DIPLOMARBEIT

Automatisierung von Hardware/Software Integration Tests auf Mikrocontroller Zielumgebungen

ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thilo Sauter
Univ.Ass. Dipl.-Ing. Stefan Christian Wilker, B.Eng.

am

Institut für Computertechnik (E384)
der Technischen Universität Wien

durch

Florian Muttenthaler BSc.
Matr.Nr. 01325603

Wien, am 10. Juni 2020 _____

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 10. Juni 2020 _____

[Florian Muttenthaler BSc.]

Kurzfassung

In der Entwicklung von funktionalen, robusten, effizienten und sicheren Embedded Systemen, werden komplexe System in Hardware und Software Komponenten gekapselt und die Komponenten unabhängig von einander entwickelt. Die Integration dieser einzelnen Komponenten zum Gesamtsystem muss verifiziert werden. Diese Arbeit beschäftigt sich mit der Analyse, Umsetzung und Evaluierung von automatisierten Hardware/Software Integration Verifiaktionmethoden auf Mikrocontroller Zielumgebungen. Dabei ist die funktionale Verifikation von Software Komponenten auf der Mikrocontroller Zielumgebung, weiterführende Software Integration Tests und weiterführende Hardware/Software Integration Tests mit zugehörigen Hardware Komponenten gemeint.

Diese Verifikationmethoden sind Teil mehrerer Prozesse des Software Process Improvement and Capability Determination (SPICE) Entwicklungsmodells, welche hier im Hinblick auf die Entwicklung von Embedded Systemen beschrieben werden. Im Speziellen wird hier ein Überblick im System Integration Prozess geschaffen, in welchem die Durchführung von Hardware/Software Integration Tests angesiedelt ist. Dabei wird die Durchführung von dezidierten Verifikationen von Software Komponenten auf den Zielumgebungen des entwickelten Systems in gewissen Safety Standards verlangt. Auf diese Anforderungen in den Standards sowie deren Ausführungen in den jeweiligen Prozessen wird in dieser Arbeit im Detail eingegangen.

Die Ausführung von Processor In the Loop (PIL) Tests am Mikrocontroller gilt als gängige Methode für die Verifikation von Software Komponenten auf einer Zielumgebung. Dabei wird auch eine Erweiterung um ein umfassendes Hardware In the Loop (HIL) Testsystem zur automatisierten Verifikation von zugehöriger Hardware Peripherie verstanden. Dieses Konzept verlangt allerdings eine weitreichende Instrumentierung, ergänzend zur produktiven Software, am Mikrocontroller und eine komplexe Synchronisierung mit dem HIL Testsystem. In dieser Arbeit werden alternative Tests am Entwicklungsrechner ausgeführt, welche einerseits mit dem HIL Testsystem interagieren und andererseits über einen Debugger mit dem Mikrocontroller kommunizieren. Dabei wird im Speziellen die Kommunikation über die Dubugging Schnittstelle analysiert.

Im Zuge eines experimentellen Aufbaus wurde eine Plattform mit einer Klassen Bibliothek erschaffen, welche die Automatisierung von Hardware/Software Integration Tests managt. Diese Plattform wurde durch die Evaluierung von Low-Level Driver (LLD) Software Komponenten auf der jeweiligen Mikrocontroller Zielumgebung geprüft. Die Effizienz der Testlaufzeit bei automatisierter Ausführung wurde mit jener der manuellen verglichen und analysiert. Ein Ausblick auf die Verwendung der Plattform für Software Komponenten und Integration Tests, sowie für Hardware/Software Integration Tests, auf einer Zielumgebung wurde geschaffen. Fehleranfällige gängige Methoden, wie das manuelle händische funktionale Verifizieren von Hardware/Software Komponenten, können abgelöst werden und das entwickelte Produkt kann einem höheren Safety Standard zugeordnet werden.

Abstract

At the development of functional, robust, efficient and secure embedded systems, complex systems are encapsulated in hardware and software components and the components are developed independently of each other. The integration of these individual components into the overall system must be verified. This thesis deals with the analysis, implementation and evaluation of automated hardware/software integration verification methods on microcontroller target environments. The functional verification of software components on the microcontroller environment, advanced software integration tests and advanced hardware/software integration tests with associated hardware components is understood.

These verification methods are part of several processes of the Software Process Improvement and Capability Determination (SPICE) development model, which are described in this work with regard to the development of embedded systems. This thesis shows an overview of the system integration process, in which the implementation of hardware/software integration tests is located. The implementation of dedicated verifications of software components on the target environment of the developed system is required in certain safety standards. This work deals in detail with these requirements and their implementation in the respective processes.

The execution of Processor In the Loop (PIL) tests on the microcontroller is a common method for the verification of software components on a target environment. An extension with a comprehensive Hardware In the Loop (HIL) test system for the automated verification of associated hardware peripherals is also sketched. This concept requires extensive instrumentation, in addition to the productive software, on the microcontroller and complex synchronization with the HIL test system. In this work, alternative tests are carried out on the development computer, which on the one hand interact with the HIL test system and on the other hand communicate with the microcontroller via a debugger. In particular, the communication via the debugging interface is analyzed.

In the course of an experimental setup, a platform with a class library was created that manages the automation of hardware/software integration tests. This platform was evaluated by verifying Low-Level Driver (LLD) software components on the respective microcontroller target environment. The efficiency of the test runtime with automated execution was compared and analyzed with that of the manual one. An outlook on the use of the platform for software components and integration tests, as well as for hardware/software integration tests on a target environment was created. Common methods that are prone to errors, such as manual functional verification of hardware/software components can be replaced and the developed product can be assigned to a higher safety standard.

Inhaltsverzeichnis

Abkürzungen

ADC	Analog Digital Conversion
ABI	Application Binary Interface
API	Application Programming Interface
ASIL	Automotive SIL
ASPICE	Automotive SPICE
AUTOSAR	AUTomotive Open System ARchitecture
BSW	Basic Software
CAN	Controller Area Network
CPU	Central Processing Unit
CS	Chip Select
DC	Direct Current
DIO	Digital Input/Output
DLL	Dynamic Link Library
DUT	Device Under Test
EABI	Embedded ABI
ECU	Electronic Control Unit
ELF	Executable and Linkable Format
eMIOS	enhanced Modular Input/Output Subsystem
EOC	End of Conversion
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPR	Floating Point Register
FPU	Floating Point Unit
GND	Ground
GPIO	General Purpose Input Output
GPR	General Purpose Register
HAL	Hardware Abstraction Layer
HIL	Hardware In the Loop
HTML	Hypertext Markup Language
IC	Integrated Circuit
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IL	Interaction Layer
IP	Intellectual Property
ISO	International Organization for Standardization
ISTO	Industry Standards and Technology Organization
I/O	Input/Output
I2C	Inter Integrated Circuit
JTAG	Joint Test Action Group

LED	Light Emitting Diode
LLD	Low-Level Driver
LVDS	Low Voltage Differential Signaling
MCAL	Microcontroller Abstraction Layer
MC/DC	Modified Condition Decision Coverage
MISRA	Motor Industry Software Reliability Association
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
NTC	Negative Temperature Coefficient
N/A	Not Available
UART	Universal Asynchronous Receiver Transmitter
uC	Microcontroller
PCB	Printed Circuit Board
PIL	Processor In the Loop
PWM	Pulse Wide Modulation
P2P	Peer to Peer
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RS	Recommended Standard
RTE	Runtime Environment
SDK	Software Development Kit
SIL	Safety Integrity Level
SIL	Software In the Loop
SPI	Serial Peripheral Interface
SPICE	Software Process Improvement and Capability Determination
SoC	System on Chip
SUT	System Under Test
TAP	Test Access Point
USB	Universal Serial Bus
WCET	Worst Case Execution Time
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Die Entwicklung einer möglichst funktionalen, robusten, effizienten und sicheren Software gewinnt in sämtlichen modernen Computersystemen mit fortschreitendem Technologiewandel immer mehr an Bedeutung. Dies ist in sämtlichen Abstraktionsebenen eines Computersystems ständig spürbar. Auch die Konzepte zur Entwicklung von Embedded Software sind im ständigen Wandel. Dabei werden sämtliche Aspekte eines solchen Entwicklungsprozesses optimiert. Die Möglichkeit zur Optimierung lässt sich auch vor allem auf die Fortschritte in der Entwicklung von Integrated Circuits (ICs) und Hardwarekomponenten in Mikrocontroller Anwendung zurückführen. Konzepte zur Verbesserung von Embedded Software orientieren sich oftmals an Konzepten verschiedenster möglicher Software Architekturen. Teil einer Embedded Software Architektur ist stets die Abstraktion der Interaktion mit der Umgebungshardware. Dabei wird zumeist modular die Wechselwirkung der unterschiedlichsten Registersets des Mikrocontrollers beschrieben. Die Erstellung von sogenannter Treiber Software hat sich als bewährt erwiesen.
Diese unterschiedlichen Treiber interagieren mit den unterschiedlichen Teilen der IC Hardware, welche in der Entwicklung von ICs in Intellectual Property (IP) Cores unterteilt sind. Diese Unterteilung basiert auf einer logischen funktionsorientierten Kapselung. Die zugehörigen Treiber entsprechen dieser Kapselung und decken zumeist den vollen möglichen Funktionsumfang des IP Cores ab. Nun werden diese Basis Treiber oftmals zur Interaktion mit externen Hardware Komponenten, zumeist in Form von zusätzlichen ICs, verwendet. Die Integration dieser Hardware Komponenten in ein Software basiertes System, ist Teil der Entwicklung einer zugehörigen Software Komponente. Die Verifikation dieser Software Komponenten und der zugehörigen Hardware basiert auf Integrationstests.

1.1 Motivation

Integrationstests bei der Entwicklung von Embedded Systems sind aufgrund von Abstraktion der Systemarchitektur in Hardware und Software Komponenten und Modulen notwendig, um die Zuverlässigkeit und Funktionsfähigkeit des Systems zu gewährleisten. Dabei spielt sich diese Integration auf unterschiedlichen Teilen des Systems ab. Die Integration zwischen dedizierten Hardware und Software Komponenten wird beispielsweise im Zuge der Software Entwicklung zu den jeweiligen System Komponenten durchgeführt. Diese Integration gehört auch ausreichend verifiziert. In Mikrocontroller Umgebungen werden für die Interaktion mit umgebender Hardware spezielle Signalschnittstellen verwendet. Die Bedienung der Signalschnittstellen wird seitens der Software über Treiber Module realisiert. Diese entwickelten Treiber müssen ebenso ausreichend

verifiziert werden, um eine weitere Integration durchführen zu können.
Bei der Verifikation von sogenannten Hardware/Software Komponenten, also Komponenten dessen Software Module zur Steuerung und Analyse von Hardware Komponenten, wie ICs, sowie zur Kommunikation mit einer peripheren Umgebung dienen, ist die Integration der Software auf einer Zielumgebung notwendig. Dabei ist nicht zwingend das zu entwickelnde Gesamtsystem gemeint, aber zumindest Teile des Systems gekapselt in funktionalen Blöcken.
Ein Gesamtsystem eines Embedded Systems wird in Abbildung 1.1 demonstriert. Eine ECU wird dabei verwendet eine Matrix von LEDs anzusteuern. Diese LED-Matrix befindet sich auf einer eigenständigen Printed Circuit Board (PCB), auf welcher LED-Matrix Manager als elektronische Bauteile für die Ansteuerung der einzelnen LEDs auf einem LED Strang zuständig sind. Die LED-Matrix Manager werden über einen Universal Asynchronous Receiver Transmitter (UART) Kanal von der ECU angesteuert. Um den Einfluss von elektromagnetischen Störungen zwischen den PCBs zu reduzieren werden die UART Frames über Peer to Peer (P2P) Controller Area Network (CAN) Verbindung übertragen [1]. Die Versorgung der LEDs erfolgt über Stromkanäle, welche mittels Buck Convertern auf der ECU eingestellt werden. Diese Buck Converter stellen jene Hardware Komponenten der System Architektur dar, welche für die beispielhafte Abstraktion von System Komponenten angeführt werden. Die zugehörige Software Komponente für die Ansteuerung der Buck Converter ist auf den Mikrocontroller der ECU realisiert. Als weitere Hardware Komponente der ECU wird ein Boost Converter für die stabile Spannungsversorgung sämtlicher Hardware Komponenten des Systems verwendet. Die Anbindung der ECU an einen Feldbus erfolgt über eine CAN Schnittstelle. Dieses System ist ein gängiges Konzept für LED-Matrix Licht System in Fahrzeugarchitekturen und wird für Anwendungen, wie das blendfreie Fernlicht verwendet.
Ein Bauteil zur Stromregelung aus dem Konzept in Abbildung 1.1, welches mittels eines Soft-

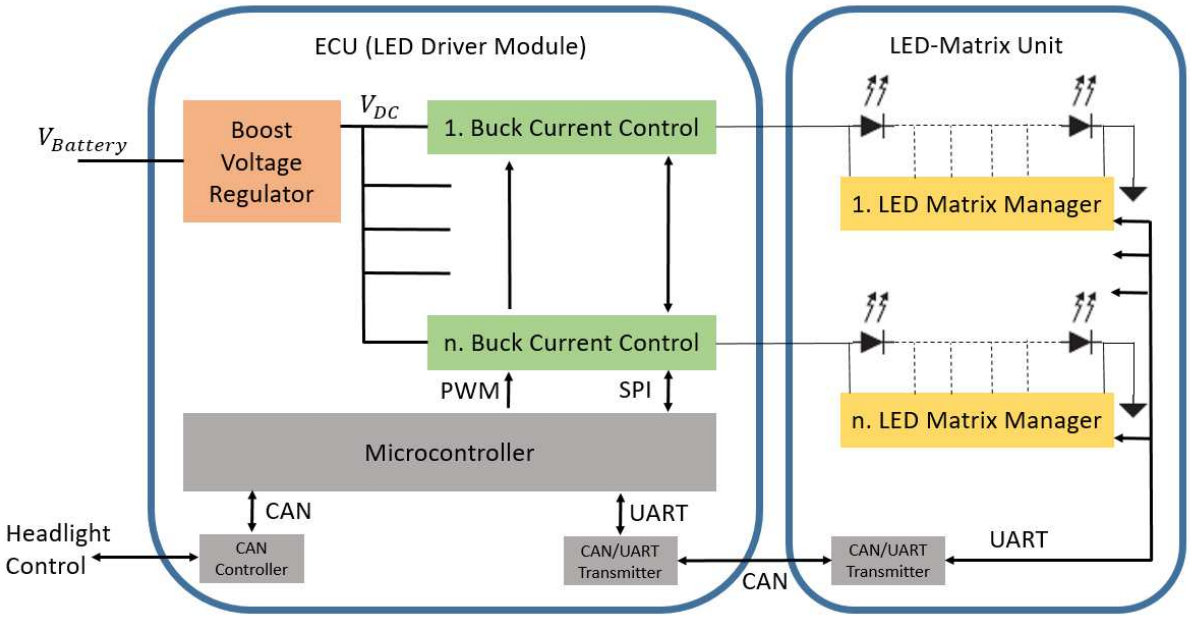

Abbildung 1.1: ECU mit Buck Converter als Stromkanäle für LED Stränge eines Lichtmoduls [1]

waretreibers angesteuert wird, stellt eine zu verifizierende Hardware/Software Komponente dar. Abbildung 1.2 demonstriert einen Buck Converter zur Stromregelung als System Komponente mit Hardware und Software Komponenten in der betrachteten System Architektur. Zur Veran-

schaulichung sind 3D Animationen der Beschaltung um einen Buck Converter IC und um den Mikrocontroller IC angeführt. Der IC mit Beschaltung um den Buck Converter entspricht dabei der Hardware Komponente. Bauteilspezifisch existiert eine Software Komponente zur Ansteuerung. Um allerdings eine Stromregelung mittels des Buck Converter IC umzusetzen, verlangt die Bauteilspezifikation eine Ansteuerung über eine serielle Serial Peripheral Interface (SPI) Schnittstelle. Damit die Software Komponente den SPI IP-Core des Mikrocontrollers verwenden kann, muss ein SPI Treiber verwendet werden. Dieser SPI Treiber ist wiederum eine eigenständige und unabhängige Software Komponente. Um die Qualität der Systemintegration einer solchen Hardware/Software Komponente zu erhöhen, ist eine automatisiert betriebene Plattform, welche Tests mit der physikalischen Zielumgebung durchführt, eine zielführende Methode.

Die Auslegung und Interpretation und damit einhergehend auch die Optimierung von Entwick-

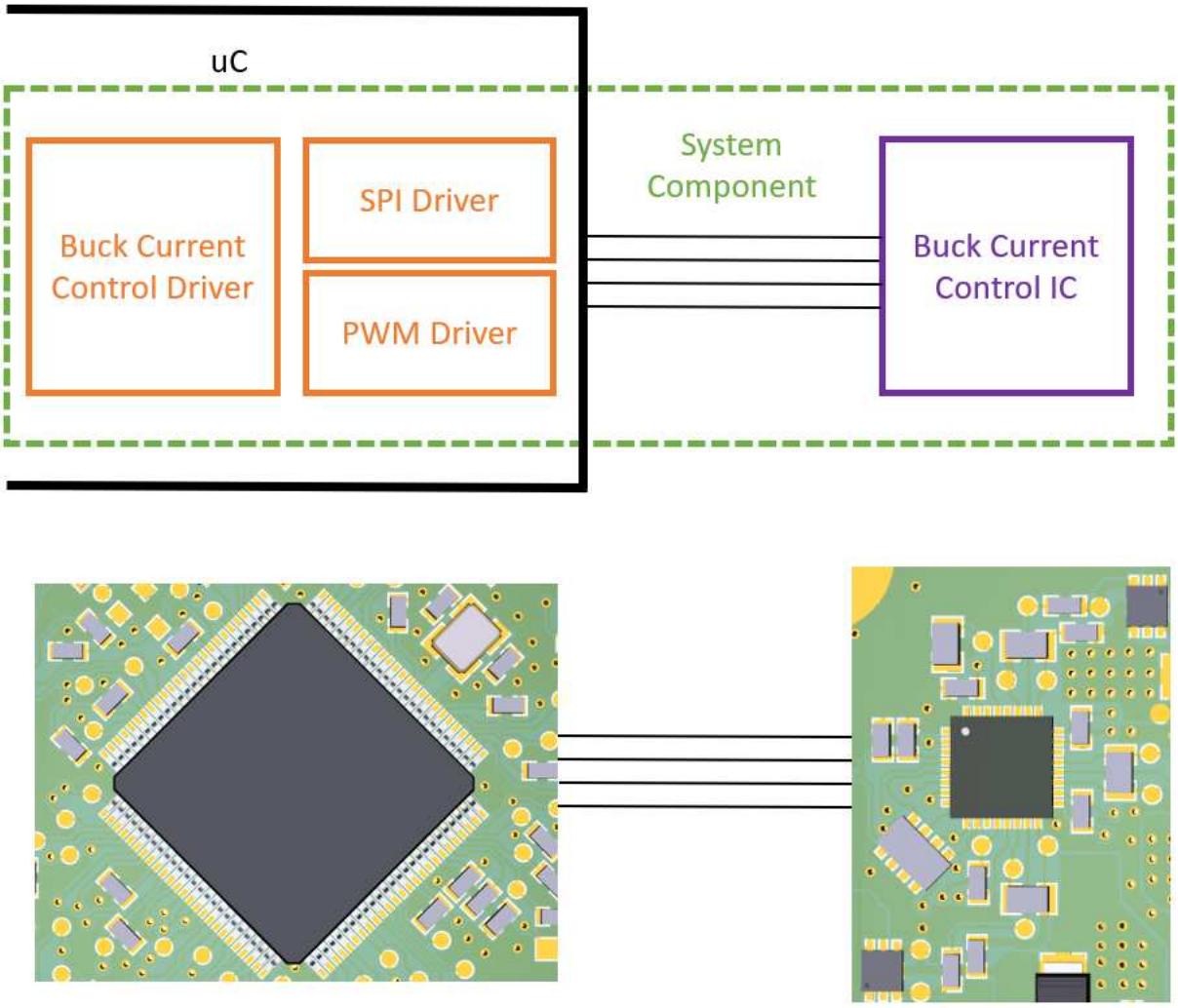

Abbildung 1.2: Buck Converter als System Komponente[1].

lungsprozessen für Embedded Systems unterscheidet sich in unterschiedlichsten Industriesparten. Dies ist vor allem durch das unterschiedliche Niveau in der Anforderung an die Entwicklung merkbar. Referenziert wird diese Anforderung durch Standards, wie z.B. der Automotive Func-

[1]3D Animationen der Bauteile sind geistiges Eigentum der ZKW Elektronik GmbH `https://zkw-group.com/`, Online: Letzter Zugriff am 08.05.2020

tional Safety Standard International Organization for Standardization (ISO) 26262 [10]. In diesem Standard wird das entwickelte System, je nach Erfüllung gewisser Kriterien, bewertet und eingestuft. Je nach Abstufung kann eine gewisse Sicherheit des Systems garantiert werden. Dies dient im Allgemeinen zur Risikoreduktion von kritischen Systemen, welche in der Automobil Industrie entwickelt und produziert werden. Um einen gewissen Safety Standard zu erreichen, muss im Allgemeinen die Entwicklungsqualität durch Verbesserung der Prozesse gesteigert werden. Diese Prozesse werden wieder durch die Bewertung eines eigenen Standards, dem Automotive Software Process Improvement and Capability Determination (ASPICE) Entwicklungsprozess Standard [9], qualifiziert und kategorisiert. Um bei den erwähnten Standards ein gewisses Level zu erreichen, sind dezidiert Integrationsprozesse mit zugehöriger Verifikation empfohlen oder gar vorgeschrieben.
Bei der Durchführung der Integrationstests von System Komponenten an deren Hardware und Software-Schnittstellen wird deren Automatisierung oftmals vernachlässigt. Die Verifikation erfolgt zumeist durch ein händisches Messen der physikalischen Signale. Bei der Entwicklung von ECUs wird dabei an eigenen Messpunkten auf der PCB mit einem Oszilloskop überprüft, ob die Signale die erwartete Form haben. Dabei werden zumeist die zugehörigen Software Komponenten über aufgelöste Symbole mittels des Debugging Interface des Mikrocontrollers manipuliert und ausgewertet. Diese gängige Methode ist allerdings äußerst fehleranfällig, zeitaufwendig und nicht nachvollziehbar für Außenstehende. Des Weiteren muss die ECU Hardware mittels Platzierung von Testpunkten auch dementsprechend entworfen sein, was wiederum Platz am PCB kostet und somit die Kosten des Produktes selbst erhöht. In Branchen wie der Automobilbranche, bei welcher ECUs in hoher Stückzahl verkauft werden, ist ein kostenoptimiertes Design notwendig, um wettbewerbsfähig zu bleiben. Der optimierte Aufwand in sämtlichen Entwicklungsprozessen, also somit auch der Hardware/Software Integration, führt damit letztendlich zu einer erhöhten Produktqualität bei niedrigerem Preis pro Stück.

1.2 Problemstellung

In dieser Arbeit soll eine Plattform zur Durchführung von Software Treiber Modul Verifikationen und weiterführenden Hardware/Software Integrationstests entwickelt werden. Dabei soll ein ausgewähltes Konzept auf einer passenden Plattform evaluiert und ein Ausblick auf plattformunabhängige Implementierungen des Konzepts geschaffen werden. Der Fokus der Evaluierung soll auf die Durchführbarkeit und Effizienz der Verifikation von LLD Software Komponenten gerichtet sein. Basierend auf den Ergebnissen soll ein Ausblick auf weitere Hardware/Software Integration Verifikation Ebenen geschaffen werden. Die entwickelte Plattform mit zugehörigem Framework soll möglichst generisch mit verschiedensten Prototypen eingesetzt werden können. Für die Auswahl einer Hardware Plattform soll ein Mikrocontroller mit zugehörigen Evaluation Board verwendet werden, welcher im Automobile Sektor in Serie eingesetzt wird. Damit soll das Konzept auf einer optimierten Umgebung erprobt werden, um dedizierten Standards, wie der ISO 26262, zu entsprechen. Die Umsetzung von dezidierten Integrationstests auf Zielumgebungen wird in [11] als notwendige Verifikation beschrieben. Ein Ausblick auf andere Zielumgebungen, welche in Zukunft auch interessant in der Automobile Branche sein könnten, soll ebenso gewonnen werden. So könnten Field Programmable Gate Arrays (FPGAs) für Image Processing Applikation in der Automotive Lichtsystem Entwicklung angewandt werden. Software Komponenten auf Hard- oder Softcore Central Processing Units (CPUs) eines solchen System on Chip (SoC) Systems müssen ebenso passend zu ihrer zugehörigen Hardware Komponente integriert und an den Schnittstellen verifiziert werden.

Nach Auswahl einer Mikrocontroller Plattform, einer Debugging Hardware und einer Hardware für die physikalischen Messungen und Instrumentierungen, soll eine Testing Plattform umgesetzt werden. Dafür soll eine Testing Software Architektur entwickelt werden, welche sowohl sämtliche Hardware abstrahiert, als auch das automatisierte Testmanagement durchführt. Die abstrahierte Hardware soll den Build Prozess der zu testenden Software Komponente triggern, die Komponente auf dem Mikrocontroller programmieren, auf die Software Symbole zugreifen und physikalische Signale triggern und messen können. Das Testmanagement soll sämtliche implementierten Tests durchlaufen und einen Report dieser Tests erstellen. Die entwickelte Architektur soll dabei im Kern möglichst generisch sein, um möglichst unabhängig von der Plattform sowie der Testing Hardware zu sein.
Um diese Ziele erreichen zu können, müssen unterschiedliche Konzepte für die Implementierung von Embedded Hardware/Software Integration Tests evaluiert werden. Im Speziellen soll auf die unterschiedlichsten Arten der Instrumentierung der Software auf der Zielhardware eingegangen werden, um deren Vor- und Nachteile zu erörtern. Abgeleitet durch die gewonnenen Erkenntnisse soll eine Architektur zur Implementierung einer solchen Hardware/Software Integration Test Umgebung erörtert und geprüft werden. Mittels einer beispielhaften Prüfung gilt es das gewählte Konzept zu analysieren. Die Schaffung eines Ausblicks für die automatisierte Durchführung von Hardware/Software Integration Tests ist das letztendliche Ziel. Vorteile gegenüber einer manuellen Durchführung von Hardware/Software Integration Tests sollen quantitativ bewertet werden. Die entwickelte Plattform soll für die Implementierung solcher Tests möglichst benutzerfreundlich und generisch einsetzbar sein.

1.3 Neuheitswert

In dieser Arbeit ist eine Alternative zur gängigen Praxis im Hardware/Software Integration Prozess in der Embedded System Entwicklung erörtert worden und deren Vor- und Nachteile wurden bewertet. Dabei ist die Fehleranfälligkeit von händisch durchgeführten Messungen mit den entsprechenden Prototypenaufbauten gegenüber automatisierten Testmethoden festgestellt worden. Der zeitliche Aufwand, welcher für die Durchführung von manuellen und automatisierten Tests notwendig ist, wurde ins Verhältnis gesetzt, um die nachhaltige Kostenersparnis von automatisierten Tests zu bewerten. Ein Ausblick für ein mögliches Konzept von plattformunabhängigen automatisierten funktionalen Verifikationen an der Schnittstelle zwischen Hardware und Software soll den Fokus auf den System Integration Prozess erhöhen. Dabei kann die Qualität des Produktes und des Entwicklungsprozesses in der Automotive ECU Entwicklung gesteigert werden.
Des Weiteren konnte ein Ausblick auf Steigerung der Effizienz durch automatisierte Tests bei der Entwicklung von elektronischen Bauteilen mit zugehöriger Treiber Software geschaffen werden. Dabei kann die Entwicklung der zugehörigen Software Komponente in einem kontinuierlichen Entwicklungsprozess eingepflegt und unabhängig von anderen System Komponenten durchgeführt werden. Zusätzlich ist festgestellt worden, dass dasselbe Konzept wie in der System Integration angewandt werden kann. Auch hier sind die Nachteile einer rein händischen Inbetriebnahme ersichtlich.

1.4 Aufbau der Arbeit

In dieser Arbeit wird zu Beginn der Stand der Technik (Kapitel 2) zu Embedded System Modul Verifikationen und Integration Verifikationen beschrieben. Dabei werden sämtliche Prozesse im V-Modell [2], welche diese Inhalte wiedergeben, genauer erörtert und beschrieben. Basierend auf bekannten Methoden werden Konzepte (Kapitel 3) für die Durchführung von Hardware/Software Integration Tests beschrieben und deren Schwachstellen erörtert. Anschließend wird das gewählte Konzept (Kapitel 4) zur Durchführung der Aufgabenstellung beschrieben. Die tatsächliche Implementierung (Kapitel 5) mit sämtlichen Teilaspekten, z.B. Messinstrumentierung und Debug-Interface, wird entsprechend dem Konzept beschrieben. Dabei wird auch die implementierte Testing-Software Architektur für die Implementierung von Hardware/Software Integrationstests erörtert. Im Hinblick auf die Evaluierung der entwickelten Plattform wurden Testszenarien und Testaufbauten geschaffen (Kapitel 6). Eine Evaluierung im Sinn des Vergleiches zwischen manuellen und automatisierten Tests (Kapitel 7) wird durchgeführt und die Ergebnisse (Kapitel 8) analysiert und diskutiert. Dabei werden alternative Lösungen für einen Vergleich herangezogen, um Vor- und Nachteile zu lokalisieren. Eine Bewertung der Vergleichsanalysen zu manuellen Tests wird beschrieben. Der Neuheitswert des geprüften Konzeptes soll verteidigt werden. Letztendlich soll ein Ausblick (Kapitel 9) auf die Durchführung von Hardware/Software Integration Tests, basierend auf der entwickelten Plattform, geschaffen werden.

2 Stand der Technik

Die Entwicklung von Embedded Systems definiert sich durch die Kombination ganz unterschiedlicher Entwicklungsdomänen. Bei der Entstehung eines solchen Systems wird dieses in Entwicklungen unterschiedlichster Disziplinen wie Software, Hardware als auch Mechanik unterteilt. Die Interaktion sowie Integration und Validierung wird dabei durch Entwicklungsarbeit auf Systemebene gesteuert und überwacht. Diese zumeist recht komplexen Prozesse können in ganz unterschiedlichen Modellen abstrahiert werden. In diesem Kapitel wird auf den Entwicklungsprozess genauer eingegangen, um letztlich die Notwendigkeit von Modul Verifikationen und Integration Verifikationen zu verdeutlichen. Dabei wird auf die einzelnen Ebenen, im speziellen für Integration Testing [12], eingegangen. Software Modul Verifikation sowie System und Software Integration Konzepte werden detailliert erörtert. Insbesondere wird auch die Hardware/Software Integration als domainenübergreifender Prozess beschrieben.

2.1 Entwicklung von Embedded Systems nach dem V-Modell

Es existieren ganz unterschiedliche Modelle, nach welchen Embedded Systems entwickelt werden können. Ein gängiges Modell hierbei ist das V-Modell [13], welches im Speziellen den Test des gesamten Systems, sowie alle weiteren Abstraktionsschichten auf die gleiche Ebene wie die Entwicklung selbst stellt. Dabei wird verdeutlicht, dass eine Verifikation und Validierung eines Produktes genauso wichtig anzusehen ist, wie die Entwicklung selbst und dass damit auch der dementsprechende Aufwand für die Entwicklung von Testsystemen rechtfertigbar ist [14]. Dabei existieren generische Praktiken und Abstraktionsebenen, um einen solch intensiven Verifikation- und Validierungsaufwand effizient umzusetzen [15]. Ein Entwicklungsprozess nach dem V-Modell ist für sämtliche Komponenten eines Embedded Systems anwendbar. Dabei können für Entwicklungs- und Testprozesse für Software, Hardware und Mechanik Komponenten formuliert werden. Diese müssen in den gesamtheitlichen Systementwicklungsprozess integriert werden. Abbildung 2.1 veranschaulicht diese Unterteilung in die einzelnen Systemkomponenten.
Zur Beschreibung des V-Modells mit einem höheren Detaillierungsgrad verdeutlicht Abbildung 2.2, wie das V-Modell bezogen auf die Systeme Entwicklung und die Software Entwicklung des Systems aussieht. Dabei ist ersichtlich, dass die einzelnen Schritte dieser Entwicklung in einzelne Prozesse unterteilt sind, welche beginnend beim linken oberen Ende des Vs starten und dann sequentiell den darauf folgenden Prozess im V triggern. Dabei stellt die linke Seite des Vs jene Prozesse dar, welche die tatsächliche Entwicklung des Systems beschreiben. Die rechte Seite des Vs stellt die Prozesse für eine Verifikation und Validierung des Systems da. Es ist erkennbar,

dass zu jedem Entwicklungsprozess in der horizontalen Ebene auch ein dedizierter Testprozess formuliert ist.

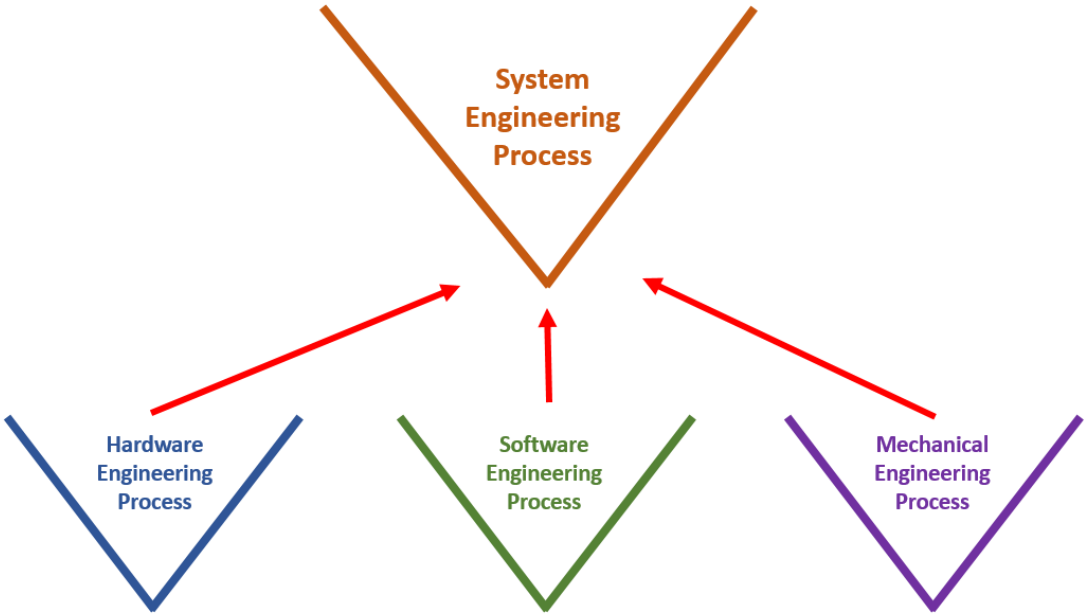

Abbildung 2.1: V-Modell für Embedded System Entwicklung erweitert um Hardware und Mechanik Entwicklung

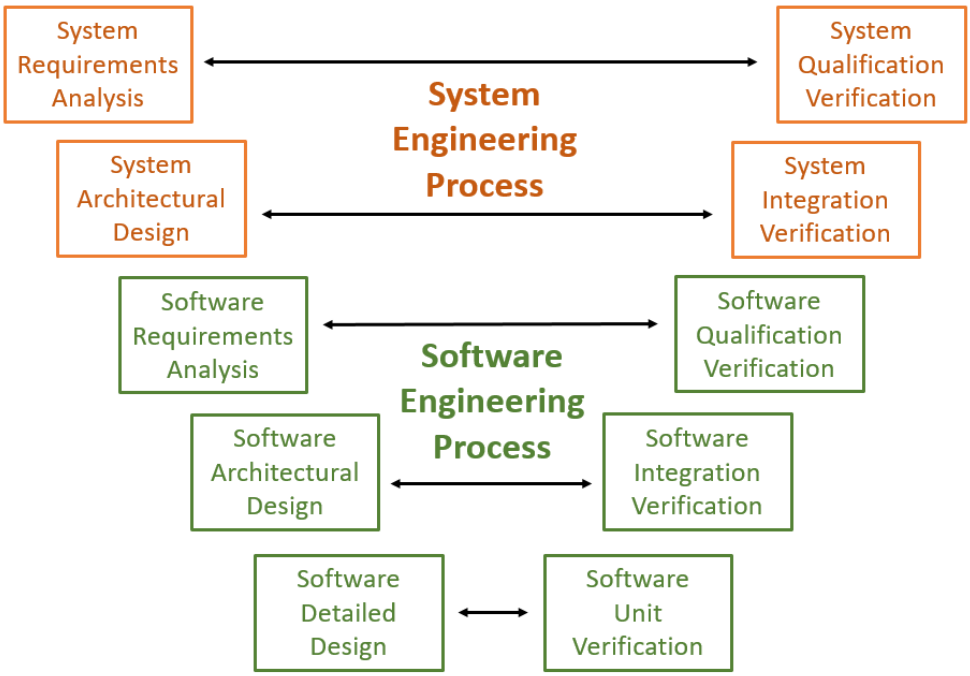

Abbildung 2.2: V-Modell für Embedded System Entwicklung [2]

Im Folgenden werden sämtliche Entwicklungsprozesse beschrieben, wobei der Komponenten Entwicklungsprozess beispielhaft am Software Entwicklungsprozess veranschaulicht wird [2]:

- **System Engineering Process**
  - **System Requirements Analysis:** In diesem Prozess werden Stakeholder Requirements in System Requirements formuliert, nach welchen dann Systeme entwickelt und validiert werden sollen. Stakeholder Requirements können von Kunden formuliert sein sowie auch Produktionsanforderungen oder interne Anforderungen, welche die Wiederverwendbarkeit und Wartbarkeit des Systems gewährleisten sollen. Dabei kann es sich um funktionale als auch nicht funktionale Anforderungen, wie Safety und Security Requirements, handeln. Das Ziel dieses Prozesses ist es, klar verständliche Requirements für sämtliche nachfolgende Prozessschritte zu formulieren. Dabei müssen Elemente der Implementierung, Verifizierung und Validierung auf eines der Requirements zurückführbar sein.
  - **System Architectural Design:** Hier wird eine System Architektur geschaffen, welche das System in einzelne System Komponenten abstrahiert. Dabei müssen die System Requirements den einzelnen Komponenten zugewiesen werden. Dabei orientieren sich die Komponenten daran, ob sie einen gezielten Zweck erfüllen. So umfasst z.B. die System Komponente des Mikrocontrollers mit zugehöriger Peripherie eine Komponente zur Erfüllung von prozessorientierten Rechenaufgaben sowie der Kühlkörper eine Komponente zur Kühlung des Systems ist. Meistens wird eine System Architektur durch ein grafisches Design mit zugehöriger Beschreibung erstellt. Eine System Architektur wird meistens möglichst modular mit möglichst einfachen Schnittstellen, welche sowohl zwischen den Komponenten als auch zur Außenwelt ausgeführt sind, aufgebaut, um die Komplexität des Gesamtsystems zu reduzieren. Dabei können auch generische Komponenten verwendet werden, welche einfach wiederverwendbar sind.
  - **System Integration Test:** Komponenten, welche in der System Architektur definiert wurden, werden nach Entwicklung ins System integriert. Somit wird das Gesamtsystem erstellt. Diese Integration, also das Zusammenspiel zwischen den einzelnen Komponenten, muss durch Tests ausreichend verifiziert sein. Die Integration und die zugehörige Verifikation sollen den Anforderungen der zugehörigen System Requriments genügen. Für die Verifikation müssen Testfälle erstellt und durchgeführt werden. Dafür erfolgt eine Auswahl der Testmethoden und Kriterien. Anfangs- und Endbedingungen sowie Schwellwerte für die Parameter der Testfälle müssen festgelegt werden. Dabei kann die Testabdeckung der Systemkomponenten eine Rolle spielen. Die Tests mit den jeweiligen Ergebnissen müssen ausreichend dokumentiert und mit den System Requirements verlinkt sein. Um die Effizienz der Integrationstests zu steigern, können diese möglichst automatisiert abgearbeitet werden.
  - **System Qualification Test:** In diesem Prozess wird das entwickelte System gegen sämtliche Stakeholder Requirements getestet. Dabei müssen sowohl sämtliche funktionalen als auch nicht funktionalen Requirements ausreichend verifiziert und validiert werden. Eine Test Strategie mit den dementsprechenden Testfällen wird dafür zurecht gelegt. Das entwickelte System ist hierbei gänzlich als Black Box zu betrachten. Die durchgeführten Tests müssen mit Verweis zu den Requirements auch ausreichend dokumentiert werden. Das Ergebnis von solchen Qualifikationstests sind zumeist einfach begutachtbare Testscripten, welche gerne automatisiert generiert werden, um die Effizienz des Prozesses zu steigern.

- **Software Engineering Process**
  - **Software Requirements Analysis:** Hier werden Software Requirements basierend auf System Requirements und denen als Software Komponenten zugewiesenen System Architektur Komponenten formuliert. Die Rückverfolgbarkeit auf die verwiesenen System Komponenten und System Requirements muss hierbei stets gewährleistet sein. Wie bereits bei den System Requirements verdeutlicht, müssen auch Software Requirements klar verständlich und eindeutig interpretierbar in sämtlichen folgenden Schritten im Software Entwicklungsprozess sein. Sämtliche Module sowie Verifizierungen müssen einem Requirement zuordenbar sein. Funktionale sowie nicht funktionale Requirements sollen letztlich ebenso abgedeckt werden. Bei der Formulierung der Software Requirements ist auch die Interaktion zu den umgebenden Hardware Komponenten und deren Requirements mit entsprechenden Verweis in einem Embedded System sinnvoll. Dabei sind die umzusetzenden Schnittstellen zur Umgebung klar zu formulieren.
  - **Software Architectural Design:** Hier wird eine Software Architektur geschaffen, welche die Software in einzelne Software Komponenten abstrahiert, um der Software die Komplexität zu nehmen. Dabei müssen die Software Requirements den einzelnen Komponenten zugewiesen werden. Die Komponenten orientieren sich daran, ob sie einen gezielten Zweck erfüllen. Meistens wird eine Software Architektur durch ein grafisches Design mit zugehöriger Beschreibung erstellt. Dabei wird die Architektur oftmals in verschiedenen Abstraktionsebenen, sowie z.B. eine Basissoftware oder Applikationssoftware Ebene, unterteilt [16]. Die in diesen Ebenen beschriebenen Komponenten werden zumeist in weitere Module, welche einen dediziert funktionalen oder nicht funktionalen Teilaspekt erfüllen, aufgeteilt. Die Interaktionen zwischen den Modulen, den Komponenten sowie zwischen den Ebenen sind ebenso ausreichend zu beschreiben. Der Einsatz von generischen Modulen sowie gar von generischen Komponenten steigert im Allgemeinen die Qualität der entwickelten Software. Auch die Integration von sogenannten *Third Party* Modulen oder Komponenten soll bereits in der Architektur ersichtlich sein.
  - **Software Detailed Design:** In diesem Prozess werden die einzelnen Module der Software Komponenten implementiert nach den in den Software Requirements und Architekturdesign beschriebenen Anforderungen. Neben der Implementierung der prozeduralen Softwarefunktionalität ist die Entwicklung der Modulschnittstellen von besonderer Wichtigkeit in Bezug auf Lesbarkeit, Einfachheit und Verwendbarkeit. Eine graphische Beschreibung der Software, über z.B. Sequenzdiagramme oder State-Maschinen, repräsentiert eine anschauliche Dokumentation. Die Kategorisierung der Funktionen in unterschiedlichste Typen, wie Tasks oder Interrupts, welche genau spezifiziert sind, ist für die Entwicklung eines Moduls ebenso sinnvoll.
  - **Software Unit Verification:** In diesem Prozess werden die einzelnen Software Module getestet. Dies umfasst mehrere mögliche Methoden zur ausreichenden Verifizierung. So kann die Qualität der entwickelten Module durch Source Code Reviews gesteigert werden. Dabei wird einerseits überprüft, ob die Software auch für eine externe fachkundige Person verständlich ist, sowie ob die Software Requirements ordnungsgemäß umgesetzt werden. Eine Toolunterstützung, auch für statische Code Analysen nach gewissen Metriken, ist hier sinnvoll einsetzbar. Des Weiteren können auch dedizierte Testfälle beschrieben und implementiert werden. Diese sogenannten Unit Tests basieren auf der Triggerung der im Modul Interface angegebenen Funktionen und der Überprüfung auf korrekte Abarbeitung der Modulfunktion nach unterschiedlichen Me-

thoden, wie der Vergleich der Werte von Rückgabewerten oder globaler Variablen. Dabei wird auf die sogenannte Code Coverage [17], welche die Abdeckung der getesteten Codes verifiziert, besonderer Wert gelegt. Bei sämtlichen Verifizierungsmethoden muss der Querverweis zu den Software Modulen, sowie zu den jeweiligen Software Requirements eindeutig sein. Um die Effizienz der Modultests zu steigern, können diese möglichst automatisiert abgearbeitet werden.

- **Software Integration Test:** Komponenten, welche in der Software Architektur definiert wurden, werden nach Entwicklung in die Gesamtsoftware integriert. Dabei kann die Integration schrittweise, durch z.B. Erstellung der Komponenten aus den einzelnen Modulen, vollzogen werden. Diese Integration, also das Zusammenspiel zwischen den einzelnen Komponenten und Modulen, muss durch Tests ausreichend verifiziert werden. Die Integration und die zugehörige Verifikation sollen den Anforderungen der zugehörigen Software Requriments genügen. Für die Verifikation müssen Testfälle erstellt und durchgeführt werden. Dafür müssen Testmethoden und Kriterien ausgewählt und formuliert werden. Anfangs- und Endbedingungen, sowie Schwellwerte für die Parameter der Testfälle, müssen festgelegt werden. Dabei kann die Testabdeckung der Softwarekomponenten eine Rolle spielen. Oftmals können für diesen Verifikationsschritt die selben Tools wie bei der Software Unit Verifikation verwendet werden. Die Tests mit den jeweiligen Ergebnissen müssen ausreichend dokumentiert und mit den Software Requirements verlinkt werden. Um die Effizienz der Integrationstests zu steigern, können diese möglichst automatisiert abgearbeitet werden.
- **Software Qualification Test:** Hier wird die entwickelte Software gegen sämtliche Software Requirements getestet. Dabei müssen sowohl sämtliche funktionalen als auch nicht funktionalen Requirtements ausreichend verifiziert werden. Eine Test Strategie wird mit den dementsprechenden Testfällen zurecht gelegt. Die entwickelte Software ist hierbei gänzlich als Black Box zu betrachten. Die durchgeführten Tests müssen mit Verweis zu den Requirements auch ausreichend dokumentiert werden. Das Ergebnis von solchen Qualifikationstests sind zumeist einfach begutachtbare Testscripten, welche gerne automatisiert generiert werden, um die Effizienz des Prozesses zu steigern.

Dieser gesamtheitliche Entwicklungsprozess kann iterativ abgearbeitet werden. Dabei können sowohl sämtliche Prozessschritte sequentiell durchlaufen werden, wie auch die Entwicklung in den einzelnen Prozessen selbst iterativ geschehen. Wichtig bleibt hierbei die richtige Verwendung von Versionsnummern für die Rückverfolgbarkeit der Entwicklungsschritte, sowie für die richtigen Querverweise zwischen den einzelnen Prozessen. Des Weiteren sei auch erwähnt, dass die einzelnen Prozessschritte nicht nacheinander abzuarbeiten sind, sondern parallel durchzuführen sind. Bloß das Triggern der Prozesse kann iterativ geschehen.
Mit steigender Komplexität von Embedded Software nimmt auch die Anzahl der benötigten Testfällen, um eine hinreichend genügende Testabdeckung zu erzielen, zu. Dabei macht oftmals Sinn Testspezifikationen auf mehreren Ebenen im V-Modell wiederzuverwenden [18]. Sowohl Software Modul Tests, Software Integration Tests und System Integration Tests als auch Hardware/-Software Integration Test können abgedeckt werden. Um diese Prozessschritte hier zu optimieren ist eine einheitliche Abstraktion der Test Interfaces notwendig [19]. Damit ist es möglich, sich Testdaten auf dem jeweiligen Layer generieren zu lassen. Je nach Konfiguration können unterschiedlichste Testdaten entstehen. Dabei werden auch sogenannte unrealistische Testdaten generiert [18], um die Grenzen des Systems auf allen Ebenen auszureizen. Bei der Generierung ist allerdings wichtig, vorab generelle Regeln, basierend auf den System Requirements, festzulegen. Basierend auf den generierten Testdaten können dann auch erzeugte Testergebnisse entsprechend

automatisiert validiert werden. Formal gesprochen bedeutet dies, dass bei Richtigkeit der Conditions, geprüft durch Assertions, die Richtigkeit der Ergebnisse stets überwacht werden kann.
Um ein Produkt mit einer möglichst guten Qualität nach unterschiedlichsten Kriterien, wie Zuverlässlichkeit, Sicherheit etc., entwickeln zu können, muss die Qualität des Entwicklungsprozesses ebenso gut sein. Als bewährte Methode zur Prüfung in einem Entwicklungsprozess ist die Entwicklung nach dem Software Process Improvement and Capability Determination (SPICE) Standard [2], auch bekannt unter ISO/IEC 33002 Standard. Dieser Standard beschreibt Qualitätslevel eines Software Entwicklungsprozesses, welche gut durch die Prozesse, angeführt im V-Modell, abstrahiert werden können. Dieser Standard kann auf sämtliche Komponenten des Embedded System, sowie auf die Systemebene selbst, ebenso angewendet werden. In der Automobile Branche wird dies durch Prüfung gegen den sogenannten ASPICE Standard in Assessments auch angewandt. Um ein möglichst hohes Qualitätslevel zu erreichen, werden unterschiedlichste Kriterien überprüft. Dabei ist die Durchführung der einzelnen Prozesse nach jeweils eigenen Gesichtspunkten genauso wichtig wie die Interaktion zwischen den Prozessen. Ziel ist die Zuordnung des Systems in eine Dimension des Reifegradmodells [9], welches folgende Level beinhaltet:

- **Level 0 (Unvollständig):** Prozesse sind nicht umgesetzt oder deren Zweck ist nicht erfüllt.
- **Level 1 (Durchgeführt):** Zwecke der Prozesse sind umgesetzt.
- **Level 2 (Gemanagt):** Prozessausführungen werden geplant, Fortschritte dokumentiert und zugehörige Aspekte implementiert.
- **Level 3 (Etabliert):** Es existiert ein weitgehend allumfassender Standardprozess, welcher für ein Projekt abgeleitet und dementsprechend konfiguriert wird.
- **Level 4 (Vorhersagbar):** Bei Prozessausführungen werden detaillierte Messungen durchgeführt und bewertet, die das quantitative Verständnis der Prozesse erhöhen und damit auch zu einer verbesserten Vohersagegenauigkeit führen.
- **Level 5 (Innovativ):** Prozesse werden ständig verbessert und weiterentwickelt. Neue Methoden werden erprobt und analysiert. Qualitativ schlechtere Prozesse werden durch neue ersetzt.

Dabei lassen sich zu den jeweiligen Reifegradstufen folgende Attribute in Tabelle 2.1 zuordnen. Es sei angemerkt, dass Attribute eines niedrigeren Levels in den höheren Leveln beinhaltet sein müssen.
Neben der Kategorisierung des entwickelten Systems nach dem Reifegradmodell, werden in der Automobilindustie Systeme auch nach dem Safety Standard ISO 26262 in vier Automotive Safety Integrity Level (ASIL) kategorisiert [10]. Wichtig ist hier zu erwähnen, dass der Umfang dieser Kategorisierung auf Systemarchitektur, Safety Management und Analyse abzielt und ein Teil des ASPICE Entwicklungsprozesses sein kann.
Ein weiteres Argument, welches diese prozessorientierte und nachhaltig optimierten Aufwendungen zur Verbesserung der Entwicklungsprozesse und damit auch des entwickelten Produkts rechtfertigt, sind Kosten. Dabei wird die Gestaltung einer kontinuierlich verbesserten Prozesslandschaft als Investment gesehen. Dabei ist oft üblich, dass EntwicklerInnen 10-20% ihrer Zeit in kontinuierliche Verbesserungen stecken [3]. Diese verhältnismäßig kleineren Anteile resultieren in robusteren und funktionaleren Designs, ohne dabei den Fokus auf die Entwicklung des Produktes zu verlieren. Abbildung 2.3 demonstriert dabei, dass Investition in Verbesserungen, welche man früh tätig, sich über die Entwicklungszeit amortisieren, da nachträglich weniger große Investments in Bugfixes

Reifegradstufe	Prozessattribute
Innovativ (Level 5):	Prozessinnovation Prozessoptimierung
Vorhersagbar (Level 4):	Quantitative Prozessanalyse Quantitative Prozesssteuerung
Etabliert (Level 3):	Prozessdefinition Prozessanwendung
Gemanagt (Level 2):	Management der Prozessausführung Management der Arbeitsprodukte
Durchgeführt (Level 1):	Prozessausführung
Unvollständig (Level 0):	

Tabelle 2.1: Attribute des Reifegradmodells [9]

und nachträglichen Verbesserungen getätigt werden müssen. Dabei wird im Allgemeinen in zwei Entwicklungsarten unterschieden:

- **Taktisch**: Beim taktischen Entwickeln liegt der Fokus auf dem Umsetzen einer dediziert angeforderten Applikation, aber nicht auf deren Wiederverwendbarkeit oder ähnlicher optimierter Desginaspekte.

- **Strategisch**: Beim Entwickeln von langlebigen, nachhaltigen und designoptimierten Systemen wird strategisch entwickelt. Dabei wird z.B. viel an Aufwand frühzeitig investiert, um eine Komponente möglichst gut an ein Gesamtsystem anzubinden und dieses auch möglichst unkompliziert wiederzuverwenden oder zu erweitern. Frühzeitig entstandene Kosten amortisieren sich hier zumeist.

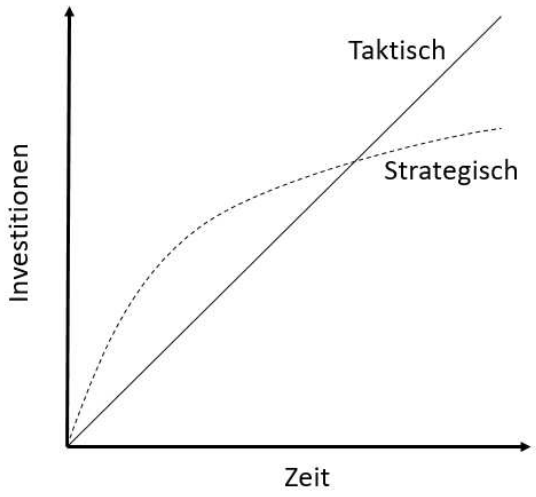

Abbildung 2.3: Aufwände nach taktischer und strategischer Systementwicklung [3]

2.2 Software Modul Verifikation

Software Modul Verifikation, welche auch gerne Unit Tests genannt werden, beschreiben entwicklungsbegleitende Tests der kleinsten modularen Einheiten der Software gegen den spezifizierten Zweck im Design und sind in Architekturen mit großem Integrationanspruch besonders effektiv [4]. Dabei werden Modultests im Allgemeinen dahingehend spezifiziert, dass die gesamte Quelldatei mit sämtlichen definierten Funktionalitäten getestet wird. Sämtliche Abhängigkeiten zu anderen Modulen werden aufgelöst und durch Ersatzfunktionen in der Testimplementierung ersetzt. Dabei existieren folgende Arten von Ersatzfunktion:

- **Mocks**: Eine Funktion mit Interface und Logik wird implementiert. Eine dynamische Bearbeitung der Eingangswerte wird umgesetzt.
- **Stubs**: Entspricht einen Mock, ohne der Möglichkeit der Überprüfung, ob die Ersatzfunktion tatsächlich aufgerufen wurde.
- **Fakes**: Das Interface einer Funktion wird implementiert, aber nicht deren Logik. Stattdessen werden feste Daten hinterlegt.

Die Auflösung der Modulabhängigkeiten hat den Sinn, externe Einflüsse beim Test auszuschließen. Für die Verwendung von Ersatzfunktionen und zur Implementierung von Testscripten werden der entsprechenden Programmiersprache des zu testenden Moduls Unit-Test-Frameworks verwendet. Abbildung 2.4 demonstriert einen Modul Tests, wobei das zu testende Modul orange, unabhängige weitere Software Module blau und zusätzliche Mocks sowie Bibliotheken grün markiert sind.

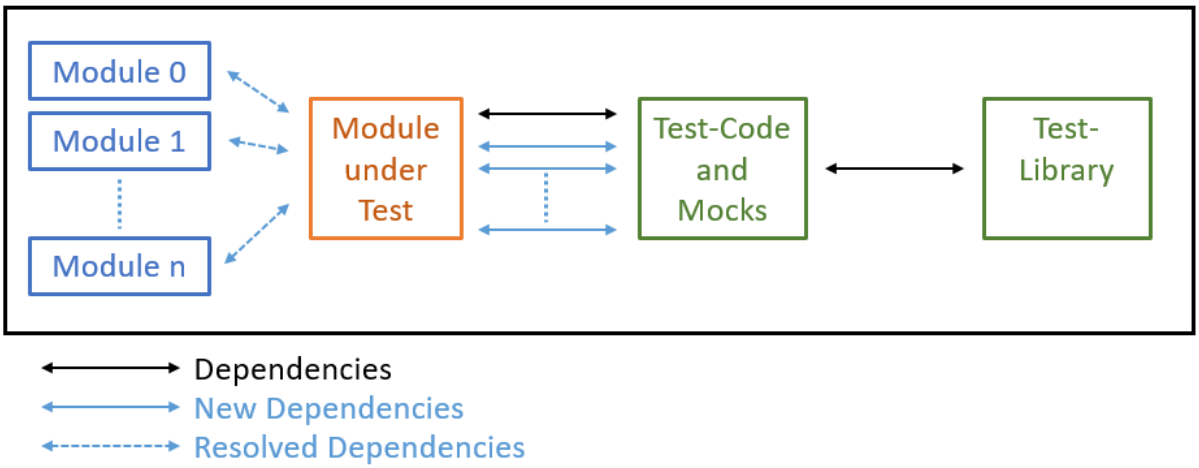

Abbildung 2.4: Aufbau eines Unit Tests

Für jedes zu testende Modul werden Testscripts erstellt, in welcher sämtliche Testmethoden definiert sind. Dabei umfasst jeder einzelne Test drei Schritte [20]:

1. **Arrange (Vorbereitung):** Hier werden sämtliche für den individuellen Test notwendigen Vorbedingungen geschaffen, sowie sämtliche benötigte Variablen angelegt und initialisiert.
2. **Act (Ausführung):** Der Test selbst wird ausgeführt.
3. **Assert (Prüfung):** Die nach dem Test erwarteten Ergebnisse werden gegen die vorbereitenden erwarteten Ergebnisse geprüft.

Bei der Erstellung von Modultests ist wichtig Testziele zu formulieren [13], wie beispielsweise die geforderte Funktionalität auf Korrektheit zu überprüfen. Dabei werden Testfälle formuliert, welche spezielle Ein-/Ausgangskombinationen der zugehörigen Parameter variieren. Ein weiteres alltägliches Testziel auf Modulebene ist der Test auf Robustheit des Moduls. Dabei wird unter Simulation des Fehlverhaltens eines oder mehrerer umgebener Module, Komponenten oder Schnittstellen das korrekte Detektieren des Fehlverhaltens und die Fortführung des gewünschten funktionalen Verhaltens des Moduls getestet. Des Weiteren können Module auch auf Wirtschaftlichkeit, also auf Speicherplatzbedarf und Rechenzeit, verifiziert werden. Im Speziellen beim Entwickeln von komplexen Algorithmen ist deren Ausführungszeit oftmals eine Spezifikation, welche verifiziert werden sollte. Auch der Umgang mit Ressourcen in einer Embedded System Umgebung macht eine dementsprechende Spezifikation sinnvoll.
Nun können gewisse Eigenschaften eines Moduls nicht durch die Entwicklung von dynamischen Tests überprüft werden. So sollte im Kontext von Wartbarkeit das Modul auf Wiederverwendbarkeit sowie Weiterentwickelbarkeit geprüft werden. Dabei soll die Codestruktur möglichst modular und einfach verständlich sein. Dabei spielt die Dokumentation eine große Rolle. Hier werden Code Reviews mit fachkundigen EntwicklerInnen, welche oft durch Tools zur statischen Code Analyse unterstützt werden, herangezogen. Dabei kann auch darauf geachtet werden, dass gewisse Metriken eingehalten werden.
Ein wichtiger Teil von dynamischen White-Box Tests ist die Analyse der Code-Coverage, mit welcher die Testabdeckung und der Testfortschritt quantitativ bestimmt und auch protokolliert werden können [17]. Dabei lässt sich die Code-Coverage mittels kontrollflussorientierten Tests erörtern. Möglichst viele Programmabschnitte sollen durch die Eingabe spezifizierter Werte in Testvektoren durchlaufen werden, um eine möglichst hohe Abdeckung zu erreichen. Zumeist werden hierfür einzelne Funktionen getriggert. Ziel einer Code-Coverage Analyse ist es, Abschnitte im Code zu lokalisieren, welche in detaillierten White-Box-Tests verifiziert werden sollten. Dabei existieren die folgenden verschiedenen Stufen von Code-Coverage, welche allerdings korrelieren können:

- **Funktionsebene:** Diese Ebene definiert sich über das Verhältnis der Anzahl der aufgerufenen Funktionen zu der Gesamtanzahl der Funktionen im Modul.

$$\text{Aufrufabdeckung} = \frac{\text{Anzahl der aufgerufenen Funktionen}}{\text{Gesamtanzahl Funktionen}} * 100\% \tag{2.1}$$

- **Anweisungsebene:** Diese Ebene definiert durch die Anzahl an Anweisungen, welche erreicht und ausgeführt werden. In dieser Teststufe ist es möglich, sogenannten Dead Code zu finden. Damit sind Sequenzen gemeint, welche nie ausgeführt werden.

$$\text{Anweisungsabdeckung} = \frac{\text{Anzahl der ausgeführten Anweisungen}}{\text{Gesamtanzahl Anweisungen}} * 100\% \tag{2.2}$$

- **Zweigebene:** Diese Ebene definiert das Verhältnis der Anzahl der durchlaufenen zu der Gesamtanzahl der möglichen Zweige.

$$\text{Zweigabdeckung} = \frac{\text{Anzahl der ausgeführten Zweige}}{\text{Gesamtanzahl Zweige}} * 100\% \tag{2.3}$$

- **Bedingungsebene:** Diese Ebene prüft wie viele Teilbedingungen in einer Verzweigung unabhängig voneinander Einfluss auf den Programmablauf haben. Dabei sind bei einer Anzahl von n Bedingungen n+1 Testfälle nötig, um 100% Modified Condition Decision Coverage (MC/DC) zu erreichen [17].

Zur Umsetzung von Code-Coverage Analysen muss der bestehende Programmcode modifiziert und ergänzt werden. Diese Instrumentierung geschieht bevor der produktive Code kompiliert wird. Dabei werden Zähler in Form von globalen Arrays angelegt, dessen Funktionalität von der Analyseebene abhängen. Dabei liegen diese globalen Zähler im Datensegment der Speicher auf der Plattform, auf welcher die Analyse durchgeführt wird. Mit einer Null Initialisierung werden die Zähler abhängig vom Durchlauf eines Programmabschnittes erhöht. Das bedeutet, dass beim produktiven Programmablauf auch der Programmteil der Code-Coverage Analyse durchlaufen wird. Dabei muss darauf geachtet werden, dass dieser keine logischen Änderungen im produktiven Programmablauf verursacht.
Durch die der Instrumentierung zugehörigen Datensegmente brauchen Random Access Memory (RAM) und Read Only Memory (ROM) Speicher, was bei Desktop Segmenten im Allgemeinen kein Problem ist. Doch in Embedded Systems liegt zumeist eine striktere Begrenzung der Ressourcen vor. Eine Möglichkeit dieses Problem mit dem zusätzlichen Speicherbedarf zu behandeln wäre die partielle Instrumentierung, bei welcher bloß kleine Teile des gesamten Programms partiell getestet werden. Des Weiteren kann auch die Größe der Zählervariablen reduziert werden, was allerdings zu Überläufen führen kann. Doch die zusätzliche Verzögerung in der Abarbeitung des produktiven Codes durch die Abarbeitung des Instrumentierungscodes führt zu einer Mehrbeanspruchung der CPU. Dabei ist es möglich, dass gewisse Timing Einschränkungen, wie sie z.B. für Bus Kommunikation existieren, nicht mehr eingehalten werden können. Als alternative Lösung ohne zusätzlichen auszuführenden Programmcode ist das Tracing am Mikrocontroller. Dabei wird über ein Debugging Interface der Programmablauf am Controller verfolgt und mitprotokolliert.
Es existieren einige Gründe Unit Tests auf der Host-Hardware und nicht auf der Ziel-Hardware durchzuführen [4]. So kann der quasi unbegrenzte virtuelle Speicher genutzt werden und Test Resultate können einfach auf dem Bildschirm ausgegeben werden. Dennoch macht es oftmals Sinn auf der Ziel-Hardware zu testen, um Compilerfehler und Fehler in Standardbibliotheken des Compilers festzustellen. Des Weiteren haben Compilerhersteller gewissen Interpretationsspielraum bei Programmiersprachen, welche zu fehlerhaften Interpretierungen führen könnten. Außerdem sind Tests von Mixed C/Assembler Programmen bloß auf der Ziel-Hardware durchführbar. Letztendlich ist es sinnvoll am Host-System Tests durchzuführen und diese zu debuggen und zu instrumentieren, bis eine gewünschte funktionale Abdeckung erreicht ist und dann auf der Ziel-Hardware zu testen, ohne weitere Instrumentierung.
Ersichtlich in diesem Abschnitt ist die Vielzahl an Möglichkeiten, welche bei entwicklungsbegleitenden Tests auf Modulebene durchgeführt werden können. Dazu gehören nicht nur automatisierte funktionale Tests, auch Code Reviews, begleitet durch Tools zur statischen Code Analyse, welche die formale Richtigkeit der Software unter Beachtung definierten Metriken analysiert [9]. Ein Beispiel für solche Metriken, welche beispielsweise in der Automobilindustrie dem Standard entsprechen, sind die Motor Industry Software Reliability Association (MISRA) Regeln [21]. Die Menge und der Detaillierungsgrad sämtlicher Analysen und Tests wird dabei oftmals durch die

Vorgabe des Erreichens eines gewissen Software Integrity Levels [22]. Dieser Institute of Electrical and Electronics Engineers (IEEE) Standard IEEE 1022 kategorisiert die Aufwandsoptimierung für Validierungs und Verifikations Prozesse der allgemeinen Systementwicklung. Auch Safety Standards wie die ISO 26262 beeinflussen die Intensität der Verifikation [23]. Unabhängig des erreichten Levels nach diversen Standards ist eine konsistente Dokumentation und Referenz zu den Requirements der einzelnen Testfälle stets notwendig. Dabei kann die Qualität der Dokumentation wiederum durch den ISO 29119 Standard [24] qualifiziert werden.

2.3 Software Integration

In diesem Kapitel wird die Verifikation der Interaktion zwischen den in der Software Architektur definierten Komponenten und Modulen beschrieben. Dabei liegt der Fokus vor allem auf die richtige Anwendung von Modulschnittstellen [25]. Die Richtigkeit und Rechtzeitigkeit von Datenflüssen, sowie die konsistente Interpretation in allen beteiligten Komponenten ist entscheidend [9]. Nebenbei sollte auch auf den Resourcenverbrauch bezogen auf die Plattform, auf welcher sich die entwickelte Software dann befindet, geachtet werden. Dabei existieren unterschiedlichste Verfahren für die Durchführung dieses Verifikationschrittes.

Beim Bottom-Up-Unit-Testing wird eine genau definierte Reihenfolge an zu testenden Modu-

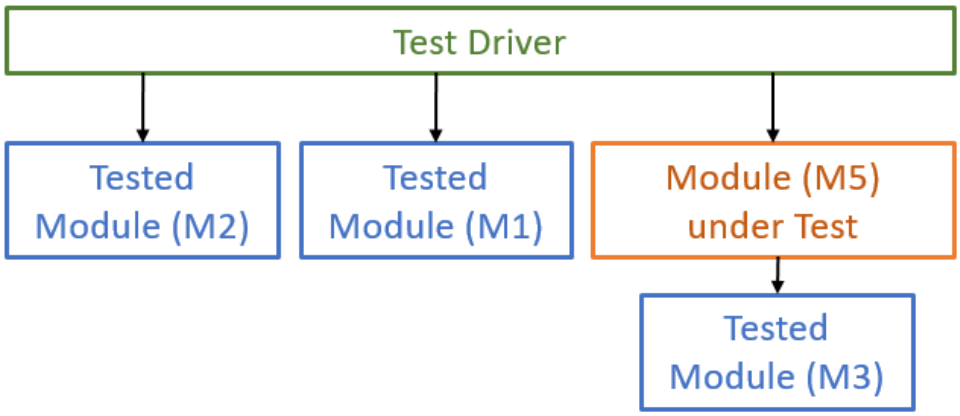

Abbildung 2.5: Bottom-Up-Unit Test

len befolgt, um einen wachsenden größeren Anteil der Software Architektur zu testen. Dabei beginnt man mit Modulen möglichst ohne weitere Abhängigkeiten, was bedeutet, dass für die Testimplementierung bloß die Testtreiber geschrieben werden müssen, ohne Mocks zu implementieren. Im nächsten Schritt werden dann Module getestet, welche die bereits getesteten Module als Abhängigkeit inkludiert haben und somit auch für deren Funktionalität verwendet werden. So arbeitet man sich in der Abhängigkeitshierachie sequentiell hoch. In Abbildung 2.5 wird demonstriert, wie das orange markierte Module M5 zu den bereits verifizierten blauen Modulen durch Tests integriert wird. Erwähnt sei an dieser Stelle, dass die Architektur der Software in diesem Abschnitt aus acht Modulen besteht und beispielhaft in sämtlichen Abbildungen der unterschiedlichen Modelle angeführt wird.

Mocks werden durch reale Module beim Bottom-up-Unit-Testing ersetzt. Eine alternative Möglichkeit der Integration eines Moduls ist der Test dieser Integration, wobei die Module, welche in Abhängigkeit zu dem zu integrierenden Modul stehen, durch sogenannte Stubs simuliert werden [25]. Die Si-

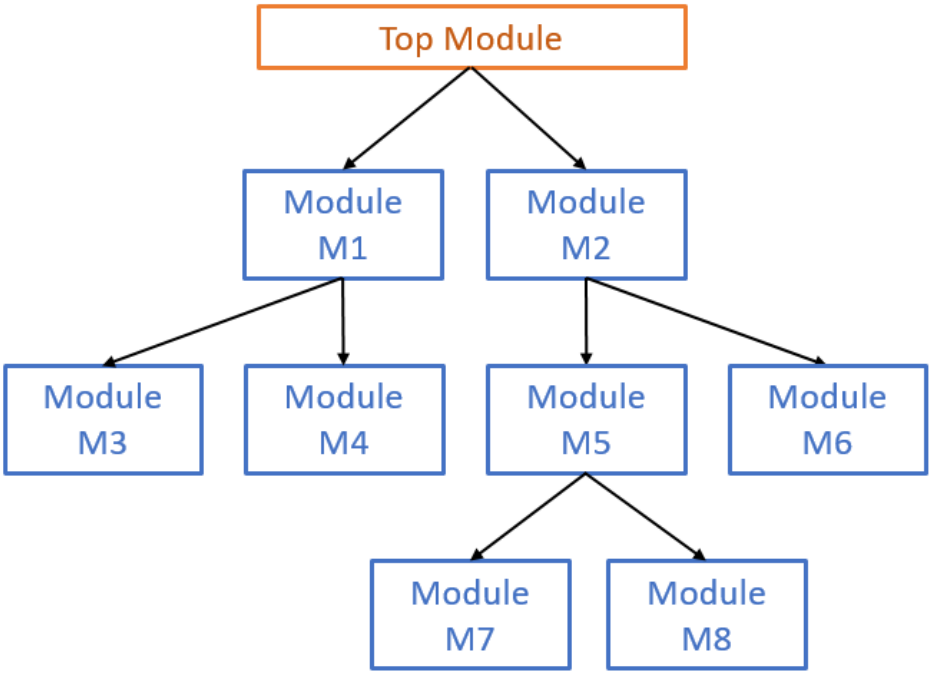

Abbildung 2.6: Top-Down Integrationsverfahren

mulation dieser Stubs wird vom Test Treiber übernommen. Dabei existieren die folgenden zwei Arten an Stubs:

- **Realistischer Stub:** Dieser simuliert den Hauptteil der Orgininalfunktionalität und kann in vielen verschiedenen Testfällen eingesetzt werden.
- **Spezifischer Stub:** Dieser simuliert eine für den dedizierten Testfall notwendige Funktionalität.

Bei der Verwendung von Stubs ist ein Bottom-Up Integration Prozess nicht notwendig, da modulfremde Funktionalitäten simuliert werden. Allerdings ist beim Top-Down Integrationverfahren der Simulationsaufwand ein recht hoher. Bei der Top-Down Software Integrationverifikation wird zu Beginn ein Top-Module mit einer *Main* Funktion getestet, während die darunterliegenden Module, abstrahiert in einer gesamtheitlichen Software Architektur, durch Stubs ersetzt werden. Bei nächsten Integrationschritten bestehen zwei Möglichkeiten:

- Sämtliche Module pro Ebene werden schrittweise integriert und die verbleibenden Stubs pro Ebene aufgelöst. In der Struktur in Abbildung 2.6 eins wäre dann die Integration der module in folgender Reihenfolge denkbar: $M1 \rightarrow M2 \rightarrow M3 \rightarrow M4 \rightarrow M5 \rightarrow M6$
- Bei Integration der Module wird ein dezidierter Pfad vom Top-Module zu einem Bottom-Module verfolgt. In der Struktur in Abbildung 2.6 eins wäre dann die Integration der Module in folgender Reihenfolge denkbar: $M1 \rightarrow M3 \rightarrow M4 \rightarrow M2 \rightarrow M5 \rightarrow M6$

Beim Top-Down Integrationverfikationsverfahren wird von Beginn an das Top-Modul, welches in Abbildung 2.6 orange markiert ist, als Test Treiber verwendet. Anschließend daran sind sämtliche

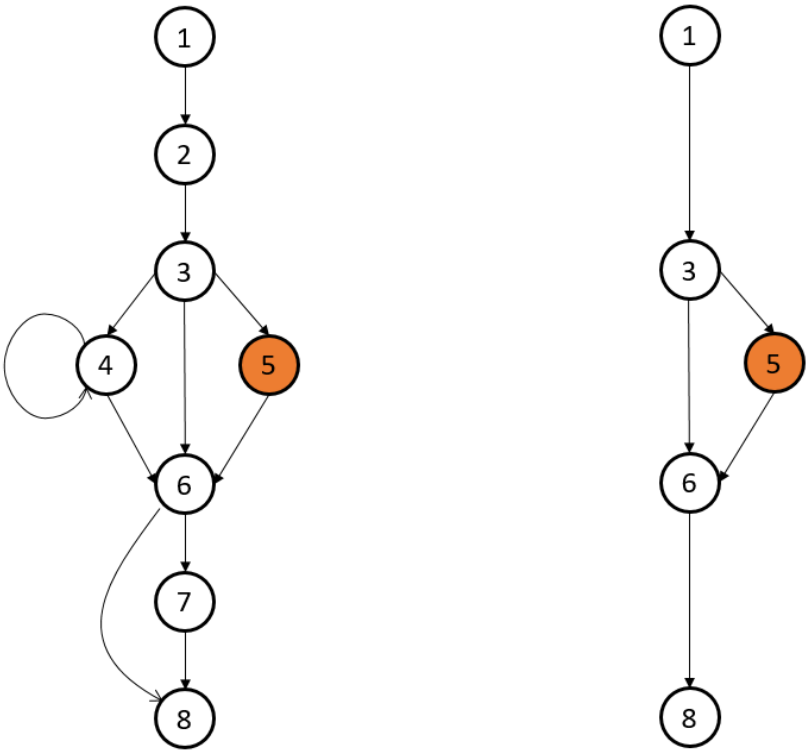

Abbildung 2.7: Strukturierter Integrationstest [4]

aufgelösten blau markierten Stubs Teil des Test Treibers.
Beim Strukturierten Integrationstest wird die Integration eines Moduls mit der gesamten Software getestet. Dabei ist die Schnittstelle zu anderen Modulen durch einen Funktionsaufruf gewährleistet. Dies wird durch den Kontrollflussgraphen in Abbildung 2.7 verdeutlicht [4]. Dabei wird deutlich, dass Knoten, welche den zu testenden orange markierten Knoten bei einem alternativen Programmdurchlauf nicht passieren, keine Aussage bezüglich des Integrationstests, des durch den markierten Knoten dargestellten Modul, haben. Der Graph kann dementsprechend auf bloß relevante Knoten reduziert werden, welche dann mit der Methode der strukturierten Unit Tests durchlaufen werden. Nachteil bei diesem Verfahren ist, dass globale Variablen, welche im markierten Modul verwendet werden, folgern, dass Module, welche eine solche Variabel verwenden, nicht reduziert werden dürfen.
Wie bereits in der Unit Verifikation, kann auch bei Integrationstests die Coverage eine Rolle spielen. Die sogenannte *Call Pair Coverage* misst den Anteil an Unterprogrammaufrufen der Software [4]. Dabei wird als Ausgang ein funktionaler System Test betrachtet, welcher die Gesamtsoftware gegen die Requirements testet. Es lässt sich leicht ableiten, wenn keine 100% Coverage erreicht wird, dass dann die Software mehr Funktionalität bietet, als die Systemtests überprüfen.
Für Software Integration Tests lässt sich zumeist dieselbe Testumgebung anwenden, wie bei Software Unit Verifikationen. Tests im gemeinsamen Kontext werden oftamls als Software In the Loop (SIL) Tests bezeichnet [26]. Dabei werden zumeist vorab getestete Module zu Komponenten integriert und getestet, welche im selben Kontext stehen. Diese Komponenten werden in die Gesamtsoftware integriert. Dabei hängt die Größe und Anzahl der zu integrierenden Module und Komponenten ganz von deren Granularität ab [25]. Des Weiteren ist es üblich, die Schnittstellen

zwischen den Modulen und Komponenten durch Monitoring Programme zu überwachen, welche auch individuell gestaltet werden können. Dabei kann ein Fehlverhalten diagnostiziert werden. Bei Code Reviews ein genaues Augenmerk auf die Modul- und Komponentenschnittstellen gelegt. Die Prüfbarkeit, Lesbarkeit und Verifizierbarkeit ist bei solchen Reviews eine Grundvoraussetzung. Eine entwicklungsbegleitende Dokumentation wird hier ebenso als Qualitätskriterium überprüft. Standards, wie bei bereits im Abschnitt 2.2 beschrieben, können hier für die Qualifizierung des Prozesses verwendet werden. Generell gelten die gleichen Kriterien bezüglich Dokumentation und Testfallkreierung hier ebenso. Dementsprechend kann zumeist auch dasselbe Toolset verwendet werden. Die Zuordnung auf die definierte Architektur Komponente ist allerdings im Zuge der Rückverfolgbarkeit hier noch zu ergänzen.
Als besonders effektive Methode zur automatisierten Verifikation der Software Komponenten Integration wird das Prinzip der Continuous Integration [9] angesehen. Dabei wird nach jedem Commit des für die Versionskontrolle verwendeten Tools eine neue lauffähige Gesamtsoftware erzeugt und verifiziert. Dabei können Fehler bei der Integration von weiterentwickelten Komponenten versioniert werden, was deren Behandlung und Rückverfolgbarkeit verbessert. Dabei soll auch durch sogenannte Smoke Tests überprüft werden, ob die Grundfunktionalität der Software noch gegeben ist.

2.4 System Integration

Ziel dieses Prozessschrittes ist es, Subsysteme an ihren in der System Architektur definierten Schnittstellen zusammenzuführen und diese gegen die System Requirements zu verifizieren. Dabei werden sowohl Bottom-Up als auch Top-Down Integrationverfahren angewendet. Bei der Verifikation werden Datenfluss und Funktionen dahingehend überprüft, dass sie ihren Zweck in den angeforderten Rahmen erfüllen und ob falsche Parameterkombinationen ein Sicherheitsrisiko darstellen [4]. Dabei ist eine schlüssige und rückverfolgbare Dokumentation unverzichtbar. Inkonsistenzen zwischen den einzelnen System sollen aufgedeckt um die Richtigkeit, Sicherheit, Zuverlässigkeit und Performance des gesamten Systems verifiziert werden [25]. Auch unerwartete Nebeneffekte sollen in der Systemintegration erkannt und in einem darauf folgenden iterativen Entwicklungsschritt behandelt werden. Letztendlich lässt sich die Systemintegration auch mit folgender Formel beschreiben [25]:

$$\text{System} = \sum \text{Subsysteme} + \sum \text{Schnittstellen} + \sum \text{Zustände} \tag{2.4}$$

Dabei wird das System als die Summe aller einzelnen Systeme mit den Schnittstellen zueinander und den dadurch sich ergebenden internen Zuständen beschrieben.
Bei der Systemintegration wird allerdings des Weiteren auch die Wechselwirkung des Systems zur Umwelt verifiziert. Das heißt, die Schnittstellen an den Systemgrenzen werden dezidiert getestet. Dabei können virtuelle Simulationumgebungen genauso eine Testumgebung darstellen, wie eine tatsächliche physikalische Triggerung.
Bei sämtlichen Integrationschritten, ganz egal ob auf Systemebene, auf Softwareebene oder auf Hardwareebene, muss auch die Integration von Komponenten, welche zugekauft sind, verifiziert werden. Dabei ist allerdings nicht notwendig, die Komponenten selbst zu verifizieren. Aber die genaue Schnittstellenbeschreibung dieser Komponenten muss ausreichend verständlich sein und dementsprechend umgesetzt werden, um eine Integration durchzuführen und zu überprüfen.
Bei der Erstellung von automatisierten System Integrationtests werden systematische Programme erstellt, um Fehler im Zusammenhang mit Schnittstellen zu detektieren. Dabei werden sämtliche

Komponenten des Systems inkrementell integriert. Dabei basiert die schrittweise Integration auf einer Umgebung mit dem Zielprozessor. Die implementierten Tests sollen direkt die Requirements verifizieren, also werden sie als Black-Box Tests durchgeführt.
Im Allgemeinen werden im Zuge der Systeme Integration sämtliche Domänen des Systems, wie

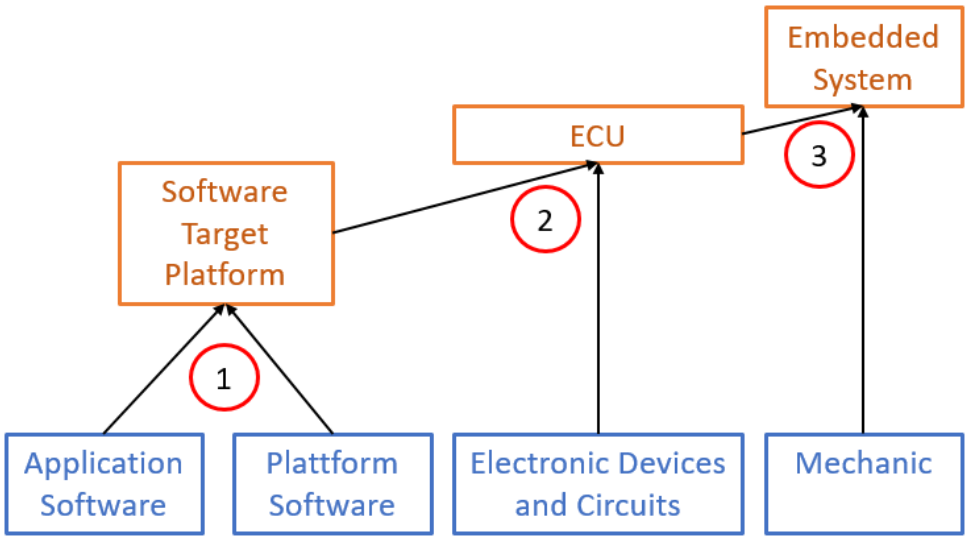

Abbildung 2.8: System Integration Prozessschritte

Mechanik, Hardware und Software, zu einem Gesamtsystem integriert. Oftmals werden in der Praxis System Integrationstests nicht als eigenständige Prozessschritte verstanden, sondern als Teil der System Qualifizierung durchgeführt in Form einer Black-Box artigen Validierung. Dabei demonstriert Abbildung 2.8 die folgenden Integrationsschritte, welche in [9] empfohlen werden:

- **Schritt 1:** Integration der entwickelten Software auf der Zielumgebung. Dabei sollen umgebungsspezifische Probleme, wie Fehler bei Speicherzuweisungen detektiert werden.
- **Schritt 2:** Integration der Controller Plattform mit den elektronischen Bauteilen samt Beschaltungen, um das Zusammenspiel aller Bauteile zu testen.
- **Schritt 3:** Integration der mechanischen Komponenten mit zugehörigen Tests.

Durch Abfolge dieser Prozessschritte soll beginnend mit einzelnen System Komponenten letztendlich das Gesamtsystem in einzelnen Integrationsschritten entstehen. Die Reihenfolge der Integrationsschritte bzw. der Integration einzelner Systemkomponenten kann dabei entscheidend sein, falls einzelne Komponenten die Integration anderer Komponenten beeinflussen. Dies verdeutlicht auch, dass bei System Integrationstests die Schnittstellenspezifikation von besonderer Bedeutung ist und bei Erstellen der System Architektur berücksichtigt werden muss. Dabei sollten die implementierten Tests auf folgendes fokussiert sein:

- Der korrekte Signalfluss zwischen System Komponenten im Hinblick auf Rechtzeitigkeit und zeitlichen Abhängigkeiten.
- Der korrekte Signalfluss zwischen System Komponenten im Hinblick auf Richtigkeit der Daten.

- Eine konsistente Interpretation aller Signalflüsse.
- Das dynamische Zusammenspiel zwischen den einzelnen System Komponenten.

Sinnhaft ist es System Integrationstests durch Konzepte wie ein Hardware In the Loop (HIL) System zu automatisieren. Bei den einzelnen Integrationsschritten wird des Weiteren empfohlen, diese mit einzelnen System Komponenten mit zugehörigen Prototypen durchzuführen und zu verifizieren. Passend zu diesen Prototypen Aufbauten werden dann dementsprechende Testfälle formuliert und die Ergebnisse genau dokumentiert. Dabei ist eine konsistente Zuordnung zu den System Requirements besonders hervorzuheben.
Das Konzept der Continious Integration kann auch auf System Ebene angewandt werden. Bei Modifikation einer System Komponente und nach Integration soll überprüft werden, ob das System noch den funktionalen Anforderungen entspricht. Dabei wird mittels sogenannten Regression Tests das System nach der Integration erneut getestet [4]. Die Ausführung von zugehörigen automatisierten Tests in Kombinationen mit der notwendigen Infrastruktur kann auch über einen Build Server mit erweiterter Verifikation getriggert werden [27].

3 Hardware/Software Integration

Oftmals wird die Entwicklung der Hardware Domäne eines Systems und der Software Domäne eines Systems in voneinander gelösten Prozessschritten durchgeführt. Doch in Embedded Systemen, bei welchen die Entwicklung hardwarenaher Software fokussiert wird, ist diese Entwicklung nicht gänzlich voneinander unabhängig. Die entwickelten Hardwarekomponenten müssen mit den zugehörigen Softwarekomponenten interagieren und somit integriert werden. Dabei wird die Software auf der Zielhardware getestet[4]. Hierbei werden auch Unit Tests auf der Zielhardware zu Hardware/Software Integrationstests gezählt. Eine Hardware/Software Integration wird oftmals im Zuge einer System Qualifizierung mitverifiziert und nicht explizite durchgeführt. Doch der Standard ISO 26262 empfiehlt eine solche explizite Integrationverifikation aus Sicherheitsgründen[11]. In der Automobilindustrie müssen bereits ab ASIL Level B Hardware/Software Schnittstellen mit einer ausreichenden Testabdeckung im Zuge der Integration verifiziert werden [10]. Dies umfasst auch Tests einzelner Software Komponenten auf der Zielumgebung. In höheren Ebenen des ASIL Standards werden noch weiter Verifiaktionmethoden Methoden, wie die Durchführung von Fault Injection Tests [28], vorgeschrieben. Diese überprüfen beispielsweise, ob die Modifiaktion der Software durch feindliche Zugriffe durch Security Maßnahmen abgefangen werden können [29]. Tests dieser ASIL Ebenen wurden in dieser Arbeit nicht behandelt.
In diesem Kapitel werden bekannte entwicklungsbegleitende Methoden für die Verifikation der Integration von Hardware und Software Komponenten vorgestellt. Dabei sollen Nachteile dieser Konzepte lokalisiert werden und mögliche Lösungsansätze gefunden werden. Ein Ausblick auf das letztlich umgesetzte Konzept soll geschaffen und dessen Auswahl begründet werden.

3.1 Methoden nach dem Stand der Technik

In diesem Abschnitt werden sowohl bekannte funktionale wie auch nicht funktionale Verifikationmethoden für Integration einer entwickelten Software in eine Hardwareplattform vorgestellt. In späterer Folge werden diese in Bezug auf die Ziele dieser Arbeit bewertet.

3.1.1 PIL Tests

Unter dem Processor In the Loop (PIL) Testkonzept versteht man die automatisierte Ausführung von Tests am Zielsystem [30]. Dabei ist es üblich funtkionale Tests, welche bei SIL Tests ebenfalls ausgeführt wurden, auf der Zielumgebung zu verifizieren um Fehler im Zusammenhang mit

dem Compiler oder Prozessorarchitektur festzustellen [26]. Bei PIL muss das Test Framework auch am Zielsystem ausgeführt, sowie die jeweiligen Resultate zwischengespeichert und an den Entwicklsrechner übertragen werden [31]. Erstellung von Laufzeitmessungen und Messung der Code Coverage können durch zusätzliche Instrumentierungen erreicht werden. Zur Interaktion zwischen Zielsystem und Entwicklungssystem wird eine Debugging Interface verwendet. Dabei können neben den herkömmlich verwendeten Universal Serial Bus (USB) Interface auch UART oder Ethernet als Interface zu Host Plattform verwendet werden. Bei der Automatisierung von PIL Tests werden dabei folgende Schritte durchlaufen:

1. Erstellung der ausführbaren Dateien
2. Download aufs Zielsystem
3. Testausführung
4. Ausgabe und Speicherung der Resultate am Entwicklungsrechner

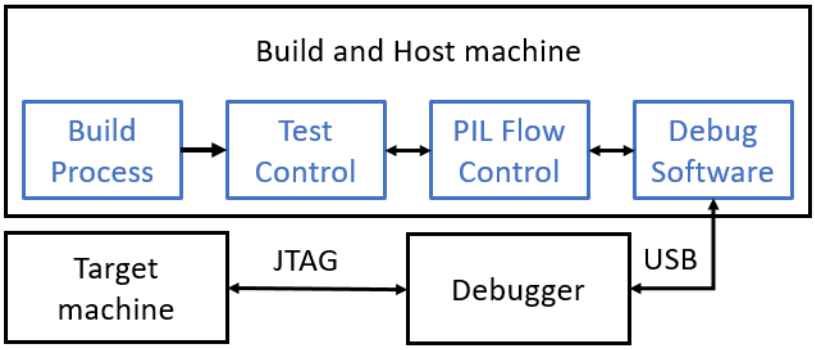

Abbildung 3.1: Ablauf eines PIL Tests

Abbildung 3.1 beschreibt den Ablauf eines PIL Tests, beginnend mit dem Build Prozess am Entwicklungsrechner, bzw. gegebenenfalls auf einem eigenen automatisierten Build Server. Im Zuge der Testablaufsteuerung werden mittels des Debuggers und des zugehörigen Frameworks die Testdaten an das Zielsystem übertragen. Diese Daten werden dann am Zielsystem geladen und am Prozessor ausgeführt. Letztendlich werden die Testresultate an den Entwicklungsrechner kommuniziert.

Beim Erstellen der ausführbaren Testdaten muss der sowohl der Zielsystem spezifische Start-Up Code, sowie das zu testende Module als auch die geschriebenen Tests mit zugehörigen Ersatzfunktion, z.B. Mocks, kompiliert und anschließend gemeinsam mit einer Test Bibliothek gelinkt werden. Abbildung 3.2 demonstriert dieses Prozedere, wobei sämtliche entwickelter oder generierten Software Code grün, zusätzlicher Maschinen Code orange und Tools für die Test Daten Generierung blau markiert sind. Hierbei werden spezifische Compiler, welche das Zielsystem unterstützen, benötigt.

Neben herkömmlichen funktionalen Tests, welche am Zielsystem ausgeführt werden und deren Ergebnisse zwischengespeichert und an den Entwicklungsrechner übertragen werden, können auch Methoden zur Messung von Laufzeiten und Code-Coverage angewendet werden. Dies benötigt allerdings eine zusätzliche Instrumentierung der Software auf der Zielumgebung.

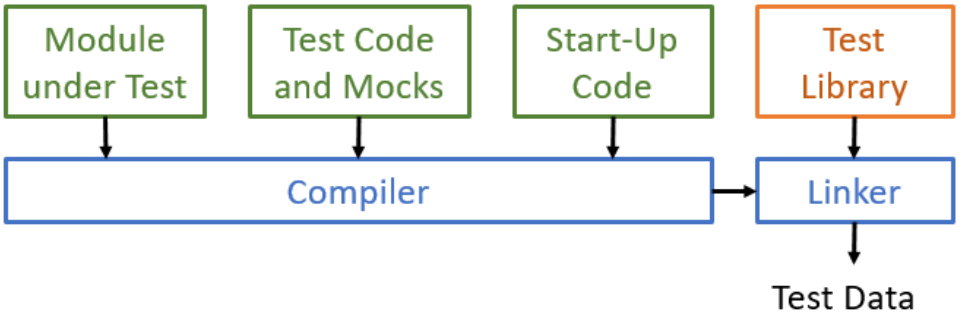

Abbildung 3.2: Build Prozess für PIL Testdateien

3.1.2 Resourcentests

Nicht funktionale Methoden, welche im Kontext der Hardware/Software Integration interessant sind, sind sogenannte Ressourcentests, wobei es um die Analyse von CPU Last und Speicherbedarf geht [4]. Dabei wird zwischen den folgenden zwei Arten unterschieden:

- **Statischer Ressourcentest:** Dieser wird ausgeführt, wenn zwar der Quellcode, allerdings nicht die Zielumgebung zur Verfügung steht. Dabei werden folgende Parameter überprüft:
  - **CPU-Last**: Die maximale Ausführungszeit von Programmen oder deren Abschnitte lassen sich mittels der sogenannten Worst Case Execution Time (WCET) analysieren [32].
  - **Statischer Speicher**: Der Umfang des Codes mit globalen Variablen kann durch das Map File, welches ein optionaler Output des Linkers ist, analysiert werden.
- **Dynamischer Ressourcentest:** Hier wird die Software am Zielsystem ausgeführt und der Speicherverbrauch und die CPU Last gemessen.
  - **CPU-Last**: Eine Variante für eine solche Messung ist es, eine Laufzeitmessung von einem Programmdurchlauf am Beginn und am Ende einen Pin zu toggeln und diesen mit einem Oszilloskop zu messen. Alternativ kann aber auch ein Instruktionszähler in die Software eingesetzt und ausgewertet werden. Dieser kann dann über einen sogenannten Profiler analysiert, oder zur Laufzeit über ein Tracing Module ausgewertet werden. Nachteil dabei ist jedoch, dass jede zusätzliche Instumentierung des produktiven Codes zu einer erhöhten Laufzeiterhöhung führt.
  - **Dynamischer Speicher**: Hier kann der verfügbare Adressraum mit einem dedizierten Wert befüllt werden. Zur Laufzeit kann regelmäßig überprüft werden, wie oft dieser Wert im Speicher liegt. Allerdings wird vor dynamischer Speicherplatz Allozierung im Heap-Speicherbereich in Embedded Systemen gewarnt, da deren Auslastung nicht vorhersagbar ist. Daher wird diese Methode zumeist im Stack-Speicherbereich verwendet.

3.1.3 Dynamische Echtzeit Analysen von Tasks auf der Zielumgebung

Für die Detektion von Scheduling Problemen und der damit einhergehenden Überschreitung von Deadlines der gemanagten Tasks, können Echtzeit Analysen automatisiert auf einer Zielumgebung umgesetzt werden [33]. Um die CPU Auslastung nicht zu steigern wird im Allgemeinen das

Debug Interface des Mikrocontrollers erweitert durch eine Tracing Hardware und Tools. Dabei muss der Joint Test Action Group (JTAG) der Debug Schnittstelle einer Kategorie im Standard IEEE-Industry Standards and Technology Organization (ISTO) 5001^{TM}-2003 (NEXUS) [34] entsprechen, welcher einen echtzeitfähigen Zugriff auf den Speicher des Mikrocontrollers im Betrieb unterstützt. Bei der Durchführung dieser Hardware Trace Analysen, welche im Vergleich zu Software Trace Verfahren ohne zusätzliche Instrumentierung der Software am Zielsystem auskommt, wird zwischen den folgenden beiden Typen unterschieden [35]:

- **Program Trace**: Hier wird die Ausführung von Funktionen aufgezeichnet. Dabei werden Events beim Eintritt und Austritt einer Funktion getriggert.
- **Data Trace**: Hier werden Daten im Speicher überwacht. Die Zuweisung zugehöriger Symbole der Software, wie z.B. globale Variablen, ist dafür notwendig.

Im Zuge der *Program Trace* Analyse werden aufgezeichneten Zeitverläufe der einzelnen Tasks werden dann auf vorab definierte Timing Requirements und Timing Metriken analysiert und ein Report erstellt. Auch ein Vergleich zu statischen Analysen der Ausführung von Tasks, wie der WCET Analyse, können zur Bewertung hergezogen werden. Basierend auf den Ergebnissen können Tasks dann neu angeordnet werden, um die Performance zu optimieren oder Überschreitungen auszumerzen.

3.1.4 Formale Verifikation von Hardware/Software Komponenten in einem System Modell

Bereits verifizierte Hardware und Software Komponenten können auch durch eine Modellierung des System Modells gemeinsam formal verifiziert werden [36]. Dabei wird die Zielumgebung, also ECU und Mikrocontroller, in einem System Modell abstrahiert, sowie die Funktionalitäten der verifizierbaren Software Komponenten. Wichtig zu beachten ist dabei die Einhaltung der genauen Hardware Architektur Spezifikationen. Dabei ist der Detaillierungsgrad im speziellen beim Abstrahieren der Hardware Ressourcen gefragt, welche von der Software gemanagt werden müssen. Konflikte basierend auf Probleme mit dem Speichermanagement oder anderer Ressourcen sollen dabei verdeutlicht werden. In diesen System Modellen können Methoden der Formalen Verifikation durchgeführt werden, um systemübergreifende Konflikte aufzuzeigen. SystemC[1] gilt hierbei als Industrie Standard Sprache für System Modellierungen. Die abstrahierte Hardware wird als Teil der Hardware Description Language (HDL) Programmierung integriert, während die Software Komponenten mit den zugehörigen C Bibliotheken ins System Modell eingebunden werden. Methoden der formalen Verifikation können dann an das erstellt Modell angewandt werden, um zu garantieren, dass das modellierte System in allen Systemzuständen valide ist [37]. Dabei werden Systemmodelle meist als Automaten dargestellt. Mögliche logische Grenzfälle sowie Fehlerzustände sind durch diese Methoden ermittelbar.

3.1.5 Messung des Energieverbrauches einer gezielten System Komponente

Automatisierte Messungen des Energieverbrauches einer Systemkomponente können im Zuge von Hardware/Software Integration Tests durchgeführt werden. Dazu werden Spannungs- und Strommessungen zur selben Zeit an Testpunkten der Schaltung durchgeführt. Oftmals wird hierbei die

[1] `https://www.accellera.org/downloads/standards/systemc`, Online: Letzter Zugriff am 20.01.2020

Systemkomponente extern versorgt. Für den messtechnischen Aufbau ist der Einsatz eines niederohmigen Shunt Widerstandes notwendig [38]. Abbildung 3.3 demonstriert den messtechnischen Aufbau. Mittels der Formel in 3.1 wird der Energieverbrauch der Systemkomponente berechnet:

$$P[W] = U * I = U_{\text{Sys}} * \frac{(U_{\text{DC}} - U_{\text{Sys}})}{R_{\text{Shunt}}} \tag{3.1}$$

Durch Spannugs- und Strommessungen lässt sich der Mittelwert der Leistung einer einzelnen System Komponente berechnen. Mittels der zugehörigen Software Komponente können elektronische Bauelemente unterschiedlich angesteuert werden und somit auch die Parameter des Energieverbrauches in Abhängigkeit von Steuerungsparametern analysiert werden. Die Grenzen der Anforderungen an das System in Bezug auf Energieverbrauch können ausgelotet werden.

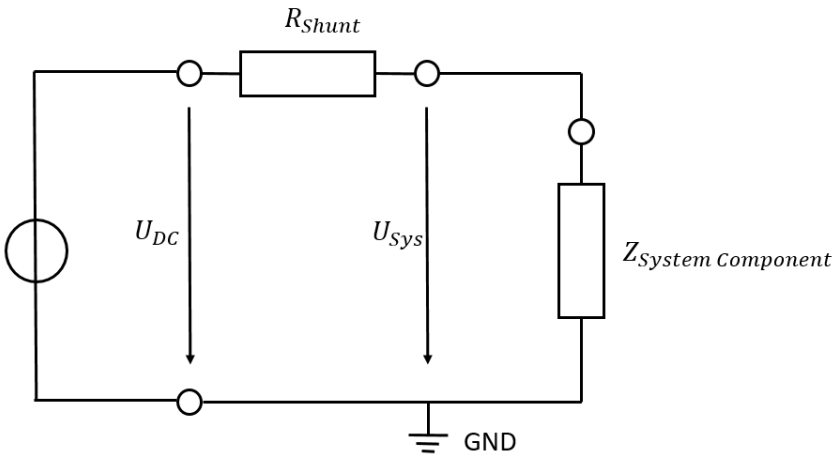

Abbildung 3.3: Messung des Energieverbrauches einer Systemkomponente

3.2 Problemstellung und konzeptionelle Lösungsansätze

Die hardwarenahe Entwicklung von Software in Embedded Systemen hebt die Notwendigkeit von Hardware/Software Integration Tests verstärkt hervor. So stellen z.B. Registerzugriffe im Code typische Hardware Flaschenhalse dar [39]. Besonders erschwert wird dieser Umstand, dass zumeist die Hardware- und Softwareentwicklung in Embedded Systemen parallel stattfindet, was auf Grund von oftmals kleinen Änderungen im Gesamtsystem zu Problemen bei der Integration zwischen Domänen führt. In [39] wird empfohlen, während der Entwicklung vorab am Entwicklungsrechner und danach auf der Zielhardware zu testen, ganz ohne Instrumentierung der Software. Falls die letztendliche Zielhardware noch nicht zur Verfügung steht, dann kann im ersten Schritt auch auf Evaluation Boards und in Simulatoren die Software getestet werden. Um möglichst effizient entwicklungsbegleitend zu testen, ist es des Weiteren ratsam, sämtliche Tests zu automatisieren. Für Hardware/Software Integration Verifikation wird ein zusätzlicher messtechnischer Aufbau benötigt, unabhängig davon, ob man mit der tatsächlichen Zielhardware oder mit Evaluation Boards testet. Unabhängig vom Verfahren, welches für solche Hardware/Software Integration Tests verwendet wird, müssen folgende zielumgebungsspezifische Eigenschaften [40] bei der Erstellung von Testfällen beachtet werden:

- **Datenerfassung**: Sämtliche Daten müssen durch die zu verifizierende Software Komponente erfasst und entsprechend ihres Umfanges im Speicher des Controllers abgelegt werden. Dabei muss ein Zugriff seitens des Test Setups gewährleistet sein.
- **Physikalische Signale**: Entsprechend der aktuellen Daten der Software Komponente müssen die jeweiligen physikalischen Signale an der Controller Schnittstelle ausgegeben und gemessen werden. Umgekehrt müssen auch physikalische Signale, welche an den Schnittstellen des Mikrocontroller instrumentiert werden durch die Software interpretierbar und letztendlich auf Daten im Speicher abbildbar sein.
- **Definitionsbereich**: Sämtliche Daten der Software, sowie physikalische Signale müssen Werten im Intervall ihres Definitionsbereichs entsprechen. Dementsprechend müssen die Ränder als Grenzwerte sämtlicher Daten und Signale spezifiziert sein.
- **Interrupts**: Die Unterbrechung bei Ausführung einer Funktion einer Software Komponente während eines Testes durch Interrupts muss verhindert werden. Dies gelingt in den meisten Fällen durch das Deaktivieren sämtlicher Interrupt Routinen. Ausnahmen bilden Funktionalitäten, deren korrekte Abarbeitung durch die Ausführung einer solchen Interrupt Routine gewährleistet sind.
- **Timing**: Bei der Durchführung funktionaler Software Tests auf der Zielumgebung ist es wichtig die Funktionen einer Komponente in der richtigen Reihenfolge auszuführen, um ihre korrekte Abarbeitung zu garantieren. Bei Hardware/Software Integration Tests kommt hinzu, dass die Instrumentierung der umgebenden physikalischen Signale berücksichtigt sein muss. Dabei können geplante Wartezeiten in der Ausführung eines Tests ein hilfreiches Mittel sein, um sicher zu stellen, dass physikalische Signale in korrekter Wechselwirkung mit den Registern der Software Treiber Komponenten sind.
- **Ressourcen Management**: Eine mehrfache Allozierung von Mikrocontroller Ressourcen in der Ausführung von Tests muss verhindert werden. Der Zugriff auf Speicherbereiche muss gemanagt werden, um valide Daten zu gewährleisten.

Ein bewährtes Integrationstestverfahren wird mittels des Bottom-Up Prinzips beschrieben, bei welchem vorab die nächsten Hardware Module in Hardware Abstraction Layer (HAL) Software Module gekapselt, entwickelt und verifiziert werden. Beginnend bei einer Modulverifikation auf der Zielhardware können nun Hardware und Software Module schrittweise ergänzt werden und durch eine dementsprechende Parametrisierung verifiziert werden.
Um die funktionale Richtigkeit von System Komponenten ausreichend verifizieren zu können, müssen die Hardware und Software Komponenten, welche in diesem Kontext in Verbindung stehen, hinreichend integriert werden. Diese Integration soll schrittweise getestet werden, wozu sich ein Bottom Up Prinzip anbietet. Ziel dieser Arbeit ist es, eine Plattform zur Durchführung von automatisierten funktionalen Hardware/Software Integration Tests zu entwickeln. In diesem Abschnitt werden die dafür möglichen Verfahren ermittelt und bewertet. Ein Ausblick auf letztliche Umsetzung soll geschaffen werden.

3.2.1 Hardware/Software Integration Tests auf der Zielumgebung

Beim Konzept der Implementierung von Tests auf der Mikrocontroller Plattform selbst ist eine bewährte Methode PIL Tests am Zielsystem zu etablieren und mittels einer Messinstrumentierung in einer automatisierten HIL Umgebung Signale auszuwerten oder zu triggern [41]. Abbildung 3.4 veranschaulicht das Prinzip. Hierbei wird die Host Plattform verwendet, um sämtliche Messdaten des HIL Systems über eine Debug Schnittstelle an die Tests auf der Target Plattform zu kommunizieren [42]. Diese Mess- und Instrumentierungsdaten werden im Zuge der ausgeführten Tests ausgewertet und analysiert. Das Host System wird des Weiteren dazu verwendet, um die Testausführung zu triggern und die Ergebnisse der Testläufe von der Controller Plattform an den Benutzer zu übertragen. Zusammengefasst bedeutet dies, dass die Kommunikation zwischen Host Rechner und Zielsystem über das Debug Interface die folgenden Aufgaben hat:

- Start der Tests auf der Zielumgebung.
- Übertragung der Testergebnisse von Zielsystem an Entwicklungsrechner.
- Übertragung der Messergebnisse des HIL Test Systems über Host zu Zielsystem zur Auswertung von physikalischen Signalen.
- Triggern von physikalischen Signalen am HIL Test System zur Messung und Auswertung auf der Zielumgebung selbst.

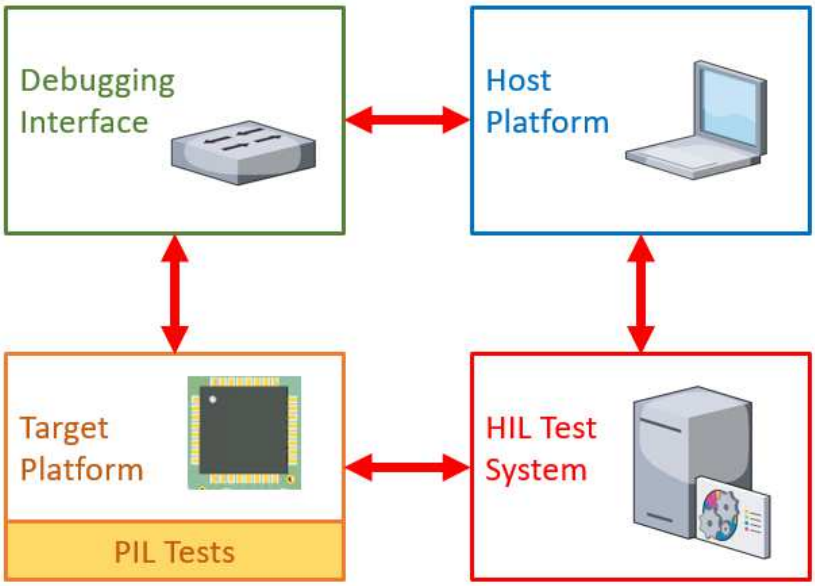

Abbildung 3.4: Hardware/Software Integrationstest basierend auf PIL Tests

Teil dieser Methode ist die Möglichkeit der Implementierung von Code-Coverage Analysen, deren Ergebnisse Teil der erstellt funktionalen Testergebnisse selbst sein könnten. Sowie die Testergebnisse können auch die Ergebnisse der Code-Coverage Analyse am Entwicklungsrechner dann interpretiert werden.

Vorteil dieser Methode ist, dass ein Echtzeitzugriff auf den Speicher des Mikrocontrollers mittels des Dubuggers nicht notwendig ist. Sämtliche Zugriffe werden am Controller selbst gemanagt und das Auslesen des entstandenen Test Reports muss nicht in Echtzeit erfolgen. Das bedeutet, dass weder der Debugger noch der Mikrocontroller eine JTAG Schnittstelle benötigen, welche einen Echtzeitzugriff auf den Speicher gewährleistet. Diese Methode ist im Allgemeinen bei Mirkocontrollern mit einem weniger komplexen Debug Schnittstelle nach dem IEEE-ISTO 5000^{TM}-2003 (NEXUS) [34] anwendbar.
Nachteil an dieser Methode sind die zusätzlichen Ressourcen wie Speicherbedarf, welche die Testdaten am Zielsystem benötigen. Im Speziellen ist in ressourcenoptimierten Embedded Mikrocontroller Zielsystemen ist Ressourcenoptimierung auch bei der schrittweisen Integration von Software Modulen und Komponenten zu berücksichtigen. Des Weiteren ist eine komplexe Synchronisierung zwischen dem HIL Test System und den PIL Tests auf der Ziel Umgebung notwendig [43]. Diese Synchronisierung müsste am Entwicklungsrechner gemanagt werden. Hierfür ist die Verwendung einer Debugger Hardware mit zugehörigen Framework notwendig, welches die Möglichkeit bietet, auf sämtliche Symbole einer Software, wie globale Variable und Funktionsaufrufe, zugreifen zu können.

3.2.2 Hardware/Software Integration Tests auf der Entwicklungsumgebung

Als Alternative, um Integrationstests automatisiert durchführen und analysieren zu können, können Module der zu testenden Software Komponenten auf der Zielhardware parametrisierbar und durch Funktionsaufrufe auf der Entwicklungsumgebung angesteuert werden. In [18] werden dabei sowohl diskrete DIO Signale instrumentiert, als auch die physikalischen Signale von Kommunikation Interfaces interpretiert. Abbildung 3.5 veranschaulicht dieses Prinzip. Hierfür ist wiederum die Verwendung einer Debugger Hardware mit zugehörigen Framework notwendig. Des Weiteren ist eine messtechnische Auswertung und Instrumentierung in einem HIL Test System notwendig, welche wiederum mit den Tests in der Entwicklungsumgebung in Wechselwirkung treten. Die tatsächlichen Test Skripten mit deren Auswertung und Interpretation werden am Host ausgeführt.
Teil dieser Methodik ist es des Weiteren, dass Code-Coverage Analysen ohne Instrumentierung auf der Zielumgebung realisiert werden müssen. Dies ist allerdings nur durch eine weiterer Trace Schnittstelle als Teil des JTAG möglich. Diese wird in der jeweiligen Kategorie des IEEE-ISTO 5001^{TM}-2003 Standard beschrieben [44]. Damit ist es einerseits möglich durch die Zuweisung von Speicherbereich die dort liegenden Daten in Echtzeit aufzuzeichnen [45]. Andererseits können auch Instruktionen in der Programmausführung aufgezeichnet werden. Somit können Ausführungspfade der Software erfasst und analysiert werden. Rückschlüsse auf den Programmfluss können gezogen werden, ohne eine Instrumentierung am Mikrocontroller selbst vorzunehmen. Sowohl Mikrocontroller als Debugger müssen die jeweiligen Hardware Treiber Bauteile zur Verfügung stellen um eine Code-Coverage Analyse über den Entwicklungsrechner umzusetzen [46].
Vorteil dieser Methodik ist, dass keine zusätzlichen Ressourcen des Controllers alloziert und sämtliche Tests direkt am Entwicklungsrechner durchgeführt werden. Auch die Initialisierung und Synchronisierung sämtlicher Module des HIL Testsystems, sowie der Mikrocontroller Zielumgebung werden durch die Host Plattfrom gemanagt.
Als Einschränkung dieses Konzepts gilt die Voraussetzung eines Echtzeitzugriffes auf den Speicher des Mikrocontrollers, um die Symbole der Software in Echtzeit auswerten und manipulieren

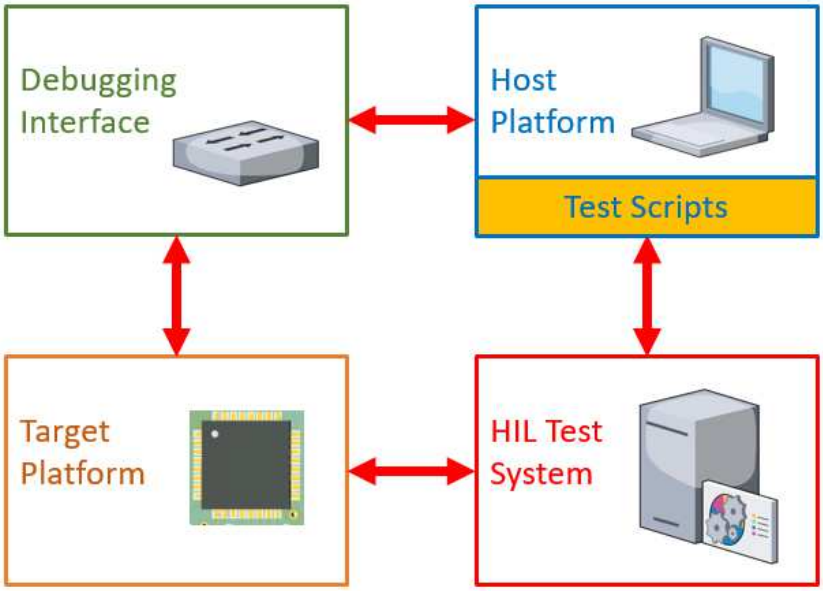

Abbildung 3.5: Hardware/Software Integrationstest basierend Test Skripten auf einem Host Rechner

zu können. Der JTAG IP-Core, welcher am IC synthetisiert ist, muss dabei einer Kategorie des IEEE-ISTO 5001^{TM}-2003 Standard entsprechen, welcher einen sogenannten Test Access Point (TAP) Controller [44] als Debug Feature verlangt. Das bedeutet, dass diese Methode nur auf Mikrocontrollern anwendbar ist, dessen TAP Controller einen Zugriff auf die Speicherelemente des Mikrocontrollers zur Laufzeit gewährleistet.

4 Konzept und Umsetzung

Als gewählte Methode zur Umsetzung einer Hardware/Software Integration Testing Plattform wurde die Implementierung von Tests auf der Host Plattform gewählt. Die möglichst rohen Software Componenten ohne zusätzlichen nicht produktiven Code sollen auch auf ressourcenoptimierten Mikrocontrollern mit geringer Speichergröße testbar sein. Um eine dedizierte Hardware/Software Komponente durch Hardware/Software Integration Tests zu verifizieren, werden die einzelnen Module nach dem Bottom-Up Prinzip integriert und schrittweise verifiziert. Abbildung 4.1 demonstriert diese Schritte:

1. **Low Level Driver Unit Hardware Software Integration**: Software Komponenten des Basis Software Layers einer herkömmlichen Embedded Software Architektur werden verifiziert. Diese LLD Komponenten werden durch funktionale Tests auf der Mikrocontroller Zielumgebung geprüft.

2. **Basis Software Komponenten Integration**: Software Komponenten, welche den Zweck der Interaktion mit einer beschalteten Umgebungshardware erfüllen sollen, werden in der Mikrocontroller Zielumgebung Integriert. Durch funktionale Tests wird dieser Software Integrationschritt verifiziert.

3. **Basis Hardware/Software Komponenten Integration**: Die Verifikation von zusammengehörigen Hardware/Software Komponenten auf der Ziel-Hardware wird durchgeführt. Die Integration der Software am Mikrocontroller, mit zusätzlichem Evaluation Board des zugehörigen elektronischen Hardware Treibers, wird durch funktionale Tests verifiziert.

In dieser Arbeit wurde eine Plattform für die Ausführung von Hardware/Software Integration Tests, welche Schritte in Abbildung 4.1 ermöglicht, mit der Methode der Ausführung der Tests auf der Host Plattform entwickelt. Die Verwendung der entwickelten Plattform wird in diesem Kapitel anhand von Beispielen erläutert. Basierend auf das Anwendungsgebiet wurden Anforderungen an die experimentelle Umsetzung formuliert.

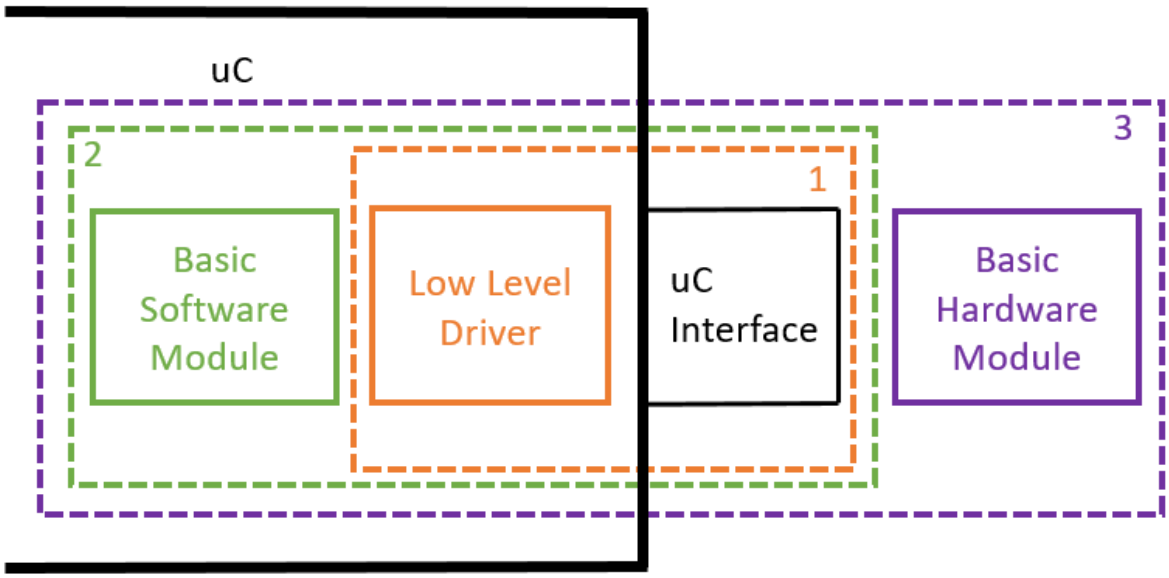

Abbildung 4.1: Hardware Software Integration Schritte

4.1 Anwendungsbeispiele

In diesem Abschnitt werden für die zuvor angeführten Schritte von Hardware/Software Integration Tests Beispiele beschrieben. Diese Bespiele decken sämtliche Ebenen des Bottom-Up Integration Testing Prinzips ab und verdeutlichen die Notwendigkeit von Hardware/Software Integration Tests für die Entwicklung von Embedded Systemen. Bei sämtlichen Beispielen wird ein Buck Converter [47] als System Komponente für eine Stromregelung eines Automotive Steuergeräts beispielhaft herangezogen. Dabei wurde der ASPICE Standard [2] für die prozessorientierte Entwicklung eines Embedded System in der Automotive Industrie und die ISO 26262 [48] als Safety Standard beachtet.

4.1.1 Fallbeispiel Bauteil Evaluierung

Ein neuer und günstiger Buck Converter soll verwendet werden. Die Entwicklung dieser System Komponente soll in zwei Schritten geschehen:

1. Bevor eine endgültige Entscheidung über den Einsatz dieses Bauteils getroffen wird, soll überprüft werden, ob es gewisse funktionale Anforderungen für das Steuergerät tatsächlich bietet. Dafür wird eine einfache Version einer zugehörigen Software Komponente implementiert. Die Software Komponente ist auf einem Mikrocontroller Evaluation Board des Mikrocontrollers, welcher fürs Steuergerät eingesetzt wird, lauffähig. Mittels der kontaktieren Pins des Evaluation Boards wird der Buck Converter, welcher sich auf einem eigenen herstellerspezifischen Evaluation Board befindet, angesteuert. Diese Kombination zwischen den beiden Demoboards wird Prototypen Aufbau genannt, siehe Abbildung 4.2. Mittels automatisierten Hardware/Software Integration Tests sollen diese Funktionalitäten verifiziert werden.

2. Falls sämtliche Grundfunktionalitäten verifiziert wurden und der Buck Converter tatsächlich am Steuergerät eingesetzt werden soll, dann wird auf Basis der Software Komponente des Schrittes 1 die Software Komponente weiterentwickelt und kontinuierlich am Prototypen verifiziert. Automatisierte Hardware/Software Tests können in diesem Fall kontinuierlich ausgeführt werden, obwohl die endgültige Steuergerät Hardware mit dem neuen und günstigeren Buck Converter noch nicht entwickelt ist. Damit ist die Entwicklung der zugehörigen Software Komponente planbarer und somit auch zeit- und kosteneffizienter. Fehler oder mögliche Probleme bei der Entwicklung der System Komponente können frühzeitig detektiert werden.

In dieser Arbeit soll für das Fallbeispiel der Bauteil Evaluierung ein Ausblick geschaffen werden. In der Evaluierung der entwickelten Hardware/Software Integration Plattform in Kapitel 7 werden Tests im Umfang dieses Fallbeispiels nicht betrachtet.

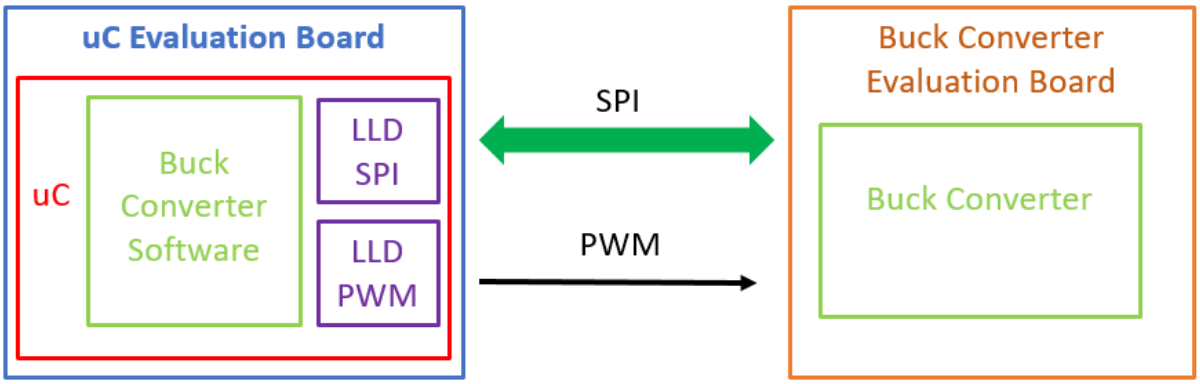

Abbildung 4.2: Prototyp für die Buck Converter Evaluierung

4.1.2 Fallbeispiel Entwicklung von Low-Level Treiber Modulen

Wie in Abbildung 4.2 ersichtlich, benötigt die Software Komponente des Buck Converters zur Ansteuerung des Bauteils selbst ein SPI als serielles Kommunikationsprotokoll, um die dementsprechenden Register am IC zu setzen bzw. auszulesen und zu analysieren. Des Weiteren ist es möglich, ein Puls Signal mittels eines Pulse Wide Modulation (PWM) Signals am Strom Kanal einzuprägen. Dazu benötigt man ebenso ein SPI Modul als auch ein PWM Modul in der Software. Will man diese selbst entwickeln um z.B. Kosten zu sparen, dann müssen diese ausreichend verifiziert werden. Diese Verifikation sollte auch auf der Mikrocontroller Zielumgebung stattfinden, um plattformspezifische Einflüsse zu testen. Dazu muss allerdings bloß die linke Seite der Abbildung 4.2 als Teil des Prototypen Aufbaus betrachtet werden.
Angemerkt sei hier, dass diese Low-Level Treiber so generisch als möglich entwickelt werden sollten, um diese möglichst vielseitig einsetzen zu können. Dabei kann diese generische Entwicklung sich auch auf die Plattformunabhängigkeit beziehen, das heißt, die Lauffähigkeit auf möglichst viele verschiedenen Mikrocontrollern. Dafür muss bloß auf sämtlichen Controllern der selbe IP Core mit dem strukturellen gleichen Aufbau des Register Sets verwendet werden. Bei der Entwicklung dieser Module sollte des Weiteren auch die Einhaltung der umgebenden Industriestandards berücksichtigt werden.
Für die Evaluierung der entwickelten Hardware/Software Integration Tests Plattform in Kapitel 7 wurden LLD Software Komponenten auf einer Mikrocontroller Zielumgebung, entsprechend der

Testaufbauten in Kapitel 6, verifiziert. Die Evaluierung entspricht damit der Verifikation, welche im Fallbeispiel der Entwicklung von LLD Treiber Modulen beschrieben ist. Somit wurde in dieser Arbeit die Machbarkeit von Tests zu diesem Fallbeispiel erprobt und bestätigt.

4.1.3 Fallbeispiel Systemintegration

Der entwickelte Buck Converter soll als Teil einer System Architektur als System Komponente integriert werden. Abbildung 4.3 beschreibt diesen Prozess als Use-Case Diagramm. Dabei stellen der System Ingenieur, der Software Ingenieur und der Hardware Ingenieur die Akteure dar. Der System Ingenieur entwickelt anhand der definierten System Requirements die System Architektur. Dabei wird der Buck Converter als System Komponente identifiziert bestehend aus einer Hardware und einer Software Komponente. Der Hardware Ingenieur wird beauftragt das Schaltungsdesign zum Bauteil zu entwickeln und zu verifizieren. Der Software Ingenieur wird mit der Entwicklung und Verifikation einer Mikrocontroller Software Komponente zur Ansteuerung des Buck Converters beauftragt. Hier sei erwähnt, dass es zumeist Sinn macht, die Verifikation sowohl der Hardware als auch der Software Komponente durch eigene Verifikation Ingenieure durchzuführen, da diese die Komponenten mittels Black-Box Tests gegen die Hardware und Software Requirements testen.

Die entwickelte und verifizierte Hardware und Software Komponente muss nun einerseits zu ei-

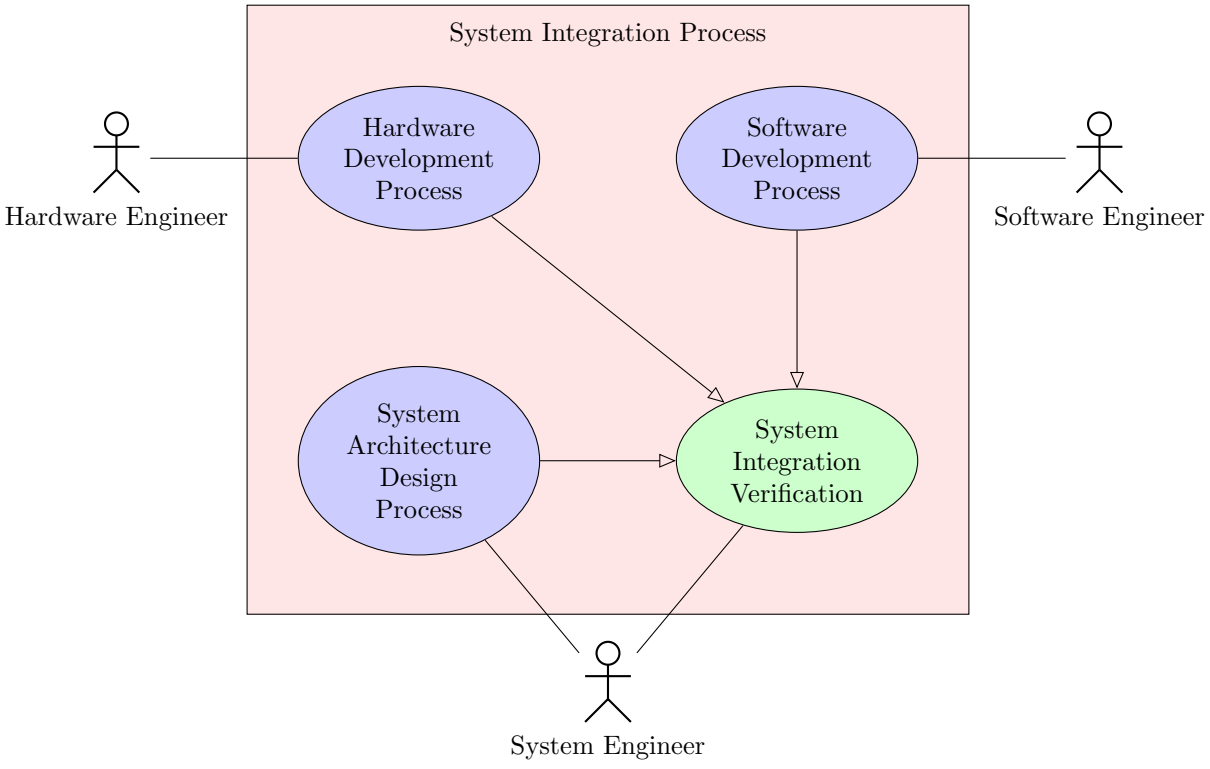

Abbildung 4.3: Fallbeispiel System Integration

ner Hardware/Software Komponente integriert werden. Andererseits muss die entstehende System Komponente auch in das Gesamtsystem integriert werden. Dies wird in Abbildung 4.3 durch den grünen Prozess dargestellt, welcher der Kernprozess des Use Case Grafen darstellt. Sämtliche anderen blau dargestellten Prozesse arbeiten dem *System Integration Verification* Prozess zu. Diese

domainenübergreifende Integration muss verifiziert werden, um den Prozess und Safety Standards in der Automobil Industrie zu entsprechen. Dazu muss der System Ingenieur die System Schnittstellen in der Architektur Beschreibung genau spezifizieren und White-Box Tests für den System Integration Prozess formulieren. Erst wenn die Integration sämtlicher System Komponenten hinreichend verifiziert ist, kann das Gesamtsystem gegen die System Requirements verifiziert und gegen die Kunden Requirements validiert werden. Dafür werden wiederum eigene Verifikation- und Validierungs Ingenieure beauftragt, um Black-Box Tests mit dem entwickelten Gesamtsystem durchzuführen.
In dieser Arbeit soll für das Fallbeispiel des System Integration Prozesses ein Ausblick geschaffen werden. In der Evaluierung der entwickelten Hardware/Software Integration Plattform in Kapitel 7 werden Tests im Umfang dieses Fallbeispiels nicht betrachtet.

4.2 Anforderungen an die experimentelle Umsetzung

Um den Safety Standards der ISO 26262 [11] für automatisierte Hardware/Software Integration Verifikationen in einer ASPICE Prozesslandschaft [2] bei der Entwicklung einer Hardware/Software Integration Test Plattform gerecht zu werden, muss die entwickelte Plattform den Spezifikationen entsprechenden Anforderungen genügen. Des Weiteren soll die entwickelte Plattform sämtliche Möglichkeiten für die Anwendungsgebiete des Abschnittes 4.1 bieten. Um dieses Ziel zu erreichen, wurden für die Plattform, welche im Zuge dieser Arbeit entwickelt werden soll, folgende Requirements formuliert:

- Eine Plattform mit Framework für die Verifikation von Low-Level Software Treiber Komponenten auf der Zielumgebung soll entwickelt werden. Die Plattform soll weiterführend für Software Integration und Hardware/Software Integration Test auf der Zielumgebung anwendbar sein.
- Zur Interaktion mit der Mikrocontroller Software sollen Hardware, Software und Tools der Firma *iSystem*[1] verwendet werden.
- Das HIL Test System mit zugehöriger Software der Firma *Vector*[2] soll verwendet werden.
- Die Tests selbst sollen mit der Programmiersprache C# entwickelt werden. Dazu soll ein Dynamic Link Library (DLL) entwickelt werden, welche die Anbindung an die Interaktion mit der Mikorcontroller Software und die Anbindung an das HIL Test System abstrahiert.
- Als Prototyp sollen die Low-Level Software Treiber Komponenten auf einem SPC560B54L5 [49] Mikrocontroller mit zugehörigen Evaluation Board [50] der Firma *STMicroelectronics*[3] verwendet werden. Der JTAG des SPC560B54L5 besitzt allerdings keine Trace Schnittstelle. Daher wurden Code-Coverage Analysen nicht im Zuge der Evaluierung an diesem Prototypen durchgeführt.
- Sämtliche Tests sollen automatisiert betrieben werden können. Dabei soll sowohl der Testoutput und die Testimplementierung selbst auf die Testfallbeschreibung und zu dem zugehörigen Requirement, verlinkt werden können, um die Rückverfolgbarkeit zu gewährleisten.

[1] `https://www.isystem.com/`, Online: Letzter Zugriff am 16.10.2019
[2] `https://www.vector.com/`, Online: Letzter Zugriff am 16.10.2019
[3] `https://www.st.com/`, Online: Letzter Zugriff am 16.10.2019

Entsprechend der formulierten Requirements wurde eine experimentelle Implementierung durchgeführte, welche in Kapitel 5 im Detail beschrieben wird. Dabei wird die ausgewählt Hardware des HIL Testsystems sowie des Debuggers beschrieben, als auch deren Integration im entwickelten *Hardware/Software Integration Verification* Framework. Das entwickelte Framework ist als Klassenbibliothek realisiert, dessen Architektur, sowie die Funktionalität der einzelnen Klassen in Kapitel 5 beschrieben ist. Eine Beschreibung wie das Framework als Teil des entwickelten Testsystem wird ebenfalls angeführt.
Bezugnehmend auf das gewählte Konzept, welches in Abbildung 3.5 als Hardware/Software Integration Tests mit Testskripten auf der Host Plattform dargestellt ist, umfasst die entwickelte Testplattform sämtliche Teile der Darstellung als eigenständige Hardware. Diese Teile sind in einem experimentellen Setup über Kabelstränge und Kommunikationskanäle miteinander verbunden. Die tatsächliche Logik als *Hardware/Software Integration Verification* Framework ist auf des Host Plattform realisiert. Dabei umfasst diese Logik die Initialisierung des Testsystems, das Management sämtlicher Hardware Komponenten des Testsystems sowie die Ausführung und Auswertung der tatsächlichen Tests.

5 Implementierung der Hardware/Software Integration Testing Plattform

In diesem Kapitel wird auf die konkrete Umsetzung im Zuge dieser Arbeit eingegangen. Entsprechend der formulierten Anforderung wird eine Beschreibung der zu verwendeten Hardware, Tools und Frameworks angeführt. Die Software Architektur des entwickelten Frameworks wird im Anschluss daran beschrieben. Sämtliche Teilaspekte dieses Frameworks werden detailliert erörtert. Als Teilaspekte werden die physikalischen Aufbauten des HIL Systems mit Debugger beschrieben, sowie eine entwickelte *Hardware/Software Integration Verification* DLL als umgesetztes Framework. Dabei steht die entwickelte Library in Wechselwirkung zu DLLs, welche sowohl als Schnittstellenabstarktion zum HIL Testsystem als auch zum Debugger dienen. Diese Wechselwirkung wird in der DLL abstrahiert, welche im Zuge dieser Arbeit entwickelt wurde. Sämtliche in Kapitel 5.3 beschrieben Teile der DLL sowie die zugehörigen Code Bespiele sind im Zuge der Entwicklung der *Hardware/Software Integration Verifikation* DLL entstanden.

5.1 Verwendete Hardware, Tools und Frameworks

Bei der Auswahl der verwendeten Hardware und Tools wurden neben den formulierten Requirements in Abschnitt 4.2 auch weitere folgende Kriterien beachtet. So ist notwendig, dass auf die Symbole der gebuildeten Software Komponenten, welche mittels des generierten Executable and Linkable Format (ELF) Files aufgelöst werden können, in Echtzeit über den Debugger zugegriffen werden kann. Des Weiteren ist es notwendig, dass sämtliche relevante physikalische Signale, welche für die Evaluierung der ausgewählten Low-Level Treiber (Kapitel 7) benötigt werden, auch instrumentierbar und messbar vom HIL Test System sind.
Im Allgemeinen lassen sich Limitierungen bei der praktischen Umsetzung von automatisierten Tests durch deren Wartbarkeit und Entwicklungsaufwand feststellen [51]. Doch um die Vorteile von automatisierten Tests, wie Erhöhung der Produktqualität, der Testabdeckung, Zuverlässigkeit, Wiederverwendbarkeit sowie die Reduktion von Kosten und Zeit bestmöglich umzusetzen, muss die Tool- und Hardwareauswahl genau vollzogen werden. Dabei müssen alle bekannten Merkmale eruiert werden, auch jene, welche nicht im direkten Zusammenhang wie dem tatsächlichen Testsystem stehen [52]. So musste in dieser Arbeit z.B. ein Compiler und eine Entwicklungsumgebung für die Ausführung der Tests am Entwicklungsrechner ausgewählt werden. Ein weiterer Compiler mit Entwicklungsumgebung ist für den Build Prozess der Mikrocontroller Software notwendig. Dabei wird als Teil dieser Entwicklungsumgebung controllerspezifische

Basissoftware mit Start-Up Code generiert. Das Prinzip der Compilierung auf einem Entwicklungsrechner für eine Zieumgebung wird Cross-Compilation genannt [4]. Auch Tools zur Test Reportage und für das Requirement- und Test-Management wurden verwendet.
Im Folgenden werden die wesentlichsten Teile der im Betrieb genommenen Hardware, Tools und Frameworks in diesem Projekt genauer beschrieben.

5.1.1 Debugger mit Framework zur Interaktion mit Software auf Zielplattform

Als Schnittstelle zwischen dem JTAG des Mikrocontroller Boards und dem Enticklungsrechner wird der On-Chip Analyzer [53] in der Abbildung 5.1 verwendet. Dieser Debugger der Firma *iSystem* unterstützt eine Vielzahl von 32-Bit Prozessorarchitekturen unterschiedlichster Hersteller. Neben herkömmlichen Debug Funktionalitäten bietet der On-Chip Analyzer auch eine externe Flashspeicherprogrammierung sowie einen Laufzeitzugriff auf Variablen, Registern und den Speicher. Ein echtzeitfähiger Zugriff auf Variablen, Registern und den Speicher ist auch notwendig für die Umsetzung von Hardware/Software Integration Tests auf der Host Plattform, da diese Funktionalität von den Tests verwendet wird. Die zugehörige Debugger Software *WinIDEA*[1] managte diese Funktionalitäten. Zur dynamischen Anbindung der Funktionen dieser Software an eine Testumgebung wird das Framework *isystem.connect* Software Development Kit (SDK)[2] verwendet. Diese unterstützt eine Vielzahl an Programmiersprachen, darunter auch C#.

Abbildung 5.1: *On-Chip Analyzer iC5000* von der Frima *iSystem*[3].

[1] `https://www.isystem.com/products/software/winidea.html`, Online: Letzter Zugriff am 16.10.2019
[2] `https://www.isystem.com/downloads/winIDEA/SDK/iSYSTEM.Python.SDK/documentation/isystem-connect-api/index.html`, Online: Letzter Zugriff am 16.10.2019

5.1.2 HIL Test System

Das verwendete HIL Test System besteht aus einzelnen Modulen der Firma *Vector*, welches ein modulares Messinstrumentierungssystem für die Interaktion mit den Input/Output (I/O) Verbindungen des sogenannten Device Under Test (DUT) ermöglicht. Dabei können an diesen Schnittstellen unterschiedlichste physikalische Signale getriggert und gemessen werden. Diese HIL Testsysteme werden im Zuge von automatisierten Tests angewandt. Abbildung 5.2 beschreibt das Prinzip dieses HIL Test Systems [54]. Das abgeschlossene HIL System von Vector wird *VT System*[4] genannt. Abbildung 5.3 zeigt ein solches *VT System*.
Sensoren und Aktuatoren, welche mit dem DUT interagieren, können zusätzlich an das *VT System* angebunden, oder durch dieses simuliert werden. Im Beispiel der ECU in Abbildung 1.1, welches hier das DUT darstellt, ist die eigenständige PCB des Licht Moduls als Aktuator im System zu sehen. Da die LED Stränge als aktive Stromlast zu interpretieren sind, müssen diese für eine korrekte Funktionalität der ECU kontaktiert oder im *VT System* simuliert werden. Die vollständige Abbildung der Umgebung der ECU ist eine notwendige Anforderung für die korrekte Durchführung von automatisierten Tests in einem HIL Testsystem. Auch die Kontaktierung oder Simulation von Sensoren, wie z.B. Negative Temperature Coefficient (NTC)s für Temperaturmessungen, sind eine gleichartige Anforderung.
Selbiges gilt auch für weitere Geräte, wie andere ECUs, welche mit dem DUT interagieren. Die Anbindung an einen gemeinsamen Feldbus, wie den CAN in Abbildung 1.1, wird zumeist simuliert, da die einzelnen ECUs unabhängig voneinander entwickelt und verifiziert werden [55]. Die Nachrichten der simulierten ECUs werden am HIL erzeugt und versendet.
Das Software Tool, welches mit dem *VT System* kommuniziert, heißt *CANoe*[5]. Mit diesem Tool können einerseits Feldbusanalysen und Simulation durchgeführt werden. Andererseits kann die Instrumentierung und Messung im *VT System* gemanagt werden. Dabei ist es möglich, automatisiert Tests zu implementieren und programmierte Testmethoden zu integrieren. Ein weiterer Teil dieses HIL Test Systems ist die Simulation von Fehlerzuständen beim DUT, wie Leitungsunterbrechungen und Kurzschlüsse. Auch der Umgang mit Fehlerbehandlung des DUT soll damit geprüft werden.
Wie bereits erwähnt besteht das *VT System* aus mehreren Modulen [8]. Diese werden im Folgenden beschrieben:

- **Load and Measurement Modul**: Dieses Modul wird dazu verwendet, um Spannungswerte des DUT zu messen und zu verarbeiten. Es könnten auch PWM Parameter, wie Frequenz, Duty Cycle und High/Low Pegel, ermittelt werden. Diese Messungen basieren auf individuellen Zeitmessungskonditionen. Dem DUT zugehörige Aktuatoren können kontaktiert oder simuliert werden. Des Weiteren können elektrische Fehler detektiert werden.

- **General-Purpose DIO Modul**: Ein großes Set an DIO Signalen ist mit diesem Modul messbar und instrumentierbar. Dabei ist der Low Pegel stets das Ground (GND) Potential des DUT und der High Pegel entspricht der DUT Versorgungsspannung oder einem externen Pegel. Dabei sind auch Schwellwerte konfigurierbar.

[3] `https://www.isystem.com/products/hardware/on-chip-analyzers/ic5000.html`, Online: Letzter Zugriff am 28.02.2020

[4] `https://www.vector.com/de/de/produkte/produkte-a-z/hardware/vt-system/`, Online: Letzter Zugriff am 16.10.2019

[5] `https://www.vector.com/at/de/produkte/produkte-a-z/software/canoe/`, Online: Letzter Zugriff am 16.10.2019

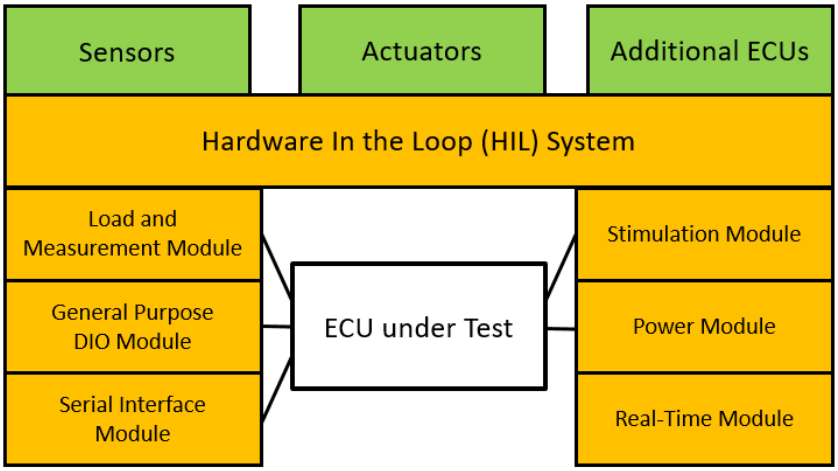

Abbildung 5.2: HIL Test System Konzept

- **Serial Interface Modul**: In diesem Modul werden serielle Schnittstellen behandelt, welche im Allgemeinen für die Kommunikation zwischen ECUs und Sensoren verwendet werden. Dabei können folgende serielle Schnittstellen mit zugehörigen Protokollen verarbeitet werden:
  - SPI
  - UART
  - Recommended Standard (RS) 232
  - RS 485
  - RS 422
  - Inter Integrated Circuit (I2C)
  - Low Voltage Differential Signaling (LVDS)

  In dieser Arbeit wurde allerdings bloß das SPI Interface näher betrachtet. Das *VT System* kann dabei sowohl als SPI Master als auch als SPI Slave konfiguriert werden. Mittels fünf Chip Select (CS) Leitungen können bis zu fünf SPI Kanäle angesprochen, bzw. simuliert werden.

- **Stimulation Modul**: Dieses Modul hat die Möglichkeit analoge Spannungswerte, PWM Signale oder eine Widerstands Dekade zu simulieren, was im Allgemeinen für Sensor Simulationen verwendet wird.

- **Power Modul**: Dieses Module wird für die elektrische Spannungsversorgung des DUT verwendet. Dabei lassen sich mehrere unterschiedliche Potentiale einstellen, sowie eine Verbindung zu einer externen Spannungsversorgung aufbauen. Auch eine zugehörige dynamische Strommessung zur Implementierung einer Sicherheitsabschaltung ist möglich. Auf konsistente Verbindung zum GND Potential muss geachtet werden.

- **Real-Time Modul**: Mittels des Real-Time Moduls wird eine High-Performance Plattform zur Verfügung gestellt. Damit sollen echtzeitfähige Tests und Simulationen durchgeführt werden. Die Treiber sämtlicher anderer Module im *VT System* können in diesem Modul

angesprochen werden. Dabei steht das Real-Time Modul über eine Ethernet Schnittstelle mit dem Entwicklungsrechner in Verbindung, welcher allerdings nicht das Echtzeitverhalten beeinflusst. Echtzeitrelevante Teile des *CANoe* Simulation und Test Projekts werden mittels dieser Verbindung direkt am *VT System* geladen und ausgeführt. Nicht echtzeitkritische Teile werden am Entwicklungsrechner ausgeführt. Zwischen der zu testenden ECU und dem Real-Time Module wird keine physikalische Verbindung hergestellt, da die Verbindung über die anderen Module des *VT System* erfolgt.

Abbildung 5.3: *VT System* der Firma *Vector*[6].

Die Applikationssoftware *CANoe* [5] wird für Restbussimulationen, also z.B. für Simulation von ECUs auf CAN Busen, verwendet. Unter der Verwendung einer *CANoe* C# Application Programming Interface (API) können Testfälle erstellt und ausgeführt werden. Mittels dieser API ist es möglich Signale und Umgebungsvariablen, welche Teil der *CANoe* Restbussimulation sind, zu manipulieren. In einem *CANoe* Projekt werden Testfälle in Testmodulen im Zuge von Testläufen ausgeführt. Dabei wird das *VT System* mittels der C# API initialisiert und die Instrumentierung und die messtechnische Auswertung der physikalischen Signale an den Schnittstellen des *VT System* getriggert. Abbildung 5.4 demonstriert diese Wechselwirkung. Dabei können Test Module in *CAPL*, einer *CANoe* spezifische Programmiersprache, beschrieben sein oder in Extensible Markup Language (XML) Modulen und .NET Modulen. Dabei bezeichnet das .NET Framework jene Software Plattform, welches für die Implementierung von C# Testmodulen verwendet wird. Als Teil der implementierten Testläufe werden Reports erstellt, dessen Resultate in XML oder Hypertext Markup Language (HTML) Reports veranschaulicht werden.

[6] `https://www.vector.com/de/de/produkte/produkte-a-z/hardware/vt-system/`, Online: Letzter Zugriff am 16.10.2019

Im *CANoe* ist es möglich, die dementsprechenden XML Files in benutzerfreundliche Forma-

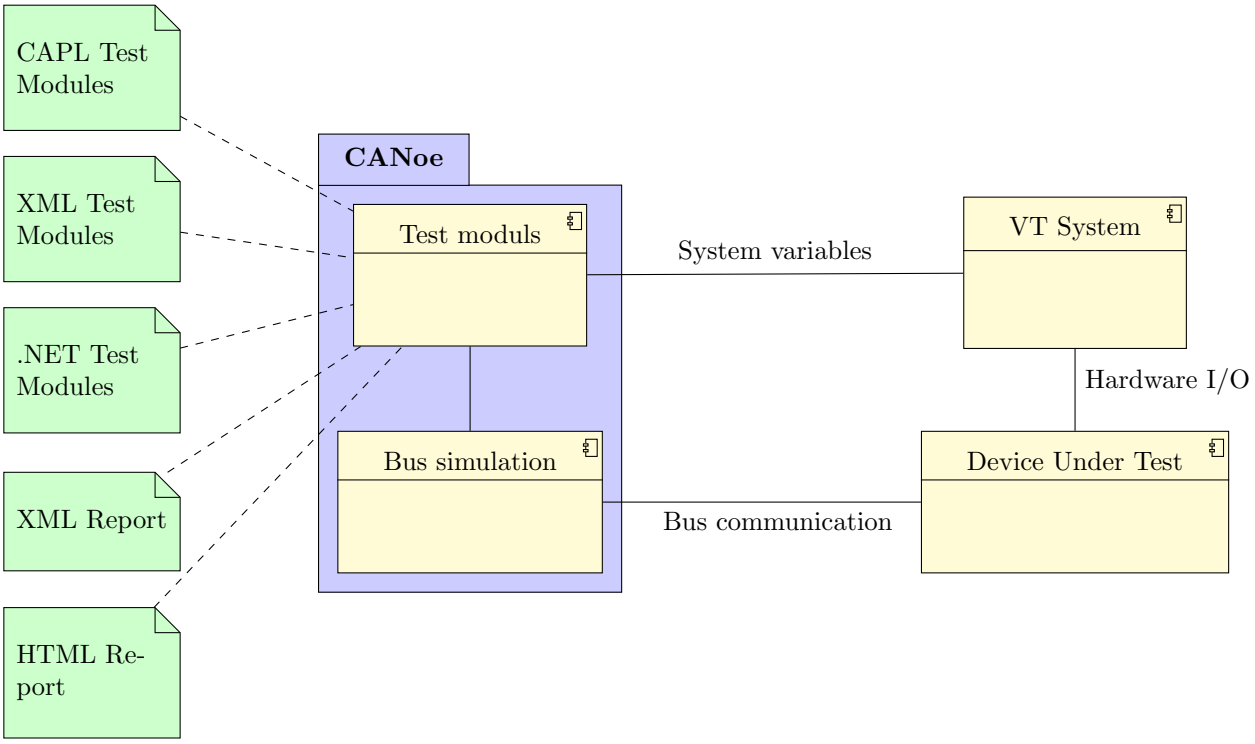

Abbildung 5.4: CANoe Testumgebung [5]

te zu transformieren. XML Formate für die Test Umgebung werden für Interaktion mit einem externen Test Management System verwendet [5]. In einem solchen Test Management System können, basierend auf testbare Requirements, Konfigurationsdaten und einer Test Case Generator Spezifikation, Test Module im XML Format generiert werden [56]. Test Module können dabei für unterschiedlichste Anwendungen Libraries beinhalten, welche auch die Interaktion mit unterschiedlichen Hardware Modulen managen. Die entwickelte Hardware/Software Integration Verifikation DLL stellt eine solche Library dar. Test Module werden in einem *CANoe* Projekt ausgeführt und die erstellten Test Reporte im XML Format werden in das Test Management System hochgeladen. Damit liegt die Kompetenz einer Test IngenieurIn in dem Beschreiben von Testfällen und der Interpretation der zugehörigen Requirements im Test Management System. Allerdings wird oftmals, im Hinblick auf Testabdeckung und Optimierung von Tests, Wert darauf gelegt, automatisierte Tests selbst zu implementieren. Dabei werden diese Tests allerdings zu den entsprechenden Testfällen und Requirements im Test Management System verlinkt. Abbildung 5.5 beschreibt diesen Workflow, welcher bei automatisierten Test Systemen, wie HIL gestützten Embedded System Tests, angewandt wird. Dieser Workflow wird auch für die Erstellung und Analyse von automatisierten Tests im System Integration Prozess angewendet.

5.2 Testing Framework Architektur

Das entwickelte Framework bedient mehrere unterschiedliche Disziplinen. So sollen sowohl Software Komponententests, als auch Software Integrationstests, sowie Hardware/Software Integrationstests und System Integrationstests automatisiert auf einer Zielumgebung durchführbar sein.

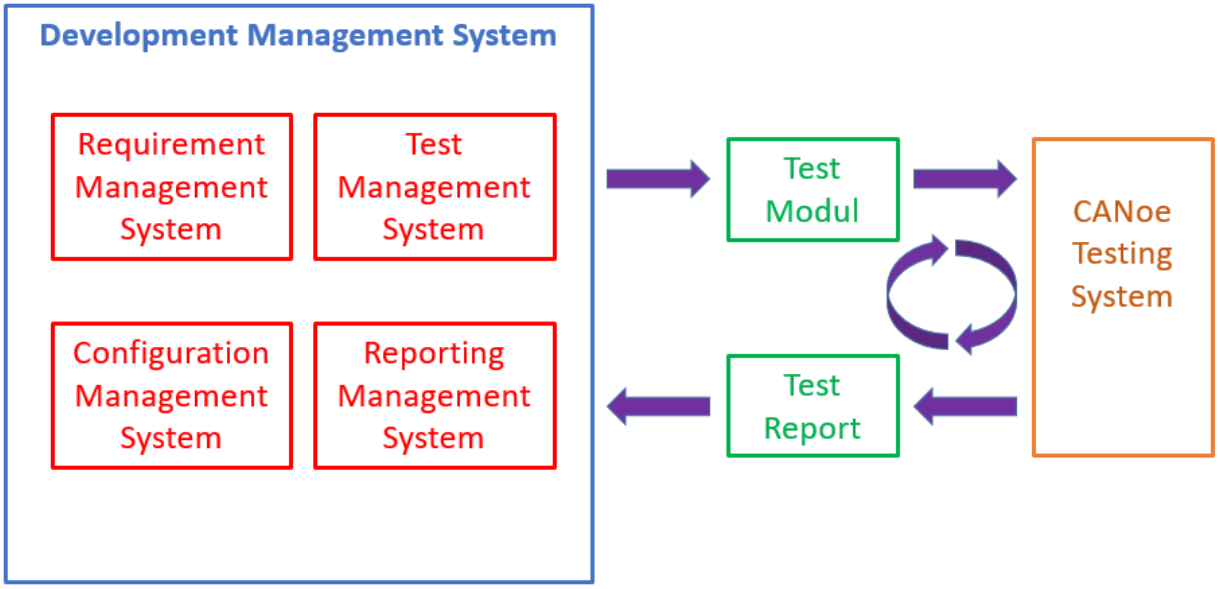

Abbildung 5.5: Workflow für automatisierte Tests [5]

Diese Bedienung von Tests auf unterschiedlichen Ebenen eines Systementwicklungsprozesses verlangt eine möglichst generische Architektur für das Testing Framework. Damit sollen Schnittstellen definiert werden, welche für gewisse Tests unter der Verwendung des Frameworks auch für Tests in anderen Verifikationsprozessen wiederverwendbar sind. Die Schnittstellen des Frameworks sollen idealerweise sogar für Codegeneratoren zur Verfügung stehen, um basierend auf einer Beschreibung für Testfälle einen ausführbaren Code generieren zu können. Dafür müssen sämtliche Schnittstellenmethoden verständlich und einfach verwendbar sein. Um ein generisches Framework auch möglichst breit einsetzen zu können, wird dieses oft konfigurierbar implementiert [18]. Bei der Implementierung eines solchen Frameworks wird seitens Funktionalität in den folgenden zwei Ebenen unterschieden:

- **Interaction Layer**: In diesem Layer werden dem User sämtliche Methoden als Schnittstelle möglichst intuitiv zur Verfügung gestellt [5]. Dabei wird zwischen einem Shallow Interface und einem Deep Module Interface unterschieden [3]. Beim Shallow Interface wird der TesterIn eine Vielzahl an möglichen Schnittstellenmethoden geboten, um eine möglichst große Flexibilität zu bieten. Dabei ist es allerdings notwendig, sich intensiv mit den Möglichkeiten dieser Schnittstellen zu beschäftigen, was eine größere Komplexität in der Verwendung verursacht. Bei einem Deep Modul Interface gibt es wenige, dafür aber auch wenig flexible Schnittstellen.

- **Abstract Machine Layer**: In diesem Layer erfolgt die tatsächliche Abarbeitung der Tasks des Frameworks. Dabei werden auch sämtliche Ressourcen verwaltet, um stets eine echtzeitfähige Performance zu gewährleisten.

Das entwickelte Framework entspricht einer entwickelten DLL, dessen einzelne Klassen in C# implementiert wurden. Abbildungen 5.6 demonstriert das zugehörige Klassendiagramm. Die jeweiligen Klassen sind im folgenden Abschnitt genauer erklärt werden.

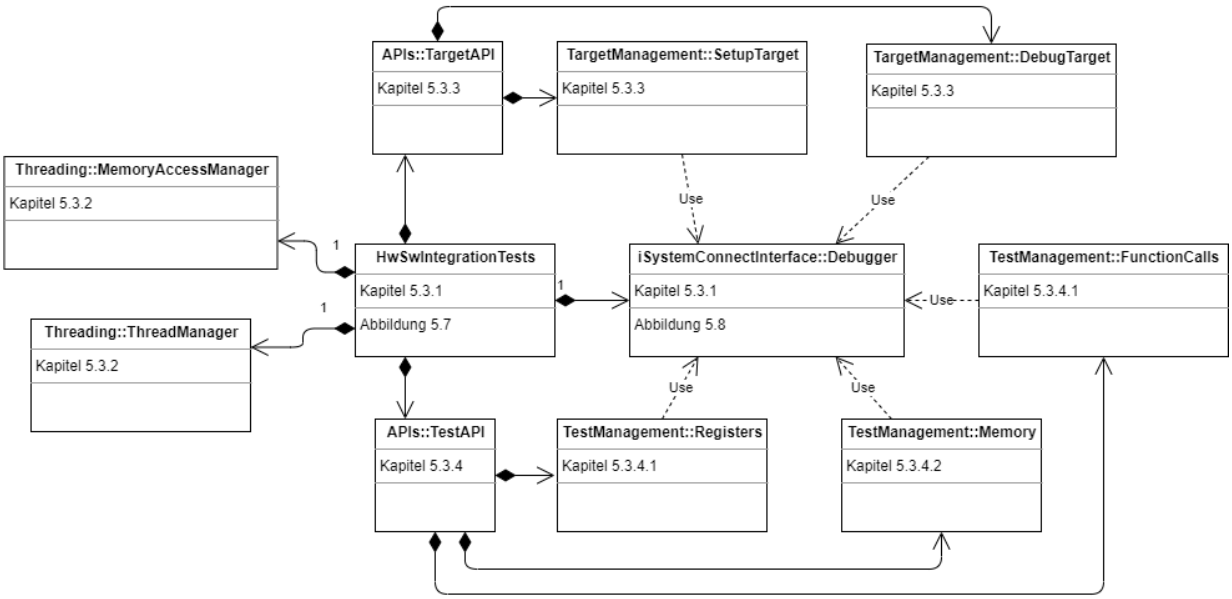

Abbildung 5.6: Klassendiagramm der Hardware/Software Integration Verifikation DLL

```
HwSwIntegrationTests tests = new
    HwSwIntegrationTests(HwSwIntegrationTests.PROCESSOR_TYPE_POWERPC,
    WinIDEAWsp, ElfFileOne);
tests.reinitializeController("main");
tests.callFunction("componentsInit");
```

Listing 5.1: Initialisierungsroutine zur Verwendung der *HwSwIntegrationTests* Library

5.3 Testing Framework Detailbeschreibung

In diesem Abschnitt werden sämtliche Klassen der *Hardware/Software Integration Verification* DLL im Detail beschrieben. Des Weiteren wird auf deren Anwendung in einem *CANoe* Testmodul eingegangen. Die Verwendung der Schnittstellen Methoden der DLL und des HIL Testsystems werden in Testfällen beschrieben und dokumentiert.

5.3.1 Schnittstellenklasse der DLL

Die Klasse *HwSwIntegrationTests* bietet sämtliche Methoden, welcher die TesterIn benötigt, um auf die Zielumgebung zuzugreifen. Der Zugriff auf weitere Klassen der DLL ist somit nicht vorgesehen, da diese in *HwSwIntegrationTests* instanziiert sind. Dabei wird diese Interface Klasse als Shallow Klasse kategorisiert, da eine große Vielzahl an Methoden zur Verwendung der DLL geboten wird. Abbildung 5.7 demonstriert diese Vielzahl an Methoden. Durch diese Vielzahl wird die Komplexität in der Verwendung der DLL gesteigert. So müssen Methoden in Listing 5.1 aufgerufen werden, um die Zielumgebung aufzusetzen und anschließend um die Komponenten testen zu können.
Beim Aufruf des Konstruktors der Interface Klasse *HwSwIntegrationTests* wird die Verbindung zum Debugger hergestellt und das *WinIDEA* Projekt mit den jeweiligen plattformspezifischen

```
try
{
    debugger = DebuggerInterface.Instance;
    target = new TargetAPI(debugger: debugger);
    test = new TestAPI(set: target.getSettingsOfTarget(), debugger:
        debugger);
    threadManager = ThreadManager.Instance;
    memAccess = MemoryAccessManager.Instance;
    threadManager.WaitForTask(task: target.setupTargetPlatform,
        parameterOne: workSpace, parameterTwo: elfFileOne,
        parameterThree: elfFileTwo, parameterFour: elfFileThree);
    threadManager.WaitForTask(task: target.deleteAllBreakpoints);
}
catch (Exception ex)
{
    Vector.Tools.Output.WriteLine(text: "Running HwSwIntegrationTests
        constructor failed!");
    Vector.Tools.Output.WriteLine(text: $"Exception Occurred
        :{ex.Message},{ex.StackTrace.ToString()}");
}
```

Listing 5.2: Erstellung sämtlicher interner Klassen der *Hardware/Software Integration Verification* Library

Konfigurationen geladen. Eines oder mehrere ELF Files können programmiert werden. Auch wird mittels eines Konfigurationparameters die Architektur des Mikrocontrollers übermittelt. Es werden alle Breakpoints, welche als Teil der Konfiguration gespeichert sein könnten, gelöscht, um zu verhindern, dass bei Ausführung der Software der Mikrocontroller stehen bleibt. Der mikrocontrollerspezifische StartUp Code der Software wird mit der Methode *reinitializeController()* bis zur Main Funktion der Software ausgeführt. Anschließend werden durch den Funktionsaufruf *callFunction("componentsInit")* sämtliche Low-Level Treiber Software Komponenten, welche Teil der Software sind, initialisiert und konfiguriert.
Nun ist es möglich an den unterschiedlichsten Stellen der Software zuzugreifen und einzugreifen, sowie diese über das Interface Embedded Application Binary Interface (EABI) [57] zu manipulieren. Es können Funktionen ausgeführt, globale Variablen beschrieben und gelesen und auf Speicherbereiche und Register zugegriffen werden. Auch das Debuggen der Software am Mikrocontroller ist durch Methoden der Interface Klasse *HwSwIntegrationTests* durchführbar. Ein Debug Prozess kann dadurch automatisiert werden.
Die Klasse *HwSwIntegrationTests* kümmert sich des Weiteren darum, dass sämtliche Zugriffe auf den Debugger in einem eigenen Thread und damit parallel zum Thread des *CANoe* Projekt ausgeführt werden. Dazu wird die Klasse *ThreadManager* instanziiert. Der Zugriff auf den Debugger und somit auch auf die Zielumgebung wird in der Klasse *Debugger* gemanagt. Dabei wird diese Klasse als Parameter beim Konstruktoraufruf an sämtliche weitere Klassen der DLL übergeben, um sicherzustellen, dass sämtliche Zugriffe auf den Debugger über die selbe Verbindung bewerkstelligt werden. Dies wird beim Konstruktoraufruf in der Klasse *HwSwIntegrationTests* durchgeführt, was in Listing 5.2 demonstriert wird.
Auch die Tasks zum Aufsetzen der Zielumgebung werden im Konstruktoraufruf durchgeführt. Sämtliche Zugriffe auf den Debugger werden in einer *try-catch* Routine behandelt. Falls es zu einem Fehler in der Ausführung kommt, wird eine Exception getriggert und der Fehler als Konsolenausgabe des *CANoe* Projekts angezeigt. Dabei werden die abgefangenen Exceptions individuell

zur Routine gestaltet [3].
Mit der Klasse *Debugger* wird die Verbindung zum *iSystem* Debugger hergestellt. Dabei handelt es sich um eine Singleton Klasse, welche bloß einmal instanziierbar ist. Damit soll verhindert werden, dass gleichzeitig in unterschiedlichen Routinen auf den Debugger und somit auch auf den Mikrocontroller zugegriffen werden kann. Mittels der Methoden dieser Klassen wird die Verbindung zum Debugger hergestellt. Des Weiteren dient die Klasse als Schnittstelle zu sämtlichen Klassen der *isystem.connect* DLL, welche in Abbildung 5.8 angeführt sind.

5.3.2 Threadmanagement

Mit der Klasse *ThreadManager* wird die asynchrone Abarbeitung von Threads [58] innerhalb der Library gemanagt. Der Zugriff auf den Debugger ist damit möglich, ohne dass andere Threads auf die Terminierung warten müssen. Dabei werden eigene Tasks mit eigenen Speicherbereichen und Variablen erzeugt. Auch sollen Tasks außerhalb der Library erzeugt und gemanagt werden können, was die Klasse *ThreadManager* zu einer Interface Klasse macht. Beispielsweise lässt sich der Build-Prozess der zu programmierenden Software in einen eigenen Thread innerhalb des *CANoe* Projektes ausführen. In Listing 5.3 wird eine Code-Sequenz zur generischen Ausführung eines Tasks mit einem Übergabeparameter und einem Rückgabewert präsentiert.
Sämtliche Tasks sowie deren Übergabeparamter werden in *private* Variablen oder Listen abgelegt. Anschließend wird der Thread *TaskExecuter* mittels der Methode *Execution.WaitForTask()* ertstellt und ausgeführt. Der aktuell laufende Thread wartet mit der weiteren Ausführung von Instruktion bis der neu geöffnete Thread terminiert wird. Der Thread kann auch nach einer übergebenen maximalen Zeitdauer vor vollständiger Ausführung terminiert werden. Innerhalb des Threads wird der Task mit seinen Übergabeparametern ausgeführt und der Rückgabewert in eine *private* Variable abgespeichert. In der *Hardware/Software Integration Verifiaction* DLL sind sämtliche Tasks Methoden der internen Klassen.
Die überladenen Methoden *WaitForTask()* sind dadurch eingeschränkt, dass sowohl Übergabeparameter als auch der Rückgabeparamter des Tasks vom Datentyp *String* sind. Das liegt daran, das bei der Deklaration von Tasks durch die Klasse *Func* keine generischen Datentypen für Paraemeter verwendet werden können [58]. Daher wurde *String* als generischer Datentyp in der *Hardware/Software Integration Verification* DLL ausgewählt. Sämtliche Variablen eines alternativen Datentyps müssen mit *ToString* gecastet werden. Ausnahmen wurden in der Klasse *MemoryAccessManager* geschaffen. Mittels der Methoden in Listing 5.4 können, in einem eigenen Task, *Byte Arrays* in den Speicher des Mikrocontrollers über den Debugger geschrieben oder von diesem gelesen werden.

5.3.3 Aufsetzen der Zielumgebung

Die Klasse *TargetAPI* wird als API verwendet, um Methoden für das Aufsetzen der Zielumgebung sowie Kontrollkommandos für das Debuggen des Mikrocontrollers zur Verfügung zu stellen. Dabei wird die Klasse *SetupTarget* dazu verwendet, um die Verbindung mit dem Debugger herzustellen, einen *WinIDEA* Workspace zu laden und die ELF Files der Software zu programmieren. Dies wird in Initialisierungsroutine der Listing 5.5 als Teil des Konstruktoraufrufes der Klasse *HwSwIntegrationTests* ausgeführt.
Um Peripherien sowie Treiber Komponenten der Mikrocontroller Software zu initialisieren, wird ein plattformspezifischer StartUp Code ausgeführt, welcher den Controller für die Ausführung einer Applikation vorbereitet. Sämtliche Ressourcen werden in Abstimmung mit der Konfiguration

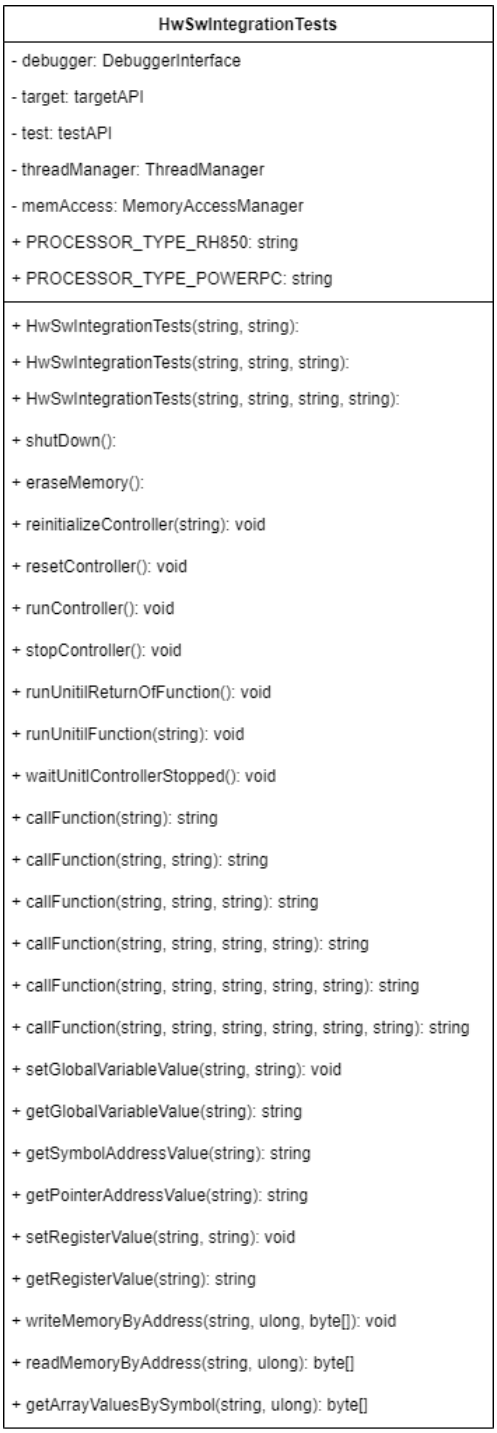

Abbildung 5.7: Methoden der Klasse *HwSwIntegrationTests*

```
private static Func<string, string> functionWithOneParam;
private static List<string> FunctionParameters;
private static string ReturnParameter = "Default";

public string WaitForTask(Func<string, string> task, string parameter,
    int maxMs = -1)
{
    functionWithOneParam = task;
    ReturnParameter = "Default";
    FunctionParameters.Clear();
    FunctionParameters.Add(item: parameter);
    int exe;
    if (maxMs != -1)
    {
        exe = Execution.WaitForTask(taskAction: TaskExecutor, maxTime:
            maxMs);
    }
    else
    {
        exe = Execution.WaitForTask(taskAction: TaskExecutor);
    }
    if (exe != EXECUTION_CORRECT)
    {
        Vector.Tools.Output.WriteLine(text: $"Execution Error with
            result: {exe.ToString()}");
    }
    return ReturnParameter;
}

private int TaskExecutor(TaskCancelToken tct)
{
    try
    {
        ReturnParameter = functionWithOneParam.Invoke(arg:
            FunctionParameters.ElementAt(0));
        return EXECUTION_CORRECT;
    }
    catch (Exception)
    {
        return EXECUTION_INCORRECT;
    }
}
```

Listing 5.3: Methode für die Ausführung eines Tasks in einen eigenen Thread

```
public string WaitForMemoryWriteTask(Func<string, ulong, byte[], string>
    task, string identifier, ulong size, byte[] data, int maxMs = -1);

public byte[] WaitForMemoryReadTask(Func<string, ulong, byte[]> task,
    string identifier, ulong size, int maxMs = -1);
```

Listing 5.4: Funktionen für das Erstellen von Threads für den Speicherzugriff auf die Zielumgebung

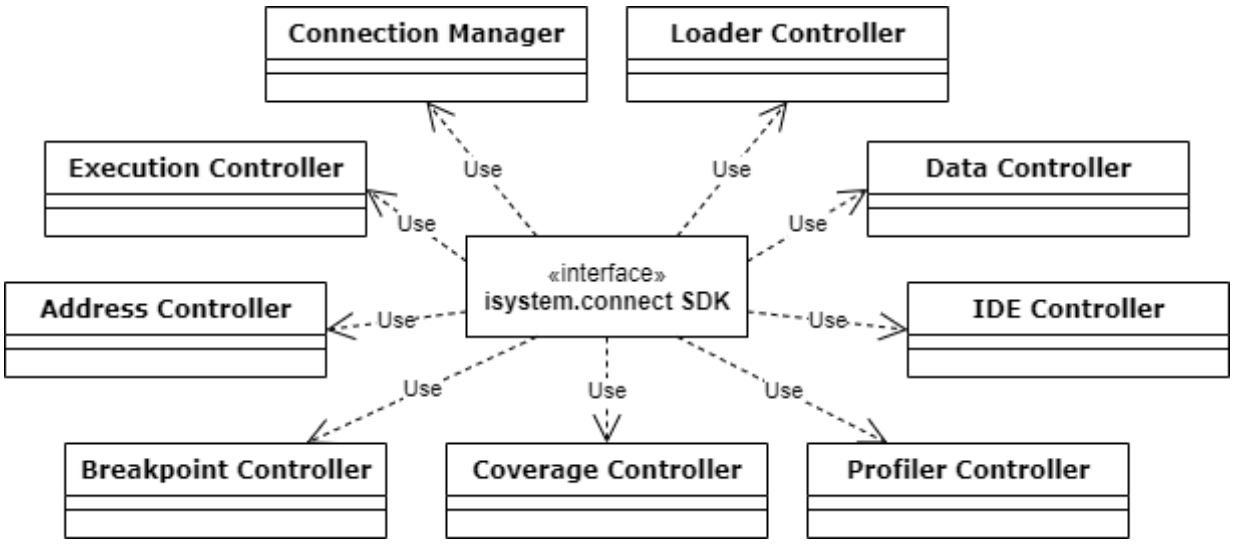

Abbildung 5.8: Klasse der DLL *isystem.connect SDK* [6]

```
setup = new SetupTarget(debug: debugger);
setup.connectDebugger();
setup.openWorkspace(path: workSpace);
setup.downloadSoftware(elfFileNameOne: elfFileOne, elfFileNameTwo:
    elfFileTwo, elfFileNameThree: elfFileThree);
```

Listing 5.5: Routine zum Aufsetzen der Zielumgebung

des Mikrocontrollers als Teil der Software initialisiert und die Register der jeweiligen IP-Cores geladen. In der Routine 5.6 wird dieser StartUp Code bis zum Funktionsaufruf der *Main* Funktion durchgeführt.
In der Klasse *DebugTarget* werden jene Methoden, die zum Debuggen der Software am Mikrocontroller verwendet werden können, abstrahiert. Kommandos wie *Reset*, *Run* und *Stop* werden der TesterIn bereitgestellt.

5.3.4 Zugriff auf Software Komponente auf der Zielumgebung

Die Klasse *TestAPI* wird als API verwendet, um Methoden für den Zugriff auf die programmierten Software Komponenten in einer Abfolge von Testschritten zur Verfügung zu stellen. Dafür muss die Zielumgebung bereits aufgesetzt sein. Die unterschiedlichen Zugriffsarten werden in den folgenden Abschnitten genauer beschrieben.

```
resetTarget();
runUntilFunction(funcname: mainFunctionName);
waitUntilStopped();
```

Listing 5.6: Routine zur Ausführung des Mikrocontroller StartUp Codes

5.3.4.1 Funktionsaufrufe

Um einen Zugriff auf die kompilierte Software auf der Zielumgebung zu erhalten, muss das Application Binary Interface (ABI) verwendet werden [59]. Die ABI definiert die Systemschnittstelle einer kompilierten Applikation. Das Binary der Software beinhaltet dabei sämtliche Parameter und Instruktionen, welche notwendig sind, um die Software auf der Zielumgebung auszuführen. Die dafür notwendigen Bibliotheken müssen beim Kompiliervorgang gelinkt werden. Mittels der generierten ELF Datei kann das Binary durch den Debugger interpretiert werden und den entsprechenden Symbolen der API zugewiesen werden. In der ELF Datei befinden sich auch sämtliche Maschineninformationen des Zielprozessors.
In der entwickelten DLL sind zwei Typen von Prozessoren verwendbar. Eine Konfiguration entspricht der ABI für Prozessoren mit einer *e500 PowerPC* Architektur. Dieser Prozessor verwendet eine Reduced Instruction Set Computer (RISC) Architektur optimiert für performante, kostengünstige Mikrocontroller [60]. Diese 64-Bit Superscalar Prozessorarchitektur ermöglicht, neben klassischen einfachen Single-Core Architekturen, die Umsetzung von Multiprozessor Features, sowie den Einsatz von Floating Point Register (FPRs) für Floating-Point Instruktionen in einer Floating Point Unit (FPU). Für die etwa 200 Maschinen Instruktionen wird ein 32-Bit langes Instruktionsformat verwendet.
Eine andere Konfiguration entspricht der ABI für Prozessoren der Firma *Renesas*[7] mit dem Prozessor *RH850* [61]. Dieser Prozessor entspricht ebenfalls einer RISC Architektur und umfasst weitgehend ähnliche Funktionalitäten wie der *e500 PowerPC*. Allerdings sind die Anwendungen der einzelnen Register der Prozessoren nicht konsistent, was eine dementsprechende Konfiguration der *Hardware/Software Integration Verification* DLL notwendig macht.
In der ABI sind Binary Informationen wie Byte Order, fundamentale Datentypen, Funktionsaufrufe und die Struktur des Stacks festgelegt. Diese Informationen werden verwendet, um seitens der entwickelten Methoden der DLL über die ABI auf die Software am Controller zuzugreifen. Die jeweiligen Register können so beschrieben und gelesen werden.
In Listing 5.7 wird ein Funktionsaufruf mit Rückgabewert und einem Übergabeparameter der DLL, konfiguriert für den *PowerPC* Prozessor, gezeigt. Sämtliche Funktionsaufrufe innerhalb dieser Methode beziehen sich auf Methoden des Objektes *Debug Manager (debugMgr)*, welche von der Klasse *Debugger* zur Verfügung gestellt werden. Zu Beginn wird die Adresse der Funktion aus dem Speicher geholt und in den Program Counter ("PC") geladen. Dabei entspricht "PC"dem Register "R0", welches nach der Definition der ABI [59] als Register für den aktuellen Adresswert des Program Counters verwendet wird. Die zugehörigen Teile der Software werden im Stack geladen und können nun ausgeführt werden. Werte von Übergabeparametern und Rückgabeparameter von Funktionen werden ebenso in Registern geladen, wie es in der ABI definiert ist. Nachdem der Stack und sämtliche Register geladen wurden, wird gewartet bis alle Instruktionen der Funktion ausgeführt werden. Letztendlich wird der Rückgabewert aus dem Register "R3"gelesen.
Die Funktionsaufrufe der Software auf der Zielumgebung werden in der Klasse *FunctionCalls* gemanagt. Doch auch ein direkter Zugriff auf die Register der Prozessoren wird mittels der Klasse *Registers* ermöglicht.

5.3.4.2 Speicherzugriff

Mittels der Klasse *Memory* werden Zugriffe auf den Speicher des Mirkocontrollers durchgeführt. Dabei können übergebene Symbole auf ihrem jeweiligen Platz im Speicher abgebildet werden.

[7] `https://www.renesas.com/eu/en/`, Online: Letzter Zugriff am 11.03.2020

```
public string call(string funcname, string param1)
{
    ulong FunctAddress;
    string returnValue;
    FunctAddress = debugMgr.getExpressionAddress(expression:
       funcname).getAddress();
    CValueType r1Value = new CValueType(bitSize: 32, value:
       FunctAddress);
    debugMgr.writeRegister(accessFlags:
       IConnectDebug.EAccessFlags.fRealTime, registerName: "PC",
       registerInfo: r1Value);
    if (string.IsNullOrWhiteSpace(param1) == false)
    {
        CValueType r3Value = new CValueType(bitSize: 32, value:
           Convert.ToUInt64(param1));
        debugMgr.writeRegister(accessFlags:
           IConnectDebug.EAccessFlags.fRealTime, registerName: "R3",
           registerInfo: r3Value);
    }
    debugMgr.stepInst();
    debugMgr.runUntilReturn();
    debugMgr.waitUntilStopped();
    debugMgr.stepInst();
    returnValue = debugMgr.readRegister(accessFlags:
       IConnectDebug.EAccessFlags.fRealTime, registerName:
       "R3").getResult();
    return returnValue;
}
```

Listing 5.7: Funktionsaufruf

```
remove_all_files_from_download_list();
add_file_to_download_list(elfFileName);
download();
```

Listing 5.8: Download Funktion

Dies ist die Methode mit welcher auf globale Variablen zugegriffen wird. Allerdings kann die zugehörige Speicheradresse ermittelt werden und mit dieser die jeweilige Stelle im Speicherbereich manipuliert werden. Das Schreiben und Lesen von mehreren Bytes auf den Speicherbereich über einer übertragen Basisadresse wird zur Verfügung gestellt. Auch auf Register, welche über die ABI [59] definiert sind, kann zur Laufzeit zugegriffen werden.
Für sämtliche beschriebene Funktionalitäten ist das Programmieren einer Software notwendig. Dies wird in der Routine 5.8 mittels Methoden der Klasse *SetupTarget* durchgeführt. Als Instanz der Klasse *Memory* werden sämtliche Referenzen zu ELF Files, welche in der *WinIDEA* Konfiguration gespeichert sind, entfernt und das übergebene hinzugefügt, um auch wirklich bloß die Software zu programmieren, welche für die Durchführung der Tests benötigt wird. Danach wird der Mikrocontroller programmiert.
Listing 5.9 beschreibt Routinen mit welcher *Byte Arrays* in den Flash Speicher des Mikrocontrollers geschrieben oder von diesem gelesen werden können. Um einen Speicherzugriff zur Laufzeit zu gewähren, muss der Mikrocontroller und der Debugger einen Zugriff über eine JTAG Schnittstelle

```
private void write_memory_by_address(string address, ulong length,
    byte[] data)
{
    VectorBYTE outBuffer = new VectorBYTE();
    ulong i;
    for (i = 0; i < length; i++)
    {
        outBuffer.Add(x: data[i]);
    }
    debugger.dbg().writeMemory(accessFlags:
        IConnectDebug.EAccessFlags.fMonitor, memArea: 0, aAddress:
        Convert.ToUInt32(address), aNumMAUs: length, bytesPerMAU: 1,
        buff: outBuffer);
}

private byte[] read_memory_by_address(string address, ulong length)
{
    byte[] data = new byte[length];
    VectorBYTE dataBytes = debugger.data().readMemory(accessFlags:
        IConnectDebug.EAccessFlags.fRealTime, memArea: 0, aAddress:
        Convert.ToUInt32(address), aNumMAUs: length, bytesPerMAU: 1);
    dataBytes.CopyTo(array: data);
    return data;
}
```

Listing 5.9: Lese- und Schreibroutine auf den Flash Speicher des Mikrocontrollers

zur Verfügung stellen. Dabei ist es notwendig, dass auf dem JTAG IP-Core des Mikrocontrollers ein TAP Controller [44] vorhanden ist und dieser die Funktion des Zugriffs auf die Speicherelemente des Mikrocontrollers zu Laufzeit zu Verfügung stellt. Mittels des TAP Controllers sind einige On-Board Debug Features am Mikrocontroller durchführbar, unter anderem auch die Manipulation von Werten hinter einer bestimmten Speicheradresse während der Laufzeit des Mikrocontrollers. Sowohl der JTAG des Debuggers als auch der TAP Controller des Mikrocontrollers müssen dabei einer Kategorie im Standard IEEE-ISTO 5001™-2003 (NEXUS) [34] entsprechen, welche einen echtzeitfähigen Zugriff auf den Speicher garantiert. Sowohl der für diese Arbeit verwendete Debugger [53] als auch der Mikrocontroller [49] unterstützen den notwendigen Standard für einen Echtzeitzugriff auf Symbole der Software auf der Zielumgebung. Dabei entspricht der TAP Controller des Mikrocontrollers der Spezifikation des Standards IEEE 1149.1-2001 [62], welcher eine Testlogik für ICs definiert. Diese Logik basiert auf sogenannten Boundary Scan Registern.

5.3.5 Umsetzung von Testfällen in einem Testmodul

Für die Implementierung von Testfällen in einem *CANoe* Projekt wird ein Testmodul angelegt, in welchem die Tests kompiliert und ausgeführt werden [5]. In einem Testmodul werden einzelne Testfälle durchgeführt, welche einzelne Variablen eines sogenannten Interaction Layer (IL) setzen und lesen und Schnittstellenmethoden ausführen. Dieser IL stellt die Verbindung zum System Under Test (SUT) her, wie in Abbildung 5.9 dargestellt. Die Testroutinen sind Teil des Appliaction Layer der Test Architektur. Im Allgemeinen hat der IL die Aufgabe die Schnittstelle zwischen den Tests und des SUT herzustellen und dieses zu managen. Teil dieses Management ist das Abbilden

von logischen auf physikalische Signale, Kalkulation im Hintergrund, sowie ein Error Handling.
Für die Umsetzung von Hardware/Software Integration Tests in dieser Arbeit sind Teil des IL

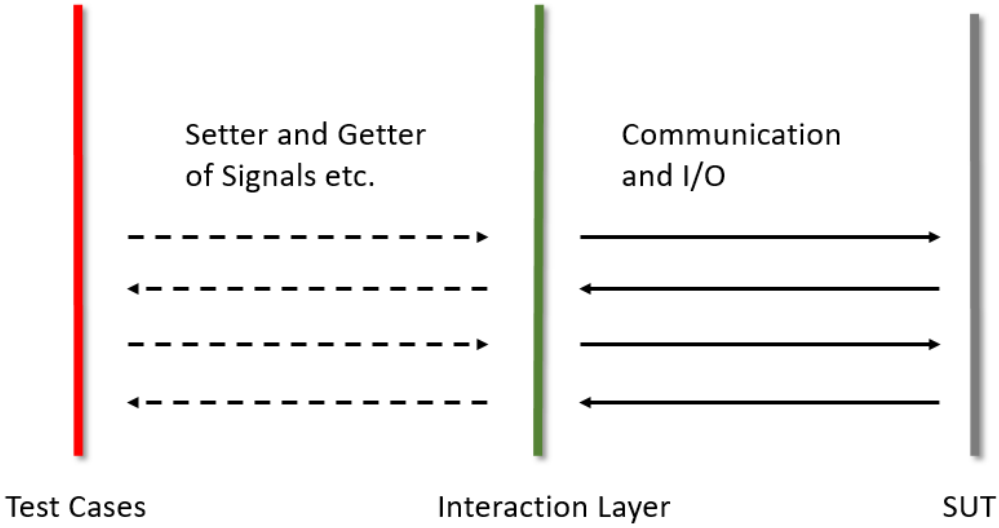

Abbildung 5.9: Demonstration des Interaction Layer in einem Testmodul [5]

eine Menge von DLLs. Abbildung 5.10 zeigt die folgende Sammlung an DLLs für ein .NET Test Modul:

- **HwSwIntegrationVerification Library**: Entwickelte DLL für Hardware/Software Integration Verifikation Tests.
- **iConnectCSLib**: Diese DLL wird von der *isystem.connect* SDK für die Interaktion mit dem Mirkorcontroller über den *iSystem* Debugger zur Verfügung gestellt.
- **Verification Library**: Diese DLL wird für die Erstellung von Testgruppen, Testfällen, Testschritten sowie Testreports verwendet.
- **CANoe Libraries**: *Vector* DLLs, welche die Interaktion mit dem HIL Test System abstrahieren.
- **PostSharp**[8]: Diese Compiler-Extension wird verwendet, um in der Verification Library Muster zu erstellen.
- **MathNet.Numerics**[9]: Diese DLL stellt Methoden und Algorithmen für numerische Berechnungen zur Verfügung.

Teil eines Testmoduls ist eine *MainTest()* Methode, welche als Startroutine für sämtliche Tests ausgeführt wird. Die programmierten Testfälle in einzelne Schritte unterteilt, welche die tatsächlichen

[8] `https://www.componentsource.com/de/product/postsharp-framework`, Online: Letzter Zugriff am 28.02.2020
[9] `https://numerics.mathdotnet.com/`, Online: Letzter Zugriff am 28.02.2020

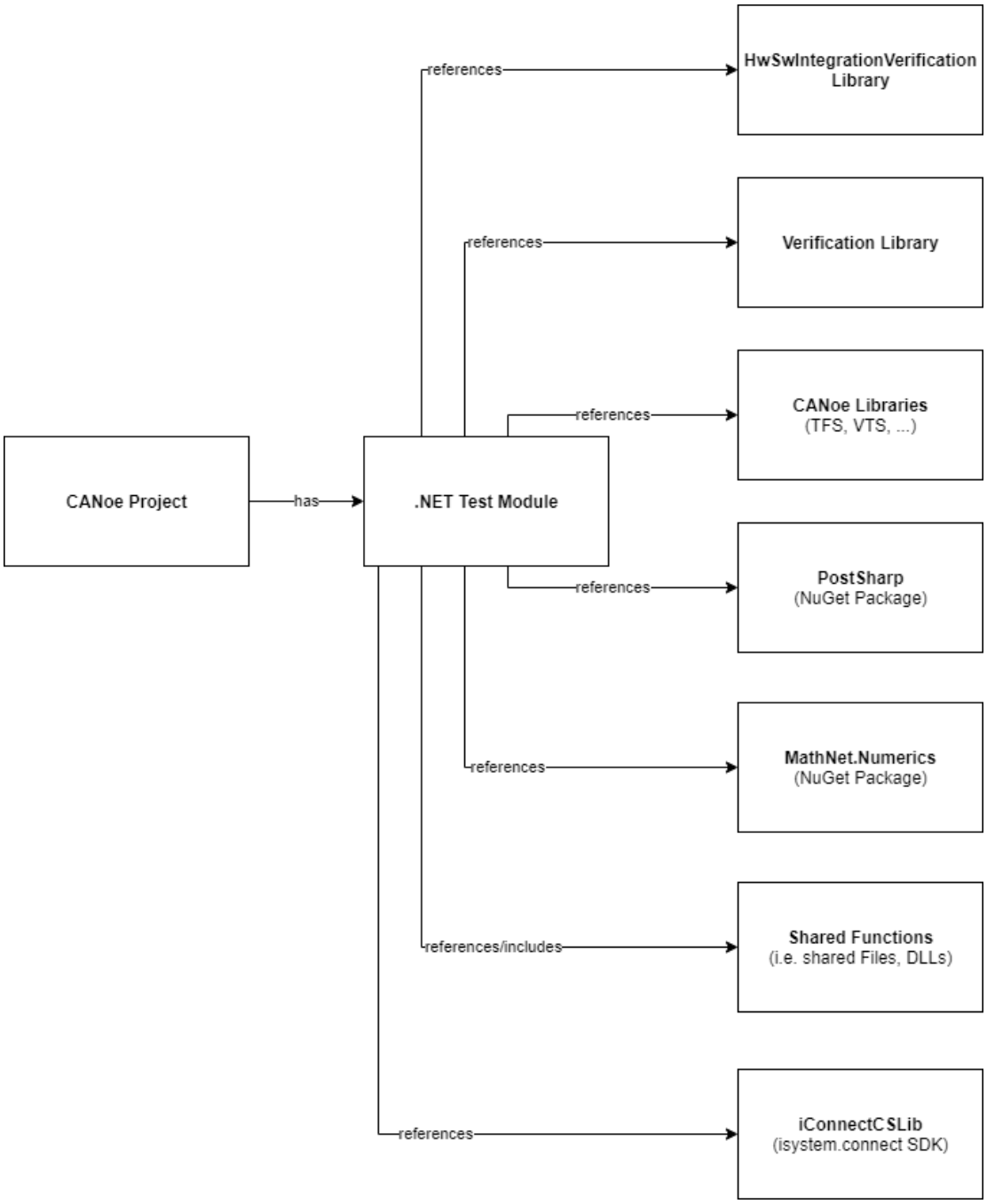

Abbildung 5.10: DLLs des Interaction Layers

Anweisungen durchführen, auswerten und ablegen. Testfälle, welche im gleichartigen Kontext eine Komponente testen, werden in Testgruppen zusammengefasst. Dabei gilt es allerdings zu beachten, dass alle Testfälle dennoch unabhängig voneinander implementiert werden müssen. Bei Ausführung der Tests wird ein Testreport erstellt, welcher die Ergebnisse der Tests dokumentiert. Dieser Report ist dabei wiederum in Testgruppen strukturiert.
In Listing 5.10 wird ein Testfall mit jeden einzelnen Schritten anhand eines Tests für eine DIO Software Komponente demonstriert. Die verwendeten *TestCases* der Testfälle dieser Arbeit sind zwar gänzlich voneinander unabhängig, können aber dennoch generische Attribute beinhalten. So können diese *Preconditions* und *Postconditions* beinhalten, welche dieselben Methoden ausführen können. Damit kann z.B. die Zielumgebung der Tests initialisiert werden. Auch die Überprüfung von Rahmenbedingungen und Einschränkungen an die Umgebung der Tests werden hier überprüft. Wenn diese den Wertebereichen interner Testparameter entsprechen, können Tests ausgeführt werden. Mittels der Klasse Report wird ein TestRport erstellt, welcher in die Testschritte *Arrange*, *Act* und *Assert* unterteilt ist. Während der Testschritte *Arrange* und *Act* werden diverse Methoden zur Interaktion mit der Software auf der Zielplattform mittels des Debuggers und mit der Hardware der Zielplattform über das HIL Testsystem durchgeführt. Im *Assert* werden erfasste Werte verglichen und dementsprechend dokumentiert. Im Allgemeinen hat ein Test bestanden, wenn er gänzlich ausgeführt wurde und alle gemessenen Werte den erwarteten Werten entsprechen. Andernfalls schlägt ein Test fehl oder eine Warnung bei fehlerhafter Ausführung wird dokumentiert.
Bei dem Test, welcher in Listing 5.10 beschrieben ist, wird funktional verifiziert, ob die Funktion *Dio_ReadChannel* der Software Low-Level Treiber Komponente DIO, tatsächlich den physikalischen Wert, welcher am Mikrocontroller Pin *DIO0* anliegt, ausliest. Dazu wird ein dementsprechender Spannungspegel mittels einer Systemvariable des HIL Testsystems am Pin angelegt. Sämtliche Interrupts, welche in der Software am Mikrocontroller definiert wurden, werden für eine störungsfreie Ausführung des Funktion *Dio_ReadChannel* deaktiviert. Der gelesen logische Pegel am Pin wird im Assert Schritt letztendlich mit dem erwarteten verglichen.
Um die einzelnen Signale eines Tests zu den geplanten Zeitpunkten zu triggern oder auszuwerten, sind Mechanismen der Flow Control notwendig. So kann die sequentielle Abarbeitung der Test Instruktionen durch Methoden wie *Execution.Wait(200)* unterbrochen werden. Dabei wird die Durchführung der Testschritte erst nach 200ms wieder fortgeführt. Es ist auch möglich erst nach Auftreten von bestimmten Events die Tests wieder fortzuführen. Dabei ist es allerdings auch möglich die Wartezeit auf ein Event bei Ablauf einer gewissen Zeit zu beenden, um die Testausführung zeitnah zu terminieren und die Abwesenheit des Events dementsprechend zu dokumentieren.
Die Interaktion mit den Modulen des *VT-Systems* werden über die *CANoe* spezifischen DLLs gemanagt. Dafür ist es allerdings notwendig modulspezifische Objekte zu instanzieren und deren Parameter zu konfigurieren. In Listing wird dies am *VT2848* Modul für die Behandlung von DIO Signalen demonstriert. Nachdem das Objekt *VT2848* erstellt und konfiguriert wurde, werden zwei Kanäle am Modul instanziiert. Dabei wird einer als digitaler Input und der andere als digitaler Output konfiguriert.

```
[TestCase(precondition: Precondition.ENABLED, versionCheck:
    VersionCheck.DISABLED, historyCheck: HistoryCheck.DISABLED,
    postcondition: Postcondition.ENABLED)]
public void CheckIfDio_ReadChannelReturnsHighValueAtDio0()
{
#region variables
string dioInput;
#endregion

#region Arrange
Report.BeginArrange();
{
    tests.callFunction("DisableAllInterrupts");
    hil.vt2848_Ch1.DigitalOutput.Value = false;
    Execution.Wait(200);
}
Report.EndArrange();
#endregion

#region Act
Report.BeginAct();
{
    hil.vt2848_Ch1.DigitalOutput.Value = true;
    Execution.Wait(200);
    dioInput = tests.callFunction("Dio_ReadChannel", DIO0);
}
Report.EndAct();
#endregion

#region Assert
Report.BeginAssert();
{
    if (String.Compare(dioInput, "1") == 0)
    {
        Report.TestStepPass("OUT", "Read Level " + dioInput +
            " at GPIO Pin " + DIO0);
    }
    else
    {
        Report.TestStepFail("OUT", "Read Level " + dioInput +
            " at GPIO Pin " + DIO0);
    }
    swComponent.integrationTests.callFunction("EnableAllInterrupts");
}
Report.EndAssert();
#endregion
}
```

Listing 5.10: Testfall

```
IVT2848 VT2848 = VTSystem.Instance.GetModule("M3_VT2848") as IVT2848;
VT2848.OutputSource1To4.Value = OutputSource.VBat;
VT2848.Threshold1To8.Value = 2.5;
IVT2848PWMMeasurementChannel vt2848_Ch1 =
    VTSystem.Instance.GetChannel("M3_Ch1") as
    IVT2848PWMMeasurementChannel;
IVT2848PWMMeasurementChannel vt2848_Ch3 =
    VTSystem.Instance.GetChannel("M3_Ch3") as
    IVT2848PWMMeasurementChannel;
vt2848_Ch1.OutputMode.Value = VT2848OutputMode.PushPull;
vt2848_Ch3.OutputMode.Value = VT2848OutputMode.Inactive;
vt2848_Ch1.CurveType.Value = CurveType.Constant;
vt2848_Ch3.CurveType.Value = CurveType.Constant;
```

Listing 5.11: HIL Initialisierung

5.4 Anwendungsbeispiel zum Testing Framework

In diesem Abschnitt soll die Verwendung des entwickelten Hardware/Software Integration Verifikation Frameworks beispielhaft demonstriert werden. Dabei werden die Implementierungen der EntwicklerIn in projektspezifischen Methoden oder in den Test Klassen behandelt, wie in Abbildung 5.11 demonstriert. Methoden der DLLs des IL werden dabei verwendet.

Beim Start der Tests wird im Testmodul die Methode *StructuredMain()* gestartet, wie im Ab-

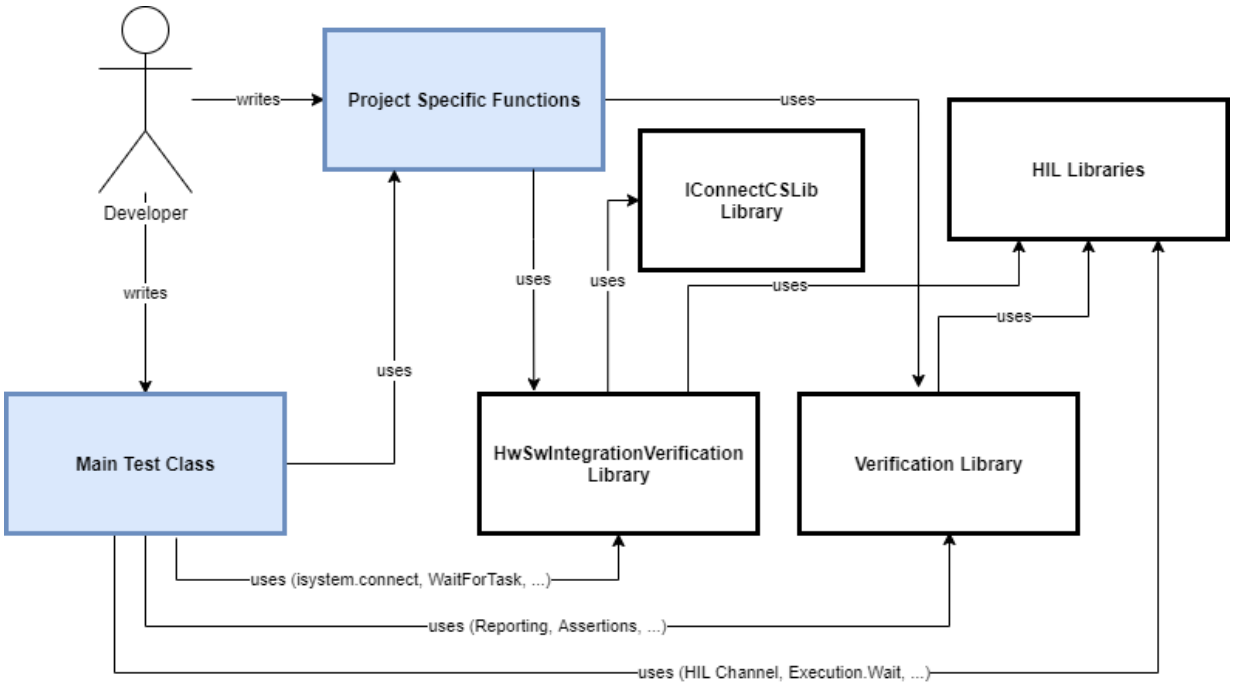

Abbildung 5.11: Verwendung des Hardware/Software Integration Verifikation Framework

laufdiagramm 5.12 demonstriert. Anschließend wird das HIL Testsystem initialisiert. Beim Aufruf des Konstruktors der *Hardware/Software Integration Verifiaction* DLL wird die Verbindung zum Debugger hergestellt, ein *WinIDEA* Workspace geladen, die Software programmiert, der Prozessor konfiguriert, der Mikrocontroller zurückgesetzt und alle Breakpoints gelöscht. Danach können

die Testfälle abgearbeitet werden. Vor der Terminierung des Testmoduls wird der *WinIDEA* Workspace geschlossen und die Verbindung zum Debugger getrennt. Letztendlich kann der erstellte Testreport ausgewertet werden.

Beispielhaft wird in Abbildung 5.13 der Ablauf eines Tests zur funktionalen Verifikation des Soft-

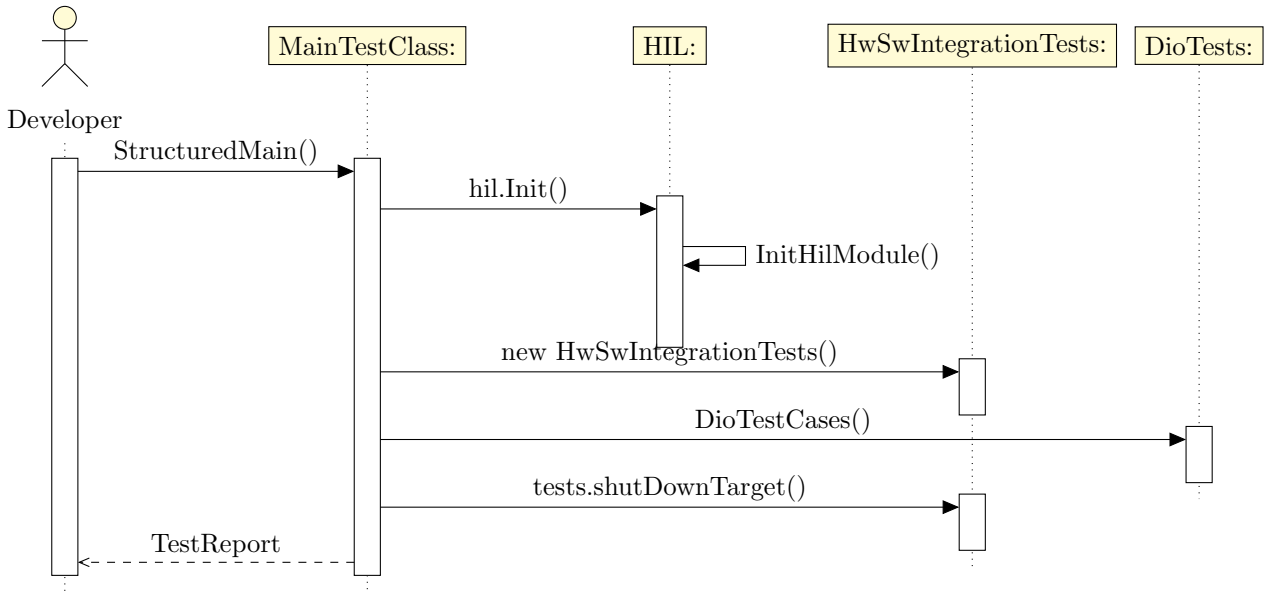

Abbildung 5.12: Ablaufdiagramm für die Implementierung eines Hardware/Software Integration Tests

ware Treiber Moduls DIO demonstriert. Es soll erörtert werden, ob die Funktion *DioReadChannel* an einem bestimmten Pin die angelegten Werte korrekt ausliest. Zu Beginn werden die Methoden zu Precondition ausgeführt, wobei die Precondition generisch für sämtliche Tests mit der gewählten Zielumgebung angewandt wird. Sowohl Precondtions als auch Postconditions werden unabhängig von den Tests implementiert. Damit soll jeder einzelne Test für sich selbst ebenfalls unabhängig sein. Sämtliche Testschritte werden entkoppelt voneinander im selben Testfall unterteilt und dokumentiert. Im Testschritt *Arrange* wird am Pin des Mikrocontrollers 0V angelegt mittels des HIL Testsystem. Im Testschritt *Act* werden dann 5V angelegt, und mittels triggern der Funktion *DioReadChannel* der Software Komponente DIO ausgewertet. Im Testschritt *Assert* wird dann überprüft, ob der Rückgabewert der Funktion *DioReadChannel* dem logischen Wert 1 entspricht. Falls dem so ist, wird der Test als bestanden dokumentiert. Auch das Fehlschlagen eines Tests wird dementsprechend dokumentiert.

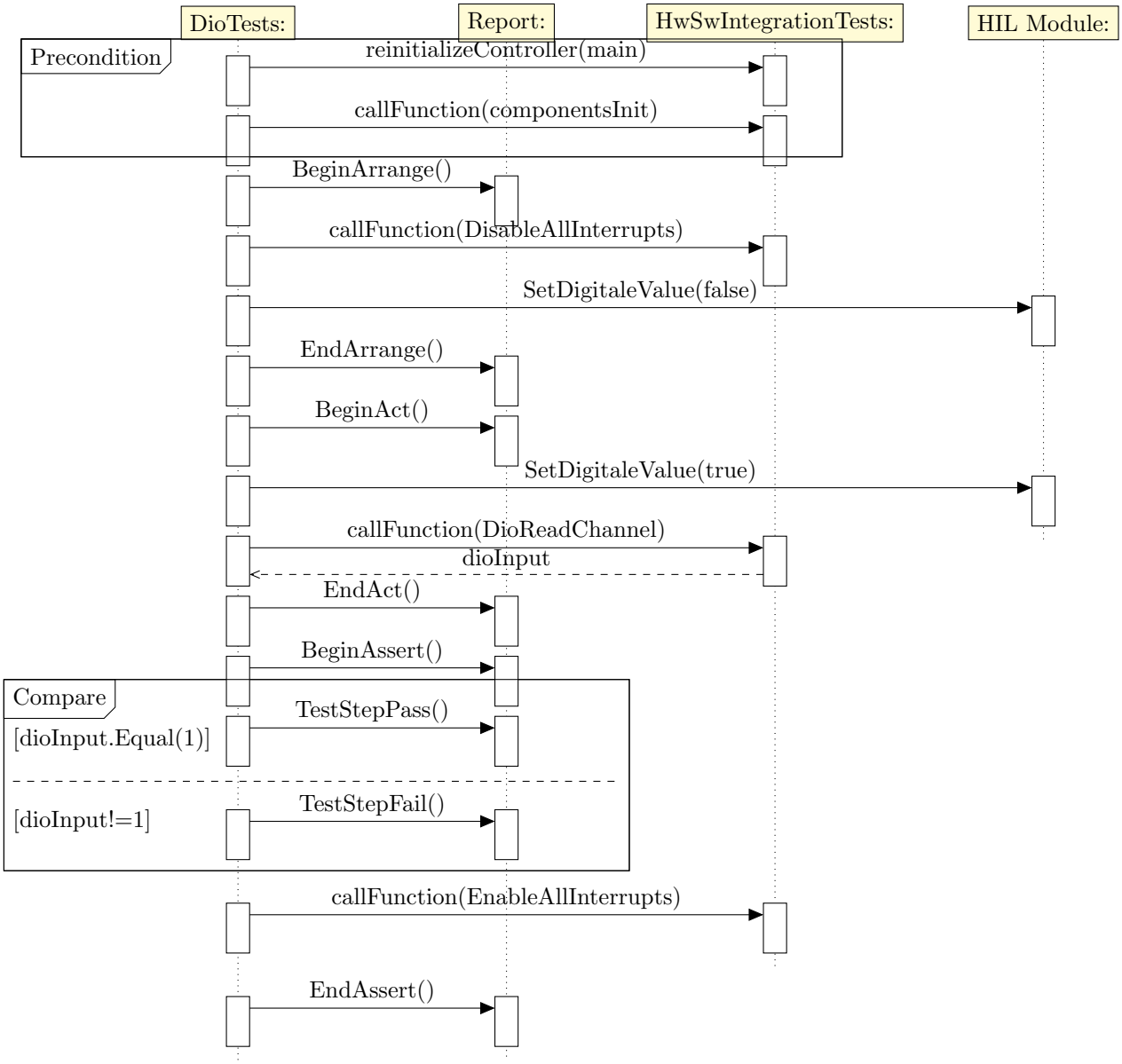

Abbildung 5.13: Ablaufdiagramm für Test CheckIfDio_ReadChannelReturnsHighValueAtDio0()

6 Experimenteller Aufbau

Die entwickelte Plattform für Hardware/Software Integration Tests soll evaluiert werden. In diesem Kapitel wird der experimentelle Aufbau zur Evaluierung der Plattform beschrieben. Dabei wurden LLD Software Module auf der Mikrocontroller Zielumgebung durch funktionale Hardware/Software Integration Tests verifiziert. Das bedeutet, Treiber Module des HAL der Software Architektur werden auf der Zielumgebung durch Unit Tests verifiziert. Im Allgemeinen versteht man unter dem HAL jenen Teil der Software, welcher als Schnittstelle zwischen Hardware und Software verwendet wird, und direkt mit den Registern des IP Cores des Mikrocontrollers interagiert [63]. Die API der Software kann über den HAL auf die umgebende Hardware zugreifen.
In Abbildung 6.1 werden sämtliche LLD Module als Teil einer sehr einfachen zugehörigen Software Architektur abgebildet, welche in dieser Arbeit evaluiert wurden. Sämtliche Module stellen dabei den Anspruch, dem AUTomotive Open System ARchitecture (AUTOSAR) Standard [64] zu entsprechen. AUTOSAR ist ein Software Architektur Standard der Automobile Industrie. Teil dieses Standards ist die genaue Spezifikation der Modulschnittstellen, sowie die interne Komponenten Architektur. Abbildung 6.2 zeigt die Abstraktion der Software Architektur nach diesem Standard [65]. Dabei ist der HAL Teil des Basic Software (BSW). Diese BSW lässt sich wiederum in drei Ebenen unterteilen [66]:

- **Microcontroller Abstraction Layer**: Diese unterste Schicht der BSW beinhaltet jene plattformspezifischen Treiber, welche notwendig sind, um auf die interne Peripherie des Mikrocontrollers zuzugreifen. Dabei soll die darüber liegende Software plattformunabhängig implementiert werden können.
- **ECU Abstraction Layer**: Über die Interaktion mit dem *Microcontroller Abstraction Layer* werden hier Treiber für die umgebende Hardware auf der ECU umgesetzt. Dabei soll die darüber liegende Software von allen elektronischen Bauteilen der ECU unabhängig werden.
- **Service Layer**: Diese oberste Schicht des BSW abstrahiert den Hardwarezugriff mittels Services, welche von der Applikation aufgerufen werden kann. Dazu wird eine Anbindung an die Runtime Environment (RTE) zur Verfügung gestellt, welche der Applikation Kommunikationservices liefert, um diese unabhängig von der BSW zu bewerkstelligen. Diese Ebene soll bereits Mikrocontroller und ECU unabhängig sein.

Diese Ebenen werden in der Abbildung 6.2 an einen Kommunikation Stack beispielhaft angeführt. Dabei verwendet jede Komponente als Teil eines Layers des Stacks die Services der darunter liegenden Komponente [3].

In [67] werden die allgemeinen Requirements für Module in der BSW Ebene formuliert. Für die Formulierung der Testfälle wurden die AUTOSAR Requirements referenziert.

Beim Start-Up der Software werden vorab sämtliche Peripherien des Controllers initialisiert,

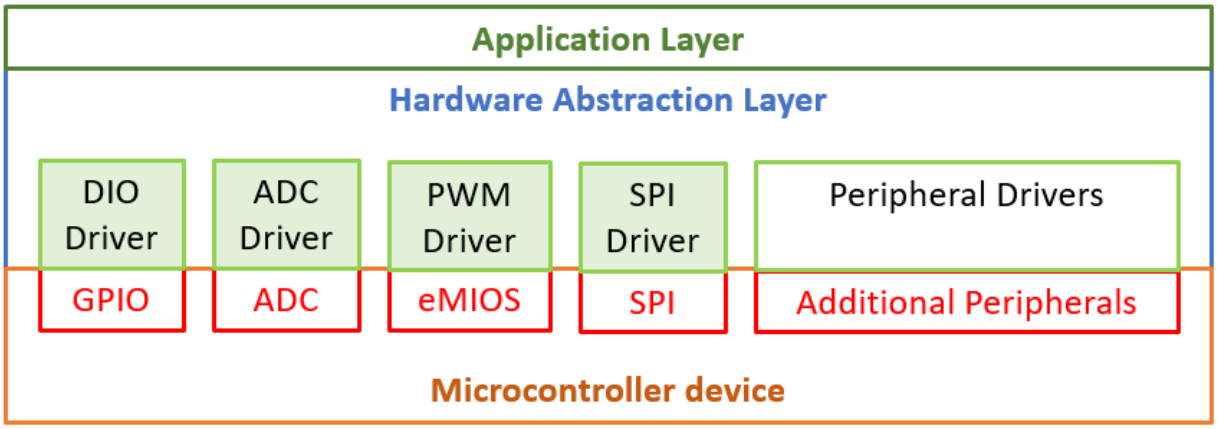

Abbildung 6.1: Software Architektur der evaluierten Low-Level Treiber Module

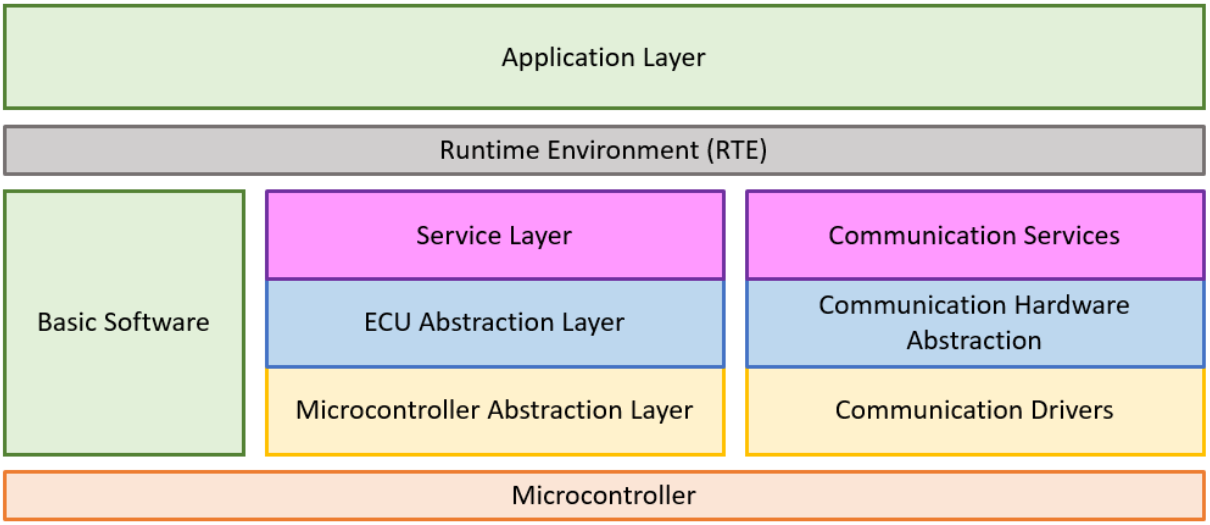

Abbildung 6.2: Allgemeine AUTOSAR Standard Architektur mit Fokus auf den Basis Software Layer [3]

bevor der HAL Layer initialisiert wird. Teil dieser HAL Initialisierung ist die komponentenspezifische Clock und Interrupt Konfiguration.

Im Folgenden wird die Evaluierung sämtlicher Low-Level Treiber Module beschrieben. Dabei wurden auch Requirements an die Tests erstellt. Bezogen auf die formulierten Requirements wurden Testfälle formuliert, deren Parameter sämtliche Grenzwerte abdecken, um eine möglichst große Testabdeckung zu erzeugen. Dabei wurde darauf geachtet, dass unterschiedliche Kanäle sowie Hardware Blöcke der verwendeten IP-Cores beansprucht wurden, um die Plattformunabhängigkeit zu verifizieren. Auch werden die messtechnischen Schaltungen am HIL Testsystem erläutert. Die finalen Test Ergebnisse werden präsentiert. Die gewonnenen Erkenntnisse zu der implementierten Hardware/Software Integration Plattform werden im Anschluss diskutiert (siehe Kapitel 8). Letztlich wurden sämtliche Module basierend auf den AUTOSAR Requirements,

interner funktionaler und nicht funktionaler Requirements zur Entwicklung generischer Software Komponenten, sowie auf der IP Core Beschreibung des Mikrocontroller Datenblattes [49] entwickelt und Unit getestet. Die Umsetzung und Resultate dieser Hardware/Software Integration Tests auf der Zielumgebung wird in diesem Kapitel beschrieben. In der Umsetzung wurde darauf geachtet, dass die Konfiguration und Initialisierung der Software Module durch die Testumgebung durchgeführt wird.

6.1 Digital Input/Output (DIO) Modul

Im DIO Modul werden einzelne Pins an zugehörigen Ports des Mikrocontrollers auf einen physikalischen Low (0V) oder High (5V) Pegel gesetzt durch das setzen bzw. lesen des zugehörigen Datenregisters [68]. Als Teil der Mikrocontroller Initialisierung sind die Pins hierzu entsprechend zu konfigurieren. Neben der Konfiguration als Ein- bzw. Ausgang besteht die Möglchkeit zur aktivierung zugehöriger Pull-Up/Pull-Down Widerstände.
Abbildung 6.3 präsentiert jene AUTOSAR DIO Treiber Architektur, welche die Funktionen zum Lesen und Schreiben an Ports des Mikrocontrollers spezifiziert [7]. Dabei wird die Port Initialisierung und Konfigurierung unabhängig von den Lese- und Schreibfunktionen strukturiert. Die implementierten Module abstrahieren die I/O Hardware zu den höheren Softwareebenen und interagieren mit den On-Chip Registern des IC. Diese Register stehen im direkten physikalischen Kontakt zu den Ports und Pins des Mikrocontrollers.
In [69] sind AUTOSAR Requirements für diese Komponente formuliert. Für die Durchführung

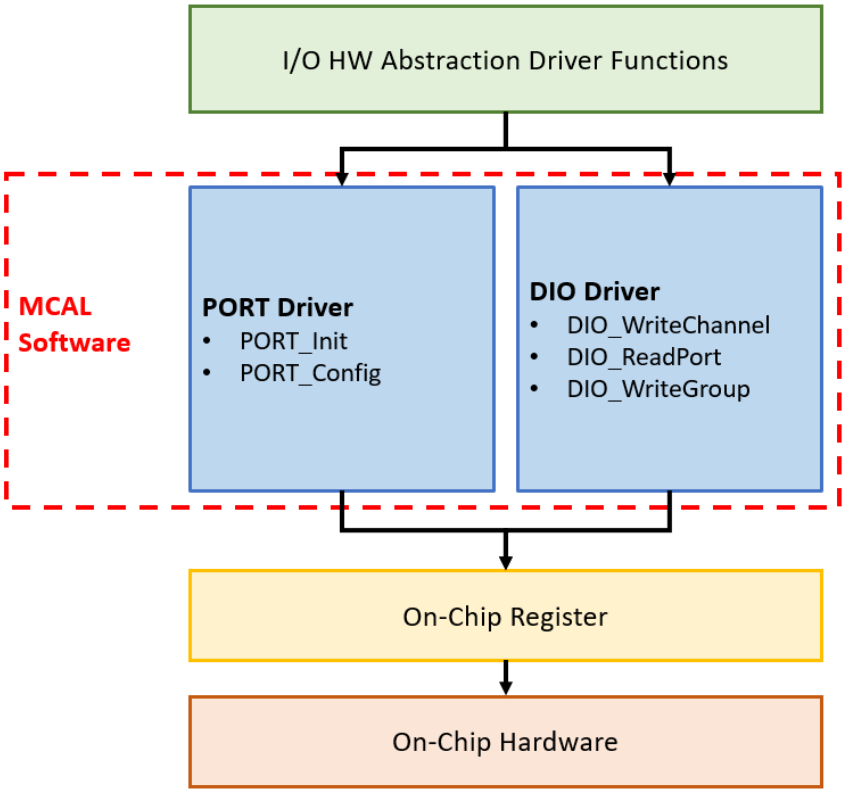

Abbildung 6.3: DIO Komponenten AUTOSAR Architektur [7]

von Hardware/Software Integration Tests wurden folgende Requirements formuliert, welche mit den implementierten Tests verlinkt sind:

- Das Auslesen von physikalischen, digitalen High (5V) und Low (GND) Werten an unterschiedlichen Pins des Mikrocontrollers soll verifiziert werden.
- Das Setzen von physikalischen, digitalen High (5V) und Low (GND) Werten an unterschiedlichen Pins des Mikrocontrollers soll verifiziert werden.

Um die formulierten Requirements auf der Zielumgebung zu verifizieren, wurde die DIO Software Komponente mit einer plattformspezifischen statischen Mikrocontroller Pinkonfiguration programmiert. Mittels Methoden der *Hardware/Software Integration Verification* DLL in einem *CANoe* Projekt wurden folgende Funktionen getestet:

- **Dio_ReadChannel**: Mittels dieser Funktion werden logische Pegel an einem Pin ausgelesen. Dabei wird der Pin als Laufnummer mittels eines Übergabeparameters an die Funktion identifiziert.
- **Dio_WriteChannel**: Mittels dieser Funktion werden logische Pegel an einem Pin angelegt, wobei der Pegel als Übergabeparameter der Funktion übergeben wird. Der Pin als Laufnummer wird mittels eines Übergabeparameters an die Funktion identifiziert.

Für die Verifikation der beschriebenen Funktionen werden die gemessenen Werte mit den erwarteten Werten verglichen, wobei eine Toleranz berücksichtigt wird. Um sämtliche Grenzen des DIO Treibers auszuloten, werden Spannungen entsprechend der logischen Pegel High (5V) und Low (0V) an den Mikrocontroller Pins angelegt und gemessen.
Am HIL Testsystem werden mittels der Schaltung in Abbildung 6.4 logische Spannungspegel am jeweiligen Pin des Mikrocontrollers zur Auswertung angelegt. Bei logischem Pegel High schaltet der obere Transistor die Versorgungsspannung V_{dd} von 5V an den Pin. Bei logischem Pegel Low schaltet der untere Transistor das gemeinsame Massepotential an den Pin des Mikrocontrollers. Bei Auswertung eines logischen Input Pegels am HIL Testsystems wird der jeweilige Kanal so konfiguriert, dass der Pegel direkt anliegt. Zum Schutz des Testsystems ist jeder Kanal durch eine 200 mA Sicherung geschützt. Angemerkt sei hier allerdings, dass die 200mA Sicherung keinen Schutz für den Mikrocontroller bietet. Dies gilt für sämtliche Sicherungen des HIL Testsystems.

6.2 Analog Digital Conversion (ADC) Modul

Im Analog Digital Conversion (ADC) Modul soll eine an einem konfigurierten Pin angelegte Spannung im Bereich zwischen High (5V) und Low (0V) Pegel dynamisch gemessen werden [68]. Dabei soll die Spannung auf einen Wert mit einer gewissen Auflösung (hier 10 Bit) abgebildet werden. Diese Werte werden aus dem jeweiligen Datenregister bei Ausführung eines Timer getriggerten End of Conversion (EOC) Interrupts ausgelesen.
Die genauen Vorgaben, nach welchen die Komponente entwickelt wurde, befinden sich in [70]. Dabei werden die Schnittstellen, Konfigurationsparameter sowie die unterschiedlichen Betriebsmodi und Sequenzdiagramme zu Funktionen beschrieben. In [71] sind AUTOSAR Requirements für diese Komponente formuliert. Für die Durchführung von Hardware/Software Integration Tests wurden folgende Requirements formuliert, welche mit den implementierten Tests verlinkt sind:

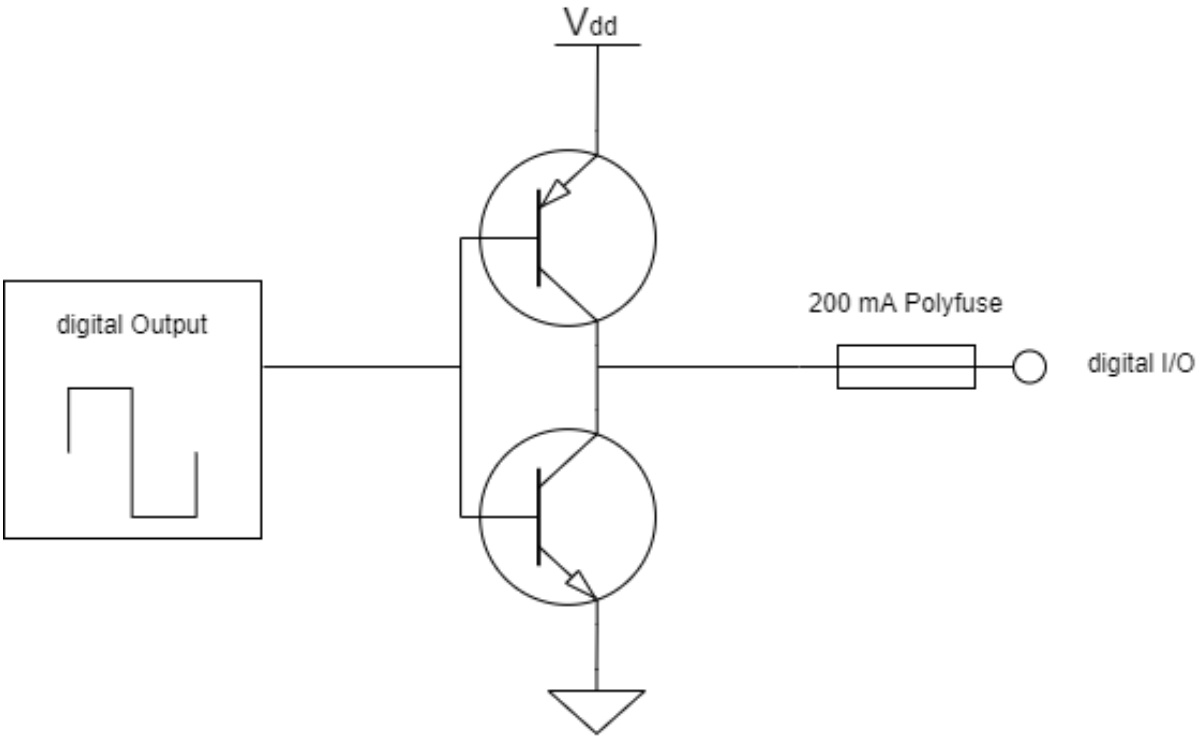

Abbildung 6.4: Messschaltung für DIO Komponente [8]

- Das Auslesen von physikalischen, analogen Werten, im Bereich 0V bis 5V, an unterschiedlichen Pins des Mikrocontrollers soll verifiziert werden.
- Das Auslesen von physikalischen, analogen Werten, im Bereich 0V bis 5V, in 10 Bit Auflösung soll verifiziert werden.

Um die formulierten Requirements auf der Zielumgebung zu verifizieren, wurde die ADC Software Komponente mit einer plattformspezifischen statischen Mikrocontroller Pinkonfiguration programmiert. Des Weiteren wurde eine statische Konfiguration der LLD Komponente gewählt. Mittels Methoden der *Hardware/Software Integration Verification* DLL in einem *CANoe* Projekt wurden folgende Funktionen getestet:

- **Adc_Init**: Mittels dieser Funktion wird die Software Komponente entsprechend ihrer statischen Konfiguration initialisiert. Teil der Konfiguration ist dabei die Zuweisung eines Hardware Treibers, welcher das Register Set des IP-Cores am Mikrocontroller abbildet. Ein Mikrocontroller kann mehrere dieser Hardware Treiber beinhalten. In den Registern des Hardware Treibers wird die Auflösung festgelegt und Parameter der ADC Konvertierung bestimmt. Die Abtastrate, so wie die Anzahl der Durchläufe pro Messung sind Teil dieser Parameter. Pro Abtastung wird der EOC Interrupt ausgeführt, welcher den aktuellen digitalen Spannungswert je nach Auflösung erfasst. Dabei wird basierend auf die Anzahl der Durchläufe, letztendlich der arithmetische Mittelwert errechnet. Des Weiteren wird eine Messung auf einem ADC Kanal durchgeführt, welcher einer entsprechenden Pin Konfiguration am Mikrocontroller zu Grunde liegt. Die ADC Komponente basiert intern auf einer State Maschine. Zur Verifikation der funktionalen Korrektheit der Funktion *Adc_Init* wird der Status der ADC Komponente abgefragt.

- **Adc_SetupResultBuffer**: Mittels dieser Funktion wird ein Buffer dem ADC Modul zugewiesen, in welchen die gemessenen Werte geschrieben werden. Basierend auf dem Rückgabewert der Funktion kann festgestellt werden, ob der Buffer richtig initialisiert wurde.
- **Adc_StartGroupConversion**: Mittels dieser Funktion wird die ADC Messung gestartet, was bedeutet, dass nach dem Auslösen des EOC Interrupts der gemessene Wert zum Lesen zur Verfügung steht. Zur Verifikation wird der interne Status der ADC Komponente abgefragt.
- **Adc_ReadGroup**: Mittels dieser Funktion wird der Buffer ausgelesen.

Für die Verifikation der beschriebenen Funktionen werden die gemessenen Werte mit den erwarteten Werten verglichen, wobei eine Toleranz berücksichtigt wird. Um sämtliche Grenzen des ADC Treibers auszuloten, werden Spannungen von 0V, 2.5V und 5V ausgewertet.
Am HIL Testsystem werden mittels der schematischen Schaltung in Abbildung 6.5 analoge Spannungswerte in Relation zum GND Potential erzeugt und dem Eingangspin des Mikrocontrollers zu Verfügung gestellt. Für die Konvertierung wird durch das HIL Testsystem ein konstanter, digitaler Spannungspegel vorgegeben. Zum Schutz des Testsystems ist jeder Kanal durch eine 200 mA Sicherung geschützt.

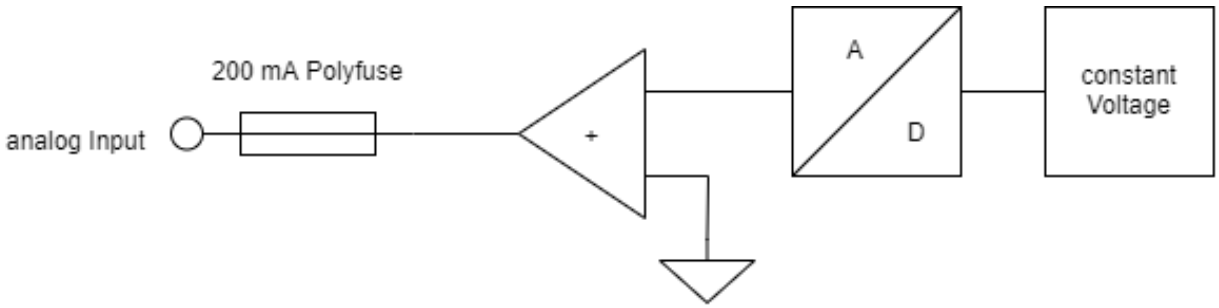

Abbildung 6.5: Messschaltung für ADC Komponente [8]

6.3 Pulse Width Modulation (PWM) Modul

Das PWM Modul wird dazu verwendet, um Pulse oder Rechtecksignale mit einer variablen Pulsweite und Frequenz zu erstellen [68]. Dabei wird der enhanced Modular Input/Output Subsystem (eMIOS) Timer des Controllers in Kombination mit Capture und Compare Registern verwendet. Die genauen Vorgaben, nach welchen die Komponente entwickelt wurde, befinden sich in [72]. Dabei werden die Schnittstellen, Konfigurationsparameter sowie Sequenzdiagramme zu Funktionen beschrieben. In [73] sind AUTOSAR Requirements für diese Komponente formuliert. Für die Durchführung von Hardware/Software Integration Tests wurden folgende Requirements formuliert, welche mit den implementierten Tests verlinkt sein sollen:

- Das Setzen von unterschiedlichen Duty Cyles eines PWM Signales soll an unterschiedlichen Pins des Mikrocontrollers verifiziert werden.
- Das Setzen von unterschiedlichen Offset eines PWM Signales soll an unterschiedlichen Pins des Mikrocontrollers verifiziert werden.

- Das Setzen von unterschiedlichen Frequenzen eines PWM Signales soll an unterschiedlichen Pins des Mikrocontrollers verifiziert werden.
- Das Setzen von unterschiedlichen Duty Cycle in % je nach Polarität soll verifiziert werden.

Um die formulierten Requirements auf der Zielumgebung zu verifizieren, wurde die PWM Software Komponente mit einer plattformspezifischen statischen Mikrocontroller Pinkonfiguration programmiert. Des Weiteren wurde eine statische Konfiguration der LLD Komponente gewählt. Mittels Methoden der *Hardware/Software Integration Verification* DLL in einem *CANoe* Projekt wurden folgende Funktionen getestet:

- **Pwm_Init**: Mittels dieser Funktion wird die Software Komponente entsprechend ihrer statischen Konfiguration initialisiert. Teil der Konfiguration ist wiederum die Zuweisung eines Hardware Treibers. In den Registern des Hardware Treibers wird die Frequenz des PWM Signals festgelegt sowie die Polarität. Dabei wird konfiguriert, welchen Spannungspegel der IDLE Zustand des PWM Signals entspricht, also ob dieser bei einem Duty Cycle von 0% gleich Low oder High ist.
- **Pwm_SetOutputToIdle**: Mittels dieser Funktion wird der Spannungspegel am Ausgang zum konfigurierten IDLE Status des PWM Kanals gesetzt.
- **Pwm_SetDutyCycle**: Mittels dieser Funktion wird an unterschiedlichen PWM Kanälen ein Duty Cycle gesetzt.
- **Pwm_SetPeriod**: Mittels dieser Funktion wird an unterschiedlichen PWM Kanälen die Periodendauer der Rechtecksignale eingestellt. Diese Periodendauer wird seitens des HIL Testsystems über eine Frequenzmessung verifiziert. Dabei werden Testfälle für Signale unterschiedlicher Frequenz ausgeführt.

Für die Verifikation der beschriebenen Funktionen werden die gemessenen Werte mit den erwarteten Werten verglichen, wobei eine Toleranz berücksichtigt wird. Um sämtliche Grenzen des PWM Treibers auszuloten, werden Duty Cycles von 0%, 50% und 100% ausgewertet.
Am HIL Testsystem werden mittels der schematischen Schaltung in Abbildung 6.6 digitale Spannungswerte in Relation zum GND Potential abgetastet. Dabei werden die Spannungspegel mit einem konfigurierten Schwellwert verglichen, um diese dann einem logischen Pegel zuzuweisen und mittels einer PWM Messung auszuwerten. Der für die Messungen konfigurierte Schwellwert entspricht 2.5V für einen Spannungsbereich im Intervall 0V bis 5V. Des Weiteren sinkt mit steigender Frequenz die Genauigkeit bei Abtastung des PWM Signals, wobei 200 kHz als Grenzfrequenz des HIL Moduls deklariert sind. Zum Schutz des Testsystems ist jeder Kanal durch eine 200 mA Sicherung geschützt.

6.4 Serial Peripheral Interface (SPI) Modul

Das SPI Modul wird dazu verwendet, um mittels einer asynchronen full-duplex Master/Slave SPI Kommunikation Hardware Peripherien des Gesamtsystems, also die zur Software Komponente zughörigen Hardware Bauteile, anzusteuern [68]. Dazu muss der Transceiver des Controllers als Master konfiguriert werden. Daten werden dabei nach dem First In First Out (FIFO) Buffer Prinzip verarbeitet. Der SPI ist eine mit vier Leitungen ausgestattete serielle Schnittstelle:

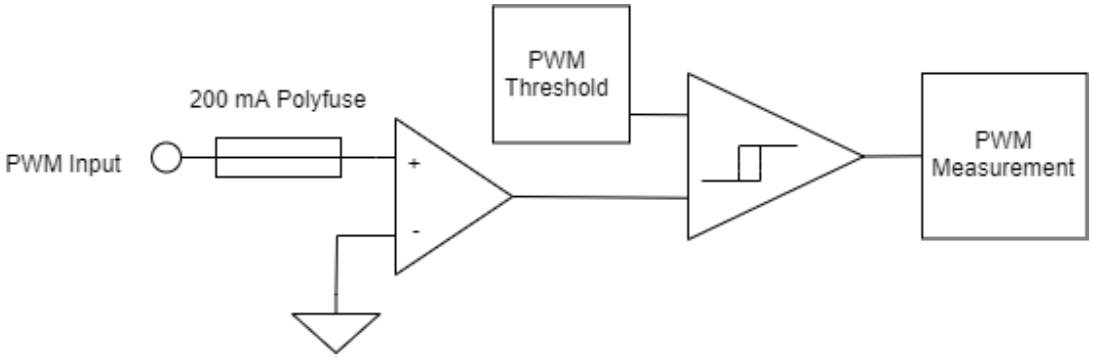

Abbildung 6.6: Messschaltung für PWM Komponente [8]

- Die CS Leitung genehmigt die Kommunikation über den Kanal.
- Die Master Output Slave Input (MOSI) Leitung stellt die serielle Datenausgabe aus Sicht des Masters zur Verfügung.
- Die Master Input Slave Output (MISO) Leitung stellt den seriellen Dateneingang aus Sicht des Masters zur Verfügung.
- Die Clock Leitung wird benötigt, um die Daten an der Gegenstelle im richtigen Moment abzutasten.

Die genauen Vorgaben, nach welchen die Komponente entwickelt wurde, befinden sich in [74]. Dabei werden die Schnittstellen, Konfigurationsparameter sowie Sequenzdiagramme zu Funktionen beschrieben. In [75] sind AUTOSAR Requirements für diese Komponente formuliert.
Für die funktionale Verifikation der SPI Software Komponente soll die asynchrone Übertragung von Lese- und Schreibenachrichten an unterschiedlichen Kanälen verifiziert werden. Um die SPI Komponente auf der Zielumgebung zu verifizieren, wurde die Software Komponente mit einer plattformspezifischen statischen Mikrocontroller Pinkonfiguration programmiert. Des Weiteren wurde eine statische Konfiguration der LLD Komponente gewählt. Mittels Methoden der *Hardware/Software Integration Verification* DLL in einem *CANoe* Projekt wurden folgende Funktionen getestet:

- **Spi_Init**: Mittels dieser Funktion wird die Software Komponente entsprechend ihrer statischen Konfiguration initialisiert. Teil der Konfiguration ist wiederum die Zuweisung eines Hardware Treibers. Die Register des Hardware Treibers beeinflussen schließlich die Signale eines SPI Kanals, welche in Abbildung 6.7 demonstriert sind. Die Übertragungsrate eines Bits sowie die Polarität der SPI Signale werden in den Registern konfiguriert. Des Weiteren werden die Wartezeiten innerhalb und zwischen den einzelnen SPI Sequenzen konfiguriert. Damit ist die Wartezeit nach dem Auslösen des CS bis zur Übertragung der Daten, die Wartezeit nach der Übertragung der Daten bis zum Ende der Sequenz und die minimale Wartezeit zwischen den SPI Sequenzen gemeint. Auch der Duty Cycle der Clock Leitung wird mittels der Register des SPI Hardware Treibers konfiguriert. Die Bits der Datenleitung können je nach Konfiguration bei steigender oder fallender Flanke interpretiert werden. Des Weiteren werden die SPI Signale über einen Kanal ausgegeben, welcher entsprechende Pin Konfigurationen am Mikrocontroller hat.

- **Spi_WriteIB**: Mittels dieser Funktion wird der Write Input Buffer der SPI MOSI Daten als Pointer zugewiesen. In diesen Buffer werden sämtliche Bytes geschrieben, welche an den Slave übertragen werden sollen. Die richtige Ausführung dieser Funktion wird über den Rückgabewert verifiziert.
- **Spi_ReadIB**: Mittels dieser Funktion wird der Read Input Buffer der SPI MISO Daten als Pointer zugewiesen. Vom diesem Buffer werden sämtliche Bytes gelesen, welche der Slave an den Master geschickt hat. Die richtige Ausführung dieser Funktion wird über den Rückgabewert verifiziert.
- **Spi_AsyncTransmit**: Mittels dieser Funktion wird die SPI Sequenz am jeweiligen Kanal ausgelöst. Im Zuge der Tests stellt das HIL Testsystem den Slave der Übertragung dar. Verifiziert wird die Funktion dadurch, dass am Slave überprüft wird, ob die Bytes im Write Input Buffer ausgelesen werden konnten und ob die Bytes im Read Input Buffer jenen Bytes entsprechen, welche am Slave zugewiesen wurden.

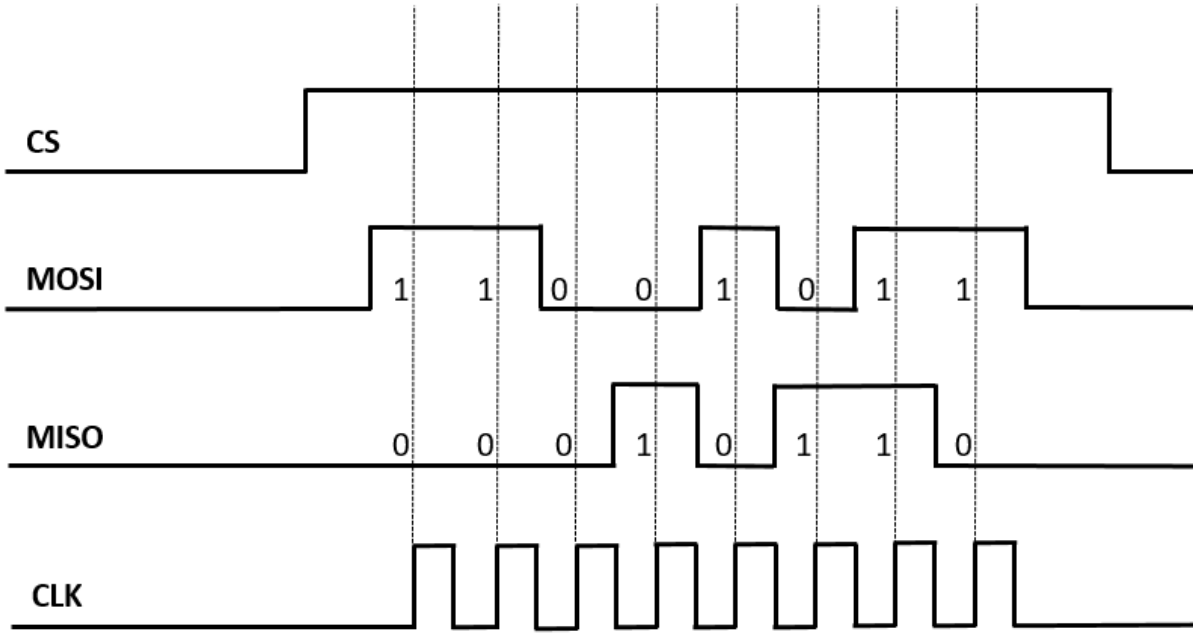

Abbildung 6.7: SPI Signale

Für die Verifikation der beschriebenen Funktionen werden die gemessenen Werte mit den erwarteten Werten verglichen, wobei eine Toleranz berücksichtigt wird. Für die richtige Interpretation der physikalischen Signale zu den Bits der seriellen Kommunikation des SPI Treibers werden Spannungen entsprechend der logischen Pegel High (5V) und Low (0V) an den Mikrocontroller Pins ausgewertet.
Am HIL Testsystem werden die Signale der SPI Leitungen auf die Werte gesetzt, welche am jeweiligen Modul des *VT System* konfiguriert sind und dann als DIO Signale eingelesen und ausgegeben [8]. Die entstandenen digitalen Signale werden interpretiert und ausgewertet, je nach Zuweisung als Master oder Slave des *VT System* Moduls. Ein SPI Modul wird am *VT System* simuliert, um ein echtzeitfähiges Verhalten der Kommunikationschnittstelle zu gewährleisten. Dabei muss diese mit denselben Parameterwerten konfiguriert werden, wie die Gegenstelle.

6.5 Test Report für automatisierte Testläufe

Für die Evaluierung der beschriebenen LLD Software Module auf einer Mikrocontroller Zielumgebung wurde ein *CANoe* Projekt erstellt, welches für die Ausführung der Tests benötigten Testknoten beinhaltet. Dieser wird als Event während der Simulation ausgeführt. Sämtliche notwendigen Module des *VT Systems* wurden entsprechend der Konfigurationen der zugewiesenen und kontaktierten LLD Software Module konfiguriert, um die Instrumentierung und Messung am HIL Testsystem zu gewährleisten. Diese Konfigurationen werden als Teil der Testgruppen Initialisierung im Testknoten ausgeführt, wie in Abbildung 6.8 ersichtlich ist. Teil dieser Testgruppen Initialisierung ist auch die automatisierte Kompilierung der zu testenden Software Komponenten. Teil dieser kompilierten Software sind die Initialisierungsroutinen und Konfigurationen sämtlicher notwendiger Hardware Komponenten des Mikrocontrollers sowie statische Konfigurationen der LLD Software Module. Des Weiteren ist auch die Initialisierung der *Hardware/Software Integration Verification* DLL und somit auch die Verbindung zum Debugger, das Öffnen eines *WinIDEA* Workspaces und das Programmieren des ELF Files, basierend auf dem Output des automatisierten Build-Prozesses, Teil der Testgruppe Initialisierung.
Nach der Initialisierung werden sämtliche Tests, welche die Funktionen der einzelnen LLD Software Module verifizieren, ausgeführt. Jedes LLD Software Modul wird in einer eigenen Testgruppe gekapselt. Als Precondition wird der Mikrocontroller vor jedem Test neu initialisiert, um eine Unabhängigkeit zwischen den einzelnen Tests zu schaffen. Um eine unterbrechungsfreie Ausführung der einzelnen Funktionen auf der Zielumgebung zu gewährleisten, werden Interrupts deaktiviert. Ausgenommen sind hier Module dessen korrekte Funktionalität auf der Ausführung von Interrupts basiert, wie z.B. der EOC Interrupt des ADC Moduls.
Abbildung 6.8 demonstriert die Ausführung eines Testlaufes. Die Ausführungszeit eines Tests, sowie der zugehörigen Testgruppe und aller Tests insgesamt, wird aufgezeichnet. Es wird erfasst, ob ein Test bestanden hat. Wenn alle Tests bestanden haben ist der Testlauf geglückt. Wenn sämtliche Tests, welche zu einem Requirement gelinkt sind, bestehen, dann gilt dieses Requirement als verifiziert und damit umgesetzt. Pro Testlauf wird ein HTML Report erstellt, welcher genauere Informationen zu den einzelnen Tests beinhaltet. Hat ein Test nicht bestanden, so kann mittels des Reports der Grund für das Fehlverhalten analysiert und detektiert werden. Abbildung 6.9 demonstriert eine Übersicht zu den Testfällen der Testgruppe *DIO Tests*. Dabei wird pro Testfall jeder einzelne Testschritt aufgezeichnet und einem Zeitstempel zugeordnet, wie in Abbildung 6.10 demonstriert. Die Bedingung für die erfolgreiche Durchführung eines Testes ist einerseits die Terminierung des Tests, andererseits kann diese auch durch eine Methode der *Verification* DLL gekennzeichnet werden. Die *Verification* DLL stellt dabei die Methoden für die Erstellung eines Reports zur Verfügung.

6.6 Testaufbau für automatisierte und manuelle Tests

Im Zuge der Evaluierung der Hardware/Software Integration Plattform (Kapitel 7) wurde die Verifikation der LLD Software Komponenten mittels automatisierten und manuellen Tests durchgeführt. Die Verifikation wurde hinsichtlich der Ausführungszeit verglichen und analysiert. Dafür wurden zwei unterschiedliche Testaufbauten geschaffen.
Für die Verifikation mit automatisierten Tests wurde der Testaufbau in Abbildung 6.11 geschaffen. Das SPC56 Evaluation Board, auf welchem die zu verifizierenden Software Komponenten programmiert sind, ist mit den folgenden Modulen des HIL Testsystems verbunden:

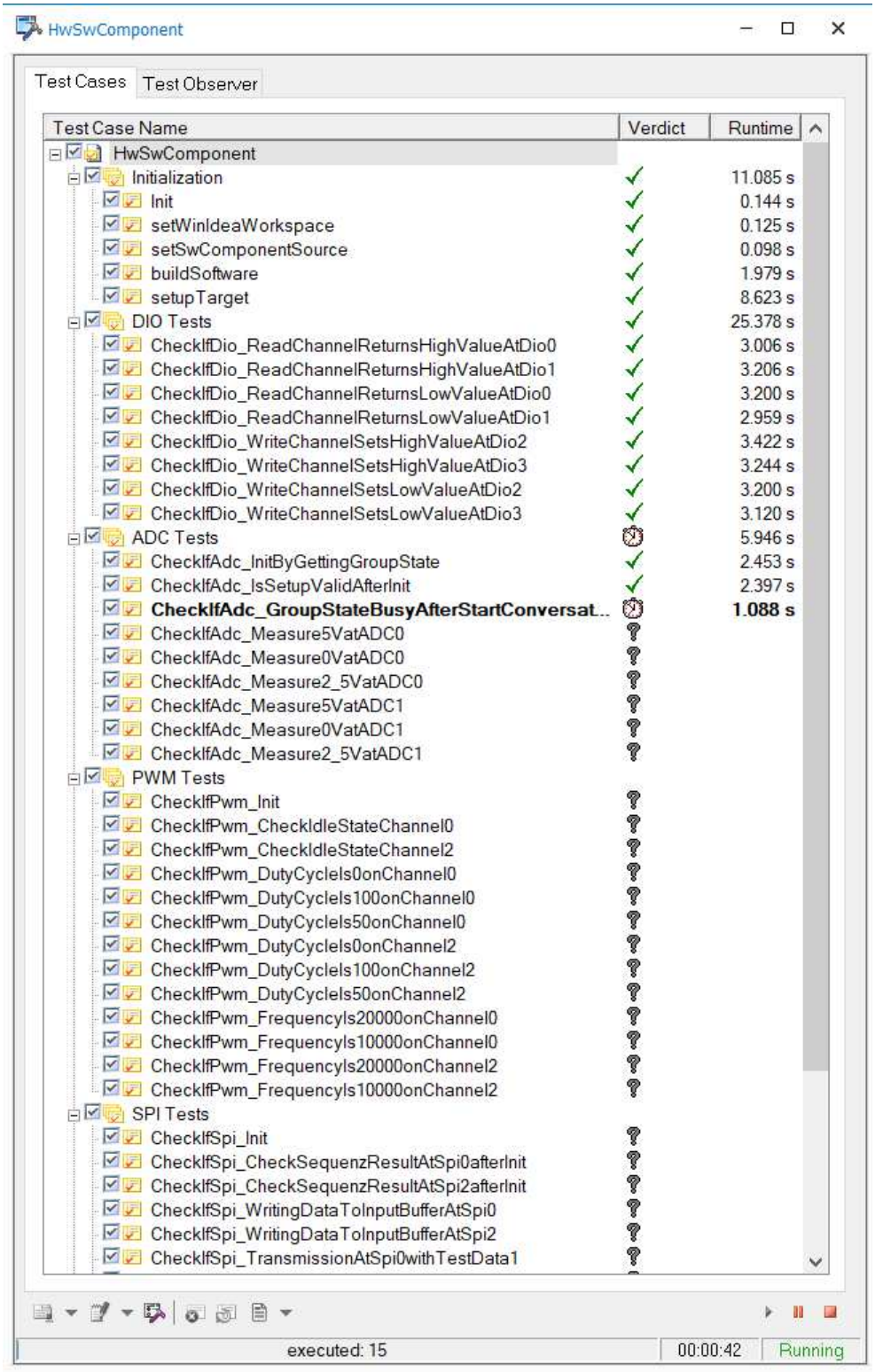

Abbildung 6.8: Ausführung der Testfälle bei automatisierten Tests

2	DIO	DIO Tests	
2.1		CheckIfDio_ReadChannelReturnsHighValueAtDio0	pass
2.2		CheckIfDio_ReadChannelReturnsHighValueAtDio1	pass
2.3		CheckIfDio_ReadChannelReturnsLowValueAtDio0	pass
2.4		CheckIfDio_ReadChannelReturnsLowValueAtDio1	pass
2.5		CheckIfDio_WriteChannelSetsHighValueAtDio2	pass
2.6		CheckIfDio_WriteChannelSetsHighValueAtDio3	pass
2.7		CheckIfDio_WriteChannelSetsLowValueAtDio2	pass
2.8		CheckIfDio_WriteChannelSetsLowValueAtDio3	pass

Abbildung 6.9: Report der automatisierten Tests der Testgruppe DIO

[−] **2.1 CheckIfDio_ReadChannelReturnsHighValueAtDio0: Passed**

Main Part of Test Case

Timestamp	Test Step	Description	Result
4759.915750		[TWO]	-
4759.915750	ARR	Started Arrange [TWO]	-
4760.498764	ARR	Finished Arrange [TWO]	-
4760.498764		[TWO]	-
4760.498764		[TWO]	-
4760.498764	ACT	Started Act [TWO]	-
4761.504810	ACT	Finished Act [TWO]	-
4761.504810		[TWO]	-
4761.504810		[TWO]	-
4761.504810	AST	Started Assert [TWO]	-
4761.504810	OUT	Read Level 1 at GPIO Pin 66 [THREE]	pass
4761.504810	AST	Finished Assert [TWO]	-
4761.504810		[TWO]	-

Abbildung 6.10: Report eines Testfalls bei automatisierten Tests

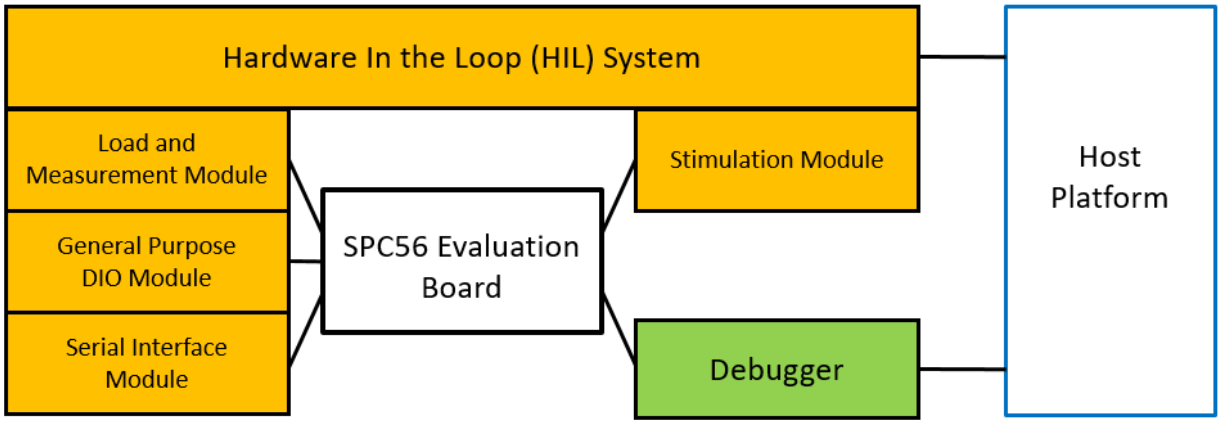

Abbildung 6.11: Testaufbau für automatisierte Tests im Zuge der Evaluierung

- **Load and Measurement Modul**: Mit diesem Modul werden die PWM Signale gemessen im Zuge der Verifikation der PWM LLD Software Komponente.
- **General-Purpose DIO Modul**: Mit diesem Modul werden die DIO Signale gemessen und instrumentiert im Zuge der Verifikation der DIO LLD Software Komponente.
- **Serial Interface Modul**: Mit diesem Modul werden die SPI Nachrichten empfangen und interpretiert im Zuge der Verifikation der DIO LLD Software Komponente.
- **Stimulation Modul**: Mit diesem Modul werden analoge Spannungspegel generiert im Zuge der Verifikation der ADC LLD Software Komponente.

Für die Implementierung und Ausführung der Tests wird das Tool *CANoe* verwendet, mit welchem die Module des HIL Testsystem auch konfiguriert wurden. Mittels der entwickelten *Hardware/-Software Integration Verification* DLL wird der Zugriff auf den Debugger und somit auch auf den Mikrocontroller bewerkstelligt. Das Management sämtlicher Tools, sowie der Zugriff auf die gesamte Hardware des Testaufbaus wird automatisiert durch die Host Plattform bewerkstelligt. Im Folgenden werden die einzelnen Arbeitsschritte erläutert, welche notwendig sind, um die DIO Software Komponente entsprechend der Funktionalitäten in Abschnitte 6.1 automatisiert mit der Hardware/Software Integration Test Plattform zu verifizieren:

1. **Erstellen des Testaufbaus**: Das SPC56 Evaluation Board wird an eine 5V Spannungsversorgung kontaktiert und der Debugger wird angeschlossen. Zwei Pins des Mikrocontroller, welche die Funktionen eines DIO Ports zu Verfügung stellen, werden durch Steckverbindung mit zwei Kanälen des *General-Purpose DIO Modul* des HIL Testsystems verbunden. Die Spannungspegel 5V und GND werden an Schnittstellen des HIL Moduls kontaktiert, um eine Interpretation von High und Low Spannungspegeln zu gewährleisten.
2. **Kompilieren der Software Komponente**: Die zu verifizierende DIO Software Komponente wird kompiliert, um ein ELF File zu erstellen. Mittels dieses ELF Files sind die Symbole der Software für den Debugger interpretierbar. Für das Kompilieren der Software Komponente ist sowohl eine Registerkonfiguration des Mikrocontrollers, als auch die Software Komponente selbst notwendig. Bei der DIO Komponente entspricht die Konfiguration der Software Komponente der Zuweisung zu den verwendeten Pins des Mikrocontrollers. Der Kompiliervorgang kann auch im Zuge der Initialisierung des Testsystems, welcher im nächsten Schritt beschrieben wird, in einem automatisierten Build Prozess ausgeführt werden.
3. **Erstellen eines *CANoe* Projekts**: In einem *CANoe* Projekt wird ein Testknoten erstellt. Entsprechend der Beschreibung zur Abbildung 5.11 werden in diesem Testknoten die Initialisierung des Testsystems und die einzelnen Testfälle programmiert. Dazu werden die Klassenbibliotheken der Abbildung 5.10 verwendet. Für die Initialisierung des HIL Testsystems bei der Verifikation der DIO werden die Spannungspegel der Kanäle des *General-Purpose DIO Modul* konfiguriert. Bei der Initialisierung der Mikrocontroller Zielumgebung wird eine Verbindung zum Debugger aufgebaut und das ELF File programmiert. Die allgemeine Initialisierungsroutine wird in Abbildung 5.12 demonstriert. Die Implementierung der einzelnen Testfälle entspricht der Routine in Abbildung 5.13. Für die Verifikation der DIO Software Komponente wurden Tests für die Funktionen *Dio_ReadChannel* und *Dio_WriteChannel* geschaffen, welche in Abschnitt 6.1 erklärt sind. Um eine möglichst hohe Testabdeckung zu erzielen wurden diese Funktionen an sämtlichen konfigurierten Ports angewandt und jeweils die High und Low Pegel instrumentiert.

4. **Ausführung der Tests**: Nach Aktivierung der Spannungsversorgung werden im *CANoe* Projekt die Testfälle ausgeführt entsprechend Abbildung 6.8. Der erstellte HTML Report kann dann entsprechend der Granularität in den Abbildungen 6.9 und 6.10 analysiert werden. Die gemessenen Laufzeiten der einzelnen Testfälle wurden in Kapitel 7 evaluiert.

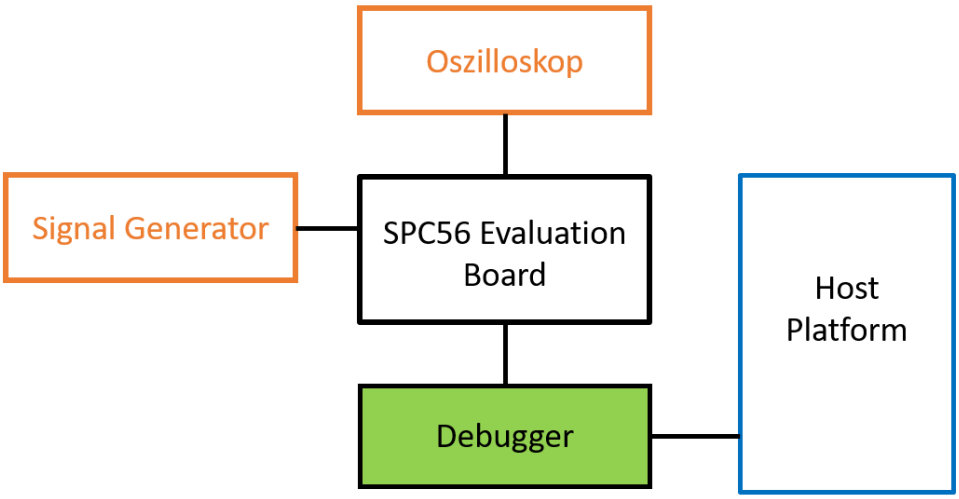

Abbildung 6.12: Testaufbau für manuelle Tests im Zuge der Evaluierung

Für die Verifikation mit manuellen Tests wurde der Testaufbau in Abbildung 6.12 geschaffen. Die TesterIn hat Zugriff auf den Debugger und auf die Software, welche am SPC56 Evaluation Board programmiert ist, über das Tool *WinIDEA*, welches auf der Host Plattform ausgeführt wird. Die Software wird Instruktion für Instruktion ausgeführt und umgebende physikalische Signale werden händisch gemessen und instrumentiert. Dazu werden Messgeräte wie ein Oszilloskop oder Signal Generatoren verwendet.
Im Folgenden werden die einzelnen Arbeitsschritte erläutert, welche notwendig sind, um die DIO Software Komponente entsprechend der Funktionalitäten in Abschnitte 6.1 manuell zu testen:

1. **Erstellen des Testaufbaus**: Das SPC56 Evaluation Board wird an eine 5V Spannungsversorgung kontaktiert und der Debugger wird angeschlossen. Ein Signalgenerator, welcher konstante Spannungspegel von 5V und GND generiert, sowie ein Oszilloskop werden verbreitet. Als Signalgenerator wird das HIL Testsystem verwendet, um händisch Spannungspegel einzuprägen.

2. **Kompilieren der Software Komponente**: Das Erstellen und Kompilieren der Software Komponente ist äquivalent zum Arbeitsschritt der automatisierten Tests. Ergänzend müssen für die Verifikationen der Funktionen *Dio_ReadChannel* und *Dio_WriteChannel* der DIO Software Komponente diese zyklisch in der Software ausgeführt werden.

3. **Erstellung der manuellen Testfälle**: Für die Erstellung der manuellen Testfälle wurde das Tool *Polarion*[1] verwendet. Mittels dieses Tools kann der Lebenszyklus eines entwickelten Systems gemanagt werden. Dabei kann ein Testlauf angelegt und Testfälle diesem zugewiesen werden. Bei der Ausführung des Testlaufes werden die gemessenen Werte der Testschritte eines Testfalls analysiert und ein Report erstellt. Abbildung 6.13 demonstriert einen Testfall in einem manuellen Testlauf für die Verifikation der Funktion *Dio_ReadChannel*. Wie

[1] `https://polarion.plm.automation.siemens.com/`, Online: Letzter Zugriff am 10.04.2020

bereits bei den automatisierten Tests wurden die Funktionen der DIO Software Komponente mit einer möglichst hohen Testabdeckung getestet.

4. **Ausführung der Tests**: Bei der Ausführung der Tests zur Verifikation der Funktionen *Dio_ReadChannel* und *Dio_WriteChannel* wurde der Mikrocontroller vor jedem Testfall in den Reset Zustand versetzt und anschließend initialisiert. Somit wurde eine Unabhängigkeit zwischen den einzelnen Testfällen geschaffen. Bei der Verifikation der Funktion *Dio_ReadChannel* wurde mittels des Signalgenerators ein High oder Low Spannungspegel an einem Mikrocontroller Pin angelegt. Beim Debuggen der Funktion wurde ein entsprechender Übergabeparameter manipuliert, um den jeweiligen Mikrocontroller Pin anzusprechen. Dazu können im Tool *WinIDEA* die Symbole der Software gesucht und modifiziert werden. Für die Verifikation der Funktion *Dio_WriteChannel* wird, je nach Modifikation der Symbole im Tool *WinIDEA*, ein Spannungspegel an einem Mikrocontroller Pin ausgegeben, welcher mit dem Oszilloskop gemessen wurde. Jeder einzelne gemessene Wert wird in den entsprechenden Testschritt des Testfalls eingetragen und bewertet. Nach Bewertung aller Testschritte werden die eingetragenen Werten des gesamten Testfalls mit den erwarteten verglichen um eine Bewertung für den gesamten Testfall abgeben zu können. Abbildung 6.13 demonstriert einen manuellen Testfall für die Verifikation der Funktion *Dio_ReadChannel*. Bei Fertigstellung der Ausführung sämtlicher Testfälle wird ein Report (siehe Abbildung 6.14) erstellt. Die gemessenen Laufzeiten der einzelnen Testfälle wurden in Kapitel 7 evaluiert.

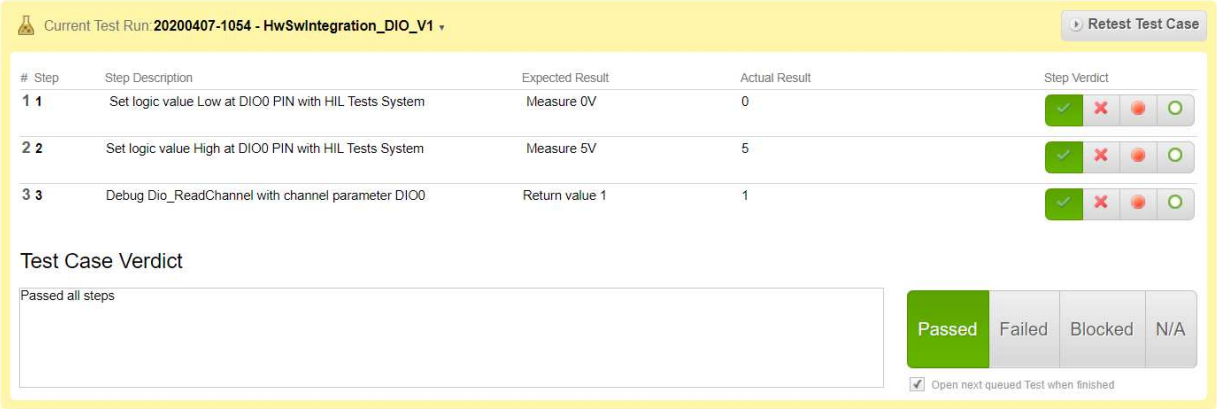

Abbildung 6.13: Struktur eines manuellen Testfalls

In Kapitel 7 wird eine Analyse der Laufzeiten der manuellen und automatisierten Tests durchgeführt. Dabei wird ausschließlich die tatsächliche Ausführungszeit eines einzelnen Tests betrachtet. Der Aufbau der Testsysteme und das Entwickeln der einzelnen Tests wurden nicht zeitlich erfasst. Diese Aspekte werden in der Diskussion zur Evaluierung in Kapitel 8 ergänzend betrachtet.

Tests - 20200407-1054 - HwSwIntegration_DIO_V1

Test Result	Test Case	Defect	Duration	Executed by	Executed
Passed	BSM-2897 - DIO Tests.CheckIfDio_WriteChannelSetsLowValueAtDio3 * Arrange: Trigger Dio_Write...		40.000 s	Muttenthaler Florian	2020-04-10 09:23
Passed	BSM-2896 - DIO Tests.CheckIfDio_WriteChannelSetsLowValueAtDio2 * Arrange: Trigger Dio_Write...		50.000 s	Muttenthaler Florian	2020-04-10 09:23
Passed	BSM-2895 - DIO Tests.CheckIfDio_WriteChannelSetsHighValueAtDio3* Arrange: Trigger Dio_Write...		45.000 s	Muttenthaler Florian	2020-04-10 09:22
Passed	BSM-2894 - DIO Tests.CheckIfDio_WriteChannelSetsHighValueAtDio2* Arrange: Trigger Dio_Write...		51.000 s	Muttenthaler Florian	2020-04-10 09:21
Passed	BSM-2893 - DIO Tests.CheckIfDio_ReadChannelReturnsLowValueAtDio1* Arrange: Set DIO1 Input t...		53.000 s	Muttenthaler Florian	2020-04-10 09:19
Passed	BSM-2892 - DIO Tests.CheckIfDio_ReadChannelReturnsLowValueAtDio0* Arrange: Set DIO0 Input t...		56.000 s	Muttenthaler Florian	2020-04-10 09:18
Passed	BSM-2891 - DIO Tests.CheckIfDio_ReadChannelReturnsHighValueAtDio1* Arrange: Set DIO1 Input...		76.000 s	Muttenthaler Florian	2020-04-10 09:15
Passed	BSM-2890 - DIO Tests.CheckIfDio_ReadChannelReturnsHighValueAtDio0 * Arrange: Set DIO0 Input...		80.000 s	Muttenthaler Florian	2020-04-10 09:14

Abbildung 6.14: Report der manuellen Tests der Testgruppe DIO

7 Evaluierung

Um die Effizienz der Plattform für die Automatisierung von Hardware/Software Integration Tests zu veranschaulichen, wurden die für die Evaluierung der LLD Software Module implementierten Testfälle sowohl automatisiert, als auch manuell durchgeführt. Die implementierten und analysierten Tests entsprechen der Verifikation der Funktionen der einzelnen Software Komponenten, wie diese in Kapitel 6 beschrieben sind. Die Messaufbauten der automatisierten und manuellen Tests sind in Kapitel 6.6 beschrieben. Die Ausführungszeiten der Testfälle beider Test-Setups wurden in den Reports in Kapitel 6 aufgezeichnet.
Sowohl bei automatisierten, also auch bei manuellen Tests ist die Laufzeit eines Tests durch die Summe der einzelnen Testschritte definiert:

$$t_{\text{Runtime}}[s] = t_{\text{Precondition}} + t_{\text{Arrange}} + t_{\text{Act}} + t_{\text{Assert}} + t_{\text{Postcondition}} \tag{7.1}$$

Diese einzelnen Testschritte werden in der Testfallbeschreibung spezifiziert und sind für automatisierte und manuelle Tests im Zuge der Evaluierung identisch. Im Besonderen gilt dies für die Vorbedingung zum Test, dessen Laufzeit durch $t_{\text{Precondition}}$ festgelegt wird. Hier wird die Initialisierung des Mikrocontrollers mit den zugehörigen LLD Software Komponenten durchgeführt. Die Initialisierung des Testsystems selbst wird nicht als Vorbedingung ausgeführt, da diese zwischen manuellen und automatisierten Tests auf Grund des unterschiedlichen Testaufbaus voneinander abweicht. Eine zeitliche Erfassung im Zuge der Evaluierung würde das Ergebnis verfälschen. Die Laufzeit der einzelnen Testschritte bei automatisierten Tests ergibt sich aus der Summe der Laufzeiten der einzelnen Tasks die das Testsystem mit integrierten *Hardware/Software Integration Verification* Framework in einem *CANoe* Testknoten benötigt. Bei manuellen Tests wird die Laufzeit eines Testschrittes durch die Bearbeitungszeit der TesterIn mittels des Debuggers und der Messgeräte definiert.

DIO Testfall	Laufzeit manueller Test [s]	Laufzeit automatisierter Test [s]	ADC Testfall	Laufzeit manueller Test [s]	Laufzeit automatisierter Test [s]
1	80	3,042	1	33	2,469
2	76	3,169	2	69	2,440
3	56	3,178	3	65	3,263
4	53	3,193	4	95	3,630
5	51	3,444	5	84	3,615
6	45	3,377	6	85	3,353
7	50	3,256	7	107	3,630
8	40	3,174	8	35	3,657
			9	83	3,743
$\sum$	451	25,891	$\sum$	656	29,839
PWM Testfall	**Laufzeit manueller Test [s]**	**Laufzeit automatisierter Test [s]**	**SPI Testfall**	**Laufzeit manueller Test [s]**	**Laufzeit automatisierter Test [s]**
1	62	2,157	1	38	2,038
2	47	2,781	2	15	2,505
3	62	2,809	3	36	2,555
4	49	3,960	4	31	2,598
5	47	3,924	5	30	2,483
6	52	3,890	6	31	2,717
7	42	4,106	7	63	2,781
8	67	4,194	8	54	2,891
9	44	4,051	9	48	3,085
10	66	3,927	10	Not Available (N/A)	2,677
11	35	3,927	11	N/A	2,657
12	33	3,634	12	N/A	2,595
13	18	4,195	13	N/A	2,678
$\sum$	624	47,555	$\sum$	N/A	34,338

Tabelle 7.1: Laufzeitmessungen manueller und automatisierter Tests der LLD Software Komponenten

Die Tabelle 7.1 zeigt die Ergebnisse der Laufzeitmessungen der einzelnen Testgruppen zur funktionalen Verifikation der einzelnen LLD Software Module. Dabei wurden jene Testaufbauten für automatisierte und manuelle Tests verwendet, welche in Kapitel 6.6 beschrieben sind. In diesen Kapitel ist die Anwendung der unterschiedlichen Testaufbauten für die funktionale Verifikation des DIO Moduls angeführt. Für die Veranschaulichung der einzelnen Testschritte für automatisierte und manuelle Tests dieser Komponente wird auf die Abbildungen 5.13 und 6.13 verwiesen. Der konzeptionelle Aufbau sämtlicher weiteren Tests der DIO Software Komponente, sowie der weiteren verifizierten Komponenten ADC, PWM und SPI, entspricht der Beschreibung zu diesen Abbildung. Dabei wurden jene Funktionen der einzelnen Komponenten verifizierte, welche in den Modulbeschreibungen des Kapitels 6 angeführt sind. In den Tabellen 7.2, 7.3, 7.4 und 7.5 werden die einzelne Testfälle zur Übersicht beschrieben. Erwähnt sei hier, dass Tests, welche nicht ausgeführt werden konnten, in der Tabelle 7.1 mit N/A gekennzeichnet wurden und auch für die weitere Analyse nicht herangezogen wurden. So konnten manuelle Tests der Testgruppe *SPI Tests* nicht ausgeführt werden, da diese das korrekte Auslesen von Bytes der MISO Leitung verifizieren. Dies konnte in den manuellen Tests nicht simuliert werden. Bei den automatisierten Tests hingegen konnte eine solche Simulation am HIL Testsystem durchgeführt werden.

Testfall	**Testfall Beschreibung**
1	Verifiziert ob die Funktion *Dio_ReadChannel* den Wert 1 zurück gibt bei anlegen von 5V an DIO Port 0
2	Verifiziert ob die Funktion *Dio_ReadChannel* den Wert 1 zurück gibt bei anlegen von 5V an DIO Port 1
3	Verifiziert ob die Funktion *Dio_ReadChannel* den Wert 0 zurück gibt bei anlegen von 0V an DIO Port 0
4	Verifiziert ob die Funktion *Dio_ReadChannel* den Wert 0 zurück gibt bei anlegen von 0V an DIO Port 1
5	Verifiziert ob die Funktion *Dio_WriteChannel* 5V an DIO Port 2 anlegt bei Übergabe des Wertes 1
6	Verifiziert ob die Funktion *Dio_WriteChannel* 5V an DIO Port 3 anlegt bei Übergabe des Wertes 1
7	Verifiziert ob die Funktion *Dio_WriteChannel* 0V an DIO Port 2 anlegt bei Übergabe des Wertes 0
8	Verifiziert ob die Funktion *Dio_WriteChannel* 0V an DIO Port 3 anlegt bei Übergabe des Wertes 0

Tabelle 7.2: Beschreibung der einzelnen Testfälle der DIO Komponente

Testfall	Testfall Beschreibung
1	Verifiziert die Terminierung der Initialisierungsroutine *Adc_Init*
2	Verifiziert ob die Funktion *Adc_SetupResultBuffer* das ADC Modul in Modulstatus *VALID* versetzt
3	Verifiziert ob die Funktion *Adc_StartGroupConversion* das ADC Modul in Modulstatus *BUSY* versetzt
4	Verifiziert ob die Funktion *Adc_ReadGroup* den Wert 5V in einer Auflösung von 10 Bit am ADC Port 0 ausließt
5	Verifiziert ob die Funktion *Adc_ReadGroup* den Wert 0V in einer Auflösung von 10 Bit am ADC Port 0 ausließt
6	Verifiziert ob die Funktion *Adc_ReadGroup* den Wert 2,5V in einer Auflösung von 10 Bit am ADC Port 0 ausließt
7	Verifiziert ob die Funktion *Adc_ReadGroup* den Wert 5V in einer Auflösung von 10 Bit am ADC Port 1 ausließt
8	Verifiziert ob die Funktion *Adc_ReadGroup* den Wert 0V in einer Auflösung von 10 Bit am ADC Port 1 ausließt
9	Verifiziert ob die Funktion *Adc_ReadGroup* den Wert 2,5V in einer Auflösung von 10 Bit am ADC Port 1 ausließt

Tabelle 7.3: Beschreibung der einzelnen Testfälle der ADC Komponente

Aus der Analyse der Laufzeitmessungen in Tabellen 7.1 können folgende Schlüsse gezogen werden:

- Bei Betrachtung der Gesamtausführung der Testgruppen *DIO Tests*, *ADC Tests* und *PWM Test* ist die Ausführungszeit von automatisierten Tests um 94,033% kürzer.

$$(1 - \frac{\sum Testgruppe_{\text{automatisiert}}}{\sum Testgruppe_{\text{manuell}}}) * 100\% = 94,033\% \tag{7.2}$$

- Bezogen auf die durchschnittliche Ausführungsdauer eines Testfalles ist die Ausführungszeit von automatisierten Tests um 99,996% kürzer.

$$(1 - \frac{\frac{\sum \text{Testfall}_{\text{automatisiert}}}{\text{Anzahl Tests}_{\text{automatisiert}}}}{\frac{\sum \text{Testfall}_{\text{manuell}}}{\text{Anzahl Tests}_{\text{manuell}}}}) * 100\% = 99,996\% \tag{7.3}$$

- Bezogen auf die durchschnittliche Ausführungsdauer einer Testgruppe ist die Ausführungszeit von automatisierten Tests um 99,337% kürzer.

$$(1 - \frac{\frac{\sum \text{Testgruppe}_{\text{automatisiert}}}{\text{Anzahl Testgruppen}_{\text{automatisiert}}}}{\frac{\sum \text{Testfall}_{\text{manuell}}}{\text{Anzahl Testgruppen}_{\text{manuell}}}}) * 100\% = 99,337\% \tag{7.4}$$

Die Auswertungen der Laufzeiten im Vergleich zwischen manuellen und automatisierten Tests zeigen, dass die Ausführung von automatisierten Tests eine enorme Reduktion in der Laufzeit mit sich bringt und damit bei der Durchführung von Verifikation im Hardware/Software Integrationsprozess Kosten spart. Die zeitaufwendige Entwicklung von Testsystemen für automatisierte Hardware/Software Integration Tests ist damit gerechtfertigt.
Durch eine genaue Analyse der Laufzeiten in der Testgruppe *DIO Tests* lassen sich Varianzen

Testfall	Testfall Beschreibung
1	Verifiziert die Terminierung der Initialisierungsroutine *Pwm_Init*
2	Verifiziert ob die Funktion *Pwm_SetOutputToIdle* den PWM Kanal 0 in den Modulstatus *IDLE* versetzt
3	Verifiziert ob die Funktion *Pwm_SetOutputToIdle* den PWM Kanal 2 in den Modulstatus *IDLE* versetzt
4	Verifiziert ob die Funktion *Pwm_SetDutyCycle* den Duty Cycle des PWM Signals am Kanal 0 auf 0% setzt
5	Verifiziert ob die Funktion *Pwm_SetDutyCycle* den Duty Cycle des PWM Signals am Kanal 0 auf 100% setzt
6	Verifiziert ob die Funktion *Pwm_SetDutyCycle* den Duty Cycle des PWM Signals am Kanal 0 auf 50% setzt
7	Verifiziert ob die Funktion *Pwm_SetDutyCycle* den Duty Cycle des PWM Signals am Kanal 2 auf 0% setzt
8	Verifiziert ob die Funktion *Pwm_SetDutyCycle* den Duty Cycle des PWM Signals am Kanal 2 auf 100% setzt
9	Verifiziert ob die Funktion *Pwm_SetDutyCycle* den Duty Cycle des PWM Signals am Kanal 2 auf 50% setzt
10	Verifiziert ob die Funktion *Pwm_SetPeriod* die Frequenz des PWM Signals am Kanal 0 auf 20 kHz setzt
11	Verifiziert ob die Funktion *Pwm_SetPeriod* die Frequenz des PWM Signals am Kanal 0 auf 10 kHz setzt
12	Verifiziert ob die Funktion *Pwm_SetPeriod* die Frequenz des PWM Signals am Kanal 2 auf 20 kHz setzt
13	Verifiziert ob die Funktion *Pwm_SetPeriod* die Frequenz des PWM Signals am Kanal 2 auf 10 kHz setzt

Tabelle 7.4: Beschreibung der einzelnen Testfälle PWM Komponente

in den unterschiedlichen Testmethoden bestimmen. Die Testgruppe *DIO Tests* eignet sich für eine solche Analyse, da sämtliche manuellen und automatisierten Tests die gleiche Anzahl an auszuführenden Testschritten pro Testfall haben. Somit sollte jeder Test die gleiche Laufzeit beanspruchen. Abbildung 7.1 demonstriert die einzelnen Laufzeiten. Dabei lassen sich folgende Analysen durchführen:

- Der arithmetische Mittelwert der Laufzeit von manuellen Tests wird folgend bestimmt:

$$\bar{t}_{\text{manuell}} = \frac{\sum \text{Testlaufzeit}}{\text{Anzahl Tests}} = 56.375s \tag{7.5}$$

- Der arithmetische Mittelwert der Laufzeit von automatisierten Tests wird folgend bestimmt:

$$\bar{t}_{\text{automatisiert}} = \frac{\sum \text{Testlaufzeit}}{\text{Anzahl Tests}} = 3,229s \tag{7.6}$$

- Die Standardabweichung der Laufzeit von manuellen Tests wird folgend bestimmt:

$$s_{\text{m}} = \sqrt{\frac{\sum(\text{Testlaufzeit} - \bar{t}_{\text{manuell}})^2}{\text{Anzahl Tests}}} = 13,331s \tag{7.7}$$

Testfall	Testfall Beschreibung
1	Verifiziert die Terminierung der Initialisierungsroutine *Spi_Init*
2	Verifiziert mittels der Funktion *Spi_GetSequenceResult* ob der SPI Kanal 0 in den Modulstatus *OK* nach der Initialisierungsroutine versetzt wurde
3	Verifiziert mittels der Funktion *Spi_GetSequenceResult* ob der SPI Kanal 2 in den Modulstatus *OK* nach der Initialisierungsroutine versetzt wurde
4	Verifiziert ob die Funktion *Spi_WriteIB* erfolgreich Testdaten (0x11) in den Input Buffer des SPI Kanals 0 geschrieben hat
5	Verifiziert ob die Funktion *Spi_WriteIB* erfolgreich Testdaten (0x11) in den Input Buffer des SPI Kanals 2 geschrieben hat
6	Verifiziert ob die Funktion *Spi_AsyncTransmit* eine Übertragung der Testdaten (0x11) am SPI Kanal 0 auslöst
7	Verifiziert ob die Funktion *Spi_AsyncTransmit* eine Übertragung der Testdaten (0x22) am SPI Kanal 0 auslöst
8	Verifiziert ob die Funktion *Spi_AsyncTransmit* eine Übertragung der Testdaten (0x11) am SPI Kanal 2 auslöst
9	Verifiziert ob die Funktion *Spi_AsyncTransmit* eine Übertragung der Testdaten (0x22) am SPI Kanal 2 auslöst
10	Verifiziert ob die Funktion *Spi_ReadIB* die die empfangenen Testdaten (0x33) bei der Übertragung durch die Funktion *Spi_AsyncTransmit* aus den Input Buffer des SPI Kanals 0 ausließt
11	Verifiziert ob die Funktion *Spi_ReadIB* die die empfangenen Testdaten (0x44) bei der Übertragung durch die Funktion *Spi_AsyncTransmit* aus den Input Buffer des SPI Kanals 0 ausließt
12	Verifiziert ob die Funktion *Spi_ReadIB* die die empfangenen Testdaten (0x33) bei der Übertragung durch die Funktion *Spi_AsyncTransmit* aus den Input Buffer des SPI Kanals 2 ausließt
13	Verifiziert ob die Funktion *Spi_ReadIB* die die empfangenen Testdaten (0x44) bei der Übertragung durch die Funktion *Spi_AsyncTransmit* aus den Input Buffer des SPI Kanals 2 ausließt

Tabelle 7.5: Beschreibung der einzelnen Testfälle SPI Komponente

- Die Standardabweichung der Laufzeit von automatisierten Tests wird folgend bestimmt:

$$s_{\mathrm{a}} = \sqrt{\frac{\sum(\text{Testlaufzeit} - \bar{t}_{\text{automatisiert}})^2}{\text{Anzahl Tests}}} = 0,119s \tag{7.8}$$

- Die Standardabweichung der manuellen Tests ist um 99,107 % höher als bei automatisierten Tests.

$$(1 - \frac{s_{\mathrm{a}}}{s_{\mathrm{m}}}) * 100\% = 99,107\% \tag{7.9}$$

Die Tatsache, dass die Varianz in der Laufzeit um ein vielfaches höher bei manuellen Tests als bei automatisierten Tests ist, ist ein Indiz dafür, dass manuelle Tests bei der Durchführung fehleranfälliger sind als automatisierte Tests. Durch die längere Laufzeit ist auch die Zeit für Fehler in der Handhabung des Testaufbaus sehr hoch. Die Nachvollziehbarkeit von Fehlern ist bei längeren Laufzeiten ebenfalls schwieriger, da diese nicht auf kleine Zeitintervalle eingegrenzt werden können.
In Abbildung 7.1 ist ersichtlich, dass die Ausführungszeit der manuellen Tests bei den ersten höher ist und dann abnimmt. Dies ist dadurch zu erklären, dass die TesterIn pro Testlauf vertrauter mit der Testumgebung wird. Die Ausführung der Tests gelingt schneller. Die Varianz in der Ausführung der automatisierten Tests lässt sich durch die Verarbeitungszeit der physikalischen Signale am HIL Testsystem im *CANoe* Projekt erklären. Signale und Umgebungsvariablen werden mit einer Granularität von 1 ms berechnet.
Obwohl die Evaluierung durch Verifikationen von LLD Software Komponenten erfolgt ist, wären die Ergebnisse der Laufzeitmessung der funktionalen Tests von Hardware/Software Komponenten, wie den Buck Converter im Fallbeispiel 4.2, gleich. Die Ausführung der einzelnen Testschritte wurde in sämtlichen Verifikationprozessen der Abbildung 4.1 dieselbe Zeit beanspruchen. Lediglich der Testaufbau würde bei Hardware/Software Integration Tests mit einer dedizierten externen Hardware Komponente, wie einem Buck Converter IC, umfangreicher sein. Somit gelten die Erkenntnisse der erhöhten Effizienz von automatisierten Hardware/Software Integration Tests gegenüber manuellen auch für Software Integration Tests auf der Zielumgebung, sowie für System Integration Tests im Allgemeinen.

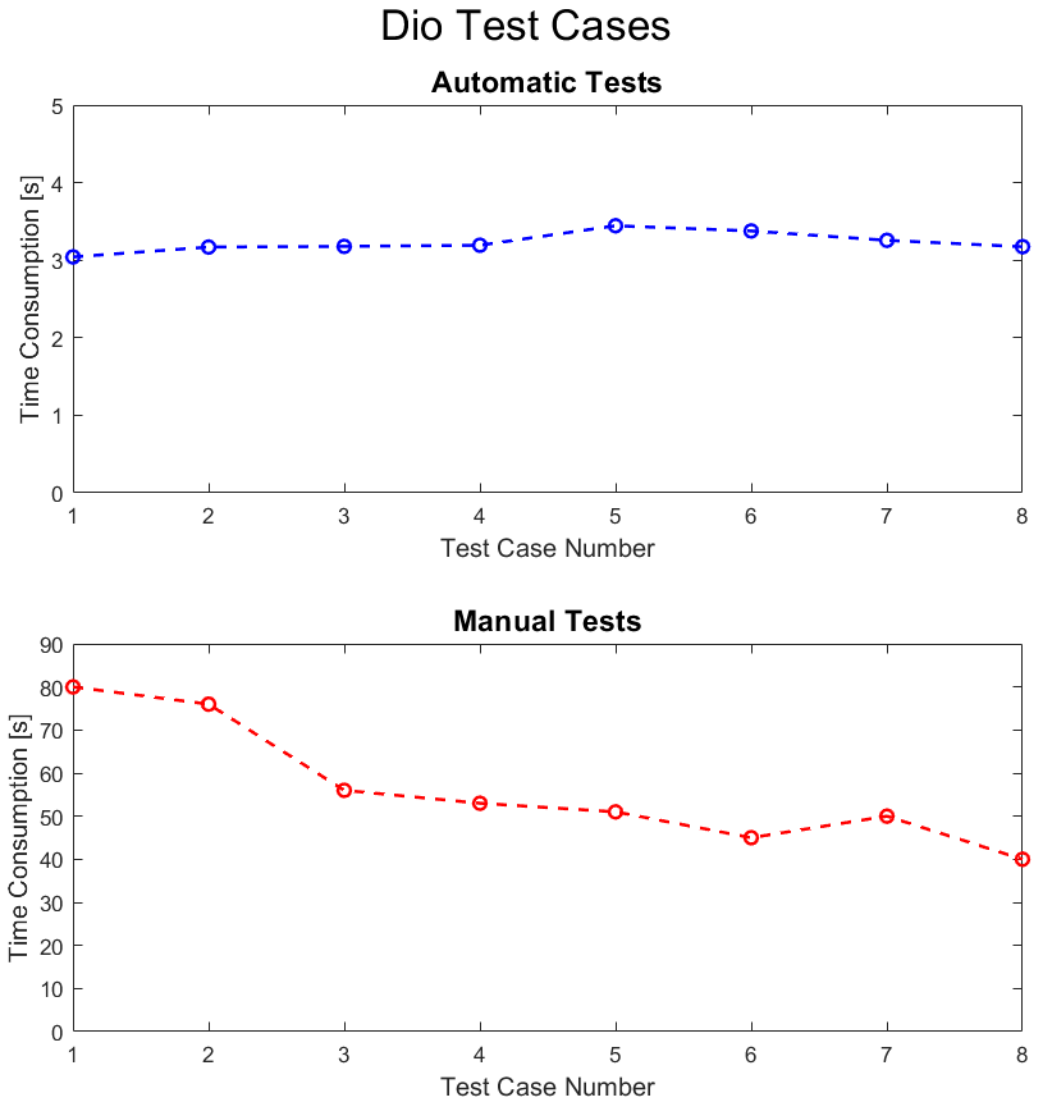

Abbildung 7.1: Laufzeitmessungen manueller und automatisierter DIO Tests

8 Diskussion der Ergebnisse

In diesem Kapitel wird die Umsetzung der Hardware/Software Integration Testplattform aus Kapitel 5, sowie deren Evaluierung aus Kapitel 7 diskutiert und analysiert. Der Neuheitswert der Plattform sowie deren Vorteile und Limitierungen werden aufgezeigt.

8.1 Nutzen der Hardware/Software Integration Testplattform

Die Umsetzung von manuellen Hardware/Software Integration Tests mittels händischen Messungen mit Messgeräten, wie Oszilloskope, bringt zwar rasche Ergebnisse, ohne große Vorbereitung des Messaufbaus. Die funktionale Korrektheit einer Implementierung sollte allerdings bloß als entwicklungsbegleitende Ergänzung im Hardware/Software Integration Prozess gesehen werden. Sich gänzlich auf deren Korrektheit zu verlassen birgt viele Probleme. So sind manuelle Tests äußerst fehleranfällig und nicht unter den gleichen Bedingungen wiederholt reproduzierbar. Die Veränderung von unbewussten Rahmenbedingungen kann unterschiedliche Ergebnisse erzeugen, welche nicht nachvollziehbar sind. Daher sollten bei der Erstellung der Requirements des zu entwickelnden Systems genaue Rahmenbedingungen für Hardware/Software Integration Tests definiert werden. Dabei sollten die Rahmenbedingungen sämtliche Grenzen der Funktionalität der zu verifizierenden System Komponenten ausloten können. Beispielsweise kann es für eine Komponente, welche eine Temperaturmessung mittels eines NTC Widerstandes vollzieht, notwendig sein, die Umgebungstemperatur zu variieren. Um die Funktionalität dieser System Komponenten unabhängig von anderen System Komponenten qualifizieren zu können, ist es notwendig, eine großflächige Erhebung der Daten durchzuführen und diese auszuwerten. Neben der hohen Fehleranfälligkeit von händischen Hardware/Software Integration Tests ist auch der zeitliche und damit auch finanzielle Aufwand, welcher für eine großflächige Erhebung und Auswertung von Daten notwendig wäre, unverhältnismäßig hoch. Durch die Automatisierung des Hardware/Software Integration Prozessschrittes können schnell und kostengünstig Reports für die unabhängige Qualifizierung von System Komponenten erstellt werden, womit deren funktionale Richtigkeit verifiziert werden kann.
Die Verifikation von unabhängigen Systemkomponenten in Black Box Tests des System Qualifizierungs Prozesses gibt zwar Aufschluss darüber ob, das gesamte System den funktionalen Requirements entspricht, allerdings ist die Identifikation einer fehlerhaften Komponente im System Qualifizierungs Prozess nicht vorgesehen. Dieser erstellt einen Report über die Funktionalität des gesamten Systems unabhängig von dessen Architektur. Für die Zuweisung auf eine fehlerhafte Komponente müssen in so einem Fall wiederum händische Tests durchgeführt werden. Daher

werden System Qualifizierungs Tests nicht als hinreichender Ersatz für automatisierte System Integration Tests verstanden.
Im Konzept des Test-Driven Developement werden funktionale Tests in der Software Entwicklung vor der eigentlichen Implementierung durchgeführt [76]. Fehler bei der Implementierung einer Funktion werden dadurch reduziert, da man sich im Zuge der Test Umsetzung bereits mit möglichen Fehlern in der Implementierung beschäftigt. Dieses Konzept lässt sich in vielen entwicklungsbegleitenden Prozessen des V-Modells anwenden, so auch im System Integration Prozess.
Wie bereits in diesem Abschnitt erläutert erzeugen automatisierte Hardware/Software Integration Tests als Teil des System Integration Prozesses eine Optimierung in der Entwicklung von Embedded System nach dem ASPICE Modell [2]. Doch auch für die Entwicklung von Software Treibern für elektronische Bauteile bieten Hardware/Software Integration Tests eine unabhängige Qualifizierung bezüglich ihrer funktionalen Richtigkeit. Durch die Verwendung von Bauteil und Mikrocontroller Evaluation Boards wird hier auch eine Unahängigkeit vom Entwicklungsstatus des zu entwickelnden Systems geschaffen. Durch diese strategische Entwicklung einer System Komponente können die Entwicklungskosten reduziert werden [3].
Die entwickelte Hardware/Software Integration Testplattform mit der *Hardware/Software Integration Verification* DLL ist Teil einer strategischen Optimierung im Entwicklungsprozess, um Entwicklungskosten im Prozessschritt der System Integration zu sparen. Höhere Aufwände bei der Erstellung eines automatisierten Testsystems, wie in Kapitel 6 beschrieben, amortisieren sich durch durch die starke Reduktion in der Testlaufzeit, wie in Kapitel 7 beschrieben. Hinzu kommt, dass die Erstellung eines automatisierten Testaufbaus bloß einmal durchgeführt werden muss, während dies bei manuellen Tests vor jeder Durchführung geschieht. Auch kann sowohl der Testaufbau, als auch die entwickelten Testfälle für die Verifikation anderen Komponenten in einen anderen Kontext verwendet werden, was die Wiederverwendbarkeit der automatisierten Tests steigert. Manuelle Tests können im Allgemeinen nicht wiederverwendet werden, da diese vor jeder Durchführung erneut aufgebaut werden müssen.
Durch den Einsatz der Hardware/Software Integration Plattform ist des Weiteren gewährleistet, dass der System Integration Test bei der Entwicklung eines Systems mit einem Safeyt Level entsprechend der ISO 26262 [10] eingestuft wird, welcher die explizite Durchführung von Integrationstests auf der Zielumgebung verlangt.

8.2 Limitierungen der Hardware/Software Integration Testplattform und Betrachtung ausgewählter kommerzieller Lösungen

Für die Evaluierung des Konzeptes zur Umsetzung von Hardware/Software Integration Tests mit Tests auf der Entwicklungsumgebung wurde ein experimenteller Prototypenaufbau geschaffen. Für diesen Aufbau wurde die dementsprechende Hardware eingekauft, sowohl für den Aufbau eines HIL Testsystems als auch ein Debugger. Dabei fiel die Entscheidung auf den Erwerb eines Messsystems der Firma *Vector Informatic*. Die Funktionalität und damit auch deren Grenzen sind bereits im Kapitel 5.1.2 erläutert. Zusätzlich wurden für die Evaluierung eines möglichen HIL Testsystems folgende Anbieter analysiert:

- *iSystem*: Mittels des *IOM6 ADIO* [77] ist die Überwachung von analogen und digitalen Signalen gewährleistet. Des Weiteren ist die Überwachung von SPI Signalen angeboten. Die Verwendung des Gerätes als HIL Plattform bietet sich für die Behandlung der beschriebenen Signale an. Auch besteht hier keine Limitierung im Hinblick auf den Einsatz des

Framework *isystem.connect* SDK. Komplexere Messungen, wie für PWM Duty Cycle oder Frequenz Messungen, werden mittels eines dementsprechenden Treibers nicht angeboten. Auch weitere herkömmliche serielle Schnittstellen sind nicht unterstützt. Die Implementierung solcher Treiber wären für diese Anwendungen notwendig. Diese müssten wiederum verifiziert werden. Basierend auf den recht großen Aufwand wurde dieser Anbieter für die Umsetzung eines möglichst generischen und umfangreichen HIL Testsystems für unzureichend erklärt.

- *Totalphase*[1]*:* Mittels der Produkte dieser Firma können DIO Signale bearbeitet werden, sowie Kommunikationschnittstellen wie I2C [78] und SPI [79]. Des Weiteren werden auch Bibliotheken in verschiedensten Sprachen, wie Python und C#, zur Verfügung gestellt, welche für automatisierte Tests verwendet werden können. Allerdings sind Instrumentierungen und Messungen von physikalischen analogen Signalen und PWM Signalen nicht unterstützt. Die Spezialisierung liegt ausschließlich auf Kommunikationsschnittstellen und ist damit nicht ausreichend für die Umsetzung eines möglichst generischen und umfangreichen HIL Testsystems.

Nachteile der weiteren beschriebenen Anbieter konnten bei HIL Testsystem der Firma *Vector Informatic* nicht festgestellt werden. Bei der Evaluierung der SPI Software Komponente werden die Signale am Modul des *VT Systems* mittels Umgebungsvariablen manipuliert und ausgewertet. In einem *CANoe* Projekt auf einem Host Rechner ist dies mit einer maximalen Geschwindigkeit von 1 ms möglich. Bei Sensor Protokollen mit harten Echtzeitanforderungen, kann die Einhaltung der Deadlines nicht garantiert werden. Abhilfe schafft hier das Real-Time Modul des *VT Systems*, welches ein *CANoe* Projekt am HIL Testsystem simuliert und dadurch schneller auf Events reagieren kann. Doch bei der Umsetzung von Hardware/Software Integration Tests mittels der *Hardware/Software Integration Verifiaction* DLL ist eine Ausführung am Host Rechner nötig, da der Zugriff auf den Dubugger über eine USB Schnittstelle erfolgt. Diese wird vom Real-Time Modul nicht zur Verfügung gestellt. Alternative kann eine Auswertung der Umgebungsvariablen, unter Einhaltung der Echtzeit Anforderungen, auf einem on-board FPGA des *VT System* Moduls für die Auswertung von Sensor Protokollen durchgeführt werden [80]. Dies wurde alledings im Zuge dieser Arbeit nicht untersucht.
Bei der Auswahl des Debuggers wurde ein Debugger [53] der Firma *iSystem* gewählt, da dieser eine DLL für C# zur Verfügung stellt, welcher in einem Testknoten in einem *CANoe* Projekt zum zugehörigen *Vector Informatic* HIL Testsystem verwendbar ist. Dementsprechend können nur Mikrocontroller verwendet werden, welche seitens des Debuggers unterstützt werden.
Die im Zuge dieser Arbeit entwickelte DLL basiert auf Methoden, welche der EABI [57] einer 32-Bit *PowerPC* und *RH850* Prozessor Architektur entsprechen. Dementsprechend können auch nur Mikorcontroller mit diesen Prozessor Architekturen verwendet werden. Die *Hardware/Software Integration Verification* DLL muss manuell erweitert werden, um die Register anderer Prozessor Architekturen abstrahieren zu können. Angemerkt sei allerdings, dass die Prozessorarchitekturen *PowerPC* und *RH850* weit verbreitet sind und damit auf vielen gängigen Mikrocontroller Plattformen zu finden sind.
Um in Echtzeit auf den Speicher und auf die Register des Mikrocontrollers zugreifen zu können, muss der JTAG IP-Core des Mikrocontrollers einen TAP Controller nach dem Standard IEEE-ISTO 5001^{TM}-2003 (NEXUS) [34] entsprechen.
Auch Limitierung der *isystem.connect* SDK werden in die entwickelte *Hardware/Software Integration Verification* DLL weiter vererbt. So ist es nicht möglich Werte von komplexen Datentypen,

[1] `https://www.totalphase.com/` Online: Letzter Zugriff am 06.02.2020

wie Strukturen in C, in die Register des Prozessors zu laden. Dementsprechend können auch keine Funktionsaufrufe, dessen Parameter Variablen mit komplexen Datentypen beinhalten, über die DLL auf der Software der Zielumgebung getriggert werden. Alternative können Speicherbereiche reserviert werden, welche die Daten einer Struktur abbilden. Diese können dann als Pointer, welcher auf den Speicherbereich verweist, einer Funktion übergeben werden. Diese Funktion muss allerdings einen Pointer als Übergabeparameter deklariert haben.

9 Fazit und Ausblick

Bezugnehmend auf die Prozesse des ASPICE V-Modells wurde deren Bewertung im Sinne des zugehörigen Reifegradmodells beschrieben. Dies war notwendig, um die Aufwände als Teil einer strategischen Prozessentwicklung zu rechtfertigen. Prozesse, welche den Kontext der Hardware/-Software Integration streifen, wurden im Detail erläutert. So wird der Software Modul Verifikation Prozess im Zuge von LLD Unit Verifikationen auf der Mikrocontrollerzielumgebung interpretiert. Weiterführend lassen sich gleichartige Methoden im Software Integration Prozess anwenden. Im besonderen Fokus wurde der System Integration Prozess beleuchtet, in welchem die Integration der Software auf der Zielumgebung, sowie die Integration von zugehörigen Hardware Komponenten als eigene Prozessschritte, beschrieben wurden. Der System Integration Prozess in der Entwicklung von Steuergeräten in der Automobilindustrie verlangt hier die Durchführung dieser Prozessschritte zur Gewährleistung des Safety Standard ASIL Level B der ISO 26262.
Verschiedene Methoden zur Durchführung von funktionaler Verifikationen von Software Komponenten auf einer Mikrocontroller Zielumgebung, sowie von Verifikation mit der zugehörigen Hardware Periphärie, wurden ermittelt. Im Speziellen wurde die Integration von PIL Tests als eigene Testfälle, ergänzend zum produktiven Code der Software Komponenten auf der Mikrocontroller Zielumgebung, als gängiges Konzept ermittelt. Auch die Einbettung der PIL Tests auf einer Zielumgebung mit einem HIL Testsystem zur Interaktion mit der umgebenden Peripherie des Mikrocontrollers wurde als Methode zur Durchführung von funktionalen Hardware/Software Integration Tests erläutert. Nachteile dieses gängigen Konzepts wurden beschrieben und ein alternatives Konzept ermittelt. So werden alternativ die Tests nicht als PIL Tests auf der Zielumgebung ausgeführt, sondern auf einer Entwicklungsumgebung. Die echtzeitfähige Kommunikation mit dem HIL Testsystem und mit dem Mikrocontroller über eine Debugging Schnittstelle, entsprechend einer echtzeitfähigen Kategorie des IEEE-ISTO 5000^{TM}-2003 (NEXUS), wurde als technologischer Hintergrund für die Umsetzung des entwickelten Konzepts festgestellt.
Die Umsetzung des gewählten Konzepts ermöglicht die funktionale Verifikation von LLD Software Komponenten auf der Mikrocontroller Zielumgebung, weiterführende Integration von Software Komponenten der Software Architektur auf der Zielumgebung sowie eine weiterführende Integration von Hardware Komponenten der System Architektur. Diese weiterführenden Integrationsschritte sind im gewählten Konzept automatisiert verifizierbar. Die Verwendung von Hardware/Software Integration Tests auf diesen unterschiedlichen Verifikationsebenen wurde anhand von Beispielen erläutert. Dabei wurde im Zuge der Evaluierung der Arbeit das Beispiel zur Verifikation von LLD Software Komponenten auf einer Mikrocontroller Zielumgebung erprobt und bestätigt. Die Ergebnisse der Evaluierung zeigen, dass das entwickelte Konzept auch in den Fällen der Bauteil Evaluierung und System Integration anwendbar ist.
Für die Implementierung einer Hardware/Software Integration Testplattform mit Tests auf der

Entwicklungsumgebung wurden einerseits sämtliche Anforderungen erhoben, andererseits auch die Hardware für dessen Umsetzung erworben und im Detail beschrieben. In Abstimmung zu den Schnittstellen der erworbenen Hardware wurde ein exemplarischer Testaufbau um ein Mikrocontroller Entwicklungsboard erstellt. Für die Interaktion und Kommunikation mit dem Mikrocontroller über einen Debugger wurde eine Klassenbibliothek entwickelt, welche in einem Projekt zu einem Tool des HIL Testsystems integrierbar ist. Somit können automatisierte Hardware/Software Integration Tests in einem solchen Projekt ausgeführt und der Report für die Qualifizierung von System Komponenten verwendet werden. Die Architektur der entwickelten Klassenbibliothek sowie deren beinhaltete Klassen selbst wurden im Detail beschrieben. Die Verwendung der entwickelten Hardware/Software Integration Plattform mit der beinhalteten *Hardware/Software Integration Verification* DLL wurde exemplarisch beschrieben.
Die entwickelte Hardware/Software Integration Testplattform wurde für die Verifikation von LLD Software Modulen verwendet. Diese Module sind durch PIL Tests nicht auf der Zielumgebung verifizierbar, da eine Messung und Instrumentierung von physikalischen Signalen an den Mikrocontroller Pins notwendig ist. Ein HIL Testsystem übernimmt die Behandlung der physikalischen Signale in der entwickelten Hardware/Software Integration Testplattform. Die integrierten LLD Software Module konnten im Zuge der Evaluierung unabhängig von anderen Software oder System Komponenten funktional verifiziert werden und entsprechen den formulierten Requirements für die jeweilige Software Komponente. Erwähnt sei hier, dass sämtliche LLD Software Module entsprechend der Hardware Treiber am Mikrocontroller SPC560B54L5 [49] umgesetzt sind. Mikrocontroller, dessen Hardware Treiber auf anderen IP-Cores basieren, können diese LLD Software Module nicht verwenden. Allerdings haben die getesteten LLD Software Module Interface Funktionen entsprechend ihrer jeweiligen AUTOSAR Spezifikation. Das bedeutet, dass die entwickelten Hardware/Software Integration Tests auf sämtlichen LLD Software Modulen mit der gleichen Funktionalität angewendet werden können, sofern diese ebenso der generischen AUTOSAR Interface Spezifikation entsprechen. Dementsprechend können die Tests auch für andere LLD Software Module auf anderen Zielumgebungen wiederverwendet werden.
Im Zuge der Evaluierung der entwickelten Hardware/Software Integration Testplattform wurde die Ausführung von Tests der Verifikation von LLD Software Modulen bewertet. Dabei wurden Unterschiede in der Laufzeit zwischen manuellen und automatisierten Tests analysiert. Die höhere Zeiteffizienz bei der Durchführung von automatisierten Hardware/Software Integration Tests gegenüber manuellen wurde festgestellt. Die Varianz in der Ausführungszeit lässt dabei Schlüsse auf die Fehleranfälligkeit der jeweiligen Methode schließen.
Die Verwendung der Hardware/Software Integration Testplattform beschränkt sich allerdings nicht bloß auf die Verifikation von LLD Software Modulen auf der Zielumgebung. Weiterführende Software Integrationstests von Komponenten einer höheren Ebene der Software Architektur können auf der Zielumgebung mit Messung und Instrumentierung von physikalischen Signalen am Mikrocontroller Pin durchgeführt werden. Somit kann die Plattform für Tests im Software Integration Prozess ebenso verwendet werden, wie als Plattform für Tests auf der Zielumgebung im Software Unit Verifikation Prozess. Wiederum weiterführend können auch Hardware und zugehörige Software Komponenten im Zuge des System Integration Prozesses verifiziert werden. Dabei erfolgt die Messung und Instrumentierung der physikalischen Signale bei den elektronischen Bauelementen der Hardware Komponente oder am Stecker des entwickelten Systems. In sämtlichen anzuwendenden Prozessen werden Reports mit den Ergebnissen der Testläufe erstellt, welche bezogen auf die formulierten Testfälle Requirements der Software und System Komponenten verifizieren können.
Auch die Erweiterung der Hardware/Software Integration Testplattform mit der zugehörigen *Hardware/Software Integration Verifiaction* DLL für eine weitere Optimierung der ASPICE Entwicklungsprozesse wäre denkbar. So kann die Durchführung von automatisierten Fault Injection

Tests [28] auf einer Zielumgebung Teil der Hardware/Software Integration Testplattform sein. Im Safety Standard ISO 26262 [10] wird die Durchführung von Fault Injection Tests auf der Zielumgebung für die Einstufung in ASIL Level D verlangt.
Durch den Einsatz von automatisierten Hardware/Software Integration Tests können gängige Methoden, wie die fehleranfällige manuelle Verifikation durch händische Messungen im System Integration Prozess, ersetzt werden. Dabei kann diese Art der Verifikation nicht bloß für funktionale Tests von LLD Software Komponenten auf Mikrocontroller Zielumgebungen, wie in der Evaluierung in Kapitel 7 erprobt, automatisiert werden. Bei der Integration von elektronischen Bauteilen mit der zugehörigen Entwicklung eines Software Treibers kann dasselbe Konzept ebenso automatisiert angewendet werden. Lediglich das Test-Setup muss um die zusätzlichen Hardware Komponenten ergänzt werden. Somit lässt sich die entwickelte Hardware/Software Integration Plattform in sämtlichen Beispielen, welche in Kapitel 4.1 beschrieben sind, anwenden. Darüber hinaus führt die Automatisierung von Hardware/Software Integration Tests zu einer optimierten Umsetzung des System Integration Entwicklungsprozesses, wodurch strategisch Kosten gespart werden können, eine höhere Kategorie im Reifegradmodell bei der Bewertung der ASPICE Prozesse erreicht und Systeme mit einem höheren Safety Standard entwickelt werden soll.

Literaturverzeichnis

[1] Texas Instruments. Led drivers for automotive exterior lighting applications. `http://www.ti.com/lit/sl/slpy006b/slpy006b.pdf`, Online: Letzter Zugriff am 11.03.2020, 2018.

[2] *Automotive SPICE Guidelines*. Verband der Automobilindustrie e.V.(VDA), Berlin Deutschland, 2017.

[3] John Ousterhout. *A Philosophy of Software Design*. 1st edition, 2018.

[4] S. Grünfelder. *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*. dpunkt.verlag, 2017.

[5] Stefan Krauß. Testing with canoe. `https://pdfs.semanticscholar.org/4aed/1aa019d0888c9a9ec92d35c81f310d95e0f0.pdf`, 10 2009.

[6] Erol Simsek Ales Kosir. Automation using python - an introduction to the isystem.connect api, 03 2020.

[7] AUTOSAR Confidential. Specification of dio driver. `https://www.autosar.org/`, 11 2016.

[8] Vector Informatik GmbH. User manuel vt system. `https://assets.vector.com/cms/content/products/vtsystem/Docs/VTSystem_Manual_EN.pdf`, 2018. Online: Letzter Zugriff am 21.08.2019.

[9] M. Müller, K. Hörmann, L. Dittmann, and J. Zimmer. *Automotive SPICE® in der Praxis: Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag, 2016.

[10] V. Gebhardt, G.M. Rieger, J. Mottok, and C. Gießelbach. *Funktionale Sicherheit nach ISO 26262: ein Praxisleitfaden zur Umsetzung*. dpunkt-Verlag, 2013.

[11] Qi Van Eikema Hommes. Review and assessment of the iso 26262 draft road vehicle - functional safety. In *SAE Technical Paper*. SAE International, 04 2012.

[12] A. Marrero Perez and S. Kaiser. Integrating test levels for embedded systems. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, pages 184–193, Sep. 2009.

[13] T. Linz and A. Spillner. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. iSQI-Reihe. dpunkt.verlag, 2012.

[14] M. Shedeed, G. Bahig, M. W. Elkharashi, and M. Chen. Functional design and verification of automotive embedded software: An integrated system verification flow. In *2013 Saudi International Electronics, Communications and Photonics Conference*, pages 1–5, 2013.

[15] V. Garousi, M. Felderer, Ç. M. Karapiçak, and U. Yilmaz. What we know about testing embedded software. *IEEE Software*, 35(4):62–69, July 2018.

[16] Dinesh Kumar Saini. Software testing for embedded systems. April 2012.

[17] R. Bär, A. Behr, and D. Fischer. Code-coverage auf embedded-systemen.

[18] Jan Peleska. @! hardware/software integration testing for the new airbus aircraft families. *http://www.informatik.uni-bremen.de/agbs/jp/papers/peleskaTestCom2002.html*, 01 2002.

[19] J. Schroeder, C. Berger, and T. Herpel. Challenges from integration testing using interconnected hardware-in-the-loop test rigs at an automotive oem – an industrial experience report. In *2015 First International Workshop on Automotive Software Architecture (WASA)*, pages 39–42, 2015.
[20] R. Osherove. *The Art of Unit Testing: 2. Auflage, deutsch.* Mitp Professional. Mitp, 2015.
[21] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, October 2004.
[22] Ieee standard for system, software, and hardware verification and validation. *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017)*, pages 1–260, Sep. 2017.
[23] J. Sini, A. Mugoni, M. Violante, A. Quario, C. Argiri, and F. Fusetti. An automatic approach to integration testing for critical automotive software. In *2018 13th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, pages 1–2, 2018.
[24] Shaukat Ali and Tao Yue. Formalizing the iso/iec/ieee 29119 software testing standard. In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *MoDELS*, pages 396–405. IEEE, 2015.
[25] M. Winter, M. Ekssir-Monfared, H.M. Sneed, R. Seidl, and L. Borner. *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration.* Hanser, 2013.
[26] L. Gomes. *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation: Applications for Design and Implementation.* IGI Global research collection. Information Science Reference, 2009.
[27] Erol Simsek Ales Kosir. Continuous integration - jenkins isystem tools, 04 2020.
[28] Ludovic Pintard, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. *Fault Injection in the Automotive Standard ISO 26262: An Initial Approach*, volume 7869, pages 126–133. 01 2013.
[29] N. Wiersma and R. Pareja. Safety != security: On the resilience of asil-d certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16, Sep. 2017.
[30] S. Letsche. *Entwicklung von Produkt- und Systemtests für Energiespeichersysteme auf einem Hardware-in-the-Loop (HiL) System.* Diplom.de, 2011.
[31] Luiz S. Martins-Filho, Adrielle C. Santana, Ricardo O. Duarte, and Gilberto Arantes Junior. Processor-in-the-loop simulations applied to the design and evaluation of a satellite attitude control. In Jan Awrejcewicz, editor, *Computational and Numerical Simulations*, chapter 8. IntechOpen, Rijeka, 2014.
[32] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 134–143, Dec 1998.
[33] Felix Martin Sebastian Ziegler. Efficient run-time analysis of autosar classic projects with os and rte, 11 2019.
[34] Inc. Ashling Microsystems. The nexus debug standard: Gateway to the embedded systems of the future. 12 2003.
[35] Erol Simsek Felix Martin. Introduction to tracing - how to utilize the hardware trace capabilities of your microcontroller in winidea, 05 2020.
[36] Susanne Kandl and Martin Elshuber. A formal approach to system integration testing. *CoRR*, abs/1404.6743, 2014.
[37] M. K. Ludwich and A. A. Fröhlich. System-level verification of embedded operating systems components. In *2012 Brazilian Symposium on Computing System Engineering*, pages 161–165, 2012.

[38] Erol Simsek Ales Kosir. Digital/analog/spi signals and isystem tools - adio how to and use cases, 04 2020.

[39] J.W. Grenning. *Test-driven Development for Embedded C*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2011.

[40] T. Fischer. *Eine Technologie fuer das durchgaengige und automatisierte Testen eingebetteter Software*. Karlsruher Institut für Technologie, 2017.

[41] Zhenhua Jiang, Rodrigo Leonard, R Dougal, H Figueroa, and A Monti. Processor-in-the-loop simulation, real-time hardware-in-the-loop testing, and hardware validation of a digitally-controlled, fuel-cell powered battery-charging station. volume 3, pages 2251 – 2257 Vol.3, 07 2004.

[42] J. Mina, Z. Flores, E. López, A. Pérez, and J. . Calleja. Processor-in-the-loop and hardware-in-the-loop simulation of electric systems based in fpga. In *2016 13th International Conference on Power Electronics (CIEP)*, pages 172–177, 2016.

[43] H. Proff. *Mobilität in Zeiten der Veränderung: Technische und betriebswirtschaftliche Aspekte*. Springer Fachmedien Wiesbaden, 2019.

[44] Randy Johnson and Stewart Christie. Jtag 101 ieee 1149.x and software debug. 01 2009.

[45] B. Scherer and G. Horváth. Microcontroller tracing in hardware in the loop tests integrating trace port measurement capability into ni veristand. In *Proceedings of the 2014 15th International Carpathian Control Conference (ICCC)*, pages 522–526, 2014.

[46] Erol Simsek Felix Martin. Autosar classic timing analysis - hardware trace based real time performance analysis of autosar classic applications, 04 2020.

[47] U. Tietze, C. Schenk, and E. Gamm. *Halbleiter-Schaltungstechnik*. Springer-Verlag GmbH, 2012.

[48] Organización Internacional de Normalización. *ISO 26262: Road Vehicles : Functional Safety*. ISO, 2011.

[49] ST Microelectronics. Rm0037. `https://www.st.com/content/ccc/resource/technical/document/reference_manual/12/0e/dd/68/7a/da/4a/62/CD00234622.pdf/files/CD00234622.pdf/jcr:content/translations/en.CD00234622.pdf`, 2016. Online: Letzter Zugriff am 21.08.2019.

[50] ST Microelectronics. Um1672. `https://www.st.com/content/ccc/resource/technical/document/user_manual/21/18/0f/79/cf/bb/4b/6f/DM00094991.pdf/files/DM00094991.pdf/jcr:content/translations/en.DM00094991.pdf`, 2013. Online: Letzter Zugriff am 21.08.2019.

[51] Dudekula Rafi, Katam Moses, Kai Petersen, and Mika Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. pages 36–42, 06 2012.

[52] Vector Informatik GmbH. How to evaluate embedded software test tools. 12 2017.

[53] iSystem. ic5000 debugger on-chip analyzer. `https://www.isystem.com/products/hardware/on-chip-analyzers/ic5000.html`. Online: Letzter Zugriff am 06.02.2020.

[54] Vector Informatik GmbH. Vt system, smart hil testing. `https://ukintpress-conferences.com/conf/10txeu_conf/pdfs/day_1/225_tuesday_vector_albert.pdf`, 2010. Online: Letzter Zugriff am 21.08.2019.

[55] F. Gao and F. Deng. Design of a networked embedded software test platform based on software and hardware co-simulation. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 375–381, 2016.

[56] All Electronics. Mil bis hil systematisch testen. `https://www.all-electronics.de/wp-content/uploads/migrated/article-pdf/117666/ael09-02-020.pdf`, Online: Letzter Zugriff am 12.05.2020, 2009.

[57] Inc Freescale Semiconductor. Powerpc embedded application binary interface (eabi): 32-bit implementation, 2004. Online: Letzter Zugriff am 21.08.2019.
[58] Joyce Farrell. *Microsoft Visual C 2015: An Introduction to Object-Oriented Programming*. Course Technology Press, Boston, MA, USA, 6th edition, 2015.
[59] Inc Freescale Semiconductor. Powerpc^{TM} e500 application binary interface user's guide, 03 2003. Online: Letzter Zugriff am 21.08.2019.
[60] Cathy May, Ed Silha, Rick Simpson, Hank Warren, and CORPORATE International Business Machines, Inc. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.
[61] Renesas Electronics. Rh850g3mh user's manual: Software. `http://pdfstream.manualsonline.com/3/349d0eef-ccc9-4e2e-ae10-1569006cf00b.pdf`, 2015. Online: Letzter Zugriff am 11.03.2020.
[62] Ieee standard test access port and boundary scan architecture. *IEEE Std 1149.1-2001*, pages 1–212, July 2001.
[63] R. Oshana. *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. EngineeringPro collection. Elsevier Science, 2013.
[64] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt, Heidelberg, 2009.
[65] AUTOSAR Confidential. Layered software architecture. `https://www.autosar.org/`, 11 2016.
[66] AUTOSAR Confidential. General specification of basic software modules. `https://www.autosar.org/`, 11 2016.
[67] AUTOSAR Confidential. General requirements on basic software modules. `https://www.autosar.org/`, 11 2016.
[68] D. Lacamera. *Embedded Systems Architecture: Explore architectural concepts, pragmatic design patterns, and best practices to produce robust systems*. Packt Publishing, 2018.
[69] AUTOSAR Confidential. Requirements on dio driver. `https://www.autosar.org/`, 11 2016.
[70] AUTOSAR Confidential. Specification of adc driver. `https://www.autosar.org/`, 11 2016.
[71] AUTOSAR Confidential. Requirements on adc driver. `https://www.autosar.org/`, 11 2016.
[72] AUTOSAR Confidential. Specification of pwm driver. `https://www.autosar.org/`, 11 2016.
[73] AUTOSAR Confidential. Requirements on pwm driver. `https://www.autosar.org/`, 11 2016.
[74] AUTOSAR Confidential. Specification of spi driver. `https://www.autosar.org/`, 11 2016.
[75] AUTOSAR Confidential. Requirements on spi driver. `https://www.autosar.org/`, 11 2016.
[76] Dietmar Winkler, Stefan Biffl, and Thomas Östreicher. Test-driven automation: Adopting test-first development to improve automation systems engineering processes. 01 2009.
[77] iSystem. Iom6 adio user manuel. `https://www.isystem.com/products/hardware/iom-accessories-82/iom6-family/iom6-adio.html`. Online: Letzter Zugriff am 06.02.2020.
[78] Totalpahse. Aardvark i2c/spi host adapter. `https://www.totalphase.com/products/aardvark-i2cspi/`. Online: Letzter Zugriff am 06.02.2020.
[79] Totalpahse. Cheetah spi host adapter. `https://www.totalphase.com/products/cheetah-spi/`. Online: Letzter Zugriff am 06.02.2020.
[80] Vector Informatik GmbH. User manual, vt system fpga manager, user programmable fpga. `https://ukintpress-conferences.com/conf/10txeu_conf/pdfs/day_1/225_tuesday_vector_albert.pdf`, Online: Letzter Zugriff am 11.03.2020, 2017.