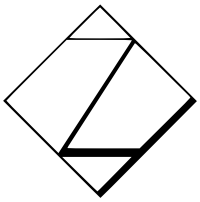

Westsächsische Hochschule Zwickau
University of Applied Sciences
Fakultät Elektrotechnik

DIPLOMARBEIT

Erstellung und Umsetzung einer generischen Ausführungsumgebung für Software-Modultests auf Embedded Hardware

zur Erlangung des

akademischen Grades

Diplomingenieur für Informationstechnik (FH)

eingereicht von

Micha Lehmann

geb. am 29. 03. 1986 in Erlabrunn

Firma:	IAV - Ingenieurgesellschaft Auto und Verkehr Kauffahrtei 45 09120 Chemnitz
Betreuer (Hochschule):	Prof. Dr.-Ing. Christian Troll
Betreuer (Betrieb):	Dipl.-Ing. Marko Meyer
Abgabedatum:	11. November 2013

Aufgabenstellung WHZ

Ziel der Arbeit ist die Entwicklung einer Ausführungsumgebung für Software-Modultests auf eingebetteten Systemen.

Die Ausführungsumgebung soll über eine Hardwareabstraktionsschicht (Hardware Abstraction Layer - HAL) auf verschiedene Systeme portierbar sein. Die Ausführungsumgebung soll zusammen mit der HAL und dem Softwaretest für das Zielsystem kompiliert werden können. Das Zielsystem kann auch ein PC sein. Die Steuerung der Ausführungsumgebung soll über eine Kommunikationsschnittstelle erfolgen. Über diese Schnittstelle soll es möglich sein, Daten für den Test nachzuladen, sowie den Testablauf zu steuern und die Ergebnisse abzuholen. Zur Steuerung kann eine PC-Software verwendet werden, die aber nicht Bestandteil dieser Diplomarbeit ist.

Es soll die Möglichkeit vorgesehen werden, die Codeabdeckung des Tests mit einer Software wie CTC++ überprüfen zu können.

Die Aufgabe umfasst das Erstellen der Softwarearchitektur und die Implementierung von Ausführungsumgebung und Anwendungsprotokoll. Zusätzlich soll eine Architektur für die Hardwareabstraktionsschicht entwickelt werden.

Die Funktion des Systems ist durch die erfolgreiche Ausführung eines Tests und Übertragung der Ergebnisse nachzuweisen. Als Zielsystem dient ein PC. Nachdem die Daten für den Test dynamisch über die Kommunikationsschnittstelle nachgeladen wurden, ist der Test mit den Methoden Setup, Test und Teardown auszuführen. Vor dem Teardown sind die Ergebnisse abzuholen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

CAN	Controller Area Network
CDT	C/C++ Development Tooling
CUTE	C++ Unit Testing Easier, Modultestframework mit Integration in Eclipse
DHCP	Dynamic Host Configuration Protocol, erlaubt das Verteilen der Netzwerkkonfiguration an Clients durch einen Server
DSP	Digitaler Signal Prozessor
FMECA	Failure Mode and Effects and Criticality Analysis
HAL	Hardware Abstraction Layer, Hardware-Abstraktionsschicht
HIL	Hardware In The Loop,Test mit Simulation der Peripherie des eingebetteten Systems
IAV	Ingenieurgesellschaft für Auto und Verkehr
IDE	Integrated Development Environment, integrierte Entwicklungsumgebung
IP	Internetprotokoll, wird häufig in Verbindung mit TCP oder UDP verwendet
MISRA-C	C-Programmierstandard, erarbeitet und veröffentlicht von der Motor Industry Software Reliability Association
OSI	Open Systems Interconnection Model
Qt	C++ Bibliothek für plattformunabhängige grafische Oberflächen
RUP	Rational Unified Process
SCRUM	Agiles Prozessmodell zur Softwareentwicklung
SOAP	Simple Object Access Protocol, Netzwerkprotokoll zum Datenaustausch und für Remote Procedure Calls
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protokoll, verbindungsloses Protokoll der Transportschicht
UDS	Unified Diagnostic Services, Kommunikationsprotokoll in der Automobiltechnik
UML	Unified Markup Language, Vereinheitlichte Modellierungssprache

TCP	Transmission Control Protocol, verbindungsorientiertes Protokoll der Transportschicht
XCP	Universal Measurement and Calibration Protocol, Netzwerkprotokoll in der Automobiltechnik
XML	eXtensible Markup Language, erweiterbare Auszeichnungssprache

Vorwort

Die Diplomarbeit wurde im Zeitraum vom 15. Juli 2013 bis zum 4. November 2013 in Zusammenarbeit mit der IAV Chemnitz angefertigt.

Die IAV bietet seit 30 Jahren Ingenieursdienstleistungen rund um Kraftfahrzeuge an. Mehr als 5000 Mitarbeiter weltweit unterstützen globale sowie lokale Projekte in den Bereichen Elektronik-, Antriebsstrang- und Fahrzeugentwicklung.

Besonderer Dank gebührt Herrn Prof. Dr. Christian Troll, der mir während der Arbeit immer wieder hilfreiche Tipps gab und es verstanden hat, meine Sichtweise und Denkansätze in die richtige Richtung zu lenken bzw. zu erweitern. Für die aktive Unterstützung während der Arbeit bedanke ich mich bei meinem Betreuer in der IAV, Herrn Marko Meyer, sowie den Mitarbeitern Enrico Walther und Karsten Juschus.

Weiterhin bedanke ich mich auch bei meiner Frau und meiner Familie, die mir besonders während der Diplomarbeit immer den nötigen Rückhalt gaben.

1. Einleitung

1.1. Hintergrund

Der Einsatz von Software ist in den letzten Jahrzehnten sehr stark angestiegen. Von einem einfachen Toaster bis hin zu komplizierten Computertomographen durchdringt Software immer mehr Lebensbereiche. Besonders in Kraftfahrzeugen nimmt Software einen stetig wachsenden Stellenwert ein. Damit einhergehend hat auch die Komplexität der verwendeten Software zugenommen. Während 1979 die Software eines Autos im Schnitt aus 2000 Codezeilen bestand, waren es 2001 schon 2 Millionen [Hau05]. In dem 2012 vorgestellten Elektroauto Opel Ampera verfünffachte sich dieser Wert noch einmal [Ilg12].

Parallel zum Anstieg der Komplexität erhöhen sich auch die sicherheitskritischen Anwendungen in Fahrzeugen. Diese reichen von Airbags über ESP-Systeme bis zu modernen Fahrerassistenzsystemen mit Lenkeingriffsmöglichkeit. Eine Fehlfunktion, auch wenn Sie nur einen kurzen Moment andauert, kann bereits lebensgefährliche Folgen haben. In diesem Zusammenhang entsteht damit eine Abhängigkeit der Sicherheit im Betrieb eines Fahrzeugs von der Qualität und Zuverlässigkeit der verwendeten Software.

Vor diesem Hintergrund werden Tests, welche geeignet sind, die Qualität einer Software nachzuweisen, immer wichtiger. Durch systematisches Testen lassen sich Fehler frühzeitig erkennen und beheben.

1.2. Problem und Zielstellung

Eine Schwierigkeit ist das Testen von separaten Softwaremodulen auf eingebetteten Systemen. Eingebettete Systeme können z.B. Steuergeräte in einem Kfz sein. Der Test wird typischerweise auf einem PC mit dem Compiler für die Zielhardware kompiliert, dorthin übertragen, ausgeführt und das Resultat zurück gelesen. Besonders bei vielen Testvektoren ist dieser Prozess sehr langwierig, da dann meist nicht alle Testvektoren mit einem Mal auf das Steuergerät gebracht werden können. Der Test müsste also in viele Teiltests aufgeteilt werden. Dies ist unübersichtlich und nur schwer automatisierbar.

Das Problem lässt sich umgehen, indem man die Tests nicht auf dem Zielsystem, sondern auf einem PC ausführt. Allerdings hat diese Vorgehensweise mindestens zwei Nachteile. Zum einen

wird ein anderer Compiler verwendet als später bei der Übertragung auf die Zielhardware. Zum anderen unterscheiden sich die Prozessoren zwischen eingebettetem System und PC in Architektur und Taktrate, weshalb nur bedingt Aussagen zur Ausführungsgeschwindigkeit oder zur Funktionalität zeitkritischer Teile des Codes gemacht werden können. Ein dritter Grund für Tests auf dem Zielsystem besteht darin, dass Module, die teils in C und teils in Assembler programmiert wurden, nur auf dem Zielsystem testbar sind.

Vor diesem Problem stand auch die IAV, weshalb nach Verbesserungsmöglichkeiten gesucht wurde. Es sollte möglich sein, das Modul und dazugehörige Tests getrennt von den Testdaten auf das eingebettete System zu übertragen. Der Test sollte vom PC aus mit definierten, sukzessive zu übertragenen Testvektoren ausgeführt werden können. Die Kommunikation könnte über eine der vorhandenen Schnittstellen der jeweiligen Zielhardware erfolgen. Dies können z.B. Ethernet, CAN (Controller Area Network) oder UART (Universal Asynchronous Receiver Transmitter) als typische Vertreter auf eingebetteten Systemen der IAV sein.

1.3. Struktur der Arbeit

Es existieren verschiedene Vorgehens- bzw. Prozessmodelle, nach denen eine Software, wie die in dieser Arbeit diskutierte Lösung, entwickelt werden kann. Zu nennen sind hier unter anderem das Wasserfallmodell, das V-Modell sowie RUP (Rational Unified Process), SCRUM (Agiles Prozessmodell zur Softwareentwicklung) und Extreme Programming.

Das Wasserfallmodell führt zu sehr strukturiertem Vorgehen, da die einzelnen Phasen klar abgegrenzt sind. Gerade durch diesen sequentiellen Ablauf ist das Modell allerdings unflexibel. Eine Weiterentwicklung stellt das Spiralmodell dar, welches zusätzlich eine Risikoabschätzung in jeder Stufe enthält. Dies ist aber eher für mittlere bis größere Projekte praktikabel. Auch RUP ist durch seinen umfassenden Ansatz und die darin enthaltene Komplexität nur für große Projekte geeignet.

SCRUM hat die Intention, komplexe Projekte in weniger komplexe, zeitlich begrenzte Abschnitte aufzuteilen. In diesen Abschnitten wird immer wieder ein Zyklus durchlaufen, der aus der Aufgabendefinition für den Zyklus, der transparenten Umsetzung, Überprüfung und Anpassung besteht. Diese feste Abfolge, die meist zwei bis vier Wochen dauert, behindert allerdings in kleineren Projekten die Flexibilität.

Extreme Programming stellt eine interessante Alternative zu den bisher betrachteten Vorgehensweisen dar, da es den Projektaufwand minimiert. Seine große Stärke spielt es aber vor allem aus, wenn noch nicht alle Anforderungen zu Beginn eines Projektes bekannt sind. Die hier gestellte Aufgabe ist bereits sehr präzise definiert, so dass dieser Vorteil ungenutzt bliebe.

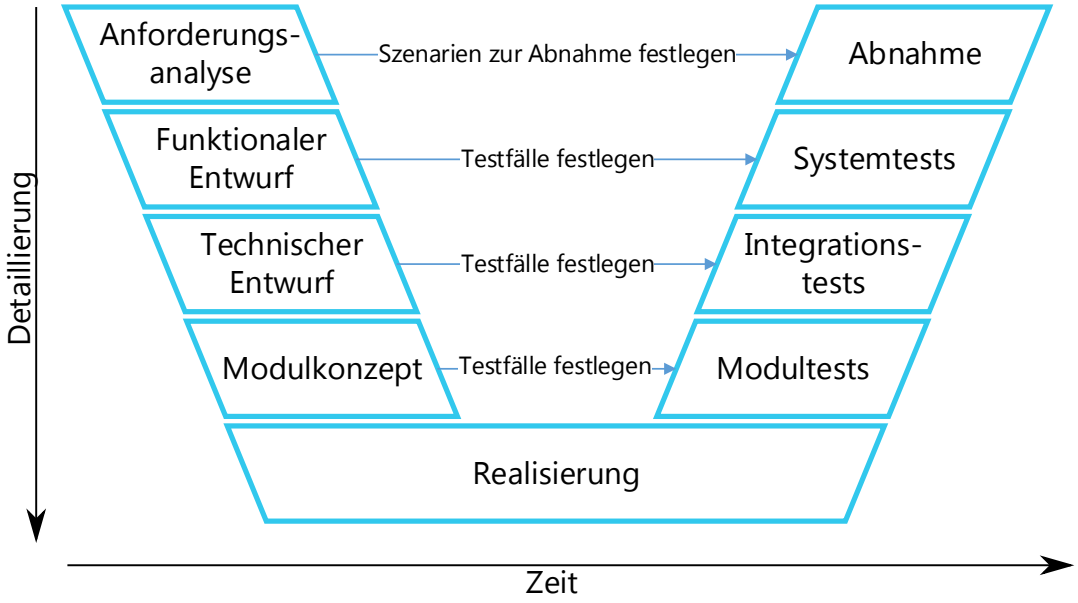

Abb. 1.1.: Übersicht der Stufen des V-Modells

Das V-Modell ist eine Vorgehensweise, die etwas verändert unter dem Namen V-Modell XT für öffentliche Softwareprojekte in der Bundesrepublik Deutschland zum Standard erhoben wurde [Bun10]. Auch in der Automobilbranche wird oft das V-Modell verwendet [DM08]. Ein Vorteil dieses Modells ist, dass Designschwächen meist früh im Entwicklungsprozess erkannt werden, da von relativ groben, aber eindeutigen Anforderungen ein immer detaillierteres Konzept erstellt wird, während gleichzeitig schon die Verifizierung bzw. Validierung in der zugehörigen Teststufe vorbereitet wird. Diese Herangehensweise wird durch viele Werkzeuge unterstützt. Die Konzeptphasen lassen sich sehr gut in UML (Unified Markup Language, Vereinheitlichte Modellierungssprache) abbilden.

Auf Grund der genannten Vorteile wurde die Entwicklung dieses Projektes an das V-Modell angelehnt. Es unterteilt sich in verschiedene Stufen, die in Abbildung 1.1 anschaulich dargestellt werden.

In der Phase der *Anforderungsanalyse* werden die benötigten Softwaremerkmale analysiert und die Ziele des Projektes definiert. Es werden funktionale sowie nichtfunktionale Anforderungen festgelegt, die später als Abnahmekriterien dienen. Grafisch modelliert werden können sie beispielsweise mit UML Use-Case Diagrammen.

Der *funktionale Entwurf* teilt die Anforderungen in einzelne, präziser definierte Funktionen auf. Es wird außerdem festgelegt, auf welche Weise diese Funktionen getestet werden können.

Der *technische Entwurf* beinhaltet eine etwas grobere Beschreibung des Gesamtsystems, welches aus den gewünschten Funktionalitäten konzipiert wurde. Schnittstellen werden identifiziert. Dies kann auch in UML geschehen. Als Darstellungsart eignet sich dafür unter anderem das

Komponentendiagramm. Zusätzlich werden Integrationstests festgelegt und der Systemtest erweitert.

Die Erstellung des *Modulkonzepts* umfasst die Aufteilung der Komponenten in Module und die Beschreibung der Module, so dass ein Programmierer sie daraus codieren kann. Dazu gehören die Modullogik, alle Schnittstellenspezifikationen, Abhängigkeiten sowie Variablen und deren Wertebereiche. Es werden Modultests festgelegt und die vorhandenen Integrationstests erweitert und ergänzt.

2. Softwaretests

2.1. Herausforderung

Fast jeder Softwareentwickler wird aus Erfahrung bestätigen können, dass Softwaretests unabdingbar sind. Trotzdem kommt das Testen in der Praxis manchmal zu kurz. Gründe dafür sind vielfältig:

Kosten Bei der Softwareentwicklung steht die Qualität meist betriebswirtschaftlichen Interessen gegenüber. Hinzu kommt, dass sich Aufwand für Softwaretests nicht sofort auszahlt bzw. nicht direkt in der Funktionalität des Endproduktes ersichtlich ist [DH08, S.10]. So kann es vorkommen, dass Budgets für das Testen zu niedrig angesetzt werden.

Termindruck Oft verlängert sich auch die Entwicklungszeit durch ungeplante Schwierigkeiten. In solchen Situationen ist die Versuchung groß, Softwaretests einzusparen, um Terminvorgaben einzuhalten.

Monotonie Testen wird oft als monotone und wenig kreative Beschäftigung empfunden. Mit den unter 2.3 auf der nächsten Seite beschriebenen Methoden, wie das parallele Testen zur Entwicklung oder der Verwendung geeigneter Werkzeuge lässt sich dieses Problem umgehen.

Designfehler Eine weitere Herausforderung für das Testen entsteht, wenn die Tests nicht in den Entwicklungsprozess mit aufgenommen, sondern hinten angestellt werden. Werden sich erst spät Gedanken über dieses Thema gemacht, kann es vorkommen, dass der Code schwierig oder nicht umfangreich genug getestet werden kann.

2.2. Notwendigkeit

Laut [Kle13, S.1] beansprucht die Qualitätssicherung von Software bis zu 70% der gesamten Entwicklungszeit. Tests sind ein wichtiger Bestandteil davon. Obwohl, wie schon erwähnt, Testen eine Herausforderung sein kann, stellt es doch eine Notwendigkeit dar. Hier einige Gründe dafür:

Vertraglich zugesicherte Qualität Kundenspezifische Softwareentwicklung im gewerblichen Bereich wird meist durch schriftliche Verträge geregelt, die genau vorschreiben, welche Restfehlerwahrscheinlichkeit oder Verfügbarkeit ein Softwaresystem bieten muss.

Schäden vermeiden Auswirkungen von Softwarefehlern können von materiellen Schäden bis hin zu Personenschäden führen. Ein historisches Beispiel für einen Softwarebug, der tödliche Folgen hatte, ist der medizinische Linearbeschleuniger Therac-25, der zur Behandlung von Karzinomen eingesetzt wurde. In den Jahren 1985 bis 1987 kam es zu sechs Bestrahlungen mit erheblicher Überdosis. Schuld waren ein Fehler, der zu einem Integerüberlauf führte, und eine fehlerhafte Programmlogik. Drei Patienten starben daraufhin [Lev95]. Welche wirtschaftlichen Schäden Softwarefehler verursachen können, wurde besonders deutlich, als 1996 die Schwerlast-Trägerrakete Ariane 5 kurz nach dem Start von ihrem Kurs abkam und sich selbst zerstörte. Der finanzielle Verlust wird auf 290 Millionen Euro geschätzt. Ursache war der Überlauf einer 16-Bit Integer-Variable [DW12].

Erweiterbarkeit Tests geben Entwicklern Sicherheit, die eine Software weiterentwickeln. Sind hinreichende Tests vorhanden, kann jederzeit der neue Softwarestand gegen die Anforderungen geprüft werden.

Konkurrenz Gerade im Bereich Standardsoftware existieren viele Konkurrenzprodukte. Bietet ein Produkt ein schlechtes Benutzererlebnis, wechseln Kunden zu anderen Anbietern.

Folgekosten Unterlassene Tests führen nicht selten zu hohen Folgekosten in der Entwicklungszeit, weil z.B. nachträgliche Konzeptänderungen weitgehende Modifikationen der Implementierung bedingen. Noch schlimmer ist es, wenn Fehler erst beim Kunden entdeckt werden. Dies führt zu kostspieligen Rückrufaktionen, Mehraufwand für Support und Imageverlust.

2.3. Effektiv Testen

Ein Softwaretest ist nur effektiv, wenn dadurch neue, nicht redundante Informationen über die Güte des Programms hinzugewonnen werden können. Es existieren verschiedene Möglichkeiten, wie Tests effektiver gestaltet werden können:

Parallel zur Entwicklung Werden sich schon während der Konzeptphase Gedanken über Tests gemacht, hat dies verschiedene Vorteile. Einer wäre, dass die Software gut modularisiert ist, da Modultests Module erfordern, die isoliert testbar sind. Das Konzept der *testgetriebenen Entwicklung* geht noch einen Schritt weiter, indem die Tests schon vor der Modulprogrammierung erstellt werden. Werden sie dagegen erst nach der Programmierung entwickelt, führt dies eher zu einem monolithischen Softwaredesign, welches nur mit viel Aufwand testbar ist.

Systematische Herangehensweise Werden Test nicht direkt im Entwicklungsprozess verankert, sondern nur als vage Anforderung beschrieben, kommt es meistens nur zu sporadischen Tests, die oft auch nicht automatisiert ausführbar sind. Softwaretests sollten daher einen festen Platz in der Entwicklung haben.

Verwendung von Testwerkzeugen Die Sammlung von Tests wächst mit dem Umfang des Projektes. Da ist es wünschenswert, die Software automatisiert testen zu können. Die verschiedenen Arten von Testwerkzeugen werden in Kapitel 2.5 auf Seite 10 aufgezeigt.

Return on invested time Wird die Zeit für das Schreiben und Ausführen der Tests ihrem Nutzen und dem Risiko nicht erkannter Fehler gegenübergestellt, ergibt sich eine Grenze, bis zu der das Testen ökonomisch sinnvoll ist. Diese Grenze kann je nach Anwendung stark variieren. Während in der Raumfahrt oder bei Verkehrsmitteln Menschenleben gefährdet sein können, hinterlässt ein Softwarefehler auf einem MP3-Player gewöhnlich keinen signifikanten Schaden. Um die geplante Zeit für das Testen gut zu nutzen, sollten nur Tests implementiert werden, die einen Mehrwert bringen. Tests, die nie fehlschlagen können oder deren Ergebnis die Teilmenge von Ergebnissen anderer Tests ist, sind nicht effektiv.

2.4. Klassifikation von Softwaretests

Softwaretests werden in verschiedenen Ebenen der Entwicklung durchgeführt. Im V-Modell wird zwischen Modultests, Integrationstests, Systemtests und Abnahmetests unterschieden (siehe Abb. 1.1 auf Seite 3). Weiterhin lassen sich Tests nach White- oder Black-Box-Test klassifizieren.

2.4.1. Klassifikation nach Entwicklungsebene

Modultests Es werden Teilfunktionen (Module) des Systems isoliert mit Testvektoren beaufschlagt und überprüft, ob die Resultate den Erwartungen entsprechen.

Integrationstests Hier wird getestet, ob die Module korrekt zusammenarbeiten. So werden z.B. Schnittstellenfehler erkannt.

Systemtests Bei diesem Test wird geprüft, ob das Ergebnis mit der Spezifikation übereinstimmt. Ausgeführt wird er gewöhnlich durch die Entwickler selbst unter Benutzung von synthetischen Testdaten. Je nach Betrachtungsweise kann ein Systemtest auch die Hardware umfassen und wäre dann kein reiner Softwaretest mehr.

Abnahmetests Der Kunde testet, typischerweise unter realen Bedingungen, ob das gelieferte Produkt seinen Anforderungen entspricht. Je nach Betrachtungsweise kann ein Abnahmetest auch die Hardware umfassen und wäre dann kein reiner Softwaretest mehr.

Mit der zu entwickelnden Testausführungsumgebung sollen Modultests sowie in begrenztem Maße auch Integrationstests durchgeführt werden können.

2.4.2. Klassifikation in White- und Black-Box-Test

Bei *White-Box Tests* wird am Code geprüft. Tests werden mit dem Bewusstsein des zu testenden Codes entwickelt. White-Box-Tests können kontrollflussorientiert oder datenflussorientiert sein. Codeüberdeckungstests gehören dieser Kategorie an. Es gibt unterschiedliche Überdeckungstests. Beispiele dafür sind Anweisungsüberdeckung, Zweigüberdeckung und Bedingungsüberdeckung.

Um 100% Anweisungsüberdeckung zu erreichen, genügt es, dass jede Anweisung (in Hochsprache) einmal ausgeführt wird. Bei einem einzeiligen if-Statement oder einer Schleife kommt es vor, dass zwar eine vollständige Anweisungsüberdeckung erreicht wird, obwohl nicht alle Pfade ausgeführt wurden. Ein Zweigüberdeckungstest stellt dies sicher. Jedoch bietet auch dieser Test keine hinreichende Bestätigung dafür, dass jede Kombination von Bedingungen eine korrekte Verzweigung hervorruft. Um dies sicherzustellen wird mindestens ein Wert aus jeder Äquivalenzklasse einer Bedingung benötigt. Die Verzweigungsbedingung muss dann für alle Kombinationen dieser Werte getestet werden, was einer Bedingungsüberdeckung entspricht. Doch selbst 100% Bedingungsüberdeckung bedeuten nicht, dass der Code keine Fehler mehr enthält. Es können beispielsweise Fehler bei der Bedingungsabfrage an den Rändern der Äquivalenzklassen entstehen oder es könnten Kontrollflussverzweigungen im Code vergessen worden sein, für die danach auch keine Testfälle entwickelt wurden. Für Codeüberdeckungstests gibt es eine umfangreiche Werkzeugunterstützung. Beispiele dafür sind gcov und CTC++, wobei letzteres auch innerhalb der IAV eingesetzt wird.

Im Gegensatz zu den White-Box Tests stehen *Black-Box Tests*, welche die Funktionalität und die Schnittstelle eines Moduls prüfen. Es werden Eingabeparameter und die dazu erwarteten Rückgabewerte festgelegt. Die Eingabeparameter werden an das Modul, die Komponente oder das System übergeben. Die Rückgabewerte werden mit Erwartungswerten verglichen. Stimmen diese überein, war der Test erfolgreich (passed). Differieren sie dagegen, schlug der Test fehl (failed).

Wie der Aufgabenstellung zu entnehmen ist, soll die Testausführungsumgebung sowohl Testvektoren für Black-Box-Tests als auch CTC++ für White-Box-Tests unterstützen.

2.4.3. Modultests

Die in der Aufgabenstellung genannte Ausführungsumgebung soll hauptsächlich für Modultests verwendet werden. Daher ist es angebracht, noch etwas näher auf diese Art Tests einzugehen.

2.4.3.1. Isolation

Um ein Softwaremodul oder eine Komponente einem Modultest unterziehen zu können, müssen sie isoliert werden. Wären noch Abhängigkeiten vorhanden, wäre bei Fehlern nicht immer klar,

ob das Modul selbst oder die abhängige Komponente den Fehler enthält. Ein weiterer Grund, Module während des Modultests zu isolieren, liegt in der Laufzeit. Modultests werden häufig mit sehr vielen verschiedenen Testvektoren ausgeführt. Zugriffe auf Hardware und externe Softwarekomponenten verlangsamen typischerweise den Test. Es ist jedoch nicht immer möglich, solche Zugriffe zu unterbinden. So können beispielsweise manche Fehler nur entdeckt werden, wenn das Modul mit echten Laufzeiten getestet wird. Solche Probleme treten allerdings eher bei Integrationstests auf.

Auch zu komplexe Module verlängern die Laufzeit und führen zu sehr vielen Testvektoren. Durch Refaktorieren eines Moduls kann es in kleinere Teile aufgelöst werden, womit sich dieses Problem oft beheben lässt.

Zur Isolation von Modultests bedient man sich Stubs oder Mocks. Stubs sind Funktionen oder Objekte, die ungeachtet ihrer Eingabewerte immer dieselben Ausgabewerte erzeugen. Mocks dagegen liefern für bestimmte Eingabeparameter definierte Ausgabewerte. Mocks sollten sparsam eingesetzt werden, da komplexe Mocks selbst wieder Potenzial für Fehler bieten.

Beispiel Angenommen, es soll ein Modul getestet werden, welches Daten über eine Hardwareschnittstelle empfängt, danach verarbeitet und die Ergebnisse über dieselbe Schnittstelle wieder ausgibt. Das Empfangen der Daten kann entweder mit einer Stubfunktion realisiert werden, welche immer dieselben Daten liefert oder über einen eingebauten Zähler mit jedem Funktionsaufruf neue Daten liefert. Eine weitere Stubfunktion ersetzt das Senden der Daten. Typischerweise ist in dieser Funktion keinerlei Logik erforderlich, so dass einfach mit dem Vermelden des Sendeerfolgs daraus zurückgekehrt wird.

2.4.3.2. Allgemeine Ausführung eines Modultests

Ein Modultest wird üblicherweise in drei Schritten durchgeführt: Setup, Test und Teardown:

Setup Es wird eine Umgebung für den Test geschaffen. Benötigte Objekte, Strukturen und Daten werden bereitgestellt, auf die das Modul später zugreift. Dies ist wichtig, damit der Test aus einem definierten Umgebungszustand heraus erfolgen kann und damit u.a. das Ergebnis reproduzierbar wird.

Test In diesem Schritt erfolgt der eigentliche Test. Das Modul wird mit den definierten Eingabeparametern aufgerufen. Die Rückgabewerte werden mit den Erwartungen verglichen.

Teardown Dieser Schritt ist nicht immer notwendig. Er wird z.B. dazu genutzt, belegten Speicher freizugeben oder andere Hardware wieder in den Ausgangszustand zu versetzen. Das Ziel ist es, verschiedene Tests in beliebiger Reihenfolge ausführen zu können, ohne dass durch eine nicht aufgeräumte Testumgebung Abhängigkeiten oder Fehler entstehen.

2.4.3.3. Ausführung von Modultests auf eingebetteten Systemen

Die Ausführung von Modultests auf eingebetteten Systemen bringt einige Herausforderungen mit sich.

- Die Zielhardware ist oft nicht beliebig oft verfügbar, so dass unter Umständen nicht jeder Entwickler ein eigenes Testsystem hat.
- Besonders auf kleinen Mikrocontrollern steht oft nur wenig RAM und Flashspeicher zur Verfügung. Wird ein Testframework für den Modultest verwendet, muss darauf geachtet werden, dass das eingebettete System genügend Ressourcen dafür zur Verfügung stellt.
- Das Laden des Tests auf die Hardware nimmt mehr Zeit in Anspruch, als den Test auf dem PC auszuführen.
- Der Transport der Testergebnisse zurück auf den PC kann aufwändig sein.

Besonders die drei letztgenannten Punkte werden in dieser Diplomarbeit aufgegriffen.

2.4.3.4. Fehlerfreiheit

Eine Garantie auf Fehlerfreiheit können Modultests, wie auch alle anderen Softwaretests, nicht geben. Drei Gründe dafür wären:

- Ein Modultest hängt immer von der Spezifikation des Moduls ab. Der Test sagt nur aus, ob die Spezifikation korrekt umgesetzt wurde. Das bedeutet nicht, dass die Spezifikation selbst korrekt ist.
- Ein Modultest kann nur Fehler finden, für die er ausgelegt wurde. Ob eine Reihe von Modultests die Spezifikation korrekt abdecken, kann mit den Modultests an sich verständlicherweise nicht nachgewiesen werden.
- Alle möglichen Testfälle abzudecken wäre auf Grund der dafür benötigten Ressourcen für viele Modultests undenkbar. Deshalb werden die Testfälle in Äquivalenzklassen eingeordnet, aus denen jeweils nur ein Testfall ausgewählt wird (evtl. noch weitere um die Ränder der Klassen abzudecken). Erfolgten bei der Auswahl der Äquivalenzklassen Fehler, kann der Modultest nur lückenhafte Ergebnisse liefern.

Modultests sind daher immer nur als Nachweis für eine bestimmte Qualitätsstufe der Software anzusehen, nicht als Nachweis von Fehlerfreiheit.

2.5. Testwerkzeuge

Werkzeuge unterstützen Softwaretests und können sie bis zu einem gewissen Grad automatisieren. Damit steigt die Effizienz bei der Entwicklung und Durchführung von Softwaretests. Werkzeuge

erstellen auch Auswertungen über den Testfortschritt, das Testergebnis sowie Prüfprotokolle als Nachweis für den Kunden. Die Wahl der Vorgehensweise beim Testen können sie allerdings nicht abnehmen. In den letzten Jahren wurden immer mehr Testwerkzeuge verfügbar, wobei die meisten davon auf sehr spezielle Anwendungsfälle spezialisiert sind.

2.5.1. Einteilung

Art des Tests	Werkzeugarten
Dynamisch	• Strukturorientiert • Funktionsorientiert • Regressionstestwerkzeuge • Leistungs- und Stresstestwerkzeuge
Statisch	• Messwerkzeuge • Stilanalysatoren • Werkzeuge zur Erzeugung von Grafiken und Tabellen • Slicing-Werkzeuge • Werkzeuge zur Analyse von Datenflussanomalien
Formale Verifikation	Werkzeuge zur • Sicherheitsprüfung • Lebendigkeitsprüfung
Modellierend und Analysierend	• FMECA-Werkzeuge (Failure Mode and Effects and Criticality Analysis) • Markov-Modell-Werkzeuge • Fehlerbaumanalysewerkzeuge
Testmanagement	• Werkzeuge zur Erfassung, Katalogisierung und Verwaltung von Testfällen • Fehlermanagementwerkzeuge • Werkzeuge zur kontinuierlichen Integration

Tab. 2.1.: Einteilung Testwerkzeuge [DL09, S. 391ff]

Testwerkzeuge lassen sich je nach Sichtweise unterschiedlich einteilen. Tab. 2.1 zeigt eine Einteilung nach der Art der Untersuchung des Testlings.

Statische Testwerkzeuge untersuchen den Code, im Gegensatz zu dynamischen Werkzeugen, außerhalb der Laufzeit. Mit solchen Hilfsmitteln kann in einer Codeanalyse der Quellcode nach

festgelegten Merkmalen untersucht werden. Die Ergebnisse werden in der Regel grafisch oder textbasiert aufbereitet. Untersucht werden können z.B.:

- Einhaltung von Codierrichtlinien
- Softwaremetrik, z.B. zyklomatische Komplexität (Anzahl der binären Verzweigungen des Kontrollflussgraphen eines Moduls)
- Erzeugung von Kontrollfluss- und Aufrufgraphen
- Codeabhängigkeiten
- Codeplausibilität (aufspüren von Datenflussanomalien)

Werkzeuge zur formalen Verifikation sind kommerziell kaum erhältlich. Die Vielzahl dieser Werkzeuge stammt aus dem universitären Bereich. Ein Beispiel für die formale Verifikation ist das Symbolic Model Checking. Damit können Zustandsräume eines Automaten darauf untersucht werden, ob sie sicher und lebendig sind. Sicher bedeutet, dass kein unsicherer Zustand erreicht werden kann, während lebendig bedeutet, dass ein bestimmter Zustand immer wieder erreicht wird [DL09, S. 398-399].

Modellierende und analysierende Werkzeuge werden zur Risikoanalyse eingesetzt. Dazu zählen u.a. die Prognose von Ausfallwahrscheinlickeit und Verfügbarkeit.

Weiter gibt es noch viele andere Testwerkzeuge, die nicht direkt einen Softwaretest ausführen, aber dabei behilflich sind. Dazu gehören beispielsweise Werkzeuge zum Testfallmanagement, mit denen übersichtlich und schnell Testfälle erstellt und verwaltet werden könnnen. Auch Software zum Fehlermanagement oder für die kontinuierliche Integration befinden sich in dieser Kategorie.

Der größte Teil von Testwerkzeugen findet sich jedoch im Bereich der dynamischen Tests. Oft werden deshalb Softwaretests als Synonym für solche verwendet. Dynamisch bedeutet in diesem Zusammenhang, dass diese Werkzeuge zur Laufzeit testen. Strukturorientierte Werkzeuge überprüfen die Codeüberdeckung während der Laufzeit, d.h. welche Codezeilen, Verzweigungen, Bedingungen und Funktionen während eines Tests durchlaufen werden (siehe Abschnitt 2.4.2 auf Seite 8). Dazu instrumentieren sie in der Regel den Code. Funktionsorientierte Werkzeuge helfen bei der Testplanung, in dem sie Äquivalenzklassen finden und Testfälle generieren. Regressionstestwerkzeuge automatisieren die Testdurchführung, indem sie die Testfälle automatisch ausführen. Damit kann nach Codeänderungen sofort überprüft werden, ob der Code noch alle Tests besteht. Leistungstests ermitteln, wie sich eine Software bei ihrem Betrieb an der Lastgrenze verhält, während Stresstests den Betrieb im Überlastbereich testen. Gerade für diese Bereiche sind Werkzeuge unabdingbar.

2.5.2. Testframeworks

Testframeworks bieten eine API zur effizienten Erstellung und Durchführung von Tests. Die entstehenden Ergebnisse können abgerufen werden, nachdem sie vom Testframework zusammengestellt worden sind. Für einige Testframeworks gibt es Anbindungen an bekannte IDEs (Integrated Development Environment, integrierte Entwicklungsumgebung). Dadurch ist es möglich, Tests einfach mit in den Entwicklungsprozess einzubinden und ein Feedback bei jedem Kompilieren zu erhalten, ob der veränderte Code alle Tests weiterhin besteht.

Testframeworks existieren in sehr unterschiedlichen Ausprägungen. Sehr einfache Varianten bestehen nur aus einer C-Headerdatei, während umfangreichere Frameworks eine grafische Oberfläche und grafische Reports ermöglichen und viele andere Werkzeuge zusätzlich enthalten, wie z.B. Testfallgeneratoren, Testfallverwaltung und Codeüberdeckungsanalysatoren.

Ein sehr bekanntes Testframework für die Programmiersprache Java ist JUnit. JUnit entstand aus dem Testframework SUnit, das ursprünglich für Smalltalk (dynamische, objektorientierte Programmiersprache) entwickelt wurde. Mit JUnit lassen sich sehr einfach automatisierte Unit-Tests erstellen, organisieren und ausführen. JUnit wurde mit den Java Development Tools in die Entwicklungsumgebung Eclipse integriert.

Viele Nachahmungen für andere Sprachen basieren auf der Idee von JUnit und werden unter dem Namen xUnit zusammengefasst. So existieren auch für die Programmiersprachen C und C++ einige Test-Frameworks auf dieser Basis, wie z.B. CUnit, CppUnit oder CppUnit-Light. Testframeworks basierend auf der xUnit-Architektur beinhalten mindestens folgende Komponenten:

Testfall Alle Modultests leiten von dieser Klasse ab

Testumgebung (Test-Fixture) Stellt eine definierte und isolierte Testumgebung bereit, damit Tests wiederholbar sind. Nach dem Test sollte wieder in den Originalzustand zurückgekehrt werden.

Testsuite Ein Satz von Tests, die dieselbe Umgebung (Fixture) benutzen

Testausführung Die Ausführung von Tests erfolgt immer in der Reihenfolge Setup, Test und Tear-Down.

Zusicherung Eine Zusicherung (englisch: assertion) ist ein Prüfkriterium für das Verhalten des zu testenden Moduls. Wird eine Zusicherung nicht eingehalten, schlägt typischerweise der Test fehl.

2.5.3. Testframeworks für eingebettete Systeme

Auf eingebetteten Systemen ergeben sich besondere Herausforderungen beim Testen. Da der Test zwar auf dem eingebetteten System ausgeführt wird, aber Kompilierung und Auswertung auf

einem PC erfolgen, ist die Handhabung nicht so unkompliziert wie ein reiner PC-basierter Test. Auch der Einsatz von gewöhnlichen Testframeworks für PCs ist nicht ohne weiteres möglich. Eine Beschränkung ist hier z.B., dass auf eingebetteten Systemen typischerweise keine dynamische Speicherallokierung genutzt werden kann.

Deshalb gibt es speziell für den Einsatz auf solchen Systemen angepasste Testframeworks. Beispiele dafür sind Embedded Unit und μCUnit. Zum Ausführen von Tests müssen diese allerdings zusammen mit dem Framework kompiliert, auf das Zielsystem übertragen, ausgeführt und die Ergebnisse zurück übertragen werden. Besonders bei großen Testvektoren, wie z.B. Bildern, kann dies eine langwierige Angelegenheit sein.

Einen anderen Ansatz liefert die Software Tessy. Tessy ist gewissermaßen ein Allround-Tool für Unit- und Integrationstests auf eingebetteter Hardware. Es analysiert den Modul-Code automatisch und extrahiert die Schnittstellen. Über die auf Eclipse basierende grafische Oberfläche lassen sich Tests anlegen, organisieren und Testfälle generieren. Die Tests können auf der Zielhardware ausgeführt und die Ergebnisse in einer Dokumentation zusammengefasst werden. Tessy 3.0 unterstützt eine Vielzahl an Plattformen [Raz13]. Modul- und Integrationstests können automatisch ausgeführt werden und eignen sich damit für Regressionstests. Auch eine Codeüberdeckungsanalyse ist möglich. Weiterhin unterstützt Tessy HIL (Hardware In The Loop,Test mit Simulation der Peripherie des eingebetteten Systems) [Hit13a].

Tessy benutzt den Debugger für das Zielsystem, um die Testumgebung herzustellen, den Test zu steuern, aber auch um dem Benutzer Eingriffsmöglichkeiten in den Test zu bieten [Hit13b].

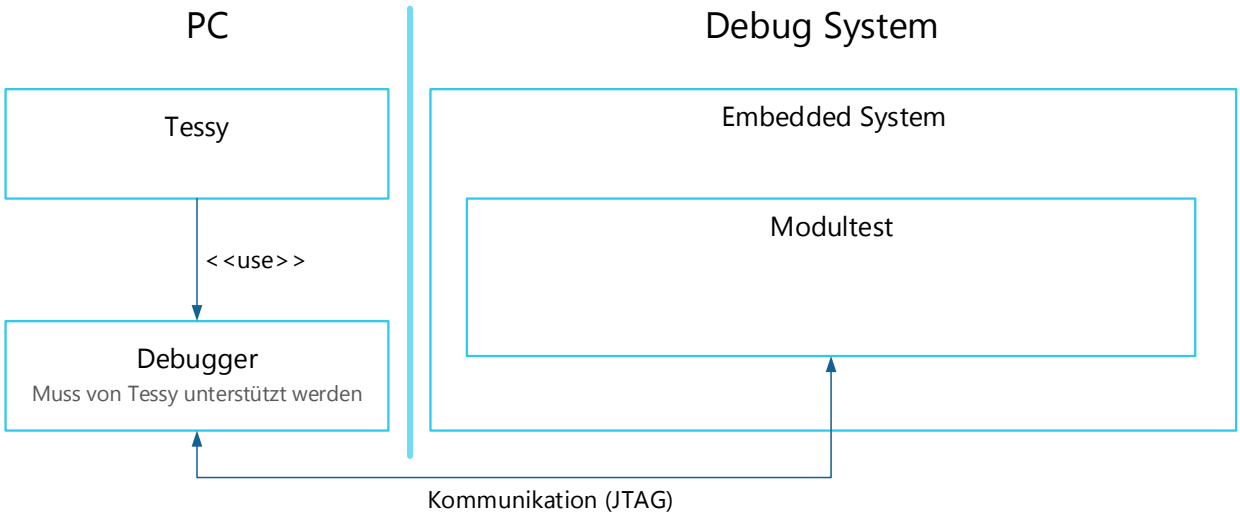

Abb. 2.1.: Konzept des Testwerkzeugs Tessy für einen Modultest auf der Zielplattform

Einen ähnlichen Umfang wie Tessy bietet VectorCAST/C++, welches vom Hersteller Vector Software als die meistautomatisierte Plattform für C und C++ Tests beworben wird. Der Unterschied zu Tessy besteht darin, dass die Testvektoren und Testergebnisse nicht zwangsläufig

über JTAG mit der Zielhardware ausgetauscht werden müssen, sondern dies auch über eine andere Kommunikationsschnittstelle der Zielhardware erfolgen kann.

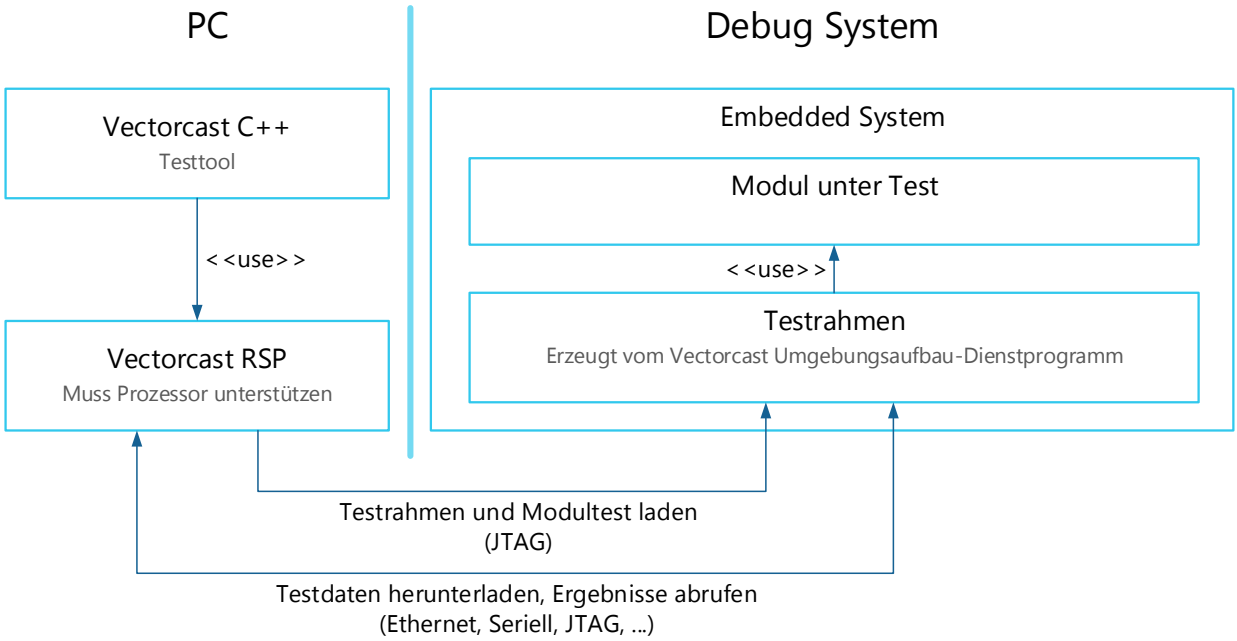

Abb. 2.2.: Konzept des Testwerkzeugs Vectorcast C++ und RSP für einen Modultest auf der Zielplattform

Diese beiden Softwareprodukte würden sich perfekt für das Testen auf eingebetteten Systemen eignen, sind allerdings nicht ganz günstig. Tessy beispielsweise kostet pro Lizenz etwa 3000 €. Außerdem bieten beide Systeme eine ungeheure Funktionsvielfalt, von der große Teile nicht von der IAV (Ingenieurgesellschaft für Auto und Verkehr) benötigt werden.

Daraus erwuchs der Entschluss, ein eigenes, an die Erfordernisse angepasstes System zu entwickeln.

3. Anforderungen

Die Anforderungen für das System wurden in funktionale Anforderungen und nichtfunktionale Anforderungen unterteilt.

3.1. Funktionale Anforderungen

Die funktionalen Anforderungen wurden in Abb. 3.1 als Anwendungsfalldiagramm dargestellt. Nachfolgend werden die Anwendungsfälle näher beschrieben.

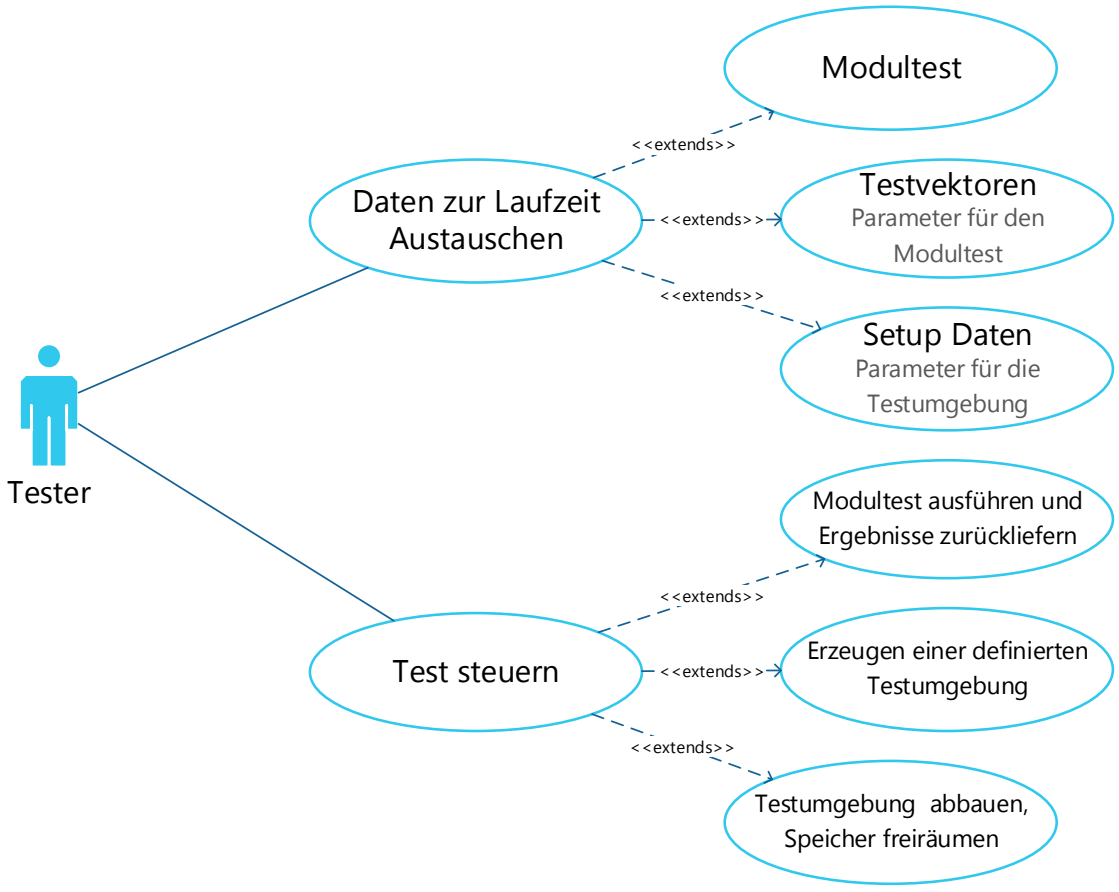

Abb. 3.1.: Funktionale Anforderungen

3.1.1. Austausch von Daten zur Laufzeit

Die Testausführungsumgebung soll, wie auch die kommerziellen Testwerkzeuge Tessy und Vectorcast, die Daten, die für den Test notwendig sind, zur Laufzeit austauschen können. Dies soll über eine vorhandene Datenschnittstelle, wie z.B. CAN, Ethernet oder Seriell, zwischen dem Steuer-PC und dem eingebetteten System möglich sein. Weiterhin soll auch der Modultest selbst über diese Schnittstelle ausgetauscht werden können. Es soll so möglich sein, vom PC aus verschiedene Tests mit jeweils mehreren Testvektoren als Stapelverarbeitung ausführen zu können.

3.1.2. Setup, Test und Teardown

Wie in vielen Testframeworks üblich, sollen die drei Methoden Setup, Test und Teardown vom PC aus einzeln aufgerufen werden können. Dabei muss die Methode Setup die Setup-Daten zur Bereitstellung der Testumgebung erhalten und die Methode Test die Testvektoren. Sofort im Anschluss der Ausführung eines Tests sollen die Ergebnisdaten zum PC übertragen werden. Die Laufzeit der Test-Funktion soll gemessen und an den PC übertragen werden können.

Im Anwendungsfall *Setup* kann für den auszuführenden Test die gewünschte Umgebung geschaffen werden, während *Test* den eigentlichen Modultest startet. Am Ende des Tests werden automatisch die Ergebnisse übertragen. *Teardown* ist dazu da, nicht mehr benötigte Ressourcen nach dem Test wieder freizugeben. Die Ausführungsumgebung hat keinen Einfluss auf die Umsetzung dieser Anwendungsfälle, da diese Befehle nur an den Modultest weitergereicht werden. Wie und ob im Modultest die einzelnen Funktionen auskodiert werden und ob ein Testframework eingesetzt wird, bleibt dem Entwickler des Tests überlassen.

3.2. Nichtfunktionale Anforderungen

3.2.1. Portabilität

Auf dem Zielsystem gibt es hohe Anforderungen an die Portabilität. Das Testframework muss sowohl auf einfachen 8-Bit Controllern, wie den Freescale HC08 und Renesas 78K0 Familien, lauffähig sein als auch auf 32-Bit Controllern, wie den Freescale MPC 56xx und Renesas V850 Familien.

Um die Software unabhängig von der Hardware zu entwickeln, wird ein HAL (Hardware Abstraction Layer, Hardware-Abstraktionsschicht) für jedes verwendete Zielsystem von der IAV bereitgestellt. Zu Beginn wird ein HAL für den PC entwickelt. Dies geschieht parallel zur Entwicklung der Testausführungsumgebung. Darin enthalten sein müssen Kommunikationstreiber

für die Übertragungsschnittstellen sowie weitere Treiber für z.B. Timer, Speicherzugriff und Watchdog.

3.2.2. Zuverlässigkeit

Zu einer Steuergerätesoftware werden oft viele Modultests erstellt, die wiederum mit vielen Testvektoren durchgeführt werden können. Weil dadurch ein nicht unerheblicher Zeitaufwand entsteht, werden solche Testreihen meist unbeaufsichtigt durchgeführt, häufig über Nacht. In diesem Sinne ist die Zuverlässigkeit der Implementierung wichtig.

Sollte ein Unit-Test einen Fehler verursachen, muss das Steuergerät automatisch in einen definierten Zustand gebracht werden. Die PC-Software muss den Fehler erkennen, festhalten und automatisch mit den nächsten Test-Vektoren oder, bei mehrfachem Fehlschlag, mit dem nächsten Unit-Test beginnen.

3.2.3. Speichereffizienz

Eingebettete Systeme verfügen nur über begrenzte Speicher-Ressourcen. Eine Version des Mikrocontrollers Freescale HC08G, welche für kleiner Projekte eingesetzt wird, verfügt beispielsweise nur über 4 kB Flash.

3.2.4. Wartbarkeit

Um die Wartbarkeit und Lesbarkeit des Codes zu steigern, werden folgende Maßnahmen angewandt:

- Anwendung von MISRA-C (C-Programmierstandard, erarbeitet und veröffentlicht von der Motor Industry Software Reliability Association)
- Codekommentare sind in einem Format anzulegen, dass daraus mit Tools wie Doxygen eine automatische Codedokumentation erzeugt werden kann.
- Die Wartbarkeit des Codes wird durch sinnvolle Unit-Tests und Integrationstests verbessert.
- Das Übertragungsprotokoll für die Kommunikations zwischen Host- und Zielsystem muss so ausgelegt werden, dass spätere Erweiterungen einfach integrierbar sind. Außerdem ist eine Dokumentation des Protokolls anzufertigen.
- Modellierung mit UML auf allen Designebenen

3.2.5. Kommunikation

Die Kommunikation soll über die vorhandenen Kommunikationsschnittstellen des eingebetteten Systems erfolgen. Da je nach System unterschiedliche Schnittstellen zur Verfügung stehen und diese auch unterschiedlich anzusprechen sind, müssen die hardwareabhängigen OSI-Schichten in der HAL umgesetzt werden. Auf diesem Kommunikationstreiber muss ein Protokoll entwickelt werden, welches die verbleibenden, hardwareunabhängig implementierbaren OSI-Schichten 5 bis 7 abdeckt. Dieses Protokoll muss unabhängig von Hardwarespezifikationen wie Alignment oder Byte-Order sein.

Es sollen die Schnittstellen Ethernet, CAN und UART betrachtet werden, da diese typischerweise in den aktuellen Embedded Systems der IAV verwendet werden.

3.2.6. Ressourcenverwaltung

Um keine aufwändige Ressourcenverwaltung, die der in einem Betriebssystem ähnlich ist, implementieren zu müssen, wurde von der IAV definiert, dass alle zu testenden Module nicht direkt auf Hardware zugreifen dürfen. Sollte ein Hardwarezugriff nötig sein, ist dieser in einem Funktionsaufruf zu kapseln. Die aufzurufende Funktion kann dann vom Modultest als Stub- oder Mockfunktion implementiert werden.

3.2.7. Programmiersprache

Für die Programmiersprache C werden Compiler auch für kleinere Controller angeboten, während C++-Compiler häufig nur für leistungsfähigere Varianten erhältlich sind. Deshalb ist C als Programmiersprache auf dem Zielsystem zu verwenden.

4. Konzept zur Realisierung

4.1. Funktionaler Entwurf

Die Funktionen, welche das System erfüllen muss, können aus den Anforderungen abgeleitet werden. Alle abgeleiteten Funktionen sind in Abb. 4.1 als Anwendungsfalldiagramm dargestellt.

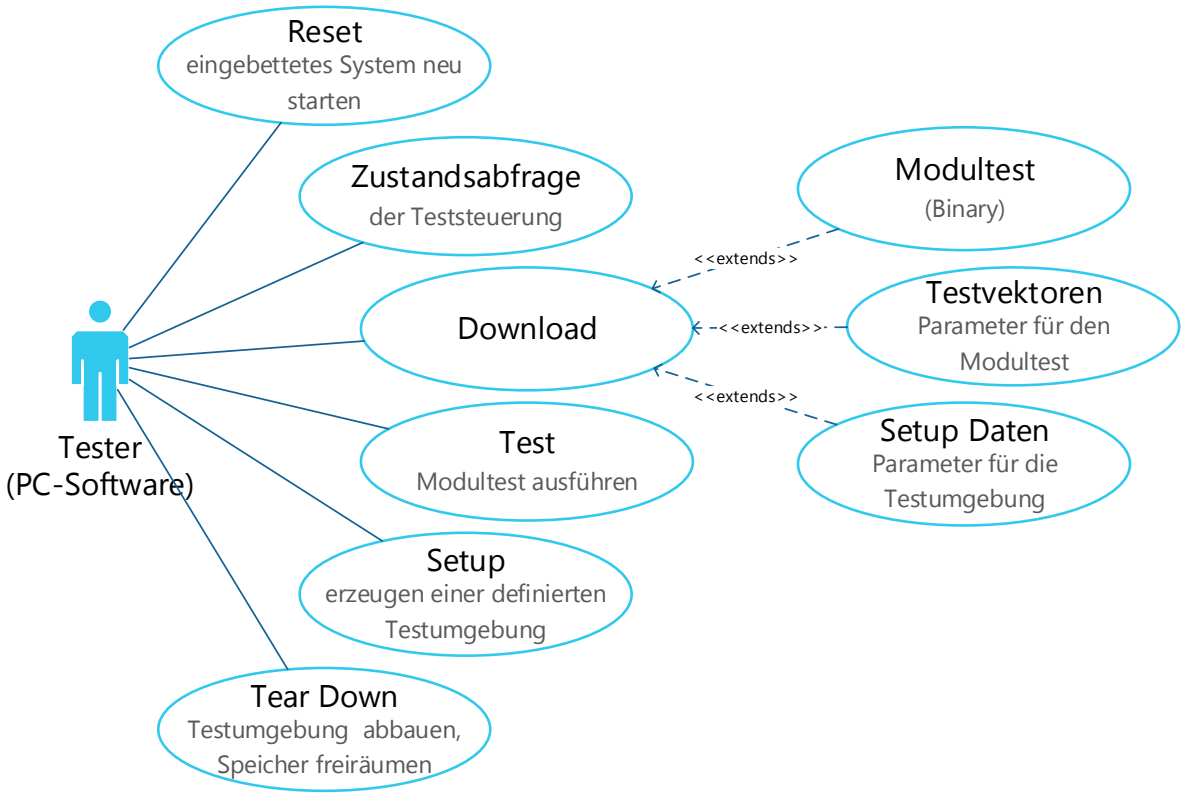

Abb. 4.1.: Aus den funktionalen Anforderungen abgeleiteter Entwurf

4.1.1. Download von Modultest, Testvektoren und Testumgebungsdaten (Setup-Daten)

Um einen Modultest auszuführen, muss dieser zuerst auf das eingebettete System gebracht werden. Vorerst wird dies durch direktes Linken an die Teststeuerung unterstützt. Die Anforderung ist aber, perspektivisch den Test auch zur Laufzeit austauschen zu können, um die automatische Ausführung mehrerer Tests beschleunigen zu können. Darum wird über einen Befehl die Möglichkeit vorgesehen, den Modultest dynamisch nachzuladen (*Programm*). Die Übertragung erfolgt dann nicht mehr über die reguläre Schnittstelle zum Flashen des Controllers, sondern auch über die Verbindung zwischen PC-Software und Testausführungsumgebung.

Weiterhin können Daten für die Einrichtung der Testumgebung (*Setup*) dynamisch übertragen werden. Auch *Testvektoren* können nachgeladen werden. Dafür wird jeweils ein Befehl reserviert. So kann ein Modultest dynamisch zusammengestellt werden und die Übertragungslast wird minimiert.

4.1.2. Teststeuerung mit Setup, Test und Teardown

Die Implementierung der Setup-, Test- und Teardown-Routine erfolgt im Modultest selbst. Allerdings muss der Aufruf dieser Anwendungsfälle korrekt an den Modultest weitergeleitet werden.

Es muss sichergestellt werden, dass die benötigten Daten verfügbar sind, bevor der Anwendungsfall aufgerufen wird. Beispielsweise darf kein Test gestartet werden, wenn noch kein Modultest auf das eingebettete System geladen wurde.

Auch muss die Abfolge der Anwendungsfälle berücksichtigt werden. Es kann z.B. kein Test gestartet werden, wenn vorher nicht das Setup erfolgreich abgeschlossen wurde.

4.1.3. Statusabfrage

Es kann vorkommen, dass die PC-Software den aktuellen Status der Ausführungsumgebung erfragen muss. Für diesen Fall wird ein Statuscode zurückgeliefert, der angibt, ob Modultest, Testvektoren und Setupdaten vorhanden sind und welcher Testschritt (Setup, Test und Teardown) als letztes ausgeführt wurde.

4.1.4. Reset

Ein Reset des eingebetteten Systems ist ebenfalls als Anwendungsfall vorgesehen. Beim Reset wird das System neu gestartet, wobei der RAM-Inhalt verloren geht. Falls die Teardown-Funktion nicht

korrekt implementiert wurde kann es zu nicht gewünschten Folgeerscheinungen bei nachfolgenden Tests kommen. Ein Reset kann in solch einem Fall einen Grundzustand wiederherstellen. So kann ein unbeaufsichtigter Test trotz eines solchen Fehlers eventuell fortgesetzt werden. Die Entscheidung, wann ein Reset durchgeführt wird, liegt bei der PC-Software. Durch den richtigen Einsatz dieses Befehls kann ein unbeaufsichtigter automatischer Ablauf von vielen Modultests nacheinander zuverlässiger durchgeführt werden.

4.2. Technischer Entwurf

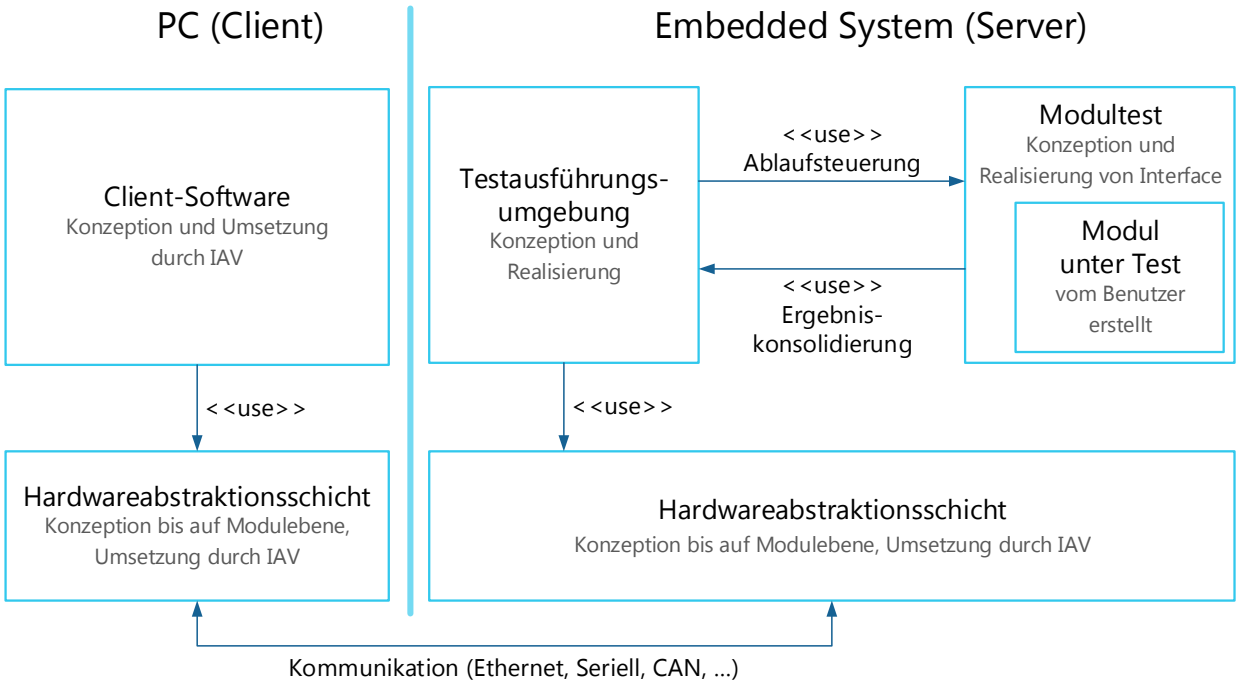

Abb. 4.2.: Übersicht des technischen Entwurfs

Die Zielstellung ist, automatisiert Tests auf eingebetteten Systemen ausführen zu können. Dazu wurde ein Konzept erarbeitet, welches in Abb. 4.2 zu sehen ist. Dieses Konzept beinhaltet auch den Entwurf des Anwendungsprotokolls zwischen Client und Server. Weiterhin ist aus diesem Konzept zu entnehmen, welche Komponenten bis zu welcher Ebene im Umfang dieser Arbeit konzipiert und realisiert werden.

Auf der Seite des Zielsystems wurde dafür bewusst eine Teilung in drei Komponenten gewählt, besonders um die Forderung nach Portabilität zu erfüllen:

Testausführungsumgebung Hierzu gehört die Übertragung von Testvektoren und anderen Daten vom PC auf das eingebettete System und die Ablaufsteuerung des Tests sowie die Konsolidierung und Übertragung der Ergebnisse zum PC. Dieser Teil stellt eine Schnittstelle für die Testdurchführung im Modultest bereit und ist *generisch* implementierbar.

Modultest In diesem Teil ist die Testlogik beschrieben. Da für die Testdurchführung das zu testende Modul bekannt sein muss, ist dieser Teil *modulspezifisch*. Das bedeutet, dass für jedes zu testende Modul mindestens ein Modultest vorhanden sein muss. Ist der Modultest nicht statisch an die Ausführungsumgebung gelinkt, kann er zur Laufzeit der Testausführungsumgebung durch diese ausgetauscht werden.

Hardwareabstraktionsschicht Hier werden von der Testausführungsumgebung und dem Modultest benötigte Systemfunktionen bereitgestellt, wie z.B. Kommunikation über Hardwareschnittstellen, Speicherzugriffe, Timerfunktionen usw. Dieser Teil ist somit *plattformabhängig*.

4.2.1. Vorteile dieses Konzepts

Die Logik der Ausführungsumgebung ist generisch und nur über ein Interface an den Test gekoppelt. Damit ist der Test zur Laufzeit der Ausführungsumgebung austauschbar. Dies ist ein Vorteil dieses Konzepts gegenüber bekannten Testwerkzeugen wie Tessy oder Vectorcast.

Weiterhin sind alle plattformspezifischen Funktionen in einer Hardwareabstraktionsschicht untergebracht. Damit lässt sich derselbe Test auf andere Plattformen portieren, wobei nur die Hardwareabstraktionsschicht ausgetauscht werden muss.

4.2.2. Client-Server-Ansatz

Das Client-Server Modell beschreibt eine Möglichkeit, Tasks in einem Netzwerk zu verteilen. Dabei schickt der Client eine Aufgabe an einen Server, welcher diese mit dem passenden Dienst dafür bearbeiten kann. In gewissem Sinn stellt auch die Testausführungsumgebung den Dienst der Teststeuerung bereit, welcher von einem Client aus mit Aufgaben versorgt werden kann. Daher kann man sagen, dass der Systementwurf einem Client-Server-System angelehnt ist (siehe Abb. 4.2 auf der vorherigen Seite).

4.2.3. Kommunikation

Die Kommunikation erfolgt immer nach dem Prinzip, dass eine Anfrage vom Client an den Server gesendet wird und der Server darauf antwortet. Dieser Datenfluss ist in Abb. 4.3 auf der nächsten Seite beschrieben, unter der Annahme, dass die Nachricht erfolgreich dekodiert und validiert werden kann.

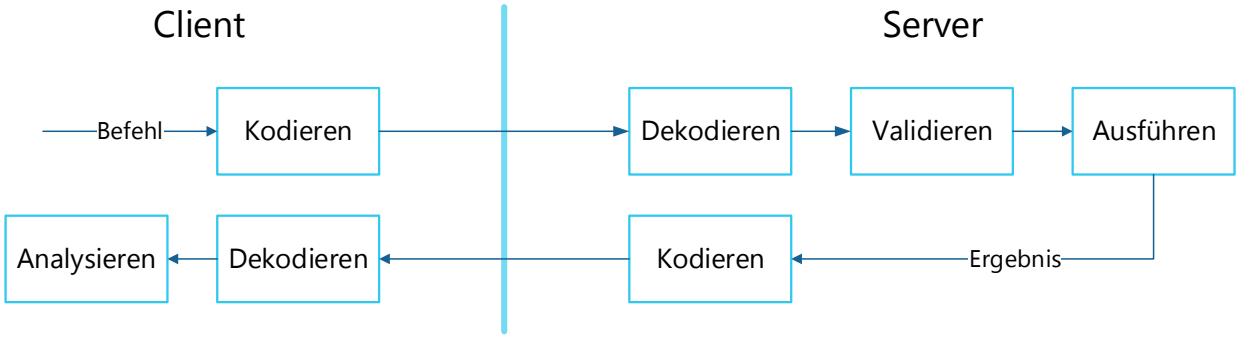

Abb. 4.3.: Datenfluss der Nachrichten zwischen Client und Server

Für die Kommunikation zwischen Server und Client wird ein Protokollstack benötigt. Da die Kommunikation in der Testausführungsumgebung generisch erfolgen soll, d.h. unabhängig von der physischen Schnittstelle, ist ein Teil des Protokollstacks in der HAL zu implementieren. In Tab. 4.1 auf der nächsten Seite wird diese Teilung am OSI-Modell festgemacht. Da man im allgemeinen sehr unterschiedlicher Auffassung ist, welche Protokolle welche Schicht (insbesondere die untersten vier Schichten) abdecken, wurde auf eine genaue Zuteilung verzichtet. Auf jeden Fall decken die aufgeführten Protokolle in den OSI-Schichten 1 bis 4 diese als Gesamtheit vollständig ab. Für jede Plattform muss für jede Schnittstelle, die zur Kommunikation zwischen Client und Testausführungsumgebung zur Verfügung stehen soll, ein eigener Treiber vorhanden sein. Die Protokolle über dem Hardwaretreiber (bis OSI 4) sind zwar nicht mehr hardwareabhängig, unterscheiden sich jedoch zwischen den Schnittstellen. Deshalb wurde der gesamte Protokollstack bis OSI 4 trotzdem in die HAL aufgenommen.

Die Kommunikation in den darüber liegenden Schichten wird von der Testausführungsumgebung realisiert, da sie unabhängig vom eigentlichen Netzwerk sind. Es ist nicht notwendig, die Sitzung gegen Unterbrechungen abzusichern oder mehrere Sitzungen aufzubauen. Daher besteht die einzige Aufgabe der Sitzungsschicht darin, die Sitzung einmalig aufzubauen bzw. auf den Aufbau durch den Client zu warten.

Die Darstellungsschicht oder Präsentationsschicht und die Anwendungsschicht werden mit NanoPB realisiert. NanoPB übernimmt die korrekte Präsentation der Daten, unabhängig von Byte-Order oder Alignment der Plattform. Hauptaufgabe von NanoPB ist jedoch die Anwendungsschicht. Mit NanoPB kann ein Protokoll in sehr geringer Codegröße definiert und implementiert werden.

Dies wird zum einen für den Nachrichtenaustausch zwischen Client und Testausführungsumgebung benötigt. Zum anderen muss auch der Modultest die Setup-Daten und Testvektoren in einem definierten Format empfangen können. Dieses Format ist modultestspezifisch und kann deshalb nicht direkt mit in das eigentliche Anwendungsprotokoll, welches ja in der generischen Testausführungsumgebung liegt, integriert werden. Jedoch erhält der Modultest über das In-

terface zur Testausführungsumgebung Zugriff auf NanoPB, so dass er nur noch die aus der Protokolldefinition generierten Dateien implementieren muss.

<table>
<tr><th>Schicht \ Verbindung</th><th>Ethernet</th><th>CAN</th><th>Seriell</th><th>Ort der Implementierung</th></tr>
<tr><td>Anwendungsschicht</td><td colspan="3">NanoPB (Sub-Nachrichtenauswertung für Testvektoren & Setupdaten)</td><td>Modultest</td></tr>
<tr><td>Anwendungsschicht</td><td colspan="3">NanoPB (Nachrichtenauswertung in der Testausführungsumgebung)</td><td rowspan="3">Test-ausführungs-umgebung</td></tr>
<tr><td>Darstellungsschicht</td><td colspan="3">NanoPB (Byte-Order, Alignment)</td></tr>
<tr><td>Sitzungsschicht</td><td colspan="3">Sitzungshandling in Testausführungsumgebung</td></tr>
<tr><td>Transportschicht</td><td rowspan="4">Ethernet (inkl. CSMA/CD), TCP, IP</td><td rowspan="4">CAN (inkl. CSMA/CR), ISO-TP</td><td rowspan="4">Serielle Direkt-verbindung, SLIP, ISO-TP</td><td rowspan="4">HAL</td></tr>
<tr><td>Vermittlungsschicht</td></tr>
<tr><td>Sicherungsschicht</td></tr>
<tr><td>Physikalische Schicht</td></tr>
</table>

Tab. 4.1.: Kommunikationskonzept im OSI-Modell

Auswahl oder Entwicklung eines Andwendungsprotokolls

Es existiert eine Vielzahl von Anwendungsprotokollen, von denen sich einige für die beschriebene Aufgabe adaptieren lassen würden. Von der IAV wurden UDS (Unified Diagnostic Services, Kommunikationsprotokoll in der Automobiltechnik) und XCP (Universal Measurement and Calibration Protocol, Netzwerkprotokoll in der Automobiltechnik) vorgeschlagen. Andererseits gibt es auch Methoden, um eigene Protokolle zu definieren. Dazu zählt u.a. Protocol Buffer. Als Alternative könnte ein XML (eXtensible Markup Language, erweiterbare Auszeichnungssprache)-Datenstrom zwischen Client und Server ausgetauscht werden. XML ist zwar keine Sprache um Protokolle zu beschreiben, da es nur den Inhalt und nicht die Abfolge der Daten festlegt. Aber für die Übertragung von Nachrichten, wie z.B. bei SOAP (Simple Object Access Protocol, Netzwerkprotokoll zum Datenaustausch und für Remote Procedure Calls) genutzt, ist es gut geeignet.

Ein Vorteil für die Verwendung und Adaptierung eines bestehenden Protokolls wäre die Verringerung der Einarbeitungszeit für Mitarbeiter der IAV, da die Protokolle UDS und XCP schon in vielen Projekten verwendet werden. Demgegenüber stehen aber mehrere schwerwiegende Nachteile:

- Ein eigenes Protokoll bietet mehr Flexibilität als die Adaptierung eines vorhandenen Protokolls
- Bibliotheken zum Enkodieren und Dekodieren sind oft nicht für alle Plattformen verfügbar
- Lizenzrechtliche Gründe verhindern die uneingeschränkte Nutzung oder den Weiterverkauf des Endprodukts
- Vorhandene Protokolle bieten Unterstützung für nicht benötigte Anwendungsfälle und vergrößern somit den Code unnötig und verlangsamen durch den Protokollüberhang die Verbindung.

Aus diesen Gründen wurde auf die Entwicklung eines eigenen Nachrichtenübertragungsformats gesetzt. XML bietet umfangreiche Beschreibungs- und Validierungsmöglichkeiten, kann eine breite Toolunterstützung vorweisen und ist für Menschen direkt lesbar. Besonders aus dem Grund, dass XML einfach lesbar ist, entsteht naturgemäß der Nachteil eines größeren Overheads, wie er z.B. durch schließende Tags hervorgerufen wird. Weiterhin ist typischerweise das XML-Schema als Text vorhanden und wird zur Laufzeit geparst.

Protocol Buffer wählt hier einen anderen Weg. Die Protokollbeschreibung wird bereits vor der Kompilierung in C-Code übersetzt. Für eingebettete Systeme steht eine leichtgewichtige C-Implementierung unter dem Namen NanoPB bereit. Eine Gegenüberstellung von XML und Protocol Buffer ist in Tab. 4.2 auf der nächsten Seite dargestellt. Diese Tabelle soll keinen vollständigen Vergleich aller Merkmale bieten, sondern ist auf den hier geforderten Anwendungsfall zugeschnitten.

Es kann also festgestellt werden, dass XML-Kodierungen in aller Regel einen längeren Bytestrom als die Protocol-Buffer Implementierung NanoPB für dieselben Daten verursacht. Weiterhin wird das XML-Schema zur Laufzeit eingelesen, was zusätzlichen Rechenaufwand benötigt. Außerdem belegt ein XML-Parser im Vergleich zu NanoPB wesentlich mehr Programmspeicher. Daher wurde NanoPB ausgewählt.

Merkmal	XML	Protocol Buffer (NanoPB)
Nachrichtendefinitionsformat	Text (z.B. XML-Schema) wird i.d.R. zur Laufzeit geparst	Text (.proto-Datei) wird vor der Kompilierung in C-Code übersetzt
Datenstrom	Text	Binärdaten
Plattformunterstützung	Breite Unterstützung, es existieren Parser sowohl für PCs als auch für Mikrocontroller	
Plattformunabhängigkeit	Byte-Order oder Alignment müssen bei der Übertragung zwischen verschiedenen Systemen nicht beachtet werden, da XML Text ist. Datentypen werden erst vom Parser ausgewählt.	Die Byte-Order kann über ein *#define* angepasst werden. Das Alignment wird automatisch eingehalten.
Sprachunterstützung	alle gängigen Programmiersprachen werden unterstützt	NanoPB ist in C implementiert, offiziell werden C++, Java und Python unterstützt. Drittanbieter siehe [Xia13]
Verwaltungsdaten	Sehr viel, durch öffnende und schließende Tags und durch die Darstellung jeglicher Information als Text.	gering, da Binärformat und Verwendung von variablem Integerformat *varint* [Goo12, „Base 128 Varints“].
Bibliotheksgröße	ca. 36kB (Mini-XML)	ca. 3kB
Binärdatenkodierung	• Base64-Konvertierung (+33% Speicherbedarf) • XOP (muss vom Parser unterstützt werden) • FastInfoset-Kodierung (kein XML mehr)	Binärdatenfelder werden einfach über Typ und Länge definiert

Tab. 4.2.: Vergleich ausgewählter Merkmale von XML und Google Protocol Buffer (NanoPB-Implementierung)

4.3. Modulkonzept

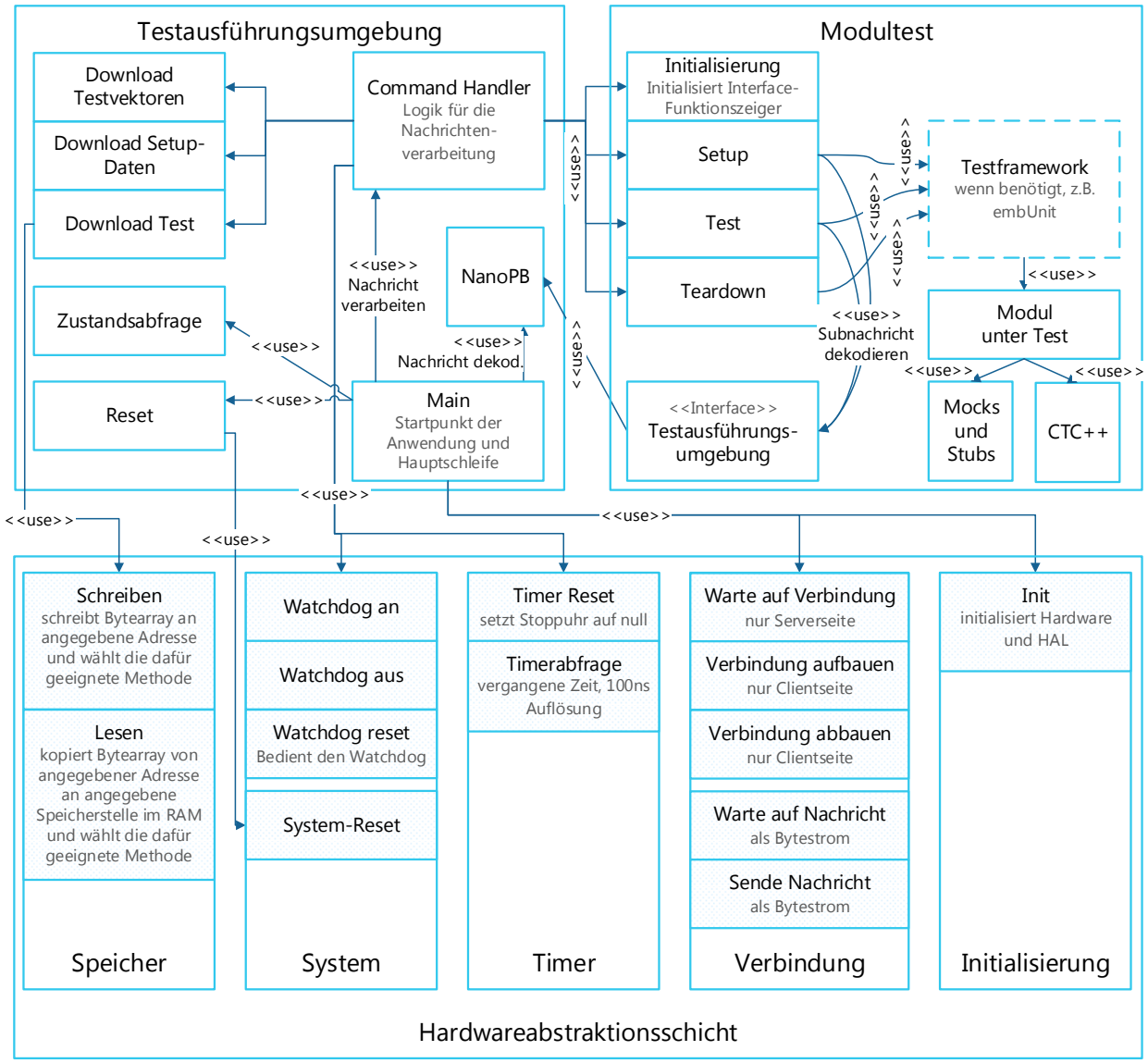

Abb. 4.4.: Modulkonzept eingebettetes System

In Abb. 4.4 sind die Module auf dem eingebetteten System und deren Beziehungen untereinander dargestellt. Die *Hardwareabstraktionsschicht* trennt die Testausführungsumgebung vom verwendeten Zielsystem. Dazu gehört die Kommunikation mit dem Steuer-PC, Zeitfunktionen zum Messen von Laufzeiten, Speicherzugriffe auf RAM und ROM sowie Systemfunktionen, besonders zum automatischen Rücksetzen des Systems bei einem Timeout.

Die *Testausführungsumgebung* enthält das Hauptprogramm (*main*-Funktion), welches beim Gerätestart ausgeführt wird. Es initialisiert zunächst die Hardware und wartet dann auf eine eingehende Verbindung eines Clients. Dazu bedient sich die Testausführungsumgebung Methoden aus der Hardwareabstraktionsschicht.

Danach nimmt die Testausführungsumgebung Befehle in Form von Nachrichten vom Client entgegen. Dazu nutzt sie wiederum die Kommunikationsfunktionen der Hardwareabstraktionsschicht.

Darauf folgend wird geprüft, ob es möglich ist, die Befehle korrekt zu parsen und ob sie im aktuellen Zustand auch ausgeführt werden dürfen. Die Anweisungen zum Download von Daten und die Abfrage des Testablaufzustands werden direkt von der Testausführungsumgebung behandelt. Befehle, die den Testablauf steuern, werden an den Modultest übergeben. Dazu wird eine definierte Schnittstelle genutzt. Zum Systemreset wird die dafür vorgesehene Funktion aus der Hardwareabstraktionsschicht aufgerufen.

Die *Modultest*-Komponente kann größtenteils frei vom Entwickler festgelegt werden. Eine korrekte Implementierung des Interfaces ist durch den Import eines C- und Headerfiles gegeben. Darin wird die Init-Funktion implementiert, welche die Zeiger der im Interface bereitgestellten Funktionen der Testausführungsumgebung in eine Funktionszeigerstruktur schreibt.

4.3.1. Testausführungsumgebung

4.3.1.1. Hauptschleife (main)

Die Testausführungsumgebung enthält die *main*-Funktion, welche beim Gerätestart aufgerufen wird. Dort wird die Initialisierung der Hardware durch die HAL veranlasst. Danach wird auf eine Verbindung gewartet. Dazu wird eine Funktion in der HAL aufgerufen, welche die weitere Ausführung so lange blockiert, bis eine Verbindung aufgebaut wurde.

Danach tritt die Ausführung in die Hauptschleife ein. Darin wird als erstes immer auf eine eingehende Nachricht gewartet. Auch dieser Aufruf ist blockierend. Eingehende Nachrichten werden versucht zu dekodieren. Falls ein Fehler beim Dekodieren auftritt, wird eine Nachricht an den Client zurückgesendet mit der entsprechenden Fehlermeldung. Der aktuelle Zustand des Automaten wird beibehalten.

Ist die Dekodierung erfolgreich gewesen, wird die Nachricht zusammen mit einer Referenz auf eine Ergebnisnachricht an das *Command Handler* Modul übergeben.

Der *Command Handler* produziert zusammen mit seinen abhängigen Modulen das Ergebnis. Dieses muss dann mit NanoPB kodiert werden und an den Client gesendet werden.

Der gesamte Ablauf der Hauptschleife ist in Abb. 4.5 auf der nächsten Seite abgebildet.

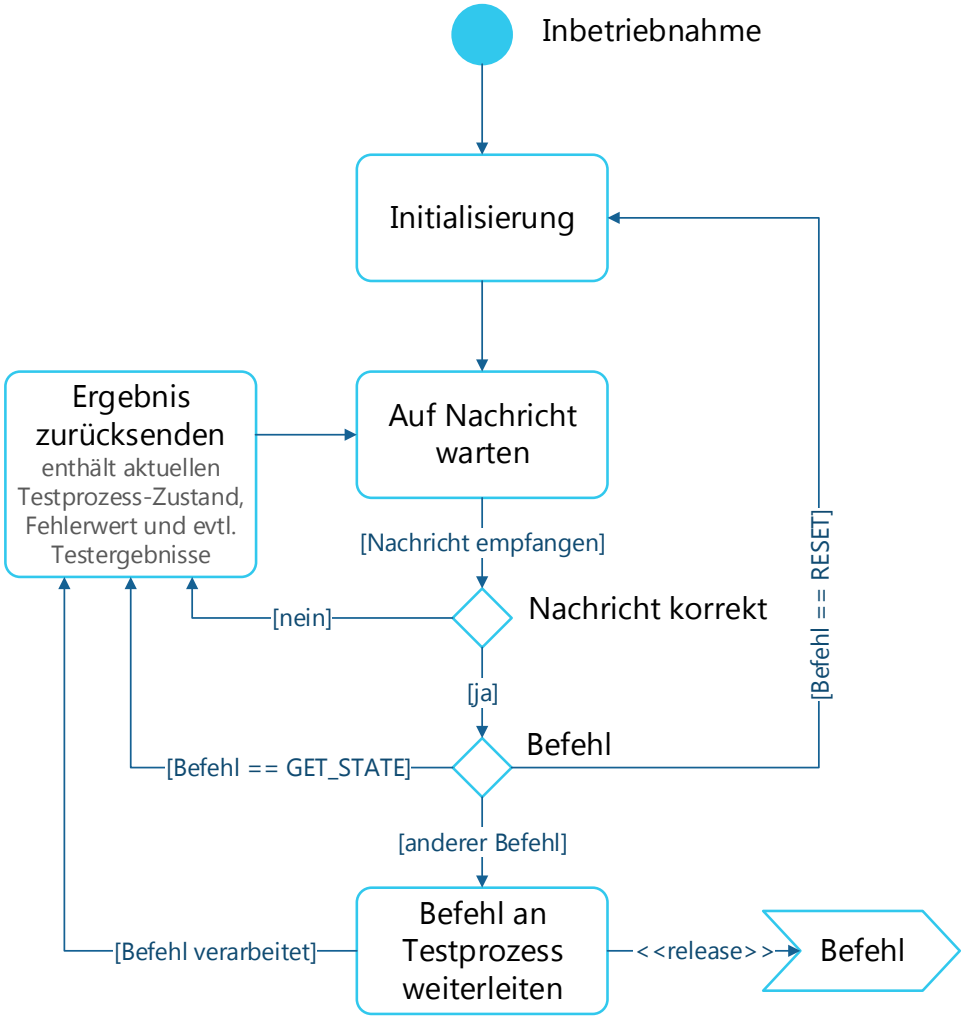

Abb. 4.5.: Zustandsdiagramm der Hauptschleife der Testausführungsumgebung

Wie man in Abb. 4.5 sieht, ist kein Endzustand vorhanden. Das bedeutet, dass die Umsetzung in einer Endlosschleife erfolgen wird. Da Endlosschleifen das Testen erheblich erschweren, wird der Inhalt der Schleife in eine neue Funktion ausgelagert.

4.3.1.2. NanoPB

Die NanoPB-Bibliothek kann als C-Quellcode von `code.google.com/p/nanopb/` heruntergeladen und einfach mit dem Projekt kompiliert werden.

Die Protokolldefinition wird in einer *.proto*-Datei festgehalten. Aus dieser wird mit dem Programm *protoc.exe* von Google und dem Python-Skript *nanopb_generator.py*, welches bei NanoPB mitgeliefert wird, eine Header- und eine C-Datei generiert, welche dann im Projekt eingebunden werden können.

Um die Anwendungsfälle für die PC-Software verfügbar zu machen, werden im Anwendungs-

protokoll Nachrichten definiert, welche diese abbilden. Die benötigten Nachrichten, um eine Anfrage an die Testausführungsumgebung zu senden, werden in Tab. 4.3 aufgeführt. Nachrichten werden dekodiert, indem der Bytestrom in eine C-Struktur konvertiert wird. Diese wird von der NanoPB-Funktion dann zurückgegeben.

War die Dekodierung erfolgreich, wird die Nachricht an das Modul *Command Handler* übergeben.

Nachricht	Daten	Erwartete Antwort
DownloadTest	Modultest und zu testendes Modul im Binärformat	neuer Zustand der Ausführungsumgebung
DownloadSetupData	Daten zur Herstellung der Testumgebung als Binärdaten	neuer Zustand der Ausführungsumgebung
DownloadTestVectors	Testvektoren als Binärdaten	neuer Zustand der Ausführungsumgebung
SetUp	-	neuer Zustand der Ausführungsumgebung
Test	-	Testergebnisse, neuer Zustand der Ausführungsumgebung
TearDown	-	neuer Zustand der Ausführungsumgebung
GetStatus	-	aktueller Zustand der Ausführungsumgebung
Reset	-	-

Tab. 4.3.: Abbildung der Anwendungsfälle auf Nachrichten

4.3.1.3. Command Handler

Der *Command Handler* bekommt von der Hauptschleife die Anfrage des Clients übergeben sowie die Referenz auf eine Ergebnisstruktur, in der Daten, welche zum Client zurückgesendet werden sollen, gespeichert werden. Für das Zusammenstellen der Ergebnisdaten ist der Modultest verantwortlich.

Im *Command Handler* wird zuerst geprüft, ob der in der Nachricht enthaltene Befehl im aktuellen Zustand ausgeführt werden kann. Falls nicht, wird eine Fehlermeldung generiert, die dann in der Hauptschleife an den Client gesendet wird. Darin wird auch der Fehlergrund angegeben. Es

kann so vom Client festgestellt werden, ob der Befehl allgemein im aktuellen Zustand unzulässig ist oder ob nur bestimmte Daten zur korrekten Ausführung fehlen.

Ist der Befehl zulässig, wird die dafür zuständige Funktion aufgerufen, worin der Befehl verarbeitet wird. Welche Befehle in welcher Reihenfolge und unter welchen Voraussetzungen ausgeführt werden dürfen, ist dem Zustandsdiagramm in Abb. 4.6 auf der nächsten Seite zu entnehmen.

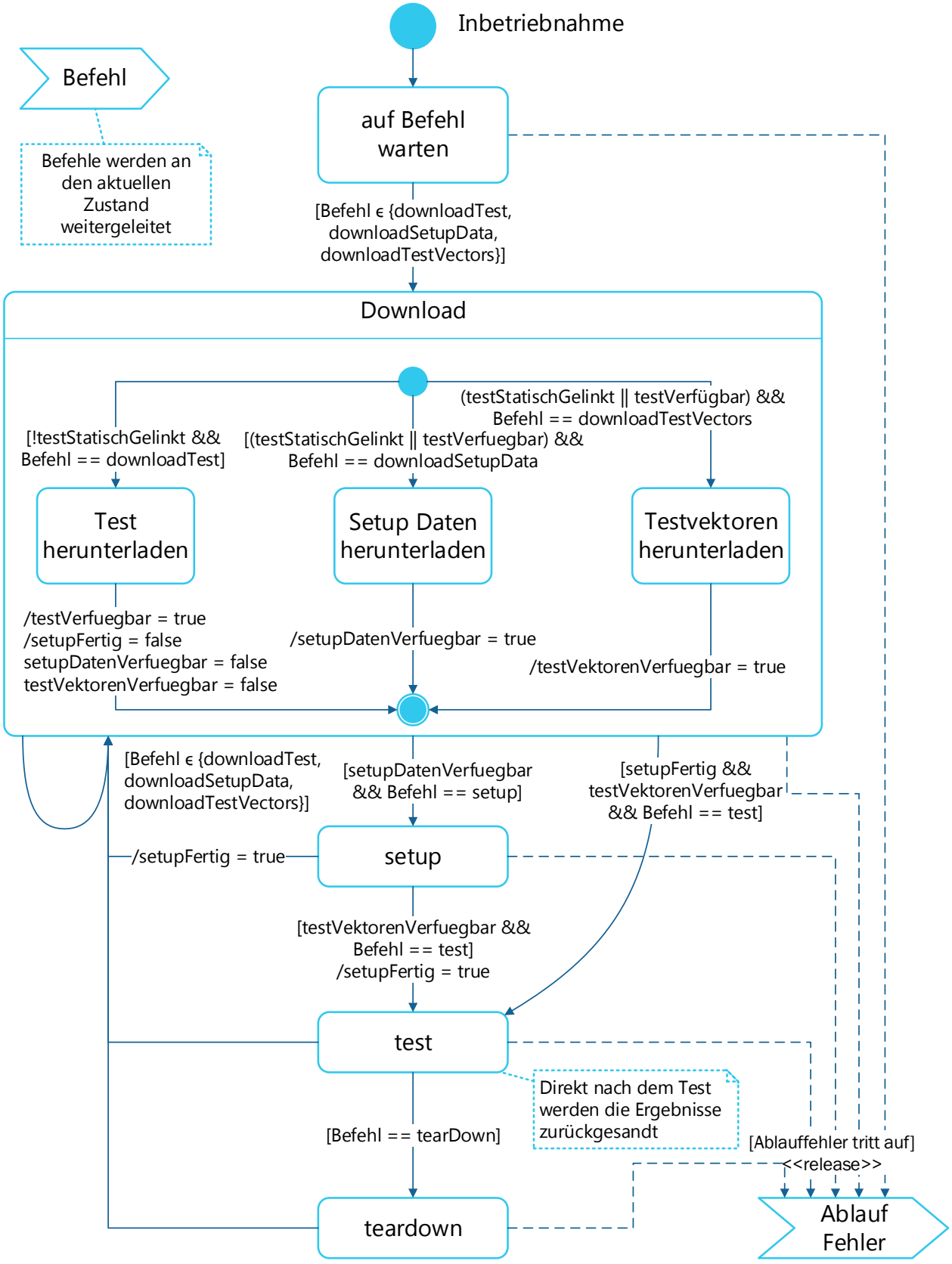

Abb. 4.6.: Zustandsdiagramm der Testablaufsteuerung

4.3.1.4. Download des Modultests

Je nach System gibt es verschiedene Möglichkeiten, von wo aus das Modultestprogramm ausgeführt werden kann. Manche Controller können Programmcode nur aus dem Flash ausführen. Dies betrifft typischerweise Prozessoren in Harvard-Architektur, da hier Programm- und Datenspeicher getrennt sind. Vor allem leistungsfähigere Controller und moderne DSP's (Digitaler Signal Prozessor) können Programme auch aus dem RAM ausführen. Auch wenn die verschiedenen Speichertypen nebeneinander im selben Adressraum liegen, kann nicht davon ausgegangen werden, dass jeder Speicher gleich angesprochen werden kann. So sind z.B. Latenzzeiten zu beachten oder Schreibschutzattribute korrekt zu setzen, um Daten schreiben zu können. Da die Testausführungsumgebung generisch realisiert werden muss, d.h. plattformunabhängig, kann dort keine Logik dafür implementiert werden. Dies muss in der HAL geschehen.

Das Binary des Modultests befindet sich nach dem Empfang der Download-Test Nachricht bereits im Puffer. Von dort kann es durch eine HAL-Funktion an den Ort der Ausführung kopiert werden.

4.3.1.5. Download von Setup-Daten und Testvektoren

Die Setup-Daten und die Testvektoren müssen jeweils aus dem Empfangspuffer in einen dafür vorgesehenen RAM-Bereich kopiert werden, um nicht von nachfolgenden Nachrichten überschrieben zu werden. Eine Referenz auf die RAM-Adresse, in die sie kopiert wurden, wird dann beim Ausführen des Setups oder des Tests als Parameter übergeben.

4.3.1.6. Setup

Mit dem Kommando Setup wird die Setup-Routine aus dem Modultest aufgerufen. Dabei wird eine Referenz auf die vorher heruntergeladenen Setup-Daten übergeben.

4.3.1.7. Test

Das Test-Kommando ruft die Test-Routine im Modultest auf. Es wird eine Referenz auf die Testvektoren, eine Referenz auf das Ergebnisdatenfeld sowie eine Referenz auf den CTC++-Datenpuffer übergeben.

Vor dem Aufruf der Test-Routine wird der Timer zurückgesetzt, damit nach dem Test die Zeit gemessen werden kann, die dafür benötigt wurde. Das Timer-Modul der HAL gibt die Zeit als Integer mit einer Auflösung von 100 ns zurück. Dieser Wert wird in das Ergebnis, welches zum Client zurückgesendet wird, mit aufgeommen.

4.3.1.8. Teardown

Über das Teardown-Kommando kann dem Modultest mitgeteilt werden, dass jetzt nicht mehr benötigter Speicher freigeräumt und die Testumgebung abgebaut werden müssen. Dazu wird die Teardown-Routine im Modultest aufgerufen.

4.3.1.9. Reset

Ein Neustart des eingebetteten Systems kann vom Client durch den *Reset*-Befehl ausgelöst werden. Der Reset-Befehl ist unabhängig vom aktuellen Zustand der Testausführung und wird immer ausgeführt.

4.3.1.10. Zustandsabfrage

Soll kein Befehl auf dem eingebetteten System ausgeführt werden, sondern nur der aktuelle Zustand des Testablaufs abgefragt werden, kann das *GET_STATE*-Kommando verwendet werden. Folgende Zustände existieren:

- Warte auf Befehl
- Modultest-Download
- Testvektoren-Download
- Setupdaten-Download
- Setup
- Test
- Teardown

4.3.2. Modultest

Die geforderte Austauschbarkeit des Modultests bringt einige Herausforderungen mit sich. Eine davon betrifft die Schnittstelle zwischen Testausführungsumgebung und Modultest. Wenn der Modultest direkt an die Testausführungsumgebung gelinkt wurde, sind die Speicherstellen der Funktionen gegenseitig bekannt. Wird der Modultest dagegen dynamisch nachgeladen, müssen die Funktionen, die im Interface offengelegt werden sollen, erst bekannt gemacht werden.

Dies betrifft zum einen die Funktionen im Modultest, welche in der Testausführungsumgebung aufgerufen werden: Init, Setup, Test und Teardown. Dies wird umgesetzt, indem Funktionszeiger auf die entsprechenden Funktionen immer am Anfang in einer festen Reihenfolge im Modultest liegen. Da der Testausführungsumgebung die Startadresse des Modultests bekannt ist sowie

in welcher Reihenfolge die Funktionszeiger dort liegen, kann sie die Funktionen im Modultest aufrufen.

In der anderen Richtung ist es sinnvoll, dem Modultest Zugriff auf die Kodier- und Dekodierfunktionen von NanoPB zu geben. Durch diese Funktionen kann der Modultest den Testvektordatenstream und den Setupdatenstream mit NanoPB verarbeiten, ohne die komplette Bibliothek integrieren zu müssen. Nur das Header- und C-File mit der Protokolldefinition muss in den Modultest selbst eingebunden werden. Um diese Funktionen dem Modultest zu veröffentlichen, werden die Funktionszeiger darauf beim Aufruf der Init-Funktion dem Modultest mitgegeben. Die Init-Funktion wird von der Testausführungsumgebung jedes Mal aufgerufen, wenn ein neuer Modultest geladen wurde oder, falls der Modultest direkt an die Testausführungsumgebung gelinkt wurde, vor Eintritt in die Hauptschleife.

4.3.3. Hardwareabstraktionsschicht

Die Hardwareabstraktionsschicht besteht aus vier Modulen: Speicherzugriff, Systemfunktionen, Timermodul und Kommunikationsmodul. Außerdem gibt es eine Funktion zur Initialisierung.

4.3.3.1. Initialisierung

Da die HAL die einzige Komponente ist, die plattformabhängigen Code enthält, ist auch hier der Code zur Initialisierung der Hardware des eingebetteten Systems enthalten. Außerdem dient diese Funktion dazu, die anderen Komponenten der HAL zu initialisieren.

4.3.3.2. Timer

Mit den Timermodul ist es u.a. möglich, die Dauer eines Tests zu bestimmen. Dazu wird vor dem Test eine Funktion aufgerufen, um den Timer zurückzusetzen. Nach dem Test wird eine zweite Funktion aufgerufen, welche die vergangene Zeit seit dem Zurücksetzen als Integer in einer Auflösung von 100 ns zurückgibt. Es ist natürlich nicht auf jedem System möglich, Zeitwerte in dieser Auflösung zu erhalten. Hat der Systemzeitgeber nur eine geringere Auflösung, muss der Wert skaliert werden, um trotzdem die Wertigkeit von 100 ns pro Einheit zu erhalten. Jedoch gibt es auch sehr schnelle Systeme, auf denen evtl. sehr kurze Tests ausgeführt werden. Daher ist die Wahl von einer 100 ns-Auflösung angebracht.

4.3.3.3. Speicherzugriff

Der zur Zeit einzige Zweck dieser Funktion ist, den Modultest dynamisch austauschen zu können. Je nachdem, ob für den Programmcode EEPROM, RAM, Flash oder externer Speicher zur Verfügung steht und auch welche Plattform verwendet wird, muss auf verschiedene

Beschränkungen geachtet werden. Z.B. kommt es bei Flashspeichern vor, dass nur ganze Blöcke auf einmal geschrieben werden können. Auch muss in manchen Controllern das Beschreiben des Programmspeichers erst freigegeben werden oder durch eine spezielle Befehlskombination bestätigt werden. Weiterhin müssen Latenzzeiten beachtet werden, da oft der Prozessor schneller ist, als Flash oder EEPROM geschrieben werden können. Das Speicherzugriffsmodul verbirgt diese Problematiken vor der Testausführungsumgebung.

Die Funktion für den Schreibzugriff bekommt als Übergabeparameter den Ort der Quelldaten im RAM, die Zieladresse und die Länge der Daten. Anhand der Zieladresse entscheidet die Funktion eigenständig, in welchen Speicher kopiert wird und welche Methode dafür zu nutzen ist. Auch die Funktion zum Lesen aus dem Speicher hat drei Übergabeparameter: Die Adresse, von der gelesen wird (in einem beliebigen Speicher), die Zieladresse im RAM und die Länge der Daten.

4.3.3.4. Systemfunktionen

Die Funktion Reset sorgt dafür, dass das eingebettete System reinitialisiert wird. Das bedeutet, dass nach dem Neustart des Geräts ein definierter Grundzustand vorliegt.

Ferner wird eine automatische Reset-Funktion in der Art eines Watchdogs implementiert. Der Watchdog kann über eine Funktion aktiviert werden, wobei eine Zeitangabe in Millisekunden mit übergeben wird, nach welcher er auslösen soll. Eine weitere Funktion zum Deaktivieren des Watchdogs wird ebenfalls vorgesehen sowie natürlich eine Routine zum Bedienen des Watchdogs.

Der Watchdog läuft wie ein Countdown. Beim Aktivieren und beim Bedienen wird der Zähler wieder auf die Ausgangszeit zurückgesetzt. Läuft der Zähler auf Null, wird ein Gerätereset ausgelöst. Sollte ein Controller keinen Watchdog implementiert haben oder eine Variante, mit dem sich die hier beschriebene Funktionalität nicht umsetzen lassen würde, kann dafür auch ein Timer-Interrupt genutzt werden. Auf dem PC wird dazu eine Betriebssystem-Funktion aufgerufen, welche den Prozess der Testausführungsumgebung beendet und neu startet.

4.3.3.5. Kommunikation

Wie schon in Abschnitt 4.2.3 auf Seite 23 erwähnt, ist die HAL für die Bereitstellung der untersten vier OSI-Schichten verantwortlich. Folgende Funktionen werden vom Kommunikationsmodul bereitgestellt:

Auf eine Verbindung warten Dies ist ein blockierender Aufruf, bei dem keine Parameter übergeben werden. Diese Funktion wartet auf eine eingehende Verbindung. Bei einer TCP/IP-Verbindung über Ethernet auf einem Windows-PC könnte dazu beispielsweise die *bind*-Funktion der Winsock2-Bibliothek genutzt werden. Ist die Verbindung aufgebaut, speichert

die Funktion intern die Referenz auf den Socket. Danach übergibt sie die Ausführung zurück an den Aufrufer.

Auf den Empfang einer Nachricht warten Dies ist ebenfalls ein blockierender Aufruf. Die Funktion muss wissen, wohin sie den empfangenen Datenstrom speichern soll und wie lang dieser maximal sein darf. Daher bekommt sie als Übergabeparameter die Adresse auf einen Puffer und die Länge des Puffers übergeben. Zurückliefern muss die Funktion die Anzahl der empfangenen Bytes. Geht man von der oben genannten Verbindung (TCP/IP über Ethernet unter MS Windows) aus, kann dazu die Funktion *recv* aus der Winsock2-Bibliothek genutzt werden. Dabei ist zu beachten, dass die Funktion selbstständig prüfen muss, ob die Verbindung noch verfügbar ist, und ggf. Maßnahmen unternehmen muss, um sie wieder herzustellen.

Eine Nachricht senden Sendet eine Nachricht an den Client und gibt die Ausführung zurück an den Aufrufer. Dazu benötigt die Funktion die Adresse der Daten im RAM und die Länge der zu sendenden Daten. Die Funktion muss selbstständig prüfen, ob die Verbindung noch verfügbar ist, und ggf. Maßnahmen unternehmen, um sie wieder herzustellen.

Durch die wechselseitig blockierenden Aufrufe wird die Kommunikation synchronisiert. Das bedeutet, dass es nicht vorkommen kann, dass Client und Server gleichzeitig senden oder empfangen möchten.

In der IAV werden hauptsächlich die drei Schnittstellen Ethernet, CAN und Seriell verwendet. Welcher Protokollstack für die einzelnen Schnittstellen vorgeschlagen wird, ist bereits in Tab. 4.1 auf Seite 25 aufgeführt worden.

5. Realisierung der Ausführungsumgebung

Um die Software zu programmieren, ist eine Entwicklungsumgebung notwendig. Daher wird im ersten Punkt auf die Auswahl einer geeigneten Entwicklungsumgebung und einer Toolkette eingegangen. Danach wird die Realisierung der einzelnen Module der Ausführungsumgebung beschrieben.

5.1. Entwicklungsumgebung

Die Entwicklungsumgebung besteht aus der Toolkette, die zur Entwicklung einer Software benutzt wird. Dies umfasst typischerweise folgende Teile:

- Projekt- und Quellcodeverwaltung
- Quellcodeeditor, meist programmiersprachenbezogen mit Zusatzfunktionen wie Syntax-Hervorhebung, Autovervollständigung, Refactoring, u.v.m.
- Buildsystem
- Compiler und/oder Interpreter
- Linker
- Debugger

Die Auswahl einer passenden Entwicklungsumgebung ist wichtig. Eine gute Entwicklungsumgebung automatisiert ständig wiederkehrende Aufgaben und unterstützt den Programmierer. Auch sollte sie sich flexibel konfigurieren lassen, besonders wenn die zu entwickelnde Software für viele Plattformen verfügbar sein soll, denn dann müssen Teile der Entwicklungsumgebung ausgetauscht werden können.

Eclipse ist eine kostenlose integrierte Entwicklungsumgebung. Integriert deshalb, weil sie die Toolkette verwaltet, den Buildprozess automatisiert und Programmieren, Erstellen und Debuggen in einer Oberfläche vereint. In Eclipse kann die Toolkette angepasst werden, so dass nahezu jeder Compiler, Interpreter, Linker oder Debugger genutzt werden kann. Eclipse hat sich in Unternehmen weit verbreitet und wird so auch bei der IAV fast ausschließlich als IDE (Integrated Development Environment, integrierte Entwicklungsumgebung) für C und C++ Programmierung eingesetzt. Diese Version nennt sich Eclipse CDT (C/C++ Development Tooling).

Allerdings werden in Eclipse von Hause aus nur wenige Compiler, Assembler und Linker unterstützt. Die Testausführungsumgebung soll aber auf vielen eingebettenen Systemen eingesetzt werden. Viele eingebettete Systeme bringen eigene Cross-Compiler und -Linker mit. Um diesem Umstand gerecht zu werden, führte die IAV Scons als Buildumgebung ein. Scons ist komplett in Python geschrieben und auch die Konfiguration wird in Python erstellt. Somit kann Scons einfach zusammen mit dem Projekt und dem Quellcode verteilt werden. Scons kann auch über das Pydef-Plugin in Eclipse integriert und so die Python-Konfiguration gedebuggt werden.

Scons hat gegenüber der integrierten Eclipse-Buildumgebung aber auch Nachteile. Der Programmierer muss Python beherrschen und sich mit der Nutzung und dem Aufbau der Projektkonfiguration in Scons vertraut machen. Ein Entwickler, der vorher noch nichts mit Python zu tun hatte, benötigt hierfür eine lange Einarbeitungszeit. Ein weiterer Nachteil ist, dass schon für kleine, individuelle Erweiterungen im Buildprozess relativ viel Code geschrieben werden muss. Trotzdem wurde von der IAV, aufgrund des Vorteils der breiten Buildtoolunterstützung, Scons für dieses Projekt vorausgesetzt.

In Abb. 5.1 ist die in diesem Projekt verwendete Entwicklungsumgebung schematisch dargestellt.

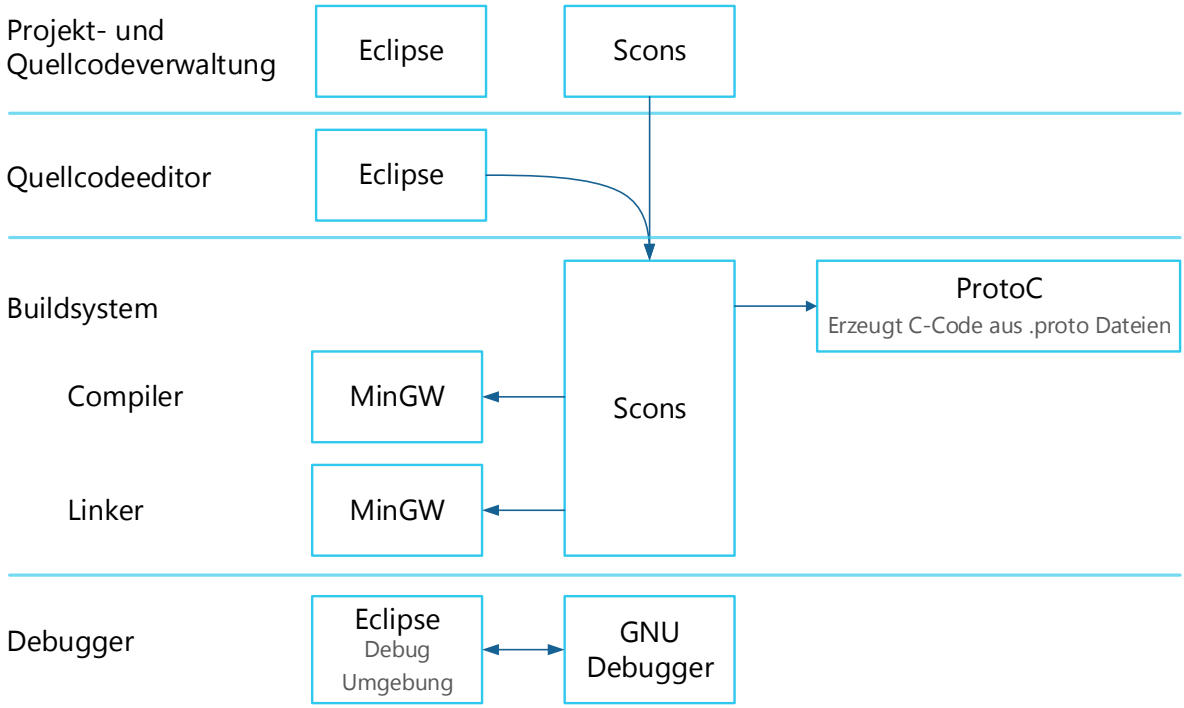

Abb. 5.1.: Übersichtsdarstellung der verwendeten Entwicklungsumgebung

5.2. main-Funktion und Nachrichtenschleife

Die Hauptschleife ist in der main-Funktion enthalten. Bevor in die Hauptschleife gewechselt wird, werden noch die Hardware und die Komponenten der HAL initialisiert, was durch den

Aufruf von *fnHal_Init* geschieht. Danach wird die HAL angewiesen, auf eine Verbindung zu warten (siehe Listing 5.1).

Listing 5.1: main: Initialisierung und Verbindungsaufbau

```
1   fnHal_Init();
2   fnHalComm_WaitForConnection();
```

Nach der Initialisierung erfolgt der Eintritt in die Hauptschleife, deren Code in Listing 5.2 gezeigt wird. Dort wird jeweils die Funktion *fnMessageLoop* aufgerufen. Der Inhalt wurde von der Endlosschleife in diese Funktion ausgelagert, um den Code testen zu können.

Listing 5.2: main: Hauptschleife

```
1  while (1)
2  {
3      fnMessageLoop();
4  }
```

In Listing 5.3 ist die Funktion *fnMessageLoop* dargestellt.

Listing 5.3: fnMessageLoop

```
1  void fnMessageLoop()
2  {
3    static TEE_STATES states = {TEE_STATE_WAIT_FOR_COMMAND,
       TEE_STATE_DOWNLOAD_TEST, TEST_LINKED_TO_TEE, false, false,
       false, false};
4    Request request;
5    Result result;
6    size_t szByteCount = MAXBUFSIZE;
7    Result_Type eResultType;
8
9    fnHalComm_ReceiveData(&ucBuffer[0],MAXBUFSIZE,&szByteCount);
10
11   eResultType = fnDecodeRequest(&ucBuffer[0], &request);
12   if (Result_Type_OK == eResultType)
13   {
14     result.type = fnTee_HandleCommand(&request, &result, &
         states);
15   }
16   else
17   {
```

```
18      result.type = eResultType;
19    }
20    szByteCount = fnEncodeResult(&result,&ucBuffer[0]);
21
22    fnHalComm_SendByteArray(ucBuffer,szByteCount);
23  }
```

Die *states*-Struktur wurde hier als statische Variable angelegt und an den Command Handler übergeben. Das bietet die Möglichkeit, in einem Modultest für den Command Handler dessen Zustand zu manipulieren. Der Command Handler hat dann keine inneren Zustände mehr, was den Aufwand zum Herstellen der Testumgebung für einen Testfall drastisch reduziert.

Mit *fnHalComm_ReceiveData* wird auf eine Nachricht vom Client gewartet, die dann in *ucBuffer* gespeichert wird. Die Länge der Nachricht kann aus *szByteCount* entnommen werden.

In Zeile 11 wird über die Funktion *fnDecodeRequest* versucht, die Nachricht zu dekodieren. Der Funktion wird dabei eine Referenz auf den Eingabedatenstrom *ucBuffer* und auf eine Request-Struktur übergeben. Die Request-Struktur wurde von NanoPB aus der Protokolldefinition generiert. Ist die Dekodierung erfolgreich, wird die Nachricht an die Command Handler Komponente übergeben (Zeile 14). Diese erhält dazu eine Referenz auf eine Result-Struktur, in der das Ergebnis des Requests gespeichert wird (zum Aufbau siehe Tab. 5.2 auf Seite 52). Weiterhin wird eine Referenz der *states*-Struktur auf den Zustand des Command Handlers übergeben.

Falls NanoPB den Eingangsdatenstrom nicht korrekt kodieren kann, wird im Result-Feld das Ergebnis *Result_Type_MESSAGE_PARSING_FAILED* hinterlegt.

In Zeile 20 wird das Ergebnis dann von NanoPB kodiert. Es ist sehr unwahrscheinlich, dass hier ein Fehler auftritt. Eventuell könnte jedoch ein Hinausschreiben über die Datenfelder der Result-Struktur zu einem Fehler führen. Bei einem Fehlerfall gibt die Methode *fnEncodeResult* als Anzahl der kodierten Bytes 0 zurück.

Der kodierte Datenstrom in *ucBuffer* wird dann mit *fnHalComm_SendByteArray* durch die HAL zurück an den Client gesendet.

5.3. Command Handler

Der Command Handler erhält mit der Request-Struktur den Typ der Anfrage übermittelt. Dabei wird zwischen den folgenden Anfragen unterschieden:

- Request_Type_GET_STATE
- Request_Type_RESET

- Request_Type_DL_TEST
- Request_Type_DL_TEST_VECTORS
- Request_Type_DL_SETUP_DATA
- Request_Type_SETUP
- Request_Type_TEST
- Request_Type_TEARDOWN

Wie aus dem Zustandsdiagramm in Abb. 4.6 auf Seite 33 entnommen werden kann, sind die ersten fünf Befehle unabhängig vom aktuellen Zustand der Testausführungsumgebung. Deshalb können diese Befehle ohne Rücksicht auf den Inhalt der *states*-Struktur ausgeführt werden. Im Listing 5.4 ist die Implementierung dieser Befehle zu sehen.

In Zeile 2 wird die *GET_STATE* Anfrage „behandelt“. Hier gibt es nichts zu tun, da der aktuelle Status am Ende der Funktion ohnehin in die *Result*-Struktur geschrieben wird, völlig unabhängig von der Anfrage.

Zeile 4 trifft bei einer Reset-Anfrage zu. Damit wird die Funktion *fnTee_Reset* aufgerufen. Diese ist nur ein Wrapper für die HAL-Funktion *fnHalSystem_ResetDevice()*.

Die Download-Nachrichten werden in den Zeilen 7 bis 9 gefiltert und in Zeile 10 an den Substate-Handler *fn_Tee_SwitchToDownloadState* weitergeleitet (siehe Download-Handler in Abschnitt 5.3 auf Seite 46).

Listing 5.4: fnTee_HandleCommand: unabhängige Befehle

```
1  switch (pRequest->type) {
2  case Request_Type_GET_STATE:
3    break;
4  case Request_Type_RESET:
5    errVal = fnTee_Reset();
6    break;
7  case Request_Type_DL_TEST:
8  case Request_Type_DL_SETUP_DATA:
9  case Request_Type_DL_TEST_VECTORS:
10   errVal = fnTee_SwitchToDownloadState(pRequest,&stSetupData,&
       stTestVectors,states);
11   break;
```

Im *default*-Zweig des *switch-case*-Konstrukts werden die weiteren Befehle behandelt (siehe Listing 5.5 auf der nächsten Seite). Da diese jedoch vom aktuellen Zustand abhängig sind, wird zuerst nach diesem unterschieden. Dies geschieht durch das *switch-case*-Konstrukt, beginnend in Zeile 2.

Vom Zustand *TEE_STATE_WAIT_FOR_COMMAND* kann nur in einen der Download-Zustände gewechselt werden. Da der Übergang zum Download-Zustand schon vorher behandelt wurde, wird in Zeile 4 immer ein Fehler festgestellt.

Vom Download-Zustand kann in den Setup- und Testzustand gewechselt werden, vorausgesetzt die Randbedingungen stimmen. Diese Logik ist in den Zeilen 6 bis 40 ausgedrückt. Dabei werden auch die Flags in der *states*-Struktur aktualisiert.

In den Zeilen 41 bis 67 sind die Übergänge Setup nach Test und Test nach Teardown kodiert. Für andere Übergänge wird immer ein Fehler produziert.

Ist der Automat aktuell im Zustand Teardown (*TEE_STATE_TEARDOWN*), kann nur in einen der Download-Zustände gewechselt werden. Daher wird in Zeile 69 immer ein Fehler produziert.

Listing 5.5: fnTee_HandleCommand: abhängige Befehle

```
1  default:
2    switch (states->eCurrentState) {
3    case TEE_STATE_WAIT_FOR_COMMAND:
4      errVal = Result_Type_INVALID_COMMAND_IN_THIS_STATE;
5      break;
6    case TEE_STATE_DOWNLOAD:
7      switch (pRequest->type) {
8      /* DOWNLOAD -> SETUP */
9      case Request_Type_SETUP:
10       if (states->bSetupDataAvailable == true) /* only start
             setup if setup data are available */
11       {
12         errVal = fnTee_Setup(&stSetupData,pRequest->timeout_sec
               );
13         if (errVal == Result_Type_OK)
14         {
15           states->eCurrentState = TEE_STATE_SETUP;
16           states->bSetupReady = true;
17         }
18       }
19       else
20       {
21         errVal = Result_Type_SETUP_DATA_NOT_AVAILABLE;
22       }
23       break;
24     /* DOWNLOAD -> TEST */
```

```
25      case Request_Type_TEST:
26        if (states->bSetupReady == false) {
27          errVal = Result_Type_SETUP_NOT_READY;
28        }
29        else if (states->bTestVectorsAvailable == false) {
30          errVal = Result_Type_TEST_VECTORS_NOT_AVAILABLE;
31        }
32        else {
33          errVal = fnTee_Test(&stTestVectors,pRequest->
              timeout_sec,&(pResult->result_data),&(pResult->
              ctcpp_data),&(pResult->time100ns));
34          if (errVal == Result_Type_OK) states->eCurrentState =
              TEE_STATE_TEST;
35        }
36        break;
37      default:
38        errVal = Result_Type_INVALID_COMMAND_IN_THIS_STATE;
39      }
40      break;
41    case TEE_STATE_SETUP:
42      switch (pRequest->type) {
43      /* SETUP -> TEST */
44      case Request_Type_TEST:
45        if (states->bTestVectorsAvailable == false) {
46          errVal = Result_Type_TEST_VECTORS_NOT_AVAILABLE;
47        }
48        else {
49          errVal = fnTee_Test(&stTestVectors,pRequest->
              timeout_sec,&(pResult->result_data),&(pResult->
              ctcpp_data),&(pResult->time100ns));
50          if (errVal == Result_Type_OK) states->eCurrentState =
              TEE_STATE_TEST;
51        }
52        break;
53      default:
54        errVal = Result_Type_INVALID_COMMAND_IN_THIS_STATE;
55      }
56      break;
57    case TEE_STATE_TEST:
58      switch (pRequest->type) {
59      /* TEST -> TEARDOWN */
```

```
60      case Request_Type_TEARDOWN:
61        errVal = fnTee_TearDown(pRequest->timeout_sec);
62        if (errVal == Result_Type_OK) states->eCurrentState =
              TEE_STATE_TEARDOWN;
63        break;
64      default:
65        errVal = Result_Type_INVALID_COMMAND_IN_THIS_STATE;
66      }
67      break;
68    case TEE_STATE_TEARDOWN:
69        errVal = Result_Type_INVALID_COMMAND_IN_THIS_STATE;
70      break;
71    }
72    break;
```

Download-Handler

Die Implementierung der Download-Zustände im Subzustandsdiagramm Download erfolgt in der Funktion *fnTee_SwitchToDownloadState*, welche in Listing 5.6 zu sehen ist.

Im switch-Konstrukt ab Zeile 4 wird zwischen den Download-Befehlen unterschieden.

Der Fall in Zeile 6 trifft zu, wenn ein neuer Test heruntergeladen werden soll. Zuerst wird geprüft, ob in diesen Zustand überhaupt gewechselt werden kann. Wenn der Test bereits an die Testausführungsumgebung gelinkt wurde, bricht die Ausführung ab mit der Meldung *Result_Type_TEST_IS_ALREADY_LINKED*. Auch wenn im Request keine Adresse mitgesendet wurde, wird eine Fehlermeldung *Result_Type_NO_TEST_ADDRESS_AVAILABLE* produziert. Sind alle Anforderungen erfüllt, wird der Test in der Funktion *fnTee_DownloadTest* (Zeile 17) heruntergeladen. Ist dies erfolgreich geschehen, werden die entsprechenden Flags (Zeilen 20 bis 23, vergleiche Abb. 4.6 auf Seite 33) und der neue Zustand (Zeilen 24 bis 25) gesetzt.

Die Behandlung für den Download von Setupdaten (Zeile 29) und Testvektoren (Zeile 46) erfolgt ganz ähnlich. Zu Beginn wird jeweils geprüft, ob in den gewünschten Zustand gewechselt werden kann. Dies wird nur verweigert, wenn kein Test verfügbar ist. Danach wird der Befehl ausgeführt, indem die entsprechenden Funktionen aufgerufen werden (Zeilen 36 und 53). Wurde der Befehl erfolgreich ausgeführt, werden die entsprechenden Flags (siehe Abb. 4.6 auf Seite 33) und der neue Zustand gesetzt.

Listing 5.6: fnTee_SwitchToDownloadState

```
1 Result_Type fnTee_SwitchToDownloadState(Request *pRequest,
    Request_data_t * pstSetupData, Request_data_t *
```

```
    pstTestVectors, TEE_STATES *states)
{
  Result_Type errVal = Result_Type_OK;
  switch (pRequest->type)
  {
    case Request_Type_DL_TEST:
      if (states->bTestAlreadyLinked)
      {
        errVal = Result_Type_TEST_IS_ALREADY_LINKED;
      }
      else if (pRequest->has_addr != true)
      {
        errVal = Result_Type_NO_TEST_ADDRESS_AVAILABLE;
      }
      else
      {
        errVal = fnTee_DownloadTest(pRequest);
        if (errVal == Result_Type_OK)
        {
          states->bTestAvailable = true;
          states->bSetupReady = false;
          states->bSetupDataAvailable = false;
          states->bTestVectorsAvailable = false;
          states->eCurrentState = TEE_STATE_DOWNLOAD;
          states->eDownloadState = TEE_STATE_DOWNLOAD_TEST;
        }
      }
      break;
    case Request_Type_DL_SETUP_DATA:
      if (!states->bTestAlreadyLinked && !states->
          bTestAvailable)
      {
        errVal = Result_Type_TEST_NOT_AVAILABLE;
      }
      else
      {
        errVal = fnTee_DownloadSetupData(pRequest,
            pstSetupData);
        if (errVal == Result_Type_OK)
        {
          states->bSetupDataAvailable = true;
```

```
40            states->bSetupReady = false;
41            states->eCurrentState = TEE_STATE_DOWNLOAD;
42            states->eDownloadState =
                 TEE_STATE_DOWNLOAD_SETUPDATA;
43          }
44        }
45        break;
46      case Request_Type_DL_TEST_VECTORS:
47        if (!states->bTestAlreadyLinked && !states->
             bTestAvailable)
48        {
49          errVal = Result_Type_TEST_NOT_AVAILABLE;
50        }
51        else
52        {
53          errVal = fnTee_DownloadTestVectors(pRequest,
               pstTestVectors);
54          if (errVal == Result_Type_OK)
55          {
56            states->bTestVectorsAvailable = true;
57            states->eCurrentState = TEE_STATE_DOWNLOAD;
58            states->eDownloadState =
                 TEE_STATE_DOWNLOAD_TESTVECTORS;
59          }
60        }
61        break;
62      default:
63        /* internal processing error, should never happen */
64        break;
65    }
66    return errVal;
67  }
```

5.4. Kommunikation

5.4.1. NanoPB

Die Implementierung von NanoPB besteht aus sechs statischen Quelldateien, die nachfolgend mit ihrer Funktion aufgelistet sind:

pb_encode.h, pb_encode.c	Methoden, um eine Nachricht aus einer C-Struktur in einen NanoPB-Binärstream zu kodieren, der dann übertragen werden kann.
pb_decode.h, pb_decode.c	Methoden, um eine Nachricht aus einem NanoPB-Binärstream zu dekodieren. Auf die Nachricht kann dann über eine C-Struktur wie üblich zugegriffen werden.
pb.h	Definitionen, die sowohl zum Enkodieren als auch zum Dekodieren benötigt werden.
stdbool.h	Definiert den Typ bool.

Die für eine einfache Kodierung oder Dekodierung verwendeten Funktionen sind:

pb_decode	Dekodiert eine Nachricht von einem NanoPB-Stream (*pb_istream_t*) unter Verwendung der *pb_field_t*-Konstante der Nachricht (aus Protokolldefinition generiert) in eine Struktur (ebenfalls aus Protokolldefinition generiert).
pb_encode	Kodiert eine Nachricht von einer Struktur (aus Protokolldefinition generiert) in einen NanoPB-Stream (*pb_ostream_t*) unter Verwendung der *pb_field_t*-Konstante der Nachricht (ebenfalls aus Protokolldefinition generiert).
pb_istream_from_buffer	Erzeugt einen NanoPB-Stream, mit welchem Daten aus einem angegebenen char-Puffer gelesen werden können.
pb_ostream_from_buffer	mit welchem Daten in einen angegebenen char-Puffer geschrieben werden können.

5.4.2. Protokolldefinition

Bei der Protokolldefinition wird unterschieden zwischen Anfragen (Requests), welche der Client an den Server richtet, und Ergebnissen (Results), welche der Server zurücksendet.

Die Protokolldefinition wird in einem *.proto*-File festgehalten (siehe Abschnitt A.1 auf Seite A1), aus welchem dann folgende Dateien generiert werden:

[Protokollbezeichnung].pb.h	Enthält Struktur-Typdefinitionen passend zur Protokollbeschreibung.
[Protokollbezeichnung].pb.c	Enthält Konstanten, welche Informationen über jedes Feld der Protokollbeschreibung enthalten. Dies sind z.B. Typ, Größe und ob es sich um ein Pflichtfeld oder ein optionales Feld handelt.

Die Protokolldefinition für Nachrichten vom Client an den Server (Request) ist in Tab. 5.1 auf der nächsten Seite dargestellt, die für Nachrichten zurück an den Client (Result) in Tab. 5.2 auf Seite 52.

Feld	optional	Beschreibung
type	nein	Legt den Befehl fest, d.h. was mit dieser Nachricht erreicht werden soll.
		Wertebereich: • GET_STATE (0x0) • RESET (0x1) • DL_TEST (0x2) • DL_TEST_VECTORS (0x3) • DL_SETUP_DATA (0x4) • SETUP (0x5) • TEST (0x6) • TEARDOWN (0x7)
addr	ja	Falls der Befehl DL_TEST gesendet wird, muss in diesem Feld die Adresse angegeben werden, wohin der Test auf dem eingebetteten System gespeichert werden soll.
		Wertebereich: vorzeichenloser 32-Bit Integer-Wert (uint32)
timeout_sec	ja	Für den Befehl TEST kann ein Timeout in Sekunden angegeben werden, nachdem der Test abgebrochen wird. Da die Testausführungsumgebung während eines laufenden Modultests keinen Einfluss auf den Programmablauf hat und nach dem Abbruch eines Modultests nicht auf einen definierten Systemzustand zurückgreifen kann, wird ein Reset durchgeführt.
		Wertebereich: vorzeichenloser 32-Bit Integer-Wert (uint32)
data	nein, darf aber leer sein	Testbinary, Setup-Daten und Testvektoren können je nach Befehl in diesem Array übertragen werden.
		Wertebereich: Bytestrom mit maximaler Länge, die im .proto-File festgelegt wird, zur Zeit 10000 Byte (char[10000]).

Tab. 5.1.: Aufbau einer Request-Nachricht

Feld	Optional	Beschreibung
type	nein	Definiert, ob ein Befehl erfolgreich ausgeführt worden ist oder warum nicht.
		Wertebereich: • OK (0) • INVALID_COMMAND_IN_THIS_STATE (-1) • TEST_IS_ALREADY_LINKED (-2) • TEST_NOT_AVAILABLE (-3) • NO_TEST_ADDRESS_AVAILABLE (-4) • SETUP_DATA_NOT_AVAILABLE (-5) • SETUP_NOT_READY (-6) • TEST_VECTORS_NOT_AVAILABLE (-7) • MESSAGE_PARSING_FAILED (-10) • NOT_IMPLEMENTED (-99)
state	nein	Der Zustand nach dem Ausführen des Befehls
		Wertebereich: • WAIT_FOR_COMMAND (1) • DL_TEST (2) • DL_TEST_VECTORS (3) • DL_SETUP_DATA (4) • SETUP (5) • TEST (6) • TEARDOWN (7)
time100ns	nein, darf aber 0 sein	Bei den Befehlen SETUP und TEST wird die Zeit gemessen, welche für die Ausführung in der Modultest-Komponente benötigt wird.
		Wertebereich: vorzeichenloser 32-Bit Integer-Wert (uint32), gibt die Zeit einer Auflösung von 100ns an. D.h. ein Wert von 10 entspricht 1μs.
result_data	nein, darf aber leer sein	Ergebnisdaten, von der Modultest-Komponente nach dem Test produziert.
		Wertebereich: Bytestrom mit maximaler Länge, die im .proto-File festgelegt wird, zur Zeit 10000 Byte (char[10000]).
ctcpp_data	nein, darf aber leer sein	CTC++ Ergebnisdaten, von CTC++ in der Modultest-Komponente während des Tests produziert
		Wertebereich: Bytestrom mit maximaler Länge, die im .proto-File festgelegt wird, zur Zeit 10000 Byte (char[10000]).

Tab. 5.2.: Aufbau einer Result-Nachricht

5.4.3. Anfrage dekodieren

Um eine Anfrage vom Client (Request) zu dekodieren wird zunächst aus dem Empfangspuffer ein NanoPB-Eingabepuffer (Typ: *pb_istream_t*) mit der maximalen Puffergröße generiert. Dies geschieht mit der Funktion *pb_istream_from_buffer*. Durch den Aufruf von *pb_decode* versucht NanoPB die im Puffer befindliche Anfrage in eine Struktur zu dekodieren. Wird dabei ein Fehler festgestellt, bricht die weitere Bearbeitung der Nachricht ab und es wird ein Ergebnis (Result) mit dem Fehlerwert *MESSAGE_PARSING_FAILED* zum Client zurückgesendet. Der Zustand der Ausführungsumgebung ändert sich dabei nicht.

Die Realisierung erfolgt in der Funktion fnDecodeRequest (siehe Listing 5.7).

Listing 5.7: fnDecodeRequest: Anfrage dekodieren

```
1  Result_Type fnDecodeRequest(uint8_t * pcInputBuffer, Request *
       request)
2  {
3    Result_Type errVal = Result_Type_OK;
4    pb_istream_t stream = pb_istream_from_buffer(pcInputBuffer,
       MAXBUFSIZE);
5
6    if (!pb_decode(&stream, Request_fields, request))
7    {
8      errVal = Result_Type_MESSAGE_PARSING_FAILED;
9    }
10   return errVal;
11 }
```

5.4.4. Ergebnis kodieren

Ein Ergebnis (Result) wird kodiert, indem vom Sendepuffer ein NanoPB-Ausgabepuffer generiert wird. Dazu wird die Funktion *pb_ostream_from_buffer* genutzt. Durch den Aufruf von *pb_encode* werden so die zu sendenden Ergebnisdaten in den Sendepuffer kodiert. War dies erfolgreich, wird die Länge des kodierten Streams zurückgegeben, sonst 0.

Die Realisierung erfolgt in der Funktion *fnEncodeResult* (siehe Listing 5.8).

Listing 5.8: fnEncodeResult: Ergebnis kodieren

```
1  int fnEncodeResult(Result * result, uint8_t * pcOutputBuffer)
2  {
3    int iStreamLen = 0;
```

```
4     pb_ostream_t stream = pb_ostream_from_buffer(pcOutputBuffer,
          MAXBUFSIZE);
5
6     if (pb_encode(&stream, Result_fields, result))
7     {
8       iStreamLen = stream.bytes_written;
9     }
10    return iStreamLen;
11  }
```

5.5. Download

Werden Daten vom Client auf das eingebettete System heruntergeladen, müssen diese zwischengespeichert werden. Dazu bedarf es mehrerer Puffer (siehe Abb. 5.2).

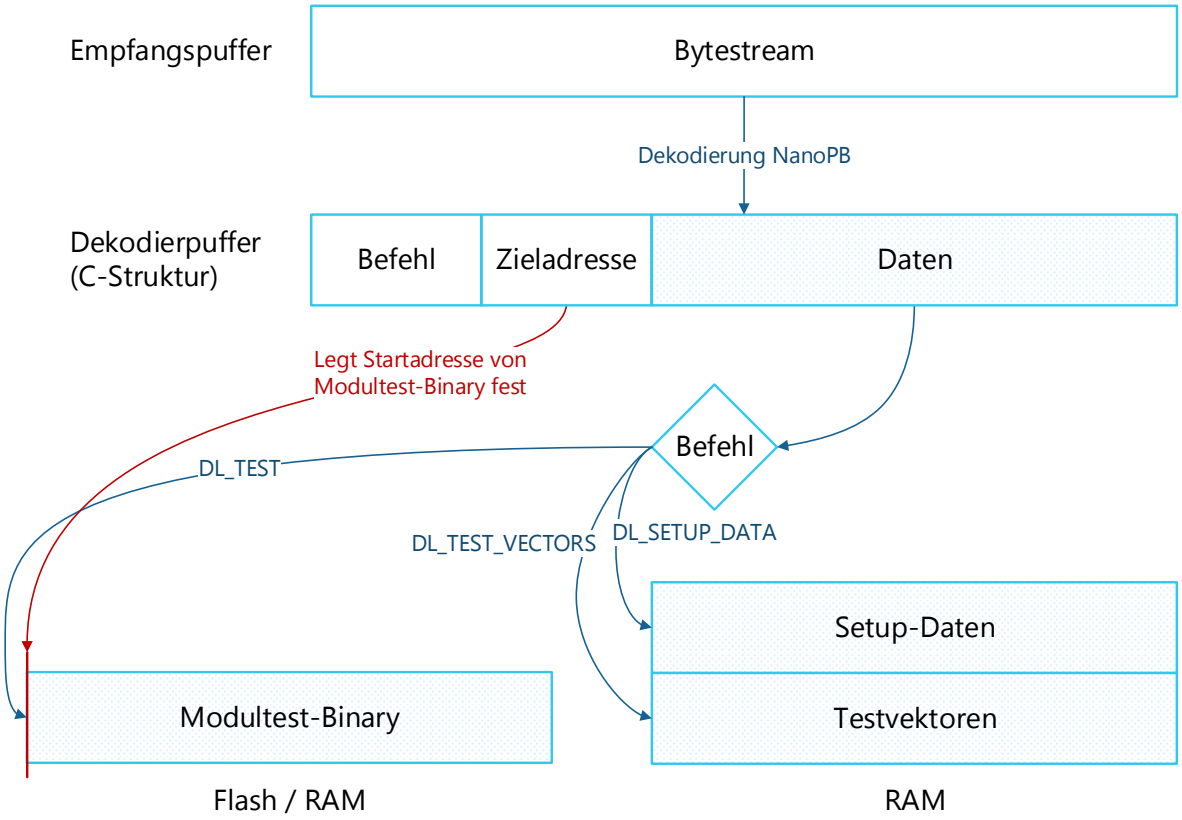

Abb. 5.2.: Übersichtsdarstellung zur Nachrichtendekodierung und der dabei verwendeten Datenpuffer und -speicher

Der Empfangspuffer wird von der HAL-Komponente beim Empfangen von Daten gefüllt. Die

HAL muss gewährleisten, dass nach dem Empfang die Nachricht vollständig in diesem Puffer liegt.

Der zweite Puffer ist die Request-Struktur. In dieser Struktur werden mit Hilfe von NanoPB die empfangenen Daten dekodiert. Der Request wird dann vom Command-Handler behandelt. Ist der Befehl ein Download-Befehl, wird das Datenfeld gesichert. Nachfolgend wird beschrieben, wie dies für die einzelnen Download-Befehle implementiert wurde.

5.5.1. Modultest-Download

Die Weiterleitung des Modultests an die Schreibfunktion der HAL erfolgt in der Funktion *fnTee_DownloadTest*. Dort wird die Funktion *fnHalMemory_Write* aufgerufen, welche die Daten an das eigentliche Ziel schreibt.

5.5.2. Setupdaten-Download

Die Setup-Daten müssen im RAM vorgehalten werden, bis der Setup-Befehl aufgerufen wird. Dazu wird im RAM ein Puffer angelegt, so groß wie die maximale Länge des Request-Datenfeldes. In diesen Puffer wird dann das Request-Datenfeld kopiert. Dazu wird die memcpy-Funktion genutzt.

5.5.3. Testvektoren-Download

Die Testvektoren müssen, wie die Setupdaten auch, im RAM vorgehalten werden. Deshalb ist der Ablauf derselbe wie in Abschnitt 5.5.2 oben beschrieben.

5.6. Testausführung

5.6.1. Setup

Die Implementierung der Setup-Funktion ist in Listing 5.9 dargestellt. Zu Beginn wird der Watchdog je nach Wert von *uiTimeoutSec* gesetzt. Danach wird die Setup-Funktion im Modultest aufgerufen. Dabei wird eine Referenz auf die Daten und die Länge der Setup-Daten übergeben. Zum Schluss wird der Watchdog wieder deaktiviert.

Listing 5.9: fnTee_Setup

```
1 Result_Type fnTee_Setup(Request_data_t * pReqData, uint32_t
    uiTimeoutSec)
```

```
2  {
3    Result_Type errVal = Result_Type_OK;
4
5    if (uiTimeoutSec > 0)
6    {
7      fnHalSystem_EnableWatchdog(uiTimeoutSec);
8    }
9    else
10   {
11     fnHalSystem_DisableWatchdog();
12   }
13
14   pstTest->fnUnitTestSetup(pReqData->bytes,pReqData->size);
15
16   fnHalSystem_DisableWatchdog();
17   return errVal;
18 }
```

5.6.2. Test

Die Implementierung der Test-Funktion ist in Listing 5.10 dargestellt. Als erstes wird der Watchdog aktiviert, wenn ein Timeout angegeben wurde. Danach wird der Timer zurückgesetzt. Daraufhin wird die Test-Funktion im Modultest aufgerufen. Übergeben wird dieser eine Referenz auf die Testvektoren, die Ergebnisdaten und den CTC++-Puffer. Außerdem wird die Länge der Testvektoren mit übergeben sowie eine Referenz auf zwei Integer-Werte, die vom Modultest mit der Länge der Ergebnisdaten und der Länge der von CTC++ generierten Daten beschrieben werden. Nach der Rückkehr aus dem Modultest wird sofort die verbrauchte Zeit gemessen. Dieser Wert wird in den Übergabeparameter *puiTestTime* gespeichert. Der Watchdog wird wieder deaktiviert. Am Ende wird mit *Result_Type_OK* in den Command Handler zurückgesprungen.

Listing 5.10: fnTee_Test

```
1  Result_Type fnTee_Setup(Request_data_t * pReqData, uint32_t
       uiTimeoutSec)
2  {
3    Result_Type errVal = Result_Type_OK;
4
5    if (uiTimeoutSec > 0)
6    {
7      fnHalSystem_EnableWatchdog(uiTimeoutSec);
```

```
8     }
9     else
10    {
11      fnHalSystem_DisableWatchdog();
12    }
13
14    pstTest->fnUnitTestSetup(pReqData->bytes,pReqData->size);
15
16    fnHalSystem_DisableWatchdog();
17    return errVal;
18  }
```

5.6.3. Teardown

Die Implementierung der Teardown-Funktion ist in Listing 5.11 dargestellt. Am Anfang wird der Watchdog aktiviert, wenn ein Timeout vorgegeben wurde. Danach wird die Teardown-Implementierung im Modultest aufgerufen. Die Teardown-Funktion des Modultests hat keine Übergabeparameter. Zum Schluss wird der Watchdog wieder deaktiviert.

Listing 5.11: fnTee_Teardown

```
1   Result_Type fnTee_TearDown(uint32_t uiTimeoutSec)
2   {
3     Result_Type errVal = Result_Type_OK;
4
5     if (uiTimeoutSec > 0)
6     {
7       fnHalSystem_EnableWatchdog(uiTimeoutSec);
8     }
9     else
10    {
11      fnHalSystem_DisableWatchdog();
12    }
13
14    pstTest->fnUnitTestTeardown();
15
16    fnHalSystem_DisableWatchdog();
17    return errVal;
18  }
```

5.7. Interface zum Modultest

Die Information, ob der Modultest an die Testausführungsumgebung gelinkt wurde oder ob er dynamisch nachgeladen werden soll, wird mit dem C-define *TEST_LINKED_TO_TEE* festgelegt. Ist dieses wahr, werden die Funktionszeiger in der Struktur *stUnitTest* konstant auf die gelinkten Funktionen festgelegt. Soll der Modultest dagegen dynamisch gelinkt werden, so müssen in diesem die ersten vier Adressen auf die angegebenen Funktionen Init, Setup, Test und Teardown zeigen. Es wird daher im Falle, dass *TEST_LINKED_TO_TEE* nicht wahr ist, nur die Adresse der Struktur *pstTest* auf die Startadresse des Modultests gelegt (siehe Listing 5.12).

Listing 5.12: Definition des Interfaces zum Modultest

```
1  typedef struct
2  {
3    void (*fnUnitTestInit)(TEE_INTERFACE * tee_interface);
4    void (*fnUnitTestSetup)(unsigned char * pucSetupData,
       unsigned int uiSetupDataLength);
5    void (*fnUnitTestTest)(
6        unsigned char * pucTestVectors, unsigned int
           uiTestVectorLength,
7        unsigned char * pucResultData, unsigned int *
           puiResultDataLength,
8        unsigned char * pucCtcppData, unsigned int *
           puiCtcppDataLength);
9    void (*fnUnitTestTeardown)();
10 } ST_UNITTEST_FUNCTIONS;
11
12 #if TEST_LINKED_TO_TEE
13   const ST_UNITTEST_FUNCTIONS stUnitTest = {
14       &fnUnitTestInit,
15       &fnUnitTestSetup,
16       &fnUnitTestTest,
17       &fnUnitTestTeardown
18   };
19   const ST_UNITTEST_FUNCTIONS * pstTest = &stUnitTest;
20 #else
21   const ST_UNITTEST_FUNCTIONS * pstTest = UNIT_TEST_ADDR;
22 #endif
```

6. Verifizierung und Validierung

An das V-Modell angelehnt (siehe Abb. 1.1 auf Seite 3), wird in diesem Abschnitt die Verifizierung und Validierung des Ergebnisses dargestellt.

Die Validierung prüft die Eignung eines Softwareteils oder der Software für die gewünschte Anwendung. Bei der Verifizierung wird dagegen festgestellt, ob die Software richtig umgesetzt wurde. Die Einhaltung von Coderegeln wird beispielsweise verifiziert.

Dazu werden geeignete Modultests, Integrationstests und ein Systemtest entwickelt. Außerdem wird ein Nachweis der Funktionalität für die IAV durch eine Beispielanwendung erbracht.

6.1. Testumgebung

Da als Entwicklungsumgebung auf Eclipse gesetzt wurde, bietet es sich an, auch die Tests dort direkt auszuführen. Seit der Eclipse CDT-Version 8.1 ist eine Unterstützung für automatisierte Modultests enthalten [The12]. Allerdings ist dies eher eine grafische Aufbereitung der Ausgabe verschiedener Testframeworks als eine eigenständige Modultestumgebung. Es werden die Frameworks Google Test, Boost und QtTest unterstützt.

Als Alternative dazu existieren noch verschiedene andere Plugins für Eclipse, die eine ähnliche Funktionalität bereitstellen. Einen großen Bekanntheitsgrad hat das CUTE (C++ Unit Testing Easier, Modultestframework mit Integration in Eclipse) C++ Testframework.

Die Auswahl des Testframeworks fiel auf Google Test. Google Test hat eine für dieses Projekt ausreichende Funktionalität, lässt sich einfach konfigurieren, benötigt nur wenige Dateien zum Testen und Google ist eine renommierte Firma. Weiterhin ist die Einbindung in die Eclipse CDT sehr einfach möglich und gut dokumentiert. Die anderen erwähnten Testframeworks dagegen haben oft einen großen Umfang, der für diese Zwecke nicht benötigt wird, und es braucht viel Zeit sie zu konfigurieren. QtTest ist, wie der Name schon sagt, hauptsächlich für Qt-basierte Anwendungen gedacht.

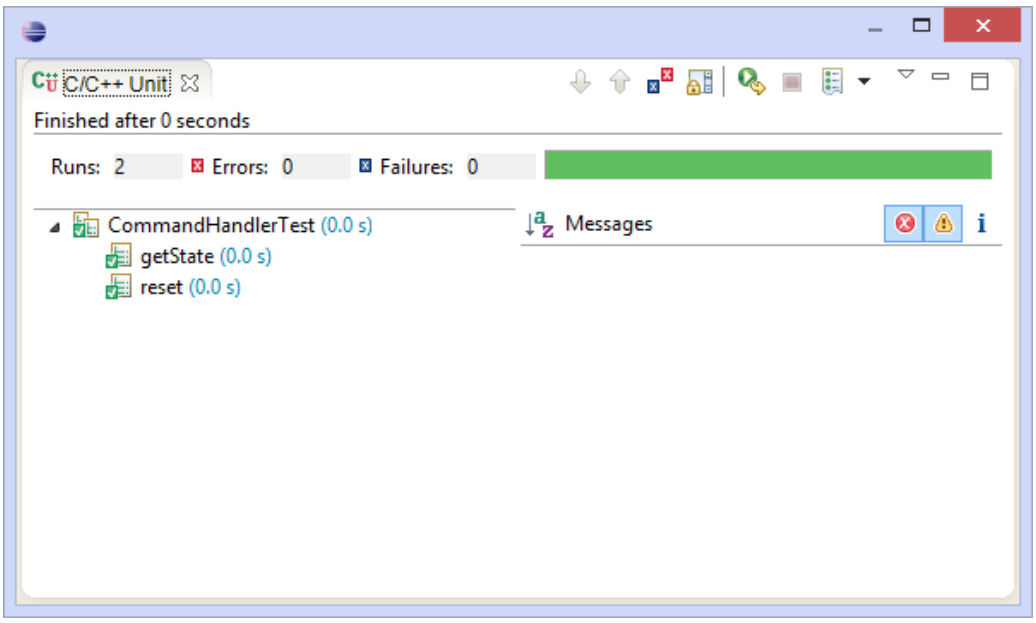

Abb. 6.1.: Eclipse Integration von Google Test mit zwei erfolgreich durchlaufenen Tests

6.2. Modultest Command-Handler

Das Command-Handler Modul ist das am kompliziertesten zu testende Modul. Ob der Übergang zu einem Zustand erlaubt ist, hängt davon ab, ob der Zielzustand vom aktuellen Zustand abhängig ist und eventuell noch von weiteren Randbedingungen. Die entstehenden Testfälle sind in Abb. 6.2 auf der nächsten Seite zu sehen.

Befehl *Request_Type_*	Zustand Hauptautomat *TEE_STATE_*	bTestAlreadyLinked	bTestAvailable	bSetupDataAvailable	bSetupReady	bTestVectorsAvailable	Zielzustand *Result_State_*	bTestAlreadyLinked	bTestAvailable	bSetupDataAvailable	bSetupReady	bTestVectorsAvailable	Fehlerwert / Besonderheit *Result_Type_*
GET_STATE	WAIT_FOR_COMMAND	x	x	x	x	x	WAIT_FOR_COMMAND	x	x	x	x	x	OK
GET_STATE	DOWNLOAD	x	x	x	x	x	DL_SETUP_DATA || DL_TEST || DL_TEST_VECTORS	x	x	x	x	x	OK
GET_STATE	SETUP	x	x	x	x	x	SETUP	x	x	x	x	x	OK
GET_STATE	TEST	x	x	x	x	x	TEST	x	x	x	x	x	OK
GET_STATE	TEARDOWN	x	x	x	x	x	TEARDOWN	x	x	x	x	x	OK
RESET	WAIT_FOR_COMMAND	x	x	x	x	x	x	x	x	x	x	x	
RESET	DOWNLOAD	x	x	x	x	x	x	x	x	x	x	x	
RESET	SETUP	x	x	x	x	x	x	x	x	x	x	x	Prüfen, ob Reset-Stub aufgerufen wurde, Fehlerwert egal
RESET	TEST	x	x	x	x	x	x	x	x	x	x	x	
RESET	TEARDOWN	x	x	x	x	x	x	x	x	x	x	x	
DL_TEST	WAIT_FOR_COMMAND	0	0	1	1	1	DL_TEST	x	1	0	0	0	OK
DL_TEST	DOWNLOAD	0	0	1	1	1	DL_TEST	x	1	0	0	0	OK
DL_TEST	SETUP	0	0	1	1	1	DL_TEST	x	1	0	0	0	OK
DL_TEST	TEST	0	0	1	1	1	DL_TEST	x	1	0	0	0	OK
DL_TEST	TEARDOWN	0	0	1	1	1	DL_TEST	x	1	0	0	0	OK
DL_TEST	WAIT_FOR_COMMAND	1	0	1	1	1	WAIT_FOR_COMMAND	x	0	1	1	1	TEST_IS_ALREADY_LINKED
DL_SETUP_DATA	WAIT_FOR_COMMAND	0	0	0	x	x	WAIT_FOR_COMMAND	x	x	0	x	x	TEST_NOT_AVAILABLE
DL_SETUP_DATA	WAIT_FOR_COMMAND	0	1	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_SETUP_DATA	WAIT_FOR_COMMAND	1	0	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_SETUP_DATA	WAIT_FOR_COMMAND	1	1	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_SETUP_DATA	DOWNLOAD	1	1	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_SETUP_DATA	SETUP	1	1	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_SETUP_DATA	TEST	1	1	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_SETUP_DATA	TEARDOWN	1	1	0	x	x	DL_SETUP_DATA	x	x	1	x	x	OK
DL_TEST_VECTORS	WAIT_FOR_COMMAND	0	0	x	x	0	WAIT_FOR_COMMAND	x	x	x	x	0	TEST_NOT_AVAILABLE
DL_TEST_VECTORS	WAIT_FOR_COMMAND	0	1	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
DL_TEST_VECTORS	WAIT_FOR_COMMAND	1	0	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
DL_TEST_VECTORS	WAIT_FOR_COMMAND	1	1	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
DL_TEST_VECTORS	DOWNLOAD	1	1	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
DL_TEST_VECTORS	SETUP	1	1	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
DL_TEST_VECTORS	TEST	1	1	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
DL_TEST_VECTORS	TEARDOWN	1	1	x	x	0	DL_TEST_VECTORS	x	x	x	x	1	OK
SETUP	DOWNLOAD	x	x	0	0	x	DL_SETUP_DATA || DL_TEST || DL_TEST_VECTORS	x	x	x	0	x	SETUP_DATA_NOT_AVAILABLE
SETUP	DOWNLOAD	x	x	1	0	x	SETUP	x	x	x	1	x	OK
SETUP	WAIT_FOR_COMMAND	x	x	1	0	x	WAIT_FOR_COMMAND	x	x	x	0	x	INVALID_COMMAND_IN_THIS_STATE
SETUP	SETUP	x	x	1	0	x	SETUP	x	x	x	0	x	INVALID_COMMAND_IN_THIS_STATE
SETUP	TEST	x	x	1	0	x	TEST	x	x	x	0	x	INVALID_COMMAND_IN_THIS_STATE
SETUP	TEARDOWN	x	x	1	0	x	TEARDOWN	x	x	x	0	x	INVALID_COMMAND_IN_THIS_STATE
TEST	DOWNLOAD	x	x	x	0	0	DL_SETUP_DATA || DL_TEST || DL_TEST_VECTORS	x	x	x	x	x	SETUP_NOT_READY
TEST	DOWNLOAD	x	x	x	0	1	DL_SETUP_DATA || DL_TEST || DL_TEST_VECTORS	x	x	x	x	x	SETUP_NOT_READY
TEST	DOWNLOAD	x	x	x	1	0	DL_TEST_VECTORS	x	x	x	x	x	TEST_VECTORS_NOT_AVAILABLE
TEST	DOWNLOAD	x	x	x	1	1	TEST	x	x	x	x	x	OK
TEST	SETUP	x	x	x	1	0	SETUP	x	x	x	x	x	TEST_VECTORS_NOT_AVAILABLE
TEST	SETUP	x	x	x	1	1	TEST	x	x	x	x	x	OK
TEST	WAIT_FOR_COMMAND	x	x	x	1	1	WAIT_FOR_COMMAND	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE
TEST	TEST	x	x	x	1	1	TEST	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE
TEST	TEARDOWN	x	x	x	1	1	TEARDOWN	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE
TEARDOWN	WAIT_FOR_COMMAND	x	x	x	x	x	WAIT_FOR_COMMAND	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE
TEARDOWN	DOWNLOAD	x	x	x	x	x	DL_SETUP_DATA || DL_TEST || DL_TEST_VECTORS	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE
TEARDOWN	SETUP	x	x	x	x	x	SETUP	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE
TEARDOWN	TEST	x	x	x	x	x	TEARDOWN	x	x	x	x	x	OK
TEARDOWN	TEARDOWN	x	x	x	x	x	TEARDOWN	x	x	x	x	x	INVALID_COMMAND_IN_THIS_STATE

Abb. 6.2.: Testfälle für das CommandHandler-Modul. Dunkelblaue Bereiche sind Eingangsbedingungen, hellblaue Bereiche stellen Flags dar, die beim Zustandswechsel verändert werden. `1` steht für wahr, `0` für falsch und `x` für nicht von Interesse

6.3. Modultest Zustandsabfrage

Die Funktion *fnTee_getState* berechnet aus dem Zustand des Hauptautomaten und des Downloadautomaten den aktuellen Gesamtzustand des Systems. Somit liefern zwei Eingangswerte einen Ausgangswert. Die dabei erforderlichen Testfälle sind in Tab. 6.1 zu sehen.

Eingangszustand Hauptautomat (TEE_STATE_)	Eingangszustand Downloadautomat (TEE_STATE_DOWNLOAD_)	Erwartete Antwort (Result_State_)
WAIT_FOR_COMMAND	x	WAIT_FOR_COMMAND
DOWNLOAD	TEST	DL_TEST
DOWNLOAD	SETUPDATA	DL_SETUP_DATA
DOWNLOAD	TESTVECTORS	DL_TEST_VECTORS
SETUP	x	SETUP
TEST	x	TEST
TEARDOWN	x	TEARDOWN

Tab. 6.1.: Erforderliche Testfälle für Modultest Zustandsabfrage

6.4. Modultest Download Setupdaten bzw. Testvektoren

Es wird ein Testfall benötigt, der prüft, ob die Daten von der Quelladresse korrekt an die Zieladresse kopiert wurden und ob die Funktion *Result_Type_OK* zurückgibt. Zu lang kann der Datenstrom nicht sein, da sonst NanoPB schon vorher einen Dekodierfehler produziert hätte. Als Grenzfall könnte man noch einen Testfall aufnehmen, bei dem das Datenfeld leer ist.

6.5. Modultest Download Test

Wie bei den beiden anderen Download-Funktionen ist nur ein Testfall notwendig. Jedoch muss die Funktion *fnHalMemory_Write* durch einen Stub ersetzt werden. Im Stub muss geprüft werden, ob Quelladresse, Zieladresse und Länge übereinstimmen. Zu lang kann der Datenstrom nicht sein, da sonst NanoPB schon vorher einen Dekodierfehler produziert hätte.

6.6. Modultest Test

Um die Funktion *fnTee_Test* zu testen, werden folgende Stub-Methoden benötigt:

- fnHalTimer_ResetTimer
- fnHalTimer_GetTimestamp
- fnHalSystem_EnableWatchdog
- fnHalSystem_DisableWatchdog
- fnUnitTestTest

Die Funktion *fnHalTimer_GetTimestamp* gibt eine Konstante zurück, womit die Übertragung der Testdauer an den Client geprüft werden kann. In der Stub-Funktion *fnUnitTestTest* wird geprüft, ob die Parameter richtig übergeben wurden, und das Ergebnis für den Test gespeichert.

6.7. Integrationstest Nachrichtenverarbeitung

Um die Nachrichtenverarbeitung zu testen, muss jeder Testfall als in NanoPB kodierter Datenstrom vorliegen. Dieser wird über die Stub-Funktion *fnHalComm_ReceiveData* in die Funktion injiziert. Daraufhin wird der Datenstrom in der Funktion verarbeitet und das Ergebnis an die Stub-Funktion *fnHalComm_SendByteArray* gesendet. Dort muss der Datenstrom zum Test wieder dekodiert und mit den Erwartungswerten verglichen werden.

Die Aufrufe in den Modultest werden durch Stub-Funktionen ersetzt.

Es ergeben sich folgende Testfälle:

- Injizierter Eingangsdatenstrom fehlerhaft, daher Dekodierung nicht möglich. Hier muss *Result_Type_MESSAGE_PARSING_FAILED* im Ergebnis stehen.
- Injizierter Eingangsdatenstrom leer, daher Dekodierung nicht möglich. Hier muss ebenfalls *Result_Type_MESSAGE_PARSING_FAILED* im Ergebnis stehen.
- Dekodieren möglich, Befehl wird im Command-Handler verarbeitet, Stub-Funktionen im Modultest werden aufgerufen, das Ergebnis mit einem Testdatenstrom aus der fnUnitTestTest-Stubfunktion zurückgegeben, kodiert und an die Stubfunktion *fnHalComm_SendByteArray* „versandt“.

6.8. Beispieltest auf dem PC

Wie schon erwähnt, wurde in der IAV parallel zur Testausführungsumgebung ein einfacher, passender Client entwickelt. Damit lassen sich über eine TCP/IP-Verbindung Nachrichten an die Testausführungsumgebung senden.

So lässt sich ein Systemtest erstellen, der auch gleichzeitig den von der IAV gewünschten Beispieltest darstellt. Der Systemtest ähnelt dem Integrationstest oben, nur dass der Datenstrom jetzt über die Kommunikationsfunktionen der HAL in die Nachrichtenschleife gelangen und auch über diese wieder an den Client zurückgesendet werden. Außerdem gibt es keine Stubfunktionen für den Modultest mehr, sondern es wurde ein einfacher Modultest entwickelt, der sinnvolle Werte für Testparameter an die Testausführungsumgebung zurückliefert.

Das zu testende Beispielmodul liefert den Quotient aus zwei Zahlen ($c = a/b * f$). Dabei ist f ein Faktor, der über die Setup-Daten gesetzt werden kann.

Die Setupdaten, welche mit der Nachricht *DOWNLOAD_SETUP_DATA* gesendet werden, sind 4-Byte lang und enthalten den 32-Bit Integer-Wert für f.

Mit der Nachricht *DOWNLOAD_TEST_VECTORS* wird der 8-Byte lange Testvektorendatenstrom verschickt. Er enthält zwei 32-Bit Integer, die a und b darstellen. Mit der Test-Routine werden die Testvektoren aus dem zwischengespeicherten Datenstrom gelesen und das Ergebnis wird berechnet.

Die Testfälle für den Systemtest werden in Tab. 6.2 auf der nächsten Seite beschrieben.

Wie man sehen kann, werden die Daten, welche mit den Befehlen DL_TEST_VECTORS und DL_SETUP_DATA übertragen wurden, korrekt beim Setup und Test berücksichtigt. Da das Zielsystem, wie schon in der Aufgabenstellung erwähnt, ein PC ist, ist es nicht möglich, den Austausch des Tests zur Laufzeit zu testen.

Mit diesem Systemtest ist auch der Nachweis für die Funktionalität der Software erbracht, der in der Aufgabenstellung gefordert wurde.

Request_Type	a/f	b	Rückgabe (Result_State, Result_Type, int aus Datenstrom)
GET_STATE			WAIT_FOR_COMMAND, OK
DL_TEST			WAIT_FOR_COMMAND, TEST_IS_ALREADY_LINKED
SETUP			WAIT_FOR_COMMAND, INVALID_COMMAND_IN_THIS_STATE
TEST			WAIT_FOR_COMMAND, INVALID_COMMAND_IN_THIS_STATE
TEARDOWN			WAIT_FOR_COMMAND, INVALID_COMMAND_IN_THIS_STATE
DL_TEST_VECTORS,	9	3	DL_TEST_VECTORS, OK
SETUP			DL_TEST_VECTORS, SETUP_DATA_NOT_AVAILABLE
DL_SETUP_DATA	1000		DL_SETUP_DATA, OK
TEST			DL_SETUP_DATA, SETUP_NOT_READY
SETUP			SETUP, OK
TEST			DL_TEST, OK, 3
DL_SETUP_DATA	10		DL_SETUP_DATA, OK
TEST			DL_SETUP_DATA, SETUP_NOT_READY
SETUP			SETUP, OK
TEST			DL_TEST, OK, 3
DL_TEST_VECTORS	12	3	DL_TEST_VECTORS, OK
TEST			TEST, OK, 4
TEARDOWN			TEARDOWN, OK
DL_SETUP_DATA	1000	0	DL_SETUP_DATA, OK

Tab. 6.2.: Testfälle für den Systemtest

7. Zusammenfassung und Perspektiven

7.1. Auswertung des Ergebnisses

In diesem Abschnitt wird diskutiert, wie gut die gestellten Ziele und Anforderungen erreicht wurden.

Austausch von Daten zur Laufzeit Das primäre Ziel dieser Arbeit war es, eine Testumgebung zu entwickeln, in der Modultest, Testumgebungsdaten (Setupdaten) und Testvektoren unabhängig voneinander (soweit sinnvoll) ausgetauscht werden können. Dies wurde erreicht, indem ein Download-Modul entwickelt wurde, welches die Testvektoren und Testumgebungsdaten als Nachrichten empfängt und dem Modultest verfügbar macht. Auch der Modultest lässt sich durch eine Nachricht vom Client austauschen. Es wurde eine Methode entwickelt, um den Modultest an eine ausführbare Stelle im Zielsystem zu bringen. Dabei wurde die plattformabhängige Logik in die Hardwareabstraktionsschicht ausgelagert. Da als Zielsystem zum Testen nur ein PC zur Verfügung stand, konnte diese Funktion leider nicht verifiziert werden.

Organisieren des Tests in Setup, Test, Tear-Down Die in vielen Testframeworks übliche Aufteilung des Tests in die Methoden Setup, Test und Teardown sollte sich auch in der Bedienung der Testausführungsumgebung widerspiegeln. Dies wurde durch drei entsprechende Befehle implementiert, die durch eine Nachricht vom Client ausgeführt werden können. Während der Test-Methode wird die Zeit gemessen und an den Client zurückgeliefert.

Portabilität Die Anforderung der Portabilität wurde durch die Aufteilung der Module in drei Kategorien gewährleistet:

- plattformunabhängig (Testausführungsumgebung)
- plattformabhängig (Hardwareabstraktionsschicht)
- modultestspezifisch (Modultest)

Für jede Plattform, auf der die Testausführungsumgebung verwendet werden soll, muss nur die Hardwareabstraktionsschicht angepasst werden.

Kommunikation abstrahieren Eine wichtige Anforderung war, die Kommunikation zwischen Client und Server über eine der auf dem Zielsystem vorhandenen Schnittstellen abzuwickeln.

Durch die strenge Trennung von plattformunabhängigem und plattformabhängigem Code ist dies möglich. Der plattformunabhängige Code in der Testausführungsumgebung legt fest, was gesendet wird, während die plattformabhängige HAL über das Wie und Worüber bestimmt.

Zuverlässigkeit Die Zuverlässigkeit der Testausführungsumgebung wurde durch Modul- und Integrationstests bis zu einem Grad bestätigt, welcher die IAV zufriedenstellt. Die Einhaltung der MISRA-Regeln führt dazu, dass typische Programmierfehler vermieden werden, wie auch nicht eindeutig spezifizierte C-Konstrukte, deren Interpretation sich in C-Compilern unterscheidet.

Speichereffizienz Da die Testausführungsumgebung auch auf leistungsschwachen Controllern mit meist wenig Speicher lauffähig sein soll, wurde besonderer Wert auf die Codegröße gelegt. Mit NanoPB wurde eine Protokollbibliothek gewählt, die nur wenig Kilobyte Codespeicher benötigt. Jedoch kann die RAM-Nutzung bei der Datenübertragung noch effizienter gestaltet werden. Bis jetzt werden, wie in Abb. 5.2 auf Seite 54 dargestellt, die empfangenen Daten erst in einem Bytestream zwischengespeichert, dann in einen weiteren Puffer dekodiert, von dem aus bei bestimmten Befehlen der Datenanhang nochmals kopiert wird.

Wartbarkeit Die Wartbarkeit einer Software wird u.a. verbessert, wenn sie besser lesbar ist. Dies wurde durch die Einhaltung der MISRA-C-Regeln erreicht sowie durch Codekommentare, aus der die Software Doxygen eine automatische Codedokumentation erstellen kann. Da das Übertragungsprotokoll in einer Metasprache verfasst ist, kann es ohne große Einarbeitung leicht erweitert werden. Auch das Protokoll wurde dokumentiert (siehe Abschnitt A.1 auf Seite A1). Weiterhin wurden im Sinne einer einfachen Einarbeitung in das Projekt alle Konzeptebenen durch verschiedene UML-Darstellungen, wie z.B. Anwendungsfalldiagramme, Zustandsdiagramme, Komponentendiagramme und Sequenzdiagramme, modelliert. Viele davon sind auch in dieser Arbeit abgedruckt.

7.2. Ausblick

7.2.1. Effiziente RAM-Nutzung

In Abb. 5.2 auf Seite 54 wurde der Datenfluss von empfangenen Nachrichten dargestellt. Wird eine neue Nachricht zum Download von Setupdaten oder Testvektoren empfangen, wird sie im Empfangspuffer der HAL zwischengespeichert. Danach wird sie in einen weiteren Puffer von NanoPB dekodiert. Die Daten werden von dort dann in einen Speicher kopiert, auf den der Modultest dann zugreift. Das bedeutet, dass immer der dreifache Speicher der maximal möglichen Datenlänge vorgehalten wird.

Diesen Prozess könnte man optimieren, indem man statt festen Feldern die Callback-Funktionalität von NanoPB nutzt. Um NanoPB anzuweisen, die Daten nicht selbst zu kopieren, muss das Feld in der Protokolldefinition im .proto-File als optional gekennzeichnet werden. Vor der Dekodierung muss in der Zielstruktur, in welcher NanoPB die dekodierten Daten ablegt, der Funktionszeiger für das Datenfeld auf die selbst zu implementierende Callback-Funktion weisen. In der Callback-Funktion könnte man die Daten direkt vom Empfangspuffer an den Speicherort kopieren, auf den der Modultest zugreift. Man spart damit einen Puffer. Die jetzt implementierte Methode und das verbesserte Konzept wird in Abb. 7.1 gegenübergestellt.

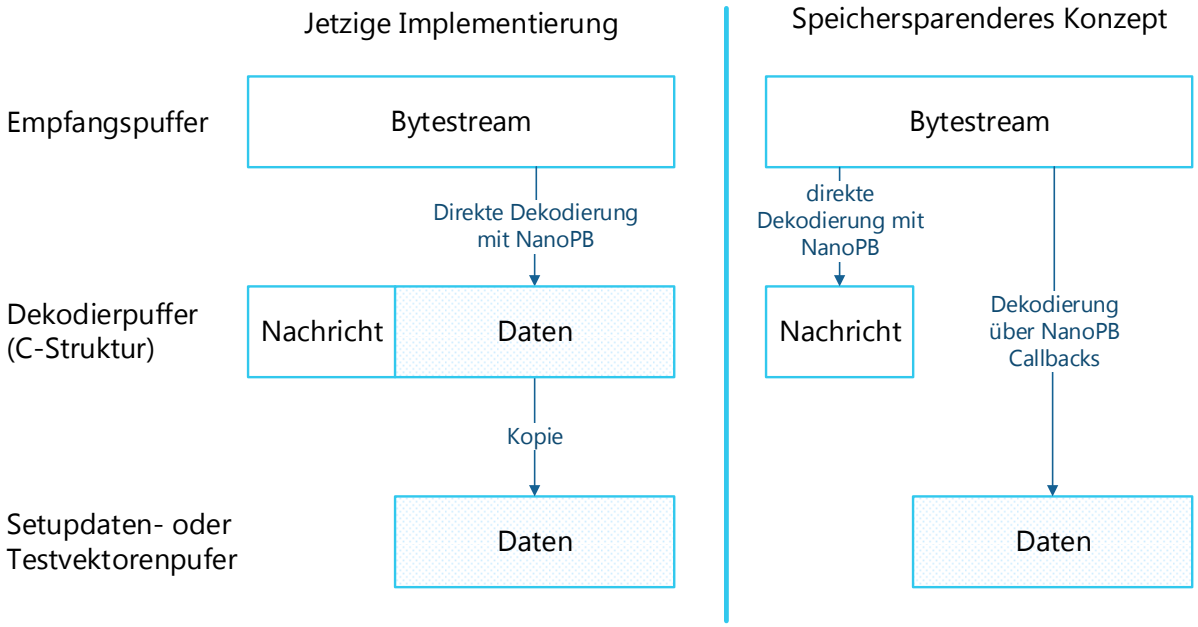

Abb. 7.1.: Optimierung des RAM-Bedarfs durch Nutzung von NanoPB Callback-Methoden

7.2.2. Effiziente Modultestübertragung

Zur Zeit stellt die Testausführungsumgebung lediglich einen Puffer für die Ergebnisse bereit, welche von der eventuell in den Modultest eingebundenen Testwell CTC++ Implementierung generiert werden.

Da CTC++ in der IAV recht häufig zum Einsatz kommt, könnte man das Host-Target Add-on mit in die Testausführungsumgebung aufnehmen. Der Vorteil ist klar: Es muss nicht jedes Mal derselbe CTC++ Programmcode auf das Zielsystem übertragen werden, wenn ein Modultest CTC++ benötigt.

Die Quellen des CTC++ Host-Target Add-on's sind reiner C-Code und plattformunabhängig, so dass sie leicht integrierbar sind. Der instrumentalisierte Code im Modultest ruft Funktionen im CTC++ Host-Target Add-on über Zeiger auf, so dass auch die Erweiterung des jetzt schon

auf Funktionszeigern basierenden Interfaces zwischen Modultest und Testausführungsumgebung kein Problem darstellen würde.

7.2.3. Client-Konzept

Durch die IAV wurde parallel zur Entwicklung der Zielsystemseite ein Client entwickelt, mit dem es möglich ist, Testnachrichten für das Zielsystem zu generieren und zu versenden. Um die Testausführungsumgebung wirklich nutzbar zu machen, ist eine wesentliche Weiterentwicklung dieser Client-Software notwendig.

Um automatisiert Tests durchführen zu können, wäre z.B. ein Client mit Kommandozeilenschnittstelle als Lösung in Betracht zu ziehen. Damit ließe sich der Client und somit die Testausführungsumgebung in die von der IAV bevorzugte Scons-Buildumgebung einbinden.

Um Tests unabhängig vom Buildprozess durchzuführen, könnte eine grafische Oberfläche für den Client erstellt werden. Der Vorteil einer grafischen Oberfläche liegt an der intuitiven Bedienbarkeit, da große Mengen von Testfällen übersichtlich aufbereitet werden können. Außerdem kann der Nutzer bei der manuellen Testfallerstellung sowie bei der Integration von extern generierten Testfällen, wie z.B. vom IAV Testvektor Generator, unterstützt werden.

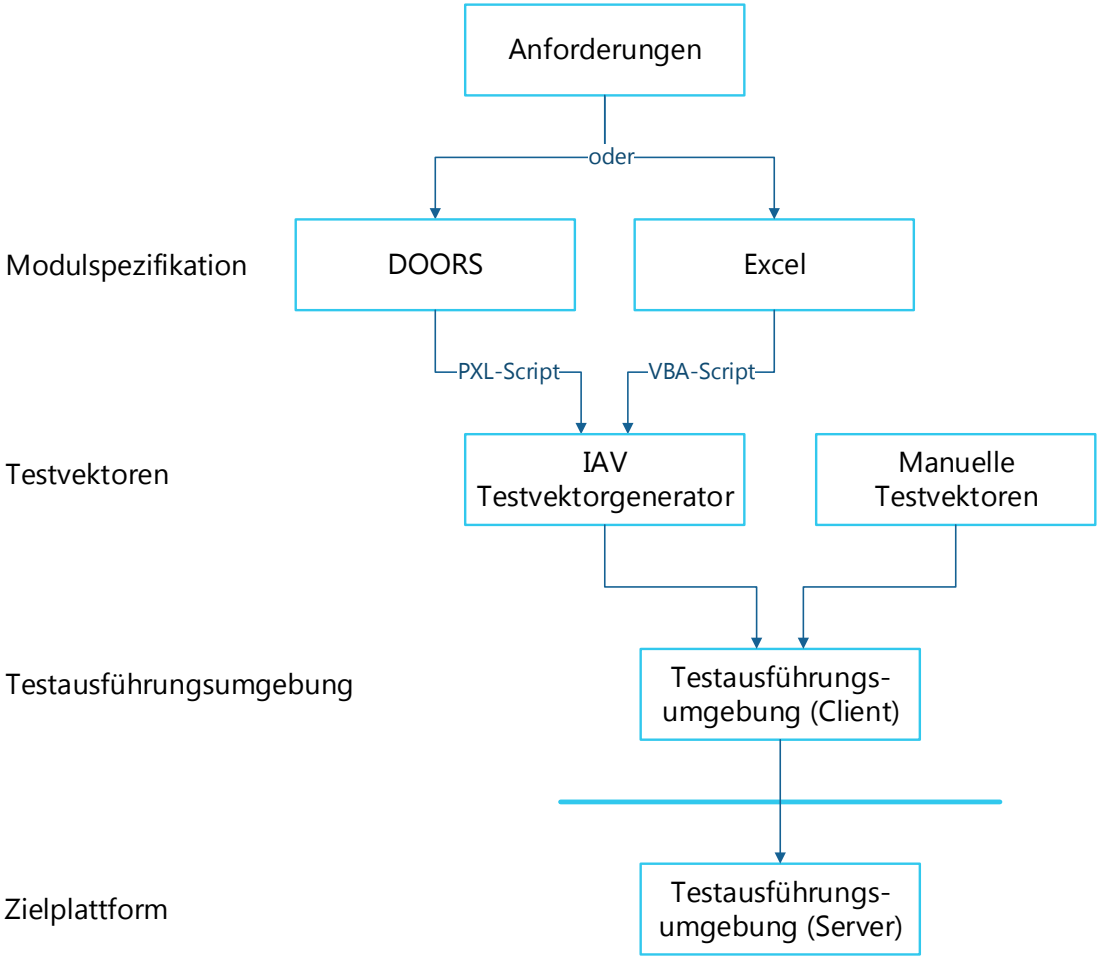

Abb. 7.2.: Toolkette: von der Anforderung bis zum Test

Außerdem wird ein Konzept erwogen, wie es in Abb. 7.2 dargestellt ist. Es soll in Zukunft möglich sein, eine Spezifikation mit einem dafür geeigneten Tool zu erstellen, woraus der IAV Testvektorengenerator automatisch Testvektoren erzeugt. Diese können über den Client der Testausführungsumgebung, zusammen mit manuell hinzugefügten Testfällen, zu Modultests zusammengestellt und in der Testausführungsumgebung ausgeführt werden. Bis dahin gibt es allerdings noch einiges zu tun.

Literaturverzeichnis

[Bun10] Bundesinnenministerium, IT-Stab: *V-Modell XT Bund*. März 2010

[DH08] Dr. Hoffmann, Dirk W.: *Software-Qualität*. Springer Berlin Heidelberg, 2008. – ISBN 978–3–530–76323–9. – Kapitel 1 gibt guten Überblick über Nichtfunktionale Anforderungen

[DL09] Dr. Liggesmeyer, P.: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag Heidelberg, 2009. – ISBN 978–3–8274–2056–5

[DM08] Dr. Meier, Erich: V-Modelle in Automotive-Projekten. In: *Automobil Elektronik* (Februar 2008), S. 36,37

[DW12] Dr. Werner, Matthias: *Verlässliche Systeme - Fallstudien*. http://osg.informatik.tu-chemnitz.de/lehre/old/ws1112/ds/ds-11-Fallstudien.pdf. Version: 2012

[Goo12] Google: *Encoding - Protocol Buffers*. https://developers.google.com/protocol-buffers/docs/encoding. Version: 04 2012

[Hau05] Hauptfleisch, K.: *Kfz-Elektronik und -Software wird immer komplexer*. http://www.channelpartner.de/200208. Version: September 2005

[Hit13a] Hitex: *Tessy - The Invaluable Test Tool*. http://www.hitex.com/index.php?id=module-unit-test. Version: 2013

[Hit13b] Hitex: *Tessy Controls Debuggers for Test Execution*. http://www.hitex.com/index.php?id=1694. Version: 2013

[Ilg12] Ilg, Peter: *Informatiker im Autobau - Mehr Software als im Kampfjet*. http://www.spiegel.de/karriere/berufsstart/auto-industrie-informatiker-dringend-gesucht-a-859701.html. Version: Oktober 2012

[Kle13] Kleuker, Stephan: *Qualitätssicherung durch Softwaretests*. Springer Fachmedien Wiesbaden, 2013. – ISBN 978–3–8348–2068–6

[Lev95] Leveson, Nancy: *Medical Devices: The Therac-25*. http://sunnyday.mit.edu/papers/therac.pdf. Version: 1995

[Raz13] Razorcat: Compilers and Debuggers/Emulators supported by Tessy. (2013). http://www.hitex.com/fileadmin/free/tessy/tessy_debug_matrix.pdf

[The12] The Eclipse Foundation: *C/C++ Unit Testing Support*. http://wiki.eclipse.org/CDT/User/NewIn81#C.2FC.2B.2B_Unit_Testing_Support. Version: 12 2012

[Xia13] Xiaofeng@google.com: *Third-Party Add-ons for Protocol Buffers - Programming Languages*. https://code.google.com/p/protobuf/wiki/ThirdPartyAddOns#Programming_Languages. Version: September 2013

A. Anhang

A.1. Protokolldefinition

Listing A.1: Protokolldefinition für die Kommunikation zwischen Client und Server

```
import "nanopb.proto";

message Request {
  enum Type {
    GET_STATE = 0;
    RESET = 1;
    DL_TEST = 2;
    DL_TEST_VECTORS = 3;
    DL_SETUP_DATA = 4;
    SETUP = 5;
    TEST = 6;
    TEARDOWN = 7;
  }
  required Type type = 1 [default = GET_STATE];
  optional uint32 addr = 2;
  required uint32 timeout_sec = 3 [default = 30];
  required bytes data = 4 [(nanopb).max_size = 10000];
}

message Result {
  enum Type {
    OK = 0;
    INVALID_COMMAND_IN_THIS_STATE = -1;
    TEST_IS_ALREADY_LINKED = -2;
    TEST_NOT_AVAILABLE = -3;
    NO_TEST_ADDRESS_AVAILABLE = -4;
    SETUP_DATA_NOT_AVAILABLE = -5;
    SETUP_NOT_READY = -6;
    TEST_VECTORS_NOT_AVAILABLE = -7;
```

```
30      MESSAGE_PARSING_FAILED = -10;
31      NOT_IMPLEMENTED = -99;
32    }
33    required Type type = 1 [default = OK];
34    enum State {
35      WAIT_FOR_COMMAND = 1;
36      DL_TEST = 2;
37      DL_TEST_VECTORS = 3;
38      DL_SETUP_DATA = 4;
39      SETUP = 5;
40      TEST = 6;
41      TEARDOWN = 7;
42    }
43    required State state = 2 [default = WAIT_FOR_COMMAND];
44    required uint32 time100ns = 3;
45    required bytes result_data = 4 [(nanopb).max_size = 10000];
46    required bytes ctcpp_data = 5 [(nanopb).max_size = 10000];
47  }
```

A.2. Dateiverzeichnis

Dateiverzeichnis des beigelegten Datenträgers

- Diplomarbeit.pdf ..... Diplomarbeit als PDF-Dokument mit Verlinkungen
- index.html ..... Übersicht des Datenträgerinhalts
- Doxygen ..... HTML-Codedokumentation, mit Doxygen generiert
- Literaturquellen ..... Auzüge aus in der Diplomarbeit verwendeten Literaturquellen
  - DHof2008-Kap01-Einleitung.pdf ..... [DH08]
  - DHof2008-Kap04-Software-Test.pdf ..... [DH08]
  - DLig2009-Softwarequalität-Kap11-Pruefstrategien.pdf ..... [DL09]
  - DLig2009-Softwarequalität-Kap12-Werkzeuge.pdf ..... [DL09]
  - DMei2008.pdf ..... [DM08]
  - DWer2012.pdf ..... [DW12]
  - Ecf2012-New-In-81-C-C++-Unit-Testing-Support.pdf ..... [The12]
  - Goo2012-Encoding-Protocol-Buffers.pdf ..... [Goo12]
  - Hau2005.pdf ..... [Hau05]
  - Hit2013a-Tessy-Kurzbeschreibung.pdf ..... [Hit13a]
  - Hit2013b-Tessy-controls-Debuggers-for-Test-Execution.pdf ..... [Hit13b]
  - Ilg2012.pdf ..... [Ilg12]
  - Kle2013.pdf ..... [Kle13]
  - Lev1995.pdf ..... [Lev95]
  - Raz2013-Tessy-Supported-Targets.pdf ..... [Raz13]
  - Vxt2010-V-Modell-XT-Bund-Gesamt.pdf ..... [Bun10]
  - xia2013-Third-Party-Add-ons-for-Protocol-Buffers.pdf ..... [Xia13]
- Source ..... Sourcecode des Projekts
  - 23b_Library ..... Bibliotheken
    - 01_HAL ..... Hardwareabstraktionsschicht für PC (teilweise implementiert)
      - include .....
        - HAL.h .....
        - HAL_Communication.h .....

- HAL_Memory.h
- HAL_Message_Log.h
- HAL_System.h
- HAL_Timer.h
- source
  - HAL.c
  - HAL_Communication.c
  - HAL_Memory.c
  - HAL_Message_Log.c
  - HAL_System.c
  - HAL_Timer.c
- 02_NanoPB ..... Nano Protocol Buffer
  - include
    - pb.h ..... NanoPB Definitionen
    - pb_decode.h ..... Methoden zum Dekodieren von Nachrichten
    - pb_encode.h ..... Methoden zum Kodieren von Nachrichten
    - stdbool.h ..... Definiert den Typ bool
  - source
    - pb_decode.c ..... Methoden zum Dekodieren von Nachrichten
    - pb_encode.c ..... Methoden zum Kodieren von Nachrichten
- 04_Protocol ..... Protokolldefinition für NanoPB
  - include
    - protocol.h ..... Gemeinsamer Header für NanoPB-Belange
    - tee.pb.h ..... Generierte Protokolldefinition
  - source
    - tee.pb.c ..... Generierte Protokolldefinition
    - tee.proto ..... Protokolldefinition
- 23d_TestExecutionServer ..... Testausführungsumgebung
  - include
    - modtest_interface.h ..... Interface zum Modultest

- tes_command_handler.h ..... Testablaufsteuerung
- tes_config.h ..... Konfigurationsparameter
- tes_control.h ..... Teststeuerung
- tes_decode_request.h ..... Nachrichtendekodierung
- tes_download.h ..... Herunterladen von Test und Daten
- tes_encode_result.h ..... Nachrichtenkodierung
- tes_init.h ..... Modultestinitialisierung
- tes_message_loop.h ..... Bearbeitungsschleife
- tes_reset.h ..... Zurückseten der Testausführungsumgebung
- tes_states.h ..... Zustandsbeschreibung Teststeuerung

module_test .....

- arithmetic.c ..... Beispielmodul Source
- arithmetic.h ..... Beispielmodul Header
- modtest_interface.c ..... Interface zur Testausführungsumgebung
- sample_module_test.c ..... Beispielmodultest

source .....

- main.c ..... HW-Initialisierung, Verbindungsaufbau
- tes_command_handler.c ..... Testablaufsteuerung
- tes_control.c ..... Teststeuerung
- tes_decode_request.c ..... Nachrichtendekodierung
- tes_download.c ..... Herunterladen von Test und Daten
- tes_encode_result.c ..... Nachrichtenkodierung
- tes_init.c ..... Modultestinitialisierung
- tes_message_loop.c ..... Bearbeitungsschleife
- tes_reset.c ..... urückseten der Testausführungsumgebung
- tes_states.c ..... Zustandsbeschreibung Teststeuerung