Library         communication.py
Resource        common.resource


*** Keywords ***
Setup Suite Wireless
    Read Csv
    Setup Device Under Test
    Setup Signal Generator


Teardown Suite Wireless
    Teardown Signal Generator
    Teardown Device Under Test


Read Csv
    @{data} =   Read Csv File   ${TEST_DATA}
    Set Suite Variable  @{FREQUENCIES}  @{data}


Test Frequency
    [Arguments]     ${frequency}

    Control Device Under Test
    Set Signal Generator    ${frequency}    ${AMPLITUDE}
    ${output} =     Get Device Under Test Output

    ${deviation} =  Evaluate    abs(${output} - ${frequency})
    Should Be True  ${deviation} <= ${MAX_DEVIATION}
```

Program 4. *Contents of frequency.resource.*

The environment is chosen with a command line parameter. The correct variable file is chosen for the chosen environment. Contents of a variable file for wireless environment are presented below in Program 5.

```
ENVIRONMENT_SETUP =     "Setup Suite Wireless"
ENVIRONMENT_TEARDOWN =  "Teardown Suite Wireless"

TEST_DATA =             "frequencies.csv"
AMPLITUDE =             1
MAX_DEVIATION =         10
```

Program 5. *Contents of config_wireless.py.*

The test suite is run with the command `robot --variable environment:wireless test_frequency.robot`. This sets the tests to use the correct configuration file, as it is imported in the `test_frequency.robot` file based on the variable value.

The test suite follows the operation sequence presented in Figure 16 and the framework architecture presented in chapter 5.3.2. Other files not presented are the `frequen-`

cies.csv, communication.py and common.resource. The file frequencies.csv contains the test data, which is different frequencies. The file communication.py is the library required to communicate with the device under test and the signal generator. File common.resource contains all the other necessary keywords and variables.

5.5 Result validation

Interpretation of the test results can be either relatively simple or complex. Simple results can be processed locally within the testing framework, whereas complex results need specialized software and expertise to be deemed acceptable. Deciding, whether a test has passed, can be done immediately by the framework for simple results. Complex results must be interpreted externally, and the framework cannot provide an immediate result regarding them.

Simple results are compared against the known correct values. The result must either match exactly, or it must be within set bounds. Exact values are known, for example, for the states of different internal components of the device.

Complex results cannot be easily determined being correct or incorrect. In the case of some test values, additional computation must be done to determine whether the result is acceptable or not. Not all of these computations can be done by the testing framework, and they must be done at a later time with specialized programs.

Some of the more complex analysis can be integrated to the testing framework, if an interface is created for the purpose. Many of the analyses are however computationally intensive. As such, they cannot be part of the test execution flow, as they would slow down the testing process. For example, regression testing should be kept reasonably quick to gain the benefits of agile development methods.

6. RESULTS AND ANALYSIS

The purpose of the thesis was to create a setup for testing a complex radio frequency embedded device, and to identify and solve the problems in creating such a setup. A working framework was created with multiple purposeful test cases, with an example of a test case utilizing the setup presented in chapter 5.4. The created setup enables automated testing of the device under test, and the setup works as a foundation for further test cases and improvements. As this was the goal of the thesis, the results can be deemed a success.

While there were problems when creating the setup, they were identified and worked around. Many of the problems could not have been discovered without creating the testing setup. Identifying the problems at this early level is a great benefit when moving forward.

The challenges in creating the testing setup are dissected in the following chapters. Additionally, the research of this thesis is compared to previous works, and the next steps for continuing development of the setup are presented. Finally, answering the research questions is analysed.

6.1 Test equipment

The equipment used for the testing setup consists of a signal generator and the device under test. For complete testing, different types of signals must be created. Creating the different signals is not possible with all signal generators, which limits the testing capabilities. In this chapter, the notable challenges, along with possible solutions, of signal generation and signal transfer are presented. The observations in the following chapters were noted when developing the testing setup.

6.1.1 Limitations of commercial signal generators

A basic signal generator can be used to create a continuous wave signal. Only few parameters of the signal can be modified, such as the frequency and the amplitude. To create more complex waveforms, different signal generators such as arbitrary waveform generators must be used.

While it is possible to create many types of signals with commercial equipment, the signal generators capable for complex signal generation are often expensive. Creating a persistent testing setup for development and regression testing requires subjecting such

generator for the setup. This may not be desired, as any expensive equipment is typically utilized as much as possible at any given time.

Limiting factors of commercial signal generators proved to be the feature sets of simple generators, and the price and availability of more featured ones. For this reason, creating and using custom testing equipment was explored. This is discussed in the following chapter 6.1.2.

6.1.2 Custom signal generator

Commercial signal generators have a vast set of features, which can be used to create different types of signals. In the scope of a development product, most likely not all of these features are needed. This promotes creating a custom signal generator that can create all the necessary signals for the required test cases.

In this thesis, the feasibility of creating a custom signal generator for test signal generation was explored. A custom signal generator has many upsides when comparing to commercial signal generators. For a specific purpose, such as testing the device under test of this thesis, the required signal properties are known. This makes it is possible to use components that fulfil all the needs of the testing setup. If only a subset of the features of a commercial signal generator is needed, the features can be implemented in the custom signal generator. Depending on the required signal features and parameters, a custom signal generator can be a significantly cheaper solution than using a commercial generator.

Creating a custom signal generator gives more control over the features of the generator. However, this is also one of the downsides. Creating a signal generator from scratch requires a lot of work, up to getting to the point where the device can be integrated to the testing flow. A commercial signal generator on the other hand can be integrated immediately, with known interfaces and performance.

Creating a custom signal generator for the testing setup seemed to be a viable option. For most of the testing, such as regression testing, a custom signal generator can be used. This frees up the commercial signal generators for other use. For complete testing, a custom signal generator alone may not be enough. For performance analysis and release testing, a reliable and tested generator may need to be used.

6.1.3 Challenges of wired and wireless testing

In chapter 5.2.1, it was identified that testing with a wired configuration provides more repeatable results. Depending on the testing situation, the tests may need to be run with

either the wired or the wireless configuration. Different configurations have different variable files, containing the same variables but with modified values. One such variable is the signal amplitude of the signal generator.

Signal amplitude is the strength of the signal being generated at the signal generator. If the amplitude is low, the signal may be too weak and will not be transferred through. Too strong signal on the other hand may cause damage to some components.

When testing with a wireless configuration, the amplitude must be high because of losses occurring in the wireless signal transfer, and because of background noise. With wired configuration, these losses are lower and the device is less susceptible to background noise, and a lower amplitude is sufficient.

Physically setting the device under test to a wired configuration means physically wiring the output of the signal generator to the antenna port of the device. The configuration is chosen when running the tests, and must be set to either wired or wireless. The different physical configurations were presented in Figure 9 and Figure 10. If the wireless tests with high amplitude are run with the wired configuration, there is a risk of damaging the device under test.

Problem with the physical configuration is that the testing setup has no knowledge whether the signal is provided by wired or wireless route. Blindly running the wireless tests uses the high amplitude setting for the signal generator, possibly causing damage. From usability perspective, it should be as hard as possible to cause damage with an incorrect configuration. To mitigate this problem, some fail-safes were proposed, one physical and one on the software side.

A physical attenuator could be added to the signal path from the signal generator. The attenuator would be placed on the signal generator side, before connecting the rest of the output to the antenna, or directly on the device under test. The attenuator would be sized in such way that using the signal generator, with the highest amplitude setting and a wired connection to the device under test, would not damage the device. The attenuator would also need to be small enough to allow testing with the wireless configuration. This sets physical limits to the signal generator, as the signal generator must have high enough maximum output amplitude that the attenuation can be ignored. The maximum possible output cannot be too strong either, as that would make the final signal too strong, even with the attenuation.

A software side fail-safe could consist of a test sequence before any actual tests take place. This pre-test would generate a simple signal with a low amplitude, which does not

harm the device under test, but is still decipherable by the device with the wired configuration. The amplitude needs to be low enough that it is undecipherable with a wireless configuration.

If this test signal is decipherable, it is possible to deduct that the device under test is connected with a wired configuration. If this is the case, the amplitude is limited to a safe level. The limit and the testing should be incorporated to the device's testing library. This way it is harder to skip the pre-test, if testing is done using the library.

Downsides of the software side fail-safe are the possibilities of bugs and accidentally setting too high amplitude limits in some parts of the tests or software. A physical fail-safe in the form of an attenuator should be preferred. With a physical limitation, no software fault or mistake can cause damage to the device under test.

The setup created in this thesis does not employ either of these fail-safes. The requirement for such was identified and left for future development.

6.2 Test interfaces

For complete testing path of the device under test, different interfaces are used. These interfaces present some problems for creating accurate test results. There are also some considerations on how to implement a testing interface to a device under test, and what interface should be used for testing. In the following chapters are observations, which were made when communicating with the device under test in the setup.

6.2.1 Remote testing

The device under test is controlled with a remote interface. The interface is used to send commands to the device to set it to correct state, and to read its output.

In some situations, the test interface is too slow. As the functionality of the device relies on hardware components, these components must be tested. Some components may require testing with quickly changing inputs, and the response of the component is gauged as the result.

A possible test case is changing the state of an internal component, and verifying that the component continues to operate correctly. These tests are hard to perform, as the incorrect behaviour may be exhibited only once in a thousand state changes. It is also possible that the incorrect behaviour happens if the component has just changed state, and is immediately commanded to change state again.

For the above reasons, the component must be rapidly switched to different states. Creating quick changes over a remote interface may be fast on human scale, but on the hardware level, it can be too slow. This was identified as a limitation when creating such test case for the device under test.

It was determined that in this kind of situation, the testing must be performed on the device, by the device. This removes the delays in communication. The test must be written at the hardware level, even if there is an interface to control the component remotely. After performing the test on the device, the device must communicate the results to the testing framework, which decides whether the test was successful. A test case utilizing such technique was not created in the scope of this thesis.

6.2.2 Communication interface

In the device under test of this thesis, communication for testing is done over the same interface as any other communication with the device. There is no separate test interface. This makes controlling the device straightforward, and at the same time, the communication interface is tested by using it.

Libraries are created in Python and Robot Framework to communicate with the device under test. As they are created this specific device in mind, the libraries only work with the same or identical devices, and they cannot be used to control different devices.

The communication libraries could be reused with other devices, if the communication interface were to be standardized. This would save time in the future when testing is performed on other devices, as there would be no need to create new libraries for each different device.

Downside of the common control interface is the need for its implementation. As presented in chapter 5.2.2, the device under test uses a high level communication interface. A simpler, lower level device could not implement the same interface. Theoretically, the test interface could be standardized to low level interface, such as a serial interface. This way the interface could be implemented to most of the devices which would be tested. In the case of the device under test of this thesis, this would require additional work, as the device already has a feasible communication interface. The additional test interface would be used only for testing, and the benefit of testing the communication interface during testing is lost.

6.3 Test accuracy and repeatability

Testing a radio frequency device is balancing between accuracy and repeatability. Accurate measurements, and therefore accurate test results, are created in an isolated environment. These do not reflect the real world, however. The device should be tested in an environment as close to the real world as possible, while still maintaining consistent test results.

This raises the question, whether the tests should be performed in an environment simulating the real world, or in an isolated environment. Basic operation of the device can be verified in an isolated environment with a test signal that stays in the nominal range. In a real world environment, with more noise and a weaker signal, the device should still work as specified. It is expected that the device will not perform as well in these conditions, but correct operation should be tested regardless.

Accuracy and repeatability must be balanced when deciding the different testing environments. A wired testing environment provides accurate results but at the cost of not testing the antenna stack, which is a big part of the radio frequency properties of the system. A wireless testing setup provides the truest results, but the tests may be inaccurate because of background noise and interference.

For repeatable and comparable results, wireless tests must be conducted in a specialized environment, as examined in chapter 3.1.4. For example, wireless testing outside a shielded environment is susceptible to background noise created by different everyday devices. This requires using test frequencies outside of the busy bands, which may not be possible or desirable.

The testing setup was created in such manner, that it is possible run the same exact tests in both environments. This provides both accuracy and repeatability. Tests passing in the wired configuration can be tested in wireless configuration, and possible failures can be pinpointed to the antenna stack, pointing to a fault in the handling of wireless signals.

6.4 Previous works

Some previous works were analysed as part of creating this thesis. Work on the setup of this thesis started as an in-house project before analysing external literature. Literature and previous works were searched on the topic, but with little success.

Embedded systems or radio frequency systems are not uncommon, and therefore they are not uncommon testing targets. Devices such as mobile phones are prevalent in today's society, and development of these devices requires testing.

As many of these devices are commercial, the testing projects for them are completed internally in their respective companies. Public works describing exactly radio frequency embedded system testing are hard to find, and challenges in testing any specific device are often specific to that device itself. If a device under test has specific testing needs, a previous project addressing different needs is only of limited use. Coincidently, similar observations have been made in at least one previous work. The work of Korhonen (2021) is about solving the problem of each product requiring a dedicated test automation system (Korhonen 2021, p. 2). In the same work, the difficulty of finding production testing literature is noted, because the relevant data is often internal data of the companies (Korhonen 2021, p. 67).

While exact previous works or literature is difficult to find, other testing projects featuring an automated testing framework and a controllable external system do exist. In recent years, there has been some works on such systems, where automated tests are performed on external hardware, and other hardware is used to create test inputs.

Turunen (2018) in the thesis "Automated UAV Testing" presents using Robot Framework for testing an unmanned aerial vehicle. The thesis features communicating with the device under test and test signal generation devices, using serial and SSH connections. Test signals are generated with external devices such as a programmable power supply and a modem. (Turunen 2018, pp. 9–12, 33)

The above setup bears similarity to the setup in this thesis. However, as noted, the device under test determines the requirements for the setup. Even though the setup of Turunen (2018) uses an automated testing framework, and similar communication methods between the device under test and the test input devices, the testing requirements are not identical to the setup in this thesis. As such, the work of Turunen (2018) cannot be used as a direct reference.

The already mentioned work by Korhonen (2021), in the thesis "Generalizing Production Testing Operations for IoT Devices", extends a testing setup to work for testing different products. In the thesis, a specialized test fixture is used in the physical testing setup, which is used to control the device under test. Possible auxiliary devices are connected to the fixture, and used to create test inputs. Automated test software is used to control the device under test and the auxiliary devices. (Korhonen 2021, pp. 2, 7–8, 31–32)

The thesis is similar in a way to this thesis, as testing is performed on an embedded device. However, the thesis does not examine in detail how the testing setup is created, as it focuses more on the problem of sharing a common testing setup. The thesis focuses on what could be the next steps of this thesis.

The above-mentioned theses were examined before starting the work on this thesis, and re-examining them after finishing the work on this one reveals some similarities. Design of the test automation by Turunen (2018) presents execution of the tests on the device under test, starting from a software build on an external server (Turunen 2018, pp. 23–24). A similar setup is the end goal of the work in this thesis. Korhonen (2021) presents using configuration files for testing different hardware on the same setup, which allows creating widely applicable test cases (Korhonen 2021, p. 69). In this thesis, different configuration files are likewise used for different environments, and to communicate with different devices, using the same setup.

Above are some similarities from the two examined works. Neither of the works overlaps completely with the work of this thesis, but similarities exist. There have been similar challenges or design goals in all the works, which have been solved in a similar fashion.

In hindsight, more influence could have been taken from previous works, per the similarities of the challenges and goals in them. However, a too deep dive would have taken time from the concrete development work of this thesis. There is also a bias, now that the challenges of this thesis have been identified. Before starting the work on the setup of this thesis, the challenges were not known, and therefore could not have been researched.

6.5 Next steps

The objective of this thesis was to create a testing framework base to build upon. In the following chapters, the next steps of the development are presented. When creating the current testing setup, these future features were taken into account.

6.5.1 Test software virtualization

Robot Framework can be run on a virtual host. For example, it is possible to create a Docker image with Robot Framework installed, and use containers of this image to run Robot Framework. A ready-made testing image with all the necessary software has many upsides.

Currently the testing software is simple. It consists of only Python, Robot Framework and some libraries. It is possible that in the future the required software starts to increase.

There can be, for example, specialized software that is needed to interface with some devices. Creating a common image with all the required software allows running the testing framework easily on different devices, such as the personal computers of developers, or on a remote server. A common image ensures that the testing software environment is identical for all developers, as well as reduces the time to setup the testing environment for each developer.

There are some downsides in using virtual clients to run test. Running Robot Framework from a virtualized client is straightforward, if communication with other devices is done over a network connection. Depending on the hardware, communication may only be possible over a serial interface or other form of physical connection. These require pass-through from the virtual client to the host computer to make communication possible.

Additional challenges may rise from proprietary software, which may be required to interface with some components. Depending on the software, it may not be possible to run it on some operating systems. A component may have drivers written only for Windows, which makes it impossible or hard to interface with the component over Linux. Typically, Docker images are based on Linux, and this may cause problems.

The device under test of this thesis is interfaced over a network connection, as is the signal generator that is used for test data generation. This makes interfacing possible from any platform supporting common networking protocols, and therefore the platform is not limited to a specific operating system.

6.5.2 Continuous integration

The setup used in this thesis makes continuous integration possible, as the software has been selected with continuous integration in mind. Below is presented how to continue the work towards the continuous integration pipeline.

For continuous integration purposes, Robot Framework can be integrated to a continuous integration tool, for example Jenkins. This allows running Robot Framework tests and displaying the results in Jenkins. (Jenkins 2022)

Connecting the framework to the continuous integration pipeline allows running the tests after each modification. The test can be run, for example, on each new commit of source code. When the tests are run automatically, there is no need to manually run tests after each change. Automated testing ensures that mistakes are caught early, and all the errors that would be caught by existing tests, are found.

Continuous integration and regression testing requires a testing setup that does not change. For the setup of this thesis, this is a system with the device under test, and an

external signal generator. These are set up with the desired radio frequency transmission configuration, which is either wireless or wired.

As pointed out in chapter 5.2.1, a setup with wireless radio transmission does not always produce identical results, unless used in a special environment. Therefore, for continuous integration, the setup should use wired transmission. The downsides are that the whole device stack cannot be tested, as examined in the same chapter. However, a quite complete stack can still be tested, which should reveal faults in most of the stack and in the software.

Continuous integration and the use of integration pipeline can be used for test result analysis. Analysing test results was discussed in chapter 5.5, where the test results were determined being either simple or complex. The more complex results require additional computation, which can be integrated to the continuous integration pipeline. As the simple test results are validated in-framework, a separate process starts to validate the more complex results. This way the test execution flow does not halt at the computationally intensive result validation, but the results will regardless be available later.

6.5.3 Robotic process automation

Robotic process automation (RPA) is automation of tasks and processes. These solutions work their way through different software interfaces, completing tasks often done by humans. Robot Framework has support for performing robotic process automation. (Robot Framework 2022d)

Instead of running the set of tasks in a test case, the tasks can be used to automate processes. An example of this is used on this thesis, when setting up the test automation environment. As examined in chapter 5.4, there is an initialization phase before performing the tests. This phase is not relevant to the tests itself, as it is only setting up the testing environment.

In a similar manner, Robot Framework can be used to automate other tasks that require controlling auxiliary devices and the device under test. For example, Robot Framework can be used to perform long tasks, which would otherwise be done by a human operator.

An example case of a long task is calibrating components in different temperatures. This involves controlling a heating cabinet by changing the internal temperature, waiting for the temperature to stabilize, and then running tests on the hardware. Instead of expending a human operator for this task for many hours, it can be automated using robotic process automation. This saves working time and ensures an identical calibration process over multiple runs.

6.5.4 Production testing and maintenance

The same testing setup can be used for future testing needs in the form of production testing and maintenance. During and after production, a unit must be tested to verify correct performance. Instead of having a designated device permanently in the testing setup for development purposes, any device could be connected to the setup to verify the correct operation of a specific unit.

For production testing purposes, instead of only checking whether a test passes, the output values of the tests will be saved. This allows pairing each unit with an exact report of all tests and their results. If a unit needs maintenance in future, the same tests can be used to determine correct performance and to pinpoint faults. Due to Robot Framework's tests' program-like nature, old tests can easily be restored from a version control system. This allows running not only similar, but the exact same tests, that have been used when developing and production testing the unit.

6.6 Research questions

There were two research questions for this thesis. How to implement an automated testing setup for a device, which requires complex measuring equipment, and what decisions contribute to the creation of a modular test automation setup.

Implementing the testing setup is presented in the chapter 5. An implementation requires a suitable testing framework, which must be able to control the device under test and the required test input generators. On physical side, interfaces on the different devices are in a key role. An additional challenge is created by the radio frequency requirements, where a balance must be found between test accuracy and repeatability.

Many decisions contributing to the modularity of the setup were identified. Feasible software choices and architectural decisions promote reusability. This is achieved by creating shared libraries, and test cases that are applicable to different testing environments. The testing setup software was discussed in chapter 5.3.

This thesis can be deemed to answer both of the research questions. The setup created in this thesis forms a working framework, which is to be developed further. Modularity being a notable feature also in the future, the design decisions made in the early phases have a far-reaching impact.

7. CONCLUSIONS

The purpose of this thesis was to create a test automation setup for Saab Finland Oy, to test a complex radio frequency embedded device. As the result of the work, an automated testing setup was created and the results of the thesis can be deemed successful.

The research questions of this thesis were how to implement an automated testing setup for a device, which requires complex measuring equipment, and what decisions contribute to the creation of a modular test automation setup. The first question is answered by a case study in creating the setup, taking in account the special requirements of the device under test. The second question is answered by identifying supporting decisions while creating the setup, such as architectural decisions to create shareable elements.

The creation of the setup required special attention due to the nature of the embedded radio frequency device and its testing requirements. Testing such device required interfacing with both the device and an external test signal generator. A purposeful test automation framework was needed to control the device under test the external signal generator, for which Robot Framework was chosen from the possible candidates.

Several challenges were observed during the creation of the setup. Many of these were caused by the difficulty of testing radio frequency equipment, and the devices required for such task.

A balance between accuracy and repeatability had to be found. Outside of specialized environments, wireless signals were susceptible to external interference. A proposed solution is using two different testing environments for different requirements, one with a wireless and the other with a wired signal transfer configuration.

The setup created in this thesis can be used to test a device with no user intervention. The tests are run automatically, and they produce a detailed report on how the tests succeeded. The testing setup was proven successful, as it was able to reveal known and unknown faults. The faults were revealed during the development work of the setup, and when running tests using it.

Currently the setup works as a standalone implementation, with no further integration to continuous integration systems. Next steps are increasing the features of the setup, such as implementing a continuous integration pipeline. The setup created in this thesis is the first iteration of the testing environment, and finishing the setup is left for the future iterations.

REFERENCES

Axelson, J. (2007). Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems. 2nd ed. Madison, WI: Lakeview Research LLC.

Banks, A. ed., Briggs, E. ed., Borgendale, K. ed. & Gupta, R. ed. (2019). MQTT Version 5.0. OASIS Standard. Available (accessed on 11.11.2022): https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

Docker. (2022). Docker: Accelerated, Containerized Application Development. Website. Available (accessed on 15.11.2022): https://www.docker.com/.

Fette, I. & Melnikov, A. (2011). RFC 6455: The WebSocket Protocol. Available (accessed on 11.11.2022): https://www.rfc-editor.org/rfc/rfc6455.html.

Fielding, R., ed., Nottingham, M., ed. & Reschke, J., ed. (2022). RFC 9110: HTTP Semantics. Available (accessed on 10.11.2022): https://www.rfc-editor.org/rfc/rfc9110.html.

IEEE Std 1149.1-2013. (2013). IEEE Standard for Test Access Port and Boundary-Scan Architecture. IEEE Standards Association.

ISO/IEC 7498-1:1994(E). (1996). Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. 2nd ed. Corrected and reprinted 1996. ISO/IEC.

Jenkins. (2022). Robot Framework | Jenkins plugin. Website. Available (accessed on 15.11.2022): https://plugins.jenkins.io/robot/.

Jorgensen, P.C. (2014). Software testing, A Craftsman's Approach. 4th ed. Boca Raton, FL: CRC Press.

Juurinen, L. (2020). Test Automation for Control Applications on Distributed Control System. Master of Science Thesis. Tampere University.

Karhula, V.-M. (2022). Ajoneuvojen mittausjärjestelmän toiminnallinen ohjelmistotestaus. Master of Science Thesis. Tampereen yliopisto.

Korhonen, K. (2021). Generalizing Production Testing Operations for IoT Devices. Master's Thesis. University of Oulu.

Krekel, H. et al. (2022a). Pytest documentation, Release 7.1. Available (accessed on 28.10.2022): https://docs.pytest.org/_/downloads/en/7.1.x/pdf/.

Krekel, H. et al. (2022b). pytest: helps you write better programs – pytest documentation. Website. Available (accessed on 26.12.2022): https://docs.pytest.org/en/7.1.x/index.html.

Krekel, H. et al. (2022c). Managing pytest's output – pytest documentation. Website. Available (accessed on 26.12.2022): https://docs.pytest.org/en/7.1.x/how-to/output.html.

Lammi-Mihaljov, H. (2020). Testivetoisen ohjelmistokehityksen soveltaminen HTTP-palvelimen kehittämisessä. Master's Thesis. Tampereen yliopisto.

Liechti, C. (2022). GitHub – pyserial/pyserial: Python serial port access library. Website. Available (accessed on 26.12.2022): https://github.com/pyserial/pyserial.

Myers, G.J., Sandler, C. & Badgett, T. (2012). The art of software testing. 3rd ed. Hoboken, New Jersey: John Wiley & Sons, Inc.

Nopanen, J. (2021). Testing of Valmet DNA machine monitoring application. Master of Science Thesis. Tampere University.

Postel, J. & Reynolds, J. (1983). RFC 854: Telnet Protocol Specification. Available (accessed on 10.11.2022): https://www.rfc-editor.org/rfc/rfc854.html.

Python. (2022). 1. Extending Python with C or C++ – Python 3.11.1 documentation. Website. Available (accessed on 21.12.2022): https://docs.python.org/3/extending/extending.html.

Robot Framework. (2022a). Robot Framework User Guide. Version 6.0. Website. Available (accessed on 26.10.2022): https://robotframework.org/robotframework/6.0/RobotFrameworkUserGuide.html.

Robot Framework. (2022b). SSHLibrary. Website. Available (accessed on 29.10.2022): https://robotframework.org/SSHLibrary/SSHLibrary.html.

Robot Framework. (2022c). Foundation | Robot Framework. Website. Available (accessed on 26.12.2022): https://robotframework.org/foundation.

Robot Framework. (2022d). RPA | Robot Framework. Website. Available (accessed on 2.12.2022): https://robotframework.org/rpa.

Rodriguez, V. (2019). Anechoic Range Design for Electromagnetic Measurements. Boston: Artech House.

SCPI Consortium. (1999). Standard Commands for Programmable Instruments (SCPI). Version 1999.0. Available (accessed on 11.11.2022): https://www.ivifoundation.org/docs/scpi-99.pdf.

Sommerville, I. (2016). Software Engineering. 10th ed., global edition. Boston: Pearson.

Turunen, M. (2018). Automated UAV Testing. Master of Science Thesis. Tampere University of Technology.

Ylönen, T. & Lonvick, C. ed. (2006). RFC 4251: The Secure Shell (SSH) Protocol Architecture. Available (accessed on 10.11.2022): https://www.rfc-editor.org/rfc/rfc4251.html.