Stefan Pointner, BSc.

Entwicklung einer AUTOSAR konformen Multicore-Prototypen-Plattform

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Softwareentwicklung - Wirtschaft

eingereicht an der

Technischen Universität Graz

Betreuer

Dipl.-Ing. Dr.techn. Christian Kreiner
Dipl.-Ing. Georg Macher

Institut für Technische Informatik

Dipl.-Ing. Dr.techn. Eric Armengaud
AVL List GmbH

Graz, Oktober 2015

Kurzfassung

Die Leistungsanforderungen an einen Prozessor sind in den letzten Jahren in sämtlichen Bereichen sehr stark gestiegen. Im Consumer Bereich sind Mehrkern-Architekturen bereits weit verbreitet und eine Selbstverständlichkeit, vor allem in Bezug auf Desktop-PC's, Smartphones oder Fernseher. Auch im automotive Bereich wird immer mehr Leistung und Sicherheit gefordert. In den letzten Jahren führte dies zu einer hohen Anzahl an Steuergeräten innerhalb moderner Personenkraftwagen (PKW). Mittlerweile beinhaltet ein PKW 80, teilweise sogar mehr, Steuergeräte, da nahezu jede einzelne elektronische Komponente mit einem solchem ausgeliefert und eingebaut wird. Dies führt logischerweise zu einem enormen Netzwerk an Kabeln (1500 Meter in einem Kompaktwagen), welche durch die Komplexität hohe Kosten verursachen. Diese Komplexität reicht vom Aufbau, über den Einbau bis hin zur Wartung. Der Trend geht daher stark zu einer Minimierung der Anzahl der Steuergeräte, indem man mehrere Funktionen in einem leistungsstarken Steuergerät zusammenfasst, welche untereinander durch ein Bussystem (zum Beispiel Controller Area Network (CAN)) kommunizieren. Um eine Zentralisierung und Parallelisierung mehrerer Funktionen zu ermöglichen, wurden auch die Steuergeräte auf Mehrkern-Architekturen weiterentwickelt. Die Verteilung der Funktionen, die Sicherstellung der Synchronisation, die Isolierung von Fehlern und die Integration vorhandener Software und Module stellt Entwickler vor eine große Herausforderung.

Diese Arbeit beschäftigt sich mit der Integration vorhandener Softwaremodule, welche an das Automotive Open Systems Architecture (AUTOSAR) Konzept angelehnt sind, um eine standardisierte CAN Kommunikation in einem modernen Steuergerät zu ermöglichen. Ziel ist es, mit überschaubarem Aufwand vorhandene Module einzubinden um somit in kurzer Zeit eine Prototypenkonfiguration für ein bereits vorhandenes Forschungsprojekt zu erreichen. Als Betriebssystem wird dazu das Open-Source Betriebssystem Embedded Real Time Kernel Architecture (ERIKA) verwendet, welches auf dem Offene Systeme für die Elektronik im Kraftfahrzeug (OSEK)/Vehicle Distributed Executive (VDX) Standard basiert. Die eingesetzten Module, welche an AUTOSAR angelehnt sind, stammen aus dem Arctic Core Standardpaket v9.0.0 von ArcCore.
Zum Abschluss wurde der Erstellungsprozess der an AUTOSAR angelehnten Prototypenplattform hinsichtlich entscheidender Erfolgsfaktoren bewertet.

Abstract

The requirements for the performance of a processor in all areas increased dramatically in the last years. Multicore architectures are widespread and taken for granted in the consumer area, especially in terms of desktop PCs, Smartphones or TVs. In the automotive field higher performance is also demanded. This has led to a high number of electronic control units in a modern passenger car within the last few years. Nowadays a car includes 80, in some cases even more, electronic control units, since almost every single electronic component comes with a built in control unit. This leads to a huge, extensive and complex network of cables (1500 meters in a compact car). This means that a complex setup, installation and maintenance is necessary. Due to this complex and extensive hardware, the trend strongly leads towards minimizing the number of electronic control units. Therefore, several functions are integrated in one high-performance control unit. These electronic control units communicate with each other through a bus system (e.g. Controller Area Network (CAN)). To enable the centralization and parallelization of several functions, the first multicore electronic control units for embedded systems have been developed. The distribution of functions, ensuring the synchronization, the isolation of faults and the integration of existing software modules are big challenges for developers.

This thesis focuses on the integration of existing software modules, which are based on the Automotive Open Systems Architecture (AUTOSAR) concept, in order to enable a standardized CAN communication in a modern control unit. The aim is to achieve a prototype configuration by integrating existing modules for an existing research project with reasonable effort. The open source operation system Embedded Real Time Kernel Architecture (ERIKA) will be used. This real time operation system is based on the OSEK/VDX standard. The used modules, which are based on AUTOSAR, are provided from ArcCore's Arctic Core Standard Package v9.0.0.
To conclude, this thesis follows the aim to rate the creation process, based on key success factors, of the prototype platform.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Statuary Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The text document, which has been uploaded to TUGRAZonline, is identical with the present master thesis.

Graz, am 29.10.2015

Ort, Datum
Place, Date

Unterschrift
Signature

Danksagung

Diese Arbeit wurde am Institut für Technische Informatik der Technischen Universität Graz in Kooperation mit der AVL List GmbH durchgeführt.
Ich möchte mich bei Georg Macher, meinem Betreuer seitens dem Institut für Technische Informatik, für die tolle Unterstützung während der Durchführung dieser Arbeit bedanken. Des weiteren bedanke ich mich bei Eric Armengaud, meinem Betreuer seitens der AVL List GmbH für die Unterstützung. Für Fragen in der Firma standen mir Ismar, Martin und Harald stets zur Verfügung, Danke dafür. Danke auch an meinem Beurteiler Christian Kreiner, für die Unterstützung und das Ermöglichen dieser interessanten Arbeit.
Mein Dank gilt auch meiner Familie, besonders meinen Eltern, welche mir mein Studium ermöglicht haben und mich immer Unterstützt haben.
Ich möchte mich auch bei meiner Freundin Julia Grundner bedanken, welche mich während der letzten Jahre immer motiviert und unterstützt hat.
Ein besonderer Dank gilt auch Andreas Voraberger für die tolle und durchaus sehr erfolgreiche Studienzeit.

Stefan Pointner
Graz, Oktober 2015

Danke an

Andreas, Christian, Eric, Georg, Harald, Ismar, Julia, Martin, Sabrina

für die Unterstützung während der gesamten Arbeit.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Gordon Moore formulierte 1965 das sogenannte Mooresche Gesetz (engl. Moore's law), welches besagt, dass sich die Anzahl der Transistoren auf einem Prozessor alle ein bis zwei Jahre bei gleichbleibenden Kosten verdoppelt. In einem vierseitigen Paper hat Gordon Moore 1965 eine Zukunft mit Home-PCs, Mobiltelefone bzw. Smartphones und automatische Kontrollsysteme für Autos vorhergesehen. [38, 42]

Ein Jahrzehnt nach seiner Aussage über die stetige Verdoppelung der Transistoren, konnte man noch keine Zeichen über einen Stopp dieses Gesetzes erkennen. Somit gilt dieses Gesetz mittlerweile 50 Jahre. Durch diese Entwicklung stehen uns heute unzählige Formen von Computern, intelligente Geräte und Sensoren zur Verfügung. Um die kontinuierliche Leistungssteigerung eines Computers zu realisieren, wurde in den vergangen Jahren nicht nur der Takt erhöht, sondern auch die Transistoren verkleinert, um den Platz auf einem Chip effektiver zu nutzen und interne Strukturen parallelisiert, um eine Erhöhung der Instruktionen pro Taktzyklus zu erreichen. [37]

Um eine effektive Leistungssteigerung zu erhalten, stieß man mit Einkern-Prozessoren bald an deren Grenzen. Eine weitere Takterhöhung, in Verbindung mit neueren Transistoren wäre bei weitem nicht so wirkungsvoll wie ein Mehrkern-Prozessor. Wenn man von Leistung eines Prozessors spricht, meint man nicht nur die Taktfrequenz oder die Anzahl der ausführbaren Instruktionen pro Taktzyklus, sondern die Kombination aus beiden:

$$Leistung = Instruktionen\ pro\ Clock(IPC) * Frequenz \tag{1.1}$$

Wenn man leistungsstarke Prozessoren entwickeln will, muss man diese beiden Faktoren berücksichtigen. Heutzutage wird versucht einen parallelisierten Ansatz zu erreichen, anstatt einfach den Takt zu erhöhen. Ein Zweikern-Prozessor hat doppelt so viele Transistoren wie ein Einkern-Prozessor, jedoch wird durch neue Designs ein gleicher oder sogar geringerer Energiebedarf erreicht. Mit dem ersten Mehrkern-Prozessoren wurde eine neue Ära der Prozessorarchitektur eröffnet, welche mehr Leistung bei weniger Verbrauch realisiert. Durch diese Prozessoren entstanden neue Möglichkeiten in den verschiedensten Bereichen: Desktop-Computer, Notebooks, Server, Smartphones und viele mehr. Mithilfe von Multitasking wurden neue Wege für die Entwicklung von Anwendungen und Spiele geschaffen. Mit einfachen Einkern-Prozessoren wären heutige, einfache Anwendungen nicht mehr realisierbar. Durch die Mehrkernarchitekturen wurde auch das Mooresche Gesetz für

die Zukunft erweitert. [23]

Auch im Automobilbereich reichen heute oft Einkern-Prozessoren nicht mehr aus, da für ein Auto hohe Leistungs- und Sicherheitsanforderungen gelten, bzw. diese in den letzten Jahren stark gestiegen sind. Ein Auto besteht heutzutage aus vielen elektronischen Systemen, welche durch Faktoren wie Energieeffizienz, Emissionsreduzierung, Sicherheitssysteme, Komfort und Fahrspaß vorangetrieben werden. Somit wird einerseits versucht die Funktionalität zu erhöhen, gleichzeitig müssen jedoch Strategien entwickelt werden um Fehler zu vermeiden, insbesondere bei solch sicherheitskritischen Systemen. [51, 36]

Um die Komplexität eines Netzwerks einzelner Steuergeräte zu minimieren, wird momentan Versucht zu einer Backbone-Architektur zu wechseln, indem wenige, leistungsstarke und sichere Netzwerke für hochperformante Steuergeräte umgesetzt werden. Eine Herausforderung bei diesem Wechsel ist, durch das integrieren mehrerer Funktionen in ein Steuergerät, eine Reduktion der Kosten zu erreichen. Um durch dieses Zusammenführen von verschiedenen Applikationen und die dadurch notwendige Erhöhung der Performance die Betriebssicherheit aller Funktionen auf einer ECU nicht zu gefährden, müssen neue Architekturen und Strategien entwickelt werden. [51]

Ein durchschnittlicher PKW besitzt heutzutage 80 - 100 Steuergeräte [51], darunter ECUs für das Antiblockiersystem (ABS), Airbag und Antriebsstrang. Viele dieser Steuergeräte arbeiten in Kooperation untereinander und nutzen gemeinsame Ressourcen, zum Beispiel die Elektronische Stabilitätskontrolle (ESP), das Antiblockiersystem (ABS) und die Antriebsschlupfregelung (ASR) haben gemeinsame Inputs und Outputs. Ein Informationsaustausch solcher Systeme kann zur Verbesserung der Betriebssicherheit beitragen. Es ist zu beachten, dass der Austausch von der Daten verschiedener Steuergeräte eine Vielzahl an Schnittstellen und somit ein komplexes Gesamtnetzwerk hervorruft, wodurch eine Weiterentwicklung erschwert wird und möglicherweise anfälliger für Fehler ist. Bereits in einem Kompaktwagen werden 1500 Meter Kabel [53] benötigt, um alle elektronischen Bauteile miteinander zu verbinden. Durch die erhöhte Komplexität eines solchen Netzwerkes, steigen neben dem Aufwand für Test und Validierung auch die Gesamtkosten der Hardware und Verkabelung sowie Einbau- und Wartungskosten.[51] In Abbildung 1.1 sieht man ein komplexes Boardnetz eines modernen Oberklasse-PKW.

Überladene Netzwerke (Abbildung 1.1) sind nicht effizient, da solch ein Netzwerk mehr Test und Validierung benötigt. Die Wahrscheinlichkeit auftretender Systemfehler ist höher und die Gesamtkosten sind nicht optimal. Ein Lösungsansatz ist eine Aufteilung in verschiedene Domänen, welche mit einem gemeinsamen Backbone verbunden sind. Abbildung 1.2 zeigt eine mögliche Aufteilung in unterschiedliche Domänen, welche über ein Bussystem verbunden sind. Ein häufig verwendetes Bussystem hierfür ist der CAN-Bus (CAN). [Siehe Kapitel 3.1] Durch die Verwendung von intelligenten Sensoren, welche schon im Vorhinein Analysen und Tests durchführen, wird die Komplexität innerhalb einer Domäne ebenfalls reduziert.

Durch die steigenden Anforderungen setzten Hersteller im Automobilbereich zunehmend auf Mehrkern-Architekturen, somit lassen sich bei gleichen Taktfrequenzen Leistungsstei-

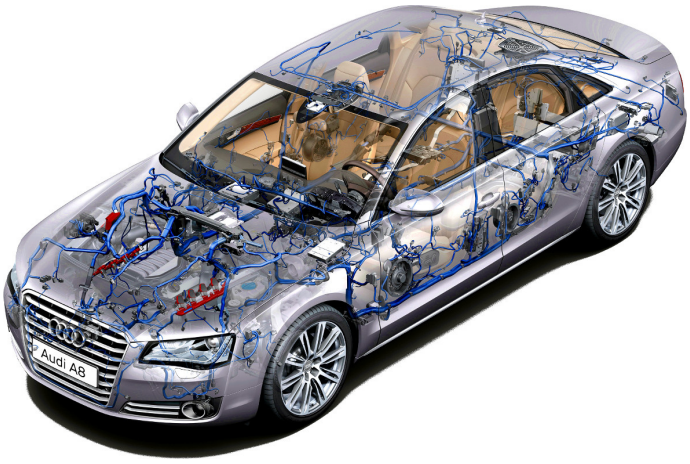

Abbildung 1.1: Boardnetz im modernen Oberklasse-PKW (Quelle: Audi AG)

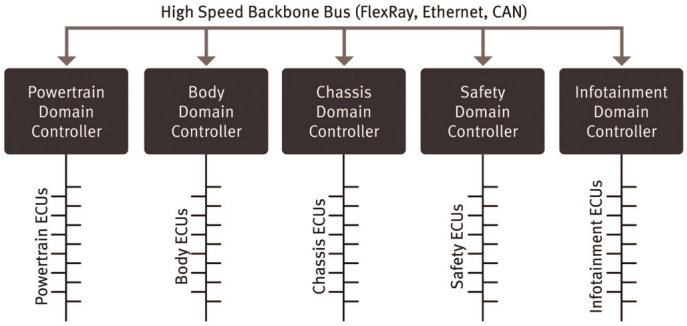

Abbildung 1.2: Boardnetz aufgeteilt in Domänen (Quelle: [57])

gerungen realisieren. Durch das zentralisieren verschiedener Funktionen werden Kosten eingespart und die Hardware-Komplexität verringert, jedoch stößt man hier auf einige Herausforderungen: Verteilung von Ressourcen, sicherer Informationsaustausch zwischen den einzelnen Kernen oder auch die Sicherstellung, dass ein Fehler auf einem Kern keine Auswirkungen auf andere hat ("Freedom from Interference").

Das Zusammenfassen mehrerer Funktionen in ein Steuergerät verringert einerseits die Komplexität im Boardnetz und erhöht diese andererseits gleichzeitig auf der Softwareebene. Eine physische Trennung einzelner Funktionen wird somit nicht mehr realisiert. Vielmehr muss diese Trennung nun in der Software umgesetzt werden oder mithilfe von Hardwaremodulen des Controllers gewährleistet sein. Eine Aufteilung von Aufgaben auf mehrere Kerne hat natürlich zur folge, dass Abhängigkeiten zwischen den Kernen entstehen, gemeinsame Ressourcen verwendet werden oder dass sich Fehler auf andere Teile auswirken können. Um diese Probleme so gut wie möglich lösen zu können, bedarf es von Beginn an einer gut durchdachten Planung und Schutzmechanismen, welche bei Echtzeitsystemen im automotive Bereich eine korrekte Ausführung, auch beim Eintreten des schlechtesten Falles sicherstellen und verhindern, dass sich ein Fehler auf andere Teile der Anwendung auswirkt. [43] Unterschiedliche Software-Standards unterstützen die Implementierung dieser komplexen Software.

Die Automobilindustrie erkannte bereits 1993 die Notwendigkeit, zusätzlich zu den Hardware-Standards, auch Standards für Software bereitzustellen, um eine Qualitätssteigerung der Anwendungsprogramme und eine Verminderung der Entwicklungskosten zu erreichen. Aus dieser Erkenntnis entstand der Standard "Offene Systeme für die Elektronik im Kraftfahrzeug (OSEK)", welcher Richtlinien für Betriebssysteme und eine, für den Automobilbereich passende, Kommunikationsbibliothek bereit stellt. Kurz darauf wurde aus OSEK, nach dem Zusammenschluss mit der französischen Initiative Vehicle Distributed Executive (VDX), OSEK/VDX. Diese Kooperation brachte nun weitere Pakete für die Standardisierung unterschiedlicher Bereiche im Automobilbereich hervor, worauf in Kapitel 3.2 näher eingegangen wird. [56]

Ein zusätzlicher Standard, welcher auf dem OSEK Konzept basiert, wurde 2003 durch eine Entwicklungspartnerschaft verschiedener Automobilhersteller und Zulieferer gegründet: Automotive Open Systems Architecture (AUTOSAR). Dadurch wurde die Möglichkeit geschaffen, vorhandene Module in ein Steuergerät unterschiedlicher Hersteller zu integrieren, ohne diese zu modifizieren. Somit existieren einheitliche Softwareschnittstellen, welche eine flexible Integration und einen variablen Austausch von Applikationen, in einem Netzwerk aus Steuergeräten, ermöglichen. Im Gegensatz zu OSEK, ermöglicht AUTOSAR seit der Version 4 die Verwendung von einem Mehrkern-Betriebssystem. [55]

1.1 Aufbau und Ziel dieser Arbeit

Ziel dieser Arbeit ist es, einer vorhandenen Einkern-ASW eine Mehrkern-BSW zur Verfügung zu stellen, um somit die bereits existierende Software auf einem Mehrkernsystem nutzen zu können. Dazu wird eine Prototypenplattform erstellt, welche an den AUTOSAR Standard angelehnt ist. Für diese Umsetzung wird Mithilfe des Arctic Core Standardpakets [6] von ArcCore [3], welches an den AUTOSAR Standard angelehnte Module bereitstellt, eine Verteilung von Softwarekomponenten auf mehrere Kerne vorgenommen. Hauptaugenmerk liegt in der Implementierung des Kommunikationsstack für den CAN Bus, welcher aus mehreren AUTOSAR Modulen besteht. Diese AUTOSAR Module werden in das INCOBAT Projekt zur Batterieüberwachung integriert. Im Kapitel 4 wird näher auf dieses Projekt eingegangen.

Mit einem AUTOSAR konformen Kommunikationsstapel wird die Qualität der Software erhöht. Des weiteren lässt sich so die Entwicklung unterschiedlicher Module verteilen und ein Zusammenführen wird durch die spezifizierten Schnittstellen unterstützt. Diese Arbeit bildet somit die Basis, für eine zum Teil AUTOSAR konforme Plattform zur Entwicklung von Prototypen. Eine Integration zusätzlicher Module bietet sich als zukünftige Erweiterung an, um somit Schritt für Schritt eine komplette AUTOSAR Architektur für die Prototypenentwicklung bereitstellen zu können.

1.2 Kontext der Arbeit zur Industrie

Diese Arbeit ist durch eine Kooperation des Instituts für Technische Informatik an der TU Graz und der AVL List GmbH entstanden.

Aufgrund der steigenden Anforderungen an eine ECU im automotive Bereich und die hohe Komplexität der Vernetzung in einem PKW, wird heutzutage versucht eine Zentralisierung unterschiedlicher Funktionen in einem Steuergerät zu erreichen. Um die dadurch benötigte Rechenleistung im gegebenem Einsatzgebiet bereitstellen zu können, werden vermehrt Mehrkern-Plattformen eingesetzt.

Auch die gängigen Standards der Softwarearchitektur im automotive Bereich haben diesen Trend erkannt und die Konzepte entsprechend erweitert bzw. angepasst. Durch diese Standards wird die Implementierung unterschiedlicher Funktionen der Steuergerätesoftware in Module geteilt, welche anhand definierter Schnittstellen untereinander kompatibel sind. Besonders im Bereich der Prototypenentwicklung ist diese Modularisierung interessant, da so schnell und mit geringem Aufwand nur jene Module implementiert und konfiguriert werden können, welche für eine Testplattform, sei es am Prüfstand oder in einem realem Fahrzeug, benötigt werden.

Im Zuge dieser Arbeit werden Teile solcher Module in ein bereits vorhandenes Projekt integriert, um so den Aufwand der Erstellung und den Nutzen einer Prototypenplattform mit Standardisierten Modulen bewerten zu können. Diese entwickelte Plattform dient als Basis für die Entwicklung von Prototypen.

Kapitel 2

Herausforderungen bei Mehrkern-Architekturen

Die Weiterentwicklungen von Prozessoren wurden in der Vergangenheit hauptsächlich durch eine Erweiterung der Befehlssätze, Erhöhung der Taktraten, Verbesserung der thermischen Verlustleistung (Thermal Design Power (TDP) [34]) und erhöhter Leistungsaufnahme realisiert. Die Effizienz einer derartigen Entwicklung wurde jedoch immer unattraktiver, wodurch der Trend in Richtung einer Reduktion der Taktraten und Leistungsaufnahmen ging. Da allerdings das Verlangen nach leistungsstärkeren Prozessoren nicht nachließ, wurde die Entwicklung der Mehrkern-Architekturen vorangetrieben. [50]

Seit 2005 wird unter anderem die Anzahl der Kerne eines Prozessors erhöht, um dem Mooresche Gesetz weiter zu folgen. [17]

Es muss jedoch beachtet werden, dass die Geschwindigkeitssteigerung durch Mehrkern Lösungen durch das Amdahl'sche Gesetz (Gleichung 2.1) begrenzt ist. Dieses besagt, dass die maximale Steigerung der Geschwindigkeit anhand von Parallelisierung durch den nicht parallelisierbaren Teil begrenzt ist:

$$S = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{s+\frac{1-s}{n}} \tag{2.1}$$

Diese Formel zeigt, dass sich die maximale Steigerung der Geschwindigkeit (S) durch die Beziehung zwischen dem parallelisierbarem Teil eines Programmes (p), dem seriellem Teil ($s = (1-p)$) und der Anzahl paralleler Prozessoren (n) ergibt. Durch einen kleinen Anteil an nicht parallelisierbaren Tasks stößt man somit schon auf die Grenzen der theoretischen Steigerung der Geschwindigkeit. Aus diesem Grund ist es wichtig, bereits von Grund auf ein Konzept zu entwickeln, welches eine gute Verteilung der Instruktionen zulässt um somit die Vorteile einer Mehrkern-Architektur optimal nutzen zu können. [54]

Man unterscheidet zwischen zwei Arten von Parallelität bei der Verteilung von Aufgaben innerhalb einer Mehrkern-Architektur. [43]:

- Datenparallelität

  Aufteilung auf mehrere Prozessoren, welche alle dasselbe machen (siehe Abbildung 2.1), somit benötigt man eine Ausführungszeit von etwa $\frac{1}{n}$, wobei n die Anzahl der Prozessoren ist.

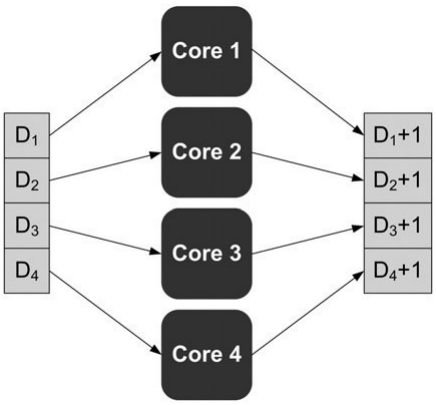

Abbildung 2.1: Parallele Ausführung gleicher Operationen (Quelle: [43])

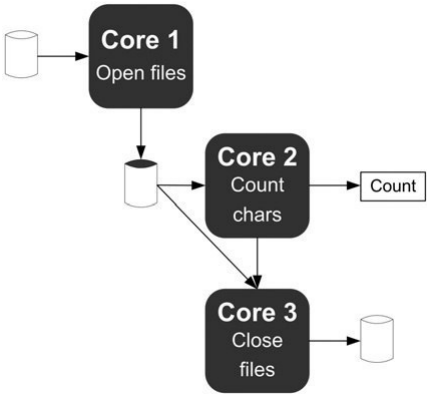

Abbildung 2.2: Parallele Ausführung verschiedener Operationen (Quelle: [43])

- Funktionsparallelität

  Hier erfolgt eine Aufteilung unterschiedlicher Aufgaben auf unterschiedliche Prozessoren (siehe Abbildung 2.2), die Ausführungszeit hängt hier von der Aufteilung der Funktionen ab.

Durch eine Verteilung auf mehrere Prozessoren ergeben sich natürlich auch Abhängigkeiten zwischen diesen, was in Abbildung 2.2 erkennbar ist. Das bedeutet, dass ein Teil des Programmes eine Berechnung durchführt und dessen Ergebnis wiederum von einem anderem Teil verwertet wird, daher gibt es einen Erzeuger und einen Verbraucher. Eine zusätzliche Herausforderung bei der Verteilung von Funktionen auf verschiedene Prozessoren ist der Austausch von Ressourcen. Da mehrere Kerne auf eine Ressource zugreifen können, muss dieser Zugriff geregelt werden. Dieses Problem hat man auch beim Einsatz von mehreren Threads auf einem Einkern-System, jedoch ist die Umsetzung von sicheren Zugriffen auf Mehrkern-Architekturen komplexer, da die einzelnen Kerne auch jeweils eine Abbildung von globalen Daten im eigenen Cache haben, welche nicht immer aktuell ist.

Als Programmierer muss man sich auf einem Einkern-System nicht weiter darum kümmern, bei Mehrkern-Systemen muss allerdings die Synchronisation zwischen den Kernen sichergestellt werden. [43]

2.1 Mehrkern-Architekturen im Consumer Bereich

Durch die rapide Entwicklung der verwendeten Geräte sowie Erhöhung der Software-Komplexität in Verbindung mit verkürzten Software-Lebensdauer, entstanden neue Herausforderungen, um vorhandene Einkern basierte Systeme auf Mehrkern-Architekturen zu portieren. Hier kommt es schon bei der Wiederverwendung als auch bei der Neuentwicklung einer Software zu Problemen. Einerseits ist es schwierig, die Vorteile von Mehrkern-Architekturen mit vorhanden Programmen zu nutzen ohne diese von Grund auf neu zu schreiben, andererseits unterstützen gängige Softwareentwicklungsprozesse meist zu spät die schnelle Entwicklung von Mehrkern-Architekturen. [49]

Man unterscheidet zwischen unterschiedlichen Typen von Mehrkern-Architekturen [28]:

- Heterogene Mehrkern-Architektur

  Hier kommen verschiedene Prozessoren für verschieden Aufgaben zum Einsatz. Die Prozessoren können sich zum Beispiel bei der Leistung und dem Verbrauch unterscheiden.

- Homogene Mehrkern-Architektur

  Bei homogenen Architekturen werden mehrere gleiche Prozessoren verwendet. Dieser Typ ist am weitesten verbreitet.

Ein großes Problem von Mehrkern-Prozessoren ist, dass vorhandene Software angepasst werden muss, um die Vorteile nutzen zu können. Oft kommt man um eine Neuentwicklung nicht herum. Dabei ist es wichtig, von Beginn an zu planen, wie man die Mehrkern-Prozessoren in einer Anwendung nutzen kann. Typische Einkern-Anwendungen haben einen sequentiellen Ablauf und sind auch dementsprechend programmiert. Eine Adaptierung solcher Programme, um eine effektive Verteilung zu erreichen, ist mit viel Aufwand verbunden. Im Consumer Bereich stellt dies oft eine nicht so große Herausforderung dar, da zum Beispiel bei Office Anwendungen Latenzen zwischen einzelnen Threads im Millisekundenbereich keine große Auswirkung haben. Problematisch wird dies jedoch in Echtzeitanwendungen. Es wird auch versucht Mithilfe von Kompilern eine Parallelisierung zu erhalten, dies funktioniert zwar zum Teil, ist jedoch noch weit vom Ideal entfernt. Für das Vorgehen beim Erstellen einer mehrkern-basierten Anwendungen bieten sich auch diverse Patterns an, welche oft ein gutes Grundgerüst für die Programmierung bieten. Das Mehrkern Pattern "Concurrent" sieht hierfür eine zentrale Klasse, welche die Synchronisationslogik übernimmt vor, um nach außen für andere Klassen eine Schnittstelle bereitzustellen. Somit ist die Synchronisation zentral in einer Klasse umgesetzt und man kann mit verschiedenen Klassen auf gemeinsame Ressourcen zugreifen. [10]
In folgendem Abschnitt wird näher auf diese Probleme im automotive Bereich eingegangen.

2.2 Mehrkern-Architekturen im Automotive Bereich

In einem sicherheitskritischen Kontext, wie dem automotive Bereich, kommt es zu weiteren Herausforderung beim Einsatz von Mehrkern-Architekturen. Um die gewünschte Leistung in einem Steuergerät bereitstellen zu können, entwickeln die Hersteller heutzutage ECUs mit Mehrkern Prozessoren, da eine Erhöhung der Taktgeschwindigkeit zu viel Leistung verbraucht und es auch zu einer drastischen Steigerung der Temperatur kommt, welche in einem geschlossenem Steuergerät nicht abgeführt werden kann. Bereits ab einer Taktrate von 300MHz reicht eine passive Kühlung in einem geschlossenem Steuergerät nicht mehr aus [12]. Zusätzlich muss man bedenken, dass die ECUs mit einer Umgebungstemperatur von -40°C bis zu +150°C zurecht kommen müssen. [51]

Durch eine Zusammenlegung von unterschiedlichen Funktionen in einem Steuergerät, erreicht man eine Kostenreduktion, da weniger ECUs, Kabel und Elektronikbauteile benötigt werden. Weiters minimiert sich auch der Aufwand für den Einbau und die Wartung der Netzwerke. Jedoch wird die Komplexität nicht nur reduziert, sondern gleichzeitig erhöht sich diese auf der Softwareebene im Steuergerät.

Bei sicherheitskritische Anforderungen kommen meist Echtzeitsysteme zum Einsatz. Dies bedeutet, dass nicht nur das Ergebnis einer Berechnung korrekt sein muss, sondern auch die Zeit, wann eine Operation ausgeführt wird und wie lange diese dauert. Um ein Echtzeitsystem garantieren zu können, bedarf es den Einsatz von effizienten Scheduling-Algorithmen in Kombination mit genauen Analysetechniken. Mit diesen Techniken muss das korrekte Verhalten beim Eintreten des schlechtesten Falles (Worst Case) bewiesen werden. Das Ziel von Echtzeit Scheduling auf Mehrkern-Prozessoren ist es, eine Applikation, bestehend aus mehreren Tasks so auszuführen, dass deren Zeitbeschränkungen immer erfüllt werden. Beim Echtzeit Scheduling unterscheidet man zwischen zwei Modellen: das periodische Taskmodell und das sporadische Taskmodell. Beim periodischem Modell werden die Tasks nacheinander in fixen Zeitintervallen abgearbeitet. Das sporadische Modell kann einen Task zu jeder Zeit ausführen. Es ist jedoch von Bedeutung die Tasks durch mindestens ein definiertes Zeitintervall zu trennen. [13, 21]

Im Gegensatz zu Mehrkern-Systemen aus dem Consumer Bereich, bei denen eine Steigerung der maximal ausführbaren Instruktionen während eines Zyklus im Vordergrund steht, hat bei Echtzeitsystemen die Einhaltung vorgegebener Zeitkriterien oberste Priorität. Diese Kriterien sind Mithilfe von exakten Hardwaremodellen zu bestimmen, was Aufgrund vieler verschiedenen Zustände, resultierend aus der Komplexität der Architektur, nur mit sehr viel Aufwand möglich ist.
Durch das Zusammenfassen verschiedener Funktionen in ein Steuergerät, können sich auch Fehler von einer Komponente in anderen auswirken. Diese Probleme hat man generell bei Mehrkern-Systemen, da durch einen Fehler auf einem Kern Interferenzen zu einem anderen entstehen können. So ist es möglich, dass sich in einem Steuergerät ein Fehler einer unkritischen Funktion, durch fehlende Schutzmechanismen, auf einen kritischen Teil auswirkt. [54]

Kapitel 3

Evolution der Automotiven Softwareentwicklung

Die steigenden Anforderungen, welche heutzutage im automotive Bereich gestellt werden, erhöhen nicht nur die Kosten, sondern auch die Komplexität der Elektronik im Hardware und Software Bereich. Ohne passende Standards und Normen, welche Schnittstellen für Kommunikation zwischen einzelnen Modulen bereitstellen, ist ein Austausch und eine Integration unterschiedlicher Komponenten nur mit viel Aufwand möglich. Der Trend geht hier ganz klar zu einheitlichen Definitionen, um eine Wiederverwendung und dynamische Integration von Systemen unterschiedlicher Hersteller zu ermöglichen. Dies führt gleichzeitig zu einer Reduktion der Kosten und der Komplexität.

Für die Kommunikation zwischen unterschiedlichen Steuergeräten wird meist der Controller Area Network (CAN) Bus verwendet, dessen Kommunikationsprotokoll in der Internationale Organisation für Normung (ISO) 11898 definiert ist. Dieser CAN Bus entstand um eine Reduzierung der Kommunikationskomplexität in einem Auto zu erreichen. [24]

Bereits 1993 erkannte die Automobilindustrie die Notwendigkeit der Entwicklung von Softwarestandards, um mit der steigenden Integration von elektronischen Bauteilen und Funktionen zurecht zu kommen. Offene Systeme für die Elektronik im Kraftfahrzeug (OSEK) wurde als Standard von Herstellern und Zulieferern der Automobilindustrie gegründet, um eine einheitliche Spezifikation für Betriebssysteme, Kommunikation und Echtzeitverhalten bereitzustellen. [56]

Eine Fortsetzung dieser Standardisierung wurde 2003, durch den Zusammenschluss von Automobilhersteller und Zulieferer, als Automotive Open Systems Architecture (AUTOSAR) gegründet, mit dem Ziel, eine Standardisierung für die Softwarearchitektur in einem Steuergerät zu entwickeln. Besonders in den letzten Jahren wurde AUTOSAR immer populärer und besteht mittlerweile schon in der Version 4.2, welche den aktuellen Trend der ECUs folgt und Spezifikationen für Mehrkern-Steuergeräte beinhaltet. [55]

3.1 Controller Area Network (CAN)

Bereits in den 1980er Jahren war die Komplexität der Vernetzung in einem Auto ein großes Problem, welches besonders für die Verkabelung und auch die Größe der Verbindungsstecker ein Maximum erreicht hatte. Aus diesem Grund entwickelte Bosch einen Netzwerkstandard, welcher besonders für die Anforderungen von Echtzeitsysteme entworfen wurde: den Controller Area Network (CAN)-Bus. Durch die Vorteile des CAN-Bussystems wurde dieser schnell in weitere Bereiche eingesetzt, dazu zählen die Automatisierung, Agrartechnik, medizinische Geräte oder auch Aufzüge. Der CAN-Bus ist auch sehr zukunftssicher, da viele namhafte Hersteller, wie zum Beispiel Intel, Motorola, Philips oder Siemens, Geräte produzieren, welche den CAN Standard verwenden. Ein CAN Netzwerk ist sehr kostengünstig herzustellen, da die Kommunikation über ein Kabel mit einem verdrilltem Adernpaar erfolgt. Der Einsatz des CAN-Bus reduziert die Komplexität der Verkabelung enorm, wie es auch in Abbildung 3.1 dargestellt ist. Durch die kurze Datenlänge des Bussystems, welche 8 Bytes beträgt, entstehen nur sehr geringe Latenzen bei der Übertragung. Die maximale Übertragungsgeschwindigkeit des CAN-Bus liegt bei 1 Mbit/s, welche jedoch aufgrund des Einsatzes von Carrier Sense Multiple Access / Collision Detect (CSMA/CD) begrenzt wird. CSMA/CD ist ein Mechanismus, um den Zugriff verschiedener Busteilnehmer auf den gemeinsamen Bus zu regeln. [60, 20]

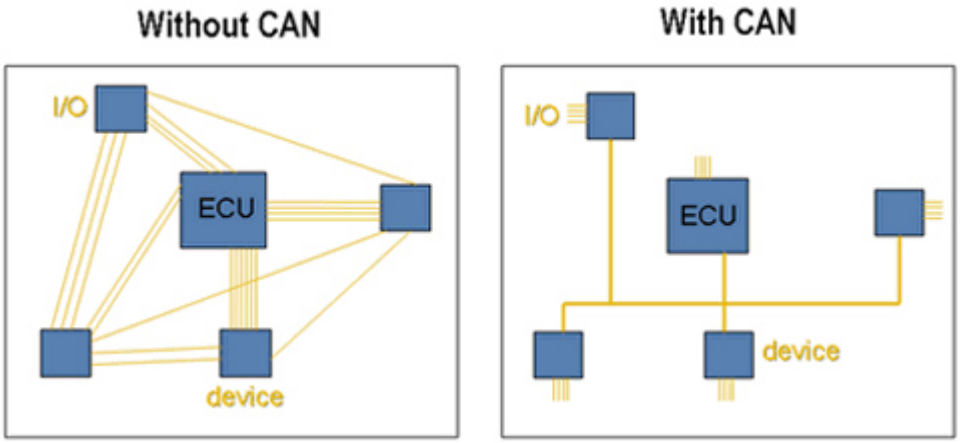

Abbildung 3.1: Vorteil von CAN [44]

Die Buslänge ist an folgende Bedingung geknüpft:

$$Buslänge <= 40 \text{ .. } 50m \ * \ \frac{1Mbit/s}{Bitrate} \tag{3.1}$$

CAN ist ein serieller Multi-Master Bus, welcher eine effiziente Übertragung von Daten ermöglicht. Beim Einsatz von CAN wird jede Botschaft ohne Angabe von Ziel- und

Absenderadresse an alle Busteilnehmer geschickt. Um die einzelnen Nachrichten identifizieren zu können, beinhalten diese eine eindeutige ID, anhand welcher jeder Teilnehmer selbst entscheidet, ob er eine Nachricht verwendet oder verwirft. Die Länge dieser ID beträgt 11 Bit beim Einsatz von Standard-Frames und 29 Bit bei Extended-Frames, beide Frame-Typen sind zueinander kompatibel. [60, 24]

3.1.1 Aufbau der Datenpakete

Der Aufbau eines CAN-Datenpakets wird in Abbildung 3.2 dargestellt. Ein Datenframe beginnt mit einem Start Bit (Start of Frame (SOF)) um den Beginn einer Nachricht zu markieren. Als nächstes folgt die ID der Botschaft, diese umfasst 11 Bit bei Standard-Frames und 29 Bit bei Extended-Frames. Diese ID dient einerseits zum zuordnen einer Nachricht und andererseits auch als Priorität. Je niedriger die ID ist, desto höher ist die Priorität der Botschaft. Wenn zwei Busteilnehmer gleichzeitig senden wollen, wird nur die höher priorisierte Nachricht akzeptiert. Abbildung 3.3 zeigt das Verhalten beim Auftreten einer Kollision von CAN-Datenpaketen.
Nach der ID folgt die Länge der eigentlichen Nutzdaten in Bytes einer Botschaft, diese wird als Data Length Code (DLC) bezeichnet. Als nächsten folgen nun die Daten der Nachricht gefolgt von einer 15 Bit Prüfsumme (Cyclic Redundancy Check (CRC)). Abschließend enthält das Paket ein ACK-Bit, welches den korrekten Empfang durch mindestens einem Busteilnehmer bestätigt, und die End of Frame (EOF) Bits, um das Ende der Nachricht zu markieren. [60]

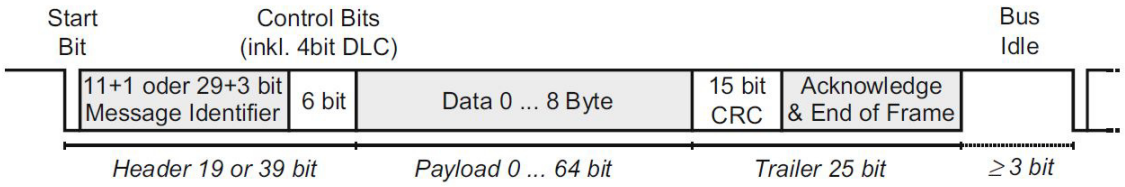

Abbildung 3.2: Aufbau eines CAN-Datenpaketes [60]

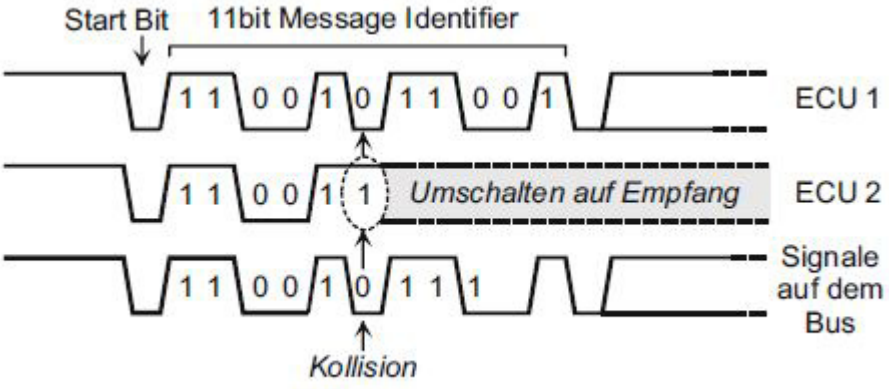

Abbildung 3.3: Kollision von CAN-Datenpaketen [60]

3.1.2 Ausblick

Durch die weite Verbreitung und Akzeptanz von CAN, wird diesem auch im AUTOSAR Konzept ein großer Teil gewidmet. Der genaue AUTOSAR-Kommunikationsstack für CAN wird im Zuge dieser Arbeit, mithilfe von Modulen aus dem Arctic Core Standardpakets [6] nachgebildet. Somit wird eine standardisierte CAN-Kommunikation für eine Prototypenplattform implementiert, welche in einem Forschungsprojekt zum Einsatz kommt.

3.2 OSEK/VDX

Die OSEK/VDX-Architektur besteht aus mehreren Modulen (Siehe Abbildung 3.4), deren Kern das OSEK OS bildet. Dabei handelt es sich um ein Echtzeit-Multitasking-Betriebssystem, welches die Möglichkeit bietet, Tasks zu synchronisieren und Ressourcen zu verwalten. Da von einem verteilten System ausgegangen wird, wird eine Interaktionsschicht zur Kommunikation benötigt, OSEK definiert hierfür die Schicht OSEK COM. Diese dient dazu, um einen Austausch von Daten zwischen einzelnen Tasks innerhalb eines Steuergeräts zu ermöglichen, zusätzlich wird eine Kommunikation zwischen Tasks unterschiedlicher Steuergeräte über ein Bussystem ermöglicht, man spricht hier von interner und externer Kommunikation. Um ein solches externes Bussystem effektiv und sicher nutzen zu können, bedarf es einer Überwachung, dazu wurde OSEK NM definiert. Die weitgehend statische Konfiguration, welche bereits während der Entwicklungsphase vorgenommen wird, ermöglicht ein effizientes System, welches nur geringe Anforderungen an Rechenleistung und Speicherplatz hat. Diese statische Konfiguration wird mittels dem OIL Format definiert, welches ein automatisches Erzeugen von Strukturen für die Taskverwaltung ermöglicht.

Die drei Hauptbestandteil OS, COM und NM sind jeweils eigenständig definiert und können daher auch unabhängig voneinander verwendet werden. Dazu existiert eine Un-

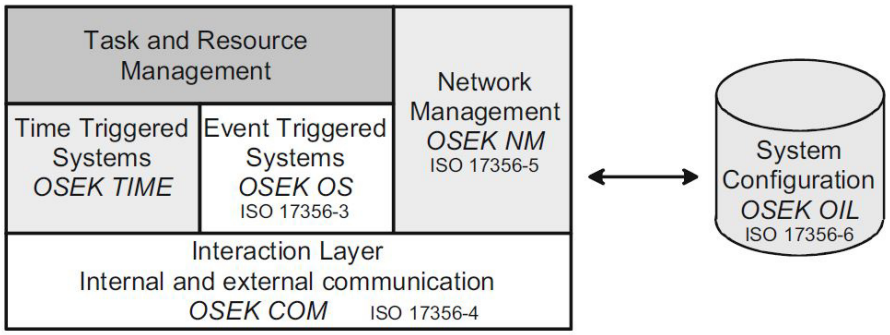

Abbildung 3.4: OSEK/VDX Konzept [61]

tergliederung in unterschiedliche Ausbaustufen. OSEK bezeichnet diese Ausbaustufen als Conformance Classes. [26, 61]

Da die Entstehung des OSEK Systems eng an den ereignisgesteuerten CAN Bus angelehnt war, wurden auch die Probleme solcher ereignisgesteuerten Systeme übernommen. Aus diesem Grund wurde eine zeitgesteuerte Betriebssystemkomponente OSEK Time und eine fehlertolerantere Kommunikationsschicht OSEK Fault Tolerant Communication (FTCOM) definiert. Diese beiden Konzepte wurden allerdings in der Praxis kaum angewandt, werden jedoch im AUTOSAR Konzept weitergeführt.

Da OSEK ein Standard ist, welcher von vielen verschiedenen Herstellern angewandt wird, ist es auch notwendig eine geeignete Schnittstelle für Tests bereitzustellen. Dieses Interface ist als OSEK RealTime Interface (ORTI) definiert worden und ermöglicht einen Zugriff auf die Laufzeitanwendung im Steuergerät durch externe Debugger. [61]

3.2.1 OSEK Operating System (OS)

Eine Unterteilung komplexer Software in einzelne Tasks ermöglicht eine scheinbare parallele Ausführung unterschiedlicher Aufgaben. Dazu definiert das OSEK Betriebssystem einen Scheduler, welcher zwischen den einzelnen Tasks umschaltet, um die Illusion eines parallelen Ablaufs zu schaffen.

Taskverwaltung

Im OSEK/VDX Konzept wird Grundsätzlich eine Unterscheidung zwischen zwei Arten von Tasks gemacht:

- Basic Tasks
- Extended Tasks

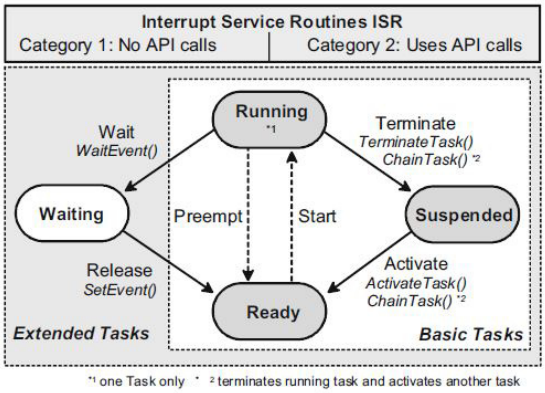

Abbildung 3.5: OSEK/VDS Task Zustandsmodell [61]

Da der Prozessor immer nur einen Task aktiv ausführen kann, werden den Tasks unterschiedliche Zustände zugeordnet (Siehe Abbildung 3.5):

- Running

  Ein Task im Zustand "Running" wird aktiv ausgeführt; es gibt immer nur einen Task in diesem Status.

- Ready

  Wenn alle Vorbedingungen um ausgeführt zu werden für einen Task erfüllt sind, erhält er den Zustand "Ready". Der Scheduler entscheidet, zwischen den Tasks mit diesem Status, welcher als nächstes ausgeführt wird.

- Suspended

  Ein Task in diesem Zustand wird aktuell nicht benötigt und kann bei Bedarf aktiviert (zu "Ready") werden.

- Waiting

  Dieser Zustand existiert nur bei den Extended Tasks und bedeutet, dass solch ein Task auf eine Aktion oder ein Ergebnis einer anderen Berechnung wartet.

Die Entscheidung, welcher Task aktuell bzw. als nächstes läuft, wird vom Scheduler anhand der Task-Priorität getroffen. Dieser Scheduler betrachtet alle Tasks mit dem Zustand "Ready" und den aktuellen "Running"-Task, demnach jene Tasks, welche aktuell ausgeführt werden können. Aus der Menge dieser Tasks, wird jener mit der höchsten Priorität gewählt und dieser erhält den Zustand "Running". Falls mehrere Tasks mit selber Priorität in Frage kommen, wählt der Scheduler, nach dem First In – First Out (FIFO) Prinzip, den ältesten aus (siehe Abbildung 3.6). Zusätzlich kann sich die Priorität während der Ausführung eines Tasks ändern, indem dieser mit dem Befehl `GetResource()` auf eine Ressource reserviert, dadurch erhält der Task die Priorität der Ressource. Nachdem der Task die Ressource mit `ReleaseResource()` freigibt, wird auch die ursprüngliche Priorität wiederherstellt. [48, 61]

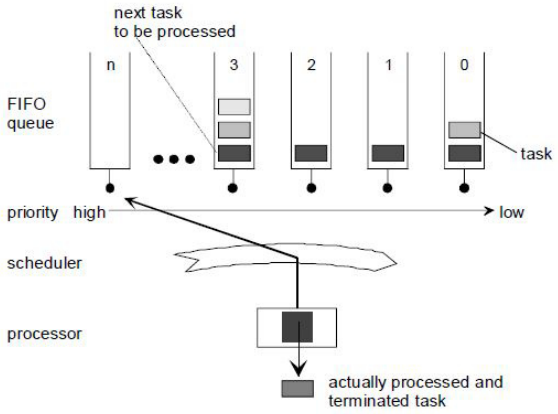

Abbildung 3.6: OSEK/VDS Scheduler Verhalten [48]

Im Allgemeinen wird Präemtives-Scheduling angewendet, das bedeutet, dass ein aktuell laufender Task unterbrochen werden kann und somit in den Zustand "Ready" versetzt wird. Dadurch ist es möglich, falls während einer Ausführung ein höher priorisierter Task bereit wird, diesen zu aktivieren, indem der Kontext des aktuellen Tasks gespeichert und deaktiviert wird. Durch dieses präemptive Vorgehen können während länger andauernder Tasks wichtigere und kürzere Tasks ausgeführt werden. Kurze Tasks werden oft auch nicht präemptiv umgesetzt um keine unnötige Verlängerung der Laufzeit durch den Scheduler zu produzieren.

Anwendungsmodi

Um einem OSEK Betriebssystem unterschiedliche Ausführungsweisen zu ermöglichen, werden im Konzept von OSEK Modi für das Betriebssystem ermöglicht, von denen es mindestens einen geben muss. Beim Start des Betriebssystem mittels `StartOS(OSMODE)` wird der gewünschte Modus übergeben. Dieser kann danach nur durch einen Neustart des Multitasking geändert werden, dazu wird `ShutdownOS()` aufgerufen, gefolgt von einem ernteutem `StartOS(OSMODE)`. Für ein Umschalten während des normalen Fahrbetriebes, zum Beispiel in das Notlaufprogramm, ist somit mit dieser OSEK Mechanismus weniger brauchbar. Alle Betriebssystemobjekte, wie zum Beispiel Tasks, Events oder Alarme, werden einem Betriebssystemmodus zugeordnet. Typische Beispiele dazu sind ein Modus für den Normalbetrieb oder ein Testmodus für den Gerätehersteller.

Durch diese Modi lassen sich erweiterte Tasks, welche im normal Betrieb nicht benötigt werden, mit wenig Aufwand durch einen Neustart hinzufügen. Die verschiedenen Modi erlauben auch getrennte Definitionen von internen Strukturen und optimieren dadurch den Speicherplatz und das Laufzeitverhalten. [48, 61]

Interrupt Service Routinen

Nicht präemptive Tasks werden zwar vom Scheduler nicht durch andere Tasks unterbrochen, jedoch kann es zu einem Interrupt kommen, wodurch auch solche Tasks unterbrochen werden. Man bezeichnet diese Unterbrechung als Interrupt Service Routine (ISR), welche durch ein Hardwaresignal des Mikrocontroller oder einem Peripheriebaustein auftritt. Wird ein Task durch eine ISR unterbrochen, wird dieser pausiert und nach dem Interrupt fortgesetzt. Im OSEK/VDX Konzept unterscheidet man zwischen zwei Kategorien von ISRs:

- ISR der Kategorie 1

  Die Ausführungszeit wird kurz gehalten, der unterbrochene Task wird danach fortgesetzt. Es werden keine Änderungen des Scheduling vorgenommen.

- ISR der Kategorie 2

  Hier wird ein Zugriff auf eine beschränkte Betriebssystem Application Programming Interface (API) zur Verfügung gestellt, um zum Beispiel einen Task aktivieren zu können.

Während einer ISR ist der Scheduler deaktiviert, die Reaktivierung erfolgt nach dem die ISR beendet wurde.

OSEK bietet auch die Möglichkeit alle Interrupts zu deaktivieren, falls dies innerhalb eines Tasks notwendig ist. Dazu existieren die API-Funktionen `DisableAllInterrupts()` bzw. `EnableAllInterrupts()`, falls man nur ISR der Kategorie 2 deaktivieren will: `SuspendOSInterrupts()` bzw. `ResumeOSInterrupts()`.

Events

Um eine Synchronisation zwischen einzelnen Tasks und Interrupt Service Routinen zu ermöglichen, bietet OSEK/VDX einen zentralen Ereignis-Mechanismus, die sogenannten Events, an. Bei einem Event handelt es sich um ein binäres Signal, welches bei der Systemkonfiguration einem Extended Task zugeordnet wird, somit kann nur dieser eine Task auf dieses eine Event mit `WaitEvent()` warten. Ausgelöst kann ein Event jedoch von einem beliebig anderem Task, sowohl Basic als auch Extended, oder einer ISR der Kategorie 2, mit dem Aufruf von `SetEvent()` werden. Während ein Task auf ein Event wartet, befindet sich dieser im Zustand "Waiting", nach dem Empfangen eines Events, wird der Task nun reaktiviert und läuft somit, unter Berücksichtigung der Priorität, weiter. Nach der Abarbeitung des Events setzt der Task dieses mit `ClearEvent()` zurück um neue Events empfangen zu können. [61]
Der Ablauf dieses Mechanismus ist in Abbildung 3.7 dargestellt.

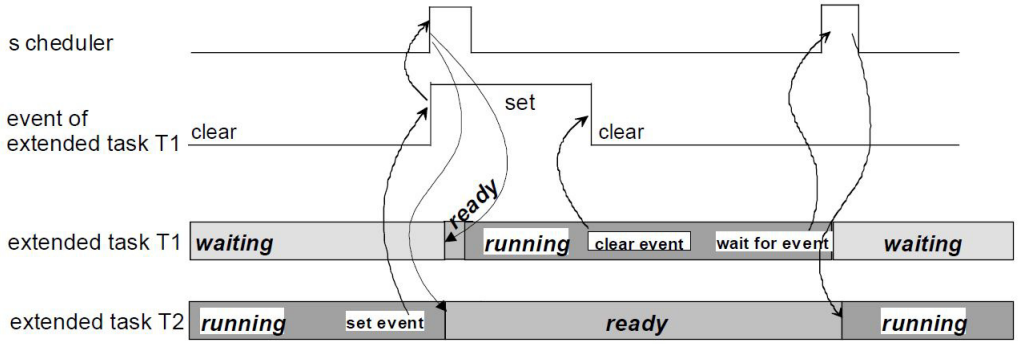

Abbildung 3.7: OSEK/VDS Event-Mechanismus [48]

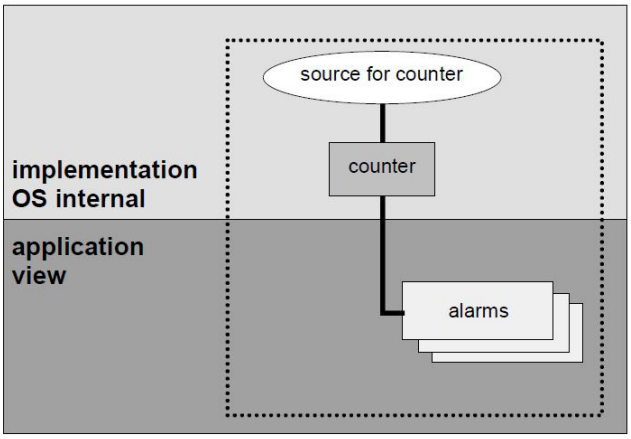

Abbildung 3.8: OSEK/VDS Alarm und Zähler Konstrukt [48]

Alarme und Zähler

Um periodisch wiederkehrende Events automatisch zu ermöglichen, hat man im OSEK/VDX System ein zweistufiges Konstrukt, bestehend aus Alarme und Zähler, geschaffen. Man hat die Möglichkeit einem Alarm unterschiedliche Aufgaben zuzuteilen, dazu zählen das Aktivieren eines Tasks, das Setzen eines Events oder das Aufrufen einer Anwenderfunktion. Damit ein Alarm zu bestimmten Zeiten auftritt, wird ein Zähler benötigt. Dieser basiert meist auf einen Zeitgeber (Timer) in einem Mikrocontroller, denkbar sind jedoch auch andere Werte als Basis, wie zum Beispiel eine Position eines Zahnrads oder auch ein Fehler innerhalb einer Applikation. Solche Zähler und Alarme werden ebenfalls während der Konfiguration mit einer OIL Datei definiert. [48, 56]
Abbildung 3.8 zeigt den Aufbau des Alarm-Zähler Konstruktes.

Auch hier bietet OSEK die Möglichkeit, durch den Aufruf von bestimmten API-Funktionen, Alarme manuell zu triggern. Die Funktionen `SetRelAlarm()` und

`SetAbsAlarm()` erlauben während der Laufzeit zu definieren, wann der Alarm ausgelöst werden soll. Hier wird der relative Wert zum aktuellen Zählerstand bzw. der absoluten Wert als Parameter übergeben. Mit dem Aufruf von `GetAlarm()` kann man den aktuellen Wert eines Alarms abfragen. Ein bereits gestarteter Alarm kann mittels `CancelAlarm` abgebrochen werden. [61]

Ressourcenverwaltung

Die Ressourcenverwaltung wird verwendet, um gleichzeitigen Zugriff auf gemeinsame Ressourcen durch unterschiedliche Tasks zu koordinieren. Will man auf eine Ressource zugreifen, muss diese zuerst durch den Benutzer angefordert werden. Erst nach einer exklusiven Zuteilung der Ressource, darf bzw. kann man diese auch verwenden. Somit wird gleichzeitiger Zugriff ausgeschlossen, man nennt dies "Mutual Exclusion". Wie bereits in Kapitel 3.2.1 erwähnt, passt das OSEK/VDX Konzept gegebenenfalls die Priorität eines Tasks an, wenn dieser Zugriff auf eine höher priorisierte Ressource erhält. Würde man diese Anpassung nicht machen, wäre es möglich, dass ein niedrig priorisierter Task eine Ressource beansprucht und bevor er diese wieder freigibt, vom Scheduler durch einen höher priorisierten Task unterbrochen wird. Dies würde dazu führen, dass der Task die Ressource solange beansprucht, bis alle höher priorisierten Tasks abgearbeitet worden sind und er selbst vom Scheduler reaktiviert wird.
Das Beispiel in Abbildung 3.9 zeigt den Mechanismus der Anhebung der Priorität des Tasks durch den Zugriff auf eine gemeinsame Ressource. Der Task T0 hat die höchste Priorität und T4 die niedrigste. Nachdem T4 die Ressource beansprucht, erhöht sich seine Priorität auf die der Ressource, somit wird T4 nur mehr durch T0 unterbrochen. Da T1 ebenfalls auf die Ressource zugreifen will, muss dieser Task nun nicht auf die Ressource warten, sondern bekommt den Zugriff sofort nachdem T4 diese freigegeben hat. Ohne diese Anpassung müsste T1 solange warten, bis T2, T3 und T4 nacheinander ausgeführt wurden. [48, 61]

3.2.2 OSEK Communication (COM)

OSEK COM beschreibt eine einheitliche Kommunikationsumgebung für interne Kommunikation, zwischen Tasks innerhalb einer ECU, und externe Kommunikation, über ein Bussystem zwischen mehreren Steuergeräten. Durch diese Umgebung wird eine Entkoppelung der Anwendungsschicht von der Kommunikationsschicht erreicht. Nachrichten und deren Eigenschaften werden statisch mittels einer OSEK OIL Datei definiert. Die offizielle OSEK COM Spezifikation beinhaltet eine detaillierte Beschreibung der API Funktionen [45].

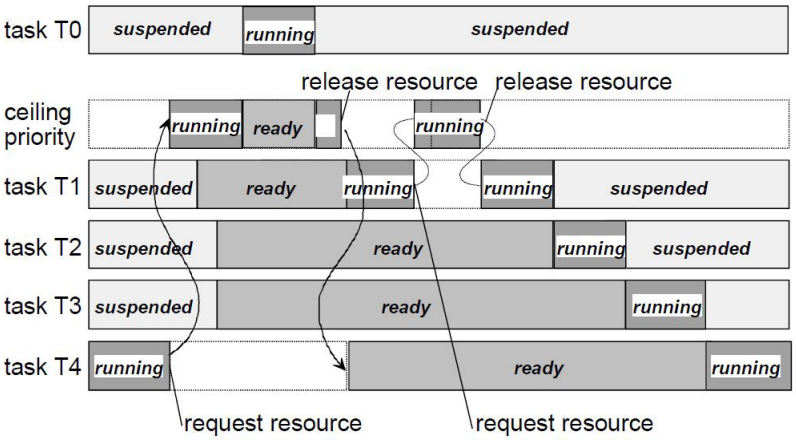

Abbildung 3.9: OSEK/VDS Ressourcenverwaltung mit Prioritätsanpassung [48]

Das OSEK COM Modell beinhaltet ganz bzw. teilweise folgende Ebenen (Siehe Abbilgung 3.10) :

- Interaktionsebene (Interaction Layer)

  Dieser Teil beinhaltet die OSEK COM API, welche Funktionen zum Senden und Empfangen von Nachrichten beinhaltet.

- Netzwerkebene (Network Layer)

  Diese Ebene behandelt, abhängig vom aktuellem Kommunikationsprotokoll, Nachrichtensegmentierung und Bestätigungen.

- Datenverbindungsebene (Data Link Layer)

  Hier werden den anderen Ebenen Schnittstellen bereitgestellt, welche den Transfer von Datenpaketen über ein Netzwerk ermöglichen.

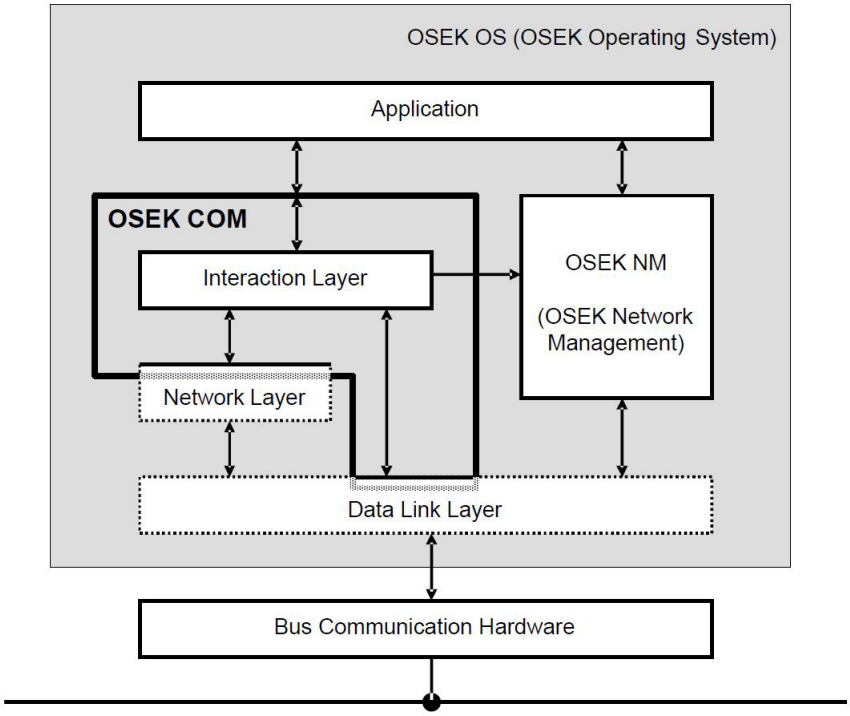

Abbildung 3.10: OSEK/VDS COM Modell [45]

3.2.3 OSEK Network Management (NM)

Das OSEK NM bietet zwei unterschiedliche Mechanismen zur Netzüberwachung an: direktes und indirektes Überwachen. Das direkte Netzwerkmanagement überwacht die Busteilnehmer, indem zusätzliche Botschaften verschickt werden. Dadurch wird eine Überwachung alle Busteilnehmer, welche ebenfalls ein Netzwerkmanagement enthalten, ermöglicht. Beim indirekten Netzwerkmanagement hingegen wird die Überwachung mittels der ohnehin versendeten Nachrichten erreicht. Jedoch können so nur jene Steuergeräte erfasst werden, welche mindestens eine Botschaft zyklisch senden. Zusätzlich muss die Herkunft dieser Nachricht eindeutig dem Steuergerät zuordenbar sein. Dadurch können beim indirektem NM auch Steuergeräte überwacht werden, welche keine NM Komponente besitzen.
Man geht davon aus, dass innerhalb eines Bussystems durchgängig, entweder indirektes oder direktes Netzwerkmanagement verwendet wird. Da es keine Zentralisierung innerhalb eines Netzwerkes zur Überwachung gibt, führt jede Electric Control Unit (ECU) seine eigene Netzwerküberwachung durch. [61]
Die Spezifikation [46] vom OSEK NM beinhaltet eine genau Beschreibung der Funktionsweise und der API-Funktionen.

3.2.4 OSEK Implementation Language (OIL)

Zur Konfiguration eines OSEK/VDX Betriebssystems wird das Dateiformat OSEK Implementation Language (OIL) verwendet. Eine OIL-Datei wird entweder manuell oder mit Hilfe eines Konfigurationstools erstellt. Ein Generator erzeugt anschließend aus der Konfigurationsdatei und der OSEK Bibliothek den Sourcecode der Anwendung. Zusätzlich gibt es die Möglichkeit eine ORTI Datei erzeugen zu lassen, welche unter anderem Informationen zu den einzelnen Tasks enthält, um somit ein Debuggen der Software zu erleichtern. [47, 61]

OSEK/VDX setzt beim Entwicklungsprozess voraus, dass das Zeitverhalten im System, die Auslastung des Speichers in Worst Case Szenarien bereits während der Entwicklung mit entsprechenden Methoden und Tools überprüft und verifiziert wurde. Dazu bietet sich messbasierte Analyse der maximalen Laufzeit (Worst Case Execution Time (WCET)) an, bei der Zeitmessungen in Kombination mit statischen Analysen der Programmstruktur durchgeführt werden. [52]

3.3 AUTOSAR

Mit der Gründung der Entwicklungskooperation "Automotive Open Systems Architecture (AUTOSAR)" entstand 2003 eine vollständige Softwarearchitektur für Steuergeräte. Jedoch wurde hier das Rad nicht neu erfunden, vielmehr übernahm man unterschiedliche Vorarbeiten aus der Automobilindustrie, darunter OSEK und Hersteller Initiative Software (HIS)[61], zusätzlich bediente man sich unter anderem den Kommunikationsstandards CAN und Local Interconnect Network (LIN). [55]

Das AUTOSAR Konzept sieht eine Entkoppelung der Anwendungs-Software (ASW) von der Basis-Software (BSW) vor, indem einzelne Module definiert sind, welche unabhängig voneinander entwickelt werden können. Diese Module lassen sich später mittels einen automatisiertem Konfigurationsprozess zu einem ganzen Projekt zusammenbauen. [59]

Die Architektur von AUTOSAR sieht eine Unterteilung in eine Hardwareabstraktionsebene und eine Serviceebene (Service-Layer) vor. Diese beiden Ebenen werden im Paket der Basis Software festgelegt. Die Serviceebene enthält unabhängige Dienste, wie das Betriebssystem, verschiedene Kommunikationsprotokolle und eine Speicherverwaltung. Die Interaktion zwischen der Basis Software Komponenten und der Anwendungssoftware erfolgt über die AUTOSAR Runtime Environment (RTE), welche hauptsächlich den Austausch von Daten regelt (Siehe Abbildung 3.11). Diese Architektur ermöglicht auch ein Verteilen einzelner Softwarekomponenten auf unterschiedliche Geräte, ohne dass dadurch funktionale Unterschiede entstehen bzw. Implementierungen angepasst werden müssen. Aufgrund der hohen Komplexität von AUTOSAR werden auch Prototypen zur Verfügung gestellt, welche einerseits die Machbarkeit nachweisen und andererseits als eine Unterstützung für die Implementierung dienen.

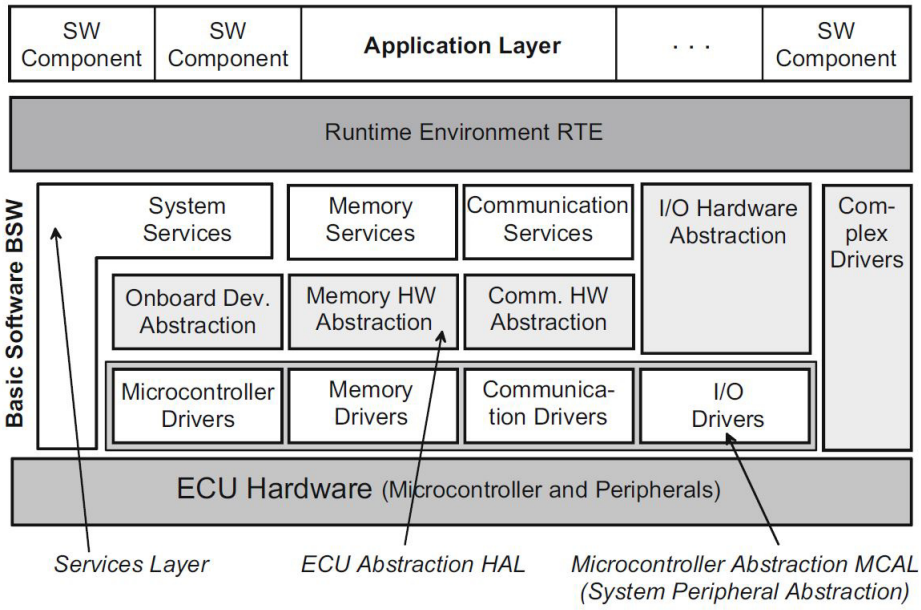

Abbildung 3.11: AUTOSAR Schichtenmodell [59]

3.3.1 AUTOSAR-Mehrkern

Da der Trend im automotive Bereich zur Zentralisierung verschiedener Funktion auf einer ECU geht, um dadurch Kosten zu sparen und die Komplexität der Hardwarevernetzung zu minimieren, wurde auch die AUTOSAR-Spezifikation angepasst. Dabei sieht das Konzept vor, dass auf jedem Prozessorkern eine eigenständige Instanz eines AUTOSAR-OS läuft. AUTOSAR bietet ebenfalls die Möglichkeit an, verschiedene Anwendungsgruppen zu definieren, um zum Beispiel Tasks und Ressourcen zusammenzufassen. Diese Gruppen werden nun zusätzlich einem CPU-Kern zugeordnet. Die Konfiguration wird statisch erstellt, wobei eine Verteilung der Lasten während der Ausführung nicht angedacht ist. Nach dem Start des Systems, bei dem die einzelnen Kerne zeitlich synchronisiert werden, läuft jeder Kern eigenständig und führt auch sein eigenes Scheduling aus. Mit verschiedenen API-Funktionen, wie zum Beispiel `SetEvent()` oder `ActivateTask()`, kann ein Betriebssystem auch Aktionen auf einem anderen Kern ausführen. Die Verwendung von Ressourcen hingegen ist zwischen den Kernen nicht möglich. Um aber dennoch eine Synchronisation zu ermöglichen, wurden die so genannten "Spinlocks" eingeführt. Wenn nun ein Task einen Spinlock mit `GetSpinlock()` beanspruchen will, welcher bereits von einem anderen CPU Kern gelockt ist, wird zyklisch versucht den Spinlock zu erhalten, bis dieser freigeben wurde. Zusätzlich kann man mit `TryToGetSpinlock()` einen nicht blockierenden Versuch ausführen, um einen Spinlock zu erhalten. [15, 59]
Eine weitere wichtige Funktion ist die sogenannten Inter-OS-Application Communication (IOC), also die Kommunikation zwischen den einzelnen Instanzen der Betriebssysteme.

Die IOC bietet eine Schnittstelle zum Senden bzw. Empfangen von Daten an. [15]

Bei der Verteilung der Funktionen und Ressourcen bedarf es einer guten Planung, um Konflikte zwischen den Kernen und Tasks zu vermeiden. Dabei empfiehlt es sich, alle Zugriffe auf die Peripherie auf einen Kern zusammenzufassen, um zu vermeiden, dass mehrere Instanzen gleichzeitig auf Ressourcen des Mikrocontroller zugreifen. Um anderen Kernen ebenfalls zugriff auf gewisse Funktionen zu ermöglichen, bietet sich der Einsatz der erwähnten IOC an. [32]

3.3.2 AUTOSAR Operating System (OS)

Der Betriebssystemkernel der AUTOSAR Architektur wird als AUTOSAR Operating System (OS) bezeichnet und basiert auf den OSEK Standard, welcher um verschiedene Schutzmechanismen für Speicher und Laufzeiten erweitert wurde. [11]
Das AUTOSAR OS übernimmt unter anderem die Taskverwaltung, die Interrupt Service Routinen, die Ressourcenverwaltung oder die Funktionalität von Alarmen und die dazugehörigen Zähler. Zusätzlich wird auch das Scheduling übernommen, jedoch gibt es hier Erweiterungen durch AUTOSAR. Der Unterschied beim Scheduling liegt darin, dass das OSEK OS ein ereignisorientiertes Multitasking-Konzept, welches auf Prioritäten basiert, einsetzt, während AUTOSAR OS eine Erweiterung um sogenannte Schedule-Tabellen umsetzt. Einen zeitgesteuerten Ablauf kann man beim OSEK OS mit dem Einsatz von Alarmen erreichen, was jedoch bei komplexen Abläufen schnell unübersichtlich wird. Die Schedule-Tabellen des AUTOSAR Konzepts ermöglichen hingegen eine Definition von Taskaufrufen mit der Hilfe von Zählern, welche sowohl auf einem Hardwaresignal oder einem Softwarezähler basieren können. Den Start einer Schedule-Tabelle kann man sowohl zyklisch als auch einmalig mit einem Offset definieren. [25]
Abbildung 3.12 zeigt den Aufbau einer Schedule-Tabelle.

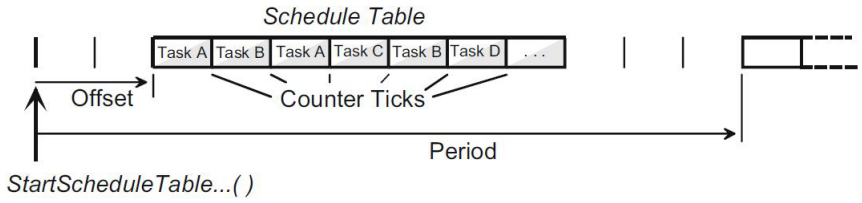

Abbildung 3.12: AUTOSAR Schedule-Tabelle [59]

Wie bereits erwähnt, bietet das AUTOSAR OS auch Schutzmechanismen zur Überwachung von Speicher und Ausführungszeiten an. Die Zeitüberwachung kann für jeden Task bzw. für jede ISR einzelnen konfiguriert werden und überwacht die Ausführungszeit (Execution Time Budget) vom Start bis hin zum Ende. Zusätzlich kann die Aufrufrate (Inter Arrival Rate) und Zeit der Ressourcennutzung (Locking Time) überwacht werden. Kommt es nun zu einer Überschreitung einer dieser Zeiten, hat man die Möglichkeit den Task neu zu starten, zu stoppen oder sogar das gesamte Betriebssystem herunterzufahren.

Auch das AUTOSAR OS bietet das Konzept der Anwendungsgruppen, welches ebenfalls von OSEK übernommen wurde. Diese Gruppen ermöglichen ein Zusammenfassen unterschiedlicher Betriebssystemobjekte, wie zum Beispiel Tasks, Events oder Alarme, welche bei der Ausführung einer Funktion eng zusammenwirken. Während innerhalb dieser Gruppe ein gegenseitiges Aufrufen und Nutzen von Ressourcen möglich ist, dürfen diese Objekte keine Teile anderer Anwendungsgruppen verwenden und können auch nicht von externen Gruppen aufgerufen werden. Um die Auslastung des Mikrocontrollers nicht zu überlasten, können einzelne Anwendungsgruppen als sichere Anwendungen (Trusted Applications) definiert und dadurch die Überwachungsmechanismen für diese deaktiviert werden. Innerhalb sicherer Anwendungen ist ein Zugriff auf andere Anwendungen erlaubt, jedoch unterliegen diese sehr strengen Zertifizierungsvorschriften.
Mikrocontroller mit integrierten Hardwareschutzmechanismen entlasten verstärkt die Auslastung durch die beschriebenen Schutzmechanismen. Durch das definieren von unterschiedlichen Betriebsmodi können Zugriffe auf bestimmte Bereiche ausgewählten Anwendungen vorbehalten werden. Man unterscheidet hier zwischen "Kernel Mode" und "User Mode". Das Betriebssystem hat durch den "Kernel Mode" vollen Zugriff auf die Hardware, währen andere Anwendungen nur eingeschränkte Zugriffsrechte haben. [59]
AUTOSAR begann ebenfalls mit dem von OSEK bekannten Konfigurationsformat OIL, jedoch wird seit der Version AUTOSAR 3.0 eine XML-basierte Konfiguration als Standard verwendet. [27]

Ablaufüberwachung

Das AUTOSAR Konzept sieht auch eine Gewährleistung der funktionalen Sicherheit nach ISO 26262 [31] vor, dazu dient der sogenannte "Watchdog Manager". Hierzu werden Sicherheitsrelevante Funktionen überwacht, um sicherzustellen, dass die Ausführungsreihenfolge korrekt ist. Diese Kontrolle ermöglicht ein Erkennen aller Laufzeitinterferenzen. [39, 33]

End-to-End (E2E) Sicherung

Eine Absicherung der Datenübertragung ist seit AUTOSAR 4.0, durch die E2E-Bibliothek möglich. Für diese Sicherung werden zusätzliche Daten, ein Zählerwert, eine Prüfsumme und die invertierten Nutzdaten, mit übertragen. Der Empfänger kann anhand dieser Daten sicherstellen, ob die Botschaft korrekt empfangen wurde. [59]

3.3.3 Basis-Software (BSW)

Grundsätzlich hat die BSW vier verschiedene Aufgaben: Systemdienste, Kommunikationsdienste, Speicherverwaltung und Hardware Ein- und Ausgaben. Jeder dieser Aufgabenbereiche streckt sich durch die verschiedenen Ebenen des Modells (Abbildung 3.11).

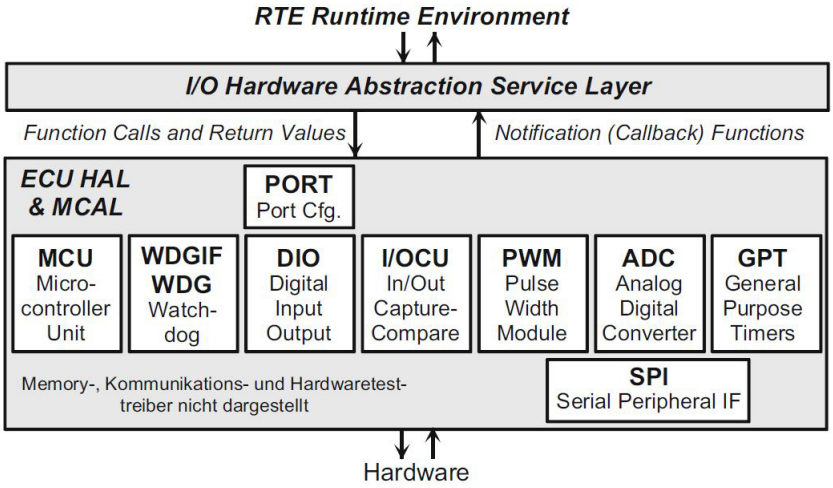

Abbildung 3.13: AUTOSAR Hardwareperipherie [59]

Die Funktion der untersten Ebene, der MicroController Abstraction Layer (MCAL), ist es, eine Schnittstelle zur Hardware selbst anzubieten. Da solch ein Interface sehr Hardwareabhängig ist, wird die MCAL vom Controller-Hersteller zur Verfügung gestellt. Die darüber liegende Schicht, die Hardware Abstraction Layer (HAL), wird vom Hersteller des Steuergeräts geliefert und dient als Interface zu den Controllern der Peripherie. Einige Hardwaretreiber, wie zum Beispiel Analog-Digital Konverter (ADC), Digital Input/Output (DIO) oder Pulsweitenmodulation (PWM), werden direkt von AUTOSAR definiert. Abbildung 3.13 gibt einen Überblick über die Hardwareperipherie. Als Ergänzung zu diesen Treibern sind noch Treiber für Speicherbausteine und Kommunikationscontroller, aber auch Module zum Selbsttest, verfügbar. Um auch auf Fehlerfälle reagieren zu können, stellt AUTOSAR noch das Modul Diagnostic Event Manager (DEM) bereit, welches beim Auftreten eines Fehlers von einer Funktion aufgerufen wird und einen zentralen Fehlerspeicher darstellt. [59]
Beim Einsatz einer Mehrkernarchitektur sieht AUTOSAR das Zusammenfassen der BSW-Module auf einem CPU Kern vor, idealerweise auf Core 0. Dadurch werden die Hardwarezugriffe und wichtige Kontrollfunktionen zentralisiert.

AUTOSAR-CAN

Zur Kommunikation in einem Kraftfahrzeug (KFZ) werden verschiedene Bussysteme eingesetzt, am häufigsten wird jedoch der CAN-Bus verwendet. Im AUTOSAR Konzept werden daher verschiedene Module für die Unterstützung des CAN-Bus definiert (siehe Abbildung 3.14).

Das CAN Modul ist der Treiber selbst, welcher an den CAN-Controller Schreib- sowie Lesebefehle schickt. Darüber liegt das CAN-Interface (CAN IF) Modul, welches verschie-

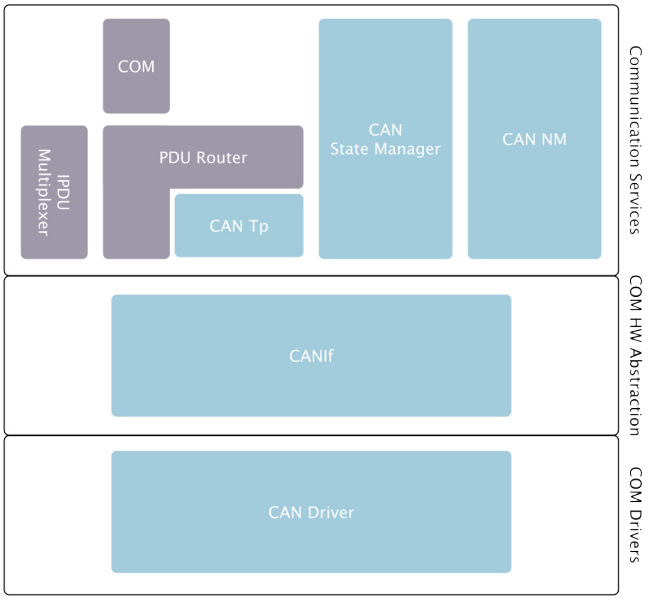

Abbildung 3.14: AUTOSAR CAN-Module [4]

dene API-Funktionen anbietet. Beim Versenden einer Botschaft über das CAN If Modul wird die Funktion `CanIf_Transmit()` verwendet, welche wiederum `Can_Write()` des CAN-Moduls aufruft, um somit die Nachricht in den Sendespeicher des CAN-Controllers zu kopieren und eine Anforderung zum Senden auszulösen. Da nach dem Aufruf von `CanIf_Transmit()` direkt zurückgekehrt wird, bekommt man nicht mit, wann die Botschaft nun tatsächlich versendet wurde. Im AUTOSAR Standard wurden für die Benachrichtigungen zwischen verschiedenen Schichten Callback-Funktionen definiert. Sobald eine Nachricht vom CAN-Controller versendet wurde, ruft der CAN-Treiber die Callback-Funktion `CanIf_TxConfirmation()` des CAN Interfaces auf. Das CAN Interface gibt wiederum dem Anwender Bescheid, indem die Callback-Funktion `User_TxConfirmation()` aufgerufen wird.
Für das Empfangen von Botschaften existieren ebenfalls solche Callback-Funktionen: der CAN Treiber meldet dem CAN Interface eine empfangene Nachricht mit `CanIf_RxIndication()` und das Interface dem Anwender mit `User_RxIndication()`. Der CAN Treiber bietet auch die Möglichkeit, einen Filter zu konfigurieren, um nur ausgewählte CAN Nachrichten anhand der CAN-ID zu akzeptieren und dem Anwender nur beim Eintreffen einer solchen Bescheid zu geben.
Das Lesen einer Nachricht mit dem CAN IF Modul erfolgt durch die API-Funktion `CanIf_ReadRxPduData`. Abbildung 3.15 zeigt einen Ausschnitt aus den API-Funktionen des CAN Treibers und des CAN Interfaces. [60]

Der weitere AUTOSAR-Protokollstapel bietet die Integration der Module PduR und COM an.

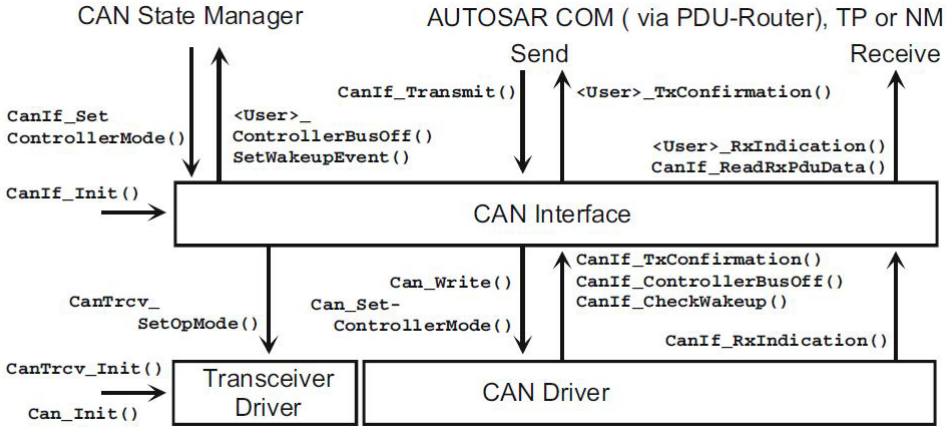

Abbildung 3.15: AUTOSAR CAN und CAN IF API Funktionen [60]

Das COM Modul basiert auf OSEK-COM und bietet eine signalorientierte API an, um einzelne Signale zu einer Botschaft (Protocol Data Unit (PDU)) zusammenzufassen und verwendet kein Transportprotokoll. Die Funktionen zum Senden bzw. Empfangen von Signalen lauten `Com_SendSignal()` und `Com_ReceiveSignal()`.
Um nun eine Botschaft aus mehreren Signalen über den entsprechenden Bus versenden zu können, bedarf es den PDU Router. Dieses PduR Modul leitet eine Nachricht an das entsprechende Interface bzw. dem Transportprotokoll Modul weiter. Im Beispiel des CAN Interfaces, ruft somit der PDU Router `CanIf_Transmit()` auf, um die Botschaft über den CAN Treiber zu versenden. Das AUTOSAR Konzept verwendet im gesamten Protokollstapel die selbe Grundstruktur, unabhängig vom verwendeten Bussystem. Beim Senden einer Nachricht wird diese über den PDU Router an das Interface des Bussystems übergeben. Beim Empfangen einer Botschaft werden die höheren Schichten durch Callback-Funktionen benachrichtigt, diese können somit die Botschaft lesen und verarbeiten. [60]
Abbildung 3.16 zeigt das Prinzip des AUTOSAR Protokollstapels.

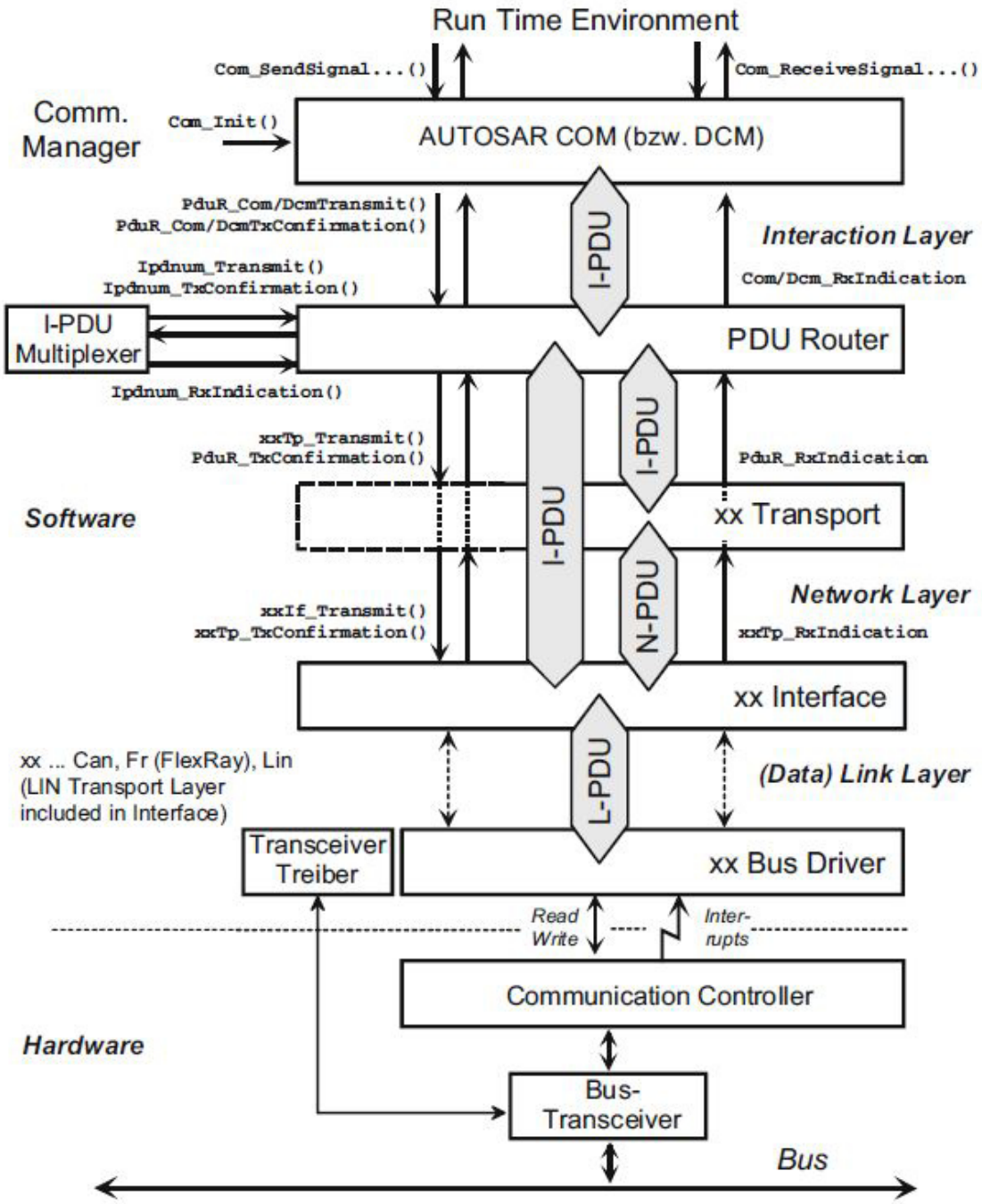

Abbildung 3.16: AUTOSAR Protokollstapel [60]

Kapitel 4

Projektumgebung

Ziel des INCOBAT (INnovative COst efficient management system for next generation high voltage BATteries) Projekts ist das Bereitstellen eines Innovativen und Kosteneffizienten Batteriemanagementsystems (Battery Management System (BMS)). Um diese Ziele zu erreichen, wird im INCOBAT Projekt ein Plattform-Konzept entwickelt, um einerseits Kosten und Komplexität zu reduzieren und andererseits Zuverlässigkeit, Flexibilität und Energieeffizienz zu erhöhen.

Das BMS spielt eine zentrale Rolle bei der Erfassung des aktuellen Ladezustands und des Gesundheitszustands einer Batteriezelle. Die Qualität dieser Datenerfassung hat direkten Einfluss auf die Batterie und somit auf die Reichweite des Fahrzeugs. Die Akzeptanz von Elektrofahrzeugen unterliegt einigen schwierigen Herausforderungen, wie zum Beispiel die Anschaffungskosten, die Reichweite oder die verschiedenen Lademöglichkeiten. Viele solcher Herausforderungen stehen in direkter Relation zum zentralen Element eines Elektrofahrzeuges, der Batterie.
Zu den angestrebten Ergebnissen des INCOBAT Projekts zählen unter anderen eine strenge Kontrolle der Batteriezellfunktionen, um eine höhere Reichweite zu ermöglichen und eine Kostensenkung des BMS. Ein weiteres wichtiges Ziel, um eine effiziente Integration in verschiedene Fahrzeugplattformen zu ermöglichen, ist die Entwicklung von modularen Konzepten für die Systemarchitektur und Partitionierung, Zuverlässigkeit sowie Verifizierung und Validierung. Um diese Ziele erreichen zu können, stützt sich das INCOBAT Projekt in erster Linie auf 12 technische Innovationen (TI), welche in vier Forschungsbereiche gruppiert (siehe Abbildung 4.1)[7]:

- Kundenbedürfnisse und Integrationsaspekte

  Dieser Bereich sorgt für eine richtige Identifizierung der Kundenanforderungen und ermöglicht eine effiziente Integration in verschiedene Plattformen.
  Um die Anforderungen der Kunden zu identifizieren, werden unterschiedliche Fahrstile, Verkehrsbedingungen sowie verschiedene Strecken betrachtet (TI-01). Zusätzlich sorgt dieser Teil für die Integration in ein Demonstrationsfahrzeug (TI-12).

- konsistentes Konzept und Spezifikation

  Zu den Zielen dieser Gruppe zählen die Optimierung der Systemarchitektur und derer konsistenten Beschreibung der Technologien und Hierarchien.

Dazu zählen ein Modell-Basiertes Systemdesign (TI-02), die effiziente Verteilung von Funktionen (TI-03) sowie die Integration unterschiedlicher Aufgaben in ein Steuergerät (TI-04).

- Elektrisch/Elektronisch (E/E) Kontrollsysteme

  Das Ziel der dritten Gruppe ist eine Verbesserung der Komponenten des E/E Kontrollsystems.
  Dazu zählt die Einführung der TriCore AURIX Plattform, um mehr Rechenleistung zur Verfügung zu stellen (TI-05) in Kombination mit einem intelligentem und integriertem Modulmanagement (TI-06). Auf der Seite der Software wird dies durch eine Modulare-Softwareplattform (TI-07) und einem optimiertem BMS-Algorithmus (TI-08) erreicht.

- Systemlaufzeitoptimierung

  Die letzte Gruppe hat als Ziel das Vertrauen auf die technische Lösung im Betrieb (TI-10), Funktionssicherheit (TI-09) und Zuverlässigkeit (TI-11) zu beweisen.
  Diese Gruppe der technischen Innovationen ist ein Laufzeit-Indikator der vorgeschlagenen Technologien und liefert dadurch Informationen, welche für die Integration und Validierung des Systems notwendig sind.

4.1 INCOBAT-ECU

Im INCOBAT Projekt wird versucht, einen seriennahen BMS-Demonstrator zu realisieren, welcher unterschiedlichen Projektpartnern erlaubt, Forschungen in diesem Bereich durchzuführen. Durch das Verwenden der neuesten Automotive-Komponenten kann die Dauer und der Aufwand, welcher betrieben werden muss um ein serienfähiges Produkt zu erreichen, verringert werden. Zusätzlich können Techniken, welche durch die Forschung entstehen bzw. optimiert werden, leichter unter realen Bedingungen angewandt und integriert werden.
Um eine optimale Integration verschiedener BMS-Funktionen zu erreichen und gleichzeitig die Komplexität der ECUs in einem Fahrzeug zu reduzieren, bedarf es eines leistungsstarken und modernen Mikrocontrollers. Aus diesem Grund setzt das INCOBAT Projekt auf die Verwendung des AURIX TC275T von Infineon. Dabei handelt es sich um einen Mehrkernprozessor, basierend auf einer innovativen Mehrkernarchitektur, welche die Ausführung unterschiedlicher Automotive Safety Integration Level (ASIL) Funktionen, bis zu ASIL-D, ermöglicht. Zusätzlich bietet die INCOBAT-BMS-ECU verschiedene Schnittstellen zur Kommunikation an, darunter mehrere Analog-Digital Konverter (ADC) Eingänge, CAN, USB und auch ein 100BaseT Netzwerk. Für Datenaufzeichnungen und Entwicklungssupport steht ein SD Kartenslot zur Verfügung.
Die Softwareentwicklung im Projekt ist modular aufgebaut, da unterschiedliche Funktionen von verschiedenen Herstellern zur Verfügung gestellt werden. Eine große Herausforderung für die Architektur der Software ist die Einhaltung und Gewährleistung der funktiona-

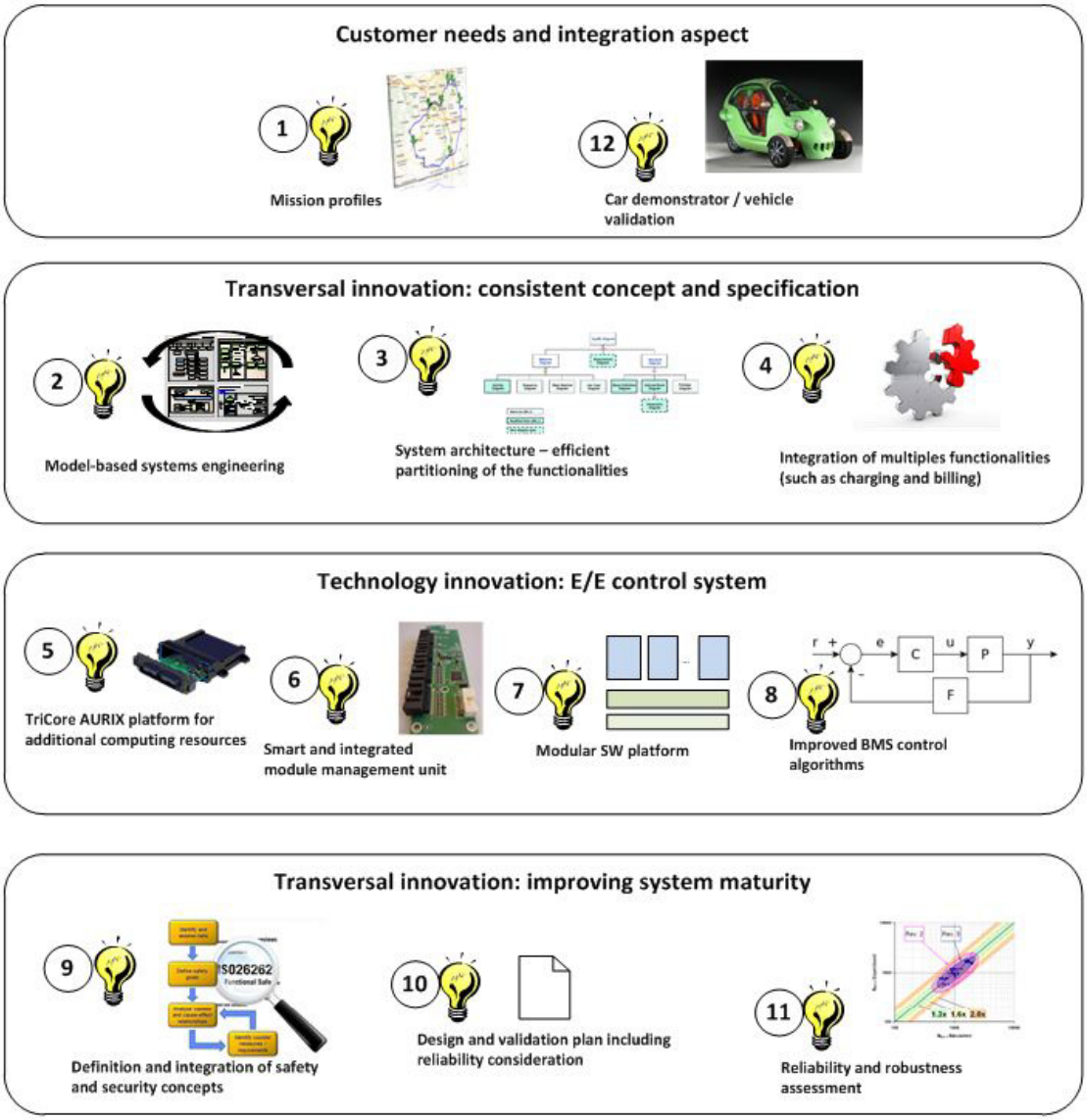

Abbildung 4.1: Die 12 technischen Innovationen (TI) gruppiert in vier Forschungsbereiche von INCOBAT [29]

len Sicherheit im automotive Bereich (ISO 26262 [31]), daher sind einzelne Module mit Schnittstellen zu definieren, welche eine korrekte Ausführung für das gesamte System garantieren. Dieser modulare Aufbau ist notwendig, um eine verteilte Entwicklung einzelner Teile zu ermöglichen. INCOBAT sieht hier den Einsatz einer Softwarearchitektur mit mehreren Schichten vor, welche Schnittstellen für den Zugriff auf die Hardware bereitstellen. [7]

Bei diesem Ansatz erkennt man eine Parallelität zu den bereits erwähnten Standards (siehe Kapitel 3).
Im Zuge dieser Arbeit wird der CAN-Protokollstapel nach AUTOSAR in das INCOBAT-Projekt integriert. So erreicht man einen modularen Aufbau des Kommunikationsstapel über mehrere Schichten hinweg nach einem weit verbreiteten Standard.

4.2 Beitrag dieser Arbeit zum INCOBAT Projekt

Die Optimierung eines BMS ist einer der wichtigsten Kriterien für die optimale Nutzung der Energie einer Batteriezelle, stellt jedoch eine große Herausforderung aufgrund der enormen Komplexität und dem Einsatz unterschiedlicher Technologien dar. Ein wichtiges Ziel des INCOBAT Projekts ist die Entwicklung einer intelligenten, zuverlässigen und modularen Rechnerplattform basierend auf einer Mehrkernarchitektur. Durch die zusätzliche Rechenleistung sind einerseits Optimierungen an den bereits existierenden Kontrollstrategien möglich, andererseits können erweiterte Messansätze integriert werden.
Diese Arbeit dient als Unterstützung für die Erstellung des modularen BMS, welches dem AUTOSAR Konzept folgt und somit die Qualität und Akzeptanz der Software erhöht. Durch die Entwicklung einer BSW, welche an dem AUTOSAR Standard angelehnt ist, wird die Implementierung weiterer Funktionssoftware erleichtert. Der Einsatz standardisierter Module auf einer Mehrkernplattform ermöglicht ein Verteilen vorhandener Software auf verschiedenen Kerne eines Prozessors. Zusätzlich wird der Wartungsaufwand aufgrund der Standardisierung minimiert, da spezifizierte Konfigurationsstrukturen zum Einsatz kommen.

Kapitel 5

Konzeptentwicklung

Das Ausgangskonzept des INCOBAT Projekts beinhaltet eine Hardwareabhängige CAN Implementierung, dies widerspricht jedoch den Zielen des Projekts, einen modularen Aufbau der Software bereitzustellen. Aus diesem Grund wurde ein neues Konzept erstellt, welches die Integration von AUTOSAR konformen Modulen aus dem Arctic Core Standardpaket in das INCOBAT-BMS, um den CAN Protokollstapel nutzen zu können, beschreibt. Das AUTOSAR Konzept ermöglicht, durch den modularen Aufbau, das Integrieren und Testen einzelner Module, wodurch sich der Entwicklungsprozess gut untergliedern lässt. Dieser modulare Aufbau trägt somit zur Erreichung der Ziele des INCOBAT Projekts bei.

Abbildung 5.1 zeigt das Ausgangskonzept des INCOBAT Projekts, dieses gliedert sich in folgende drei Teile:

- INCOBAT ASW

  Die Anwendungs-Software des INCOBAT Projekts umfasst das BMS. In der Ausgangssituation enthält dieses BMS bereits eine CAN Implementierung, welche jedoch Hardwarespezifisch umgesetzt wurde und somit nicht flexibel einsetzbar ist.

- INCOBAT MCAL

  Die MCAL stellt die Schnittstelle zur Hardware zur Verfügung, diese stammt in diesem Fall von Infineon.

- Hardware

  Dieser Teil stellt die Hardware dar, welche im Zuge dieser Arbeit eingesetzt wurde.

Der Nachteil dieses Konzepts ist die Hardwarespezifische Implementierung der Funktionen für den CAN-Bus. Dadurch ist ein Austausch der Hardware nur mit viel Aufwand möglich, dies gilt auch für die Wartung der Nachrichten aufgrund fehlender Spezifikationen.

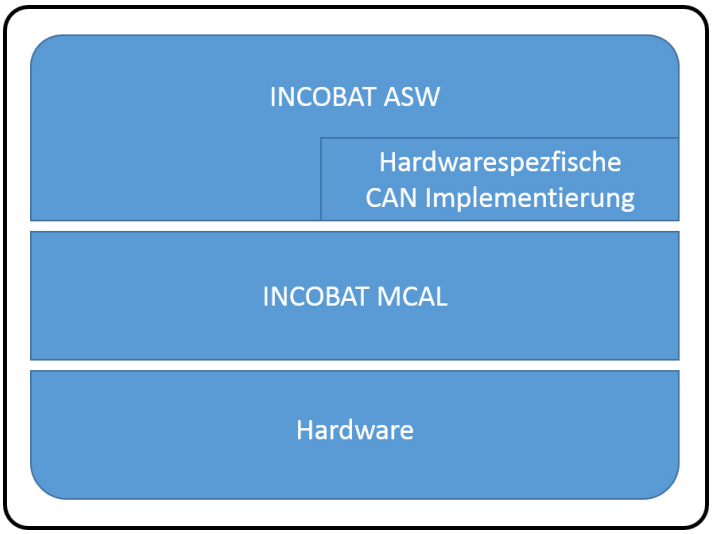

Abbildung 5.1: Ausgangskonzept

Das neu entwickelte Konzept sieht einen AUTOSAR konformen Aufbau der CAN Kommunikation vor um so eine modulare Software zu erhalten (siehe Abbildung 5.2).

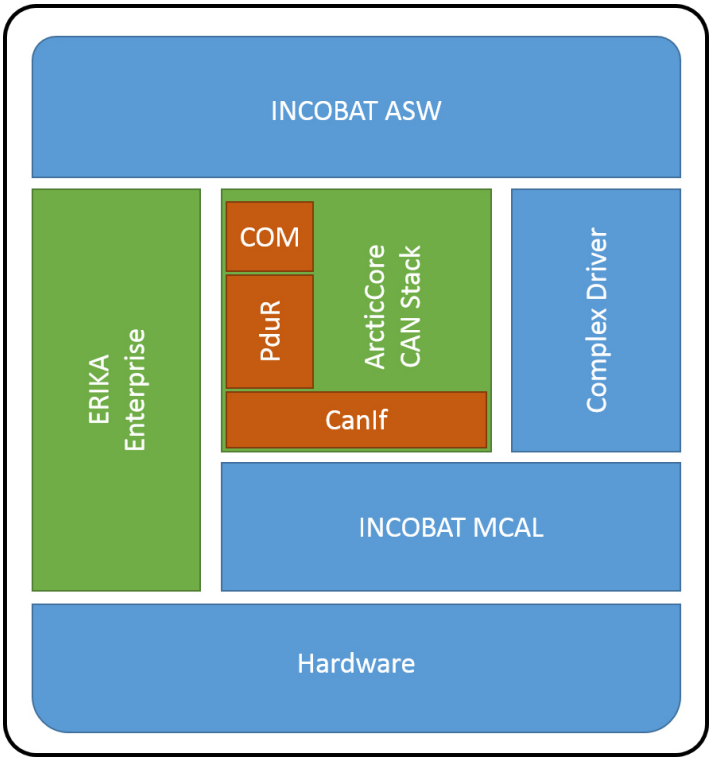

Abbildung 5.2: Entwickeltes Konzept

Die einzelnen Komponenten dieses entwickelten Konzepts beinhalten folgende Aufgaben:

- INCOBAT ASW

  Dieser Teil umfasst wieder das BMS des INCOBAT Projekts, welches nun jedoch über den AUTOSAR konformen Aufbau der CAN Kommunikation Hardware-unabhängig kommuniziert.

- ERIKA Enterprise

  Als Real Time Operating System (RTOS) kommt ERIKA Enterprise zum Einsatz, dadurch wird eine Verteilung unterschiedlicher Funktionen auf verschiedene Kerne des Mikrocontroller ermöglicht.

- ArcticCore CAN Stack

  Der ArcticCore CAN Stack umfasst die Integration und Konfiguration der an den AUTOSAR Standard angelehnten Module, um eine modulare, standardisierte CAN-Kommunikation zu ermöglichen. Dies umfasst die Module CanIf, PduR und COM.

- Complex Driver

  Die Complex Driver beinhalten zusätzliche Software Module, welche nicht durch AUTOSAR definiert sind. Diese werden im Zuge dieser Arbeit nicht näher betrachtet.

- INCOBAT MCAL

  Die eingesetzte MCAL bleibt unverändert und wird von den entsprechenden Module des CAN-Aufbaus verwendet.

- Hardware

  Die eingesetzte Hardware bleibt ebenfalls wie im Ausgangskonzept.

5.1 INCOBAT Integration

Der modulare Softwareaufbau des INCOBAT Projekts ermöglicht eine getrennte Entwicklung einzelner Module, welche definierte Schnittstellen besitzen, um sie mit überschaubarem Aufwand integrieren zu können. Für die CAN Kommunikation sollen diese Module nach dem bekannten AUTOSAR Standard entwickelt und integriert werden, um somit eine Basis für die weitere Entwicklung nach AUTOSAR zu bilden. Die Vorteile dieses Vorgehens liegen klar auf der Hand, denn durch das anerkannte und bewährte AUTOSAR Konzept steigt die Qualität der Software des BMS von INCOBAT. Zugleich ist keine eigene Softwarearchitektur zu entwickeln und der Austausch zwischen den einzelnen Projektbeteiligten hinsichtlich der Software wird immens erleichtert. Das Open-Source Standardpaket von Arctic Core bietet sich für diese Integration perfekt an, da dieses an den aktuellen AUTOSAR Standards angelehnt ist.

INCOBAT BMS Betriebssystem

Das Betriebssystem ist die zentrale Komponente der Software, kümmert sich um die Verwaltung von einzelnen Aufgaben sowie Ressourcen, und hat die Echtzeitfähigkeit des gesamten Systems sicherzustellen. Verzögerungs- und Bearbeitungszeiten dürfen daher ein konfiguriertes Limit auf keinen Fall überschreiten.

Als Betriebssystem des Battery Management System (BMS) von INCOBAT kommt "ERIKA (Embedded Real Time Kernel Architecture) Enterprise" zum Einsatz, dabei handelt es sich um ein freies Echtzeitbetriebssystem. ERIKA Enterprise basiert auf den bereits erwähnten OSEK Standard und ist auf dem Infineon AURIX ausführbar [19].
Zu den Eigenschaften von ERIKA Enterprise zählen unter anderen [18]:

- OSEK/VDX Zertifizierung
- Echtzeit Kernel
- RTOS-API für: Tasks, Events, Alarme, Ressourcen, Anwendungsmodi
- Preemptives und Nicht-Preemptives Scheduling
- Verwaltung geteilter Ressourcen
- periodische Ausführungen mit Alarme

ERIKA Enterprise ist kein AUTOSAR RTOS, jedoch wird versucht die Implementierungen an AUTOSAR anzulehnen um den aktuellen AUTOSAR Anforderungen gerecht zu werden. Da ERIKA bereits bevor AUTOSAR eine Mehrkernumgebung anbot, gibt es in diesem Zusammenhang zum Teil große Unterschiede.

5.2 Arctic Core

Arccore bietet ein komplettes Produktportfolio an, um die Softwareentwicklung und Konfiguration von eingebetteten Systemen im automotive Bereich zu unterstützen. Zu den Hauptprodukten zählt das Arctic Studio, eine vollwertige Entwicklungsumgebung für eingebettete automotive Systeme, welches auf dem AUTOSAR Standard basiert. Ein weiteres Hauptprodukt von Arccore ist die transparente und variable Plattform Arctic Core, welche einen AUTOSAR Stack zum Entwickeln von Steuergeräten implementiert. Das Arctic Core Standardpaket beinhaltet alle notwendigen Funktionen, welche in einer ECU im automotive Bereich benötigt werden. Dazu zählen Kommunikations-, Diagnose- und Sicherheitsfunktionen. Die Flexibilität dieses Pakets erlaubt es nur jene Funktionen zu verwenden, welche auch tatsächlich benötigt werden. Dadurch können die Kosten für die Hardware gering gehalten werden und zusätzlich wird die Komplexität nicht unnötig erhöht. Da Arctic Core auf den bekannten AUTOSAR Standard basiert, werden auch die Anforderungen der Automobilhersteller und Zulieferer abgedeckt, zusätzlich bleibt das Paket auch zukunftssicher. Die Entwicklung von Arctic Core erfolgt in Übereinstimmung mit den Anforderungen der sicherheitskritischen und komplexen eingebetteten Systemen im automotive Bereich. Für diese Arbeit wurde Arcitc Core in der Version 9.0.0 verwendet, welche an AUTOSAR 4.1.1

angelehnt ist. Des weiteren werden die Konformitätsklassen ICC1 bis ICC3 unterstützt, welche unterschiedliche Implementierungsgrade definieren. [6]

5.2.1 Arctic Core CAN

Zu den AUTOSAR-basierten Kommunikationsmodulen von Arctic Core zählt auch der generische Protokollstapel (siehe Kapitel 3.3.3) samt der Module für die CAN Implementierung, welche im Zuge dieser Arbeit in das INCOBAT Projekt eingliedert wurden.
Zu diesen Modulen zählen (siehe Abbildung 3.14):

- CAN Treiber (Can)

  Der Treiber ist Teil der untersten Schicht des AUTOSAR Modells, führt Zugriffe auf die Hardware durch und stellt dafür eine unabhängige API für die darüber liegenden Schichten zur Verfügung. Zum Senden und Empfangen von Botschaften stehen mehrere Nachrichtenobjekte zur Verfügung, welche sich die einzelnen CAN Controller teilen. In der Konfiguration des CAN Treibers werden Eigenschaften, wie die Baudrate oder der Zeitgeber, festgelegt.

- CAN Interface (CanIf)

  Für die Kommunikation zwischen dem CAN Treiber der unteren Schicht und dem PDU Router bzw. dem COM Modul der oberen Schichten, steht das CAN Interface zu Verfügung. Hier ist es auch Möglich, Filter bezüglich der Akzeptanz von Nachrichten anhand deren ID zu definieren.

- PDU Router (PduR)

  Um dem Anwender einen generischen Zugriff auf unterschiedliche Kommunikationsmodule zu ermöglichen, wird der PDU Router eingesetzt. Dieser gibt API-Funktionsaufruf des Anwenders an das entsprechende Interface bzw. dem Transportprotokoll-Modul weiter. Die Routingpfade werden statisch konfiguriert und können nicht während der Laufzeit erweitert bzw. modifiziert werden.

- COM

  Das COM Modul ermöglicht dem Anwender eine Kommunikation mit einzelnen Signalen anstatt mit gesamten Nachrichten. Diese Modul ist ebenfalls protokollunabhängig. Die Konfiguration beinhaltet einzelne Signale und die Verweise zu den entsprechenden PDUs.

5.2.2 Arctic Core Konfigurationsmodel

Die Konfiguration der einzelnen Module erfolgt in den jeweiligen C-Source und Header Dateien. Im folgenden einige Ausschnitte aus den Modulkonfigurationen, weitere Details werden in der Arctic Core Dokumentation [5] behandelt.

Quelltext 5.1: CAN Interface Konfigurationsstruktur [6]

```
typedef struct
{
    uint32 CanIfConfigSet;

    /** Size of Rx PDU list. */
    uint32 CanIfNumberOfCanRxPduIds;
    /** Size of Tx PDU list. */
    uint32 CanIfNumberOfCanTXPduIds;

    uint16 CanIfNumberOfTxBuffers;

    // Containers
    /* Tx Buffer List */
    const CanIf_TxBufferConfigType *CanIfBufferCfgPtr;
    /** Hardware Object Handle list */
    const CanIf_InitHohConfigType *CanIfHohConfigPtr;
    /** Rx PDU's list */
    const CanIf_RxPduConfigType *CanIfRxPduConfigPtr;
    /** Tx PDU's list */
    CanIf_TxPduConfigType *CanIfTxPduConfigPtr;

} CanIf_InitConfigType;
```

CAN Interface Konfiguration

Die Konfiguration des CAN Interfaces erfolgt großteils in der `CanIfInifCfg` (siehe Quelltext 5.1), welche alle PDUs und deren Konfiguration enthält. Für jeden CAN Controller, welcher zum Versenden von Nachrichten verwendet wird, muss ein Buffer deklariert werden, welcher auf ein Hardware-Sende-Objekt (Hardware Transmit Handle (HTH)) referenziert. Für das Empfangen existiert eine ähnliche Konfiguration (Hardware Receive Handle (HRH)).
Ein `CanIfTxPduCfg` Objekt wird für jede Botschaft, welche versendet werden soll, angelegt. Diese Konfiguration verweist auf eine PDU und einen Buffer.
Ähnlich der Transmit-PDU-Konfiguration, muss für eingehende Nachrichten ein `CanIfRxPduCfg` angelegt werden, diese beinhaltet jedoch anstatt dem Buffer eine Referenz auf ein Hardware-Empfangs-Objekt (HRH).

PDU Router Konfiguration

Das PduR Modul bietet Funktionen zum Weiterleiten von Botschaften, zwischen dem höherem COM Modul und den darunter liegenden Interfaces, an. Da im Laufe dieser Arbeit nur das CAN Interface integriert wird, kommt im PduR Modul der sogenannte "Zero cost operation mode" zum Einsatz. Dieser Modus bedeutet, dass durch den Einsatz des PduR Moduls keine längeren Ausführungszeiten entstehen, da der Aufruf einer PduR-API Funktion mittels einer Definition durch die Funktion eines anderen Moduls ersetzt wird. Quelltext 5.2 zeigt, wie dieses Ersetzen eines Funktionsaufrufs, anhand der Callback-Funktionen zwischen den Modulen CanIf und COM implementiert bzw. konfiguriert wird.

Quelltext 5.2: PduR: Zero cost operation mode [6]

```
#if PDUR_ZERO_COST_OPERATION == STD_ON
// Zero cost operation support active.
    #if PDUR_CANIF_SUPPORT == STD_ON
        #include "Com.h"
        #define PduR_CanIfRxIndication Com_RxIndication
        #define PduR_CanIfTxConfirmation Com_TxConfirmation
    #else
        #define PduR_CanIfRxIndication(... )
        #define PduR_CanIfTxConfirmation(...)
    #endif
#endif
```

COM Konfiguration

Beim COM Modul handelt es sich um eine netzwerkunabhängige Komponente, deren Hauptaufgabe es ist, dem Anwender ein Signalinterface bereitzustellen. Die Konfiguration dieses Moduls ist in fünf Hauptblöcke aufgeteilt (siehe Abbildung 5.3). Das COM Modul unterstützt keine Signalreplikation und auch keine Filter für Botschaften und Signale. Quelltext 5.3 zeigt die Struktur der COM-Konfiguration. [5]:

- ComIPdu

  Die `ComIPdu` referenziert zu einer PDU in der ECU Configuration (EcuC) und kann als eine Art Sammlung mehrerer Signale gesehen werden. Ein Beispiel für so eine PDU wäre eine CAN-Botschaft.

- ComSignal

  Ein Signal verweist auf eine `ComIPdu` und enthält Attribute, wie es aus dieser PDU extrahiert werden kann (zum Beispiel Start-Bit und Signallänge). Es existieren auch noch weitere Eigenschaften, darunter auch ein Initialisierungswert.

- ComSignalGroup

  Mit einer `ComSignalGroup` ist es möglich, mehrere Signale in eine Gruppe zusammenzufassen, dies wird für gewöhnlich angewendet, wenn Signale in einer Beziehung zueinander stehen.

- ComGroupSignal

  Signale, welche in einer `ComSignalGroup` zusammengefasst werden, werden als `ComGroupSignal` bezeichnet.

- ComIpduGroup

  Die `ComIpduGroup` bietet die Möglichkeit mehrere `ComIPdu` Objekte zu gruppieren, um diese gemeinsam zu aktivieren bzw. deaktivieren.

Für die Integration im Rahmen dieser Arbeit werden keine COM Signalgruppen verwendet, die Konfiguration des COM Moduls ermöglicht jedoch eine Definition der Signale.

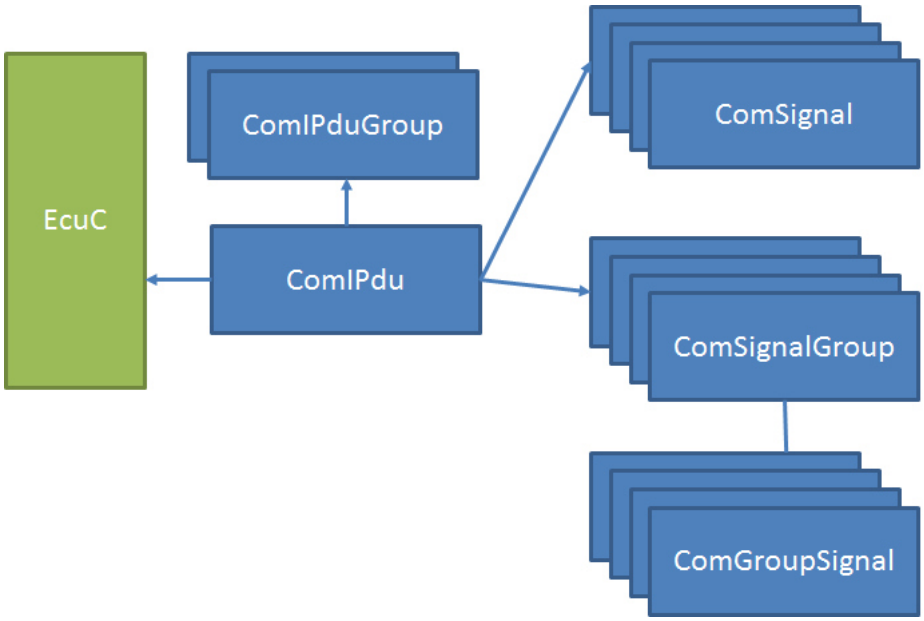

Abbildung 5.3: COM Konfigurationsmodel [5]

Quelltext 5.3: Konfigurationsstruktur einer COM Moduls [6]

```
typedef struct
{
    /** The ID of this configuration. This is returned by
        Com_GetConfigurationId(); */
    const uint8 ComConfigurationId;
    /* The number of IPdus */
    const uint16 ComNofIPdus;
    /* The number of signals */
    const uint16 ComNofSignals;
    /* The number of group signals */
    const uint16 ComNofGroupSignals;
    /* Signal Gateway mappings   */
    const ComGwMapping_type * ComGwMappingRef;
    /** IPDU definitions */
    const ComIPdu_type * const ComIPdu;
    /** IPDU group definitions */
    const ComIPduGroup_type * const ComIPduGroup;
    /** Signal definitions */
    const ComSignal_type *const  ComSignal;
    // Signal group definitions
    //ComSignalGroup_type *ComSignalGroup;
    /** Group signal definitions */
    const ComGroupSignal_type * const ComGroupSignal;
    /** Reference to Gateway source description */
    const ComGwSrcDesc_type * ComGwSrcDesc;
    /** Reference to Gateway destination description */
    const ComGwDestnDesc_type * ComGwDestnDesc;
} Com_ConfigType;
```

5.3 Software Verteilung

Die Verteilung der unterschiedlichen Funktionen des Battery Management System und der Basis-Software Module auf die drei CPU-Kerne des AUTRIX TC275 Mikrocontroller erfolgt wie in Abbildung 5.4 dargestellt.

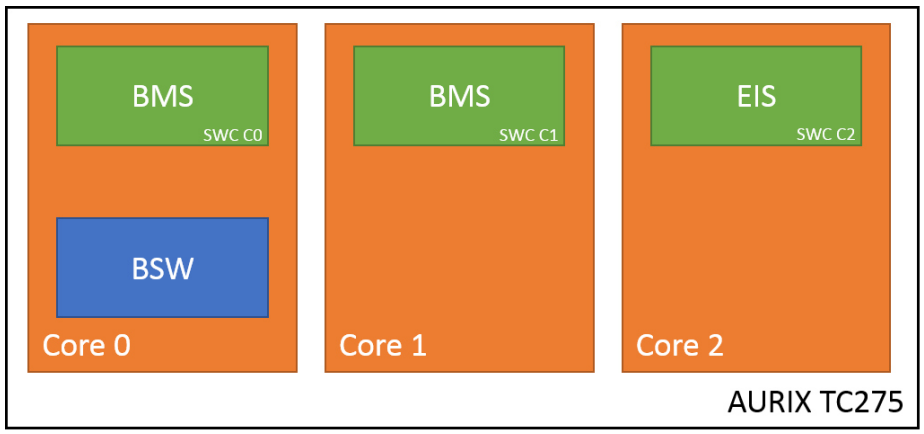

Abbildung 5.4: Software Core-Verteilung

Die Aufteilung der BSW und der einzelnen SWCs (Software Component) wird wie folgt umgesetzt:

- Core 0
  - BMS: Diese Softwarekomponente umfasst die Standard Funktionen des BMS. Dazu zählt die SoX Berechnung, welche den State of Charge (SoC), den State of Health (SoH) und den State of Function (SoF) der Batteriezellen über entsprechende CAN-Signale auswertet um somit die Zustände der Batterienzellen zu analysieren.
  - BSW: Die Basissoftware wird ebenfalls auf dem Core 0 ausgeführt, diese beinhaltet den im Zuge dieser Arbeit implementierten CAN Protokollstapel.
- Core 1
  - Core 1 dient als Support für die SoX Berechnung am Core 0.
- Core 2
  - Der Core 2 steht für die rechenintensive Electrochemical Impedance Spectroscopy (EIS) Berechnung zur Verfügung, diese wird in [7] näher betrachtet.

Die Basissoftware, welche den im Zuge dieser Arbeit entwickelten CAN Kommunikationsstapel enthält, wird auf Core 0 ausgeführt. Es ist generell Ratsam, die gesamte BSW auf einem Kern zusammenzufassen, da ansonsten die Gefahr besteht, dass unterschiedliche Prozessorkerne gleichzeitig Hardwarezugriffe durchführen wollen. Diese wiederum führen zu erhöhtem Schedulingaufwand und somit zu mehr Zeit- und Ressourcenverbrauch.

Kapitel 6

Implementierung

Diese Arbeit umfasst die Integration des CAN-Protokollstapel nach dem AUTOSAR Konzept Mithilfe der Open-Source Module des Arctic Core Standardpakets in das INCOBAT Projekt. Dazu wurden die Arctic Core Module CanIf, PduR, und COM integriert, konfiguriert und angepasst. Für die Softwareimplementierung, das Portieren auf die Hardware und der Überprüfung der Funktionalität werden geeignete Werkzeuge benötigt, welche im Folgendem näher betrachtet werden.

6.1 Entwicklungsumgebung und Tools

Basis für die Softwareentwicklung is eine geeignete Entwicklungsumgebung (Integrated Development Environment (IDE)), um die einzelnen Softwareteile zusammenzufassen. Von dieser IDE wird auch das Kompilieren gestartet, um einen Binärcode zu erhalten. Da für das INCOBAT Projekt bereits die IDE Eclipse [16] verwendet wird, wurden auch die Implementierungsarbeiten im Zuge dieser Arbeit mit Eclipse umgesetzt.

Das Echtzeitbetriebssystem (Real Time Operating System (RTOS)) zählt zu den wichtigsten aller Softwarekomponenten, da es für die Ausführung der einzelnen Tasks zuständig ist. Des Weiteren verwaltet das Betriebssystem auch die Zugriffe auf diverse Ressourcen und Interfaces, welche zur Kommunikation zwischen den einzelnen Kernen und dem Zugriff auf die Hardware dienen.

Die Basis-Software umfasst die Schnittstellenkonfigurationen bezogen auf die eigentliche Hardware und ist somit, unter anderem, auch für die Initialisierung der Busschnittstellen zuständig. Um die Entwicklung und somit die Hardwarezugriffe zu erleichtern, verwendet die BSW meist eine sogenannte "Low Level Driver" Bibliothek. Die Integration der AUTOSAR konformen Module für die CAN Kommunikation in das INCOBAT Projekt erfolgt in den Schichten dieser BSW.

Die Anwendungs-Software des INCOBAT Battery Management System wurde während dieser Arbeit weitgehend unverändert belassen.

Die im INCOBAT Projekt eingesetzte Entwicklungsplatine nutzt als Mikrocontroller den Mehrkern-Prozessor AURIX TC275, dieser verfügt über drei 32Bit Kerne mit jeweils 200Mhz.[30]
Als Compiler kommt der Altium Tasking Compiler 4.3R1 [2] zum Einsatz, welcher aus dem

Sourcecode, eine für den Infineon Tricore kompatible Binärdatei, erstellt. Dieser Compiler erfüllt die aktuellen Sicherheitszertifizierungen und ist somit in der Industrie zugelassen. Mit einem Flashingprogramm wird diese Binärdatei anschließend auf die Hardware übertragen. Zum Testen und Debuggen der Implementierung wird über die Joint Test Action Group (JTAG) Schnittstelle auf den Prozessor zugegriffen. Für das Flashen und Debuggen wird das Programm Trace32 in Kombination mit entsprechendem Debugger der Firma Lauterbach[35] verwendet. Mit diesem Programm lassen sich die Binärdateien auf den AURIX TC275 portieren, des Weiteren hat man auch die Möglichkeit, den Sourcecode zu debuggen. Trace32 ermöglicht auch einen Zugriff auf sämtliche Register des Systems, um so das korrekte bzw. auch fehlerhafte Verhalten nachvollziehen zu können.
Zum Simulieren von eingehender bzw. Überprüfen ausgehender CAN Nachrichten wurde die Software CANalyzer von der Firma Vector [58] verwendet, welche Mithilfe eines "CANcase", ebenfalls von Vector, über USB einen CAN Bus überwachen und auch beschreiben kann.

6.1.1 Anwendungs-Software (ASW)

Als ASW kommt das BMS aus dem INCOBAT Projekt zum Einsatz, dieses verwendet verschiedene CAN Nachrichten zur Überwachung verschiedener Zustände von Batteriezellen. Der Aufbau eines AUTOSAR konformen CAN Protokollstapel erfordert kleine Anpassungen der ASW, um die entsprechenden API-Funktionen nutzen zu können.

6.1.2 Basis-Software (BSW)

Die Basis-Software (BSW) ist für die Initialisierung der verwendeten Hardware zuständig und stellt auch Schnittstellen für den Hardwarezugriff zur Verfügung. Die Treiber der verschiedenen Hardwarekomponenten stammen aus dem Infineon Low Level Driver (ILLD) Paket, welches die Initialisierung und Konfiguration der einzelnen Module im Mikrocontroller ermöglicht. Der CAN Protokollstapel nach dem AUTOSAR Standard wird Mithilfe der entsprechenden Module aus dem Arctic Core Standardpaket implementiert.

Betriebssystem

Eine wichtige Komponente der BSW stellt das Betriebssystem dar, welches die einzelnen Aufgaben und Ressourcen verwaltet und auch für die Gewährleistung der Echtzeitfähigkeit des Systems verantwortlich ist. Um das Echtzeitverhalten zu gewährleisten, dürfen definierte maximal Zeiten von den auftretenden Bearbeitungs- und Verzögerungszeiten nicht überschritten werden.

Als Betriebssystem kommt in diesem Projekt das freie Echtzeitbetriebssystem (Real Time Operating System (RTOS)) ERIKA Enterprise zum Einsatz, welches auf dem in Kapitel 3.2 erwähntem OSEK Standard basiert. [19]

6.2 Softwareimplementierung

Die Implementierung umfasst die Integration und Konfiguration der Arctic Core Module CanIf, PduR und Com in das BMS des INCOBAT Projekts. Diese Module werden beim Starten des Betriebssystem vom ECU Manager, durch den Aufruf der jeweiligen Init-Funktion, initialisiert. Diese Aufrufe erhalten als Parameter einen Referenz auf die jeweilige Konfigurationsstruktur (siehe Kapitel 5.2.2).

6.2.1 Modul-Integration

Die Integration der einzelnen Module umfasst das Eingliedern und Anpassen der C-Source und Header Files in das bestehende Projekt. Begonnen wurde mit der Anpassung des bereits vorhandenen CAN Moduls, um die AUTOSAR konformen Funktionen bereitzustellen und die entsprechenden Aufrufe des Infineon Low Level Treibers zu tätigen.

6.2.1.1 CAN Treiber (Can) Integration

Das vorhandene CAN Modul des Projekts wurde so erweitert bzw. angepasst, dass die Aufrufe der Funktionen dem AUTOSAR Konzept entsprechen. Zu den betroffenen Methoden zählen unter anderen:

- `void Can_Init(const Can_ConfigType* Config)`: Initialisiert den CAN Treiber, der Parameter `Config` verweist auf die Konfiguration.
- `Std_ReturnType Can_ChangeBaudrate(uint8 Controller, const uint16 Baudrate)`: Diese Funktion dient zum Ändern der Baudrate eines CAN Controllers. Der Returnwert gibt an, ob die Änderung der Baudrate erfolgreich war (`E_OK`) oder nicht (`E_NOT_OK`).
- `Can_ReturnType Can_SetControllerMode(uint8 Controller, Can_StateTransitionType Transition)`: Mit dieser Methode ist eine Änderung des Controller Modi möglich. Mögliche Modi sind: `CAN_T_START`, `CAN_T_STOP`, `CAN_T_SLEEP`, `CAN_T_WAKEUP`. Der Returnwert gibt an ob die Änderung erfolgreich war (`CAN_OK`) oder nicht (`CAN_NOT_OK`).
- `void Can_DisableControllerInterrupts(uint8 Controller)`: Diese Funktion wird zum Deaktivieren von Interrupts eines Controllers verwendet.
- `void Can_EnableControllerInterrupts(uint8 Controller)`: Für die Aktivierung von Interrupts eines Controllers wird diese Methode aufgerufen.
- `Can_ReturnType Can_Write(Can_HwHandleType Hth, const Can_PduType* PduInfo)`: Diese Funktion dient zum Schreiben einer Botschaft (`PduInfo`) auf den CAN-Bus unter Verwendung des angegebenen Hardware Transmit Handle (HTH). Wenn während der Ausführung der Funktion ein Error auftritt, wird `CAN_NOT_OK` zurückgegeben. Falls kein Buffer Verfügbar ist, ist

der Returnwert CAN_BUSY. Bei einer erfolgreichen Ausführung der Funktion wird CAN_OK zurückgegeben.

Durch die Erweiterungen des CAN Moduls um die AUTOSAR konformen Funktionen und Strukturen, ist die Basis für die weitere Integration des CAN Interface Moduls gegeben.

6.2.1.2 CAN Interface (CanIf) Integration

Die Integration des CAN Interfaces umfasst mehrere Dateien für Logik, Konfiguration und Strukturen:

- CanIf.c und CanIf.h umfassen unter anderen die API Funktionen CanIf_Transmit, CanIf_ReadRxPduData und CanIf_Init. Letztere wird zum Initialisieren und Konfigurieren des Interfaces verwendet, dazu wird ein Pointer auf eine Konfigurationsstruktur übergeben. Im CanIf Modul findet man auch Funktionen zum Ändern der Baudrate oder des Modi eines CAN Controller, welche die entsprechende Funktion des CAN Moduls aufrufen.
- CanIf_ConfigTypes.h enthält Strukturen, welche für die Konfiguration benötigt werden, darunter CanIf_TxPduConfigType, CanIf_RxPduConfigType und auch CanIf_ConfigType, welche für die Initialisierung des Moduls verwendet wird.
- CanIf_Cfg.c und CanIf_Cfg.h enthalten unterschiedliche Definitionen bezüglich der Verfügbarkeit von Funktionalitäten, so kann zum Beispiel mit der Definition CANIF_PUBLIC_READRXPDU_DATA_API die Verfügbarkeit der API-Funktion CanIf_ReadRxPduData beeinflusst werden.
- CanIf_PBCfg.c und CanIf_PBCfg.h bilden die "PostBuild" Konfigurationen ab. Diese beinhalten die eigentliche Konfiguration des Moduls, wie zum Beispiel die unterschiedlichen Botschaften, Hardware-Objekte und die Initialparameter.
- CanIf_Types.h stellt zusätzliche Typen und Strukturen zur Verfügung, welche in den Konfigurationen, aber auch für die Aufrufe von API-Funktionen, verwendet werden.
- CanIf_Cbk.h beinhaltet die Callback-Funktionen, welche für die Benachrichtigungen zwischen den Modulen verwendet werden, um zum Beispiel das Versenden einer Botschaft zu bestätigen.

Das CAN Interface wird im ECU Manager durch den Aufruf von CanIf_Init und Übergabe einer Referenz auf die CanIf Konfiguration initialisiert. Vor dieser Initialisierung erfolgt der Aufruf der Initialfunktion das CAN Moduls.

6.2.1.3 PDU Router (PduR) Integration

Das PduR Modul wird in dieser Arbeit nur im Zusammenhang mit dem CAN Interface verwendet, daher wird für das Routing der in Kapitel 5.2.2 erwähnte "Zero cost operation

mode" angewendet. Für dieses Modul wurden mehrere Dateien integriert, die Wichtigsten werden nun angeführt.

- `PduR.h` enthält Definitionen für die Module, welche vom PDU Router unterstützt werden, für diese Arbeit wird der Support für das COM und das CanIf Modul aktiviert.
- In der `PduR_Cfg.h` werden die Definition für das Routing zwischen den PDUs der Module CanIf und COM definiert. Des weiteren sind hier auch die Definitionen für das Ersetzen von Funktionsaufrufen enthalten, Quelltext 6.4 zeigt so eine Definition. Der bereits erwähnte "Zero cost operation mode" wird ebenfalls in dieser Datei mit `#define PDUR_ZERO_COST_OPERATION STD_ON` aktiviert.
- Die weiteren Dateien, welche für das PduR Modul integriert wurden, enthalten Definitionen von Typen und Strukturen zur Modulkonfiguration, sowie weitere Funktionen.

Auch der PDU Router stellt eine Funktion zum Initialisieren zur Verfügung (`PduR_Init`). Da jedoch im Zuge dieser Arbeit das PduR Modul mit dem "Zero cost operation mode" verwendet wird, ist diese Initialisierung nicht notwendig.

6.2.1.4 COM Integration

Das letzte Module des CAN Protokollstapels welches im Zuge dieser Arbeit integriert wurde, ist das COM Modul. Hier handelt es sich um ein signalorientiertes Modul, welches dem Anwender ein einfaches Versenden und Lesen von einzelnen Signalen (Teile einer Nachricht) durch entsprechende API-Funktionen ermöglicht.

- Die Dateien `Com.c` und `Com.h` enthalten die Init-Funktion des Moduls, welche die einzelnen PDUs und Signale initialisieren und auch Zuweisungen von Buffern durchführen.
- `Com_Com.c` und `Com_Com.h` stellen der ASW die API-Funktionen zum Senden (`Com_SendSignal`) und Lesen (`Com_ReceiveSignal`) von Signalen zur Verfügung.
- `Com_Cfg.h` definiert unter anderen die Größe des Buffers, welcher zur Übertragung der Signale verwendet wird. Auch eine maximale Anzahl an PDUs und Signale wird in dieser Datei festgelegt.
- Die eigentliche Konfiguration des Moduls findet in den Dateien `Com_PbCfg.c` und `Com_PbCfg.h` statt. Hier werden neben der allgemeinen Modulkonfiguration auch die PDUs und die darin enthaltenen Signale definiert. Die Header File enthält für jede PDU und jedes Signal eine Definition, welche die Verwendung der unterschiedlichen API-Funktionen erleichtern.
- In den restlichen COM bezogenen Dateien sind weitere Typen, Strukturen und Funktionen definiert, welche sowohl von der internen Logik, als auch für die API-Funktionen benötigt werden.

Die Initialisierung des COM Moduls erfolgt durch den Aufruf der Funktion `Com_Init`, welche als Parameter eine Referenz auf die Konfigurationsstruktur erhält. Diese Funktion wird durch den ECU Manager nach der Initialisierung der Module CAN und CanIf aufgerufen.

6.2.2 Modul-Konfiguration

Neben der Integration der einzelnen AUTOSAR konformen Module aus dem Arctic Core Standardpaket für die Abbildung des CAN Protokollstapel, wurden auch entsprechende Konfigurationen durchgeführt. Diese Anpassungen sind einerseits notwendig, um eine Kommunikation mit dem bereits vorhandenem CAN Treiber von Infineon zu ermöglichen und andererseits, um die Module für die Kommunikation des INCOBAT BMS entsprechend der CAN Botschaften und deren Signale zu konfigurieren.

6.2.2.1 CAN Treiber (Can) Konfiguration

Da das bereits vorhandene CAN Modul schon entsprechend der verwendeten Umgebung konfiguriert war, mussten hier nur Anpassungen durchgeführt werden, um eine AUTOSAR konforme API und Konfiguration zu realisieren, welche für die weiteren Module, vor allem dem CanIf Modul, notwendig sind.
Eine detaillierter Auszug aus der Konfiguration des CAN Moduls wird in Anhang A.1 dargelegt.

6.2.2.2 CAN Interface (CanIf) Konfiguration

Das CAN Interface befindet sich zwischen dem CAN Treiber der unteren Schicht und der höheren Kommunikationsschicht, zu denen der PDU Router und das COM Modul zählen. In der Konfiguration des CanIf Moduls wird jede Botschaft, welche gesendet oder empfangen werden soll, definiert und für jeden CAN Controller ein Buffer angelegt. Diese Nachrichtendefinitionen sind jeweils in Arrays (`CanIfTxPduConfigData` und `CanIfRxPduConfigData`) gespeichert, welche ein einfacheres Hinzufügen weiterer Nachrichten ermöglichen.
Die Konfiguration einer Botschaft im CAN Interface enthält unter anderem die CAN ID (`CanIfCanTxPduIdCanId`), die Länge der Nachricht (`CanIfCanTxPduIdDlc`) und auch das CAN ID Format (`CanIfTxPduIdCanIdType`), welches Standard (11 Bit ID) oder Extended (29 Bit ID) sein kann. Zusätzlich enthält die Konfiguration einer ausgehenden Botschaft eine Referenz auf einen Buffer (`CanIfTxPduBufferRef`). Bei den eingehenden Nachrichten gibt es die Möglichkeit einen Filter (`CanIfCanRxPduCanIdMask`) zu definieren, um eine Vorauswahl an Nachrichten treffen zu können. Quelltext 6.1 zeigt die Konfiguration einer Transmit-PDU, welche im Zuge dieser Arbeit in das INCOBAT-Projekt integriert wurde. Im Quelltext 6.2 wird hingegen die Konfiguration einer Botschaft

zum Empfangen, aus dem Projekt, gezeigt.
Anhang A.2 zeigt einen Auszug aus der Konfiguration des CAN Interfaces.

Quelltext 6.1: Konfiguration einer Botschaft zum Versenden

```
const CanIf_TxPduConfigType CanIfTxPduConfigData[] =
{
    {
        .CanIfTxPduId               = PDUR_REVERSE_PDU_ID_FREQIND,
        .CanIfCanTxPduIdCanId       = 258,
        .CanIfCanTxPduIdDlc         = 8,
        .CanIfCanTxPduType          = CANIF_PDU_TYPE_STATIC,
        .CanIfTxPduPnFilterEnable   = STD_OFF,
      #if ( CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API == STD_ON )
        .CanIfReadTxPduNotifyStatus = FALSE,
      #endif
        .CanIfTxPduIdCanIdType      = CANIF_CAN_ID_TYPE_11,
        .CanIfUserTxConfirmation    = PDUR_CALLOUT,
        /* [CanIfBufferCfg] */
        .CanIfTxPduBufferRef        = &CanIfBufferCfgData[0],
        .PduIdRef                   = NULL,
    }
};
```

Quelltext 6.2: Konfiguration einer Botschaft zum Empfangen

```
const CanIf_RxPduConfigType CanIfRxPduConfigData[] =
{
    {
        .CanIfCanRxPduId            = PDUR_PDU_ID_RX_PDU,
        .CanIfCanRxPduLowerCanId    = 0x200,
        .CanIfCanRxPduUpperCanId    = 0x200,
        .CanIfCanRxPduDlc           = 8,
      #if ( CANIF_PUBLIC_READRXPDU_DATA_API == STD_ON )
        .CanIfReadRxPduData         = FALSE,
      #endif
      #if ( CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API == STD_ON )
        .CanIfReadRxPduNotifyStatus = FALSE,
      #endif
        .CanIfCanRxPduHrhRef        = &CanIfHrhConfigData_CanIfInitHohCfg[0],
        .CanIfRxPduIdCanIdType      = CANIF_CAN_ID_TYPE_11,
        .CanIfUserRxIndication      = PDUR_CALLOUT,
        .CanIfCanRxPduCanIdMask     = 0x7FF,
        .PduIdRef                   = NULL,
    }
};
```

6.2.2.3 PDU Router (PduR) Konfiguration

Der PDU Router stellt Funktionen zum Weiterleiten von Botschaften zwischen dem COM Modul und einer Kommunikationsschnittstelle oder einem Transportprotokoll zur Verfügung. Dazu werden für die einzelnen PDUs Definitionen angelegt, um das interne Routing zu ermöglichen (siehe Quelltext 6.3). Diese dienen dazu, die Verbindung zwischen dem oberen COM Modul und dem unterem Interface bzw. Transportprotokoll, in diesem Fall das CAN-Interface, zu ermöglichen. Da der sogenannte "Zero cost operation mo-

de" angewendet wird, erfolgen Funktionsaufrufe, wie zum Beispiel der Aufruf der API-Funktion `CanIf_Transmit()`, ebenfalls durch ein Definition in der Konfiguration des PDU-Router. Quelltext 6.4 zeigt, wie der Aufruf von `PduR_ComTransmit()` durch die entsprechende CanIf-Funktion ersetzt wird. Dadurch entstehen keine längeren Ausführungszeiten und es kommt zu keiner zusätzlichen Ressourcennutzung, also keine weiteren "Kosten". Falls man diesen "Zero cost operation mode" nicht verwendet, hat man die Möglichkeit, Modifikationen in der Logik des PduR Moduls vorzunehmen. Da für die Integration im Zuge dieser Arbeit solche Anpassungen nicht notwendig waren, wurde dieser Modus aktiviert und die Logik des PduR Moduls unverändert belassen, einzig die Definitionen für die interne Kommunikation zwischen dem CAN Interface und dem COM Modul waren zu erstellen.

Quelltext 6.3: PduR: Definitionen einer PDU

```
#define PDUR_PDU_ID_RX_PDU_BatHiLAuxData0
    ComConf_ComIPdu_RX_PDU_BatHiLAuxData0

#define PDUR_REVERSE_PDU_ID_RX_PDU_BatHiLAuxData0
    CANIF_PDU_ID_RX_PDU_BatHiLAuxData0
```

Quelltext 6.4: PduR: Aufruf der Transmit Funktion von CanIf (zero cost operatoin mode)

```
#if PDUR_ZERO_COST_OPERATION == STD_ON
    #if PDUR_COM_SUPPORT == STD_ON
        #include "CanIf.h"
        #define PduR_ComTransmit CanIf_Transmit
    #else
        #define PduR_ComTransmit(... )  (E_OK)
    #endif
#endif
```

6.2.2.4 COM Konfiguration

Das COM Modul ermöglicht der RTE einen Signal-basierten Datenaustausch, wozu die einzelnen Signale einer Botschaft in der Konfiguration des COM Moduls definiert werden (siehe Quelltext 6.5). Die `ComHandleId` ist die interne ID eines Signals, während die `ComIPduHandleId` die ID der gesamten PDU ist. Der Anwender hat somit die Möglichkeit, direkt einzelne Signale zu empfangen bzw. versenden, ohne wissen zu müssen, zu welcher Botschaft ein Signal gehört bzw. wie die dazugehörige Nachricht aufgebaut ist, also an welcher Stelle sich das gewünschte Signal befindet. Die interne Verarbeitung über das PduR Modul erfolgt mit den bereits erwähnten Definitionen im PduR Modul.
Anhang A.3 enthält einen Auszug aus der Konfiguration des COM Moduls, welcher einzelne Signale und weitere Einstellungen enthält.

Quelltext 6.5: COM Signale einer Botschaft (Auszug)

```
#define ComConf_ComIPdu_RX_PDU_BatHiLAuxData0                    0
#define ComConf_ComIPdu_RX_PDU_BatHiLAuxData1                    1

#define ComConf_ComSignal_RX_TiBatOff                            0
#define ComConf_ComSignal_RX_BcuSt                               1
#define ComConf_ComSignal_RX_SrvCmd                              2

const ComSignal_type ComSignal[] =
{
    {
        .ComHandleId                = ComConf_ComSignal_RX_TiBatOff,
        .ComIPduHandleId            = ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
        .ComFirstTimeoutFactor      = 0,
        .ComNotification            = COM_NO_FUNCTION_CALLOUT,
        .ComTimeoutFactor           = 0,
        .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
        .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
        .ComTransferProperty        = PENDING,
        .ComUpdateBitPosition       = 0,
        .ComSignalArcUseUpdateBit   = FALSE,
        .ComSignalInitValue         = &Com_SignalInitValue_TiBatOff,
        .ComBitPosition             = 0,
        .ComBitSize                 = 16,
        .ComSignalEndianess         = COM_LITTLE_ENDIAN,
        .ComSignalType              = UINT16,
        .Com_Arc_IsSignalGroup      = 0,
        .ComGroupSignal             = NULL,
        .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
        .ComSigGwRoutingReq         = FALSE,
        .Com_Arc_EOL                = 0
    },
    {
        .ComHandleId                = ComConf_ComSignal_RX_BcuSt,
        .ComIPduHandleId            = ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
        .ComFirstTimeoutFactor      = 0,
        .ComNotification            = COM_NO_FUNCTION_CALLOUT,
        .ComTimeoutFactor           = 0,
        .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
        .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
        .ComTransferProperty        = PENDING,
        .ComUpdateBitPosition       = 0,
        .ComSignalArcUseUpdateBit   = FALSE,
        .ComSignalInitValue         = &Com_SignalInitValue_TiBatOff,
        .ComBitPosition             = 16,
        .ComBitSize                 = 2,
        .ComSignalEndianess         = COM_LITTLE_ENDIAN,
        .ComSignalType              = UINT8,
        .Com_Arc_IsSignalGroup      = 0,
        .ComGroupSignal             = NULL,
        .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
        .ComSigGwRoutingReq         = FALSE,
        .Com_Arc_EOL                = 0
    }
    ...
};
```

6.3 Anwendung

Für das BMS des INCOBAT Projekt wird zur Kommunikation mit dem CAN Bus die API des COM Moduls verwendet. Jedoch bietet auch das CAN Interface eine API an, mit der eine CAN Kommunikation ermöglicht wird. Nachfolgend werden Einsatzbeispiele unterschiedlicher Funktionen der Module CanIf und COM gezeigt. Bevor die einzelnen Module verwendet werden können, müssen diese durch den ECU Manager initialisiert werden. Quelltext 6.6 zeigt einen Ausschnitt der Initialisierungen durch den ECU Manager. Die einzelnen Konfigurationsreferenzen verweisen auf die jeweilige Modulkonfigurationsstruktur, welche in Kapitel 6.2.2 näher betrachtet wurden.

Quelltext 6.6: Modulinitialisierung durch den ECU Manager (Ausschnitt)

```
...
Can_Init(&CanConfigData);
CanIf_Init(&CanIf_Config);
...
Com_Init(&ComConfiguration);
...
```

6.3.1 CAN Interface

Das CAN Interface bietet die API Funktionen `CanIf_ReadRxPduData` und `CanIf_Transmit` an. Quellcode 6.7 zeigt den Einsatz dieser beiden Funktionen. Beim Datentyp `PduIdType` handelt es sich um ein `uint8` Objekt, welches für die Angabe der CAN ID verwendet wird. Die Daten welche empfangen bzw. versendet werden, sind in einem Objekt des Typs `PduInfoType` abgebildet. Dieser Typ enthält eine Referenz (`SduDataPtr`) auf einen `uint8` Speicherbereich, welcher für die Daten der Botschaft verwendet wird. Des Weiteren existiert noch das Feld `SduLength`, welches die Anzahl der Daten einer ausgehenden Nachricht in Bytes angibt.

Quelltext 6.7: API-Funktionen von CanIf

```
PduIdType can_rx_id = 512;
PduInfoType can_rx_data;

uint8 data_rx_ptr[8];
can_rx_data.SduDataPtr = data_rx_ptr;

CanIf_ReadRxPduData(can_rx_id, &can_rx_data);


PduIdType can_tx_id = 256;
PduInfoType can_tx_data;

uint8 data_tx_ptr[8] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}

can_tx_data.SduDataPtr = data_tx_ptr;
can_tx_data.SduLength = 8;

CanIf_Transmit(can_tx_id, &can_tx_data);
```

6.3.2 COM

Für den Einsatz des AUTOSAR konformen CAN Kommunikationsstapel ist die API des COM Moduls interessanter, da hier ein einfaches signalbasiertes Kommunizieren über den CAN Bus möglich ist. Quellcode 6.8 zeigt den Einsatz der COM API-Funktionen `Com_ReceiveSignal` und `Com_SendSignal`. Der Aufruf der Empfangsfunktion speichert den aktuellen Wert des angegebenen Signals in den Speicher der übergebenen `uint16` Referenz. Hier ist es wichtig, den Datentyp, also die Länge des Signals in der CAN Nachricht, zu kennen. Quelltext 6.5 zeigt die Definition des hier verwendeten Signals samt der internen COM-Signal ID. Die Sendefunktion legt den Wert der übergebenen Variable auf das angegebene Signal, welches somit beim dem nächsten Versenden der gesamten Botschaft auf den CAN Bus geschrieben wird.

Quelltext 6.8: API-Funktionen von COM

```
uint16 rx_signal_tibatoff = 0;
Com_ReceiveSignal(ComConf_ComSignal_RX_TiBatOff, &rx_signal_tibatoff)

uint8 tx_signal_result = 64;
Com_SendSignal(ComConf_ComSignal_TX_ResultSig, &tx_signal_result)
```

6.4 Evaluierung

6.4.1 Aufbau

Die Implementierung des CAN Kommunikationsstapel nach dem AUTOSAR Standard im BMS des INCOBAT Projekts wurde mithilfe von verschiedenen Tools durchgeführt. Abbildung 6.1 zeigt den Aufbau der Testumgebung. Die in Eclipse entwickelte Software wurde mithilfe des Tasking Compilers zu einem ausführbaren Binärcode übersetzt. Mit dem Programm Trace32 und einem entsprechenden Debugger, welcher über USB an den Entwicklungs-PC angeschlossen wird und über die JTAG Schnittstelle mit dem AURIX TC275 verbunden ist, wurde zunächst der Binärcode auf die Prototypenplattform übertragen.
Zusätzlich zu dieser Verbindung zur JTAG Schnittstelle, existiert auch noch ein CAN-Bus. Dieser wird durch den Einsatz eines CANcase, welches über USB an den Entwicklungs-PC angeschlossen ist und von diesem aus mit dem Programm CANalyzer angesprochen wird, gebildet. Das CANcase ist über ein CAN-Kabel mit dem AURIX TC275 verbunden. Somit enthält dieser CAN-Bus zwei Busteilnehmer: den Entwicklungs-PC und den AURIX TC275.

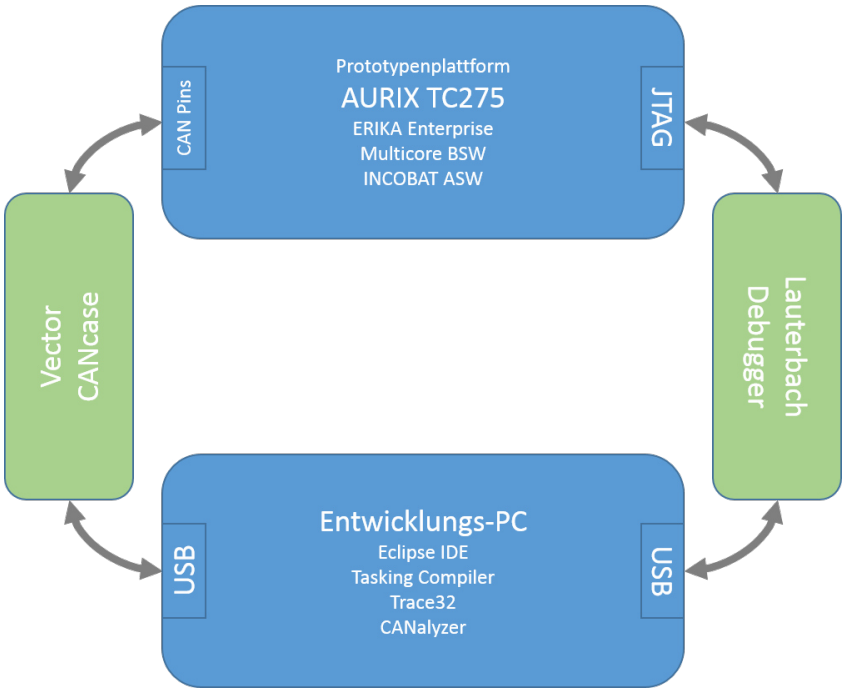

Abbildung 6.1: Testaufbau

6.4.2 Durchführung

Mithilfe einer entsprechenden Definitionsdatei, welche die Nachrichten und Signale einer Batterie enthält, wurde vom CANalyzer über das CANcase eine Nachricht auf den Bus geschrieben. Auf der Seite des BMS wurden einzelne Signale mit der API-Funktion `Com_ReceiveSignal` ausgelesen und durch den Einsatz des Debuggers von Lauterbach in Kombination mit der Software Trace32 deren Richtigkeit überprüft und bewiesen. Die Funktionalität des Kommunikationsstapels wurde ebenso in die andere Richtung verifiziert, indem CAN Botschaften vom BMS versendet wurden, deren Korrektheit wiederum mit dem CANalyzer verifiziert wurde. Diese Tests wurden für verschiedene Signale durchgeführt, des weiteren wurden auch weitere API-Funktionen der Module CanIf und COM erfolgreich getestet.

Kapitel 7

Bewertung der Erfolgsfaktoren

Nach der Integration und Konfiguration der AUTOSAR konformen Modulen von Arctic Core v9.0.0, zur Abbildung des CAN Kommunikationsstapel, in das BMS des INCOBAT Projekts wurde anhand ausgewählter Faktoren eine Bewertung durchgeführt. Ziel dieser Beurteilung ist es zu zeigen, dass mithilfe von entsprechenden quelloffenen Modulen eine Plattform erstellt werden kann, welche den bewährten AUTOSAR Standard umsetzt, um so eine verteilte und flexible Entwicklung zu ermöglichen. Durch die definierten Schnittstellen der Module hat man auch den Vorteil, keine eigene Softwarearchitektur entwickelnd zu müssen. Diese Umsetzung verspricht eine Prototypenülattform mit hoher Qualität und Flexibilität. Weitere Funktionalitäten können durch das Hinzufügen neuer Module realisiert werden. Anpassungen der Konfiguration einzelner Module werden durch die definierten Konfigurationsstrukturen unterstützt.
Nachfolgend wird nun eine Bewertung entscheidender Erfolgsfaktoren, welche für dir Durchführung dieser Arbeit notwendig bzw. relevant waren, durchgeführt.

7.1 Vorbedingungen

Für die Umsetzung dieser Arbeit ist es notwendig, sich mit unterschiedlichen Thematiken und Standards auseinanderzusetzen. Eine gewisse Programmierfähigkeit ist für die Integration und das Verständnis der Module notwendig. Der Aufbau des umfangreichen AUTOSAR Standards ist ebenfalls zu verstehen, wie auch das CAN Protokoll.

Programmierfähigkeit	Für die Umsetzung dieser Arbeit sind entsprechende Fähigkeiten in der Programmiersprache C erforderlich.
AUTOSAR	Die Methodik des AUTOSAR Konzepts, welche eine Austauschbarkeit und flexible Zusammenstellung unterschiedlicher Module von verschiedenen Herstellern vorsieht, ist auf jeden Fall zu verstehen. Des Weiteren ist eine detaillierte Analyse des CAN Kommunikationsstapel notwendig, um das Zusammenspiel der Module CanIf, PduR und COM nachvollziehen zu können.
Arctic Core	Hinsichtlich der Integration der Module aus dem Artic Core Standardpaket ist es notwendig, die entsprechenden Dateien, welche für den Kommunikationsstapel benötigt werden, herauszufinden und evtl. benötigte Anpassungen für die Eingliederung umzusetzen. Hier werden die C-Fähigkeiten und das Verständnis des AUTOSAR Standards angewandt.
Kommunikationsprotokolle	Ein gewisses Verständnis des CAN Protokolls ist für die Implementierung ebenfalls von Vorteil, um so das Verhalten am CAN Bus interpretieren zu können.

Tabelle 7.1: Notwendiges Know-How

Zusätzlich zu dem in Tabelle 7.1 erwähntem Wissen, ist auch ein Verständnis der unterstützenden Tools (siehe Kapitel 6.1) zur Implementierung, Portierung und Überprüfung notwendig.

7.2 Anbieter

Die Wahl eines geeigneten Anbieters für die Entwicklung einer Prototypenplattform ist ein sehr wichtiger Punkt. Besonders für die Prototypenentwicklung wird eine kostengünstige Lösung angestrebt, welche offen für Anpassungen ist. Des weiteren wird ein schnelle Verfügbarkeit erwartet und der Overhead für das Starten mit einem neuen Produkt soll so gering wie möglich sein.

ArcCore [3]

Die verwendeten Module zur Integration stammen aus dem Arctic Core Standardpaket der Version 9.0.0 [6] des schwedischen AUTOSAR Softwareanbieter ArcCore [3].
ArcCore bietet neben der transparenten und variablen Plattform Arctic Core noch weitere Produkte zur Unterstützung der Softwareentwicklung und Konfiguration von eingebetteten Systemen im automotive Bereich an (siehe Kapitel 5.2).

Entwicklung	Im Jahr 2006 wurde das Unternehmen "3core" gegründet, welches Produktentwicklung für eingebettete Systeme in diversen Bereichen, wie zum Beispiel Automotiv oder Telekommunikation, anbot. 2009 entschied sich 3core neue Produkte als Open-Source Lösungen anzubieten. Des Weiteren wurde 3core zu einem Partner von AUTOSAR, aus diesen Ereignissen entstand ArcCore. Die weiteren Entwicklung des AUTOSAR Standards wurden ebenfalls übernommen.
Marktanteil	ArcCore weist mittlerweile Kunden auf der ganzen Welt vor, welche die Produkte nicht nur zur Forschung und Entwicklung einsetzten, sondern auch vermehrt zur Serienproduktion nutzen. Nach der Einführung der ArcCore Produkte wurde ein Umsatzwachstum von 50% innerhalb drei Jahre erzielt. [1]
Produktportfolio	ArcCore bietet neben dem in dieser Arbeit verwendetem Arcitc Core Paket auch noch weitere Produkte zur Entwicklung von eingebetteten Systemen an. Das Arctic Studio stellt eine komplette Entwicklungsumgebung für die Softwareentwicklung zur Verfügung. Ein weiteres Produkt ist der Arctic Bootloader, welcher nach dem Start des Prozessors ausgeführt wird und Prüfsummen berechnet oder auch das Portieren neuer Software ermöglicht. Zusätzlich bietet ArcCore auch noch sogenannte Starter-Kits an, dabei handelt es sich um Prototyp-Entwicklungsboards von unterschiedlichen Herstellern, deren Mikrocontroller von den ArcCore Produkten unterstützt werden. Somit kann man schnell eine funktionierende Arbeitsumgebung erreichen. [3]
Partner	Eine Zusammenarbeit mit unterschiedlichen technologischen Partnern ermöglicht ArcCore die Bereitstellung von Lösungen. Zu diesen Zählen auch die in dieser Arbeit betrachteten bzw. verwendeten Standards und Produkte AUTOSAR, Eclipse und Lauterbach. Für die Entwicklung der Mikrocontrollertreiber stehen unter anderem die Partnerunternehmen Freescale und TexasInstruments zur Verfügung. [3]

Tabelle 7.2: Eigenschaften von ArcCore [3]

Arctic Core Standardpaket [6]

Da sich diese Arbeit mit der Integration von Modulen aus dem Standardpaket von Arctic Core befasst, wird dieses im Folgendem näher betrachtet.

Entwicklung	Die Entwicklung von Arctic Core ist an das AUTOSAR Konzept angelehnt. Die aktuelle Version 9.0.0 wurde auf Basis von AUTOSAR 4.0.2, 4.0.3 und 4.1.1 entwickelt.
Funktionen	Das Paket beinhaltet komplette Kommunikationsstapel für CAN, Lin und Ethernet. Des Weiteren ist auch ein Echtzeitbetriebssystem (RTOS) enthalten.
Aufbau	Das modulare Design bietet eine Flexibilität der Integration und Konfiguration an, was zu einer Kostenreduzierung führt, da nur jene Module umgesetzt werden, welche auch tatsächlich benötigt werden. Der uneingeschränkte Zugriff auf den Quelltext ermöglicht beliebige Anpassungen für die jeweilige Situation.
Lizenzierung	Für die unterschiedlichen Phasen eines Projektes, von der Evaluierung und Prototypenerstellung bis hin zur Entwicklung und Serienproduktion, stehen eigene Lizenzmodelle zur Verfügung.
Systemanforderungen	Da es sich bei dem Arctic Core Paket um ein reines Softwarepaket handelt, werden hier keine Anforderungen an ein Entwicklersystem gestellt. Der Einsatz des Arctic Studio, basierend auf Eclipse, setzt eine aktuelle Version eines Windows Betriebssystems voraus.

Tabelle 7.3: Eigenschaften des Arctic Core Standardpakets [6]

Der modulare Aufbau nach dem aktuellem AUTOSAR Standard ermöglichte in dieser Arbeit eine flexible Integration der benötigten Module unter Einhaltung der definierten Schnittstellen.

7.3 Kosten, Verfügbarkeit und Service

Wichtige Faktoren bei der Wahl von Softwarepakten oder Entwicklungsumgebungen sind die Kosten und die Verfügbarkeit. Besonders bei der Entwicklung einer Prototypen-Plattform sind oft hohe Kosten ein KO-Kriterium.
In Tabelle 7.4 werden die unterschiedlichen Kosten und Services rund um Arctic Core näher betrachtet. Besonders ein angebotener Support stellt oftmals einen Mehrwert eines Produktes dar, in diesem Zusammenhang bietet ArcCore nicht nur Entwicklungs- und Integrationssupport an, sondern auch Schulungen über das umfangreiche Themengebiet von AUTOSAR.

Anschaffungskosten	Für die im Rahmen dieser Arbeit verwendeten Arctic Core Module fallen keine Kosten an, für deren Einsatz in einer Serienproduktion würden jedoch Kosten anfallen.
Supportkosten	ArcCore bietet unterschiedlich intensive Trainings bezüglich des AUTOSAR Konzepts an, zusätzlich gibt es auch Integrationssupport und Entwicklungssupport.
Lizenzmodell	Das Lizenzmodell sieht verschiedene Modelle für die unterschiedlichen Projektphasen vor. So kann man mit einer Evaluierungslizenz, welche im Laufe dieser Arbeit verwendet wurde, die Produkte näher analysieren und den Aufwand bzw. die Funktionalität der Umsetzung bewerten.

Tabelle 7.4: Kosten und Verfügbarkeit

Da für die Implementierungen im Zuge dieser Arbeit lediglich eine Evaluierungslizenz zur Verfügung stand, können weder die Kosten für Lizenzen hinsichtlich einer Serienproduktion, noch die Kosten für die verschiedenen Supportarten näher bewertet werden.

7.4 Standards

Standardisierte Softwaremodule und Vorgehensweisen sind besonders hinsichtlich steigenden Komplexität im automotive Bereich ein wichtiger Punkt für die Umsetzung neuer Plattformen. Besonders wichtig ist die Flexibilität einzelner Softwaremodule, um somit die Austauschbarkeit zu gewährleisten. Des weiteren wird der Wartungsaufwand durch entsprechende Standards deutlich verringert. Eine Standardisierte Plattform kann durch die gute Skalierbarkeit mit geringem Aufwand für andere Projekte wiederverwendet werden.
Da das Arctic Core Standardpaket auf dem Konzept von AUTOSAR basiert, erreicht man mit einer Entwicklung mit Arctic Core ein Produkt, welches aufgrund des AUTOSAR Standards bestimmten Definitionen folgt und dadurch eine hohe Qualität vorweist. Dieser Vorteil ist natürlich auch für die Entwicklung einer Prototypenplattform geeignet, da so die Akzeptanz steigt und eine Portierung zu einem Serienprodukt erleichtert wird. Des Weiteren wird durch den Einsatz eines geeigneten Entwicklungsprozesse und durch die Einhaltung von Branchenspezifischen Zertifizierungen die Qualität der ArcCore Produkte sichergestellt. Da es sich bei Artic Core um ein Open-Source Produkt handelt, ist auch die Qualitätssicherung des Quelltextes erforderlich, hierfür greift ArcCore auf den Programmierstandard der Automobilindustrie "Motor Industry Software Reliability Association (MISRA)" zurück.

AUTOSAR	Die aktuelle hier verwendete Version wurde basierend auf die AUTOSAR Versionen 4.0.2, 4.0.3 und 4.1.1 entwickelt.
ISO	Die Entwicklung von Arctic Core ist ebenso an die Vorgaben der ISO 26262 [31] angelehnt, um die funktionale Sicherheit einzuhalten und zu gewährleisten.
Konformitätsklassen	Die Implementierungskonformitätsklassen Implementation Conformance Classes (ICC) 1 bis ICC 3 werden unterstützt, ebenso die sogenannte AUTOSAR Scalability Classes 1 bis 3.
MISRA-C	Arctic Core wurde nach den C-Programmierstandards MISRA-C2 und MISRA-C3, welche von Motor Industry Software Reliability Association (MISRA)[41, 40] definiert wurden, geprüft.
Entwicklungsprozess	Die Entwicklung wurde in Übereinstimmung mit Level 3 von Automotive SPICE (Software Process Improvement and Capability Determination) veranlagt. Dieses Level 3 besagt, dass einheitliche Richtlinien für die gesamte Organisation existieren. Abbildung 7.1 zeigt das Automotive SPICE Modell.

Tabelle 7.5: Arctic Core Standards

7.5 Produktreifegrad

Der Produktreifegrad von einer Software lässt sich zum Beispiel mit Automotive SPICE (Software Process Improvement and Capability Determination) bewerten. Dabei handelt es sich um eine objektive Prozessbewertung, welche 6 Stufen (Level 0 - Level 5) des Reifegrads definiert (siehe Abbildung 7.1). Die Definition von Level 0 besagt, dass einige Prozesse nicht ausgeführt werden, so sind wichtige Arbeitsunterlagen, wie zum Beispiel Pläne, Designdokumente oder Spezifikationen, nicht vorhanden. Diese Dokumente sind in Level 1 alle vorhanden, somit werden auch die geforderten Prozesse angewandt. Eine systematische Planung und Nachverfolgung wird in Level 2 erreicht. Werden in der gesamten Organisation einheitliche Richtlinien befolgt, spricht von vom Level 3. Level 4 und 5 sehen zusätzlich noch statistische Messungen und Optimierungen der Prozesse vor. [14]

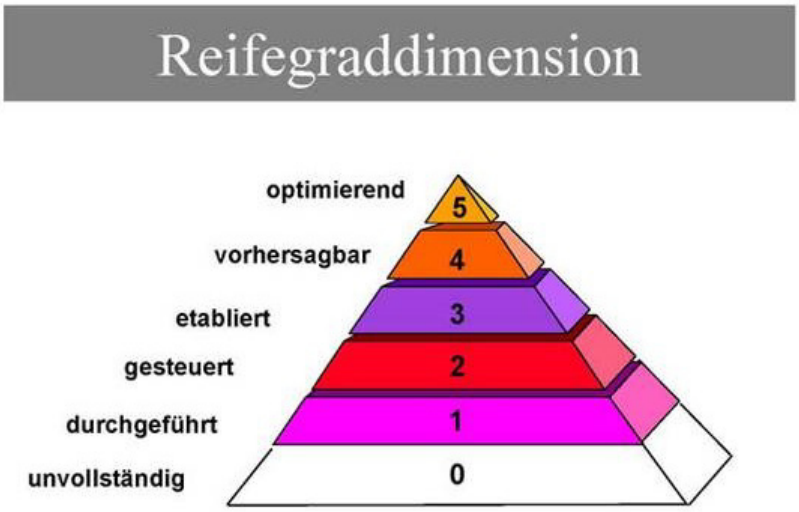

Abbildung 7.1: Automotive SPICE Modell [14]

Für die Entwicklung eines Prototypen wird mindestens eine Level 1 Bewertung empfohlen.
Arctic Core erreicht eine im Automotive SPICE Modell eine Level 3 Bewertung und ist somit bestens für eine Prototypenentwicklung geeignet. Daher wurde Arctic Core im Zuge dieser Arbeit für die Erstellung einer AUTOSAR konformen Prototypenplattform eingesetzt. Besonders die Tatasche, dass der Quellcode von Arctic Core gänzlich zugänglich ist, macht die Nutzung im Rahmen einer Prototypenentwicklung interessant. Jedoch geht das Einsatzgebiet von Arctic Core noch viel weiter. Da unterschiedliche Standards der Branche strickt eingehalten werden und die Qualität durch die eingesetzten Entwicklungsprozesse gewährt wird, spricht nichts gegen den Einsatz in Serienprodukte. Ende 2011 konnte ArcCore bereits Kunden in Europa und Asien verzeichnen, darunter auch die ersten auf ArcCore basierten Produkte in Serie. Neben Automobilhersteller und deren Zulieferer zählen auch Universitäten und Forschungseinrichtungen zu den Kunden von ArcCore.
Um Arctic Core erfolgreich einsetzen zu können, sind entsprechende Kenntnisse über das Konzept und die Methodik von AUTOSAR erforderlich. Des Weiteren sind fortgeschrittene Fähigkeiten in der Programmiersprache C notwendig, um so den Quellcode von Arctic Core interpretieren zu können. Folgende Tabelle bewertet das Anwendungsgebiet und entsprechende Randbedingungen von Arctic Core.

Einarbeitungszeit	Die Einarbeitungszeit ist schwierig zu beurteilen, da es für den Einstieg wichtiger ist, fachspezifische Vorkenntnisse zu haben. Arctic Core selbst ist übersichtlich aufgebaut und gut dokumentiert, zusätzlich wird mithilfe von Demos das Zusammenspiel unterschiedlicher Teile gezeigt.
Produktivität	Der Aufwand welcher von Beginn eines neuen Projekts bis zur Endversion bzw. zum ersten Prototypen notwendig ist, lässt sich nicht generell bewerten. Vor allem, wenn eine Integration von Modulen aus Arctic Core in ein bereits vorhandenes Projekt angestrebt wird, hängt der Aufwand hauptsächlich von der Struktur des Projekts ab.
Anwendungsgebiet	Das Anwendungsgebiet der ArcCore Produkte lässt sich nicht verallgemeinern. Für Forschung und Entwicklung ist vor allem die Erstellung von Prototypen durch die Open-Source Produkte interessant, hingegen können Automobilhersteller und Zulieferer mit Hilfe der angebotenen Produkte ebenso eine Serienproduktion anstreben.
Weiterentwicklung	Da bereits die letzten AUTOSAR Standards in dem Arctic Core Paket implementiert wurden und der Erfolg von Arctic Core sehr groß ist, wird wahrscheinlich auch die zukünftige Entwicklung in diese Richtung weiter gehen.
Multi Core	Ein wichtiger Aspekt der Entwicklung und Programmierung von Steuergeräten im automotive Bereich ist der Einsatz von Mehrkern-Architekturen. Durch die Anlehnung an den aktuellen AUTOSAR Standard unterstützt auch Arctic Core den Einsatz von Mehrkern-Betriebssystemen nach AUTOSAR mit entsprechender Funktionalität.
Mikrocontroller Architektur	Das Arctic Core Paket ist standardmäßig für unterschiedliche Mikrocontroller Architekturen verfügbar: PowerPC, ARM (Cortex) und Renesas (RH850). Zusätzlich werden andere Architekturen nach Kundenanforderungen hinzugefügt.

Tabelle 7.6: Arctic Core Anwendungsgebiet

Die Integration der Arctic Core Module in das INCOBAT Projekt war vor allem durch die bereits vorhandene Hardwarespezifische CAN Kommunikationsimplementierung, welche statische Zugriffe auf diverse Mikrocontroller-Register durchführte, erschwert.
Eine Flexibilität bei der Integration der unterschiedlichen Module ist auf jeden Fall gegeben, so wurde beim Integrieren mit dem CanIf Modul begonnen und dieses auch einzeln getestet, ehe mit der weiteren Eingliederung und Überprüfung der Module PduR und COM fortgefahren wurde. Diese Flexibilität ist besonders für die Entwicklung von Prototypen interessant, da so schneller eine Lauffähige Software für reale Tests mit entsprechender

Funktionalität gegeben ist, welche durch wenig Modifikationen mit weiteren Modulen erweitert werden kann.

7.6 Benutzerfreundlichkeit

Ein weiterer wichtiger Aspekt bezüglich der Integration und Konfiguration von Software Modulen stellt die Benutzerfreundlichkeit dar. Ein benutzerfreundliches System erleichterte den Einstieg in die Arbeit und unterstützt die Implementierung von Beginn bis zum Abschluss des Projekts. Dazu zählen aber nicht nur die Bedienbarkeit der verwendeten Programme, sondern auch der Kontext dieser. Besonders eine gute Dokumentation und übersichtlicher Sourcecode sind erleichtern einen Softwareentwickler die Arbeit.
Da es ich bei Arctic Core um ein Softwarepaket, also eine Sammlung von C-Source und Header Dateien handelt, kann man hier eine Benutzerfreundlichkeit nur auf Basis der Dokumentation und den Support beurteilen. Da jedoch für die Integration und Konfiguration eine geeignete Entwicklungsumgebung eingesetzt wurden, wird diese im folgendem bezüglich deren Benutzerfreundlichkeit und Erweiterbarkeit betrachtet.

7.6.1 Entwicklungsumgebung Eclipse

Als Entwicklungsumgebung kommt Eclipse [16] zum Einsatz, welches für verschiedene Programmiersprachen eingesetzt werden kann und auch eine Reihe an Erweiterungen und Anpassungen zulässt. Für diese Erweiterungen steht der sogenannte Eclipse Marketplace [22] zur Verfügung. Zusätzlich bietet Eclipse die Möglichkeit Erweiterungen direkt aus anderen Marketplaces zu installieren.

Zusatzfunktionen	Eclipse lässt hinsichtlich der Erweiterungen kaum Wünsche offen. Besonders beliebt ist die Erweiterung für die Versionsverwaltung SVN. Die einzelnen Plugins werden hauptsächlich durch eine große Community entwickelt.
Schnittstellen	Für den Import von vorhanden Projekten bzw. den Export von Projekten stehen entsprechende Schnittstellen zur Verfügung. Des Weiteren steht auch eine API zur Verfügung, welche natürlich für die Entwicklung von Erweiterungen benötigt wird.
Updates	Die Versionen von Eclipse werden primär durch Projektnamen beschrieben, die aktuelle Version heißt "Mars" und wurde im Juni 2015 veröffentlicht. Die Namensgebung basiert auf einer alphabetischen Reihenfolge des ersten Buchstaben. Für die Entwicklung während dieser Arbeit wurde Eclipse "Luna" aus 2014 verwendet.

Tabelle 7.7: Entwicklungsumgebung

Die große Verteilung und Akzeptanz von Eclipse, gepaart mit den Erweiterungsmöglichkeiten, sind Indikatoren für die Qualität dieser Entwicklungsumgebung. Besonders die enorme Individualisierung mithilfe von Plugins war für die Implementierung hilfreich. Beispielsweise die Integration des Kompilier- und Portierungsprozesses direkt in die IDE, erleichterten die Arbeit enorm.

7.6.2 Entwicklungssupport

Zu einer optimalen Benutzerfreundlichkeit zählt auch ein umfangreicher Support für Entwickler, sei es um den Einstieg zu erleichtern oder als Unterstützung während der Projektlaufzeit.
Im Zuge dieser Arbeit wurde kein Support von Arctic Core beansprucht, somit kann dieser nicht beurteilt werden. ArcCore bietet jedoch unterschiedliche Schulungen und Unterstützungen an. Während die Schulungen hauptsächlich das Thema AUTOSAR fokussieren, stehen für die Integration und Entwicklung eigene Services zur Verfügung. Besonders wenn zum ersten Mal ein Projekt mit der kompletten Produktkette von ArcCore realisiert werden soll, ist entsprechender Integrationssupport ratsam, welcher einen Prozess aufsetzt, bei der Integration hilft und die Ergebnisse bewertet. Im Falle einer bereits bestehenden ArcCore Plattform, welche durch neue Module, Treiber oder Tools erweitert werden soll, bietet sich die Inanspruchnahme eines Entwicklungssupports an. [3]

Schulungen	Unterschiedlich intensive Schulungen sollen das AUTOSAR Konzept näher bringen und somit eine Basis für die weitere Arbeit mit ArcCore Produkte bilden. Des Weiteren werden auch Veranstaltungen für Universitäten und Forschungseinrichtungen angeboten.
Integrationssupport	Der Integrationssupport unterstützt beim Aufsetzten und Integrieren einer ArcCore basierten Lösung.
Entwicklungssupport	Für die Weiterentwicklung bereits vorhandener ArcCore Projekte steht der Entwicklungssupport zur Verfügung.

Tabelle 7.8: ArcCore Entwicklersupport

Die unterschiedlichen Servicetypen erscheinen als sehr sinnvoll, da sie den Prozess vom Einarbeiten in die AUTOSAR Thematik über das Durchführen eines ersten Projektes bis hin zur Erweiterung und Wartung von vorhanden Projekten abdecken.

7.6.3 Dokumentation

Ein sehr wichtiger Punkt, hinsichtlich der Benutzerfreundlichkeit ist eine geeignete Dokumentation. Während der Integration und der Konfiguration der Module ist die Dokumentation oftmals die erste Anlaufstelle. In folgender Tabelle werden unterschiedliche Faktoren

in Bezug auf die Dokumentation näher betrachtet.
ArcCore biete einer Benutzerdokumentation für die Produkte Arctic Studio und Arctic Core an, welche für die verschiedenen Versionen verfügbar ist. Diese Dokumentationen enthalten sogenannte "Getting Started" Anleitungen, welche den Einstieg erleichtern. Zusätzlich sind Hinweise zu den einzelnen Produktversionen und eine Auflistung typischer Fehler vorhanden. ArcCore bietet nicht für alle Funktionen eine Dokumentation an, sondern verweist bei AUTOSAR konformen Funktionen, durch ein Kommentar im Sourcecode, auf den entsprechenden Abschnitt der offiziellen AUTOSAR Spezifikation [8].
Tabelle 7.9 betrachtet unterschiedliche Faktoren der Dokumentation des Arctic Core Pakets näher, auf die Dokumentation des Arctic Studio wird hier nicht näher eingegangen.

Verfügbarkeit	Die Dokumentation von Arctic Core ist Online verfügbar [5], jedoch benötigt man für den Zugang einen ArcCore Benutzeraccount. Da für die Durchführung dieser Arbeit eine Evaluierungslizenz angefordert wurde, war auch ein entsprechender Account vorhanden.
Vollständigkeit	Der Arctic Core Teil der Dokumentation umfasst vor allem eine detaillierte Beschreibung der Konfiguration der verschiedenen Modulen, wie auch eine Erklärung über das Zusammenwirken dieser. Ein großer Teil wird hierzu dem Kommunikationsstapel gewidmet, vor allem die Dokumentation des CAN Aufbaus und der COM Konfiguration waren für die Durchführung dieser Arbeit hilfreich.
Aktualität	Für die aktuelle Version 9.0.0 von Arctic Core, welche für diese Arbeit verwendet wurde, steht eine Dokumentation zur Verfügung. Des Weiteren existieren auch die Dokumentationen für die Versionen 8.x.x und 7.x.x.
Mehrsprachigkeit	ArcCore bietet die Dokumentationen der einzelnen Produkte in Englisch an.
Fehlerdokumentation	Eine umfangreiche Dokumentation von Fehler existiert nicht, es werden nur typische Fehler, welche beim Kompilieren der Software auftreten, erläutert.
Quellcode	Der Quellcode ist zum Teil mit Kommentaren dokumentiert. Bei AUTOSAR konformen Funktionen bestehen Verweise auf die offiziellen AUTOSAR Dokumentationen. Quelltext 7.1 zeigt einen Ausschnitt der Datei `Com_Com.h`, welche Definitionen für COM API-Funktionen und entsprechende Vermerke (`@req COM197` und `@req COM198`), welche auf die Funktionsdokumentation in der COM AUTOSAR Dokumentation verweisen [9], enthält.

Demos	Das Softwarepaket Arctic Core enthält unterschiedliche Demoprojekte, welche für den Einstieg sehr hilfreich sind, da hier das Zusammenwirken unterschiedlicher Module gut erkennbar ist.

Tabelle 7.9: Arctic Core Dokumentation

Quelltext 7.1: Dokumentationsverweise von COM API-Funktionen

```
/* @req COM197 */
Com_SendSignal(Com_SignalIdType SignalId, const void *SignalDataPtr);

/* @req COM198 */
Com_ReceiveSignal(Com_SignalIdType SignalId, void* SignalDataPtr);
```

7.7 Konfiguration

Die Konfiguration wird hinsichtlich unterschiedlicher Erfolgsfaktoren betrachtet. Neben den Aufwand für eine Erstkonfiguration, sind auch Faktoren wie die Erweiterbarkeit, die Skalierbarkeit oder die Wiederverwendbarkeit zu beachten. Zusätzlich ist auch die Unterstützung während der Konfiguration ein wichtiges Kriterium.

Der folgende Abschnitt wird sich mit Themen rund um die Konfiguration der Arctic Core Module befassen. Neben der Integration der einzelnen Module ist deren Konfiguration hinsichtlich des Einsatzgebietes durchzuführen. Einerseits ist, besonders für das CAN Modul, eine Konfiguration an die Systemumgebung anzupassen um eine AUTOSAR konforme Kommunikation mit dem Infineon Low Level Driver zu ermöglichen, andererseits sind alle Relevanten CAN Botschaften und deren Signale aufzunehmen. Nachdem die Grundkonfiguration der Module erfolgt war, wurden sämtliche Nachrichten und Signale in die entsprechenden Einstellungen aufgenommen, das bedeutet ein Hinzufügen von etwa 20 Zeilen pro Signal und pro Nachricht in verschiedene Dateien. Für das BMS des INCOBAT Projekts existieren 25 Botschaften mit ca. 90 Signale.
Zusätzlich zu den eigentlichen Konfigurationen ist auch die Möglichkeit der Erweiterung um zusätzliche AUTOSAR konforme Module ein interessanter Aspekt.

Konfigurationsaufwand	Der Aufwand der Konfiguration ist schwer zu bewerten, da dieser natürlich stark von den Anforderungen abhängig ist. Grundsätzlich ist die Grundkonfiguration, welche notwendig ist damit die Module einsatzbereit sind, der Hauptteil, obwohl das Einbinden der Nachrichten bzw. Signale in diesem Fall mehr Aufwand darstellt.

Konfigurationsunterstützung	Im Zuge dieser Arbeit wurden für die Konfiguration lediglich die Dokumentation und die vorhandenen Demos als Unterstützung verwendet. Der Einsatz des Arctic Studio würde die Konfiguration noch weiter unterstützen und erleichtern.
Komplexität	Wichtig während der Konfiguration ist es, den Überblick zu bewahren, besonders die internen Funktionsaufrufe der verschiedenen Module erfordern viele zusätzliche Definitionen, welche zum Beispiel für die Zuordnung von einem COM Signal zu einer PDU im CAN Interface über das PduR Modul dienen.
Erweiterbarkeit	Die Erweiterung um zusätzliche CAN Botschaften lässt sich im diesem Projekt durch entsprechende Konfigurationserweiterungen und zusätzliche Definitionen realisieren. Der modulare AUTOSAR basierte Aufbau von Artic Core ermöglicht auch das Erweitern des Projekts um zusätzliche Module.
Austauschbarkeit	Die Methodik von AUTOSAR sieht eine leichte Austauschbarkeit von einzelnen Modulen vor, welche durch genaue Schnittstellendefinitionen ermöglicht wird. Dadurch ist es möglich, einzelne Module oder gar das Betriebssystem durch kompatible Implementierungen mit wenig Aufwand auszutauschen.
Kompatibilität	Da die verwendeten Module streng an den AUTOSAR Standard angelehnt sind, ist automatisch eine Kompatibilität zu Modulen anderer Hersteller gegeben, solange diese auch den AUTOSAR Standard implementieren. Natürlich sind evtl. kleinere Anpassungen notwendig, vor allem wenn man selbst bereits an der Implementierung oder Definition der Module Änderungen durchgeführt hat.
ASW Voraussetzungen	Damit die ASW die durch die Module angebotene API Nutzen kann, sind die entsprechenden Funktionen korrekt zu verwenden. In Kapitel 6.3 werden Beispiele solcher Aufrufe näher betrachtet.
Skalierung	Während dieser Arbeit wurde zuerst das CAN Interface integriert und so konfiguriert, dass erste kleine Tests erfolgen konnten. Erst danach wurde mit dem weiteren Kommunikationsstapel durch die Integration der Module PduR und COM fortgefahren. Diese wurden ebenso mit Minimalkonfiguration getestet, ehe die Konfigurationen für das Einsatzgebiet erweitert wurden. Die Skalierung hinsichtlich der einzelnen Konfigurationen läuft somit problemlos ab.

Wiederverwendbarkeit	Ein Projekt, welches den Aufbau des AUTOSAR Standards mithilfe des Arctic Core Pakets umsetzt, lässt sich leicht für weitere Projekte wiederverwenden, in dem man die ASW austauscht und notwendige Anpassungen der unterschiedlichen Konfigurationen vornimmt.

Tabelle 7.10: Arctic Core Konfiguration

Die Konfiguration der verwendeten Arcitc Core Module lassen sich mit Hilfe der Dokumentationen und dem notwendigen Wissen über den AUTOSAR Standard und dem CAN Bus mit überschaubaren Aufwand realisieren. Eine großer Herausforderung der Konfiguration war auch das Anpassen der Module CanIf bzw. Can, um die Kommunikation mit dem Infineon Low Level Driver zu ermöglichen. Hierfür war besonders der Einsatz des Debuggers von Lauterbach hilfreich, um das interne Verhalten des CAN Controllers nachvollziehen zu können.

7.8 Integration von externen Tools

Für die Entwicklung der Prototypenplattform wurden mehrere Tools eingesetzt, welche die sogenannte Toolchain darstellen. Zusätzlich kam auch weitere Hardware als Unterstützung zum Einsatz.

Neben der Entwicklungsumgebung wird ein geeigneter Compiler benötigt, um einen für den AURIX TC275 Mikrocontroller ausführbaren Binärcode zu erzeugen, man erhält für jeden Kern des Controllers eine eigene Binärdatei. Für den Vorgang des Kompilierens wurde eine entsprechende Build-Umgebung, welches das Tool "make" nutzt, verwendet. Damit ist es zum Beispiel möglich, nur den relevanten Teil für einen CPU-Kern zu kompilieren. Für die Portierung dieser Binärcodes wird als nächstes eine geeignete Hardwareverbindung zwischen dem PC und dem TC275 benötigt, über welche die Binärdateien auf das Board portiert werden können, dazu wird wiederum eine passende Software eingesetzt. Um nun die neue Konfiguration testen bzw. deren Verhalten beobachten zu können, wird einerseits ein Debugger benötigt und andererseits wird mit einer eigenen Umgebung der CAN Bus extern überwacht und auch beschrieben.

Tasking Compiler	Als Compiler kommt der Altium Tasking Compiler 4.3R1 [2] zum Einsatz, dieser erzeugt für jeden Kern des AURIX TC275 Mikrocontrollers eine separate ausführbare Binärcode. Der Build-Prozess wurde durch den Einsatz des Build-Tools "make" vereinfacht.
Trace32	Das Programm Trace32 von Lauterbach [35] ermöglicht ein Portieren der Binärdateien auf die Hardware. Dazu wird ein Debugger, welcher über die JTAG Schnittstelle der Hardware und über USB mit dem Entwicklungscomputer verbunden ist, eingesetzt. Die Kombination aus dem Debugger und dem Trace32 Programm, ermöglicht nun auch einen Zugriff auf sämtliche Register des Controllers, wie auch ein Debuggen der portierten Software.
CANalyzer	Um als letzten Punkt das Verhalten des CAN Kommunikationsstapel der portierten Software testen zu können, kommt der sogenannte CANalyzer der Firma Vector [58] zum Einsatz. Dieser ermöglicht über ein CANcase, welches über eine USB Schnittstelle verbunden ist, ein Lesen und Schreiben am CAN Bus und stellt somit einen Busteilnehmer dar.

Tabelle 7.11: Zusätzliche Tools

Mithilfe der erwähnten Programme und Hardware wurde die Implementierung, das Portieren und das Testen stark erleichtert. Im Zuge dieser Arbeit konnte ein reibungsloser Ablauf dieser Toolchain beobachtet werden.

7.9 Zusammenfassung der Bewertung

Die in diesem Kapitel betrachteten und bewerteten Erfolgsfaktoren fassen die Erkenntnisse dieser Arbeit zusammen. Die Bewertungen der ArcCore Produktpalette, den angebotenen Services sowie dem Arctic Core Paket bilden als Einheit einen Überblick bezüglich des Einstiegs in diese Thematik. Für den Einstieg ist jedoch ein umfangreiches fachspezifisches Wissen bezüglich Standards, Protokolle und Programmiersprachen notwendig.
Die Erläuterung der möglichen Anwendungsgebiete zeigt, dass Arctic Core aufgrund der Qualität und Verfügbarkeit, einerseits hervorragend für die Forschung und Entwicklung geeignet ist, andererseits sind Entwicklungen für Serienproduktionen mithilfe der ArcCore Produktpalette, den angebotenen Schulungen und dem umfangreichen Support ebenso realisierbar.
Die im Zuge dieser Arbeit eingesetzte Entwicklungsumgebung Eclipse erweist sich vor allem durch die zahlreichen Möglichkeiten der Erweiterungen als besonders geeignet. Obwohl Eclipse grundsätzlich eine für Java optimierte IDE darstellt, lässt es sich mit entsprechenden Erweiterungen auch für die Softwareentwicklung im automotive Bereich nutzen.

Während der Integration und Konfiguration der Module aus dem Arctic Core Standardpaket, waren auch die entsprechende Dokumentationen von ArcCore, sowie auch die Unterlagen von AUTOSAR, sehr hilfreich. Die Konfiguration der Module teilt sich in zwei Teile: zuerst wird eine Grundkonfiguration vorgenommen, um die Module nach und nach miteinander zu verbinden und zu aktivieren, danach erfolgt die Konfiguration hinsichtlich des Einsatzgebietes, also die Aufnahme der einzelnen CAN Nachrichten und deren Signale in die entsprechenden Konfigurationsstrukturen.
Für das Kompilieren, Debuggen und Testen wurden noch weiter Programme und auch zusätzliche Hardware eingesetzt, welche sich während der Durchführung als eine sehr gut geeignete Kombination an unterschiedlichen Tools herausstellten.

Kapitel 8

Zusammenfassung und Ausblick

In diesem letzten Kapitel wird eine Zusammenfassung der Ergebnisse dieser Arbeit, sowie ein Ausblick auf mögliche Erweiterungen an der Prototypenplattform hinsichtlich des AUTOSAR Standards gegeben.

8.1 Zusammenfassung

Die Anforderungen an die Rechenleistung eines Prozessors sind in den letzten Jahren in den verschiedensten Bereichen stark gestiegen. Davon blieb auch der automotive Bereich nicht aus. Während im Consumer Bereich schon vor vielen Jahren diesen steigenden Anforderungen durch die Einführung von Mehrkern-Architekturen entgegen gewirkt wurde, ist dieser Trend erst in letzter Zeit für den Einsatz im Automobil zu verzeichnen. Aufgrund der großen Anzahl an ECUs in einem modernen KFZ, kommt es auch zu einer enormen Komplexität der Vernetzung dieser ECUs untereinander. Der Umstieg auf Mehrkern-Mikrocontroller soll somit einerseits die notwendige Rechenleistung für die derzeitigen Anforderungen zur Verfügung stellen, andererseits wird durch die Zentralisierung von mehreren Funktionen in eine ECU eine Reduzierung der Komplexität erreicht. Jedoch bringt dieser Umstieg auch eine Reihe von Herausforderungen mit sich und führt zu einer komplexeren Software. Zusätzlich muss man sich Gedanken um die Verteilung unterschiedlicher Funktionen und das gemeinsame Verwenden von Ressourcen machen.

Während dieser Arbeit entstand in Zusammenarbeit mit der AVL List GmbH eine Prototypen-Platform für ein Forschungsprojekt. Diese Projekt befasst sich unter anderem mit der Entwicklung eines geeigneten Battery Management System, welches die Performance und somit die Akzeptanz von Elektrofahrzeugen erhöhen soll. Dazu wurde die bereits vorhandene Software analysiert und die Integration und Konfiguration von mehreren Modulen zur Abbildung des CAN Kommunikationsstapel durchgeführt. Diese Module stammen aus dem Open-Source Standardpaket von Arctic Core und sind an dem Standard AUTOSAR angelehnt. Dieses Vorgehen steigert einerseits die Qualität der Software, andererseits wird durch die Methodik von AUTOSAR ein modularer und flexibler Aufbau der Software erreicht.

Diese Implementierung würde grob in zwei Teile gegliedert: der Integration der Module und deren Konfiguration. Während der Integration wurden die benötigten Dateien aus dem

Artic Core Paket analysiert, in das vorhanden Projekt portiert und durch eine angepasste Minimalkonfiguration lauffähig gemacht. Dadurch konnten die einzelnen Module bereits vorab getestet werden, um so die Basis für den zweiten Teil, der Konfiguration, zu legen. Die Konfiguration befasst sich nun mit der Eingliederung der relevanten CAN Nachrichten und deren Signale in die Konfigurationen der unterschiedlichen Module. Dadurch wird der ASW ein AUTOSAR konformer API Zugriff auf die Funktionalitäten des CAN Bus ermöglicht.

Das Ergebnis dieser Arbeit stellt nun einen AUTOSAR konformen Kommunikationsstapel für die CAN-Kommunikation im BMS des INCOBAT Forschungsprojektes zur Verfügung und trägt damit zur Erreichung der Ziele dieses Projekts bei.

Im Kapitel 7 "Bewertung der Erfolgsfaktoren" wurden die Ergebnisse dieser Arbeit und die dafür verwendete Entwicklungsumgebung anhand verschiedener Erfolgsfaktoren bewertet. Dadurch wird ein Überblick von den Vorbedingungen über die verwendeten Tools bis hin zur Konfiguration gegeben.

8.2 Ausblick

Die Implementierungen im Zuge dieser Arbeit zeigen das Einbinden von AUTOSAR konformen Modulen in ein bereits vorhandenes Prototypen-Projekt. Durch den Erfolg dieser Umsetzung ist eine gute Basis für die Integration weiterer AUTOSAR Module gegeben, um somit die Qualität, Flexibilität und Akzeptanz dieser Plattform weiter zu erhöhen. Durch diese Standardisierung erreicht man eine besonders gute Prototypenplattform, welche als Basis für zukünftige Projekte dient. Dadurch wird eine Entwicklung dem weitverbreiteten AUTOSAR Standard ermöglicht. Eine entsprechend konfigurierte BSW, welche der AUTOSAR Spezifikation folgt, führt dadurch zu qualitativ hochwertigen Prototypen, welcher mit geringerem Aufwand erstellt werden können. Des weiteren stellt solch ein Standardisierung einer perfekte Basis für die Portierung zu einem Serienprodukt dar. Eine modulare und standardisierte Softwarearchitektur, welche die gesamte AUTOSAR Architektur abbildet, ist hinsichtlich der komplexen Funktionen im automotive Bereich der beste Weg zu qualitativ hochwertigen Produkten und sollte somit angestrebt werden.

Anhang A

Anhang

A.1 Konfiguration des CAN Moduls (Auszug)

Quelltext A.1: Konfiguration des CAN Moduls (Auszug)

```
/*
 * Generator version: 5.0.0
 * AUTOSAR version:   4.1.2
 */

#include <stdlib.h>
#include "Can.h"
#include "CanIf_Cbk.h"

#include "CanIf.h"

PduIdType  Can_swPduHandles_CanController[5];

#define BSW_START_SEC_CONST_UNSPECIFIED
#include "MemMap.h"

const Can_HardwareObjectType CanHardwareObjectConfig_CanController[] = {
    {
        .CanObjectId        =   CanConf_CanHardwareObject_HWObj_1,
        .CanHandleType      =   CAN_ARC_HANDLE_TYPE_BASIC,
        .CanIdType          =   CAN_ID_TYPE_STANDARD,
        .CanObjectType      =   CAN_OBJECT_TYPE_RECEIVE,
    },
    {
        .CanObjectId        =   CanConf_CanHardwareObject_HWObj_2,
        .CanHandleType      =   CAN_ARC_HANDLE_TYPE_BASIC,
        .CanIdType          =   CAN_ID_TYPE_STANDARD,
        .CanObjectType      =   CAN_OBJECT_TYPE_TRANSMIT,
    },
};



const uint8 Can_MailBoxToSymbolicHrh_CanController[]= {
    CanConf_CanHardwareObject_HWObj_1,

};

const Can_ControllerBaudrateConfigType Can_SupportedBaudrates_CanController[] =
{
    {
        .CanControllerBaudRate =        500,
        .CanControllerPropSeg =         7,
        .CanControllerSeg1 =            4,
        .CanControllerSeg2 =            4,
        .CanControllerSyncJumpWidth =   2,
    },
};

const Can_ControllerConfigType CanControllerConfigData[] =
{
    {
        .CanControllerActivation =  TRUE,
        .CanHwUnitId =          FLEXCAN_A,
        .CanControllerDefaultBaudrate = 500,
        .CanControllerSupportedBaudrates = Can_SupportedBaudrates_CanController,
        .CanControllerSupportedBaudratesCount = 1,
        .Can_Arc_Flags =            (CAN_CTRL_BUSOFF_PROCESSING_INTERRUPT |
                                     CAN_CTRL_RX_PROCESSING_INTERRUPT |
                                     CAN_CTRL_TX_PROCESSING_INTERRUPT |
```

```
                                    0 |
                                    CAN_CTRL_ACTIVATION |
                                    0 |
                                    0 |
                                    0),
        .CanWakeupSourceRef =       0, // Unsupported
        .Can_Arc_Hoh =              &CanHardwareObjectConfig_CanController[0],
        .Can_Arc_HohCnt =           2,
#if defined(CFG_CAN_USE_SYMBOLIC_CANIF_CONTROLLER_ID)
        .Can_Arc_CanIfControllerId = CanIfConf_CanIfCtrlCfg_CanIfCtrlCfg,
#endif
    },
};

const uint8 Can_HthToSymbolicCtrl[] = {
    [CAN_ARC_CANHARDWAREOBJECT_HWOBJ_2] = CanConf_CanController_CanController,
};

const uint8 Can_HthToHohMap[] = {

    [CAN_ARC_CANHARDWAREOBJECT_HWOBJ_2] = 1,
};

const uint8 Can_HwUnitToControllerId[] = {
     [FLEXCAN_A] = CanConf_CanController_CanController,
};

const Can_HwHandleType Can_SymbolicHohToInternalHohMap [] = {
    [CanConf_CanHardwareObject_HWObj_1] = CAN_ARC_CANHARDWAREOBJECT_HWOBJ_1,
    [CanConf_CanHardwareObject_HWObj_2] = CAN_ARC_CANHARDWAREOBJECT_HWOBJ_2,
};

const Can_CallbackType CanCallbackConfigData = {
      NULL, //CanIf_CancelTxConfirmation
    CanIf_RxIndication,
    CanIf_ControllerBusOff,
    CanIf_TxConfirmation,
    NULL, //CanIf_ControllerWakeup,
    NULL,
    CanIf_ControllerModeIndication,
};

const Can_ConfigSetType CanConfigSetData =
{
    .CanController = CanControllerConfigData,
    .CanCallbacks = &CanCallbackConfigData,
    .ArcHthToSymbolicController = Can_HthToSymbolicCtrl,
    .ArcHthToHoh = Can_HthToHohMap,
    .ArcSymbolicHohToInternalHoh =  Can_SymbolicHohToInternalHohMap,
    .ArcHwUnitToController = Can_HwUnitToControllerId,
};

const Can_ConfigType CanConfigData = {
    .CanConfigSetPtr =  &CanConfigSetData,
};

#define BSW_STOP_SEC_CONST_UNSPECIFIED
#include "MemMap.h"
```

A.2 Konfiguration des CanIf Moduls (Auszug)

Quelltext A.2: Konfiguration des CanIf Moduls (Auszug)

```
/*
 * Generator version: 5.0.0
 * AUTOSAR version:   4.0.3
 */

#include "CanIf.h"
#include "CanIf_PBCfg.h"

#if defined(USE_CANTP)
#include "CanTp.h"
#include "CanTp_Cbk.h"
#endif

#if defined(USE_J1939TP)
#include "J1939Tp.h"
#include "J1939Tp_Cbk.h"
#endif

#if defined(USE_CANNM)
#include "CanNm.h"
#include "CanNm_PBCfg.h"
#endif

#if defined(USE_PDUR)
#include "PduR.h"
#include "PduR_PbCfg.h"
#endif
#if defined(USE_XCP)
#include "Xcp.h"
#include "XcpOnCan_Cbk.h"
#endif

SECTION_POSTBUILD_DATA const CanIf_HthConfigType
    CanIfHthConfigData_CanIfInitHohCfg[] =
{
    {
        .CanIfHthType               = CAN_ARC_HANDLE_TYPE_BASIC,
        .CanIfCanControllerIdRef    = CanIfConf_CanIfCtrlCfg_CanIfCtrlCfg,
        .CanIfHthIdSymRef           = CanConf_CanHardwareObject_HWObj_2,
        .CanIf_Arc_EOL              = TRUE,
    },
};

SECTION_POSTBUILD_DATA const CanIf_HrhConfigType
    CanIfHrhConfigData_CanIfInitHohCfg[] =
{
    {
        .CanIfHrhType               = CAN_ARC_HANDLE_TYPE_BASIC,
        .CanIfSoftwareFilterHrh     = TRUE,
        .CanIfCanControllerHrhIdRef = CanIfConf_CanIfCtrlCfg_CanIfCtrlCfg,
        .CanIfHrhIdSymRef           = CanConf_CanHardwareObject_HWObj_1,
        .CanIf_Arc_EOL              = TRUE,
    },
};

SECTION_POSTBUILD_DATA const CanIf_InitHohConfigType CanIfHohConfigData[] = {
    {
        .CanIfHrhConfig     = CanIfHrhConfigData_CanIfInitHohCfg,
        .CanIfHthConfig     = CanIfHthConfigData_CanIfInitHohCfg,
        .CanIf_Arc_EOL      = TRUE,
```

```
    },
};

SECTION_POSTBUILD_DATA const CanIf_TxBufferConfigType CanIfBufferCfgData[] = {
    {
        .CanIfBufferSize = 0,
        .CanIfBufferHthRef = &CanIfHthConfigData_CanIfInitHohCfg[0],
        .CanIf_Arc_BufferId = 0
    },
};

SECTION_POSTBUILD_DATA const CanIf_TxPduConfigType CanIfTxPduConfigData[] = {
    {
        .CanIfTxPduId               = PDUR_REVERSE_PDU_ID_FREQIND,
        .CanIfCanTxPduIdCanId       = 258,
        .CanIfCanTxPduIdDlc         = 8,
        .CanIfCanTxPduType          = CANIF_PDU_TYPE_STATIC,
        .CanIfTxPduPnFilterEnable   = STD_OFF,
#if ( CANIF_PUBLIC_READTXPDU_NOTIFY_STATUS_API == STD_ON )
        .CanIfReadTxPduNotifyStatus = FALSE,
#endif
        .CanIfTxPduIdCanIdType      = CANIF_CAN_ID_TYPE_11,
        .CanIfUserTxConfirmation    = PDUR_CALLOUT,
        /* [CanIfBufferCfg] */
        .CanIfTxPduBufferRef        = &CanIfBufferCfgData[0],
        .PduIdRef                   = NULL,
    },
};

SECTION_POSTBUILD_DATA const CanIf_RxPduConfigType CanIfRxPduConfigData[] = {
        {
            .CanIfCanRxPduId            = PDUR_PDU_ID_RX_PDU,
            .CanIfCanRxPduLowerCanId    = 0x200,
            .CanIfCanRxPduUpperCanId    = 0x200,
            .CanIfCanRxPduDlc           = 8,
    #if ( CANIF_PUBLIC_READRXPDU_DATA_API == STD_ON )
            .CanIfReadRxPduData         = FALSE,
    #endif
    #if ( CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API == STD_ON )
            .CanIfReadRxPduNotifyStatus = FALSE,
    #endif
            .CanIfCanRxPduHrhRef        = &CanIfHrhConfigData_CanIfInitHohCfg[0],
            .CanIfRxPduIdCanIdType      = CANIF_CAN_ID_TYPE_11,
            .CanIfUserRxIndication      = PDUR_CALLOUT,
            .CanIfCanRxPduCanIdMask     = 0x7FF,
            .PduIdRef                   = NULL,
        },
        {
            .CanIfCanRxPduId            = PDUR_PDU_ID_RX_PDU,
            .CanIfCanRxPduLowerCanId    = 0x201,
            .CanIfCanRxPduUpperCanId    = 0x201,
            .CanIfCanRxPduDlc           = 8,
    #if ( CANIF_PUBLIC_READRXPDU_DATA_API == STD_ON )
            .CanIfReadRxPduData         = FALSE,
    #endif
    #if ( CANIF_PUBLIC_READRXPDU_NOTIFY_STATUS_API == STD_ON )
            .CanIfReadRxPduNotifyStatus = FALSE,
    #endif
            .CanIfCanRxPduHrhRef        = &CanIfHrhConfigData_CanIfInitHohCfg[0],
            .CanIfRxPduIdCanIdType      = CANIF_CAN_ID_TYPE_11,
            .CanIfUserRxIndication      = PDUR_CALLOUT,
            .CanIfCanRxPduCanIdMask     = 0x7FF,
            .PduIdRef                   = NULL,
        },
};
```

```
SECTION_POSTBUILD_DATA const CanIf_TxBufferConfigType *const
    CanIfCtrlCfg_BufferList[] = {
    /* CanIfBufferCfg */
    &CanIfBufferCfgData[0],
};


SECTION_POSTBUILD_DATA const CanIf_Arc_ChannelConfigType CanIf_Arc_ChannelConfig[
    CANIF_CHANNEL_CNT] = {
    {
        /* CanIfCtrlCfg */
        .CanControllerId        = CanConf_CanController_CanController,
        .NofTxBuffers           = 1,
        .TxBufferRefList        = CanIfCtrlCfg_BufferList,
        .CanIfCtrlWakeUpSupport = STD_OFF,
        .CanIfCtrlWakeUpSrc     = 0,
        .CanIfCtrlPnFilterSet   = STD_OFF,
    },
};


// This container contains the init parameters of the CAN
// Multiplicity 1.
SECTION_POSTBUILD_DATA const CanIf_InitConfigType CanIfInitConfig =
{
    .CanIfConfigSet                     = 0, // Not used
    .CanIfNumberOfCanRxPduIds           = CAN_PORT0_RXMSGSIZE,
    .CanIfNumberOfCanTXPduIds           = 2,
    .CanIfNumberOfDynamicCanTXPduIds    = 0, // Not used
    .CanIfNumberOfTxBuffers             = 1,

    // Containers
    .CanIfBufferCfgPtr                  = CanIfBufferCfgData,
    .CanIfHohConfigPtr                  = CanIfHohConfigData,
    .CanIfRxPduConfigPtr                = CanIfRxPduConfigData,
    .CanIfTxPduConfigPtr                = CanIfTxPduConfigData,
};

// This container includes all necessary configuration sub-containers
// according the CAN Interface configuration structure.
SECTION_POSTBUILD_DATA const CanIf_ConfigType CanIf_Config =
{
    .InitConfig                     = &CanIfInitConfig,
    .CanIfTransceiverConfig         = NULL,
    .Arc_ChannelConfig              = CanIf_Arc_ChannelConfig,
};
```

A.3 Konfiguration des COM Moduls (Auszug)

Quelltext A.3: Konfiguration des COM Moduls (Auszug)

```
/*
 * Generator version: 1.0.0
 * AUTOSAR version:   4.0.3
 */

#define USE_PDUR
#if defined(USE_PDUR)
#include "PduR.h"
#include "PduR_PbCfg.h"
#endif

#include "CanIf_PBCfg.h"

#if defined(USE_CANTP)
#include "CanTp.h"
#include "CanTp_PBCfg.h"
#endif

#include "Com.h"
#include "Com_PbCfg.h"

#if (COM_MAX_BUFFER_SIZE < 384)
#error Com: The configured ram buffer size is less than required! (384 bytes
    required)
#endif
#if (COM_MAX_N_IPDUS < 4)
#error Com: Configured maximum number of Pdus is less than the number of Pdus in
    configuration!
#endif
#if (COM_MAX_N_SUPPORTED_IPDU_COUNTERS < 0)
#error Com: Configured maximum number of Pdus with counter(
    ArcMaxNumberOfSupportedIPduCounters) is less than the number of Pdus
    configured with counter !
#endif
#if (COM_MAX_N_SIGNALS < 5)
#error Com: Configured maximum number of signals is less than the number of
    signals in configuration!
#endif
#if (COM_MAX_N_GROUP_SIGNALS < 0)
#error Com: Configured maximum number of groupsignals is less than the number of
    groupsignals in configuration!
#endif
#if (COM_MAX_N_SUPPORTED_GWSOURCE_DESCRIPTIONS < 0)
#error Com: Configured maximum number of gateway source description is less than
    the number of ComGwSourceDescription in configuration!
#endif

#define SECTION_POSTBUILD_DATA

#define BSW_START_SEC_CONST_UNSPECIFIED
#include "MemMap.h"

/*
 * Signal init values.
 */
SECTION_POSTBUILD_DATA const uint8 Com_SignalInitValue_TiBatOff = 0x0;
SECTION_POSTBUILD_DATA const uint8 Com_SignalInitValue_BcuSt = 0x0;
SECTION_POSTBUILD_DATA const uint8 Com_SignalInitValue_SrvCmd = 0x0;
SECTION_POSTBUILD_DATA const uint8 Com_SignalInitValue_MdulBalnAcv = 0x0;
SECTION_POSTBUILD_DATA const uint16 Com_SignalInitValue_MCooltFlowMdul = 0x0;
```

```
SECTION_POSTBUILD_DATA const uint8 Com_SignalInitValue_TCooltMdulIn = 0x0;
...


//SECTION_POSTBUILD_DATA const uint16 Com_SignalInitValue_TiBatOff = 0x0;


/*
 * Group signal definitions
 */
SECTION_POSTBUILD_DATA const ComGroupSignal_type ComGroupSignal[] = {
    {
        .Com_Arc_EOL = 1
    }
};

/*
 * Signal definitions
 */

SECTION_POSTBUILD_DATA const ComSignal_type ComSignal[] = {
        {
                .ComHandleId                = ComConf_ComSignal_RX_TiBatOff,
                .ComIPduHandleId            =
                    ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
                .ComFirstTimeoutFactor      = 0,
                .ComNotification            = COM_NO_FUNCTION_CALLOUT,
                .ComTimeoutFactor           = 0,
                .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
                .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
                .ComTransferProperty        = PENDING,
                .ComUpdateBitPosition       = 0,
                .ComSignalArcUseUpdateBit   = FALSE,
                .ComSignalInitValue         = &Com_SignalInitValue_TiBatOff,
                .ComBitPosition             = 0,
                .ComBitSize                 = 16,
                .ComSignalEndianess         = COM_LITTLE_ENDIAN,
                .ComSignalType              = UINT16,
                .Com_Arc_IsSignalGroup      = 0,
                .ComGroupSignal             = NULL,
                .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
                .ComSigGwRoutingReq         = FALSE,
                .Com_Arc_EOL                = 0
            },
            {
                .ComHandleId                = ComConf_ComSignal_RX_BcuSt,
                .ComIPduHandleId            =
                    ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
                .ComFirstTimeoutFactor      = 0,
                .ComNotification            = COM_NO_FUNCTION_CALLOUT,
                .ComTimeoutFactor           = 0,
                .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
                .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
                .ComTransferProperty        = PENDING,
                .ComUpdateBitPosition       = 0,
                .ComSignalArcUseUpdateBit   = FALSE,
                .ComSignalInitValue         = &Com_SignalInitValue_BcuSt,
                .ComBitPosition             = 16,
                .ComBitSize                 = 2,
                .ComSignalEndianess         = COM_LITTLE_ENDIAN,
                .ComSignalType              = UINT8,
                .Com_Arc_IsSignalGroup      = 0,
                .ComGroupSignal             = NULL,
                .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
                .ComSigGwRoutingReq         = FALSE,
                .Com_Arc_EOL                = 0
            },
```

```
    {
        .ComHandleId                = ComConf_ComSignal_RX_SrvCmd,
        .ComIPduHandleId            =
            ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
        .ComFirstTimeoutFactor      = 0,
        .ComNotification            = COM_NO_FUNCTION_CALLOUT,
        .ComTimeoutFactor           = 0,
        .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
        .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
        .ComTransferProperty        = PENDING,
        .ComUpdateBitPosition       = 0,
        .ComSignalArcUseUpdateBit   = FALSE,
        .ComSignalInitValue         = &Com_SignalInitValue_SrvCmd,
        .ComBitPosition             = 18,
        .ComBitSize                 = 2,
        .ComSignalEndianess         = COM_LITTLE_ENDIAN,
        .ComSignalType              = UINT8,
        .Com_Arc_IsSignalGroup      = 0,
        .ComGroupSignal             = NULL,
        .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
        .ComSigGwRoutingReq         = FALSE,
        .Com_Arc_EOL                = 0
    },
    {
        .ComHandleId                = ComConf_ComSignal_RX_MdulBalnAcv,
        .ComIPduHandleId            =
            ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
        .ComFirstTimeoutFactor      = 0,
        .ComNotification            = COM_NO_FUNCTION_CALLOUT,
        .ComTimeoutFactor           = 0,
        .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
        .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
        .ComTransferProperty        = PENDING,
        .ComUpdateBitPosition       = 0,
        .ComSignalArcUseUpdateBit   = FALSE,
        .ComSignalInitValue         = &Com_SignalInitValue_MdulBalnAcv,
        .ComBitPosition             = 24,
        .ComBitSize                 = 8,
        .ComSignalEndianess         = COM_LITTLE_ENDIAN,
        .ComSignalType              = UINT8,
        .Com_Arc_IsSignalGroup      = 0,
        .ComGroupSignal             = NULL,
        .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
        .ComSigGwRoutingReq         = FALSE,
        .Com_Arc_EOL                = 0
    },
    {
        .ComHandleId                = ComConf_ComSignal_RX_MCooltFlowMdul
            ,
        .ComIPduHandleId            =
            ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
        .ComFirstTimeoutFactor      = 0,
        .ComNotification            = COM_NO_FUNCTION_CALLOUT,
        .ComTimeoutFactor           = 0,
        .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
        .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
        .ComTransferProperty        = PENDING,
        .ComUpdateBitPosition       = 0,
        .ComSignalArcUseUpdateBit   = FALSE,
        .ComSignalInitValue         = &Com_SignalInitValue_MCooltFlowMdul
            ,
        .ComBitPosition             = 32,
        .ComBitSize                 = 10,
        .ComSignalEndianess         = COM_LITTLE_ENDIAN,
        .ComSignalType              = UINT16,
        .Com_Arc_IsSignalGroup      = 0,
        .ComGroupSignal             = NULL,
```

```
                .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
                .ComSigGwRoutingReq         = FALSE,
                .Com_Arc_EOL                = 0
            },
            {
                .ComHandleId                = ComConf_ComSignal_RX_TCooltMdulIn,
                .ComIPduHandleId            =
                    ComConf_ComIPdu_RX_PDU_BatHiLAuxData0,
                .ComFirstTimeoutFactor      = 0,
                .ComNotification            = COM_NO_FUNCTION_CALLOUT,
                .ComTimeoutFactor           = 0,
                .ComTimeoutNotification     = COM_NO_FUNCTION_CALLOUT,
                .ComErrorNotification       = COM_NO_FUNCTION_CALLOUT,
                .ComTransferProperty        = PENDING,
                .ComUpdateBitPosition       = 0,
                .ComSignalArcUseUpdateBit   = FALSE,
                .ComSignalInitValue         = &Com_SignalInitValue_TCooltMdulIn,
                .ComBitPosition             = 48,
                .ComBitSize                 = 8,
                .ComSignalEndianess         = COM_LITTLE_ENDIAN,
                .ComSignalType              = UINT8,
                .Com_Arc_IsSignalGroup      = 0,
                .ComGroupSignal             = NULL,
                .ComRxDataTimeoutAction     = COM_TIMEOUT_DATA_ACTION_NONE,
                .ComSigGwRoutingReq         = FALSE,
                .Com_Arc_EOL                = 0
            },
    {
        .Com_Arc_EOL                = 1
    }
};

/*
 * IPdu signal lists.
 */

SECTION_POSTBUILD_DATA const ComSignal_type * const
    ComIPduSignalRefs_RX_PDU_BatHiLAuxData0[] = {
    &ComSignal[ComConf_ComSignal_RX_TiBatOff],
    &ComSignal[ComConf_ComSignal_RX_BcuSt],
    &ComSignal[ComConf_ComSignal_RX_SrvCmd],
    &ComSignal[ComConf_ComSignal_RX_MdulBalnAcv],
    &ComSignal[ComConf_ComSignal_RX_MCooltFlowMdul],
    &ComSignal[ComConf_ComSignal_RX_TCooltMdulIn],
    NULL
};

/*
 * Gateway signal description handle
 */


/*
 * I-PDU definitions
 */
SECTION_POSTBUILD_DATA const ComIPdu_type ComIPdu[] = {
    { // RX_PDU
        .ArcIPduOutgoingId          = PDUR_REVERSE_PDU_ID_RX_PDU_BatHiLAuxData0,
        .ComRxIPduCallout           = COM_NO_FUNCTION_CALLOUT,
        .ComTxIPduCallout           = COM_NO_FUNCTION_CALLOUT,
        .ComIPduSignalProcessing    = DEFERRED,
        .ComIPduSize                = 64,
        .ComIPduDirection           = RECEIVE,
        .ComIPduGroupRefs           = ComIpduGroupRefs_RX_PDU,
        .ComTxIPdu = {
            .ComTxIPduMinimumDelayFactor    = 0,
            .ComTxIPduUnusedAreasDefault    = 0,
```

```
            .ComTxModeTrue = {
                .ComTxModeMode                      = NONE,
                .ComTxModeNumberOfRepetitions       = 0,
                .ComTxModeRepetitionPeriodFactor    = 0,
                .ComTxModeTimeOffsetFactor          = 0,
                .ComTxModeTimePeriodFactor          = 0,
            },
        },
        .ComIPduSignalRef           = ComIPduSignalRefs_RX_PDU_BatHiLAuxData0,
        .ComIPduDynSignalRef        = NULL,
        .ComIpduCounterRef          = NULL,
        .ComIPduGwMapSigDescHandle  = NULL,
        .ComIPduGwRoutingReq        = FALSE,
        .Com_Arc_EOL                = 0
    },
    {
        .Com_Arc_EOL                = 1
    }
};

SECTION_POSTBUILD_DATA const Com_ConfigType ComConfiguration = {
    .ComConfigurationId             = 1,
    .ComNofIPdus                    = 25,
    .ComNofSignals                  = 92,
    .ComNofGroupSignals             = 0,
    .ComIPdu                        = ComIPdu,
    .ComIPduGroup                   = ComIPduGroup,
    .ComSignal                      = ComSignal,
    .ComGroupSignal                 = ComGroupSignal,
    .ComGwMappingRef                = ComGwMapping,
    .ComGwSrcDesc                   = ComGwSourceDescs,
    .ComGwDestnDesc                 = ComGwDestinationDescs
};

#define BSW_STOP_SEC_CONST_UNSPECIFIED
#include "MemMap.h"
```

Verzeichnisse

Abkürzungsverzeichnis

ABS	Antiblockiersystem
ADC	Analog-Digital Konverter
API	Application Programming Interface
ASIL	Automotive Safety Integration Level
ASR	Antriebsschlupfregelung
ASW	Anwendungs-Software
AUTOSAR	Automotive Open Systems Architecture
BMS	Battery Management System
BSW	Basis-Software
CAN	Controller Area Network
COM	Communication
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access / Collision Detect
DEM	Diagnostic Event Manager
DLC	Data Length Code
DIO	Digital Input/Output
E2E	End-to-End
E/E	Elektrisch/Elektronisch
ECU	Electric Control Unit
EcuC	ECU Configuration
EIS	Electrochemical Impedance Spectroscopy
EOF	End of Frame
ERIKA	Embedded Real Time Kernel Architecture
ESP	Elektronische Stabilitätskontrolle
FIFO	First In – First Out
FTCOM	Fault Tolerant Communication
HAL	Hardware Abstraction Layer
HIS	Hersteller Initiative Software

HTH	Hardware Transmit Handle
HRH	Hardware Receive Handle
ICC	Implementation Conformance Classes
IDE	Integrated Development Environment
INCOBAT	INnovative COst efficient management system for next generation high voltage BATteries
ILLD	Infineon Low Level Driver
IOC	Inter-OS-Application Communication
ISO	Internationale Organisation für Normung
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
KFZ	Kraftfahrzeug
LIN	Local Interconnect Network
MCAL	MicroController Abstraction Layer
MISRA	Motor Industry Software Reliability Association
NM	Network Management
OIL	OSEK Implementation Language
ORTI	OSEK RealTime Interface
OS	Operating System
OSEK	Offene Systeme für die Elektronik im Kraftfahrzeug
PDU	Protocol Data Unit
PKW	Personenkraftwagen
PWM	Pulsweitenmodulation
RTE	Runtime Environment
RTOS	Real Time Operating System
SoC	State of Charge
SoF	State of Function
SOF	Start of Frame
SoH	State of Health
SPICE	Software Process Improvement and Capability Determination
SWC	Software Component
TI	technische Innovation
USB	Universal Serial Bus
VDX	Vehicle Distributed Executive
WCET	Worst Case Execution Time

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] MEXPERTS AG. ArcCore AB führt neue AUTOSAR Produkte ein. http://www.presseagentur.com/arccore/detail.php?pr_id=3566&lang=de, Oktober 2015.

[2] Altium. *TASKING VX-toolset for TriCore User Guide*, v4.3 edition, 07 2013.

[3] ArcCore. ArcCore. https://www.arccore.com/, Oktober 2015.

[4] ArcCore. Arctic Core - Can Cluster Configuration. https://www.arccore.com/documentation/release-9xx/UserDoc9.x.x/Can-Cluster-Configuration_10649606.html, Oktober 2015.

[5] ArcCore. Arctic Core - Documentation. https://www.arccore.com/documentation/release-9xx/UserDoc9.x.x/ArcCore-Documentation_49905847.html, Oktober 2015.

[6] ArcCore. Arctic Core Standard Package. https://www.arccore.com/products/arctic-core/standard-package, Oktober 2015.

[7] Armengaud, E. and Kurtulus, C. and Macher, G. and Groppo, R. and Novaro, M. and Hofer, G. and Schmidt, H. A Smart Computing Platform for Dependable Battery Management Systems. In Schulze, Tim and Müller, Beate and Meyer, Gereon, editor, *Advanced Microsystems for Automotive Applications 2015*, Lecture Notes in Mobility, pages 239–250. Springer International Publishing, 2016.

[8] AUTOSAR Development Cooperation. AUTOSAR AUTomotive Open System ARchitecture, 2015.

[9] AUTOSAR Development Cooperation. Specification of Communication, 2015.

[10] Bendiuga, Volodymyr. Multi-Core Pattern. 2012.

[11] Bertrand, D. and Faucou, S. and Trinquet, Y. An analysis of the AUTOSAR OS timing protection mechanism. In *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–8, Sept 2009.

[12] Claraz, D and Grimal, F and Laydier, T and Mader, R and Wirrer, G. Introducing Multi-Core at Automotive Engine Systems. In *ERTS 14*. ERTSS, 2014.

[13] Davis, Robert I. and Burns, Alan. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[14] Verband der Automobilindustrie e. V. (VDA). Automotive SPICE. http://vda-qmc.de/en/software-processes/automotive-spice/, Oktober 2015.

[15] Devika, K. and Syama, R. An Overview of AUTOSAR Multicore Operating System

Implementation. In *International Journal of Innovative Research in Science, Engineering and Technology*, number Issue 7, 2013.

[16] Eclipse Foundation. About the Eclipse Foundation, Oktober 2015.

[17] Esmaeilzadeh, H. and Blem, E. and St.Amant, R. and Sankaralingam, K. and Burger, D. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, June 2011.

[18] Evidence. ERIKA Enterprise - Features and Benefits. http://erika.tuxfamily.org/drupal/features-and-benefits.html, Oktober 2015.

[19] Evidence Srl. *ERIKA Enterprise Manual*, Oktober 2015.

[20] Farsi, M and Ratcliff, K and Barbosa, Manuel. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.

[21] Fisher, N. How Hard is Partitioning for the Sporadic Task Model? In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 2–5, Sept 2009.

[22] Eclipse Foundation. Eclipse MarketPlace. https://marketplace.eclipse.org/, Oktober 2015.

[23] Gepner, P. and Kowalik, M.F. Multi-Core Processors: New Way to Achieve High System Performance. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pages 9–13, Sept 2006.

[24] Hanxing Chen and Jun Tian. Research on the Controller Area Network. In *Networking and Digital Society, 2009. ICNDS '09. International Conference on*, volume 2, pages 251–254, May 2009.

[25] Hladik, P.-E. and Deplanche, A. and Faucou, S. and Trinquet, Y. Adequacy between AUTOSAR OS specification and real-time scheduling theory. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 225–233, July 2007.

[26] Homann, M. *OSEK: Betriebssystem-Standard für Automotive und Embedded Systems*. mitp-Verlag, 2005. [Konfiguration, Kommunikation, Netzwerkmanagement, Design Patterns, detaillierte Dokumenatation aller API-Funktionen, mit zeitlich unbegrenzter Demoversion (Windows) auf CD].

[27] Honekamp, U. The Autosar XML Schema and Its Relevance for Autosar Tools. *Software, IEEE*, 26(4):73–76, July 2009.

[28] Abeer Hyari. A comparative study on heterogeneous and homogeneous multiprocessors. 2009.

[29] INCOBAT. INCOBAT Project. http://www.incobat-project.eu/, Oktober 2015.

[30] Infineon Technologies AG. *Application Kit TC2X5 User's Manual*. 81726 Munich, Germany, 2015.

[31] ISO - International Organization for Standardization. ISO 26262 Road vehicles Functional Safety Part 1-10, 2011.

[32] Jongtaek Han and Jin Seo Park and Michael Deubzer and Jens Harnisch and Patrick Leteinturier. Efficient Multi-Core Software Design Space Exploration for Hybrid Control Unit Integration. In *SAE Technical Paper 2014-01-0260*. SAE International, 04 2014.

[33] Krause, Matthias and Weich, Carsten. Intrinsic safety of AUTOSAR Basic Software - Integrating software components with different safety relevance in one ECU. *VECTOR Technical Article*, pages 1–4, 2012.

[34] Krishnan, S. and Garimella, Suresh V. and Chrysler, G.M. and Mahajan, R.V. Towards a Thermal Moore's Law. *Advanced Packaging, IEEE Transactions on*, 30(3):462–474, Aug 2007.

[35] Lauterbach. Lauterbach. http://www.lauterbach.com/, Oktober 2015.

[36] Macher, Georg and Atas, Muesluem and Armengaud, Eric and Kreiner, Christian. Automotive Real-time Operating Systems: A Model-based Configuration Approach. *SIGBED Rev.*, 11(4):67–72, January 2015.

[37] Mack, C. The Multiple Lives of Moore's Law. *Spectrum, IEEE*, 52(4):31–31, April 2015.

[38] Mack, C.A. Keynote: Moore's Law 3.0. In *Microelectronics and Electron Devices (WMED), 2013 IEEE Workshop on*, pages xiii–xiii, April 2013.

[39] Marc Graniou and Hakan Sivencrona and Rickard Svenningsson. Advantages and Challenges of Introducing AUTOSAR for Safety-Related Systems. In *SAE Technical Paper 2009-01-0750*. SAE International, 04 2009.

[40] MISRA. MISRA Home. http://www.misra.org.uk/, Oktober 2015.

[41] MISRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, October 2004.

[42] Moore, G.E. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[43] Moyer, Bryon. *Real World Multicore Embedded Systems*. Newnes, Newton, MA, USA, 1st edition, 2013.

[44] National Instuments. Controller Area Network (CAN) – eine Übersicht. http://www.ni.com/white-paper/2732/de/, Oktober 2015.

[45] OSEK/VDX Steering Committee. OSEK/VDX Communication. http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf, 2004.

[46] OSEK/VDX Steering Committee. OSEK/VDX Network Management. http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf, 2004.

[47] OSEK/VDX Steering Committee. OSEK/VDX System Generation OIL: OSEK Implementation Language. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, 2004.

[48] OSEK/VDX Steering Committee. OSEK/VDX Operating Systems. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, February 2005.

[49] Park, S. and Park, Y.B. A Multi-Core Architectural Pattern Selection Method for the Transition from Single-Core to Multi-Core Architecture. In *IT Convergence and Security (ICITCS), 2014 International Conference on*, pages 1–5, Oct 2014.

[50] Parkhurst, Jeff and Darringer, John and Grundmann, Bill. From Single Core to Multi-core: Preparing for a New Exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pages 67–72, New York, NY, USA, 2006. ACM.

[51] Patrick Leteinturier and Simon Brewerton and Klaus Scheibert. MultiCore Benefits & Challenges for Automotive Applications. In *SAE Technical Paper 2014-01-0260*. SAE International, 04 2008.

[52] Rafael Zalman and Alexander Griessing and Paul Emberson. Timing Correctness in Safety-Related Automotive Software. In *SAE Technical Paper 2011-01-0449*. SAE International, 04 2011.

[53] Rees Jürgen, Wettach Silke. Rollender Datensammler. http://www.zeit.de/mobilitaet/2014-03/auto-ueberwachung-autoindustrie-vernetzung, 2014.

[54] Salloum, C.E. and Elshuber, M. and Hoftberger, O. and Isakovic, H. and Wasicek, A. The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems. In *2012 15th Euromicro Conference on Digital System Design (DSD)*, pages 105–113, 09 2012.

[55] Schelling, Helmut. AUTOSAR - Equipped for Everything. *VECTOR Technical Article*, pages 1–4, March 2014.

[56] Schoof, Jochen. OSEK/VDX-OS — Betriebssystemstandard für Steuergeräte in Kraftfahrzeugen. In Holleczek, Peter, editor, *PEARL 2000*, Informatik aktuell, pages 43–52. Springer Berlin Heidelberg, 2000.

[57] Simon Brewerton. Höchste Sicherheit mit Echtzeit-Performance. http://www.elektronikpraxis.vogel.de/automotive/articles/375822/, Oktober 2015.

[58] Vector. CANalyzer. http://vector.com/vi_canalyzer_de.html, Oktober 2015.

[59] Zimmermann, Werner and Schmidgall, Ralf. AUTOSAR-Softwarearchitektur für Kfz-Systeme. In *Bussysteme in der Fahrzeugtechnik*, ATZ/MTZ-Fachbuch, pages 367–413. Springer Fachmedien Wiesbaden, 2014.

[60] Zimmermann, Werner and Schmidgall, Ralf. *Bussysteme in der Fahrzeugtechnik*. Springer Fachmedien Wiesbaden, Wiesbaden, 2014.

[61] Zimmermann, Werner and Schmidgall, Ralf. Software-Standards: OSEK und HIS. In *Bussysteme in der Fahrzeugtechnik*, ATZ/MTZ-Fachbuch, pages 331–365. Springer Fachmedien Wiesbaden, 2014.