Masterarbeit

Jan Heimann

Testgetriebene Entwicklung von eingebetteter Software bei simultaner Entstehung der Zielhardware

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Hamburg University of Applied Sciences
Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

Jan Heimann

Testgetriebene Entwicklung von eingebetteter Software bei simultaner Entstehung der Zielhardware

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Henning Dierks
Zweitgutachter: Prof. Dr. Kristina Schädler

Eingereicht am: 22. Dezember 2020

Jan Heimann

Thema der Arbeit

Testgetriebene Entwicklung von eingebetteter Software bei simultaner Entstehung der Zielhardware

Stichworte

Testgetriebene Entwicklung, Eingebettete Systeme, Dual-Targeting

Kurzzusammenfassung

Die testgetriebene Entwicklung ist eine Arbeitsweise der Softwareentwicklung, die es vorsieht, den Code inkrementell zu schreiben. Es wird zunächst ein Testfall formuliert, der die Software auf ein bestimmtes Verhalten überprüft. Anschließend wird gerade ausreichend Code implementiert, um den Testfall erfolgreich zu bestehen. Diese Arbeit untersucht, wie die testgetriebene Entwicklung für eingebettete Software eingesetzt werden kann, ohne die finale Zielhardware zum Ausführen der Tests bereitstehen zu haben. Dafür werden drei Test-Frameworks auf ihre Eignung hin untersucht. Anschließend werden Methoden, wie das Dual-Targeting und der Einsatz von Doubles und Mock Objekten vorgestellt, gefolgt von einer Betrachtung der Auswirkungen des Ansatz auf die Softwareentwicklung. Es wird eine Toolchain beschrieben, die für den testgetriebenen Ansatz und die vorgestellten Methoden geeignet ist. Danach wird auf das Thema Automatisierung eingegangen und beschrieben, wie sich der Ansatz mithilfe von Continuous Integration für größere Projekte skalieren lässt. Abschließend wird die testgetriebene Entwicklung in den Kontext von agilen Vorgehensmodellen eingeordnet.

Jan Heimann

Title of Thesis

Test-driven development for embedded software while the target-hardware is simultaneously developed

Keywords

test-driven development, embedded systems, dual-targeting

Abstract

Test-driven development is a method of software development where the code is written incrementally. First a testcase is written which tests the software for a specific behaviour. Next, just enough logic is implemented for the test to pass successfully. This thesis examines how test-driven development can be used for embedded software under the assumption that the final target-hardware is not yet available for executing the tests. Three test-frameworks will be analyzed regarding their suitablity. Followed by an introduction of a set of methods which allow for testing without the hardware, including dual-targeting and the use of test-doubles and mock objects. A toolchain is proposed, which enables the application of the described methods and test-driven development. The impact of the approach on the software development itself is described. Automation of the workflow and scalability of the approch for more complex projects gets explored. Finally, the test-driven development is put into the context of agile methodologies.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADC Analog Digital Converter.

CAN-Bus Central Area Network Bus.
CD Continuous Delivery.
CI Continuous Integration.
CPU Central Processing Unit.

DAC Digital Analog Converter.
DIP Dual-Inline Package.
DMA Direct Memory Access.

FPU Floating Processing Unit.

GPIO General Purpose Input/Output.

HAL Hardware Abstraction Layer.

I/O Input-Output.
I2C Inter-Integrated Circuit.
IC Integrated Circuit.
IDE Integrated Development Environment.
IoT Internet of Things.

JTAG Joint Test Action Group.

MMU Memory Management Unit.

OOP Objektorientierte Programmierung.
OSAL Operating System Abstraktion Layer.

PLL Phase Locked Loop.

QA Quality Assessment.

RAM Random Access Memory.
RF Radio Frequency.
ROM Read-Only Memory.
RTOS Real-Time Operating System.

SoC System-on-a-Chip.
SPI Serial Peripheral Interface.
SWD Single Wire Debug.

TCP Transmission Control Protocol.
TDD Test-Driven Development.

UART Universal Asyncronuous Receiver Transmitter.
USB Universal Serial Bus.

XP Extreme Programming.

Listings

1 Einleitung

Eingebettete Systeme finden sich in sämtlichen Bereichen des täglichen Lebens und in vielen Industrieanwendungen. Beispiele reichen von Geräten in der Heimautomatisierung, über vernetzte Sensoren in Städten und Industrieanlagen, bis hin zu Teilsystemen in Fahrzeugen. Jeder Mensch interagiert täglich mit einer Vielzahl an eingebetteten Systemen. Charakteristisch für diese Systeme ist, dass sie für die Erfüllung eines bestimmten Zwecks ausgelegt sind. Dies wird meist durch eine Kombination aus dedizierter Hardware und einer darauf ausgeführten Software erreicht. Anders als bei einem Computer oder einem Smartphone, sind sie nicht dafür ausgelegt, generische Aufgaben erledigen zu können.
Aufgrund der hohen Verbreitung von eingebetteten Systemen und ihrer direkten Interaktion mit der Umwelt, stellt eine Fehlfunktion ein hohes potentielles Risiko dar. Zugleich wächst stetig die Komplexität der Anwendungen und somit auch die Komplexität der Software. Die Wahrscheinlichkeit für Softwarefehler, auch als Defekte bezeichnet, steigt dadurch. Auswirkungen von Defekten können katastrophale wirtschaftliche Folgen haben, oder im schlimmsten Fall sogar Menschenleben gefährden.
Software stellt oft ein vereinfachtes Modell der Realität dar. Leider sind es häufig nicht vorhersehbare Ausnahmebedingungen, die einen Defekt zum Vorschein bringen. Eine umfassende Abbildung der Realität ist aufgrund der beschränkten Ressourcen nahezu unmöglich. Diesem Problem kann mithilfe verschiedener Methoden entgegengewirkt werden. Die testgetriebene Entwicklung ist eine davon. Sie macht von Komponententests, auch *Unit Tests* genannt, Gebrauch. Ein Komponententest prüft einen für sich selbst stehenden, funktionalen Teil des Codes auf dessen korrekte Funktionsweise, bei einer bestimmten Kombination aus Parametern. Bei der testgetriebenen Entwicklung wird, wie der Name bereits andeutet, die Implementierung der Funktionalität durch die Komponententests getrieben. Es wird also zunächst der Testfall für eine bestimmte Anforderung formuliert und als Komponententest implementiert. Erst danach wird der eigentliche Code geschrieben. Durch die Kombination aus mehreren Tests für eine Code-Einheit, wird sichergestellt, dass sich die Software-Einheit sich so verhält, wie die Anforderungen es vorgeben.

Der Begriff *testgetriebene Entwicklung* etablierte sich, als Kent Beck das erste xUnit-Framework für die Programmiersprache Smalltalk schrieb. Die Idee, Tests zu schreiben bevor die Logik implementiert wird (*Test-First*), war zu dem Zeitpunkt nicht neu. Allerdings wird sie von der testgetriebenen Entwicklung durch das inkrementelle Vorgehen erweitert. Anstatt erst alle Testfälle zu schreiben, wird nur ein Test zur Zeit geschrieben und dann die Implementierung soweit voran getrieben, bis dieser Testfall erfüllt ist. Testgetriebene Entwicklung ist daher vor allem als eine Arbeitsweise der Softwareentwicklung anzusehen. Besonders in der agilen Softwareentwicklung wird die testgetriebene Entwicklung gerne als Maßnahme der Qualitätssicherung eingesetzt.

Mehrere Fallstudien im industriellen und akademischen Umfeld zeigen, dass die testgetriebene Entwicklung Defekte begrenzt und zu einer Qualitätssteigerung führen kann. So wurde bei einer Studie, durchgeführt mit vier Teams von Microsoft und IBM, festgestellt, dass eine Reduktion der Defekt-Dichte zwischen 40 % und 90 % stattfand [Nag+03]. Eine Umfrage mit Programmiererinnen und Programmieren ergab, dass ein Großteil der Probanden den Ansatz für sehr effektiv hält, da ein besseres Verständnis für die Anforderungen gefördert und der Debugging Aufwand signifikant reduziert wird [GW03]. Bei einer weiteren Untersuchung der Thematik bei IBM, kam man zu dem Schluss, dass der Ansatz Probleme bei der Integration von Software-Einheiten deutlich früher aufzeigte [MW03].
Auch für eingebettete Software ist die testgetriebene Entwicklung daher sehr interessant. Jedoch ergeben sich bei eingebetteten Systemen einige Problematiken, die für das Feld spezifisch sind. Eine besondere Herausforderung resultiert aus der engen Kopplung von der Software an ihre Zielhardware. Häufig existiert die Zielhardware zum Zeitpunkt der Softwareentwicklung noch nicht. Vielmehr wird sie simultan zur Software entwickelt. Das erschwert die testgetriebene Entwicklung, da die Komponenten-Tests nicht sofort auf der finalen Hardware ausgeführt werden können.

In dieser Arbeit wird untersucht, wie die testgetriebene Entwicklung für eingebettete Software adaptiert werden kann. Der Fokus liegt insbesondere auf den Strategien, welche es erlauben, die Software auch ohne die finale Zielhardware testgetrieben zu entwickeln. Es wird sich dabei auf die Programmiersprache C begrenzt, da diese die größte Relevanz für eingebettete Systeme hat. Zunächst werden die Grundlagen zu eingebetteten Systemen und testgetriebener Entwicklung behandelt, um die Restriktionen und Umgebungsbedingungen besser zu verstehen. Im Hauptteil folgt eine Untersuchung der einzelnen Punkte,

welche für die testgetriebene Entwicklung wichtig sind. Neben technischen Aspekten, wie den gängigen Test-Frameworks, der Toolchain und der Test-Automatisierung, werden verschiedene Test-Strategien anhand eines Beispiel-Projekts vorgestellt. Es wird zudem untersucht, inwiefern sich die testgetriebene Entwicklung auf die Software-Entwicklung und die Qualität des Codes auswirkt. Anschließend folgt eine Betrachtung, wie die testgetriebene Entwicklung in den Kontext von agilen Vorgehensmodellen einzuordnen ist. Im Fazit werden die Vor- und Nachteile des Ansatzes abschließend gegenübergestellt und bewertet.

2 Grundlagen Eingebettete Systeme

Dieses Kapitel behandelt die Grundlagen eingebetteter Systeme. Es wird zunächst auf die technischen Eigenschaften eingebetteter Systeme und den Entwicklungsprozess eingegangen. Gefolgt von einer Betrachtung der Programmiersprache C, sowie den Ressourcen und den Anforderungen, die für eingebettete Systeme zutreffen.

2.1 Definition und Eigenschaften

Der Begriff *eingebettete Systeme* ist nicht klar eingegrenzt. Um eine Grundlage für diese Arbeit zu schaffen, soll an dieser Stelle definiert werden, wie der Begriff im Folgenden zu verstehen ist.
Elicia White beschreibt eingebettete Systeme als rechnergestützte Systeme, die für eine spezielle Anwendung entwickelt sind [Whi12, S. 1]. Laut Jack Ganssle haben alle eingebetteten Systeme gemein, dass das Endprodukt kein Computer ist [Gan08, S. 4]. Dies sind die beiden wichtigsten Eigenschaften, mit denen in dieser Arbeit eingebettete Systeme charakterisiert werden:

1. Ein eingebettetes System ist für die Erfüllung einer bestimmtem Anwendung ausgelegt.
2. Ein eingebettetes System ist keine Plattform, die generelle Aufgaben ausführen kann.

Zur Erfüllung seiner Aufgabe führt ein eingebettetes System Software aus. Diese Software läuft auf einer dedizierten Hardware, welche eigens für den jeweiligen Anwendungszweck spezifiziert und entwickelt wurde. Die möglichen Anwendungsgebiete sind sehr vielseitig. Möglich ist dabei alles von einem vernetzen Sensor, Haushaltsgeräten, Elektrowerkzeugen, bis hin zu Teilsystemen in einem Kraftfahrzeug. Ein Mobiltelefon wird in dieser Arbeit nicht als eingebettetes System angesehen, da es zum Ausführen einer generellen Software genutzt werden kann. Auch werden keine Systeme betrachtet, die auf einem

Embedded Linux basieren, da die vorgestellten Methoden dafür nicht unbedingt zutreffen.
Abbildung 2.1 zeigt ein Modell der verschiedenen Ebenen, welche in einem eingebetteten System beteiligt sind. Die unterste Ebene wird durch die Hardware gebildet. Sie ist in jedem eingebetteten System zu finden. Darauf bauen die zwei Software Ebenen auf. Die System Software Ebene kann neben einem Betriebssystem auch sogenannte *Middleware* enthalten. Anwendungs-Code ist in der Anwendungssoftware-Ebene zu finden. Sie ist in jedem Projekt vorhanden. Optional hingegen ist die System Software Ebene. Systeme, welche auf ein Betriebssystem verzichten, werden oft mit dem Begriff *Bare Metal* beschrieben.

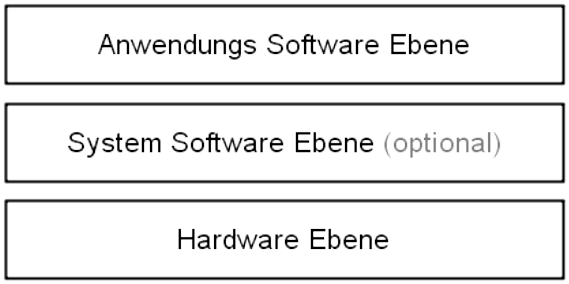

Abbildung 2.1: Modell-Darstellung der Architektur eines eingebetteten Systems.

Die Hardware, im Folgenden auch als *Zielsystem* bezeichnet, kann bei jedem Projekt komplett unterschiedlich aussehen. Das zentrale Element wird dabei stets durch einen Mikroprozessor gebildet, welcher die Central Processing Unit (CPU) enthält. Ergänzt wird der Prozessor durch den Read-Only Memory (ROM), typischerweise ein nicht-flüchtiger Speicher, und durch den Random Access Memory (RAM), ein flüchtiger Speicher. Verbunden werden diese Speicherbausteine mit dem Prozessor durch eines Bus. Sind die Speicherbausteine mit dem Prozessor auf einem Chip integriert, so wird dieser als Mikrocontroller bezeichnet. Der Prozessor interagiert neben den Speicherelementen auch noch mit verschiedenen anderen Peripherien. In Abbildung 2.2 ist ein Block Diagramm eines Mikrocontrollers inklusive Peripherie-Elementen zu sehen. Welche Peripherie-Elemente ein Mikrocontroller bietet, kann sehr unterschiedlich ausfallen. Häufig werden sie speziell für bestimmte Anwendungsgebiete zugeschnitten und enthalten die dafür notwendigen Peripherien.

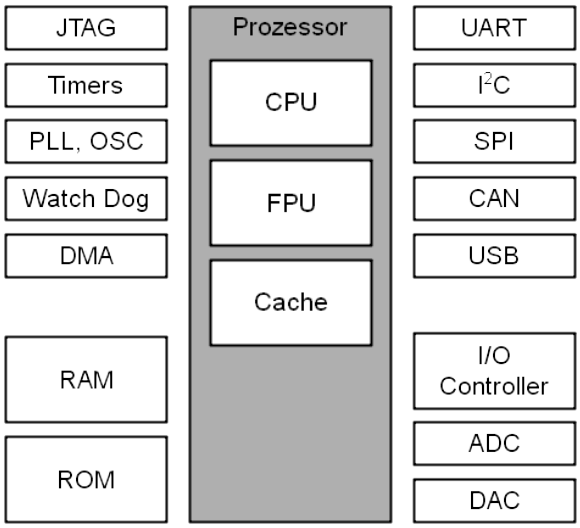

Abbildung 2.2: Exemplarisches Mikrocontroller Block Diagramm.

Neben der CPU könnte der Prozessor eine Floating Processing Unit (FPU) oder ein Cache haben. Es stehen meist eine Vielzahl an Kommunikations-Schnittstellen, wie UART, I2C, SPI, etc., zur Verfügung. Sie dienen der Kommunikation mit anderen Recheneinheiten, zum Beispiel Smart Sensoren oder weiteren Prozessoren. Ein wichtiges Element bildet der I/O Controller. Über ihn lassen sich digitale Ein- und Ausgänge auslesen bzw. ansteuern. Auf diese Weise kann ein System mittels Aktoren mit seiner Umwelt interagieren oder mittels Sensoren Daten über die Umwelt einlesen. Weitere Bausteine stellen dem Mikrocontroller zusätzliche Funktionalitäten zur Verfügung. So können mittels Timern Zeitintervalle gemessen werden oder über den Direct Memory Access (DMA) Baustein Daten direkt in den Speicher geschrieben werden, ohne den Prozessor zu belasten. Eine JTAG-Schnittstelle kann zum Testen, Debuggen oder Programmieren des Controllers dienen. Die Peripherie kann von der Software aus über Kontroll- und Steuerregister konfiguriert und angesteuert werden. Diese Register haben eine Speicheradresse im adressierbaren Bereich des Prozessors und werden daher auch als *Memory-Mapped Register* bezeichnet. Der Zugriff erfolgt über Zeiger-Variablen [Bar07, S. 185].
Aufgrund des starken Preisdrucks werden bevorzugt mehrere Elemente auf einem Integrated Circuit (IC) kombiniert, auch System-on-a-Chip (SoC) genannt. Ein Beispiel wäre hierfür ein Chip, der einen Mikrocontroller mit einen RF-Transceiver, für die drahtlose Kommunikation, auf einem IC vereint.

Ein Produkt besteht, je nach Komplexität, aus einem System oder mehreren untereinander verbundenen Teilsystem. Diese werden gewöhnlich auf Platinen umgesetzt, welche neben dem Mikrocontroller noch eine Auswahl an weiteren Bauteilen enthalten können. Wie man sieht, hat die Hardware, welche in eingebetteten Systemen zu finden ist, eine enorme Bandbreite. Von einer Platine mit nur wenigen physikalischen Bauteilen, bis hin zu mehreren Teilsystemen mit hunderten an Bauteilen, ist hier alles möglich. Gemein haben diese Systeme, dass sich auf ihren Anwendungszweck speziell zugeschnitten werden und dabei möglichst Ressourcen schonend sein müssen.

2.2 Entwicklungsprozess

Bei der Entwicklung eines eingebetteten Systems spielen mehrere Bereiche eine Rolle. Im Folgenden sollen diese vorgestellt werden, um die einzelnen Zuständigkeiten und Tätigkeiten zu etablieren.

2.2.1 Rollen in der Produktentwicklung

Je komplexer ein System bzw. die Anwendung ist, desto größer ist auch das Entwicklungsteam. Mit wachsender Anzahl an Teammitgliedern, steigt meist auch die Ausdifferenzierung der Rollen. Die nachfolgend vorgestellten Rollen beschränken sich auf die für diese Thesis relevanten Tätigkeiten. In der Realität gibt es oft noch viele weitere, anwendungsspezifische Rollen.
Grob lassen sich die Tätigkeiten in das Projektmanagement, die Hardware-Entwicklung, die Software-Entwicklung und das Quality Assessment (QA) einteilen. Das Projektmanagement ist für die Anforderungs-Analyse, die Produktspezifikation, die Zeitplanung und die Einteilung von Ressourcen zuständig. Das Software-Team kümmert sich um die Entwicklung der Software. Dementsprechend entwickelt das Hardware-Team die Hardware-Komponenten. Das Quality Assessment verifiziert, dass die Anforderungen der Spezifikation erfüllt wurden und validiert das System auf Einhaltung der Anforderungen der Anwendung. Darüber hinaus suchen Tester in Software und Hardware nach Schwachstellen und berichten diese an die entsprechenden Entwicklungs-Teams.
Die im Folgenden aufgelisteten Rollen sind besonders für die Softwareentwicklung relevant [Bec05, S. 73 ff.].

- Das **Projektmanagement** ist zuständig für die Projektplanung und die Kommunikation zwischen Team, Kunden, Lieferanten und anderen Organisationseinheiten.
- Das **Produktmanagement** gibt die zukünftige Richtung für ein Projekt vor und legt die entsprechenden Anforderungen fest.
- Die **Software Architektur** umfasst das Software Design auf Basis der Spezifikation.
- Die **Entwicklung** umfasst die Implementierung des Software Entwurfs und die Instandhaltung des Quellcodes.
- Die **Konfigurationsverwaltung** ist zuständig für das Integrieren von Software-Einheiten und die Bereitstellung der Software.
- Die **technische Redaktion** erstellt Dokumentationen, Anleitungen und weitere Materialien für die Kunden des Produkts.
- Das **Qualitätsmanagement (QA)** verifiziert und validiert das Produkt.

Die Entwicklung ist eng mit den Software-Architektur und dem Qualitätsmanagement verbunden.

2.2.2 Lebenszyklus der Produktentwicklung

Der Lebenszyklus der Produktentwicklung eines eingebetteten Systems wird durch das angewandte Vorgehensmodell bestimmt. Mit der Wahl des Vorgehensmodells können einzelne Aspekte, wie z.B. die Entwicklungszeit oder die funktionale Sicherheit priorisiert werden. Häufig wird das Vorgehensmodell allerdings entweder durch das Unternehmen oder durch eine in dem Gebiet zuständige Norm vorgegeben. Eines der ersten verbreiteten Vorgehensmodelle war das das Wasserfall-Modell.

Abbildung 2.3 zeigt die einzelnen Phasen des Wasserfall-Modells. Namensgebend ist die Eigenschaft, dass eine Phase jeweils in die nächste übergeht bis hin zur Fertigstellung des Projekts. Der Fokus liegt auf der strikten Spezifikation des Produkts bereits zum Anfang des Projekts. Voraussetzung ist, dass die Anforderungen sich klar definieren lassen [Pat06, S. 33 f.]. Der Ablauf ist hier sehr stringent strukturiert und lässt sich daher gut dokumentieren und später auch wieder nachvollziehen. Es ist eigentlich nicht vorgesehen, dass zu einer vorherigen Phase zurückgesprungen werden kann. Problematisch ist hierbei, dass nur schwer auf Änderungen reagiert werden kann, weshalb viele weiterentwickelte Varianten des Modells zusätzlich den Rücksprung auf die vorherigen Schritt zulassen. Ein Beispiel hierfür ist in Abbildung 2.4 dargestellt. Bei Reviews nach Abschluss eines

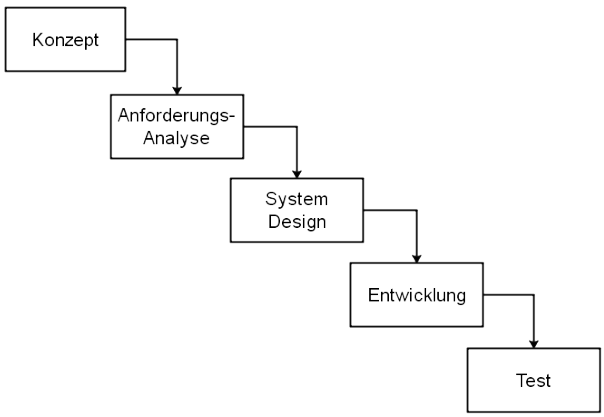

Abbildung 2.3: Die Phasen des Wasserfall-Modells.

Schrittes werden hier Schwachstellen festgestellt, die in der nächsten Iteration in dem Entwurf beseitigt werden können. Wichtig ist jedoch anzumerken, dass sich die diskreten Schritte nicht zeitlich überschneiden.

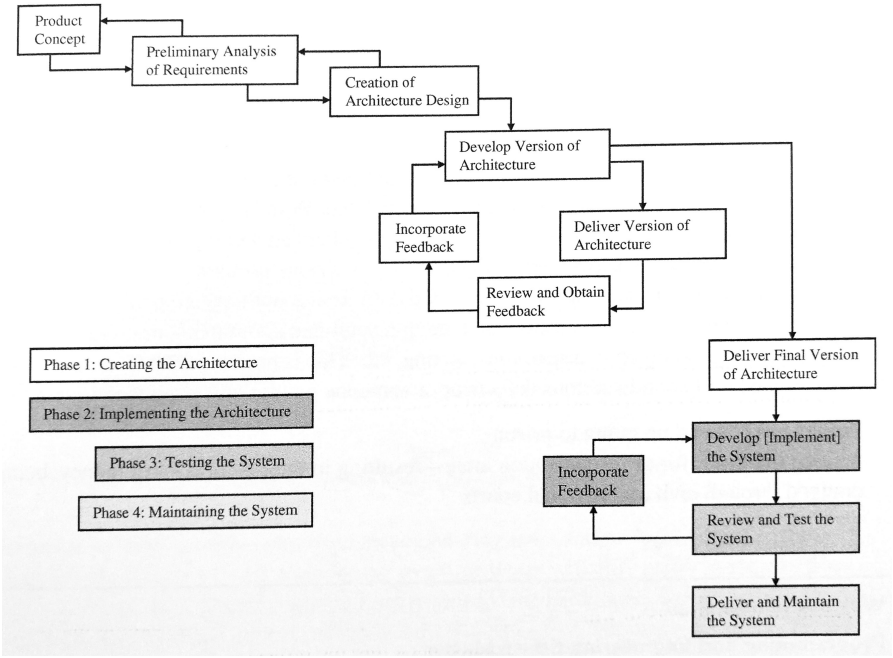

Abbildung 2.4: Durch Iterationen erweitertes Wasserfall-Modell [Noe13, S. 13].

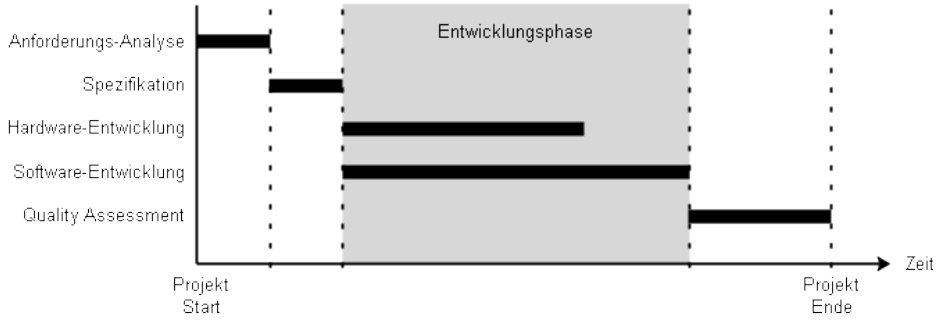

Abbildung 2.5: Zeitlicher Verlauf eines Projekts mit dem Wasserfall-Modell.

Eine Schwachstelle dieses Ansatzes wird deutlich, wenn man den Projektverlauf zeitlich betrachtet (Abbildung 2.5). Die Testphase bzw. das Quality Assessment ist die letzte Phase im Projekt. Sie läuft damit Gefahr durch zeitliche oder finanzielle Engpässe, welche in den vorherigen Phasen entstanden sind, im ihrem Umfang eingegrenzt zu werden. Besonders in Anwendungen, die hohe Anforderungen an die funktionale Sicherheit, die Zuverlässigkeit oder die Verfügbarkeit stellen, ist dies problematisch. Hinzu kommt, dass das Finden und Beheben von hier gefundenen Defekten deutlich teurer ist, als ein Defekt, der bereits in der Entwicklungsphase auffällt. Modernere Ansätze, wie das *Agile Development*, versuchen diesem Problem entgegenzuwirken, indem sie die Tests und das QA bereits parallel zur Entwicklung ausführen. Das Vorgehen basiert hier auf vielen, kurzen Iterationsschritten anstelle diskreter Abschnitte. Eine vertiefende Betrachtung des Agile Developments auf seine Eignung für eingebettete Systeme ist im Kapitel 4.7 zu finden.

Aus der Abbildung 2.5 lässt sich ein weiterer für diese Thesis zentraler Punkt ableiten: Die Entwicklung der Software und Hardware erfolgt simultan. Häufig wird die Zielhardware erst in einem stark fortgeschrittenen Stadium des Projekts finalisiert. Aus zeitlichen Gründen ist es nicht möglich, die Software-Entwicklung erst an diesem Punkt starten zu lassen.

Basierend auf dem Wasserfall-Modell wurde speziell für die Software-Entwicklung das V-Modell entworfen (Abbildung 2.6). Von der Anforderungs-Analyse bis hin zur letztendlichen Implementierung der Funktionalität steigt mit jedem Schritt der Detaillierungsgrad an. Auch wird hier häufig ein Rücksprung zum vorherigen Schritt vorgesehen, um Ände-

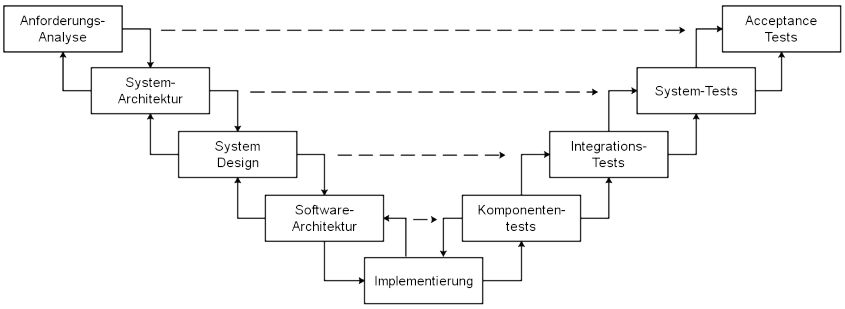

Abbildung 2.6: Darstellung des V-Modells.

rungen die bei den Reviews vorgeschlagen werden einfließen zu lassen. Es bietet die selben Vorteile des Wasserfall-Modells, legt darüber hinaus aber einen Fokus auf die Testbarkeit des Produkts. Wie in der Darstellung durch die gestrichelten Pfeile angedeutet, ergeben sich aus jedem Schritt der linken Seite die Testfälle der rechten Seite. Die auszuführenden Tests sind somit früh im Projekt klar definiert. Durch die Unterteilung der Test in die verschieden Abstraktions-Ebenen kann eine umfassend Überprüfung des *System Under Test* sichergestellt werden. Die Komponententests stellen eine korrekte Funktionalität der einzelnen Software-Module sicher. Die Integrations-Tests ermöglichen eine Betrachtung der Module im Zusammenspiel miteinander. Die System-Tests betrachten das System als Ganzes aus der Sicht des Anwenders und die Acceptance Test stellen zuletzt sicher, dass das System die Anforderungen des Kunden zufrieden stellt.

2.3 Eingebettete Software mit C

An dieser Stelle wird die Programmiersprache C genauer vorgestellt. Wie bereits erwähnt, ist sie bei eingebetteten Systemen die dominierende Programmiersprache. Die stärksten Argumente für C sind ihre Tauglichkeit zur Umsetzung sehr ressourcenschonende Anwendungen und die direkte Interaktionsmöglichkeit mit der Hardware. Jedoch bringt diese Eigenschaft auch Nachteile mit sich, welche die testgetriebene Entwicklung erschweren können. Im Folgenden wird die Sprache genauer vorgestellt und auf ihre potentiellen Schwachstellen eingegangen.

C wurde Anfang der 1970er von Dennis Ritchie in den Bell Labs als Sprache für die Implementierung von Unix Betriebssystemen in Leben gerufen [Rit93]. Seit dem wurde sie durch die Unterstützung von verschiedenen Compilern für immer mehr Systeme adaptiert. Insbesondere Mikroprozessor-Architekturen bieten meist Unterstützung für C an. Bis heute ist sie die am meisten verwendete Sprache für eingebettete Systeme.
Bei C handelt es sich um eine prozedurale Programmiersprache. Als solche ermöglicht sie es, das Problem in mehrere kleine Prozeduren bzw. Funktionen zu unterteilen. Im Gegensatz zu jüngeren Sprachen, wie C++, Java oder C#, gehört sie nicht zu den Objekt-Orientierten Sprachen. Diese basieren auf der Idee, eine Abstraktion der Realität in Form von Klassen bzw. Objekten zu beschreiben. Dennoch ist es auch in C möglich, Daten und Funktionen logisch zu kapseln. Eine solche Einheit, auch Modul[1] genannt, wird in zwei Dateien implementiert. Das ist zum einen die Header-Datei, in der Teile des Moduls auf die an anderer Stelle zugegriffen werden muss enthalten sind. Dabei kann es sich um Funktions-Prototypen, Typdefinitionen oder globale Variablen handeln. Die Source-Datei enthält die eigentliche Implementierung der öffentlichen Funktion, sowie Variablen, Typdefinitionen und Funktionen, die nur innerhalb der Source-Datei sichtbar sind. Deutlich unterscheidet sich C auch von anderen Hochsprachen durch ihre Beschränkung auf primitive Datentypen. Sie bietet allerdings die Möglichkeit, mittels dem Sprachkonstrukt `struct` mehrere Variablen logisch zusammenzufassen.
Im Vergleich zu Objekt-Orientierten Sprachen stehen in C wesentlich weniger Features, welche die Software-Entwicklung erleichtern, zur Verfügung. Es gibt weder eine eingebaute Ausnahmebehandlung noch einen Garbage Collector. Dennoch hat es bisher kaum eine andere Sprache geschafft, sich auf dem Gebiet der eingebetteten Software in einem ähnlichen Maße zu etablieren. Ausschlaggebend sind dafür zwei Gründe [Bar07, S. 33 f.]:

1. C ist im Vergleich sehr ressourcensparend im Hinblick auf Speicherbedarf und Performance und bietet dennoch den Vorzug der guten Verständlichkeit einer Hochsprache. Effizienter lässt sich ein Prozessor nur bei direkter Verwendung seines Anweisungssatzes (Assembly) programmieren. Dabei steigt allerdings die Entwicklungszeit deutlich an.
2. C erlaubt eine direkte Adressierung des Speichers. Dies ist bei eingebetteten Systemen unverzichtbar, da zur Steuerung von Peripherie oder Features des Prozessors

[1] Dieser Begriff wird hier bewusst eingesetzt, um auch sprachlich auszudrücken, dass es sich eben nicht um eine Klasse handelt.

häufig direkt auf Register zugegriffen werden muss. Sie ist damit eine sehr hardwarenahe Sprache.

Der Sprachumfang von wird seit ihrer ersten Veröffentlichung bewusst auf ein Minimum begrenzt. Der Abstraktionsgrad soll gering bleiben, um die Nähe der Sprache zum Zielsystem zu erhalten. Grundsätzlich gibt es keine Einschränkungen für Anwendungen, die mit C implementiert werden können, jedoch erfordert dies oft mehr Aufwand bei der Implementierung. Brian Kernighan und Dennis Ritchie führen dazu das Folgende an [KR88, S. 3].

> *„C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.“*

Es ist ein Leichtes eine C Anwendung zu schreiben, die beispielsweise durch zu unvorsichtigen Umgang mit Zeigern eine unvorhergesehenes Verhalten produziert. Der Compiler verhindert dies in keinster Weise. Aus diesem Grund ist es in der Industrie üblich, durch ergänzende Software-Werkzeuge sicherzustellen, dass Best Practices und Coding Guidelines befolgt werden.

2.3.1 Die C Build-Pipeline

Eine der Besonderheiten bei der Entwicklung von eingebetteter Software ist der Einsatz von Cross-Compilern. Sprich ein Compiler, welcher den Quellcode für eine andere Hardware-Architektur übersetzt als das System, auf dem entwickelt wird. Abbildung 2.7 stellt diesen Zusammenhang grafisch dar. Auf dem Entwicklungssystem wird der Code, meist mit Hilfe einer Integrierten Entwicklungsumgebung, geschrieben. Der Cross-Compiler übersetzt dann in eine ausführbare Datei (engl. Executable). Dieses Executable besteht aus einer Folge der für die Zielhardware spezifischen Anweisungen. Um die Anwendung ausführen zu können, muss sie zunächst auf die Zielhardware übertragen werden. Dazu kommt ein Programmiergerät zum Einsatz, das den Inhalt des Executables in den Programmspeicher lädt. Nun kann die Anwendung auf der Zielhardware ausgeführt werden. Mittels eines Debuggers ist es möglich, vorausgesetzt die Zielhardware bietet dafür eine Schnittstelle, einen Einblick in die Anwendung zur Ausführungszeit zu bekommen.

Der Cross-Compiler ist speziell für den jeweiligen Prozessor ausgelegt. Verwendet man nur den Standard-Sprachumfang von C ist der Quellcode zwar in der Theorie unabhängig

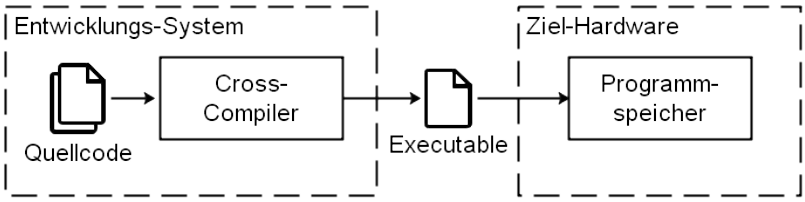

Abbildung 2.7: Entwicklungs-System und Zielsystem.

von der Hardware und vom Compiler, jedoch ist nicht sichergestellt, dass der selbe Code sich auf unterschiedlichen System auch identisch verhält. Beispielsweise zeigt sich bei der Verwendung der Standard-Datentypen wie `int` und `uint`, dass sie die zugrundeliegende Verarbeitungsbreite des Zielsystems annehmen. In eingebetteten Systemen ist von 8-Bit bis hin zu 64-Bit Architekturen alles vertreten.
Ein gutes Verständnis der Prozessor-Architektur und dem entsprechenden Cross-Compiler helfen, unerwartetes Verhalten der Anwendung zu verhindern.
Die Abbildung 2.8 zeigt einen typischen Build-Vorgang für ein C Projekt. Er besteht aus drei Schritten. Zunächst werden die Source-Dateien, an den Preprocessor übergeben. Er fügt den Inhalt von eingebundenen Header-Dateien in die jeweilige Datei ein, führt Preprocessor-Macros aus und entscheidet mittels Preprocessor-Flags, welche Code-Abschnitte tatsächlich beim Übersetzen einbezogen werden. Das Resultat ist eine *Übersetzungseinheit*. Sie enthält keine Preprozessor-Anweisungen mehr und auch die Kommentare wurden entfernt. Die meisten Compiler erlauben Zugriff auf diese Übersetzungseinheiten, um die korrekte Ausführung des Preprozessors verifizieren zu können. Im zweiten Schritt erfolgt die eigentlich Übersetzung des Quellcodes. Zunächst wird der Code in die Assembly-Instruktionen der entsprechenden Ziel-Architektur übersetzt. Es folgt eine weitere Übersetzung des Assembly Codes in den tatsächlichen Maschinencode. Jede Übersetzungseinheit resultiert in einer sogenannten Objekt-Datei. Sie ist nicht mehr direkt für Menschen lesbar [Ami19, S. 63 ff.]. Daten und Anweisungen werden an dieser Stelle in Speicher-Segmente eingeteilt. Diese sind allerdings noch nicht festen Adressbereichen zugeordnet. Dies wird auch als *Object-Placement* bezeichnet und ist ein wichtiges Feature für eingebettete Systeme, da sie häufig über mehrere Speichertypen verfügen und für viele Anwendungen eine bestimmte Positionierung der Segmente im Speicher wichtig ist [Noe13, S. 550]. Die tatsächliche Platzierung der Segmente erfolgt schließlich im dritten Schritt. Dem Linker wird mit einer Linker-Datei die genaue Speicheraufteilung vorgegeben. Darüber hinaus bindet er auch die System-Bibliotheken ein. Das Ergebnis ist

ein Executable. Die meisten Build-Pipelines kombinieren, soweit nicht anders spezifiziert, die ersten beiden Schritte.
Ein Beispiel für eine Build-Pipeline mit einem Cross-Compiler ist der GNU Arm Embedded Toolchain[2], welche für eine Reihe Prozessoren, die auf einer ARM 32-Bit Architektur basieren, einsetzbar ist.

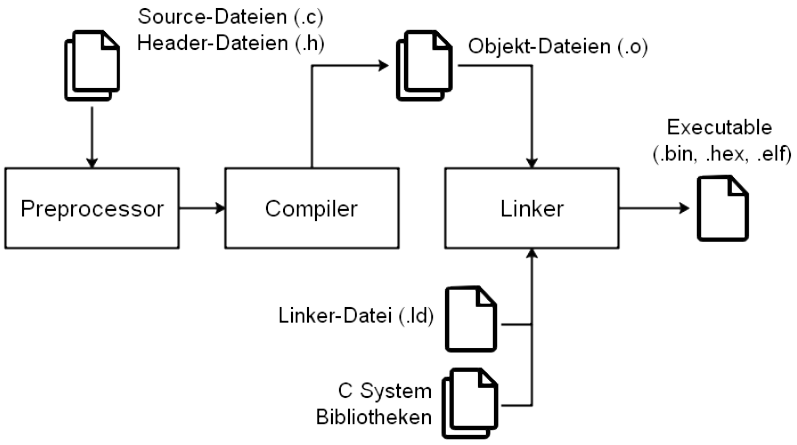

Abbildung 2.8: Build-Vorgang für ein eingebettetes System.

2.4 Ressourcen und Anforderungen

Bei der Auslegung eines eingebetteten Systems spielen verschiedene Ressourcen eine Rolle. Diese haben sowohl auf den Hardware- als auch auf den Software-Entwurf einen Einfluss. Sie lassen sich in die drei grundlegenden Kategorien Speicherbedarf, Rechenleistung und Energiebedarf einteilen. Im Folgenden werden diese Kategorien und ihre Auswirkungen auf den Entwicklungsprozess vorgestellt.

Speicherbedarf

In eingebetteten Systemen sind meist zwei Typen von Speicher vorhanden. Zum einen der ROM. Hier wird der Code, die Initialisierungswerte für statische Variablen und Konstan-

[2] `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm`

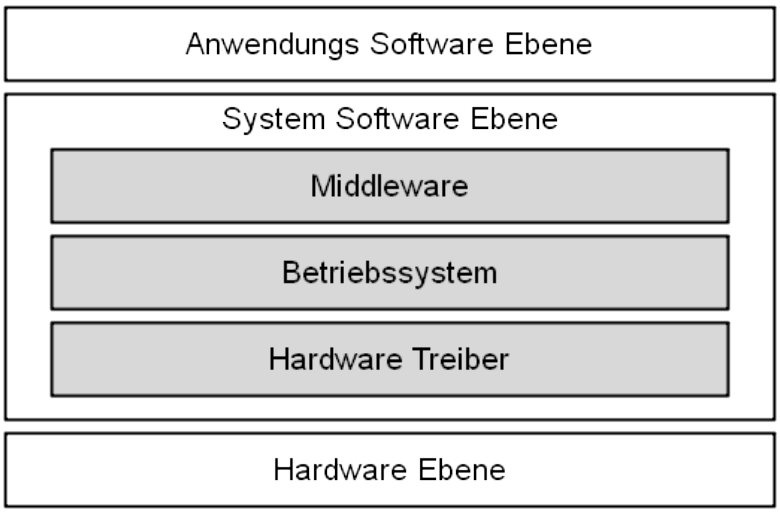

Abbildung 2.9: Die Software Ebenen eines eingebetteten Systems.

ten gespeichert. Zum anderen der RAM. Er enthält zur Ausführungszeit den Stack und den Heap. Üblich sind bei Mikrocontrollern Speicherkapazitäten für beide Speichertypen von einigen Kilobytes bis hin zum einstelligen Megabyte Bereich. Physikalischer Speicher ist einer der wichtigsten Faktoren für den Stückpreis von Mikrocontrollern. Geschuldet ist dies dem enormen Platzbedarf von Speicherelementen auf dem Chip. Vergleicht man beispielsweise zwei Versionen[3] der i.MX RT-Serie von NXP miteinander wird deutlich, wie groß der Einfluss der Speicherkapazität auf den Stückpreis eines Mikrocontrollers ist. Die beiden Varianten haben neben der Speicherkapazität eine nahezu identische Spezifikation. Bei einer Anzahl von Tausend Stück liegt der Preisunterschied bei 1,28 € pro Einheit. Geht man davon aus, dass diese Differenz bei steigender Abnahmezahl konstant bleibt, so würde sich bei einer Anzahl von einer Million Stück, was eine durchaus realistische Zahl für eingebettete Systeme ist, eine Kostendifferenz von über einer Million Euro ergeben. Es wird klar, dass es eine Anforderung an eingebettete Software ist, mit möglichst wenig Speicher auszukommen. Dies gilt insbesondere, da sich gesparte Kosten für Hardware-Bauteile häufig unmittelbar in Gewinn für ein Unternehmen übersetzen lassen. Idealerweise nutzt die Anwendung, die letztendlich in Produktion geht, den vor-

[3] MIMXRT1051CVL5A (96 KB ROM und 512 kB RAM) und MIMRT1061CVL5A (128 kB ROM und 1 MB RAM)

handenen Speicher zu einem Großteil, unter Vorbehalt einer bestimmten Kapazität als Puffer zur Sicherheit, aus.

Rechenleistung

Während moderne Prozessoren für PCs und Server mit einigen Gigahertz getaktet werden, bewegen sich Mikroprozessoren im Megahertz Bereich. Auch hier lässt sich ein direkter Zusammenhang zwischen dem Prozessortakt und den Kosten eines Prozessors erkennen. Je schneller der Prozessor desto teurer. Da eingebettete Systeme unmittelbar mit ihrer Umwelt agieren, müssen sie entsprechende Echtzeit-Bedingungen erfüllen. Kann die Anwendung nicht im vorgegebenen Zeitraum auf ein Event reagieren, hat man entweder die Option, auf einen schnelleren Prozessor umzusteigen oder die Ausführungsgeschwindigkeit des Codes zu optimieren. Ersteres ist aus Kostengründen meist keine Option.

Energiebedarf

Viele eingebettete Systeme werden mit einer Batterie oder einem Akku mit Energie gespeist. Sie sollen dabei möglichst lange mit der vorhandenen Kapazität auskommen. Nicht nur das Hardware-Design hat Einfluss auf den Energiebedarf eines Systems. Zum großen Teil wird er durch die Auslastung des Prozessors bestimmt. Daraus ergibt sich die Anforderung für die Software, mit einem begrenzten Kapazität möglichst lange auszukommen. Muss der Energiebedarf eines Produkts reduziert werden, so kann dies durch eine kleinere Taktfrequenz erreicht werden [Bar07, S. 376]

Diese drei Ressourcen werden meist zum Projektbeginn bereits festgelegt. Sie sind wichtige Eckdaten für das Hardware-Design. Die Komponenten werden so ausgewählt, dass sie die Anforderungen erfüllen und nicht unnötig mehr Ressourcen bieten, die von der Anwendung nicht benötigt werden [Bar07, S. 372]. Das ausschlaggebende Argument sind hier die Kosten. Einsparung durch Komponenten schlagen sich meist unmittelbar in Gewinn nieder. Problematisch ist hierbei, dass eine Einschätzung der benötigten Ressourcen einer Software-Anwendung häufig nicht genau vorgenommen werden kann. Dies wird dadurch erschwert, dass je nach Auslegung der Software, die Schonung bestimmter Ressourcen priorisiert werden kann. Eine solche Priorisierung kann sich negativ auf die anderen Ressourcen auswirken. Beispielsweise kann die Ausführungszeit verringert werden, indem eine häufig aufgerufene Funktion mit `inline` gekennzeichnet wird. Dies veranlasst den

Compiler diesen Code-Abschnitt an jeder Stelle einzufügen, an der die Funktion aufgerufen wird. Die Konsequenz ist allerdings das der Code im Gegenzug mehr Platz im ROM benötigt [Bar07, S. 363]. Eine Software die gut testbar ist und einen Fokus auf die Software Qualität legt, hat einen größeren Bedarf an den oben genannten Ressourcen. Es gilt also bei der testgetriebenen Entwicklung, den Ressourcenbedarf im Auge zu behalten und gegebenenfalls Kompromisse zu finden.

3 Grundlagen Testgetriebene Softwareentwicklung

In diesem Kapitel werden zunächst die grundlegenden Konzepte der Testgetriebenen Entwicklung vorgestellt. Darauf folgt eine Betrachtung des aktuellen Stands der Technik. Schließlich wird der Bogen zur Testgetriebenen Entwicklung von eingebetteter Software gespannt in Form einer Analyse der besonderen Herausforderungen, die sich hier ergeben.

3.1 Prinzipien und Vorgehen

Testgetriebene Entwicklung, im Englischen auch als *Test-Driven Development (TDD)* bezeichnet, ist eine Methodik, welche von Software-Entwicklern bei der Implementierung von Software eingesetzt werden kann. Sie beruht dabei auf den folgenden fünf Grundprinzipien [Lab17, S. 129]:

1. Tests werden geschrieben, bevor der Code geschrieben wird.
2. Es wird gerade soviel Code geschrieben, der benötigt wird, um den Test zu bestehen.
3. Es werden nur Tests geschrieben die den Code testen.
4. Es wird in kurzen Zyklen gearbeitet.
5. Der Code wird ständig refaktorisiert.

Der Begriff *Test* bezieht sich im Kontext der testgetriebenen Entwicklung stets auf einen Komponententest, auch *Unit Test* genannt. Er lässt sich wie folgt definieren [LL13, S. 491].

> *„Bei diesem Test werden einzelne, überschaubare Programmeinheiten getestet, je nach verwendeter Programmiersprache also z.B. Funktionen, Unterprogramme oder Klassen.“*

Die Verantwortung für das Formulieren und Implementieren der Komponententests liegt bei den Entwicklerinnen und Entwicklern selbst [Som18, S. 70]. Da diese intime Kenntnisse über die Details des Quellcodes haben, handelt es sich bei einem Komponententest stets um einen White Box Test. Der Test betrachtet die Software zur Ausführungszeit, daher handelt es sich um einen dynamischen Test. Ziel eines Komponententests ist es, zu verifizieren, dass sich eine Programmiereinheit unter bestimmten Bedingungen auf die bei der Implementierung vorgesehene Weise reagiert. Dieses vorgesehene Verhalten ist dabei die Interpretation der Spezifikation und des System Entwurfs. Zu beachten ist, dass diese Interpretation nicht unbedingt auch den Ansprüchen des Kunden oder anderen Stakeholdern entspricht.
Meist werden Komponententests in drei Schritten implementiert. Zunächst werden die Vorbedingungen für den Test hergestellt. Im zweiten Schritt folgt dann die Ausführung der zu testenden Programmiereinheit. Der dritte und letzte Schritt vergleicht dann das erwartete Ergebnis mit dem tatsächlichen Ergebnis [Lab17, S. 99]. Stimmt das Ergebnis mit dem erwarteten überein, war der Test erfolgreich. Listing 3.1 zeigt ein Beispiel für einen Komponententest für eine Funktion, geschrieben in Java, die zwei Integer-Werte addiert. Geschrieben wurde der Test mit dem *JUnit*-Framework. Es lassen sich gut die drei Schritte erkennen. Die dreistufige Struktur wird häufig mit den drei Schlüsselwörtern *Arrange*, *Act* und *Assert* beschrieben.

```
1  @Test
2  void integerAddition() {
3      // 1. Arrange: Vorbedingungen schaffen
4      int a = 10;
5      int b = 20;
6      int result;
7
8      // 2. Act: Aufruf der zu testenden Funktion
9      result = MyMathImplementation.IntegerAdd(a, b);
10
11     // 3. Assert: Verifizieren des Ergebnisses
12     assertEquals(30, result);
13 }
```

Listing 3.1: Ein Komponententest in Java basierend auf dem jUnit Framework.

Eine Programmiereinheit wird in der Regel durch eine Vielzahl an Komponententests überprüft. Wichtig ist, dass die Tests nicht Abhängigkeiten zueinander aufweisen. Folglich ist die Reihenfolge der durchgeführten Test egal. Zum Schreiben der Tests wird ein

Test-Framework verwendet. Diese sind speziell für die jeweilige Programmiersprache ausgelegt. In Kapitel 4.1 wird auf einige Frameworks für die Programmiersprache C genauer eingegangen. Ein guter Komponententest hat die folgenden Eigenschaften [She19, S. 13 ff.].

- Der Test ist **deterministisch**.
- Der Test ist **reproduzierbar**.
- Der Test ist **atomar**.
- Der Test ist von anderen Tests **unabhängig**.

In der herkömmlichen Vorgehensweise werden die Komponententests geschrieben nachdem die Implementierung der Programmiereinheit fertig gestellt wurde. Der testgetriebene Ansatz hingegen zieht den Test vor die eigentliche Implementierung. Zudem erfolgt die Implementierung in vielen kurzen Iterationsschritten. Kent Beck sieht für jede Iteration die folgenden fünf Schritte vor [Bec03, S. 1].

1. Einen neuer Test wird hinzugefügt.
2. Alle Tests werden ausgeführt. Der neue Test schlägt fehl.
3. Der Code wird angepasst, um den neuen Test bestehen zu lassen.
4. Erneut werden alle Test ausgeführt. Der neue Test sollte nun erfolgreich sein.
5. Der Code wird refaktorisiert, um Duplikate im Code zu entfernen.

Natürlich sollte auch nach dem Refaktorisieren, durch Ausführen aller Tests, sichergestellt werden, dass das Verhalten nicht verändert wurde. Eine Programmiereinheit entsteht durch ständige Wiederholung dieser Schritte. Ihre Implementierung ist abgeschlossen, sobald die gesamte geforderte Funktionalität mit einer Reihe von Tests beschrieben wird. Der geschriebene Code muss zu einer erfolgreichen Ausführung aller Test führen.

Komponententests helfen Defekte im Code ausfindig zu machen. Zudem steigern sie die Code Qualität, da testbarer Code modular geschrieben sein muss. Sie können darüber hinaus als Dokumentation fungieren, da sie nicht nur zeigen, wie der Code aufgerufen werden muss, sondern auch die Intention des Programmierers festhalten.
Der größte Gewinn des testgetriebenen Ansatzes ist die verkürzte Zeitspanne, zwischen dem Moment, an dem ein Defekt entsteht und dem, an dem er gefunden bzw. behoben wird. Dies ist ein wichtiger Punkt, denn die Kosten, die das Beheben eines Fehlers verursachen, steigen mit der Zeit immer weiter an. Wird gar ein Rückruf des Produkts durch

einen Defekt verursacht, kommt es zu einer Kostenexplosion. Grenning führt weiterhin die folgenden Vorteile an [Gre11, S. 8 f.].

- Es entstehen weniger neue Defekte beim Modifizieren von bestehenden Code, da die Tests die Einschränkungen und Anforderungen widerspiegeln.
- Es wird weniger Zeit für das Debuggen aufgewendet. Die Fehlerquelle ist auf den geänderten Teil des Codes eingrenzbar.
- Der getestete Code gibt dem Team mehr Selbstbewusstsein.

Während es sich bei Komponententests klar um White Box Tests handelt, ist dies bei Tests, die geschrieben werden, bevor der zu testende Code existiert, nicht mehr ganz so eindeutig. Aus diesem Grund werden sie auch als *Grey Box* Tests bezeichnet [Gre11, S. 68]. Darin steckt ein weiterer Vorteil. Sie prüfen exakt die funktionalen Anforderungen an den Code. Während White Box Tests immer Gefahr laufen, so geschrieben zu werden, dass sie mit dem existierenden Code bestehen. Der Ansatz verhindert, dass sich die Software Entwickler mit ihren eigenen Tests belügen.

Besonders bei Objekt Orientierten Sprachen und bei den Webtechnologien ist die testgetriebene Entwicklung inzwischen sehr etabliert. Viele davon haben fest in den Sprachumfang integrierte Möglichkeiten, Komponententests zu schreiben. Im nächsten Abschnitt wird der aktuelle Stand der Technik der testgetriebenen Entwicklung, mit besonderen Augenmerk auf eingebettete Software, betrachtet.

3.2 Stand der Technik

Die testgetriebene Entwicklung wird durch mehrere andere Bereiche stark beeinflusst bzw. hat selber auch Auswirkungen auf andere Bereiche. Um sich ein genaues Bild von dem Stand der Technik machen zu können, lohnt es sich demnach, auch einen Blick auf diese Bereiche zu werfen. Im Folgenden werden demnach unterschiedliche Quellen der Fachliteratur vorgestellt und diskutiert.

Bei der Auseinandersetzung mit testgetriebener Entwicklung stößt man unweigerlich auf eine Gruppe von Autoren, die in ihren Werken besonders für diesen Ansatz werben. Es handelt sich hierbei um Kent Beck, James Grenning, Martin Fowler, Robert Martin,

David Thomas und Andrew Hunt. Besser bekannt sind sie als Mitglieder einer Gruppe von Softwareingenieuren, die im Jahr 2001 das *Manifesto for Agile Software Development* [Bec+01] gemeinsam formuliert hat. Dieses Manifest stellt mit seinen zwölf Grundprinzipien die Grundlage für die agile Entwicklung dar. Die testgetriebene Entwicklung ist ein integraler Teil von agilen Vorgehensmodellen, auch wenn sie keinesfalls auf diese beschränkt ist. Folglich sind die Werke, die von diesem Personenkreis verfasst wurden, eine wertvolle Quelle, wenn es um testgetriebene Entwicklung geht.
Beck stellt in seinem Buch *Test-Driven Developement by Example* die Grundprinzipien des testgetriebenen Ansatzes anschaulich anhand von praktischen Codebeispielen vor.
Für Programmiersprachen wie C++, C# oder JavaScript existiert eine große Auswahl an Literatur, die sich speziell auf die testgetriebene Entwicklung oder Komponententests im Bezug auf die jeweilige Sprache beziehen. Leider schaut dies für C insbesondere im Kontext von eingebetteter Software anders aus. Das von James Grenning verfasste *Test-Driven Developement for Embedded C* [Gre11] bildet hier die Ausnahme und kann sich somit als Standardwerk für die testgetriebene Softwareentwicklung bezeichnen lassen. Es geht direkt auf Herausforderungen ein, die sich aus der engen Kopplung von Hardware und Software ergeben, und stellt Test-Frameworks vor, die für eingebettete Anwendungen genutzt werden können. Das im Kapitel 4.2 behandelte Dual-Targeting ist ebenfalls ein Konzept, welches Grenning in diesem Buch beschreibt.
In *The Pragmatic Programmer* [TH20] von David Thomas und Andrew Hunt wird ein guter Überblick zur Arbeitsweise und Grundhaltung eines Software Ingenieurs geliefert. Relevant ist dies insbesondere, da der testgetriebene Ansatz doch eine deutliche Umstellung beim Vorgehen von einem Programmierer abverlangt. Im Hinblick auf agile Vorgehensmodelle gibt das Buch *Clean Agile* [Mar20] von Robert Martin einen umfassenden Überblick. Das Kapitel zu Technischen Praktiken ist besonders interessant. Neben der testgetriebenen Entwicklung wird hier auch auf andere Praktiken wie das *Pair Programming* und *Refactoring* eingegangen, mit denen es möglich ist die Software Qualität zu steigern. Refactoring ist gleichzeitig ein wichtiger Bestandteil der Iterationsschritte in dem testgetriebenen Vorgehen. Das von Martin Fowler verfasste *Refactoring* [Fow19] ist das Standardwerk zu diesem Thema. Es geht auf die Grundprinzipien ein, die beim Refactoring zum Einsatz kommen. Zudem stellt es die Verbindung zum Testen von Software her.
Eines der hauptsächlichen Ziele der testgetriebenen Entwicklung ist es, die Software Qualität zu fördern. Die beiden Bücher *Clean Code* [Mar09] und *Clean Agile* [Mar20] beschäftigen sich mit dieser Thematik. *Clean Code* liefert Einblicke wie beim Software Entwurf, die Software-Qualität aktiv gefördert werden kann. *Clean Agile* ist eine gute Einführung

in agile Vorgehensmodelle.
Die Literatur der bis hier vorgestellten Autoren wird sehr häufig auch in anderen Autoren als Quellen angegeben. Sie birgt jedoch die Gefahr, von der testgetriebene Entwicklung und von agilen Praktiken, eine einseitige Sichtweise zu erlangen. Daher ist sinnvoll auch einen Blick auf Quellen zu werfen, welche unabhängig von den bisher genannten Werken entstanden und der Personengruppe um das *Manifesto for Agile Software Development* sind.

Software Testing [Pat06] von Ron Patton behandelt das übergreifende Thema des Software Tests aus einem von der zugrundelegenden Technologie unabhängigen Blickpunkt. Es diskutiert ausführlich Punkte wie die Ziele des Testens, Testabdeckung, Nachverfolgung und Dokumentation von Tests und dem Test-Reporting. Wenngleich große Teile des Buchs eher dem Bereich der Qualitätssicherung bzw. der Tätigkeit von Software Testerinnen und Testern angehörig sind, ist die Betrachtung dieser angrenzenden Bereiche dennoch interessant. Der Fokus liegt hierbei auf dem Finden von Schwachstellen in Code. Um dies zu erreichen, betrachten wird die zu testende Software sehr strukturiert und analytisch betrachtet. Viele der dafür eingesetzten Techniken lassen sich auch gut für die testgetriebene Entwicklung adaptieren. Die Bücher *Software-Test für Embedded Systems* [Grü17] von Stephan Grünfelder und *Testen von Software und Embedded Systems* [Vig10] von Uwe Vigenschow beschäftigen sich insbesondere mit dem Testen und dem Qualitätsmanagement von eingebetteten Systemen. Sie stellen weiterführende Techniken der Qualitätssicherung wie Modellbasiertes Testen und Echtzeittests vor, gehen auf die geltenden Normen und Standards ein und behandeln den Bereich der Testautomatisierung.
Auch in agilen Vorgehensmodellen spielt das Qualitätsmanagement eine große Rolle. Im Gegensatz zum Wasserfall-Modell findet das Testen jedoch bereits auch schon in frühen Projektstadien im Rahmen der Iterationen statt. *Agile Testing* [CG09] stellt die Arbeit des Qualitätmanagements in diesem Kontext dar.
Software Entwicklerinnen und Entwickler schreiben häufig nicht nur Komponententests. Oft formulieren sie auch automatisierte *Integrations-Test*, welche die Interaktion von mehreren Programmiereinheiten untersuchen, und *System-Tests*, die das System als gesamtes testen. Tarlinder geht in seinem Buch *Developer Testing* [Tar17] auf sämtliche Formen von Software Tests ein, welche in den Bereich der Software Entwicklerinnen und Entwickler fallen. Unter anderem beschreibt er wichtige Konzepte, wie Stubs, Spies und Mock Objekte, die auch in dem Kapitel 4.3.2 noch genauer analysiert werden.

Eines der Grundprinzipien zur Förderung von Software Qualität ist die Wiederverwendung. Dies kann sich sowohl auf fertige Softwaremodule beziehen, als auch auf den Einsatz von Entwurfsmustern, sogenannten *Design Patterns*. Das Standardwerk für Entwurfsmuster ist das Buch *Design Patterns* [Gam+95]. Es ist ein umfassendes Nachschlagewerk, welches viele Muster abhandelt, mit denen immer wiederkehrende Probleme in der Software-Entwicklung auf standardisierte Weise gelöst werden können. Auch die Tests selber können von Design Patterns profitieren. Meszaros beschreibt in *xUnit Test Patterns* [Mes07] Entwurfsmuster für Unit Tests. Die Grundlage für eine gute Code Qualität muss bereits in der Entwurfs-Phase im Projekt gelegt werden. Hier muss, wenn die Software Tests vorgesehen sind, auch schon an die Testbarkeit des Codes gedacht werden. Wie Software Architektur entworfen werden muss, um sie testbar zu machen, wird unter anderem in dem Buch *Software Architecture in Practice* [BCK12] beschrieben.

Eine wissenschaftliche Auseinandersetzung mit der testgetriebenen Softwareentwicklung ist bisher nur in einem eher geringeren Maße vorhanden. Hervortun können sich jedoch einige Fallstudien von Laurie Williams von der North Carolina State University. Sie war unter anderen an den Studien *"Realizing quality improvement through test driven development: results and experiences of four industrial teams"* [Nag+03], *"Assessing Test-Driven Development at IBM"* [MW03] und *"An Initial Investigation of Test Driven Development in Industry"* [GW03] beteiligt.

Ein präziser Umgang mit der Programmiersprache C ist für die testgetriebene Entwicklung eingebetteter Software essentiell. Das Standardwerk zur Sprache wurde vom Erfinder Dennis M. Ritchie persönlich als Co-Autor verfasst. Obwohl *The C Programming Language* [KR88] aus dem Jahr 1988 stammt, so hat es dennoch seine Relevanz behalten. Dies liegt vor allem daran, dass die Sprache nur wenig im Umfang ausgebaut wurde. Damit auch Software in C gut testbar ist, gibt es die Notwendigkeit, komplexere Programmierkonzepte, wie beispielsweise den Einsatz von Klassen, zu imitieren. *Extreme C* [Ami19] gibt genau in diese Techniken Einblick und liefert darüber hinaus eine gute Beschreibung der Build-Pipeline. Auch in C können Programmierer sich Entwurfsmuster zu Nutzen machen. Dieses Thema behandelt Powel Douglass in *Design Patterns for Embedded Systems in C* [PD11] mit besonderen Augenmerk auf eingebettete Systeme.

Insgesamt lässt sich festhalten, dass deutlich weniger Fachliteratur bezüglich Testgetriebener Entwicklung von eingebetteter Software verfügbar ist, als für andere Software-

Technologien. Das Thema Qualitätssicherung ist im eingebetteten Bereich wesentlich etablierter und auch standardisierter. Daher lohnt es sich auch diese verwandte Disziplin zu betrachten und die testgetriebene Entwicklung davon beeinflussen zu lassen. Viele modernere Programmiersprachen bieten nicht nur fest integrierte Möglichkeiten Tests zu schreiben, sondern unterstützen auch durch Sprachfunktionalitäten dabei, eine gute Software Qualität zu erzielen. C hingegen verzichtet auf solche Features gänzlich zu Gunsten eines geringeren Ressourcenbedarfs. Es liegt stattdessen in der Verantwortung der Entwicklerinnen und Entwickler, die Qualität in der Software zu priorisieren und zu fördern. Deshalb lohnt sich in diesem Feld der Einsatz der testgetriebenen Entwicklung. Sie hilft dabei, Qualität von Beginn an in die Software zu integrieren.
Gezwungenermaßen ist das Testen von eingebetteter Software meist auf individuelle Lösungen für das jeweilige Projekt angewiesen. Der Aufwand, einen guten und funktionierenden Workflow zum Testen zusammenzustellen, ist nicht unerheblich und ist daher auch ein Kostenfaktor. Dies wird in der Literatur zwar anerkannt, jedoch werden nur wenig Lösungsansätze dafür geliefert. Daher soll auch in dem Kapitel 4 ein besonderer Augenmerk darauf gelegt werden, Ansätze zu finden, die das Implementieren von Tests einfacher machen, leichter zu automatisieren sind und unabhängiger von der Hardware bzw. Entwicklungsplattform sind.

Im folgenden Kapitel wird etabliert, welche Herausforderungen sich bei eingebetteten Systemen für die testgetriebene Entwicklung ergeben. Daraus werden Anforderungen formuliert, aufgrund derer die in Kapitel 4 untersuchten Prinzipien und Konzepte bewertet werden.

3.3 Herausforderungen von eingebetteter Software

Der größte Unterschied von eingebetteter Software im Vergleich zu anderen Software-Disziplinen, wie beispielsweise Webtechnologien oder Anwender-Programmen, ist ihre enge Koppelung an die zugrundeliegende Hardware. Es gibt direkte Interaktion zwischen Software und Hardware. Durch die System Software-Ebene wird zwar die Hardware abstrahiert, es gibt jedoch keine strenge Trennung, die es verhindert, dass aus der Anwendungsebene auf Hardware-Ressourcen zugegriffen wird. Anders als bei Windows- oder Linux-Systemen wird beispielsweise der Speicher direkt adressiert, anstatt mit virtuellen Speicheradressen zu arbeiten. Eine Trennung von Anwendungssoftware und Hardware

muss somit explizit im Softwaredesign vorgesehen und bei der Implementierung befolgt werden.
Diese Unterschiede machen sich auch beim Schreiben von Softwaretests bemerkbar. Der Verlauf von Tests hängt stark von dem Zustand und dem Verhalten der Hardware ab. Es wäre daher von Vorteil, die Test auch direkt auf der Zielhardware auszuführen. In Projekten bei denen die Software und Hardware gleichzeitig entwickelt werden, wird dies erst sehr spät im Projektverlauf möglich. Dadurch wird die Zeitspanne von der Entstehung eines Defekts und dem Ausführen der Test, die den Defekt potentiell aufdecken könnten, extrem in die Länge gezogen. Wie bereits erwähnt nehmen die Kosten eines Defekts mit der Zeit stark zu. Es ist daher wünschenswert auch eingebettete Software testgetrieben entwickeln zu können. Die Rückmeldezyklen für Defekte und die Qualität der Software ließen sich damit drastisch verkürzen.
Neben der späten Verfügbarkeit der Zielhardware, kommt hinzu, dass unter Umständen nicht jedem Entwickler ein entsprechender Cross-Compiler oder ein Hardware-Prototyp zur Verfügung steht. Der Grund ist, dass die Kosten für die Compiler-Lizenzen und die Prototypen schlicht zu hoch sind, um sie für alle Entwicklerinnen und Entwickler zu beschaffen [Gre11, S. 77 f.].

In den folgenden Kapiteln werden Prinzipien vorgestellt, die es dennoch erlauben, Software für eingebettete Systeme testgetrieben zu entwickeln.

4 Testgetriebene Entwicklung eingebetteter Software

4.1 Frameworks

Dieses Kapitel stellt drei verschiedene Test-Frameworks vor und bewertet sie auf Eignung für die testgetriebene Entwicklung von eingebetteter Software in C. Bei den Frameworks handelt es sich um *Unity*, *CMocka* und *CppUTest*. Zunächst werden Kriterien ausgearbeitet, an denen sich die Eignung der Frameworks bewerten lässt.

Kriterien

- Für das *Dual-Targeting*, ein Prinzip auf das näher im Kapitel 4.2 eingegangen wird, ist es notwendig, das Test-Framework sowohl für das Entwicklungssystem als auch für das Zielsystem übersetzen zu können. Die Frameworks sind hauptsächlich für die Ausführung auf Windows- oder Linux-Systemen ausgelegt. Daher müssen sie für die Zielarchitektur portiert werden. Interessant ist, wie groß der Aufwand für die Portierung ausfällt und ob eine entsprechende Dokumentation existiert.
- Um auch mit Zielsystemen kompatibel zu sein, die nur über geringe Speicherkapazitäten verfügen, sollte das Executable, welches beim Übersetzen der Tests entsteht, möglichst klein ausfallen. Ist das Executable zu groß, um in den ROM eines Mikrocontrollers zu passen, besteht keine Möglichkeit, die Tests auf der Zielhardware auszuführen und zu verifizieren.
- Es ist ratsam in C Datentypen mit fixer Wortbreite zu verwenden. Daher sollten für diese Datentypen gesonderte Assert-Makros verfügbar sein.
- Die Anzahl der Tests kann schon bei Projekten mit geringer Komplexität sehr groß ausfallen. Eine Möglichkeit die Testfälle zu gruppieren und für die einzelnen Gruppen Setup- und Teardown-Funktionen schreiben zu können, hilft bei der Organisation und Übersichtlichkeit der Tests.

- Eine aussagekräftige Textausgabe bei Ausführung der Test, gibt wichtige Hinweise auf die Fehlerquellen und beschleunigt das Beheben der Defekte.
- Der Quellcode des Frameworks sollte aktiv Instand gehalten werden, um sicherzustellen, dass die Qualität auf einem hohen Standard gehalten wird und Fehler im Framework schnell behoben werden.

Die drei hier vorgestellten Frameworks wurden mit *Ubuntu 20.04* auf einer *x86*-Architektur als Entwicklungssystem und einem *STM32F303RE* Mikrocontroller als Zielhardware evaluiert. Übersetzt wurden die Tests mit einer *GNU Toolchain*. Für das Entwicklungssystem wurde dementsprechend der native *gcc* Compiler eingesetzt und für das Zielsystem der *arm-none-eabi-gcc* Cross-Compiler. Als Optimierungsgrad wurde `-Og` gewählt. Der Compiler führt bei dieser Option eine Optimierung des Codes durch, jedoch werden bestimmte Optimierungen deaktiviert, die ein Debuggen des Programms verhindern könnten. Der Optimierungsgrad wird für die Verwendung im Edit-Compile-Debug Zyklus empfohlen [Sta+19, S. 128] und ist daher auch die Wahl für das iterative Vorgehen in der testgetriebenen Entwicklung. Diese Compiler unterstützen viele unterschiedliche Architekturen und sollten somit einen guten Einblick in die Kompatibilität der Frameworks geben.

Unity

Unity[1] ist ein komplett in C geschriebenes Test-Framework. Es ist quelloffen und hat eine MIT Lizenz. Somit ist es auch für kommerzielle Zwecke einsetzbar. Der Quellcode steht über Github[2] zur Verfügung und erhält dort regelmäßige Commits von mehreren Personen. Externe Abhängigkeiten gibt es lediglich zu den Standard C Bibliotheken, die für die meisten Systeme problemlos verfügbar sind.
Listing 4.1 zeigt einen simplen Komponententest mit Unity. Wie man sehen kann wird der Testfall einer Gruppe zugeordnet. Für die Datentypen mit fixer Wortbreite gibt spezifische Makros wie `TEST_ASSERT_EQUAL_UINT8`.

```
1 TEST(testgroupName, testcaseName)
2 {
3     // 1. Arrange: Vorbedingungen schaffen
4     uint8_t a = 10;
5     uint8_t b = 20;
6     uint8_t result;
7
```

[1] http://www.throwtheswitch.org/unity
[2] https://github.com/ThrowTheSwitch/Unity

```
8     // 2. Act: Aufruf der zu testenden Funktion
9     result = addUnsigned8BitIntegers(a, b);
10
11    // 3. Assert: Verifizieren des Ergebnisses
12    TEST_ASSERT_EQUAL_UINT8(30, result);
13 }
```

Listing 4.1: Ein exemplarischer Komponententest basierend auf dem Unity-Framework.

Die Text-Ausgabe von Unity ist in Listing 4.2 zu sehen. Sie enthält für einen fehlgeschlagenen Test den Dateinamen, die Zeile, den Namen der Testgruppe und den des Testfalls. Darüber hinaus wird Auskunft gegeben, welcher Wert erwartet wurde und welcher sich tatsächlich ergab.

```
1 Unity test run 1 of 1
2 .test/Tests.c:30:TEST(testgroupName, testcaseName):FAIL: Expected 30
     Was 29
3
4 -----------------------
5 1 Tests 1 Failures 0 Ignored
6 FAIL
7 }
```

Listing 4.2: Eine exemplarische Ausgabe für einen fehlgeschlagenen Test des Unity-Frameworks.

Um das Framework für einen Mikrocontroller zu portieren muss mittels dem Makro `UNITY_OUTPUT_CHAR(a)` die Text-Ausgabe umgeleitet werden. Dies geschieht in der `unity_config.h` Header-Datei, in der dank weiteren Preprozessor-Makros, auch noch andere Features des Frameworks weg- oder zugeschaltet werden können. Die Ausgabe-Daten werden für das Zielhardware-Executable auf eine serielle Schnittstelle (UART) geschrieben. Auf diese Weise kann mittels eines Serial Terminal Programms die Ausgabe angezeigt werden.
Zu beachten ist weiterhin, dass in der Main-Funktion für den Test auf der Zielhardware zunächst die Hardware initialisiert werden muss. Es müssen, bevor das Framework ausgerufen wird, erst die System-Clock, die IO und das UART konfiguriert und initialisiert werden. Das Portieren des Frameworks wird auf der Unity Webseite ausführlich erklärt. Das Executable für den STM32F303RE hat für den oben dargestellten Test eine Größe von 12,7 kByte.

CMocka

CMocka[3] ist ebenfalls ein komplett in C geschriebenes, quelloffenes Komponententest-Framework. Es steht unter einer Apache 2.0 Lizenz und ist somit auch für kommerzielle Anwendungen geeignet. Der Quellcode ist bei GitLab[4] zu finden. Die Codebasis erhält regelmäßige Commits. Das Projekt wird demnach aktiv Instand gehalten. Zum Übersetzen werden nur die Standard C Bibliotheken benötigt. Wie der Name bereits sagt, legt das Framework einen besonderen Schwerpunkt auf Mock-Objekte.
Listing 4.3 zeigt den Aufbau eines typischen Komponententests mit dem CMocka-Framework. Zu beachten ist, dass CMocka keine spezifischen Assert-Makros für Datentypen mit fixer Wortbreite bietet.

```
1  static void test_testcaseName(void **state)
2  {
3      // 1. Arrange: Vorbedingungen schaffen
4      uint8_t a = 10;
5      uint8_t b = 20;
6      uint8_t result;
7
8      // 2. Act: Aufruf der zu testenden Funktion
9      result = addUnsigned8BitIntegers(a, b);
10
11     // 3. Assert: Verifizieren des Ergebnisses
12     assert_int_equal(30, result);
13 }
```

Listing 4.3: Ein exemplarischer Komponententest basierend auf dem CMocka-Framework.

Die Ausgabe (Listing 4.4) für einen fehlgeschlagenen Test auf dem Entwicklungssystem ist sehr übersichtlich und liefert genaue Informationen über den fehlgeschlagenen Testfall.

```
1 [==========] tests: Running 1 test(s).
2 [ RUN      ] test_testcaseName
3 [  ERROR   ] --- 0x1e != 0x1d
4 [   LINE   ] --- test/exampleTest.c:28: error: Failure!
5 [  FAILED  ] test_testcaseName
6 [==========] tests: 1 test(s) run.
7 [  PASSED  ] 0 test(s).
8 [  FAILED  ] tests: 1 test(s), listed below:
```

[3] https://cmocka.org/
[4] https://gitlab.com/cmocka/cmocka/

```
 9 [  FAILED  ] test_testcaseName
10
11 1 FAILED TEST(S)
```

Listing 4.4: Eine exemplarische Ausgabe für einen fehlgeschlagenen Test des CMocka-Frameworks auf dem Entwicklungssystem.

Wird der gleiche Test jedoch für die Zielhardware übersetzt, so fällt auf, dass obwohl das Programm problemlos durchläuft, die Ausgabe wesentlich magerer ausfällt. Es wird nur noch Auskunft über den fehlgeschlagenen Test gegeben, allerdings fehlen jegliche Informationen über die Quell-Datei, die Zeile und die Annahme. Es wäre ein Eingriff in die Codebasis des Frameworks notwendig, um diese Problematik zu beheben.

```
1 [==========] tests: Running 1 test(s).
2 [ RUN      ] test_testcaseName
3 [  FAILED  ] test_testcaseName
4 [==========] tests: 1 test(s) run.
```

Listing 4.5: Eine exemplarischer Ausgabe für einen fehlgeschlagenen Test des CMocka-Frameworks auf dem Zielsystem.

Das Übersetzen für die Zielplattform erwies sich als zeitintensiv. In einer `config.h` Header-Datei müssen über Preprozessor-Makros einzelne Features des Frameworks konfiguriert werden. Zudem muss über ein Makro die `printf`-Funktion umgeleitet werden, sodass die Ausgabe, wie auch schon bei Unity über die serielle Schnittstelle erfolgen kann. Da dieser Vorgang nur recht spärlich dokumentiert ist, wird hier deutlich mehr Einsatz gefordert als bei dem Unity-Framework.
Das Executable für die Zielhardware hat eine Größe von 30,9 kByte.

CppUTest

Im Gegensatz zu den ersten beiden Frameworks, handelt es sich bei CppUTest[5] um ein Framework geschrieben in C++. Neben Bas Vodde gehört James Grenning zu den Hauptauthoren dieses quelloffenen Frameworks. In seinem Buch *Test-Driven Developement for Embedded C* stellt Grenning die testgetriebene Entwicklung mit diesem Framework ausführlich vor. Die Codebasis ist bei Github[6] unter der BSD 3-Clause Lizenz verfügbar und

[5] `http://cpputest.github.io/`
[6] `https://github.com/cpputest/cpputest`

ist daher auch für kommerzielle Zwecke nutzbar.
Im Vergleich zu den anderen beiden Frameworks ist CppUTest am komfortabelsten zu benutzen. Während Unity und CMocka es erfordern, jeden Testfall einzeln in den Testrunner einzuhängen, geschieht das bei CppUTest automatisch.
Listing 4.6 zeigt einen exemplarischen Komponententest mit CppUTest.

```
1  TEST(testgroupName, testcaseName)
2  {
3      // 1. Arrange: Vorbedingungen schaffen
4      uint8_t a = 10;
5      uint8_t b = 20;
6      uint8_t result;
7
8      // 2. Act: Aufruf der zu testenden Funktion
9      result = addUnsigned8BitIntegers(a, b);
10
11     // 3. Assert: Verifizieren des Ergebnisses
12     CHECK_EQUAL(30, result);
13 }
```

Listing 4.6: Ein exemplarischer Komponententest basierend auf dem CppUTest-Framework.

Die Ausgabe im Falle eines fehlgeschlagenen Tests ist detailliert und bietet alle Informationen, die notwendig sind, um den Defekt zu lokalisieren.

```
1  test/LinkedListTest.cpp:38: error: Failure in TEST(testgroupName,
       testcaseName)
2  expected <30>
3  but was  <29>
4  difference starts at position 0 at: <          29        >
5
6  .
7  Errors (1 failures, 1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered
       out, 0 ms)
```

Listing 4.7: Eine exemplarische Ausgabe für einen fehlgeschlagenen Test des CppUTest-Frameworks.

Problematisch ist hingegen das Übersetzen des Frameworks für die Zielplattform. Der Build-Prozess ist wesentlich komplexer als bei den anderen Frameworks. Daher war es leider nicht möglich, das Framework für die Zielarchitektur zu portieren. Grundsätzlich soll dies laut Autor allerdings möglich sein. Jedoch würde dafür eine Einarbeit in die

Codebasis notwendig werden, was in einem Projektverlauf zeitlich sehr wahrscheinlich nicht zu rechtfertigen wäre.

Wahl eines Frameworks

Für den weiteren Verlauf dieser Arbeit wird das Unity-Framework eingesetzt werden. Es überzeugt besonders durch seine gute Portierbarkeit. Wenngleich alle drei Frameworks grundsätzlich für eingebettete Systeme übersetzt werden können, ist der Aufwand für CMocka und CppUTest zu hoch. Weiter verfügt Unity über eine Reihe an Assert-Makros, welche besonders für eingebettete Software nützlich sind. Auch wenn der Feature-Umfang im Vergleich am geringsten ist, bietet Unity dennoch alle wesentlichen Features, welche für die testgetriebene Entwicklung benötigt werden. Gerade wegen des geringeren Umfangs haben die Executables eine kleinere Größe, was wiederum sehr positiv zu bewerten ist. Setzt man keine Priorität darauf, die Tests auch auf der Zielhardware auszuführen, wäre CppUTest die beste Wahl. Es sei an dieser Stelle erwähnt, dass im CppUTest-Repository Scripte enthalten sind, die eine Umwandlung von einem CppUTest-Projekt in ein Unity-Projekt erlauben.

4.2 Dual-Targeting

In diesem Kapitel wird das Prinzip des Dual-Targetings vorgestellt. Es stellt das wichtigste Konzept dar, welches eine testgetriebene Entwicklung von eingebetteter Software, ohne über die finale Hardware zu verfügen, überhaupt erst ermöglicht. Die grundlegende Idee ist, die Software und insbesondere die Anwendungsebene, also den Teil der hauptsächlich einen Gegenwert für die Kosten schafft, für zwei unterschiedliche Architekturen zu entwickeln. Die Komponententests mit denen die Entwicklung getrieben wird, können also sowohl auf dem Entwicklungssystem als auch auf der Zielhardware ausgeführt werden. James Grenning, der in *Test-Driven Developement for Embedded C* dieser Thematik ein ganzes Kapitel widmet formuliert es wie folgt [Gre11, S. 79]:

> *„Dual-targeting solves several problems. It allows you to test code before the hardware is ready, and you can avoid the hardware bottleneck throughout the development cycle. You also avoid the finger pointing that goes with simultaneous hardware software debugging. It is a practice that keeps you moving fast.“*

Aus verschiedenen Gründen bezeichnet er die Hardware als Flaschenhals. So nimmt er zum einen an, dass die Hardware erst spät oder nicht allen Programmiererinnen zur Verfügung steht und zum anderen sagt er, das Übersetzen und Hochladen des Codes auf die Zielhardware, dauert länger als bei dem Entwicklungssystem. Diese Aussage lässt sich leicht prüfen, indem man die Dauern für die jeweiligen Build-Vorgänge misst. Für das Unity-Test-Projekt welches im Kapitel 4.1 genutzt wurde, wird für einen Zielsystem Build und Upload 1,901 s benötigt, während der Entwicklungssystem Build nach 0,347 s abgeschlossen ist. Dies mag nicht signifikant wirken, bedenkt man jedoch, dass bei der testgetriebenen Entwicklung unter Umständen mehrmals pro Minute die Tests ausgeführt werden, wird das Einsparpotential klar. Anzumerken ist hier, dass dieses Projekt nur einen einzigen Testfall hatte. In einem richtigen Projekt können mehrere tausende Testfälle zusammenkommen. Dies hätte eine entsprechende Auswirkung auf die Build-Dauer.
Natürlich hat das Dual-Targeting auch einige Nachteile. Die Compiler unterstützen ggf. verschiedene Sprachfeatures, weisen unterschiedliche Defekte auf oder haben verschiedene Runtime Bibliotheken. Je nach System könnten die primitiven Datentypen unterschiedliche Wortbreiten oder eine andere Byte-Reihenfolge haben [Gre11, S. 80]. Dies sind aller-

dings Probleme, die sich gut antizipieren lassen. Es kann sich lohnen, bei Unklarheiten speziell diese Probleme in einzelnen Unit-Tests zu untersuchen. Auf diese Weise wird bei Unstimmigkeiten zwischen den zwei Systemen oder zwischen unterschiedlichen Bibliotheken eine Fehlermeldung generiert. Somit ist das Problem für den weiteren Projektverlauf bereits bekannt und durch den Testfall dokumentiert.

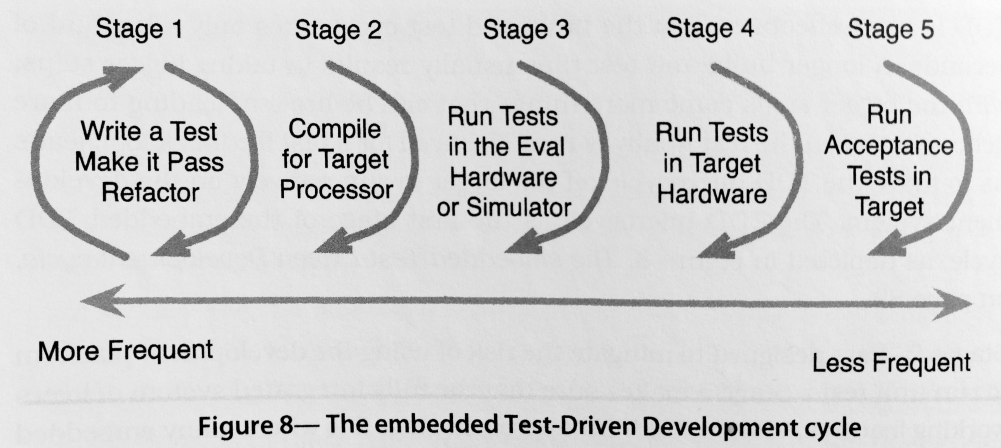

Abbildung 4.1: Der testgetriebene Entwicklungs-Zyklus [Gre11, S. 82]

Abbildung 4.1 zeigt den Lebenszyklus für Testgetriebene Entwicklung bezogen auf den Einsatz von verschiedenen Zielarchitekturen. Je nach Budget und Zielhardware gibt es neben dem Entwicklungssystem und der finalen Zielhardware noch weitere Möglichkeiten, die Tests auszuführen. So werden für die meisten Mikrocontroller Evaluationsboards angeboten. Diese sind neben dem Mikrocontroller meist noch mit weiteren grundlegenden Peripherien bestückt, sodass sich damit leicht einfache Prototypen der letztendlichen Hardware umsetzen lassen. Darüber hinaus gibt es Simulatoren, welche auf dem Entwicklungssystem ein virtuelles Abbild des Prozessors simulieren, in dem der Code ausgeführt werden kann. Für gewöhnlich ist der Build-Vorgang für diese Ziele identisch mit dem eigentlichen Target-Build. Grenning schlägt in seinem Lebenszyklus vor, die Tests und den Code auf dem Entwicklungssystem auszuführen und zu entwickeln (Stage 1). Die Tests werden dort am häufigsten ausgeführt. Ergänzend wird das Executable auch für die Zielarchitektur übersetzt und entweder auf der Evaluations-Plattform oder dem Simulator (Stage 3) getestet. Sobald eine Version der Zielhardware bereitsteht, werden die Tests schließlich auch darauf ausgeführt (Stage 4). Mittels Acceptance-Tests (Stage 5), welche von den Software-Testerinnen und Testern entworfen werden, wird schließlich validiert, ob sich das System sinnvoll mit seinen Umgebungsbedingungen und Aufgaben verhält.

4.3 Test-Strategien

Beim Entwickeln und Testen auf dem Entwicklungssystem ergeben sich Situationen, in denen der Code Abhängigkeiten zu anderen Modulen aufweist. In diesem Zusammenhang wird ein Modul auch als *Collaborator* bezeichnet. Solange diese Module keine weiteren Abhängigkeiten aufweisen, also eine in sich abgeschlossene Einheit darstellen, ist dies unproblematisch. Vorausgesetzt das Modul wurde bereits implementiert, kann es im Komponententest direkt eingesetzt werden. Schwieriger hingegen wird es, wenn es sich um Abhängigkeiten handelt, die im Bezug zur Hardware oder zum Betriebssystem (RTOS) stehen. Selbst wenn für diese Module bereits Implementierungen vorhanden sind, so lassen sie sich nicht auf dem Entwicklungssystem ausführen. Dies liegt an ihrer engen Kopplung zur Zielarchitektur. Sie greifen meist direkt auf Register der Hardware zu, die der Steuerung von Peripherie oder dem Prozessor selbst dienen und nutzen Interrupts. Ein solches Modul auf dem Entwicklungssystem auszuführen, würde einen erheblichen Portierungsaufwand bedeuten, dessen Kosten bei weitem den Nutzen übersteigen würden.
Die sogenannten *Test-Doubles* stellen die Lösung für dieses Problem dar. Beim Testen von Software für generelle Rechensysteme werden sie beispielsweise eingesetzt, um Datenbank-Zugriffe oder Datenströme zu ersetzen. Bill Laboon stellt folgenden Anspruch an einen Komponententest [Lab17, S. 109].

> *„A unit test should be a localized test; that is, it should check the particular method or function under test, and not worry about other aspects of the system.“*

Die Doubles erlauben genau dies. Sie ersetzen für die Dauer des Tests den Collaborator durch eine deutlich vereinfachte Implementierung.
Ein wichtiges Ziel der Doubles ist es Interaktionen, welche innerhalb der getesteten Funktion stattfinden, überwachen zu können. Solche Interaktionen würden sich sonst, rein über den Rückgabewert der Funktion nicht verifizieren lassen. Darüber hinaus geben sie dem Test die Kontrolle über alle Zustände im Verlauf des Testfalls. So können zum Beispiel Fehlerzustände in Peripheriegeräten simuliert werden, die andernfalls nur schwer herzustellen wären.
Test-Doubles gibt es in verschiedenen Detaillierungsgraden. In der Literatur existieren für die einzelnen Arten von Doubles unterschiedliche Definitionen. Um für die folgenden Abschnitte eine gemeinsame Diskussionsgrundlage zu schaffen, sollen an dieser Stelle die

Begrifflichkeiten definiert werden. Die Definitionen folgen der Beschreibung von James W. Grenning [Gre11, S. 114].

Dummy

Ein Dummy ist eine Funktion, die von den Testfällen nie tatsächlich aufgerufen wird. Sie dient lediglich dazu eine leere Implementierung zu Verfügung zu stellen, damit sich der Quellcode kompilieren lässt.

Stub

Eine Funktion, die ein durch den Testfall vorgeschriebenen Rückgabewert liefert.

Spy

Ein Spy weist die selbe Funktionalität auf wie ein Stub, speichert darüber hinaus die übergebenden Parameter, sodass diese im Testfall verifiziert werden können.

Fake

Ein Fake implementiert die Funktionalität des eigentlichen Collaborators soweit es für die Testfälle notwendig ist. Die Implementierung wird dabei vereinfacht.

Mock

Ein Mock erlaubt die Verifizierung einer Folge von mehreren Interaktionen und kann für jede Interaktion individuell festgelegte Rückgabewerte liefern.

Der Schwerpunkt von Stubs, Spies und Fakes liegt darauf, in einem Test den Endzustand verifizieren zu können. Ein Mock hingegen dient der Betrachtung des Verhaltens.

Wie der Begriff *Double* bereits vermuten lässt, handelt es sich um eine alternative Implementierung des Collaborators zu der des eigentlichen Quellcodes. Für das Test-Executable

wird der eigentliche Quellcode mit dem des Doubles ersetzt. C bzw. die Build-Pipeline bieten hierfür drei Mechanismen [Gre11, S. 113 ff.].

Präprozessor-Substitution

Bei der Präprozessor-Substitution wird sich zu Nutzen gemacht, dass der Präprozessor mittels eines Makros einen Funktionsaufruf durch einen anderen ersetzten kann. In Listing 4.8 wird dargestellt, wie ein Funktionsaufruf von `GPIO_ReadInput()` durch den Aufruf von dessen Stub `stub_GPIO_ReadInput()` ausgetauscht wird. Grundsätzlich gilt, der Einsatz von Präprozessor-Makros sollte, soweit möglich, vermieden werden. Sie wirken sich negativ auf die Lesbarkeit des Codes aus und verschleiern die eigentliche Funktionalität des Programms.

```
1 #define GPIO_ReadInput(address, number) stub_GPIO_ReadInput(address,
     number)
2
3 GPIO_State_t double_GPIO_ReadInput(volatile uint32_t*
     gpioBankAddress, uint8_t gpioNumber);
```

Listing 4.8: Eine exemplarische Präprozessor-Substitution einer Funktion mit ihrem Test-Double.

Link-Time-Substitution

Dieser Mechanismus nutzt die Build-Pipeline von C-Compilern, genauer gesagt den Linker-Schritt. Da in Header- und Quelldateien unterschieden wird, ist es möglich für eine Schnittstelle (definiert in der Header-Datei) unterschiedliche Implementierungen in mehreren Quelldateien zu erstellen. Jede Quelldatei stellt eine eigene Übersetzungseinheit dar (siehe Kapitel 2.3.1) und erzeugt somit auch eine eigene Objekt-Datei. Beim Linken kann nun eine Objekt-Datei durch eine andere ersetzt werden, die ein Double des Moduls enthält. Solange das Double sich an die Schnittstelle hält, kann die komplette Implementierung des Moduls einfach ausgetauscht werden. Nachteilig ist, dass immer nur eine Implementierung simultan für das gesamte Test-Executable eingesetzt werden kann. Soll auch der Collaborator getestet werden, so müssen zwei separate Test-Executables erzeugt und ausgeführt werden.

Funktionszeiger-Substitution

Die Funktionszeiger-Substitution ist die flexibelste Methode, eine Implementierung des Collaborators durch ein Double zu ersetzen. Statt das zu testende Modul direkt die Funktionen des Collaborators aufrufen zu lassen, werden ihm Funktionszeiger zu den Funktionen übergeben. Dadurch ist es möglich, zur Laufzeit die Implementierung auszutauschen. Listing 4.9 und 4.10 zeigen, wie eine Funktionszeiger-Substitution umgesetzt werden könnte. Das Beispiel macht deutlich, auch hier wird der Quellcode schwerer lesbar, da er einen größeren Umfang hat. Es wird zudem die Komplexität erhöht, was zu weiteren potentiellen Fehlerquellen führen könnte. Auch wird durch diese Methode etwas mehr RAM benötigt. Was allerdings nur in den seltensten Fällen kritisch werden dürfte.

```
1  // Funktionszeiger der readInput Funktion
2  static GPIO_State_t (*readInput)(volatile uint32_t* gpioBankAddress,
       uint8_t gpioNumber) = NULL;
3
4  // Setter des Funktionszeigers
5  void setReadInputFunction(GPIO_State_t (*readInput)(volatile
       uint32_t* gpioBankAddress, uint8_t gpioNumber) function)
6  {
7      readInput = function;
8  }
9
10 // Aufruf der readInput Funktion
11 bool getSwitchState(void)
12 {
13     if(readInput == NULL) return false;
14     bool result = false;
15     GPIO_State_t state = readInput(GPIO_REG_ADDR, 1);
16     if(state == GPIO_STATE_HIGH)
17     {
18         result = true;
19     }
20     return result;
21 }
```

Listing 4.9: Eine exemplarische Implementierung eines Moduls mit einem Funktionszeiger und der entsprechenden Setter-Funktion.

```
1  // Implementierung des Stubs
2  static GPIO_State_t stub_readInput(volatile uint32_t*
       gpioBankAddress, uint8_t gpioNumber)
```

```
3 {
4     return true;
5 }
6
7 // Setzen des Funktionszeigers
8 setReadInputFunction(stub_readInput);
9
10 // Aufruf der zu testenden Funktion
11 bool state = getSwitchState();
```

Listing 4.10: Der Funktionszeiger wird auf eine Stub-Implementierung der Funktion gesetzt.

Je nach Situation ist individuell abzuschätzen, welche der drei vorgestellten Methoden sich am besten eignet.

In den folgenden zwei Abschnitten wird anhand eines Beispielprojekts demonstriert, wie Spies, Fakes und Mocks dazu eingesetzt werden können, Module testgetrieben auf dem Entwicklungssystem zu entwickeln und zu testen.

4.3.1 Spies und Fakes

Zunächst soll an dieser Stelle kurz das Beispielprojekt vorgestellt werden, an dem für diese Arbeit die behandelten Themen praktisch nachvollzogen und analysiert wurden.

Das Beispielprojekt setzt ein Entgleisungs-Warn-System (*Derailing Alarm System*) für Züge um und wurde testgetrieben auf dem Entwicklungssystem programmiert. Abbildung 4.2 zeigt das Blockschaltbild der Hardware-Architektur. Es handelt sich um ein Subsystem, das über einen TIA-485 Bus (auch als RS-485 bezeichnet) mit anderen Bus-Teilnehmern mittels eines simplen Binär-Protokolls kommuniziert. Über den Kommunikations-Bus teilt es den anderen Busteilnehmern mit, wenn die Beschleunigungskräfte oder Winkelgeschwindigkeiten des Wagons, in dem es verbaut wird, die gesetzten Schwellwerte überschreiten. Die Entgleisungs-Erkennung erfolgt mit Hilfe des Smart-Sensors BNO055[7] der Firma Bosch. Dieser kombiniert einen Beschleunigungsmesser, ein Magnetometer und ein Gyrometer und nimmt jeweils für drei Achsen Messwerte auf. Er wird mit einer I2C-Schnittstelle angebunden. Das System baut auf dem STM32F303RE[8] Mikrocontroller

[7] https://www.bosch-sensortec.com/products/smart-sensors/bno055.html
[8] https://www.st.com/en/microcontrollers-microprocessors/stm32f303re.html

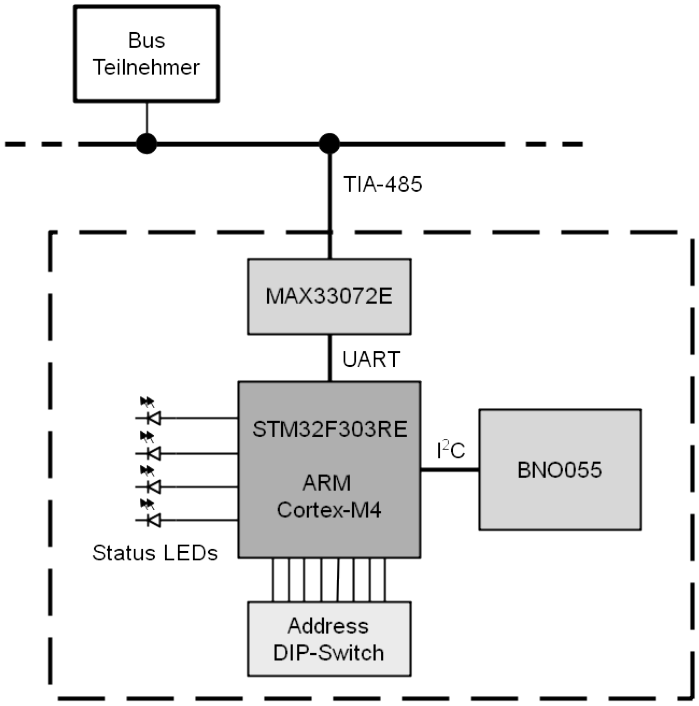

Abbildung 4.2: Blockschaltbild des Derailing Alarm Systems.

von STM auf. Der Prozessor ist performant genug, um ein Betriebssystem, in diesem Fall FreeRTOS[9], ausführen zu können. Weitere Peripherie besteht aus einem DIP-Switch zum Einstellen der Bus-Adresse und Status LEDs. Die vollständige Beschreibung findet sich im Anhang A.1.

In diesem Kapitel wird anhand des LED-Treibers gezeigt, wie unter Zuhilfenahme von Stubs, Spies und Fakes das Modul für die testgetriebene Entwicklung auf dem Entwicklungssystem von Hardware-Abhängigkeiten isoliert werden kann. Das Modul weist Abhängigkeiten zu dem GPIO-Modul der Hardware-Abstraktionsebene und zu dem Timer-Modul der Betriebssystem-Abstraktionsebene auf. Abbildung 4.3 zeigt diese Abhängigkeiten in Form eines Klassendiagramms. Die Einführung der Abstraktionsebenen hilft direkte Abhängigkeiten zur Hardware oder dem Betriebssystem in der Anwendungsebene zu vermeiden. Die Zugriffe werden durch die Definition von Schnittstellen Plattformun-

[9] `https://www.freertos.org/`

abhängig, sodass sie auch auf dem Entwicklungssystem getestet werden können. Kapitel 4.4.1 geht auf den Aspekt der Abstraktionsebenen im Systementwurf genauer ein.

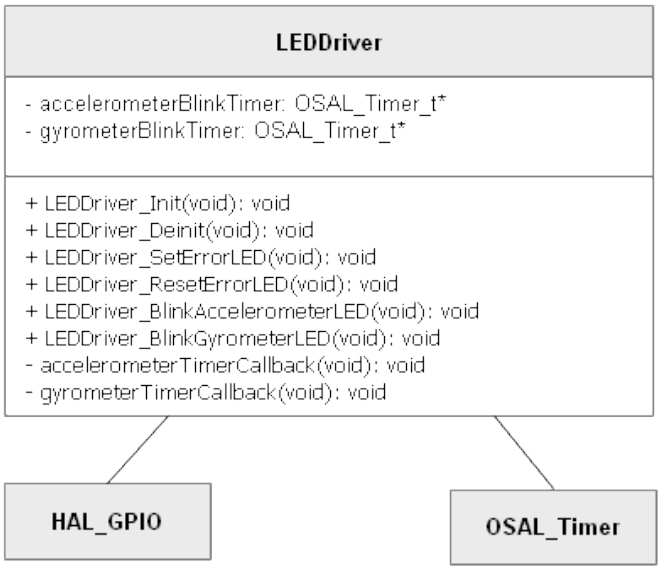

Abbildung 4.3: Klassendiagramm des LED Treibers.

Wie in dem Klassendiagramm zu erkennen ist, können über die Schnittstelle vom `LEDDriver`-Modul drei Status-LEDs angesteuert werden. Die Error-LED soll interne Fehlzustände signalisieren und die Accelerometer- und Gyrometer-LEDs sollen für eine kurze Zeitspanne blinken, wenn das System eine Schnellwertüberschreitung detektiert. Jede der Methoden muss, um das gewünschte Verhalten umzusetzen, die entsprechenden GPIOs der LEDs ansteuern. Die Blink-Methoden nutzen einen Software-Timer, um die LEDs für eine bestimmte Zeitspanne blinken zu lassen. Folgende Liste enthält die Testfälle die notwendig sind, um zu verifizieren, dass das gewünschte Verhalten umgesetzt wird.

- `InitConfiguresPins0to2AsOutputs`
- `InitTurnsLEDsOffByDefault`
- `InitUsesCorrectGPIOBank`
- `InitCreatesTwoTimersWith500msPeriods`
- `SetErrorLEDTurnsOnLED0`
- `ResetErrorLEDTurnsOffLED0`
- `BlinkAccelerometerTurnsOnLED1`

- BlinkAccelerometerStartsTimer
- BlinkAccelerometerTimerCallbackTurnsOffLED1
- BlinkAccelerometerTimerCallbackResetsTimer
- BlinkGyrometerTurnsOnLED2
- BlinkGyrometerStartsTimer
- BlinkGyrometerTimerCallbackTurnsOffLED2
- BlinkGyrometerTimerCallbackResetsTimer

Die ersten vier Testfälle überprüfen zunächst, ob bei der Initialisierung die entsprechenden GPIOs als Outputs konfiguriert und die notwendigen zwei Timer erzeugt weden. Die weiteren Tests prüfen, ob die LED-Ansteuerungen korrekt implementiert sind.

```
13 #include "LEDDriver.h"
14
15 #include "OSAL_Timer.h"
16 #ifdef PRODUCTION
17     #include "HAL_GPIO.h"
18 #else
19     #include "HAL_GPIODouble.h"
20 #endif
```

Listing 4.11: Einbindung der Collaboratoren-Header in LEDDriver.c

Das HAL_GPIO-Modul wurde durch ein Spy substituiert. Hier kam es für die Testfälle ausschließlich darauf an, die Parameter der Funktionsaufrufe zu überprüfen. Für das OSAL_Timer-Modul musste darüber hinaus auch ein geringer Anteil der eigentlichen Funktionalität imitiert werden. Daher wurde hier zusätzlich zu den Spy-Techniken auch die Fake-Strategie angewandt. Listing 4.11 liefert einen Hinweis, welche Substitutions-Methoden für die beiden Collaboratoren eingesetzt wurden. Für den OSAL_Timer konnte die Link-Time-Substitution genutzt werden. Die eigentlichen Implementierung des Moduls befindet sich in einem Verzeichnis, welches in den Test-Builds nicht eingebunden wird. Es kann daher zu keinen Kollisionen beim Build kommen. Das HAL_GPIO hingegen wird in dem selben Test-Projekt ebenfalls testgetrieben entwickelt. Somit ist es die einfachste Option, an dieser Stelle auf die Link-Time-Substitution zu setzen. Über einen Präprozessor-Switch (PRODUCTION) der nur bei einem Build-Vorgang des Production-Codes definiert ist, wird die Header-Datei HAL_GPIO.h gegen die des Doubles (HAL_GPIODouble.h) ersetzt. Diese führt die eigentliche Präprozessor-Substitution

durch (siehe Listing im Anhang A.4).

```
51 /**
52  * @{
53  * @name Functions for controlling the double
54  */
55 void Double_HAL_GPIO_Reset(void);
56 volatile uint32_t* Double_HAL_GPIO_GetLastBankAddress(void);
57 uint8_t Double_HAL_GPIO_GetLastGPIONumber(void);
58 void Double_HAL_GPIO_SetGPIOState(HAL_GPIO_State_t state, uint8_t
       gpioNumber);
59 HAL_GPIO_State_t Double_HAL_GPIO_GetGPIOState(uint8_t gpioNumber);
60 Mode_t Double_HAL_GPIO_GetGPIOMode(uint8_t gpioNumber);
61 /** @} */
```

Listing 4.12: Die Spy Schnittstelle des `HAL_GPIO`-Doubles (`HAL_GPIODouble.h`).

In Listing 4.12 sind die Funktionsprototypen zu sehen, welche die Schnittstelle des Spies bilden. Typischerweise bieten Spies eine Reset-Methode, die es erlaubt den Spy zwischen den Testfällen wieder in einen definierten Ausgangszustand zu bringen. Die weiteren Methoden dienen dem Abfragen der letzten Aufrufs-Parameter und des aktuellen Zustands.

```
32 /**
33  * @{
34  * @name Timer double control functions
35  */
36 void Double_OSAL_Timer_Reset(void);
37 void Double_OSAL_Timer_InvokeCallback(OSAL_Timer_t* timer);
38 OSAL_Timer_t* Double_OSAL_Timer_GetLastTimer(void);
39 OSAL_Timer_t* Double_OSAL_GetTimer(uint8_t index);
40 /** @} */
```

Listing 4.13: Die Schnittstelle des `OSAL_Timer`-Doubles (`OSAL_TimerDouble.h`).

Die Schnittstelle des `OSAL_Timer`-Doubles ist in Listing 4.13 abgebildet. Neben der bereits beschriebenen Methoden für den Spy, gibt es auch eine `InvokeCallback`-Funktion. Diese führt einen Callback aus, der nach Ablauf eines Timers geschehen muss. Dies ist zwingend notwendig, um das zu testende Modul vollständig betrachten zu können.
Der in Listing 4.14 dargestellte Testfall überprüft, ob beim Aufruf der Blink-Funktion

tatsächlich ein Timer gestartet wird. Dazu holt er sich mit `GetLastTimer()` den letzten Timer, der an das Timer-Modul übergeben wurde und prüft dessen Zustand.

```
110 TEST(LEDDriverTest, BlinkAccelerometerStartsTimer)
111 {
112     // act
113     LEDDriver_BlinkAccelerometerLED();
114     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
115
116     // assert
117     TEST_ASSERT(timer->started);
118 }
```

Listing 4.14: Testfall der prüft, ob der Timer durch den Funktionsaufruf gestartet wird. (`LEDDriverTest.c`).

Der in Listing 4.15 zu sehende Testfall, ist ein gutes Beispiel für den Einsatz eines Spies. Über die `GetLast`-Funktionen kann mit Assertions geprüft, ob der gewünschte Zustand hergestellt wurde. In diesem Beispiel wird getestet, ob der korrekte I/O und die entsprechende GPIO-Bank angesteuert wird. Darüber hinaus wird geschaut, ob der Ausgang tatsächlich in den richtigen Zustand versetzt wurde. In Zeile 110 findet sich der Aufruf der `InvokeCallback`-Funktion. Ohne eine Implementierung im Fake des Timer-Moduls, wäre dieser Testfall nicht durchführbar gewesen.

```
120 TEST(LEDDriverTest, BlinkAccelerometerTimerCallbackTurnsOffLED1)
121 {
122     // arrange
123     LEDDriver_BlinkAccelerometerLED();
124     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
125
126     // act
127     Double_OSAL_Timer_InvokeCallback(timer);
128
129     // assert
130     TEST_ASSERT_EQUAL_PTR(GPIO_BANK,
        Double_HAL_GPIO_GetLastBankAddress());
131     TEST_ASSERT_EQUAL_HEX8(1, Double_HAL_GPIO_GetLastGPIONumber());
132     TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW,
        Double_HAL_GPIO_GetGPIOState(1));
133 }
```

Listing 4.15: Testfall der prüft, ob der Aufruf der Callback-Funktion die LED wieder ausgeschaltet (`LEDDriverTest.c`).

Mit der Hilfe von den beiden Doubles ist es möglich, das `LEDDriver`-Modul komplett von seinen Abhängigkeiten zu entkoppeln. Die Testfälle können so individuelle, atomare Szenarien prüfen. Auch bei vielfacher Ausführung der Tests bleiben die Ergebnisse stets gleich. Zwar bedeutet das Umsetzen der Doubles einen Mehraufwand, dieser ist jedoch gerechtfertigt, da es anderenfalls nicht möglich wäre, das Modul auf dem Entwicklungssystem zu testen. Die Testfälle laufen sowohl auf dem Entwicklungssystem als auch auf dem Zielsystem problemlos. Es ist somit möglich, dieses Modul vollständig zu testen und komplett auf dem Entwicklungssystem zu implementieren.
Der vollständige Code, welcher für diese Tests notwendig ist, findet sich im Anhang A.2.

4.3.2 Testen von Hardware Treibern mit Mocks

Das Prinzip der Mock Objekte wurde im Jahr 2000 erstmals von Tim Mackinnon, Steve Freeman und Philip Craig in ihrem Paper *Endo-Testing: Unit Testing with Mock Objects* beschrieben [MFC00]. Sie waren auf der Suche nach einer Methodik, die es ihnen erlaubte, komplexere Interaktion, wie beispielsweise die mit einer Datenbank, zu testen. Die in dem Paper beschriebenen Mocks arbeiten nach dem folgenden Muster [MFC00, S. 8].

1. Erzeugen einer Instanz des Mock Objekts.
2. Festlegen des Zustands des Mock Objekts.
3. Definieren der Erwartungen.
4. Aufruf des zu testenden Codeeinheit.
5. Verifizieren ob die Erwartungen erfüllt wurden.

Die Methode erlaubt es, statt lediglich Zustände durch einen Test zu überprüfen, das Verhalten zu testen. Alexander Tarlinder schreibt über Mock Objekte das Folgende [Tar17, S. 164].

> *„Mock objects are game changers in a way, as they shift a test's focus from state to behaviour. [...] Their goal is to verify that certain interactions have occured between the mock object and the tested code or another collaborator.“*

Das im vorigen Kapitel vorgestellte Testprojekt nutzt einen BNO055 Smart Sensor, um Schnellwertüberschreitungen zu detektieren. Auf der Software-Seite wird es durch ein

Treiber Modul angesteuert. Die Kommunikation läuft über einen I2C-Bus. Dieser erlaubt das Schreiben und Lesen von einzelnen Registern des Sensors. Viele der Interaktionen mit dem Sensor setzen sich dabei aus mehreren aufeinanderfolgenden Schreib- und Leseoperationen zusammen. Das Treiber-Modul soll diese komplexen Interaktionen vor dem aufrufenden Code verstecken und die Funktionalitäten über eine möglichst einfache Schnittstelle zur Verfügung stellen. Dadurch wird allerdings das Testen des Moduls erschwert, da die Rückgabewerte der Funktionen in keinster Weise die Komplexität der Implementierung an sich widerspiegeln können. Es ist daher der perfekte Fall für den Einsatz eines Mock Objekts.

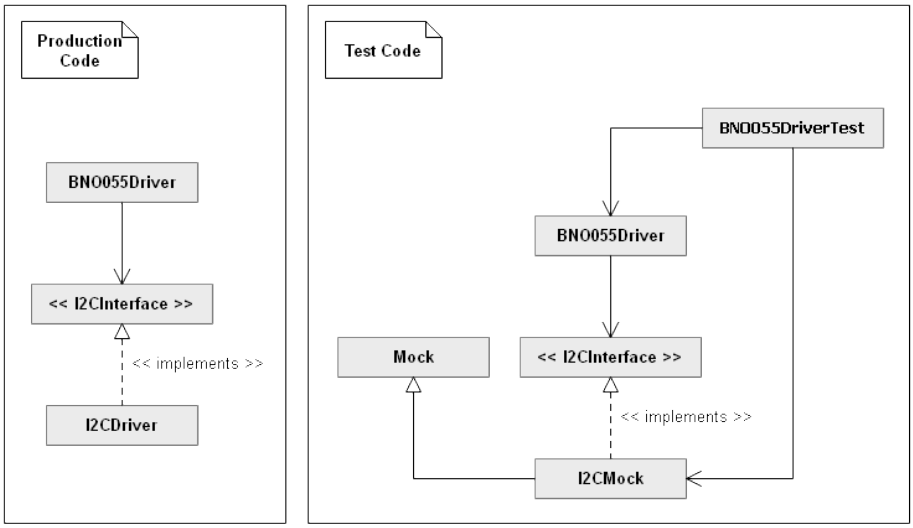

Abbildung 4.4: Klassendiagramme des BNO055-Treiber Moduls im Produktions Kontext und im Test Kontext.

Abbildung 4.4 stellt in dem linken Klassendiagramm die Abhängigkeiten des Sensor-Treiber Moduls dar. Es greift auf ein I2C-Treiber Modul zu, um die entsprechenden Schreib- und Lesezugriffe auf den Bus zu senden. Das `I2CDriver`-Modul implementiert eine Schnittstelle, welche von dem `I2CInterface` (Abbildung 4.5) beschrieben wird. Es definiert Methoden zum Lesen und Schreiben von einem Register und zum Lesen mehrerer aufeinanderfolgender Register.

Um den Sensor-Treiber auf dem Entwicklungssystem programmieren zu können, muss der `I2CDriver` durch einen Mock substituiert werden. Im rechten Klassendiagramm

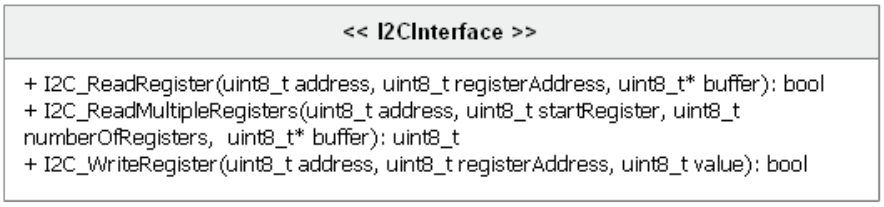

Abbildung 4.5: Das I2C Interface.

(Abbildung 4.4) ist dargestellt, wie die Abhängigkeiten zwischen den Modulen für die Tests aussehen. Der `I2CDriver` wird durch `I2CMock` ausgetauscht. Dieses erbt von einem `Mock`-Modul. Aus Gründen der Wiederverwendung wurden die grundlegenden Funktionen des Mocks in einem eigenen Modul implementiert. So ist es möglich schneller Mock Objekte mit für verschiedene Zwecke umzusetzen.

```
21 /**
22  * @brief Mock object struct.
23  */
24 typedef struct _Mock Mock_t;
25
26 /**
27  * @{
28  * @name Mock initialization and deinitialization functions
29  */
30 Mock_t* Mock_Create(uint8_t maxExpectations);
31 void Mock_Destroy(Mock_t* handle);
32 /** @} */
33
34 /**
35  * @{
36  * @name Mock expectation control functions
37  */
38 bool Mock_AddExpectation(Mock_t* handle, void* expectation);
39 void* Mock_GetNextExpecation(Mock_t* handle);
40 /** @} */
41
42 /**
43  * @{
44  * @name Mock check functions
45  */
```

```
46 void Mock_Check(Mock_t* handle, bool condition, const char* format,
       ...);
47 /** @} */
48
49 /**
50  * @{
51  * @name Mock verify and report functions
52  */
53 bool Mock_Verify(Mock_t* handle);
54 char* Mock_Report(Mock_t* handle);
55 /** @} */
```

Listing 4.16: Die Schnittstelle des `Mock`-Moduls (`Mock.h`).

`Mock` stellt die in Listing 4.16 gezeigten Methoden zur Verfügung. Sie erlauben Modulen, die von `Mock` erben, das Hinzufügen von Erwartungen, das Überprüfen von Bedingungen und das Verifizieren der erwarteten Sequenz. Mit `Mock_Report()` kann ein Ausgabetext generiert werden, der abgefangene Fehlverhalten beschreibt. Die vollständige Implementierung von `Mock` findet sich im Anhang A.3.
Der `I2CMock` implementiert ebenfalls die von `I2CInterface` definierte Schnittstelle (Listing 4.17) und prüft bei deren Aufruf, ob die Bedingungen, die durch die Erwartungen festgelegt wurden, erfüllt werden. Die Erwartungen werden mit dem Datentyp `Expectation_t` beschrieben.

```
17 /**
18  * @brief Struct for holding the attributes of an mock expectation.
19  */
20 typedef struct
21 {
22     Interaction_Type_t type;
23     uint8_t address;
24     uint8_t registerAddress;
25     uint8_t* data;
26     uint8_t numberOfRegisters;
27 } Expectation_t;
28
29 /**
30  * @brief Mock instance
31  */
32 static Mock_t* mock;
33 /**
34  * @brief I2C interface implementation
35  */
```

```
36 static I2C_Interface_t io;
37
38 static bool readRegister(uint8_t address, uint8_t registerAddress,
       uint8_t* buffer);
39 static uint8_t readMultipleRegisters(uint8_t address, uint8_t
       startRegister, uint8_t numberOfRegisters, uint8_t* buffer);
40 static bool writeRegister(uint8_t address, uint8_t registerAddress,
       uint8_t value);
41
42 /**
43  * @{
44  * @name I2CMock initialization and deinitialization functions
45  */
46 /**
47  * @brief      Initializes the instance of the I2C Mock.
48  *
49  * @param[in]  maxExpectations  The maximum number of expectations
50  */
51 void I2CMock_Create(uint8_t maxExpectations)
52 {
53     io.readRegisterFunction = readRegister;
54     io.readMultipleRegistersFunction = readMultipleRegisters;
55     io.writeRegisterFunction = writeRegister;
56
57     mock = Mock_Create(maxExpectations);
58 }
```

Listing 4.17: Ausschnitt der Implementierung des `I2CMock`-Moduls (`I2CMock.c`).

Für den Austausch von `I2CDriver` durch `I2CMock` wurde die Funktionszeiger-Substitution gewählt. Es bietet sich in diesem Fall an, da die Interface-Schnittstelle bereits vorsieht, dass ein `struct` mit den Funktionszeigern übergeben wird.

Um zu demonstrieren, wie das Mock Objekt eingesetzt wird, soll hier ein Testfall betrachtet werden, der prüft ob der `BNO055Driver` detektieren kann, dass der Selbsttest des Sensors fehlschlägt. Das Sequenzdiagramm in Abbildung 4.6 zeigt, welche Interaktionen stattfinden müssen, um zunächst den Selbsttest zu starten und anschließend das Resultat abzufragen. Als Erstes wird der Sensor in den Konfigurations-Modus versetzt, gefolgt vom Starten des Selbsttest und einer Überprüfung, ob der Test läuft. Zum Schluss wird das Ergebnis ausgelesen.

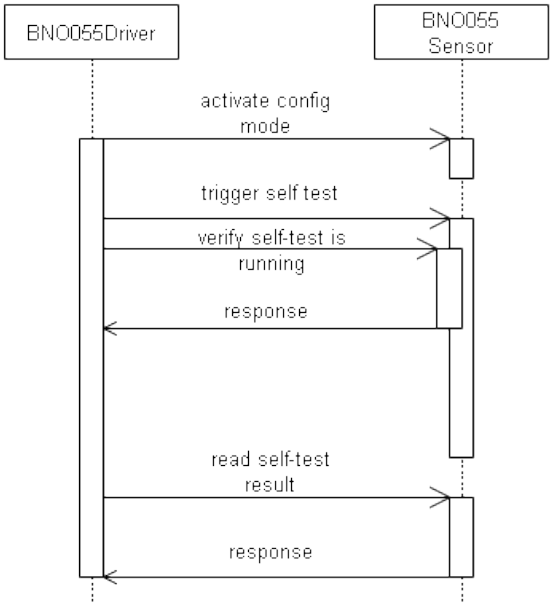

Abbildung 4.6: Sequenz-Diagramm der Interaktion zwischen dem Treiber Modul und dem Sensor zur Durchführung des Selbst-Tests.

In Listing 4.18 ist der Testfall gezeigt, der verifiziert, dass der `BNO055Driver` einen fehlgeschlagenen Selbsttest des Sensors erkennt. Er stellt ein gutes Beispiel dar, wie Testfälle die Mocks nutzen, formuliert werden. Zunächst werden die Erwartungen festgelegt. In diesem Fall werden vier Interaktionen erwartet, angefangen mit zwei Schreibzugriffen und gefolgt von zwei Lesezugriffen. Dann erfolgt der Aufruf der zu testenden Funktion `BNO055_ExecuteSelfTest()`. Es folgt der Aufruf von `I2CMock_Verify()`, die auswertet, ob die erwarteten Interaktionen wie beschrieben stattgefunden haben. Mit einer Assertion wird zum Schluss noch geprüft, ob tatsächlich festgestellt wurde, dass der Selbsttest fehlgeschlagen ist.

```
90 TEST(BNO055DriverTest, ExecuteFailingSelfTest)
91 {
92     // arrange
93     uint8_t dataOprModeReg = 0x00;
94     I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3D,
       &dataOprModeReg, 1);
95     uint8_t dataSysTrigReg = 0x01;
```

```
96      I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3F,
        &dataSysTrigReg, 1);
97      uint8_t dataSysStatusReg = BNO_EXECUTING_SELFTEST;
98      I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x39,
        &dataSysStatusReg, 1);
99      uint8_t dataSysErrorReg = BNO_SELF_TEST_FAILED;
100     I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x3A,
        &dataSysErrorReg, 1);
101
102     // act
103     bool result = BNO055_ExecuteSelfTest();
104
105     // assert
106     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
107     TEST_ASSERT_FALSE(result);
108 }
```

Listing 4.18: Testfall zur Überprüfung eines fehlgeschlagenen Sensor-Selbsttests mit Hilfe eines Mock Objekts (`BNO055DriverTest.c`).

Durch den Einsatz von Mock Objekten wird es möglich, nahezu jede Interaktion mit Peripherie-Geräten auf dem Entwicklungssystem zu testen. Besonders Peripherien, die über Schnittstellen wie I2C, UART, SPI oder ähnlichen angebunden werden, können mit Mocks sehr einfach testgetrieben entwickelt werden, ohne die Hardware final vorliegen zu haben. Ein weiterer Vorteil ist, dass Zustände des Collaborators simuliert werden können, wie der hier gezeigte fehlgeschlagene Selbsttest, die in der Hardware nicht ohne weiteres herzustellen wären. Der vollständige Quellcode zu diesem Kapitel findet sich im Anhang A.4

4.3.3 Testen von Zugriffen auf Register

Ein häufiges Problem, das besonders beim Testen von Hardware-Abstraktionen auf dem Entwicklungssystem auftritt, ist das direkte Adressieren von Speicher-Registern. Mit solchen Registern werden beispielsweise Peripherien konfiguriert und angesteuert. Ein grundlegender Unterschied des Entwicklungssystem (wahrscheinlich ein Linux oder Windows Betriebssystem) zur Zielhardware ist der Einsatz einer Memory Management Unit (MMU). Sie sorgt unter anderem dafür, dass Prozesse nur mit virtuellen Speicheradressen arbeiten. Auf einem Mikroprozessor ist hingegen meist keine MMU vorhanden. Daher

verweisen die Zeiger direkt auf den physikalischen Speicher. Führt man nun auf dem Entwicklungssystem Code aus, der einen Zugriff über einen Zeiger auf einen Speicherregister enthält, ist es sehr wahrscheinlich, dass die MMU einen illegalen Speicherzugriff erkennt und die Ausführung des Prozesses vom Betriebssystem terminiert wird. Dieses Kapitel stellt einen Ansatz vor, der es dennoch ermöglicht, Zugriffe auf Register auf dem Entwicklungssystem zu entwickeln.

Als Beispiel dient der GPIO-Treiber des Derailing Alarm Systems. Seine Aufgabe ist das Konfigurieren und Ansteuern der GPIO-Pins des Mikrocontrollers. Dazu greift er auf verschiedene Register zu, die mit Hilfe eines Zeigers adressiert werden.
Listing 4.19 zeigt die Schnittstellen-Beschreibung des GPIO-Treibers. Damit die hier vorgestellte Strategie funktioniert, ist es wichtig, dass Speicheradressen von außen an das zu testende Modul übergeben werden können. In diesem Fall hat jede der Methoden einen Parameter `volatile uint32_t* gpioBankAddress`, mit dem die Adresse der jeweiligen GPIO-Bank an die Methode übergeben werden kann. Auch denkbar wäre, beim Erstellen einer Instanz eines Moduls, einmalig eine Speicheradresse zu übergeben. Der `volatile` Modifier ist ein Hinweis für den Compiler, dass es sich bei dem adressierten Element um einen flüchtigen Speicher handelt. Dadurch wird der Compiler instruiert, dass scheinbar redundante Zugriffe auf diese Adresse nicht optimiert werden dürfen, da sich dadurch das Verhalten ändern könnte [KR88, S. 211].

```
38 /**
39  * @{
40  * @name GPIO mode configuration functions
41  */
42 void HAL_GPIO_ConfigureInput(volatile uint32_t* gpioBankAddress,
       uint8_t gpioNumber);
43 void HAL_GPIO_ConfigureOutput(volatile uint32_t* gpioBankAddress,
       uint8_t gpioNumber);
44 /** @} */
45
46 /**
47  * @{
48  * @name GPIO read status functions
49  */
50 HAL_GPIO_State_t HAL_GPIO_ReadInput(volatile uint32_t*
       gpioBankAddress, uint8_t gpioNumber);
51 HAL_GPIO_State_t HAL_GPIO_ReadOutput(volatile uint32_t*
       gpioBankAddress, uint8_t gpioNumber);
52 /** @} */
```

```
53
54 /**
55  * @{
56  * @name GPIO state control functions
57  */
58 void HAL_GPIO_SetOutput(volatile uint32_t* gpioBankAddress, uint8_t
       gpioNumber);
59 void HAL_GPIO_ResetOutput(volatile uint32_t* gpioBankAddress,
       uint8_t gpioNumber);
60 /** @} */
```

Listing 4.19: Schnittstellen-Beschreibung des GPIO-Treibers (`HAL_GPIO.h`).

In Listing 4.20 ist zu sehen, wie für das Testen auf dem Entwicklungssystem die adressierten Register durch eine statische Variable ersetzt werden können. Es wird zunächst ein Array `gpioRegisters` mit 21 Elementen angelegt. Er soll als Ersatz für den Speicherbereich dienen, auf den der GPIO-Treiber zugreift. Dann wird ein Zeiger `gpioBankAddress` deklariert, dem in der Setup-Funktion die Adresse, des ersten Elements des Arrays, zugewiesen wird. So kann umgangen werden, in den Testfällen mit absoluten Adressen arbeiten zu müssen. Stattdessen nutzt man nun die vom Betriebssystem zugewiesenen virtuellen Adressen. Auf diese Weise können Zugriffe auf nicht erlaubte Speicherbereiche ausgeschlossen werden.

```
14 volatile static uint32_t gpioRegisters[21];
15 volatile static uint32_t* gpioBankAddress;
16
17 /**
18  * @{
19  * @name HAL_GPIO test group
20  */
21 TEST_GROUP(HAL_GPIOTest);
22
23 TEST_SETUP(HAL_GPIOTest)
24 {
25     gpioRegisters[0] = 0;
26     gpioRegisters[4] = 0;
27     gpioRegisters[8] = 0;
28     gpioRegisters[12] = 0;
29     gpioRegisters[16] = 0;
30     gpioRegisters[20] = 0;
31
32     gpioBankAddress = &gpioRegisters[0];
33 }
```

Listing 4.20: Setup für die Tests des GPIO-Treibers. (HAL_GPIOTest.c).

Ein weiterer Vorteil dieser Strategie ist, dass die volle Kontrolle über den Inhalt des Speicherbereichs bei den Tests liegt. Wie in Listing 4.21 demonstriert, kann der Speicher zum Beispiel mit einem bestimmten Bitmuster gefüllt werden. Das ermöglicht eine Verifizierung von bitweisen Operationen mit dem Register.

```
45 TEST(HAL_GPIOTest, SettingOneInputMakesCorrectChangesToModeRegister)
46 {
47     // arrange
48     gpioRegisters[0] = 0x5555;
49
50     // act
51     HAL_GPIO_ConfigureInput(gpioBankAddress, 0);
52
53     // assert
54     TEST_ASSERT_EQUAL_HEX32(0x5554, gpioRegisters[0]);
55 }
```

Listing 4.21: Testfall zum Überprüfen eines Register-Zugriffs (HAL_GPIOTest.c).

Der vollständige Quellcode zu diesem Kapitel findet sich im Anhang A.5.

4.4 Auswirkungen auf die Software-Entwicklung

Die testgetriebene Entwicklung von eingebetteter Software verändert nicht nur die Arbeitsweise beim Programmieren, sondern hat auch Einfluss auf den System Entwurf und die Programmierung in C. In diesem Kapitel wird beleuchtet, wie sich diese Einflüsse bemerkbar machen.

4.4.1 System Entwurf

Um testgetrieben eingebettete Software entwickeln zu können, muss der Code gut testbar sein. Insbesondere die Anwendung des Dual-Targetings muss auch bereits beim System Entwurf bedacht werden. Robert C. Martin schreibt zu diesem Thema [Mar20, S. 261]:

> *„When embedded code is structured without applying clean architecture principles and practices, you will often face the scenario in which you can test your code only on the target. If the target is the only place where testing is possible, the target bottleneck will slow you down.“*

Er kritisiert, dass eingebettete Software in vielen Fällen nicht streng genug zwischen Anwendungsebene und Systemebene trennt. Die Anwendungsebene sollte keinerlei direkten Abhängigkeiten zu plattformspezifischen Funktionalitäten enthalten. D.h. die Anwendungsebene darf weder direkt auf Speicherregister zugreifen, noch darf sie direkt mit der Peripherie interagieren. Er geht sogar noch einen Schritt weiter und untersagt der Anwendungsebene die direkte Interaktion mit dem Betriebssystem, soweit vorhanden [Mar20, S. 261 ff.]. Sowohl Abhängigkeiten zur Hardware, als auch Abhängigkeiten zum Betriebssystem in der Anwendungsebene führen dazu, dass Tests nur noch auf dem Zielsystem durchgeführt werden können. Die Lösung sind Abstraktionsschichten, von Martin als Hardware Abstraction Layer (HAL) und Operating System Abstraktion Layer (OSAL) bezeichnet, welche als Schnittstelle zwischen Anwendungs- und Systemebene dienen. Sie abstrahieren die Funktionalität von Hardware und Betriebssystem. Auf diese Weise können die Schnittstellen, mit den in Kapitel 4.3 beschriebenen Strategien, für die Tests ersetzt werden. Den Zweck der Abstraktionsebenen fasst Martin wie folgt zusammen [Mar20, S. 272].

> *„A clean embedded architecture is testable within the layers because modules interact through interfaces. Each interface provides that seam or substitution point that facilitates off-target testing.“*

Abbildung 4.7 zeigt, wie sich die HAL und die OSAL Abstraktionsebenen in die Software Architektur einfügen. Sie befinden sich zwischen der Anwendungsebene (in der Abbildung mit *Software* bezeichnet) und der Betriebssystem-Ebene bzw. der Systemebene (mit *Firmware* bezeichnet).

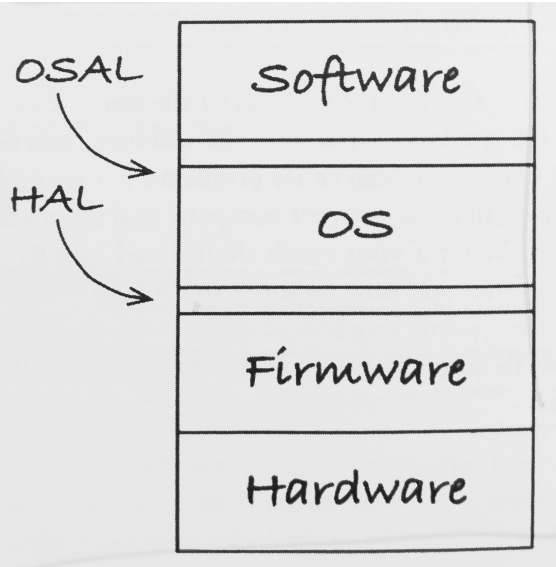

Abbildung 4.7: Die HAL und OSAL Abstraktionsebenen in der Software Architektur [Mar20, S. 270].

Neben der verbesserten Testbarkeit haben die Abstraktionsebenen auch einen positiven Einfluss auf die Portierbarkeit der Anwendungssoftware. Da sie keinerlei Abhängigkeiten mehr zur Hardware oder zum Betriebssystem hat, kann sie durch Anpassung der Abstraktionsebenen einfacher auf anderen Plattformen zum Laufen gebracht werden.

Beim System Entwurf sollte generell Wert auf die Reduzierung von Komplexität gelegt werden. Dies ist auch für die Testbarkeit wichtig. Denn das Testen soll nicht nur Defekte in der Software aufzeigen, sondern auch dabei helfen, ihre Ursache zu identifizieren. Es ist somit notwendig, das Programm für einen Test in einen definierten Zustand bringen zu können. Ist die Komplexität zu hoch, wird dies deutlich erschwert, da an zu vielen Stellen eingegriffen werden muss. Um die Komplexität zu verringern, sollten die Abhängigkeiten

zwischen einzelnen Modulen möglichst gering gehalten werden [BCK12, S. 167]. Die Abhängigkeiten zwischen Modulen kann reduziert werden, indem die Module ein möglichst reduziertes öffentliches Interface aufweisen und Details und komplexere Funktionalitäten in der Implementierung vor den anderen Modulen verstecken. Diese Design-Technik wird auch als *Deep Module* bezeichnet [Ous18, S. 22 f.].

Code-Einheiten, welche durch einen Komponententest überprüft werden sollen, meistens Funktionen, sollten so entworfen werden, dass sie für eine bestimmte Kombination aus Parametern ein deterministisches Verhalten aufweisen und das Resultat über geeignete Rückgabewerte klar kommunizieren. Deterministisches Verhalten hängt stark von der Möglichkeit ab, das Programm vor dem Test in einen definierten Zustand bringen zu können. Dies muss bereits beim Entwurf bedacht werden.

4.4.2 Implementierung in C

Auch an die Implementierung der Funktionalität in C entstehen durch die testgetriebene Entwicklung andere Ansprüche. Im vorherigen Abschnitt wurde bereits für eine klarere Struktur plädiert. Wichtig sind weiterhin auch Modularität und Kapselung. Alle diese Aspekte werden durch eine prozedurale Programmiersprache nicht aktiv fokussiert und liegen in der Verantwortung der Programmiererinnen und Programmierer. Gerade Objektorientierte Programmierung (OOP) fördert die genannten Eigenschaften wesentlich besser. Daher lohnt es sich, auch in C die Konzepte der Objektorientierung aufzugreifen. Das zentrale Element der OOP ist sicherlich das Konzept der Klassen, also eine Einheit, die sowohl Daten (Attribute) als auch Funktionalität (Methoden) logisch zusammenfasst. Im Folgenden wird darauf eingegangen, wie in C Objektorientierung imitiert werden kann.

Kapselung von Attributen

C bietet mit `struct` bereits ein Feature, das es erlaubt, Daten zusammenzufassen. Mittels einer Typdefinition (`typedef`) ist es sogar möglich, einem `struct` einen eigenen Datentypnamen zuzuweisen. Die grundlegende Idee ist es, für jede Klasse ein eigenes `struct` zu definieren, welches die Klassenattribute enthält. Listing 4.22 zeigt wie eine solche Kapselung aussehen könnte.

```
typedef struct _Class Class_t;

struct _Class
{
    uint8_t attribute1;
    ...
};
```

Listing 4.22: Ein `struct` kapselt zusammengehörige Attribute.

Ein weiteres wichtiges Konzept der OOP ist die Sichtbarkeit von Klassen-Membern. Attribute die für aufrufenden Code nicht relevant sind, können in ihrer Sichtbarkeit auf die eigene Klasse beschränkt werden. An dieser Stelle kommt die sogenannte *Forward Declaration* zum Tragen. Das bedeutet, die Typdefinition findet in der Header-Datei statt, während das eigentliche `struct` in der Quelldatei definiert wird. Listings 4.23 und 4.24 demonstrieren dieses Vorgehen.

```
typedef struct _Class Class_t;
```

Listing 4.23: Typdefinition in der Header-Datei.

```
#include "Class.h"

typedef struct _Class
{
    uint8_t attribute1;
    ...
} Class_t;
```

Listing 4.24: Definition des `struct` in der Quelldatei.

Auf diese Weise wird erreicht, dass der Datentyp zwar außerhalb der Klasse durch einbinden des Headers bekannt ist, jedoch das Innere seiner Struktur verborgen bleibt. Dies funktioniert, da in C die Datei den Scope für Variablen vorgibt.

Kapselung von Methoden

Die Kapselung von Methoden gestaltet sich schwieriger, als die von Attributen. Anders als in der Objektorientierte Programmierung ist es nicht möglich eine Funktion fest an ein Objekt zu binden. Es kann jedoch, wie in Listing 4.25 gezeigt, bei jeder Methode

mit ein Parameter ein Zeiger auf die Klassen-Struktur übergeben werden. Somit kann die Implementierung der Methode die Klassen-Attribute manipulieren.

```
1 void Class_Method1(Class_t* instance, uint8_t parameter);
```

Listing 4.25: Prototyp einer Klassen-Methode.

Listings 4.26 und 4.27 zeigen exemplarisch wie eine komplette Klasse in C umgesetzt werden kann. Die beiden Funktionen `Class_Create()` und `Class_Destroy()` erfüllen den selben Zweck wie ein Constructor und ein Destructor. Sie erzeugen bzw. zerstören eine Instanz der jeweiligen Klasse. Der Zeiger auf die Klassen-Struktur hat die selben Zweck, der in einer OOP durch eine Objekt-Referenz erfüllt werden würde.

```
1  #ifndef CLASS_H
2  #define CLASS_H
3
4  typedef struct _Class Class_t;
5
6  Class_t* Class_Create(void);
7  void Class_Destroy(Class_t* instance);
8
9  void Class_Method1(Class_t* instance, uint8_t parameter);
10 ...
11
12 #endif /* CLASS_H */
```

Listing 4.26: Ein exemplarischer Klassen-Header in C.

```
1  #include "Class.h"
2
3  typedef struct _Class
4  {
5      uint8_t attribute1;
6      ...
7  } Class_t;
8
9  Class_t* Class_Create(void)
10 {
11     Class_t* instance = (Class_t*) malloc(sizeof(Class_t*));
12     if(instance != NULL)
13     {
14         instance->attribute1 = 0;
15         ...
16     }
```

```
17      return instance;
18  }
19
20  void Class_Destroy(Class_t* instance)
21  {
22      if(instance != NULL)
23      {
24          free(instance);
25      }
26  }
27
28  void Class_Method1(Class_t* instance, uint8_t parameter)
29  {
30      instance->attribute1 = parameter;
31      ...
32  }
33  ...
34
35  #endif CLASS_H
```

Listing 4.27: Eine exemplarische Klassen-Implementierung in C.

Wichtig ist an dieser Stelle anzumerken, dass es in sicherheitskritischen Systemen vermieden werden sollte, dynamisch Speicher zu allokieren [Lac18, S. 103]. Ein Fehler im Heap-Management könnte katastrophale Folgen haben. Wie Listings 4.28 und 4.29 zeigen, können Klassen in C aber auch statisch erzeugt werden. Anstatt den Speicher dynamisch in der `Create`-Funktion zu reservieren, wird er in diesem Fall zum Kompilierzeitpunkt zur Verfügung gestellt. Der Nachteil ist, dass der genutzte Speicher für die komplette Laufzeit des Programms belegt bleibt, obwohl die Instanz unter Umständen nicht weiter benötigt wird.

```
1  Class_t instance;
2  Class_Create(&instance);
```

Listing 4.28: Deklaration und Initialisierung einer nicht dynamisch allokierten Klassen-Instanz.

```
1  void Class_Create(Class_t* instance)
2  {
3      instance->attribute1 = 0;
4      ...
5  }
```

Listing 4.29: Initialisierung einer Klassen-Instanz ohne dynamisch allokierten Speicher.

Das Buch *Extreme C* [Ami19] von Kamran Amini liefert zu dem Thema Objektorientierte Programmierung in C detaillierte Ausführungen und geht auch auf weiterführende Aspekte, wie Vererbung oder Polymorphismus ein.

4.5 Toolchain

Die testgetriebene Entwicklung ist allen voran eine Arbeitsweise. Sie verändert grundlegend das Vorgehen bei der Implementierung von Code. Ziel ist es, die Zeit zwischen Entstehung eines Defekts und dessen Erkennung drastisch zu verkürzen. Dazu wird in sehr kurzen Zyklen der Code iterativ geschrieben. Der Code muss daher, teilweise mehrfach pro Minute, kompiliert und die Testfälle ausgeführt werden müssen. Um hier nicht unnötige Zeitverluste zu erzeugen, ist es essentiell, die Toolchain effektiv einzusetzen. In diesem Kapitel wird beschrieben, welche Tools notwendig sind und wie sie sich in den testgetriebenen Zyklus einfügen. Die betrachteten Tools wurden für das in Kapitel 4.3 vorgestellte Beispielprojekt eingesetzt. Natürlich stellen sie nicht die einzige Lösung für eine Toolchain dar, liefern aber Anhaltspunkte, wie eine Toolchain für die testgetriebene Entwicklung aussehen könnte.

Das Beispielprojekt wurde komplett auf einem Ubuntu 20.04 Betriebssystem umgesetzt. Die Entscheidung mit einer Linux Distribution anstelle eines Windows Betriebssystems zu arbeiten, wurde aufgrund der großen Auswahl an quelloffenen Software-Werkzeuge getroffen. Die im Folgenden vorgestellten Tools sind zwar durchgehend auch für Windows verfügbar, sind dort allerdings wesentlich aufwendiger zu installieren. Zudem wäre es sehr wahrscheinlich notwendig, eine Softwareumgebung wie Cygwin[10] zu nutzen. Die Toolchain wird dadurch komplexer, während mit Linux alle Tools nativ ausgeführt werden können. Ist nur ein System mit Windows verfügbar, so stellt eine virtuelle Maschine mit einer Linux Distribution eine gute Alternative dar. Hinzu kommt, dass Build-Server meist ebenfalls auf einem Linux System aufbauen. Die Verwendung von Linux für das Entwicklungssystem verhindert, dass an dieser Stelle Fehler ergeben, aufgrund von Unterschieden in den zugrundeliegenden Plattformen.

Integrated Development Environment (IDE)

Integrated Development Environments kombinieren einen Text-Editor, eine Build-Pipeline, einen Debugger und viele weitere Funktionalitäten in einer einzigen grafischen Oberfläche. Sie werden meist von den Chip-Herstellern selber angeboten. Es existiert eine große Anzahl am Markt an IDEs für eingebettete Software. Unterschiede liegen zum Beispiel

[10] `https://www.cygwin.com/`

in den unterstützen Compilern, den Debugging-Schnittstellen, aber auch in den verfügbaren Lizenzen. Während die IDEs ein mächtiges Werkzeug darstellen, so können sie für die testgetriebene Entwicklung von eingebetteter Software zum Hindernis werden. Schließlich ist es wichtig, viele der in jeder Iteration vorkommenden Arbeitsschritte zu automatisieren. Die grafische Oberfläche ist dafür eher hinderlich. David Thomas und Andrew Hunt haben zu dem Thema folgenden Kritikpunkt [TH20, S. 78 f.].

> *„GUI environments are normally limited to the capabilities that their designers intended. If you need to go beyond the model the designer provided, you are usually out of luck–and more often than not, you do need to go beyond the model.“*

Sie plädieren für den Einsatz von Plain-Text und Shell-Scripten, um die Einschränkungen der IDE zu überkommen.
Dieser Ansatz wurde auch für das Beispielprojekt gewählt. Das Dual-Targeting hat die Anforderung, den Code mit zwei unterschiedlichen Build-Pipelines übersetzten zu können. Der Aufwand dies mit einer grafischen Oberfläche zu realisieren, die für dieses Konzept nicht ausgelegt ist, wäre erheblich. Spätestens für den Build-Vorgang auf einem Build-Server, wäre es zwingend notwendig, die IDE zu umgehen. Ein integrales Verständnis für den Build-Vorgang hilft, die in den Kapiteln 4.2 und 4.3 beschriebenen Konzepte wirksam einzusetzen.
Natürlich bieten Integrated Development Environments aber auch Vorteile. So können sie beim Refactoring durch Features unterstützen, die zum Beispiel ein leichtes Editieren von Benamungen oder das Extrahieren von Funktionen ermöglichen.

Text-Editor

Verzichtet man gänzlich auf eine IDE, sollte ein Text-Editor gewählt werden, der die Programmierung in C aktiv fördert. Das Mindeste ist ein gutes Code-Highlighting, um das Navigieren des Quellcodes zu erleichtern und Syntax-Fehler hervorzuheben. Meist beschränken sich die Features zum Refactoring leider auf das Umbenennen von Namen. Auch die Vervollständigung von Code kommt nicht an die Qualität einer IDE heran. Ist für ein Projekt ein Integrated Development Environment vorhanden, das über einen guten Editor verfügt, so ist es ratsam, sie zumindest für diesen Zweck einzusetzen.

Build-Automation

Der Build-Vorgang sollte für die testgetriebene Entwicklung von eingebetteter Software dringend von der IDE entkoppelt werden. Für das Dual-Targeting müssen zwei Build-Pipelines realisiert werden, eine für das Entwicklungssystem und eine für das Zielsystem. Mit diesen beiden Pipelines werden dann unterschiedliche Executables erzeugt. Es gibt ein Executable, welches den Production-Code, also die Anwendung die mit der Hardware ausgeliefert wird, enthält. Für die Testfälle gibt es mindestens zwei Executables, eines für das Entwicklungssystem und eines für die Zielhardware. Je nach Bedarf, kann es strategisch auch lohnenswert sein, die Testfälle auf mehrere Executables aufzuteilen. Die Folge ist, dass der Überblick über eine große Anzahl an Build-Vorgängen mit unterschiedlichen Konfigurationen behalten werden muss. Abhilfe kann hier ein Werkzeug wie *GNU make*[11] schaffen. Es erlaubt mit Hilfe von sogenannten *Makefiles* die Beschreibung und Automatisierung von Build-Vorgängen. Dabei beschreiben einzelne *Targets* die gewünschten Executables. In dem Beispielprojekt wurden ein Target für den Production-Code, eines für die Tests auf dem Entwicklungssystem und eines für Tests auf dem Zielsystem festgelegt. Die Abhängigkeiten können für jedes Target seperat beschrieben werden. Sind die Abhängigkeiten sauber definiert, sorgt das Werkzeug dafür, dass nur Einheiten, die seit dem vorherigen Build modifiziert wurden, neu übersetzt werden. Das minimiert die Build-Dauer und erhöht somit die Geschwindigkeit der einzelnen Iterationen. Mit *Rules* wird festgelegt, wie einzelne Elemente zu erstellen sind. *Managing Projekts with GNU Make* [Mec05] bietet ein gute Einführung in das Werkzeug.
James Grenning stellt folgende Anforderungen an einen Makefile für die testgetriebene Entwicklung [Gre11, S. 293].

- Das Makefile muss einen schnellen schrittweisen Build basierend auf den Abhängigkeiten ermöglichen.
- Die Tests müssen mit jedem Build ausgeführt werden.
- Die Test-Dateien haben Priorität gegenüber Production-Code.
- Build-Artefakte dürfen sich nicht im Projekt-Verzeichnis sammeln.

Für das Beispielprojekt wurden für die Tests zwei Targets, je eines pro System, definiert. Sie werden einfach aus der Kommandozeile durch `$ make check` bzw. `$ make check_target` aufgerufen. Sie sorgen nicht nur dafür, dass die jeweiligen Executables erzeugt werden, sondern führen die Testfälle im Anschluss auch direkt auf der jeweiligen

[11] `https://www.gnu.org/software/make/`

Plattform aus. Beim Programmieren reicht daher ein einziger Befehl in der Kommandozeile aus, um die Änderungen der aktuellen Iteration zu überprüfen. Auf dem Entwicklungssystem benötigt der Build und die Ausführung der Tests etwa 1,6 s. Für das Zielsystem werden etwa 9,7 s beansprucht. Dies liegt unter anderem auch an dem zeitintensiven Programmiervorgang des Hardware. Es wird deutlich, wie das Dual-Targeting die Entwicklungs-Zyklen enorm beschleunigen kann.

Scripte

In den einzelnen Iterationen der testgetriebenen Entwicklung müssen bestimmte Tätigkeiten wiederholt ausgeführt werden. Geschieht dies manuell, beanspruchen diese Tätigkeiten immer wieder Zeit und führen zur Frustration beim Arbeiten. Dies ist für die testgetriebene Arbeitsweise unbedingt zu vermeiden, denn es könnte aufgrund ihrer repetitiven Natur auf Dauer zu Nachlässigkeiten in der Arbeitsweise führen. Es lohnt sich daher Aufgaben, die in jeder Iteration vorkommen, möglichst früh zu automatisieren. Mehr zu dem Thema Automatisierung folgt im Kapitel 4.6. Für die Automatisierung kann eine Script-Sprache der Wahl eingesetzt werden. Sie haben den Vorteil, dass simple Aufgaben mit ihnen schnell und effizient umgesetzt werden können. Das Beispielprojekt setzt verschiedene *Bash*-Scripte ein. So wird beispielsweise der *Test Runner*, also die Quelldatei, welche die Main-Funktion zur Ausführung der Testfälle enthält, automatisch bei jedem Test-Build erneut generiert. Das Programmieren des Mikrocontrollers wird ebenfalls mit einem Script umgesetzt.

Debugging

Beim Debugging zeigt sich ein weiterer Vorteil des Dual-Targetings. Das Debuggen kann auf dem Entwicklungssystem durchgeführt werden. Dies ist in der Regel einfacher, als auf dem Zielsystem zu debuggen. Es muss nicht zunächst einmal eine Verbindung über eine Debugging-Schnittstelle, wie JTAG oder SWD, aufgebaut werden. Stattdessen kann das Test-Executable direkt zur Laufzeit auf dem Entwicklungssystem debuggt werden. Generell gilt, dass durch die testgetriebene Entwicklung die Zeit, die mit Debuggen verbracht wird, sich deutlich reduziert. Durch die kurzen Iterationszyklen lässt sich die Ursache für einen neuen Defekt häufig auf wenige Zeilen Code eingrenzen. Es sollte jedoch darauf geachtet werden, einen Defekt sofort zu beheben, sobald dieser von einem fehlgeschlagenen Test erkannt wurde.

Für das Debugging bietet sich ein Werkzeug wie *GNU gdb*[12] an. Es ermöglicht nicht nur das Debuggen auf dem selben System, sondern kann auch mittels einer Server-Client-Architektur zum Debuggen von Anwendungen auf anderen Systemen eingesetzt werden. In Verbindung mit einem Programm wie *OpenOCD*[13] kann gdb auch genutzt werden, um auf der Zielhardware selber zu debuggen. Bei OpenOCD handelt es sich um eine Software, die, neben anderen Funktionalitäten, auch das sogenannte *On-Chip Debugging* erlaubt. Abbildung 4.8 zeigt ein exemplarisches Setup für das On-Chip Debugging.

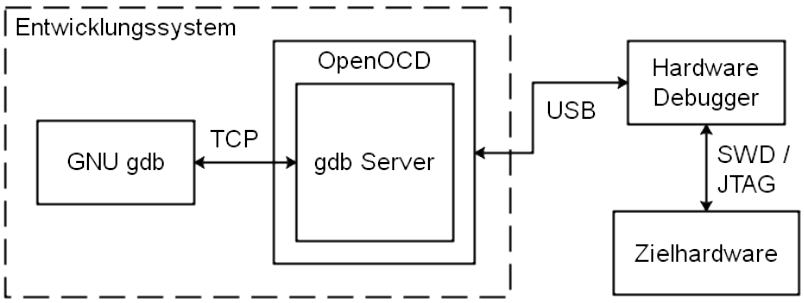

Abbildung 4.8: Blockschaltbild eines Setups für das On-Chip Debugging.

Die OpenOCD-Anwendung baut mit Hilfe eines Hardware Debuggers eine Verbindung zur Zielhardware auf und kann diese über die gewählte Debug-Schnittstelle kontrollieren. Gleichzeitig wird eine gdb Server-Instanz geöffnet, zu der sich GNU gdb via TCP verbinden kann. Das On-Chip Debugging kann nützlich sein, um unterschiedliche Verhaltensweisen zwischen dem Test-Executable auf dem Entwicklungssystem und der Zielhardware zu untersuchen.

Testabdeckung

Eine für die testgetriebene Entwicklung interessante Metrik ist die Testabdeckung. Sie gibt Aufschluss über den Anteil des Quellcodes, welcher durch die Ausführung aller Testfälle abgedeckt wird. Mit einem Werkzeug wie *gcov*[14] kann die Testabdeckung für den Quellcode ermittelt werden. Es liefert sowohl einen Prozentwert, wie viele der betrachteten Codezeilen durch die Testfälle erreicht wurden, als auch einen detaillierten Bericht in

[12] `https://www.gnu.org/software/gdb/`
[13] `http://openocd.org/`
[14] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

Klartext. Dieser Bericht enthält Informationen über einzelne Codezeilen und Anweisungen. Das Werkzeug kann dazu eingesetzt werden, die Testfälle zu optimieren. So können etwaige Redundanzen in den Testfällen ermittelt und anschließend eliminiert werden. Des Weiteren wird durch Zeilen oder Pfade, die durch die bisherigen Testfälle nicht abgedeckt werden, deutlich, welche Ergänzungen für die Test-Suite sinnvoll wären [Pat06, S. 118]. Unterschieden wird zwischen drei verschiedenen Kategorien, dessen Metriken alle auf dem Kontrollfluss des Programms basieren. Sie lassen sich wie folgt beschreiben [Wit18, S. 117 f.].

- **Anweisungsüberdeckung** - *„Bei der Anweisungsüberdeckung wird der prozentuelle Anteil der ausgeführten Anweisungen an der Gesamtzahl der Anweisungen gemessen.“*
- **Zweigüberdeckung** - *„Die Entscheidungs-/Zweigüberdeckung ist mächtiger als die Anweisungsüberdeckung, weil es hier nicht reicht, in jeder Anweisung mindestens einmal gewesen zu sein, sondern auch jeder Zweig mindestens einmal durchlaufen werden muss.“*
- **Bedingungsüberdeckung** - *„Die Bedingungsüberdeckung [...] betrachtet die Bedingungen (alle booleschen Wertekonstellationen) in ihren verschiedenen Komplexitätsgeraden.“*

Mit GNU gcov ist die Ermittlung von Anweisungsüberdeckung und Zweigüberdeckung möglich. Somit liefert es wichtige Einblicke in die Vollständigkeit der Test-Suite. Die Ausgabe in der Kommandozeile eines Aufrufs des Werkzeugs ist in Listing 4.30 zu sehen.

```
1 $ gcov src/BNO055Driver.c -o output/test_host/ -n -b
2
3 File 'src/BNO055Driver.c'
4 Lines executed:100.00% of 70
5 Branches executed:100.00% of 24
6 Taken at least once:87.50% of 24
7 Calls executed:100.00% of 16
```

Listing 4.30: Ausgabe des GNU gcov Werkzeugs für das `BNO055Driver`-Modul.

In diesem Fall wurde sowohl für die Anweisungs- als auch für die Zweigüberdeckung ein Wert von 100 % erreicht.

Build Server

Um die testgetriebene Entwicklung auf ein großes Team zu skalieren, ist es nützlich einen Build Server für das jeweilige Projekt aufzusetzen. Seine Aufgabe ist es nicht nur, die Executables zu erzeugen, sondern führt zudem auch die Testfälle aus. Steht aus Kostengründen nicht allen Entwicklerinnen ein Cross-Compiler auf dem Entwicklungssystem zur Verfügung, stellt der Build Server die einzige Möglichkeit dar, den Quellcode für das Ziel-System zu übersetzten. Automation Server wie *Jenkins*[15] erlauben komplexere Automationen von Build-Vorgängen und Test-Durchläufen. Darüber ermöglichen sie den Einsatz von *Continuous Integration*, ein wichtiger Bestandteil der Automation. Eine eingehende Auseinandersetzung mit dem Thema Continuous Integration folgt im Kapitel 4.6.2.

[15] `https://www.jenkins.io/`

4.6 Automatisierung

Die Komponententests, welche bei der testgetriebenen Entwicklung geschrieben werden, sind anders als Integrations- oder System-Tests, wesentlich leichter zu automatisieren. Der Einsatz von Doubles und Mocks eliminiert äußerliche Abhängigkeiten und ermöglicht somit die isolierte Betrachtung der zu testenden Code-Einheit. Die Testfälle können mithilfe diese Strategien ohne weiteres von dem Test-Framework wiederholt und ohne manuelle Eingriffe ausgeführt werden. Ein Komponententest, der nicht ohne manuellen Eingriff auskommt, muss dringend neu geschrieben werden. Aber nicht nur die Testfälle müssen automatisch ablaufen. Die testgetriebene Entwicklung bedarf bei jeder Iteration häufig ähnliche oder sogar identische Tätigkeiten. Es besteht die Gefahr, dass diese meist simplen Tätigkeiten zu einem Hindernis für den Arbeitsfluss werden. Da es für die testgetriebene Entwicklung aber essentiell ist, dass die Testfälle so oft es geht ausgeführt werden, darf der Arbeitsfluss nicht behindert werden. Daher ist es ratsam alle Tätigkeiten, die wiederholt ausgeführt werden müssen, zu automatisieren. Automatisierung spart nicht nur Zeit, sondern verhindert zusätzlich auch potentielle Fehler, welche sich durch stupide und repetitive Abläufe ergeben [CG09, S. 258 f.].

4.6.1 Automatisierung des Workflows

An dieser Stelle soll anhand eines Beispiels gezeigt werden, wie Automatisierung eingesetzt werden kann, um den Workflow in der testgetriebenen Entwicklung möglichst reibungslos zu gestalten. Dafür kommen die bereits im vorigen Kapitel beschriebenen Software-Werkzeuge zum Einsatz.
In jedem Iterationszyklus der testgetriebenen Entwicklung muss mindestens einmal das Test-Executable erzeugt und ausgeführt werden. Zunächst muss also der Build-Vorgang für das Executable, mittels eines Aufrufs des entsprechenden Targets des Makefiles, gestartet werden. Ist der Build erfolgreich, muss als nächstes das Executable ausgeführt werden. Auch wenn dieser Prozess nicht sehr aufwändig erscheint, so kann der Zeitaufwand sich mit der Zeit akkumulieren. Schlimmer ist allerdings der Ermüdungseffekt, der durch die notwendigen manuellen Eingriffe erforderlich wird. Dies verleitet dazu, in den Iterationsschritten zunehmend mehr und größere Modifikationen am Quellcode vorzunehmen. Das resultiert in einer Minderung der Effizienz der testgetriebenen Entwicklung. Mehr Änderungen pro Iteration bedeuten eine Verlängerung der Rückmeldeschleife.

Soll nun auf der Zielhardware oder auf dem Evaluationsboard getestet werden, sind weitere Schritte notwendig. Nach dem Build muss das Executable auf den Mikrocontroller übertragen werden. Anschließend muss ein Terminal-Programm für die serielle Kommunikation gestartet werden, um die Text-Ausgabe des Test-Frameworks auswerten zu können. Ein Shell-Script ist in diesem Fall ein guter Weg, um diese Arbeitsschritte auf einen Aufruf in der Kommandozeile zu reduzieren. Das in Listing 4.31 dargestellte Bash-Script übernimmt genau dies.

```
1  #!/bin/bash
2
3  ######################################################
4  #
5  # This script can be used to generate the test
6  # executables.
7  #
8  ######################################################
9  # check.sh: will generate the TestRunner source
10 #   file
11 ######################################################
12 # check.sh host: will generate the executable for
13 #   the host development system
14 ######################################################
15 # check.sh target: will generate the executable for
16 #   the target system and program the device
17 ######################################################
18
19 # size of ROM of target hardware
20 rom_size=64000
21
22 # generate the Unity test runner
23 scripts/generate_unity_test_runner.sh \
24     || { echo 'Generating the Unity TestRunner failed!' ; exit 1; }
25
26 if [ $# -eq 0 ];
27 then
28     exit 0
29 fi
30
31 # build and run host test executable
32 if [ $1 == "host" ];
33 then
34     make test_host >> /dev/null \
```

```
35          || { echo 'Building the Host Test executable failed!' ; exit
    1; }
36
37      ./output/test_host/derail_alarm_system
38
39
40  # build, program and run target test executable
41  elif [ $1 == "target" ];
42  then
43      make test_target >> /dev/null \
44          || { echo 'Building the Target Test executable failed!' ;
    exit 1; }
45
46      executable_size=$(scripts/get_executable_size.sh
    output/test/derail_alarm_system.elf)
47      if [[ "$executable_size" -ge "$rom_size" ]]
48      then
49          echo 'Executable is too big for target ROM!'
50          exit 1
51      else
52          ./scripts/program_target_test.sh \
53              || { echo 'Unable to program target!' ; exit 1; }
54
55          grabserial -d /dev/ttyACM0 -b 115200 --endtime=10
    --quitpat="(OK\n|FAIL\n)" -o output/test/testLog.txt
56      fi
57  fi
```

Listing 4.31: Bash-Script zur Automatisierung des testgetriebenen Iterationszyklus mit Dual-Targeting.

Eine der wichtigsten Aufgaben des Scripts ist es, in jeder Iteration den Test-Runner für das Unity-Framework zu erzeugen. Dies erfolgt über den Aufruf eines weiteren Scripts (`generate_unity_test_runner.sh`) in Zeile 7. Das Unity-Framework hat den Nachteil, dass jeder Testfall dem Framework bekannt gemacht werden muss. Dafür muss der Testfall in der Main-Funktion in den Test-Runner eingehängt werden. Erfolgt dies nicht, wird der Test schlicht nicht ausgeführt. Sehr wahrscheinlich würde dies nicht einmal auffallen, da die Text-Ausgabe aus strategischen Gründen minimal gehalten wird. Nur fehlgeschlagene Tests erzeugen tatsächlich auch eine Ausnahme. Wird der Test-Runner manuell editiert, ist die Gefahr groß, einen Testfall zu übersehen. Dieses Risiko wird durch die Automatisierung mittels eines Scripts eliminiert und verdeutlicht den enormen

Nutzen der Automatisierung. Das `generate_unity_test_runner`-Script scannt das Verzeichnis, welches die Quelldateien mit den Testgruppen enthält. Danach werden aus jeder Datei die Namen der Testfunktionen extrahiert. Basierend auf einer Vorlage wird dann die Quelldatei erzeugt, welche den Test-Runner und die Main-Funktion enthält. Neben der Reduzierung des Risikos von nicht angemeldeten Tests, ist auch die Zeitersparnis durch das Script enorm.

4.6.2 Continuous Integration

In einem Projekt, welches eingebettete Software enthält, ergibt sich das Problem der Integration von einzelnen Quellcode-Einheiten miteinander. Diese Quellcode-Einheiten werden simultan an mehreren Entwicklungssystemen geschrieben und müssen anschließend in eine gemeinsame Codebasis zusammengeführt werden. Diese Tätigkeit fällt in den Bereich der Konfigurationsverwaltung. Sie lässt sich wie folgt definieren [LL13, S. 558].

> *„Die Konfigurationsverwaltung ist diejenige Rolle oder Organisationseinheit, die die Software-Einheiten und Konfigurationen identifiziert, verwaltet, bei Bedarf bereitstellt und ihre Änderungen überwacht und dokumentiert.“*

Wird testgetrieben entwickelt, so muss sichergestellt werden, dass vor dem Integrieren von neuen Quellcode-Einheiten alle Testfälle erfolgreich ausgeführt werden. Dies bezieht sich nicht nur auf die zur Code-Einheit gehörigen Tests, sondern auf alle Testfälle im gesamten Projekt. Eine neue Code-Einheit sollte nicht in die gemeinsame Codebasis integriert werden, wenn sie dafür sorgt, dass ein Test fehlschlägt. Die Codebasis muss durchgängig in einem Zustand sein, in dem alle Testfälle erfolgreich durchlaufen. Dies ist essentiell, um auch im weiteren Projektverlauf testgetrieben entwickeln zu können. Der damit zusammenhängende Aufwand kann, je nach Größe des Teams und des Projekts, erheblich ausfallen. Wird die Konfigurationsverwaltung manuell ausgeführt, läuft man Gefahr, dass durch Nachlässigkeiten Tests nicht ausgeführt werden und die Codebasis in einen nicht funktionalen Zustand übergeht. An dieser Stelle setzt die Continuous Integration (CI) an. Ihr Ziel ist es, die Konfigurationsverwaltung zu automatisieren. Durch die Automatisierung kann die testgetriebene Entwicklung auch auf ein großes Team skaliert werden. Paul Duvall schreibt dazu das Folgende [DMG07, S. XX].

> *„[...] As the complexity of a project increases (even just adding one more person), there is a greater need to integrate and ensure that software components work together–early and often.“*

In seinem Vortrag *Multi-Stage-CI with Jenkins in an Embedded World* [Mar14] beschreibt Robert Martin, wie Continuous Integration bei BMW für die Automatisierung der Konfigurationsverwaltung eingesetzt wird. Das vorgestellte Projekt weist über 100 Millionen Codezeilen eingebetteter Software auf. Etwa 400 Entwicklerinnen und Entwicklern waren daran beteiligt. Mit CI war es möglich, die Integrationsdauer für neue Code-Einheiten von etwa fünf Wochen auf unter eine Stunde zu reduzieren. Ziel der Automatisierung war es, eine kurze Rückmeldedauer für erkannte Defekte zu priorisieren. Eine Besonderheit, die auch für diese Thesis eine Relevanz besitzt, ist, dass den Entwicklerinnen und Entwicklern keine Zielhardware zur Verfügung steht. Aufgrund der hohen Kosten für Prototypen im Automotive Sektor wurde sich darauf beschränkt, Tests im Zuge der Integration auf einer zentralen Zielhardware auszuführen. Die in dem Vortrag vorgestellten Erkenntnisse zeigen den Nutzen den Continuous Integration für die testgetriebene Entwicklung eingebetteter Software bedeuten kann. Im Folgenden wird auf das Konzept der CI näher eingegangen.

Abbildung 4.9 zeigt die grundlegenden Komponenten, aus denen sich ein CI-System zusammensetzt. Die Infrastruktur hat mehrere Entwicklungssysteme. Von diese werden Änderungen an der Codebasis in einen Versionsverwaltungssystem eingecheckt. Ein CI-Server erkennt Änderungen, die an die Versionsverwaltung übermittelt werden und löst ein durch ein Build-Script beschriebenen Build-Vorgang aus. Über einen Rückmelde-Mechanismus werden erkannte Defekte und Fehler den zuständigen Entwicklerinnen mitgeteilt.

Für die Versionsverwaltung wird ein Software-Werkzeug wie beispielsweise *Git*[16] eingesetzt. Git erlaubt eine über mehrere Systeme verteilte Verwaltung von Quellcode. Der Quellcode eines Projekts wird dabei in einem sogenannten *Repository* aufbewahrt. Diese Repositories können auf mehreren Systemen gleichzeitig existieren. Nach Bedarf werden sie miteinander synchronisiert und integriert. Die Versionsverwaltung ist ein zentraler Bestandteil der CI-Infrastruktur, da sie es ermöglicht, an mehreren Entwicklungssystemen gleichzeitig an der selben Codebasis zu arbeiten.

[16] `https://git-scm.com/`

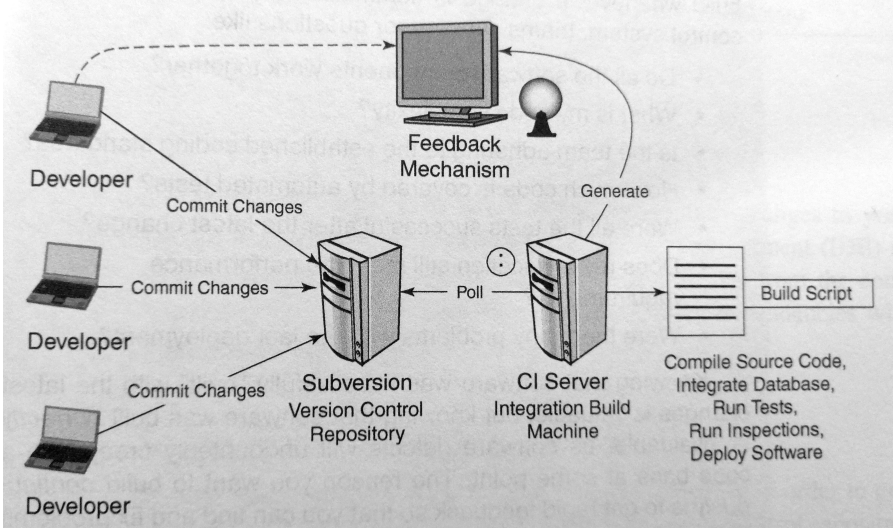

Abbildung 4.9: Komponenten eines simplen Continuous Integration Systems [DMG07, S. 5]

Der CI-Server ist dafür zuständig, automatisch neu eingecheckte Änderungen an der Codebasis zu detektieren. Dies kann entweder durch regelmäßiges Pollen des Git-Server erfolgen oder es wird durch Events getriggert, die von dem Versionsverwaltungsserver ausgelöst werden. Ein populäres Beispiel für einen CI-Server ist *Jenkins*[17]. Wird bei Continuous Integration von einem *Build* gesprochen, so wird sich dabei auf mehr als nur das Übersetzen des Quellcodes bezogen [DMG07, S. 4].

> *„A build may consist of the compilation, testing, inspection, and deployment– among other things. A build acts as the process for putting source code together and verifying that the software works as a cohesive unit.“*

Bevor eine Änderung am Quellcode also in die Codebasis integriert wird, muss sie durch den Build verifiziert werden können. Zunächst muss natürlich das Übersetzten erfolgreich verlaufen. Anschließend müssen alle Tests bestanden werden. Das können zum einen die durch die testgetriebene Entwicklung entstandenen Unit-Tests sein. Darüber hinaus werden auch Integrations-, System- und falls vorhanden, Acceptance-Tests ausgeführt. Weiterhin werden im Build statische Tests laufen gelassen und der Quellcode nach den geltenden Coding-Guidelines überprüft. Wird die eingecheckte Änderung durch den Build erfolgreich verifiziert, werden die notwendigen Executables durch den CI-Server bereitgestellt. Schlägt der Build fehl, so generiert der Server einen entsprechenden Bericht zu

[17] `https://www.jenkins.io/`

den erkannten Defekten und übermittelt diesen an die Verantwortlichen. Meistens erfolgt dies per Mail.

Eine weitere Parallele von Continuous Integration zur testgetriebenen Entwicklung ist der Grundsatz, dass bei einem fehlgeschlagenen Test/Build zunächst das gesamte Fortschreiten zur nächsten Iteration pausiert wird, bis das Problem behoben wurde. Damit ein CI-Ansatz tatsächlich gute Ergebnisse liefern kann, ist es unumgänglich, dass für das Projekt eine ausführliche Test-Suite existiert. Die testgetriebene Entwicklung schafft dafür eine gute Grundlage. Deutlich wird allerdings, der Übersetzungs-Vorgang für den Production-Executable, als auch für das Test-Executable, muss zwingend vollständig automatisierbar sein. Kann der Quellcode bereits auf dem Entwicklungssystem nicht ohne manuellen Eingriff übersetzt werden, ist der automatisierte Build auf dem CI-Server definitiv nicht möglich.

Continuous Integration (CI) bietet einige Vorteile, die sich gut mit den Zielen der testgetrieben Entwicklung decken. So wird das Risiko reduziert, dass Defekte in der Software erst spät erfasst werden. Die Codebasis wird stets in einem funktionierenden Zustand gehalten, was durch die Test-Suite sichergestellt wird. Das Vertrauen des Teams in die Software wächst, da verhindert wird, dass sich der Quellcode über lange Zeiträume in einem nicht verifizierten Zustand befinden kann [DMG07, S. 29 ff.]. Der Ansatz ist nahezu beliebig skalierbar. Die in Abbildung 4.9 dargestellte Infrastruktur kann nach Bedarf ausgebaut und durch weitere Komponenten ergänzt werden. Robert Martin beschreibt in seinem Vortrag [Mar14] beispielsweise, wie der Integration-Build in mehrere Stadien unterteilt werden kann und die Builds auf mehreren verteilten Systemen ablaufen. Natürlich erfordert der Betrieb einer CI-Infrastruktur auch einen erheblichen Aufwand. Spätestens bei größeren Projekten müssen hier gesonderte Personal-Ressourcen eingeplant werden.

Bei Continuous Delivery (CD) handelt es sich um ein Konzept, was auf der Grundidee von Continuous Integration aufsetzt und diese ausbaut. Änderungen an der Codebasis werden nicht nur automatisiert integriert, sondern werden darüber hinaus auch direkt *deployed*, sprich den Endkunden bereitgestellt. Für eingebettete Software hat dieses Konzept derzeit weniger Relevanz, da es für die Hersteller recht schwierig ist, die Firmware ihrer Produkte nach Markteinführung zu updaten. Allerdings bringt der Trend des Internet of Things zunehmend eingebettete Geräte hervor, die an ein Netz angebunden sind und

somit leichter Updates der Firmware erfahren könnten. Insofern dürfte auch Continuous Delivery in Zukunft für eingebettete Software immer wichtiger werden.

4.7 Testgetriebene Entwicklung in Agilen Vorgehensmodellen

Die testgetriebene Softwareentwicklung wird hauptsächlich bei Projekten eingesetzt, die einem agilen Vorgehensmodell folgen. In diesem Kapitel soll daher betrachtet werden, welche Rolle die testgetriebene Entwicklung in die agilen Praktiken einnimmt.

Das Fundament, auf dem die meisten agilen Praktiken basieren, wurde 2001 mit dem *Manifesto for Agile Software Development* [Bec+01] gelegt. Dieses besteht aus insgesamt zwölf grundlegenden Prinzipien, denen agile Modelle versuchen, gerecht zu werden. Eine Reihe an Praktiken, wie *Scrum*, *Kanban* und *Extreme Programming*(XP) lassen sich den agilen Methoden zuordnen. Insbesondere das Extreme Programming befasst sich auch mit den technischen Details eines agilen Projekts. Daher werden im Folgenden insbesondere die damit zusammenhängenden Praktiken eingehender betrachtet. Die Methoden des Extreme Programming wurden erstmals von Kent Beck in seinem Buch *Extreme Programming Explained* [Bec05] beschrieben.

Der entscheidende Unterschied zwischen den agilen und dem Wasserfall-Vorgehensmodellen ist, dass das Projekt in viele kleine Iterationen unterteilt wird. Jede Iteration (oft auch als *Sprint* bezeichnet) umfasst dabei eine Analyse-, eine Design- und eine Implementierungsphase [Mar20, S. 24]. Für jede Iteration werden eine Gruppe von Features für das zu entwickelnde Produkt ausgewählt, die umgesetzt werden sollen. Diese werden in sogenannten *Stories* beschrieben. Am Ende einer Iteration werden die Resultate bewertet und somit Daten zu dem Projekt generiert. Diese Daten geben Aufschluss über den derzeitigen Stand des Projekts und können dazu eingesetzt werden, Prognosen zu dem weiteren Projektverlauf zu erstellen. Das Projektmanagement erfolgt über den Scope, also den Umfang des Projekts [Mar20, S. 31]. Dafür werden allen Stories eine Priorität zugewiesen. Die Stories mit den höchsten Prioritäten werden als erstes umgesetzt. Über den Verlauf des Projekts können die Gewichtungen der Stories immer wieder angepasst, Stories gestrichen oder neue hinzugefügt werden. Auf diese Weise kann wesentlich flexibler auf Änderungen oder neue Erkenntnisse reagiert werden als mit dem Wasserfall-Modell. Einer der wichtigsten Aspekte an dem agilen Vorgehen ist die Priorisierung von schnellem Feedback. Dies zeichnet sich auch sehr deutlich im Bereich der Software-Tests ab. Neben der testgetriebenen Entwicklung gibt es noch weitere Praktiken, die ein schnelles

Feedback unterstützen. Abbildung 4.10 zeigt den *Circle of Life* des Extreme Programmings.

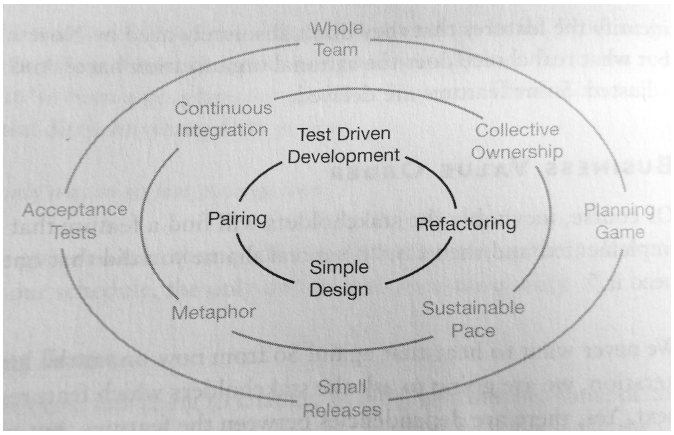

Abbildung 4.10: Der Circle of Life in Extreme Programming. [Mar20, S. 32]

Er besteht insgesamt aus drei Schichten, die jeweils Praktiken enthalten, derer sich das Extreme Programming bedient. Auf der äußeren Schicht sind die Praktiken enthalten, die auf geschäftliche Aspekte abzielen. Die mittlere Schicht enthält Praktiken, welche die Arbeit im Team betreffen. Die innerste Schicht enthält schließlich die technische Praktiken [Mar20, S. 32 ff.]. Die testgetriebene Entwicklung ist ein Teil der innersten Schicht und somit ein integraler Part des Extreme Programmings. Zusammen mit den weiteren Praktiken, Refactoring, Simple Design und Pairing, wird ein Fokus auf kontinuierliche Verbesserung und Wissensaustausch gelegt. Auf der mittleren Schicht ist, die bereits im Kapitel 4.6.2 beschriebene, Continuous Integration zu finden. Auf der äußeren Schicht sind die sogenannten *Acceptance Tests* beheimatet. Durch die Acceptance Tests wird am Ende einer Iteration verifiziert, ob die Anforderungen der bearbeiteten Story erfüllt wurden und die Story somit erfolgreich umgesetzt wurde. Das Zusammenspiel von Acceptance Tests, Continuous Integration und testgetriebener Entwicklung sorgt dafür, dass jederzeit Klarheit über den aktuellen Stand des Projekts existiert. Die aus der testgetriebenen Entwicklung resultierenden Komponententests stellen dies auf der kleinsten, funktionalen Ebene sicher. Die Acceptance Tests validieren die Software aus Sicht der Stakeholder und Kunden. Mittels Continuous Integration wird dafür gesorgt, dass die Software-Tests jederzeit automatisch ausgeführt werden können.
Die agilen Vorgehensmodelle redefinieren die Rolle der Software-Testerinnnen und -Tester von Grund auf. Anstatt eine Test-Phase als abschließende Phase des Projekts zu haben, wird nun kontinuierlich in jeder Iteration getestet. Es kann so nicht mehr passieren,

dass der Umfang der Test-Phase, aufgrund von Zeit- oder Budgetgründen, eingeschränkt wird. Die Tests liefern wichtige Erkenntnisse zum Zustand des Projekts über den gesamten Projektverlauf. In allen Iterationen werden alle Testfälle durchgeführt. Daraus resultiert, dass die Tests unbedingt automatisiert werden müssen, um in den Builds des Continuous Integration-Systems ausgeführt werden zu können. Ob eine Iteration erfolgreich abgeschlossen wurde, wird durch die Acceptance Tests ermittelt. Zu Beginn der Iteration werden sie in Zusammenarbeit von Kunden bzw. Stakeholdern und Qualitätsmanagement formuliert. Die Softwareentwicklerinnen und -Entwickler sind anschließend dafür zuständig, die Tests für den CI-Build zu automatisieren [Mar20, S. 90]. Die Rolle des Qualitätsmanagements ändert sich komplett. Kent Beck beschreibt dies wie folgt [Bec05, S. 74].

> *„Testers on an XP team help customers choose and write automated system-level tests in advance of implementation and coach programmers on testing techniques. On XP teams much of the responsibility for catching trivial mistakes is accepted by the programmers. Test-first programming results in a suite of tests that help keep the project stable.“*

Die Entwicklerinnen und -Entwickler nehmen demnach ebenfalls mehr Verantwortung für das Testen auf sich als sonst üblich. Sie arbeiten mit dem Qualitätsmanagement gemeinsam an den Acceptance Tests und praktizieren testgetriebene Entwicklung.

Der größere Austausch zwischen Qualitätsmanagement und das Selbstverständnis, dass die Entwicklung Verantwortung für die Software-Test trägt, unterstützen den Einsatz der testgetriebenen Entwicklung. Die Qualität der Software rückt somit in den Fokus. Des Weiteren profitiert die testgetriebene Entwicklung von den anderen Mechanismen im Vorgehensmodell, die darauf abzielen Rückmeldezyklen zu verkürzen.
Sicherlich ist der Einsatz der testgetriebenen Entwicklung nicht auf die agilen Vorgehensmodelle begrenzt, jedoch ergeben sich hier nützliche Synergieeffekte durch die iterative Arbeitsweise und das neue Verständnis des Qualitätsmanagements.

5 Fazit

Ziel dieser Thesis war es, zu untersuchen, ob die testgetriebene Entwicklung von eingebetteter Software eingesetzt werden kann, um auch bei nicht bereitstehender Zielhardware den Quellcode zu testen. Es soll dadurch verhindert werden, dass die Software erstmals mit dem verfügbar werden der Zielhardware ausgeführt und geprüft wird. Dafür wurden unter anderem verschiedene Test-Frameworks auf ihre Eignung analysiert und Strategien vorgestellt, die helfen, die technischen Herausforderungen der fehlenden Zielhardware zu überkommen. Anschließend wurde betrachtet, welche Auswirkung das testgetriebene Vorgehen hat. Es wurde beschrieben, wie Continuous Integration eingesetzt werden kann, um den Ansatz für größere Teams und Projekte zu skalieren. Abschließend wurde die testgetriebene Entwicklung im Kontext von agilen Vorgehensmodellen betrachtet. Im Folgenden soll nun ein Fazit gezogen werden, indem Vorteile und Nachteile gegenübergestellt werden und auf Kritik an der testgetriebenen Entwicklung eingegangen wird.

Die wohl wichtigste Strategie, um die testgetriebene Entwicklung unabhängig von der Zielhardware zu ermöglichen, ist das Dual-Targeting. Sie sieht vor, die Testfälle bevorzugt auf dem Entwicklungssystem auszuführen. Erreicht wird dies, indem eine weitere Build-Pipeline aufgebaut wird, welche den in C geschriebenen Quellcode, insbesondere das Test-Projekt, für das Entwicklungssystem übersetzt. Hierzu können theoretisch alle nativen C-Compiler eingesetzt werden. Die Auswahl ist groß und bei Verwendung eines quelloffenen Compilers, wie beispielsweise gcc, entstehen nicht einmal zusätzliche Lizenzkosten. Der Nachteil ist, dass der Konfigurationsaufwand steigt. Statt einer Build-Pipeline müssen nun zwei aufgesetzt und instand gehalten werden. Ein Ausführen der Testfälle auf der Zielhardware wird von der entsprechenden Literatur nicht ausdrücklich priorisiert. Dies birgt ein Risiko, denn Unterschiede zwischen den beiden Systemen, wie etwa die Endianness, die native Verarbeitungsbreite oder Unterschiede in den Standard C-Bibliotheken, können zu unterschiedlichen Ergebnissen der Testfälle führen. Um sicherzustellen, dass diese schwer vorherzusehenden Probleme nicht unbemerkt bleiben,

ist es unabdingbar, die Testfälle auch auf der Zielhardware ausführen zu können. Dies muss zwar nicht so hochfrequent passieren, wie die Tests auf der Entwicklungsplattform, aber sollte dennoch in regelmäßigen Abständen durchgeführt werden. Glücklicherweise ist es dafür nicht zwingend notwendig, die finale Zielhardware vorliegen zu haben. Es reicht vollkommen aus, wenn ein Evaluationsboard oder ein Emulator des entsprechenden Prozessors zur Verfügung steht. Diese sind meist wesentlich früher im Projektverlauf zugänglich als ein Prototyp der Hardware.
Zum Schreiben der Komponententests ist es ratsam, ein Test-Framework einzusetzen. Auch hier ist die Auswahl zunächst groß. Grundsätzlich können Frameworks genutzt werden, die für C oder C++ entwickelt wurden. Die Untersuchung von drei Frameworks zeigte allerdings Probleme auf. Obwohl die Frameworks alle damit beworben wurden, auch für eingebettete Software ausgelegt zu sein, kam es zu erheblichen Schwierigkeiten, sie für die Zielarchitektur zu übersetzten. So zeigte das CMocka-Framework bei Ausführung auf dem Zielsystem ein anderes Verhalten auf als auf dem Entwicklungssystem. Es wäre ein Eingriff in den Quellcode des Frameworks nötig gewesen, um dies zu korrigieren. Das CppUTest-Framework erwies sich als sehr komplex in seiner Konfiguration, sodass eine Übersetzung für die Zielarchitektur einen enormen Aufwand bedeutet hätte, der im Projekt nicht zu rechtfertigen wäre. Darüber hinaus ist es in C++ geschrieben, was bedeutet, dass die für die Build-Pipeline ein C++-Cross-Compiler vorhanden sein muss. Einzig das Unity-Framework zeigte konstante Resultate zwischen dem Entwicklungs- und dem Zielsystem. Es hat den geringsten Feature-Umfang der betrachteten Frameworks, bietet aber dennoch die wichtigsten Funktionalitäten, um effektiv Komponententests schreiben zu können. Ein Vorteil des geringeren Umfangs ist die kleine Größe des Executables. Das Unity-Framework sollte somit auch für Architekturen mit nur kleinen Speicherkapazitäten geeignet sein. Der Konfigurationsaufwand, um Unity auf der Zielplattform ausführen zu können, ist absolut vertretbar. So reicht es aus eine Funktion zu implentieren, welche die Textausgabe des Frameworks auf eine serielle Verbindung des Mikrokontrollers umleitet.
Generell lässt sich bei den in C geschriebenen Test-Frameworks feststellen, dass sie weniger komfortabel zu nutzen sind als vergleichbare Frameworks für andere Hochsprachen. So erfordern sowohl Unity und CMocka, dass jeder hinzugefügte Testfall gesondert in dem Test-Runner bekannt gemacht wird. Zwar kommen beide Frameworks mit einem Script, welches diesen Schritt automatisiert, dies ist jedoch für die Zielarchitektur nicht ohne weiteres einsetzbar. Der Grund ist, dass die Peripherie des Mikroprozessors initialisiert werden muss, bevor der Test-Runner aufgerufen wird. Es bleibt also die Wahl das mitgelieferte Script zu editieren oder, wie in dieser Arbeit geschehen, ein neues Script

zu schreiben. Ohne dieses Script würde der manuelle Arbeitsschritt den Zyklus der testgetriebenen Entwicklung zu sehr verlangsamen. Ein weiteres Feature, welches den Frameworks fehlt, ist das der parametrisierbaren Testfälle. Dies führt dazu, dass viele der Testfälle Redundanzen enthalten. Eine Integration der Komponententests in die entsprechenden IDEs, wie man sie beispielsweise von *VisualStudio* kennt, ist ebenfalls nicht vorhanden. Die Resultate der Frameworks werden ausschließlich über die Kommandozeile oder, bei Ausführung auf der Zielhardware, in dem Serial Terminal ausgegeben.

Beim Schreiben der Tests kann die enge Kopplung der Software an die Hardware zum Problem werden. Dem kann mit verschiedenen Strategien entgegengewirkt werden. Als wichtiges Werkzeug stellen sich hier die Doubles und Mock Objekte heraus. Mit ihnen können direkte Abhängigkeiten zur Hardware oder zu hardwarenahen Software-Einheiten umgangen werden. Sie ersetzen den entsprechenden Quellcode durch vereinfachte Versionen. Mocks erlauben es, komplexere Interaktionen, zum Beispiel mit Treibern für serielle Schnittstellen, zu verifizieren. Obwohl die Doubles und Mocks einen Mehraufwand beim Schreiben der Testfälle bedeuten, sind sie sehr wertvoll, wenn es darum geht, die Tests auf der Entwicklungsplattform auszuführen. Darüber hinaus ermöglichen sie es, den zu testenden Code tatsächlich von allen Abhängigkeiten zu isolieren. Ein Merkmal, welches ein gut implementierter Testfall unbedingt aufweisen sollte.
Um mit den Doubles und Mocks effizient arbeiten zu können, muss der Code allerdings stärker abstrahiert werden. Die Software-Architektur sollte unbedingt eine Abstraktionsschicht der Hardware und, falls vorhanden, eine für das Real-Time Operating System vorsehen. Diese Abstraktionsschichten stellen wichtige Schnittstellen dar, an denen mit Mocks und Doubles angesetzt werden kann. Sind sie vorhanden, ist es ohne weiteres möglich, die Anwendungsschicht der Software ausschließlich auf dem Entwicklungssystem, ohne Zielhardware, Evaluationsboard oder Emulator zu entwickeln. Der erhöhte Abstraktionsgrad führt leider auch zu einem größeren Speicherbedarf des Executables und kann sich gegebenenfalls negativ auf die Performance auswirken. Aber er hat auch Vorteile. Die Software-Einheiten gewinnen an Plattform-Unabhängigkeit. Sollte sich im Projekt-Verlauf eine Änderung an der zugrundeliegenden Hardware ergeben, kann durch Modifikation der Abstraktionsschichten der Quellcode leichter portiert werden. Eine Anpassung der Anwendungs-Logik wird bei einer sauberen Trennung nicht nötig.
Eine testgetriebene Entwicklung von Hardware-Treibern ist ebenfalls möglich. Diese interagieren meist direkt mit Kontroll- und Steuerregistern des Mikrocontrollers. Auf diese Weise können Peripherien konfiguriert und gesteuert werden. Während bei Mikropro-

zessoren die physikalischen Speicherbereiche direkt adressiert und modifiziert werden können, ist dies auf dem Entwicklungssystem nicht ohne weiteres möglich. Das Betriebssystem ordnet den Prozessen virtuelle Adressen zu. Um dieses Problem für das Dual-Targeting zu umgehen, müssen die Hardware-Treiber so implementiert werden, dass Zeiger auf Variablen übergeben werden können. Auf diese Weise lassen sich auch solche Interaktionen mit Speicherelementen gut testen.

Es lässt sich festhalten, dass durch den Einsatz von Doubles und Mocks, in Kombination mit dem Dual-Targeting, die Software sehr gut auf dem Entwicklungssystem implementiert werden kann. Insbesondere auf der Anwendungsebene, die mit Sicherheit den größten Wert für Kunden und Stakeholder inne hat, ergeben sich dadurch viele Vorteile. Es kann wesentlich früher ausgiebig getestet werden und somit die Funktionalität des Codes eingeschätzt und verifiziert werden. Dies mindert das Risiko, dass sich, sobald die Hardware verfügbar wird, eine Reihe unvorhersehbarer Schwierigkeiten auftun. Sowohl die Kunden und Stakeholder als auch das Entwicklungsteam erlangen so mehr Vertrauen in den Quellcode.
Ein weiterer positiver Effekt ergibt sich beim erstmaligen Kombinieren des Quellcodes und der Zielhardware. So werden mögliche Fehlerursachen eingegrenzt. Schließlich können die Gründe für Fehler nicht nur im Code, sondern auch an der Zielhardware liegen. Durch die vorhandene Test-Suite lassen sich in einem solchen Fall einfacher Rückschlüsse auf potentielle Ursachen ziehen.
Natürlich profitiert der Code ebenfalls von den allgemein bekannten Vorteilen der testgetriebenen Entwicklung. Der fest vorgesehene Iterationsschritt des Refactorings steigert die allgemeine Code-Qualität. Diese wird somit über den Verlauf des Projekts auf einem höheren Niveau gehalten, was wiederum dafür sorgt, dass der Code später leichter modifiziert werden kann. Auch die Wiederverwendbarkeit steigt. Es entsteht eine einfach zu automatisierende Test-Suite.

Das Dual-Targeting ließe sich grundsätzlich natürlich auch mit anderen Arbeitsweisen nutzen, um die Software zunächst rein auf dem Entwicklungssystem zu entwickeln. So könnte beispielsweise auch der Test-First Ansatz gewählt werden, der ebenfalls das Schreiben der Test vor die Implementierung setzt. Die Kombination des Dual-Targetings mit der testgetriebenen Entwicklung bietet jedoch den Vorteil, dass sehr leicht eine hohe Codeüberdeckung der Testfälle erzielt werden kann. Eine Studie von Boby George und Laurie Williams ergab, dass die testgetriebene Entwicklung im Mittel zu 92 % Anweisungsüber-

deckung und 97 % Zweigüberdeckung führte [GW03]. Dies müsste bei dem Test-First Vorgehen speziell forciert werden. Ein weiterer Vorteil der testgetriebenen Entwicklung ist im Vergleich die iterative Arbeitsweise. Sie stellt sicher, dass die zu lösenden Probleme in kleine Teile runter gebrochen werden, was hilft, die Rückmeldedauern zu verkürzen.

Mit wachsendem Umfang und Komplexität steigt die Notwendigkeit, die Konfigurationsverwaltung zu automatisieren. Für die testgetriebene Arbeitsweise ist es absolut essentiell, die Rückmeldedauer für erkannte Probleme kurz zu halten. Müssen manuell Versionen des Quellcodes miteinander integriert werden oder gar Build-Vorgänge und Testdurchläufe manuell angestoßen werden, wird die Effektivität des Ansatzes deutlich eingeschränkt. Daher sollte die Konfigurationsverwaltung unbedingt um ein Continuous Integration-System ergänzt werden. Nur so kann sichergestellt werden, dass stets alle Tests ausgeführt werden und Änderungen, die zu Defekten führen, schnell erkannt und gemeldet werden. Der Betrieb eines Continuous Integration-Systems bedeutet unweigerlich einen höheren Personalbedarf.
Generell ergibt sich ein Mehraufwand durch die testgetriebene Entwicklung. Eine Fallstudie, die bei Microsoft und IBM durchgeführt wurde, ergab eine Steigerung der Entwicklungsdauer zwischen 15 % und 35 %. Jedoch wird argumentiert, dass die geringeren Instandhaltungskosten, die aus der Steigerung der Codequalität resultieren, die längere Entwicklungsdauer wieder ausgleichen [Nag+03]. Die Adaption des testgetriebenen Ansatzes stellt natürlich einen großen Eingriff in die Arbeitsweise der Programmiererinnen und Programmierer dar, was potentiell die Moral des Teams negativ beeinflussen könnte. Dies wird auch durch das Paper *An Initial Investigation of Test Driven Development in Industry* gestützt. Über die Hälfte der Probanden gab an, zunächst Schwierigkeiten mit der testgetriebenen Arbeitsweise gehabt zu haben [GW03].
Ein anderer Kritikpunkt an der testgetriebenen Entwicklung kommt aus dem Bereich des Software Entwurfs. John Ousterhout schreibt in seinem Buch *A Philosophy of Software Design* das Folgende [Ous18, S. 155].

> *„The problem with test-driven development is that if focuses attention on getting specific features working, rather than finding the best design.“*

Auch David Thomas und Andrew Hunt sehen dieses Problem. Sie kritisieren den *Bottom-Up* Ansatz der testgetriebenen Entwicklung. Dennoch glauben sie an den Nutzen. Sie schlagen vor, den Fokus auf die Ende-zu-Ende Implementierung von Funktionalitäten zu legen, um sich nicht in Details zu verlieren [TH20, S. 217 f.]. Diese Forderung lässt sich

gut durch die Kombination von testgetriebener Entwicklung mit einem agilen Vorgehen erfüllen. Die Umsetzung eines kompletten Features in einer der Iterationen hilft, das übergreifende Ziel der Software-Einheit im Auge zu behalten. Ergänzend sollten weitere Praktiken, wie Code-Reviews und Pair-Programming, eingesetzt werden, um sicherzustellen, dass das Software Design nicht vernachlässigt wird.

Abschließend lässt sich festhalten, dass die testgetriebene Entwicklung einen großen Nutzen haben kann, wenn es darum geht eingebettete Software zu schreiben ohne vorhandene Zielhardware. Sie kann potentielle Risiken begrenzen, indem sie ein Fokus auf die Code-Qualität und umfassendes Testen legt. Aber sie bringt auch einen Mehraufwand mit sich und hängt stark von der Akzeptanz des Teams ab. Das Schreiben der Komponententests in C ist definitiv aufwendiger, als in anderen Hochsprachen. Um anfängliche Hindernisse bei der Adaption der Arbeitsweise zu schmälern, wäre es hilfreich, wenn die Test-Frameworks speziell auf die Anwendung im eingebetteten Bereich ausgelegt werden würden. Auch eine bessere Integration der Komponententests und eventuell sogar des Dual-Targetings in die IDEs wäre sicherlich förderlich.

Literatur

[Ami19] Kamran Amini. *Extreme C*. 1. Aufl. Birmingham: Packt, 2019.

[Bar07] Micheal Barr. *Programming Embedded Systems*. 1. Aufl. Sebastopol: O'Reilly Media, 2007.

[BCK12] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. 3. Aufl. Boston: Pearson Education, 2012.

[Bec+01] Kent Beck u. a. *Manifesto for Agile Software Development*. Abgerufen am 03.11.2020. 2001. URL: `https://agilemanifesto.org/`.

[Bec03] Kent Beck. *Test-Driven Developement by Example*. 1. Aufl. Boston: Pearson Education, 2003.

[Bec05] Kent Beck. *Extreme Programming Explained*. 2. Aufl. Upper Saddle River: Pearson Education, 2005.

[CG09] Lisa Crispin und Janet Gregory. *Agile Testing*. 1. Aufl. Upper Saddle River: Pearson Education, 2009.

[DMG07] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous Integration*. 1. Aufl. Boston: Pearson Education, 2007.

[Fow19] Martin Fowler. *Refactoring*. 2. Aufl. Boston: Pearson Education, 2019.

[Gam+95] Erich Gamma u. a. *Design Patterns*. 1. Aufl. Indianapolis: Pearson Education, 1995.

[Gan08] Jack Ganssle. *The Art of Designing Embedded Systems*. 2. Aufl. Burlington: Elsevier, 2008.

[Gre11] James W. Grenning. *Test-Driven Developement for Embedded C*. 1. Aufl. Dallas: The Pragmatic Bookshelf, 2011.

[Grü17] Stephan Grünfelder. *Software-Test für Embedded Systems*. 2. Aufl. Heidelberg: dpunkt.verlag, 2017.

[GW03] Boby George und Laurie Williams. "An Initial Investigation of Test Driven Development in Industry". In: *Proceedings of the 2003 ACM symposium on Applied computing* (2003).

[KR88] Brian W. Kernighan und Dennis M. Ritchie. *The C Programming Language*. 2. Aufl. Upper Saddle River: Prentice-Hall, 1988.

[Lab17] Bill Laboon. *A Friendly Introduction to Software Testing*. 1. Aufl. Leipzig: Amazon Distribution, 2017.

[Lac18] Daniele Lacamera. *Embedded System Architecture*. 1. Aufl. Birmingham: Packt Publishing, 2018.

[LL13] Jochen Ludewig und Horst Lichter. *Software Engineering*. 3. Aufl. Heidelberg: dpunkt.verlag, 2013.

[Mar09] Robert C. Martin. *Clean Code*. 1. Aufl. Upper Saddle River: Prentice Hall, 2009.

[Mar14] Robert Martin. *Multi-Stage-CI with Jenkins in an Embedded World*. Abgerufen am 12.12.2020. 2014. URL: https://www.youtube.com/watch?v=AB5RTabEtEI.

[Mar20] Robert C. Martin. *Clean Agile*. 1. Aufl. Boston: Pearson Education, 2020.

[Mec05] Robert Mecklenburg. *Managing Projekts with GNU Make*. 3. Aufl. Sebastopol: O'Reilly Media, 2005.

[Mes07] Gerard Meszaros. *xUnit Test Patterns*. 1. Aufl. Upper Saddle River: Pearson Education, 2007.

[MFC00] Tim Mackinnon, Steve Freeman und Philip Craig. *Endo-Testing: Unit Testing with Mock Objects*. Abgerufen am 28.11.2020. 2000. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.3214.

[MW03] Michael E. Maximilien und Laurie Williams. "Assessing Test-Driven Development at IBM". In: *Proceedings of the 25th International Conference on Software Engineering* (2003).

[Nag+03] Nachiappan Nagappan u. a. "Realizing quality improvement through test driven development: results and experiences of four industrial teams". In: *Empirical Software Engineering* 13 (2003).

[Noe13] Tammy Noergaard. *Embedded Systems Architecture*. 2. Aufl. Kidlington: Elsevier, 2013.

[Ous18] John Ousterhout. *A Philosophy of Software Design*. 1. Aufl. Palo Alto: Yaknyam Press, 2018.

[Pat06] Ron Patton. *Software Testing*. 2. Aufl. Indianapolis: Sams Publishing, 2006.

[PD11] Bruce Powel Douglass. *Design Patterns for Embedded Systems in C*. 1. Aufl. Burlington: Elsevier, 2011.

[Rit93] Dennis Ritchie. *The Development of the C Language*. Abgerufen am 22.10.2020. 1993. URL: http://csapp.cs.cmu.edu/3e/docs/chistory.html.

[She19] Jianjun Shen. *Software Testing*. 1. Aufl. Leipzig: Dahuo Books, 2019.

[Som18] Ian Sommerville. *Software Engineering*. 10. Aufl. Hallbergmoos: Pearson, 2018.

[Sta+19] Richard M. Stallman u. a. *Using the GNU Compiler Collection*. Abgerufen am 28.11.2020. 2019. URL: https://gcc.gnu.org/onlinedocs/.

[Tar17] Alexander Tarlinder. *Developer Testing*. 1. Aufl. Boston: Pearson Education, 2017.

[TH20] David Thomas und Andrew Hunt. *The Pragmatic Programmer*. 2. Aufl. Boston: Pearson Education, 2020.

[Vig10] Uwe Vigenschow. *Testen von Software und Embedded Systems*. 2. Aufl. Heidelberg: dpunkt.verlag, 2010.

[Whi12] Elecia White. *Making Embedded Systems*. 1. Aufl. Sebastopol: O'Reilly Media, 2012.

[Wit18] Frank Witte. *Metriken für das Testreporting*. 1. Aufl. Wiesbaden: Springer Vieweg, 2018.

A Anhang

A.1 Beschreibung - Derailing Alarm System

Bei dem Derailing Alarm System handelt es sich um ein, an einen Kommunikationsbus angeschlossenes, Warn-System für Züge. Es misst mittels eines Sensors Beschleunigungskräfte und Winkelgeschwindigkeiten. Sobald die Messwerte einen bestimmten Schwellwert überschreiten, werden Warnungsmeldungen über den Kommunikationsbus geschickt. Dabei ist es vorgesehen, dass an den Bus mehrere solche Systeme angeschlossen werden können. Auf diese Weise kann in jedem Wagon ein Alarm System installiert werden und den entsprechenden Wagon überwachen. Abbildung A.1 zeigt das Blockschaltbild des Systems.

Der Kommunikationsbus basiert auf dem TIA-485 (RS-485) Standard. Dabei handelt es sich um einen asynchronen seriellen Kommunikationsstandard. Die physikalische Datenübertragung findet symmetrisch statt und ist daher resistent gegen elektromagnetische Störungen. Als zentrale Recheneinheit dient ein STM32F303RE Mikrocontroller. Dieser verfügt über einen ARM Cortex-M4 Prozessor und hat somit genug Leistung, um ein RTOS, wie in diesem Fall FreeRTOS, auszuführen. Die Schnittstelle zu dem TIA-485 Bus wird über eine UART-Schnittstelle des Mikrocontrollers realisiert. Ein MAX33072E Chip konvertiert dabei zwischen den physikalischen Übertragungsstrandards. Zur Erfassung der Messdaten wird ein BNO055 Smart-Sensor von Bosch eingesetzt. Hierbei handelt es sich um einen drei Achsen Sensor für Beschleunigungs-, Winkelgeschwindigkeits- und magnetische Flussdichten-Messungen. Der Sensor ist über I2C an den Mikrocontroller angebunden. Status LEDs dienen der visuellen Darstellung des System-Zustands und Schwellwertüberschreitungen. Über einen DIP-Switch kann die Bus-Adresse des Systems konfiguriert werden.

Der Quellcode zu diesem Projekt ist auf der beigefügten CD enthalten.

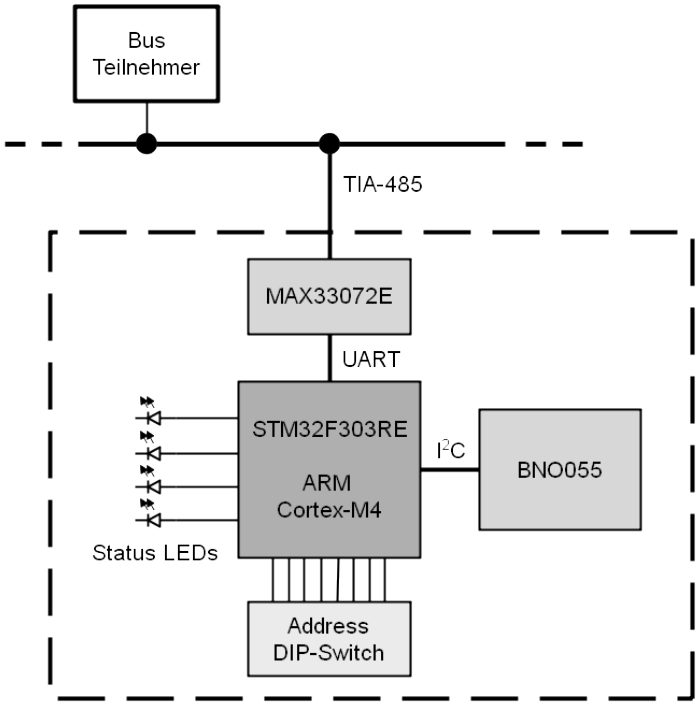

Abbildung A.1: Blockschaltbild des Derailing Alarm Systems.

A.1.1 Bus Protokoll

Das Bus Protokoll setzt auf dem TIA-485 Standard als Anwendungsschicht auf. Es ist ein Binärprotokoll, welches es erlaubt, Einträge einer auf dem System lokalen Daten Tabelle zu lesen und zu schreiben (siehe A.1.2). Es umfasst Lese- und Schreibanfragen, sowie Event-Nachrichten. Anfragen werden entweder mit einer Antwort-Nachricht oder einer Fehler-Nachricht quittiert. Das Protokoll kann 255 Bus Teilnehmer individuell adressieren. Im Folgenden werden die Datenfelder beschrieben die alle Nachrichten-Frames gemeinsam haben.

Operation Code (`OP_CODE`)

Der Operation Code (Tabelle A.1) gibt an, um welchen Frame-Typ es sich bei einer Nachricht handelt. Er wird in einem vier Bit breiten Feld gehalten. Es gibt Request-, Response-, Event- und Error-Frames.

OP_CODE	Operation
0x1	Request
0x2	Response
0x4	Event Message
0x8	Error Response

Tabelle A.1: Die Operation Codes des Bus Protokolls.

R/W	Beschreibung
0x1	Read
0x2	Write
0x0	Event

Tabelle A.2: Die Read/Write Codes des Bus Protokolls.

Read/Write (`R/W`)

Das Read/Write-Element (Tabelle A.2) gibt an, ob es sich um einen Lese- oder Schreibzugriff handelt. Es wird ebenfalls in einem vier Bit breitem Feld gehalten und bildet gemeinsam mit dem Operation Code das erste Byte jedes Nachrichten-Frames.

Destination Address (`DEST_ADDR`)

Die Destination Address gibt die Ziel Adresse einer Nachricht an. Jeder Bus Teilnehmer verfügt dabei über eine einzigartige 8-Bit Adresse. Die Destination Address ist jeweils das zweite Byte eines jeden Frames.

Source Address (`SRC_ADDR`)

Die Source Address gibt die Ziel Adresse einer Nachricht an. Bus-Teilnehmer reagieren nur auf Nachrichten, die an sie adressiert sind. Ist die Adresse auf `0xFF` gesetzt, so handelt es sich um eine Broadcast-Nachricht. Diese sind an alle Bus-Teilnehmer gewandt. Broadcast-Nachrichten werden nicht mit einer Antwort quittiert. Die Source Address ist jeweils das dritte Byte eines jeden Frames.

Entry ID (`ENTRY_ID`)

Mit der Entry ID wird angegeben, auf welchen Eintrag der Daten Tabelle zugegriffen werden soll. Insgesamt können 256 verschiedene Einträge adressiert werden. Die Entry ID ist jeweils das vierte Byte eines jeden Frames.

Cyclic Redundancy Check (`CRC`)

Das CRC-Element enthält jeweils die CRC-Checksumme aller vorherigen Elemente des Frames. Bei Empfang einer Nachricht wird die Checksumme vom Empfänger erneut berechnet und mit der übertragenen Checksumme abgeglichen. Nur wenn der Abgleich erfolgreich ist, wird ein empfangener Frame als valide angesehen. Das CRC-Element bildet die jeweils zwei letzten Bytes eines jeden Frames.

Read Request

Ein Read Request fordert einen Lesezugriff auf einen bestimmten Tabellen Eintrag eines anderen Bus-Teilnehmers an. Ist der Zugriff valide, wird mit einer Read Response geantwortet, die in ihrem `VALUE`-Feld den aktuellen Wert des Tabellen Eintrags zurückliefert. Die Read-Frames sind in Abbildung A.2 dargestellt. Ist der Zugriff nicht valide, wird mit einer Error Response (siehe A.1.1) geantwortet.

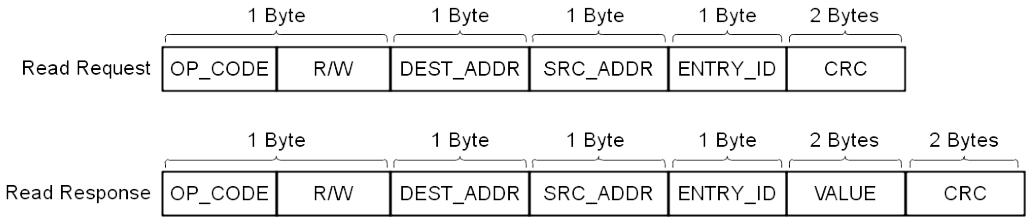

Abbildung A.2: Die Read Request und Response Frames des Bus-Protokolls.

Abbildung A.3 zeigt das Sequenzdiagramm eines validen Lesezugriffs.

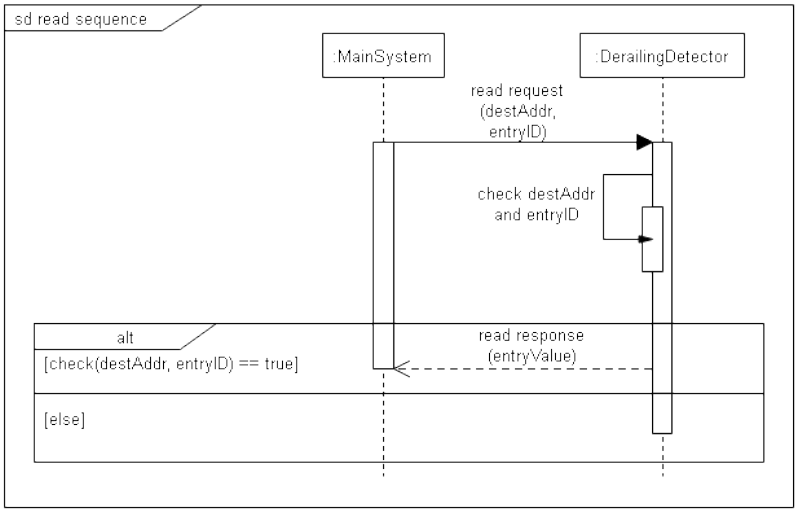

Abbildung A.3: Sequenzdiagramm eines Lesezugriffs des Bus Protokolls.

Write Request

Ein Write Request fordert einen Schreibzugriff auf einen bestimmten Tabellen Eintrag eines anderen Bus-Teilnehmers an. Das `VALUE`-Element hält den zu schreibenden Wert. Ist der Zugriff valide, wird mit einer Write Response der Zugriff quittiert. Die Write-Frames sind in Abbildung A.4 dargestellt. Ist der Zugriff nicht valide, wird mit einer Error Response (siehe A.1.1) geantwortet.

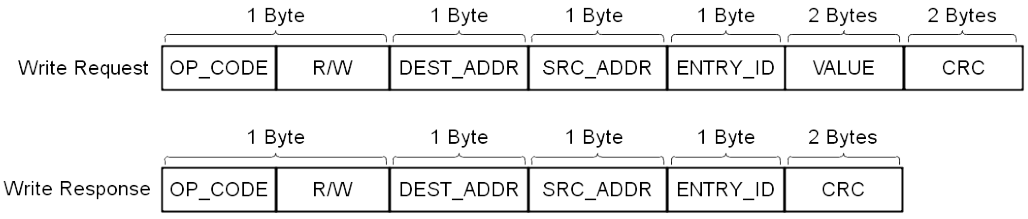

Abbildung A.4: Die Write Request und -Response Frames des Bus-Protokolls.

Abbildung A.5 zeigt das Sequenzdiagramm eines Schreibzugriffs.

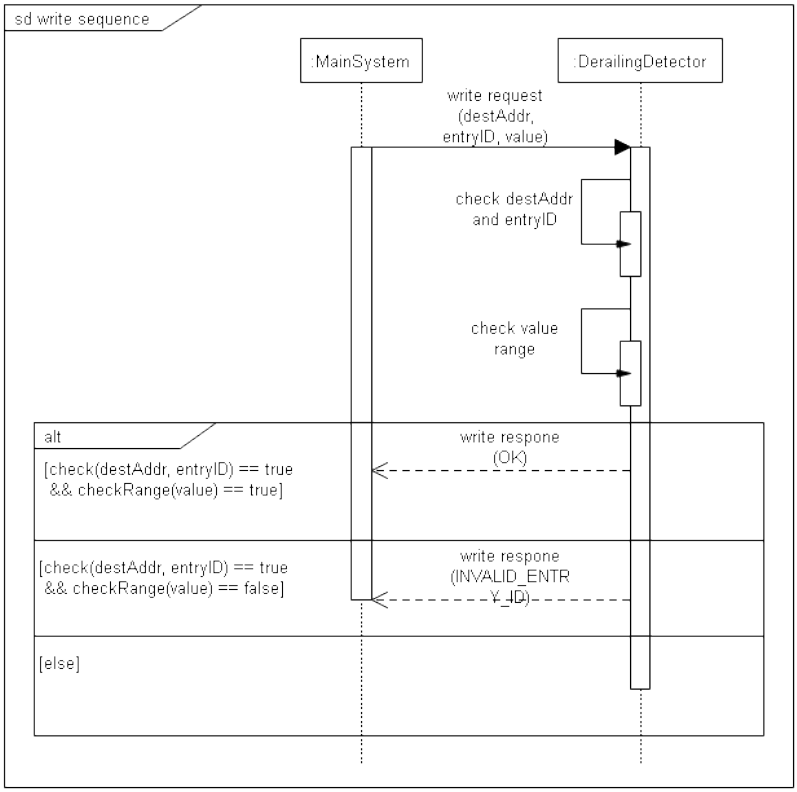

Abbildung A.5: Sequenzdiagramm eines Schreibzugriffs des Bus Protokolls.

Error Response

Ist ein empfangener Schreib- oder Lesezugriff invalide, so wird mit einer Error Response geantwortet. Gründe für Fehler umfassen, ein Wert außerhalb des erlaubten Bereichs, Adressierung eines nicht vorhandenen Tabellen Eintrags und fehlende Zugriff-Berechtigungen. Sie werden durch das ERROR_CODE-Element (siehe A.3) angegeben. Abbildung A.6 zeigt den Error Frame.

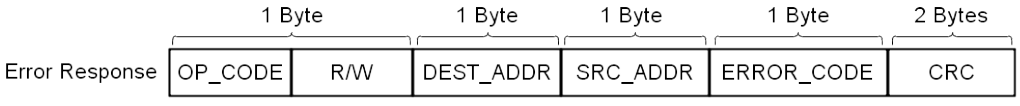

Abbildung A.6: Der Error Frame des Bus Protokolls.

ERROR_CODE	Beschreibung
0x00	Wert außerhalb des erlaubten Bereichs
0x01	Tabellen Eintrag existiert nicht
0x02	Keine Lese-Berechtigung
0x03	Keine Schreib-Berechtigung
0x04	Unbekannter Fehler

Tabelle A.3: Die Error Codes des Bus Protokolls.

Event Message

Ereignisse, wie etwa eine Schwellwertüberschreitung, werden mit Event Messages anderen Bus-Teilnehmern mitgeteilt. Event Messages müssen nicht mit einer Antwort vom Adressaten quittiert werden. Abbildung A.7 zeigt den Frame einer Event Nachricht. Das ENTRY_ID und das VALUE Element enthalten den aktuellen Wert des Tabellen Eintrags, der durch die Event Nachricht anderen Bus-Teilnehmern mitgeteilt werden soll.

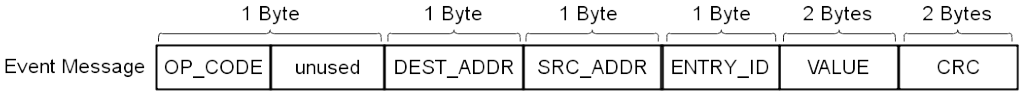

Abbildung A.7: Der Event Frame des Bus Protokolls.

A.1.2 Daten Tabelle

Die Daten Tabelle kann bis zu 256 Einträge enthalten. Jeder Eintrag hält dabei einen 16-Bit großen Wert. Dieser Wert kann entweder System intern aktualisiert werden oder von einem Bus-Teilnehmer gesetzt werden. Jedem Eintrag können individuell Lese- und Schreibrechte zugeordnet werden. Einträgen denen eine Schreibberechtigung zugeordnet ist, kann eine Überprüfung des Wertebereichs für Schreibzugriffe hinzugefügt werden. Über die ENTRY_ID wird bestimmt auf welchen Eintrag zugegriffen werden soll.

Tabelle A.4 listet die Tabellen Einträge des Derailing Alarm Systems auf.

ENTRY_ID	R/W	Beschreibung	Wertebereich
0x00	R	System Status	n.a.
0x01	R	System Error	n.a.
0x11	R/W	Accelerometer Schwellwert	n.a.
0x12	R/W	Gyrometer Schwellwert	n.a.
0x13	R/W	Refresh Rate in ms	10 - 3000
0x21	R	Accelerometer X-Wert	n.a.
0x22	R	Accelerometer Y-Wert	n.a.
0x23	R	Accelerometer Z-Wert	n.a.
0x31	R	Gyrometer X-Wert	n.a.
0x32	R	Gyrometer Y-Wert	n.a.
0x33	R	Gyrometer Z-Wert	n.a.
0x41	R	Magnetometer X-Wert	n.a.
0x42	R	Magnetometer Y-Wert	n.a.
0x43	R	Magnetometer Z-Wert	n.a.

Tabelle A.4: Die Einträge der Daten Tabelle.

A.1.3 BNO055 Sensor

Bei dem BNO055 handelt es sich um einen Smart Sensor der eine Sensor-Fusion betreibt. Für dieses Projekt wird diese allerdings nicht eingesetzt. Stattdessen werden die rohen Messdaten von Accelerometer, Gyrometer und Magnetometer periodisch ausgelesen und in die entsprechenden Einträge der Daten Tabelle geschrieben. Neben der I2C-Schnittstelle ist der Sensor auch noch über einen Interrupt-Pin mit dem Mikrocontroller verbunden. Bei dem Konfigurationsvorgang des Sensors werden Interrupts für Überschreitung bestimmter Schwellwerte von Accelerometer und Gyrometer aktiviert. Findet eine Überschreitung statt, signalisiert der Sensor dies über das Interrupt-Signal. Danach muss der Mikrocontroller die Interrupt-Quelle durch Auslesen des entsprechenden Registers feststellen und den Wert der zur Überschreitung geführt hat auslesen.
Der Sensor verfügt über einen Selbsttest. Dieser sollte nach einem Reset ausgeführt werden, um den korrekten Zustand des Sensors zu verifizieren.

A.1.4 Status LEDs

Error LED

Die Error LED zeigt interne Fehlerzustände an. Ist das System betriebsbereit und hat keine Fehler erkannt, so ist die LED ausgeschaltet. Bei Auftreten eines Fehlers wird sie angeschaltet.

Accelerometer & Gyrometer LED

Diese beiden LEDs zeigen Schwellwertüberschreitungen in der Beschleunigung bzw. Winkelgeschwindigkeit an. Tritt eine Überschreitung auf, wird die entsprechende LED für 500 ms eingeschaltet.

A.2 Quellcode zu Kapitel 4.3.1

```c
/**
 * @file       LEDDriver.h
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file declares the public interface for LEDDriver.
 *
 * LEDDriver is responsible for controlling the status LEDs. This
 * includes setting and resetting the Error LED and blinking the
 * Accelerometer and Gyrometer threshold step over LEDs.
 */

#ifndef LED_DRIVER_H
#define LED_DRIVER_H

#include <stdint.h>

/**
 * @brief Blink period for accelerometer and gyrometer LED in ms
 */
#define LED_BLINK_PERIOD                    (500)

/**
 * @{
 * @name LEDDriver initialization and deinitialization functions
 */
void LEDDriver_Init(void);
void LEDDriver_Deinit(void);
/** @} */

/**
 * @{
 * @name Error LED control functions
 */
void LEDDriver_SetErrorLED(void);
void LEDDriver_ResetErrorLED(void);
/** @} */

/**
 * @{
 * @name Accelerometer and gyrometer control functions
 */
void LEDDriver_BlinkAccelerometerLED(void);
void LEDDriver_BlinkGyrometerLED(void);
/** @} */

#endif /* LED_DRIVER_H */
```

Listing A.1: `LEDDriver.h`

```c
/**
 * @file       LEDDriver.c
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file contains the implementation for the LEDDriver.
 *
 * LEDDriver is responsible for controlling the status LEDs. This
 * includes setting and resetting the Error LED and blinking the
 * Accelerometer and Gyrometer threshold step over LEDs.
 */

#include "LEDDriver.h"

#include "OSAL_Timer.h"
```

```
16 #ifdef PRODUCTION
17     #include "HAL_GPIO.h"
18 #else
19     #include "HAL_GPIODouble.h"
20 #endif
21
22 static OSAL_Timer_t* accelerometerBlinkTimer;
23 static OSAL_Timer_t* gyrometerBlinkTimer;
24
25 static void accelerometerTimerCallback(void);
26 static void gyrometerTimerCallback(void);
27
28 /**
29  * @brief      Initializes the LEDDriver and the LED GPIOs.
30  */
31 void LEDDriver_Init(void)
32 {
33     // initialize LED GPIO outputs
34     HAL_GPIO_ConfigureOutput(GPIO_BANK_B_BASE_ADDR, 0);
35     HAL_GPIO_ConfigureOutput(GPIO_BANK_B_BASE_ADDR, 1);
36     HAL_GPIO_ConfigureOutput(GPIO_BANK_B_BASE_ADDR, 2);
37
38     HAL_GPIO_ResetOutput(GPIO_BANK_B_BASE_ADDR, 0);
39     HAL_GPIO_ResetOutput(GPIO_BANK_B_BASE_ADDR, 1);
40     HAL_GPIO_ResetOutput(GPIO_BANK_B_BASE_ADDR, 2);
41
42     // create blink timers
43     accelerometerBlinkTimer = OSAL_Timer_Create(LED_BLINK_PERIOD, accelerometerTimerCallback);
44     gyrometerBlinkTimer = OSAL_Timer_Create(LED_BLINK_PERIOD, gyrometerTimerCallback);
45 }
46
47 /**
48  * @brief      Deinitializes the LEDDriver.
49  */
50 void LEDDriver_Deinit(void)
51 {
52     OSAL_Timer_Destroy(accelerometerBlinkTimer);
53     OSAL_Timer_Destroy(gyrometerBlinkTimer);
54 }
55
56 /**
57  * @brief      Sets the Error LED.
58  */
59 void LEDDriver_SetErrorLED(void)
60 {
61     HAL_GPIO_SetOutput(GPIO_BANK_B_BASE_ADDR, 0);
62 }
63
64 /**
65  * @brief      Reset the Error LED.
66  */
67 void LEDDriver_ResetErrorLED(void)
68 {
69     HAL_GPIO_ResetOutput(GPIO_BANK_B_BASE_ADDR, 0);
70 }
71
72 /**
73  * @brief      Blinks the accelerometer threshold step-over LED.
74  */
75 void LEDDriver_BlinkAccelerometerLED(void)
76 {
77     HAL_GPIO_SetOutput(GPIO_BANK_B_BASE_ADDR, 1);
78     OSAL_Timer_Start(accelerometerBlinkTimer);
79 }
80
81 /**
82  * @brief      Blinks the gyrometer threshold step-over LED.
83  */
84 void LEDDriver_BlinkGyrometerLED(void)
```

```
85 {
86     HAL_GPIO_SetOutput(GPIO_BANK_B_BASE_ADDR, 2);
87     OSAL_Timer_Start(gyrometerBlinkTimer);
88 }
89
90
91 /**
92  * @{
93  * @name Private functions
94  */
95 /**
96  * @brief      Callback function for accelerometer timer.
97  * Resets the accelerometer threshold step-over LED.
98  */
99 static void accelerometerTimerCallback(void)
100 {
101     HAL_GPIO_ResetOutput(GPIO_BANK_B_BASE_ADDR, 1);
102     OSAL_Timer_Reset(accelerometerBlinkTimer);
103 }
104
105 /**
106  * @brief      Callback function for gyrometer timer.
107  * Resets the gyrometer threshold step-over LED.
108  */
109 static void gyrometerTimerCallback(void)
110 {
111     HAL_GPIO_ResetOutput(GPIO_BANK_B_BASE_ADDR, 2);
112     OSAL_Timer_Reset(gyrometerBlinkTimer);
113 }
114 /** @} */
```

Listing A.2: `LEDDriver.c`

```
1  /**
2   * @file       HAL_GPIO.h
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file declares the public interface for the Hardware Abstraction
7   * Layer (HAL) GPIO.
8   */
9
10 #ifndef HAL_GPIO_H
11 #define HAL_GPIO_H
12
13 #include <stdint.h>
14
15 /**
16  * @brief Enum defining the possible IO states.
17  */
18 typedef enum
19 {
20     HAL_GPIO_LOW = 0x00,                                        ///< GPIO low state
21     HAL_GPIO_HIGH = 0x01                                        ///< GPIO high state
22 } HAL_GPIO_State_t;
23
24 /**
25  * @{
26  * @name GPIO-bank base address defines
27  */
28 #define GPIO_BANK_A_BASE_ADDR           ((uint32_t*)0x48000000)     ///< GPIO-bank A base address
29 #define GPIO_BANK_B_BASE_ADDR           ((uint32_t*)0x48000400)     ///< GPIO-bank B base address
30 #define GPIO_BANK_C_BASE_ADDR           ((uint32_t*)0x48000800)     ///< GPIO-bank C base address
31 #define GPIO_BANK_D_BASE_ADDR           ((uint32_t*)0x48000C00)     ///< GPIO-bank D base address
32 #define GPIO_BANK_E_BASE_ADDR           ((uint32_t*)0x48001000)     ///< GPIO-bank E base address
33 #define GPIO_BANK_F_BASE_ADDR           ((uint32_t*)0x48001400)     ///< GPIO-bank F base address
34 #define GPIO_BANK_G_BASE_ADDR           ((uint32_t*)0x48001800)     ///< GPIO-bank G base address
35 #define GPIO_BANK_H_BASE_ADDR           ((uint32_t*)0x48001C00)     ///< GPIO-bank H base address
```

```
36 /** @} */
37
38 /**
39  * @{
40  * @name GPIO mode configuration functions
41  */
42 void HAL_GPIO_ConfigureInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
43 void HAL_GPIO_ConfigureOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
44 /** @} */
45
46 /**
47  * @{
48  * @name GPIO read status functions
49  */
50 HAL_GPIO_State_t HAL_GPIO_ReadInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
51 HAL_GPIO_State_t HAL_GPIO_ReadOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
52 /** @} */
53
54 /**
55  * @{
56  * @name GPIO state control functions
57  */
58 void HAL_GPIO_SetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
59 void HAL_GPIO_ResetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
60 /** @} */
61
62 #endif /* HAL_GPIO_H */
```

Listing A.3: `HAL_GPIO.h`

```
1  /**
2   * @file       HAL_GPIODouble.h
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file declares the interface of the test double
7   * for the HAL_GPIO-module and the necessary preprocessor substitutions
8   */
9
10 #ifndef HAL_GPIO_DOUBLE_H
11 #define HAL_GPIO_DOUBLE_H
12
13 #include "HAL_GPIO.h"
14 #include <stdint.h>
15
16 /**
17  * @brief Enum defining the I/O modes
18  */
19 typedef enum
20 {
21     INPUT,
22     OUTPUT
23 } Mode_t;
24
25 /**
26  * @{
27  * @name Preprocessor substitutions macros
28  * @brief These macros replace the original function calls with
29  * calls of the functions of the double
30  */
31 #define HAL_GPIO_ConfigureInput(gpioBankAddress, gpioNumber)
       Double_HAL_GPIO_ConfigureInput(gpioBankAddress, gpioNumber)
32 #define HAL_GPIO_ConfigureOutput(gpioBankAddress, gpioNumber)
       Double_HAL_GPIO_ConfigureOutput(gpioBankAddress, gpioNumber)
33 #define HAL_GPIO_ReadInput(gpioBankAddress, gpioNumber) Double_HAL_GPIO_ReadInput(gpioBankAddress,
       gpioNumber)
34 #define HAL_GPIO_ReadOutput(gpioBankAddress, gpioNumber)
       Double_HAL_GPIO_ReadOutput(gpioBankAddress, gpioNumber)
```

```
35 #define HAL_GPIO_SetOutput(gpioBankAddress, gpioNumber) Double_HAL_GPIO_SetOutput(gpioBankAddress,
       gpioNumber)
36 #define HAL_GPIO_ResetOutput(gpioBankAddress, gpioNumber)
       Double_HAL_GPIO_ResetOutput(gpioBankAddress, gpioNumber)
37 /** @} */
38
39 /**
40  * @{
41  * @name Double functions for replacing the original functions
42  */
43 void Double_HAL_GPIO_ConfigureInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
44 void Double_HAL_GPIO_ConfigureOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
45 HAL_GPIO_State_t Double_HAL_GPIO_ReadInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
46 HAL_GPIO_State_t Double_HAL_GPIO_ReadOutput(volatile uint32_t* gpioBankAddress, uint8_t
       gpioNumber);
47 void Double_HAL_GPIO_SetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
48 void Double_HAL_GPIO_ResetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
49 /** @} */
50
51 /**
52  * @{
53  * @name Functions for controlling the double
54  */
55 void Double_HAL_GPIO_Reset(void);
56 volatile uint32_t* Double_HAL_GPIO_GetLastBankAddress(void);
57 uint8_t Double_HAL_GPIO_GetLastGPIONumber(void);
58 void Double_HAL_GPIO_SetGPIOState(HAL_GPIO_State_t state, uint8_t gpioNumber);
59 HAL_GPIO_State_t Double_HAL_GPIO_GetGPIOState(uint8_t gpioNumber);
60 Mode_t Double_HAL_GPIO_GetGPIOMode(uint8_t gpioNumber);
61 /** @} */
62
63 #endif /* HAL_GPIO_DOUBLE_H */
```

Listing A.4: `HAL_GPIODouble.h`

```
1  /**
2   * @file       HAL_GPIODouble.c
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the implementation of the test double
7   * for the HAL_GPIO-module.
8   */
9
10 #include "HAL_GPIODouble.h"
11
12 #include <stdbool.h>
13 #include <stddef.h>
14
15 /**
16  * @brief      Defines the number of IOs per IO-bank
17  */
18 #define IOS_PER_BANK                    (16)
19
20 static Mode_t gpioModes[IOS_PER_BANK] = {INPUT};
21 static HAL_GPIO_State_t gpioStates[IOS_PER_BANK] = {HAL_GPIO_LOW};
22
23 volatile static uint32_t* lastBankAddress = NULL;
24 static uint8_t lastGPIONumber = 0;
25
26
27 static void saveParameters(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
28 static bool checkGPIONumberRange(uint8_t gpioNumber);
29
30 /**
31  * @{
32  * @name Double replacement functions
33  */
```

```
34 /**
35  * @brief      Replaces the HAL_GPIO_ConfigureInput function.
36  *
37  * @param[in]  gpioBankAddress  The GPIO bank address
38  * @param[in]  gpioNumber       The GPIO number
39  */
40 void Double_HAL_GPIO_ConfigureInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
41 {
42     saveParameters(gpioBankAddress, gpioNumber);
43 
44     if(checkGPIONumberRange(gpioNumber))
45     {
46         gpioModes[gpioNumber] = INPUT;
47     }
48 }
49 
50 /**
51  * @brief      Replaces the HAL_GPIO_ConfigureOutput function.
52  *
53  * @param[in]  gpioBankAddress  The GPIO bank address
54  * @param[in]  gpioNumber       The GPIO number
55  */
56 void Double_HAL_GPIO_ConfigureOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
57 {
58     saveParameters(gpioBankAddress, gpioNumber);
59 
60     if(checkGPIONumberRange(gpioNumber))
61     {
62         gpioModes[gpioNumber] = OUTPUT;
63     }
64 }
65 
66 /**
67  * @brief      Replaces the HAL_GPIO_ReadInput function.
68  *
69  * @param[in]  gpioBankAddress  The GPIO bank address
70  * @param[in]  gpioNumber       The GPIO number
71  *
72  * @return     Returns the GPIO state.
73  */
74 HAL_GPIO_State_t Double_HAL_GPIO_ReadInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
75 {
76     saveParameters(gpioBankAddress, gpioNumber);
77 
78     HAL_GPIO_State_t state = HAL_GPIO_LOW;
79     if(checkGPIONumberRange(gpioNumber))
80     {
81         state = gpioStates[gpioNumber];
82     }
83     return state;
84 }
85 
86 /**
87  * @brief      Replaces the HAL_GPIO_ReadOutput function.
88  *
89  * @param[in]  gpioBankAddress  The GPIO bank address
90  * @param[in]  gpioNumber       The GPIO number
91  *
92  * @return     Returns the GPIO state.
93  */
94 HAL_GPIO_State_t Double_HAL_GPIO_ReadOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
95 {
96     saveParameters(gpioBankAddress, gpioNumber);
97 
98     HAL_GPIO_State_t state = HAL_GPIO_LOW;
99     if(checkGPIONumberRange(gpioNumber))
100     {
101         state = gpioStates[gpioNumber];
102     }
```

```
103     return state;
104 }
105
106 /**
107  * @brief      Replaces the HAL_GPIO_SetOutput function.
108  *
109  * @param[in]  gpioBankAddress  The GPIO bank address
110  * @param[in]  gpioNumber       The GPIO number
111  */
112 void Double_HAL_GPIO_SetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
113 {
114     saveParameters(gpioBankAddress, gpioNumber);
115
116     if(checkGPIONumberRange(gpioNumber))
117     {
118         gpioStates[gpioNumber] = HAL_GPIO_HIGH;
119     }
120 }
121
122 /**
123  * @brief      Replaces the HAL_GPIO_ResetOutput function.
124  *
125  * @param[in]  gpioBankAddress  The GPIO bank address
126  * @param[in]  gpioNumber       The GPIO number
127  */
128 void Double_HAL_GPIO_ResetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
129 {
130     saveParameters(gpioBankAddress, gpioNumber);
131
132     if(checkGPIONumberRange(gpioNumber))
133     {
134         gpioStates[gpioNumber] = HAL_GPIO_LOW;
135     }
136 }
137
138 /**
139  * @brief      Replaces the HAL_GPIO_Reset function.
140  */
141 void Double_HAL_GPIO_Reset(void)
142 {
143     lastBankAddress = NULL;
144     lastGPIONumber = 0;
145
146     uint8_t i;
147     for(i = 0; i < IOS_PER_BANK; i++)
148     {
149         gpioModes[i] = INPUT;
150         gpioStates[i] = HAL_GPIO_LOW;
151     }
152 }
153 /** @} */
154
155 /**
156  * @{
157  * @name Double control functions
158  */
159 /**
160  * @brief      Gets the last bank address provided as parameter
161  * to the double.
162  *
163  * @return     Returns address pointer to the last bank address
164  * provided as parameter to the double.
165  */
166 volatile uint32_t* Double_HAL_GPIO_GetLastBankAddress(void)
167 {
168     return lastBankAddress;
169 }
170
171 /**
```

```
172  * @brief      Gets the last GPIO number provided as parameter
173  * to the double.
174  *
175  * @return     Returns the last GPIO number provided as
176  * parameter to the double.
177  */
178 uint8_t Double_HAL_GPIO_GetLastGPIONumber(void)
179 {
180     return lastGPIONumber;
181 }
182 
183 /**
184  * @brief      Sets the current GPIO state used for double replacement
185  * functions.
186  *
187  * @param[in]  state       The GPIO state
188  * @param[in]  gpioNumber  The GPIO number
189  */
190 void Double_HAL_GPIO_SetGPIOState(HAL_GPIO_State_t state, uint8_t gpioNumber)
191 {
192     if(checkGPIONumberRange(gpioNumber))
193     {
194         gpioStates[gpioNumber] = state;
195     }
196 }
197 
198 /**
199  * @brief      Gets the current GPIO state of the provided gpioNumber.
200  *
201  * @param[in]  gpioNumber  The GPIO number
202  *
203  * @return     Returns the GPIO state.
204  */
205 HAL_GPIO_State_t Double_HAL_GPIO_GetGPIOState(uint8_t gpioNumber)
206 {
207     HAL_GPIO_State_t state = HAL_GPIO_LOW;
208     if(checkGPIONumberRange(gpioNumber))
209     {
210         state = gpioStates[gpioNumber];
211     }
212     return state;
213 }
214 
215 /**
216  * @brief      Gets the current GPIO mode of the provided gpioNumber.
217  *
218  * @param[in]  gpioNumber  The GPIO number
219  *
220  * @return     Returns the GPIO mode.
221  */
222 Mode_t Double_HAL_GPIO_GetGPIOMode(uint8_t gpioNumber)
223 {
224     Mode_t mode = INPUT;
225     if(checkGPIONumberRange(gpioNumber))
226     {
227         mode = gpioModes[gpioNumber];
228     }
229     return mode;
230 }
231 /** @} */
232 
233 /**
234  * @brief      Saves parameters provided in function call so they
235  * can be retrieved with the get last functions.
236  *
237  * @param[in]  gpioBankAddress  The GPIO bank address
238  * @param[in]  gpioNumber       The GPIO number
239  */
240 static void saveParameters(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
```

```
241 {
242     lastBankAddress = gpioBankAddress;
243     lastGPIONumber = gpioNumber;
244 }
245 
246 /**
247  * @brief      Checks whether gpioNumber is inside valid range.
248  *
249  * @param[in]  gpioNumber  The GPIO number
250  *
251  * @return     Returns true when valid, false otherwise.
252  */
253 static bool checkGPIONumberRange(uint8_t gpioNumber)
254 {
255     bool result = false;
256     if(gpioNumber < IOS_PER_BANK)
257     {
258         result = true;
259     }
260     return result;
261 }
```

Listing A.5: HAL_GPIODouble.c

```
1  /**
2   * @file       OSAL_Timer.h
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file declares the public interface for the Operating System Abstraction
7   * Layer (HAL) Timer.
8   */
9  
10 #ifndef OSAL_TIMER_H
11 #define OSAL_TIMER_H
12 
13 #include <stdint.h>
14 #include <stdbool.h>
15 
16 /**
17  * @brief Type definition of struct holding timer attributes.
18  */
19 typedef struct _OSAL_Timer OSAL_Timer_t;
20 
21 /**
22  * @brief Type definition for timer callback functions.
23  */
24 typedef void (*timerCallback_fctn)(void);
25 
26 /**
27  * @{
28  * @name Timer initialization and deinitalization functions
29  */
30 OSAL_Timer_t* OSAL_Timer_Create(uint16_t period, timerCallback_fctn callback);
31 void OSAL_Timer_Destroy(OSAL_Timer_t* timer);
32 /** @} */
33 
34 /**
35  * @{
36  * @name Timer control functions
37  */
38 bool OSAL_Timer_Start(OSAL_Timer_t* timer);
39 bool OSAL_Timer_Reset(OSAL_Timer_t* timer);
40 bool OSAL_Timer_Stop(OSAL_Timer_t* timer);
41 /** @} */
42 
43 #endif /* OSAL_TIMER_H */
```

Listing A.6: OSAL_Timer.h

```
/**
 * @file       OSAL_TimerDouble.h
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file declares the interface of the test double
 * for the OSAL_Timer-module.
 */

#ifndef OSAL_TIMER_DOUBLE_H
#define OSAL_TIMER_DOUBLE_H

#include "OSAL_Timer.h"

/**
 * @brief Maximum amount of timers
 */
#define MAX_TIMERS                      (4)

/**
 * @brief Struct for holding the attributes for a timer double.
 */
typedef struct _OSAL_Timer
{
    uint16_t period;
    timerCallback_fctn callback;
    bool started;
    bool stopped;
    bool reset;
} OSAL_Timer_t;

/**
 * @{
 * @name Timer double control functions
 */
void Double_OSAL_Timer_Reset(void);
void Double_OSAL_Timer_InvokeCallback(OSAL_Timer_t* timer);
OSAL_Timer_t* Double_OSAL_Timer_GetLastTimer(void);
OSAL_Timer_t* Double_OSAL_GetTimer(uint8_t index);
/** @} */

#endif /* OSAL_TIMER_DOUBLE_H */
```

Listing A.7: OSAL_TimerDouble.h

```
/**
 * @file       OSAL_TimerDouble.c
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file contains the implementation of the test double
 * for the OSAL_Timer-module.
 */

#include "OSAL_TimerDouble.h"

#include <stddef.h>
#include <stdlib.h>


static OSAL_Timer_t* lastTimer = NULL;
static void* timerList[MAX_TIMERS] = {NULL};
static uint8_t timerCount = 0;
```

```
19
20 /**
21  * @{
22  * @name Double functions for OSAL_Timer-module.
23  */
24 /**
25  * @brief      Double funtion for OSAL_Timer_Create().
26  *
27  * @param[in]  period    The timer period
28  * @param[in]  callback  The timer callback function
29  *
30  * @return     Returns a pointer to a timer struct when successful, NULL otherwise.
31  */
32 OSAL_Timer_t* OSAL_Timer_Create(uint16_t period, timerCallback_fctn callback)
33 {
34     OSAL_Timer_t* timer = (OSAL_Timer_t*) malloc(sizeof(OSAL_Timer_t));
35
36     if(timer != NULL)
37     {
38         timer->period = period;
39         timer->callback = callback;
40         timer->started = false;
41         timer->stopped = false;
42         timer->reset = false;
43     }
44
45     lastTimer = timer;
46     if(timerCount < MAX_TIMERS)
47     {
48         timerList[timerCount++] = timer;
49     }
50
51     return timer;
52 }
53
54 /**
55  * @brief      Double function for OSAL_Timer_Destroy().
56  *
57  * @param      timer  Pointer to the timer instance
58  */
59 void OSAL_Timer_Destroy(OSAL_Timer_t* timer)
60 {
61     if(timer == NULL) return;
62     uint8_t i;
63     for(i = 0; i < MAX_TIMERS; i++)
64     {
65         if(timerList[i] == timer)
66         {
67             timerList[i] = NULL;
68         }
69     }
70
71     free(timer);
72 }
73
74 /**
75  * @brief      Double function for OSAL_Timer_Start().
76  *
77  * @param      timer  Pointer to the timer instance
78  *
79  * @return     Returns true when successful, false otherwise.
80  */
81 bool OSAL_Timer_Start(OSAL_Timer_t* timer)
82 {
83     timer->started = true;
84     lastTimer = timer;
85     return true;
86 }
87
```

```
88 /**
89  * @brief      Double function for OSAL_Timer_Reset().
90  *
91  * @param      timer  Pointer to the timer instance.
92  *
93  * @return     Returns true when successful, false otherwise.
94  */
95 bool OSAL_Timer_Reset(OSAL_Timer_t* timer)
96 {
97     timer->reset = true;
98     lastTimer = timer;
99     return true;
100 }
101
102 /**
103  * @brief      Double function for OSAL_Timer_Stop().
104  *
105  * @param      timer  Pointer to the timer instance.
106  *
107  * @return     Returns true when successful, false otherwise.
108  */
109 bool OSAL_Timer_Stop(OSAL_Timer_t* timer)
110 {
111     timer->stopped = true;
112     lastTimer = timer;
113     return true;
114 }
115
116 /**
117  * @brief      Double function for OSAL_Timer_Reset().
118  */
119 void Double_OSAL_Timer_Reset(void)
120 {
121     lastTimer = NULL;
122     uint8_t i;
123     for(i = 0; i < MAX_TIMERS; i++)
124     {
125         timerList[i] = NULL;
126     }
127     timerCount = 0;
128 }
129 /** @} */
130
131 /**
132  * @{
133  * @name Double control functions
134  */
135 /**
136  * @brief      Invokes the callback for provided timer instance.
137  *
138  * @param      timer  Pointer to the timer instance.
139  */
140 void Double_OSAL_Timer_InvokeCallback(OSAL_Timer_t* timer)
141 {
142     timer->callback();
143 }
144
145 /**
146  * @brief      Gets last timer provided as parameter by last
147  * double function call.
148  *
149  * @return     Returns pointer to the last timer instance.
150  */
151 OSAL_Timer_t* Double_OSAL_Timer_GetLastTimer(void)
152 {
153     return lastTimer;
154 }
155
156 /**
```

```
157  * @brief      Gets timer instance specified by index.
158  *
159  * @param[in]  index  The timer index
160  *
161  * @return     Returns pointer to the timer instance.
162  */
163 OSAL_Timer_t* Double_OSAL_GetTimer(uint8_t index)
164 {
165     if(index < timerCount)
166     {
167         return (OSAL_Timer_t*) timerList[index];
168     }
169     else
170     {
171         return NULL;
172     }
173 }
174 /** @} */
```

Listing A.8: `OSAL_TimerDouble.c`

```
1  /**
2   * @file       LEDDriverTest.c
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the unit tests for the LEDDriver-module.
7   */
8
9  #include "LEDDriver.h"
10 #include "unity_fixture.h"
11
12 #include "HAL_GPIODouble.h"
13 #include "OSAL_TimerDouble.h"
14
15 #include <stdint.h>
16
17 #define GPIO_BANK                           GPIO_BANK_B_BASE_ADDR
18
19 /**
20  * @{
21  * @name LEDDriver test group
22  */
23 TEST_GROUP(LEDDriverTest);
24
25 TEST_SETUP(LEDDriverTest)
26 {
27     Double_HAL_GPIO_Reset();
28     Double_OSAL_Timer_Reset();
29
30     LEDDriver_Init();
31 }
32
33 TEST_TEAR_DOWN(LEDDriverTest)
34 {
35     LEDDriver_Deinit();
36 }
37 /** @} */
38
39 /**
40  * @{
41  * @name LEDDriver unit tests
42  */
43 TEST(LEDDriverTest, InitConfiguresPins0to2AsOutputs)
44 {
45     // assert
46     TEST_ASSERT(Double_HAL_GPIO_GetGPIOMode(0) == OUTPUT);
47     TEST_ASSERT(Double_HAL_GPIO_GetGPIOMode(1) == OUTPUT);
```

```
48      TEST_ASSERT(Double_HAL_GPIO_GetGPIOMode(2) == OUTPUT);
49  }
50
51  TEST(LEDDriverTest, InitTurnsLEDsOffByDefault)
52  {
53      // assert
54      TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW, Double_HAL_GPIO_GetGPIOState(0));
55      TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW, Double_HAL_GPIO_GetGPIOState(1));
56      TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW, Double_HAL_GPIO_GetGPIOState(2));
57  }
58
59  TEST(LEDDriverTest, InitUsesCorrectGPIOBank)
60  {
61      // assert
62      TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
63  }
64
65  TEST(LEDDriverTest, InitCreatesTwoTimersWith500msPeriods)
66  {
67      OSAL_Timer_t* timer1 = Double_OSAL_GetTimer(0);
68      OSAL_Timer_t* timer2 = Double_OSAL_GetTimer(1);
69
70      // assert
71      TEST_ASSERT_NOT_NULL(timer1);
72      TEST_ASSERT_EQUAL_UINT16(LED_BLINK_PERIOD, timer1->period);
73      TEST_ASSERT_NOT_NULL(timer2);
74      TEST_ASSERT_EQUAL_UINT16(LED_BLINK_PERIOD, timer2->period);
75  }
76
77  TEST(LEDDriverTest, SetErrorLEDTurnsOnLED0)
78  {
79      // act
80      LEDDriver_SetErrorLED();
81
82      // assert
83      TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
84      TEST_ASSERT_EQUAL_HEX8(0, Double_HAL_GPIO_GetLastGPIONumber());
85      TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_HIGH, Double_HAL_GPIO_GetGPIOState(0));
86  }
87
88  TEST(LEDDriverTest, ResetErrorLEDTurnsOffLED0)
89  {
90      // act
91      LEDDriver_ResetErrorLED();
92
93      // assert
94      TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
95      TEST_ASSERT_EQUAL_HEX8(0, Double_HAL_GPIO_GetLastGPIONumber());
96      TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW, Double_HAL_GPIO_GetGPIOState(0));
97  }
98
99  TEST(LEDDriverTest, BlinkAccelerometerTurnsOnLED1)
100 {
101     // act
102     LEDDriver_BlinkAccelerometerLED();
103
104     // assert
105     TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
106     TEST_ASSERT_EQUAL_HEX8(1, Double_HAL_GPIO_GetLastGPIONumber());
107     TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_HIGH, Double_HAL_GPIO_GetGPIOState(1));
108 }
109
110 TEST(LEDDriverTest, BlinkAccelerometerStartsTimer)
111 {
112     // act
113     LEDDriver_BlinkAccelerometerLED();
114     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
115
116     // assert
```

```
117     TEST_ASSERT(timer->started);
118 }
119
120 TEST(LEDDriverTest, BlinkAccelerometerTimerCallbackTurnsOffLED1)
121 {
122     // arrange
123     LEDDriver_BlinkAccelerometerLED();
124     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
125
126     // act
127     Double_OSAL_Timer_InvokeCallback(timer);
128
129     // assert
130     TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
131     TEST_ASSERT_EQUAL_HEX8(1, Double_HAL_GPIO_GetLastGPIONumber());
132     TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW, Double_HAL_GPIO_GetGPIOState(1));
133 }
134
135 TEST(LEDDriverTest, BlinkAccelerometerTimerCallbackResetsTimer)
136 {
137     // arrange
138     LEDDriver_BlinkAccelerometerLED();
139     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
140
141     // act
142     Double_OSAL_Timer_InvokeCallback(timer);
143
144     // assert
145     TEST_ASSERT(timer->reset);
146 }
147
148 TEST(LEDDriverTest, BlinkGyrometerTurnsOnLED2)
149 {
150     // act
151     LEDDriver_BlinkGyrometerLED();
152
153     // assert
154     TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
155     TEST_ASSERT_EQUAL_HEX8(2, Double_HAL_GPIO_GetLastGPIONumber());
156     TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_HIGH, Double_HAL_GPIO_GetGPIOState(2));
157 }
158
159 TEST(LEDDriverTest, BlinkGyrometerStartsTimer)
160 {
161     // act
162     LEDDriver_BlinkGyrometerLED();
163     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
164
165     // assert
166     TEST_ASSERT(timer->started);
167 }
168
169 TEST(LEDDriverTest, BlinkGyrometerTimerCallbackTurnsOffLED2)
170 {
171     // arrange
172     LEDDriver_BlinkGyrometerLED();
173     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
174
175     // act
176     Double_OSAL_Timer_InvokeCallback(timer);
177
178     // assert
179     TEST_ASSERT_EQUAL_PTR(GPIO_BANK, Double_HAL_GPIO_GetLastBankAddress());
180     TEST_ASSERT_EQUAL_HEX8(2, Double_HAL_GPIO_GetLastGPIONumber());
181     TEST_ASSERT_EQUAL_HEX8(HAL_GPIO_LOW, Double_HAL_GPIO_GetGPIOState(2));
182 }
183
184 TEST(LEDDriverTest, BlinkGyrometerTimerCallbackResetsTimer)
185 {
```

```
186     // arrange
187     LEDDriver_BlinkGyrometerLED();
188     OSAL_Timer_t* timer = Double_OSAL_Timer_GetLastTimer();
189
190     // act
191     Double_OSAL_Timer_InvokeCallback(timer);
192
193     // assert
194     TEST_ASSERT(timer->reset);
195 }
196 /** @} */
```

Listing A.9: `LEDDriverTest.c`

A.3 Umsetzung des Mock Objekts

```c
/**
 * @file       Mock.h
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file declares the public interface for the Mock module.
 *
 * The Mock module enables the user to easily implement Mock objects
 * for testing purposes. It features include handling of expectations,
 * running checks, verifying the expectations and creating a report in
 * text form.
 */

#ifndef MOCK_H
#define MOCK_H

#include "stdint.h"
#include "stdbool.h"
#include "stdarg.h"

/**
 * @brief Mock object struct.
 */
typedef struct _Mock Mock_t;

/**
 * @{
 * @name Mock initialization and deinitialization functions
 */
Mock_t* Mock_Create(uint8_t maxExpectations);
void Mock_Destroy(Mock_t* handle);
/** @} */

/**
 * @{
 * @name Mock expectation control functions
 */
bool Mock_AddExpectation(Mock_t* handle, void* expectation);
void* Mock_GetNextExpecation(Mock_t* handle);
/** @} */

/**
 * @{
 * @name Mock check functions
 */
void Mock_Check(Mock_t* handle, bool condition, const char* format, ...);
/** @} */

/**
 * @{
 * @name Mock verify and report functions
 */
bool Mock_Verify(Mock_t* handle);
char* Mock_Report(Mock_t* handle);
/** @} */
#endif /* MOCK_H */
```

Listing A.10: `Mock.h`

```c
/**
 * @file       Mock.c
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file contains the implementation for the Mock module.
```

```
 *
 * The Mock module enables the user to easily implement Mock objects
 * for testing purposes. It features include handling of expectations,
 * running checks, verifying the expectations and creating a report in
 * text form.
 */

#include "Mock.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

/**
 * @brief Mock status codes
 */
typedef enum
{
    MOCK_OK,                            ///< status OK
    MOCK_FEWER_CALLS_THAN_EXPECTED,     ///< fewer calls detected than expected
    MOCK_MORE_CALLS_THAN_EXPECTED       ///< more calls detected than expected
} Mock_StatusCode_t;

/**
 * @brief Mock struct holding private attributes.
 */
typedef struct _Mock
{
    void** expectations;                ///< array of expectation pointers
    uint8_t maxExpectations;            ///< maximum number of expectations
    uint8_t expectationCount;           ///< current number of expectations
    uint8_t currentExpectation;         ///< current number of resolved expectations
    bool failed;                        ///< failed status
    Mock_StatusCode_t status;           ///< mock status
    char* report;                       ///< report string
} Mock_t;

static bool checkCorrectNumberOfCalls(Mock_t* handle);

/**
 * @brief      Creates a new Mock instance.
 *
 * @param[in]  maxExpectations  The maximum number of expectations
 *
 * @return     Returns a pointer to the Mock instance when successful, otherwise NULL.
 */
Mock_t* Mock_Create(uint8_t maxExpectations)
{
    Mock_t* handle = (Mock_t*) malloc(sizeof(Mock_t*));
    if(handle == NULL) return NULL;

    // allocate memory for expectations
    handle->maxExpectations = maxExpectations;
    handle->expectations = (void**) malloc(sizeof(void*) * handle->maxExpectations);
    if(handle->expectations != NULL)
    {
        // initialize expectations
        uint8_t i;
        for(i = 0; i < handle->maxExpectations; i++)
        {
            handle->expectations[i] = NULL;
        }

        handle->expectationCount = 0;
        handle->currentExpectation = 0;
        handle->failed = false;
        handle->status = MOCK_OK;
        handle->report = NULL;
    }
```

```
76      else
77      {
78          handle = NULL;
79      }
80
81      return handle;
82  }
83
84  /**
85   * @brief      Destroys mock instance.
86   *
87   * @param      handle  Pointer to mock instance
88   */
89  void Mock_Destroy(Mock_t* handle)
90  {
91      if(handle == NULL) return;
92
93      // free expectation memory space
94      uint8_t i;
95      for(i = 0; i < handle->expectationCount; i++)
96      {
97          if(handle->expectations[i] != NULL)
98          {
99              void* expectation = handle->expectations[i];
100             free(expectation);
101         }
102     }
103
104     // free report memory space
105     if(handle->report != NULL)
106     {
107         free(handle->report);
108     }
109
110     free(handle);
111 }
112
113 /**
114  * @brief      Adds a new Expectation to mock instance.
115  *
116  * @param      handle       Pointer to mock instance
117  * @param      expectation  The expectation to be added
118  *
119  * @return     Returns true when successful, false otherwise.
120  */
121 bool Mock_AddExpectation(Mock_t* handle, void* expectation)
122 {
123     if(handle->expectationCount >= handle->maxExpectations) return false;
124
125     handle->expectations[handle->expectationCount++] = expectation;
126
127     return true;
128 }
129
130 /**
131  * @brief      Gets the next expectation. Should be called first in functions
132  * replacing the actual collaborator functions.
133  *
134  * @param      handle  Pointer to mock instance
135  *
136  * @return     Returns the pointer to the next expectation if available, NULL otherwise.
137  *
138  * @attention  The returned void* should be cast to the specific expectation type.
139  */
140 void* Mock_GetNextExpecation(Mock_t* handle)
141 {
142     // check wheter there is a next expecation
143     if(handle->expectations == NULL) return NULL;
144     if(handle->currentExpectation >= handle->expectationCount)
```

```
145     {
146         handle->currentExpectation++;
147         handle->status = MOCK_MORE_CALLS_THAN_EXPECTED;
148         return NULL;
149     }
150 
151     return handle->expectations[handle->currentExpectation++];
152 }
153 
154 /**
155  * @brief      Checks a condition and updates the mock instance status accordingly.
156  * Supposed to be used in functions replacing the actual collaborator functions.
157  *
158  * @param      handle     Pointer to mock instance
159  * @param[in]  condition  Condition to be checked
160  * @param[in]  format     Format string for report messages.
161  * @param[in]  ...        Parameters for formatted report message.
162  */
163 void Mock_Check(Mock_t* handle, bool condition, const char* format, ...)
164 {
165     if(condition) return;
166     if(handle->failed) return;
167 
168     // update mock status
169     handle->failed = true;
170 
171     // generate report message
172     va_list arglist;
173     va_start(arglist, format);
174     char buffer[128];
175     size_t length = vsnprintf(buffer, sizeof(buffer), format, arglist);
176 
177     // allocate memory for report message
178     handle->report = (char*) malloc(length);
179     if(handle->report == NULL) return;
180 
181     strcpy(handle->report, buffer);
182 }
183 
184 /**
185  * @brief      Verify the mock interactions. Should be called at the end
186  * of the unit test.
187  *
188  * @param      handle  Pointer to mock instance.
189  *
190  * @return     Returns true when no conflict was detected, false otherwise.
191  */
192 bool Mock_Verify(Mock_t* handle)
193 {
194     bool result = true;
195 
196     result &= checkCorrectNumberOfCalls(handle);
197     result &= !(handle->failed);
198 
199     return result;
200 }
201 
202 /**
203  * @brief      Generates the final report.
204  *
205  * @param      handle  Pointer to mock instance
206  *
207  * @return     Returns a char* to the report terminated with \0.
208  */
209 char* Mock_Report(Mock_t* handle)
210 {
211     // determine wheter there were failed checks
212     if(handle->report == NULL)
213     {
```

```
214         handle->report = (char*) malloc(17);
215         if(handle->report != NULL)
216         {
217             snprintf(handle->report, 17, "No checks failed");
218         }
219     }
220 
221     // generate report including report messages from failed checks
222     if(handle->status != MOCK_OK)
223     {
224         char buffer[256];
225         size_t length = snprintf(buffer, sizeof(buffer), "%s. Interface was called %d times
     instead of expected %d\n",
226                 handle->report, handle->currentExpectation, handle->expectationCount);
227 
228         handle->report = (char*) realloc(handle->report, length);
229         if(handle->report != NULL)
230         {
231             snprintf(handle->report, length, "%s", buffer);
232         }
233     }
234 
235     return handle->report;
236 }
237 
238 /**
239  * @brief      Checks whether the number of expected interactions
240  * matches the actual number of expectations.
241  *
242  * @param      handle  Pointer to the mock instance
243  *
244  * @return     Returns true when expectations were met, false otherwise.
245  */
246 static bool checkCorrectNumberOfCalls(Mock_t* handle)
247 {
248     bool result = true;
249 
250     if(handle->currentExpectation == handle->expectationCount)
251     {
252         handle->status = MOCK_OK;
253     }
254     else if(handle->currentExpectation < handle->expectationCount)
255     {
256         handle->status = MOCK_FEWER_CALLS_THAN_EXPECTED;
257         result = false;
258     }
259     else
260     {
261         result = false;
262     }
263     return result;
264 }
```

Listing A.11: `Mock.c`

A.4 Quellcode zu Kapitel 4.3.2

```
1  /**
2   * @file       I2CInterface.h
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file declares the I2C interface.
7   *
8   * Inlcudes function type definitions for reading a single register,
9   * reading multiple registers and writing a single register. Function
10  * pointers can be bundled in a I2C_Interface_t struct.
11  */
12
13 #ifndef I2C_INTERFACE_H
14 #define I2C_INTERFACE_H
15
16 #include <stdint.h>
17 #include <stdbool.h>
18
19 /**
20  * @brief Type definition for reading a single register function.
21  *
22  * @param[in]   address         I2C device bus address.
23  * @param[in]   registerAddress Address of register to be read.
24  * @param[out]  buffer          Buffer for read value of register
25  *
26  * @return Returns true when successful.
27  */
28 typedef bool (*I2C_ReadRegister)(uint8_t address, uint8_t registerAddress, uint8_t* buffer);
29
30 /**
31  * @brief Type definition for reading multiple register functions.
32  *
33  * @param[in]   address             I2C device bus address.
34  * @param[in]   startRegister       Start address of registers to be read.
35  * @param[in]   numberOfRegisters   Number of registers to be read.
36  * @param[out]  buffer              Buffer for read data of register.
37  *
38  * @return Returns the number of register that were read. Returns 0 if operation failed.
39  */
40 typedef uint8_t (*I2C_ReadMultipleRegisters)(uint8_t address, uint8_t startRegister, uint8_t
        numberOfRegisters, uint8_t* buffer);
41
42 /**
43  * @brief Type definition for writing a single register function.
44  *
45  * @param[in]   address         I2C device bus address.
46  * @param[in]   registerAddress Address of register to be read.
47  * @param[out]  value           New value to be written to register.
48  *
49  * @return Returns true when successful.
50  */
51 typedef bool (*I2C_WriteRegister)(uint8_t address, uint8_t registerAddress, uint8_t value);
52
53 /**
54  * @brief Type definition of struct holding function pointers for the I2CInterface functions.
55  */
56 typedef struct
57 {
58     I2C_ReadRegister readRegisterFunction;                      ///< read a single register
        function pointer
59     I2C_ReadMultipleRegisters readMultipleRegistersFunction;    ///< read multiple register
        function pointer
60     I2C_WriteRegister writeRegisterFunction;                    ///< write a single register
        function pointer
61 } I2C_Interface_t;
62
```

```
63 #endif /* I2C_INTERFACE_H */
```

Listing A.12: I2CInterface.h

```
1  /**
2   * @file       BNO055Driver.h
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the interface of the BNO055Driver-module.
7   *
8   * A driver for the BNO055 smart-sensor by Bosch. Includes functions for configuring
9   * the sensor, running the self-test, configuring the interrupts and reading x-, y-
10  * and z-axis data.
11  */
12
13 #ifndef BNO055_DRIVER_H
14 #define BNO055_DRIVER_H
15
16 #include "I2CInterface.h"
17
18 #include <stdint.h>
19 #include <stdbool.h>
20
21 /**
22  * @brief I2C bus address of sensor.
23  */
24 #define BNO055_I2C_ADDRESS                  ( 0x28 )
25
26 /**
27  * @brief System status codes.
28  */
29 typedef enum
30 {
31     BNO_SYSTEM_IDLE                         = 0x00,         ///< sensor is idle
32     BNO_SYSTEM_ERROR                        = 0x01,         ///< error occured
33     BNO_INITIALIZING_PERIPHERALS            = 0x02,         ///< sensor initializing periperals
34     BNO_SYSTEM_INITIALIZATION               = 0x03,         ///< sensor initializing
35     BNO_EXECUTING_SELFTEST                  = 0x04,         ///< self-test is executing
36     BNO_SENSOR_FUSION_RUNNING               = 0x05,         ///< sensor running in fusion mode
37     BNO_SYSTEM_RUNNING                      = 0x06          ///< system is running fine
38 } BNO055_SystemStatus_t;
39
40 /**
41  * @brief Sensor error codes.
42  */
43 typedef enum
44 {
45     BNO_NO_ERROR                            = 0x00,         ///< no error occured
46     BNO_PERIPHERAL_INITIALIZATION_ERROR     = 0x01,         ///< peripheral initialization failed
47     BNO_SYSTEM_INITIALIZATION_ERROR         = 0x02,         ///< system initialization failed
48     BNO_SELF_TEST_FAILED                    = 0x03,         ///< self-test failed
49     BNO_REGISTER_MAP_VALUE_OUT_OF_RANGE     = 0x04,         ///< register value not in valid range
50     BNO_REGISTER_MAP_ADDRESS_OUT_OF_RANGE   = 0x05,         ///< register address out of range
51     BNO_REGISTER_MAP_WRITE_ERROR            = 0x06,         ///< no write permissions for register
52     BNO_LOW_POWER_MODE_NOT_AVAILABLE        = 0x07,         ///< can't activate low power mode
53     BNO_ACC_POWER_MODE_NOT_AVAILABLE        = 0x08,         ///< accelerometer power mode not
       available
54     BNO_FUSION_ALGORITHM_CONFIG_ERROR       = 0x09,         ///< sensor fusion configuration failed
55     BNO_SENSOR_CONFIG_ERROR                 = 0x0A          ///< sensor configuration failed
56 } BNO055_ErrorCode_t;
57
58 /**
59  * @brief Enum for determining sensor interrupt sources.
60  */
61 typedef enum
62 {
63     BNO_NO_INTERRUPT_OCCURED                = ( 0x00 ),     ///< no interrupt occured
```

```
64      BNO_ACC_BSX_DRDY                        = (1 << 0),     ///< accerleometer BSX data ready
         interrupt
65      BNO_MAG_DRDY                            = (1 << 1),     ///< magnetometer data ready interrupt
66      BNO_GYRO_AM                             = (1 << 2),     ///< gyrometer any motion interrupt
67      BNO_GYR_HIGH_RATE                       = (1 << 3),     ///< gyrometer high rate interrupt
68      BNO_GYR_DRDY                            = (1 << 4),     ///< gyrometer data ready interrupt
69      BNO_ACC_HIGH_G                          = (1 << 5),     ///< accelerometer high-g interrupt
70      BNO_ACC_AM                              = (1 << 6),     ///< accelerometer any moting interrupt
71      BNO_ACC_NM                              = (1 << 7)      ///< accelerometer no motion interrupt
72  } BNO_Interrupt_t;
73
74  /**
75   * @brief Struct holding configuration register data.
76   */
77  typedef struct
78  {
79      uint8_t address;        ///< register address
80      uint8_t data;           ///< register value
81  } BNO055_ConfigData_t;
82
83  /**
84   * @{
85   * @name Driver initialization and deinitialization
86   */
87  void BNO055_Create(I2C_Interface_t* i2cIO);
88  void BNO055_Destroy(void);
89  /** @} */
90
91  /**
92   * @{
93   * @name Sensor control functions
94   */
95  bool BNO055_Reset(void);
96  bool BNO055_ActivateConfigMode(void);
97  bool BNO055_ActivateAMGMode(void);
98  bool BNO055_ConfigureSensor(void);
99  bool BNO055_ExecuteSelfTest(void);
100 /** @} */
101
102 /**
103  * @{
104  * @name Status getter functions
105  */
106 BNO055_SystemStatus_t BNO055_GetSystemStatus(void);
107 BNO055_ErrorCode_t BNO055_GetSystemError(void);
108 BNO_Interrupt_t BNO055_GetInterruptStatus(void);
109 /** @} */
110
111 /**
112  * @{
113  * @name Sensor data getter functions
114  */
115 bool BNO055_GetAccerlometerData(uint16_t* x, uint16_t* y, uint16_t* z);
116 bool BNO055_GetMagnetometerData(uint16_t* x, uint16_t* y, uint16_t* z);
117 bool BNO055_GetGyrometerData(uint16_t* x, uint16_t* y, uint16_t* z);
118 /** @} */
119
120 #endif /* BNO055_DRIVER_H */
```

Listing A.13: `BNO055Driver.h`

```
1  /**
2   * @file       BNO055Driver.c
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the implementation of the BNO055Driver-module.
7   *
```

```
 * A driver for the BNO055 smart-sensor by Bosch. Includes functions for configuring
 * the sensor, running the self-test, configuring the interrupts and reading x-, y-
 * and z-axis data.
 */

#include "BNO055Driver.h"

/**
 * @{
 * @name        Operation modes
 */
#define BNO_OPR_MODE_REG            (0x3D)          ///< operation mode register
#define BNO_CONFIG_MODE             (0b00000000)    ///< configuration operation mode
#define BNO_AMG_MODE                (0b00000111)    ///< AMG (accelerometer, magnetometer,
     gyrometer) operation mode
/** @} */

/**
 * @{
 * @name        System register addresses
 */
#define BNO_SYS_STATUS_REG          (0x39)          ///< system status register
#define BNO_SYS_ERROR_REG           (0x3A)          ///< system error register
#define BNO_SYS_TRIG_REG            (0x3F)          ///< system trigger register
#define BNO_ST_RESULT_REG           (0x36)          ///< self-test result register
#define BNO_INT_STATUS_REG          (0x37)          ///< interrupt status register
/** @} */

/**
 * @{
 * @name        Output data register addresses
 */
#define BNO_ACC_DATA_X_MSB          (0x09)          ///< acceleration data x-axis MSB register
#define BNO_ACC_DATA_X_LSB          (0x08)          ///< acceleration data x-axis LSB register
#define BNO_ACC_DATA_Y_MSB          (0x0B)          ///< acceleration data y-axis MSB register
#define BNO_ACC_DATA_Y_LSB          (0x0A)          ///< acceleration data y-axis LSB register
#define BNO_ACC_DATA_Z_MSB          (0x0D)          ///< acceleration data z-axis MSB register
#define BNO_ACC_DATA_Z_LSB          (0x0C)          ///< acceleration data z-axis LSB register
#define BNO_MAG_DATA_X_MSB          (0x0F)          ///< magnetometer data x-axis MSB register
#define BNO_MAG_DATA_X_LSB          (0x0E)          ///< magnetometer data x-axis LSB register
#define BNO_MAG_DATA_Y_MSB          (0x11)          ///< magnetometer data y-axis MSB register
#define BNO_MAG_DATA_Y_LSB          (0x10)          ///< magnetometer data y-axis LSB register
#define BNO_MAG_DATA_Z_MSB          (0x13)          ///< magnetometer data z-axis MSB register
#define BNO_MAG_DATA_Z_LSB          (0x12)          ///< magnetometer data z-axis LSB register
#define BNO_GYR_DATA_X_MSB          (0x15)          ///< gyrometer data x-axis MSB register
#define BNO_GYR_DATA_X_LSB          (0x14)          ///< gyrometer data x-axis LSB register
#define BNO_GYR_DATA_Y_MSB          (0x17)          ///< gyrometer data y-axis MSB register
#define BNO_GYR_DATA_Y_LSB          (0x16)          ///< gyrometer data y-axis LSB register
#define BNO_GYR_DATA_Z_MSB          (0x19)          ///< gyrometer data z-axis MSB register
#define BNO_GYR_DATA_Z_LSB          (0x18)          ///< gyrometer data z-axis LSB register
#define BNO_TEMP                    (0x34)          ///< temperature data register
/** @} */


extern BNO055_ConfigData_t bno055_configData[13];

static I2C_Interface_t* io;

static uint16_t convertToUint16(uint8_t msb, uint8_t lsb);
static bool getXYZData(uint8_t startRegister, uint16_t* x, uint16_t* y, uint16_t* z);

/**
 * @brief      Initializes the instance BNO055Driver instance.
 *
 * @param      i2cIO  I2C interface struct pointer
 */
void BNO055_Create(I2C_Interface_t* i2cIO)
{
    io = i2cIO;
```

```
76 }
77
78 /**
79  * @brief      Deinitializes the BNO055Driver instance
80  */
81 void BNO055_Destroy(void)
82 {
83
84 }
85
86 /**
87  * @brief      Resets the BNO055 sensor.
88  *
89  * @return     Returns true when successful.
90  */
91 bool BNO055_Reset(void)
92 {
93     return io->writeRegisterFunction(BNO055_I2C_ADDRESS, BNO_SYS_TRIG_REG, 0b00010000);
94 }
95
96 /**
97  * @brief      Activates the configuration mode of the BNO055 sensor.
98  *
99  * @return     Returns true when successful.
100  */
101 bool BNO055_ActivateConfigMode(void)
102 {
103     return io->writeRegisterFunction(BNO055_I2C_ADDRESS, BNO_OPR_MODE_REG, BNO_CONFIG_MODE);
104 }
105
106 /**
107  * @brief      Activates the Accelerometer-Magnetometer-Gyrometer mode.
108  *
109  * @return     Returns true when successful.
110  */
111 bool BNO055_ActivateAMGMode(void)
112 {
113     return io->writeRegisterFunction(BNO055_I2C_ADDRESS, BNO_OPR_MODE_REG, BNO_AMG_MODE);
114 }
115
116 /**
117  * @brief      Configures the BNO055 sensor according to settings specified in BNO055Config.c
118  *
119  * @return     Returns true when successful.
120  */
121 bool BNO055_ConfigureSensor(void)
122 {
123     bool result = true;
124
125     result &= BNO055_ActivateConfigMode();
126
127     // set configuration registers
128     uint8_t i;
129     for(i = 0; i < 13; i++)
130     {
131         result &= io->writeRegisterFunction(BNO055_I2C_ADDRESS, bno055_configData[i].address,
     bno055_configData[i].data);
132         if(!result)
133         {
134             break;
135         }
136     }
137
138     return result;
139 }
140
141 /**
142  * @brief      Triggers and executes self-test of the BNO055 sensor.
143  *
```

```
144  * @return     Returns true when the self-test is successful, else returns false.
145  */
146 bool BNO055_ExecuteSelfTest(void)
147 {
148     bool result = true;
149 
150     result &= BNO055_ActivateConfigMode();
151 
152     // trigger self test
153     result &= io->writeRegisterFunction(BNO055_I2C_ADDRESS, BNO_SYS_TRIG_REG, 0x01);
154 
155     // check whether self test is running
156     uint8_t sysStatusReg;
157     result &= io->readRegisterFunction(BNO055_I2C_ADDRESS, BNO_SYS_STATUS_REG, &sysStatusReg);
158     result &= (sysStatusReg == BNO_EXECUTING_SELFTEST);
159 
160     // wait 400 ms for self test to complete
161     // TODO: wait 400 ms
162 
163     // check result
164     if(result)
165     {
166         uint8_t sysErrorReg;
167         result &= (io->readRegisterFunction(BNO055_I2C_ADDRESS, BNO_SYS_ERROR_REG, &sysErrorReg));
168         result &= (sysErrorReg != BNO_SELF_TEST_FAILED);
169     }
170 
171     return result;
172 }
173 
174 /**
175  * @brief      Gets the system status.
176  *
177  * @return     Returns the system status code.
178  */
179 BNO055_SystemStatus_t BNO055_GetSystemStatus(void)
180 {
181     uint8_t status;
182     io->readRegisterFunction(BNO055_I2C_ADDRESS, BNO_SYS_STATUS_REG, &status);
183 
184     return (BNO055_SystemStatus_t) status;
185 }
186 
187 /**
188  * @brief      Gets the system error state.
189  *
190  * @return     Returns the system error code.
191  */
192 BNO055_ErrorCode_t BNO055_GetSystemError(void)
193 {
194     uint8_t error;
195     io->readRegisterFunction(BNO055_I2C_ADDRESS, BNO_SYS_ERROR_REG, &error);
196 
197     return (BNO055_ErrorCode_t) error;
198 }
199 
200 /**
201  * @brief      Returns the sensor interrupt status.
202  *
203  * @return     Retruns the interrupt source.
204  */
205 BNO_Interrupt_t BNO055_GetInterruptStatus(void)
206 {
207     BNO_Interrupt_t status = BNO_NO_INTERRUPT_OCCURED;
208     uint8_t buffer;
209     io->readRegisterFunction(BNO055_I2C_ADDRESS, BNO_INT_STATUS_REG, &buffer);
210 
211     // determine interrupt source
212     if(buffer & BNO_ACC_BSX_DRDY)       status = BNO_ACC_BSX_DRDY;
```

```
213     else if(buffer & BNO_MAG_DRDY)      status = BNO_MAG_DRDY;
214     else if(buffer & BNO_GYRO_AM)       status = BNO_GYRO_AM;
215     else if(buffer & BNO_GYR_HIGH_RATE) status = BNO_GYR_HIGH_RATE;
216     else if(buffer & BNO_GYR_DRDY)      status = BNO_GYR_DRDY;
217     else if(buffer & BNO_ACC_HIGH_G)    status = BNO_ACC_HIGH_G;
218     else if(buffer & BNO_ACC_AM)        status = BNO_ACC_AM;
219     else if(buffer & BNO_ACC_NM)        status = BNO_ACC_NM;
220
221     return status;
222 }
223
224 /**
225  * @brief      Gets the current accelerometer values.
226  *
227  * @param      x     Buffer for the x-axis value.
228  * @param      y     Buffer for the y-axis value.
229  * @param      z     Buffer for the z-axis value.
230  *
231  * @return     Returns true when successful.
232  */
233 bool BNO055_GetAccerlometerData(uint16_t* x, uint16_t* y, uint16_t* z)
234 {
235     return getXYZData(BNO_ACC_DATA_X_LSB, x, y, z);
236 }
237
238 /**
239  * @brief      Gets the current Magnetometer values.
240  *
241  * @param      x     Buffer for the x-axis value.
242  * @param      y     Buffer for the y-axis value.
243  * @param      z     Buffer for the z-axis value.
244  *
245  * @return     Return true when successful.
246  */
247 bool BNO055_GetMagnetometerData(uint16_t* x, uint16_t* y, uint16_t* z)
248 {
249     return getXYZData(BNO_MAG_DATA_X_LSB, x, y, z);
250 }
251
252 /**
253  * @brief      Gets the current Gyrometer values.
254  *
255  * @param      x     Buffer for the x-axis value.
256  * @param      y     Buffer for the y-axis value.
257  * @param      z     Buffer for the z-axis value.
258  *
259  * @return     Return true when successful.
260  */
261 bool BNO055_GetGyrometerData(uint16_t* x, uint16_t* y, uint16_t* z)
262 {
263     return getXYZData(BNO_GYR_DATA_X_LSB, x, y, z);
264 }
265
266 /**
267  * @{
268  * @name Private functions
269  */
270 /**
271  * @brief      Converts two individual bytes to a unsigned 16-bit integer.
272  *
273  * @param[in]  msb   Most significant byte.
274  * @param[in]  lsb   Least significant byte.
275  *
276  * @return     Returns the converted unsigned 16-bit integer.
277  */
278 static uint16_t convertToUint16(uint8_t msb, uint8_t lsb)
279 {
280     uint16_t value = (((uint16_t)msb) << 8) + (uint16_t) lsb;
281
```

```
282     return value;
283 }
284
285 /**
286  * @brief      Reads x-, y- and z-axis value from the sensor
287  *
288  * @param[in]  startRegister  The start register address.
289  * @param      x              Buffer for the x-axis value.
290  * @param      y              Buffer for the y-axis value.
291  * @param      z              Buffer for the z-axis value.
292  *
293  * @return     Returns true when successful.
294  */
295 static bool getXYZData(uint8_t startRegister, uint16_t* x, uint16_t* y, uint16_t* z)
296 {
297     bool result = true;
298     uint8_t buffer[6];
299
300     // block read the register values containing the xyz data
301     if(io->readMultipleRegistersFunction(BNO055_I2C_ADDRESS, startRegister, 6, buffer) == 6)
302     {
303         *x = convertToUint16(buffer[1], buffer[0]);
304         *y = convertToUint16(buffer[3], buffer[2]);
305         *z = convertToUint16(buffer[5], buffer[4]);
306         result = true;
307     }
308
309     return result;
310 }
311 /** @} */
```

Listing A.14: `BNO055Driver.c`

```
1  /**
2   * @file       I2CMock.h
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file declares the public interface for the I2CMock.
7   *
8   * @attention  This mock object is based on the basic mock implementation.
9   */
10
11 #ifndef I2C_MOCK_H
12 #define I2C_MOCK_H
13
14 #include "Mock.h"
15 #include "I2CInterface.h"
16
17 #include <stdint.h>
18 #include <stdbool.h>
19
20 /**
21  * Enum defining interaction types.
22  */
23 typedef enum
24 {
25     READ_REGISTER,
26     READ_MULTIPLE_REGISTERS,
27     WRITE_REGISTER
28 } Interaction_Type_t;
29
30 /**
31  * @{
32  * @name Mock initialization and deinitialization functions
33  */
34 void I2CMock_Create(uint8_t maxExpectations);
35 void I2CMock_Destroy(void);
```

```
36 /** @} */
37
38 /**
39  * @{
40  * @name I2C interface functions
41  */
42 I2C_Interface_t* I2CMock_GetInterface(void);
43 /** @} */
44
45 /**
46  * @{
47  * @name Mock control functions
48  */
49 bool I2CMock_AddExpectation(Interaction_Type_t type, uint8_t address, uint8_t registerAddress,
       uint8_t* data, uint8_t numberOfRegisters);
50 bool I2CMock_Verify(void);
51 char* I2CMock_Report(void);
52 /** @} */
53
54 #endif /* I2C_MOCK_H */
```

Listing A.15: `I2CMock.h`

```
1  /**
2   * @file       I2CMock.c
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the implementation for the I2CMock.
7   *
8   * @attention  This mock object is based on the basic mock implementation.
9   */
10
11 #include "I2CMock.h"
12 #include "unity.h"
13
14 #include <stdlib.h>
15 #include <string.h>
16
17 /**
18  * @brief Struct for holding the attributes of an mock expectation.
19  */
20 typedef struct
21 {
22     Interaction_Type_t type;
23     uint8_t address;
24     uint8_t registerAddress;
25     uint8_t* data;
26     uint8_t numberOfRegisters;
27 } Expectation_t;
28
29 /**
30  * @brief Mock instance
31  */
32 static Mock_t* mock;
33 /**
34  * @brief I2C interface implementation
35  */
36 static I2C_Interface_t io;
37
38 static bool readRegister(uint8_t address, uint8_t registerAddress, uint8_t* buffer);
39 static uint8_t readMultipleRegisters(uint8_t address, uint8_t startRegister, uint8_t
       numberOfRegisters, uint8_t* buffer);
40 static bool writeRegister(uint8_t address, uint8_t registerAddress, uint8_t value);
41
42 /**
43  * @{
44  * @name I2CMock initialization and deinitialization functions
```

```
45  */
46 /**
47  * @brief      Initializes the instance of the I2C Mock.
48  *
49  * @param[in]  maxExpectations  The maximum number of expectations
50  */
51 void I2CMock_Create(uint8_t maxExpectations)
52 {
53     io.readRegisterFunction = readRegister;
54     io.readMultipleRegistersFunction = readMultipleRegisters;
55     io.writeRegisterFunction = writeRegister;
56 
57     mock = Mock_Create(maxExpectations);
58 }
59 
60 /**
61  * @brief      Destroys the instance of the I2C Mock.
62  */
63 void I2CMock_Destroy(void)
64 {
65     io.readRegisterFunction = NULL;
66     io.readMultipleRegistersFunction = NULL;
67     io.writeRegisterFunction = NULL;
68 
69     Mock_Destroy(mock);
70 }
71 /** @} */
72 
73 /**
74  * @{
75  * @name I2C interface functions
76  */
77 /**
78  * @brief      Gets the struct containing the I2C interface function pointers.
79  *
80  * @return     Returns the I2C interface struct.
81  */
82 I2C_Interface_t* I2CMock_GetInterface(void)
83 {
84     return &io;
85 }
86 /** @} */
87 
88 /**
89  * @{
90  * @name I2CMock control functions
91  */
92 /**
93  * @brief      Adds a new expectation to the I2C Mock.
94  *
95  * @param[in]  type               The interaction type
96  * @param[in]  address            The I2C address
97  * @param[in]  registerAddress    The I2C register address
98  * @param      data               Pointer to the buffer / data used in interaction
99  * @param[in]  numberOfRegisters  The number of registers to be read
100 *
101 * @return     Returns true when successful, false otherwise.
102 */
103 bool I2CMock_AddExpectation(Interaction_Type_t type, uint8_t address, uint8_t registerAddress,
      uint8_t* data, uint8_t numberOfRegisters)
104 {
105     // check for valid numberOfRegisters
106     if(type != READ_MULTIPLE_REGISTERS && numberOfRegisters > 1) return false;
107 
108     bool result = false;
109 
110     // generate expectation instance
111     Expectation_t* expectation = (Expectation_t*) malloc(sizeof(Expectation_t));
112     if(expectation != NULL)
```

```
113     {
114         expectation->type = type;
115         expectation->address = address;
116         expectation->registerAddress = registerAddress;
117         expectation->data = data;
118         expectation->numberOfRegisters = numberOfRegisters;
119 
120         // add the expectation
121         result = Mock_AddExpectation(mock, expectation);
122         if(!result)
123         {
124             free(expectation);
125         }
126     }
127     else
128     {
129         result = false;
130     }
131 
132     return result;
133 }
134 
135 /**
136  * @brief      Verifies the interactions with the I2CMock.
137  *
138  * @return     Returns true when actual interactions match the expectations, false otherwise.
139  */
140 bool I2CMock_Verify(void)
141 {
142     return Mock_Verify(mock);
143 }
144 
145 /**
146  * @brief      Generates the reports for the I2CMock.
147  *
148  * @return     Returns pointer to char array holding the report (\0 terminated).
149  */
150 char* I2CMock_Report(void)
151 {
152     return Mock_Report(mock);
153 }
154 /** @} */
155 
156 /**
157  * @{
158  * @name Mock functions of the interface declared in I2CInterface.h
159  */
160 /**
161  * @brief      Mock function for the read single register function.
162  *
163  * @param[in]  address          The I2C address
164  * @param[in]  registerAddress  The I2C register address
165  * @param      buffer           Buffer for the register data
166  *
167  * @return     Returns true.
168  */
169 static bool readRegister(uint8_t address, uint8_t registerAddress, uint8_t* buffer)
170 {
171     // get next expectation
172     Expectation_t* expectation = (Expectation_t*) Mock_GetNextExpecation(mock);
173     if(expectation == NULL) return false;
174 
175     // check whether all expected conditions are met
176     Mock_Check(mock, (expectation->type == READ_REGISTER),"%s:%d:MOCK Unexpected call type to be
         0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->type, READ_REGISTER);
177     Mock_Check(mock, (expectation->address == address),"%s:%d:MOCK Expected I2C address to be
         0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->address, address);
```

```
178     Mock_Check(mock, (expectation->registerAddress == registerAddress),"%s:%d:MOCK Expected
         registerAddress to be 0x%02X but was 0x%02X", __FILE__, __LINE__,
         expectation->registerAddress, registerAddress);
179
180     // fake the function logic
181     *buffer = *(expectation->data);
182
183     return true;
184 }
185
186 /**
187  * @brief      Mock function of the read multiple registers function.
188  *
189  * @param[in]  address            The I2C address
190  * @param[in]  startRegister      The start register address
191  * @param[in]  numberOfRegisters  The number of registers to be read
192  * @param      buffer             Buffer for the read data
193  *
194  * @return     Returns the number of read registers.
195  */
196 static uint8_t readMultipleRegisters(uint8_t address, uint8_t startRegister, uint8_t
         numberOfRegisters, uint8_t* buffer)
197 {
198     // get next expectation
199     Expectation_t* expectation = (Expectation_t*) Mock_GetNextExpecation(mock);
200     if(expectation == NULL) return 0;
201
202     // check whether all expected conditions are met
203     Mock_Check(mock, (expectation->type == READ_MULTIPLE_REGISTERS),"%s:%d:MOCK Unexpected call
         type to be 0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->type,
         READ_MULTIPLE_REGISTERS);
204     Mock_Check(mock, (expectation->address == address),"%s:%d:MOCK Expected I2C address to be
         0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->address, address);
205     Mock_Check(mock, (expectation->registerAddress == startRegister),"%s:%d:MOCK Expected
         startRegister to be 0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->registerAddress,
         startRegister);
206     Mock_Check(mock, (expectation->numberOfRegisters == numberOfRegisters),"%s:%d:MOCK Expected
         numberOfRegisters to be %d but was %d", __FILE__, __LINE__, expectation->numberOfRegisters,
         numberOfRegisters);
207
208     // fake the function logic
209     memcpy(buffer, expectation->data, numberOfRegisters);
210
211     return numberOfRegisters;
212 }
213
214 /**
215  * @brief      Mock function of the write single register function.
216  *
217  * @param[in]  address          The I2C address
218  * @param[in]  registerAddress  The I2C register address
219  * @param[in]  value            The value to be written to register
220  *
221  * @return     Returns true.
222  */
223 static bool writeRegister(uint8_t address, uint8_t registerAddress, uint8_t value)
224 {
225     // get next expectation
226     Expectation_t* expectation = (Expectation_t*) Mock_GetNextExpecation(mock);
227     if(expectation == NULL) return false;
228
229     // check whether all expected conditions are met
230     Mock_Check(mock, (expectation->type == WRITE_REGISTER),"%s:%d:MOCK Unexpected call type to be
         0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->type, WRITE_REGISTER);
231     Mock_Check(mock, (expectation->address == address),"%s:%d:MOCK Expected I2C address to be
         0x%02X but was 0x%02X", __FILE__, __LINE__, expectation->address, address);
232     Mock_Check(mock, (expectation->registerAddress == registerAddress),"%s:%d:MOCK Expected
         registerAddress to be 0x%02X but was 0x%02X", __FILE__, __LINE__,
         expectation->registerAddress, registerAddress);
```

```
233     Mock_Check(mock, (*(expectation->data) == value),"%s:%d:MOCK Expected value to be 0x%02X but
         was 0x%02X", __FILE__, __LINE__, *(expectation->data), value);
234
235     // fake the function logic
236     return true;
237 }
238 /** @} */
```

Listing A.16: `I2CMock.c`

```
1  /**
2   * @file       BNO055DriverTest.c
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the unit tests for the BNO055Driver-module.
7   */
8
9  #include "BNO055Driver.h"
10 #include "unity_fixture.h"
11
12 #include "I2CInterface.h"
13 #include "I2CMock.h"
14
15 #include <stdint.h>
16 #include <stdbool.h>
17
18 extern BNO055_ConfigData_t bno055_configData[13];
19
20 /**
21  * @{
22  * @name BNO055Driver test group
23  */
24 TEST_GROUP(BNO055DriverTest);
25
26 TEST_SETUP(BNO055DriverTest)
27 {
28     I2CMock_Create(20);
29     BNO055_Create(I2CMock_GetInterface());
30 }
31
32 TEST_TEAR_DOWN(BNO055DriverTest)
33 {
34     BNO055_Destroy();
35     I2CMock_Destroy();
36 }
37 /** @} */
38
39 /**
40  * @{
41  * @name BNO055Driver unit tests
42  */
43 TEST(BNO055DriverTest, ActivateConfigMode)
44 {
45     // arrange
46     uint8_t data = 0x00;
47     I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3D, &data, 1);
48
49     // act
50     BNO055_ActivateConfigMode();
51
52     // assert
53     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
54 }
55
56 TEST(BNO055DriverTest, ActivateAMGMode)
57 {
58     // arrange
```

```
59      uint8_t dataOprModeReg = 0x07;
60      I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3D, &dataOprModeReg, 1);
61
62      // act
63      bool result = BNO055_ActivateAMGMode();
64
65      // assert
66      TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
67      TEST_ASSERT(result);
68  }
69
70  TEST(BNO055DriverTest, ExecuteSuccessfulSelfTest)
71  {
72      // arrange
73      uint8_t dataOprModeReg = 0x00;
74      I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3D, &dataOprModeReg, 1);
75      uint8_t dataSysTrigReg = 0x01;
76      I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3F, &dataSysTrigReg, 1);
77      uint8_t dataSysStatusReg = BNO_EXECUTING_SELFTEST;
78      I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x39, &dataSysStatusReg, 1);
79      uint8_t dataSysErrorReg = BNO_NO_ERROR;
80      I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x3A, &dataSysErrorReg, 1);
81
82      // act
83      bool result = BNO055_ExecuteSelfTest();
84
85      // assert
86      TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
87      TEST_ASSERT(result);
88  }
89
90  TEST(BNO055DriverTest, ExecuteFailingSelfTest)
91  {
92      // arrange
93      uint8_t dataOprModeReg = 0x00;
94      I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3D, &dataOprModeReg, 1);
95      uint8_t dataSysTrigReg = 0x01;
96      I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3F, &dataSysTrigReg, 1);
97      uint8_t dataSysStatusReg = BNO_EXECUTING_SELFTEST;
98      I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x39, &dataSysStatusReg, 1);
99      uint8_t dataSysErrorReg = BNO_SELF_TEST_FAILED;
100     I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x3A, &dataSysErrorReg, 1);
101
102     // act
103     bool result = BNO055_ExecuteSelfTest();
104
105     // assert
106     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
107     TEST_ASSERT_FALSE(result);
108 }
109
110 TEST(BNO055DriverTest, SystemReset)
111 {
112     // arrange
113     uint8_t dataSysTrigReg = 0b00010000;
114     I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3F, &dataSysTrigReg, 1);
115
116     // act
117     bool result = BNO055_Reset();
118
119     // assert
120     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
121     TEST_ASSERT(result);
122 }
123
124 TEST(BNO055DriverTest, GetSystemStatusReturnsCorrectStatus)
125 {
126     // arrange
127     uint8_t dataSysStatusReg = BNO_SYSTEM_INITIALIZATION;
```

```
128     I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x39, &dataSysStatusReg, 1);
129
130     // act
131     BNO055_SystemStatus_t status = BNO055_GetSystemStatus();
132
133     // assert
134     TEST_ASSERT_EQUAL_HEX8(dataSysStatusReg, status);
135     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
136 }
137
138 TEST(BNO055DriverTest, GetSystemErrorReturnsCorrectError)
139 {
140     // arrange
141     uint8_t dataSysErrorReg = BNO_SYSTEM_INITIALIZATION_ERROR;
142     I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x3A, &dataSysErrorReg, 1);
143
144     // act
145     BNO055_ErrorCode_t error = BNO055_GetSystemError();
146
147     // assert
148     TEST_ASSERT_EQUAL_HEX8(dataSysErrorReg, error);
149     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
150 }
151
152 TEST(BNO055DriverTest, SensorIsConfiguredCorrectly)
153 {
154     // arrange
155     uint8_t dataOprModeReg = 0x00;
156     I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, 0x3D, &dataOprModeReg, 1);
157     uint8_t configData[sizeof(bno055_configData)];
158     uint8_t i;
159     for(i = 0; i < 13; i++)
160     {
161         configData[i] = bno055_configData[i].data;
162         I2CMock_AddExpectation(WRITE_REGISTER, BNO055_I2C_ADDRESS, bno055_configData[i].address,
     &configData[i], 1);
163     }
164
165     // act
166     bool result = BNO055_ConfigureSensor();
167
168     // assert
169     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
170     TEST_ASSERT(result);
171 }
172
173 TEST(BNO055DriverTest, GetAccelerometerDataReadsDataCorrectly)
174 {
175     // arrange
176     uint16_t x, y, z;
177     uint8_t accDataRegs[] = {0xAA, 0x11, 0xBB, 0x22, 0xCC, 0x33};
178     I2CMock_AddExpectation(READ_MULTIPLE_REGISTERS, BNO055_I2C_ADDRESS, 0x08, accDataRegs, 6);
179
180     // act
181     bool result = BNO055_GetAccerlometerData(&x, &y, &z);
182
183     // assert
184     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
185     TEST_ASSERT_EQUAL_HEX16(0x11AA, x);
186     TEST_ASSERT_EQUAL_HEX16(0x22BB, y);
187     TEST_ASSERT_EQUAL_HEX16(0x33CC, z);
188     TEST_ASSERT(result);
189 }
190
191 TEST(BNO055DriverTest, GetMagnetometerDataReadsDataCorrectly)
192 {
193     // arrange
194     uint16_t x, y, z;
195     uint8_t magDataRegs[] = {0xAA, 0x11, 0xBB, 0x22, 0xCC, 0x33};
```

```
196     I2CMock_AddExpectation(READ_MULTIPLE_REGISTERS, BNO055_I2C_ADDRESS, 0x0E, magDataRegs, 6);
197
198     // act
199     bool result = BNO055_GetMagnetometerData(&x, &y, &z);
200
201     // assert
202     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
203     TEST_ASSERT_EQUAL_HEX16(0x11AA, x);
204     TEST_ASSERT_EQUAL_HEX16(0x22BB, y);
205     TEST_ASSERT_EQUAL_HEX16(0x33CC, z);
206     TEST_ASSERT(result);
207 }
208
209 TEST(BNO055DriverTest, GetGyrometerDataReadsDataCorrectly)
210 {
211     // arrange
212     uint16_t x, y, z;
213     uint8_t gyroDataRegs[] = {0xAA, 0x11, 0xBB, 0x22, 0xCC, 0x33};
214     I2CMock_AddExpectation(READ_MULTIPLE_REGISTERS, BNO055_I2C_ADDRESS, 0x14, gyroDataRegs, 6);
215
216     // act
217     bool result = BNO055_GetGyrometerData(&x, &y, &z);
218
219     // assert
220     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
221     TEST_ASSERT_EQUAL_HEX16(0x11AA, x);
222     TEST_ASSERT_EQUAL_HEX16(0x22BB, y);
223     TEST_ASSERT_EQUAL_HEX16(0x33CC, z);
224     TEST_ASSERT(result);
225 }
226
227 TEST(BNO055DriverTest, GetInterruptStatusReturnsCorrectInterruptSource)
228 {
229     // arrange
230     BNO_Interrupt_t intStatus[9];
231     uint8_t intStatusReg[] = {0, 0b00000001, 0b00000010, 0b00000100, 0b00001000, 0b00010000,
        0b00100000, 0b01000000, 0b10000000};
232     I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x37, &intStatusReg[0], 1);
233     uint8_t i;
234     for(i = 0; i <= 7; i++)
235     {
236         I2CMock_AddExpectation(READ_REGISTER, BNO055_I2C_ADDRESS, 0x37, &intStatusReg[i+1], 1);
237     }
238
239     // act
240     uint8_t j;
241     for(j = 0; j < 9; j++)
242     {
243         intStatus[j] = BNO055_GetInterruptStatus();
244     }
245
246     // assert
247     TEST_ASSERT_MESSAGE(I2CMock_Verify(), I2CMock_Report());
248     TEST_ASSERT_EQUAL_HEX8(BNO_NO_INTERRUPT_OCCURED, intStatus[0]);
249     TEST_ASSERT_EQUAL_HEX8(BNO_ACC_BSX_DRDY, intStatus[1]);
250     TEST_ASSERT_EQUAL_HEX8(BNO_MAG_DRDY, intStatus[2]);
251     TEST_ASSERT_EQUAL_HEX8(BNO_GYRO_AM, intStatus[3]);
252     TEST_ASSERT_EQUAL_HEX8(BNO_GYR_HIGH_RATE, intStatus[4]);
253     TEST_ASSERT_EQUAL_HEX8(BNO_GYR_DRDY, intStatus[5]);
254     TEST_ASSERT_EQUAL_HEX8(BNO_ACC_HIGH_G, intStatus[6]);
255     TEST_ASSERT_EQUAL_HEX8(BNO_ACC_AM, intStatus[7]);
256     TEST_ASSERT_EQUAL_HEX8(BNO_ACC_NM, intStatus[8]);
257 }
258 /** @} */
```

Listing A.17: BNO055DriverTest.c

A.5 Quellcode zu Kapitel 4.3.3

```
/**
 * @file       HAL_GPIO.h
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file declares the public interface for the Hardware Abstraction
 * Layer (HAL) GPIO.
 */

#ifndef HAL_GPIO_H
#define HAL_GPIO_H

#include <stdint.h>

/**
 * @brief Enum defining the possible IO states.
 */
typedef enum
{
    HAL_GPIO_LOW = 0x00,                                        ///< GPIO low state
    HAL_GPIO_HIGH = 0x01                                        ///< GPIO high state
} HAL_GPIO_State_t;

/**
 * @{
 * @name GPIO-bank base address defines
 */
#define GPIO_BANK_A_BASE_ADDR           ((uint32_t*)0x48000000)     ///< GPIO-bank A base address
#define GPIO_BANK_B_BASE_ADDR           ((uint32_t*)0x48000400)     ///< GPIO-bank B base address
#define GPIO_BANK_C_BASE_ADDR           ((uint32_t*)0x48000800)     ///< GPIO-bank C base address
#define GPIO_BANK_D_BASE_ADDR           ((uint32_t*)0x48000C00)     ///< GPIO-bank D base address
#define GPIO_BANK_E_BASE_ADDR           ((uint32_t*)0x48001000)     ///< GPIO-bank E base address
#define GPIO_BANK_F_BASE_ADDR           ((uint32_t*)0x48001400)     ///< GPIO-bank F base address
#define GPIO_BANK_G_BASE_ADDR           ((uint32_t*)0x48001800)     ///< GPIO-bank G base address
#define GPIO_BANK_H_BASE_ADDR           ((uint32_t*)0x48001C00)     ///< GPIO-bank H base address
/** @} */

/**
 * @{
 * @name GPIO mode configuration functions
 */
void HAL_GPIO_ConfigureInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
void HAL_GPIO_ConfigureOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
/** @} */

/**
 * @{
 * @name GPIO read status functions
 */
HAL_GPIO_State_t HAL_GPIO_ReadInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
HAL_GPIO_State_t HAL_GPIO_ReadOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
/** @} */

/**
 * @{
 * @name GPIO state control functions
 */
void HAL_GPIO_SetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
void HAL_GPIO_ResetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber);
/** @} */

#endif /* HAL_GPIO_H */
```

Listing A.18: HAL_GPIO.h

```c
/**
 * @file       HAL_GPIO.c
 * @author     Jan Heimann
 * @date       2020
 *
 * @brief      This file contains the implementation for the Hardware Abstraction
 * Layer (HAL) GPIO.
 */

#include "HAL_GPIO.h"

/**
 * @{
 * @name Defines of the register offsets
 */
#define REG_MODER_OFFSET                (0x00)
#define REG_OTYPER_OFFSET               (0x04)
#define REG_OSPEED_OFFSET               (0x08)
#define REG_PUPDR_OFFSET                (0x0C)
#define REG_IDR_OFFSET                  (0x10)
#define REG_ODR_OFFSET                  (0x14)
/** @} */

static void modifyRegister(volatile uint32_t* registerAddress, uint32_t value, uint8_t gpioNumber);
static uint32_t shiftForGPIONumber(volatile uint32_t value, uint8_t gpioNumber);
static HAL_GPIO_State_t readState(volatile uint32_t* registerAddress, uint8_t gpioNumber);

/**
 * @{
 * @name GPIO mode configuration functions
 */
/**
 * @brief      Configures the specified IO as Input
 *
 * @param      gpioBankAddress  The GPIO bank address
 * @param[in]  gpioNumber       The GPIO number
 */
void HAL_GPIO_ConfigureInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
{
    // set GPIO mode to input
    modifyRegister(gpioBankAddress + REG_MODER_OFFSET, 0, gpioNumber);

    // configure pull-up
    modifyRegister(gpioBankAddress + REG_PUPDR_OFFSET, 1, gpioNumber);
}

/**
 * @brief      Configures the specified IO as Output.
 *
 * @param      gpioBankAddress  The GPIO bank address
 * @param[in]  gpioNumber       The GPIO number
 */
void HAL_GPIO_ConfigureOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
{
    // set GPIO mode to output
    modifyRegister(gpioBankAddress + REG_MODER_OFFSET, 1, gpioNumber);

    // set output type to push-pull configuration
    *(gpioBankAddress + REG_OTYPER_OFFSET) &= ~(1 << gpioNumber);

    // set output speed
    modifyRegister(gpioBankAddress + REG_OSPEED_OFFSET, 1, gpioNumber);

    // configure pull-up-down register
    modifyRegister(gpioBankAddress + REG_PUPDR_OFFSET, 0, gpioNumber);
}
/** @} */

```

```
69  /**
70   * @{
71   * @name GPIO read status functions
72   */
73  /**
74   * @brief      Reads the state of the specified input.
75   *
76   * @param      gpioBankAddress  The GPIO bank address
77   * @param[in]  gpioNumber       The GPIO number
78   *
79   * @return     Returns the inputs state.
80   */
81  HAL_GPIO_State_t HAL_GPIO_ReadInput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
82  {
83      return readState(gpioBankAddress + REG_IDR_OFFSET, gpioNumber);
84  }
85
86  /**
87   * @brief      Reads the state of the specified output.
88   *
89   * @param      gpioBankAddress  The GPIO bank address
90   * @param[in]  gpioNumber       The GPIO number
91   *
92   * @return     Returns the outputs state
93   */
94  HAL_GPIO_State_t HAL_GPIO_ReadOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
95  {
96      return readState(gpioBankAddress + REG_ODR_OFFSET, gpioNumber);
97  }
98  /** @} */
99
100 /**
101  * @{
102  * @name GPIO state control functions
103  */
104 /**
105  * @brief      Sets the specified output.
106  *
107  * @param      gpioBankAddress  The GPIO bank address
108  * @param[in]  gpioNumber       The GPIO number
109  */
110 void HAL_GPIO_SetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
111 {
112     *(gpioBankAddress + REG_ODR_OFFSET) |= 1 << gpioNumber;
113 }
114
115 /**
116  * @brief      Resets the specified output.
117  *
118  * @param      gpioBankAddress  The GPIO bank address
119  * @param[in]  gpioNumber       The GPIO number
120  */
121 void HAL_GPIO_ResetOutput(volatile uint32_t* gpioBankAddress, uint8_t gpioNumber)
122 {
123     *(gpioBankAddress + REG_ODR_OFFSET) &= ~(1 << gpioNumber);
124 }
125 /** @} */
126
127 /**
128  * @{
129  * @name Private functions
130  */
131 /**
132  * @brief      Modifies the specified register.
133  *
134  * @param      registerAddress  The register address
135  * @param[in]  value            The value
136  * @param[in]  gpioNumber       The GPIO number
137  */
```

```
138 static void modifyRegister(volatile uint32_t* registerAddress, uint32_t value, uint8_t gpioNumber)
139 {
140     uint32_t mask = shiftForGPIONumber(3, gpioNumber);
141 
142     *registerAddress &= ~mask;
143     *registerAddress |= shiftForGPIONumber(value, gpioNumber) & mask;
144 }
145 
146 /**
147  * @brief      Shifts a unsigned 32-bit interger value according to the gpioNumber.
148  *
149  * @param[in]  value       The value
150  * @param[in]  gpioNumber  The GPIO number
151  *
152  * @return     Returns the shifted value.
153  */
154 static uint32_t shiftForGPIONumber(volatile uint32_t value, uint8_t gpioNumber)
155 {
156     uint32_t shiftedValue = value << (gpioNumber * 2);
157     return shiftedValue;
158 }
159 
160 /**
161  * @brief      Reads the state of a GPIO from the according GPIO control register.
162  *
163  * @param      registerAddress  The register address
164  * @param[in]  gpioNumber       The GPIO number
165  *
166  * @return     The GPIO state.
167  */
168 static HAL_GPIO_State_t readState(volatile uint32_t* registerAddress, uint8_t gpioNumber)
169 {
170     uint32_t mask = 1 << gpioNumber;
171     uint32_t state = *(registerAddress) & mask;
172 
173     HAL_GPIO_State_t result;
174     if(state == 0)
175     {
176         result = HAL_GPIO_LOW;
177     }
178     else
179     {
180         result = HAL_GPIO_HIGH;
181     }
182 
183     return result;
184 }
185 /** @} */
```

Listing A.19: HAL_GPIO.c

```
1  /**
2   * @file       HAL_GPIOTest.c
3   * @author     Jan Heimann
4   * @date       2020
5   *
6   * @brief      This file contains the unit tests for the HAL_GPIO-module.
7   */
8  
9  #include "HAL_GPIO.h"
10 #include "unity_fixture.h"
11 
12 #include <stdint.h>
13 
14 volatile static uint32_t gpioRegisters[21];
15 volatile static uint32_t* gpioBankAddress;
16 
17 /**
```

```
18  * @{
19  * @name HAL_GPIO test group
20  */
21 TEST_GROUP(HAL_GPIOTest);
22
23 TEST_SETUP(HAL_GPIOTest)
24 {
25     gpioRegisters[0] = 0;
26     gpioRegisters[4] = 0;
27     gpioRegisters[8] = 0;
28     gpioRegisters[12] = 0;
29     gpioRegisters[16] = 0;
30     gpioRegisters[20] = 0;
31
32     gpioBankAddress = &gpioRegisters[0];
33 }
34
35 TEST_TEAR_DOWN(HAL_GPIOTest)
36 {
37
38 }
39 /** @} */
40
41 /**
42  * @{
43  * @name HAL_GPIO unit tests
44  */
45 TEST(HAL_GPIOTest, SettingOneInputMakesCorrectChangesToModeRegister)
46 {
47     // arrange
48     gpioRegisters[0] = 0x5555;
49
50     // act
51     HAL_GPIO_ConfigureInput(gpioBankAddress, 0);
52
53     // assert
54     TEST_ASSERT_EQUAL_HEX32(0x5554, gpioRegisters[0]);
55 }
56
57 TEST(HAL_GPIOTest, SettingTwoInputsMakesCorrectChangesToModeRegister)
58 {
59     // arrange
60     gpioRegisters[0] = 0x5555;
61
62     // act
63     HAL_GPIO_ConfigureInput(gpioBankAddress, 0);
64     HAL_GPIO_ConfigureInput(gpioBankAddress, 1);
65
66     // assert
67     TEST_ASSERT_EQUAL_HEX32(0x5550, gpioRegisters[0]);
68 }
69
70 TEST(HAL_GPIOTest, SettingOneOutputMakesCorrectChangesToModeRegister)
71 {
72     // arrange
73     gpioRegisters[0] = 0x4444;
74
75     // act
76     HAL_GPIO_ConfigureOutput(gpioBankAddress, 0);
77
78     // assert
79     TEST_ASSERT_EQUAL_HEX32(0x4445, gpioRegisters[0]);
80 }
81
82 TEST(HAL_GPIOTest, SettingTwoOutputMakesCorrectChangesToModeRegister)
83 {
84     // arrange
85     gpioRegisters[0] = 0x4444;
86
```

```
87      // act
88      HAL_GPIO_ConfigureOutput(gpioBankAddress, 0);
89      HAL_GPIO_ConfigureOutput(gpioBankAddress, 2);
90
91      // assert
92      TEST_ASSERT_EQUAL_HEX32(0x4455, gpioRegisters[0]);
93  }
94
95  TEST(HAL_GPIOTest, ConfigureOutputModifiesOutputTypeRegisterCorrectly)
96  {
97      // arrange
98      gpioRegisters[4] = 0xFF;
99
100     // act
101     HAL_GPIO_ConfigureOutput(gpioBankAddress, 0);
102
103     // assert
104     TEST_ASSERT_EQUAL_HEX32(0xFE, gpioRegisters[4]);
105 }
106
107 TEST(HAL_GPIOTest, ConfigureOutputModifiesOutputSpeedRegisterCorrectly)
108 {
109     // arrange
110     gpioRegisters[8] = 0xFF;
111
112     // act
113     HAL_GPIO_ConfigureOutput(gpioBankAddress, 0);
114
115     // assert
116     TEST_ASSERT_EQUAL_HEX32(0xFD, gpioRegisters[8]);
117 }
118
119 TEST(HAL_GPIOTest, ConfigureOutputModifiesPullUpDownRegisterCorrectly)
120 {
121     // arrange
122     gpioRegisters[12] = 0xFF;
123
124     // act
125     HAL_GPIO_ConfigureOutput(gpioBankAddress, 0);
126
127     // assert
128     TEST_ASSERT_EQUAL_HEX32(0xFC, gpioRegisters[12]);
129 }
130
131 TEST(HAL_GPIOTest, MixedInputAndOutputConfigurationChangesModeRegisterCorrectly)
132 {
133     // act
134     HAL_GPIO_ConfigureOutput(gpioBankAddress, 0);
135     HAL_GPIO_ConfigureInput(gpioBankAddress, 2);
136     HAL_GPIO_ConfigureOutput(gpioBankAddress, 3);
137
138     // assert
139     TEST_ASSERT_EQUAL_HEX32(0x0041, gpioRegisters[0]);
140 }
141
142 TEST(HAL_GPIOTest, ConfigureInputDoesNotChangeTypeAndSpeedRegisters)
143 {
144     // arrange
145     gpioRegisters[4] = 0xAAAAAAAA;
146     gpioRegisters[8] = 0xAAAAAAAA;
147
148     // act
149     HAL_GPIO_ConfigureInput(gpioBankAddress, 0);
150
151     // assert
152     TEST_ASSERT_EQUAL_HEX32(0xAAAAAAAA, gpioRegisters[4]);
153     TEST_ASSERT_EQUAL_HEX32(0xAAAAAAAA, gpioRegisters[8]);
154 }
155
```

```
156 TEST(HAL_GPIOTest, ConfigureInputSetsPullUpDownRegisterToPullUp)
157 {
158     // arrange
159
160     // act
161     HAL_GPIO_ConfigureInput(gpioBankAddress, 0);
162
163     // assert
164     TEST_ASSERT_EQUAL_HEX32(0x01, gpioRegisters[12]);
165 }
166
167 TEST(HAL_GPIOTest, ReadInputReturnsHighWhenInputIsHigh)
168 {
169     // arrange
170     gpioRegisters[16] = 0x00000001;
171
172     // act
173     HAL_GPIO_State_t state = HAL_GPIO_ReadInput(gpioBankAddress, 0);
174
175     // assert
176     TEST_ASSERT(state == HAL_GPIO_HIGH);
177 }
178
179 TEST(HAL_GPIOTest, ReadInputReturnsLowWhenInputIsLow)
180 {
181     // arrange
182     gpioRegisters[16] = 0xFFFFFFFE;
183
184     // act
185     HAL_GPIO_State_t state = HAL_GPIO_ReadInput(gpioBankAddress, 0);
186
187     // assert
188     TEST_ASSERT(state == HAL_GPIO_LOW);
189 }
190
191 TEST(HAL_GPIOTest, MultipleReadInputCallsReturnTheCorrectState)
192 {
193     // arrange
194     gpioRegisters[16] = 0xD;
195     HAL_GPIO_State_t state[4];
196
197     // act
198     state[0] = HAL_GPIO_ReadInput(gpioBankAddress, 0);
199     state[1] = HAL_GPIO_ReadInput(gpioBankAddress, 1);
200     state[2] = HAL_GPIO_ReadInput(gpioBankAddress, 2);
201     state[3] = HAL_GPIO_ReadInput(gpioBankAddress, 3);
202
203     // assert
204     TEST_ASSERT(state[0] == HAL_GPIO_HIGH);
205     TEST_ASSERT(state[1] == HAL_GPIO_LOW);
206     TEST_ASSERT(state[2] == HAL_GPIO_HIGH);
207     TEST_ASSERT(state[3] == HAL_GPIO_HIGH);
208 }
209
210 TEST(HAL_GPIOTest, ReadOutputReturnsHighWhenOutputIsHigh)
211 {
212     // arrange
213     gpioRegisters[20] = 0x01;
214
215     // act
216     HAL_GPIO_State_t state = HAL_GPIO_ReadOutput(gpioBankAddress, 0);
217
218     // assert
219     TEST_ASSERT(HAL_GPIO_HIGH == state);
220 }
221
222 TEST(HAL_GPIOTest, ReadOutputReturnsLowWhenOutputIsLow)
223 {
224     // arrange
```

```
225     gpioRegisters[20] = 0xFFFFFFFE;
226
227     // act
228     HAL_GPIO_State_t state = HAL_GPIO_ReadOutput(gpioBankAddress, 0);
229
230     // assert
231     TEST_ASSERT(HAL_GPIO_LOW == state);
232 }
233
234 TEST(HAL_GPIOTest, MultipleReadOutputCallsReturnTheCorrectState)
235 {
236     // arrange
237     gpioRegisters[20] = 0xD;
238     HAL_GPIO_State_t state[4];
239
240     // act
241     state[0] = HAL_GPIO_ReadOutput(gpioBankAddress, 0);
242     state[1] = HAL_GPIO_ReadOutput(gpioBankAddress, 1);
243     state[2] = HAL_GPIO_ReadOutput(gpioBankAddress, 2);
244     state[3] = HAL_GPIO_ReadOutput(gpioBankAddress, 3);
245
246     // assert
247     TEST_ASSERT(state[0] == HAL_GPIO_HIGH);
248     TEST_ASSERT(state[1] == HAL_GPIO_LOW);
249     TEST_ASSERT(state[2] == HAL_GPIO_HIGH);
250     TEST_ASSERT(state[3] == HAL_GPIO_HIGH);
251 }
252
253 TEST(HAL_GPIOTest, SetOutputModifiesODRRegisterCorrectly)
254 {
255     // arrange
256     gpioRegisters[20] = 0x00;
257
258     // act
259     HAL_GPIO_SetOutput(gpioBankAddress, 0);
260
261     // assert
262     TEST_ASSERT_EQUAL_HEX32(0x01, gpioRegisters[20]);
263 }
264
265 TEST(HAL_GPIOTest, MultipleSetOutputCallsModifyODRRegisterCorrectly)
266 {
267     // arrange
268     gpioRegisters[20] = 0x00;
269
270     // act
271     HAL_GPIO_SetOutput(gpioBankAddress, 0);
272     HAL_GPIO_SetOutput(gpioBankAddress, 1);
273     HAL_GPIO_SetOutput(gpioBankAddress, 2);
274     HAL_GPIO_SetOutput(gpioBankAddress, 3);
275
276     // assert
277     TEST_ASSERT_EQUAL_HEX32(0x0F, gpioRegisters[20]);
278 }
279
280 TEST(HAL_GPIOTest, ResetOutputModifiesODRRegisterCorrectly)
281 {
282     // arrange
283     gpioRegisters[20] = 0xFF;
284
285     // act
286     HAL_GPIO_ResetOutput(gpioBankAddress, 0);
287
288     // assert
289     TEST_ASSERT_EQUAL_HEX32(0xFE, gpioRegisters[20]);
290 }
291
292 TEST(HAL_GPIOTest, MultipleResetOutputCallsModifyODRRegisterCorrectly)
293 {
```

```
294     // arrange
295     gpioRegisters[20] = 0xFF;
296
297     // act
298     HAL_GPIO_ResetOutput(gpioBankAddress, 0);
299     HAL_GPIO_ResetOutput(gpioBankAddress, 1);
300     HAL_GPIO_ResetOutput(gpioBankAddress, 2);
301     HAL_GPIO_ResetOutput(gpioBankAddress, 3);
302
303     // assert
304     TEST_ASSERT_EQUAL_HEX32(0xF0, gpioRegisters[20]);
305 }
306
307 TEST(HAL_GPIOTest, MixedResetAndSetOutputCallsModifyODRRegisterCorrectly)
308 {
309     // arrage
310     gpioRegisters[20] = 0x00000000;
311
312     // act
313     HAL_GPIO_ResetOutput(gpioBankAddress, 0);
314     HAL_GPIO_SetOutput(gpioBankAddress, 1);
315     HAL_GPIO_ResetOutput(gpioBankAddress, 2);
316     HAL_GPIO_SetOutput(gpioBankAddress, 3);
317
318     // assert
319     TEST_ASSERT_EQUAL_HEX32(0x0A, gpioRegisters[20]);
320 }
321 /** @} */
```

Listing A.20: `HAL_GPIOTest.c`

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg	22.12.2020	
Ort	Datum	Unterschrift im Original