# Application of Software Quality Measures to Bare-metal Firmware for Optical Coherence Tomography

Florian Hinterleitner


MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

embedded systems design

in Hagenberg

im Juni 2022

Advisor:

Langer, Rankl, Zorin

# Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, June 15, 2022

Florian Hinterleitner

# Contents

# Abstract

**ToDo:** vorläufige Fassung, vom Beginn der Masterarbeit

Well established measures of quality for software development allow insightful analysis of the inspected software. These measures, for the bigger part, rely on an underlying operating system, and compilation of the inspected software on the host-system or an equivalent one. To gain similar insight to cross-compiled bare-metal firmware, the application of mentioned quality-measures shall be researched. Obstacles are expected from the necessary cross-compilation and the absence of an underlying operating-system.

This master thesis contains such a firmware project, that forms part of the controlling system of an OCT-systems (optical coherence tomography). For 2-dimensional measurement, OCT-systems require galvanometer-scanners, set up in an x/y-mode. These are highly-dynamical rotational drives for optical application with up to 20° of angle for forward and back-rotation at rates up to 1kHz. These galvanometer-scanners manipulate a light beam from a focused coherent light-source, by deflection via rotating mirrors. Manipulation of the light beam in two dimensions allows OCT-systems to scan areas, instead of single points. The RECENDT GmbH is an Austrian, non-university research institute specialized in non-destructive testing. This company is active in the development of OCT-Systems.

The chosen scanner-models require control signals to create scanning areas, which usually are of rectangular form. Therefore, two synchronous analogue ramp signals, a slow and a fast one, are necessary. The task of this thesis is, to program an existing microcontroller-hardware to form a two-channel arbitrary signal generator, called OCTane. This contains firmware modules to access digital-analogue-converters, trigger-units, for correct timing and synchronisation. Furthermore USB-connectivity in a SCPI-style is necessary, as the control of the unit happens via USB. Various types of code coverage determine the quality level of the resulting firmware.

# Kurzfassung

Die Firma RECENDT GmbH entwickelt und baut OCT-Systeme (optical coherence tomography), für die im Rahmen dieser Masterarbeit ein Teil der Steuerung entworfen werden soll. Zur 2-dimensionalen Messung mit OCT-Systemen kommen Galvanometer-Scanner im X/Y-Betrieb zum Einsatz. Das sind hochdynamische Drehantriebe für optische Anwendungen, die mit einer Rate von rund 1kHz etwa 20° vor- und rückwärts rotieren können. Sie tragen mitrotierende Spiegel um den optischen Pfad in 2 Dimensionen auszulenken und somit flächige Scans zu ermöglichen.

Die ausgewählten Galvanometer-Scanner benötigen Steuersignale zur Erzeugung der Scan-Muster. Typischerweise sind dies zwei synchrone Rampen-Signale, eines schnell, eines langsam. Aufgabe ist es nun, auf bestehender Mikrocontroller-Hardware einen 2-kanaligen arbiträren Signalgenerator, namens OCTane zu programmieren. Dieser soll sowohl Rampen-Signale als auch arbiträr gewählte Signalformen erzeugen können. Dies beinhaltet FW-Module für die Digital-Analog-Wandler, Trigger-Einheit für das Timing sowie Synchronisation der Kanäle. Weiters ist die USB-Kommunikation per SCPI-Protokoll zu programmieren. Die Anbindung an eine übergeordnete Steuerung des OCT-Systems erfolgt per USB.

Etablierte Qualitätsmaße der Software-Entwicklung erlauben tiefe Einblicke in die untersuchte Software. Diese Maße stützen sich, großteils, auf ein unterliegendes Betriebssystem, sowie die Tatsache, daß die untersuchte Software auf dem Zielsystem oder einem Äquivalenten kompiliert wird. Um ähnlich tiefgreifenden Aufschluss über cross-kompilierte bare-metal Firmware zu erhalten, soll die Anwendbarkeit erwähnter Qualitätsmaße untersucht werden. Als hinderlich dabei ist zu erwarten, daß Firmware generell cross-kompiliert werden muss und eventuell kein unterstützendes Betriebssystem enthält.

# Chapter 1

# Introduction

## 1.1 Motivation

Firmware for industrial application requires intensive and thorough testing, that often happens in a behavioural manner. This denotes in testing from outside, via the systems interfaces. Quality measures, that inspect the inner processes of a firmware, allow additional insight to increase reliability and stability. Absence of such measures might lead to firmware containing major unnoticed errors. During execution of the firmware, these errors possibly cause malfunction and even destruction of the host system.

The project 'OCTane', a signal-generator, consists of firmware and hardware to control galvanometer-scanners. Galvanometer-scanners are key elements of OCT-systems. They allow the scanning of areas, instead of only point-wise measurements, by manipulating a laser-beam. Two separate steering-voltages enable this manipulation in x-, and y-direction. An existing microcontroller-board, providing two analogue outputs, hosts the firmware to generate such steering-voltages. USB-connectivity provides control over the unit. Additionally, the firmware must include a HAL (hardware abstraction layer), utilizing several miscellaneous functionalities of the microcontroller. This is an appropriate project to apply quality measures, to ensure adequate reliability of the resulting signal-generator. An insufficient level of quality might even lead to damage of attached components.

## 1.2 Optical Coherence Tomography

Optical coherence tomography (OCT) is an imaging method for the analysis of transparent and semi-transparent materials. It shows similarities to the measurement processes via ultrasound or radar. A light source applies an electromagnetic wave on the sample under test. Relevant properties of the resulting 'echoes' are their times of flight, as well as their intensities. This measured properties contain information of the sample-geometry, including the layer structure and also the maximum penetration depth of the applied wave. This creates a single point 1D-measurement, containing the lateral structure of the material.
A coherent broadband light source generates this electromagnetic wave in the visible, up to the near infrared spectrum. Coherent means, that several wave-bundles of a light

source must have a fixed phase relationship to each other. This is necessary to obtain stable interference patterns.
For the detection of the echoes, conventional photodetectors or cameras do not suffice. On the one hand due to the propagation speed of light, on the other hand due to the low reflected light intensities. Therefore, OCT-systems use interferometry to detect the back reflected light. Fig 1.1 visualizes the structure of an interferometer, suitable for OCT-imaging. In interferometry, a laser beam is split into two waves. One wave travels on an optical reference path of known length, the other to the surface of the sample. The interferometer superimposes the reflections, the returning waves. Depending on the nature of the sample material, this results in constructive or destructive interference. A spectrometer detects these interferences and assembles several of them into an interferogram.
The term 'A-scan' denotes a single point measurement and its depth information about

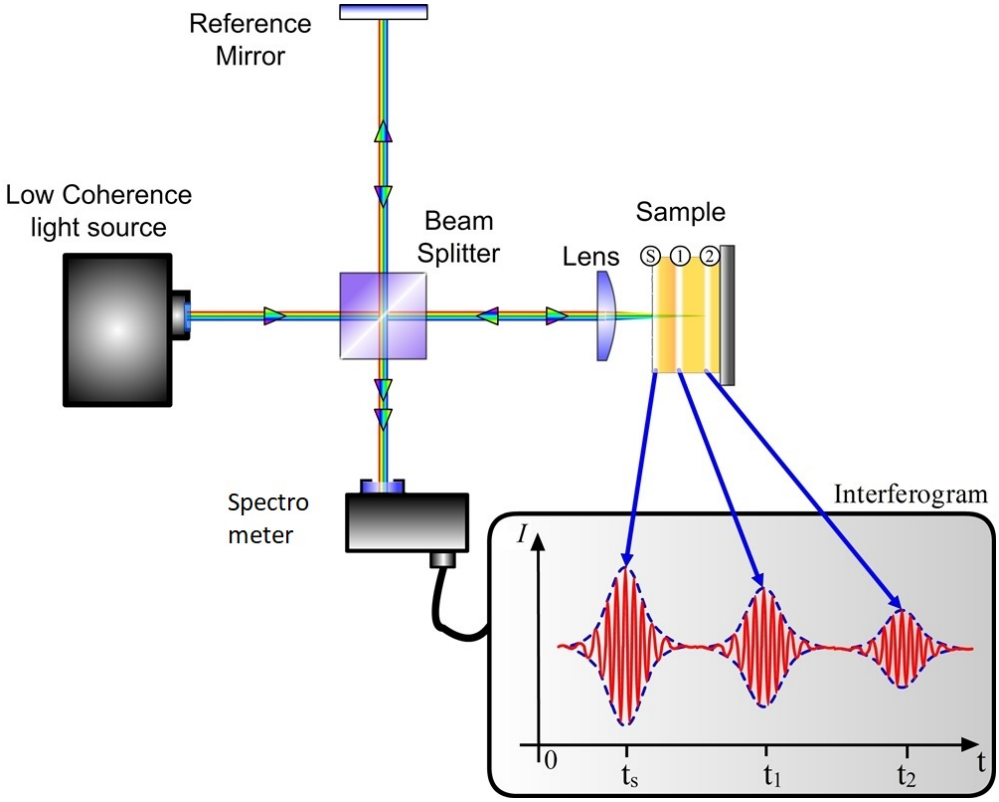


**Figure 1.1:** OCTPrinciplesTDOCT

the material under test. An aggregation of A-scans along a line (x-direction) across the sample material, forms a B-scan. Aggregation of B-scans along a line in the y-direction result in a volume scan, i.e. a spatial, three-dimensional image of the sample material. Relevant parameters of OCT systems are the penetration depth, the axial and lateral measurement range, axial and lateral resolution and the measurement speed. While the penetration depth of ultrasound typically reaches a few centimetres and a resolution in the millimetre range, OCT allows only to look a few millimetres below the surface, but

with micrometer resolutions. Measurable areas, or field-of-view, in ultrasound is in the order of centimetres, with OCT in the order of millimetres. Achievable measurement speed results in up to 100kHz of A-scan rate.
The term 'optical coherence tomography' results on the one hand from the coherent light source. The other two parts of the name, 'tomos' means slice or section, and 'graphein' stand for writing or drawing, and both come from Greek. They reflect that the resulting image is an assembly of individual slices or sectional images.
Rotatably mounted mirrors, one for the x- one for the y- direction, allow for the manipulation of the light beam along the mentioned lines. The faster this rotation is possible, the higher the measurement speed of OCT-images. One widespread technical realization, allowing very fast rotation of the mirrors, is a galvanometer-mirror or -scanner.

## 1.3 Galvanometer-Scanners

Galvanometer scanners (colloquial: galvos) are highly dynamic opto-mechanical components, based on the classic galvanometer according to Hans Christian Oersted: A current-carrying conductor in the proximity of a rotatable, magnetizable object, e.g. a magnetic needle, deflects this object from its initial position. Wiring a high number of windings of the electric conductor around the deflectable object, creating an electrical inductance, a coil. This improves the low sensitivity of the described effect. Placing the coil between a rigidly positioned iron-cylinder inside and a permanent-magnet outside the coil, linearizes the relation between current and deflection angle to a first-order approximation. [1]. If this classic galvanometer carries a mirror as a rotatable object, it can manipulate optical paths, specifically: the beam-path of a point light source, in one space dimension. Feasible for technical applications is, that this manipulation happens nearly linear with the current at the galvo coil. The galvanometer-scanner in use for this master-thesis, already includes power-electronics for the conversion of control signals to the required coil-currents. Therefore, furthermore, only 'control signals' will be discussed, instead of currents and voltages. A combination of two galvanometer scanners in a suitable geometric arrangement, irradiated with a point laser, allows to manipulate this point in two dimensions. Such an arrangement is shown in fig. 1.2. At sufficient speed, at which the laser point is deflected, 2-dimensional contours illuminate the target area, resulting in a 'stationary' image for the human eye. It shall be noted, that only closed geometric figures are possible, as long as the light source itself cannot be turned off. These components are commercially available as laser scanner and used for light effects at music events, art installations and in discotheques. In OCT-systems, on the other hand, galvanometer scanners expand the measurement area from a single point of interest to an area. Scanning mentioned coherent light over an area of the examined sample, allows for three dimensional analysis of the sample. Steering dedicated x- and a y-mirrors with a slow and a fast ramp, respectively, results in a rectangular illumination of the sample. This is the preferred illumination pattern for OCT-systems, as it allows raster scanning of samples.

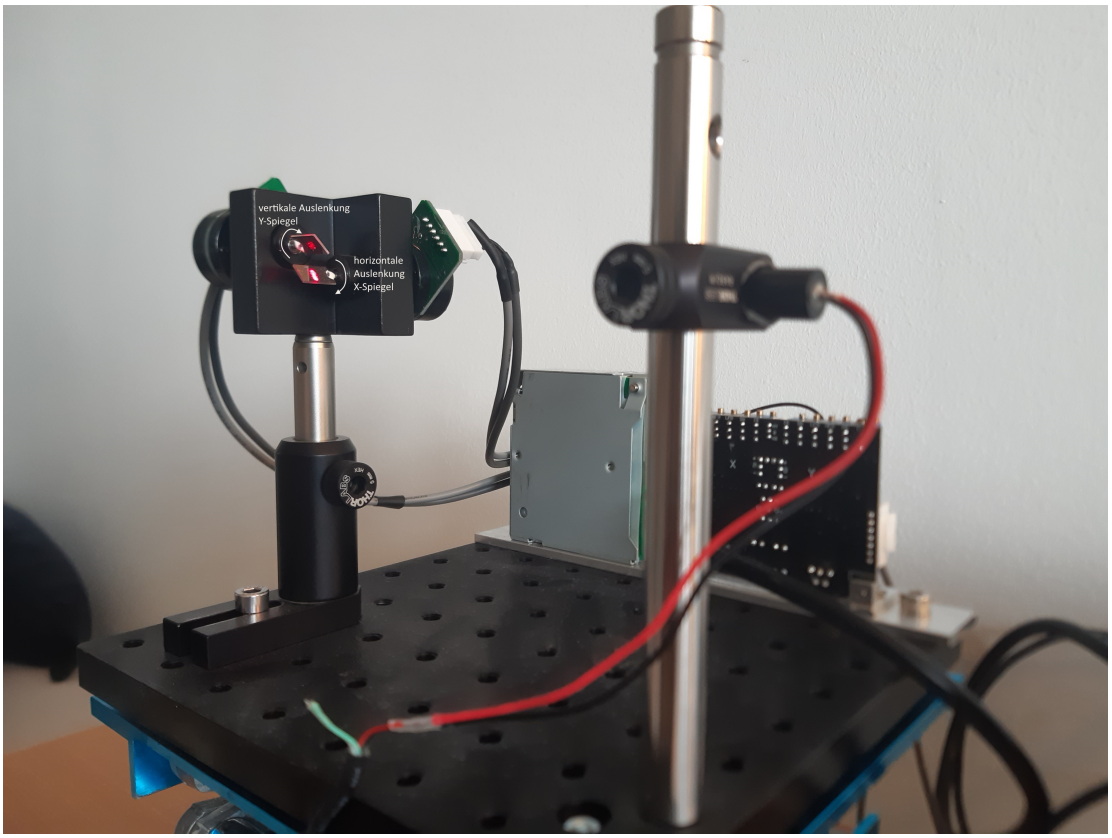


**Figure 1.2:** DetailGalvoOn

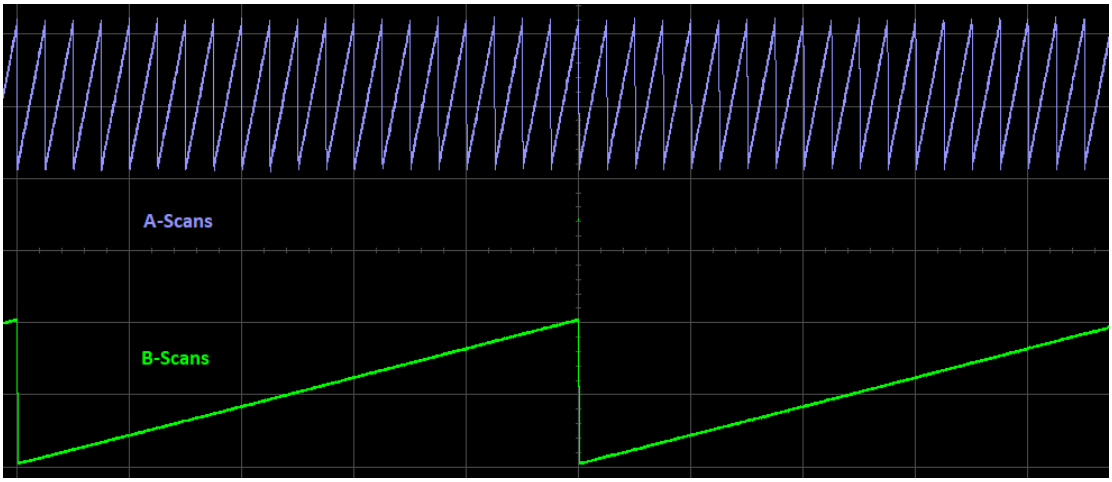


**Figure 1.3:** GalvoRamps01

## 1.4 Control of Galvanometer-Scanners

Commercially available galvanometer scanners usually allow control in the form of analogue voltage inputs with a range of ± 10volts. The angle of the rotated mirror follows

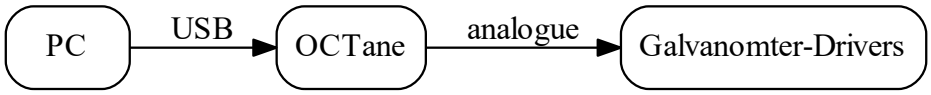


**Figure 1.4:** Integration of an OCTane

that control-voltage in a linear manner for sufficiently low frequencies. Fig. 1.3 shows suitable signal-forms to generate the rectangular scan-patterns, as described in the previous section. An existing microcontroller-board, including 16-bit analogue outputs, USB-connectivity and coaxial trigger outputs hosts a firmware. This firmware utilises the features of the underlying hardware to form an arbitrary signal generator for mentioned ramp-signals. Aside from signal-generation and USB-connectivity, the firmware has to provide access to additional features, such as user-controllable relays, utilisation of watchdog-timer, UART-, I2C- and SPI-ports as well as analogue inputs. The combination of hard- and firmware goes by the name OCTane. Fig. 1.4 demonstrates the integration of an OCTane into an OCT-system.
Quality measures have to accompany the implementation-process, to ensure sufficient reliability and stability. Obtaining the required performance data of a firmware for such quality measures constitutes a certain challenge. Analysis tools for software quality rely on a file-system to gather performance data. Implementing a file-system, though, is not feasible for the present firmware project, as it would only serve for measuring purposes and not contribute to the actual functionality. Analysing the firmware performance on the hosting system would exceed it's computational capabilities. An alternative way to transfer performance data from the host- to the analysing system is necessary. This leads to the scientific question: How to apply measures of software quality to a bare-metal firmware?

# Chapter 2

# Fundamentals - Code Quality and Real-Time

## 2.1 Motivation

The areas of applications for microcontrollers, as well as the complexity of their software continues to grow. Also the demands towards correct and reliable function of those systems increase accordingly. Therefore software development requires significantly more effort than development of the corresponding hardware. As the lifespan of a software surpasses that of the corresponding hardware, these efforts becomes a sensible investment. To obtain the most value from that software-product, it has to fulfil certain quality measures. Neglecting these quality aspects would lead to an unjustifiable amount of work necessary for maintenance, when the product is already in service. Examinations of the dynamics of software development indicate, that, the later in a project lifecycle, changes become necessary, the higher the expenses. The worst case scenario: errors are induced in the implementation, persist unnoticed through testing and verification. Finally, this errors appear as faults in operation at the customer site. Furthermore, if the project suffers from lazy documentation and insufficient structure, identifying these errors and correcting them becomes even more expensive. The increasing complexity of nowadays software further escalates the problem. An even worse phenomenon can arise: Insufficient understanding of a faulty piece of software can lead to introducing even more errors with additional code, that is actually intended to fix a bug. Especially when poor structuring masks hidden dependencies between modules. A general rule of thumb is: The earlier an error originates, for example in the phase of gathering requirements, the more extensive changes are necessary, later on. In other words: The earlier an error is originates and the later it emerges, the more expensive are its consequences. Pursuing a sufficient quality level from the beginning of the project, significantly reduces waste of money, hours and enthusiasm on unnecessary maintenance. It may also substantially increase the customers approval.

The described problems have suitable solutions. Applying the appropriate methods, implemented software can become reliable, easy to change, inexpensive to maintain and allow a more intuitive understanding. Distinguishable into two categories, these methods are either analytical or constructive.

Best practice is to apply a circular combination of constructive and analytical methods. Constructive methods alone assist in preventing errors, but do not guarantee their absence. Analytical methods are capable of demonstrating the absence of errors, but not their prevention. Therefore a large amount of preventable errors might emerge, by exclusively applying analytical methods. The combined use consists of constructive methods during every phase of a development project, and assessing the intermediate results with analytical methods by the end of each phase. This process is called the "principle of integrated quality assurance" [3]. If the intermediate results do not meet the arranged quality criteria, the current state of the project does not reach the next phase, but remains in the extended current phase. This implies, that the current state of the product requires further development, until it fulfils all necessary quality criteria. This phase- and quality driven conduct, supports the development team in detection of errors at an early point and their removal at reasonable effort. Ideally, developers detect and eliminate all errors in the by the end of the same phase, where theses errors originate. This furthermore helps in minimizing the number of errors in a product over several phases.
The described process makes it evident, that testing only a finished product is no sufficient way of ensuring high quality. Already the first intermediate result requires examination for deviations from the quality goals. Upon detection of deviations, measures have to be taken for correction at an early stage. Also an integration of constructive and analytical quality measures significantly improves the development process. While constructive methods are most helpful during the implementation activities of a phase, the corresponding assessment benefits primarily from analytical measures.

The early definition of quality goals is a key factor, as it allows to achieve the intended quality. It constitutes of the specification of the desired quality features, not of defining the software requirements. This has to happen even before the phase of requirement-definition, as the requirements themself are affected by aforementioned quality goals. On the other hand, testing results against quality features is also of vital importance.
The typical approach of developers is, to test a program in its early stages. This is already an informal dynamic test. Inspecting the code after implementation for structural errors is the informal equivalent of a static analysis. Application of these informal methods is very common, while formal processes of testing is much less established. Ideally, testing generates reproducible results, while following well defined procedures.

While hardware quality assurance often results in quantitative results, this is usually not the case for software. But processes exist for both worlds, to ensure systematic development, as well as quality assurance. Developers of systems integrating both hardware and software have to be aware of their differences. Combining the quality measures for soft- and hardware allows to consider interactions between soft- and hardware. This prevents interactions to corrupt quality goals. Developers have to specify and verify quality properties for the complete system. It is insufficient to perform this tasks on separate modules. The correct behaviour of the whole system requires demonstration, as the test results of individual modules, usually, can not be superimposed.

**ToDo:** ... bis hier nach Langer-Review korrigiert

## 2.2 Terminology and Definitions of Terms

To clarify regularly used terms, here are definitions in accordance with either [2] or [3]

### 2.2.1 Quality, Quality Requirements, Quality Features, Quality Measures

- Quality, according to the standard 'DIN 55350 - Concepts for quality management', is defined as: The ability of a unit, or device, to fulfil defined and derived quality requirements.
- Quality requirements describe the aggregate of all single requirements regarding a unit or device.
- Quality features describe concrete properties of a unit or device, relevant for the definition and assessment of the quality. While it does not make quantitative statements, or allowed values of a property, so to say, it very well may have a hierarchical structure: One quality feature, being composed of several detailed sub-features. A differentiation into functional and non-functional features is advised. Also features may have different importance for the customer and the manufacturer. Overarching all these aspects, features may interfere with each other in an opposing manner. As a consequence, maximizing the overall level of quality, regarding every aspect, is not a feasible goal. The sensible goal is to find a trade-off between interfering features, and achieve a sufficient level of quality for all relevant aspects. Typical features, regarding software development include: Safety, security, reliability, dependability, availability, robustness, efficiency regarding memory and runtime, adaptability portability, and testability.
- Quality measures define the quantitative aspects of a quality feature. These are measures, that allow conclusions to be drawn about the characteristics of certain quality features. For example, the MTTF (mean time to failure), is a widespread measure for reliability.


**Figure 2.1:** Causal chain

### 2.2.2 Error, Failure, Fault

- Error, the root cause of a device or unit to fail, may originate from operation outside the specification, or from human mistakes in the design.
- Failure, or defect is the incorrect internal state of a unit, and is the result of an error. It exists either on the hard- or software-side and is the cause of a fault, but not necessarily.
- Fault is the incorrect behaviour of the unit, or it's complete cease of service, observable by the user. It is caused by a failure.
- These definitions are in accordance with [3] and [2] and have causal dependencies, depicted in Fig. 2.1.
- While an error can be classified by its persistence, being permanent or transient, failures and faults are classified more detailed into consistent/inconsistent, permanent/transient and benign/malign, among other categories.

### 2.2.3 Correctness

Correctness is the binary feature of a unit or device, loosely described as 'the absence of failures'. A more specific description would be, that a correct software operates consistent to its specification. This implies, that no conclusion about correctness is possible, without an existing specification.

### 2.2.4 Completeness

Completeness describes, that all functionalities, given in the specification are implemented. This includes normal intended operation, as well as the handling of error-states. It is a necessary, but not a sufficient criterion for correctness.

### 2.2.5 Testability

Testability describes the property of a unit, to include functionality dedicated only to facilitate the verification of that unit. Supporting concepts include the

- Partitioning of the whole unit into modules, that are testable in isolation. These modules should have little to no side-effects with each other.
- A dedicated debug-unit, making the actual state of the unit observable from outside further assists Testability.
- Another concept is, to specify only as much input space as is necessary, resulting in fewer necessary test-cases to ensure a high coverage.

The aggregate of these concepts is called **design-for-testability**. Generally, time-triggered units support testability to a higher degree, than event-triggered systems.

### 2.2.6 Safety and Security

- **Safety** means, that a unit is fit for its intended purpose and provides reliable operation within a specified load- and fault-hypothesis.

- **Security**, though, is the resistance of unit against malicious and deliberate misusage.

### 2.2.7 Reliability

Reliability is a dynamic property, giving the probability, that a unit is operational after given time **t**.

$$\text{Reliability ...} \quad R(t) = e^{-\lambda(t-t_o)}$$

$$\text{failure rate ...} \quad \lambda = \frac{1}{MTTF}$$

An exponential function, decaying from 100% at time = t0, where a unit was known to be operating. $\lambda$ is the failure rate with dimension 'failures/h'

### 2.2.8 Maintainability

Maintainability is the probability, that a system is repaired and functioning again within a given time after a failure. Note that this includes also the time required to detect the error. A quantified measure for it is the mean-time-to-repair (MTTR).

### 2.2.9 Availability

Availability combines reliability and maintainability into a measure, giving the percentage of time, a unit is operational, providing full functionality.

$$\text{Availability ...} \quad A = \frac{MTTF}{MTTF + MTTR}$$

It is apparent, that a low time-to-repair and a high time-to-failure leads to high availability.

### 2.2.10 Robustness

Robustness is actually a property of the specification and requirements. While correctness rates, how far the implemented software complies with the specification, it correct software still might fail in critical situations, that are not covered in the specification. Therefore, to achieve robustness, the specifications have to be examined and ensured, that all critical situations are covered and the expected behaviour of the device under these conditions is defined.

### 2.2.11 Dependability

Dependability finally, is composed of sufficiently fulfilled levels of

- Reliability
- Availability
- Maintainability
- Safety

...assembled into the common acronym **R.A.M.S.**

## 2.3 Function-oriented Testing

This chapter establishes methods to design test-cases, to verify a given piece of software against its specification. The first method, named 'equivalence partitioning', assists in reducing all possible inputs to an examined unit, down to a sufficient set of inputs, while the second method 'state based testing' aims to sufficiently cover code, whose behaviour relies heavily on its own condition and history. Both are best suited in a white-box scenario, that means, that the inner structure of the examined software must be known to the tester, for example in form of the not compiled source-code. Equivalence class partitioning might be applied in a black-box scenario, where only a specification is present, but the consequential flaws of such an approach will become apparent in the following chapter.

### 2.3.1 Equivalence Class Partitioning

This method is applied most beneficial on a unit- or module level testing. The input- and output spaces of various functions might allow an extreme amount of values, testing them all would lead to unacceptable amount of test-cases and would prevent their execution in a feasible time. Then again, many of those possible inputs would take the same paths through the examined module, in other words, excite the module to the same behaviour. Such a sub-set of inputs forms a common class, a so called 'equivalent class', that can ideally by represented by one input and therefore one test-case. A distinction of cases inside a module would form separate paths for the information to take, therefore form different behaviours of the module itself. Each of those distinctions call for a separate equivalence class and their own test-case. The aggregate of test-cases to cover all possible paths through a unit, or to trigger all possible kinds of behaviour of a unit, form a sufficient set for function-oriented testing. This method, that applies the ancient concept of 'divide and conquer', partitions a unit into low levels of complexity, that can be represented by one single equivalent test-case, thus giving it the name 'equivalence partitioning'.

Equivalence classes should initially be derived from the software's specification and can be distinguished into input- and output-classes. While forming a specific output-class it shall be noted, that an according choice of input values has to be defined, presumably exciting the tested unit to the desired or specified output values.
An equivalence class, representing valid input or output values is hence called 'valid equivalence class'. For input or output values, that are specified as invalid, or not specified at all, according 'valid equivalence classes' must be formed as well, to test a units capability in handling those exceptional situations and possibly reveal errors inside a unit. This differentiation in types of test-classes is illustrated in Tab. 2.1.

| | port-wise | |
|---|---|---|
| validity-wise | valid input class | valid output class |
| | invalid input class | invalid output class |

**Table 2.1:** distinguishing equivalence classes

While output classes are much less common in everyday programming, their importance shall not be neglected: Identical inputs might very well result in different outputs, depending on varying side-effects, that have influence on the inspected unit. This has to be accounted in separate equivalence classes for expected outputs.

Following this first steps of partitioning, the resulting classes shall further be separated into sub-classes that take into account distinction of cases within a module, where data might travel several different paths or branches of the source code. This step is only possible in a white-box-scenario, as it requires direct inspection of the source-code. While demanding additional effort, this allows to examine also rather hidden corners of the source-code, that otherwise might go unnoticed and possibly mask hidden errors.

Some examples demonstrate the correct application of the described method:

- valid/invalid input classes:
  input is specified as a floating point number between 0 and 20 volts
  → valid class: $0.00 \leq$ 'test-value' $\leq 20.00$
  → invalid class: $0.00 > test - value$ and
  → invalid class: <test-value> $> 20.00$

- output class:
  output is specified for given input filenames as: 0 if file exists, -1 if file does not exist.
  → valid class: Filename of an existing file
  → invalid class: Filename of an inexistent file
  → invalid class: String with a malformed file-path

- dedicated allowed values:
  addressed module can be chosen from TriggerA, TriggerB, or TriggerC.
  → valid class: TriggerB
  → invalid class: TriggerK
  → invalid class: Trucker

A visual explanation of the first example is given in Fig. 2.2

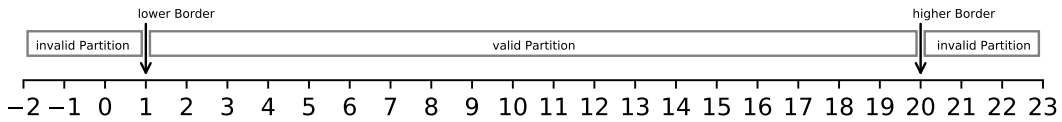


**Figure 2.2:** basic equivalence partitions

### 2.3.2 Boundary Value Analysis

Until now it might seem, that test-values can be chosen randomly from gathered classes, which is often a sufficient case. But a closely related method called 'boundary value anal-

ysis' refines the selection of test-values. From a set of integers between 10 and 100, with a known code structure to be free from case distinctions, the representative test-value can truly be chosen randomly as 15, 60, or 78. In more complex numerical structures, like floating-point numbers, overarching '0' and negative numbers as input space, a single value becomes insufficient. It is then advisable to deliberately choose values close to the bounding values of a function and in the given case also values close to the '0'. Further explanation of choosing useful values will be given on a slight variation of the first equivalent classes-example: Assumed is a function specified for floating point input values in the range of ± 10V. The given set, visualized in 2.3 **ToDo:** saubere Grafik

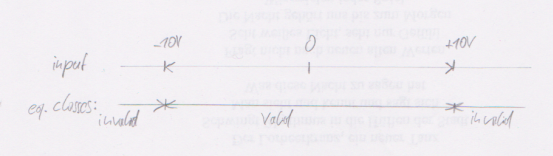


**Figure 2.3:** equivalent class for composite numerical input

wie 2.2 , has obvious bounding values of +10 and -10, giving the first to test-values. Small values deviating from ± 10V are also values to test for. Furthermore, '0' and small values deviating from 0 in positive and negative direction will reveal the units stability, in case, the input is used as a divisor.

→ valid classes: +10V, +9.99V, -10V, -9.99V, 0V, +0.01V, -0.01V

→ invalid classes: +10.01V, -10.01V

Every value has to be applied via a separate testcase, to alleviate which values cause problems, in case of failing tests.
Boundary value analysis and equivalence class partitioning are closely related and often mentioned in unison, nevertheless, their separate description in this chapter is intended to specify their different applications.

## 2.4 Coverage Metrics

Code coverage belongs to the group of dynamic testing techniques, and concerns itself with the structure of software. It is part of the class of control-flow- and structure-oriented dynamic testing techniques. The primary application of this white-box technique lies in testing modules and units, thus 'testing on a small scale'. On the level of integration-tests, coverage has valid applications and is rather common among developers. There is contradicting literature, whether or not code coverage is a suitable technique on a system level of testing (see [4] vs. [3] )
The term 'coverage' refers to the amount of source code, a program executes and therefore 'covers', during testing. Covered code delivers results that require assessment of their correctness against a specification. Uncovered code requires additional test cases ensuring coverage of those areas. Code coverage allows the combination with other test techniques, that describe the generation of those test cases, because code coverage does not specify rules for that.

The following list shows control-flow-oriented techniques, relevant for this thesis:

- Statement coverage
- Decision or branch coverage
- Basic condition coverage
- Condition/decision coverage
- Boundary-interior coverage, structured path test
- Modified condition/decision coverage (MCDC)
- Modified boundary-interior test
- Path coverage
- Multiple condition coverage
- Mutation analysis

Fig. 2.4 demonstrates the implicative relations of the coverage types in aforementioned list. For example, complete decision coverage implies full statement. Multi-Condition and path coverage on the other hand have no relevant relation to each other and both constitute the strongest coverage metric in their own aspect.

Statement coverage is the most basic metric and only requires to execute every line of code, existing in the code under test. A sufficient amount of test-cases should always achieve 100% statement coverage, otherwise 'unreachable' code sections indicate design-flaws.

Decision or branch coverage indicates, if a program executes all branches. It subsumes statement coverage and extends the concept, in that, every decision in the source code must evaluate to true and false. A decision is possibly composed of multiple elementary conditions.

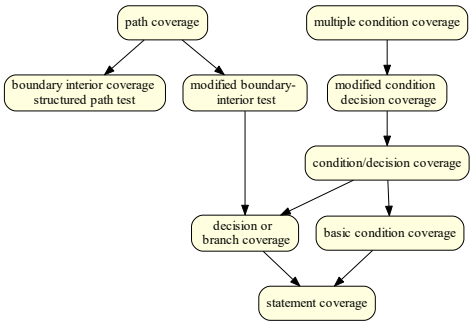


**Figure 2.4:** Subsumption Hierarchy

Basic condition coverage indicates, if all elementary conditions evaluate to both possibilities. It requires, that in compound decisions, every elementary condition separately results to true and false.

Condition/decision coverage consists of all requirements from basic condition coverage and decision coverage combined. It is a relevant metric in case of complete evaluation of compound decisions, in contrast to short-circuit evaluation. It subsumes statement, decision and basic condition coverage.

The structured path test defines criteria for sufficient testing of loops, while boundary-interior coverage is the corresponding metric. It requires separate test-cases that enter a loop, with (interior) and without (boundary) incrementing it. Because of these special requirements, it cannot be subsumed by previous metrics. Therefore, it resides in a separate branch of the hierarchy-tree in fig. 2.4.

Modified condition/decision coverage (MCDC) is a refinement of condition/decision coverage. It requires every elementary condition to result in true and false. Additionally, it demonstrates the impact on the compound decision, separately for every elementary condition.

The modified boundary-interior test is similar to the structured path test, but requires separate test-cases for every loop, especially nested loops. Paths, that execute a loop and vary in the execution of nested loops, count as one path. Paths that only vary outside the examined loop do not require separate consideration. Finally it explicitly requires complete branch coverage, regardless of the previous rules.

Path coverage refers to the execution of all possible paths, present in the code under test. It is the strongest requirement towards evaluation of program-flow and subsumes boundary-interior coverage as well as the modified boundary-interior test.

Multiple condition coverage extends basic condition coverage, by imposing the strongest

requirement towards compound decisions: Every possible result of one elementary condition, true or false, has to be combined with every possible result of every other elementary condition.

Mutation analysis deliberately manipulates the code under test to demonstrate, whether the present test-cases detect these changes. It is an auxiliary method to assess capabilities of test-cases, though not a type of coverage itself.

The major similarity of all coverage types, is the intention to indicate the completeness of the code under test, albeit with regards to different aspects. Also, they all strive to execute all possible variants of decisions, branches and parameter spaces. This aims to reveal all possible errors, as software under execution will either show the intended or divergent behaviour. In the latter case, measures to eradicate found errors during development are necessary. A lack of coverage indicates insufficient testing, while not every metric allows to achieve 100% coverage in feasible testing-time. Adding test-cases is a feasible way to improve coverage. Branches, decisions or statements, that no sensible test-case can reach, suggest a flaw in the software design.

The most prominent limitation of any type of coverage lies in the aspect, that it can exclusively test, what is implemented: Coverage can not reveal functionalities, that are part of the Specification, but do not have an Implementation.

**ToDo:** Probekapitel geht bis hier

Chapter 3

# Requirements

# Chapter 4

# Implementation

### 4.0.1 Concept

FSM

Trigger-Diagramme

Timer usage

- 3 capture compare timers for signal-generation
- 1 timer basic for kex debouncing
- 1 timer basic for reading timeouts
- 1 timer basic for flashing LEDs

Debug-Unit

A debug-unit, offering eight digital outputs via set.. and rst.. - functions, was established. Fig. 4.1 shows a 'ladder' setting and resetting all debug-Pins upon initialization of the module.

## 4.1 Hardware

### 4.1.1 STM32F4

### 4.1.2 Wandler, Level-Shifter, HighSider

## 4.2 Software tools

### 4.2.1 CubeIDE

### 4.2.2 Gcov

### 4.2.3 Valgrind

### 4.2.4 Wavedrom

### 4.2.5 WireShark/USBPcap

### 4.2.6 Gitlab Runner

### 4.2.7 HIL-Setup

## 4.3 Firmware-Requirements

### 4.3.1 FW-REQ

### 4.3.2 Load-Hypothesis, Fault-Hypothesis

### 4.3.3 Traceability-Matrix

Linking Requirements by there tags, to the SW-modules, where they are fulfilled


**Figure 4.1:** dbgUnitLogic

# Chapter 5

# Measurements

## 5.1 Oszi, Debug-Unit und Opto-Detektoren

# Chapter 6

# Results

6.1 Test-Res

6.2 Coverages

6.3 Review Remarks

6.4 Gavlo-Performance

6.5 Project-status

6.6 ...

Chapter 7

# Conclusion

# Appendix A

# Supplementary Materials

List of supplementary data submitted to the degree-granting institution for archival storage (in ZIP format).

## A.1 PDF Files

Path: /

| | |
|---|---|
| thesis.pdf . . . . . | Master/Bachelor thesis (complete document) |

## A.2 Media Files

Path: /media

| | |
|---|---|
| *.ai, *.pdf . . . . . | Adobe Illustrator files |
| *.jpg, *.png . . . . . | raster images |
| *.mp3 . . . . . | audio files |
| *.mp4 . . . . . | video files |

## A.3 Abbreviations and Acronyms

| | |
|---|---|
| uC | MicroController |
| FW | Firmware, the Software, running on the uC |
| OCT | Optical Coherence Tomography |
| SW | Software, the Software, running on the OCT-System |
| FSM | Finite State Machine |
| CRC | Cyclic Redundancy Check |
| IO | Input-Output, bidirectional Communcation Lines |
| USB | Universal Serial Bus |
| VCP | Virtual Com Port, a serial connection via USB |
| USB | Universal Serial Bus |
| SCPI | Standard Commands for Programmable Instruments, as defined by IEEE 488.2 |
| LUT | Look-up-table |
| IRQ | Interrupt request |
| ISR | Interrupt-service-routine, a function within the FW, that is called by an IRQ |
| HW | Hardware, the entirety of uC, the PCB and peripherals |
| SLD | Super luminiscence Diode |
| AIM | Aiming Laser |
| CAM | Camera |
| LED | Light emitting diode |
| LSB | Least significant bit |

**Table A.1:** Abbreviations

# References

## Literature


[1] Joseph F. Keithley. *The Story of Electrical and Magnetic Measurements From 500 BC to the 1940s*. New York, USA: IEEE Press, 1999 (cit. on p. 3).

[2] Hermann Kopetz. *Real-time systems - design principles for distributed embedded applications*. Vol. 395. The Kluwer international series in engineering and computer science. Kluwer, 1997 (cit. on pp. 8, 9).

[3] Peter Liggesmeyer. *Software-Qualität - testen, analysieren und verifizieren von Software*. Spektrum Akadem. Verl., 2002 (cit. on pp. 7–9, 14).

[4] Jeff Tian. *Software quality engineering - testing, quality assurance, and quantifiable improvement*. Wiley, 2005 (cit. on p. 14).