OCTane - Applying software quality-measures to bare-metal firmware

Florian Hinterleitner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

embedded systems design

in Hagenberg

im Juni 2022

Advisor:

Langer, Rankl, Zorin

Declaration

Hagenberg, June 15, 2022

Florian Hinterleitner

Contents

Abstract

The company RECENDT GmbH develops and produces OCT-systems (optical coherence tomography), for which, as part of this master thesis, a part of the controlling system will be designed. For 2-dimensional measurement with OCT-systems, galvanometer-scanners are employed in an x/y-mode. These are highly-dynamical rotational drives for optical application with approximately 20° of angle for forward an back-rotation at rates up to 1kHz. Carrying mirrors, that are rotating along, these galvanometer-scanners can manipulate an optical path, typically from a focused coherent light-source, in 2 dimensions and therefore allow to scan whole surfaces with OCT-systems.

The chosen scanner-models require control signals to create scan-patterns, usually rectangles. Therefore, two synchronous ramp signals, a slow and a fast one are necessary. The task at hand is now, 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 the synchronisation. Furthermore USB-connectivity in a SCPI-style fashion has to be implemented, as the control of the unit shall be provided via USB.

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 the inspected software to be compiled 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 to expected from the necessary cross-compilation and the absence of an underlying operating-system.

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

The RECENDT GmbH researches, develops and produces, among other technology areas, measurement systems employing optical coherence tomography. A key element of such OCT-systems are galvanometer-scanners. These allow for investigation of areas, instead of only point-wise measurements, by manipulating a laser-beam. This manipulation again, has to be controlled via two separate steering-voltages, one for manipulation in x- and y-axis. An existing microcontroller-board, providing sufficient precise and fast DACs, is to be programmed, to form the 'OCTane', a signal-generator for mentioned steering-voltages, controllable via USB. The resulting Firmware shall also incorporate a HAL, utilizing several other functionalities, the microcontroller has to offer. Optionally, adapted curves for the steering voltages shall be investigated to allow linear control over galvanometer-scanners in higher frequency ranges.

1.2 Optical coherence tomography

Optical coherence tomography (OCT) is an imaging measurement method for the analysis of transparent and semi-opaque materials and shows similarities to the measurement processes via ultrasound or radar. The sample to be measured is subjected to an electromagnetic wave, the resulting 'echoes' are analysed with regard to their times of flight. From these run-times again, the geometric structure is determined, including the layer structure of the sample and also the maximum penetration depth of the applied wave. This creates measured point with an in-depth resolution. Usually this EM wave is of a coherent broadband light source 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, however, 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 Interferometry is used to determine the depth of penetration of a photon, being reflected from the sample. In interferometry, a laser beam is split into two waves of half the initial optical power. One wave is sent on an optical reference path of known length, the other to the surface of the sample. The reflections, the returning waves are superimposed and, depending on the nature of the sample material, result in constructive or destructive Interference. This interference can be detected using a photodetector[1], or a spectrometer[2] and used for further processing. A single measured point and its depth information about the material under test is called an A-scan. Aggregates a lot of A-scans along a line (X-direction) across the sample material, forms a B-scan and the aggregation of B-scans along a line in the Y direction 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 micron resolutions. Measurable areas, or field-of-view, in ultrasound is in the order of centimetres, with OCT in the order of millimetres[1]. achievable speed al results from A-scan rates up to 100kHz. 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 assembled from individual slices or sectional images. The manipulation of the light beam along the mentioned lines takes place with rotatably mounted mirrors, one for the X one for the Y direction. The faster this rotation is possible, the faster OCT-images can be created. One widespread technical realization, allowing very fast rotation of the mirrors called 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 rotatable, magnetizable object, e.g. a magnetic needle, that will be deflected from its position in the proximity of a current-carrying conductor. The low sensitivity of the effect on the current is improved by a high number of windings of the electric conductor around the deflectable object, creating an electrical Inductance, a coil. The non-linear connection between current and deflection angle can be linearized to a first-order approximation, by placing the coil between a rigidly positioned iron-cylinder inside and a permanent-magnet, which is arranged outside the coil[2]. If this classic Galvanometer is equipped with a mirror as a rotatable object, optical paths, specifically: the beam-path of a point light source, can be manipulated in one space dimension. Feasible for technical applications is, that this manipulation can be controlled reasonably linear by the current or voltage at the galvo coil. With the galvanometer-scanner employed for this master-thesis, power-electronics for the conversion of control signals to the required coil-currents and -voltages were already included. 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. []. At sufficient speed with which the laser point is deflected, 2-dimensional contours can be projected, 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. Applied to OCT-systems, on the other hand, galvanometer scanners are used to expand measurement from a single point of interest. Deflecting mentioned coherent light source of an area of an examined sample, allows for two-dimensional analysis of the sample. If a dedicated x- and a y-galvo are to be steered with a slow and a fast ramp, respectively, this results in a rectangle on the sample, which is the desired 'image shape' of the laser dot for OCT systems. In this way, samples can be scanned in a grid pattern.

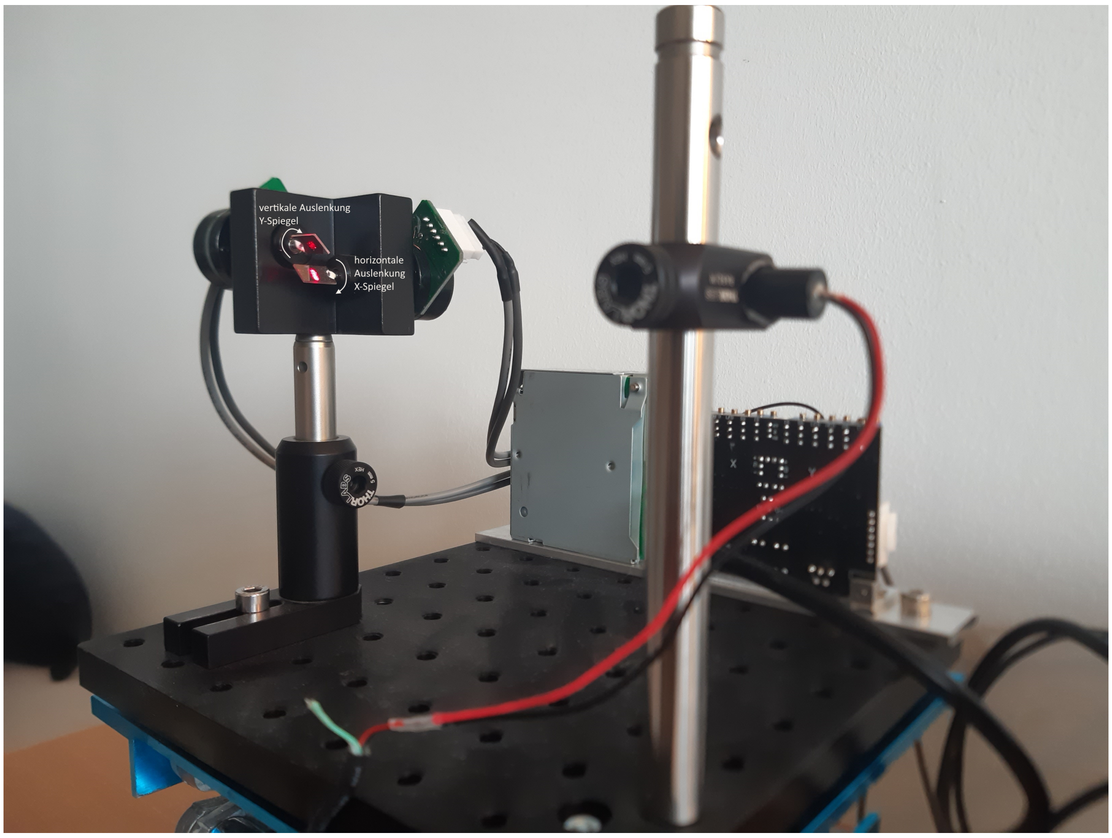

Figure 1.1: DetailGalvoOn

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 that control-voltage in a linear manner for sufficiently low frequencies. The signal-forms to result in the rectangular scan-grids, as described in the previous section, are depicted in Fig. 1.4. To achieve these signals, a microcontroller-board was designed, including 16-bit-DACs, USB-PHY and coaxial Trigger-IOs, among other features. To utilise these features and form an arbitrary signal generator for mentioned ramp-signals, a firmware is necessary. This should be done in a fashion employing quality-assurance

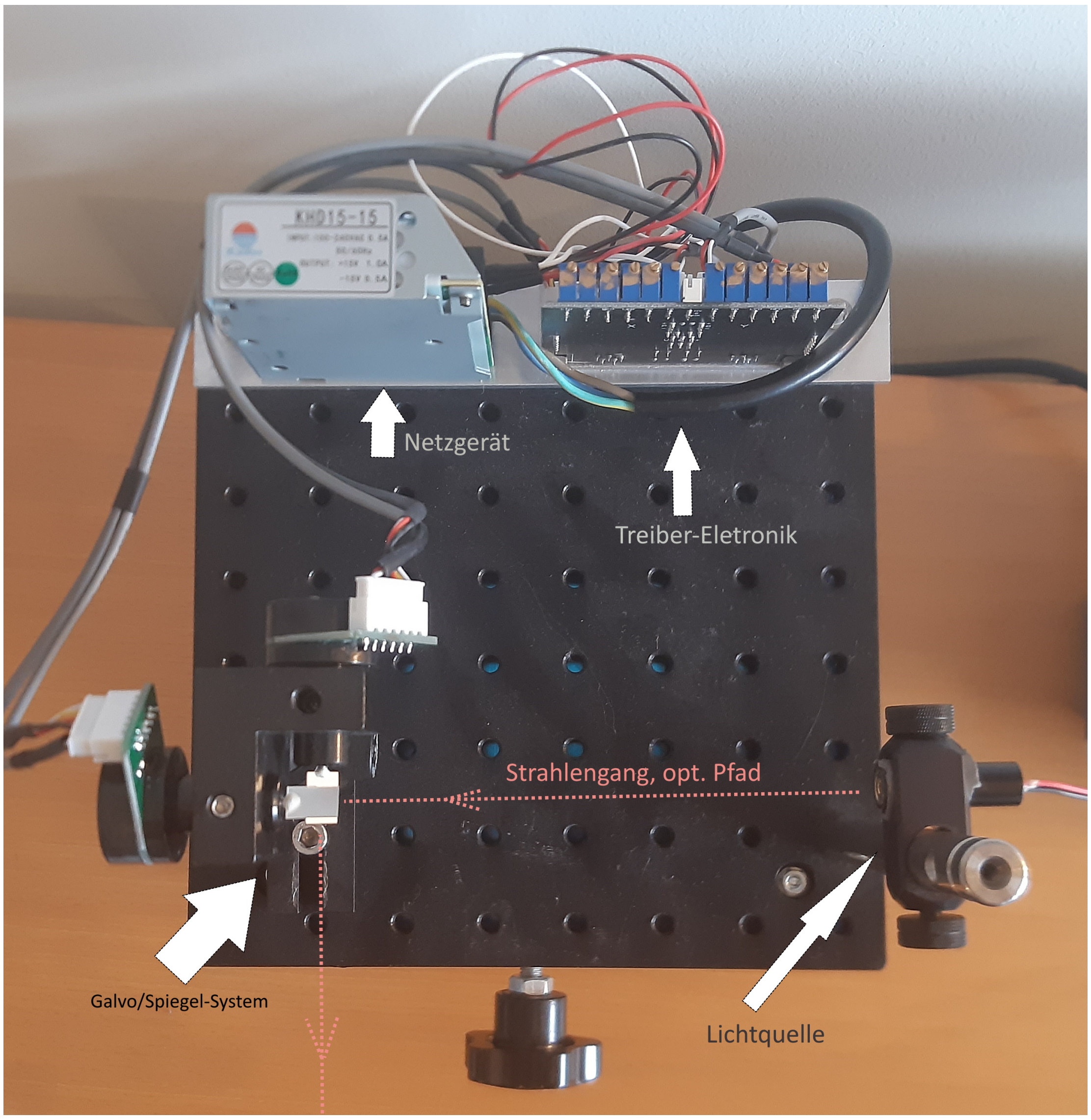

Figure 1.2: DutTop02

during the implementation-process, as well as the verification-phase of the firmware and its supporting hardware. Aside from signal-generation and USB-connectivity, additional features are desirable, 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 will be called OCTane. Fig. 1.3 demonstrates the involvement of an OCTane into an OCT-System.

This leads to the scientific problem at hand:

How can measures of quality be applied to a bare-metal firmware?

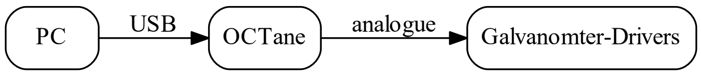

Figure 1.3: Integration of an OCTane

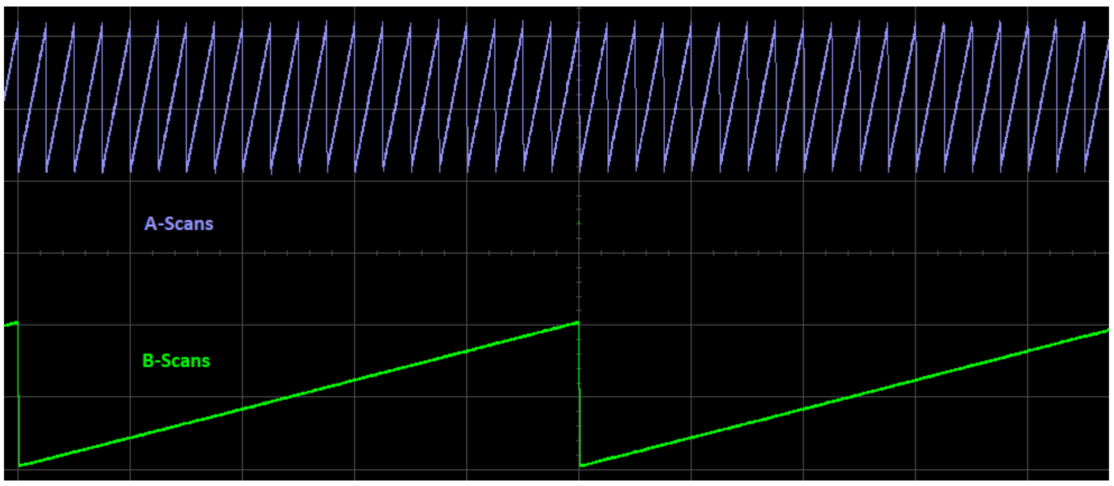

Figure 1.4: GalvoRamps01

Chapter 2

Fundamentals - Code Quality and Real-time

2.1 Motivation

As 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. This causes the efforts, put into software-development, to usually surpass efforts for the corresponding hardware. As the software's lifespan also surpasses that of the hardware, these efforts becomes a sensible investment. On the other hand, 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. Investigating the dynamics of software development efforts leads to the result, that, the later in a project lifecycle, changes become necessary,the higher the expenses. More detailed expressed, 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, insufficient structure, identifying these errors and correcting them becomes consuming in time, cost and resources. This phenomenon is further escalated with the increasing complexity of nowadays software. An even worse phenomenon can arise: Insufficient understanding of a faulty piece of software at hand, 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 induced and the later it is discovered, the more expensive are its consequences. Achieving a sufficient quality level from the beginning of the project, can significantly reduce a waste of money, hours and enthusiasm on unnecessary maintenance and also substantially reduce the customers disapproval.

But these problems have suitable solutions at hand. It is not compulsive to plunge money, hours and employee motivation on unnecessary maintenance. Employing the right 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 employ a circular combination of constructive and analytical methods. Constructive means alone can assist in preventing errors in the intended product, but not guarantee their absence. Analytical means are capable of demonstrating the absence of errors, but not their prevention, so a large amount of preventable errors might emerge, when only analytical means are put to use. The combined use advisable would be, to employ 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' [0]. If these intermediate results do not meet the arranged quality criteria, the current state of the project is not passed on to the next phase, but the current phase has to be extended. This implies, that the current state of the product requires further development, until all necessary criteria are met. 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 all errors induced in a development phase are detected and eliminated by the end of the same phase. This should further help 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 has to be investigated for deviations from the quality goals and measures have to taken for correction at an early stage. Also an integration of constructive and analytical quality measures is required. While constructive methods are advised during the implementation activities of a phase, it should be followed by the corresponding analysis.

A key factor in ensuring the intended quality lies in the early definition of these quality goals. It constitutes not of defining the requirements, but the specification of the desired quality features. 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 central importance. The typical approach of every developer is, to call a written program with a few sets of inputs and observe the program for expected, or divergent behaviour. This already constitutes for an informal dynamic test. Inspecting the code after, implementing it, for structural errors is the informal equivalent of a static analysis. As these informal methods are widespread among programmers, employing formal processes of testing is rather disregarded among programmers, as well as the knowledge about their effectiveness. Ideally, testing is aimed at generating reproducible results, while following well defined procedures.

While hardware quality assurance often results in quantitative results, same is not the case for software, at least not to the same extent as for hardware. 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. Also, strictly separating the quality measures for software and hardware is not an advisable way to go. The quality properties have to be specified and verified for the complete system and not just its separate modules. The test results of individual modules, usually, can not be superimposed, but the correct behaviour of the whole system has to be demonstrated. Therefore, the deviating aspects of hardware and software quality assurance have to be regarded.

2.2 Terminology and definitions of terms

To clarify regularly used terms, here are definitions in accordance with either [0] or [0]

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 describes 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.

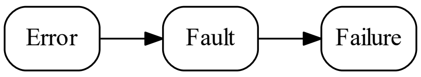

Figure 2.1: Causal chain

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 [0] and [0] 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 said 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. λ 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 employed 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, depending of what was specified. 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.

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

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

Table 2.1: distinguishing equivalence classes

data might travel several different paths or branches of the source code. This step is only possible in a white-box-scenario, as it affords 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
  $\rightarrow$ valid class: $0.00 \leq$ 'test-value' ≤ 20.00
  $\rightarrow$ invalid class: $0.00 > test - value$ and
  $\rightarrow$ 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.
  $\rightarrow$ valid class: Filename of an existing file
  $\rightarrow$ invalid class: Filename of an inexistent file
  $\rightarrow$ invalid class: String with a malformed file-path

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

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

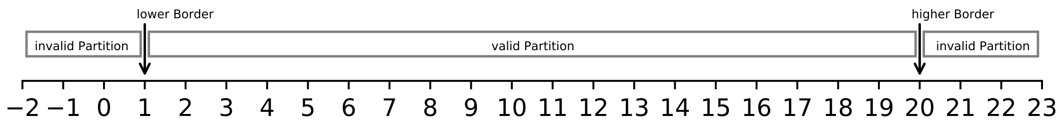

Figure 2.2: basic equivalence partitions

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 analysis' 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, has obvious bounding

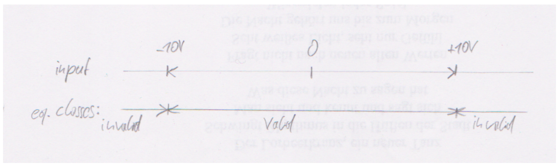

Figure 2.3: equivalent class for composite numerical input

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. **To-Do:** saubere Grafik wie 2.2

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

→ invalid classes: +10.01V, -10.01V

Every value has to 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.

2.3.3 State based testing

2.4 Coverage metrics

This chapter describes dynamic testing techniques that assess the completeness of the test based on the coverage of the software source code. Coverage describes the amount of source code, that is executed during testing of the examined software. Therefore, they are referred to as structure-oriented testing techniques. The techniques described are based on the control structure or the control flow of the software to be tested. For this reason one speaks of control flow-oriented, structure-oriented test techniques. This group of test techniques is of great practical importance. This applies in particular to their use in module testing, the so-called 'testing on a small scale'. The group of control flow oriented testing techniques is well supported by test tool vendors. In addition, there are accepted minimum criteria in the field of control flow-oriented tests that should be considered in terms of an adequate test. A as minimal, the so-called branch coverage test is a necessary acceptance test procedure. In particularly critical areas of application, relevant standards require more extensive tests, for example a so-called condition coverage test. Certain control-flow-oriented test techniques are of such a fundamental nature that an examination that does not use these techniques, particularly in the module test, must be rated as insufficient.

2.4.1 Properties and goals

Because control flow-oriented test techniques belong to the group of structure-oriented test techniques, they have their respective advantages and disadvantages. Test completeness is assessed based on coverage of the control structure or control flow. A corresponding specification is required for the assessment of the expenditure. Like all structure-oriented test techniques, control flow-oriented test techniques do not define any rules for the generation of test cases. It is only important that the test cases cause corresponding coverage in the structure. This degree of freedom in test case generation is extremely important, as it allows other test case generation techniques to be combined with structure-oriented testing techniques.

The most important area of application of the control flow-oriented test techniques is the unit test. Control flow-oriented test techniques can still have a certain importance in integration testing, while they are not used in system testing. Control flow oriented testing techniques look at the structure of the code. In particular, aspects of the processing logic that are represented in the software as instructions, branches, conditions, loops or paths are considered. The disadvantage is that this approach is blind to omission errors in the software. Unrealized but specified functions are only recognized by chance, there is no code to test for these functions.

The test basis is the so-called control flow graph The control flow graph can be created for any program implemented in an imperative programming language. For this reason, the control flow-oriented test techniques can be applied equally to all programs created in an imperative language via the representation of a software module to be tested as a control flow graph. Tools to support control flow-oriented testing techniques usually generate control flow graphs, as portrayed in fig. 2.4, and use them to represent the test results.

To-Do: Günther: bitte bis hier durchlesen

2.4.2 statement coverage test

The statement coverage test is the simplest control flow oriented test method. It is also referred to as the C_0 test for short. The aim of statement coverage is to execute all statements of the program to be tested at least once, i.e. to cover all nodes of the control flow graph. The degree of statement coverage achieved is defined as a test measure. It is the ratio of the executed instructions to the total number of instructions in the test

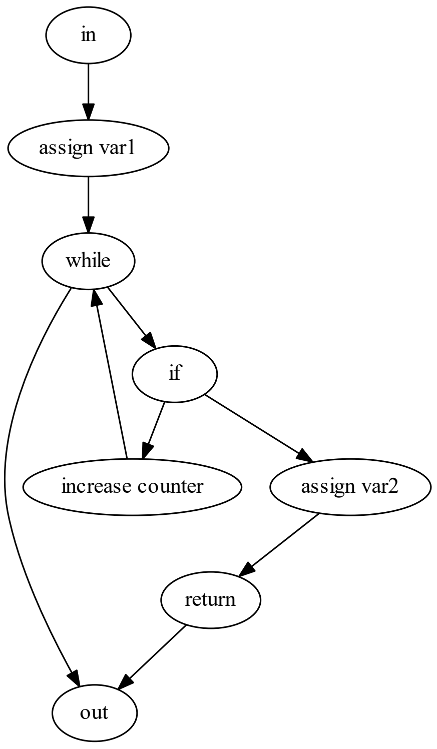

Figure 2.4: exemplatory control flow graph of a looped function

object.

$$C_0 = \frac{\text{number of executed statements}}{\text{number of statements}}$$

If all statements of the module to be tested have been executed at least once by the entered test data, then complete statement coverage is achieved. The strategy of executing all statements that a programmer has assembled into a program at least once with test data is immediately obvious. It ensures that there are no instructions in the software under test that have never been executed. On the one hand, the execution of a statement is certainly a necessary criterion that must be met in order to find an error contained in the statement. On the other hand, this is not a sufficient criterion, as the occurrence of the error effect can be dependent to the execution with certain input data.

Even if the misconduct has occurred, it cannot be guaranteed that it will be recognized by the testing person. The erroneous situation must propagate to an externally observable point. The statement coverage test tries to fulfil the necessary criterion of the execution of potentially faulty parts of the software. The execution of all instructions is part of almost all important test procedures, which also take other aspects into account. As an independent test procedure, the statement coverage test takes a subordinate position and is directly supported by only a few tools. The statement coverage test offers the possibility of detecting non-executable statements, so-called 'dead code'. The statement coverage measure can be used to quantify the achieved test coverage. Statement coverage testing is rarely the main function of testing tools. It occurs as a by-product of tools supporting branch coverage testing and other testing tools. The statement coverage test is considered to be too weak a criterion for meaningful test execution. It is of minor practical importance.

The minimum criterion of the control flow-oriented test techniques is the so-called branch coverage test, which contains the statement coverage test. It is therefore not advisable to use the applications coverage test alone as a structure-oriented test completeness criterion.

2.4.3 branch coverage test

Branch coverage testing is a more rigorous testing technique than statement coverage testing. The statement coverage test is fully contained, or: subsumed, in the branch coverage test. The branch coverage test is generally considered to be the minimum criterion in the area of control flow-oriented testing. It is also referred to as the C_1 test for short.

$$C_1 = \frac{\text{number of executed branches}}{\text{number of branches}}$$

The goal of branch coverage testing is to execute all branches of the program under test. This requires running through all edges of the control flow graph. The ratio of the number of executed branches to the number of branches present in the software under test is usually used as a simple measure for branch coverage. The central position of the branch coverage test is particularly illustrated by its position within the test procedure. As a necessary test technique, it is subsumed by most other test procedures **To-Do:** (Bild mit Test-Arten). The branch coverage test forms the largest common subset of all of these testing techniques. The branch coverage test provides the ability to detect non-executable program branches. This is the case when no test data can be generated that causes the execution of a branch that has not yet been run through. Software parts that are run through particularly often can be identified and specifically optimized.
In some cases it may be difficult to run all branches of the program for various reasons. Often unexecuted branches would be executable but the test cases are difficult to generate, for example, because operating system states or file constellations cannot be created with justifiable effort, or the test cases are difficult to derive from the program itself. In principle, branches can also not be executable. This would be a design error that resulted in an unnecessary branch.

As with the statement coverage test, it also applies to the branch coverage test, that software with a coverage rate of 100

The simple branch coverage measure proves to be problematic. Since all branches are weighted equally without considering dependencies between them, there is no linear relationship between the achieved coverage rate and the ratio between the number of test cases required. The uncritical use of the simple coverage measure leads to an overestimation of the test activities carried out, since the coverage rate is always greater than the number of test cases carried out so far in relation to the total number of test cases with complete coverage of all branches.

The reason is that each test case executes a sequence of branches, some of which generally do not belong to just one path, but are part of multiple paths and thus multiple test cases. The test cases performed at the beginning of the test already cover a large number of these branches. A relatively high coverage rate is achieved after a few test runs. The remaining test cases also cover these branches, but only increase the coverage rate by executing the branches not contained in the already tested paths. A relatively large number of test cases are therefore required at the end of the test in order to achieve a small increase in the coverage rate.

One approach to solving this problem considers the execution dependencies between branches as a criterion for influencing the degree of coverage. A branch is not considered if it is executed whenever another branch is executed. The branches that do not have this property are called primitive or essential. Since the execution of the primitive branches ensures the execution of all non-primitive branches due to the dependency described, it is sufficient to use only the primitive branches for the calculation of the coverage measure.

$$C_{primitive} = \frac{\text{number of executed primitive branches}}{\text{number of primitive branches}}$$

$\mathrm{C}_{\mathrm{primitive}}$ gives a more linear relationship between coverage and number of test cases than the simple measure of coverage.

The branch coverage test is the minimum criterion of structure-oriented software testing, a superset of the statement coveage and even prescribed by various safety standards.

A variety of supporting tools for different programming languages exist for the branch coverage test. Such tools usually work instrumentally. The tool analyses the control structure of the software module present in the source code, locates branches and inserts additional instructions (counters) allowing to trace the control flow. If branches are not represented by corresponding statements in the program text, e.g. B. if the optional else construct of a branch is not used, a corresponding statement is generated by the tool. This so-called instrumented version of the DUT is compiled and the generated executable program is executed with test data. The information collected by the added instructions during the test runs can then be evaluated. Additional test cases must be created for branches that are not run through. In addition, the tool can display the degree of branch coverage achieved. Some tools offer the possibility of displaying the branches executed by the test case entered last, which is used in particular to check the control flow, and keep overall statistics to identify branches that have not been executed and program parts that have been run through particularly frequently.

Because of the importance of branch coverage testing as a necessary test criterion and the excellent availability of appropriate testing tools, the use this technique is highly advisable. Whoever, this should be done with tool support, as carrying out structure-oriented tests by hand becomes unnecessary cumbersome.

2.4.4 Condition Coverage Test

The condition coverage test considers the logical structure of decisions of the software under test. Different forms exist, the weakest of which - the simple condition coverage test - does not subsume the statement and branch coverage test! The so-called multiple condition coverage test subsumes branch coverage, but has other weaknesses that will be discussed in more detail below. The minimal multiple condition coverage test and the so-called modified condition/decision coverage test represent a feasible middle ground.

The basic idea of condition coverage tests is the thorough examination of composite decisions of the software under test. For a complete branch coverage test it is sufficient that the evaluation of all decisions delivers the value true and false once. This strategy does not take into account the often complicated structure of the decisions, which often contain nested logical links of partial decisions on several levels.

In the general case, it cannot be guaranteed that the simple condition coverage test subsumes the branch coverage test. Whether the branch coverage test is subsumed determines the way decisions are evaluated. If compound decisions are implemented by the compiler in such a way that they are incompletely evaluated when the software is run, the branch coverage test is included in the simple condition coverage test. This way of evaluating decisions is the standard case. Since in this case composite decisions are only checked until the truth value of the overall decision is known, this form of decision evaluation reduces the execution time.

In a full evaluation of the decisions, the branch coverage test is not included in the simple condition coverage test. Some compilers offer the choice of whether decisions should be fully or partially evaluated. Since branch coverage testing is considered a necessary testing technique, and it is certainly unacceptable if the fulfillment of necessary criteria depends on the compiler used or its settings, the simple condition coverage test must usually be considered insufficient.

2.4.5 Condition/Decision Coverage Test

The condition/decision coverage test guarantees full branch coverage testing in addition to simple condition coverage. It explicitly requires branch coverage to be established in addition to Condition coverage. Since the simple condition coverage test already ensures this in the case of an incomplete evaluation of decisions, this technique is only relevant in the case of a complete evaluation of decisions.

2.4.6 Minimum Multiple Condition Coverage Test

The minimum multiple condition coverage test requires that, in addition to the atomic sub-decisions and the overall decision, all composite sub-decisions are also checked against true and false. This technique subsumes the condition/decision coverage test. Since decisions can be hierarchically structured, it makes sense to take this structure into account when testing. The minimum multiple condition coverage test requires that all decisions - regardless of whether they are atomic or not, are tested against both truth values. This form of condition coverage takes the structure of decisions into account better than the techniques presented above, since all nesting levels of a complicated decision are considered equally.

The requirement to cover all partial decisions makes sense, since it must apply to every atomic or non-atomic decision that it can assume both truth values. If no test cases can be generated for a partial decision that would cause it to assume a truth value that has not yet been tested, then it is invariant. In this case, the decision can be equivalently transformed so that the corresponding partial decision is omitted. The invariant partial decisions correspond to the non-executable branches of the branch coverage test or the non-executable statements of the statement coverage test. They can be removed and indicate a software error.

On the one hand, the minimal multiple condition coverage test subsumes the branch coverage test. He considers the logical structure of decisions and identifies invariant partial decisions. On the other hand, it is only partially able to recognize erroneous logical operators, especially in the case of the complete evaluation of decisions.

2.4.7 Modified Condition/Decision coverage test

The modified condition/decision coverage test requires test cases that demonstrate that each atomic sub-decision can affect the truth value of the overall decision independently of the other sub-decisions. In other words, the technology aims to test the logic of composite decisions as comprehensively as possible with a reasonable test effort. The relationship between the number of atomic sub-decisions of a decision and the number of required test cases is linear. At least n+1 test cases are required to test a decision with n sub-decisions.

The modified condition/decision coverage test subsumes the minimum multiple condition coverage test. In the incomplete evaluation of decisions, the branch coverage test at the object code level and the minimal multiple condition coverage test at the source code level correspond to each other.

Multiple Condition Coverage Test

The multiple condition coverage test requires the testing of all truth value combinations of the atomic sub-decisions. This approach undoubtedly yields a very comprehensive test of composite decisions. In addition, when all possible combinations are taken into account, it is ensured that both truth values are taken into account for the overall decision, regardless of the linking logic of composite decisions. The multiple condition coverage test therefore in any case subsumes the branch coverage test and all other condition coverage testing techniques. The disadvantage is its high testing effort. A decision made up of n sub-decisions always requires 2^n test cases. This is referred to as exponential growth of the test effort. Such an exponential growth in the number of test cases - and thus the test effort - is usually unacceptable, except for a very small amount of sub-divisions.

The condition coverage tests are particularly interesting as testing techniques when there is complicated processing logic that leads to complicated decisions. In terms of the best compromise between performance and testing effort, the minimum multiple condition coverage test and the modified condition/decision coverage test are recommended.

2.4.8 Techniques for testing loops

Loops often cause an extremely high number of program paths, theoretically one for each repetition, if a counting variable is involved. The execution of these paths is not feasible in this case. A solution to this problem is provided through structured path tests and boundary interior coverage methods. These approaches divide paths into 'equivalence classes' and only execute appropriate proxies from those classes of paths. The two techniques are closely related, the boundary interior test is a special case of the structured path test. On the one hand, the techniques in the primary literature are not described with sufficient precision, so that in the case of complicated loop structures it is not entirely clear which requirements have to be met. On the other hand, the aim of these techniques is to define a test criterion for loops that can be carried out with reasonable effort and that complies with certain rules. Thus, one will require that a full branch coverage test be achieved as a constraint, because we are looking for a testing technique that sits between the branch coverage test and the path coverage test. Depending on the underlying definition of the process, there are loop structures for which one of the requirements mentioned is not met. The structured path test as a third method has both reasonable execution effort, as well as sufficient coverage.

Structured path test and boundary interior path test

The number of different paths of a software module can become extremely high in the presence of loops, one for each repetition. However, this does not apply to every type of loop. Counting loops with a constant number of repetitions do not pose a problem in this respect. It makes sense to define constraints for testing those paths that loop through. This is done by combining paths from a certain number of loop runs into classes, which are considered to be sufficiently tested by selecting a test path of the class. The boundary-interior approach according to **To-Do:** Howden75 distinguishes test cases into 'boundary tests', where a loop is entered, but not iterated. The 'interior test' on the other hand enters the loop and also iterates it at least one time. A more general look at Howdens definition reveals the distinction into 'no loop execution', 'one-time loop execution' and 'multiple loop execution'. It shall be pointed out, that pre-checked loops remain untouched by cases of the first type.

Modified boundary interior test technique

Following, a modified boundary interior test technique is suggested:

1. Requirement for test cases neglecting loops: All executable paths that do not enter rejecting loops and do not repeat non-rejecting loops must be tested.
2. Requirements for test cases considering loops:
3. For each rejecting loop, all executable partial paths are to be tested that
    - execute the loop body exactly once and do not differ only in the iteration of nested loops.
    - Paths that only have differences outside of the rejection loop under consideration do not have to be distinguished.

- For each rejecting and each non-rejecting loop, all executable subpaths are to be tested that execute the loop body at least twice and the first two executions of the loop body differ not only in the iteration of nested loops.
- Paths that only have differences outside the loop under consideration or inside the loop from the third pass do not have to be differentiated.

4. The rules mentioned are to be applied separately for each loop.
5. If branches are not tested, corresponding additional test cases are required.

The first requirement ensures a path coverage test with the omission of loops. This can usually be done for reasonably designed modules with a reasonable amount of effort. The requirements listed furthermore essentially correspond to those of the boundary interior test for dealing with loops. By considering a single loop at a time and ignoring the control structures surrounding it and neglecting paths that result from nested loops, a reduction in the number of test cases is achieved. The last requirement ensures branch coverage regardless of the reachability of certain paths.

The modified boundary interior test technique strives to reduce the test effort by considering the software to be tested in a modular way. For the areas outside of loops, test cases are required separately from the increase in complexity caused by loops. Each loop is considered individually, and nested loops are also ignored. However, this does not mean that each loop has to be tested individually. And finally, the branch coverage test is ensured by a corresponding explicit requirement.

When testing loops, also the maximum number of loop iterations shall be tested as well as exceeding the maximum number of iterations. In contrast to a small number of loop iterations, a large number of iterations is not necessarily considered appropriately by the boundary interior test, since interior tests can be aborted after the second loop iteration. Choosing a high number of repetitions is, however, entirely possible in interior tests. This is strongly recommend.

The modified boundary interior test can very easily be generalized to a modified structured path test (with parameter k) by transferring the above requirements from k=2 to values greater than 2.

Since the boundary interior test is a special case of the structured path test, a similar performance can be expected.

Chapter 3

Requirements

Chapter 4

Implementation

4.0.1 Concept

FSM

Trigger-Diagramme

Timer usage

3 Timers necessary, old concept: double frequency and on modulo 2 will be decided if PinSet and DACwrite, or PinClear new conspt: output compare Timer etiher the advanceds from the F4 for TrigB and C with separate ISRs to set and clear OR three GP-Triggers with dedicated output lines, that are set/reset by Timer itself (PSC, ARR and pulse) $\rightarrow$ see schematic wich Trigger has wich line and Graph against according Timers!

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

4.2.6 gitlab

4.2.7 runner

4.2.8 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

4.3.4 TCs

4.3.5 Unit-Tests

4.3.6 Module-Tests

4.3.7 Integration-Tests

4.3.8 Load/Fault Tests

Chapter 5

Measurements

5.1 Oszi, Debug-Unit und Opto-Detektoren

Chapter 6

Results

6.1 Test-Res

6.2 Coverages

6.3 Review-remakrs

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 Online Sources (PDF Captures)

Path: /online-sources

Reliquienschrein-Wikipedia.pdf [0]

References

Literature

[0] 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).

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

Online sources

[0] *Reliquienschrein*. Oct. 20, 2020. URL: https://de.wikipedia.org/wiki/Reliquienschrein (visited on 05/12/2021) (cit. on p. 28).