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

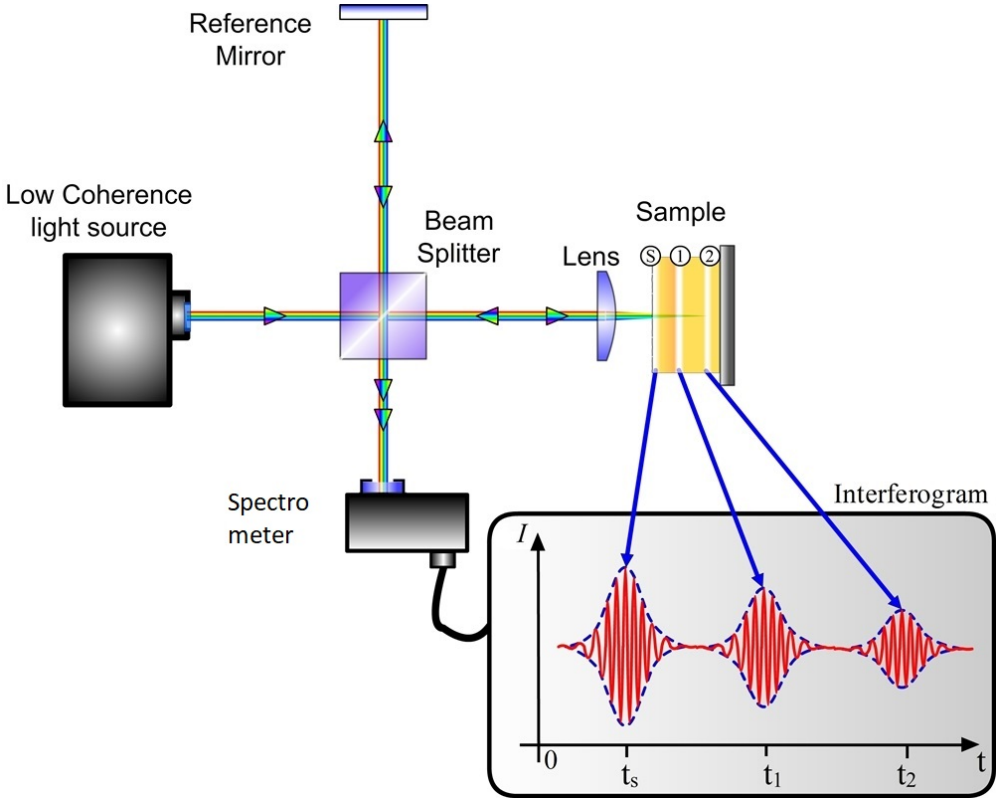


Figure 1.1: OCTPrinciplesTDOCT

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. [12]. 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.

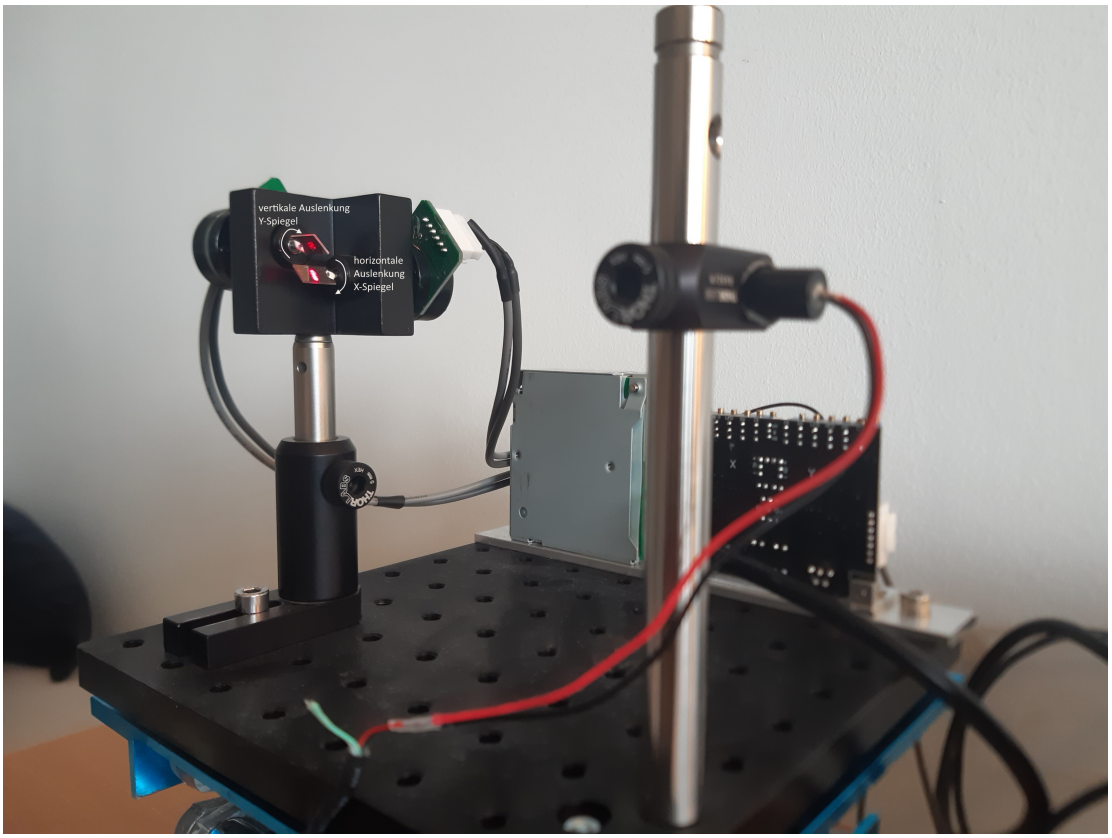


Figure 1.2: DetailGalvoOn

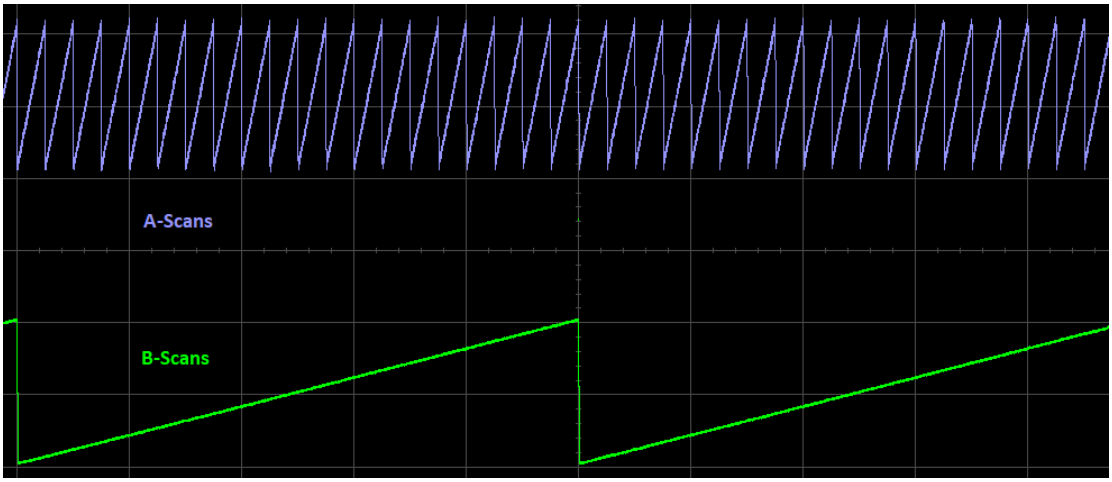


Figure 1.3: GalvoRamps01

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

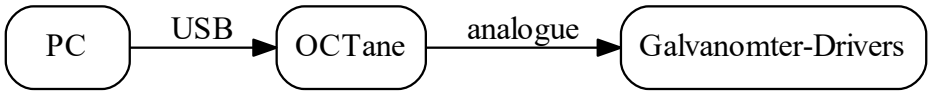


Figure 1.4: Integration of an OCTane

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. [16]

ToDo: Beizer-Zitat, citen: In the significant book Software Testing Techniques [2], which contains the most complete catalog of testing techniques, Beizer stated that “the act of designing tests is one of the most effective bug preventers known,” which extended the definition of testing to error prevention as well as error detection activities. This led

to a classic insight into the power of early testing.

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" [15]. 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 [14] or [15]

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.

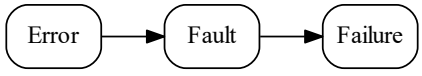


Figure 2.1: Causal chain

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 [15] and [14] 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.

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.

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.

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.

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.

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'

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).

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.

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.

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 an integer number between 1 and 20 volts
  → valid class: $1 \leq$ 'test-value' ≤ 20
  → invalid class: $1 > test - value$ and
  → invalid class: <test-value> > 20

- 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

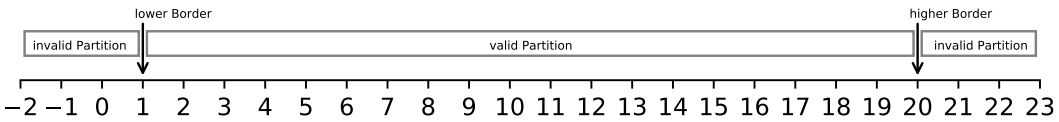


Figure 2.2: basic equivalence partitions

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

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 Fig. 2.3, has obvious

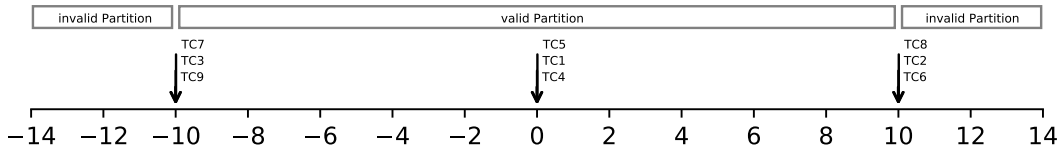


Figure 2.3: equivalent class for floating-point numerical input

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.
This results in the following classes and their according values for test-cases:

$$\begin{aligned} 0.00 &\rightarrow Test - Case1 \\ 10.00 &\rightarrow TC2 \\ -10.00 &\rightarrow TC3 \\ -0.01 &\rightarrow TC4 \\ 0.01 &\rightarrow TC5 \\ 9.99 &\rightarrow TC6 \\ -9.99 &\rightarrow TC7 \\ 10.01 &\rightarrow TC8 \\ -10.01 &\rightarrow TC9 \end{aligned}$$

And their categorization into valid/invalid values:

→ 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 [20] vs. [15])
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
- Branch or decision 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.

Branch or decision 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 or atomic conditions.

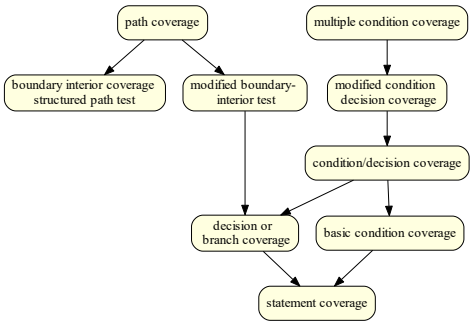


Figure 2.4: Subsumption Hierarchy [15]

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 branch or 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, branch or decision and basic condition coverage.

Boundary-interior coverage is a metric for the sufficient testing of loops, while the structured path test describes, how to design corresponding test-cases. 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 (MC/DC) 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 practicable 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.

Limitations

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.
Furthermore, the higher ranking a type of coverage, the more difficult it becomes, to achieve complete coverage. In this context, a higher level in fig. 2.4 indicates a higher ranking coverage. While a complete statement coverage is a minimal criterion for sufficiently testing a program, same is not the case for the more complex types of coverage. In many cases it is neither possible nor necessary, to test every path or combination of conditions. Nonetheless, In these situations, the according coverage metric is useful in pointing out 'unreachable' states of the program. It is then upon the developer to interpret the coverage report. If it indicates a design flaw, changes to the program are necessary. If the report points out states in the program that are reliably unreachable, neither during testing, nor during execution, the responsible developer may decide these states to be 'not worthy of testing'. Nevertheless, this requires to profoundly argue such a decision, communicate it all affected team-members and documented it. Lastly, reports of uncovered states can be useful as hints, to construct complexer test-cases to also cover these states.

The basis for all control-flow oriented tests is the control-flow graph. Fig. 2.5 shows an example of a simple function, containing a loop and an if-condition. This graphical representation visualizes, in an intuitive way, which paths can be traversed during execution of a delimited piece of code. To maintain clarity in the resulting image, a single graph should only visualize small parts of a program, typically a single function. Larger amounts of code rapidly lead to overwhelmingly complex control-flow graphs. This type of graph is able to represent any program, written in an imperative programming lan-

guage.

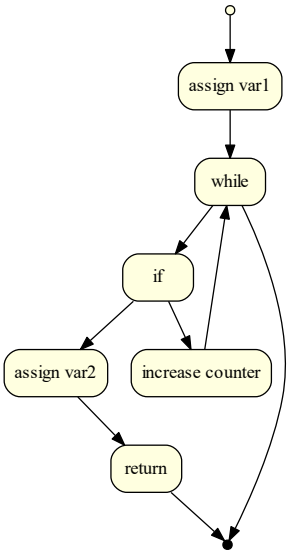


Figure 2.5: exemplary control flow graph of a looped function

The following sections offer more detailed descriptions of coverage metrics.

2.4.1 Statement Coverage Test

The basic control flow oriented test method is the statement coverage, also known by the abbreviation C_0. It aims to execute every statement, or line of code, at least once. Regarding the control flow graph, this means to execute every node of the graph. The definition of this metric is the amount of executed code versus the amount of existing code:

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

Usually, coverage reports state this and similar metrics in percent. 100% or complete statement coverage denotes the execution of every line of code at least once. One obvious benefit lies in the immediate detection of non-executable code sections, also known as 'dead code'. Another benefit lies in revealing possible errors, present in the tested code: Incomplete statement coverage encourages a developer to add test-cases, to execute and test also the remaining areas of code. If their execution triggers an existing error, this error becomes apparent by causing either faulty output or even a crash of the program. And only after an error becomes apparent, a developer can apply counter-measures against it.
This ability of the statement coverage is limited to errors, that are sensitive to execution, regardless of input. This metric only requires to execute code and doe not specify,

which input values to apply. Furthermore, conditional execution has only minor impact on statement coverage.
In general, statement coverage is usually a by-product of other coverage metrics and not determined via an independent test procedure, because many other coverage metrics imply the statement metric. Nevertheless, as basis for most higher-level coverage metrics, statement coverage is still worth studying and understanding. Because of the inherent shortcomings, it is a necessary, but not sufficient metric for adequate software testing.

2.4.2 Branch or Decision Coverage Test

The branch coverage, also called decision coverage, concerns itself with the conditional execution of code. It is a more advanced method than the statement coverage and represents the minimum criterion in control-flow oriented testing. Again, regarding the control flow graph, this means to traverse every edge of the graph.
Obviously, statement coverage does not guarantee the execution of every branch, as nodes may have multiple edges leading to and from them. The example in fig. 2.5 contains an edge to the very right, that would require traversal for branch coverage, but may be ignored by statement coverage. The alternative label 'decision coverage' refers to the inherent requirement, that every decision must at least once result to 'true' or 'false'.
Branch coverage goes by the abbreviation C_1 and measures the amount of executed branches versus the amount of existing branches, again in percent.

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

Based on its definition, branch coverage is able to quantify the execution of branches and draw conclusions upon the resulting categories.

- not executed branches
- seldom executed branches
- often executed branches

On the one hand, not executed branches may turn out to be unnecessary, signify a design error and could be eliminated. On the other hand, it can indicate, that preconditions to reach a branch are difficult to generate via test-cases. The test report may then contain hints towards the design of suitable test-cases. Often executed branches represent candidates for optimization. Speeding up code, that is run very often is, obviously, more beneficial, than optimizing code that executes rather seldom.

Drawbacks of branch coverage contain insufficient assessment of loops, as this metric only requires to execute a loop-body once, regardless of loop-counters. Also not all possible combinations of edges within the control-flow graph have to be traversed. This may hide errors that would emerge only upon traversal of very specific execution paths.

These special paths are not necessarily contained in test-cases, that would otherwise suffice for branch coverage. Furthermore, compound decisions are not reviewed in detail.

2.4.3 Basic Condition Coverage Test

The next more sophisticated metric is the (Basic) Condition coverage. It extends the concept of branch coverage, by closer examination of compound or composite decisions. Fig. 2.6 gives an example of such a decision.

```
if( (strPtr != NULL && !isObsolete(strPtr) || isValid(strPtr) )
{   // ... handle string-data
    return 0;
}else   // error
{   return -1;
}
```

Figure 2.6: compound decision

Condition coverage requires every separate elementary decision to result in 'true' and 'false', while branch coverage demands this only for the complete decision. In particular, condition coverage demands, that

- strPtr != NULL
- isObsolete(strPtr) and
- isValid(strPtr))

have to result to 'true' and 'false', independently.

In the usual case of short-circuit evaluation, the if-condition can possibly leave out execution of second or third arguments at all. In the given case, the subroutine isOsbolete (strPtr) is not guaranteed to undergo execution, would it be not deliberately required by condition coverage. The developer might force the compiler to abstain from short-circuit evaluation and therefore achieve the execution of every single argument. Nevertheless, the thorough way, is to design separate test-cases, aiming for the examination of elementary conditions. This is because during debugging, separate test-cases indicate the source of failure within a compound decision more specific, than just one test-case for the whole decision. Furthermore, complete evaluation would prevent condition coverage, to completely subsume branch coverage. [11] **ToDo:** hmm, wohin damit: [19]

Condition coverage, partially, considers the complexity of compound decisions, which is neglected by decision coverage.

2.4.4 Condition/Decision Coverage Test

Condition/Decision Coverage extends the requirements of the basic condition coverage, by literally demanding the execution of every branch. As described earlier, this is not guaranteed by basic condition coverage, in case, the compiler performs complete evaluation. For reasons, already described, this thesis only considers short-circuit evaluation. In this case, condition/decision coverage and basic condition coverage become identical.

ToDo: elementary conditions -> elementary decisions, (Compound Condition? laut Jorgensen) A common weakness of both these metrics lies in only dissecting compound decisions into its elementary conditions. While this addresses problems arising from short-circuit evaluation, it does not regard inter-dependencies between elementary decisions.

2.4.5 Modified Condition/Decision coverage test MC/DC

The modified condition/decision coverage targets nested logical links between elementary decisions and is abbreviated 'MC/DC'. It achieves this by demanding to demonstrate every single elementary decisions impact on the compound decision, independently. This is not guaranteed by condition/decision coverage. It is a thorough method to examine compound decisions, at reasonable effort. This is due to the linear relation between the complexity of a compound decision and the test effort: According to [6] MC/DC affords N+1 test-cases to comprehensively assess a compound decision containing N elementary decisions.

ToDo: [5] Tab. 2.2 demonstrates the possible combinations of four elementary decisions for the compound condition (A||B)\&\&(C||D) .

Testcase	A	B	C	D	A||B	C||D	(A||B)&&(C||D)
1	0	0	0	0	0	0	0
2	0	0	0	1	0	1	0
3	0	0	1	0	0	1	0
4	0	0	1	1	0	1	0
5	0	1	0	0	1	0	0
6	0	1	0	1	1	1	1
7	0	1	1	0	1	1	1
8	0	1	1	1	1	1	1
9	1	0	0	0	1	0	0
10	1	0	0	1	1	1	1
11	1	0	1	0	1	1	1
12	1	0	1	1	1	1	1
13	1	1	0	0	1	0	0
14	1	1	0	1	1	1	1
15	1	1	1	0	1	1	1
16	1	1	1	1	1	1	1

Table 2.2: test-cases for MC/DC

Following the rule, to demonstrate each elementary decisions impact on the compound condition, only the test-cases number 2, 6, 9, 10 and 11 are necessary. This amounts to five test-cases, which equals the number of elementary decisions+1.
Considering logic dependencies among elementary decisions can reduce the number of test-cases, even more. The decision in fig. 2.7 implies a truth-table similar to tab. 2.2,

but with $2^5 = 32$ entries or lines. Obviously, the left input operand is identical in all five elementary conditions. Therefore, those elementary conditions are mutually exclusive, as, for example, col can not be RED and BLUE at the same time. This leads to the conclusion, that only five test-cases with valid input values and one invalid test-case, are necessary. In this case, six test-cases are sufficient for MC/DC coverage instead of 32. The similarity of this example with a 'switch-case' statement **ToDo:** flappsig formuliert: suggests to apply MC/DC also to those kinds of constructs.
Although, this chapter emphasis on the design of suitable test-cases, MC/DC does not

```
if(col == BLACK || col == WHITE ||  col == RED ||  col == GREEN ||  col == BLUE)
```

Figure 2.7: compDeciMutualEx

specify this. Only the outcome of MC/DC is defined, but free tools are not available, so emphasis on test cases to achive MC/DC in Blindflug are goiven The advantage of the MC/DC, evidently, lies in the thorough test of compound conditions with feasible effort. For a condition, composed of four elementary decisions, only five of sixteen possible input combinations require test-cases. It implies all aforementioned coverage metrics.
The described advantages come at the price of higher efforts in identifying suitable stimuli as test-cases. While the test-cases to achieve other coverage metrics are often self-evident from code or fragmentary coverage reports, filtering out the necessary input combinations for MC/DC often require examination of truth-tables for every single compound decision.

ToDo: evtl auf Hayhurst verwesien **ToDo:** statement cov -> statement or line coverage

ToDo: warum keine C-Notation mehr ab C2 ? weil nicht mehr eindeutig, zB C2c = structPath, anderswo: C2 = branch/condition coverage

2.4.6 Multiple Condition Coverage Test

Multiple condition coverage is the superior method with respect to conditional execution. It demands to test all possible truth value combinations of all single elementary conditions within a compound decision. This constitutes the most thorough examination of conditions within a program, because, literally, all possible sub-decisions are included. Inherently, this also contains compound conditions to result to 'true' and 'false'.
Multiple condition coverage shows no regard, whether, short-circuit evaluation takes place or not. Furthermore, it does not consider logical links, like mutual exclusions, between elementary conditions. Therefore, multiple condition coverage subsumes all aforementioned coverage metrics and constitutes a brute-force approach to the examination of conditional execution.
ToDo: verweis auf die 16-truth-table With those strict requirements, comes an exponential effort for actual testing of a program: A compound decision composed of N elementary conditions calls for 2^N individual test-cases. For small values of N, this is acceptable, but rapidly becomes unfeasible for complex decisions with a large number of elementary conditions. Furthermore, elementary conditions, that mutually exclude each other, do not allow for meaningful sets of input values. Fig. 2.7 is an expressive

example for that, as the input variable 'col' can not be two different colours at the same time. But this would be an actual requirement of multiple condition coverage.
The strict demands of multiple condition coverage, resulting in excessive efforts for testing, renders this metric impractical for reasonable examination of software. Furthermore, with MC/DC, there is a practical approach at hand, that allows nearly the same insight into the code under test, as multiple condition coverage. **ToDo:** Formulierungen?

Conclusio to cond/deci coverages Ligges-Zitat 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.

2.4.7 Techniques for Testing Loops

The methods described until now, are not really feasible for programs containing loops: Every single execution of a loop (constitutes a separate path/ demands a separate test-case), according to requirements imposed by, for example, branch coverage. Neither is this practical nor worthwhile for high loop-counts: Though the number of test-cases increases linear with the loop-count, this number can be extremely high in terms of test-case-count. On the other hand, if no structural differences happen inside a loop-body, no additional insight is gained from testing the same body with every single count-value. Only count-values with consequences to the internal control flow of a loop-body justify separate test cases. Examples for that would be if-statement nested in a loop.
Solutions at hand come in the form of the following test-methods specifically targeting looped programs.

- Boundary Interior Path Test and
- Structured Path Test
- Modified Boundary Interior Test Technique

ToDo: Formulierungen?

Boundary Interior Path Test

This method inspects the different possible paths through a given program and partitions them into classes, each class affording a separate test-case. Paths form a common class, if they have the same control-flow and differ, for example in numerous different loop-counts. Classes, then again, have different control-flow paths, therefore requiring separate test-cases. These differing control-flow paths, for example, might stem from different loop-counts, which have varying impact on internal control-structures on a loop. This leaves out numerous paths with no, or insignificant differences and groups them into one common class with one representative test-case. [8] The boundary-interior path test incorporates ideas from equivalence class partitioning, to identify separate classes of input values, in this case: loop-counts. It distinguishes loop-counts by varying effects on the internal control-flow of a loop. Consequentially, only one representative value of the loop-count, per class, has to be tested.
These attributes distinguish between different classes:

- a loop is not entered, only code before and after it is executed.

- a loop is entered, but only executed once. (boundary)
- a loop is entered, executed at least twice. (interior)
- a loop is entered and different paths through the loop are possible (e.g.: nested loops, if-statements). This again, needs differentiation regarding boundary- or interior- classes. **ToDo:** ahem, evtl gehort nested nur zu modified B/I test

ToDo: baundary/interior, struc path: wrum heissens net coverage ?

```
preProccess();

while( ptr != NULL )
{   ptr++;
    processData(ptr);
}

// post processing
```

Figure 2.8: example code, boundary interior path test

The fragment of code in fig. 2.8 requires one test-case where the loop body is not executed, therefore the ptr-variable has to be NULL, from the outset. Boundary interior path test requires also one test-case, that executes the loop-body only once, constituting the boundary-test-case. The ptr-variable has to become NULL after one increment. A third test-case, that executes the loop-body at least twice, is necessary. It constitutes the interior-test-case and requires the ptr-variable to result to NULL, earliest after two increments.

Structured Path Test

The structured path test is a generalization of the boundary interior path test and subsumes it. On the other hand it is a special case, a subset of path coverage. The goal is, to test many different paths, but exclude blocks from further test-cases after a limited number of iterations. This number is a variable by the denomination 'k'. The structured path test attempts to loosen the rigorous requirements of path coverage, resulting in feasible testing effort, while maintaining a valuable depth of analysis of a given piece of code. By choosing $k = 2$, the structured path test becomes identical with the boundary interior test. [7]

Modified Boundary Interior Test Technique

The introduction of a third, closely related method, the modified boundary interior test, addresses weaknesses of the boundary interior and the structured path test. The most prominent weaknesses are the possibility of not executable paths within loops, rapid

increase of test-cases caused by nested control structures and no testing of higher loop-counts. Implicitly, both methods do not subsume any other coverage metric from fig. **??** The modified requirements, distinguishing path classes, consist of:

- classes, that do not enter a loop, but only only execute code before and after it. These classes strive for path coverage surrounding the loop.
- classes, that enter a loop and execute it once, and do not exclusively differ in the execution of nested loops. Paths executing a loop, that only differ in code surrounding that loop, do not call for a separate class.
- classes, that enter a loop and execute it at least twice, and do not exclusively differ in the execution of nested loops. Paths executing a loop at least twice, that only differ in code surrounding that loop, do not call for a separate class.
- These rules have to be applied separately for every loop, in particular, also for every nested loop.
- classes, that execute branches, if they are not already covered by aforementioned classes.

The first requirement tests the surroundings of loops, while neglecting loops themselves, resulting in rather little effort. The 2nd and 3rd requirement are slight adaptions of the boundary-interior test, incorporating the possibility of nested loops. The last requirement, deliberately, aims for a complete test to subsume decision coverage. Additionally, separate test-cases should cover the maximum number of loop-iterations, as well as exceed them, to demonstrate maximum computation times and detect errors from overflows or exceeding limited resources.

Fig. 2.9 contains exemplary code to demonstrate the application of these requirements.

Classes, and therefore separate test-cases, are necessary for both possible results of the if-statement, as well as one that executes the first while-loop once and one that iterates it twice. The second while loop requires a class to test single execution of its body and one for iterating of the loop body twice. As the inner loop has a fixed count of iterations, only one test-case is required in that regard. Best practice suggests to add a test-case for the maximum possible number of increments, as well as one that tests beyond that number.

The modified boundary interior test delivers significant insight in the tested code, while affording reasonable effort during testing. It has a significance similar to the MC/DC method, but focuses on the verification of program parts containing loops.

2.4.8 Path Coverage Test

The path coverage test is a superior method, requiring to execute all possible paths through a programs control-flow graph. This may seem appealing from the viewpoint of gathering as much insight into a program as possible. Alas, with increasing complexity, the efforts to test all those paths, growing exponentially, quickly surpass the gainable advantages. Comparably significant test data arises from modified boundary interior as well, with significantly lower efforts. Therefore, the path coverage test has a rather niche existence, but nevertheless its importance, as related methods are derived from it.

ToDo: spec (or scenario) coverage

```
if( !isFull(ptr) )
{
    while( !isFull(ptr) )
        append(ptr, getItem() )
}else
{
    blockItems();
}

while( isValid(ptr) )    // outer loop
{
    ptr++;

    for(idx = 0; idx < 4; idx++) // inner loop
    {
        itm[idx] = decomposeItem(ptr, idx);
    }

    processItem( itm[3], itm[2], itm[1], itm[0]);
}

free(ptr);
```

Figure 2.9: Example code, modified boundary interior test

ToDo: Probekapitel geht bis hier

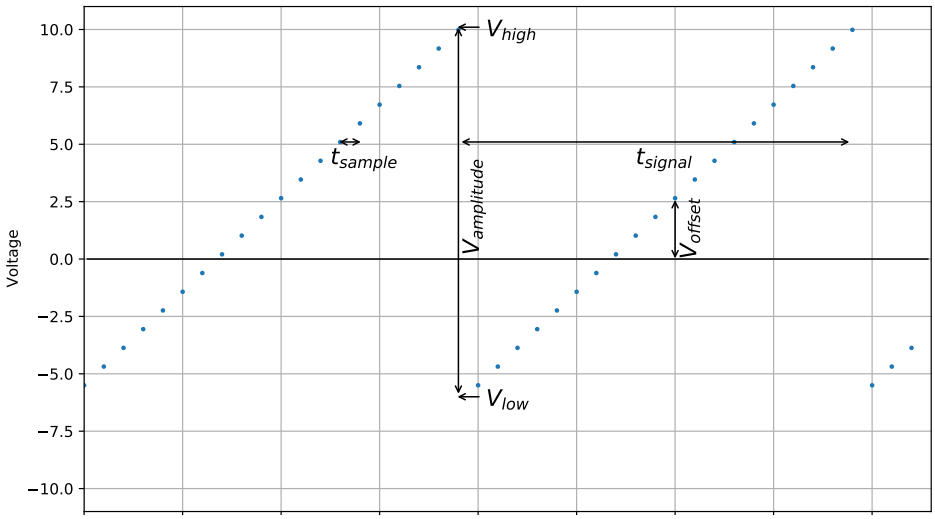
Chapter 3

Requirements

ToDo: Formulieren Two meetings, including all stakeholders of the OCTane, resulted in all requirements towards the OCTane and especially the firmware running on it. As these are imposed by the stakeholders, or, in other words, by the users of the OCTane, they are called 'user requirements'. Accompanying tags in the form 'RU-xx' allow for tracing relations between a requirement and according test-cases or points of implementation inside the source-code.

3.1 User requirements

Tag RU-1	Basic Functionality	Module	General
Description	The FirmWare has to access available hardware, to generate two-channel signals in ramp-, constant or arbitrary form, with • sample-rates ($\neq$ signal-frequency) up to 250kSPS • a resolution of 16bit • resulting in $\pm$10 volts of output voltage		



t_{sample} ... sampling-time or -period, alias: trigger-rate

t_{signal} ... signal-time or -period

f_{sample} ... sampling-frequency or -rate

f_{signal} ... signal-frequency or -rate

N ... sample-count, length of the signal-vector

$$f_{sample} = \frac{1}{t_{sample}} = N \cdot f_{signal}$$

$$f_{signal} = \frac{1}{t_{signal}} = \frac{1}{N \cdot t_{sample}}$$

$V_{amplitude}$ ... difference between maximum and minimum voltage of a signal.

V_{offset} ... deviation of a signal from 0 volts.

V_{high} ... maximum voltage of a signal

V_{low} ... minimum voltage of a signal

$$V_{amplitude} = V_{high} - V_{low}$$

$$V_{offset} = \frac{V_{high} + V_{low}}{2}$$

$$\rightarrow V_{high} = V_{offset} + \frac{V_{amplitude}}{2} \quad V_{low} = V_{offset} - \frac{V_{amplitude}}{2}$$

Tag RI-5	Priorities	Module	General
Description	In case of temporal overlapping tasks, first priority lays with analogue signal generation, second prio with USB-connectivity, third prio with Miscellaneous functions.		

Tag RU-2	Last Command Counts	Module	General
Description	The last submitted and accepted value for each parameter is the valid one.		

Tag RU-3	Parameters	Module	General
Description	The FirmWare has to implement user-adjustable parameters according to Tab. 1.		

Parameter	Values	reset value	Dim.	Type
TriggerA State	off|idle|arm|run	idle		enum
TrigA Mode	finite|infinite (freerun)	finite		enum
TrigA Input	USB|ext|TrigB|butt0	TrigB		enum
TrigA Signal-Rate	100m ... 125k	30.00e3	Hz	float
TrigA Signal-Period	8u ... 10	3.33e-5	s	float
TrigA Size	0 ... 250000	1000	samples	int
TriggerB State	off|idle|arm|run	idle		enum
TrigB Signal-Rate	100m ... 125k	30	Hz	float
TrigB Mode	finite|infinite (freerun)	finite		enum
TrigB Input	USB|ext|TrigC|butt1	TrigC		enum
TrigB Signal-Period	8u ... 10	3.33e-2	s	float
TrigB Size	0 ... 250000	1000	samples	int
TriggerC State	off|idle|arm|run	idle		enum
TrigC Mode	finite|infinite (freerun)	finite		enum
TrigC Input	USB|ext|butt2	USB		enum
TrigC Signal-Rate	20m ... 125k	3e-2	Hz	float
TrigC Signal-Period	8u ... 50	33.33	s	float
TrigC Size	0 ... 250000	1	samples	int
SourceA Mode	triggered|detached|singleshot	triggered	-	enum
SourceA Function	ramp|arbitrary	ramp	-	enum
SourceA Symmetry	0 ... 100	0	percent	float
SourceA Amplitude	0 ... 20	20	volts	float
SourceA Offset	-10 ... +10	0	volts	float
SourceA High-Volt	-10 ... +10	+10	volts	float
SourceA Low-Volt	-10 ... +10	-10	volts	float
SourceA Const-Volt	-10 ... +10	0	volts	float
SourceA timeout	0 ... 1000	0	ms	float
SourceB Mode	triggered|detached|singleshot	triggered	-	enum
SourceB Function	ramp|arbitrary	ramp	-	enum
SourceB Symmetry	0 ... 100	0	percent	float
SourceB Amplitude	0 ... 20	20	volts	float
SourceB Offset	-10 ... +10	0	volts	float
SourceB High-Volt	-10 ... +10	+10	volts	float
SourceB Low-Volt	-10 ... +10	-10	volts	float
SourceB Const-Volt	-10 ... +10	0	volts	float
SourceB timeout	0 ... 1000	0	ms	float
I2C mode	off|USB|slave	off	-	enum
UART mode	off|USB|slave	off	-	enum
Galvo-Relay	off|on	off	-	bool
SLD-Relay	off|on	off	-	bool
AIM-Relay	off|on	off	-	bool
CAM-Relay	off|on	off	-	bool
Relay5	off|on	off	-	bool
Relay6	off|on	off	-	bool
Watchdog	off|reset|powerdown|keepalive		-	enum
WDGTimeout	0 ... 1000	1000	ms	int
CRCmode	off|on	off	-	bool
VerboseMode	off|on	on	-	bool
A-in mode	off|USB|trig'd	-	-	enum
A-in value	0 ... 2^{12}	-	LSB	int
D-IO mode	off|in|out	-	-	enum
D-IO value	0 ... 2^{16}	-	bin-vect	int

Table 3.1: user-adjustable parameters

Tag RU-4	USB-Protocol	Module	USB-Stack
Description	The device has to provide the user with a USB-Interface. It has to be in the form of a VCP, text-based and SCPI-oriented. Messages in either direction may be up to 100 characters long and have to be delimited by the linefeed symbol '\n'.		

Tag RU-5	USB-Actions	Module	USB-Stack
Description	The FirmWare has to perform actions and state transitions as requested by USB-messages.		

Tag RU-6	Verbose	Module	USB-Stack
Description	The FW has to reply to every USB-command with a meaningful answer. This is called a 'verbose'-mode, has to be active on startup, but detachable by SCPI-command. Opposite is called *laconic* - mode		

Tag RU-7	USB-Timing	Module	USB-Stack
Description	USB-messages sent from the device to the host must be sent with a minimum interval of 1ms. The device must receive USB-messages in intervals up to 1ms.		

Tag RU-8	Case-Insensitivity	Module	USB-Stack
Description	The SCPI-detection has to be case-insensitive, and respond to the long form as well as the short form of SCPI commands.		

Tag RU-9	USB-turnoff	Module	USB-Stack
Description	The FirmWare has to deactivate USB-reactivity during A-, B- or C-scans, unless in freerun-mode. On startup, this functionality is active.		

Tag RU-10	SCPI	Module	USB-Stack
Description	The FirmWare has to parse USB-messages in a SCPI-fashion as defined in document "USB-Protocol.pdf", into FW-internal data structures.		

Tag RU-11	Restart	Module	USB-Stack
Description	The FirmWare has to perform a complete System-restart, when requested by USB-command.		

Tag RU-12	Standard-SCPIs	Module	USB-Stack
Description	The FirmWare has to implement mandatory SCPI-command according to IEEE 488.2		

Tag RU-13	Arbitrary Signal Vectors	Module	Signals
Description	The FirmWare must provide functionality to load user-defined arbitrary signal vectors, individually for both channels. In *verbose* - mode, every single transmitted value will be replied with a meaningful message, in *laconic* - mode, only the average value of the final vector will be replied Values will be transmitted one value per USB-command. optional: Transmit-mode to submit values chunk-wise.		

Tag RU-14	Vector Length	Module	Signals
Description	The FirmWare must provide functionality to set a user-defined signal vector length, either for ramp- and arbitrary signal, individually for both channels.		

Tag RU-15	source states	Module	Signals
Description	Signal generation must contain the following operational modes: $triggered$, $detached$, $single-shot$ • $triggered$: each pulse of the corresponding trigger causes the next vector value to be represented at the analogue output (default) • $detached$: analogue output holds a certain constant level, regardless of trigger and vector values (alias: $ref-pos$ - mode) • $single-shot$: analogue output holds a certain constant level, and returns to 0 volt after a specified timeout.		

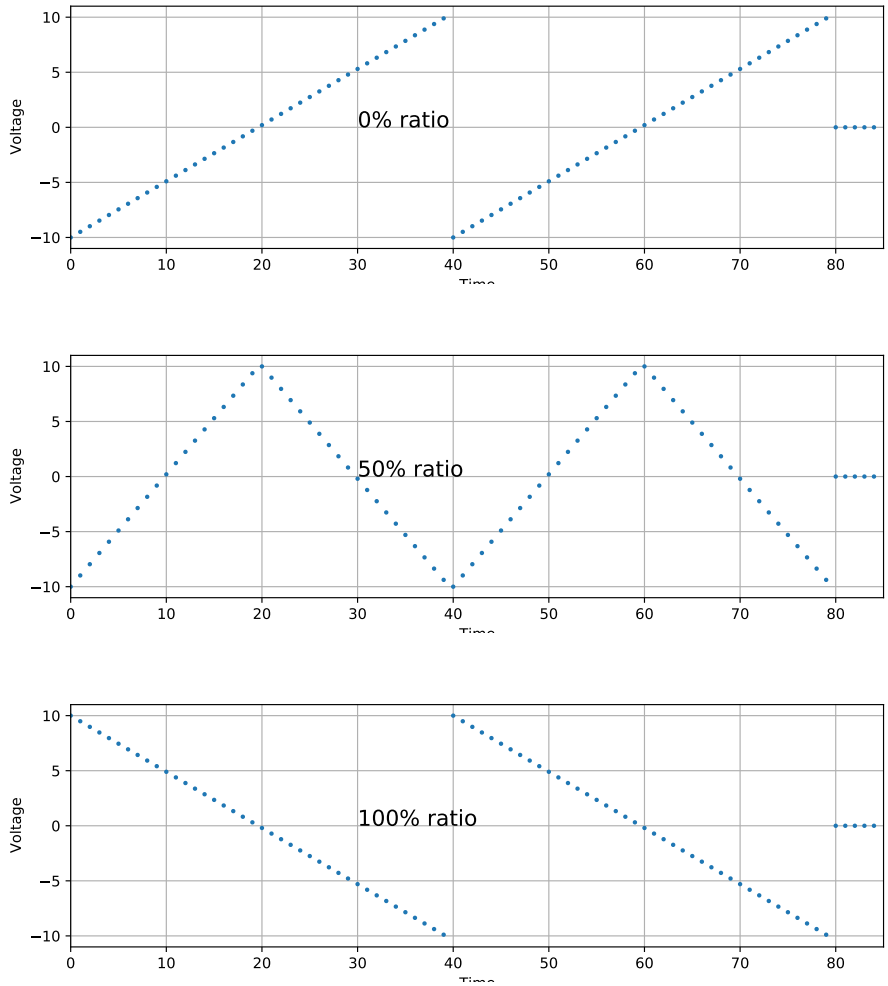
Tag RU-16	Default Ramp Signals	Module	Signals
Description	By default, signal vectors are to be loaded with ramp signals.		

Tag RU-17	signal-end	Module	Signals
Description	The FirmWare has to stop signal generation upon completion of all vector lengths and reset analogue outputs to 0V.		

Tag RU-18	free-run	Module	Signals
Description	The FirmWare has to provide a freerun mode. This mode continues signal generation, until a specific stop command is submitted via USB.		

Tag RU-19	Adjustable Signal Parameters	Module	Signals
Description	Signal generation has to be adjustable in amplitude and offset **or** high and low-voltage, signal-freq, **or** -period). This values apply to ramp- as well as arbitrary signals and will be applied to the signal vectors in a overwriting manner.		

Tag RU-20	Ramp symmetry	Module	Signals
Description	Ramp signals must have adjustable symmetry/asymmetry between 0% and 100%. The according meaning is depicted in the following graphics.		



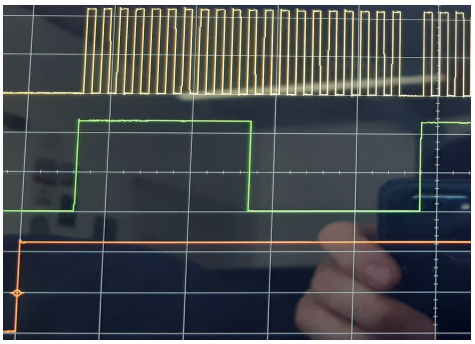
Tag RU-21	Trigger-IO	Module	Triggers
Description	Internal Trigger-Pulses must be put out via corresponding Trigger-outputs		

Tag RU-22	Trigger-Source	Module	Triggers
Description	Trigger-Modules must be implemented to handle timing of the signal-generation, comprising following input-sources: USB, $Trigger - Input$, $Superior\text{-}Trigger$, $Push - button$		

Tag RU-23	Timing-Parameters	Module	Triggers
Description	The FirmWare has to accept signal frequency or signal period and signal vector length as parameters. It has to reply with the actual frequency/period or an error message.		

Tag RU-24	Timing-Calc	Module	Triggers
Description	The FirmWare has to derive necessary sample-rates and trigger-periods from signal period and vector length, either by calculation or by selection from a look-up-table.		

Tag RU-25	Sequences	Module	Triggers
Description	The FirmWare has to generate sequences of A, B and C-Triggers. A-Trigger pulses have a duty-cycle of 50%, B and C-Trigger have falling edges upon completion.		



Tag RU-26	Buttons,LEDs	Module	Miscellaneous
Description	The FirmWare must access the available push-buttons and state-LEDs.		

Tag RU-27	Button-Function	Module	Miscellaneous
Description	Push-buttons must be programmed to cause transitions to the devices internal state, in a debounced manner.		

Tag RU-28	LED-Function	Module	Miscellaneous
Description	State-LEDs have to represent the current internal state of the device: *idle*, *armed*, *running* or *error*.		

Tag RU-29	Relays	Module	Miscellaneous
Description	The FirmWare has to provide access to the available relays. Access must consist of *close*, *open* and *read*-functions		

Tag RU-30	Additional IOs	Module	Miscellaneous
Description	The FirmWare has to provide access for available UART-, I^2C-, SPI-modules, as well as digital IOs and analogue inputs.		

Tag RU-31	Additional IO-Modes	Module	Miscellaneous
Description	Functionality for USART-, I^2C-, SPI-modules, the digital IOs and analogue inputs must consist of *activation*, $de-activation$, *write* and *read*.		

Tag RU-32	IO Read	Module	Miscellaneous
Description	*read*-Function must send received information to the host via USB. *read*-Function must be performed upon USB-command, or slave-action.		

Tag RU-33	CRC	Module	Miscellaneous
Description	The FirmWare has to implement functions to perform cyclic-redundancy-check calculations and apply it on verification of incoming strings and adaption of outgoing strings		

Tag RU-34	Watchdog Functionality	Module	Miscellaneous
Description	The FirmWare has to implement functions to enable the processors built-in watchdog and set its parameters. Available modes have to be *reset*, *powerdown*, *keepalive*		

3.2 USB-Protocol

SCPI Commands can be in 'short form', defined by the capital letters, or in 'long form', defined by the whole string. OCTane accepts both forms as commands and is case-insensitive. [18]

Sub-sys	Parameter	Value	Command	Response
Trigger A	State	off|idle|arm|run	TRIGgerA:STATe OFF	<state>|<error>
	State		TRIGgerA:STATe IDLE	<state>|<error>
	State		TRIGgerA:STATe ARM	<state>|<error>
	State		TRIGgerA:STATe RUN	<state>|<error>
	Mode (freerun)	finite	TRIGgerA:MODE FINite	<mode> |<error>
	Mode	infinite	TRIGgerA:MODE INFinite	<mode> |<error>
	Input	USB	TRIGgerA:INput USB	<input>|<error>
	Input	external input	TRIGgerA:INput EXTernal	<input>|<error>
	Input	Trigger B	TRIGgerA:INput TRIGgerB	<input>|<error>
	Input	Trigger C	TRIGgerA:INput TRIGgerC	<input>|<error>
	Input	Button	TRIGgerA:INput BUTTon	<input>|<error>
	Signal-Rate	1.0e-1 ... 125e3	TRIGgerA:RATE <freq>	<time>|<error>
	Signal-Period	8e-6 ... 10	TRIGgerA:PERIod <time>	<time>|<error>
	Vector-Size	1...250000	TRIGgerA:SIZE <size>	<size>|<error>
Trigger B	State	off|idle|arm|run	TRIGgerB:STATe OFF	<state>|<error>
	State		TRIGgerB:STATe IDLE	<state>|<error>
	State		TRIGgerB:STATe ARM	<state>|<error>
	State		TRIGgerB:STATe RUN	<state>|<error>
	Mode (freerun)	finite	TRIGgerB:MODE FINite	<mode> |<error>
	Mode	infinite	TRIGgerB:MODE INFinite	<mode> |<error>
	Input	USB	TRIGgerB:INput USB	<input>|<error>
	Input	External	TRIGgerB:INput EXTernal	<input>|<error>
	Input	Trigger C	TRIGgerB:INput TRIGgerC	<input>|<error>
	Input	Button	TRIGgerB:INput BUTTon	<input>|<error>
	Signal-Rate	1.0e-1 ... 125e3	TRIGgerB:RATE <freq>	<time>|<error>
	Signal-Period	8e-6 ... 10	TRIGgerB:PERIod <time>	<time>|<error>
	Vector-Size	1...250000	TRIGgerB:SIZE <size>	<size>|<error>
Trigger C	State	off|idle|arm|run	TRIGgerC:STATe OFF	<state>|<error>
	State		TRIGgerC:STATe IDLE	<state>|<error>
	State		TRIGgerC:STATe ARM	<state>|<error>
	State		TRIGgerC:STATe RUN	<state>|<error>
	Mode (freerun)	finite	TRIGgerC:MODE FINite	<mode> |<error>
	Mode	infinite	TRIGgerC:MODE INFinite	<mode> |<error>
	Input	USB	TRIGgerC:INput USB	<input>|<error>
	Input	External	TRIGgerC:INput EXTernal	<input>|<error>
	Input	Button	TRIGgerC:INput BUTTon	<input>|<error>
	Signal-Rate	1.0e-1 ... 125e3	TRIGgerC:RATE <freq>	<time>|<error>
	Signal-Period	8e-6 ... 10	TRIGgerC:PERIod <time>	<time>|<error>
	Vector-Size	1...250000	TRIGgerC:SIZE <size>	<size>|<error>
Source-A	Mode	triggered	SOURceA:MODE TRIGgered	<mode>|<error>
	Mode	detached	SOURceA:MODE DETached	<mode>|<error>
	Mode	singleshot	SOURceA:MODE SINGleshot	<mode>|<error>
	Function	Ramp	SOURceA:FUNCtion:SHAPe RAMP	<func>|<error>
	Function	Arbitrary	SOURceA:FUNCtion:SHAPe ARBitrary	<func>|<error>
	Symmetry	0 ... 100	SOURceA:RAMP:RATIO <ratio>	<ratio>|<error>
	Arb load	-	SOURceA:ARBitrary:LOAD	<count>|<error>
	Arb val	±10.000	SOURceA:ARBitrary:VALUe <idx, val>	<idx, val>|<error>
	Amplitude	0.000...20.000	SOURceA:FUNCtion:AMPlitude <ampl>	<ampl>|<error>
	Offset	±10.000	SOURceA:FUNCtion:OFFset <offs>	<offs>|<error>
	High	±10.000	SOURceA:FUNCtion:HIgh <high>	<high>|<error>
	Low	±10.000	SOURceA:FUNCtion:LOw <low>	<low>|<error>
	Constant	±10.000	SOURceA:VOLTage:LEVel <volts>	<volts>|<error>
	Timeout	1...1000ms	SOURceA:PULSe:WIDth <time>	<time>|<error>
Source-B	Mode	trig|det|single	SOURceB:MODE TRIGgered	<mode>|<error>

	Mode	trig|det|single	SOURceB:MODE DETached	<mode>|<error>
	Mode	trig|det|single	SOURceB:MODE SINGleshot	<mode>|<error>
	Function	Ramp	SOURceB:FUNCtion:SHAPe RAMP	<func>|<error>
	Function	Arbitrary	SOURceB:FUNCtion:SHAPe ARBitrary	<func>|<error>
	Symmetry	0 ... 100	SOURceB:RAMP:RATIO <ratio>	<ratio>|<error>
	Arb load	-	SOURceB:ARBitrary:LOAD	<count>|<error>
	Arb val	±10.000	SOURceB:ARBitrary:VALUe <idx, val>	<idx, val>|<error>
	Amplitude	0.000...20.000	SOURceB:FUNCtion:AMPlitude <ampl>	<ampl>|<error>
	Offset	±10.000	SOURceB:FUNCtion:OFFset <offs>	<offs>|<error>
	High	±10.000	SOURceB:FUNCtion:HIgh <high>	<high>|<error>
	Low	±10.000	SOURceB:FUNCtion:LOw <low>	<low>|<error>
	Constant	±10.000	SOURceB:VOLTage:LEVel <volts>	<volts>|<error>
	Timeout	1...1000ms	SOURceB:PULSe:WIDth <time>	<time>|<error>
Relays	Galvo	close|open|read	ROUTe:<CLOSe|OPEN|STATE?> GAL	<state>|<error>
	SLD	close|open|read	ROUTe:<CLOSe|OPEN|STATE?> SLD	<state>|<error>
	AIM	close|open|read	ROUTe:<CLOSe|OPEN|STATE?> AIM	<state>|<error>
	CAM	close|open|read	ROUTe:<CLOSe|OPEN|STATE?> CAM	<state>|<error>
I2C	mode	OFF	I2C::MODE OFF	<mode>|<error>
	mode	USB	I2C::MODE USB	<mode>|<error>
	mode	slave-action	I2C::MODE SLAVeaction	<mode>|<error>
	write	0 ... 255	I2C::WRITe <val>	<val>|<error>
	read	0 ... 255	I2C::READ	<val>|<error>
UART	mode	OFF	UART:MODE OFF	<mode>|<error>
	mode	USB	UART:MODE USB	<mode>|<error>
	mode	slave-IRQ	UART:MODE SLAVeaction	<mode>|<error>
	write	0 ... 255	UART:WRITe <val>	<val>|<error>
	read	0 ... 255	UART:READ	<val>|<error>
DIO	mode	OFF	DIGIO:MODE OFF	<val>|<error>
	mode	input	DIGIO:MODE IN	<val>|<error>
	mode	output	DIGIO:MODE OUT	<val>|<error>
	write	0 .. 65535	DIGIO:WRIte <val>	<val>|<error>
	read	0 .. 65535	DIGIO:READ	<val>|<error>
AnalogIN	mode	OFF	ANAlog0|1|2|3:MODE OFF	<val>|<error>
	mode	USB	ANAlog0|1|2|3:MODE USB	<val>|<error>
	mode	triggered	ANAlog0|1|2|3:MODE TRIGA	<val>|<error>
	mode	triggered	ANAlog0|1|2|3:MODE TRIGB	<val>|<error>
	mode	triggered	ANAlog0|1|2|3:MODE TRIGC	<val>|<error>
	read	0 ... 4095	ANAlog0|1|2|3:READ	<val>|<error>
System	CRCmode	OFF	SYStem:CRC16 OFF	<state>|<error>
	CRCmode	on	SYStem:CRC16 ON	<state>|<error>
	ShutDown	-	SYStem:POWerdown	POWD|<error>
	ListSCPI	-	SYStem:LISt	<list>|<error>
	RESEt	-	SYStem:RESEt	RESE|<error>
	RESTart	-	SYStem:RESTart	REST|<error>
	Verbosity	OFF	SYStem:VERBose OFF	<mode>|<error>
	Verbosity	on	SYStem:VERBose ON	<mode>|<error>
	Watchdog	OFF	SYStem:WATchdog OFF	<mode>|<error>
	Watchdog	on	SYStem:WATchdog ON	<mode>|<error>
	Time	1...1000ms	SYStem:WATchdog <time>	<time>|<error>

Table 3.2: OCTane USB-Protocol, commands

Sub-sys	Parameter	possible messages	occurence		
Trigger A|B|C	State	TrigX idling|armed|running	sent on every state change		
Trigger A|B|C	Input	-200	error, if button in use		
Trigger A|B|C	Signal-Rate	-200	error, if out-of-range		
Trigger A|B|C	Signal-Period	-200	error, if out-of-range		
Trigger A|B|C	Vector-Size	-200	error, if out-of-range		
Source A|B	Arb load	-200	error, if not in Arb-mode		
Source A|B	Arb val	VectorX complete	if sufficient amount of values was sent		

Source A|B	Arb val	-200	error, if out-of-range		
Source A|B	Arb val	-200	error, if exeeds vector-size		
Source A|B	Symmetry	-200	error, if out-of-range		
Source A|B	Amplitude	-200	error, if out-of-range		
Source A|B	Offset	-200	error, if out-of-range		
Source A|B	High	-200	error, if out-of-range		
Source A|B	Low	-200	error, if out-of-range		
Source A|B	Constant	-200	error, if out-of-range		
Source A|B	Timeout	-200	error, if out-of-range		
AIN	input value	AINx: <value>	sent on every corresp. Trigger		
DIN	input value	DIN: <value>	sent on every DIO:READ-Command		
UART	input value	UART: <value>	sent on every corresp. Trigger		
I2C	input value	I2C: <value>	sent on every corresp. Trigger		

Table 3.3: OCTane USB-Protocol, responses

Table 3.3 specifies the responses by the OCTane, if they are not described sufficiently in the previous table.

Command	Description	Action	Return
*CLS	Clear Status Command		
*ESE	Standard Event Status Enable Command		
*ESE?	Standard Event Status Enable Query	-	
*ESR?	Standard Event Status Register Query	-	
*IDN?	Identification Query	-	ID-String
*OPC	Operation Complete Command		
*OPC?	Operation Complete Query	-	
*RST	Reset Command		
*SRE	Service Request Enable Command		
*SRE?	Service Request Enable Query	-	
*STB?	Read Status Byte Query	-	Status Byte
*TST?	Self-Test Query	-	
*WAI	Wait-to-Continue Command		

Table 3.4: IEEE 488.2 mandatory commands

3.2.1 Analogue outputs Resolution and LSB

mapping 20Vpp Voltage space to a resolution of 16bit

- 0 ... 30000 ... 60000
- 1000 ... 31000 ... 61000
- 0 ... 32767 ... 65535
- ???

$\rightarrow$ LSB $\triangleq$ ...mV

3.2.2 Load-Hypothesis, Fault-Hypothesis

3.2.3 Traceability-Matrix

Linking Requirements by there tags, to the SW-modules, where they are fulfilled The traceability Matrix establishes the relations between user requirements and test cases. Furthermore it is good practice to also include requirement IDs inside the source code on the exact point of implementation. The convenient layout for a traceability matrix

is to list requirement IDs column-wise, while noting the IDs of the test cases row wise. The reason being that one requirement may have multiple test cases and and it is more convenient to note these multiple IDs in rows, rather than columns.

3.2.4 V-Model

3.2.5 Test-cases

Automated test-cases are the primary method to ensure code-quality, because they allow for the assessment of functionalities against requirements and produce according documentation of conducted tests. Furthermore, they help identifying errors in case of failures and secure implemented functionalities during future adaptions. To demonstrate a complete assessment of the firmware, for every existing user-requirement, at least one test-case is necessary. For practical reasons, these cases must be implemented as python-scripts, running on a proxy host-device, controlling the device under test via USB. The first practical aspect, being, that the resulting test-data are directly available on a device with capable processing power and high memory capacity, facilitating the automated generation of test-reports. The python programming language provides the package 'pytest-html', allowing for the automated generation of test-reports in HTML-format. Every test-case has to be simple enough, to render verification of the test-code itself unnecessary. A test-case, so complex, that i would require a superordinate test-case, indicates, that said test-case should be split into several cases of lesser respective complexity. Fig. 3.1 contains an exemplary test-case, assessing the devices correct reply to an identification query. [2]

```
def test_IDN():
    reply = serTxRx('*IDN?')
    assert 'RECENDT GmbH' == reply.split(', ')[0]
    assert 'OCTane' == reply.split(', ')[1]
```

Figure 3.1: testCaseExample

The general procedure of this test-case consists of

- Sending a command to the device under test,
- Gathering resulting test-data and
- Verification of retrieved data against requirements.

Upon execution via 'pytest-html', this test-case either results as 'passed' or 'failed', depending on the contained assertions. Furthermore, it causes an entry in the resulting report-file, likewise to the extract in fig. 3.2

In case of commands resulting in digital, analogue or serial output-signals, these signals shall be automatically measured as part of a test-case. Automated measurement is required if performable with justifiable effort and available measurement instruments. Apart from this method of fully automated testing, few requirements demand assessment in a manual fashion. For example, oscillograms of the resulting analogue signals require evaluation buy the eye of a skilled engineer. Automation of this process via spectral analysis or automated comparison with reference signals would require unjustifiable

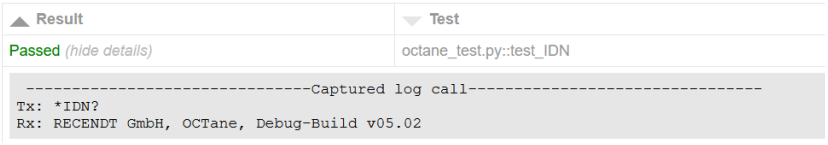


Figure 3.2: reportExample

effort, compared to an evaluation via visual inspection.
Test-cases, usually, belong to one of the following classes, **ToDo:** according to the V-Model:

3.2.6 Unit-Tests

Unit-tests are test-cases that evaluate the correctness of single functions, methods or procedures. A single variable, array or data-set also constitutes such a unit, if the contained data demands deliberate assessment via a test-case. Viable inputs exist in the form of binary values with separate cases for 'true' and 'false'. In case of numerical input, be they of integer or floating-point nature, the boundary-value and equivalence-class methods deliver suitable input values. For inputs in text form, all specified valid texts, and at least one invalid text form a set of useful input values. [9]

3.2.7 Integration-Tests

The next level of tests concern the interactions between units and their correct cooperation to form correctly working modules and sub-systems. A major focus in integration testing lies on the verification of units and modules to ensure their correct interaction. As this aspect is of a lesser concern during unit-testing, integration-testing is an established branch of verification in its own right. Furthermore, side-effects of units, which are hardly a concern during unit-testing, are important aspects during integration-testing. Testing and demonstrating seamless interoperability and collaboration of modules are the prime objectives. [17] The strategy of integration, happening during implementation has significant influence on the design of suitable integration-tests. These are the most common approaches:

- Top-down integration
- Bottom-up integration
- Ad-hoc integration
- Backbone integration

3.2.8 System-Tests

TODOkann ma den da reinschummeln? [3]

3.2.9 Load/Fault Tests

ToDo: cite: IEEE830.pdf, Crowder-REQs, Buttazzo, system-deadlocks (Coffman) und Datenblätter

3.2.10 End to end Test

The core concern is if a test is intended and designed as a unit or an end to end test, regardless of additional resources of the system being used. Even though unit tests are not into end-to-end tests they may very well use the outer interfaces of the system under test.

3.2.11 Code Coverage

Code Coverage is an accompanying metric to test-cases, indicating their accuracy and completeness of assessment. The goal for this project is to achieve complete statement and branch coverage for the original, self-written source-code. Third-party libraries and HAL-modules from the processor-vendor are excluded from this requirement.

Chapter 4

Concept

4.1 system architecture

Modules-Skizze + HW-Graph und ein meta-graph der diese verbindet.

4.1.1 Modules of the Firmware

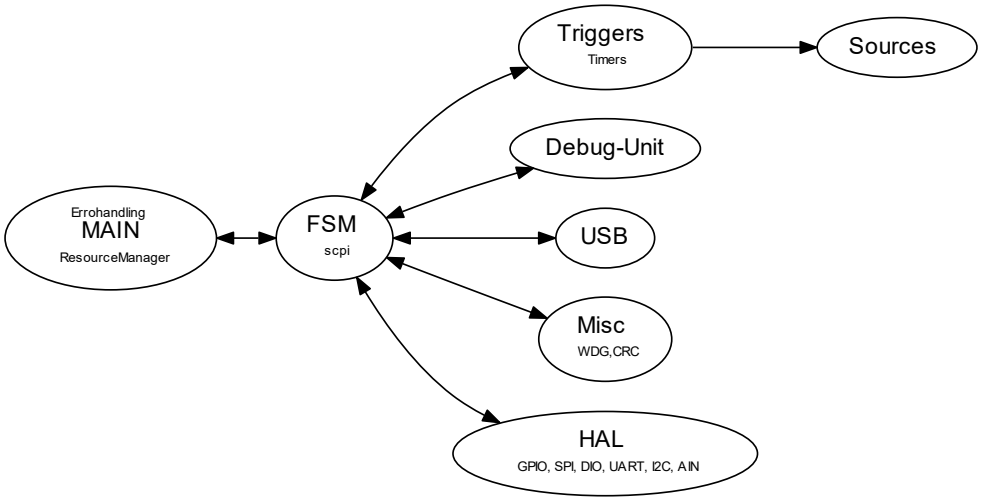


Figure 4.1: Modular structure

4.2 Module Integration Strategy

These strategies constitute the most promising approaches in module integration:

- Top-down integration
- Bottom-up integration

- Ad-hoc integration
- Backbone integration

Top-down integration approaches the code under test from its entry point and external interfaces, non-existing sub-systems require replacement through stubs delivering dummy-data.

Bottom-up integration builds modules and subs-systems based upon existing low-level functions, while non-existing higher-ranking systems require replacement through test-drivers delivering dummy-commands.

Ad-hoc integration is the least formal integration-strategy, where components undergo integration directly after integration, regardless of their level or rank within the complete system. The effort in planning integration is negligible, while non-existing components demand higher effort for their replacements. Furthermore, the lack of a thoroughly planned integration phase might diffuse into the resulting software exhibiting an erratic and patchy structure.

Backbone integration [4] is the formal equivalent of ad-hoc integration, where the initial task is, to build an overarching backbone or skeleton for the whole project and afterwards integrate components in arbitrary order. This leaves substantial freedom to the order of developing components, alas requires thorough planning up front and notably effort initially, to implement the backbone.

Backbone integration is the most promising option and consequently the selected choice for the project at hand. The appeal of that strategy stems from the possibility to develop and integrate components in any order. This allows to work on another component, if one imposes seemingly unsolvable problems and come back to that problematic component at a later point. In contrast to the ad-hoc method, backbone integration maintains order and structure over a software project, while leaving mentioned degrees of freedom.

4.3 FSM

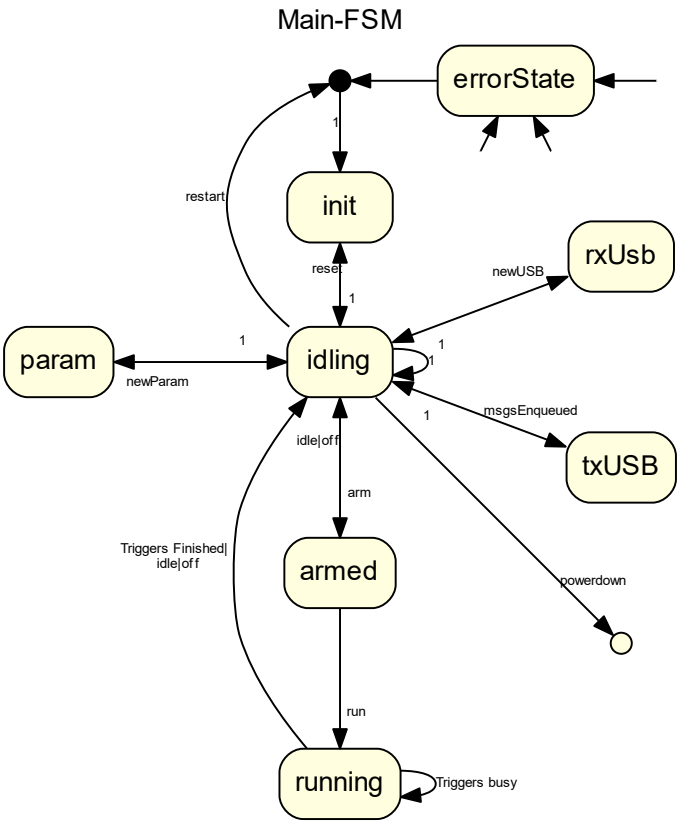


Figure 4.2: overarching Finite state machine

Triggers and Voltage - Outputs

association of Triggers and their analogue outputs

- TriggerB → SourceB → Vout1
- TriggerA → SourceA → Vout2

4.3.1 Standard operation procedures (SOP)

4.4 Hardware

4.4.1 STM32F4

SOURce1:FUNCtion:Amplitude 6	
SOURce1:FUNCtion:Offset 3	
SOURce2:FUNCtion:Amplitude 4	
SOURce2:FUNCtion:Offset -4	
TRIGgerC:STATe RUN	start scan sequence

Table 4.1: one Volume-Scan

SOUR2:VOLT:LEV 4.5	both Galvos in fixed positions
SOUR1:VOLT:LEV -2.95	no Triggers
...	
SOUR2:VOLT:LEV 0	Send galvos home afterwards
SOUR1:VOLT:LEV 0	

Table 4.2: A-Scan in one position

TRIGgerB:STATe stop	deactivate
TRIGgerA:STATe stop	in exactly this order
SOUR1:VOLT:LEV 0	send Galvo home
SOUR1:mode:trig	reattach Galvo to TriggerB
TRIGgerB:MODE trigC	reattach TriggerB to TriggerC

Table 4.3: B-Scan in one position, continuous A-Scans, 'A-Freerun' Mode-'infinite'

SOUR1:MODE free	detach Galvo from its Trigger
SOURce2:FUNCtion:Amplitude 3.5	
SOURce2:FUNCtion:Offset 1.95	
TRIGgerB:Mode CONTinuous	...Trigger will run forever
TRIGA:PRE 4	
TRIGA:tcou 74	...40kHz A-Scans
TRIGB:pre 64	....10Hz B-Scans
TRIGA:cou 1550	1550 samples
TRIGB:tcou 36500	10Hz
TRIGgerB:STATe RUN	activate
TRIGgerB:STATe stop	activate
TRIGA:cou 1250	1250 samples
TRIGB:tcou 14600	25Hz
TRIGgerB:STATe RUN	activate
TRIGgerB:STATe stop	deactivate
TRIGA:cou 620	620 samples
TRIGB:tcou 7300	50Hz
TRIGgerB:STATe run	activate
TRIGgerB:STATe stop	deactivate in exactly
TRIGgerA:STATe stop	this order
SOUR1:VOLT:LEV 0	send Galvo home
SOUR1:mode:trig	reattach Galvo to TriggerB
TRIGgerB:MODE trigC	reattach TriggerB to TriggerC

Table 4.4: Ivan Patch

4.4.2 Wandler, Level-Shifter, HighSider

4.4.3 Connection of Galvos and Triggers

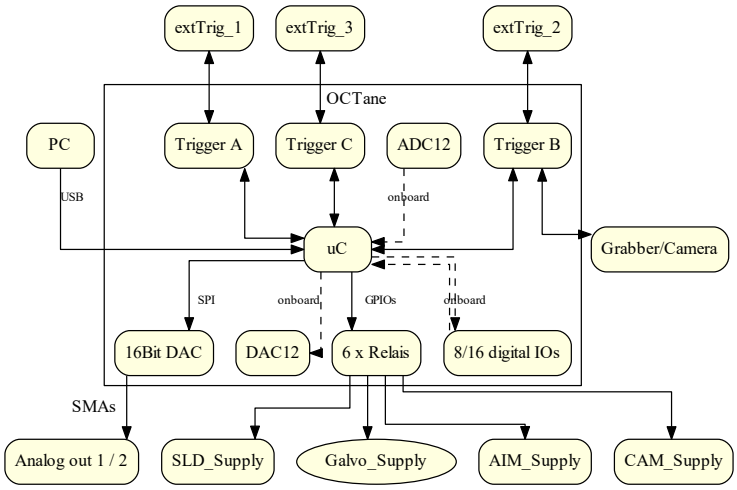


Figure 4.3: HardWare

Source1 - Galvo y (slow)	Trigger B
Source2 - Galvo x (fast)	Trigger A

Table 4.5: Assignment of Triggers and according analogue outputs

Chapter 5

Implementation

5.1 Trigger-Diagramme

5.2 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

5.2.1 Trigger-Lines and Timers

utilisation of the output compare - timers

- TrigA $\hat{=}$ $TRIG\_2$ $\hat{=}$ PB3 $\leftarrow$ $TIM2_CH2$
- TrigB $\hat{=}$ EN_3 $\hat{=}$ PC6 $\leftarrow$ $TIM8_CH1$
- TrigC $\hat{=}$ EN_4 $\hat{=}$ PC7 $\leftarrow$ $TIM3_CH2$

5.2.2 Debug-Unit

A debug-unit, offering eight digital outputs via set.. and rst.. - functions, was established. Fig. 5.1 shows a 'ladder' setting and resetting all debug-Pins upon initialization of the module.

Figure 5.1: dbgUnitLogic

5.3 Software tools

5.3.1 CubeIDE

5.3.2 Gcov

'gcov' is a suitable software tool to measure code coverage. It is part of the gnu compiler collection 'gcc'. It is free of charge, open source and operates on various platforms like Linux, Windows and Mac and supports target platforms ranging from arm microprocessors to x86 processors. It is able to perform statement as well as branch coverage analysis. While the actual GCC compiler produces instrumented code suitable for coverage measurement, gcov merely converts coverage data post factum into human readable reports.

Instrumentation

The production code per se is not suited for coverage measurement, because no data is generated to base coverage analysis on. Therefore it is necessary to insert counter variables into the source code that keep track on how often every block of code was executed and which branches of decisions were taken and which not. Furthermore data management of those counters is necessary as well as exporting generated data onto a host system for further processing. Instrumentation is the process of inserting mentioned counters and the according data management into the original source code. Additionally

the compiler generates information about basic blocks, arcs and information to relate them to line numbers in the source code. Basic blocks are sequences of statements without any branching statements and branch targets. Arcs on the other hand are branch operations and their according targets that link basic blocks together.

The instrumented source code is not identical to production source code! In case of the gcov software tool the instrumentation is done by the actual GCC compiler and not the gcov tool itself. To force the compiler to produce instrumented code, the flags '-fprofile-arcs' and '–ftest-coverage' are necessary. The instrumented code is merely an intermediate by-product, not directly visible to the developer. It is good practice, to generate instrumented code only for one module at a time and not a complete project at once, especially on a bare-metal firmware-project. They binary footprint of the instrumented source-code is substantially larger than the non-instrumented code. Therefore, the demand in memory might overwhelm the limited resources of the embedded processor, if attempting to measure coverage for a complete project at once. Post-processing tools for gcov allow for assembling data-sets of several modules into one report. With the mentioned flags, the compiler measures how often a program reaches a branch instruction and how often it actually performs a branch operation. For this purpose GCC creates a control flow graph and an accompanying spanning tree. Statements not contained in this spanning tree require counting, as they are not executed in every case.

Producing Raw Data

Executing instrumented code produces a 'gcda'-file, holding information about control-flow-graph and the spanning tree. By default, the instrumenting instructions rely on syscalls to generate and fill files with the gathered counter-data. This step requires a workaround on bare-metal systems, as there, the mentioned syscalls only exist as dummies. A subsequent chapter describes this workaround. Now it is known which branch operations the program performed and how often it executed each statement and block of code. This information allows to write to generate a 'gcov' file and a statistical analysis. The 'gcov' file is essentially the original source code, which the gcov-tool prepends with annotations about execution-counts of statements, branching information and omitted statements, retrieved from the binary gcda-file.

The '-g' compiler option might be helpful for measurement of coverage data, even though it is not mandatory. It forces the compiler to generate debug information in the executable binary. This allows for example to perform breaks during execution of an instrumented program. This is a useful option for simulating failed memory allocations. Executing statements that attempt to access memory that was faulty allocated demonstrates how a program handles faulty memory regions. It is an exceptional situation for the program and therefore complicated to provoke during test cycles. Therefore, simulating exceptions via the debuggers break-functionality can be a useful method to test handling of said exceptions. So, to achieve complete branch coverage it, might be necessary to incorporate the GDB-debugger into the process. This thesis omits a detailed explanation of this debugger, as it is not the core topic. [13]

To produce the raw coverage information, at least one execution of the instrumental program is necessary. Repeated execution of the program results in the instrumenting code parts to append additional counter information to the already existing gcda file. This step again requires a certain amount of work around on bare metal systems.

Processing Raw Data

After one or several executions of the instrumented program, gcov is able to link and analyse the information contained in the gcda-file the gcno-file and the original source code. This results in the mentioned gcov-file and statistic data about the source-codes coverage. By applying the '-b' flag upon execution of gcov, branch coverage information is generated as well. Typically gcov states the percentages of executed code lines, of executed branches, of taken branches and of executed function-calls, in one module. The '-c' option allows to retrieve statistics about branching operations in absolute values rather than percentages. The '-f' option delivers statement coverage also for every separate function, additionally to the statement coverage of a whole module.

Post-processing

Up to this point the existing analysis data is already of valuable insight for a developer, but scattered among modules and not in a presentable format. This calls for post-processing via tools like 'lcov', 'llvm-cov', 'kcov' or 'gcovr', who are able to generate an aggregate report of several modules in HTML-, JSON-, CSV-, and XML-format. [21] [10] Fig. 5.3 depicts an extract of the coverage report of one submodule and 5.2 contains a composite report about code coverage over the whole firmware-project. As the figures originate from an early phase of the project, they contain only one module called 'DebugUnit'. The software tool 'gcovr' produces such expressive reports as HTML-files. The summarizing statistics, the coverage metrics appear on the right side of the header of every report, with a colouring-scheme similar to traffic-lights, to indicate problematic metrics. The total report replicates this concept, itemized for every single module in the body part of the output. Reports of separate modules list the according source-code in the body part, indicating covered code-portions in green and not executed parts in red. This representation provides an intuitive overview of areas, that require further testing.

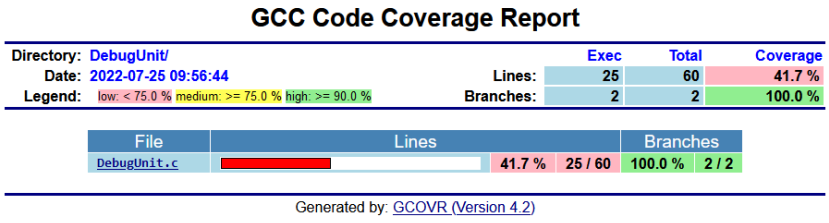


Figure 5.2: gcovReport01

GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	DebugUnit/	Lines:	25	60	41.7 %
File:	DebugUnit/DebugUnit.c	Branches:	2	2	100.0 %
Date:	2022-07-25 09:56:44				

```
Line  Branch  Exec  Source
   1                /*
   2                 * DebugUnit.c
   3                 *
   4                 *  Created on: 14.04.2022
   5                 *      Author: r.hif
   6                 */
   7
   8                #include "main.h"
   9                #include "OCTaneHAL.h"
  10                #include "ResMan.h"
  11
  12
  13                #ifdef DEBUG
  14                      void setDbgPin1(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN01);            }
  15                2     void clrDbgPin1(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN01);          }
  16                      bool readDbgPin1(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN01);}
  17
  18                      void setDbgPin2(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN02);            }
  19                2     void clrDbgPin2(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN02);          }
  20                      bool readDbgPin2(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN02);}
  21
  22                      void setDbgPin3(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN03);            }
  23                2     void clrDbgPin3(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN03);          }
  24                      bool readDbgPin3(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN03);}
  25
  26                      void setDbgPin4(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN04);            }
  27                2     void clrDbgPin4(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN04);          }
  28                      bool readDbgPin4(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN04);}
  29
  30                      void setDbgPin5(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN05);            }
  31                2     void clrDbgPin5(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN05);          }
  32                      bool readDbgPin5(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN05);}
  33
  34                      void setDbgPin6(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN06);            }
  35                2     void clrDbgPin6(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN06);          }
  36                      bool readDbgPin6(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN06);}
  37
  38                      void setDbgPin7(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN07);            }
  39                2     void clrDbgPin7(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN07);          }
  40                      bool readDbgPin7(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN07);}
  41
  42                      void setDbgPin8(void)   {       SET_BIT(DBGPORT->ODR, DBGPIN08);            }
  43                2     void clrDbgPin8(void)   {       CLEAR_BIT(DBGPORT->ODR, DBGPIN08);          }
  44                      bool readDbgPin8(void)  {       return READ_BIT(DBGPORT->ODR, DBGPIN08);}
  45
```

Figure 5.3: gcovReportDebugUnit01

5.3.3 Measuring Code Coverage on bare-metal systems

Code coverage imposes a significant challenge on bare metal systems: Measurement-tools assume an underlying operating system, especially a file-system, that receives the measured raw data. This requires a workaround, as 'bare-metal' literally means 'without an operating system'. It is very-well possible to establish a file-system, alas at significant demands in processing power and memory. As these resources require economical utilization, transferring data directly out of the device under test during test-cycles is an appealing alternative. Most ARM-based processors provide ITMs, instrumentation trace macro-cells, coupled with SWO (serial wire output), a dedicated debug-interface. This allows to define debug-informations in firmware and provide them via SWO. Debug-probes, like ST-Link2, support this interface and pass the data on to a host-system. The processors serial interfaces like USB or RS-232 are unaffected by this and remain available for user-application, instead of debugging-purposes.

Fig. 5.4 illustrates the process, applying the described components for coverage measurement: A compiler generates executable binary-code from the instrumented code under test, a programming tool transfers the compiled code to the target platform. As gcov utilises the syscalls `_open()` , `_write()` and `_exit()` , additional code redirects these calls to the processors ITM-functions. During test-cycles, the IT-macrocell provides the emerging coverage information via SWO, while a debug-probe picks up this information. The probe then passes the information on to a host-PC, capable of storing large volumes of data and further processing. An ad-hoc-script, written in python, converts the

raw coverag information into gcda-files, compatible with gcov and its derivates. Either a combination of gcov and lcov, or gcovr then leads to a final HTML-report, similar to fig. refgcovReportDebugUnit01.

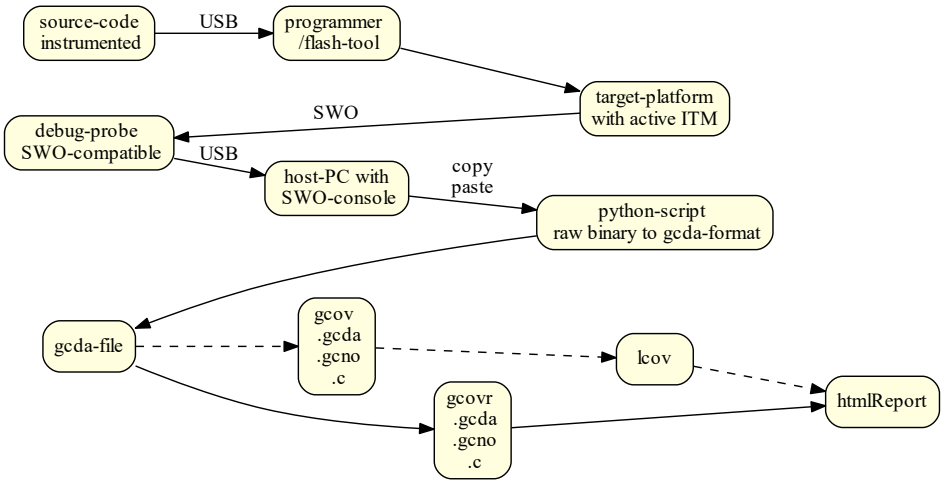


Figure 5.4: coverageFlow

Special attention regarding versions of the compiler, the coverage tool and the reporting tool is necessary. Report generation easily fails, if the versions of the tools generating gcno-, gcda- and html-files do not match up exactly. Furthermore, only one exact version (v5) of the arm-compiler generates instrumented source code applicable for coverage measurement on STM32 processors. Lcov does not produce any reports containing any coverage metrics at all, at least not with the compiler- and gcov-versions in use.

Unfortunately, only versions up to 4.2 of gcovr provides compatibility among compiler,

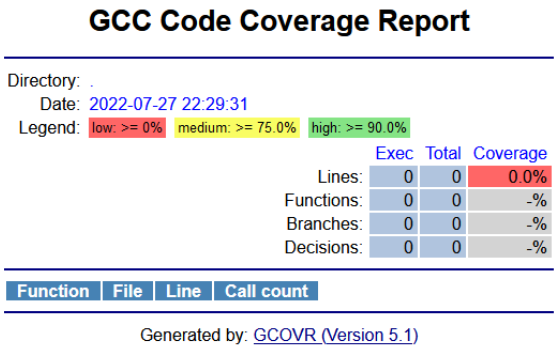


Figure 5.5: gcvor52

coverage-tool and the platform performing the coverage analysis. While the most recent version 5.2 delivers separate metrics for function-calls and decisions (see fig. 5.5), version 4.2 is only capable of providing statement and branch-coverage[1] .

The described process is rather cumbersome, hardly automated and requires some manual intervention. Nevertheless, the resulting reports are of significant value to the developer, which again justifies the efforts.

5.3.4 python tools, pytest-html

Chapter 6

Results

6.1 Measurements

Messaufbau, werte, ergebnisse, interpretation

6.1.1 Oszi, Debug-Unit und Opto-Detektoren

6.2 Test-Res

6.3 Test-Cases

6.4 Code Coverage

6.5 Code Review

6.6 Review Remarks

6.7 Gavlo-Performance

6.8 Project-status

6.9 ...

Chapter 7

Conclusion

7.1 Test Cases

Failed *(hide details)* | octane_test.py::test_SourceB_DC | 0.06

```
def test_SourceB_DC():
        reply = serTxRx("System:verbose on")
        reply = serTxRx("System:debug off")

        reply = serTxRx("SOURceB:VOLTage:LEVel?")
        assert "SourceB voltage is: " in reply.strip()

        reply = serTxRx("SOURceB:VOLTage:LEVel 10.00")
>       assert "setting Src B voltage to: 10.000" == reply.strip()
E    AssertionError: assert 'setting Src ...ge to: 10.000' == 'SourceB voltage is: 10.000'
E      - SourceB voltage is: 10.000
E      + setting Src B voltage to: 10.000

octane_test.py:210: AssertionError
```

Figure 7.1: exampleHelpfulFailedTest01

Fig. 7.1 contains an example of a helpful test-case that failed in the beginning. The direct comparison of expected and actual results allows to backtrack the problem via searching for the unexpected result-string. This leads to the accompanying enum-ID of that string, that was misplaced because of a typing error of one single letter. The Fig. 7.2 depicts the according place in the source code, with the erroneous ID TX_SOURB_VOLT_LEVq instead of TX_SOURB_VOLT_LEV . This example highlights

```
rSCPIReplys[TX_SOURB_VOLT_LEVq], getDC(&srcB)
```

Figure 7.2: exampleHelpfulFailedTestSrc01

the advantages of numerous simple test-cases, already during the debugging and early verification of newly implemented functionalities.

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] the gcovr authors. *gcovr Documentation*. Version Release 5.1. July 4, 2022 (cit. on p. 52).

[2] Aravind Balaji, S Sasikumar, and Dr. Ramesh K. "SCPI based integrated test and measurement". *IOP Conference Series: Materials Science and Engineering* 1045 (Feb. 2021) (cit. on p. 38).

[3] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. 1st ed. MR: referenziert aus Binder1999 (u.a. im Hinblick auf data flow testing strategies). Verlag John Wiley and Sons, Inc, 1995 (cit. on p. 39).

[4] Boris Beizer. *Software Testing Techniques (2nd Ed.)* USA: Van Nostrand Reinhold Co., 1990 (cit. on p. 42).

[5] John Chilenski. "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion" (Jan. 2001) (cit. on p. 21).

[6] John Chilenski and Steven Miller. "Applicability of modified condition/decision coverage to software testing". *Software Engineering Journal* 9 (Oct. 1994), pp. 193–200 (cit. on p. 21).

[7] William E. Howden. "An evaluation of the effectiveness of symbolic testing". *Software: Practice and Experience* 8 (1978) (cit. on p. 24).

[8] William E. Howden. "Methodology for the Generation of Program Test Data". *IEEE Transactions on Computers* C-24 (1975), pp. 554–560 (cit. on p. 23).

[9] Paul C. Jorgensen. *Software Testing: a Craftsman's Approach*. Fourth. Auerbach Publications, 2013 (cit. on p. 39).

[10] Simon Kagstrom. *kcov - code coverage for fuzzing*. Version 4.00. May 21, 2020 (cit. on p. 49).

[11] Mikhail Kalkov and Dzmitry Pamakha. "Code coverage criteria and their effect on test suite qualities". In: 2013 (cit. on p. 20).

[12] 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).

[13] Michael Kerrisk. *gcov manual pdf*. Version 1st edition. Aug. 27, 2021 (cit. on p. 48).

[14] 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).

[15] Peter Liggesmeyer. *Software-Qualität - testen, analysieren und verifizieren von Software*. Spektrum Akadem. Verl., 2002 (cit. on pp. 7–9, 15, 16).

[16] Lu Luo. "Software Testing Techniques". In: 2001 (cit. on p. 6).

[17] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations*. Heidelberg: dpunkt., 2005 (cit. on p. 39).

[18] *Standard Commands for Programmable Instruments: (SCPI).. Command reference*. Bd. 2. SCPI Consortium, 1993. URL: https://books.google.at/books?id=mzZpmwEACAAJ (cit. on p. 35).

[19] Kuo-chung Tai. "Program Testing Complexity and Test Criteria". *IEEE Transactions on Software Engineering* SE-6 (1980), pp. 531–538 (cit. on p. 20).

[20] Jeff Tian. *Software quality engineering - testing, quality assurance, and quantifiable improvement*. Wiley, 2005 (cit. on p. 15).

Online sources

[21] Linux-Kernel team. *LCOV user Manual*. 2019. URL: http://ltp.sourceforge.net/coverage/lcov/lcov.1.php (visited on 07/27/2022) (cit. on p. 49).