FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Power-Management in Automotiven Systemen – Integration und Umsetzung am Beispiel der PLASA-Plattform

Robert Urs Dörfel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Bernd Brügge, Ph. D.
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. Uwe Baumgarten
	2. Univ.-Prof. Dr. Johann Schlichter

Die Dissertation wurde am 17.12.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.04.2016 angenommen.

Kurzfassung

Die heutigen Automotiven Systeme benötigen immer mehr Energie, um die immer steigende Anzahl von Fahrzeug-Funktionen zu erfüllen. Im Gegensatz dazu stehen die EU-Regulierung des CO_2-Ausstoßes einer Fahrzeug-Flotte und die steigenden Benzin-Preise, was zu höheren Kosten sowohl für die Automobil-Industrie als auch für den Automobil-Nutzer führt. Aus diesen Gründen sollten die Automotiven Systeme energieeffizienter gestaltet werden, um damit den CO_2-Ausstoß und den Benzin-Verbrauch zu senken, indem nur die Verbraucher aktiv sind, die für die Erfüllung der momentan aktiven Fahrzeug-Funktionen benötigt werden.

Durch den aktuellen Trend immer mehr Fahrzeug-Funktionen sowohl im Sicherheitsbereich als auch im Komfort-Bereich ins Automobil zu integrieren, verstärkt sich noch das Bedürfnis nach einer energieeffizienter Auslegung der Automotiven Systeme. Denn mit der Zunahme an Fahrzeug-Funktionen steigt auch die Anzahl der Energieverbraucher. Da viele der Komfort-Funktionen individuell von den Fahrzeug-Nutzern ein- bzw. ausgeschaltet werden, müssen für eine energieoptimierte Auslegung des Automotiven Systems die nicht benötigten Verbraucher deaktiviert werden, um somit Energie einzusparen. Jedoch sind die bisherigen Automotiven Systeme weder nach energieeffizienten Gesichtspunkte ausgelegt noch dafür entwickelt worden.

An dieser Stelle setzt diese Arbeit an und untersucht, wie ein energieeffizientes Power-Management in die System- und Software-Architektur von Automotiven Systemen integriert werden kann, um ein für die Fahrzeug-Funktionen möglichst transparentes Power-Management zu realisieren. Es wurde eine spezielle Architektur für ein automotives Power-Management entwickelt und in die PLASA-Plattform integriert, die aus Teilen aus dem Lehrbetrieb entstanden ist, und anschließend an definierten Anwendungsfällen getestet. Der Stromverbrauch der Plattform wurde mit Hilfe eines für die PLASA-Plattform speziell entwickelten Mess-System gemessen, um das korrekte Verhalten des Power-Managements nachzuweisen.

Danksagung

An erster Stelle möchte ich mich bei Prof. Dr. Uwe Baumgarten für die Betreuung dieser Arbeit bedanken und, dass er es mir ermöglichte, mit dem interessanten und spannendem Thema über das Power-Management in Automotiven Systemen zu promovieren. Er hat mir stets die notwendigen Freiräume gelassen, um das Thema in all seinen Facetten zu beleuchten und die bearbeiteten Schwerpunkte in dieser Arbeit selbst zu setzen. An zweiter Stelle möchte ich meinen Dank Prof. Dr. Johann Schlichter aussprechen, dass er das Zweitgutachten dieser Dissertation übernommen hat und mit seinen wertvollen Kommentaren zu dieser Arbeit beigetragen hat.

Des Weiteren möchte ich mich bei Herrn Michael Haunreiter bedanken, der das von seiner Firma Miray Software AG entwickelte Betriebssystem Symobi für den Lehrbetrieb bereitgestellt hat. Dies wurde zum einen für die Steuerung der Modell-Fahrzeuge in dem angebotenen Betriebssystem-Praktikum verwendet. Zum anderen sind viele studentische Arbeiten im Umfeld von Symobi entstanden, durch die ich meine Kenntnisse über Betriebssysteme und deren Funktionsweisen vertiefen konnte.

Außerdem gilt mein Dank den vielen Studenten, die mit ihren Abschluss-Arbeiten oder mit ihren Implementierungen in den Praktika das PLASA-Projekt unterstützt und die Plattform erweitert haben. Stellvertretend für die vielen Studenten möchte ich Herrn Fabian Otto namentlich nennen, der neben zwei studentischen Arbeiten auch als studentische Hilfskraft einen großen Anteil an der Realisierung des Projekts hatte.

Ebenfalls möchte ich mich bei meinen vielen Kollegen am Fachgebiet für Betriebssysteme bedanken. Die geführten Diskussionen waren für die Erarbeitung der Themen in dieser Dissertation stets förderlich und hilfreich. Stellvertretend möchte ich Herrn Dr. Wolfgang Haberl nennen, mit dem in vielen Diskussionen u. a. die Idee entstanden ist, Modell-Fahrzeuge in dem bereits angebotenen Betriebssystem-Praktikum zu verwenden.

Ebenso möchte ich mich bei meinen Eltern Marianne und Peter Dörfel bedanken, die mich immer in meinen Entscheidungen unterstützt und mir das Informatik-Studium und später die Promotion ermöglicht haben. Mein herzlichster Dank geht an meine Frau Dr. Katrin Dörfel, die mich schon während meines Studiums bis hin zur Fertigstellung dieser Arbeit stets liebevoll unterstützt hat und mich immer ermutigt hat, diese Arbeit zu beenden. In der letzten Phase hat sie durch das Korrekturlesen dieser Arbeit dazu beigetragen, dass ich die Dissertation in dieser Form fertigstellen konnte. Vielen herzlichen Dank dafür.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den letzten Jahrzehnten ist die Durchschnittstemperatur auf der Erde um 0,74 °C gestiegen. Die Ursache für diese Temperaturerhöhung ist der erhöhte Ausstoß von Treibhausgasen wie Kohlendioxid (CO_2), Methan (CH_4) oder Distickstoffmonoxid (N_2O), die in den oberen Atmosphären-Schichten der Erde den sogenannten Treibhaus-Effekt verursachen. Die kurzwellige Sonnenstrahlung kann die angesammelten Gase ungehindert passieren. Jedoch wird die von der Erde emittierte langwellige Strahlung durch die Treibhausgase wieder auf die Erde reflektiert. Dadurch kommt es zu einer globalen Erwärmung auf der Erde, was auch als Klimawandel bezeichnet wird.

Durch die Temperaturerhöhung kommt das Ökosystem auf der Erde aus dem Gleichgewicht, was u. a. ein Aussterben des Tierartenreichtums zu Folge hat. Des Weiteren schmelzen die Gletscher und das arktische Eis, was zu einem Ansteigen des Meeresspiegels führt. Durch den steigenden Meeresspiegel werden die niedrigen Landesteile überflutet und ganze Inseln und Küstenlandstriche gehen mit ihrem Tierartenreichtum verloren. Auch sind weitere Folgen durch eine Temperaturerhöhung nicht abzuschätzen [140].

Das Ziel der internationalen Klima-Politik ist es, die globale Erwärmung auf weniger als 2 °C gegenüber dem Niveau vor der Industrialisierung zu begrenzen. Als eine Möglichkeit, um den Anstieg der Temperatur aufzuhalten oder wenigstens zu verlangsamen, wird die Reduzierung des Ausstoßes von Treibhausgasen gesehen. CO_2, eines der Treibhausgase, entsteht u. a. bei der Verbrennung in Industrieanlagen. Ein anderer Anteil des CO_2-Ausstoßes entsteht durch die Mobilität der Erdbevölkerung, die ebenfalls in den letzten Jahrzehnten durch die Globalisierung zugenommen hat. Sowohl die Turbinen von Flugzeugen als auch die Verbrennungsmotoren von Kraftfahrzeugen tragen zum CO_2-Ausstoß bei.

Um den CO_2-Ausstoß im Fahrzeug zu verringern, hat die Europäische Union Grenzwerte für den CO_2-Ausstoß festgelegt. Damit darf der CO_2-Ausstoß der Fahrzeug-Flotte eines Kraftfahrzeug-Hersteller im Durchschnitt einen festgelegten Grenzbetrag nicht überschreiten. Falls jedoch diese Grenzen nicht eingehalten werden, sind über die Jahre steigende Strafzahlungen für die Kraftfahrzeug-Hersteller fällig [154].

Diese Regulierungen des CO_2-Ausstoßes durch die Europäische Union führen u. a. zu

den Bestrebungen in der Industrie, die Mobilität umweltfreundlicher und energieeffizienter zu gestalten. Ein Beispiel dafür ist die Nationale Roadmap Embedded Systems [1], die der Zentralverband Elektrotechnik- und Elektronikindustrie e. V. (ZVEI) 2009 veröffentlicht hat. In dieser wird als eine Herausforderung die nachhaltige Mobilität genannt, die es gilt, in den nächsten Jahren anzugehen.

Dies führt unweigerlich zu der Frage, wie die Kraftfahrzeuge mit ihren Automotiven Systemen energieeffizienter gestaltet werden können. Jedoch sollte, bevor diese Frage beantwortet werden kann, geklärt werden, was Automotive Systeme sind, wie sie sich in die Gruppe der Embedded Systeme einordnen lassen und welche Eigenschaften diese besitzen.

1.1 Einordnung der Automotiven Systeme

Um die Frage zu klären, wie sich Automotive Systeme in der Welt der Embedded Systeme einordnen lassen, müssen zuerst die Embedded Systeme und deren Eigenschaften betrachtet werden. Dabei ist der Begriff der Embedded Systeme weit gefasst und beinhaltet eine große Menge an unterschiedlichen Systemen. Grundsätzlich werden Embedded Systeme als Systeme definiert, die in einem technischen Prozess eingebettet sind und dabei Mess-, Kontroll- und Steueraufgaben übernehmen.

Im Unterschied zu den General Purpose-Rechensystemen wie der Desktop-Rechner oder ein Server-System, das unterschiedliche Dienste bereitstellt, ist ein Embedded System dafür ausgelegt, nur bestimmte Funktionen auszuführen. Meist ist auf den ersten Blick nicht erkennbar, dass sich ein Rechen-System hinter einem Geräte verbirgt. Als Beispiele lassen sich Unterhaltungselektronik wie Fernseher oder Musikanlagen, aber auch Haushaltsgeräte wie Waschmaschinen oder Backöfen nennen. In diesen Geräten sind Mikrocontroller verbaut, die einen physikalischen Prozess überwachen und mit Hilfe einer Regelschleife in den physikalischen Prozess eingreifen, um die gewünschte Funktion des Geräts zu erfüllen.

Ein Beispiel aus der Domäne der Embedded Systeme ist das WSN (engl. *wireless sensor network*), in dem hauptsächlich die Überwachung der Umwelt im Vordergrund steht. Hier werden mit Hilfe von Sensoren bestimmte physikalische Größen der Umwelt wie z. B. Temperatur, Luftdruck oder Luftfeuchtigkeit erfasst und drahtlos an eine zentrale Datenbank gesendet. Das Besondere an dem WSN ist, dass die Sensoren in großer Anzahl ausgebracht werden und im Anschluss über einen längeren Zeitraum nicht mehr von Technikern vor Ort betreut werden. A. Mainwaring *et al.* beschreiben die Anforderungen und die System-Architektur der WSNs anhand eines konkreten Forschungsprojekts zur Überwachung des Lebensraums von Vögeln [106]. Zu den Anforderungen der WSNs zählen u. a. eine lange Lebensdauer der Sensor-Knoten, der Zugang zu den Sensor-Knoten über das Internet und die Administration der Sensor-Knoten aus der Ferne. Daraus ergibt sich eine geschichtete System-Architektur, bei der die Sensor-Knoten ein drahtloses Netzwerk bilden und über ein Gateway mit einer Basis-Station kommunizieren. Diese Basis-Station ist anschließend mit dem Internet verbunden, über das die gemessenen und gesammelten Sensor-Daten an eine Datenbank gesendet werden. Die Domäne der WSNs zeigt einen Ausschnitt der Embedded Systeme, wobei hier der

Fokus rein in der Erfassung der Umwelt liegt.

Durch die unterschiedlichen Aufgaben, die die Embedded Systeme erbringen, ist es schwierig, eine klare Abgrenzung bezüglich Hard- und Software für die Embedded Systeme zu treffen. Dennoch ergeben sich Gemeinsamkeiten in der Hard- und Software, die S. Heath in seinem Buch „Embedded Systems Design“ beschreibt [69]. Die Software in Embedded Systemen ist meist proprietär für das konkrete System entwickelt worden. Jedoch wachsen die Embedded Systeme immer stärker und die Funktionen werden immer mehr von Software realisiert. Dadurch ergeben sich weitere Herausforderungen an die Software für Embedded Systeme, die E. Lee in seiner Veröffentlichung nennt [94]. Durch das Anwachsen der Software-Systeme in den Embedded Systemen ist die Notwendigkeit gestiegen, diese umfassend zu spezifizieren. Dabei hebt P. Marwedel in seinem Buch „Embedded System Design“ verschiedene Möglichkeiten für die Spezifikation von Embedded Systemen hervor [107].

In der jüngsten Vergangenheit kam der Begriff der Cyber-Physical Systeme (CPS) auf, der die Verschmelzung von Computer-Systemen mit deren Netzwerk mit einem physikalischen Prozess in den Mittelpunkt rückt. Dabei überwacht das Computer-System mit Hilfe von Sensoren den physikalischen Prozess und berechnet daraus, in wie weit er mit Hilfe sogenannter Aktoren Einfluss auf den physikalischen Prozess nehmen muss. Somit entsteht ein Regelkreis, in dem die Computer-gestützte Berechnung Einfluss auf den physikalischen Prozess nimmt und der physikalische Prozess die Berechnung beeinflusst.

Die zuvor genannten WSNs können hierbei in den CPS die Überwachung der physikalischen Prozesse übernehmen. Jedoch steuern diese nicht den physikalischen Prozess. Für ein vollständiges CPS muss im Anschluss noch ein separates System vorhanden sein, das mit Aktoren Einfluss auf die Umwelt nimmt.

R. Rajkumar *et al.* beschreiben die Vision der zukünftigen CPSs im Alltag und welche Herausforderungen sich daraus ergeben [126]. So sehen sie u. a. folgende Visionen für die zukünftigen CPSs, wobei hier Beispiele aus dem alltäglichen Leben genannt werden:

Unterbrechungsfreie Stromversorgung. Eine Herausforderung der CPS ist die globale unterbrechungsfreie Stromversorgung der Bevölkerung, die es ermöglichen soll, auf verschieden Gegebenheiten zu reagieren. Als Beispiel wird die Energiegewinnung aus der Windkraft genannt, die nur an windigen Tagen zur Verfügung steht. An Tagen ohne Wind muss ebenfalls für eine Stromversorgung aus anderen Energiequellen gesorgt werden. Damit eine globale unterbrechungsfreie Stromversorgung garantiert werden kann, muss das CPS auf alle Situationen selbstständig und intelligent auf die jeweilige Situation reagieren.

Ständige Assistenz im Alltag. Für ältere oder behinderte Menschen sollen die CPS Unterstützung bereitstellen, die deren Alltag durch eine ständige Assistenz erleichtern. So können Geräte diese bei Alltagstätigkeiten wie z. B. beim Einkaufen, Kochen oder bei Reinigungsarbeiten unterstützen. Ebenfalls können Geräte mit Bildschirmen Erinnerungshilfen für demente oder vergessliche Menschen für bestimmte Tätigkeiten darstellen.

Verzögerungs- und Unfall-freie Mobilität. Ein Aspekt dieser Vision ist auch die verzögerungsfreie Mobilität. Dies beinhaltet, dass es z. B. im Berufsverkehr zu keinem

Stau kommt, indem die CPSs dafür sorgen, dass die Fahrzeuge intelligent auch mit Hilfe von Infrastruktur-Komponenten mit dieser Situation umgehen und eigenständig Routen finden, um Staus zu vermeiden. Der zweite Aspekt der Vision ist die Vermeidung von Fahrzeug-Unfällen. So kommt es in der Vision zu fast keinen Unfällen mehr oder falls ein Unfall passiert, so bleibt es nur bei geringen Verletzungen und Schäden.

Diese Aufzählung der Visionen zeigt u. a., in welchen Bereichen der Embedded Systeme die CPSs eingesetzt werden. Außerdem wird ersichtlich, welchen Platz die Automotiven Systeme im Bereich der CPSs einnehmen. Die Vision der unfallfreien Mobilität führt vor Augen, welche Herausforderungen auf die Automotiven Systeme zukommen und welchen Einfluss dies auf die zukünftige Entwicklung der Automotiven Systeme hat, auf die in Abschnitt 2.5 eingegangen wird.

E. Lee nennt dabei als Herausforderungen im Design der CPSs die Zuverlässigkeit der Systeme [93]. Dabei hebt er hervor, dass vor allem die rechtzeitige Lieferung der Berechnungsergebnisse Schwierigkeiten bereitet. Diese Problematik ist auch als Einhaltung von Deadlines bekannt, was im Bereich der Echtzeit-Systeme (engl. *real-time system*) ebenfalls untersucht wird. Bereits Ende der 1980er erkannte J. Stankovic die falschen Auffassungen des Real-time Computings und nannte die größten Herausforderungen, an denen seitdem immer inkrementell und ausschnittsweise gearbeitet wurde [143].

In ihrem Buch „High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing“ beschreibt M. Wolf die Domäne der Embedded Systeme mit ihren angepassten Programmen, deren Betriebssysteme und Prozessoren [166]. Hierbei legt sie den Fokus auf Multiprozessor-Architekturen und die für die Embedded Systeme angepasste Multiprozessor-Software. Das Buch schließt sie mit einem Kapitel über die CPS ab.

H. Kopetz beschreibt die Grundlagen der embedded Echtzeit-Systeme in seinem Buch „Real-Time Systems: Design Principles for Distributed Embedded Applications“ [90]. H. Kopetz untersuchte dabei in einem verteilten System die Echtzeit-Anforderungen, womit eine weitere Domäne genannt wird: die verteilten Systeme (engl. *distributed systems*). A. Tanenbaum und M. van Steen nennen in ihrem Buch „Distributed Systems: Principles and Paradigms“ die Grundlagen der Verteilten Systeme unabhängig von Echtzeit-Anforderungen [150].

Der Begriff der verteilten Systeme beschreibt Computer-Systeme, die über ein Netzwerk verbunden sind und eine Funktion über das Netzwerk erbringen. Darunter fallen auch Server, die bestimmte Dienste (engl. *service*) im Netzwerk bereitstellen. Für diese Systeme wurden unterschiedliche Konzepte der Kommunikation entworfen, die unter den Begriff der verteilten Systeme zusammengefasst wurden.

Die Automotiven Systeme liegen in der Schnittmenge aus den CPS, den Echtzeit-Systemen und den verteilten Systemen. Diese Einordnung der Automotiven Systeme ist in Abbildung 1.1 noch einmal graphisch dargestellt. Bereits in der Vision für die CPS wurde dargelegt, dass die Automotiven Systeme eine Teilmenge der CPSs sind.

Automotive Systeme sind stark verteilte Systeme, die historisch gewachsen sind. Denn heute werden bis zu 80 ECUs (engl. *electronic control unit*) in einem voll ausgestatteten Fahrzeug mit allen Fahrassistenz- und Komfortfunktionen eingesetzt. Diese Funktionen

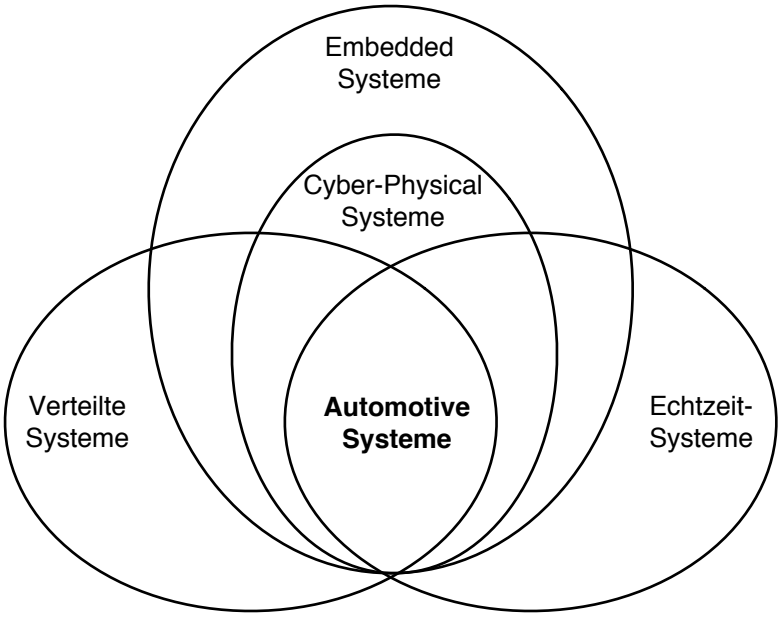


Abbildung 1.1: Einordnung der Embedded Systeme, der Cyber-Physical Systeme und der Automotiven Systeme.

werden dabei verteilt von verschiedenen ECUs erbracht, die über mehrere Bus-Systeme und einem zentralen Gateway miteinander kommunizieren. Die aktuelle Architektur der Automotiven Systeme wird im Abschnitt 2.2.1 im Detail vorgestellt.

Zusätzlich sind die ECUs sehr heterogen und stark an ihre Funktion, die sie erbringen müssen, angepasst. Dies führt dazu, dass verschiedenste Mikrocontroller, Speicher- und Kommunikationskomponenten in einer ECU verbaut werden. Daraus ergibt sich, dass Automotive Systeme keine Homogenität in ihrer Hardware aufweisen, was eine weitere Herausforderung darstellt. Auf die heterogene Hardware der Automotiven Systeme wird im Detail in Abschnitt 2.3 eingegangen.

Eine weitere Folge der heterogenen Hardware ist, dass die Software ebenfalls heterogen ist und jeweils an die zu erbringende Funktion der ECU angepasst ist. Dennoch haben sich im Laufe der Zeit Industrie-Standards für automotive Software entwickelt, die in Abschnitt 2.4 näher vorgestellt werden. In den automotive Software-Standards ist der Aspekt der Verteilung sehr stark verankert und somit ist auch die Kommunikationsfähigkeit zwischen den ECUs ein elementarer Bestandteil.

Automotive Systeme sind ebenfalls harte Echtzeit-Systeme, bei denen das Nicht-Einhalten einer Deadline zu einem hohen Schaden führt. Bei den Fahr-, Lenk- und Bremsfunktionen kann das Reißen einer Deadline im schlimmsten Fall zu einem tragischen Unfall mit Verletzten oder sogar Toten führen. Aus diesem Grund ist die Echtzeit-Fähigkeit der Automotiven Systeme ein wichtiger Aspekt.

Andererseits sind Teile der Automotiven Systeme vor allem im Bereich der Komfortfunktionen nicht hart Echtzeit-kritisch und haben weiche Echtzeit-Anforderungen. Als Beispiele lassen sich das Entertainment-System, in dem das Radio oder die Navigation

enthalten ist, oder die Sitzheizung nennen, bei denen ein nicht Einhalten der Deadline zu einer Verärgerung des Fahrers führt. Aber es werden dabei keine sicherheitsrelevanten Funktionen des Fahrzeugs beeinträchtigt. Aus diesem Grund sind die Automotiven Systeme in einen sicherheitsrelevanten, Echtzeit-kritischen Bereich und in einen nicht sicherheitsrelevanten Teil, bei dem die Echtzeit-Anforderungen weicher sind, getrennt. Diese Trennung ist in der Architektur heutiger Automotiver Systeme fest verankert und wird in Abschnitt 2.2.1 näher beschrieben.

1.2 Power-Management in Automotiven Systemen

Der Beginn dieses Kapitels wurde mit der Frage eingeleitet, wie Automotive Systeme energieeffizienter gestaltet werden können, um ihren Beitrag an der Reduzierung des CO_2-Ausstoß zu leisten. Daraus ergibt sich die Frage, wie die einzelnen Komponenten des Automotiven Systems energieeffizienter werden können, damit das gesamte Fahrzeug energieeffizienter wird.

Automotive Systeme sind ein stark verteiltes System mit ca. 80 ECUs, deren Anzahl jeweils abhängig vom Fahrzeug und deren Ausstattung variieren kann. Neben den ECUs sind zusätzlich Sensoren und Aktoren wie der Antrieb, die Lenkung, Klima-Anlage, die elektrischen Sitz- und Heckscheibenheizungen verbaut. Im Weiteren wird nicht der Sachverhalt betrachtet, wie die einzelnen Hardware-Komponenten der Automotiven Systeme wie die Sensoren oder Aktoren energieeffizienter gebaut werden können. Sondern der Fokus dieser Arbeit liegt auf dem Power-Management für Automotive Systeme, das die Hardware-Komponenten zu den Zeitpunkten, an denen sie nicht mit ihrer vollen Leistung benötigt werden, in den niedrigsten Power-Modus schaltet, der gerade dazu ausreicht, alle zu diesem Zeitpunkt benötigten Fahrzeug-Funktionen auszuführen. Dies beinhaltet auch das komplette Ausschalten einer ECU, wenn deren zu erbringenden Funktionen nicht benötigt werden.

Dieses Konzept, das sich zunächst sehr einfach anhört und bereits in jedem modernen Embedded System wie z. B. einem Smartphone integriert ist, gestaltet sich in einem typischen Automotiven System durch die große Anzahl an Fahrzeug-Funktionen und deren starke Verteilung auf ECUs gepaart mit den dort herrschenden Echtzeit-Anforderungen als schwierig.

Eine offene Frage ist, ob sich Aktoren in einem Automotiven System überhaupt in einem niedrigeren Power-Modus betreiben lassen. Dies ist an dieser Stelle oft nicht möglich, da Aktoren nur Energie verbrauchen, wenn sie aktiv sind. Dies bedeutet, dass sie nur Energie benötigen, wenn sie auch Einfluss auf den physikalischen Prozess nehmen. Dieser Einfluss ist meist durch den Fahrer des Fahrzeugs gewollt und kann somit nicht reduziert werden. Als Beispiel lassen sich der Antrieb oder die Lenkung nehmen. Wenn der Fahrer beschleunigen oder lenken möchte, dann müssen die Systeme auch diesem Wunsch entsprechen und die Energie aufwenden, um zu beschleunigen oder den Lenk-Einschlag zu unterstützen.

Somit bleiben für die Steuerung durch ein automotives Power-Management nur die Sensoren und die Hardware-Komponenten übrig, die auf einer ECU verbaut sind und ebenfalls Strom verbrauchen. Der Verbrauch der Hardware-Komponenten wie der des

Mikrocontrollers, des Speichers oder des Kommunikationscontrollers wird auch als Logik-Anteil bezeichnet. Nach einer Analyse der heutigen Automotiven Systeme laufen die auf einer ECU verbauten Mikro- und Kommunikationscontroller meist unter Volllast, obwohl eventuell nur ein Bruchteil ihrer Rechen- oder Kommunikationsleistung zu einem gewissen Zeitpunkt benötigt wird. Ziel wäre es, die Power-Modi, die bereits automotive Mikrocontroller integriert haben, zu nutzen. Damit wird der Stromverbrauch und somit über die Zeit der Energieverbrauch des Logik-Anteils reduziert.

Der Grund dafür, dass die Power-Modi der Hardware-Komponenten nicht genutzt wurden, ist, dass bei der Entwicklung der ECUs bis zur Einführung der Grenzwerte für den CO_2-Ausstoßes einer Fahrzeug-Flotte und der Einführung von Strafzahlungen für die Kraftfahrzeug-Hersteller ausschließlich die Funktionserfüllung im Vordergrund stand und die Energieeffizienz der ECUs keine Anforderung der Automobil-Hersteller war. Denn nach einer heutigen Analyse beträgt der Energieverbrauch des Logik-Anteils nur ca. 10 % des gesamten Energieverbrauchs im Fahrzeugs. Der Entwicklungsaufwand für die Nutzung der Power-Modi stand in keiner Relation zu den möglichen Energieeinsparungen durch ein optimales Power-Management bei gleichbleibender Erfüllung der Fahrzeug-Funktionen. Doch die EU-Auflagen bezüglich des CO_2-Ausstoßes veranlassten die Automobil-Hersteller hier ebenfalls umzudenken und die Energie-Einsparmöglichkeiten im Bereich des Logik-Anteils einer ECU nutzen zu wollen.

Zusätzlich trägt der Anstieg der Fahrzeug-Funktionen in den zukünftigen Fahrzeug-Generationen dazu bei, den Energieverbrauch im Logik-Anteil nicht zu vernachlässigen. Denn die neuen Fahr- und Komfortfunktionen im Fahrzeug werden meist durch Software, die teilweise auch rechenintensiv sein kann, realisiert. Dies führt dazu, dass zusätzliche oder leistungsstärkere Mikrocontroller im Fahrzeug verbaut werden, was einen Anstieg des Energieverbrauchs durch den Logik-Anteil zur Folge hat. Aus diesem Grund lohnt es sich, neben den Effizienz-Steigerungen der mechanischen Fahrzeug-Komponenten, der elektromagnetischen Fahrwerkskomponenten oder der Klima-Anlagen auch die Energieeffizienz des Logik-Anteils im Fahrzeug zu steigern, um den Energieverbrauch im Fahrzeug zu optimieren.

Bei einer genaueren Betrachtung verbraucht nur die Hardware Energie, auf der die Software läuft und ihre Funktionalität erbringt. Um Energie einzusparen, bietet die Hardware unterschiedliche Power-Modi an, in denen die Hardware ihre Rechenleistung vermindert und einzelne Hardware-Komponenten in den Ruhemodus fahren oder sogar ganz abschalten kann. Dadurch verringert sich der Stromverbrauch der Hardware, was über die Laufzeit eine Energieeinsparung bedeutet. Das automotive Power-Management steuert dabei, welche von der Hardware angebotene Power-Modi in dem entsprechenden Fahrzeug-Zustand eingenommen werden. Hierbei ist für das automotive Power-Management wichtig, dass alle Funktionen, die in dem gerade aktiven Fahrzeug-Zustand notwendig sind, bei den niedrigsten Power-Modi erbracht werden.

Für ein möglichst gutes Power-Management ist die Granularität der Power-Modi, die die Hardware anbietet, entscheidend. Je mehr Power-Modi die Hardware anbietet, desto mehr verschiedene Möglichkeiten stehen dem Power-Management zur Verfügung, um den Energieverbrauch der Hardware optimal auslegen zu können. Auf die automotive Hardware wird in Abschnitt 2.3 näher eingegangen. Der Fokus auf das Power-Management für Automotive Systeme liegt in Kapitel 4, in dem auch in Abschnitt 4.3.1

die verschiedenen Power-Modi der Hardware vorgestellt werden.

Durch das gestiegene Interesse der Automobil-Hersteller an energieeffizienten ECUs hat sich das automotive Power-Management im Industrie-Standard AUTOSAR (engl. *automotive open system architecture*) etabliert. Der AUTOSAR-Standard wird im Detail in Abschnitt 2.4.2 vorgestellt. Die konkreten Konzepte für die Steigerung der Energieeffizienz in AUTOSAR werden in Kapitel 4 erläutert, wenn das Power-Management in Automotiven Systemen im Mittelpunkt steht.

1.3 Das PLASA-Projekt

Ein zentraler Bestandteil dieser Arbeit ist nicht nur die Untersuchung des bestehenden Power-Managements in Automotiven Systemen, sondern auch die Umsetzung und Integration eines Power-Managements in eine automotive Plattform. Dafür wurde die PLASA-Plattform (engl. *platform for automotive systems and applications*) gewählt, die am Fachgebiet für Betriebssysteme im Institut für Informatik der Technischen Universität München ab 2009 entstanden ist, um zum einen automotive Themen den Studenten in einem Praktikum der technischen Informatik näher zu bringen und zum anderen um verschiedene automotive Themen wissenschaftlich zu untersuchen.

Die PLASA-Plattform besteht grundlegend aus einem handelsüblichen 1:10 RC-Modell-Fahrzeug, das durch einen Elektromotor angetrieben wird. Von Beginn an wurde das Modell-Fahrzeug nicht wie üblich durch einen RC-Funkempfänger und einer RC-Fernbedienung gesteuert, sondern durch eine eingebaute ECU, die die Steuerung des Motors und des Lenk-Servos übernimmt. Die RC-Fernbedienung, mit der üblicherweise die RC-Modell-Fahrzeuge gesteuert werden, wurde durch eine Nintendo Wii Remote ausgetauscht. Die Nintendo Wii Remote ist über Bluetooth mit der ECU im Modell-Fahrzeug drahtlos verbunden. Dabei erfolgt die Steuerung des Modell-Fahrzeugs über das Neigen der Nintendo Wii Remote.

Neben der manuellen Steuerung war das automatisierte Fahren eine weitere Aufgabe in den Praktika. Dabei wurden verschiedene Aufgaben, wie das automatisierte Einparken oder das Folgen eines voraus fahrenden zweiten Modell-Fahrzeugs, gelöst. Während dieser Aufgaben wurde die PLASA-Plattform um verschiedene Sensoren wie einem Geschwindigkeitssensor oder Abstandssensoren erweitert, um die Aufgabenstellungen zu erfüllen.

Daraus hat sich ab 2010 die PLASA-Plattform entwickelt, die im Kapitel 5 näher beschrieben und für die Untersuchungen des Power-Managements für automotive Systeme herangezogen wird. In dieser werden zwei verschiedene Klassen von ECUs eingesetzt, wobei die eine Klasse ein 600 MHz getakteter ARM-Applikationsprozessor und die andere Klasse ein 8-Bit Mikrocontroller mit einem Takt von 16 MHz ist. Diese beiden ECU-Klassen werden in der PLASA-Plattform für unterschiedliche Zwecke eingesetzt. Während der Applikationsprozessor für rechenintensive Aufgaben verwendet wird und durch dessen drahtlose Kommunikationsfähigkeiten die Schnittstelle zur Außenwelt des Modell-Fahrzeugs darstellt, werden die Mikrocontroller für das Auslesen der Sensoren und für die Steuerung der Aktoren benutzt.

Alle ECUs sind über einen gemeinsamen Fahrzeug-Bus miteinander verbunden, um

die Fahrzeug-Funktionen verteilt zu erfüllen. Die ECUs, die für das Auslesen der Sensoren zuständig sind, versenden die Sensor-Werte über den Bus, damit diese von den anderen ECUs für deren Steuerungsberechnungen verwendet werden können.

In der PLASA-Plattform werden unterschiedliche Betriebssysteme eingesetzt. Auf dem Applikationsprozessor kamen im Laufe der Zeit zwei verschiedene Betriebssysteme zum Einsatz. Für das Praktikum wurde Symobi, ein Echtzeit-fähiges Microkernel-Betriebssystem, auf dem Applikationsprozessor verwendet. Symobi wird als Vertreter der Microkernel-Betriebssysteme näher in Abschnitt 3.2.2 vorgestellt. Im Laufe der Zeit kam es zu einem Generationswechsel der Hardware für die Klasse der ARM Applikationsprozessoren. Hier wurde auf einen Gumstix zurückgegriffen, der neben der Bluetooth-Schnittstelle noch eine Kommunikation über WLAN erlaubte. Für den Gumstix wird als Betriebssystem Linux eingesetzt, welches auch für die Untersuchung des Power-Managements für Automotive Systeme verwendet wird. Linux ist ursprünglich für eine Server- und Desktop-Umgebung entwickelt worden. Doch gab es schon immer Bestrebungen, Linux ebenfalls in den Embedded Systeme einzusetzen. Inzwischen wird der Linux-Kernel als Basis für Betriebssysteme in Smartphones und seit den letzten Jahren auch im automotiven Bereich eingesetzt. In der GENIVI-Allianz, einem Industrie-Standard für automotive Infotainment-Systeme, besteht das Referenz-System aus einem Linux-Kernel, worauf später in Abschnitt 2.4.3 detaillierter eingegangen wird.

Für die zweite ECU-Klasse der Mikrocontroller wird das Komponenten-basierte Betriebssystem TinyOS verwendet, das im Forschungsumfeld der WSNs entstanden ist. In Abschnitt 3.3 wird TinyOS im Detail vorgestellt, wo ebenfalls die Anforderungen der WSNs an ein Betriebssystem näher beschrieben werden. Nach genauer Betrachtung dieser Anforderungen wird ersichtlich, dass sich die Anforderungen der WSNs mit denen der Automotiven Systeme teilweise überschneiden. Eine der Anforderungen in den WSNs ist ein niedriger Energiebedarf, der unmittelbar in die Architektur von TinyOS eingeflossen ist. Da die PLASA-Plattform ebenfalls für die Untersuchung eines Power-Managements für Automotive Systeme eingesetzt wird, bietet sich die Gelegenheit, die Konzepte des Power-Managements in TinyOS für ein automotives Power-Management wiederzuverwenden.

Da TinyOS für den Einsatz in WSNs entwickelt wurde, gibt es in TinyOS vorrangig Software-Module für die Kommunikation und für das Auslesen von Sensoren. Aus diesem Grund fehlten für die PLASA-Plattform automotive Software-Komponenten, die die Steuerung des Motors oder des Lenk-Servos übernehmen. Deshalb wurden die fehlenden Software-Komponenten für TinyOS entwickelt, um die Ansteuerung des Motors und des Lenk-Servos in der PLASA-Plattform zu ermöglichen.

Die PLASA-Plattform hat sich im Laufe der Zeit gewandelt. Es wurden unterschiedliche ECUs mit verschiedenen Betriebssystemen eingesetzt. Im Kapitel 5 wird im Detail die PLASA-Plattform beschrieben, wie sie für die Untersuchung des automotiven Power-Managements genutzt wird. In diesem Kapitel werden zu Beginn die Anwendungsfälle der PLASA-Plattform erläutert. Aus den verschiedenen Fahrzeug-Funktionen, die während der Praktika von Studenten entwickelt wurden, ist eine Auswahl getroffen worden, die für die Untersuchung des automotiven Power-Managements genutzt werden. Aus der Auswahl der Anwendungsfälle wird eine logische Architektur definiert, die als Ausgangslage für das automotive Power-Management der PLASA-Plattform dient. In die-

sem Kapitel wird ebenfalls die Hard- und Software-Architektur der PLASA-Plattform zusammen mit den für TinyOS entwickelten Software-Komponenten vorgestellt.

1.4 Problemstellung der Arbeit

Aus der Historie sind verschiedene Spezifikationen, Konzepte und Mechanismen für das Power-Management in anderen Domänen als dem Automobil-Bereich entstanden. Die aktuellen Entwicklungen in den Automotiven Systemen gehen dahin, auch im Bereich des Logik-Anteils Energie einzusparen, um den Kraftstoffverbrauch des Fahrzeugs zu senken. Somit ist es von Interesse, bereits bekannte Konzepte und Mechanismen aus anderen Domänen, u. a. aus dem Bereich der WSNs zu untersuchen und gegebenenfalls in ein automotives Power-Management zu übernehmen. Besonders in Sensor-Knoten ist ein energieeffizientes Power-Management wichtig, da einmal ausgebrachte Sensor-Knoten typischerweise nicht mehr eingesammelt werden und nur solange in Betrieb sind, wie die Energie der beim Ausbringen eingesetzten Batterie ausreicht.

Ebenfalls existieren im Bereich der Desktops Mechanismen zur Steuerung der verschiedenen Power-Modi, die die Hardware bereitstellt. So findet das ACPI (engl. *advanced configuration and power interface*) seine Anwendung in Desktop-Rechnern, das eine standardisierte Schnittstelle bietet, um die Hardware vom Betriebssystem aus steuern zu können.

Allerdings unterscheiden sich die Anforderungen anderer Domänen, so dass Anpassungen in den Konzepten für die Automotiven Systeme gemacht werden müssen. Die bisherigen Konzepte des Power-Managements beschränken sich dabei auf ein alleinstehendes Gerät mit einem Prozessor bzw. einem Mikrocontroller. In einem Automotiven System dagegen sind bis zu 80 ECUs in einem heterogenen Netzwerk miteinander verknüpft. Somit sollte ein wirkungsvolles Power-Management für Automotive Systeme nicht über den lokalen Zustand der jeweiligen ECU seine Entscheidungen treffen, sondern es muss den Zustand des gesamten Fahrzeugs mit allen ECUs in Betracht ziehen.

Eine weitere Schwierigkeit neben dem verteilten System sind die Echtzeit-Anforderungen, die in Automotiven Systemen herrschen. Während bei den Sensor-Knoten in WSNs weiche Echtzeit-Anforderungen vorliegen, unterliegen Automotive Systeme harten Echtzeit-Bedingungen. Dies hat ebenfalls Auswirkungen auf ein automotives Power-Management. Während man in weichen Echtzeit-Systemen die Zeit, die für den Wechsel der Power-Modi benötigt wird, eventuell vernachlässigen kann, spielen diese in harten Echtzeit-Systemen eine wesentliche Rolle.

Mit der PLASA-Plattform als Basis soll in dieser Arbeit ein automotives Power-Management untersucht und weiterentwickelt werden. In erster Linie liegt der Fokus in Erweiterungen im Betriebssystem, um einen generalisierten Ansatz für einen energieeffizienten Betrieb des Fahrzeugs zu gewährleisten. Dabei sollen nur diejenigen Fahrzeug-Funktionen aktiv sein, die benötigt werden, um das Fahrzeug in dem entsprechenden Zustand zu steuern. Alle nicht benötigten Fahrzeug-Funktionen sollen demzufolge deaktiviert werden. Dadurch können Hardware-Komponenten in einen niedrigeren Power-Modus gefahren oder nicht mehr benötigte sogar abgeschaltet werden, um somit Energie zu sparen und einen energieeffizienten Betrieb zu gewährleisten.

An dieser Stelle setzt diese Arbeit an und geht der Frage nach, wie ein Power-Management für Automotive Systeme aussehen könnte und versucht gleichzeitig die Konzepte für ein Power-Management aus anderen Domänen der Embedded Systeme auf eine automotive Plattform zu übertragen.

1.5 Gliederung der Arbeit

Um einen Überblick über diese Dissertation zu geben, wird im Folgenden die weitere Gliederung der Arbeit vorgestellt.

Kapitel 2 — Automotive Systeme. In diesem Kapitel werden die Automotiven Systeme vorgestellt, wozu zuerst deren Geschichte und die Entstehung der heutigen Architektur beschrieben wird. Für die Software-Entwicklung von Automotiven Systemen werden verschiedene Abstraktionsebenen definiert, die im Anschluss kurz aufgezählt werden, bevor deren Hardware-Architektur präsentiert wird. Im Laufe der Zeit haben sich automotive Software-Standards etabliert, die danach vorgestellt werden. Das Kapitel wird mit den Trends für die Zukunft der Automotiven Systeme abgeschlossen.

Kapitel 3 — Software- und Betriebssysteme. Als Grundlage beleuchtet dieses Kapitel die Prinzipien der Software-Architektur und der Betriebssysteme. Zuerst werden die verschiedenen Arten von Betriebssystemen wie die Monolithischen, die Microkernel- und die Komponenten-basierten Betriebssysteme sowie deren Unterschiede vorgestellt. Auf TinyOS als ein Vertreter der Komponenten-basierten Betriebssysteme wird im Anschluss detaillierter eingegangen, da es in dieser Arbeit als Basis für die weiteren Untersuchungen des Power-Managements für Automotive Systeme herangezogen wurde.

Kapitel 4 — Power-Management in Automotiven Systemen. Das Kapitel fasst den Aufbau und die grundlegende Funktionsweise eines DPMs (engl. *dynamic power management*) für Automotive Systeme zusammen. Dafür werden zuerst die Anforderungen an ein Power-Management für Automotive Systeme gesammelt, bevor die Konzepte und die Architektur eines DPMs für Automotive Systeme beschrieben werden. Hierfür werden ebenfalls verwandte Arbeiten auf diesem Gebiet genannt. Der Fokus liegt im Anschluss auf dem PM (engl. *power manager*) in einer ECU, der die Hardware- und Software-Komponenten nach der Strategie des DPMs steuert.

Kapitel 5 — Die PLASA-Plattform. Für die Untersuchung eines DPMs für Automotive Systeme wurde die PLASA-Plattform herangezogen, die in diesem Kapitel im Detail vorgestellt wird. Dabei wird mit der Beschreibung der Anwendungsfälle begonnen, die die Grundlage für deren logischen Architektur bilden. Daraus leitet sich die Software-Architektur der PLASA-Plattform ab, die zum großen Teil auf der Basis von TinyOS aufgesetzt ist. An dieser Stelle wird näher darauf eingegangen, wie TinyOS um automotive Komponenten erweitert wurde und wie

das DPM für Automotive Systeme in Form eines PMs in TinyOS realisiert wurde. Vollständigkeitshalber ist ebenfalls die Hardware-Architektur der PLASA-Plattform am Ende des Kapitels beschrieben.

Kapitel 6 — Messungen aus der PLASA-Plattform. Zum Nachweis, dass das automotive Power-Management, das für die PLASA-Plattform entwickelt wurde, korrekt funktioniert, ist speziell für die PLASA-Plattform ein Mess-System entworfen worden, das zu Beginn des Kapitels kurz vorgestellt wird. Der Fokus in diesem Kapitel liegt allerdings auf der Präsentation der Messergebnisse, die mit Hilfe des Mess-System in der PLASA-Plattform gewonnen wurden. Aus den Messungen lassen sich durch Referenz-Messungen, die zuvor bei bekannten System-Zuständen durchgeführt wurden, die Entscheidungen des PMs nachvollziehen und zeigen, dass das DPM in der PLASA-Plattform korrekt funktioniert.

Kapitel 7 — Zusammenfassung und Ausblick. Zum Abschluss dieser Arbeit werden die Erkenntnisse aus der Implementierung des DPMs für die PLASA-Plattform noch einmal zusammengefasst. Außerdem werden noch deren Erweiterungs- und Verbesserungsmöglichkeiten genannt, die während der Entwicklung des DPMs für die PLASA-Plattform erkannt wurden. Als Ausblick wird geschildert, wie die Erkenntnisse, die aus der prototypischen Umsetzung des DPMs für die PLASA-Plattform erlangt wurden, in reale Automotive Systeme Einzug finden können und welche Herausforderungen dort noch bevorstehen.

Kapitel 2

Automotive Systeme

Die heutigen Automotiven Systeme sind historisch gewachsen. Zu Beginn waren die Automobile rein mechanische Systeme, in denen keine Elektronik verbaut wurde. Auch die ersten Motoren waren dabei ein mechanisches System, das ohne Elektronik zündete. Selbst die Scheibenwischer waren mechanisch und wurden von Hand betätigt. Nach und nach wurden die mechanischen Fahrzeug-Funktionen durch elektrische Systeme ersetzt. So wurde z. B. der Scheibenwischer von einem Elektromotor angetrieben und musste nicht mehr von Hand betätigt werden.

Ende der 1970er Jahre wurden die ersten Motorsteuergeräte entwickelt, die einen Mikrocontroller enthielten. Der Mikrocontroller steuerte mit Hilfe von am Motor angebrachte Sensoren die Zündung. Jedoch war die Steuer-Software rein von den lokal angeschlossenen Sensoren abhängig. Es war zu dieser Zeit noch kein Netzwerk von mehreren ECUs vorhanden, die sich gegenseitig Informationen zuschickten und verarbeiteten.

Im Lauf der Jahre kamen weitere Fahrzeug-Funktionen hinzu, die wiederum von eigenen, unabhängigen ECUs erfüllt wurden. Jedoch mussten die autarken ECUs Nachrichten austauschen, um ihre Funktion zu erfüllen. Als Beispiel sind hier einfache Sensor-Werte zu nennen, die von einer ECU an weitere ECUs gesendet werden. Somit mussten die ECUs über Bus-Systeme verbunden werden, um den Nachrichtenaustausch zwischen den ECUs zu ermöglichen. Dafür wurden speziell für den automotiven Bereich Bus-Systeme wie z. B. der CAN-Bus (siehe Abschnitt 2.3.2) entwickelt, die die speziellen automotiven Anforderungen erfüllen. Mit Hilfe der neuen Bus-Technologien wurden die vorhandenen ECUs im Fahrzeug zu einem Netz verbunden.

Aus der historischen Entwicklung der Automotiven Systeme wurde der Wandel ersichtlich, wie immer mehr Fahrzeug-Funktionen, die mechanisch realisiert waren, durch elektromagnetische Systeme erbracht werden, die über Software gesteuert werden [95]. Als Beispiel wurde der von Hand betriebene Scheibenwischer durch einen automatischen Scheibenwischer mit Regensensor ersetzt oder die über eine Kurbel betriebenen Fensterherber wurden nun in modernen Fahrzeugen elektrisch betrieben, wobei ebenfalls Software die Motoren für das Anheben und Absenken der Fenster steuert.

Somit wuchs über die letzten Jahrzehnten der Software-Anteil in Automotiven Systemen extrem an. Nicht nur der Wandel von den mechanischen Systemen hin zu den

elektromagnetischen Systemen, sondern auch der Anstieg der Fahrzeug-Funktionen trug dazu bei. Dieser wachsende Software-Anteil führt u. a. zu Herausforderungen in Automotiven Systemen, die im nächsten Abschnitt beleuchtet werden.

Über die Jahre hat sich eine für die Automotiven Systeme spezifische Architektur entwickelt, auf die im darauf folgenden Abschnitt eingegangen wird. Um die Automotiven Systeme mit ihrer Vielzahl an ECUs und ihren großen Software-Anteilen beherrschen zu können, werden die Automotiven Systeme in verschiedene Abstraktionsebenen eingeteilt, die ebenso in dem Abschnitt vorgestellt werden. Eine der Ebenen ist die sogenannte Elektrik/Elektronik Architektur (E/E-Architektur), in der beschrieben wird, welche ECUs verbaut sind und über welche Bus-Systeme diese vernetzt sind.

Ein wesentlicher Bestandteil eines Automotiven Systems ist die verbaute Hardware, die darauf folgend in Abschnitt 2.3 vorgestellt wird. In diesem Abschnitt werden die Hardware-Komponenten einer ECU näher untersucht und deren Aufbau kurz erläutert. Ein wesentlicher Bestandteil dieses Abschnitts sind die in den ECUs verbauten Mikrocontroller. Zwei unterschiedlich leistungsstarke Mikrocontroller werden in diesem Abschnitt vorgestellt: der 8-Bit Atmel ATmega1284P Mikrocontroller und der OMAP3530 Applikationsprozessor von Texas Instruments. Ebenfalls wird auf die automotiven Bus-Systeme mit ihren Eigenschaften eingegangen, bevor die Funktionsweise der Sensoren und der Aktoren beschrieben wird.

Neben den automotiven Hardware-Komponenten werden drei automotive Software-Standards vorgestellt: die OSEK/VDX-Spezifikation, der AUTOSAR-Standard und der GENIVI-Standard. Die OSEK/VDX-Spezifikation und der AUTOSAR-Standard beschreiben die automotive Software-Architektur Domänen-unabhängig, während sich der GENIVI-Standard auf die Infotainment-Domäne beschränkt.

Um dieses Kapitel über die Automotiven Systeme abzuschließen, werden die zukünftigen Trends der Automotiven Systeme zum Abschluss beleuchtet. An dieser Stelle wird dargestellt, in welche Richtung sich die Automotiven Systeme entwickeln werden.

2.1 Herausforderungen von Automotiven Systemen

Bevor die konkreten Herausforderungen der Automotiven Systeme diskutiert werden, muss zuvor die Domäne der Automotiven Systeme analysiert werden. Dazu gehört u. a. das Arbeitsmodell, wie die Automotiven Systeme entwickelt werden, aber auch die wirtschaftlichen Umstände in der Automobil-Industrie.

In der Automobil-Branche hat sich historisch folgendes Arbeitsmodell etabliert: Die Automobil-Hersteller, die in der Automobil-Branche auch als OEM (engl. *original equipment manufacturer*) bezeichnet werden, bestimmen die Fahrzeug- und Komfortfunktionen für ihre verschiedenen Modelle. Im Anschluss werden die Teilfunktionen auf die verschiedenen ECUs verteilt. Dieser Arbeitsschritt wird Partitionierung genannt, der ebenfalls vom OEM erbracht wird. Im Anschluss entscheiden die OEMs, welche ECU sie selbst entwickeln und welche ECUs sie von einen Automobil-Zulieferer zukaufen oder entwickeln lassen. Dadurch übernehmen die Automobil-Zulieferer, die auch als Tier 1 bezeichnet werden, einen Großteil der Fahrzeug-Funktionsentwicklung, da gewöhnlich die OEMs viele der ECUs von den Tier 1 entwickeln lassen. Sobald die ECUs vom

Tier 1 fertiggestellt wurde, werden diese vom OEM in das Gesamt-Fahrzeug integriert und getestet. Der OEM übernimmt hier die Verantwortung für das Gesamtfahrzeug, während der Zulieferer die Verantwortung einer einzelnen ECU übernimmt [34].

Ein weiterer Einfluss-Faktor in der Entwicklung der Fahrzeug-Funktionen ist der Kosten-Druck, der in der Automobil-Branche vorherrscht. Da die ECUs in Millionen von Fahrzeugen verbaut werden, ist die Einsparung in den Stückkosten pro ECU von ein paar Cents bis zu Euros entscheidend. Durch die hohen Stückzahlen wird aus dem geringen Betrag pro ECU schnell eine größere Summe für die gesamten produzierten Fahrzeuge. Somit lohnt es sich für die OEMs sehr genau auf die Stückkosten zu achten und hier jede mögliche Einsparung vorzunehmen und zu nutzen.

Dies führt zu einer Kosten-getriebenen Entwicklung, in der jede ECU mit der jeweils günstigsten Hardware bestückt wird, die benötigt wird, um die von dem OEM geforderten Funktionen zu erfüllen. Somit werden die Rechen-Leistungen, der verbaute Speicher und die Art der Kommunikationscontroller auf die entsprechenden Anforderungen für die jeweilige ECU ausgesucht und abgestimmt. Um weitere Kosten zu sparen, wird meist nicht auf die aktuelle Generation der Mikrocontroller gesetzt, sondern auf eine ältere Generation. Denn diese Hardware-Komponenten sind meist günstiger als die aktuelle Generation.

Der Kosten-Druck hat allerdings nicht nur Auswirkungen auf die Hardware, sondern auch auf die Kommunikationsbusse. Selbst die Busse werden Kosten-getrieben entwickelt. Für einfache Bus-Systeme, die für den Anschluss von Sensoren vorgesehen sind, wird auf eine zusätzliche Absicherung im Bus-Protokoll und damit auf eine aufwendigere Bus-Kommunikation verzichtet. Denn diese aufwendigere Bus-Kommunikation führt zu größeren Logik-Anteilen, die in Software oder Hardware realisiert werden müssen. Damit entstehen höhere Entwicklungskosten für die Funktionen, die es aus Kostengründen zu vermeiden gilt. Zusätzlich führt dies unweigerlich auch zu größeren Hardware-Kosten, da entweder zusätzliche Hardware verbaut werden muss, wenn der Logik-Anteil in Hardware realisiert wird, oder ein leistungsstärkerer Mikrocontroller muss verwendet werden, wenn der Logik-Anteil in Software implementiert ist. Sofern es aufgrund der Anforderungen möglich ist, gilt es, unnötigen Aufwand und somit Kosten zu vermeiden.

Aus der spezifischen Architektur der Automotiven Systeme, die im nächsten Abschnitt beschrieben wird, ergeben sich Herausforderungen für die Software-Entwicklung, die sich wie folgt nach M. Broy [33] zusammenfassen lassen:

Verteiltes System. Automotive Systeme sind stark verteilte Systeme, in denen die Fahrzeug-Funktionen von unterschiedlichen ECUs erbracht werden. Dabei müssen Informationen eventuell über verschiedene Teilnetze ausgetauscht werden.

Größe der Software. Die Software für Automotive Systeme ist in den letzten 35 Jahren entstanden und hat inzwischen die Code-Größe von ca. 10.000.000 Code-Zeilen erreicht. Dies ist ein immenses Wachstum, auch mit dem Hintergrund, dass die Software von Fahrzeug-Generation zu Fahrzeug-Generation auf Grund der starken Hardware-Abhängigkeit des Codes neu geschrieben werden musste und eine Wiederverwendbarkeit des bereits geschriebenen Codes nicht gegeben war.

Harte Echtzeit-Anforderungen. In den Domänen der sicherheitskritischen Fahrzeug-

Funktionen herrschen harte Echtzeit-Anforderungen vor, die nicht nur Auswirkungen auf die Software hat, sondern durch die Vernetzung mit anderen ECUs auch die Bus-Systeme beeinflusst. Somit werden in diesen Teilnetzen Echtzeit-fähige Bus-Systeme benötigt, die die Einhaltung der Echtzeit-Anforderungen garantieren können.

Nicht-funktionale Anforderungen. Selbst in Domänen mit weichen Echtzeit-Anforderungen existieren nicht-funktionale Anforderungen bezüglich der Aufstart-Zeit der ECUs und der Fahrzeug-Funktionen. In der Infotainment-Domäne lässt sich als Beispiel die maximale Zeitdauer bis zur Ausgabe des Radio-Tons auf die Lautsprecher nach Aufstarten der Head-Unit nennen.

Zuverlässigkeit der Software. Die sicherheitsrelevanten Funktionen müssen zuverlässig sein. Neben den Echtzeit-Anforderungen dürfen in Automotiven Systemen die Fahrzeug-Funktionen nicht ausfallen. Selbst in den nicht sicherheitskritischen Bereichen akzeptiert ein Fahrzeug-Nutzer nicht den Ausfall einer Funktion. Im Desktop-Bereich ist dagegen die Akzeptanz eines Absturzes viel größer. Somit ist der Anspruch bezüglich der Qualität der Software in Automotiven Systemen um einiges höher.

Update-Fähigkeit. Eine weitere Herausforderung für die Software in Automotiven Systemen ist die Schwierigkeit die Software in bereits verkauften Fahrzeugen zu aktualisieren. Diese Problematik erhöht sich noch durch den hohen Qualitätsanspruch an die Software. Somit ist es nicht leicht, bereits gefundene Software-Fehler durch Updates in bereits verkauften Fahrzeugen zu beheben. Dies ist für den Fahrzeug-Nutzer nur durch den Besuch einer Werkstatt möglich.

Software Logistik. Durch die große Varianten-Vielfalt durch unterschiedliche Modell- und Konfigurationsmöglichkeiten in den Fahrassistenz- und Komfortfunktionen ist es ebenfalls schwierig, die für die vorhandenen Hardware-Konfiguration passende Software bereitzustellen.

Neben den technischen Herausforderungen in Automotiven Systemen, die zuvor genannt wurden, existieren zusätzlich noch Herausforderungen im Bereich der Arbeitsweise in der Automobil-Branche.

Anforderungsmanagement. Zu den technischen Herausforderungen eines stark verteilten Systems kommt zusätzlich hinzu, dass die ECUs von verschiedene Tier 1 Zulieferern entwickelt und an den OEM geliefert werden. Der OEM muss dafür sorgen, dass die Anforderungen an alle Tier 1 Zulieferer, die für die verschiedenen ECUs verantwortlich sind, übereinstimmen und die einzelnen ECUs mit ihren Teilfunktionen, die Gesamtfunktion erfüllen.

Kosten. Wie in der Einführung über die Automotiven Systemen beschrieben, herrscht in der Automobil-Branche ein enormer Kosten-Druck, der ebenfalls Auswirkungen auf die Entwicklung der ECUs hat. Somit ist es erforderlich, dass Kosten-Modelle, die bis jetzt noch nicht existieren, Einzug in der ECU-Entwicklung finden.

Produktzyklen. Für jede Fahrzeug-Generation wird die Software neu programmiert, um die Hardware-nahe Software an die neue Hardware-Generation anzupassen. Jedoch muss die neue Software wieder den gesamten Prozess der Zertifizierung durchlaufen. Dies bedeutet einen erhöhten Aufwand, der durch eine Wiederverwendung der vorherigen Software vermieden werden könnte. Allerdings müsste dann ein Großteil der Software Hardware-unabhängig implementiert sein.

Energieeffizienz. Durch die immer steigenden Benzin-Preise und durch die Strafabgaben für die Automobil-Hersteller bei einer Nicht-Einhaltung des durchschnittlichen CO_2-Ausstoßes in dessen Fahrzeug-Flotte steigt ebenfalls das Energie-Bewusstsein in der Automobil-Branche. Somit wird das Bedürfnis nach energieeffizienten Erfüllung der Fahrzeug-Funktionen immer größer.

Mit diesen Herausforderungen ist eine solide Architektur der Automotiven Systeme notwendig, die als nächstes vorgestellt wird.

2.2 Architektur der Automotiven Systeme

Mit der zu Beginn dieses Kapitels beschriebenen Historie der Automobil-Branche lässt sich die große Heterogenität in den Automotiven Systemen erklären, die bis heute Auswirkungen auf die Architektur der Automotiven Systeme hat.

Zum Einstieg in die Architektur der Automotiven Systeme wird die E/E-Architektur der heutigen Fahrzeuge vorgestellt, die sich aufgrund der unterschiedlichen Anforderungen in den verschiedenen Fahrzeug-Domänen, die ebenfalls in dem nächsten Abschnitt zusammen mit der E/E-Architektur genannt werden, entwickelt hat. Jedoch ist es nicht ausreichend, die verschiedenen Fahrzeug-Domänen in eigenen Netzwerken zusammenzufassen. Denn die Fahrzeug-Funktionen, vor allem im Bereich der Fahrassistenz, werden Domänen-übergreifend und verteilt über mehrere ECUs realisiert. Um nach Gesichtspunkten des Software-Engineerings ein Automotives System ausreichend zu beschreiben, muss eine Architektur mit unterschiedlichen Abstraktionsebenen erstellt werden, die im darauf folgenden Abschnitt beleuchtet wird.

2.2.1 E/E-Architektur der Automotiven Systeme

Wie in den vorigen Kapiteln beschrieben, sind in einem Fahrzeug bis zu 80 ECUs verbaut, die jeweils spezifische Fahrzeug-Funktionen oder Teilfunktionen erfüllen. Die ECUs sind über ein Bus-System miteinander verbunden, um Informationen auszutauschen. Aus Gründen der unterschiedlichen Echtzeit-Anforderungen, der Sicherheits- und Zuverlässigkeitsansprüche sind die ECUs nicht über einen gemeinsamen Bus verbunden, sondern sind in unterschiedliche Teilnetze aufgeteilt. Um jedoch einen Informationsaustausch zwischen den Teilnetzen zu ermöglichen, sind alle Teilnetze durch das sogenannte Zentrale Gateway (ZGW) verbunden, das die Nachrichten bei Bedarf von einem Teilnetz in ein anderes weiterleitet.

Diese Architektur wird als E/E-Architektur bezeichnet, die für jedes Automotive System spezifisch ist. Die E/E-Architektur beschreibt dabei nicht nur die logische Archi-

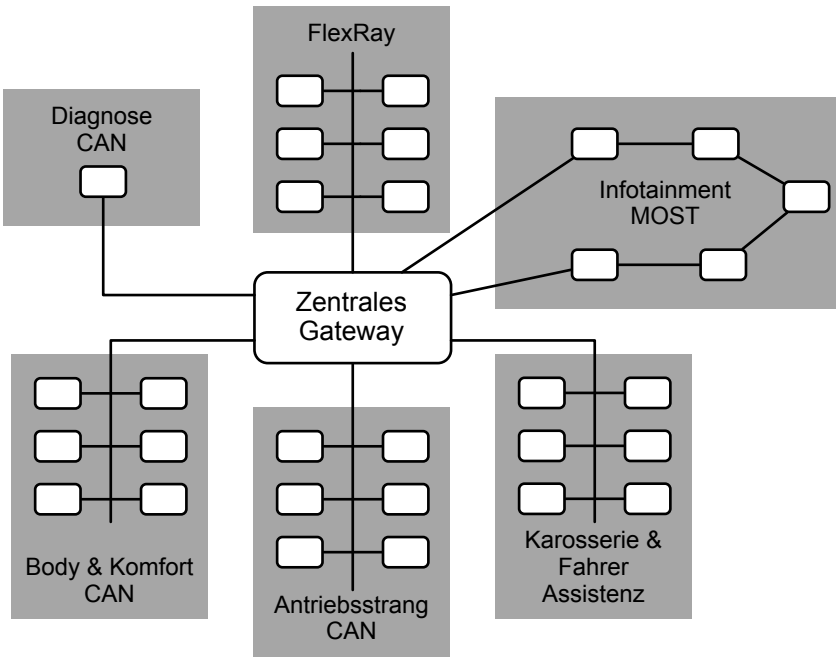


Abbildung 2.1: Übersicht über ein Automotives System (angelehnt an eine Abbildung aus der Veröffentlichung von C. Schmutzler *et al.* [132]).

tektur des Automotiven Systems sondern auch die physikalische Architektur mit den konkreten Stromversorgungs- und Daten-Leitungen der ECUs [133].

Typischerweise werden den verschiedenen funktionalen Domänen, die jeweils eigene Anforderungen bezüglich der Zuverlässigkeit, der Sicherheit oder der Echtzeit-Fähigkeit haben, eigene Teilnetze zugeordnet, die sogar über unterschiedliche automotive Bus-Systeme verbunden sein können. In einem Fahrzeug werden dabei bis zu fünf verschiedene Domänen unterschieden: Antriebsstrang, Body & Komfort, Karosserie & Fahrer Assistenz, Infotainment und Diagnose. In Abbildung 2.1 ist ein typisches Automotives System mit seinen Teilnetzen dargestellt, wie es in den heutigen Fahrzeugen zu finden ist [132, 134].

Die in der Abbildung dargestellten Fahrzeug-Domänen haben unterschiedliche Aufgabenbereiche, die im Folgenden vorgestellt werden:

Antriebsstrang. Der Antriebsstrang (engl. *power train*) enthält u. a. das Motorsteuergerät, das die Zündungen im Motor kontrolliert, und somit für den Antrieb des Fahrzeugs sorgt. Des Weiteren sind die ECUs für das Getriebe oder die Funktion der Geschwindigkeitsregelung in dieser Domäne.

Body & Komfort. In der Fahrgestell-Domäne (engl. *body*) sind die ECUs für die Fahrstabilisierung miteinander vernetzt. Außerdem sind ebenfalls die Sitzmodule in dieser Domäne, die für die elektrische Sitzverstellung für Fahrer und Beifahrer verantwortlich sind.

Karosserie & Fahrer-Assistenz. In dieser Domäne werden ECUs, die in der Karosserie

verbaut sind, miteinander verbunden. Dazu zählen u. a. die Scheibenheber, aber auch die notwendigen Sensoren für die Fahrer Assistenz-Systeme, wie Abstandssensoren oder Video-Kameras.

Infotainment. In der Domäne Infotainment sind die Informations- und Entertainment-Funktionen eines Fahrzeugs zusammengefasst. Darunter fallen die Head-Unit, der Audio-Verstärker oder der CD-Wechsler. Die Head-Unit vereinigt hierbei die Funktionen Radio, Multimedia, Navigation und Informationsterminal für den Fahrer in einer ECU.

FlexRay. In dieser Domäne werden ECUs, auf denen sicherheitskritische sowie Bandbreiten-intensive Applikationen ausgeführt werden, über den deterministischen, Echtzeit-fähigen und Fehler-toleranten FlexRay-Bus miteinander verbunden. Diese Applikationen können meistens der Karosserie und Fahrer-Assistenz Domäne logisch zugeordnet werden. Da der FlexRay-Bus allerdings ein eigenes Bus-System darstellt und wie die anderen Domänen an das ZGW angeschlossen ist, wird dieser in eine eigene Domäne ausgelagert.

Diagnose. Der Diagnose-Bus wird für die OBD (engl. *on-board diagnostics*) verwendet. Damit ist es möglich, eine Diagnose der im Fahrzeug verbauten ECUs durchzuführen. Dies ist für Werkstatt-Mechaniker von Interesse, wenn ein Fahrzeug mit einem Fehler zum Service in die Werkstatt kommt.

In den Automotiven Systemen herrscht eine große Heterogenität bei den ECUs, bei den Bus-Systemen, bei den Sensoren und bei den Aktoren vor. Jede Domäne hat ihre eigenen Anforderungen, auf die die Hardware angepasst und kostenoptimiert wird. Somit kommen unterschiedliche Mikrocontroller in den ECUs vor, die näher in Abschnitt 2.3.1 beschrieben werden. Auch die Ausstattung der einzelnen ECUs unterscheiden sich je nach deren Funktionalität. Jede ECU besitzt genau diejenigen I/O Schnittstellen, die sie für die Erfüllung ihrer Funktion benötigt. Damit ergibt sich eine Heterogenität der ECUs, die neben der Vielzahl an Steuergeräte in einem Automotiven System bei der System- und Software-Entwicklung berücksichtigt werden muss.

Zusätzlich werden in den verschiedenen Domänen unterschiedliche Bus-Systeme eingesetzt, die in Abschnitt 2.3.2 näher vorgestellt werden. Dabei unterscheiden sich die Domänen in ihren Anforderungen, womit nicht nur die ECUs, sondern auch die Bus-Systeme speziell an die Anforderungen der Domäne angepasst sind. In den Domänen Body und Antriebsstrang kommt der für den automotiven Bereich entwickelte CAN-Bus zum Einsatz. In der Infotainment-Domäne ist dagegen der MOST-Bus verbreitet, der durch eine hohe Datenübertragung für die Übertragung der Medieninhalte ausgelegt ist.

Die jeweilige E/E-Architektur eines Automotiven Systems ist abhängig vom Modell und der Fahrzeug-Konfiguration, die der Käufer beim Kauf eines Fahrzeugs durch die Auswahl der verschiedenen Ausstattungsmöglichkeiten bestimmt. Durch die Kosten-Optimierung werden nur die ECUs im Fahrzeug verbaut, die auch benötigt werden, um die vom Käufer konfigurierten Fahrzeug-Funktionen zu erfüllen. Daraus ergibt sich allerdings, dass es zu einer vermehrten Anzahl an unterschiedlichen Kombinationen von Sensoren, ECUs und Aktoren kommt, die es bei der System-Entwicklung zu berücksichtigen gilt.

Der Einsatz von neuen Bus-Systemen wird in Automotiven Systemen erschwert. Die Gründe liegen zum einen an den langen Produktzyklen eines Fahrzeug-Modells, die bei ca. sieben Jahren liegen. Zum anderen ist der Austausch eines alten Bus-Systems durch ein neues mit zusätzlichen Kosten verbunden, die in der Kosten-getriebenen Automobil-Branche vermieden werden. Denn eine neue E/E-Architektur mit neuen Bus-Systemen bedeutet, dass bereits vorhandene ECUs für die neuen Bus-Systeme neu entwickelt werden müssen, obwohl die zu erbringende Funktionalität eventuell gleich bleibt. Somit ist der Einsatz eines neuen Bus-Systems vorerst mit zusätzlichen Kosten verbunden – ohne einen Gewinn an zusätzlichen Fahrzeug-Funktionen. Aus diesem Grund wird es vermieden, ECUs neu zu entwickeln und die alten Bus-Systeme bleiben so lange wie möglich in den Automotiven Systemen vorhanden.

2.2.2 Abstraktionsebenen und Modell-basierte Entwicklung in Automotiven Systemen

In einem Fahrzeug sind je nach Ausstattung und Klasse insgesamt bis zu 80 ECUs verbaut, die zusammen ein verteiltes Echtzeit-System bilden. Auf diesen ECUs laufen bis zu 10.000.000 Source Code Zeilen [33]. Dabei ist jede ECU einzeln eventuell von unterschiedlichen Entwickler-Teams programmiert worden, die nur die Spezifikation einer ECU umgesetzt haben. Dabei haben die Entwickler-Teams nur in den wenigsten Fällen das gesamte Automotive System mit den ca. 2000 verteilten Fahrzeug-Funktionen betrachtet und besitzen selten den Zugriff auf den Source Code der übrigen ECUs, um ein genaues Verständnis der anderen Funktionen zu erhalten [35].

Mit dieser hohen Anzahl an ECUs und der enormen Software-Größe ist es notwendig, die Komplexität des gesamten Automotiven Systems zu reduzieren. Dies wird durch die Definition mehrere Abstraktionsebenen erreicht, die unterschiedliche Schwerpunkte haben. M. Broy *et al.* schlagen dafür sechs Ebenen vor, die im Folgenden erläutert werden [34].

Usage Level. Das *Usage Level* bietet eine Sicht, welche Funktionen dem Benutzer eines Fahrzeugs angeboten werden. Dabei sind als Benutzer nicht nur der Fahrer sondern auch die Beifahrer und die Werkstatt-Mechaniker und weitere Personen für die Wartung des Fahrzeuges eingeschlossen. Es wird eine Funktionshierarchie (engl. *function hierarchy*) aufgebaut, in der zusammenhängende Dienste nach der Art, wie das Fahrzeug die Dienste anbietet, gruppiert werden. Dabei wird der Datenfluss und die Interaktion der Benutzer mit den Diensten u. a. mit Message Sequence Charts dargestellt.

Design Level. In der Ebene *Design Level* wird die logische Architektur des Fahrzeugs beschrieben. Die Fahrzeug-Funktionen werden in verteilte logische Komponenten unterteilt, die über definierte Schnittstellen miteinander kommunizieren. Es wird nicht darauf geachtet, ob die Komponenten von Hardware oder Software realisiert werden. Das Verhalten der logischen Komponenten wird dabei mit Hilfe von abstrakten Algorithmen beschrieben.

Cluster Level. Das *Cluster Level* ist eine Zwischenschicht, in der die logische Architektur zu sogenannten Clustern gruppiert wird, um daraus leichter die Software-Architektur in der darunter liegenden Ebene abzuleiten. Somit ist diese Ebene eine Übergangsschicht hin zur Software-Ebene, um den Schritt der Übertragung von der logischen Architektur zu der Software-Architektur zu vereinfachen.

Software Level. In der Ebene *Software Level* wird zum einen die Software-Architektur beschrieben und zum anderen die Prozess- und Task-Sicht inklusive deren Scheduling. Dabei wird eingeteilt, welche Funktionen im Betriebssystem von Treibern und Diensten oder auf Applikationsebene in Software-Komponenten erfüllt werden. Eine detaillierte Architektur-Beschreibung von Betriebssystemen und Applikationsebene erfolgt im Abschnitt 3.1.

Hardware Level. Die Hardware-Architektur wird in der Ebene *Hardware Level* festgelegt. Hier werden Hardware-Komponenten wie ECUs, Bus-Systeme, Sensoren und Aktoren sowie deren Vernetzung dargestellt. Eine nähere Beschreibung der automotiven Hardware erfolgt in Abschnitt 2.3.

Deployment. In der Ebene *Deployment* wird das Hardware/Software-Codesign des Automotiven Systems in einer technischen Architektur beschrieben. Daraus ergibt sich die konkrete Realisierung der logischen Architektur, indem die Komponenten aus der logischen Architektur durch das Zusammenspiel aus Hardware und Software in der technischen Architektur realisiert werden.

Während der Entwicklung eines Automotiven Systems muss sich eine Abstraktionsebene von der darüber liegenden Ebene ableiten. So ergibt sich die logische Architektur des Automotiven Systems aus den Kundenfunktionen, die der OEM für das System festlegt. Die logische Architektur dient dann als Ausgangslage für die Software- und Hardware-Architektur des Automotiven Systems, die über die Zwischenebene *Cluster Level* entwickelt werden. Zum Schluss werden die Komponenten aus der logischen Architektur auf die Komponenten die technische Architektur zugeordnet, die für jede ECU sowohl die Hardware-Komponenten, wie Sensor und Aktoren als auch die Software-Funktionen festlegt.

Die Abbildung von der logischen Architektur hin zur technischen Architektur soll anhand eines kleinen Beispiels, das in Abbildung 2.2 dargestellt ist, kurz verdeutlicht werden. Dabei werden die Abstraktionsebenen *Cluster Level*, *Software Level* und *Hardware Level* nicht weiter detailliert, um darzustellen, wie eine logische Architektur auf eine technische Architektur abgebildet werden kann. Die Software- und Hardware-Architektur unterscheiden sich je nach konkretem System. In Kapitel 3 über die Software- und Betriebssysteme werden unterschiedliche Software-Architekturen vorgestellt. Zusätzlich wird in Kapitel 5 die Software- und Hardware-Architektur der PLASA-Plattform beschrieben.

Auf der oberen Hälfte der Abbildung 2.2 ist eine logische Architektur mit den drei unterschiedlichen Blöcken dargestellt. Auf der linken Seite sind die Datenquellen Quelle_1 bis Quelle_4, die die Sensor-Daten aufzeichnen und diese an die logischen Funktionen Funktion_1, Funktion_2 und Funktion_3 weiterleiten. Über die zwei weiteren Funktionen

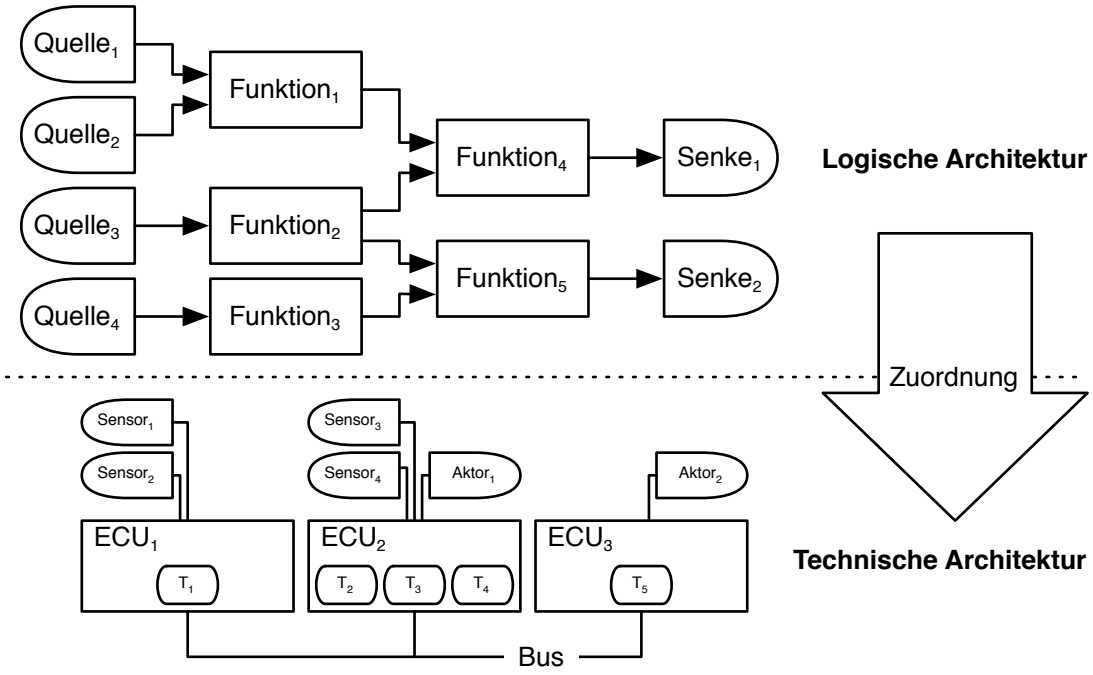


Abbildung 2.2: Zuordnung der logischen auf die technische Architektur eines Automotiven Systems.

Funktion_4 und Funktion_5 werden die Daten an die Daten-Senken Senke_1 und Senke_2 auf der rechten Seite gesendet, die am Ende der Kette ihren Einfluss auf den physikalischen Prozess nehmen.

In einem Schritt des Software-Entwicklungsprozesses wird die logische Architektur auf die technische Architektur abgebildet. Das bedeutet, dass jeder Block in der logischen Architektur einer ECU zugeordnet wird. Dabei sind in den meisten Fällen Randbedingungen zu beachten, wie die technischen Anschlüsse einer ECU oder dass eine ECU nicht überlastet wird. Dafür ist in den meisten Fällen Expertenwissen notwendig und kann nicht rein automatisch durch Software-Werkzeuge erledigt werden. In der Abbildung 2.2 wird eine mögliche Zuordnung der bereits vorgestellten logischen Architektur in der unteren Hälfte als technische Architektur graphisch dargestellt. Die technische Architektur in diesem Beispiel besteht aus den drei ECUs ECU_1, ECU_2 und ECU_3, die über einen Bus miteinander verbunden sind. An der ECU_1 hängen die Sensoren Sensor_1 und Sensor_2, an der ECU_2 die Sensoren Sensor_3 und Sensor_4 sowie der Aktor Aktor_1 und entsprechend an der ECU_3 der Aktor Aktor_2. Über die Zuordnung werden die Daten-Quellen Quelle_1 bis Quelle_4 aus der logischen Architektur entsprechend auf die Sensoren Sensor_1 bis Sensor_4 zugeordnet und die Daten-Senken Senke_1 und Senke_2 den Aktoren Aktor_1 und Aktor_2. Die logischen Funktionen werden den Software-Tasks T_1 bis T_5 zugeteilt, die auf die drei ECUs verteilt werden müssen. In diesem Beispiel läuft der Software-Task T_1 auf der ECU_1, die Software-Tasks T_2, T_3 und T_4 auf der ECU_2 und der Software-Task T_5 auf der ECU_3.

Dies ist nur ein kleines und übersichtliches Beispiel für eine logische und technische

Architektur. Für ein komplettes Fahrzeug ist die logische Architektur mit den ca. 2000 Kundenfunktionen um ein vielfaches größer. Die technische Architektur entspricht dabei der bereits im Abschnitt 2.2.1 vorgestellten E/E-Architektur, die ebenfalls mit den bis zu 80 ECUs abhängig von der Fahrzeug-Klasse und deren Ausstattung um ein vielfaches größer ausfällt.

Für jede einzelne Abstraktionsebene existieren unterschiedliche Möglichkeiten, wie die Ebenen beschrieben werden können. So wird für die Modellierung von Echtzeit-Systemen, unter die auch die Automotiven Systeme fallen, auch UML (engl. *unified modeling language*) vorgeschlagen, um eine agile Software-Entwicklung zu ermöglichen [77]. Um jedoch die Aspekte der Automotiven Systeme zu berücksichtigen, wurde in einem EU-Forschungsprojekt die Architektur-Beschreibungssprache EAST-ADL (Electronics Architecture and Software Technology - Architecture Description Language) für Automotive Systeme entwickelt, um die Architekturen für die verschiedenen Abstraktionsebenen beschreiben zu können [40, 39].

In der Forschung wird eine Modell-basierte Entwicklung vorgeschlagen, in der für die verschiedenen Abstraktionsebenen unterschiedliche Modelle genutzt werden. Im Englischen hat sich dafür auch der Begriff MDD (engl. *model-driven development*) etabliert. Das Ziel der Modell-basierten Entwicklung ist es, das Software-Engineering für Automotive Systeme komfortabler und zuverlässiger zu machen. Aus der Sicht des Software-Engineerings lassen sich daraus Herausforderungen ableiten, die A. Pretschner *et al.* in ihrer Veröffentlichung darlegen [123]. In der Wissenschaft sind viele Beiträge über eine Modell-basierte Entwicklung in Automotive Systemen zu finden. Z. B. wurde an der Technischen Universität München in einem Projekt ein durchgängig Modell-gestützter Ansatz untersucht, um aus der Modellierungssprache CoLa (Component Language) Maschinen-Code zu erzeugen und diesen anschließend verteilt auf die Ziel-Steuergeräte zu bringen [63]. Dabei ist ebenfalls das Scheduling, die Verteilung und das Modell-basierte Auffinden von Fehlern untersucht worden.

Die bisherigen Beispiele stammen vorrangig aus der Forschung. Allerdings hat auch die Automobil-Industrie aufgrund der Herausforderungen, die in den Automotiven Systemen herrschen, mehrere Software-Standards geschaffen, die in Abschnitt 2.4 vorgestellt werden. Einer der Standards ist AUTOSAR, der eine logische Architektur für ein Automotives System, eine ECU Software-Architektur und eine Methodik für die Entwicklung eines Automotiven Systems definiert. Allerdings deckt AUTOSAR nicht alle Abstraktionsebenen ab, da der Schwerpunkt bei AUTOSAR in der Software-Entwicklung liegt, wie später ausführlich in Abschnitt 2.4.2 erläutert wird.

Die Abstraktionsebenen fließen auch in unterschiedliche Modellierungs- und Entwicklungswerkzeuge für Automotive Systeme ein. So teilt PREEvision, ein Modellierungswerkzeug, die E/E-Architektur ebenfalls in vier Ebenen ein, die den vorgestellten Abstraktionsebenen zugeordnet werden können [76].

2.3 Hardware in Automotiven Systemen

Nachdem die Architektur der Automotiven Systeme im vorherigen Abschnitt vorgestellt wurde, liegt der Schwerpunkt dieses Kapitels in der Hardware und den Bus-Systemen,

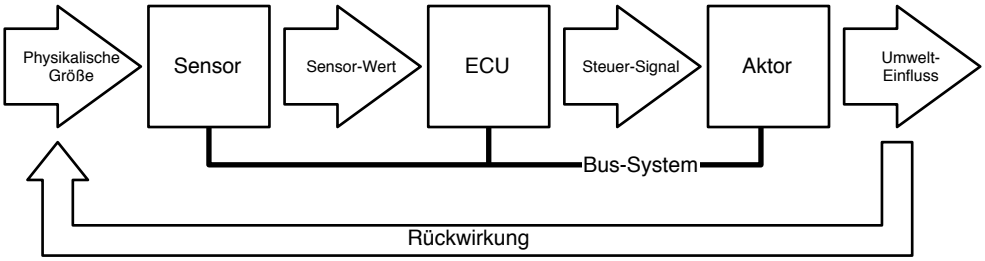


Abbildung 2.3: Übersicht über die Hardware und deren Einbettung in den physikalischen Prozess.

die in den heutigen Automotiven Systemen eingesetzt werden. Darunter fallen die ECUs zusammen mit den Sensoren und den Aktoren.

Auf den ECUs – auch Steuergeräte genannt – werden die Sensor-Daten verarbeitet und die Aktoren gesteuert. Dafür wird auf den ECUs die spezifische Software ausgeführt, die die Fahrzeug-Funktionen realisiert. Wie bereits im Abschnitt 2.2 über die Architektur Automotiver Systeme beschrieben wurde, sind die ECUs über Bus-Systeme miteinander vernetzt, so dass diese gemeinsam kommunizieren und Daten austauschen können. In der Regel wird eine Fahrzeug-Funktion nicht alleine von einer ECU erfüllt, sondern diese wird verteilt von mehreren ECUs erbracht.

In Abbildung 2.3 ist der Zusammenhang zwischen Sensor, ECU, Aktor und Bus-System noch einmal übersichtlich dargestellt. Zuerst müssen die Sensoren die Umwelt erfassen und einen Sensor-Wert einer physikalischen Größe der ECU bereitstellen. Diese werden von einer oder mehreren ECUs verarbeitet. Daraus wird das Steuer-Signal für den Aktor generiert, der im Anschluss Einfluss auf die Umwelt nimmt. Dieser hat dann Rückwirkung auf die gemessene physikalische Größe, die von dem Sensor aufgenommen wird. Die Sensoren, ECUs und Aktoren sind über verschiedene Bus-Systeme und Fahrzeug-Netzwerke miteinander verbunden, wie es schon im vorherigen Kapitel beschrieben wurde.

Im Weiteren wird der Aufbau und die Funktionsweise der ECUs vorgestellt. Dabei sind einer der zentralen Komponenten einer ECU die Mikrocontroller, auf denen die Steuer-Software ausgeführt wird. Im Anschluss wird eine Übersicht über die automotiven Bus-Systemen gegeben, bevor in den weiteren Abschnitten auf die Sensoren und Aktoren eingegangen wird.

2.3.1 ECUs in Automotiven Systemen

Die zentrale Rolle in der E/E-Architektur kommt den ECUs zu, durch die die Fahrzeug-Funktionen realisiert werden. Wie im Abschnitt 2.1 über die Herausforderungen der Automotiven Systeme beschrieben wurde, sind die einzelnen ECUs in ihrem Aufbau und ihren Funktionen sehr heterogen, da diese von verschiedenen Automobil-Zulieferern entwickelt werden und sehr stark auf ihre zu erbringende Funktion ausgelegt und optimiert sind. Dennoch lassen sich die ECUs in logische Teilkomponenten unterteilen, die im Folgenden vorgestellt werden.

DC/DC Wandler. In Automotiven Systemen wird ein Bord-Netz mit einer Versorgungsspannung von 12 V verwendet. Wenn die Teilkomponenten einer ECU nicht mit einer Versorgungsspannung von 12 V arbeiten, muss ein DC/DC-Wandler die 12 V in eine meist niedrigere Versorgungsspannung auf der ECU transformieren.

Mikrocontroller. Die zentrale Steuer- und Berechnungseinheit auf der ECU sind die Mikrocontroller. Je nach Steuer- oder Berechnungsaufwand, die eine ECU bereitstellen muss, sind die Mikrocontroller unterschiedlich leistungsstark. C. Schmutzler *et al.* geben einen Überblick über die Leistungsbereiche und definieren die Bereiche *Low End*, *Mid End*, *High End* und *High End Powertrain* [132].

Speicher. Ebenfalls enthält jede ECU Speicher, in dem die benötigten Daten abgelegt sind. Dabei wird zwischen flüchtigem und nicht-flüchtigem Speicher unterschieden. In dem flüchtigen Speicher gehen die Daten verloren, wenn der Speicher nicht mehr mit Strom versorgt wird. Im Gegensatz dazu behält der nicht-flüchtige Speicher seinen Inhalt. Zudem wird bei dem nicht-flüchtigen Speicher zwischen Programm- und Daten-Speicher unterschieden. Im Programm-Speicher wird nur ausführbarer Code abgelegt, während im nicht-flüchtigen Daten-Speicher die Daten persistent abgespeichert werden.

Kommunikationscontroller. Die ECUs sind über ein Bus-System miteinander verbunden. Die Kommunikationscontroller sind die Schnittstelle zum jeweiligen Bus, an dem die ECU angeschlossen ist. Die Kommunikationscontroller sind dafür verantwortlich, die Daten auf den physikalischen Kanal zu legen. Sie decken typischerweise die Schicht 1 (Bitübertragungsschicht (engl. *physical layer*)) und Schicht 2 (Sicherungsschicht (engl. *data link layer*)) des ISO/OSI Referenz-Modells ab.

Analoge Sensoren. Für das Erfassen der physikalischen Größen sind auf einer ECU analoge Sensoren enthalten, die eine physikalische Größe in einen Strom- oder Spannungswert umwandeln. Dabei gibt es unterschiedliche Arten von Sensoren, die je nach Funktion der ECU und Bedarf verbaut werden. Eine Beschreibung der Sensoren erfolgt später in Abschnitt 2.3.3.

ADC (engl. analog-to-digital converter). Um die analogen Werte der Sensoren auf dem Mikrocontroller verarbeiten zu können, müssen diese zuerst von einem ADC digitalisiert werden. Diese digitalisierten Sensor-Werte werden im Anschluss an den Mikrocontroller weitergeleitet, der diese für weitere Berechnungen benutzt.

Die Mikrocontroller sind auf den ECUs die zentrale Komponente, die sich je nach Mikrocontroller wiederum in weitere Controller aufteilen lassen. Der Mikroprozessor ist in dem Mikrocontroller die Berechnungseinheit, die aus der ALU und den Registern besteht. Neben dem Mikroprozessor sind noch weitere Peripherie-Bausteine wie Timer, Kommunikationscontroller oder ADCs in einem Mikrocontroller enthalten. Diese Kombination aus Timern, Kommunikationscontroller und ADCs hat sich für die Steuerung von Embedded Systemen bewährt und hat sich auch in den Automotiven Systemen durchgesetzt. Die Vereinigung mehrerer Controller zusammen mit dem Mikroprozessor auf einem Chip wird auch als SoC (engl. *system on a chip*) bezeichnet.

In der Domäne der Smartphones hat sich dagegen der Begriff des Applikationsprozessors (engl. *application processor*) etabliert. Hier steht im Vergleich zu den Mikrocontrollern nicht die Steuerung von physikalischen Prozessen, sondern die Ausführung von Anwendungen mit einer graphischen Benutzeroberfläche und die Verarbeitung von Daten im Vordergrund. Als Beispiel für die Applikationen können ein mobiler Internet-Browser, E-Mail-Clients oder Navigationsapplikationen genannt werden. In Automotiven Systemen werden die Applikationsprozessoren in Head-Units eingesetzt, bei der die Ausführung von Anwendungen mit einer graphischen Benutzeroberfläche im Fokus liegt [127]. So werden Navigations-, Multimedia- oder Büro-Anwendungen, wie Email-Clients oder Kontaktbuch auf der Head-Unit ausgeführt, die höhere Anforderungen bezüglich der Rechenleistung an den Mikroprozessor haben.

Im Folgenden wird ein Mikrocontroller und ein Applikationsprozessor mit ihren Peripherie-Bausteinen näher vorgestellt. Als Beispiel für einen Mikrocontroller wird der Atmel ATmega1284P, der aus dem 8-Bit Low End Bereich stammt, beschrieben. Als Applikationsprozessor ist an dieser Stelle der Texas Instruments OMAP3530 genannt, von dem die leistungsstärkere, automotive-fähige Variante DRA74x OMAP existiert [152].

Der Atmel ATmega1284P Mikrocontroller

Zuerst wird der Atmel ATmega1284P Mikrocontroller näher vorgestellt, der ein 8-Bit Mikrocontroller mit einem Systemtakt von bis zu 20 MHz ist [7]. Der darin enthaltene Mikroprozessor besitzt 32 8-Bit Register und die AVR RISC-Architektur mit 131 Befehlen, die in der Regel in einem Systemtakt ausgeführt werden können [9]. Dabei besitzen die Befehle keine unterschiedlichen Privileg-Stufen, so dass keine Systemwechsel zwischen den privilegierten und den nicht-privilegierten Modus des Prozessors stattfinden. Ebenfalls gibt es beim ATmega1284P keine Unterstützung für eine virtuelle Adressierung durch eine MMU (engl. *memory management unit*). D. h. es existiert keine durch die Hardware unterstützte Adressraum-Trennung für Applikationen, die näher im Abschnitt 3.2 über die Software-Architektur und Betriebssysteme erläutert wird.

Neben dem Mikroprozessor besitzt der ATmega1284P Mikrocontroller auch Speicher auf dem Chip. Der Flash-Speicher für Programme beträgt 128 kB, während der EEPROM-Speicher für die nicht-flüchtigen Daten 4 kB groß ist. Als internes RAM stehen dem ATmega1284P 16 kB zur Verfügung.

Eine Übersicht über den ATmega1284P gibt Abbildung 2.4, bei der ebenfalls weitere Peripherie-Komponenten dargestellt sind. Diese lassen sich auf dem ATmega1284P in Timer, Kommunikationscontrollern und ADC einteilen. Der ATmega1284P unterstützt mit seinen Kommunikationscontrollern verschiedene Bus-Systeme wie den USART (engl. *universal synchronous/asynchronous receiver transmitter*), den I^2C-Bus (engl. *inter-integrated circuit*), den SPI-Bus (engl. *serial peripheral interface*) oder auch für Debugging Zwecke den JTAG (engl. *joint test action group*).

Um mit der Außenwelt zu interagieren, besitzt der ATmega1284P Reset-, Stromversorgungs-, Clock- sowie 32 Daten-Pins, von denen jeweils 8 zu einem sogenannten Port zusammengefasst sind. Somit existieren in dem ATmega1284P insgesamt 4 Ports, die mit Port A bis Port D benannt werden. Die dazu gehörigen Daten-Pins werden entsprechend PA0 bis PA7 oder PD0 bis PD7 benannt. Die zuvor beschriebenen Controller legen

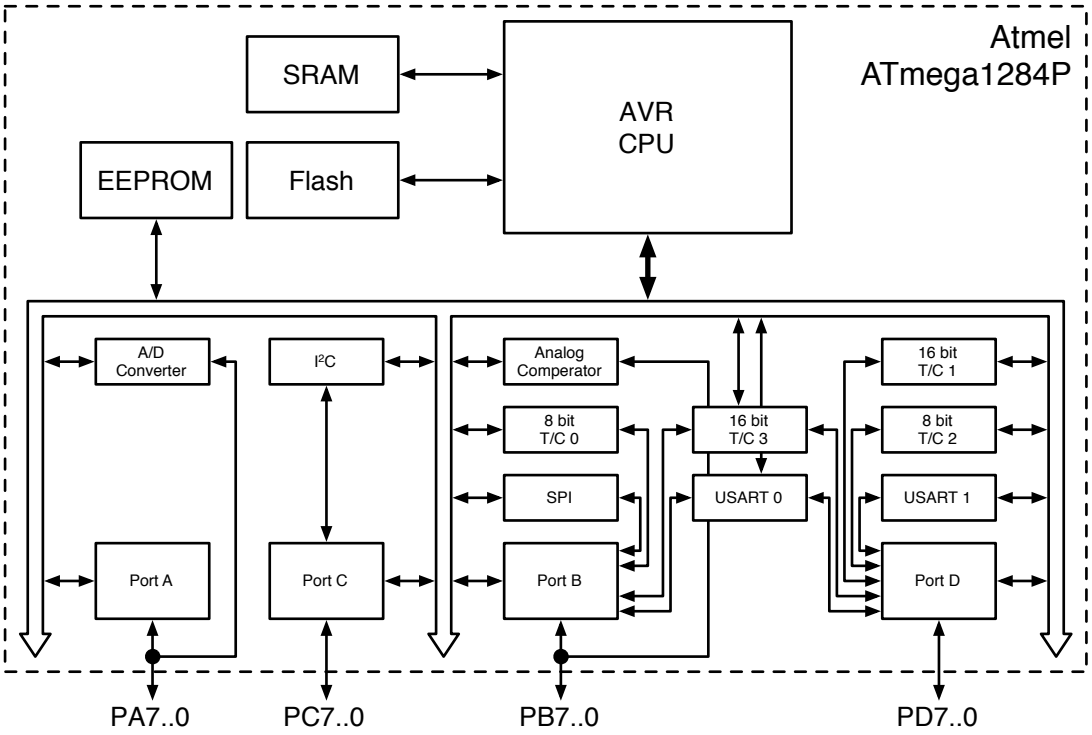


Abbildung 2.4: Blockbild des Atmel ATmega1284Ps (angelehnt an eine Abbildung aus dessen Spezifikation [7]).

ihre Datensignale an dem spezifizierten Pin. Daneben können die Pins auch einzeln als Ausganspins angesteuert oder als Eingangspins über den jeweiligen Port ausgelesen werden.

Der ATmega1284P besitzt zwei 8-Bit und zwei 16-Bit Timer. Timer sind Zähler, die im Systemtakt des Chips ein Register hoch zählen, bis ein Überlauf im Registers stattfindet. Dies verursacht einen Overflow Interrupt, den anschließend die Software dafür nutzen kann, um einen Timer zu implementieren, der nach einer voreingestellten Zeit abläuft. Neben dem Zählregister hat der ATmega1284P auch zwei Vergleichsregister, die ständig mit dem Zählregister verglichen werden. Sobald eines der Vergleichsregister den selben Wert wie das Zählregister hat, wird ein weiterer Interrupt erzeugt, der wiederum in der Software genutzt werden kann. Aufbauend auf den Vergleichsregistern kann der ATmega1284P PWM (engl. *pulse-width modulation*) Signale an den dafür vorgesehenen Ausgangspins generieren. Die beiden Flankenwechsel des PWM-Signals entstehen zum einen beim Überlauf des Zählregisters zum anderen bei Gleichheit zwischen Zähl- und Vergleichsregister.

Für die Kommunikation bietet der ATmega1284P zwei Controller für die USART Schnittstelle an. Zusätzlich besitzt der ATmega1284P die Möglichkeit, über einen Kommunikationscontroller für den SPI-Bus zu kommunizieren. Über den SPI-Bus kann der Flash-Speicher für die Programm-Daten des ATmega1284P beschrieben werden. Dafür besitzt der ATmega1284P einen speziellen Zustand, in dem er über den SPI-Bus pro-

grammiert werden kann. Als weiteren Controller besitzt der ATmega1284P einen I^2C-Bus Kommunikationscontroller, der sowohl als Master als auch als Slave auf dem I^2C-Bus betrieben werden kann.

Neben dem Kommunikationscontroller ist auf dem ATmega1284P auch ein Analog-Comperator enthalten, der zwei Spannungswerte vergleicht und einen Interrupt erzeugt, falls die positive Eingangsleitung eine höhere Spannung aufweist als die negative Eingangsleitung. Über eine Einstellung kann der erzeugte Interrupt zu einem Timer/Counter des ATmega1284P geleitet werden, der beim Auftreten des Interrupts den aktuellen Zählerstand in einem zweiten dafür vorgesehenen Register speichert.

Für die Analog-Digital-Wandlung besitzt der ATmega1284P intern einen SAR (engl. *successive approximation register*) ADC mit einer 10-Bit Auflösung. Darüber lassen sich über einen Multiplexer bis zu 8 verschiedene Signale, die am Port A anliegen, gegenüber einer Referenz-Spannung messen und digitalisieren.

Der Texas Instruments OMAP3530 Applikationsprozessor

Der Applikationsprozessor OMAP3530 von Texas Instruments Instruments besitzt einen ARM Cortex-A8 Mikroprozessor [153], der bis zu 700 MHz getaktet werden kann. Neben dem Mikroprozessor sind noch ein DSP (engl. *digital signal processor*) für eine Audio- und Video-Beschleunigung und ein POWERVR SGX530 Graphik-Beschleuniger auf dem Chip. Somit ist der OMAP3530 für Anwendungen im Bereich des Video-Streamings, der 2D/3D Videospiele und der Video-Konferenzsysteme ausgelegt und findet seine Integration in Smartphones, in Navigationsgeräten, in mobilen Spiele-Konsolen oder in Infotainment-Systemen.

In Abbildung 2.5 wird ein Überblick über den Texas Instruments OMAP3530 Applikationsprozessor mit seinen Peripherie-Komponenten gegeben. Als zentraler Mikroprozessor kommt in dem OMAP3530 der ARM Cortex-A8 zum Einsatz, der nach der ARM v7 Referenz-Architektur implementiert ist [6]. Dieser enthält 16 32-Bit Register, wobei davon 13 für den freien Gebrauch verwendet werden können und drei eine zugewiesene Aufgabe wie den Stack-Zeiger oder den Programmzähler enthalten. Die Befehle des Cortex-A8 Mikroprozessors besitzen unterschiedliche Privileg-Stufen, so dass ein Kernel-Modus, in dem die Hardware-Ressourcen zugeteilt werden können, vom Cortex-A8 Hardware-mäßig unterstützt wird. Ebenfalls enthält der Cortex-A8 Mikroprozessor eine MMU, die eine Umsetzung der virtuellen Adressen in eine physikalische Adresse Hardware-unterstützt vornimmt.

Neben dem Cortex-A8 Mikroprozessor enthält der OMAP3530 noch weitere Peripherie-Komponenten auf dem Chip, die in Abbildung 2.5 dargestellt sind. Für die Anbindung von flüchtigem und nicht-flüchtigem Speicher werden zwei verschiedene Speicher Controller verwendet, die sowohl den NAND- und NOR-Flash-Standard als auch den SDRAM-Standard unterstützen.

Für den Datentransfer zwischen den verschiedenen Peripherie-Komponenten besitzt der OMAP3530 einen DMA Controller mit 32 Kanälen. Dieser ermöglicht es, Speicherbereiche ohne die Verwendung des Mikroprozessors zu kopieren. Somit ist es möglich, dass die Peripherie-Komponenten die Daten in den Hauptspeicher kopieren können, ohne dass dafür der Mikroprozessor, der in der Zwischenzeit für weitere Berechnungen

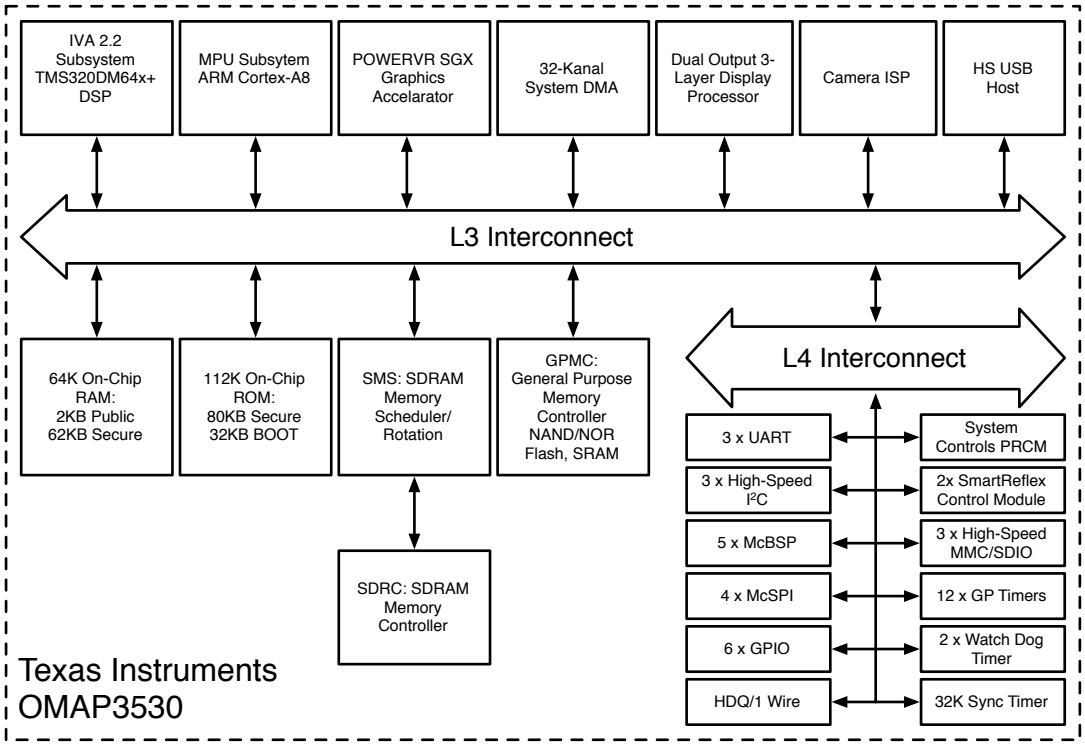


Abbildung 2.5: Blockbild des Texas Instruments OMAP3530s (angelehnt an eine Abbildung aus dessen Spezifikation [153]).

verwendet werden kann, benötigt wird.

Der OMAP3530 enthält den POWERVR SGX530 2D/3D Graphik Beschleuniger. Dieser bietet neben einer 2D-Beschleunigung auch eine Beschleunigung im 3D-Bereich an. Ebenfalls besitzt der OMAP3530 für die Bildschirm-Ausgabe einen Display Controller, mit dem LCDs (engl. *liquid crystal display*) gesteuert werden können. Somit erhält der OMAP3530 die Möglichkeit, Bildinhalte zu berechnen und an Displays auszugeben, ohne dabei an zusätzliche externe Controller angewiesen zu sein. Zusätzlich besitzt der OMAP3530 auch einen ISP (engl. *image signal processor*) mit einem Kamera Eingang. Darüber können Bilder und Videos von einem Bild-Sensor aufgenommen und weiterverarbeitet werden, bevor die Bilddaten in den Hauptspeicher geschrieben werden.

Des Weiteren besitzt der OMAP3530 verschiedene Kommunikationscontroller. So ist ein USB (engl. *universal serial bus*) Host-Subsystem verbaut, das 3 USB-Ports enthält, womit weitere USB-Geräte angeschlossen werden können. Neben dem USB Host-Subsystem besitzt der OMAP3530 zusätzlich noch einen USB-OTG (engl. *universal serial bus on-the-go*) Controller, mit dem es auch möglich ist, den OMAP3530 als USB-Device anzusprechen.

Neben dem USB-Controller stehen eine Reihe weiterer Kommunikationscontroller zur Verfügung, über die verschiedene Hardware-Komponenten mit dem OMAP3530 verbunden werden können. So besitzt der OMAP3530 einen UART (engl. *universal asynchronous receiver transmitter*) Controller für die Anbindung externer Geräte über die se-

rielle Schnittstelle, worüber auch Infrarot-Komponenten angeschlossen werden können. Zusätzlich sind Kommunikationscontroller für den I^2C-Bus, den SPI-Bus und den BSP (engl. *buffered serial port*) enthalten.

Ebenfalls ist der OMAP3530 für den Anschluss externer Speicher-Karten gerüstet und besitzt ein MMC/SD-Card-Interface, das es ihm erlaubt Speicher-Karten auszulesen.

Für die Zeitmessung sind im OMAP3530 11 Timer für den freien Gebrauch enthalten. Diese können wie beim Atmel dazu genutzt werden, um PWM Signale an verschiedenen Ausgängen anzulegen. Der OMAP3530 besitzt auch zwei Watchdog Timer, über die zum einen die MPU zum anderen der IVA-DSP überwacht werden kann.

Für das Abfragen (engl. *capture*) einzelner Eingangspins und für die Steuerung einzelner Ausgangspins besitzt der OMAP3530 sechs GPIO Module, wobei jedes Modul 32 Pins steuern kann.

2.3.2 Automotive Bus-Systeme

Wie am Anfang dieses Kapitels über die historische Entwicklung der Automotiven Systeme beschrieben wurde, sind im Laufe der Zeit immer mehr ECUs ins Fahrzeug integriert worden. Diese arbeiteten zuerst nur selbstständig und waren nicht von anderen ECUs abhängig. Als Abhängigkeiten lassen sich z. B. Daten von Sensoren, die physikalisch an andere ECUs angeschlossen sind oder Funktionszustände nennen, die in anderen ECUs verwaltet werden.

Um benötigte Sensor-Werte oder Funktionszustände zwischen den ECUs auszutauschen, wurden die ECUs im Laufe der Zeit immer mehr über ein Bus-System vernetzt. Einer der ersten Fahrzeug-Busse war der CAN-Bus (engl. *controller area network*) [84], der von der Robert Bosch GmbH ab 1983 entwickelt und schließlich 1987 veröffentlicht wurde. Inzwischen ist der CAN-Bus als ISO 11898 standardisiert worden und wird immer noch in den heutigen Automotiven Systemen eingesetzt. Der CAN-Bus besitzt zwei unterschiedliche Raten. Die schnellere Highspeed hat eine Datenübertragungsrate von bis zu 1 MBit/s [85] und die Lowspeed eine Rate von 125 kBit/s [86].

Der Bus arbeitet nach Arbitrierungsprinzip CSMA/CA (engl. *carrier sense multiple access/collision avoidance*), so dass keine Kollisionen beim Bus-Zugriff durch mehrerer Bus-Teilnehmer auftreten. Dies wird durch eine verlustfreie Bit-Arbitrierung realisiert, indem jeder Sender ebenfalls den Bus überwacht. Beim Senden des Identifiers überschreibt das dominante Bit das rezessive Bit, so dass der unterlegene Sender beim Erkennen des dominanten Bits auf dem Bus die Kommunikation beendet. Dies hat allerdings Auswirkungen auf das Echtzeit-Verhalten des Systems, da der nicht-dominante Sender seine Nachricht nicht übertragen kann und das Senden zu einem späteren Zeitpunkt wiederholen muss.

Doch durch die steigende Zahl der ECUs war es schwierig, mit dem Event-getriebenen CAN-Bus die Echtzeit-Anforderungen zu erfüllen. Deshalb wurde auf Basis des CAN-Bus der TT-CAN (engl. *time-triggered controller area network*) [87] entwickelt, der eine Echtzeit-Erweiterung des CAN-Bus darstellt.

Dennoch stiegen die Echtzeit-Anforderungen in den sicherheitskritischen Teilnetzen, so dass ein Echtzeit-fähiger Bus von Grund auf neu entwickelt wurde. Um diese Anforderungen zu bewältigen, wurde der FlexRay-Bus entwickelt. Hier wurde als Arbi-

Name	Bitrate	Topologie	Bemerkung
CAN-Bus	1 Mbit/s	Bus	Event-triggered automotiver Bus.
FlexRay	10 Mbit/s je Kanal	Bus	Echtzeit-fähiger Bus für sicherheitskritische Domänen.
MOST-Bus	23 MBaud	Ring	Bus mit hoher Daten-Rate für die Infotainment-Domäne.
LIN-Bus	19,2 kbit/s	Bus	Kostengünstiger Bus für den Anschluss von Sensoren.

Tabelle 2.1: Übersicht über die automotiven Bus-Systeme.

trierungsprinzip TDMA (engl. *time division multiple access*) eingesetzt, bei dem jeder Teilnehmer einen fest definierten Zeit-Slot für seine Übertragung bekommt.

Im Abschnitt 2.2 über die Architektur der heutigen Fahrzeuge wurden die Domänen in Teilnetze unterteilt, die jeweils ganz spezifische Anforderungen besitzen. Diese Anforderungen wirken sich auch auf die Bus-Systeme aus. So unterscheiden sich die Anforderungen für die Infotainment-Domäne so stark, so dass der MOST-Bus (engl. *media oriented systems transport*) entwickelt wurde, der für eine hohe Datenübertragung ausgelegt ist. Diese ist in der Infotainment-Domäne für die Verteilung der Audio- und Video-Signale zwischen den ECUs notwendig [60].

Neben der Vernetzung von ECUs müssen auch intelligente Sensoren oder Aktoren an die ECUs angeschlossen werden. Da der CAN-Bus, der FlexRay-Bus oder der MOST-Bus für eine einfache Anbindung eines Sensors oder Aktors zu aufwendig und deshalb zu teuer ist, wurde für diesen Zweck der LIN-Bus (engl. *local interconnect network*) entwickelt, der eine einfachere, für Sensoren ausreichende Kommunikation erlaubt [61].

In Tabelle 2.1 wird eine Übersicht über die hier genannten automotiven Bus-Systeme, die in den heutigen Fahrzeugen eingesetzt werden, gegeben. Ebenfalls wird auf die Übertragungsrate und die Topologie der einzelnen Bus-Systeme eingegangen. Eine weitere Übersicht über die Kommunikation kann bei N. Navet *et al.* nachgelesen werden [113, 114].

2.3.3 Sensoren in Automotiven Systemen

Für eine exakte Steuerung der physikalischen Prozesse durch Aktoren muss zuerst die Umwelt durch Sensoren erfasst werden. In Automotiven Systemen werden eine Reihe unterschiedlicher Sensoren, wie Temperatur-, Druck-, Luftzug- oder Abgas-Sensoren eingesetzt, wie es am Beispiel der Motorsteuerung von J. Cook *et al.* in ihrer Veröffentlichung beschrieben ist [37]. Diese Art von Sensoren werden für die Steuerung des Motors genutzt. Je nach Aktor müssen unterschiedliche physikalische Größen erfasst werden. In modernen Fahrzeugen kommen abhängig von der zu realisierenden Fahrzeug-Funktion weitere Sensoren wie Helligkeitssensoren, Abstandssensoren oder sogar Video-Kameras zum Einsatz, um Fahrassistenz-Funktionen umzusetzen [164].

Um die Funktionsweise von Sensoren besser zu verstehen, wird in Abbildung 2.6 der Datenfluss und die Digitalisierung der analogen Sensor-Werte dargestellt. Als erstes wird die zu messende physikalische Größe von einem sogenannten analogen Sensor aufgenommen und in einen Spannungs- oder Stromwert gewandelt. Dieser analoge Sensor-Wert wird im zweiten Schritt von einem ADC digitalisiert, der anschließend von einem Mi-

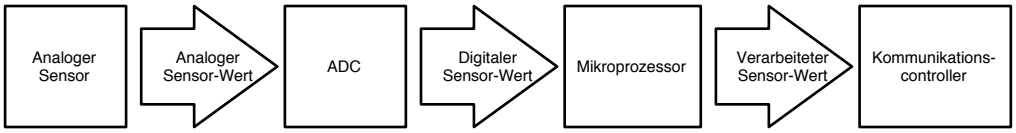


Abbildung 2.6: Datenfluss bei Sensoren.

kroprozessor weiterverarbeitet wird. Je nach Sensor wird der digitale Sensor-Wert im Mikrocontroller umgerechnet und weiterverarbeitet, bevor er anschließend über einen Kommunikationscontroller auf den Bus gelegt wird.

In Automotiven Systemen sind die ECU-Grenzen bei diesem Datenfluss nicht festgelegt. D. h. je nach Sensor und ECU kann sich der analoge Sensor auf der ECU befinden oder über Leitungen an die ECU angeschlossen sein. Ebenfalls ist es denkbar, dass ein Sensor die komplette Datenfluss-Kette abdeckt und als Ausgabe ein digitalen Sensor-Wert liefert. In diesem Fall spricht man von einem sogenannten digitalen Sensor.

Ein wichtiger Teil dieser Kette ist die Digitalisierung des Sensor-Werts im ADC. Es gibt unterschiedliche Wandlungstechniken, die unterschiedliche lang dauern können und unterschiedliche Bitbreiten haben. Je nach Anforderungen an den Sensor kommen unterschiedliche Umwandlungstechniken zum Einsatz.

2.3.4 Aktoren in Automotiven Systemen

Nachdem die Umwelt mit Hilfe von Sensoren erfasst und der vorzunehmende Eingriff in den physikalischen Prozess von den ECUs berechnet wurde, nehmen nun die Aktoren Einfluss auf den physikalischen Prozess. Dabei gibt es eine Vielzahl von unterschiedlichen Aktoren wie Ventile, Pumpen, Zündungen oder elektrische Motoren. Für die Steuerung eines Verbrennungsmotors werden z. B. Ventile und Zündungen benötigt, die von dem Motorsteuergerät kontrolliert werden [37]. Auf einer höheren Abstraktionsebene kann der Motorblock mit dem Motorsteuergerät, den Sensoren und der Vielzahl von kleinen Aktoren als ein einzelner Aktor gesehen werden, der das Fahrzeug antreibt.

Anhand des Beispiels des Verbrennungsmotors ist ersichtlich, dass ein Aktor ebenfalls ein abgeschlossenes System aus Sensoren und Aktoren ist, das nach außen hin eine Aktor-Funktion anbietet. Neben dem Antrieb muss das Fahrzeug auch gelenkt oder gebremst werden, wozu ebenfalls Systeme für die Lenkkraft-Unterstützung oder Brems-Systeme als Aktoren benötigt werden [164]. Für die Fahr-Stabilisierung sind elektromagnetische Stabilisatoren im Fahrzeug verbaut, die das Fahrzeug bei unebenen Untergrund ruhig halten.

Eine andere Art von Aktor, der nicht für die Antriebs-, Lenk-, Brems- oder Fahrstabilierungsfunktion zuständig ist, ist die Beleuchtung im Fahrzeug. Diese setzt sich aus unterschiedlichen Scheinwerfern oder Signal-Leuchten zusammen.

Für die Komfort-Funktionen sind ebenfalls eine Reihe weiterer Aktoren notwendig, wie z. B. die Heizsysteme, die von der Heckscheiben- bis zur Sitz-Heizung reichen, oder die elektrischen Fensterhebern oder elektrische Schiebedächer. Für die Temperatur und Feuchtigkeitsregelung im Fahrerraum gibt es die Klimaautomatik, die ebenfalls als ein Aktor aufgefasst werden kann.

Die kurze Aufzählung der verschiedenen Aktoren zeigt, dass in den modernen Fahrzeugen eine Vielzahl unterschiedlicher Aktoren verbaut ist, die auf ihre Weise Einfluss auf die physikalischen Prozesse nehmen. Wie die Anzahl der Aktoren so unterscheiden sich auch die Möglichkeiten, wie die Aktoren angesteuert werden. So lassen sich Aktoren über PWM-Signale ansteuern, bei denen das Verhältnis von High- und Low-Signalen den Aktor steuert. Bei anderen Aktoren kann es sein, dass das Steuergerät mit dem Fahrzeug-Bus verbunden ist und seine Steuer-Signale über Bus-Nachrichten erhält.

2.4 Automotive Software-Standards

Wie am Anfang des Kapitels über die Entwicklung der Automotiven Systeme beschrieben worden ist, waren die ersten Fahrzeuge mechanische Systeme, die sich im Laufe der Zeit zu elektromagnetischen Systemen wandelten. Der Anteil der Software, die eine ECU steuert, war gering. Jedoch stiegen die Fahrzeug-Funktionen stetig an, womit die Codegröße der System-Software einer ECU ebenfalls wuchs.

Erschwerend kommt zu der Software-Größe hinzu, dass die Kundenfunktionen verteilt über mehrere ECUs erbracht werden. Dies führt dazu, dass ein Austausch der Informationen über die ECU-Grenzen hinweg erfolgen muss. Da die Steuergeräte eventuell von mehreren Zulieferern entwickelt und gefertigt werden, müssen die Protokolle für den Datenaustausch zwischen den verschiedenen Zulieferern einheitlich sein, um die gesamte Kundenfunktion realisieren zu können.

Aus diesen beiden Gründen sind mehrere Industrie-Standards entstanden, die im Folgenden vorgestellt werden. Einer der ersten Spezifikationen für die automotiven Betriebssysteme war der OSEK/VDX-Standard, der sich auf die Standardisierung eines Echtzeit-Betriebssystems konzentrierte. Aufbauend auf dem OSEK/VDX-Standard wurde ab 2004 der AUTOSAR-Standard eingeführt, der eine Software-Architektur für die ECUs spezifizierte. Neben AUTOSAR wurde 2009 die GENIVI-Allianz gegründet, die sich die Standardisierung der Software-Komponenten auf der Head-Unit als Ziel gesetzt hat. In den folgenden Abschnitten werden die verschiedenen Standards im einzelnen vorgestellt.

2.4.1 Der OSEK/VDX-Standard

Im Mai 1993 gründeten die Projektpartner BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW und die Universität Karlsruhe als Koordinator OSEK (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) für die Standardisierung einer offenen Architektur für die verteilten ECUs im Fahrzeug [119]. Die französischen Automobil-Hersteller PSA und Renault schlossen sich 1994 dem Standard an und brachten ihren VDX (engl. *vehicle distributed executive*) Ansatz in die Standardisierung mit ein. Beide Ansätze wurden im Anschluss zusammengeführt, woraus der OSEK/VDX-Standard im Oktober 1997 veröffentlicht wurde.

Das Ziel des OSEK/VDX-Standards ist es, die Portabilität und Wiederverwendbarkeit der automotiven Anwendungssoftware zu gewährleisten. Dies wird durch die Definition einer abstrakten und anwendungsunabhängigen Schnittstelle für die Applikationssoftware erreicht, die zusätzlich unabhängig von der darunter liegenden Hardware ist. Des

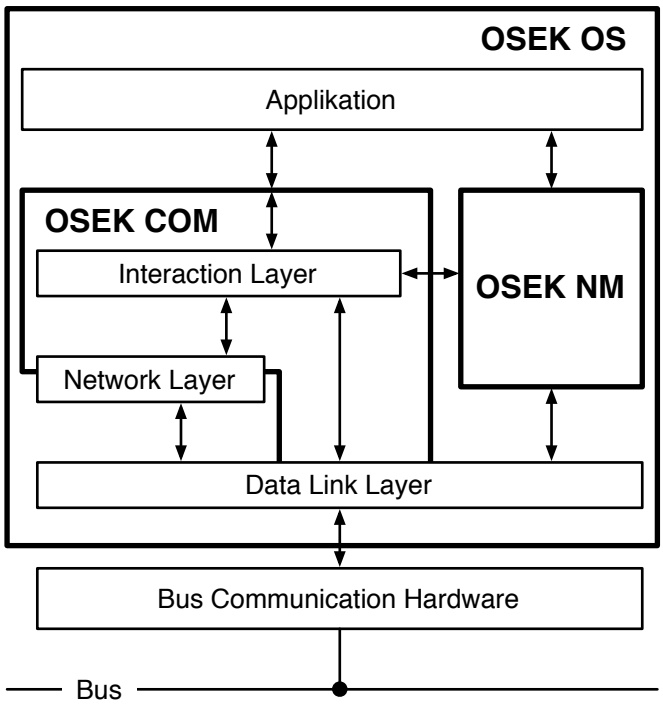


Abbildung 2.7: Übersicht über den OSEK/VDX-Standard (angelehnt an eine Abbildung aus der Spezifikation des Teilbereichs OSEK COM [116]).

Weiteren ist bei dem Standard darauf geachtet worden, dass die definierten Funktionen konfigurierbar und skalierbar sind. Dadurch wird eine optimale Auslegung der Software an die verwendete Hardware ermöglicht, so dass diese optimal ausgenutzt wird. Dies sorgt dann für eine Optimierung der Hardware-Kosten, was eine Anforderung und ebenfalls Herausforderung in Automotiven Systemen ist, wie es in Abschnitt 2.1 geschildert wurde.

Ein weiterer Vorteil, der bei der Entstehung des OSEK/VDX-Standard gesehen wurde, ist, dass es durch die Definition eines Software-Standards möglich ist, die funktionale Verifikation der Anwendungssoftware zuvor in ausgewählten Prototypen vorzunehmen, bevor die Software in der Serienentwicklung im endgültigen Produkt eingesetzt wird. Dadurch wird eine verkürzte Entwicklungszeit des Serienprodukts erreicht, was wiederum Kosten spart. Ebenfalls wird die Qualität der Software durch die vorherige Implementierung in den Prototypen gesteigert, da die Anwendungssoftware eine längere Absicherung erfährt, als wenn sie nur direkt für das finale Produkt entwickelt wird.

In Abbildung 2.7 ist eine Übersicht über den OSEK/VDX-Standard mit dessen drei Teilbereichen OSEK OS (engl. *OSEK/VDX operating system*), OSEK COM (engl. *OSEK/VDX communication*) und OSEK NM (engl. *OSEK/VDX network management*) zu sehen, die im Folgenden näher erläutert werden.

OSEK OS. Im Bereich der Betriebssysteme definiert der OSEK/VDX-Standard die grundlegenden Dienste, die der Anwendungssoftware bereitgestellt werden. Dar-

unter fällt die Definition der Tasks und deren Management, wie deren Aktivierung und der Task-Wechsel. Des Weiteren wird die Synchronisation der Tasks und der Zugriff auf die bereitgestellten, exklusiven Ressourcen definiert. Das Interrupt-Management sowie die Alarme, die sowohl absolut als auch relative Zeitangaben enthalten können, werden ebenfalls standardisiert. Für den Austausch von Daten zwischen den Modulen auf einem Prozessor wird die Intra-Prozessor-Kommunikation über Nachrichten angeboten. Abgerundet wird der Teilbereich OSEK OS mit der Unterstützung der Anwendungssoftware bei der Fehlerbehandlung [118].

OSEK COM. In dem Bereich der Kommunikation definiert der OSEK/VDX-Standard hauptsächlich den sogenannten *Interaction Layer*, der die Dienste für den Nachrichten-Transfer enthält. Dafür wird den Applikationsmodulen eine Schnittstelle für das Senden und Empfangen von Nachrichten angeboten. Dabei wird im *Interaction Layer* entschieden, ob die Nachricht intern auf einer ECU verteilt wird oder über den externen Bus gesendet werden muss. Des Weiteren enthält der Teilbereich OSEK COM die Anforderungen für die beiden darunter liegenden Schichten, den *Network Layer* und den *Data Link Layer*. Diese Schichten werden nicht mit einer Schnittstelle definiert, sondern es werden nur die minimalen Anforderungen, die der *Interaction Layer* benötigt, festgelegt. OSEK COM sieht für den *Network Layer* abhängig von dem jeweiligen Kommunikationsprotokoll, das auf dem Bus verwendet wird, beim Versenden von Daten-Paketen deren Segmentierung bzw. das Zusammensetzen der empfangenen Daten-Pakete und deren Bestätigung vor. Der *Data Link Layer* sorgt für den Transfer einzelner nicht-bestätigter Daten-Pakete auf dem Netzwerk. Ebenfalls definiert hier der OSEK COM nur die minimalen Anforderungen, die für den *Interaction Layer* notwendig sind [116].

OSEK NM. Für das Netzwerk-Management sorgt der Teilbereich OSEK NM, dessen Aufgabe die Sicherstellung einer zuverlässigen Kommunikation zwischen den verschiedenen ECUs im Fahrzeug-Netzwerk ist. Dafür werden die benötigten Dienste für das Initialisieren der Ressourcen zur Kommunikation auf einem Knoten wie z. B. der Netzwerk-Controller bereitgestellt. Des Weiteren sorgt das OSEK NM für das Aufstarten des Netzwerks und das Bereitstellen der Netzwerk-Konfiguration innerhalb eines Knotens. Der Teilbereich OSEK NM definiert verschiedene Mechanismen für das Überwachen der Netzwerk-Knoten, die ebenfalls im Teilbereich OSEK NM verwaltet werden. Der Teilbereich OSEK NM erkennt, bearbeitet und signalisiert die Netzwerk- und Knoten-Zustände. Ebenfalls können spezifischen Parameter des Netzwerk und des Knotens über den Teilbereich OSEK NM gelesen und gesetzt werden. Die Koordination der Netzwerk-weiten, globalen Operationsmodi steuert auch die OSEK NM in einer ECU. Zusätzlich unterstützt er die Diagnose des Netzwerk-Knotens und bietet den Applikationen Informationen über alle benötigten Knoten im Netzwerk [117] an.

Der OSEK/VDX-Standard dient als Grundlage für den AUTOSAR-Standard, auf den im nächsten Abschnitt eingegangen wird.

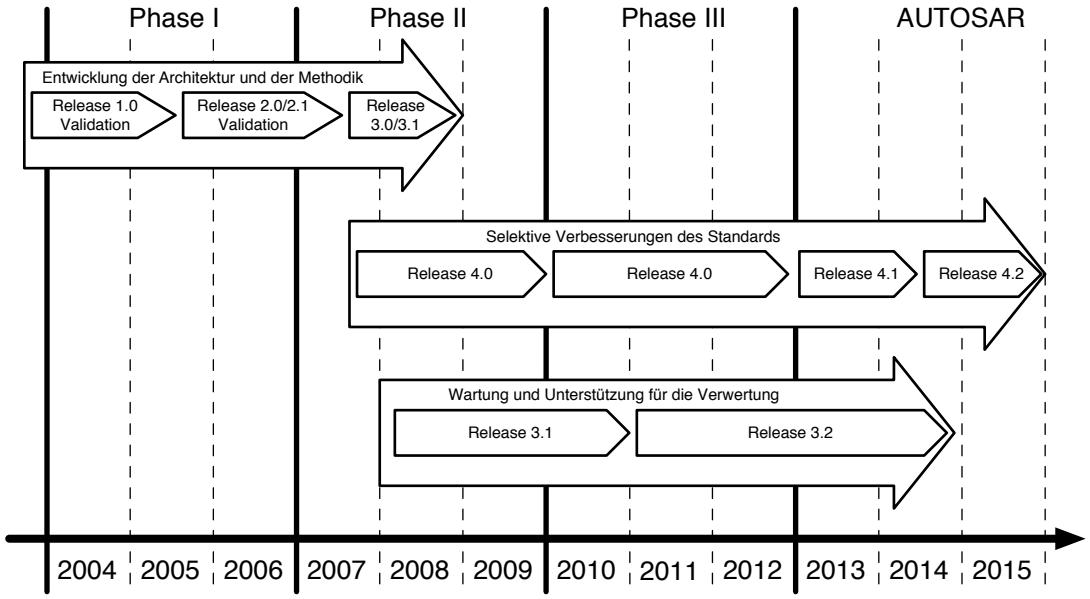


Abbildung 2.8: Zeitliche Entwicklung des AUTOSAR-Standards (angelehnt an eine Abbildung aus der Veröffentlichung von S. Fürst *et al.* [49] und einem Vortrag von D. Aiello [2]).

2.4.2 Der AUTOSAR-Standard

AUTOSAR ist eine offene und von der Industrie standardisierte Software-Architektur für Automotive Systeme. Sie wird von einem Verbund von Automobil-Herstellern, deren Zulieferern und Software-Werkzeugherstellern definiert. Dabei herrscht bei dieser Partnerschaft das Motto „Cooperate on standards, compete on implementation“ vor. Dies zeigt, dass das Interesse vorrangig daran liegt, einen gemeinsamen Software-Standard zu schaffen, für den jeder Partner seine eigenen Implementierungen erstellt, die im Anschluss gegenseitig konkurrieren.

Das Ziel des AUTOSAR-Standards ist es, dem stetig wachsenden Software-Anteil in den ECUs gerecht zu werden. Dazu gehört, dass die Performance und die Zuverlässigkeit der Software in den Steuergeräten gewährleistet bleibt. Zusätzlich bietet der Standard die Möglichkeit, einzelne Software-Komponenten in die ECU zu integrieren oder im Bedarfsfall in einem Automotiven System vorhandene Software-Komponenten durch andere Software-Komponenten zu ersetzen, die eine kompatible Schnittstelle besitzen.

Seit der Gründung von AUTOSAR im Jahr 2003 hat sich der Standard in drei Phasen stetig weiterentwickelt [49]. Jede Phase dauerte drei Jahren, in denen der Fokus auf unterschiedlichen Schwerpunkten lag. In Abbildung 2.8 sind die Phasen mit ihren Schwerpunkten graphisch dargestellt. In Phase I wurde hauptsächlich die Architektur und die Methodik von AUTOSAR entwickelt und in der Release 1.0 veröffentlicht. Mit Release 2.1 wurde Phase I Ende 2006 abgeschlossen. Release 3.0 und 3.1 wurden in Phase II fertig gestellt, in der immer noch an der Festigung des Standards im Bereich der Architektur und der Methodik gearbeitet wurde. Gleichzeitig lag in dieser Phase der

Schwerpunkt darauf, selektive Verbesserungen in den Standard einzubringen, die gegen Ende der Phase II mit Release 4.0 im Jahr 2009 veröffentlicht wurden. In Phase II und Phase III, die bis 2012 andauerte, legte man den Fokus auf die Pflege der älteren AUTOSAR-Releases und die Unterstützung bei der Integration und bei der Verwertung des AUTOSAR-Standards. Seit 2013 wird an der Fortführung von AUTOSAR gearbeitet, wobei entschieden wurde, nicht mehr in Phasen mit definierten Enddatum zu arbeiten sondern den AUTOSAR-Standard kontinuierlich zu erweitern. Aus diesem Grund wird der Begriff „Phase“ fallengelassen und der Zeitabschnitt ab 2013 wird nur noch mit AUTOSAR bezeichnet [25]. Anfang 2013 wurde der Release 4.1 und im Dezember 2014 der Release 4.2 veröffentlicht [2]. Seitdem wird weiter an Release 4.2 gearbeitet, der 2015 in der Version 4.2.2 verabschiedet wurde.

In AUTOSAR werden neben der Methodik zwei unterschiedliche Ebenen der automotiven Software definiert. Nach den Abstraktionsebenen, die in Abschnitt 2.2.2 vorgestellt wurden, definiert AUTOSAR eine logische und eine Software-Ebene. Für die logische Ebene führt AUTOSAR den sogenannten VFB (engl. *virtual functional bus*) ein, über den die SW-Cs (engl. *software component*) eines Automotiven Systems miteinander kommunizieren. In dieser Schicht wird weder die Partitionierung der SW-Cs auf ECUs festgelegt, noch wird die konkrete Hard- und Software betrachtet. In der Software-Ebene wird die Software-Architektur einer ECU spezifiziert. Für die Definition des VFBs und der Software-Architektur definiert AUTOSAR eine Methodik, in der Arbeitsschritte festgelegt werden, um aus den zwei eingeführten Ebenen ein komplettes Automotives System zu entwickeln. Ebenfalls werden die Arbeitsschritte definiert, um ausführbaren Code für jede AUTOSAR kompatible ECU zu generieren.

Zuerst wird die logische Architektur des VFBs und im Anschluss die ECU-Software-Architektur aus AUTOSAR vorgestellt, bevor im weiteren Abschnitt auf die in AUTOSAR definierten Methodik eingegangen wird.

Logische Architektur von AUTOSAR

In der logischen Architektur werden die SW-Cs des Automotiven Systems definiert und wie diese untereinander über den VFB kommunizieren [26]. Die Kommunikation zwischen den SW-Cs erfolgt über die sogenannten Ports. Die SW-Cs sind in dieser logischen Architektur noch keiner ECU zugeteilt und besitzen noch keine konkrete Implementierungen. Jedoch existieren im AUTOSAR-Standard unterschiedliche Arten von SW-Cs, von denen die wichtigsten in Tabelle 2.2 aufgelistet sind.

Den SW-Cs ist eine Interaktion untereinander nur über Ports erlaubt, über die die Kommunikation erfolgt. Jeder Port gehört dabei genau zu einer SW-C. Dabei gibt es zwei Arten von Ports: der PPort und der RPort. Der PPort bietet seine Funktionalität an (engl. *provide*) und der RPort benötigt eine Funktionalität (engl. *require*). Somit ist nur eine Kommunikation zwischen einem PPort und einem RPort möglich. Eine Verbindung zweier gleicher Ports ist nicht erlaubt. In Tabelle 2.3 sind die möglichen Port-Interfaces zusammengefasst. Jeder der Interfaces besitzt ein eigenes graphisches Symbol, das in den Architektur-Diagrammen verwendet wird. Neben den Port-Interfaces für das Sender-Empfänger Modell und das Client-Server Modell existieren noch weitere Port-Interfaces für den Austausch von Parameter-Werten, das Triggern von Ereignissen

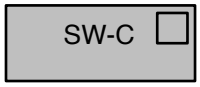
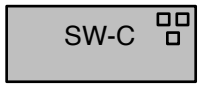
Art der SW-C	Symbol	Beschreibung
Sensor/Aktor-SW-C		Die Sensor/Aktor-SW-C enthalten die Implementierungen eines Sensors oder Aktors und sind ebenfalls atomare SW-Cs. Diese interagieren allerdings direkt mit der Hardware-Abstraktion einer ECU.
Atomare SW-C		Die atomaren SW-Cs lassen sich nicht in weitere SW-C aufteilen und enthalten die Applikationslogik.
Komposition-SW-C		Um den Abstraktionsgrad zu erhöhen und mehrere SW-Cs zu einer zusammenzufassen, stellt der AUTOSAR-Standard die Komposition-SW-C zur Verfügung. In dieser SW-C sind SW-Cs enthalten, die über Ports verbunden sind. Zusätzlich werden definierte Ports nach außen geleitet, die dann die Ports der Komposition-SW-C sind.

Tabelle 2.2: Übersicht über die SW-Cs in AUTOSAR.

oder das Verteilen einer Modus-Änderung, die im Automotiven System aufgetreten ist.

Für die Kommunikation zwischen den SW-Cs definiert AUTOSAR grundlegend zwei Kommunikationsmodelle. Das erste ist das Sender-Empfänger Modell (engl. *sender-receiver*) und das zweite das Client-Server Modell, die beide eigene Port-Interfaces besitzen.

Sender-Empfänger Modell. Das Sender-Empfänger Modell erlaubt es, Informationen auf eine asynchrone Weise zu verteilen. Der Sender sendet dabei Daten an einen oder mehrere Empfänger. Er wird dabei nicht blockiert und erhält keine Antworten von dem Empfänger. Er stellt nur die Informationen bereit und die Kommunikationsinfrastruktur ist für die Verteilung der Informationen an die Empfänger zuständig.

Client-Server Modell. Im Client-Server Modell bietet der Server einen Dienst (engl. *service*) an, an den sich ein oder mehrere Clients verbinden können. Der Client initiiert bei diesem Modell die Kommunikation und sendet dem Server eine Anfrage, bei der ebenfalls Parameter übergeben werden können. Der Server wartet auf die Anfragen der Clients und beantwortet diese. Der Client kann, während er auf die Antwort des Servers wartet, auf seinen Wunsch hin blockiert werden, was einer synchronen Kommunikation entspricht. Ebenfalls kann dieser nicht blockiert werden, womit eine asynchrone Kommunikation realisiert werden kann.

Für ein konkretes Automotives System werden nun die SW-Cs mit ihren Interfaces definiert und untereinander logisch über den VFB verbunden. Um dies näher zu

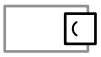
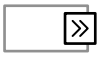
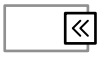
Art des Interfaces	Art des Ports	Symbol	Beschreibung
Sender-Receiver	PPort		Die SW-C stellt Werte von Daten-Elementen bereit.
	RPort		Die SW-C liest Werte von Daten-Elementen.
Client-Server	PPort		Die SW-C stellt die Funktionen bereit, die durch das Interface definiert sind.
	RPort		Die SW-C benötigt die Funktionen, die durch das Interface definiert sind.
Parameter Interface	PPort		Die SW-C stellt Parameter Daten bereit.
	RPort		Die SW-C liest Parameter Daten.
Trigger Interface	PPort		Die SW-C ist eine Quelle von Triggern.
	RPort		Die SW-C ist eine Senke von Triggern.
Mode Switch Interface	PPort		Die SW-C besitzt einen Mode Switch Manager.
	RPort		Die SW-C benutzt den Mode Switch.

Tabelle 2.3: Übersicht über die Port-Interfaces in AUTOSAR.

erläutern, ist in Abbildung 2.9 ein VFB mit fünf SW-Cs exemplarisch dargestellt. In diesem Beispiel sind die SW-C_1 und SW-C_5 Sensor/Aktor SW-Cs und die SW-C_2, SW-C_3 und SW-C_4 atomare SW-Cs. Die SW-C_1 besitzt zwei Ports – einen RPort des Client Server Interfaces und einen RPort des Sender-Empfänger Interfaces. Beide Ports sind entsprechend mit den PPorts der SW-C_2 verbunden. Die SW-C_2 besitzt zusätzlich noch zwei RPorts des Sender-Empfänger Interfaces. Einer der Ports ist mit dem PPort der SW-C_5 verbunden, die ihren Daten-Wert ebenfalls an den RPort der SW-C_3 sendet. Der zweite RPort der SW-C_2 erhält seine Daten von dem PPort der SW-C_3. Die restlichen Verbindungen lassen sich analog der vorherigen Beispiele ablesen.

Auf diese Weise werden SW-Cs in AUTOSAR und deren Kommunikation festgelegt. An dieser Stelle ist noch keine Zuordnung der SW-Cs zu ECUs erfolgt. Dies wird über mehrere Arbeitsschritte erreicht, deren Methodik der AUTOSAR-Standard ebenfalls vorgibt. Dabei ist die Generierung der RTE (engl. *runtime environment*), die für jede ECU unterschiedlich ist, der zentraler Bestandteil. Dafür wird der sogenannte RTE-Generator eingesetzt, der aus der Beschreibung der ECU-Konfigurationen die Realisie-

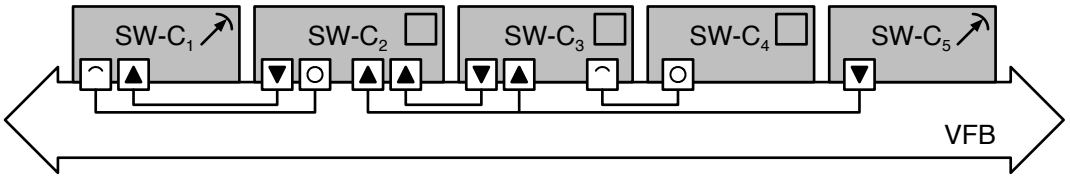


Abbildung 2.9: Der VFB in AUTOSAR mit fünf SW-Cs.

rung des VFBs für eine konkrete ECU erzeugt. Somit stellt die RTE die Realisierung der Interfaces dar, die im VFB für die Applikationen einer ECU definiert sind. Die RTE verbindet die SW-Cs sowohl untereinander als auch die SW-Cs mit der sogenannten BSW (engl. *basic software*). Dabei leitet die RTE die Nachrichten direkt zu den SW-Cs, die sich auf der selben ECU befinden. Falls die SW-C einer anderen ECU zugeteilt ist, werden die Nachrichten durch die BSW geleitet und über das Bus-System an die entsprechende ECU versandt. Die BSW wird ebenfalls in AUTOSAR spezifiziert und wird im nächsten Abschnitt über die Software-Architektur von AUTOSAR vorgestellt. Neben der Realisierung der Kommunikation zwischen den SW-Cs übernimmt die RTE noch die zweite wichtige Aufgabe des Schedulings von SW-Cs [24].

Dies wird ebenfalls an dem zuvor eingeführten Beispiel erläutert. Eine mögliche Zuteilung der SW-Cs ist in Abbildung 2.10 dargestellt. Die SW-C_1 und die SW-C_2 sind auf die ECU_1 und die SW-C_3 und SW-C_5 auf die ECU_2 zugeteilt worden. An dieser Verteilung der SW-Cs ist ersichtlich, dass die Kommunikation zwischen SW-C_1 und SW-C_2 auf der selben ECU in der RTE_1 stattfindet. Dies gilt ebenfalls für die Kommunikation zwischen SW-C_3 und SW-C_5 auf ECU_2 in der RTE_2. Die Kommunikation zwischen SW-C_2 und SW-C_3 bzw. SW-C_5 erfolgt über die ECU-Grenzen hinweg über den physikalischen Bus des Systems.

Wie bereits Abbildung 2.10 zeigt, besteht die Software auf einer ECU in AUTOSAR aus mehreren Schichten, die nun in der Software-Architektur vorgestellt werden.

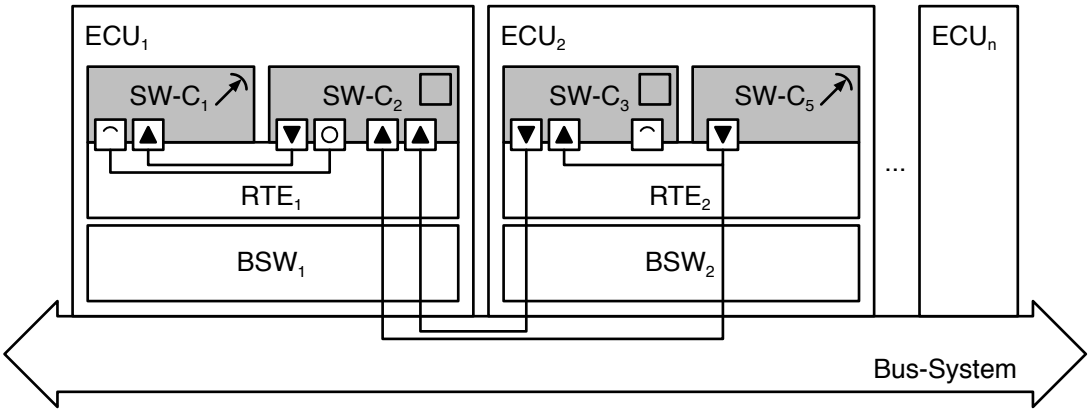


Abbildung 2.10: Generierung der RTE aus dem VFB in AUTOSAR.

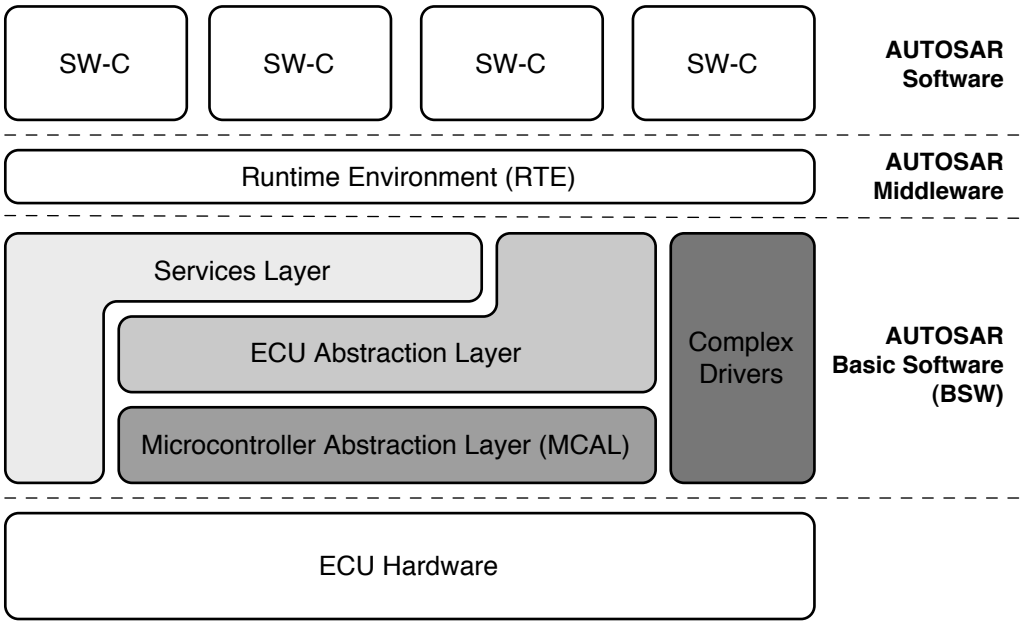


Abbildung 2.11: ECU-Software-Architektur in AUTOSAR (angelehnt an eine Abbildung aus deren Spezifikation [11]).

ECU-Software-Architektur in AUTOSAR

Neben der Definition des VFBs als logische Architektur besitzt der AUTOSAR-Standard ebenfalls eine geschichtete ECU-Software-Architektur [11]. Eine Übersicht ist in Abbildung 2.11 dargestellt. In der AUTOSAR Software-Schicht laufen die SW-Cs, die in der logischen Architektur definiert sind und dort über den VFB miteinander kommunizieren. Da der VFB zusammen mit den SW-Cs Hardware- und Software-unabhängig definiert werden, müssen unterhalb der AUTOSAR Software-Schicht zwei weitere Software-Schichten eingezogen werden. Diese erlauben es den Infrastruktur-unabhängigen SW-Cs, dass sie auf einer ECU ausgeführt werden können. Die erste Schicht ist die AUTOSAR Middleware, die nur aus der RTE besteht [24]. Die RTE wird mit Hilfe der AUTOSAR Software-Werkzeugen aus der logischen Beschreibung des VFB, der SW-Cs und der ECUs automatisch für jede einzelne ECU generiert. Als zweite Infrastruktur-abhängige Schicht liegt zwischen der RTE und der Hardware die AUTOSAR BSW, die direkt auf der Hardware/Software-Schnittstelle einer ECU aufsetzt.

Die BSW ist selbst wiederum in mehrere Software-Schichten unterteilt, um mehrere Abstraktionsebenen der Hardware in der BSW zu ermöglichen. Die Module, die in der BSW enthalten sind, und deren Schnittstellen werden ebenfalls in dem AUTOSAR-Standard definiert. Diese lassen sich in folgende vier Abstraktionsebenen einteilen.

MCAL (engl. microcontroller abstraction layer). Die MCAL setzt direkt auf der Mikrocontroller-Hardware auf und hat das Ziel, den darüber liegenden Schichten die Mikrocontroller-Hardware zu abstrahieren und diesen über Treiber einen standardisierten Zugriff auf den Mikrocontroller zu bieten.

ECU Abstraction Layer. Um einen ECU unabhängigen Zugriff auf Hardware-Komponenten zu erhalten, wird der *ECU Abstraction Layer* definiert. Dieser stellt für die höheren Software-Schichten Schnittstellen bereit, die sowohl einen Zugriff auf die Mikrocontroller internen als auch auf die Mikrocontroller externen Hardware-Komponenten erlaubt, die auf einer ECU neben dem Mikrocontroller verbaut sind. Darunter können u. a. Kommunikationscontroller oder ADCs fallen. Somit enthält diese Schicht Treiber für die Hardware-Komponenten, die nicht im Mikrocontroller enthalten, sondern an den Mikrocontroller angeschlossen sind.

Complex Drivers. Die Schicht *Complex Drivers* erlaubt es in AUTOSAR, spezielle Funktionen in die BSW zu integrieren, die nicht im AUTOSAR-Standard spezifiziert werden. Ebenfalls ist diese Schicht für die Migration älterer, nicht zum AUTOSAR-Standard kompatiblen Software oder für sehr Zeit-kritische Software-Funktionen geeignet.

Services Layer. Der *Services Layer* ist die höchste Schicht in der BSW und enthält die grundlegenden Betriebssystem-Dienste, auf die die SW-Cs zugreifen können. Nur der *ECU Abstraction Layer* stellt den Zugriff auf die I/O-Funktionen bereit. Somit enthält der *Services Layer* unter anderem Funktionen für die Fahrzeug-Buskommunikation, Speicherfunktionen, Diagnose-Funktionen sowie das ECU-Zustands- und das Mode-Management.

Abgesehen von den *Complex Drivers* lassen sich die drei übrigen BSW Schichten in vier verschiedenen Service-Bereiche einteilen: die System Services, die Memory Services, die Communication Services und die I/O Services. Dabei ziehen sich die Bereiche durch alle drei Schichten der BSW, wie es in Abbildung 2.12 dargestellt ist. Die MCAL enthält hauptsächlich die Treiber für die Mikrocontroller-Komponenten, für die Speicher-Controller, für die im Mikrocontroller enthaltene Kommunikationscontroller und für die Ein- und Ausgabe. Aufbauend auf diesen Treibern enthält die *ECU Abstraction Layer* die Abstraktionen der ECU-Hardware für die vier Service-Bereiche. Der *Service Layer* bietet den SW-Cs die Schnittstelle für die vier Bereiche Hardware-unabhängig an.

Jede der BSW-Schichten enthält Module, die in der AUTOSAR-Spezifikation „List of Basic Software Modules“ übersichtlich aufgelistet sind [12]. Jedes der Module wird einem BSW-Bereich in einer Schicht zugeordnet und besitzt eine eigene AUTOSAR-Spezifikation, in der die Schnittstellen der Module definiert sind.

Anhand der CAN-Bus Kommunikation soll die Schichtung der BSW erläutert werden. Es gibt sechs BSW Module, die für die CAN-Bus Kommunikation zuständig sind. Auf der MCAL Ebene im Bereich der *Communication Drivers* ist das Modul *CAN Driver* definiert, das den Zugriff auf die CAN-Bus-Controller Hardware bereitstellt [15]. Das Interface ist dabei Controller-unabhängig für die darüber liegende Schicht definiert. In AUTOSAR hat nur das Modul *CAN Interface* in der Ebene *ECU Abstraction Layer* im Bereich der *Communication HW Abstraction* Zugriff auf das Modul *CAN Driver* [16]. Das Modul *Can Interface* abstrahiert unterschiedliche CAN-Bus Hardware-Konfigurationen, die hauptsächlich aus den CAN-Bus Controller und CAN-Bus Transceivers besteht. Ebenfalls auf der selben Ebene und im selben Bereich definiert der

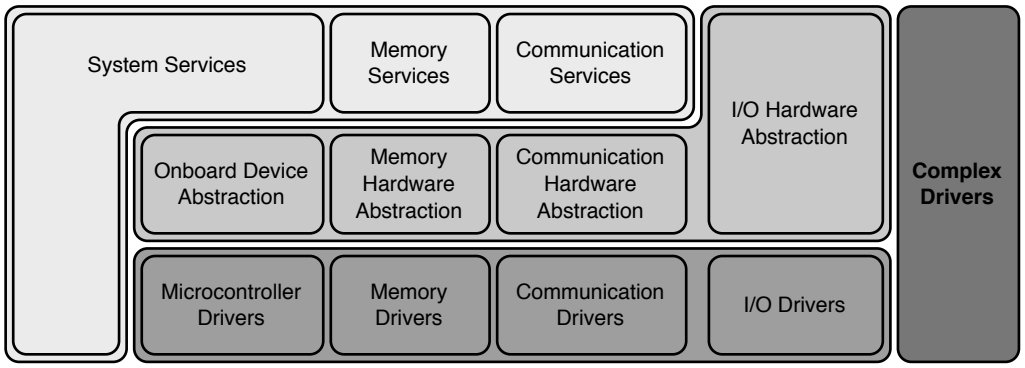


Abbildung 2.12: AUTOSAR-BSW mit ihren Service-Bereichen (angelehnt an eine Abbildung aus deren Spezifikation [11]).

AUTOSAR-Standard das Modul *CAN Tranceiver Driver* [19]. Dieser stellt für die CAN-Bus Tranciever eine abstrakte Schnittstelle bereit, die ebenfalls vom Modul *CAN Interface* für dessen Dienste genutzt wird.

Aufbauend auf dem Modul *CAN Interface* sind in der BSW Schicht *Service Layer* im Bereich der *Communication Services* die Module *CAN Transport Layer*, *CAN Network Management* und *CAN State Manager* definiert. Für das Segmentieren im Fall des Versendens von CAN-Bus-Nachrichten und für das Zusammensetzen im Fall des Empfangens von CAN-Bus-Nachrichten ist der *CAN Transport Layer* verantwortlich [20]. Daneben kümmert er sich um die Kontrolle des Datenflusses, das Erkennen von Fehlern in der Segmentierung sowie um das Abbrechen eines Empfangs- oder Sende-Vorgangs. Das Modul *CAN Network Management* ist für das Netzwerk-Management auf dem CAN-Bus zuständig [17]. Darunter fällt die Koordination des Wechsels zwischen den Bus-Schlaf-Modi und dem normalen Operationsmodus, das Entdecken aller anderer aktiver Netzwerk-Knoten und die Erkennung, ob die anderen Netzwerk-Knoten für einen Wechsel in den Bus-Schlaf-Modus bereit sind. Dieses Modul stellt dabei das Bindeglied zwischen dem allgemeinen AUTOSAR Netzwerk-Management Modul, das in der Schicht *Service Layer* im Bereich der *Communication Services* definiert ist [22], und dem Modul *CAN Interface* dar. Zuletzt ist das Modul *CAN State Manager* für das Zustandsmanagement des CAN-Bus verantwortlich [18]. D. h. das Modul sorgt für die Benachrichtigung der AUTOSAR-Module im Fall einer Zustandsänderungen des CAN-Bus und für das Behandeln der Zustandsänderungen auf dem CAN-Bus.

Dieses Beispiel für die Kommunikation über den CAN-Bus zeigt, wie in AUTOSAR die Funktionalität in Module aufgeteilt wird und wie die Module den BSW-Schichten zugeordnet werden. Analog wird dies für andere Bus-Systeme wie den FlexRay oder den LIN-Bus in AUTOSAR gehandhabt. Jedoch sind in der AUTOSAR Schicht *System Service* zusätzlich drei weitere Module hervorzuheben, die unterschiedliche Aufgaben neben der Kommunikation ausführen. Das Modul *OS* enthält die Spezifikation der grundlegenden Betriebssystem-Operationen [23]. Hierfür wird als Grundlage der OSEK/VDX-Standard herangezogen, der in Abschnitt 2.4.1 vorgestellt wurde. Zusätzlich enthält das AUTOSAR Modul *OS* Erweiterungen, die noch nicht durch den OSEK/VDX abge-

deckt sind. Für den ECU Zustand und deren Wechsel ist das BSW Modul *ECU State Manager* verantwortlich [21]. Darunter fallen die ECU Power-Zustände wie Sleep oder Shutdown. Es sorgt für Initialisierung des OS, die Konfiguration der ECU bezüglich der Power-Zustände und die Verwaltung der Weck-Signale. Dagegen sorgt das BSW Modul *BSW Mode Manager* für die Verwaltung der Fahrzeug- und Applikationszustände [14] innerhalb der BSW. Darunter fällt die Arbitrierung der Anfragen seitens der SW-Cs nach einem bestimmten Modus. Dies wird über einfache Regeln erreicht und führt zu Aktionen, die ebenfalls in diesen Regeln festgelegt sind.

Über die beschriebene Schichtung in AUTOSAR und durch die Definition der BSW Modul-Schnittstellen wird es ermöglicht, dass eine Wiederverwendung der Hardware-unabhängigen BSW-Module auf verschiedenen Hardware-Platformen möglich ist. Dadurch wird ebenfalls erreicht, dass die RTE und die darauf laufenden SW-Cs unabhängig von der Infrastruktur definiert und ausgeführt werden können.

Methodik-Definitionen in AUTOSAR

Im AUTOSAR-Standard wird ebenfalls die Methodik definiert, die für eine komplette Software-Entwicklung notwendig ist. Dafür werden mehrere Teilschritte festgelegt, die sich in mehrere Teildomänen aufteilen lassen [13]. An dieser Stelle wird die AUTOSAR-Methodik nur in einer groben Übersicht beschrieben.

Definition des VFB. Ziel dieser Teildomäne ist es, eine Beschreibung des VFBs zu erarbeiten. Darunter fällt die Definition des Datenmodells, der Komponenten und des Timing-Modells für den VFB. Die Spezifikation der SW-Cs erfolgt über die Aufteilung der zu erbringenden Fahrzeug-Funktionen auf verschiedene SW-Cs. Bei der Definition des Datenmodells werden die Datentypen, die Schnittstellen und die Modi des VFBs festgelegt.

Entwicklung der SW-Cs. In diesem Teilbereich wird das Software-Design der verschiedenen atomaren SW-Cs festgelegt. Darunter fällt die Definition der Runnables, der Events und der Variablen, die zwischen den Runnables gemeinsam verwendet werden. Nach dem Design der SW-C wird die Beschreibung generiert, nach der die SW-Cs implementiert werden. Allerdings umfasst dieser Teilbereich weder die Art der Implementierung noch die Qualitätsanforderungen der SW-Cs. Auch das Testen der SW-Cs ist in dieser Teildomäne enthalten. In diesem Teilbereich steht am Ende der Arbeitsschritte die Ermittlung der Ressourcen-Anforderungen einer SW-C, die ebenfalls in der Beschreibung der SW-Cs enthalten ist.

Entwicklung des Systems und der Teilsysteme. Das Ziel dieser Teildomäne ist die Erstellung der System-Beschreibung. Dies schließt die Definition der System-Topologie mit ein, wie auch die Definition der ECU-Beschreibungen sowie die Kommunikationsverbindungen zwischen den einzelnen ECUs. In diesem Schritt kann auch die Festlegung der konkreten Implementierung für eine SW-C enthalten sein. Ein weiterer elementarer Arbeitsschritt ist die Verteilung der SW-C auf die ECUs in dem System unter Beachtung der geltenden Randbedingungen sowie der Kommunikationsbandbreiten auf den Bus-Systemen.

Entwicklung der BSW. In dieser Domäne werden die Arbeitsschritte für die Entwicklung der BSW Module definiert. Darunter fallen die Schritte für die Definition der Daten-Typen aber auch die Definition der Schnittstellen eines BSW-Moduls.

Software-Integration auf einer ECU. In diesem Teilbereich erhält jede ECU ihre Konfiguration und die ausführbaren Binärdateien. Dafür wird zuerst aus der System-Beschreibung die Konfiguration der ECU vorbereitet, um daraus die BSW und die RTE für die ECU zu konfigurieren und im Anschluss zu generieren. Am Ende werden die ausführbaren Binärdateien für die ECU gebaut, die dann auf die ECU gespielt werden.

Für die Definition der Arbeitsschritte wird in AUTOSAR ein Domänen-übergreifendes Meta-Modell festgelegt, nach dem die einzelnen Arbeitsschritte beschrieben werden. Dafür werden folgende fünf Elemente in der AUTOSAR-Methodik eingeführt [13].

Arbeitsschritt. Das zentrale Element in der AUTOSAR-Methodik ist der Arbeitsschritt, der von einer Rolle ausgeführt wird. Neben der auszuführenden Rolle kann es noch eine unterstützende Rolle geben oder der Arbeitsschritt wird mit Hilfe eines Werkzeugs erledigt. Die Gesamtdauer eines Arbeitsschritts kann von wenigen Stunden bis zu einigen Tagen dauern. Jeder Arbeitsschritt hat notwendige und optionale Arbeitsprodukte als Eingang und erzeugt Arbeitsprodukte als Ausgang.

Arbeitsprodukte. Die Arbeitsprodukte werden in einem Arbeitsschritt erzeugt, konsumiert oder verändert. In der AUTOSAR-Methodik können Arbeitsprodukte u. a. XML Dateien, Source Code, Binärdateien oder Text sein.

Rollen. Eine Rollendefinition legt die Verantwortung, die Fähigkeiten, die Kompetenz und die Qualifikation einer Person oder einer Personengruppe fest, um einen Arbeitsschritt auszuführen. Dabei ist es auch möglich, dass eine Rolle von mehreren Personen übernommen wird. Ebenfalls ist es nicht ausgeschlossen, dass eine Person mehrere verschiedene Rollen einnimmt.

Werkzeuge. Die Werkzeuge unterstützen die Rollen bei der Ausführung der Arbeitsschritte. Die Definition des Werkzeugs beschreibt die Eigenschaften der CASE (engl. *computer-aided software engineering*) Werkzeuge. Dabei können die Werkzeuge nützlich, empfohlen oder notwendig für einen Arbeitsschritt sein.

Beratung. Für die Arbeitsprodukte, die Rollen und die Werkzeuge kann es noch beratende Elemente wie Dokumentation, Werkzeug-Mentoren oder Veröffentlichungen geben.

Diese im AUTOSAR-Standard genau definierte Methodik erlaubt es, dass die OEMs, die Tier 1 und die weiteren Zulieferer eng zusammenarbeiten und nach genau definierten Arbeitsschritten das Automotive System entwickeln können.

2.4.3 Die GENIVI-Allianz

In den letzten Jahren hat sich Linux auch in den Automotiven Systemen etabliert. Vor allem in IVI-Systemen (engl. *in-vehicle infotainment*), das auf der Head-Unit läuft, kann OSS (engl. *open-source-software*) eingesetzt werden. Dazu wurden die ersten Untersuchungen 2006 gestartet, die in einem zweiten Schritt mit Hilfe eines Pilot-Projekts validiert wurden. Um die kritische Masse für einen Erfolg zu erreichen und den Einstieg in den Markt zu schaffen, wurde 2009 die GENIVI-Allianz gegründet [138]. Im Folgenden wird nun die Historie der GENIVI-Allianz vorgestellt, wobei auch der Einsatz der OSS in Automotiven Systemen weiter beleuchtet wird.

In der ersten Phase der Untersuchung wurde sehr schnell festgestellt, dass in der Software für IVI-Systeme nicht nur Betriebssystem-Software vorhanden ist, sondern auch ein großer Anteil an Middleware-Software, die nicht vom Betriebssystem-Zulieferer, sondern vom Tier 1 bereitgestellt wird. Dabei unterscheidet sich die Middleware-Software der verschiedenen Tier 1 nicht in der Funktionalität, sondern in der Qualität der Software. Ebenfalls stellte sich heraus, dass der Prozentsatz der Middleware-Software im Vergleich zu der Betriebssystem-Software von der Head-Unit im Einsteiger-Bereich bis hin zu der high-end Head-Unit immer mehr ansteigt.

Aus diesen Erkenntnissen wurde die Entscheidung getroffen, die Entwicklung eines IVI-Systems mit einem Betriebssystem zu beginnen, das bereits in der Welt der Unterhaltungselektronik eingesetzt wird und ebenfalls in der Größe skalieren kann. Denn es schien leichter zu sein, ein solches Betriebssystem um die automotiven Anteile zu erweitern, als den umgekehrten Ansatz zu verfolgen, in ein automotives Betriebssystem die Anteile für ein Infotainment-System zu integrieren.

Während dieser Zeit setzte sich Linux und OSS in der Welt der Smartphones und Netbooks durch und wurde als Vorbild für die Entwicklung eines IVI-Systems gesehen. Der Ansatz des Open-Source-Gedankens, das der Source Code für alle Beteiligten zur Verfügung steht und für den jeweiligen Einsatz-Zweck angepasst werden kann, erschien allen Teilnehmer der Untersuchung als ein Vorteil. Jedoch stellte sich die Frage, ob sich dieses Arbeitsmodell auch in der automotiven Industrie durchsetzen kann, in der wegen den Zulieferer-Ketten andere Regeln als in der Industrie der Unterhaltungselektronik herrschen.

Außerdem erhoffte man sich, dass durch die Produkte in der Unterhaltungselektronik, die auf OSS basieren, wie Set-Top Boxen, Media Player, Navigationsgeräte oder Mobiltelefone, der Grundstein für die Entwicklung einer automotiven Head-Unit gelegt ist. Um dies zu validieren, wurde nach der ersten Phase der Untersuchung ein Pilot-Projekt gestartet, bei dem die Realisierung einer Head-Unit auf der Basis von OSS überprüft wird. An diesem Projekt waren über sechs Monate durchschnittlich 40 Entwickler beteiligt. Dabei nahm BMW die Rolle des OEMs, Magneti Marelli die Rolle des Tier 1, Wind River die Rolle des Betriebssystem- und Middleware-Lieferanten und Intel die Rolle des Hardware-Lieferanten ein. Dieses Pilot-Projekt zeigte, dass es möglich ist, eine Head-Unit mit Hilfe von OSS zu entwickeln [138]. Allerdings wurde auch erkannt, dass für eine vollständige Realisierung einer Head-Unit, die zu einem großen Anteil aus OSS besteht, die Gruppe der Entwickler vergrößert werden musste.

Aus diesen Gründen ist die GENIVI-Allianz im Anschluss an das Pilot-Projekt 2009

gegründet worden. Dort sind die OEMs, die Tier 1, die Betriebssystem- und Middleware-Zulieferer als auch die Hardware-Hersteller vertreten. Neben den Gründungsmitgliedern traten weitere Partner bei, so dass die GENIVI-Allianz im April 2012 aus ca. 160 Mitglieder bestand, deren Zahl sich bis 2015 auf diesem Niveau gehalten hat [55]. Ziel der Allianz ist es, eine Spezifikation einer IVI-Plattform zu definieren, die eine Schnittstelle der Middleware für die darauf laufenden Applikationen darstellt und die Anforderungen der Automotiven Systeme erfüllt. Dabei wurde darauf geachtet, dass die Spezifikation die Kompatibilität zwischen den möglichen Middleware-Implementierungen garantiert, so dass diese im Anschluss gegenseitig konkurrieren. Somit stehen für eine konkrete Produktentwicklung eine Vielzahl von Middleware-Implementierungen zur Auswahl, womit sich die Vorteile eines Wettbewerbs ergeben.

Durch die Existenz verschiedener IVI-Plattformen mit definierten Schnittstellen ergibt sich ein verkürzter Zeitraum bis zur Marktreife eines Head-Unit Projekts, da eine funktionierende Middleware bereits vorhanden ist und nur die Applikationen und Dienste implementiert werden müssen.

Zusätzlich ergibt sich für die Entwickler der Applikationen und Dienste der Vorteil, dass mit einer definierten Middleware-Schnittstelle sich die Entwicklungskosten für die Applikationen und Diensten reduzieren, da sich ein größerer Absatzmarkt für die Applikationen und Diensten bei einer einheitlichen Schnittstelle der Middleware ergibt.

In einer Allianz wird ebenfalls die Erhöhung des Innovationsgrads gesehen, da eine gemeinsame Beobachtung des Head-Unit Marktes durch die verschiedenen Allianz-Mitglieder, die aufgrund ihrer Tätigkeit eine unterschiedliche Wahrnehmung des Marktes haben, erfolgen kann und dadurch die Anzahl an Kundenfunktionen in der Head-Unit sich durch Innovationen erhöht.

In Abbildung 2.13 ist das GENIVI-Referenz-System abgebildet. Dies setzt sich aus der Hardware, der GENIVI-Plattform, den GENIVI-Referenz-Applikationen und der Referenz-HMI (engl. *human machine interface*) zusammen. Die GENIVI-Allianz verfolgt die Ansätze Adopt, Adapt und Create, die im Folgenden verdeutlicht werden. Die GENIVI-Plattform besteht zum einen aus dem Community Code, der Open Source zur Verfügung steht. Dieser wird unverändert in der GENIVI-Plattform verwendet (engl. *adopt*) und besteht aus ca. 80 % der GENIVI-Plattform. Die weiteren 20 % der Plattform setzen sich aus dem GENIVI-Code zusammen, der in der Allianz entwickelt wird. Hier wird dabei noch einmal zwischen den Erweiterungen der OSS (engl. *adapt*) und dem automotive-spezifischen Code-Anteil (engl. *create*) unterschieden. Die Erweiterungen der OSS machen ca. 15 % der GENIVI-Plattform aus, die nach der Anpassung an die Anforderungen der Automotiven Systemen an die Maintainer der Open-Source-Projekte übergeben werden. Hierbei wird das Ziel verfolgt, dass die Optimierungen und Anpassungen für die Automotiven Systeme in den Mainstream Open Source Code aufgenommen werden. Die restlichen 5 % der GENIVI-Plattform sind spezieller Code für die Automotiven Systeme, der in eigenständigen Projekten in der GENIVI-Allianz entwickelt wird. Darunter fällt z. B. die Software für das DLT (engl. *diagnostic log and trace*) oder das IVI-Layer-Management.

Dabei stellt die GENIVI-Plattform keine Linux-Distribution dar, in der spezifische Software-Komponenten für ein vollständiges IVI-System bereitgestellt werden. Sondern es wird in der GENIVI-Spezifikation festgelegt, welche Software-Komponenten

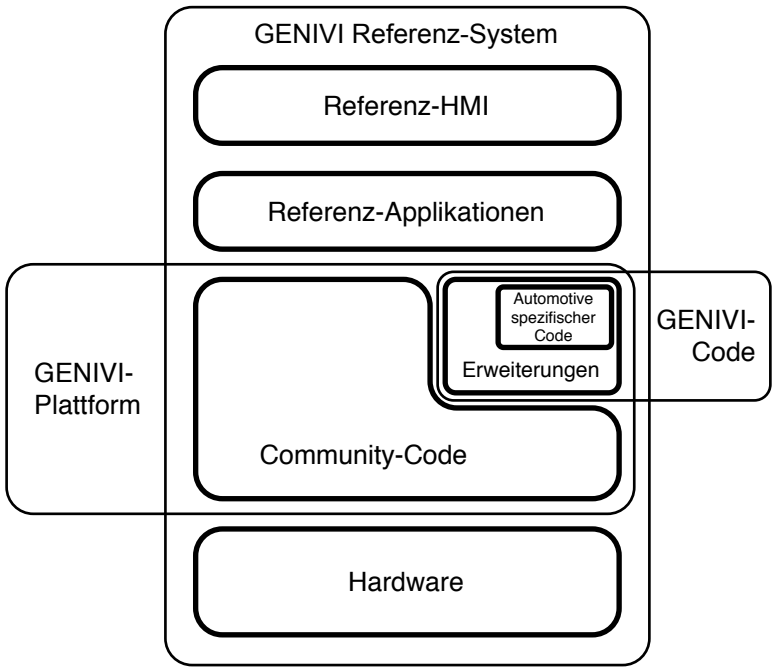


Abbildung 2.13: Übersicht über die GENIVI-Plattform (angelehnt an eine Abbildung aus der Veröffentlichung von G. Smethurst [138]).

in der GENIVI-Plattform enthalten sein müssen. Dabei unterscheidet die GENIVI-Spezifikation zwischen drei unterschiedlichen Kategorien von Software-Komponenten:

Platzhalter-Komponente. Wenn nur die Anforderungen an eine Software-Komponente definiert sind, spricht die GENIVI-Spezifikation von einer Platzhalter-Komponente (engl. *placeholder component*). In einer GENIVI-konformen Plattform muss für jede Platzhalter-Komponente eine Software-Komponente enthalten sein, die die in der GENIVI-Spezifikation definierten Anforderungen erfüllt.

Abstrakte Komponente. Das Verhalten einer abstrakten Software-Komponente (engl. *abstract component*) ist durch eine Schnittstellen-Beschreibung in der GENIVI-Spezifikation definiert. Falls eine Plattform GENIVI-konform ist, existiert in der Plattform eine Software-Komponente, die die spezifizierte Schnittstelle und das Verhalten erfüllt.

Spezifische Komponente. Falls eine Software-Komponente als spezifische Komponente (engl. *specific component*) definiert ist, muss in der GENIVI-konformen Plattform genau die spezifizierte Software-Komponente in der angegebenen Version enthalten sein. Hierbei handelt es sich entweder um OSS oder um eine Software-Komponente, die in der GENIVI-Allianz als Projekt entwickelt wurde.

Die Allianz hat die in Abbildung 2.14 dargestellte Organisation. Dabei unterteilt sich die Organisation in folgende Gruppen, die unterschiedliche Aufgaben haben. Der Allianz steht das *GENIVI Board of Directors* vor, in dem sowohl die OEMs, die Tier 1,

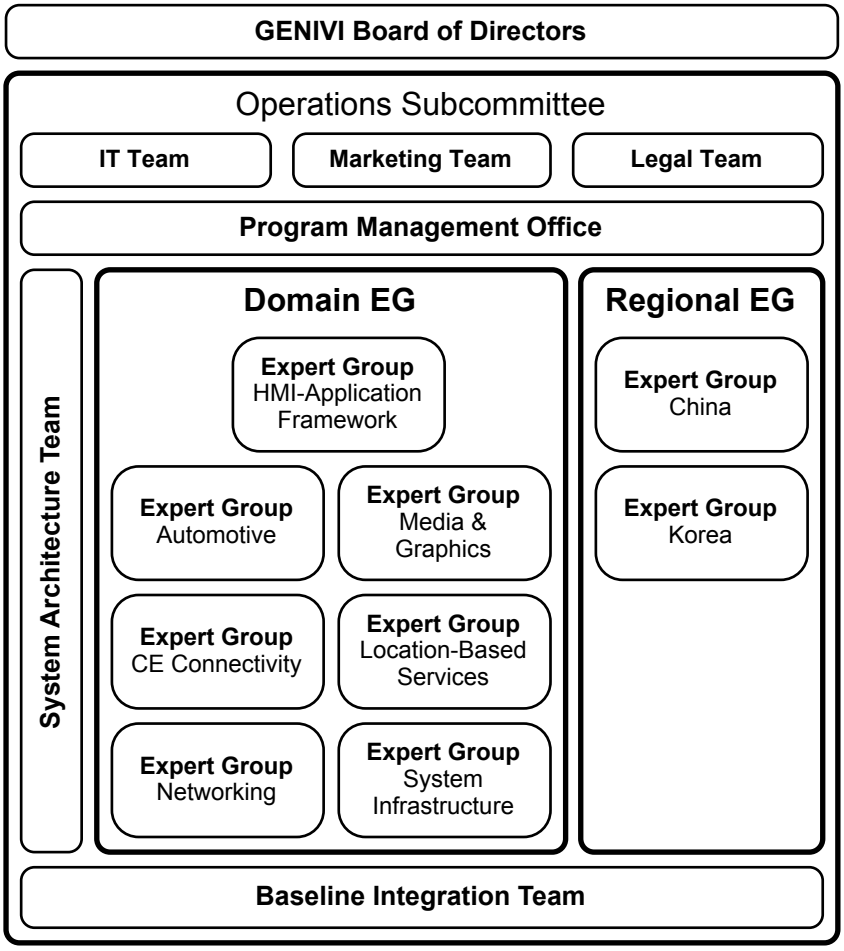


Abbildung 2.14: Organisation der GENIVI-Allianz.

die Betriebssystem- und Middleware-Zulieferer als auch die Hardware-Lieferanten vertreten sind. Das *Operations Subcommittee* untersteht direkt dem *Board of Directors* und unterstützt es in deren Aufgaben. Das *IT Team* ist für die Bereitstellung der IT-Infrastruktur der GENIVI-Allianz zuständig, wie eine Web-Server oder den Email-Verteiler für den Informationsaustausch der einzelnen Gruppen in der Allianz. Zusätzlich wird das *Operations Subcommittee* durch ein *Marketing Team* und einem *Legal Team* unterstützt, das jeweils Aufgaben im Marketing-Bereich oder im rechtlichen Umfeld wahrnimmt.

Nachdem kurz die Gruppen für den operativen Teil vorgestellt wurden, werden die Teams für die Standardisierung der GENIVI-Spezifikation und die Bereitstellung der Referenz-Plattform erklärt. Das PMO (engl. *program management office*) übernimmt hierbei das Projekt-Management, wobei die einzelnen EGs (engl. *expert group*) dem PMO ihren Status berichten. Die einzelnen EGs sind für den Standardisierungsprozess in ihrem jeweiligen Themengebiet verantwortlich. In der GENIVI-Allianz gibt es sieben Domain EGs und zwei regionale EGs, die für jeweils eine Sprachregion zuständig sind. Da-

bei unterteilen sich die Domain EG in folgende Gruppen mit ihrem Zuständigkeitsbereichen.

HMI-Application Framework. In der EG *HMI-Application Framework* wird ein gemeinsames Framework für die Applikationen auf Benutzer-Ebene definiert. Darunter zählt u. a. das Verwalten und dynamische Laden der Applikationen sowie auch ein Framework für die Sprach-Eingaben.

Automotive EG. In den Zuständigkeitsbereich dieser EG liegt die Diagnose Automotiver Systeme, die Fahrzeug-Schnittstellen, die spezifischen Fahrzeug-Netzwerke, sowie das spezifische Software-Management in Automotiven Systemen.

Für die Diagnose kommen für das Fahrzeug angepasste Programme wie das DLT zum Einsatz, die in dieser EG entwickelt werden. Ebenfalls wird hier die Inter-Node-Communication spezifiziert, die die Kommunikation zu den anderen ECUs festlegt. Hier werden ebenfalls die spezifischen Bus-Systeme, wie CAN-Bus und MOST-Bus betrachtet, die bereits in Abschnitt 2.3.2 näher vorgestellt wurden. In diesen Bereich fällt auch die Anbindung von GENIVI an den AUTOSAR-Standard.

Ein weiterer Bereich ist das Software-Management in Automotiven Systemen. Dazu zählt u. a. das Updaten und Installieren neuer Software in IVI-Systemen. Dabei werden ebenfalls unterschiedliche Ansätze betrachtet, wie das Installieren oder Erneuern der Software initiiert werden kann. Dies kann sowohl von der Software in der Head-Unit automatisch oder per Benutzeranfrage über die HMI erfolgen.

Media & Graphics EG. Für die Grafik der HMI und das Audio-Management ist die EG *Media & Graphics* zuständig. Dabei reicht hier die Spannweite vom Graphic Back-End bis zum Layer-Management, was das Anzeige-Konzept in IVI-Systemen darstellt. Anders als in Desktop-Systemen wird hier die Anzeige über sogenannte Layers bewerkstelligt, die für den automotiven Einsatz ausgelegt sind. Das Audio-Management in Automotiven Systemen hat andere Anforderungen als in Desktop-Systemen. Im Fahrzeug müssen die Audio-Quellen nach definierten Regeln automatisch geschaltet werden, damit der Fahrer während der Fahrt nicht durch Benutzer-Eingaben für das Umschalten abgelenkt wird. Aus diesem Grund übernimmt die Head-Unit nach definierten Regeln die Schaltung der Audio-Quellen. Als Beispiel können hier ein eingehender Telefon-Anruf, das Abspielen einer Audio-Quelle oder die Ansagen der Navigation genannt werden.

CE Connectivity EG. In dieser EG steht die Anbindung der CE-Geräte (engl. *consumer electronics*) an die Head-Unit im Fokus. Dabei werden vorrangig die Verbindungen über USB, aber auch drahtlose Verbindungen über Bluetooth und WLAN betrachtet. Allerdings werden die Betrachtungen nicht auf diese Verbindungstypen beschränkt sondern werden auch auf andere Technologien wie NFC (engl. *near field communication*) ausgedehnt.

Neben den Anbindungsmöglichkeiten wird in dieser EG auch das Abspielen von Medieninhalten, die sich auf den CE-Geräten befinden, definiert. Darunter fallen

ebenso die Funktionen für die Indizierung sowie das Durchsuchen und Auswählen der Medien-Inhalte auf den CE-Geräten.

Location-Based Services EG. In der EG *Location-Based Services* werden die Themen Positionierung, Zugang zu den Karten-Daten, die Navigation, die Fahrer-Assistenz, der Erhalt und die Anzeige von Verkehrsinformationen sowie ein Flotten-Management behandelt.

Networking EG. In dieser EG steht die Daten-Verbindung zwischen dem IVI-System und der IT-Infrastruktur im Fokus. Als Daten-Verbindung werden zum einen die Mobilfunknetze als auch die WLAN-Access-Points betrachtet, wie es in Abschnitt 2.5 noch vorgestellt wird. Aus diesem Grund arbeitet die EG an einer generischen Software-Schnittstelle für beliebige Daten-Verbindungen, so dass Applikationen im IVI-System unabhängig von der physikalischen Daten-Verbindung mit der IT-Infrastruktur kommunizieren können.

Einen weiteren Bereich, den die EG *Networking* bearbeitet, sind die Schnittstellen für die Telefonie im Fahrzeug. Neben den Schnittstellen zur Applikation werden hier auch die Protokolle und physikalischen Verbindungen wie z. B. Bluetooth betrachtet.

Außerdem arbeitet die EG *Networking* an den Schnittstellen eines Browser-Framework, mit dessen Hilfe die HMI Web-Inhalte darstellen kann. Dabei werden die Schnittstellen zwischen der HMI und der Browser-Applikation definiert. Für das Hoch- und Herunterladen von Dateien aus dem Internet wird dabei ein zentraler Dienst spezifiziert, der diese Aufgaben übernimmt.

System Infrastructure EG. Im Fokus der EG *System Infrastructure* steht die Betriebssystem-Komponenten des IVI-Systems, die für das Aufstarten und Herunterfahren des Systems notwendig sind. Daneben kümmert sich die EG auch um die Software-Komponenten für die Ressourcen-Verwaltung. Zusätzlich werden ebenfalls Themen wie Lifecycle, Persistenz, Benutzerverwaltung oder Personalisierung des IVI-Systems behandelt.

Auch die Unterstützung der Basis-Betriebssystem-Funktionalität fällt in den Bereich dieser EG. Darin sind u. a. die Themen IPC (engl. *inter-process communication*) und die Definition der Komponenten-Schnittstelle durch eine gemeinsame IDL (engl. *interface description language*) enthalten.

Neben den Domain EGs gibt es noch zwei regionale EGs, die sich um die speziellen regionalen Bedürfnisse der asiatischen Ländern in GENIVI kümmern. Bis jetzt existiert eine EG für China und eine für Korea. Die Aufgaben der EGs reichen von Anpassungen im Betriebssystem für die asiatischen Schriftzeichen bis zu speziellen Sprachanpassungen. Neben den Sprach- und Schriftanpassungen liegen in diesen EGs auch die Umsetzungen von regionalen Standards, die für Automotive Systeme gelten, im Fokus.

Neben den Domain und Regional EGs sorgt das SAT (engl. *system architecture team*) für eine einheitliche System-Architektur. Die Ergebnisse der EGs bezüglich der Architektur werden von dem SAT überprüft und gegebenenfalls werden Nachbesserungen

gefordert. Somit wird sichergestellt, dass die GENIVI-Architektur über die einzelnen EGs hinweg einheitlich ist.

An das SAT ist ebenfalls das Security Team angehängt, das für sicherheitsrelevante Themen einer Head-Unit verantwortlich ist. Dabei steht das Security Team im Austausch mit den Domain EGs, die die Sicherheitsanforderungen, mögliche Angriffe und Abwehrmaßnahmen der jeweiligen Domäne erarbeiten. Der gesamte Prozess für die Sicherheit in der Head-Unit wird dann von dem Security Team koordiniert und die Ergebnisse der EG zusammengetragen und in eine einheitliche Spezifikation gegossen.

Für die Bereitstellung der GENIVI Referenz-Plattform ist das BIT (engl. *baseline integration team*) zuständig. Dabei stellt nicht das BIT selbst die Referenz-Plattform zur Verfügung, sondern definiert das Vorgehen, wie verschiedene Referenz-Plattformen für ein GENIVI-Release – die sogenannten Baselines – bereitgestellt werden können. Jede Gruppe kann einen Antrag bei dem BIT stellen, dass ihre Baseline auf die Liste der GENIVI konformen Baselines aufgenommen wird. Nach der Prüfung, ob die definierte Organisationsstruktur einer Baseline sowie die GENIVI-Spezifikation von der beantragten Baseline erfüllt ist, wird der Baseline die GENIVI-Konformität bestätigt.

In den Beschreibungen der einzelnen EGs mit deren Aufgaben und Projekten zeigt sich an einigen Stellen, inwiefern sich die OSS, die frei im Desktop und im CE-Geräte Umfeld zur Verfügung steht, an Automotive Systeme angepasst werden müssen und wie weit sich die Anforderungen zwischen den Desktop- und Automotiven Systemen unterscheiden. Die Referenz-Architektur, die in den EGs erarbeitet wird, kann mit ihren Komponenten in dem technischen Bericht „Reference Architecture“ der GENIVI-Allianz nachgeschlagen werden [56].

An dieser Stelle wird nicht im Detail auf die Software-Architektur des Linux-Systems eingegangen, sondern es wird auf den Abschnitt 3.2 verwiesen, in dem detaillierter die Software-Systeme beschrieben werden. Unter anderem wird auch Linux im Abschnitt 3.2.1 im Zusammenhang mit den Monolithischen Betriebssystemen beleuchtet.

2.5 Trends für die Zukunft der Automotiven Systeme

Neben der Bewältigung der Herausforderungen von Automotive Systemen mit neuen Techniken im Software-Engineering als auch durch die automotiven Software-Standards werden auch neue Zukunftstrends in den Automotiven Systemen sichtbar.

Eine der auffälligsten Entwicklung ist das Zunehmen von Fahrassistenz- und Komfortfunktionen im Fahrzeug. Dies geht von der automatischen Schilderkennung, Hindernis- und Personenerkennung, der Einpark-Hilfe über das ACC (engl. *adaptive cruise control*) bis hin zu den modernen Multimedia-Anwendungen in Automotiven Systemen, die aus dem Umfeld der Home-Cinema-Technologien ins Fahrzeug übertragen werden. Dies sind alles in heutigen Fahrzeug bereits integrierte Funktionen. Kurzfristig werden weitere Fahrzeug-Funktionen entwickelt, die den Fahrer beim alltäglichen Fahren unterstützen. So wird es als Beispiel Stau-Assistenten geben, die dem Fahrer sowohl das Lenken als auch das Anfahren und Abbremsen abnehmen. Ebenfalls werden neue Fahrzeug-Funktionen bei der lästigen Parkplatz-Suche Hilfestellungen leisten.

Ein mittelfristiges Ziel ist Drive by wire. Dies bedeutet, dass die Lenkung nicht mehr

durch eine Lenk-Säule mechanisch mit elektronischer Unterstützung erfolgt, sondern komplett elektronisch ohne mechanische Verbindung zwischen Lenkrad und Lenkachse. Dies würde neben der variablen Anordnung der Lenkachse und des Lenkrads auch größere Möglichkeiten für Assistenz-Funktionen in der Lenkung ergeben. Auch die Art der Lenkung könnte revolutioniert werden. Jedoch ist die Zuverlässigkeit der automotiven Software nicht hoch genug, als dass eine solch sicherheitskritische Fahrzeug-Funktion den heutigen Sicherheitsstandards erfüllen würde [33].

Wie in der Einführung beschrieben wurde, ist das langfristige Ziel im Bereich der Fahrassistenz-Funktionen das vollständig automatisierte Fahren und die automatische Stau-Vermeidung. Bis zu diesem Ziel werden die Fahrassistenz-Funktionen schrittweise verbessert, um diesem großen Ziel näher zu kommen. Dafür werden aber auch immer neue und leistungsstärkere Technologien benötigt. Im Bereich der Mikrocontroller bedeutet dies eine Erhöhung der Rechenleistung. Im Bereich der Sensoren müssen ebenfalls immer leistungsstärkere Sensoren in das Fahrzeug integriert werden, um eine detailliertere Umgebungswahrnehmung zu ermöglichen. Dies wird dazu führen, dass 3D-Scanner im Fahrzeug verbaut werden, die ein detailliertes 3D-Bild der Umgebung für die Realisierung der Fahrassistenz-Funktionen bereitstellt.

Diese Entwicklung hat allerdings unweigerlich zur Folge, dass die Software-Systeme, die die neuen Fahrzeug-Funktionen bereitstellen werden, immer weiter und stärker als bisher anwachsen. Somit ist es um so wichtiger, den Software-Entwicklungsprozess in Automotiven Systemen durch Methodiken und Werkzeug-Unterstützung aus dem Bereich des Software-Engineerings zu verbessern, um der steigenden Zahl an Fahrzeug-Funktionen gerecht zu werden.

Ein weiterer Trend wird der Ausbau der Fahrzeug-Kommunikation zu Infrastruktur-Komponenten oder anderen Fahrzeugen sein. Dies würden weitere Fahrassistenz-Funktionen ermöglichen, indem zusätzliche Umgebungsinformationen über die neuen Kommunikationswege in das Automotive System gelangen. Als Beispiele lassen sich Stau-Informationen oder Benachrichtigungen über einen vorausliegenden Unfall nennen. Um den erhaltenen Informationen vertrauen zu können, sind ebenfalls noch Erweiterungen in den Sicherheitskonzepten für eine Realisierung notwendig [42].

Wie bereits dargelegt wurde, müssen in den verschiedenen Domänen unterschiedliche Zuverlässigkeitsanforderungen erfüllt werden. Aus diesem Grund sind die Funktionen in den verschiedenen Teilnetzen realisiert worden, um eine strikte Trennung zwischen den zuverlässigkeitskritischen Fahrzeug-Systemen und den weniger kritischen Systemen zu gewährleisten. Jedoch ergibt sich die Problematik, dass Funktionen mit unterschiedlichen Zuverlässigkeitsanforderungen auf dem selben Steuergerät ausgeführt werden müssen, da beide Funktionen dieselbe Hardware wie z. B. das Display benötigen. Ein Beispiel wäre das Fahrer-Display, auf dem zum einen sicherheitskritische Fahrinformationen, wie die aktuelle Geschwindigkeit, zum anderen auch Informationen aus der Infotainment-Domäne wie die Richtungsanweisungen des Navigationssystems oder der aktuell abgespielte Liedtitel des Multimedia-Systems dargestellt werden. Hier muss dafür gesorgt werden, dass die Fahrzeug-Funktionen, die eine weniger zuverlässigkeitsrelevante Anforderung haben, die sicherheitskritischen Funktionen unter keinen Umständen beeinträchtigen. Dies ist einer der Anwendungsfälle, bei dem eine Virtualisierung von Steuergeräten auf einer einzelnen Hardware genutzt werden kann und deshalb für Automo-

tive Systeme untersucht wird. Die Virtualisierung ist eine Technik, die im Server- und Desktop-Bereich schon lange eingesetzt wird und auch hier wissenschaftlich untersucht worden ist. Auf diese Weise wird auch in Automotiven Systemen untersucht, ob die Virtualisierung im Punkt der Trennung zwischen unterschiedlichen sicherheitskritischen Funktionen oder die Ausführung von mehreren Binärdateien von unterschiedlichen Zulieferern auf einer ECU eine Lösung darstellen könnte.

Um die große Anzahl an Steuergeräten zu minimieren und dadurch sowohl Gewicht als auch Stromverbrauch zu reduzieren, wird über den Einsatz von Multicore-Prozessoren im Fahrzeug nachgedacht. Damit hat ein Steuergerät mehr Rechenleistung, die auch parallel im Vergleich zu einer reinen Erhöhung der Prozessor-Taktrate genutzt werden kann. Allerdings spielen hier ähnliche Parallelisierungsprobleme eine Rolle, wie sie z. B. auch im Server- oder Desktop Bereich anzutreffen sind [89, 112, 130].

Dies sind nur einige Beispiele an Zukunftstrends, in denen sich die Automotiven Systeme weiterentwickeln und verbessern, damit die in der Einführung genannte Vision der verzögerungs- und unfallfreien Mobilität Wirklichkeit wird.

Kapitel 3

Software- und Betriebssysteme

Im vorhergehenden Kapitel wurden Automotive Systeme mit ihrer Entwicklung, ihrer spezifischen Hardware und den Software-Standards wie OSEK/VDX, AUTOSAR oder GENIVI beschrieben. Dieses Kapitel soll einen besseren Einblick in die Software-Architektur geben, um den Zusammenhang zwischen einem Power-Management für Automotive Systeme und den Kundenfunktionen auf Applikationsebene besser zu verstehen.

Die Software-Architektur der Automotiven Systemen unterscheidet sich nicht wesentlich von der Software-Architektur, die in den Server- und Desktop-Systemen eingesetzt wird. Somit leiteten sich die automotiven Software-Systeme grundsätzlich von der Software-Architektur gängiger Rechensysteme ab, die sich in den 1960er und 1970er Jahren entwickelt haben. In Abschnitt 2.4.2 wurden bereits die Konzepte des AUTOSAR-Standards beschrieben. Im Verlauf dieses Kapitels wird sich zeigen, dass dieser Standard ähnliche Architektur-Konzepte wie typische Desktop- oder Server-Betriebssysteme beinhaltet und um spezifische automotive Ansätze erweitert wurde.

Im Folgenden wird zuerst eine Übersicht über die gängige Software-Architektur gegeben, bevor detailliert auf die Betriebssysteme eingegangen wird. Es wird sich an mehreren Stellen im Verlauf dieses Kapitels zeigen, dass sich die Software-Systeme an die darunter liegende Hardware mit den verschiedenen Ressourcen-Ausprägungen anpassen.

Im Anschluss wird der Fokus dieses Kapitels auf die Betriebssysteme gelegt und die verschiedenen Betriebssystem-Arten vorgestellt. Angefangen von der klassischen monolithischen Kernel-Architektur wird die Microkernel-Architektur bis hin zu der Architektur der Komponenten-basierten Betriebssystemen beschrieben. Die verschiedenen Betriebssystem-Architekturen können und werden zum Teil schon in Automotiven Systemen eingesetzt, worauf in den einzelnen Abschnitten speziell eingegangen wird.

Zum Abschluss dieses Kapitels wird ein konkretes Betriebssystem als Fallbeispiel vorgestellt, das in der PLASA-Plattform eingesetzt wird: das Komponenten-basierte Betriebssystem TinyOS. Die in dem Abschnitt dargelegte, detaillierte Beschreibung der Software-Architektur ist für das weitere Verständnis der PLASA-Plattform und des dort implementierten Power-Managements notwendig.

3.1 Übersicht über die Software-Systeme

Alle Software-Systeme sind von der Hardware abhängig, auf der sie laufen. Denn sie benötigen immer Hardware-Ressourcen, um ihre Funktionen zu realisieren. Darunter fallen u. a. der Prozessor mit dessen Rechen-Fähigkeit oder der Speicher für die Speicher-Fähigkeit der Daten. Um die Hardware-Ressourcen optimal ausnutzen zu können, sind die Software-Systeme immer für eine bestimmte Hardware-Klasse ausgelegt. Nach A. Tanenbaum lassen sich Software-Systeme mit deren Betriebssystemen in folgende Klassen einordnen, wobei ein konkretes Software-System auch mehreren Klassen zugeordnet werden kann [148]:

Mainframe-Systeme. Mainframe-Systeme sind die größten Rechen-Systeme, die vor allem in Daten-Zentren eingesetzt werden. Als Merkmal neben deren Größe ist die große Datenverarbeitungs- und Speicherfähigkeit. Außerdem sind diese Software-Systeme dafür ausgelegt, viele Tasks simultan auszuführen.

Server-Systeme. Ein Server stellt einen oder mehrere verschiedene Dienste für unterschiedliche Benutzer über ein Netzwerk bereit. Die Benutzer teilen sich somit die Hardware- und Software-Ressourcen des Server-Systems. Als Beispiel lassen sich Druck-, Datei- oder Web-Server nennen.

Multiprozessor-Systeme. In dieser Klasse sind alle Systeme enthalten, bei der die Hardware mehrere Prozessor-Kerne besitzt. Diese Software-Systeme sind meistens Abwandlungen der Server-Systeme mit zusätzlichen Funktionen in den Bereichen Kommunikation, Verbindungen und Daten-Konsistenz, um den Anforderungen von Multiprozessor-Systemen gerecht zu werden. In den letzten Jahren sind auch kleine Multiprozessor-Systeme im Desktop-Bereich anzutreffen, so dass auch die Desktop-Betriebssysteme mit mehreren Prozessor-Kernen umgehen können.

Desktop-Systeme. Die Desktop-Systeme sind die bekannteste Klasse an Software-Systemen. Dies besteht aus einem Rechner mit einer graphischen Oberfläche und Eingabegeräte für Benutzer wie Tastatur oder Maus. Das Ziel des Software-Systems ist die gute Unterstützung eines einzigen Benutzers zu einem Zeitpunkt.

Handheld und Smartphone-Systeme. Eine weitere Klasse von Software-Systeme ist die für Handhelds oder PDAs (engl. *personal digital assistant*). Daraus entwickelten sich die Software-Systeme für Smartphones. Kernkompetenzen der Software-Systeme sind Telefonieren, Nachrichten-Austausch, Kommunikation sowie Fotographie für die eingebaute Kamera. Der Unterschied zu den Desktop-Systemen liegt in den eingeschränkten Rechen- und Speicher-Ressourcen sowie den unterschiedlichen Eingabe-Möglichkeiten für den Benutzer, der den Handheld vorrangig über einen Touch-Screen bedient.

Embedded Systeme. Die Embedded Systeme werden meist in unterschiedlichen Bereichen eingesetzt und sind tief in den Geräten eingebettet. Meistens ist es auf Anhieb nicht ersichtlich, dass sich ein Software-System hinter dem Gerät verbirgt. In der

Einleitung im Abschnitt 1.1 wurde bereits vertieft auf die Embedded Systeme eingegangen.

Echtzeit-Systeme. In diesen Systemen spielt die Ausführungszeiten der Funktionen und das Einhalten der Deadlines die maßgebliche Rolle. Dabei werden die Systeme in harte und weiche Echtzeit-Systeme eingeteilt. In den harten Echtzeit-Systemen ist das Verletzen der Deadline mit hohen Kosten verbunden und somit schwerwiegender als in weichen Echtzeit-Systemen, in denen das Verletzen der Deadline keine großen Schäden nach sich zieht.

Sensor-Knoten. Sensor-Knoten sind in der Einleitung neben den Embedded Systemen ebenfalls vorgestellt worden. Diese Systeme erfassen die Umwelt über Sensoren und verteilen die gesammelten Daten in drahtlosen Netzwerken weiter. Neben den sehr geringen Rechen- und Speicher-Ressourcen spielt das Power-Management eine wesentliche Rolle, da die Knoten nur einen begrenzten Energie-Speicher besitzen, der in seltensten Fällen wieder aufgeladen werden kann.

Smart-Card-Systeme. Dies sind die kleinsten Software-Systeme, die auf Smart-Karten zum Einsatz kommen und auf dem CPU-Chip ausgeführt werden. Diese haben keine eigene Stromversorgung, sondern werden über den Kartenleser entweder über Pins oder induktiv mit Energie versorgt. Die Software-Systeme sind Java-orientiert, wobei ein Java-Byte-Code-Interpreter die sogenannten Java-Applets im Chip ausführt.

Die verschiedenen Klassen der Software-Systeme zeigen, in welchen Einsatz-Bereichen sie zur Anwendung kommen und dass den Systemen unterschiedliche Ressourcen in ihrer Art und Größe zur Verfügung stehen. Die ältesten Software-Systeme sind die Mainframe- und Server-Systeme. Daraus haben sich die Software-Systeme für die übrigen Systeme abgeleitet, so dass die grundlegende Software-Architektur über alle Klassen hinweg ähnlich strukturiert ist. Deshalb lässt sich in allen Systemen eine gemeinsame Software-Architektur erkennen. Sie lässt sich wesentlich in drei Teile gliedern, wie es in Abbildung 3.1 darstellt ist. Direkt auf der Hardware setzt das Betriebssystem auf, das die Basis des Software-Systems bildet. Darauf baut die Middleware auf, die für die Applikationen weitere Dienste bereitstellt, die nicht im Betriebssystem-Kern realisiert sind. Auf obersten Ebene sind die Applikationen angesiedelt, die die Funktionen des Systems realisieren.

In den verschiedenen Software-Systemen können die Schichten jeweils unterschiedlich groß ausfallen, womit auch die enthaltene Funktionalität unterschiedlich ist. Die Schichtung verfolgt den Zweck, Abstraktionsebenen in die Software einzuziehen und dadurch den darüber liegenden Software-Schichten eine einheitliche und besser zu bedienende Schnittstelle bereitzustellen. Die Grenzen zwischen den einzelnen Schichten sind fließend und je nach Betriebssystem unterschiedlich gelegt. Somit können u. U. einzelne Software-Funktionen je nach Ausprägung im konkreten Betriebssystem nicht klar einer Schicht zugeordnet werden. Dennoch lassen sich die einzelnen Schichten wie folgt charakterisieren:

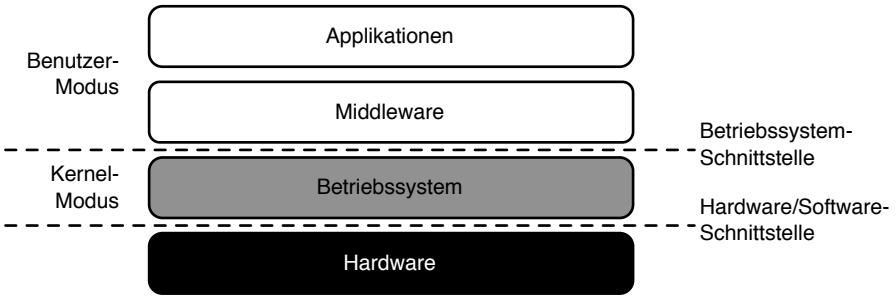


Abbildung 3.1: Klassische Schichtung der Software-Systeme.

Betriebssystem. Das Betriebssystem (engl. *operating system*) oder auch der Betriebssystem-Kern (engl. *kernel*) bildet die Basis des Software-Systems und setzt direkt auf der Hardware/Software-Schnittstelle auf. Nach A. Tanenbaum [148] ist die Aufgabe des Betriebssystems die schwer bedienbare Schnittstelle der Hardware zu einer leicht für die Applikationen zu bedienende Schnittstelle zu verwandeln. Diese Schnittstelle wird als Betriebssystem-Schnittstelle bezeichnet. Daneben ist eine wesentliche Aufgabe des Betriebssystems die Ressourcen-Verwaltung. Dabei verwaltet das Betriebssystem nicht nur Ressourcen, die die Hardware zur Verfügung stellt, sondern es werden ebenfalls virtuelle Ressourcen verwaltet, die es für die darüber liegende Schicht erschafft. H.-J. Siegert und U. Baumgarten sprechen hier auch von logischen Geräten [137]. Für die Ausführung der Applikationen gibt das Betriebssystem Konzepte für das gesamte Software-System vor und stellt die dafür notwendigen virtuelle Ressourcen bereit. Hier wird auch von einer Veredelung der Hardware gesprochen.

Middleware. Diese Schicht stellt weitere Dienste für die Applikationsschicht bereit, die nicht im Betriebssystem laufen. Die Middleware nutzt dabei die Betriebssystem-Schnittstelle und realisiert dabei ihre eigene zusätzliche Funktionalität außerhalb des Betriebssystems. A. Tanenbaum führt den Begriff in seinem Buch als Zwischenschicht in einem verteilten System ein, die die Kommunikation zwischen den einzelnen Knoten abstrahiert [150]. Weitere typische Software-Komponenten, die in der Middleware zu finden sind, sind neben den Kommunikationsdiensten Datenbank-Dienste oder eine graphische Bedienungsoberfläche, sofern diese nicht bereits im Betriebssystem realisiert wurde.

Applikationen. In der Applikationsschicht werden die spezifischen Funktionen des Systems ausgeführt. Dabei nutzen die Applikationen, die auch als Programme bezeichnet werden, die von der Middleware oder von dem Betriebssystem bereitgestellten generischen Dienstfunktionen, um ihre spezifischen Funktionen zu realisieren. Damit Applikationen nicht ständig neue Schnittstellen im Betriebssystem oder in der Middleware vorfinden und deshalb neu geschrieben werden müssen, haben sich im Laufe der Jahre Standards für Schnittstellen entwickelt, die die Applikationen nutzten können. Ein Beispiel dafür ist die POSIX-Schnittstelle [81]. Die Applikationen werden typischerweise in eigenen Umgebungen, den sogenannten Prozessen,

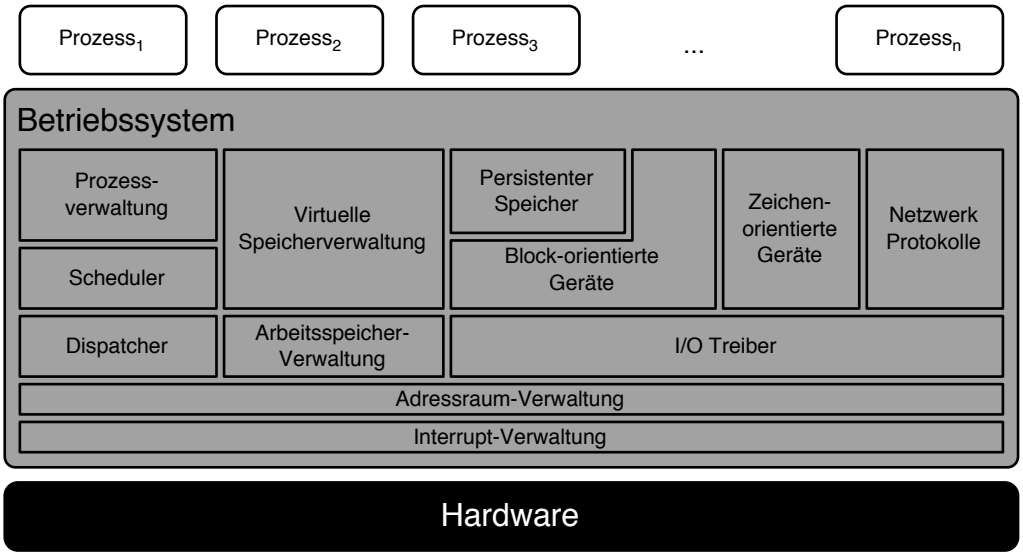


Abbildung 3.2: Teilkomponenten eines Betriebssystems.

ausgeführt. Prozesse sind ein Konzept, das das Betriebssystem für die Ausführung der Applikationen bereitstellt. Dabei enthält das Prozess-Konzept auch den Schutz eines Prozesses gegenüber anderen Prozessen vor dem ungewollten Lesen oder Manipulieren der Daten.

Auch wenn die Grenzen der drei Schichten fließend sind, so kann das Betriebssystem ganz klar von der Middleware und der Applikationsebene abgegrenzt werden. Sofern die Hardware unterschiedliche Privileg- oder Berechtigungsstufen anbietet, läuft das Betriebssystem alleine in der höchsten Privileg-Stufe, während die Middleware und die Applikationen in der nicht-privilegierten Stufe ausgeführt werden. Je nach Mikroprozessor werden unterschiedliche Privileg-Stufen angeboten. Dabei können bestimmte Prozessor-Befehle nur in der privilegierten Stufe aufgerufen werden, die somit nur dem Betriebssystem vorbehalten sind. Darunter fallen vor allem die Befehle für die Speicher- und Interrupt-Verwaltung. Für die unterschiedlichen Privileg-Stufen haben sich je nach Hardware-Hersteller unterschiedliche Begriffe etabliert. Für den privilegierten Modus spricht man auch vom Kernel-Modus (engl. *kernel mode*) oder Supervisor-Modus (engl. *supervisor mode*). Für den nicht-privilegierten Modus wird auch der Begriff Benutzer-Modus (engl. *user mode*) verwendet.

Eine der Aufgaben des Betriebssystems ist die Verwaltung der Hardware-Ressourcen. Dafür enthält jedes Betriebssystem typischerweise Teilkomponenten, die bestimmte Tätigkeiten übernehmen. In Abbildung 3.2 sind die einzelnen Teilkomponenten eines Betriebssystems dargestellt. Je nach Ausprägung der Hardware und den Anforderungen an das System sind diese Teilkomponenten unterschiedlich realisiert oder sind eventuell gar nicht enthalten. Als Beispiel lassen sich kleine Embedded Systeme nennen, in denen der Mikrocontroller keine MMU besitzt und somit die virtuelle Speicherverwaltung im Betriebssystem ggf. nicht möglich ist.

Grundlegend für die Steuerung der Hardware ist die Interrupt- und die Adressraum-Verwaltung, die jeweils eine Teilkomponente des Betriebssystems übernimmt. Auf diese Teilkomponenten bauen die anderen Teilkomponenten im Betriebssystem auf. Die Interrupt-Verwaltung setzt über die Hardware/Software-Schnittstelle die Interrupt-Register der Hardware und leitet die Interrupts an die dafür zuständigen Komponenten weiter, sobald ein Interrupt von der Hardware ausgelöst wurde. Über die Adressraum-Verwaltung wird der physikalische Adressraum der Hardware organisiert. Dort können die übrigen Teilkomponenten des Betriebssystems Bereiche für ihre Zwecke reservieren. Nicht nur die Arbeitsspeicher-Verwaltung greift darüber auf den Speicher zu, sondern auch die Treiber erhalten darüber Zugriff auf die Register der Hardware-Controller, die in der Abbildung 3.2 als I/O Treiber bezeichnet werden.

Wie schon erwähnt, werden die Applikationen oder Programme, die die Funktionen des Systems realisieren, in den sogenannten Prozessen ausgeführt, die vom Betriebssystem bereitgestellt werden. Damit ist das Betriebssystem für die Erzeugung und die Verwaltung der Prozesse verantwortlich. Beim Start einer Applikation wird deren Code in den Speicherbereich des Prozesses geladen. Sobald ein Prozess gestartet ist, muss das Betriebssystem für die faire Ausführung aller Prozesse im System sorgen. Die zwei Teilkomponenten im Betriebssystem, die für die Prozesse verantwortlich sind, sind der *Dispatcher* und der *Scheduler*. Der Dispatcher ist dafür zuständig, dass einem laufenden Prozess der Prozessorkern entzogen und der neue Prozess auf dem Prozessorkern zur Ausführung gebracht wird. Dabei bestimmt der Scheduler den neuen Prozess nach einer für das System bestimmten Strategie. Im Laufe der Jahre sind verschiedene Scheduling-Strategien entwickelt worden, die die verschiedenen Anforderungen des Gesamtsystems erfüllen. Je nach Betriebssystem ist es auch möglich, dass ein Prozess mehrere Ausführungsfäden oder Threads besitzt. Dann werden die Threads quasi-parallel im Adressraum des Prozesses ausgeführt. Für die Synchronisation der Threads muss das Betriebssystem Synchronisationshilfsmittel wie z. B. Semaphoren anbieten, damit der Zugriff auf die gemeinsamen Daten durch die Threads synchronisiert erfolgt. Je nach Betriebssystem werden unterschiedliche Arten von Synchronisationshilfsmittel angeboten.

Eine weitere Ressource neben dem Prozessor ist der Speicher, in dem die Applikationen ihre Daten ablegen können. Das Betriebssystem stellt den Prozessen Speicher bereit und sorgt dafür, dass ein Prozess nicht auf die Daten eines anderen Prozesses zugreifen kann. Dies beinhaltet nicht nur das Lesen, sondern auch das Ändern der Daten. Für diesen Zweck wird jedem Prozess ein eigener virtueller Adressraum bereitgestellt, über den er alleine verfügen kann. Für diese Aufgabe ist, wie es der Name schon andeutet, die virtuelle Speicherverwaltung zuständig. Dabei erhält das Betriebssystem für diese Aufgabe Unterstützung von der Hardware, die mit Hilfe der MMU realisiert wird. In kleinen Mikrocontrollern, in denen keine MMU verbaut ist, wird auf eine virtuelle Speicherverwaltung verzichtet und der Schutz der Prozess-Daten muss über einen anderen Mechanismus erfolgen, sofern dieser vom Betriebssystem bereitgestellt wird.

Durch die virtuelle Adressierung besteht zudem die Möglichkeit, dass das Betriebssystem allen Prozessen mehr Speicher zur Verfügung stellt, als dass die Hardware besitzt. Sobald der sogenannte Arbeitsspeicher ausgeschöpft ist, lagert das Betriebssystem zur Zeit nicht benötigte Speicherseiten in den Hintergrund-Speicher aus. Dies wird auch als

Paging bezeichnet. Für die Bestimmung der auszulagernden Seite sind ebenfalls viele Strategien entstanden, die unterschiedliche Anforderungen erfüllen.

Eine weitere Aufgabe des Betriebssystems ist die Steuerung der Hardware. Hier bietet das Betriebssystem eine abstrahierte Schnittstelle für die Prozesse an, so dass diese nicht die konkreten Hardware/Software-Schnittstelle kennen müssen. Dafür bietet üblicherweise das Betriebssystem eine funktionale Schnittstelle an. Dafür gibt es die Block-orientierten Geräte und die Zeichen-orientierten Geräte. Die Block-orientierten Geräte werden in Blöcke eingeteilt, die einzeln über eine Block-Nummer gelesen oder beschrieben werden können. Bei den Zeichen-orientierten Geräten werden Zeichenströme gelesen oder geschrieben.

Für die persistente Speicherung der Prozess-Daten bieten die Betriebssysteme meist ein hierarchisch strukturiertes Dateisystem an. Prozesse können in Dateien Daten Byteweise ablegen und zu einem späteren Zeitpunkt wieder auslesen. Das Betriebssystem legt dabei nicht die Struktur der Dateien fest. Somit kann jede Applikation die Struktur der Dateien selbst bestimmen. Jedoch erhalten die Applikationen keinen direkten Zugriff auf die Speichermedien, sondern nur auf das virtuelle Dateisystem, das von dem Betriebssystem angeboten wird. Das Betriebssystem bestimmt über Dateisystem-Formate wie die Dateisystem-Struktur und die Inhalte der Dateien Block-weise auf dem Speichermedium abgelegt werden. Dafür bieten die Treiber des Speichermediums die Block-orientierte Schnittstelle an, über die die Daten Block-weise auf die Speichermedien geschrieben werden. Allerdings werden in kleinen Embedded Systemen mit wenig persistentem Speicher eventuell andere Schnittstellen zur persistenten Datenspeicherung vom Betriebssystem angeboten.

Bei der Netzwerk-Kommunikation bietet das Betriebssystem eine Socket-Schnittstelle für die Datenkommunikation an. Die Applikationen können darüber Zeichen-orientiert Daten empfangen oder versenden. In vielen Software-Systemen sind die Netzwerk-Protokolle ebenfalls als Teilkomponente im Betriebssystem enthalten. Somit müssen die Applikationen nicht die Netzwerk-Protokolle oder sogar die Hardware/Software-Schnittstelle der Netzwerkkarte kennen, um Nachrichten-Pakete zu empfangen oder zu versenden.

Die beschriebenen Teilkomponenten geben nur einen Überblick über die Funktionen und die Ressourcen-Verwaltung in Betriebssystemen. Je nach konkretem Betriebssystem können sich die Konzepte und Schnittstellen dafür unterscheiden.

3.2 Arten von Betriebssystemen

Wie bereits in der Übersicht über die Software-Systeme beschrieben wurde, sind die Betriebssysteme die Basis des Software-Systems und bestimmen somit grundlegend den Aufbau des restlichen Systems. Im Laufe der Zeit haben sich drei Betriebssystem-Arten mit deren verschiedenen Architekturen und Konzepten entwickelt: Die Monolithischen Betriebssysteme, die Microkernel-Betriebssysteme und die Komponenten-basierten Betriebssysteme.

Im folgenden werden die drei Arten der Betriebssysteme im einzelnen vorgestellt, um ein besseres Verständnis für die einzelnen Betriebssystem-Architekturen zu bekommen. Dabei wird bei jeder Betriebssystem-Art auf unterschiedliche Konzepte eingegangen. Der

Fokus liegt hierbei auf Betriebssystemen für Embedded Systeme. Jedoch bedeutet dies nicht, dass die beschriebenen Architekturen und Konzepte sich auf Embedded Systeme beschränken. Diese finden ebenfalls in den anderen Hardware-Systemen Anwendung.

Zusätzlich wird neben den drei bereits genannten Betriebssystem-Arten das Singularity-Projekt von Microsoft Research vorgestellt, das prinzipiell die Architektur eines Microkernel-Betriebssystems verfolgt. Jedoch besitzt es einen neuartigen Ansatz der Adressraum-Trennung zwischen den Benutzerprozessen, weshalb es als eigene Betriebssystem-Art aufgefasst werden kann.

3.2.1 Monolithische Betriebssysteme

Die ältesten Betriebssysteme sind die Monolithischen Betriebssysteme, bei denen alle Treiber und viele Dienste des Software-Systems im Betriebssystem-Kern laufen. Dabei teilen sich alle Betriebssystem-Komponenten einen gemeinsamen Adressraum und laufen im privilegierten Modus des Prozessors wie es in der Übersicht dieses Kapitels beschrieben wurde. Die bekanntesten Vertreter der Monolithischen Systeme ist das Betriebssystem Windows von Microsoft [128] oder das OSS-Betriebssystem Linux [32, 38].

Wie zuvor beschrieben wurde, setzt sich ein Software-System aus dem Betriebssystem-Kern, der Middleware und den Applikationen zusammen. Für den Linux-Kernel haben sich sogenannte Linux-Distributionen entwickelt, die sich über ein Paket-Management je nach System-Anforderungen konfigurieren lassen. Viele der Linux-Distributionen bedienen sich der OSS und stellen daraus ein komplettes Software-System zusammen. Je nach Anwendungsgebiet gibt es unterschiedliche Linux-Distributionen, die die dort herrschenden Anforderungen erfüllen.

Für Embedded Systeme oder Smartphones existieren Linux-Distributionen, die unterschiedliche Gemeinschaften oder Firmen im Laufe der Zeit entwickelt haben. So hat Nokia für ihre Smartphones Maemo [105] oder Intel Moblin [111] bereitgestellt. Beide Firmen haben sich zusammengeschlossen, um die Projekte Maemo und Moblin in MeeGoo zu vereinigen. Jedoch ist nur kurze Zeit später das Projekt MeeGoo in Tizen [156] aufgegangen. Beide Plattformen sind ebenfalls für Automotive IVI-Systeme ausgelegt, da beide Plattformen GENIVI-konform sind, wie auf der GENIVI-Homepage veröffentlicht ist [54].

Auf dem Smartphone-Markt hat sich vor allem das Android-System von Google durchgesetzt [59], in dem aufbauend auf einem Linux-Kern die virtuelle Maschine Dalvik läuft [124]. Die Applikationen sind in der Programmiersprache Java geschrieben, die in der virtuellen Umgebung die speziell für die Android-Plattform spezifizierten Schnittstellen nutzen.

Für den Fall, dass eine Linux-Distribution benötigt wird, die speziell für eine embedded Plattform angepasst ist, bietet das Build-Framework OpenEmbedded die Möglichkeit, mit Hilfe sogenannter Rezepte aus OSS eine Linux-Distribution zu generieren [115]. Ein Rezept ist dabei eine Anleitung, wie aus einem OSS-Projekt, das den Source Code öffentlich im Internet anbietet, ein Installationspaket erstellt wird, das die übersetzten Binär-Dateien für die embedded Plattform enthält. Dies erfolgt durch einzelne Schritte, die wie folgt hintereinander ausgeführt werden: Zuerst werden die offenen Sourcen des einzelnen Projekts aus dem Internet geladen, wobei das Rezept die Download-Quelle

für den Source Code enthält. Anschließend wird mit Hilfe des Cross-Compilers, der zuvor für das Embedded System erstellt wurde, der Quell-Code des Projekts kompiliert und anschließend zu einem oder mehreren Installationspaketen zusammengefasst. Das Rezept enthält wiederum die Anleitungen für das Kompilieren, sowie das Aufteilen der kompilierten Dateien auf die Installationspakete. Im letzten Schritt werden die Inhalte der benötigten Installationspakete zu einem gepackten Verzeichnis zusammengestellt, das auf der Zielplattform im Wurzel-Verzeichnis entpackt wird.

Ein Nachteil der Monolithischen Betriebssysteme ist, dass alle Treiber und Dienste gemeinsam in dem Betriebssystem-Kern laufen. Falls ein Treiber fehlerhaft ist und einen Absturz verursacht, bringt er das gesamte System zum Absturz und es muss neu gestartet werden. Fehlerhafte Treiber sind in Windows XP mit 85 % der häufigste Grund für einen Absturz [146]. A. Chou *et al.* untersuchten in ihrer Studie die Fehler in Betriebssystemen und kamen zum Ergebnis, dass die Treiber eine drei bis siebenfache höhere Fehlerrate als die übrigen Komponenten des Betriebssystems haben [36]. Vor allem in Betriebssystemen, in denen der Kern nicht versiegelt ist, d. h. dass Code dynamisch zur Laufzeit in den Kern nachgeladen werden kann, verringert das die Zuverlässigkeit des Gesamtsystems erheblich.

Ein weiterer Nachteil in Monolithischen Betriebssystemen ist die Sicherheit. Da alle Komponenten einen gemeinsamen Adressraum im Betriebssystem-Kern teilen, hat jede Komponente im Betriebssystem Zugriff auf die gesamten Daten. Somit können die Daten von einer bösartigen Komponente gelesen oder sogar manipuliert werden. A. Tanenbaum gibt einen Überblick, wie Betriebssysteme robuster gegen Abstürze und sicherer gegenüber Angreifern gemacht werden können [149]. Er schlägt dabei viele Varianten vor, die die Konzepte der virtuellen Maschinen [139] nutzen oder über Microkernel-Betriebssysteme realisiert werden, auf die im nächsten Abschnitt eingegangen wird.

3.2.2 Microkernel-Betriebssysteme

Einen umgekehrten Ansatz zu den Monolithischen Betriebssystemen verfolgen die Microkernel-Betriebssysteme, bei denen sich nur die Treiber im Betriebssystem-Kern befinden, die auch im privilegierten Modus des Prozessors laufen müssen. Alle übrigen Treiber und Dienste laufen als Benutzerprozesse im nicht-privilegierten Modus des Prozessors.

Durch das Auslagern der Treiber und Dienste in Benutzer-Prozessen wird das gesamte Software-System zuverlässiger und robuster. Denn falls ein Treiber oder Dienst abstürzt, was eine der häufigsten Ursachen für einen Absturz eines Monolithischen Betriebssystem ist, bleibt der Microkernel davon unberührt und das System läuft bis auf den fehlerhaften Treiber weiter. Falls die notwendigen Prozesse überwacht werden, kann im Notfall auch der fehlerhafte Treiber neu gestartet werden, um ein voll funktionstüchtiges Software-System wieder herzustellen. Ebenfalls wird die Sicherheit in einer Microkernel-Architektur erhöht, da es viele abgeschottete Prozesse mit kleineren Funktionen gibt und nicht wie bei Monolithischen Systemen einen großen Kern, in dem alle Betriebssystem-Komponenten einen Adressraum teilen und auf Daten anderer Komponenten zugreifen können.

Im Folgenden wird Symobi als Vertreter der Microkernel-Betriebssysteme vorgestellt, das sich aus dem Betriebssystem μnOS entwickelt hat und in Embedded Systemen ein-

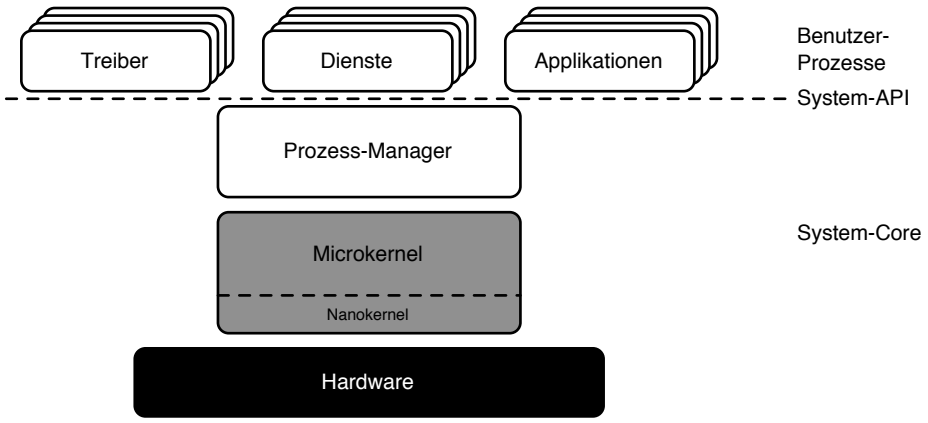


Abbildung 3.3: Architektur von Symobi (angelehnt an eine Abbildung aus der Veröffentlichung von R. Dörfel *et al.* [41]).

gesetzt wird [68]. Neben Symobi sind ab 1990 weitere Microkernel-Betriebssysteme in der Forschung entstanden. Einige von diesen wurden später als kommerzielle Betriebssysteme eingesetzt. Einer der Vertreter ist QNX [75], das ebenfalls in Automotiven Systemen vertreten ist. Dort kommt QNX vor allem in Head-Units zum Einsatz, in denen dessen Echtzeit-Fähigkeit und Zuverlässigkeit gefragt sind. Ein weiterer Vertreter der Microkernel-Betriebssysteme ist der L4 Kernel, der von J. Liedtke ebenfalls in den 1990er Jahren entwickelt wurde [101]. Der L4 Kernel ist über die Jahre weiterentwickelt worden und kommt in verschiedenen Versionen wieder zum Einsatz. So wird er als OKL4 als Hypervisor in Embedded Systemen vorrangig in Mobiltelefonen eingesetzt [70, 71]. Ein weiterer Vertreter der Microkernel-Betriebssysteme ist der MINIX-Kernel, der an der Vrije Universität in Amsterdam entstanden ist [73].

Symobi leitet sich aus „System for mobile applications“ ab und hat somit den Fokus auf Embedded Systemen [110]. Symobi läuft auf verschiedenen Hardware-Plattformen wie der x86-Plattform, der ARM-Plattform oder der PowerPC-Plattform. Um einen besseren Überblick über die Microkernel-Betriebssysteme zu erhalten, ist in Abbildung 3.3 eine Übersicht über die Architektur von Symobi abgebildet. Der Prozess-Manager und der Microkernel bilden den sogenannten System-Core, auf den die Benutzerprozesse über die System-API zugreifen können. Allerdings läuft nur der Microkernel im privilegierten Modus des Prozessors, was mit dem grauen Hintergrund dargestellt ist. Treiber und Dienste, die aus dem Kern ausgelagert sind, laufen neben den Anwendungen als Benutzerprozesse im nicht-privilegierten Modus des Prozessors. Der Prozess-Manager hat in Symobi eine Sonderrolle. Er ist ein besonderer Prozess, der zwar zum System-Core gehört, ab nicht im privilegierten Modus des Prozessors läuft. Dieser wird vom Microkernel nach dessen Start als erstes ausgeführt. Die Aufgaben sind wie folgt aufgeteilt: Der Microkernel enthält die Mechanismen, während der Prozess-Manager die Strategien dafür bereitstellt. Im Unterschied zu den Monolithischen Betriebssystemen sind alle weiteren Treiber und Dienste aus dem Kern ausgelagert und laufen als eigenständige Benutzerprozesse im nicht-privilegierten Modus des Prozessors. Der Microkernel hat eine HAL (engl. *hardware abstraction layer*), die in der Symobi-Architektur Nanokernel

genannt wird. Dieser stellt für die oberen Schichten des Microkernels eine einheitliche Schnittstelle zur Verfügung, die auf allen Hardware-Plattformen gleich ist.

Symobi erfüllt mit seiner Architektur vier Anforderungen, die in Embedded Systemen vorherrschen: Zuverlässigkeit, Echtzeit, Skalierbarkeit und Portabilität [41]. Die Zuverlässigkeit des Betriebssystems wird durch die Auslagerung der Treiber und Dienste in die Benutzer-Prozesse erreicht, was bereits zuvor erläutert wurde. Die Skalierbarkeit des Systems ist ebenfalls dadurch gegeben, da nur die zu einem bestimmten Zeitpunkt benötigten Treiber und Dienste dynamisch in einem Prozess gestartet werden können.

Die geringe Abhängigkeit von der Hardware-Plattform wird durch den Nanokernel mit dessen gleichbleibender Schnittstelle über alle Plattformen hinweg erreicht. Aufbauend auf dieser Schnittstelle ist der restliche Microkernel in Hardware-unabhängigen C-Code geschrieben, so dass der Microkernel für jede Hardware-Plattform kompiliert werden kann, sofern ein C-Compiler für die Hardware-Plattform existiert. Alle über dem Microkernel liegenden Software-Komponenten sind in Symobi in C^{++} programmiert, so dass auch hier eine leichte Portierung gegeben ist, sofern ein C^{++}-Compiler für die Hardware-Plattform vorhanden ist. Nur im Nanokernel sind Hardware-abhängige Assembler-Routinen enthalten, die für eine Portierung auf eine neue Hardware-Plattform angepasst werden müssen. Aus diesem Grund kann Symobi mit verhältnismäßig geringem Aufwand portiert werden.

Symobi stellt für die Prozesse, in denen die Treiber, Dienste und Applikationen laufen, geschützte Adressräume bereit. Somit kann kein Prozess auf den Speicherbereich eines anderen Prozesses zugreifen. Für den Datenaustausch zwischen den Prozessen stellt Symobi eine IPC bereit. Nur über die IPC-Mechanismen ist es den Prozessen möglich, Daten zwischen ihnen auszutauschen. Symobi bietet eine IPC über Nachrichten nach dem Client-Server Prinzip an. Jeder Server erstellt für einen Service einen Kanal, mit dem sich beliebig viele Client-Threads verbinden können. Dabei kann aber der Server entscheiden, mit welchen Clients er eine Verbindung eingehen möchte und mit welchen er keine Verbindung aufbauen will. Nachdem eine Verbindung aufgebaut wurde, d. h. der Server die Anfrage vom Client angenommen hat, kann der Client Anfragen an den Server schicken. Der Client-Thread bleibt dabei solange blockiert, bis der Server die Antwort auf derselben Verbindung zurückgeschickt hat. Somit besitzt Symobi eine synchrone, bidirektionale Kommunikation. Allerdings kann der Server keine Anfragen an den Client-Thread initiieren. Er muss immer auf eine Anfrage vom Client warten und kann erst anschließend darauf antworten. Somit ist es keine symmetrische Kommunikation. Allerdings erlaubt es Symobi, dass zwei Prozesse wechselseitig Kanäle öffnen und sich dann anschließend mit den Kanälen des jeweils anderen Threads verbinden. Dieser Zusammenhang ist in Abbildung 3.4 noch einmal graphisch dargestellt.

Diese Form der IPC ist mit Performance-Einbußen verbunden. Denn die Daten müssen von einem Prozess-Adressraum in den Prozess-Adressraum des anderen Prozesses kopiert werden. Vor allem in einer Microkernel-Architektur wird dieser Performance-Verlust gegenüber Monolithischen Systemen noch einmal verstärkt, da jeder Treiber und Dienst in einem eigenen Prozess läuft. Somit müssen z. B. Daten, die über die Netzwerk-Verbindung kommen über mehrere Prozesse kopiert werden, bevor diese z. B. in einer Browser-Applikation dargestellt werden können. So empfängt der Treiber für die Netzwerk-Karte die Daten von dem Hardware-Controller und überträgt diese an den

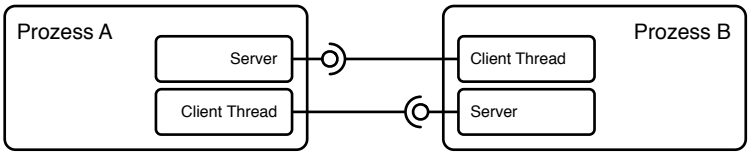


Abbildung 3.4: Interprozesskommmunikation zweier gleichberechtigter Prozesse in Symobi (entnommen aus der Veröffentlichung von R. Dörfel *et al.* [41]).

Netzwerk-Stack, der in einem weiteren Prozess läuft, bevor die Browser-Applikation die Daten vom Netzwerk-Stack erhält. Während dieser Kette finden zusätzlich Prozess-Wechsel statt, die neben dem Kopieren der Daten einen zusätzlichen Overhead und somit ein Performance-Verlust bedeuten. Im Vergleich dazu findet in einem Monolithischen Betriebssystem, in dem der Treiber und der Netzwerk-Stack im Betriebssystem-Kern laufen, kein Kopieren der Daten und keine Prozess-Wechsel statt. Erst wenn die Daten an die Browser-Applikation übergeben werden, müssen die Daten aus dem Betriebssystem-Adressraum in den Prozess-Adressraum kopiert und der Prozess-Kontext gewechselt werden.

Um das in vielen Situationen unnötige Kopieren der Daten zu vermeiden, bietet Symobi einen weiteren Mechanismus für den Daten-Austausch zwischen Prozessen an: den gemeinsamen Speicher. Hierbei fordert ein Prozess beim System-Core einen gemeinsamen Speicher einer gewissen Größe an, den er beliebig nutzen und für andere Prozesse freigeben kann. Dabei kann der Prozess, der sich den Speicher beim System-Core reserviert hat, weiteren Prozessen je nach Bedarf nur Lese-Rechte, aber auch Schreibrechte an dem gemeinsamen Speicher geben. Sobald der gemeinsame Speicher für den weiteren Prozess freigegeben wurde, können Daten zwischen den Prozessen ausgetauscht werden, ohne dass diese kopiert werden müssen. Jedoch muss der Zugriff auf die gemeinsamen Daten über Nachrichten synchronisiert werden, damit kein gleichzeitiger Zugriff auf die Daten stattfinden kann.

Somit wird die Zuverlässigkeit und die höhere Robustheit in Microkernel-Betriebssystemen durch einen Performance-Verlust erkauft. An dieser Stelle müssen die Anforderungen an das Software-System analysiert werden, um festzustellen, ob die höhere Zuverlässigkeit den Performance-Verlust in einem konkreten System rechtfertigt.

3.2.3 Das Singularity-Projekt

Um die Frage zu untersuchen, wie eine moderne Software-Architektur mit den heutigen Entwicklungswerkzeugen aussehen könnte, startete 2003 das Singularity-Projekt bei Microsoft Research. Denn die gängigen Betriebssysteme teilen, wie in den vergangenen Abschnitten beschrieben wurde, dieselbe Methodik sowie dieselben Grundlagen und Techniken, die in den 1960ern und 1970ern Jahre Stand der Technik waren. Dabei untersuchten G. Hunt *et al.* ebenfalls die Probleme, die in der gängigen Software-Architektur und in den Betriebssystem vorrangig vorherrschen und in welchen Gebieten im Bereich der Betriebssysteme weiter geforscht und Ergebnisse erzielt werden können [79]. Er zählt dabei folgende fünf Problemgebiete auf:

Verlässlichkeit. Das System muss verlässlich sein und die Erwartungen des Benutzers erfüllen. Es muss vermieden werden, dass sich das System für den Benutzer unerwartet verhält.

Sicherheit. Das System muss sicher sein, d. h. die verschiedenen Benutzer des Systems und deren Daten müssen geschützt sein. Fehlerhafte Programme oder schädliche Software dürfen keinen Zugriff auf die Daten erlangen oder den Ablauf der Programme beeinflussen.

System-Konfiguration. Die Abhängigkeiten von System-Komponenten sind meist nur informell charakterisiert und nicht vollständig mit deren Abhängigkeiten spezifiziert. Daraus können sich in größeren Systemen Schwierigkeiten ergeben, wenn z. B. unterschiedliche Versionen von gemeinsam genutzte Bibliotheken benötigt werden und dies nicht erkannt wird.

System-Erweiterbarkeit. Viele System-Komponenten bieten die Möglichkeit, Funktionserweiterungen nachträglich zu laden, um den Funktionsumfang der Komponente an die Bedürfnisse des Benutzers anzupassen. Dies führt allerdings in den heutigen Systemen zu einem Sicherheitsrisiko, da der dazu geladene Code nicht von der Hauptkomponente isoliert ausgeführt wird.

Multi-Prozessor Programmierung. Die Anzahl der Prozessor-Kerne wächst in den letzten Jahren immer mehr. Jedoch fehlt die Unterstützung der Multiprozessoren in der Programmierung. Somit ist eine der Herausforderungen, die wachsende Zahl der Prozessor-Kerne in der System-Architektur besser zu unterstützen.

Nach genauer Betrachtung kämpfen die Automotiven Systeme ebenfalls mit diesen Herausforderungen und versuchen Teile der Probleme mit Hilfe des Industrie-Standards AUTOSAR, der in Abschnitt 2.4.2 vorgestellt wurde, und des GENIVI-Standards (siehe Abschnitt 2.4.3) in den Griff zu bekommen. Aus diesem Grund werden die Erkenntnisse aus dem Singularity-Projekt an dieser Stelle kurz vorgestellt.

Für die Entwicklung des Singularity-OS lagen drei Design-Prinzipien zu Grunde, die J. Larus und G. Hunt wie folgt beschreiben [92]:

1. *Sichere höhere Programmiersprachen sollen im größtmöglichen Umfang eingesetzt werden.* Das gesamte System sollte in einer Typ-sicheren Programmiersprache geschrieben sein, das viele der kritischen Fehler möglichst frühzeitig erkennt, wie z. B. Puffer-Überläufe.

2. *Software-Fehler sollten nicht zu System-Fehlern führen.* Trotz der weiterentwickelten Programmiersprachen wird eine fehlerfreie Software nie existieren. Aus diesem Grund sollte ein auftretender Fehler lokal bleiben und nicht die Robustheit des gesamten Systems beeinträchtigen.

3. *Systeme sollten in allen Abstraktionsschichten selbst-beschreibend sein.* Durch formale Verifikationsmethoden ist es möglich, dass jede einzelne System-Komponente überprüft werden kann, ob sie ihre Spezifikation der Schnittstelle erfüllt. Dies setzt

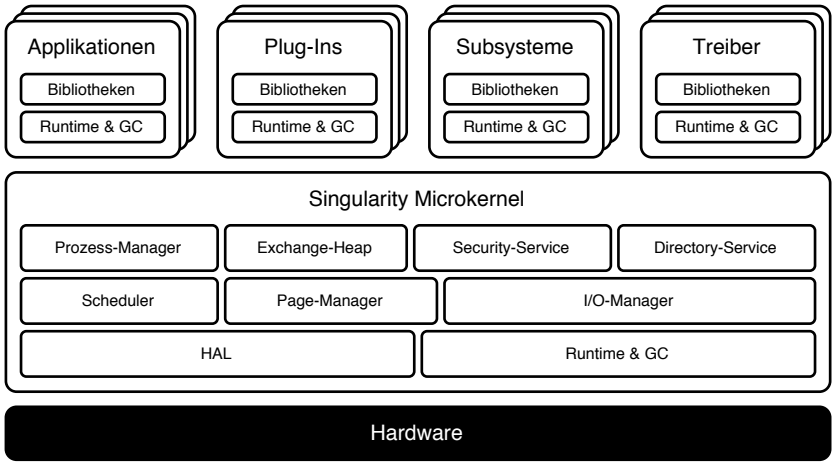


Abbildung 3.5: Architektur von Singularity (angelehnt an eine Abbildung aus der Veröffentlichung von G. Hunt *et al.* [80]).

allerdings voraus, dass für jede System-Komponente eine Spezifikation vorhanden ist.

Aus diesen Design-Prinzipien entstand eine Software-Architektur, die in Abbildung 3.5 dargestellt ist [80]. Wie aus der Abbildung ersichtlich ist, wird als grundlegende Architektur des Singularity-OS die Microkernel-Architektur verwendet, wie sie im vorhergehenden Abschnitt 3.2.2 beschrieben wurde. Jedoch unterscheidet sich die Software-Architektur erheblich von den bis dahin bekannten Architekturen. Der Hauptunterschied ist, dass auf Typ-sichere Programmiersprachen gesetzt wurde, um die Zuverlässigkeit des Systems zu erreichen. So ist der Singularity-Kernel zum größten Teil mit dem Typ-sicheren Sprachanteil der Programmiersprache C# oder in Sing# geschrieben, die eine Erweiterung der Sprache C# darstellt. Nur kleine Teile in der HAL enthalten den unsicheren Sprachanteil der Programmiersprache C#, C^{++} oder Assembler-Code. Dieser Teil wird in der Architektur von Singularity als TCB (engl. *trusted computing base*) bezeichnet, weil vertraut werden muss, dass in diesem Teil des Singularity-Kernels keine Fehler enthalten sind.

Die übrigen Teile des Systems sind mit sicheren Programmiersprachen geschrieben, die in die Typ-sichere MSIL (engl. *Microsoft intermediate language*) übersetzt wurden. Die MSIL ist die Implementierung der ECMA CIL (engl. *common intermediate language*) [45] von Microsoft. Zur Installationszeit wird dann der MSIL-Code vom Bartok-Compiler in nativen Code übersetzt. Dabei wird dem Bartok-Compiler vertraut, dass dieser vertrauenswürdigen Code erzeugt und ist deshalb in der TCB enthalten.

Für die Ausführung der MSIL wird die Laufzeitumgebung CLR (engl. *common language runtime*) benötigt, die wiederum die Implementierung der ECMA CLI (engl. *common language infrastructure*) [44] von Microsoft ist. Diese Laufzeitumgebung ist in jedem Prozess und im Kernel enthalten. Zu dieser Laufzeitumgebung gehört ebenfalls ein GC (engl. *garbage collector*), der sich um die Freigaben der nicht mehr benötigten Objekte kümmert. Mit dem Singularity-Projekt konnte leider nicht die Frage nach einem

effizienten GC im Kernel beantwortet werden [92].

Aufbauend auf der beschriebenen Architektur sind drei neue Konzepte im Singularity-Projekt entstanden: die SIPs (engl. *software-isolated process*), die Contract-Based Channels und die MBPs (engl. *manifest-based program*), auf die im Folgenden eingegangen wird [78].

Software-Isolated Processes

Eines der wesentlichen Konzepte in Betriebssystemen sind die Prozesse, wie sie in Abschnitt 3.1 vorgestellt wurden. Deren Eigenschaft ist u. a., dass der Adressraum eines Prozesses gegenüber anderen Prozessen geschützt ist. Ein Informationsaustausch zwischen zwei Prozessen erfolgt dann via IPC. Typischerweise werden die Prozess-Adressräume mit Hilfe der MMU, also mit Hardware-Unterstützung, geschützt. Dieser Adressraum-Schutz wird in den Veröffentlichungen aus dem Singularity-Projekt auch als HIP (engl. *hardware-isolated process*) bezeichnet.

Im Singularity-OS wird dagegen der Adressraum der Prozesse durch die Sicherheit der modernen Programmiersprachen geschützt, wie es zuvor in der Architektur-Beschreibung dargestellt ist. Dies wird mit der Typ-sicheren Sprache Sing# erreicht. Bis auf die Art wie die Daten eines Prozesses geschützt werden, unterscheiden sich die SIPs nicht von den Prozessen herkömmlicher Betriebssysteme. Sie besitzen Code und Daten, die in Speicher-Seiten abgelegt werden. Für die Ausführung des Codes wird ebenfalls das Konzept der Threads genutzt, die im Prozess-Kontext ausgeführt werden.

Jedoch sind die SIPs wie der Kernel im Singularity-OS versiegelt. Dies bedeutet, dass kein Code nach dem Start des SIPs nachträglich dynamisch geladen werden kann und somit der gesamte Code beim Start des SIPs bekannt sein muss. Dies erlaubt es, dass der gesamte Code statischen Analysen durch Verfikationswerkzeugen unterzogen werden kann.

Zusätzlich unterstützt das Singularity-OS die sogenannten HIPs. Dies sind Hardware-geschützte Domänen, in denen ein oder mehrere SIPs laufen können. Die Domänen werden mit Hilfe der MMU geschützt, wie es in den gängigen Betriebssystemen üblich ist. Dadurch wird die Untersuchung unterschiedlicher System-Konfigurationen ermöglicht, in denen verschiedene SIPs in einem HIP laufen. Somit kann das System konfiguriert werden, dass alle SIPs in einer Hardware-Domäne mit dem Kernel zusammen enthalten sind. In dieser Konfiguration teilen sich alle SIPs und der Kernel einen Adressraum und laufen im privilegierten Modus des Prozessors. Dies ermöglichte die Untersuchung, wie viel Performance durch die MMU und durch die Kontext-Wechsel der Hardware verloren gehen. M. Aiken *et al.* stellten bei ihren Analysen einen Performance-Gewinn von bis zu 33 % je nach Applikationsart fest [3]. Dabei untersuchten sie auch unterschiedliche System-Konfigurationen, bei denen die SIPs und der Kernel in unterschiedlich Domänen konfiguriert sind.

Contract-Based Channels

Neben dem neuen Konzept für den Adressraum-Schutz der Prozesse ist für Singularity-OS eine Sprachunterstützung für die IPC entwickelt worden [46]. Daten zwischen den

geschützten Prozessen werden im Singularity-OS mit Hilfe von bidirektionalen, asymmetrischen Kanälen ausgetauscht, über die nach einem definierten Vertrag (engl. *contract*) mit Hilfe von Nachrichten kommuniziert wird. Der Vertrag ist in der Programmiersprache Sing# geschrieben, in dem über einen endlichen Automaten definiert wird, welche Nachrichten in welchem Zustand des Kanals gesendet bzw. empfangen werden. Der Compiler kann mit Hilfe des Vertrags prüfen, ob die Nachrichten in der vorgeschriebenen Reihenfolge verschickt werden.

Für den Daten-Austausch zwischen SIPs muss sichergestellt werden, dass die SIPs keinen Zugriff auf die Daten eines anderen SIP erhalten. Dafür muss garantiert werden, dass der sendende SIP keinen Zugriff auf die Nachricht mehr hat, nachdem er diese versendet hat. Ebenfalls erhält der zu empfangende SIP erst Zugriff auf die Nachricht, wenn die Nachricht vollständig versendet wurde. Dies wird über den sogenannten Exchange Heap erreicht, um die Daten von einem Prozess in den anderen zu verschieben. Die Nachricht wird vom sendenden SIP im Exchange Heap angelegt und anschließend versendet. Dabei verliert er den Zugriff auf die Nachricht und der empfangende Prozess erhält den Zugang auf die gesendete Nachricht.

Manifest-based Programs

Die dritte architekturelle Neuerung im Singularity-Projekt sind die MBPs. Zu jedem Programm muss es ein Manifest geben, das die benötigten Code-Ressourcen, die benötigten System-Ressourcen, die gewünschten Berechtigungen und die Abhängigkeiten zu anderen Programmen beschreibt. Bevor eine Applikation ausgeführt wird, wird überprüft, ob alle notwendigen System-Ressourcen für die Ausführung zur Verfügung stehen.

Die MBPs wurden vor allem bei Treibern untersucht, die im Singularity-OS ebenfalls außerhalb des Kerns in eigenen SIPs laufen und als MBP installiert werden [141]. Durch das Manifest entstehen selbst beschreibende Treiber, durch die die Probleme der Monolithischen Betriebssysteme im Bezug auf die unzuverlässigen Treiber angegangen werden. Durch das Konzept der MBPs werden auch Ressourcen-Konflikte während der Laufzeit vermieden.

Dabei ist das Manifest keine zusätzliche Dokumentation des Treibers. Sondern es ist Maschinen-les- und überprüfbar. So wird während der Installation des Treibers mit Hilfe des Manifests garantiert, dass der Treiber für das System geeignet ist. Anschließend wird der Treiber optimiert, indem die MSIL in nativen Code übersetzt wird. Der übersetzte native Code wird anschließend in Verzeichnissen abgelegt und die notwendigen Informationen in der System-Beschreibung festgehalten.

Das Manifest wird allerdings beim Installationsprozess ebenfalls dazu genutzt, automatisch Code für die Allokation der System-Ressourcen zu generieren. Damit wird eine mögliche Fehlerquelle ausgeschlossen und dem Programmierer des Treibers Arbeit abgenommen. Dadurch wird garantiert, dass der Treiber auch nur diejenigen Ressourcen beansprucht, die auch im Manifest hinterlegt sind.

Zusätzlich ist der Singularity-Kern um zwei Aspekte erweitert worden. Zum einen erkennt er Ressourcen-Konflikte, bevor die Treiber ausgeführt werden, und zum anderen wird die Reihenfolge, in der die Treiber während des Boot-Vorgangs geladen werden, aus den Ressourcen-Beschreibungen in den Manifesten abgeleitet. Dadurch wird eine

erhöhte Zuverlässigkeit und Wartbarkeit des Singularity-Systems bezüglich der Treiber-Problematik in Monolithischen Systemen erreicht.

3.2.4 Komponenten-basierte Betriebssysteme

In den vorhergehenden Abschnitten sind die Monolithischen, die Microkernel-Betriebssysteme und das Singularity-Projekt von Microsoft Research vorgestellt worden. Diese haben alle den Ansatz eines sogenannten GPOS (engl. *general-purpose operating system*), d. h. dass das Betriebssystem nicht für eine bestimmte Anwendung konzipiert und optimiert worden ist, sondern dass es für verschiedene Anwendungen möglichst flexibel ausgelegt ist. Vor allem sind die Applikationen während der Konzeption des Betriebssystems nicht alle bekannt. Diese Flexibilität und Modularität ist in vielen Embedded Systemen nicht anwendbar, da die für die Verwaltung zusätzlich benötigten Ressourcen zu teuer sind. Dieser Zusammenhang ist bereits in Abschnitt 2.1 für die Automotiven Systeme, die eine Untermenge der Embedded Systeme sind, beschrieben worden.

Aus diesem Grund haben sich für die Embedded Systeme die Gruppe der ASOS (engl. *application-specific operating system*) etabliert, die speziell für eine Applikation oder Applikationsdomäne zugeschnitten sind [48]. Im Gegensatz zu den GPOS werden alle für die Applikation unnötigen Funktionen und Komponenten im Betriebssystem entfernt, so dass nur die für die Applikation benötigten Komponenten übrig bleiben. Dadurch wird eine höhere Performance und ein kleinerer Ressourcen-Bedarf erreicht, was für Embedded Systeme entscheidend ist.

Zu der Gruppe der ASOS gehören die Komponenten-basierten Betriebssysteme, die zur Entwicklungszeit aus vorgefertigten Komponenten zusammengesetzt werden. Dabei ist bereits vorab genau bekannt, welche Arten von Komponenten gebraucht werden, so dass nur die benötigten Komponenten im System enthalten sind. Zur Laufzeit kann dann in den seltensten Fällen eine dynamische Konfiguration und Erweiterung des Software-Systems erfolgen. TinyOS ist ein typischer Vertreter der Komponenten-basierten Betriebssysteme, das in WSNs eingesetzt wird und ausführlich als Fallbeispiel in Abschnitt 3.3 vorgestellt wird. Weitere Vertreter sind das Betriebsystem Pebble [50] und MMLite [72], die L. Friedrich *et al.* neben einer Übersicht über weitere Komponenten-basierte Betriebssysteme nennt [48].

Die GPOS bieten zwar auch die Möglichkeit die Betriebssystem-Kerne zur Entwicklungszeit zu konfigurieren. Jedoch ist die Konfiguration auf einer hohen Abstraktionsebene, z. B. ob das Betriebssystem eine Funktionalität wie z. B. die virtuelle Speicherverwaltung oder eine Netzwerk-Kommunikation unterstützen soll oder nicht. Diese Entscheidung wird bei Komponenten-basierten Betriebssysteme dagegen, wie es der Namen andeutet, auf der Komponenten-Ebene getroffen. Dabei stützen sie sich auf wiederverwendbare Komponenten, die über Schnittstellen genau definiert sind. Eines der wichtigen Werkzeuge bei den Komponenten-basierten Betriebssystemen sind die Konfigurationswerkzeuge, die genau die Komposition der Komponenten für das System analysieren und festlegen.

Die Komponenten enthalten eine vorgefertigte Funktionalität, die sie zur Verfügung stellen. Dabei sind sie die Funktionsblöcke, die zusammengesetzt das Gesamtsystem ergeben. Die Komponenten kapseln ihre Funktion und können nicht in kleinere Kom-

ponenten geteilt oder nur halb in das System integriert werden. Damit sie in verschiedenen Systemen wieder verwendet werden können, enthalten die Komponenten keinen persistenten Zustand, d. h. es kann nicht zwischen einer Komponente und deren Kopien unterschieden werden. Daraus ergibt sich, dass die Komponenten unabhängig von ihrem Kontext laufen, was eine Voraussetzung für eine schnelle Integration in neue Systeme ist. Im Umkehrschluss bedeutet dies wiederum, dass neue Systeme sehr schnell aus den bereits vorhandenen Komponenten erzeugt werden können.

Die Komponenten können sich in ihrer Größe und in ihrer Art der Implementierung unterscheiden. Für die Größe einer Komponente gibt es keine Vorgaben und auch die Frage nach einer optimalen Größe ist nicht klar zu beantworten. Denn die optimale Größe einer Komponente ist, wenn sie am besten wiederverwendet werden kann. Jedoch erhöht eine gute Wiederverwendbarkeit die Kontext-Abhängigkeit einer Komponente, was nach der Definition der Komponente vermieden werden sollte. Wenn alle benötigten Schnittstellen in einer Komponente implementiert werden, verringert dies die Kontext-Abhängigkeit und erhöht aber automatisch die Größe der Komponente. Jedoch geht damit auch die Modularität und die Wiederverwendbarkeit der Komponente verloren. Aus diesem Grund müssen beide Ziele ausbalanciert werden, um bei der richtigen Größe der Komponente zu landen.

Es ist nicht zwangsläufig notwendig, dass die Komponenten als White-Box-Komponenten mit ihrem Quelltext zur Verfügung stehen. Es ist durchaus möglich, dass die Komponenten als Black-Box-Komponenten in ein Gesamtsystem integriert werden. Dafür müssen die Schnittstellen und das Verhalten der Komponente bekannt und vollständig definiert sein. Auf diese Weise ist es möglich, Komponenten von Fremdherstellern in das eigene System zu integrieren, ohne dass der Quelltext der Komponente herausgegeben werden muss.

Um eine Komponente ausreichend zu spezifizieren, müssen deren Schnittstellen definiert werden. Dabei sind die Schnittstellen eine Ansammlung von Funktionen, die von anderen Komponenten aufgerufen werden können. Ein wichtiges Kriterium für die Schnittstelle ist, dass diese unabhängig von der Implementierung der Komponente spezifiziert werden muss, damit im Anschluss die Unabhängigkeit der Komponente garantiert werden kann. Über die Schnittstellen sind die Komponenten in einem System miteinander verbunden, womit die Schnittstellen auch als ein wohl-bekannter Vertrag zwischen den Komponenten angesehen werden können, der angibt, wie auf die Funktionalität der jeweiligen Komponente zugegriffen werden kann. Neben der funktionalen Beschreibung kann eine Schnittstelle auch die nicht-funktionalen Anforderungen an die Komponente enthalten, wie z. B. Angaben über die Reaktionszeiten oder die Performance.

Für die Komponenten-basierten Betriebssysteme sind in der Forschung verschiedene Frameworks entstanden, um die Komponenten-basierten Betriebssysteme zu analysieren. Ein Framework zum Erstellen eines Komponenten-basierten Betriebssystem-Kerns ist THINK, das aus der Komponenten-Bibliothek KORTEX verschiedene Betriebssystem-Kerne unterschiedlicher Größe bauen kann [47]. Ein weiteres Framework für Embedded Echtzeit-Systeme ist VEST (engl. *Virginia embedded systems toolset*), das an der Universität von Virginia bei J. Stankovic entstanden ist [145, 144]. Das Framework enthält neben den Werkzeugen für die Komposition des Betriebssystems auch Analyse-Werkzeuge, die die Zuverlässigkeit und die Echtzeit-Bedingungen des Embedded Systems prüfen.

Darunter fällt z. B. die Analyse nicht-funktionaler Anforderungen, wie die Einhaltung der Deadlines in dem Echtzeit-System.

3.3 Fallbeispiel TinyOS

Wie in der Einleitung in Abschnitt 1.1 beschrieben wurde, herrschen in der Domäne WSNs spezifische Anforderungen, die sich ebenfalls auf die Betriebssystem-Architektur auswirken. So wurden spezielle Betriebssysteme für die WSNs entwickelt, wie Contiki, das einen Event-getriebenen Microkernel besitzt und Code für die Programme dynamisch zur Laufzeit nachladen kann [43]. Bei der Entwicklung wurde auf die geringen Speicher-Ressourcen der Hardware geachtet und Contiki dahingehend optimiert. Jedoch kommen hier immer noch die Ansätze der Desktop- oder Server-Systeme zum Einsatz, die nur abgespeckt wurden, um den Hardware-Ressourcen in den WSNs zu genügen.

Da sich die Anforderungen jedoch stark von den Desktop- oder Server-Systeme unterscheiden, wurde bei der Entwicklung von TinyOS auf die Anforderungen der WSNs geachtet und TinyOS als Komponenten-basiertes Betriebssystem von Grund auf neu entwickelt [99]. TinyOS entstand aus einer Kooperation zwischen der University of California in Berkeley sowie den Firmen Intel Research und Crossbow Technology. Die erste Version von TinyOS wurde 2000 veröffentlicht. Für die stetige Weiterentwicklung bildete sich eine Gemeinschaft, der sowohl Partner aus dem Forschungsbereich als auch aus der Industrie angehören. Diese Gemeinschaft gliedert sich wiederum in mehrere Arbeitsgruppen, die sich um eine Standardisierung von TinyOS kümmern. Dabei spezifiziert die Gemeinschaft über sogenannte TEPs (engl. *TinyOS enhancement proposal*) die TinyOS-Komponenten und Schnittstellen. Über die letzten Jahre sind um die 40 TEPs entstanden, die u. a. die Bereiche Kommunikation, Datenerfassung oder Power-Management abdecken. Die TEPs werden in den Arbeitsgruppen über das Internet spezifiziert und haben aus diesem Grund kein Datum der Veröffentlichung [155].

Im Folgenden wird TinyOS näher und ausführlicher vorgestellt. Als erstes werden die Anforderungen erläutert, die ausschlaggebend für die Architektur von TinyOS waren, bevor auf das Komponenten-Modell in TinyOS eingegangen wird. In TinyOS kommt ein Task-Modell zum Einsatz, das im Anschluss im Abschnitt 3.3.3 erläutert wird. Zum Abschluss wird noch die Hardware-Abstraktion und die Schnittstellen von TinyOS beschrieben, um einen guten Einblick in die Architektur von TinyOS zu erlangen. Dies ist für das weitere Verständnis dieser Arbeit wichtig, da TinyOS in der PLASA-Plattform in drei von vier ECUs eingesetzt wird. Auf die Software-Architektur der PLASA-Plattform wird später in Abschnitt 5.3 detailliert eingegangen.

3.3.1 Anforderungen in TinyOS

Einsatzgebiet und Anwendungsfälle zwischen den Sensor-Knoten und den Desktop- oder Server-Systemen unterscheiden sich erheblich. Sogar zu anderen mobilen Embedded Systemen, wie Smartphones oder CE Geräten, differenzieren sich die Sensor-Knoten in ihren Anwendungsfällen. Sensor-Knoten bauen auf einer kleinen und kostengünstigen Hardware auf, die in einer großen Anzahl in der Umwelt ausgesetzt werden. Wie in Abschnitt 1.1 bereits beschrieben, vermessen sie dort die Umwelt oder übernehmen die Lokalisierung

von Objekten, um nur eine kleine Auswahl der Anwendungsfälle der Sensor-Knoten zu nennen.

Aus diesem Grund ergeben sich für die Sensor-Knoten andere Anforderungen, die es bei der System-Entwicklung zu beachten gilt. Aus diesem Grund wurden hauptsächlich die folgenden vier Anforderungen als Grundlage für das Design von TinyOS herangezogen [99]:

Eingeschränkte Hardware-Ressourcen. Da die Sensor-Knoten für ihre Anwendungen meist in sehr großer Stückzahl benötigt werden, werden in den Sensor-Knoten meist die stromsparenden und kostengünstigen 8-Bit Mikrocontroller eingesetzt, die nur einen Programm-Speicher zwischen 32 KB bis 256 KB haben. Der Arbeitsspeicher bewegt sich bei diesen Mikrocontroller nur zwischen 2 KB bis 8 KB SRAM. Aus diesem Grund muss TinyOS mit diesen geringen Speicher-Ressourcen umgehen können und dafür ausgelegt sein.

Reaktive Nebenläufigkeit. Ein Sensor-Knoten hat viele Aufgaben, die er nebenläufig ausführen muss. Neben dem Messen der Umwelt durch seine Sensoren, hat er noch weitere Aufgaben wie die Verarbeitung und teilweise Auswertung der gemessenen Daten. Im Anschluss muss er diese Ergebnisse noch über Funk an eine zentrale Stelle weiterleiten. Viele dieser Aufgaben sind an Echtzeit-Bedingungen geknüpft, wie das Vermessen der Umwelt oder das Versenden der Messergebnisse über Funk. Aus diesem Grund muss von Betriebssystem-Seite ein Mechanismus zur Verfügung gestellt werden, um diese beschriebene Nebenläufigkeit mit den Echtzeit-Anforderungen möglichst fehlerfrei bewältigen zu können.

Flexibilität. Durch den Einsatz unterschiedlichster Hardware und die Vielfalt der Applikationen auf Sensor-Knoten muss ein Konzept entwickelt werden, dass möglichst flexiblen Einsatz der Applikationen erlaubt. Dafür sollte ein Großteil der Software unabhängig von der Hardware und modular geschrieben sein, damit die Software für die verschiedenen Applikationen in den WSNs wiederverwendet werden kann.

Niedriger Energiebedarf. Während Desktop- und Server-Systeme ortsgebunden sind und eine ständige Stromversorgung besitzen, werden die Sensor-Knoten in freier Umwelt batteriebetrieben eingesetzt. Dabei besteht meistens nach deren Aussetzen keine Möglichkeit mehr, die Batterien wieder aufzuladen. Dies führt zu der Anforderung, dass die Sensor-Knoten besonders energiesparend sein müssen. Ebenfalls spielen die Mechanismen für ein Power-Management eine besondere Rolle im Betriebssystem, da dieses möglichst flexibel an den jeweiligen Zustand des Sensor-Knotens angepasst werden muss.

Um diese Anforderungen zu erfüllen, wurde das Konzept des Komponenten-basierten Betriebssystems entwickelt, was in TinyOS umgesetzt ist. Dafür wurde eigens die Sprache nesC definiert [52], mit der die Betriebssystem-Komponenten und die Applikationen für TinyOS programmiert werden [51]. Dabei ist nesC ein C-Dialekt, der die Sprache C um das Konzept der Komponenten und deren Schnittstellen erweitert. Ebenfalls enthält nesC Erweiterungen, um die Anforderungen der reaktiven Nebenläufigkeit zu erfüllen.

Dafür wurde ein Ereignis-getriebenes Ausführungsmodell entwickelt, in dem der Begriff des Tasks eine zentrale Rolle spielt. In Abschnitt 3.3.3 wird dieses Ausführungsmodell näher beschrieben. Doch zunächst wird im nächsten Abschnitt auf das Komponenten-Modell in TinyOS eingegangen, das die geforderte Modularität erfüllt und eine hohe Wiederverwendbarkeit der in nesC geschriebenen Software-Komponenten erlaubt.

3.3.2 Komponenten-Modell in TinyOS

Um die genannten Anforderungen in WSNs zu erfüllen, hat sich in TinyOS ein Komponenten-Modell entwickelt, das eine hohe Flexibilität und Wiederverwendbarkeit des geschriebenen Codes erlaubt. P. Levis und D. Gay beschreiben in ihrem Buch „TinyOS Programming“ [98] die Konzepte des Komponenten-basierten Betriebssystems sowie verschiedene Design Patterns, um die zu entwickelnde Software möglichst modular, flexibel und wiederverwendbar zu gestalten.

In TinyOS kommt, wie in Komponenten-basierten Betriebssystemen üblich, den Komponenten und den Schnittstellen eine große Bedeutung zu. Unter TinyOS wird die Funktionalität in Komponenten gegliedert, die genau spezifizierte Schnittstellen (engl. *interface*) entweder anbieten (engl. *provide*) oder benutzen (engl. *use*). Es ist dabei nur erlaubt, eine Komponente mit einer anderen Komponente über eine Schnittstelle zu verbinden (engl. *wire*), die die eine Komponente anbietet und die andere Komponente nutzt.

Schnittstellen in TinyOS

Die Schnittstellen in TinyOS beschreiben die funktionale Beziehung zweier Komponenten und besitzen zwei Arten von Funktionen: die einen sind Befehle (engl. *command*) und die anderen sind Ereignisse (engl. *event*). Der Unterschied zwischen Befehlen und Ereignissen ist, dass Befehle Aktionen ausführen, während Ereignisse Benachrichtigungen darstellen, die als asynchrone Rückantwort auf einen bereits ausgeführten Befehl kommen. Dadurch sind die Schnittstellen in TinyOS bidirektional und definieren eine asynchrone Kommunikation zwischen zwei Komponenten. Dies ist dem Umstand geschuldet, dass die Hardware/Software Schnittstelle der Mikrocontroller ebenfalls asynchron arbeitet. Für eine Aktion in der Hardware, z. B. für das Versenden eines Nachrichtenpakets durch einen Kommunikationscontroller, werden die Register des Controllers durch den Software-Treiber entsprechend gesetzt, um eine Daten-Einheit zu versenden. Im Anschluss springt der Treiber aus dem Befehl zurück in die aufrufende Routine und der Kommunikationscontroller versendet die Daten-Einheit asynchron. Während der Kommunikationscontroller den Befehl ausführt, können im Mikroprozessor andere Tasks ausgeführt werden. Sobald der Kommunikationscontroller die Übertragung abgeschlossen hat, signalisiert er mit einem Interrupt den Erfolg seiner Tätigkeit. Der Software-Treiber fängt den Interrupt ab und ruft in TinyOS ein Event auf, um der Komponente, die den Befehl für die Übertragung beauftragt hat, zu benachrichtigen, dass der Kommunikationscontroller die Daten-Einheit versendet hat. Dieser beschriebene Mechanismus wird in TinyOS auch split-phase genannt, wobei die Befehle im Regelfall immer nach unten und die Ereignisse nach oben in den Schichten der Software-Architektur aufgerufen werden.

```
1 interface Read<val_t>
2 {
3   command error_t read();
4   event void readDone( error_t result, val_t val );
5 }
```

Quelltext 3.1: Die Schnittstelle `Read` als Beispiel für eine Schnittstelle in TinyOS.

In Quelltext 3.1 wird als einfaches Beispiel die Schnittstelle `Read` dargestellt, die zum Lesen eines Sensor-Werts genutzt wird. Dafür wird in Zeile 3 der Befehl `read` definiert, der keine Parameter enthält und als Rückgabewert einen Fehler vom Typ `error_t` liefert. Die verknüpfte Komponente sorgt bei Aufruf des Befehls dafür, dass das Lesen des Sensor-Werts angestoßen wird. Die Messung wird im Anschluss asynchron von der Plattform ausgeführt. Sobald die Messung abgeschlossen ist, wird dies mit dem Ereignis `readDone`, das in Zeile 4 definiert wird, der aufrufenden Komponente signalisiert. Als Parameter des Ereignisses wird einen möglicherweise aufgetretenen Fehler vom Typ `error_t` und der gemessene Wert generischen Typs übergeben.

Wie in dem Beispiel der Schnittstelle `Read` können bei Bedarf in TinyOS die Schnittstellen generisch sein, indem diese Parameter besitzen. Die Parameter sind Variablen-Typen, die in den Befehls- und Ereignis-Definitionen genutzt werden können. Hier besitzt die Schnittstelle `Read` den Parameter `val_t`, der in spitzen Klammern gesetzt wird und anschließend im Ereignis `readDone` beim zweiten Parameter verwendet wird. Dies bedeutet, dass der Parameter für den Ergebniswert generisch ist. Erst mit der Benutzung der Schnittstelle `Read` in einer Komponente, wird der Typ des Ergebniswerts festgelegt. Werden zwei Komponenten in einer Konfiguration miteinander verbunden, so müssen die Parameter der Schnittstelle ebenfalls identisch sein. Auf diese Weise wird garantiert, dass die verbundenen Schnittstellen beider Komponenten kompatibel sind.

Die generischen Schnittstellen bieten die Möglichkeit die Einheit einer Messgröße in die Schnittstelle aufzunehmen. Dafür wird die Einheit als Parameter zu der Schnittstelle hinzugefügt. Allerdings wird der Parameter anschließend nicht in den Befehls- und Ereignisdefinitionen verwendet. Jedoch lassen sich keine zwei Komponenten miteinander verknüpfen, die nicht dieselben Schnittstellen-Parameter besitzen. Auf diese Weise lassen sich in TinyOS Einheiten in Schnittstellen hinzufügen, die dann ebenfalls beim Verbinden zweier Komponenten geprüft werden. Als Beispiel ist in Quelltext 3.2 die Schnittstelle `Timer` dargestellt, in der die Vorteile verdeutlicht werden.

In allen Befehls- und Ereignisdefinitionen der Schnittstelle `Timer` kommt der Parameter `precision_tag` nicht vor. Dennoch können keine zwei Komponenten miteinander verbunden werden, die nicht die gleiche Einheit haben. Auf diese Weise wird verhindert, dass eine Komponente, die als Einheit für den Timer Millisekunden erwartet, mit einer zweiten Komponente verbunden wird, die eine andere Einheit als Millisekunden verwendet.

Anhand der Beispiele `Read` und `Timer` wurden die Schnittstellen in TinyOS vorgestellt, die im Anschluss von Komponenten angeboten oder genutzt werden können. Im Folgenden wird auf die Komponenten und deren Definition in TinyOS eingegangen.

```
1  interface Timer<precision_tag>
2  {
3    command void startPeriodic(uint32_t dt);
4    command void startOneShot(uint32_t dt);
5    command void stop();
6    event void fired();
7    command bool isRunning();
8    command bool isOneShot();
9    command void startPeriodicAt(uint32_t t0, uint32_t dt);
10   command void startOneShotAt(uint32_t t0, uint32_t dt);
11   command uint32_t getNow();
12   command uint32_t gett0();
13   command uint32_t getdt();
14 }
```

Quelltext 3.2: Die Schnittstelle `Timer` als Beispiel für die Einheitsüberprüfung in TinyOS.

Komponenten in TinyOS

In TinyOS wird die Software in Komponenten aufgeteilt, die jeweils Schnittstellen anbieten oder nutzen. Somit wird in TinyOS ein Programm aus Komponenten als Bausteine zusammengesetzt. Die Schnittstellen sind dabei die genau definierte Signatur einer jeder Komponente. TinyOS unterscheidet zwischen zwei Arten von Komponenten. Auf der einen Seite gibt es Konfigurationen (engl. *configuration*), die selbst wieder Subkomponenten enthalten und die Verbindungen (engl. *wire*) zwischen den enthaltenen Subkomponenten festlegen. Auf der anderen Seite gibt es Module (engl. *module*), die die Befehle aller angebotenen Schnittstellen und die Ereignisse aller genutzten Schnittstellen implementieren.

Die Komponenten in TinyOS sind im Regelfall Singletons. Dies bedeutet, dass, wenn eine Komponente in mehreren Konfigurationen enthalten ist, immer dieselbe Komponente mit deren Daten verwendet wird. Zusätzlich kennt TinyOS generische Komponenten, die bei Bedarf in Konfigurationen erzeugt werden können. Dabei erstellt der nesC-Compiler aus der generischen Definition der Komponente eine Instanz, die im Anschluss in der Konfiguration verwendet wird. Auf die generischen Komponenten sowie die Konfigurationen und die Module wird im Folgenden weiter eingegangen.

Konfigurationen in TinyOS In TinyOS enthalten die Konfigurationen die sogenannten Verbindungen (engl. *wiring*) zwischen den Komponenten. Dabei können nur zwei Komponenten miteinander verbunden werden, wenn die eine Komponente dieselbe Schnittstelle nutzt, die die andere Komponente anbietet. Die Konfigurationen werden ebenfalls in nesC definiert [52]. In Quelltext 3.3 ist als Beispiel die Konfiguration `HilTimerMilliC` dargestellt.

Nach dem Schlüsselwort `configuration` in Zeile 3 folgt der Name der Konfiguration mit den angebotenen Schnittstellen, die in den Zeilen 4 bis 6 genannt werden. Das Schlüsselwort `implementation` trennt die Schnittstellen-Definition und die Implementierung der Konfiguration, die im Quelltext von Zeile 9 bis Zeile 28 steht. Mit dem

```
1  #include "Timer.h"
2
3  configuration HilTimerMilliC {
4    provides interface Init;
5    provides interface Timer<TMilli> as TimerMilli[uint8_t num];
6    provides interface LocalTime<TMilli>;
7  }
8  implementation {
9
10   enum {
11     TIMER_COUNT = uniqueCount(UQ_TIMER_MILLI)
12   };
13
14   components AlarmCounterMilli32C;
15   components new AlarmToTimerC(TMilli);
16   components new VirtualizeTimerC(TMilli, TIMER_COUNT);
17   components new CounterToLocalTimeC(TMilli);
18
19   Init = AlarmCounterMilli32C;
20
21   //LocalTime
22   LocalTime = CounterToLocalTimeC;
23   CounterToLocalTimeC.Counter -> AlarmCounterMilli32C.Counter;
24
25   //Timer
26   TimerMilli = VirtualizeTimerC;
27   VirtualizeTimerC.TimerFrom -> AlarmToTimerC.Timer;
28   AlarmToTimerC.Alarm -> AlarmCounterMilli32C.Alarm;
29 }
```

Quelltext 3.3: Definition einer Konfiguration in TinyOS.

Schlüsselwort `components` werden die enthaltenen Komponenten in der Konfiguration aufgelistet, deren Schnittstellen im Anschluss mit dem Operator `->` (Zeile 19 bis einschließlich Zeile 28) verbunden werden. Durch den Operator `=` werden die Schnittstellen der Konfiguration selbst auf Schnittstellen der enthaltenen Komponenten zugeordnet. In dem Beispiel `HilTimerMilliC` werden drei generische Komponenten verwendet, die mit dem Schlüsselwort `new` erzeugt werden. Die generischen Komponenten `AlarmToTimerC`, `VirtualizeTimerC` und `CounterToLocalTimeC` sind Komponenten, die in TinyOS als Bibliotheksfunktionen mitgeliefert werden. Da die Implementierungen dieser Module Hardware-unabhängig ist, können sie für verschiedene Hardware-Plattformen wiederverwendet werden. Das generische Modul `AlarmToTimerC` nutzt, wie es der Name bereits andeutet, die Schnittstelle `Alarm` um die Schnittstelle `Timer` zu realisieren. Gleich verhält es sich bei der generischen Komponente `CounterToLocalTimeC`, die die `Counter` Schnittstelle nutzt, um eine Schnittstelle `LocalTime` anzubieten. Das generische Modul `VirtualizeTimerC` virtualisiert aus einem Timer, der im Regelfall ein Hardware-Timer ist, mehrere Timer, die über die parametrisierte Schnittstelle im Anschluss zur Verfügung gestellt werden.

Die Konfigurationen können ebenfalls graphisch dargestellt werden, um einen besseren Überblick über die Konfiguration mit ihren enthaltenen Komponenten zu bieten.

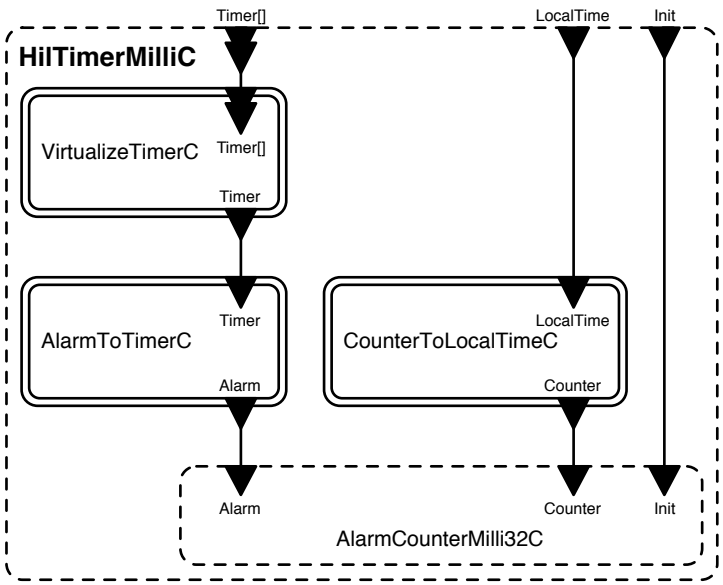


Abbildung 3.6: Graphische Darstellung einer Konfiguration in TinyOS.

In Abbildung 3.6 ist eine graphische Darstellung der Konfiguration `HilTimerMilliC` mit deren Verbindungen zu sehen. Komponenten mit durchgezogener Umrandung sind Module, wogegen die Komponenten mit unterbrochener Umrandung Konfigurationen sind. Falls es sich um eine generische Komponente handelt, ist diese mit einer doppelten Umrandung dargestellt. Die Dreiecke an den Umrandungen einer jeden Komponente sind die Schnittstellen, die eine Komponente anbietet bzw. benutzt. Bei Schnittstellen, die angeboten werden, zeigt das Dreieck nach innen, wogegen bei Schnittstellen, die benutzt werden, das Dreieck nach außen zeigt. Zusätzlich unterstützt TinyOS sogenannte parametrisierte Schnittstellen (engl. *parameterized interface*), die graphisch mit einem doppelten Dreieck dargestellt werden. Eine parametrisierte Schnittstelle bietet ihre Schnittstelle mehrfach an, wobei die konkrete Instanz der Schnittstelle über einen Index bestimmt wird. Die von der Konfiguration selbst angebotenen oder genutzten Schnittstellen sind an die Ränder der Konfiguration gesetzt. In der Abbildung 3.6 bietet die Konfiguration `HilTimerMilliC` drei Schnittstellen an, wobei die Schnittstellen `LocalTime` und `Init` einfache Schnittstellen sind und die Schnittstelle `Timer` eine parametrisierte Schnittstelle darstellt.

Auf diese Weise wird definiert, wie Komponenten miteinander verbunden werden. Die in einer Konfiguration enthaltene Komponente können wiederum Konfigurationen sein oder Module, die die angebotenen Schnittstellen implementieren.

Module in TinyOS Während die Verbindungen zwischen den Komponenten in Konfigurationen enthalten sind, werden die Implementierungen der Befehle und Ereignisse zusammen mit den benötigten Daten in Module gekapselt. Auf die in einem Modul enthaltenen Daten kann nur von den Befehlen oder Ereignissen innerhalb des Moduls zugegriffen werden.

```
1  generic module Atm1284pAlarmC(typedef frequency_tag, typedef timer_size
       @integer())
2  {
3    provides interface Alarm< frequency_tag, timer_size >;
4    uses interface HplAtm1284pCompare< timer_size > as Compare;
5    uses interface HplAtm1284pCounter< timer_size > as Counter;
6  }
7  implementation
8  {
9    async command void Alarm.start(timer_size dt)
10   {
11     call Alarm.startAt(call Counter.get(), dt);
12   }
13   /*
14     ...
15   */
16   async command void Alarm.startAt(timer_size t0, timer_size dt)
17   {
18     timer_size cv;
19     cv = t0 + dt;
20     call Compare.set(cv);
21     call Compare.reset();
22     call Compare.start();
23   }
24   /*
25     ...
26   */
27   async event void Compare.fired()
28   {
29     call Compare.stop();
30     signal Alarm.fired();
31   }
32 }
```

Quelltext 3.4: Definition des Moduls `Atm1284pAlarmC`.

Die Syntax von nesC in einem Modul ist eine Erweiterung von C, die in der Sprachspezifikation von nesC nachgelesen werden kann [52]. In Quelltext 3.4 ist das generische Modul `Atm1284pAlarmC` auszugsweise als Beispiel abgebildet. Die Definition leitet mit dem Schlüsselwort `module` und dem Modul-Namen ein. Da es sich bei `Atm1284pAlarmC` um ein generisches Modul handelt, ist noch das Schlüsselwort `generic` vorangestellt und nach dem Modul Namen folgen noch die Parameter, die in der Implementierung genutzt werden können. Die Parameter werden bei der Erstellung einer Instanz übergeben. Im Anschluss werden die angebotenen und genutzten Schnittstellen in der Zeile 3 bis einschließlich Zeile 5 genannt. Auf das Schlüsselwort `implementation` folgt ab Zeile 9 die Implementierung des Moduls. Alle Befehle aus den angebotenen Schnittstellen müssen mit dem Schlüsselwort `command` und alle Ereignisse aus den genutzten Schnittstellen mit dem Schlüsselwort `event` definiert werden. Nach dem Rückgabe-Parameter folgt die Schnittstelle und, mit einem Punkt getrennt, der Name des Befehls bzw. des Ereignisses, bevor die Parameter in runden Klammern folgen.

Befehle einer genutzten Schnittstelle werden mit dem Schlüsselwort `call` aufgerufen und Ereignisse einer angebotenen Schnittstelle werden mit dem Schlüsselwort `signal` signalisiert. Hier wird wiederum der Name der Schnittstelle vor dem Befehl bzw. Ereignisses vorangestellt, damit eindeutig definiert ist, welcher Befehl aufgerufen bzw. welches Ereignis signalisiert wird.

Neben den Befehlen und Signalen unterstützt nesC den C Sprachschatz, so dass in den Befehls- und Ereignis-Definitionen arithmetische Ausdrücke, Bedingungen und Schleifen genutzt werden können, um das Verhalten des Moduls zu programmieren.

Auf die beschriebene Weise werden Komponenten und deren Schnittstellen in TinyOS definiert und über die Konfigurationen genutzt. Eine Applikation in TinyOS ist eine Konfiguration, die als Start-Konfiguration dem nesC-Compiler übergeben wird. Dieser ermittelt über eine Tiefen-Suche alle benötigten Komponenten und generiert daraus C-Code, der mit Hilfe eines C-Compilers in Maschinen-Code für den genutzten Mikroprozessor übersetzt wird.

3.3.3 Task-Modell in TinyOS

Um die reaktive Nebenläufigkeit in den WSNs beherrschen zu können, wurde in TinyOS ein eigens für diese Umgebung angepasstes Ausführungsmodell entwickelt. Dieses basiert auf dem zentralen Begriff der Tasks, in denen die Applikationen in TinyOS ausgeführt werden. Die Tasks laufen dabei in einem synchronen Ausführungskontext ab und können sich dabei gegenseitig nicht unterbrechen. Somit werden die Tasks nicht präemptiv in TinyOS ausgeführt. Neben der synchronen Code-Ausführung unterscheidet TinyOS explizit den asynchronen Programm-Ablauf, in denen die ISRs (engl. *interrupt service routine*) ablaufen. Die Tasks können nur durch ISRs unterbrochen werden, die die Behandlung der Interrupts im asynchronen Kontext durchführen. Sobald die ISR vollständig durchlaufen wurde, wird der davor ausgeführten Task an der unterbrochenen Stelle wieder fortgesetzt.

Um längere Tasks zu unterbrechen, setzt TinyOS auf ein kooperatives Task-Modell, bei dem jeder Task freiwillig die Kontrolle an den nächsten Task abgibt. Falls ein Task längere Arbeiten ausführen muss, unterteilt er seine Arbeit in mehrere Schritte und übergibt am Ende eines jeden Schritts die Kontrolle zurück an den Scheduler. Jedoch ruft er sich vor seiner Beendigung wieder selbst auf, um bei seinem nächsten Aufruf mit dem nächsten Schritt seiner Berechnung fortzufahren.

Sobald ein Task seine Programmausführung abgegeben hat, wird der Scheduler in TinyOS aufgerufen, der in der Liste der anstehenden Tasks nach einer Strategie den als nächstes auszuführenden Task auswählt. Standardmäßig ist in TinyOS ein Scheduler nach dem Prinzip „First come, first served“ (FCFS) implementiert, der jedoch durch jede Implementierung einer anderen Strategie ausgetauscht werden kann [100]. Falls kein Task zur Ausführung bereitsteht, nutzt der Scheduler die Schnittstelle `McuSleep`, um den Mikrocontroller schlafen zu legen und damit Energie zu sparen. Die Schnittstelle `McuSleep` wird im nächsten Abschnitt näher vorgestellt.

Um die Tasks in TinyOS besser zu unterstützen, bietet nesC für deren Definition und deren Aufruf Spracherweiterungen an [52]. Tasks werden mit dem Schlüsselwort `task` wie ein Befehl oder Ereignis definiert und enthalten im Anschluss den auszuführenden

Programm-Code. Ein Task kann in einem Modul mit dem Schlüsselwort `post` erzeugt und in die Warteschlange der Tasks aufgenommen werden. Dabei ist jeder Task nur einmal in der Warteschlange vorhanden und kann nicht mehrmals gestartet werden.

Für die Synchronisierung der gemeinsamen Variablen bietet TinyOS ebenfalls ein Konzept an [98]. Dies zeichnet sich dadurch aus, dass bei der Definition einer Schnittstelle für jeden Befehl und jedes Ereignis festgelegt wird, ob er asynchron oder synchron ausgeführt wird. Standardmäßig laufen die Befehle und Ereignisse im synchronen Kontext. Soll ein Befehl oder Ereignis im asynchronen Kontext laufen, so muss dies mit dem Schlüsselwort `async` bei der Definition des Befehls oder des Ereignisses in der Schnittstelle gekennzeichnet werden. Dies führt zur folgenden Regel, dass synchrone Befehle oder Ereignissen keine asynchrone und umgekehrt keine asynchrone Befehle oder Ereignisse synchrone aufrufen können. Die Interrupt-Behandlung startet immer mit einem asynchronen Event. Um aus einer Interrupt-Behandlung in den synchronen Kontext zu springen, muss ein Task gestartet werden, der im synchronen Kontext ausgeführt wird.

Daraus lassen sich ebenfalls Regelungen für den Datenzugriff ableiten, um nicht in Probleme aufgrund von wechselseitigen Zugriffen auf die gemeinsamen Daten zu laufen. Falls synchrone Befehle oder Ereignisse ausgeführt werden, muss der Zugriff auf die gemeinsame Variable, die nur in synchronen Kontexten verwendet werden, nicht synchronisiert werden, da synchrone Befehle oder Ereignissen nur in Tasks laufen, die sich nicht gegenseitig unterbrechen können und immer zu Ende ausgeführt werden. Daraus ergibt sich, dass nur gemeinsame Daten, die auch in asynchronen Kontext verwendet werden, synchronisiert werden müssen. Dafür bietet nesC mit dem Schlüsselwort `atomic` atomare Code-Bereiche an, die ohne Unterbrechung durchlaufen werden. Nur in diesen atomaren Bereichen darf der Zugriff auf die gemeinsamen Variablen erfolgen, damit es bei wechselseitigen Zugriffen zu keinen Fehlern kommt.

3.3.4 Standardisierte TinyOS-Schnittstellen

In Abschnitt 3.3.2 wurde das Komponenten-Modell von TinyOS vorgestellt, woraus aus den einzelnen Komponenten eine TinyOS-Applikation erstellt wird. Ebenfalls wurde zu Beginn des Abschnitts über TinyOS die Anforderung der Flexibilität genannt, die in WSNs erforderlich ist, um mit den unterschiedlichen Hardware-Komponenten bereits geschriebene Applikationen wieder verwenden zu können. Die Komponenten in TinyOS sollten deshalb möglichst durch generische Schnittstellen wiederverwendbar und möglichst Hardware-unabhängig gestaltet sein. Doch es muss immer einen Hardware-abhängigen Teil in einem Betriebssystem geben, der unmittelbar auf der Hardware/-Software Schnittstelle aufbaut und den höheren Komponenten bis hin zur Applikation eine Hardware-unabhängige Schnittstelle bereitstellt. Um dies in TinyOS zu erfüllen und die Hardware-Unabhängigkeit für die Applikationen zu gewährleisten, definiert TinyOS die HAA (engl. *hardware abstraction architecture*) [66], die eine Schichtung der Komponenten in TinyOS vornimmt. Dabei werden in der HAA drei unterschiedliche Schichten definiert, die jeweils verschiedene Aufgaben übernehmen [65]:

HPL (engl. hardware presentation layer). Die HPL ist für die Präsentation der Hardware für die darüber liegende Schicht verantwortlich. Die darin enthaltenen Kom-

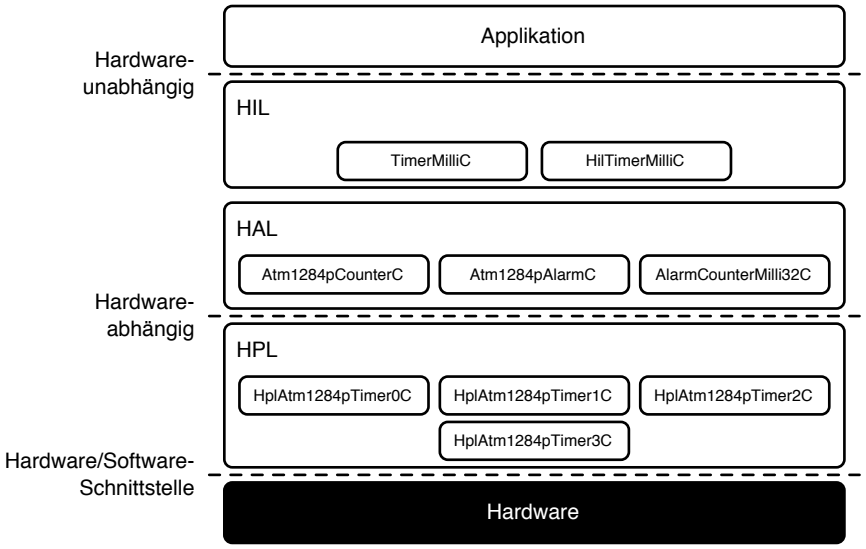


Abbildung 3.7: Die HAA von TinyOS mit der Applikationsschicht und der Hardware.

ponenten bieten eine sehr Hardware-bezogene Schnittstelle an, bei der meist die Hardware-Register gelesen und gesetzt werden. Dabei werden keine aufwendigen Mechanismen bereitgestellt, sonder die Schicht bietet nur den Zugriff auf die Hardware über eine definierte und über die Namen der Befehle und Ereignisse gut lesbare Schnittstelle an.

HAL (engl. hardware adaptation layer). Zwischen der HPL und der HIL befindet sich die HAL. Diese hat die Aufgabe, aus der sehr Hardware-nahen HPL den Schritt zur Hardware-unabhängigen HIL zu bewerkstelligen. Diese Schicht ist je nach Ausprägung der Hardware und der angebotenen Schnittstelle auf HIL-Ebene unterschiedlich dick.

HIL (engl. hardware interface layer). Die HIL-Schicht bietet eine Hardware-unabhängige Schnittstelle an, auf die die Applikationen zugreifen können. Dabei wird diese Schnittstelle über die verschiedenen Hardware-Konfigurationen einheitlich bereit gestellt, so dass bei Austausch der Hardware diese Schnittstelle stabil bleibt. Dafür muss die HIL-Schicht standardisiert werden. Dies geschieht in TinyOS über Arbeitsgruppen, die in den sogenannten TEPs die Schnittstellen spezifizieren, wie es schon im einführenden Abschnitt über TinyOS erklärt wurde. Auf diese Weise werden u. a. die Schnittstellen für den Timer, den ADC oder das Versenden und Empfangen von Nachrichten definiert.

In Abbildung 3.7 ist die Schichten-Architektur von TinyOS noch einmal graphisch an konkreten Komponenten dargestellt. Diese Strukturierung in Schichten ist für TinyOS entscheidend, um die Anforderung an eine flexible und modulare Architektur zu erfüllen, die möglichst für die unterschiedlichste Hardware wiederverwendbar und möglichst leicht erweiterbar ist. Als Beispiel ist ein Auszug aus der Timer-Schnittstelle dargestellt, die

ebenfalls im nächsten Abschnitt vorgestellt wird. Die Applikationen greifen auf der HIL-Schicht auf die Hardware-unabhängige Schnittstelle `Timer` zu, die von den Komponenten `TimerMilliC` und `HilTimerMilliC` angeboten werden. Die Komponenten `Atm1284pCounterC`, `Atm1284pAlarmC` und `AlarmCounterMilli32C` übersetzen in der HAL-Schicht die Hardware-unabhängige Schnittstelle `Timer` in die Hardware-abhängige Schnittstelle für den Atmel Mikrocontroller ATmega1284P. Auf der HPL-Schicht repräsentieren die vier Komponenten `HplAtm1284pTimer0C` bis `HplAtm1284pTimer4C` die vier Timer des ATmega1284P auf Hardware-Ebene. Diese Komponenten besitzen eine für den Atmel ATmega1284P spezifische Timer-Schnittstelle.

Den Applikationen ist es in TinyOS ebenfalls gestattet auf die HAL- oder HPL-Schicht zuzugreifen. Allerdings sind die Applikationen dann nicht mehr Hardware-unabhängig und müssen entsprechend an die Plattform angepasst werden. Dennoch ist dies in manchen Situationen sinnvoll, wenn für einen Dienst oder Hardware-Komponente keine Hardware-unabhängige HIL-Schicht vorhanden ist oder aus Performance-Gründen direkte Aufrufe notwendig sind.

Im Folgenden werden nun die wichtigsten Schnittstellen in TinyOS vorgestellt, die u. a. für die Realisierung der PLASA-Plattform benötigt werden.

Timer unter TinyOS

Die Timer sind einer der wichtigsten Hardware-Komponenten in Sensor-Knoten, da sie typischerweise einen periodischen Ablauf haben. Sie wachen periodisch auf, nehmen alle Messwerte auf und schicken diese im Anschluss drahtlos an den nächsten Knoten weiter. Daraufhin legen sie sich bis zu ihrem nächsten periodischen Aufwachen wieder schlafen. Somit spielen die Timer in Sensor-Knoten eine wichtige Rolle, weshalb die Schnittstellen für die Timer im TEP 102 standardisiert worden sind [136].

Die Schnittstelle `Timer`, die in Quelltext 3.5 abgebildet ist, spezifiziert die Timer unter TinyOS. Mit den Befehlen `startOneShot` und `startPeriodic` wird der Timer für ein einziges Mal oder periodisch gestartet. Ein gestarteter Timer kann mit dem Befehl `stop` wieder gestoppt werden. Sobald ein Timer abgelaufen ist, wird dies über das Ereignis `fired` signalisiert.

```
1  interface Timer<precision_tag>
2  {
3    command void startPeriodic(uint32_t dt);
4    command void startOneShot(uint32_t dt);
5    command void stop();
6    event void fired();
7    command bool isRunning();
8    command bool isOneShot();
9    command void startPeriodicAt(uint32_t t0, uint32_t dt);
10   command void startOneShotAt(uint32_t t0, uint32_t dt);
11   command uint32_t getNow();
12   command uint32_t gett0();
13   command uint32_t getdt();
14 }
```

Quelltext 3.5: Die Schnittstelle `Timer` für die Abstraktion der Timer.

Über diese Grundfunktionalität hinaus sind weitere Befehle für die Timer definiert, um z. B. Abfragen über einen Timer zu erlauben. So ist es möglich, mit den Befehlen `isRunning` oder `isOneShot` abzufragen, ob ein Timer gerade aktiv ist oder ob er für ein einzelnen Durchlauf gestartet wurde.

Neben den oben beschriebenen Befehlen `startOneShot` und `startPeriodic` bietet die Schnittstelle `Timer` noch die Befehle `startOneShotAt` und `startPeriodicAt` an, um zu einem bestimmten Zeitpunkt `t0`, der üblicherweise in der Vergangenheit liegt, einen Timer für einen weiteren Durchlauf bzw. periodisch zu setzen. Die drei Befehle `getNow`, `gett0` und `getdt` liefern dabei die aktuellen Werte des Timers für die aktuelle Zeit, den Anker `t0` oder die Verzögerung `dt` zurück.

Da die Hardware-Timer in den meisten Mikrocontrollern auf zwei bis vier Timer beschränkt sind, hat TinyOS die Timer virtualisiert, um mit einem Hardware-Timer den Applikationen mehrere virtuelle Timer zur Verfügung zu stellen. Dafür definiert das TEP 102 auf der HIL Schicht die Komponente `HilTimerMilliC`, die auf jeder Plattform vorhanden sein muss. Im Quelltext 3.6 wird die Konfiguration `HilTimerMilliC` dargestellt, wobei die Implementierung der Konfiguration für jede Plattform spezifisch ist. Auf dieser Komponente setzen die Hardware-unabhängigen System-Komponenten von TinyOS auf, um virtuelle Timer zu realisieren. Wie im Quelltext 3.6 ersichtlich ist, bietet die Konfiguration `HilTimerMilliC` die drei Schnittstellen `Init`, `Timer` und `LocalTime` an.

```
1  configuration HilTimerMilliC
2  {
3    provides interface Init;
4    provides interface Timer<TMilli> as TimerMilli[ uint8_t num ];
5    provides interface LocalTime<TMilli>;
6  }
7  implementation
8  {
9    //Plattform-spezifische Implementierung
10 }
```

Quelltext 3.6: Die Konfiguration `HilTimerMilliC` für die Realisierung virtueller Timer.

Schnittstellen für Quellen- und Senken-unabhängige Treiber

Über die Timer können Applikationen sich periodisch wecken lassen, um z. B. Sensor-Werte zu erfassen. Dafür sind in TinyOS ebenfalls generische Schnittstellen definiert, wobei auf die Unabhängigkeit der Schnittstellen von konkreten Daten-Quellen geachtet wurde. In diesem Zusammenhang sind auch die Schnittstellen für Daten-Senken definiert worden. TEP 114 fasst die Definition der Schnittstellen für Daten-Quellen, worunter auch die Sensoren fallen, und für Daten-Senken zusammen [157].

Für Daten-Quellen definiert TEP 114 die Schnittstellen `Read`, `ReadStream`, `ReadNow` und `Get`, die im Folgenden vorgestellt werden.

Die Schnittstelle `Read`, die in Quelltext 3.7 dargestellt ist, erlaubt es, die Daten von einer Quellen abzuholen. Dafür ist der Befehl `read` definiert, der das Auslesen der Daten

```
1  interface Read<val_t>
2  {
3    command error_t read();
4    event void readDone( error_t result, val_t val );
5  }
```

Quelltext 3.7: Die Schnittstelle `Read` für das Auslesen eines Daten-Werts.

anstößt. Sobald die Quelle die Daten bereitstellen kann, wird dies mit dem Ereignis `readDone` signalisiert, das auch den ermittelten Daten-Wert als Parameter enthält.

```
1  interface ReadStream<val_t>
2  {
3    command error_t postBuffer( val_t* buf, uint16_t count );
4    command error_t read( uint32_t usPeriod );
5    event void bufferDone( error_t result,val_t* buf, uint16_t count );
6    event void readDone( error_t result );
7  }
```

Quelltext 3.8: Die Schnittstelle `ReadStream` für das periodische Auslesen eines Daten-Werts.

Um mehrere Werte in einer definierten Periode ohne die Änderung der Konfiguration ermittelt zu können, definiert TEP 114 die Schnittstelle `ReadStream`, die in Quelltext 3.8 abgebildet ist. Über den Befehl `postBuffer` wird ein Puffer übergeben, den die Daten-Quelle mit den ermittelten Daten-Werten füllen soll. Das Ereignis `bufferDone` signalisiert, wenn der übergebene Puffer mit den Daten gefüllt ist. Mit dem Befehl `read` wird im Anschluss die Daten-Ermittlung mit der gewünschten Periode gestartet. Ein währenddessen auftretender Fehler wird mit dem Ereignis `readDone` signalisiert, wobei als Parameter der Fehlercode zurückgegeben wird.

```
1  interface ReadNow<val_t>
2  {
3    async command error_t read();
4    async event void readDone( error_t result, val_t val );
5  }
```

Quelltext 3.9: Die Schnittstelle `ReadNow` für das Auslesen eines Sensor-Werts im asynchronen Kontext.

Falls jedoch das Auslesen einer Daten-Quelle im asynchronen Kontext benötigt wird, bietet TinyOS noch die Schnittstelle `ReadNow` an, die in Quelltext 3.9 abgebildet ist. Sie enthält wie die synchrone Schnittstelle `Read` den Befehl `read` und das Ereignis `readDone`. Jedoch sind der Befehl und das Ereignis für den asynchronen Kontext definiert, was nur einen Aufruf in asynchronen Funktionen erlaubt. Damit ist es möglich, die Daten-Quellen ohne einen Wechsel in den synchronen Kontext auszulesen, was zu einer schnelleren Verarbeitung der ermittelten Daten-Werte führt.

Anders verhält es sich mit Daten-Quellen, die sofort ihre Daten-Werte liefern können.

Dafür muss von den höheren Schichten keine Daten-Erfassung angestoßen werden, sondern der Daten-Wert kann mit dem Befehl `get` aus der Schnittstelle `Get`, die in Quelltext 3.10 abgebildet ist, sofort abgefragt werden.

```
1 interface Get<val_t>
2 {
3   command val_t get();
4 }
```

Quelltext 3.10: Die Schnittstelle `Get` für das Auslesen eines Daten-Werts.

Neben der Schnittstelle `Get` definiert TEP 114 zusätzlich die Schnittstelle `Notify`, um die Änderung eines Daten-Werts zu signalisieren. Die Schnittstelle `Notify` ist in Quelltext 3.11 dargestellt. Mit dem Ereignis `notify` wird die Änderung des Daten-Werts signalisiert, was entsprechend mit den Befehlen `enable` und `disable` ein- bzw. ausgeschaltet werden kann.

```
1 interface Notify<val_t>
2 {
3   command error_t enable();
4   command error_t disable();
5   event void notify( val_t val );
6 }
```

Quelltext 3.11: Die Schnittstelle `Notify` für das Benachrichtigen über eine Änderung eines Daten-Werts.

Für Daten-Senken spezifiziert TEP 114 nur die Schnittstelle `Set`, die im Quelltext 3.12 abgebildet ist. Diese enthält nur den Befehl `set`, mit dem ein Daten-Wert in die Daten-Senke geschrieben werden kann.

```
1 interface Set<val_t>
2 {
3   command void set( val_t val );
4 }
```

Quelltext 3.12: Die Schnittstelle `Set` für das Schreiben eines Daten-Werts.

ADC-spezifische Schnittstellen in TinyOS

Für Sensor-Knoten ist der ADC neben dem Mikrocontroller einer der wichtigsten Hardware-Bestandteile. Dieser wandelt die analogen Eingangssignale in digitale Werte um, wie es in Abschnitt 2.3.3 über die Sensoren beschrieben ist. Diese werden anschließend weiter verarbeitet und über das drahtlose Netzwerk geschickt. Bevor jedoch die Daten vom ADC gelesen werden können, muss dieser konfiguriert werden. Für die Hardware-unabhängigen Konfiguration eines ADCs spezifiziert TEP 101 die Schnittstelle `AdcConfigure` [67], mit der die unterschiedlichen Parameter eines ADCs, wie der

Kanal oder die Referenz-Spannung eingestellt werden können. Dafür bietet die Schnittstelle `AdcConfigure` den Befehl `getConfiguration` an, mit dem die höheren Schichten die gewünschten Parameter des ADCs konfigurieren können. Wie im Quelltext 3.13 dargestellt ist, wird die Konfiguration als Schnittstellen-Parameter übergeben, der je nach Hardware unterschiedlich sein kann.

```
1 interface AdcConfigure<config_type>
2 {
3   async command config_type getConfiguration();
4 }
```

Quelltext 3.13: Die Schnittstelle `AdcConfigure` für die Konfiguration eines ADCs.

Nachdem der ADC konfiguriert wurde, können die höheren Schichten über die bereits vorgestellten Schnittstellen `Read`, `ReadStream` und `ReadNow` auf die Wandlungsergebnisse des ADCs zugreifen.

Kommunikationsschnittstellen in TinyOS

Eine der Aufgaben von Sensor-Knoten ist, neben der Messung der Umwelt, das drahtlose Versenden und Weiterleiten der Messwerte. Um die in den WSNs herrschenden Anforderungen im Bereich der Kommunikation zu erfüllen, spezifiziert TinyOS ebenfalls Schnittstellen für das Versenden und Empfangen von Nachrichten. Für die Protokolle, die Sensor-Knoten verwenden, hat TinyOS den Begriff der *Active Message* eingeführt, die sich in zwei TEPs wiederfinden. Zum einen wird die Nachrichtenstruktur `message_t` in TEP 111 definiert, die eine Repräsentation eines Nachrichtenpaket in TinyOS darstellt [96]. Zum anderen werden Plattform-unabhängige Schnittstellen in TEP 116 spezifiziert, über die Applikationen Nachrichtenpakete empfangen und verschicken können [97]. Zusätzlich werden Schnittstellen für die Applikationen definiert, über die Hardware- und Nachrichten-unabhängig auf den Paketinhalt, auf die Quell- und Ziel-Adressen der Nachrichtenpakete zugegriffen werden kann. Im Weiteren wird kurz auf die Nachrichtenstruktur `message_t` eingegangen, bevor die Schnittstellen zum Senden und Empfangen der Nachrichtenpakete näher beschrieben werden.

Nachrichtenpakte in TinyOS Um in TinyOS einen möglichst Plattform-unabhängigen Zugriff auf die Nachrichtenpakete zu erlauben, wird die Nachrichtenstruktur `message_t` definiert, die in allen Kommunikationsbefehlen und Ereignissen als Parameter verwendet wird. Die Struktur `message_t` ist dabei so ausgelegt, dass sie alle existierenden Datenpakete mit unterschiedlicher Länge aufnehmen kann. Ebenfalls wurde darauf geachtet, dass sie für die unterschiedlichen Kommunikationsschichten alle möglichen Protokolle unterstützen kann.

Um den Applikationen dennoch einen möglichst einfachen Zugriff auf die Nachrichten-Pakete zu erlauben, definiert TinyOS die Schnittstellen `Packet` und `AMPacket` [97], die einen Plattform- und Nachrichten-unabhängigen Zugriff auf die Pakete erlauben. Für den Zugriff auf den Inhalt einer Nachricht ist die Schnittstelle `Packet` spezifiziert worden, die im Quelltext 3.14 dargestellt ist. Über die Befehle `payloadLength`

und `maxPayloadLength` kann die aktuelle bzw. die maximale Länge des Nachrichten-Inhalts ermittelt werden. Mit dem Befehl `getPayload` wird ein Zeiger auf den Paketinhalt zurückgegeben, der für das Auslesen oder Schreiben der Nachricht verwendet werden kann. Für das Setzen der Länge des Nachrichten-Inhaltes sorgt der Befehl `setPayloadLength`. Mit dem Befehl `clear` kann der Inhalt eines Nachrichtenpakets komplett gelöscht werden, um das Paket für eine neue Nachricht wieder zu verwenden.

```
1  interface Packet {
2    command void clear(message_t* msg);
3    command uint8_t payloadLength(message_t* msg);
4    command void setPayloadLength(message_t* msg, uint8_t len);
5    command uint8_t maxPayloadLength();
6    command void* getPayload(message_t* msg, uint8_t len);
7  }
```

Quelltext 3.14: Die Schnittstelle `Packet` für den standardisierten Zugriff auf ein Nachrichtenpaket.

Für die Adress-orientierte Kommunikation ist zusätzlich die Schnittstelle `AMPacket` definiert worden, die im Quelltext 3.15 dargestellt ist. Mit Hilfe der Befehle `address` und `localGroup` kann die Netzwerkadresse des Sensor-Knotens bzw. die Gruppe der Kommunikationsschnittstelle abgefragt werden. Für das Abfragen und Setzen der Ziel- bzw. Quell-Adresse des Pakets sind in der Schnittstelle die Befehle `destination` und `setDestination` bzw. `source` und `setSource` enthalten. Der Typ bzw. die Gruppe der Nachricht kann über `type` bzw. `group` abgefragt und über `setType` bzw. `setGroup` gesetzt werden. Der Befehl `isForMe` ermittelt, ob das Nachrichtenpaket an den eigenen Sensor-Knoten gesendet wurde oder ob die Nachricht weitergeleitet werden muss.

```
1  interface AMPacket
2  {
3    command am_addr_t address();
4    command am_addr_t destination(message_t* amsg);
5    command am_addr_t source(message_t* amsg);
6    command void setDestination(message_t* amsg, am_addr_t addr);
7    command void setSource(message_t* amsg, am_addr_t addr);
8    command bool isForMe(message_t* amsg);
9    command am_id_t type(message_t* amsg);
10   command void setType(message_t* amsg, am_id_t t);
11   command am_group_t group(message_t* amsg);
12   command void setGroup(message_t* amsg, am_group_t grp);
13   command am_group_t localGroup();
14 }
```

Quelltext 3.15: Die Schnittstelle `AMPacket` für die Adress-orientierte Kommunikation.

Über die beiden Schnittstellen `Packet` und `AMPacket` erlaubt TinyOS einen standardisierten Zugriff auf die Nachrichtenpakete, die über die Kommunikationsschnittstellen versendet werden können.

Sende- und Empfangsschnittstellen unter TinyOS Nachdem die interne Repräsentation eines Nachrichtenpakets in TinyOS und die Schnittstellen für den Zugriff darauf beschrieben worden sind, wird im Folgenden das Plattform-unabhängige Empfangen und Versenden von Nachrichten vorgestellt, das in TEP 116 spezifiziert wird.

Für den Empfang von Nachrichten ist in TinyOS die Schnittstelle `Receive` eingeführt worden, mit der die Applikationen über den Empfang einer Nachricht informiert werden. Wie im Quelltext 3.16 dargestellt ist, enthält die Schnittstelle `Receive` das Ereignis `receive`, über das der Komponente der Erhalt einer Nachricht signalisiert wird. Als Parameter wird die Nachricht als `message_t` übergeben, auf das die Komponente im Anschluss über die Schnittstellen `Packet` oder `AMPacket` zugreifen kann.

```
1 interface Receive
2 {
3   event message_t* receive(message_t* msg, void* payload, uint8_t len);
4 }
```

Quelltext 3.16: Die Schnittstelle `Receive` für das Empfangen von Nachrichtenpaketen.

Das Senden erfolgt über die Schnittstellen `Send` und `AMSend`. Beide Schnittstellen bieten den Befehl `send` für das Versenden einer Nachrichten an. Der Unterschied liegt in den unterschiedlichen Modi der Adressierung. Während die Schnittstelle `Send` für die Adress-freien Protokolle wie das CTP (engl. *collection tree protocol*) [57] verwendet wird, kommt die Schnittstelle `AMSend` für die Adress-orientierte Kommunikation zum Einsatz. Die Schnittstelle `Send` ist in Quelltext 3.17 abgebildet, während das Interface `AMSend` im Quelltext 3.18 zu sehen ist.

```
1 interface Send
2 {
3   command error_t send(message_t* msg, uint8_t len);
4   command error_t cancel(message_t* msg);
5   event void sendDone(message_t* msg, error_t error);
6
7   command uint8_t maxPayloadLength();
8   command void* getPayload(message_t* msg, uint8_t len);
9 }
```

Quelltext 3.17: Die Schnittstelle `Send` für das Adress-freie Versenden von Nachrichten.

Beide Schnittstellen haben die gleichen Befehle und das gleiche Ereignis. Sie unterscheiden sich darin, dass bei der Schnittstelle `AMSend` beim Befehl `send` noch die Zieladresse der Nachricht übergeben wird. Aufgrund des gleichen Verhaltens der Befehle und des Ereignisses werden die Befehle beider Schnittstellen im Folgenden erklärt. Über den Befehl `send` wird das Senden eines Nachrichten-Pakets veranlasst, was über den Befehl `cancel` bei Bedarf abgebrochen werden kann. Sobald das Senden des Nachrichten-Pakets erfolgreich war, wird die Komponente über das Ereignis `sendDone` darüber informiert. Die beiden Befehle `maxPayloadLength` und `getPayload` sind für den einfacheren Zugriff auf die Nachricht enthalten. Somit muss die nutzende Komponente nicht zusätzlich mit der Schnittstelle `Packet` verbunden werden, um auf den Paketinhalt zugreifen zu

können. Der Befehl `maxPayloadLength` gibt damit die maximale Paketlänge zurück, während der Befehl `getPayload` einen Zeiger auf den Inhalt des Pakets liefert.

```
1  interface AMSend
2  {
3    command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
4    command error_t cancel(message_t* msg);
5    event void sendDone(message_t* msg, error_t error);
6
7    command uint8_t maxPayloadLength();
8    command void* getPayload(message_t* msg, uint8_t len);
9  }
```

Quelltext 3.18: Die Schnittstelle `AMSend` für das Adress-orientierte Versenden von Nachrichten.

Um die Kommunikationsschnittstellen der Applikation zur Verfügung zu stellen, wird in TEP 116 die Komponente `ActiveMessageC` auf der HIL-Ebene definiert, die die Schnittstellen für das Versenden und Empfangen von Nachrichten bereitstellt [97]. Im Quelltext 3.19 ist die Signatur der Konfiguration `ActiveMessageC` abgebildet. Diese stellt die zuvor beschriebenen Schnittstellen `Packet`, `AMPacket`, `AMSend` und `Receive` den Anwendungen zur Verfügung. Über die Schnittstelle `PacketAcknowledgements`, die nicht näher beschrieben wurde, können die Paket-Bestätigungen für den Sensor-Knoten gesteuert werden.

```
1  configuration ActiveMessageC
2  {
3    provides {
4      interface Init;
5      interface SplitControl;
6
7      interface AMSend[uint8_t id];
8      interface Receive[uint8_t id];
9      interface Receive as Snoop[uint8_t id];
10
11     interface Packet;
12     interface AMPacket;
13     interface PacketAcknowledgements;
14   }
15 }
```

Quelltext 3.19: Die Konfiguration `ActiveMessageC` für den Zugriff auf die Kommunikationsinfrastruktur auf HIL-Ebene.

Auffällig bei der Komponente `ActiveMessageC` ist, dass `AMSend` und `Receive` eine parametrisierte Schnittstelle haben. Als Parameter wird der AM-Pakettyp übergeben, mit der die Nachricht versendet oder empfangen werden soll. Auf diese Weise wird in der Konfiguration der AM-Pakettyp festgelegt, der die Komponente mit der Konfiguration `ActiveMessageC` miteinander verbindet. Dies lässt eine modulare Konfiguration für die Kommunikation zu. Denn die Komponenten, die für das Versenden oder Empfangen der

Nachrichten zuständig sind, beschränken sich hierbei um den Paketinhalt und müssen nicht den AM-Pakettyp festlegen oder kennen. Dies übernimmt eine Konfiguration in der Applikation, die an einer zentralen Stelle alle Kommunikationskomponenten mit der Komponente `ActiveMessageC` verbindet und die AM-Pakettypen durch die Verbindungen festlegt. Bei einer Änderung der AM-Pakettypen muss nur die zentrale Konfiguration angepasst werden und nicht alle Kommunikationskomponenten.

Neben dem `Receive` gibt es noch die Schnittstelle `Snoop`, mit der ebenfalls Pakete empfangen werden können. Der Unterschied zu der Schnittstelle `Receive` besteht darin, dass alle Nachrichten hier empfangen werden können, auch diejenigen, die nicht an den Sensor-Knoten selbst gerichtet sind. Dies ist für die Weiterleitung eingehender Nachrichten notwendig, was ebenfalls eine Aufgabe eines Sensor-Knotens ist.

Neben den Schnittstellen für das Senden und Empfangen von Nachrichten enthält die Konfiguration `ActiveMessageC` ebenfalls die Schnittstellen `Packet` und `AMPacket`, die für den Zugriff auf die Nachrichtenpakete zuständig sind. Somit bietet die Komponente `ActiveMessageC` für die Applikationen die notwendigen Hardware- und Plattformunabhängigen Schnittstellen für den Zugriff auf die Nachrichten an. Als Folge müssen die Applikationen nur den Aufbau des Paketinhalts wissen, für den sie verantwortlich sind.

Für die Initialisierung der Kommunikationskomponenten ist die Schnittstelle `Init` in der Konfiguration `ActiveMessageC` enthalten. Dies sorgt für die Initialisierung der gesamten Komponenten, die für die Kommunikation benötigt werden. Im Zuge der Initialisierung werden auch die Komponenten der HAL- und HPL-Schicht initialisiert, damit im Anschluss der Sensor-Knoten kommunikationsbereit ist.

Zusätzlich lässt sich die Kommunikation eines Sensor-Knotens über die Schnittstelle `SplitControl` ein- und ausschalten. Damit ist es der Applikationsschicht möglich, die Kommunikation des Sensor-Knotens komplett abzuschalten, wenn diese nicht mehr benötigt wird, um Energie einzusparen. Die Schnittstelle `SplitControl` ist Teil der Power-Management Schnittstellen in TinyOS, die im Folgenden näher vorgestellt wird.

Power-Management Schnittstellen in TinyOS

Wie in den einleitenden Kapiteln über TinyOS beschrieben wurde, spielt das Power-Management in Sensor-Knoten eine entscheidende Rolle, um eine möglichst lange Laufzeit der ausgebrachten Sensor-Knoten zu gewährleisten. Dafür sind in der TinyOS-Architektur ebenfalls Mechanismen und Schnittstellen entwickelt worden, die im Folgenden beschrieben werden. Zum einen sind zwei generische Schnittstellen für das Power-Management von nicht-virtualisierten Geräten im TEP 115 standardisiert worden, die das Starten und Stoppen von Geräten erlauben [88]. Zum anderen ist ein Mechanismus in TinyOS integriert, der es erlaubt, den Mikrocontroller immer in den Energie-ärmsten Schlaf-Zustand zu setzen, sobald keine Tasks mehr ausgeführt werden müssen. Für diesen Mechanismus, auf den auch die Treiber und Applikationen Einfluss nehmen können, definiert TEP 112 eine Schnittstelle für das Power-Management des Mikrocontrollers [147].

Für das Starten und Stoppen von Geräten unter TinyOS sorgen die zwei Schnittstellen `StdControl` und `SplitControl` [88]. Im Quelltext 3.20 ist die Schnittstelle `StdControl` abgebildet. Sie definiert die zwei Befehle `start` und `stop`, die jeweils das Gerät ein-

bzw. ausschalten. Bei dem Rücksprung aus dem Befehl ist das Gerät je nach Wunsch bereits ein- bzw. ausgeschaltet.

```
1  interface StdControl
2  {
3    command error_t start();
4    command error_t stop();
5  }
```

Quelltext 3.20: Die Schnittstelle `StdControl` für das Ein- und Ausschalten von Geräten.

Falls Hardware-bedingt eine Split-Phase für das Ein- bzw. Ausschalten notwendig ist, muss die Schnittstelle `SplitControl`, die im Quelltext 3.21 dargestellt ist, verwendet werden. Hier wird mit dem Befehl `start` der Einschaltvorgang gestartet, der mit dem Ereignis `startDone` vollständig ausgeführt wurde. Gleich verhält es sich mit dem Ausschaltvorgang. Mit dem Befehl `stop` wird dieser initiiert und das Ereignis `stopDone` bestätigt das vollständige Ausschalten.

```
1  interface SplitControl
2  {
3    command error_t start();
4    event void startDone(error_t error);
5    command error_t stop();
6    event void stopDone(error_t error);
7  }
```

Quelltext 3.21: Die Schnittstelle `SplitControl` für das Ein- und Ausschalten von Geräten mit einer Split-Phase.

Mit diesen beiden Schnittstellen können die höheren Schichten bis hin zur Applikation den Power-Zustand der Hardware steuern, indem sie Dienste ein- bzw. ausschalten. Als Beispiel wurde bereits in Quelltext 3.19 der Kommunikationsdienst mit der Komponente `ActiveMessageC` genannt, den die Applikationen über die Schnittstelle `SplitControl` steuern. Die Kommunikationskomponente schaltet ebenfalls über diese Schnittstelle die tiefer liegenden Komponenten. Am Ende erhält der Treiber auf der HPL-Schicht den Befehl für das Ein- bzw. Ausschalten der Hardware. Ebenfalls ist es darüber möglich, dass höhere Schichten, die mehrere Hardware-Ressourcen verwalten, alle Ressourcen über die Schnittstelle `StdControl` bzw. `SplitControl` schalten. Somit müssen die Applikationen weder über die genutzten Hardware-Komponenten noch deren Power-Zustände Bescheid wissen, um die Hardware in einen niedrigeren Power-Zustand zu versetzen. Dies übernehmen die Komponenten aus der HAL- und HPL-Schicht.

```
1  interface McuSleep {
2    async command void sleep();
3  }
```

Quelltext 3.22: Die Schnittstelle `McuSleep` für das Versetzen des Mikrocontrollers in den Schlaf-Modus.

Wie in dem Hardware-Kapitel 2.3.1 über Mikrocontroller dargelegt ist, besitzen die Mikrocontroller unterschiedliche Power-Zustände, in denen sie unterschiedliche Teile der Hardware ein- oder ausgeschaltet haben. Das Power-Management muss hierbei den gewünschten Power-Zustand aus den aktuell erbrachten Funktionen ermitteln und anschließend den niedrigsten Power-Zustand setzen. In TinyOS ruft der Scheduler den Befehl `sleep` aus der Schnittstelle `McuSleep` auf, sobald keine weiteren Tasks in der Task-Queue vorhanden sind. Die Schnittstelle `McuSleep` ist in Quelltext 3.22 dargestellt.

```
1  module McuSleepC
2  {
3    provides interface McuSleep;
4    provides interface McuPowerState;
5    uses interface McuPowerOverride;
6  }
7  implementation
8  {
9    //Plattform-spezifische Implementierung
10 }
```

Quelltext 3.23: Die Komponente `McuSleepC` aus der HIL-Schicht für die Steuerung des Schlaf-Modus.

Bei der Erstellung der TinyOS Applikation wird dabei automatisch der Scheduler mit der Plattform-spezifischen Komponente `McuSleepC` aus der HIL-Schicht verbunden, deren Definition laut TEP 112 in Quelltext 3.23 abgebildet ist [147]. Die Komponente `McuSleepC` ist nicht nur für das Versetzen des Mikrocontrollers in den Schlaf-Zustand zuständig, sondern sie berechnet auch den niedrigsten Power-Zustand, den der Mikrocontroller einnehmen kann. Die Berechnung des niedrigsten Power-Zustandes kostet in der Regel einige Rechenzyklen, die nicht vor jedem `sleep` Befehl aus der Schnittstelle `McuSleep` durchlaufen werden sollten. Zudem ändert sich der einzunehmende Power-Zustand nur in den seltensten Fällen zwischen zwei Schlaf-Perioden. Somit sollte die Berechnung des Power-Zustandes nur angestoßen werden, wenn eine andere Komponente im System Bedarf daran sieht. Dies kann über den Befehl `update` aus der Schnittstelle `McuPowerState`, die in Quelltext 3.24 dargestellt ist, angestoßen werden. Wird der Befehl `update` aufgerufen, so berechnet die Komponente `McuSleepC` erneut den niedrigsten einzunehmenden Power-Zustand für den nächsten `sleep` Befehl.

```
1  interface McuPowerState {
2    async command void update();
3  }
```

Quelltext 3.24: Die Schnittstelle `McuPowerState` für die erneute Ermittlung des niedrigsten Power-Zustandes.

Die Komponente `McuSleepC` muss während der Berechnung des niedrigsten Power-Zustands den Befehl `lowestState` aus der Schnittstelle `McuPowerOverride` aufrufen, die in Quelltext 3.25 abgebildet ist. Als Rückgabewert erhält sie von den verbundenen

Komponenten einen aus deren Sicht niedrigsten Power-Zustand, der in der Berechnungen berücksichtigt werden muss. Der Grund für das Abfragen der anderen Komponenten im System ist, dass die Applikation ein besseres Wissen über die nächsten Aktionen des Sensor-Knotens hat und somit Einfluss auf die Berechnung nehmen sollte. Auf diese Weise kann die Applikation z. B. verhindern, dass der Sensor-Knoten einen niedrigeren Power-Zustand einnimmt, wenn als nächstes das Messen eines Sensor-Werts oder das Versenden eines Nachrichten-Pakets ansteht.

```
1 interface McuPowerOverride {
2   async command mcu_power_t lowestState();
3 }
```

Quelltext 3.25: Die Schnittstelle `McuPowerOverride`.

Über die zwei vorgestellten Mechanismen wird in TinyOS das Power-Management gesteuert, um möglichst energieeffizient mit den Energie-Ressourcen des Sensor-Knotens umzugehen. Zum einen können die Applikationen über die Schnittstellen `StdControl` und `SplitControl` nicht mehr benötigte Dienste oder Hardware-Komponenten deaktivieren. Zum anderen sorgt die Komponente `McuSleepC` dafür, dass, sobald kein Task mehr ansteht, der Mikrocontroller und seine Peripherie in dem niedrigsten Power-Modus schlafen gelegt werden.

3.4 Zusammenfassung

Seit 1945 haben sich die Software-Systeme mit den ersten Rechen-Systemen entwickelt [148]. Die Grundlagen wurden in den Mainframe-Systemen gelegt, die ihre Dienste als Server-Systeme mehreren Nutzern zur Verfügung gestellt haben. Ab den 1980er Jahren haben sich immer mehr die Desktop-Systeme verbreitet, deren Software-Systeme von den Mainframe- und Server-Systemen abgeleitet und angepasst wurden. Die Embedded Systeme bis hin zu den Automotiven Systemen enthalten Software, die sich in ihrer Architektur nicht grundlegend von den Server- oder Desktop-Systemen unterscheidet. Jedoch sind die Software-Systeme immer an die Domänen-spezifischen Anforderungen angepasst, wodurch sich die verschiedenen Software-Systeme im Detail unterscheiden.

In diesem Kapitel wurde eine grundlegende Übersicht über die Software-Systeme mit dem Betriebssystem als Basis gegeben und die verschiedene Betriebssystem-Arten vorgestellt, die sich im Laufe der Zeit entwickelt haben. Die erste Art waren die Monolithischen Betriebssysteme, in denen alle Treiber und Dienste im Betriebssystem-Kern und nur die Applikationen in den Benutzer-Prozessen außerhalb des Betriebssystem-Kerns laufen. Ein Möglichkeit, die Fehleranfälligkeit der Monolithischen Betriebssysteme einzudämmen, ist die Treiber und Dienste aus dem Betriebssystem-Kern auszulagern und in eigenen Benutzer-Prozessen laufen zu lassen. Der Betriebssystem-Kern besteht dann nur noch aus den Komponenten, die nur im privilegierten Modus des Prozessors laufen können. Diese zweite Betriebssystem-Art wird als Microkernel-Betriebssystem bezeichnet. Viele der Konzepte, die heute noch in den Betriebssystemen zu finden sind, stammen aus den 1960er und 1970er Jahren und haben sich in den Software-Systemen

bis heute erhalten. Mit dem Ziel, die Software-Systeme nach dem heutigen Stand der Technik weiterzuentwickeln, startete 2003 das Singularity-Projekt bei Microsoft Research. Hierbei wurde der Frage nachgegangen, wie ein modernes Software-System mit aktuellen Software-Werkzeugen aussehen könnte. Daraus hat sich das Singularity-OS entwickelt, das aufbauend auf einer Microkernel-Architektur drei neuartige Konzepte enthält. Der Adressraum der SIPs, in den die Treiber, Dienste und Applikationen ausgeführt werden, wird nicht durch Hardware-Unterstützung sondern durch die Typ-Sicherheit der Programmiersprache Sing# geschützt. Die sogenannten Contract-Based Channels sichern zweitens die Kommunikation zwischen den SIPs ab, indem der Nachrichtenaustausch durch sogenannte Verträge festgelegt wird. Als drittes neues Konzept sorgen die MBPs für eine zusätzliche Robustheit des Systems, indem vor dem Start der Programme mit Hilfe von Manifesten, die die Programm beschreiben und deren Ressourcen-Anforderungen enthalten, Ressourcen-Konflikte aufgedeckt werden, bevor das System in einen instabilen Zustand gebracht wird.

Die Monolithischen Betriebssysteme und die Microkernel-Betriebssysteme werden als GPOS ausgeführt, bei denen das Betriebssystem nicht für eine bestimmte Anwendung konzipiert ist. In der Domäne der Embedded Systeme, die aufgrund der eingeschränkten Hardware-Ressourcen von kleinen Software-Systemen geprägt ist, wurde ein weiterer Weg im Bereich der Betriebssysteme eingeschlagen, woraus sich die sogenannten ASOS entwickelt haben. Zu dieser Gruppe gehören u. a. die Komponenten-basierten Betriebssysteme, die nur aus den für die Applikation benötigten Komponenten zur Entwicklungszeit zusammengesetzt und später während der Ausführung nicht mehr angepasst werden. Ein Vertreter der Komponenten-basierten Betriebssysteme ist TinyOS, das aufgrund der Verwendung in der PLASA-Plattform in diesem Kapitel im Detail vorgestellt wurde. TinyOS bietet mit seinen Kommunikationsfähigkeiten und dem Energie-optimierten Power-Management eine solide Basis für die Software in der PLASA-Plattform und für die Untersuchung eines automotiven Power-Managements. Die in TinyOS verwendeten Schnittstellen für die verschiedenen Funktionen, worunter u. a. die Timer-, ADC- und Kommunikationsfunktionen fallen, werden in sogenannten TEPs definiert. Auf einige dieser Schnittstellen setzen ebenfalls die Applikationen der PLASA-Plattform auf, weshalb diese neben dem Task-Modell von TinyOS ausführlich in diesem Kapitel beschrieben wurden.

Kapitel 4

Power-Management in Automotiven Systemen

Um die Automotiven Systeme energieeffizienter zu machen, gibt es verschiedene Möglichkeiten. Eine davon ist, die Hardware-Komponenten selbst energieeffizienter zu realisieren, indem sie mechanisch optimiert werden, indem z. B. die Aktoren, die Einfluss auf den physikalischen Prozess nehmen, mit weniger Reibungsverluste ihre Funktion erfüllen. Eine weitere Alternative ist die Verlust-Leistung der elektronischen Bauelementen wie Mikrochips durch elektrotechnischen Methoden zu verringern.

Eine andere Möglichkeit ist, Sensoren, Aktoren oder sogar ganze ECUs abzuschalten, falls diese nicht bei der Funktionserfüllung des Automotiven Systems benötigt werden. Darunter fällt auch die Steuerung der von der Hardware angebotenen Power-Zustände. Zusammengefasst bedeutet dies, dass die gesamte Hardware des Automotiven Systems den niedrigsten Power-Zustand einnehmen sollte, der ausreicht, um die benötigten Funktionen in dem jeweiligen Fahrzeug-Zustand zu erfüllen.

Im Umfeld der Automotiven Systeme und des AUTOSAR-Standards haben sich folgende drei Konzepte für die Energieeffizienz der Automotiven Systeme etabliert: *ECU Degradation*, *Pretended Networking* und *Partial Networking*.

Bei dem Konzept *ECU Degradation* wird ein Herunterfahren des Mikrocontrollers veranlasst, so dass der Energieverbrauch des Mikrocontrollers vermindert wird. Zusätzlich stellt das Konzept das Abschalten der Peripherie bereit, so dass der Energieverbrauch einer ECU abgesenkt werden kann. Bei diesem Konzept wird der Mikrocontroller niemals ganz ausgeschaltet, so dass dieser immer noch die Nachrichten-Pakete auf dem Bus empfangen und beantworten kann. Somit ist auch der Kommunikationscontroller immer aktiv und kann mit Hilfe dieses Konzepts nicht heruntergefahren werden. Allerdings ermöglicht das Konzept einzelne Kerne des Mikrocontrollers auszuschalten. Da sich das Steuergerät nach außen hin wie im voll betriebenen Modus verhält, muss kein weiteres Steuergerät über den Wechsel des Power-Zustands informiert werden. Die Energieeinsparung ergibt sich aus dem Heruntertakten des Mikrocontrollers und dem Ausschalten der Peripherie. Der Implementierungsaufwand beschränkt sich hierbei auf das Bereitstellen der Schnittstellen für das Steuern der Power-Zustände, die die Hardware bereitstellt.

Für dieses Konzept der Energieeinsparung spielt das automotive Power-Management eine maßgebliche Rolle, da dieses die Power-Zustände der ECU bestimmt und entsprechend die Power-Zustände der Hardware-Komponenten schaltet.

Beim Konzept *Pretended Networking* wird der Mikrocontroller komplett ausgeschaltet und die Kommunikation der ECU übernimmt der sogenannte ICOM (engl. *intelligent communication controller*). Dieser empfängt die Nachrichten und verschickt, je nach Konfiguration, selbstständig die Antwort-Pakete, so dass es für die anderen ECUs im System wirkt, als ob die ECU voll betrieben wird. Somit arbeitet das Konzept *Pretended Networking* ebenfalls nur lokal, da andere ECUs nicht an das Konzept angepasst werden müssen. Jedoch lässt sich hier mehr Energie einsparen, da der Mikrocontroller komplett heruntergefahren wird und die Kommunikation das ICOM übernimmt, das weniger Energie als der Mikrocontroller für die Kommunikation benötigt. C. Schmutzler *et al.* schlagen das Konzept ebenso für weitere primitive Aufgaben wie das Erzeugen eines PWM-Signals vor [132].

Im Gegensatz zu den Konzepten *ECU Degradation* und *Pretended Networking*, die nur lokal auf einer ECU arbeiten und den übrigen ECUs nicht die Maßnahmen für die Energieeinsparungen mitteilen, ist das Konzept *Partial Networking* ein verteilter Ansatz. Hier wird die gesamte ECU ausgeschaltet und die übrigen ECUs darüber informiert. Jedoch sind für den FlexRay- und CAN-Bus aufgrund den Bus-Eigenschaften ein spezielles ICOM notwendig. Für den FlexRay-Bus wurde dies von C. Schmutzler *et al.* untersucht und in einem speziellen FlexRay-Controller implementiert [133]. Somit ist der Implementierungsaufwand für das Konzept *Partial Networking* höher, der sich aber durch eine höhere Energieeinsparung wieder rechtfertigt.

Um diese Konzepte anwenden zu können, ist ein DPM für Automotive Systeme notwendig, das im weiteren Verlauf dieses Kapitels im Fokus steht. Ein DPM für Systeme wurde wissenschaftlich bereits ausgiebig untersucht. L. Benini *et al.* beschreiben das DPM als Methodik für dynamisch konfigurierbare Systeme, um die benötigten Dienste, Funktionen und Leistungsstufen mit einer minimalen Anzahl an aktiven Komponenten oder einer minimalen Last auf den einzelnen Komponenten bereitzustellen [30]. Das Ziel des DPMs ist es, die benötigten Funktionen des Systems möglichst energieeffizient bereitzustellen, indem der Energieverbrauch der benötigten Hardware-Komponenten auf ein Minimum reduziert wird.

Die bereits gewonnenen Erkenntnisse und Methoden des DPMs gilt es auch auf die Automotiven Systeme anzuwenden, was in Abschnitt 4.2 erfolgt. Zuvor jedoch werden die Anforderungen an ein Power-Management für Automotive Systeme dargelegt, die sich aus den besonderen Eigenschaften der Domäne ableiten lassen. Da die Automotiven Systeme durch die E/E-Architektur in verschiedene Fahrzeug-Domänen und Teilnetze gegliedert sind, wird ein Power-Management für Automotive Systeme ebenfalls hierarchisch gegliedert sein, wie es A. Barthels *et al.* in ihrer Veröffentlichung beschreiben [28].

In Abschnitt 4.3.2 wird der Fokus auf den PM gelegt, der in jeder einzelnen ECU vorhanden ist und die Power-Zustände der jeweiligen ECU-Hardware steuert. Dieser ist als OSPM (engl. *OS-based power management*) im Betriebssystem in einer jeden ECU verankert. Jedoch existieren ebenfalls PMs, die in Hardware realisiert sind. Das Zusammenspiel und die Möglichkeit eines in Hardware realisierten PMs in Automotiven Systemen wird nachfolgend kurz beleuchtet.

4.1 Anforderungen an ein Power-Management in Automotiven Systemen

Bevor auf die Architektur eines DPMs für Automotive Systeme eingegangen wird, werden dessen Anforderungen beschrieben, die es zu erfüllen gilt. Diese leiten sich aus den Herausforderungen und Anforderungen der Automotiven Systeme ab, die in Abschnitt 2.1 vorgestellt wurden.

Niedrigster Energieverbrauch. Das DPM für Automotive Systeme hat mit seiner Strategie das Ziel, für den niedrigsten Energieverbrauch im gesamten Automotiven Systems zu sorgen. Dabei hat sich aus den Erfahrungen anderer Domänen gezeigt, dass die Wechsel des Power-Zustands möglicherweise Zeit und auch Energie kosten, so dass bei hochfrequenten Wechseln keine Energie gespart wird oder dass eventuell das Verhalten energetisch schlechter ist [30].

Einhalten der funktionalen Sicherheit. Durch das Ein- und Ausschalten der Hardware-Komponenten und die Wechsel der Power-Zustände vergrößert sich die Programm-Logik einer jeden ECU. Dadurch erhöhen sich u. U. ebenfalls die funktionalen Zustände der ECU, deren funktionalen Sicherheit garantiert werden muss.

Einhalten der Echtzeit-Fähigkeit. Neben der funktionalen Sicherheit müssen auch die Echtzeit-Anforderungen des Automotiven Systems eingehalten werden. Die von der Hardware benötigten Zeit bei einem Wechsel des Power-Zustands kann dazu führen, dass die Echtzeit-Bedingungen des Systems verletzt werden. Deshalb müssen diese Zeit-Verzögerungen in die Echtzeit-Betrachtungen miteinbezogen werden, was u. U. die Echtzeit-Analyse erschwert.

Keine wesentliche Steigerung des Ressourcen-Verbrauchs. Die Komponenten für das DPM benötigen neben der Programm-Logik zusätzliche Hardware-Ressourcen, um ihre Aufgaben zu erfüllen. Sofern die Komponente für das Power-Management einer ECU in Software realisiert ist, braucht diese für ihre Ausführung Arbeitsspeicher, um ihre Daten darin abzulegen, und zusätzlichen Programm-Speicher für deren Code. Da die Automotiven Systeme möglichst auf geringen Ressourcen-Verbrauch ausgelegt sind, um Hardware-Kosten einzusparen, gilt dies auch für das DPM Automotiver Systeme, damit die benötigten Speicher-Ressourcen nicht erheblich ansteigen und somit zusätzliche Kosten verursacht werden. Ebenso sollte das Power-Management nicht zuviel Rechenlast für die Bestimmung und Wechsel der Power-Zustände in der Hardware erzeugen. Durch die Rechenlast wird zusätzliche Energie benötigt, die in den Energieverbrauch der ECU miteinberechnet werden muss. Genauso sollte durch das Power-Management keine oder nur wenig zusätzliche Buslast erzeugt werden, um einerseits keine zusätzliche Energie zu verbrauchen und andererseits den Bus nicht unnötig zu beanspruchen, was zu Verzögerungen anderer Bus-Nachrichten führen kann. Dadurch wäre wiederum das Einhalten der funktionalen Sicherheit und der Echtzeit-Fähigkeit gefährdet.

Leichter Austausch der Power-Management Konfiguration. Die Automotiven Systeme sind ein sehr stark verteiltes System, das sich mit jedem Modell oder mit

der Ausstattungsvariante ändert. Somit kommen je nach Modell und Ausstattung des Fahrzeugs unterschiedliche Kombinationen von ECUs zum Einsatz. Da die Funktionen verteilt auf den ECUs erfüllt werden, muss auch die Konfiguration des Power-Managements leicht an die Fahrzeug-Konfiguration und dessen Architektur zur Entwicklungszeit der ECU-Software angepasst und ausgetauscht werden können.

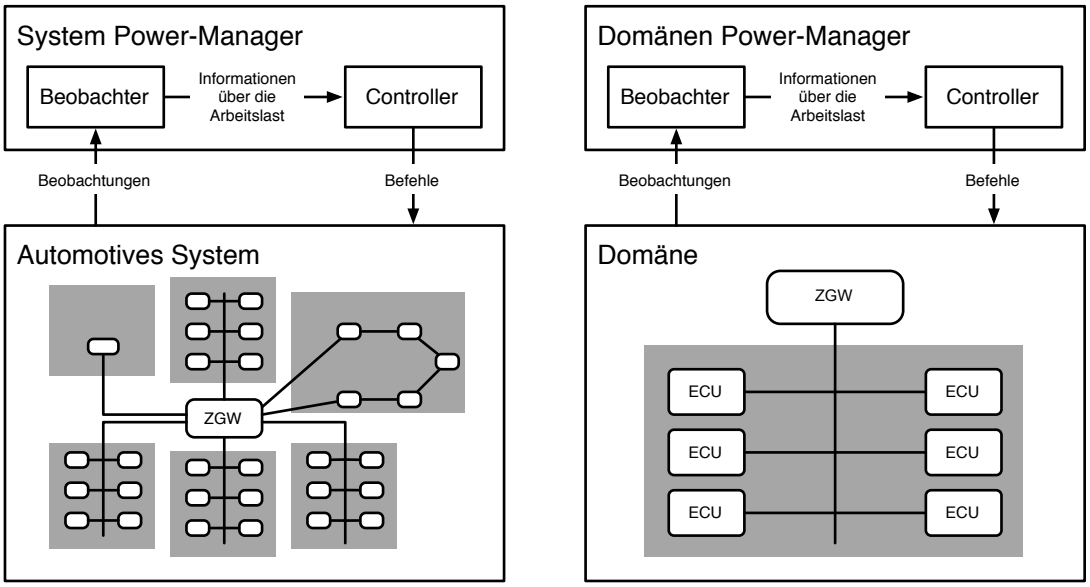
Leichter Umgang des Power-Managements mit heterogener Hardware. In Automotiven Systemen kommt eine heterogene Hardware zum Einsatz, für das ein DPM ausgelegt werden muss. Damit entsteht die Anforderung, dass ein DPM für Automotive Systeme leicht mit der heterogenen Hardware umgehen können muss.

Anbieten von Power-Zuständen durch die Hardware. Durch den Kosten-Druck, der in der Automobil-Branche herrscht, werden die ECUs möglichst kostengünstig entwickelt und produziert, wodurch jede nicht benötigte Funktionalität wegfällt. Damit allerdings ein DPM möglich ist, müssen die Hardware-Komponenten verschiedene Power-Zustände anbieten. Dafür müssen auch Sensoren und Aktoren, die extern an die ECU angeschlossen sind, verschiedene Power-Zustände anbieten, die ein DPM des Automotiven Systems steuern kann. Um das DPM effizienter zu gestalten, müssen die Hardware-Komponenten fein-granulare Power-Zustände anbieten. Jedoch widerspricht dies der Anforderung niedriger Kosten für Automotive Systeme.

4.2 Hierarchisches Power-Management in Automotiven Systemen

Bevor auf das Power-Management in Automotiven Systemen eingegangen wird, werden die Grundlagen des DPMs, die L. Benini *et al.* in ihrer Übersicht über das DPM gelegt haben [30], kurz vorgestellt. Im Anschluss werden diese auf die Automotiven Systeme angewendet. Das DPM ist die Methodik, um sogenannte PMCs (engl. *power manageable components*) zu steuern. Dabei sind die PMCs für das DPM abstrakt, d. h. die genaue Funktionsweise der PMCs ist für das DPM nicht von Bedeutung und werden bis auf die unterschiedlichen Power-Zuständen als Black-Box betrachtet. Die verschiedenen Power-Zustände der PMCs werden in einer PSM (engl. *power state machine*) abstrahiert, die der PM steuert, indem er das System beobachtet und aus den daraus gewonnenen Informationen Befehle zur Steuerung der Power-Zustände an das System schickt. Es wurde bereits das DPM für Systeme mit vielen Geräten, die ihr Power-Management selbstständig vornehmen, untersucht. Hierfür wurde ein hierarchisches Power-Management mit bestärkendem Lernen (engl. *reinforcement learning*) vorgeschlagen [158]. In der Domäne der WSNs werden für das DPM in den Sensor-Knoten meist stochastische Ansätze gewählt. Eine aktuelle Übersicht über die stochastische Ansätze für ein DPM in WSNs gibt die Veröffentlichung von A. Pughat und V. Sharma [125].

In Automotiven Systemen, in denen die bereitgestellten Funktionen je nach Ausstattung des Systems von bis zu 80 ECUs verteilt erbracht werden, gelten andere Randbedingungen für das Power-Management als in gängigen Desktop- oder in den meisten



(a) PM auf System-Level.

(b) PM auf Domänen-Level.

Abbildung 4.1: PMs auf unterschiedlichen System-Ebenen (angelehnt an eine Abbildung aus der Veröffentlichung von L. Benini *et al.* [30]).

Embedded Systemen. Typischerweise sind die ECUs über mehrere Bus-Systeme mit einem Gateway als Bindeglied untereinander verbunden, wie es in Abschnitt 2.2.1 über die E/E-Architektur beschrieben ist. Dies führt im Unterschied zu Desktops oder Servern dazu, dass die Funktionalität des gesamten Systems von mehreren ECUs erbracht wird. Somit muss ein automotives Power-Management verteilt über alle ECUs erfolgen. Das Ziel dabei ist, dass jede einzelne ECU den niedrigsten Power-Zustand einnimmt, der benötigt wird, um die momentan benötigten Funktionen des gesamten Automotiven Systems ausführen zu können.

A. Barthels *et al.* schlagen deshalb vor, das Power-Management in dem Automotiven System hierarchisch vorzunehmen [28]. Neben dem PM auf Gesamtsystem-Ebene existiert auf Domänen-Ebene ebenso ein PM, der das DPM für eine Domäne übernimmt. Dies ist in Abbildung 4.1a und in Abbildung 4.1b exemplarisch dargestellt. Der PM ist nach L. Benini *et al.* in zwei Teilkomponenten unterteilt, die unterschiedliche Aufgaben haben [30]. Der Beobachter überwacht das System und analysiert dessen momentanen Zustand. Daraus generiert er Informationen über die Arbeitslast des Systems und übergibt diese dem Controller, der daraus Befehle für das System ableitet und diese im Anschluss versendet. Die grundlegenden Arbeitsweisen der PMs auf System- und Domänen-Ebene sind identisch. Nur der Bereich der Beobachtung und die daraus generierten Befehle sind für die unterschiedlichen Ebenen ausgelegt.

Auf System-Ebene existiert ein DPM, das die Power-Zustände des gesamten Automotiven Systems steuert. Dieser Zusammenhang ist in Abbildung 4.1a dargestellt. Auf dieser Ebene wird die übergeordnete Strategie des Automotiven Systems festgelegt, die sehr von dem Fahrzeug-Fahrer als Nutzer beeinflusst wird. Darunter fällt u. a. ob das

Fahrzeug gerade geparkt und abgeschlossen ist, ob es sich in einem fahrbereiten Zustand befindet oder ob der Motor läuft und das Fahrzeug fährt. Der PM setzt sich aus einem Beobachter (engl. *observer*) und Controller zusammen. Der Beobachter zieht seine Beobachtungen aus dem System und informiert den Controller über den Zustand des Systems. Dieser entscheidet über den einzunehmenden Power-Zustand und schickt seine Befehle zur Einnahme der Power-Zustände an das gesamte System.

Auf Domänen-Ebenen werden Befehle des DPMs auf System-Ebene weiter heruntergebrochen und durch das DPM auf Domänen-Ebene umgesetzt, was in Abbildung 4.1b veranschaulicht ist. Allerdings werden nicht nur die Befehle des DPMs auf System-Ebene berücksichtigt, sondern das System auf Domänen-Ebene wird ebenfalls beobachtet und die Befehle für die Domäne daraus abgeleitet.

Dies lässt sich auf ECU-Ebene entsprechend weiter anwenden. Jede ECU enthält einen PM, der die Befehle des DPM auf Domänen-Ebene entgegen nimmt und die Arbeitslast der PMCs auf der ECU-Ebene beobachtet und diese steuert.

Auf diesen Zusammenhang des DPMs auf Domänen-Ebene wird im Folgenden eingegangen, wobei auch der PM auf ECU-Ebene einbezogen wird. Dazu wird sowohl die logische als auch die technische Architektur des Automotiven Systems herangezogen, was in Abbildung 4.2 graphisch an einem einfachen Beispiel dargestellt ist. Als Grundlage wird die Abbildung 2.2 aus Abschnitt 2.2.2 verwendet, in der die Zuordnung der logischen Architektur auf die technische Architektur der Automotiven Systeme erläutert wurde. Diese Abbildung wurde um die Aspekte, die für das DPM für Automotive Systeme notwendig sind, erweitert.

In einem bestimmten Fahrzeug-Zustand wird Senke_1 benötigt und Senke_2 nicht. Daraus ergibt sich aus der Rückwärts-Traversierung der benötigten Funktionsblöcke und Sensoren in der logischen Architektur, dass die logischen Quellen Quelle_1, Quelle_2 und Quelle_3 sowie die Funktionen Funktion_1, Funktion_2 und Funktion_4 aktiv geschaltet werden müssen und die logischen Komponenten Quelle_4, Funktion_3 und Funktion_5 in diesem Fahrzeug-Zustand deaktiviert werden können. Durch die Zuordnung der logischen Komponenten auf die technische Architektur des Automotiven Systems ergibt sich, dass der Sensor_1 und der Sensor_2 sowie der Software-Task T_1 auf ECU_1 aktiv sind. Auf der ECU_2 ist der Sensor_3 und die Software-Tasks T_2 und T_4 und der Aktor_1 aktiv, während der Sensor_4 und der Software-Task T_3 deaktiviert werden können. Der Software-Task T_5 und der Aktor_2 auf der ECU_3 sind deaktiviert. Somit ist es möglich, die komplette ECU_3 abzuschalten, da ihre Funktionalität in dem momentan aktiven Fahrzeug-Zustand nicht benötigt wird.

In der technischen Architektur im unteren Teil der Abbildung ist ersichtlich, dass jede ECU einen PM enthält, der aufgrund des aktuellen Fahrzeug-Zustands die PMCs der ECU steuert. Dabei setzt sich der Power-Zustand der ECU aus den Power-Zuständen aller PMCs einer ECU zusammen. Dazu zählen sowohl die angeschlossenen Sensoren als auch die Aktoren sowie der Mikrocontroller inklusive Kommunikationscontroller, die in der ECU verbaut sind.

Damit ein DPM in Automotiven Systemen möglichst energieeffizient arbeiten kann, ist die Zuordnung der logischen Architektur auf die technische Architektur sehr entscheidend. Denn der Energieverbrauch einer ECU ist am geringsten, wenn diese komplett ausgeschaltet werden kann, da sie für einen bestimmten Fahrzeug-Zustand keine Funk-

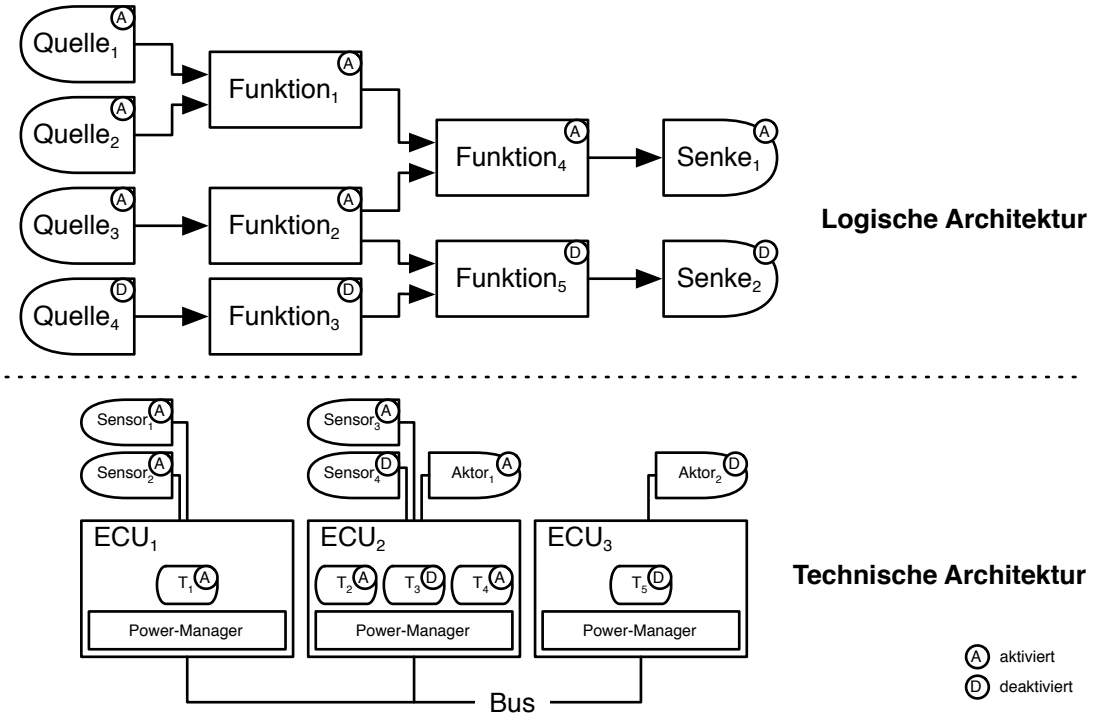


Abbildung 4.2: DPM auf Domänen-Ebene unter Betrachtung der logischen und technischen Architektur eines Automotiven Systems.

tionalität bereitstellen muss. In dem Beispiel aus Abbildung 4.2 ist es möglich die ECU_3 komplett auszuschalten, da T_5 und $Aktor_2$ für den aktuellen Fahrzeug-Zustand nicht benötigt werden. Falls die Zuordnung der $Funktion_2$ auf ECU_3 erfolgt wäre, müsste die ECU_3 den Software-Task T_2 ausführen und könnte so nicht komplett ausgeschaltet werden. Bis jetzt erfolgt die Zuordnung der logischen Architektur auf die technische Architektur in Automotiven Systemen nicht unter dem Aspekt der Energieeffizienz, sondern meist nur aus funktionalen Aspekten. Um die Zuordnung der logischen Architektur auf die technischen Architektur unter energetischen Aspekten zu untersuchen, hat G. Walla *et al.* ein Framework entwickelt [159, 160, 163]. Dabei ermöglicht der Simulator ITE-Sim den Energieverbrauch von verschiedenen Alternativen der Zuordnung bereits in frühen Entwicklungsphasen zu ermitteln und zu evaluieren [162]. Um das Problem der Zuordnung der Funktionen auf ECUs unter energieeffizienten Aspekten für die automotive Domäne zu lösen, verwenden G. Walla *et al.* ein sogenanntes MILP-Modell (engl. *mixed-integer linear programming*) mit einer Kostenfunktion, die den Leistungsverbrauch des Prozessors und der Kommunikationscontroller einschließt [161]. Ein DPM für Automotive Systeme kann somit erst durch eine energieeffiziente Zuordnung seiner Funktionen auf ECUs seine vollen Möglichkeiten ausschöpfen.

Die hierarchische Struktur des DPMs in Automotiven Systemen lässt sich in einer ECU weiter auf die Hardware-Ebene fortsetzen. Die PMCs auf ECU-Ebene können wiederum einen PM in Hardware besitzen, der für die Steuerung der Power-Zustände

seiner PMCs auf Hardware-Ebene verantwortlich ist. Ein PM auf Hardware-Ebene bietet den Vorteil schnellerer Reaktionszeiten gegenüber einem PM auf ECU-Ebene, der als OSPM in Software realisiert ist. Jedoch kann der Beobachter des PMs auf Hardware-Ebene nur weiter eingeschränkt die Informationen des Systems erfassen.

4.3 Power-Management auf ECU-Ebene

Wie im vorherigen Abschnitt beschrieben, ist das DPM in Automotiven Systemen hierarchisch aufgebaut. Im weiteren Verlauf dieses Abschnitts wird detailliert auf das DPM auf ECU-Ebene eingegangen. Als erstes werden die PMCs charakterisiert, die der PM in der ECU steuert. Der PM auf ECU-Ebene wird als OSPM realisiert, der als Software im Betriebssystem läuft. An dieser Stelle können die bisherigen Methoden des DPMs aus den Embedded Systemen wie z. B. der Sensor-Knoten herangezogen werden. Allerdings muss eine ECU die Befehle des PMs auf Domänen-Ebene befolgen und mit in seine Entscheidungen für die Steuerung der PMCs auf ECU-Ebene einbeziehen. Bevor der PM auf ECU-Ebene beschrieben wird, wird auf die PMCs auf ECU-Ebene eingegangen.

4.3.1 PMCs auf ECU-Ebene

Der PM auf ECU-Ebene steuert seine PMCs. In Automotiven Systemen sind dies nicht nur Hardware-Komponenten, sondern auch die Software-Tasks, die je nach Fahrzeug-Zustand den entsprechenden Zustand einnehmen müssen. Wie bereits im vorherigen Abschnitt beschrieben, wird anhand der logischen Architektur des Automotiven Systems festgelegt, welchen Power-Zustand ein PMC in einem bestimmten Fahrzeug-Zustand einnehmen muss. Im Folgenden werden zwischen Hardware- und Software-PMCs unterschieden, auf die in den nächsten Abschnitten detaillierter eingegangen wird.

Hardware-PMCs

Hardware-Komponenten in Desktop-Systemen sind bereits als PMCs untersucht worden. So beschreiben L. Benini *et al.* wie Hardware-Komponenten als PMCs modelliert werden können, so dass eine einheitliche Sicht für den PM auf die verschiedenen Ausprägungen der Hardware gewährleistet ist [30]. Dafür wird ein endlicher Automat vorgeschlagen, dessen Zustände die verschiedenen Power-Zustände der Hardware abbilden, wobei jeder Zustand mit einem Leistungsverbrauch charakterisiert ist. Die Transitionen zwischen den Zuständen im Automaten stellen die Übergänge zwischen Power-Zuständen der Hardware dar. Dabei enthalten die Transitionen die Zeit für den Übergang und der dafür benötigte Energieaufwand. Diese Abstraktion der Hardware wird PSM genannt. Somit ist für den PM nicht die genau Funktionsweise der Hardware in einem Power-Zustand wichtig, sondern wird als Black-Box betrachtet. Der PM entscheidet aufgrund der Informationen, die in der PSM hinterlegt sind, wie er die PMCs steuert.

In Desktop- und Server-Systemen sind Hardware/Software-Schnittstellen dafür spezifiziert worden, wie Software-Komponenten die Power-Zustände der Hardware ansteuern können. Das APM (engl. *advanced power management*) [83] ist eine der Spezifikationen, die später von dem ACPI [74] abgelöst wurde. Die Weiterentwicklung beim ACPI-

Standard liegt darin, dass die Wechsel der Power-Zustände vom Betriebssystem aus gesteuert werden können. Aus diesem Grund stellt die Hardware eine detailliertere Beschreibung der verschiedenen Power-Zustände bereit, die der PM im Betriebssystem steuert. Im ACPI-Standard wird das gesamte Computer-System betrachtet, das die definierten, globale Zustände G_0 bis G_3 einnehmen kann. Zusätzlich werden Geräte, wie Festplatten, Laufwerke oder Displays betrachtet, die sich jeweils in den eigenen Power-Zustände D_0 bis D_3 befinden können. Eine gesonderte Rolle spielt dabei der Prozessor, der explizit als Komponente die im ACPI-Standard ausgewiesenen Power-Zustände C_0 bis C_n erhält. Der globale Schlaf-Zustand G_1 ist im ACPI-Standard wiederum in die verschiedenen Schlaf-Zustände S_0 bis S_5 unterteilt.

Das im ACPI-Standard spezifizierte Vorgehen kann auch auf das DPM in Automotiven Systemen auf ECU-Ebene angewendet werden. Denn die in Automotiven Systemen verbaute Hardware besitzt, wie in Abschnitt 2.3.1 über die Hardware in Automotive Systeme dargestellt, prinzipiell den gleichen Aufbau wie die Hardware in Desktop-Systemen.

Jedoch spielen im Gegensatz zu Desktop-Systeme die Sensoren und Aktoren eine wichtige Rolle für die Funktionserfüllung in Automotiven Systemen. Diese lassen sich wie Geräte im ACPI-Standard betrachten und somit verschiedene Power-Zustände zuweisen, die über den PM auf ECU-Ebene gesteuert werden. Darüber können die an eine ECU angeschlossenen Sensoren und Aktoren ein- bzw. ausgeschaltet werden. Als Voraussetzung dafür müssen die angeschlossenen Sensoren und Aktoren verschiedene Power-Zustände inklusive eines kompletten Ausschaltens anbieten. Bei Aktoren kommt erschwerend hinzu, dass der Leistungsverbrauch im aktiven Power-Zustand sehr stark von der physikalischen Einflussnahme abhängt und somit starken Schwankungen in der Leistungsaufnahme unterliegt. Dies erschwert die Modellierung des Aktors mit Hilfe einer PSM, bei der die Leistungsaufnahme in jedem Power-Zustand festgelegt wird.

Der Hardware für die Bus-Kommunikation kommt eine besondere Rolle zu. Denn diese kann nicht ohne Weiteres in einem Automotiven System deaktiviert werden. Der Grund dafür ist, dass die ECUs zyklisch Nachrichten senden, die von anderen ECUs benötigt werden. Falls die zyklischen Nachrichten durch einen Schlaf-Zustand der ECU ausbleiben, werden Fehler-Einträge im System produziert. Aus diesem Grund untersucht C. Schmutzler *et al.* ob es möglich ist, dass ein Großteil der ECU sich im Schlaf-Zustand befindet, während ein intelligenter Kommunikationscontroller die Kommunikation nach außen aufrecht erhält [132, 133].

Die Mikrocontroller einer ECU besitzen wie die Prozessoren in Desktop-Systeme folgende Möglichkeiten, um Energie einzusparen [30, 131]:

Clock Gating. Diese Möglichkeit schaltet für bestimmte Bereiche des Mikrocontrollers die Clock ab. Damit wird die Frequenz-abhängige Verlust-Leistung für diesen Bereich reduziert. Allerdings ist der Bereich nicht mehr aktiv und kann keine Berechnungen mehr durchführen. Da der Bereich weiter mit Strom versorgt wird, gehen keine Kontext-Informationen verloren. Somit kann beim Einschalten der Clock ohne Nachladen der Kontext-Informationen die Ausführung fortgesetzt werden.

Power Gating. Über das *Power Gating* werden bestimmte Bereiche des Mikrocontrollers nicht mehr mit Strom versorgt. Damit wird die gesamte Verlust-Leistung für

diesen Bereich des Mikrocontrollers reduziert. Allerdings gehen auch die Kontext-Informationen verloren, was beim erneuten Aktivieren des Bereichs zur Folge hat, dass alle Kontext-Informationen nachgeladen werden müssen. Dies verzögert somit den Einschaltvorgang. Jedoch sind die Energieeinsparungen größer als beim *Clock Gating*.

Dynamic Voltage and Frequency Scaling. Diese Möglichkeit des *Dynamic Voltage and Frequency Scalings* erlaubt es, dynamisch an die benötigte Rechenleistung des Mikrocontrollers die Frequenz und die Versorgungsspannung anzupassen. Damit wird die Frequenz und die Versorgungsspannung abgesenkt, wenn nicht die komplette Rechenleistung des Mikrocontrollers benötigt wird.

Die Schwierigkeit ist, aus den von der Hardware angebotenen Möglichkeiten diskrete Power-Zustände für die PSM zu abstrahieren, mit der der Mikrocontroller durch den PM gesteuert werden kann. Vor allem durch das *Dynamic Voltage and Frequency Scaling*, das nicht zwangsläufig immer diskrete Zustände hat, sondern kontinuierliche Power-Zustände bereitstellt, müssen für die PSM diskrete Power-Zustände festgelegt werden. Die Wahl dieser diskreten Power-Zustände für die PSM hat ebenfalls Einfluss auf die Energieeffizienz des DPMs und muss somit für den konkreten Anwendungsfall optimiert werden, um die größtmögliche Energieeinsparung durch das DPM zu erhalten.

Bei den Mikrocontrollern handelt es sich um SoCs, bei denen viele Controller in einem Chip enthalten sind. Jeder der Controller ist dabei als ein Gerät nach dem ACPI-Standard zu verstehen, der unterschiedliche Power-Zustände einnehmen kann. Somit besteht der Mikrocontroller neben dem Mikroprozessor aus weiteren Geräten, die als eigene PMCs modelliert werden können. Somit ist es für den PM möglich, nicht benötigte Controller (z. B. Bus-Controller) auf dem Mikrocontroller auszuschalten und sobald der Controller wieder benötigt wird, diesen wieder zu aktivieren.

Dies wird anhand eines konkreten Beispiels erläutert. Der Atmel ATmega1284P Mikrocontroller, der bereits im Abschnitt 2.3.1 vorgestellt wurde, verfügt über sechs unterschiedliche Schlaf-Zustände, bei denen verschiedene Clock-Domänen und Oszillatoren des Mikrocontrollers abgeschaltet werden [7]. Die Tabelle 4.1 fasst die verschiedenen Schlaf-Zustände des Atmel ATmega1284P zusammen und gibt dabei an, für welche Domänen die Clock und welche Oszillatoren aktiv sind sowie welche Interrupts den Mikrocontroller aus dem Schlaf-Zustand wieder aufwecken.

Damit nutzt der Atmel ATmega1284P hauptsächlich das *Clock Gating* um Energie einzusparen. Wie viel Leistung dieser in den jeweiligen Schlaf-Zuständen benötigt, ist von der anliegenden Versorgungsspannung und der Takt-Frequenz des Mikrocontrollers abhängig. Für eine genaue Darstellung des Leistungsverbrauchs wird auf Abschnitt 6.2 verwiesen, in dem der Verbrauch des AtmelATmega1284P in den verschiedenen Schlaf-Zuständen im PLASA-Projekt vermessen wird.

Der leistungsfähigere Applikationsprozessor OMAP3530 von Texas Instruments bietet ebenfalls verschiedene Power-Zustände an. Da die Strom-Aufnahme für die höhere Rechenleistung um einiges größer ist, können die Power-Zustände des OMAP3530 feingranularer gesteuert werden, um einen sparsamen Betrieb je nach Benutzung des Applikationsprozessors zu ermöglichen. Neben dem *Clock Gating* wird ebenfalls das *Power Gating*

Schlaf-Zustand	Aktive Clock Domänen					Oszillatoren		Weck-Quellen						
	$\mathbf{clk}_{CPU}$	$\mathbf{clk}_{FLASH}$	$\mathbf{clk}_{IO}$	$\mathbf{clk}_{ADC}$	$\mathbf{clk}_{ASY}$	**Aktive Main Clock**	**Aktiver Timer Oszillator**	**INT2:0 und Pin-Veränderung**	**Anliegen der TWI Adresse**	**Timer2**	**SPM/EEPROM**	**ADC**	**WDT Interrupt**	**Andere E/A**
Idle			•	•	•	•	•	•	•	•	•	•	•	•
ADCNRM				•	•	•	•	•	•	•	•	•	•	
Extended Standby					•	•	•	•	•	•			•	
Standby						•		•	•				•	
Power-Save					•		•	•	•	•			•	
Power-Down								•	•				•	

Tabelle 4.1: Schlaf-Zustände des Atmel ATmega1284Ps (entnommen aus dessen Spezifikation [7]).

und das *Dynamic Voltage and Frequency Scaling* unterstützt. Dafür werden 18 sogenannte *Power Domains* definiert, die einzeln ab- bzw. zugeschaltet werden können [153].

Aus den beiden Beispielen für Mikrocontroller bzw. Applikationsprozessoren wird ersichtlich, dass die Hardware eine Vielzahl an unterschiedlichen Power-Zuständen anbietet. Diese in eine PSM zu transferieren, ist ebenfalls Teil der Aufgabe, ein DPM für ein konkretes System zu definieren.

Software-PMCs

Die Besonderheit des DPM in Automotiven Systemen ist, dass Software-Tasks ebenfalls PMCs sind. Dies bedeutet, dass die Software-Tasks vom PM der jeweiligen ECU gesteuert werden. Dies ist in typischen Desktop- und Embedded Systemen normalerweise nicht vorgesehen. In Desktop- oder Server-Systemen werden die Applikationen vom Benutzer gestartet und beendet. Der PM steuert aufgrund seiner Beobachtungen des Systems die Power-Zustände der Hardware. In Bezug auf die Software-Tasks, die in den Applikationen ausgeführt werden, bedeutet dies, dass der PM die Prozessor-Auslastung kontrolliert und demnach die Hardware-Zustände schaltet. Somit laufen die Applikationen meist ohne genaue Kenntnis, welchen Power-Zustand die Hardware eingenommen hat und damit ist für die Applikationen das DPM transparent.

In Embedded Systemen wie den WSNs beobachtet das DPM nur die Ausführung der Software-Tasks und kontrolliert je nach System-Auslastung die Power-Zustände der Hardware. Jedoch kontrolliert der PM nicht direkt die Zustände der Software-Tasks oder verändert deren Operationsmodus. In WSNs bestimmen die Software-Tasks selbst ihre Aufruf-Zyklen und ihren Operationsmodus.

In Automotiven Systemen verhält sich dies anders, da die gesamte Funktionalität typischerweise verteilt über eine Kette von Software-Tasks, die auf mehrere ECUs verteilt laufen können, erbracht wird. Somit können die Software-Tasks nicht selbstständig entscheiden, in welchem Operationsmodus sie arbeiten müssen, damit die gewünschte

Funktionalität des Gesamt-Systems erbracht wird. Deshalb sind die Software-Tasks darauf angewiesen, dass ihnen ihr Operationsmodus mitgeteilt wird. AUTOSAR bietet hierfür das sogenannte Mode Switch Interface an, wie es in Abschnitt 2.4.2 erwähnt wurde. Jedoch ist im AUTOSAR-Standard nicht festgelegt, welche Komponenten den Software-Tasks den Operationsmodus mitteilen. Für ein DPM für Automotive Systeme kann diese Aufgabe der PM auf ECU-Ebene übernehmen, der den Power-Zustand der ECU und somit auch die Software-Tasks als PMCs steuert.

Damit der PM die Software-Tasks als PMCs steuern kann, müssen die Software-Tasks allerdings folgende Voraussetzungen erfüllen: Diese müssen unterschiedliche Operationsmodi bereitstellen und deren Steuerung über eine Schnittstelle erlauben, die vom PM genutzt werden kann. Der PM entscheidet dann, in welchem Operationsmodus die Software-Tasks in einem Power-Zustand geschaltet werden.

Als eine Möglichkeit für die Steuerung der Software-Tasks als PMCs schlagen A. Barthels *et al.* vor, die Software-Tasks zusammen mit den Hardware-PMCs über sogenannte Power-Management-Pläne zu steuern [29]. Dies bedeutet, dass ein bestimmter Power-Management-Plan in dem jeweiligen ECU-Zustand ausgeführt wird. Dieser schaltet die Hardware-PMCs in deren vorgesehenen Power-Zustand und regelt über das angepasste Scheduling die Ausführung der Software-Tasks. Darüber hinaus können die Software-Tasks zusätzlich eine Schnittstelle anbieten, über die der PM den Operationsmodus steuern kann.

4.3.2 PM auf ECU-Ebene

Der PM auf ECU-Ebene steuert die PMCs, die im vorherigen Abschnitt vorgestellt wurden. Dies sind sowohl die Hardware-PMCs als auch die Software-PMCs, die einer ECU zugeteilt sind. Wie zu Beginn dieses Kapitels bereits eingeführt wurde, besteht der PM aus einem Beobachter und einem Controller. Der Beobachter erfasst den aktuellen Zustand des Systems und leitet die Informationen an den Controller weiter, der daraus Befehle für das System generiert und somit direkten Einfluss auf das System nimmt. Der Controller besitzt dabei eine Strategie (engl. *policy*), wie er aus den Informationen des Beobachters die Befehle für das System ableitet.

Auf Automotive Systeme angewandt, bedeutet dies, dass der PM auf ECU-Ebene den Fahrzeug-Zustand erfassen und daraus die Power-Zustände der PMCs ableiten und setzen muss. In Abbildung 4.3 ist der PM detaillierter mit seinen Komponenten für Automotive Systeme dargestellt. Auf der linken Seite befindet sich der Beobachter, der den Fahrzeug-Zustand erfasst, indem er die Kommunikation der Software-Tasks im Automotiven System mitverfolgt und die beobachteten Informationen an den Controller weiterleitet. Dieser bestimmt mit Hilfe seiner PM-Logik und seiner PM-Wissensbasis, in welchem Zustand die PMCs auf der ECU über die PMC-Abstraktion gesetzt werden sollen. Im Folgenden wird sowohl auf den Beobachter als auch auf den Controller mit seinen Komponenten detaillierter eingegangen.

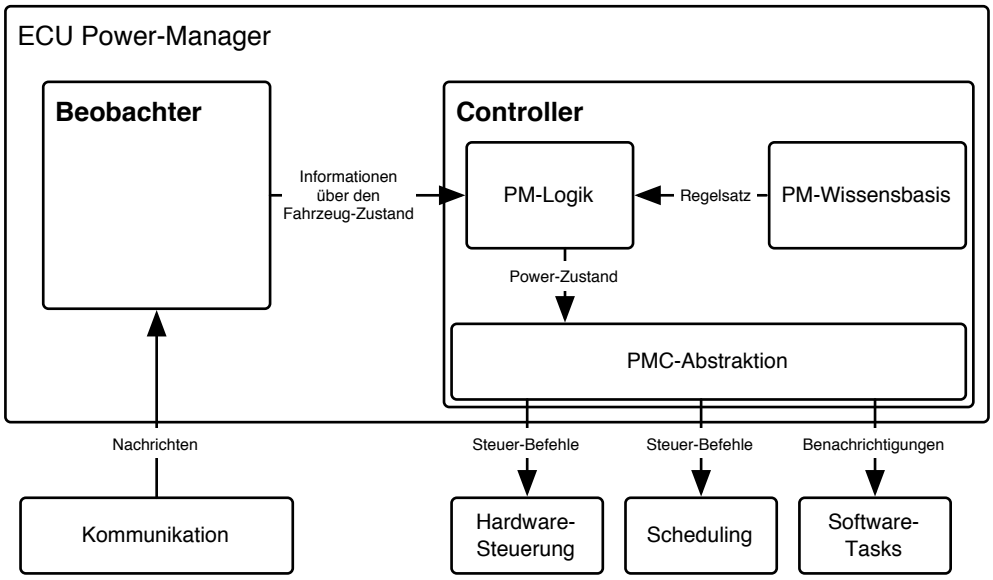


Abbildung 4.3: Übersicht über den PM auf ECU-Ebene.

Beobachter des PMs auf ECU-Ebene

Die Aufgabe des Beobachters ist es, den Fahrzeug-Zustand zu erkennen und an den Controller weiterzuleiten. Dies erscheint im ersten Augenblick als eine einfache Aufgabe. Jedoch besteht das Problem darin, dass der Zustand des gesamten Automotiven Systems auf allen ECUs in den Software-Tasks und Hardware-Komponenten verteilt ist und somit typischerweise eine Komponente alleine nicht den gesamten Fahrzeug-Zustand kennt. Außerdem sind die Automotiven Systeme in verschiedene, abgegrenzte Domänen strukturiert, wodurch die Erkennung des Fahrzeug-Zustands mit Hilfe von Informationen, die in einer anderen Fahrzeug-Domäne gehalten werden, zusätzlich erschwert wird.

Der Beobachter im PM muss die entsprechenden Informationen aus den Software-Tasks extrahieren, um daraus den Fahrzeug-Zustand zu bestimmen. Jedoch müsste dafür eine zusätzliche Schnittstelle definiert werden, über die Software-Tasks ihren Zustand mitteilen können. Außerdem befinden sich die Mehrheit der Software-Tasks nicht auf derselben ECU, was zu einer erhöhten Bus-Kommunikation führen würde, wenn jeder PM den Zustand eines jeden Software-Tasks abfragen würde. Eine erhöhte Bus-Kommunikation bis zur einer Überflutung des Buses durch diese Abfragen der PMs ist zu vermeiden, um die verschiedenen Anforderungen, die in Automotiven Systemen herrschen, zu gewährleisten, wie z. B. die Einhaltung der Echtzeit-Anforderungen.

Die Lösung des Problems liegt darin, dass sich der Beobachter des PMs für die Ermittlung des Fahrzeug-Zustandes auf die Nachrichten stützt, die für die Erfüllung der Funktionen zwischen den Software-Tasks verschickt werden. Dabei sind hier nicht nur die Nachrichten auf den Kommunikationsbussen enthalten, sondern auch die Nachrichten, die zwischen zwei Software-Tasks innerhalb einer ECU ausgetauscht werden. Dies bedeutet, dass die Kommunikation zwischen den Software-Tasks auf eine ECU zentralisiert werden muss, damit der Beobachter des PMs dort die Möglichkeit hat, alle Nachrichten

mitzuverfolgen.

Ebenfalls ist es möglich, dass Nachrichten, die explizit den Fahrzeug-Zustand enthalten, zusätzlich versendet werden, damit den Beobachtern die Erkennung des Fahrzeug-Zustandes erleichtert wird. Über Zustandsnachrichten lässt sich auch das Problem der abgegrenzten Domänen lösen, indem das ZGW die von anderen Domänen benötigten Fahrzeug-Zustandsnachrichten weiterleitet, damit garantiert werden kann, dass die Beobachter der PMs den Zustand des Automotiven Systems korrekt erfassen können.

Damit beschränkt sich der zu erfassende Fahrzeug-Zustand auf die versendeten Nachrichten der Software-Tasks. Unter Umständen ist mit dieser Methode nur eine Teilmenge des Fahrzeug-Zustands für die Beobachter erkennbar, was die Möglichkeiten der PMs einschränkt. Hier kommt vor allem die Granularität der Software-Tasks zum Tragen. Denn je mehr Funktionalität in einem Software-Task zusammengefasst wird, desto weniger Nachrichten werden zwischen Software-Tasks verschickt, da viele Daten, die jetzt nur intern in den Software-Tasks gebraucht werden, die Software-Tasks nicht mehr verlassen. Damit wird es den Beobachtern der PMs erschwert, den Fahrzeug-Zustand oder Änderungen im Fahrzeug-Zustand zu erkennen. Ist die Anzahl der unterteilten Software-Tasks zu hoch, so erhöht sich auch die Anzahl der Nachrichten, die zwischen den Software-Tasks ausgetauscht werden. Dies erleichtert zwar den Beobachtern die Zustandserkennung, jedoch müssen in dem Automotiven System viel mehr Nachrichten versendet werden, was mit einem Kommunikationsoverhead verbunden ist. Somit muss die Anzahl der Software-Tasks, die schon in der Definition der logischen Architektur beginnt, sorgfältig und unter der Berücksichtigung des DPMs vorgenommen werden. Aus diesem Grund ist ein wesentlicher Gesichtspunkt für die Effektivität des DPMs der Abstraktionsgrad der Funktionsblöcke in einem Automotiven Systems.

Controller des PMs auf ECU-Ebene

Der Controller des PMs auf ECU-Ebene hat die Aufgabe aus dem Fahrzeug-Zustand, den der Beobachter aus den abgehörten Nachrichten extrahiert hat, den Power-Zustand seiner PMCs abzuleiten und diesen entsprechend seiner PM-Wissensbasis zu setzen.

Dafür gliedert sich der Controller in die drei Teilkomponenten, die ebenfalls in Abbildung 4.3 dargestellt sind: die PM-Logik, die PM-Wissensbasis und die PMC-Abstraktion [129]. Die PM-Logik ist der Kern des PM-Controllers, der den Fahrzeug-Zustand und dessen Änderung analysiert und daraus den Power-Zustand der PMCs ableitet. Die dafür benötigten Konfigurationen und Regeln sind in der PM-Wissensbasis enthalten. Dies entspricht dem Prinzip der Trennung zwischen Strategie und Mechanismus, wie sie auch in anderen Betriebssystem-Teilen zu finden ist [135]. Die PM-Logik ist dabei der Mechanismus, der die Strategie von der PM-Wissensbasis erhält. Die PMC-Abstraktion ist für die PM-Logik eine Abstraktionsschicht für die zu steuernden Hardware- und Software-PMCs. Damit stellt die PMC-Abstraktion eine einheitliche Schnittstelle für die PM-Logik bereit, um die heterogenen PMCs generisch anzusprechen.

Die PM-Logik hat die Aufgabe, aus dem aktuellen Fahrzeug-Zustand den Power-Zustand aller PMCs auf der ECU zu bestimmen. Wie in den vorherigen Abschnitten beschrieben, muss die PM-Logik die Nachrichten, die in logischen Architektur zwischen den Software-Tasks versendet werden, interpretieren. Dabei muss sie auch den Nachrich-

teninhalt für die Analyse heranziehen. Jedoch muss die PM-Logik eine möglichst deterministische Zeit für das Interpretieren der Nachrichten benötigen, damit das DPMs die harten Echtzeit-Anforderungen des Automotiven Systems unter keinen Umständen verletzt. Da die verschickten Nachrichten unterschiedlich aufgebaute Nachrichtenstrukturen haben, muss ein Algorithmus für die Nachrichten-Analyse entwickelt werden, deren Laufzeit möglichst deterministisch oder zumindest mit einem maximalen Wert abgeschätzt werden kann. Dies steht allerdings im Widerspruch, dass es auch für die Nachrichten-Analyse möglich sein muss, unterschiedlich viele Daten-Elemente, die einen beliebigen Werte-Bereich haben können, aus der Nachrichtenstruktur zu untersuchen.

Die Trennung zwischen der PM-Logik und der PM-Wissensbasis hat den Vorteil, dass die Implementierung der PM-Logik für alle ECUs gleich erfolgen kann. Nur die PM-Wissensbasis muss für jede ECU angepasst werden. Diese ist meistens eine Konfiguration der PM-Logik mit den entsprechenden Regeln. Somit besitzt jede ECU ihren eigenen Regelsatz, der aus der Analyse der logischen Architektur des Automotiven Systems abgeleitet und am besten automatisiert generiert wird. Dabei muss eine Regel-Tabelle für die PM-Logik gefunden werden, die für alle ECUs anwendbar ist. Mit dieser Trennung zwischen PM-Logik und PM-Wissensbasis wird die Anforderung erfüllt, dass das DPM für Automotive Systeme leicht an die heterogenen ECUs angepasst werden kann.

A. Barthels *et al.* schlagen für die PM-Logik einen endlichen Automaten vor, der die Nachrichten aus dem Automotiven System in einen Power-Zustand der ECU übersetzt. Aus dem Power-Zustand wird im Anschluss ein sogenannter Power-Management-Plan abgeleitet, der sowohl den Power-Zustand der Hardware-PMCs setzt als auch einen zyklischen Scheduling-Plan der Software-Tasks festlegt [29].

Die PMC-Abstraktion bietet der PM-Logik eine einheitliche Schnittstelle für die Steuerung der verschiedenen, heterogenen PMCs. Die PM-Logik teilt dabei der PMC-Abstraktion den Power-Zustand einer jeden PMC mit. Die PMC-Abstraktion sorgt im Anschluss dafür, dass die PMCs die geforderten Power-Zustände einnehmen. Dafür nutzt die PMC-Abstraktion von den Betriebssystem-Komponenten die Hardware- und Software-Steuerung sowie das Scheduling.

Für die Hardware-PMCs erfolgt in der PMC-Abstraktion die Anbindung des PMs an die Treiber im Betriebssystem. Typischerweise sind die Zugriffsfunktionen der Hardware-Treiber über die Vielzahl an Hardware-Controller nicht standardisiert, so dass hiermit für die PM-Logik eine Abstraktionsschicht eingeführt wird. In der PMC-Abstraktion erfolgt die Abbildung der abstrakten Power-Zustände der PM-Logik auf die konkreten Power-Zustände der Hardware-PMCs. Für jede Hardware-PMC muss somit in der PMC-Abstraktion eine Steuerung implementiert werden, wie die abstrakten Power-Zustände auf der konkreten Schnittstelle der Hardware-PMC umgesetzt werden können. Dies hat den Vorteil, dass bei der Anbindung einer neuen Hardware-Komponente an den PM nur die PMC-Abstraktionsschicht angepasst werden muss und nicht die PM-Logik selbst, die damit unverändert bleibt.

Für die Steuerung der Software-PMCs muss die PMC-Abstraktion das Scheduling der Software-Tasks auf einer ECU beeinflussen können. Je nach Power-Zustand eines Software-Tasks wird dessen Scheduling entsprechend den Echtzeit-Anforderungen angepasst. Jedoch wird zuvor der Software-Task über die Änderung des Power-Zustandes informiert, damit dieser bei Bedarf seinen internen Zustand entsprechend anpassen kann.

4.4 Zusammenfassung

Im Umfeld des AUTOSAR-Standards sind die drei Konzepte *ECU Degradation*, *Pretended Networking* und *Partial Networking* entstanden, die als Mechanismus einen energieeffizienten Betrieb eines Automotiven Systems ermöglichen. Dennoch muss ein DPM für Automotive Systeme dafür sorgen, dass diese Mechanismen im Gesamt-System angewendet werden. Dabei sind die Anforderungen an ein DPM für Automotive Systeme zu beachten, worunter u. a. das Erreichen eines niedrigen Energieverbrauchs, das Einhalten der funktionalen Sicherheit und der Echtzeit-Fähigkeit, keine wesentliche Steigerung des Ressourcen-Verbrauchs, der leichte Austausch der Power-Management-Konfiguration sowie der leichte Umgang mit der heterogenen Hardware fallen.

Das DPMs auf System-Ebene sind bereits wissenschaftlich untersucht worden und liefern Grundlage und Begriffsbildung, auf die für das automotive DPM aufgesetzt werden kann. Ebenfalls trägt der ACPI-Standard als Schnittstelle für das Power-Management in Server- und Desktop-Systemen als Basis und Orientierung für das DPM für Automotive Systeme bei. Dennoch muss das aus den Server-, Desktop- oder Embedded Systemen bekannte DPM für Automotive Systeme wegen der dort herrschenden Domänen-Struktur mit den insgesamt ca. 80 ECUs angepasst werden. Als logische Konsequenz wird das DPM für Automotive Systeme hierarchisch aufgebaut, womit es einen PM auf System-Ebene gibt, der das Gesamt-System beobachtet und die einzelnen Domänen als PMCs steuert. Hierbei werden insbesondere die Nutzer-Eingaben des Fahrers betrachtet, woraus die Befehle für die Domänen abgeleitet werden. Auf nächst tieferer Ebene arbeitet der PM auf Domänen-Ebene, der zum einen seine Domäne beobachtet und zum anderen die Befehle des PMs auf System-Ebene entgegennimmt. Seine Befehle gehen an den PM auf ECU-Ebene, der die PMCs auf ECU-Ebene steuert. Der PM auf ECU-Ebene ist vergleichbar mit dem PM aus den Server- und Desktop-Systemen, die die Hardware-PMCs kontrollieren. Jedoch leitet der PM auf ECU-Ebene seine Befehle nicht nur aus der Beobachtung ab, sondern erhält seine Befehle ebenfalls von dem übergeordneten PM auf Domänen-Ebene. In Automotiven Systemen werden auch die Software-Tasks als PMCs betrachtet, die vom PM auf ECU-Ebene gesteuert werden. Die Nutzer-Eingaben gehen bereits auf System-Ebene ein und bringen das Gesamt-System in einen bestimmten Zustand, aus dem die PMs auf ECU-Ebene die benötigten Software-Tasks bestimmen und diese zur Ausführung bringen.

Damit der PM auf ECU-Ebene die PMCs steuern kann, muss er den Zustand des Gesamt-Systems erkennen und daraus den Power-Zustand seiner PMCs ermitteln. Dafür leitet der PM auf ECU-Ebene aus allen Nachrichten, die von den lokalen Software-Tasks innerhalb der ECU entweder empfangen oder versendet werden, den für ihn relevanten Fahrzeug-Zustand ab. Daraus werden die Power-Zustände der PMCs ermittelt und diese entsprechend gesetzt. Der Controller des PMs auf ECU-Ebene ist in die Unterkomponenten PM-Logik, die PM-Wissensbasis sowie die PMC-Abstraktion aufgeteilt. Dabei sorgt die Trennung zwischen PM-Logik und PM-Wissensbasis dafür, dass der PM möglichst leicht konfiguriert werden kann, was eine der Anforderungen an das DPM für Automotive Systeme ist. Um die Unterstützung der heterogenen Hardware als spezielle Anforderung Automotiver Systeme zu erfüllen, liefert die PMC-Abstraktion eine einheitliche Schnittstelle für alle zu steuernden PMCs.

Kapitel 5

Die PLASA-Plattform

Am Fachgebiet für Betriebssysteme im Institut für Informatik der Technischen Universität München wurde im PLASA-Projekt eine automotive Plattform nachgebildet, die in ein 1:10 Modell-Fahrzeug eingebaut wurde. Zum einen diente die PLASA-Plattform dazu, automotive Themen in einem Praktikum der technischen Informatik den Studenten näher zu bringen und zum anderen, um verschiedene automotive Themen wissenschaftlich zu untersuchen.

Der Fokus dieser Arbeit liegt darin, ein DPM für Automotive Systeme umzusetzen und in die PLASA-Plattform zu integrieren. Dabei wurden Software-Komponenten und OSS aus den Domänen der mobilen Embedded Systeme und der WSNs eingesetzt. Damit sind die bereits vorhandenen Mechanismen für ein DPM aus diesen Domänen automatisch in die PLASA-Plattform gewandert und müssen lediglich noch für den automotiven Einsatz adaptiert werden. Zusätzlich werden die in der PLASA-Plattform fehlenden automotiven Komponenten entwickelt und in das Modell-Fahrzeug integriert.

In dem Modell-Fahrzeug, das in Abbildung 5.1 zu sehen ist, sind alle ECUs, Sensoren und Aktoren verbaut, so dass die Plattform die im nächsten Abschnitt beschriebenen Anwendungsfälle selbstständig erfüllen kann.

Das Automotive System der PLASA-Plattform besteht aus vier verschiedenen ECUs, die über einen gemeinsamen Bus miteinander kommunizieren können. Auf die Hardware-Architektur der einzelnen ECUs wird vertieft in Abschnitt 5.4.2 eingegangen. An dieser Stelle sollen die vier ECUs mit ihren Aufgaben nur kurz vorgestellt werden: Das PSB (engl. *power supply board*) erzeugt aus der Batterie-Spannung von 7,2 V die 5 V Bordnetz-Spannung, die es dem gesamten Modell-Fahrzeug zur Verfügung stellt. Neben der Spannungswandlung übernimmt das PSB auch die Batterie-Überwachung und meldet, wenn diese eine zu niedrige Spannung aufweist. Das DAB (engl. *drive assistant board*) übernimmt die Ansteuerung des Lenk-Servos und des Fahrten-Reglers und sorgt somit für die Bewegungen des Modell-Fahrzeugs. Das LSB (engl. *light and sensor board*) steuert die Fahrzeug-Beleuchtung und liest die in der Karosserie verbauten Abstands- und Licht-Sensoren aus. Der Gumstix bildet in dem Modell-Fahrzeug die Head-Unit eines Automotiven Systems nach. Außerdem stellt er die Schnittstelle zur Außenwelt des Modell-Fahrzeugs dar, indem über ihn eine WLAN- oder Bluetooth-Verbindung mit

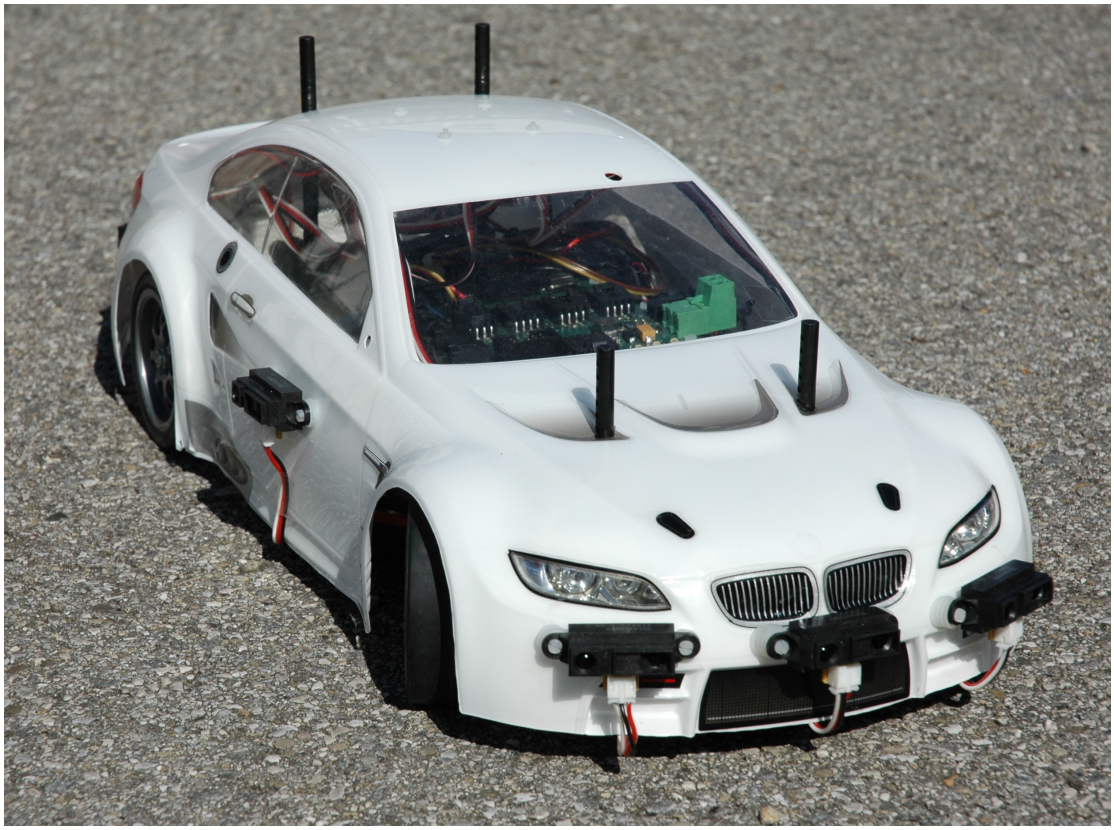


Abbildung 5.1: Das Modell-Fahrzeug der PLASA-Plattform.

dem Modell-Fahrzeug aufgebaut werden kann.

Neben den vier ECUs sind Sensoren und Aktoren im Modell-Fahrzeug verbaut, die schematisch in Abbildung 5.2 dargestellt sind. Als Sensoren kommen sieben Abstandssensoren zum Einsatz, die wie in der Darstellung am Modell-Fahrzeug angebracht sind. Drei der Sensoren sind nach vorne, drei nach hinten und einer zur rechten Seite gerichtet. Neben den Abstandssensoren ist zusätzlich in der Karosserie ein Helligkeitssensor eingebaut. Über einen an der Kurbelwelle des Fahrzeugs angebrachten Rotationssensor kann die Fahrzeug-Geschwindigkeit gemessen werden.

Als Aktoren kommen der Lenk-Servo, der Motor und zehn LEDs (engl. *light-emitting diode*) als Scheinwerfer zum Einsatz. Die modellierten Scheinwerfer sind der eines Fahrzeugs üblichen Beleuchtung nachempfunden. Doch aufgrund der Restriktionen, die sich aus der mechanischen Modell-Konstruktion ergeben, sind nur die folgenden fünf Scheinwerfer im Modell integriert:

Frontlichter. Die zwei weißen Frontscheinwerfer werden für das Stand-, Abblend-, Fern- und Kurvenlicht verwendet. Diese zwei LEDs können getrennt von einander mehrere Helligkeiten annehmen, so dass die verschiedenen Lichtarten simuliert werden können.

Blinker rechts. Die gelben LEDs auf der rechten Seite werden als Blinker verwendet.

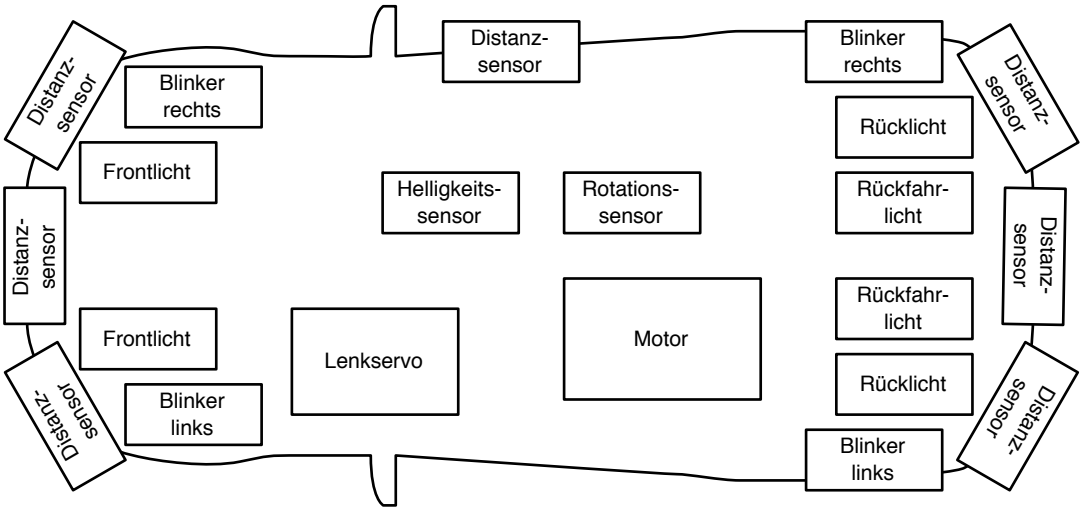


Abbildung 5.2: Schematische Abbildung des Modell-Fahrzeugs mit den vorhandenen Sensoren und Aktoren.

Diese LEDs können nur gemeinsam und in einer Helligkeit aufleuchten.

Blinker links. Die gelben LEDs auf der linken Seite werden ebenfalls als Blinker verwendet. Diese LEDs können wie die Blinker auf der rechten Seite nur gemeinsam und in einer Helligkeit gesteuert werden.

Rücklichter. Die zwei Rücklichter werden als Rückleuchten und als Bremslichter genutzt. Diese LEDs können nur gemeinsam aber in unterschiedlichen Helligkeiten gesteuert werden.

Rückfahrlicht. Die LEDs für die Rückfahrscheinwerfer werden bei einer Rückwärtsfahrt eingeschaltet und können ebenfalls nur in einer Helligkeit gemeinsam aktiviert werden.

Um zusätzliche Sensoren und Steuerelemente für die Bedienung des Modell-Fahrzeugs bereitzustellen, kommt eine Nintendo Wii Remote als Fernsteuerung zum Einsatz, deren Zustandsdaten über Bluetooth an das Modell-Fahrzeug übertragen werden. In Abbildung 5.3 ist die Nintendo Wii Remote schematisch mit ihren Steuerelementen und LEDs abgebildet. Die Nintendo Wii Remote hat 11 Tasten, die genutzt werden können, um das Modell-Fahrzeug zu steuern. Dabei haben diese die unterschiedliche Bezeichnungen „A“, „B“, „1“, „2“, „+“, „-“ und „Home“. Die Taste „B“ sitzt an der Unterseite der Nintendo Wii Remote, weshalb sie mit unterbrochenen Linien dargestellt ist. In dem Steuerkreuz sind die restlichen vier Bedienelemente untergebracht, wobei jede Richtung eine Taste enthält. Eine Ausnahme ist die Taste „Power“, dessen Status von der Nintendo Wii Remote nicht übertragen wird. Bei deren Drücken schaltet sich die Nintendo Wii Remote automatisch aus, nachdem sie die Bluetooth-Verbindung abgebaut hat. Die LEDs L1 bis L4 können ebenfalls kontrolliert werden und können dafür genutzt werden, um Fahrzeug-Zustände anzuzeigen. Zusätzlich besitzt die Nintendo Wii Remote

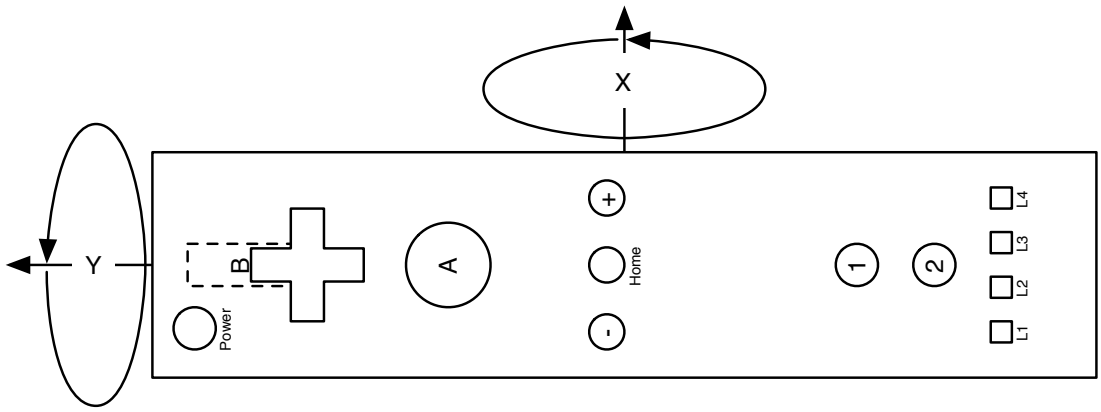


Abbildung 5.3: Schematische Abbildung der Nintendo Wii Remote.

Beschleunigungssensoren, die in der Abbildung mit den Achsen-Bezeichnungen „X“ und „Y“ dargestellt sind. Mit Hilfe dieser Beschleunigungssensoren kann der Neigungswinkel der Nintendo Wii Remote gemessen werden, über den das Modell-Fahrzeug gelenkt wird. In der PLASA-Plattform steuert eine Drehung um die X-Achse die Lenkung des Modell-Fahrzeugs und eine Drehung um die Y-Achse regelt die Leistung des Motors.

Im weiteren Verlauf wird die PLASA-Plattform nach den unterschiedlichen Abstraktionsebenen beschrieben, wie sie in Abschnitt 2.2.2 dargelegt sind. Als erstes werden die Anwendungsfälle der PLASA-Plattform aufgezählt, die als Funktionsumfang für die Untersuchung des DPMs dienen. Auf der Grundlage dieser Anwendungsfälle wird anschließend eine logische Architektur des Modell-Fahrzeugs mit seinen Funktionen entwickelt, die wiederum die Basis für die in der PLASA-Plattform eingesetzten Software-Architektur bildet. Ebenfalls wird erläutert, wie die logische Architektur auf die Software-Architektur abgebildet wird und wie die logische Architektur die Grundlage für das DPM der PLASA-Plattform darstellt. Zum Abschluss erfolgt in Abschnitt 5.4 eine detailliertere Beschreibung der Hardware-Architektur, wie sie in dem Modell-Fahrzeug verbaut ist, und wie die Software-Komponenten auf die ECUs verteilt werden.

5.1 Anwendungsfälle der PLASA-Plattform

Bevor die logische Architektur der PLASA-Plattform im Detail beschrieben wird, werden die Anwendungsfälle, die für die PLASA-Plattform konzipiert wurden, informell vorgestellt. Diese Anwendungsfälle dienen dabei als Basis für die logische Architektur, deren Definition im nächsten Abschnitt erfolgt. Der Umfang der Anwendungsfälle fällt im Vergleich zu den gängigen Automotiven Systemen mit ihren ca. 2000 Kundenfunktionen sehr gering aus. Jedoch ist der Umfang ausreichend, um mögliche Konzepte für ein automotives Power-Management zu untersuchen und zu testen.

Parkzustand. Im Parkzustand reagiert das Modell-Fahrzeug nur auf die Steuereingaben der Nintendo Wii Remote die das Auto in den Zustand des manuellen Fahrens

bringt. Ansonsten zeigt das Auto keine Reaktion auf andere Steuer-Eingaben. In diesem Zustand wird nur die Nintendo Wii Remote als Sensor benötigt. Die Aktoren werden in diesem Fahrzeug-Zustand nicht angesprochen.

Manuelles Fahren mit der Nintendo Wii Remote. Das Modell-Fahrzeug wird in diesem Zustand mit der Nintendo Wii Remote gelenkt. Die Beschleunigungssensoren der Nintendo Wii Remote werden dazu genutzt, das Fahrzeug zu steuern. Per Tastendruck werden verschiedene Funktionen aktiviert, die in den nächsten Anwendungsfällen erklärt werden.

Automatische Geschwindigkeitsregelung. Das Modell-Fahrzeug hält die aktuelle Geschwindigkeit konstant, wobei der Lenkeinschlag noch über die Nintendo Wii Remote gesteuert wird.

Fahrsequenz-Regelung. Das Modellauto fährt eine definierte Strecke ab, die zuvor festgelegt wurde. Somit steuert das Modell-Fahrzeug sowohl die Geschwindigkeitsregelung als auch den Lenkeinschlag selbstständig.

Hindernis-Erkennung. Das Modell-Fahrzeug soll automatisch mit Hilfe der Abstandssensoren Hindernisse in Fahrtrichtung erkennen.

Nothalt. Nach der Erkennung eines Objektes in Fahrtrichtung wird eine Vollbremsung durchgeführt. Die Nothalt-Funktion wird über die Nintendo Wii Remote per Tastendruck aktiviert und deaktiviert.

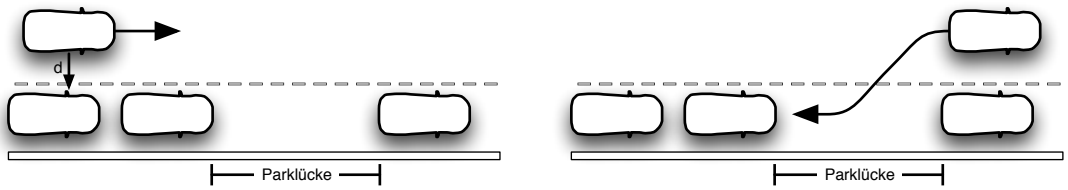
Blinker. Per Tastendruck an der Nintendo Wii Remote können die Blink-Lichter des Fahrzeugs auf der entsprechenden Seite aktiviert bzw. deaktiviert werden.

Automatische Helligkeitssteuerung der Front-Lichter. Die Helligkeit der Front-Lichter wird automatisch mit Hilfe eines Helligkeitssensors gesteuert.

Kurvenlichter. Mit Hilfe der Frontlichter soll ein Kurvenlicht realisiert werden. Dabei soll bei einem Lenkeinschlag nach rechts das rechte Frontlicht heller aufleuchten. Entsprechend soll bei einem Linkseinschlag des Lenk-Servos das linke Frontlicht heller werden [91].

Bremslichter. Die Rücklichter des Modell-Fahrzeugs sollen bei einer Bremsung aufleuchten. Dabei soll die Intensität der Lichter höher sein als bei eingeschaltetem Abblendlicht.

Automatisches Einparken. Per Tastendruck an der Nintendo Wii Remote soll automatisch eingeparkt werden. Dazu werden zwei Schritte ausgeführt, die den Abbildungen 5.4a und 5.4b dargestellt sind. Zuerst wird eine passende Parklücke gesucht. Dabei fährt das Modell-Fahrzeug mit konstanter Geschwindigkeit geradeaus vorwärts. Der Abstandssensor an der rechten Seite misst den seitlichen Abstand d. Sobald dieser über den Grenzwert von 15 cm ansteigt, wird die gefahrene Strecke s gemessen. Der seitliche Abstand muss über dem Grenzwert von 15 cm bleiben, um erfolgreich die Parklücke zu finden. Sobald die gefahrene Strecke s über 80 cm



(a) Die Suche nach einer Parklücke. (b) Das Einparken in eine Parklücke.

Abbildung 5.4: Das automatische Einparken in eine Parklücke.

steigt, ist die Parklücke gefunden. Die gefahrene Strecke s wird mit Hilfe des Rotationssensors gemessen. Sobald die Parklücke gefunden ist, wird mit dem Einparken begonnen. Dazu wird mit Hilfe der Fahrsequenz-Regelung die Fahrsequenz für das Einparken abgefahren.

Batterie-Überwachung. Die Spannung der Batterie im Modell-Fahrzeug soll überwacht werden. Sobald diese unter eine definierte Grenze fällt, soll das Modell-Fahrzeug in den Parkzustand gehen und dies über die LEDs der Nintendo Wii Remote anzeigen.

Die beschriebenen Anwendungsfälle zeigen den kompletten Funktionsumfang des Modell-Fahrzeugs, der als Grundlage für die weiteren Architekturen der PLASA-Plattform dient.

5.2 Logische Architektur der PLASA-Plattform

Die logische Architektur der PLASA-Plattform ergibt sich aus den zuvor beschriebenen Anwendungsfällen, die den Funktionsumfang der Plattform und somit auch die logische Architektur vorgeben. Die logische Architektur basiert auf den sogenannte Funktionsketten, die den Datenfluss von den Daten-Quellen über die Funktionsblöcke zu den Daten-Senken festhalten. Die Funktionsblöcke verarbeiten die eingehenden Daten und berechnen daraus Ausgaben, die an den nächsten Funktionsblock oder an die Datensenke geschickt werden.

In den nächsten Abschnitten wird die logische Architektur der PLASA-Plattform im Detail vorgestellt. Zuerst werden die Sensoren als Daten-Quellen, anschließend die Aktoren als Daten-Senken und die Funktionsblöcke beschrieben, bevor die Nachrichten, die in der logischen Architektur verwendet werden, vorgestellt werden. Aus diesen Komponenten werden die Funktionsketten zusammengestellt, die die logische Architektur der PLASA-Plattform bilden.

5.2.1 Logische Sensoren der PLASA-Plattform

Als Daten-Quellen für die logische Architektur der PLASA-Plattform dienen die Sensoren, die in der Übersicht über die PLASA-Plattform mit ihrer Position in dem Modell-Fahrzeug bereits beschrieben wurden. Diese Sensoren werden in der logischen Architektur verwendet und mit eindeutigen Bezeichnungen versehen. Die Sensoren liefern

Sensor	Typ	Beschreibung
SEN_DIST_FL	Distanz	Abstandssensor vorne links.
SEN_DIST_FC	Distanz	Abstandssensor vorne mitte.
SEN_DIST_FR	Distanz	Abstandssensor vorne rechts.
SEN_DIST_RL	Distanz	Abstandssensor hinten links.
SEN_DIST_RC	Distanz	Abstandssensor hinten mitte.
SEN_DIST_RR	Distanz	Abstandssensor hinten rechts.
SEN_DIST_S	Distanz	Distanz-Sensor seitlich.
SEN_LIGHT_INTENS	Lichtintensität	Helligkeitssensor im Modell-Fahrzeug.
SEN_ODO	Hodometer	Hodometer des Modell-Fahrzeugs.
SEN_SPEED	Geschwindigkeit	Geschwindigkeitssensor des Modell-Fahrzeugs.
SEN_WII_BUTTONS	Knöpfe	Knöpfe der Nintendo Wii Remote.
SEN_WII_ACCEL	Beschleunigung	Beschleunigungssensor der Nintendo Wii Remote.
SEN_BATT_V	Spannung	Spannungssensor der Batterie.

Tabelle 5.1: Logische Sensoren in der PLASA-Plattform.

Datenwerte, die an die logischen Funktionsblöcken zur Verarbeitung weitergereicht werden. In Tabelle 5.1 sind alle Sensoren, die in der PLASA-Plattform verbaut sind, mit ihrer Bezeichnung, ihrem Typ und einer kurzen Beschreibung aufgelistet.

Dabei sind unterschiedliche Sensoren verbaut, die sich in folgende Klassen gliedern lassen: Distanzsensoren, Helligkeitssensoren, Hodometer, Geschwindigkeitssensor, Beschleunigungssensoren der Nintendo Wii Remote, die Knöpfe der Nintendo Wii Remote und die Batteriespannung. Jeder dieser Sensoren versendet seine Messwerte in einer Nachricht, die an die weiteren Funktionsblöcke in der logischen Architektur geleitet werden. Dabei werden die Nachrichten nach den Sensoren benannt, die die Nachricht versenden. Die Sensor-Nachrichten sind zusammen mit den übrigen Nachrichten in dem späteren Abschnitt über die Nachrichten der PLASA-Plattform zusammengefasst.

Die Sensoren mit der Bezeichnung SEN_DIST sind die Abstandssensoren, die an allen Seiten der Karosserie angebracht sind. Diese messen den Abstand zu einem nahe liegenden Objekt und geben diesen Abstand zurück. Der Sensor SEN_LIGHT_INTENS misst die Lichthelligkeit in der Karosserie, die für eine adaptive Lichtsteuerung genutzt werden kann. Für die Fahrsequenz-Regelung und das automatische Einparken wird ein Hodometer benötigt, um die gefahrene Strecke zu messen. Dies wird von dem Sensor SEN_ODO erfüllt. Neben dem Hodometer besitzt das Modell-Fahrzeug noch den Geschwindigkeitssensor SEN_SPEED, der die aktuelle Geschwindigkeit des Fahrzeugs misst. Diese wird hauptsächlich für die Geschwindigkeitsregelung verwendet. Neben den in dem Modell-Fahrzeug verbauten Sensoren wird die Nintendo Wii Remote als Fernbedienung genutzt. Dabei dient der Beschleunigungssensor der Nintendo Wii Remote SEN_WII_ACCEL für die Steuerung des Fahrzeugs, wogegen mit den Tasten der Nintendo Wii Remote SEN_WII_BUTTONS verschiedene Funktionen wie z. B. Blinker, Lichter oder die Einpark-Automatik des Fahrzeugs aktiviert und deaktiviert werden können. Um die Batterie des Modell-Fahrzeugs überwachen zu können, ist in der PLASA-Plattform ein Batterie-Sensor SEN_BATT_V verbaut, der die Batterie-Spannung misst.

Aktor	Art	Beschreibung
ACT_MOTOR	Motor	Motor des Modell-Fahrzeugs.
ACT_SERVO	Servo	Lenk-Servo des Modell-Fahrzeugs.
ACT_LIGHT_FL	Licht	Frontlichter (engl. *front lights*) des Modell-Fahrzeugs.
ACT_LIGHT_RL	Licht	Rücklichter (engl. *rear lights*) des Modell-Fahrzeugs.
ACT_LIGHT_BL	Licht	Blinklichter (engl. *blinker*) des Modell-Fahrzeugs.
ACT_LIGHT_RVL	Licht	Rückfahrlicht (engl. *reversing lights*) des Modell-Fahrzeugs.

Tabelle 5.2: Logische Aktoren in der PLASA-Plattform.

5.2.2 Logische Aktoren der PLASA-Plattform

Neben den Sensoren als Daten-Quellen besitzt die logische Architektur die Aktoren als Daten-Senken. In dem einführenden Abschnitt über die PLASA-Plattform wurden die verschiedenen Aktoren vorgestellt, die in der logischen Architektur wie die Sensoren mit festen Bezeichnungen versehen werden. In Tabelle 5.2 sind alle Aktoren mit ihrer Bezeichnung, Art und Beschreibung aufgelistet.

In der PLASA-Plattform sind zwei Arten von Aktoren in dem Modell-Fahrzeug verbaut. Die eine Art ist für die Bewegung zuständig, worunter der Motor ACT_MOTOR, der das Modell-Fahrzeug antreibt, und der Lenk-Servo ACT_SERVO, der den Vorderrad-Einschlag des Modell-Fahrzeug steuert, fallen. Die Nachrichten, die die Bewegungsaktoren regeln, sind nach dem zu steuernden Aktor benannt. So werden die Steuer-Nachrichten für den Motor mit MSG_MOTOR und die für den Lenk-Servo mit MSG_SERVO bezeichnet. Eine Übersicht über alle Nachrichten der PLASA-Plattform ist im Abschnitt 5.2.4 zu finden.

Die zweite Art von Aktoren sind die Lichter ACT_LIGHT. Dabei werden die unterschiedlichen Lichter als eigener Aktor aufgefasst. So werden die Front-Lichter als ACT_LIGHT_FL, die roten Rücklichter als ACT_LIGHT_RL und die weißen Rückfahrlichter als ACT_LIGHT_RVL bezeichnet. Die rechten und linken Blink-Lichter werden in der logischen Architektur der PLASA-Plattform als ACT_LIGHT_BL zusammengefasst.

5.2.3 Logische Funktionsblöcke der PLASA-Plattform

Neben den Sensoren und Aktoren existieren in der logischen Architektur Funktionsblöcke, die Eingaben verarbeiten und daraus Ausgaben erzeugen, die wieder als Nachrichten versendet werden. Dabei können die Funktionsblöcke unterschiedliche Funktionen, wie Steuerungsaufgaben oder das Umrechnen von Sensor-Werten, übernehmen. In allen Fällen erhalten sie Nachrichten als Eingabe, aus denen wieder Nachrichten als Ausgabe verschickt werden. Die Frage, wie die Funktionsblöcke einer Plattform geschnitten werden, beruht in der PLASA-Plattform auf Ingenieursleistung.

Die Funktionsblöcke können in verschiedenen Funktionsketten verwendet werden. Allerdings handelt es sich bei jeder Verwendung um dieselbe Instanz des Funktionsblocks. Falls dieselbe Funktionslogik mit unterschiedlichen Parametern oder Zuständen in mehreren Funktionsketten benötigt wird, muss ein neuer Funktionsblock mit eigener Instanz

Funktionsblock	Beschreibung
FCN_CAR_STATE	Berechnung und Kontrolle des Fahrzeug-Zustands aus den Benutzer-Eingaben über die Knöpfe der Nintendo Wii Remote.
FCN_MANUAL_MOTOR	Berechnung der Motor-Leistung aus den Beschleunigungswerten der Nintendo Wii Remote.
FCN_MANUAL_SERVO	Berechnung des Lenk-Einschlags aus den Beschleunigungswerten der Nintendo Wii Remote.
FCN_SPEED_SET	Setzen der aktuellen Geschwindigkeit als Soll-Geschwindigkeit des Geschwindigkeitsreglers.
FCN_SPEED_CTRL	Geschwindigkeitsregler über die Leistung des Motors aus der aktuell gefahrenen Geschwindigkeit und einer Soll-Geschwindigkeit.
FCN_DRIVE_SEQUENCE	Regler für das Abfahren einer Fahrsequenz.
FCN_FIND_PARKING_SPACE	Überwachung des seitlichen Abstandssensors für die Suche nach einer passenden Parklücke.
FCN_PARKING_SEQUENCE	Generator der Fahrsequenz für das automatische Einparken.
FCN_PARKING_END	Überwachung des Einpark-Vorgangs und Versenden des Zustands beim erfolgreichem Einparken.
FCN_OBSTACLE_F	Überwachung der vorderen Abstandssensoren und Erkennung eines Hindernisses in der Vorwärtsfahrt.
FCN_OBSTACLE_R	Überwachen der interen Abstandssensoren und Erkennung eines Hindernisses in der Rückwärtsfahrt.
FCN_EMERG_STOP	Auslösen des Nothalts beim Erkennen eines Hindernis in Fahrtrichtung.
FCN_BATT_CTRL	Überwachung des Batterie-Sensors und Versenden einer Nachricht bei zu niedriger Batterie-Spannung.

Tabelle 5.3: Logische Funktionsblöcke in der PLASA-Plattform.

dafür erzeugt werden.

Um einen Überblick über alle Funktionsblöcke zu geben, sind in Tabelle 5.3 die Funktionsblöcke aus der logischen Architektur der PLASA-Plattform mit einer kurzen Beschreibung aufgelistet. Einige Funktionsblöcke werden im Folgenden exemplarisch mit ihren Aufgaben detaillierter beschrieben.

FCN_MANUAL_MOTOR. Der Funktionsblock FCN_MANUAL_MOTOR ist für die Berechnung der Motor-Leistung, die anschließend an den Motor gesendet wird, aus den Beschleunigungswerten auf der Y-Achse der Nintendo Wii Remote zuständig.

FCN_MANUAL_SERVO. Der Einschlag des Lenk-Servos wird entsprechend den Beschleunigungswerten auf der X-Achse der Nintendo Wii Remote im Funktionsblock FCN_MANUAL_SERVO berechnet. Im Anschluss wird dieser an den Lenk-Servo geschickt.

FCN_SPEED_CTRL. Der Funktionsblock FCN_SPEED_CTRL implementiert ein Geschwindigkeitsregler, der eine zuvor eingestellte Soll-Geschwindigkeit einregelt. Dabei benötigt er die aktuelle Geschwindigkeit als Eingabe-Wert. Als Ausgabe erzeugt er die Nachricht MSG_MOTOR, mit der er die Leistung des Motors steuert.

FCN_DRIVE_SEQUENCE. Die Funktion der Fahrsequenz-Regelung ist im Funktionsblock FCN_DRIVE_SEQUENCE gekapselt. Dabei wird die Strecke in Segmente eingeteilt, die jeweils Soll-Geschwindigkeit, Beschleunigung, Lenk-Einschlag und Länge des Segments enthalten. Die Regelung sorgt dafür, dass die Segmente der Reihe nach abgefahren werden. Neben der Sequenz benötigt der Funktionsblock als Eingabe die aktuell gefahrene Strecke, die er vom Hodometer bekommt. Mit der Nachricht MSG_ADD_SEG werden neue Segmente zu der Liste hinzugefügt. Mit der Nachricht MSG_DRIVE_STATUS wird der aktuelle Status der Fahrsequenz-Regelung verschickt, in der die Segment-Nummer und die aktuell gefahrene Strecke im Segment enthalten sind. Zum Beginn eines neuen Segments wird die Soll-Geschwindigkeit an den Geschwindigkeitsregler und der Lenk-Einschlag an den Servo gesendet.

FCN_OBSTACLE_F. Für die Hindernis-Erkennung in der Vorwärtsfahrt des Modell-Fahrzeugs ist der Funktionsblock FCN_OBSTACLE_F zuständig. Diese bekommt als Eingabe die Sensor-Werte der drei Abstandssensoren, die vorne an der Karosserie angebracht sind. Daraus wird der Abstand zu einem Objekt berechnet, das sich vor dem Modell befindet. Dieser berechnete Abstand wird in der Nachricht MSG_OBSTACLE weiter versendet.

FCN_OBSTACLE_R. Die Hindernis-Erkennung in der Rückwärtsfahrt des Modell-Fahrzeugs übernimmt der Funktionsblock FCN_OBSTACLE_R. Dabei werden die Sensor-Werte der Abstandssensoren, die hinten am Modell-Fahrzeug befestigt sind, für die Berechnung herangezogen. Der Funktionsblock sendet den Abstand zum erkannten Objekt in derselben Nachricht MSG_OBSTACLE wie der Funktionsblock FCN_OBSTACLE_F, der die Hindernis-Erkennung für die Vorwärtsfahrt übernimmt.

FCN_EMERG_STOP. Falls ein Hindernis in Fahrtrichtung zu nah ist, sorgt der Funktionsblock FCN_EMERG_STOP für das Auslösen des Nothalts. Dabei nimmt er die aktuelle Geschwindigkeit und den Abstand des nächsten Objekts und berechnet daraus, ob ein Nothalt ausgelöst werden muss. Im Anschluss wird mit der Nachricht MSG_EMERG_STOP das Auslösen des Nothalts signalisiert.

Auf eine detailliertere Beschreibung der übrigen Funktionsblöcke wird an dieser Stelle verzichtet und auf die Übersicht in Tabelle 5.3 verwiesen, in der alle Funktionsblöcke kurz erläutert wurden.

5.2.4 Logische Nachrichten der PLASA-Plattform

Nachdem Sensoren, Aktoren und Funktionsblöcke der logischen Architektur beschrieben wurden, werden als nächstes die logischen Nachrichten, die von den Blöcken versendet werden, näher erläutert. Die Nachrichten sind in der logischen Architektur als Broadcast-Nachrichten zu verstehen, die einmal ausgesendet von allen logischen Komponenten empfangen werden, die diese erwarten. Dabei werden bei den Nachrichten zwischen Daten- und Zustandsnachrichten unterschieden. Während die Daten-Nachrichten

Nachricht	Beschreibung
MSG_ODO	Aktuell gefahrene Strecke.
MSG_SPEED	Aktuelle Geschwindigkeit des Modell-Fahrzeugs.
MSG_WII_BUTTONS	Zustand der Nintendo Wii Remote Knöpfe.
MSG_WII_ACCEL	Beschleunigungswerte der Nintendo Wii Remote.
MSG_DIST_FL	Messwert des Abstandssensors vorne links.
MSG_DIST_FC	Messwert des Abstandssensors vorne mitte.
MSG_DIST_FR	Messwert des Abstandssensors vorne rechts.
MSG_DIST_RL	Messwert des Abstandssensors hinten links.
MSG_DIST_RC	Messwert des Abstandssensors hinten mitte.
MSG_DIST_RR	Messwert des Abstandssensors hinten rechts.
MSG_DIST_S	Messwert des seitlichen Abstandssensors.
MSG_LIGHT_INTENS	Messwert des Helligkeitssensors.
MSG_BATT_V	Messwert der Batterie-Spannung.
MSG_MOTOR	Leistung des Motors.
MSG_SERVO	Lenkeinschlag für den Lenk-Servo.
MSG_SPEED_CTRL	Setzen der Soll-Geschwindigkeit für die Geschwindigkeitsregelung.
MSG_ADD_SEG	Hinzufügen eines Segments in der Fahrsequenz-Regelung.
MSG_DRIVE_STATUS	Status der Fahrsequenz-Regelung.
MSG_OBSTACLE	Abstand des nächsten Hindernisses in Fahrtrichtung.

Tabelle 5.4: Logische Daten-Nachrichten in der PLASA-Plattform.

in den logischen Funktionsketten die Daten-Werte zwischen den Blöcken weiterleiten und somit zur Regelung der Aktoren benötigt werden, senden die logischen Funktionsblöcke Zustandsnachrichten, um einen neuen Fahrzeug-Zustand im System zu verteilen.

Die Aktoren müssen periodisch gesteuert werden, wofür die Daten-Nachrichten in der logischen Architektur der PLASA-Plattform existieren. Diese werden von den Sensoren oder Funktionsblöcken versendet und können wiederum von Funktionsblöcken oder am Ende einer Funktionskette schließlich von Aktoren empfangen werden. In Tabelle 5.4 sind die logischen Daten-Nachrichten der PLASA-Plattform mit ihrem Namen und kurzer Beschreibung aufgelistet.

Im ersten Teil der Tabelle sind die Nachrichten, die von Sensoren erzeugt werden, und deren Messwerte enthalten. So übermittelt die Nachricht MSG_SPEED die aktuelle Geschwindigkeit des Modell-Fahrzeugs, die vom Geschwindigkeitssensor ermittelt wurde. Das Hodometer sendet die Nachricht MSG_ODO mit der aktuell gefahrenen Strecke. Ebenfalls die von der Nintendo Wii Remote empfangenen Sensor-Werte werden mit den beiden Nachrichten SEN_WII_BUTTONS und SEN_WII_ACCEL für die Knöpfe bzw. den Beschleunigungswerten an die zu verarbeitenden Funktionsblöcke geschickt. Außerdem versenden die an der Karosserie angebrachten Abstands- und Helligkeitssensoren ihre Messwerte in den entsprechenden Nachrichten.

Die Aktoren erhalten Nachrichten für deren Steuerung. So existieren die Nachrichten MSG_MOTOR und MSG_SERVO, die jeweils für die Ansteuerung des Motors und des Lenk-Servos dienen. Allerdings werden diese Nachrichten auch für die Steuerung der Lichter verwendet, wie z. B die Nachricht MSG_MOTOR die Rückfahrlichter steuert, da

Nachricht	Beschreibung
MSG_MANUAL	Zustandsnachricht für das manuelle Fahren.
MSG_SPEED_CTRL	Zustandsnachricht für das Fahren mit Geschwindigkeitsregelung.
MSG_PARKING_SPACE	Zustandsnachricht für das Suchen einer Parklücke.
MSG_PARKING	Zustandsnachricht für das Einparken.
MSG_PARKED	Zustandsnachricht für den Park-Zustand.
MSG_EMERG_ON	Zustandsnachricht für das Einschalten des Nothalts.
MSG_EMERG_OFF	Zustandsnachricht für das Ausschalten des Nothalts.
MSG_EMERG_STOP	Zustandsnachricht für das Einleiten eines Nothalts.
MSG_LIGHT_ON	Zustandsnachricht für das Einschalten der Scheinwerfer.
MSG_LIGHT_OFF	Zustandsnachricht für das Ausschalten der Scheinwerfer.
MSG_BLINKER_ON	Zustandsnachricht für das Einschalten Blink-Lichter.
MSG_BLINKER_OFF	Zustandsnachricht für das Ausschalten Blink-Lichter.
MSG_BATT_LOW	Zustandsnachricht für eine zu niedrige Batterie-Spannung.

Tabelle 5.5: Logische Zustandsnachrichten in der PLASA-Plattform.

in der Nachricht auch enthalten ist, ob das Modell-Fahrzeug gerade rückwärts fährt.

Weitere Nachrichten übergeben manchen Funktionsblöcken ihre Parameter. So setzt die Nachricht MSG_SPEED_CTRL die Soll-Geschwindigkeit für die Geschwindigkeitsregelung. Auch die Fahrsequenz-Regelung erhält über die Nachricht MSG_ADD_SEG die Segmente, die sie im Anschluss zu ihrer Segment-Liste hinzufügt und abfährt. Damit weitere Funktionsblöcke auf den aktuellen Status der Fahrsequenz-Regelung reagieren können, wird dieser über die Nachricht MSG_DRIVE_STATUS an die übrigen Komponenten der logischen Architektur verteilt.

Dies war ein kleiner Auszug aus der Beschreibung der logischen Daten-Nachrichten, die in den Funktionsketten zur Steuerung der Aktoren verwendet werden. Als zweite Art von Nachrichten, die in der logischen Architektur der PLASA-Plattform verwendet werden, sind die Zustandsnachrichten zu nennen, die, wie der Name es schon ausdrückt, Änderungen im Fahrzeug-Zustand signalisieren. In der logischen Architektur der PLASA-Plattform bedeutet dies, dass einzelne Funktionsketten aktiv geschaltet oder deaktiviert werden. Um abhängig von Sensor-Werten eine Fahrzeug-Zustandsänderung herbeizurufen, werden zusätzliche Funktionsblöcke benötigt. Diese erhalten als Eingaben die benötigten Sensor-Werte oder Nachrichten, die aus den Sensor-Werten abgeleitet werden, und berechnen daraus einen Fahrzeug-Zustand, den sie als Zustandsnachricht versenden. In den Funktionsketten, die im nächsten Abschnitt vorgestellt werden, werden Zustandsnachricht mit einem senkrechten gestrichelten Pfeil nach oben bzw. nach unten gesondert gekennzeichnet, um darzustellen, dass diese nicht Teil des Datenpfades von Sensor zu Aktor sind. Eine Übersicht über die Zustandsnachrichten, die in der PLASA-Plattform verwendet werden, gibt Tabelle 5.5.

Die Zustandsnachrichten enthalten im Wesentlichen die Aktivierung und Deaktivierung von bestimmten Fahrzeug-Funktionen, die im Abschnitt 5.1 über die Anwendungsfälle der PLASA-Plattform vorgestellt wurden. So wird die Nothalt-Funktion von der Nachricht MSG_EMERG_ON aktiviert, während die Nachricht MSG_EMERG_OFF diese deaktiviert. Sobald der Funktionsblock FCN_EMERG_STOP eine Nothalt-Situati-

on erkennt, versendet dieser die Nachricht MSG_EMERG_STOP. Dies versetzt das Modell-Fahrzeug in einen neuen Zustand, der das Modell zu einer Vollbremsung veranlasst. Beim Einparken verhält es sich ähnlich. Zum Starten des automatischen Einparkens wird die Nachricht MSG_PARKING_SPACE gesendet. Sobald die Parklücke in der ersten Phase des Einparkens gefunden wurde, wird die Nachricht MSG_PARKING von dem verantwortlichen Funktionsblock FCN_FIND_PARKING_SPACE gesendet. Dies sorgt dafür, dass nun das Modell-Fahrzeug über die Fahrsequenz-Regelung in die gefundene Parklücke einparkt. Für die Lichtsteuerung existieren ebenfalls jeweils zwei Nachrichten, um die Front- und Rücklichter ein- und auszuschalten, sowie die Blinker zu aktivieren.

Somit ändern die Zustandsnachrichten den Zustand des Fahrzeugs, was in der logischen Architektur eine Aktivierung oder Deaktivierung der Funktionsketten bedeutet, die im nächsten Abschnitt vorgestellt werden.

5.2.5 Logische Funktionsketten der PLASA-Plattform

Nachdem die Sensoren, die Aktoren, die dazwischen liegenden Funktionsblöcke und die Nachrichten erläutert wurden, werden die Funktionsketten in der PLASA-Plattform beschrieben. Hierbei dienen die Anwendungsfälle als Grundlage, woraus die logischen Funktionsketten generiert werden.

Die Funktionsketten zeigen, welche Komponenten aus der logischen Architektur für die Steuerung eines Aktors notwendig sind. Dabei besitzt jeder Aktor eine Aktiv-Bedingung, woraus sich über die Funktionskette ableitet, welche weiteren Komponenten wie Sensoren und Funktionsblöcke in der Aktiv-Bedingung des Aktors benötigt werden.

In der PLASA-Plattform dienen diskrete Fahrzeug-Zustände als Aktiv-Bedingungen. Jeder Aktor hat einen deterministischen Automaten, nach dessen Zuständen die entsprechenden Funktionsketten den Aktor steuern. Die Zustandsübergänge sind Fahrzeug-Zustandsnachrichten, die ebenfalls von Funktionsblöcken aufgrund von Sensor-Daten versendet werden. Sollte keine Funktionskette für einen Aktor aktiv sein, wird der Aktor nicht benötigt und kann ausgeschaltet werden. Ausgehend von den Aktoren in den jeweiligen Fahrzeug-Zuständen, können die benötigten Funktionsblöcke und Sensoren durch eine Tiefen-Suche ermittelt werden. Die in der dazu komplementären Menge befindlichen Sensoren und Funktionsblöcke werden nicht benötigt, und können in dem jeweiligen Fahrzeug-Zustand ausgeschaltet bzw. nicht mehr ausgeführt werden.

Neben der Aktiv-Bedingung besitzt ein jeder Aktor eine Regel-Frequenz, mit der er mindestens gesteuert werden muss. D. h. der Aktor bekommt mit dieser Frequenz seine Nachrichten mitgeteilt. Aus der Funktionskette wird dabei ersichtlich, in welcher Frequenz die Daten-Quellen, also die Sensoren, ihre Daten senden müssen. Im Anschluss müssen die Funktionsblöcke aus jeder empfangenen Nachricht eine Ausgangsnachricht verschicken.

Im Folgenden werden nun für die einzelnen Aktoren die Funktionsketten mit deren dazugehörigen Aktiv-Bedingungen beschrieben. Zur besseren Übersicht können auch mehrere Aktoren in einer Funktionskette enthalten sein, wenn diese von denselben Sensoren und Funktionsblöcken geregelt werden und dieselben Aktiv-Bedingungen haben.

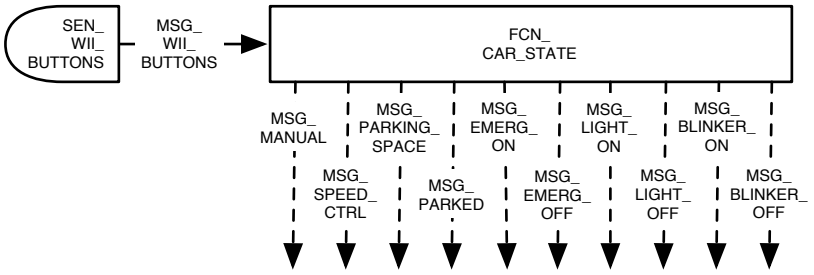


Abbildung 5.5: Funktionskette für die Knöpfe der Nintendo Wii Remote.

Tasten der Nintendo Wii Remote

Als Eingabe-Gerät für die Steuerung des Modell-Fahrzeugs kommt die Nintendo Wii Remote zum Einsatz. Die Aktivierung der Lichter und Scheinwerfer oder der Nothalt-Funktion geschieht mit Hilfe der Knöpfe der Nintendo Wii Remote. Aus diesem Grund gibt es die in Abbildung 5.5 dargestellte Funktionskette, die keine Aktoren steuert, sondern nur die Fahrzeug-Zustandsnachrichten erzeugt, die im Anschluss weitere Funktionsketten aktiv schaltet.

Je nach gedrückter Taste wird eine entsprechende Fahrzeug-Zustandsnachricht gesendet. Dabei besitzt der Funktionsblock FCN_CAR_STATE einen Zustandsautomat, der bei Tastendruck bestimmte Funktionen ein- und ausschaltet.

Motor ACT_MOTOR und der Lenk-Servo ACT_SERVO

Für die beiden Aktoren ACT_MOTOR und ACT_SERVO, die für die Bewegungen des Modell-Fahrzeugs verantwortlich sind, gibt es einen gemeinsamen Zustandsautomaten, nach dem deren Funktionsketten geschaltet werden. In der Tabelle 5.6 sind die dazugehörenden Zustände, die aus den Anwendungsfällen der PLASA-Plattform durch Ingenieursleistung erarbeitet worden sind, aufgelistet. So gibt es für die verschiedenen Steuerungsarten des Modell-Fahrzeugs unterschiedliche Zustände. In einem Zustand werden die beiden Aktoren manuell über die Nintendo Wii Remote gesteuert oden in einem weiteren Zustand wird die Leistung des Motors über eine Geschwindigkeitsreglung kontrolliert. Es existieren weitere Zustände für die in zwei Phasen geteilte Einpark-Automatik, für den Parkzustand oder für den Nothalt.

Zustand	Funktionale Beschreibung
S_MANUAL	Manuelles Fahren.
S_SPEED_CTRL	Geschwindigkeitsgeregelte Fahrt.
S_PARKING_SPACE	1. Teil des Einparkens: Suche der Parklücke.
S_PARKING	2. Teil des Einparkens: Einparken in die Parklücke.
S_EMERG_STOP	Ausführen des Nothalts.
S_PARKED	Parkzustand des Modell-Fahrzeugs.

Tabelle 5.6: Zustände für die Aktoren ACT_MOTOR und ACT_SERVO.

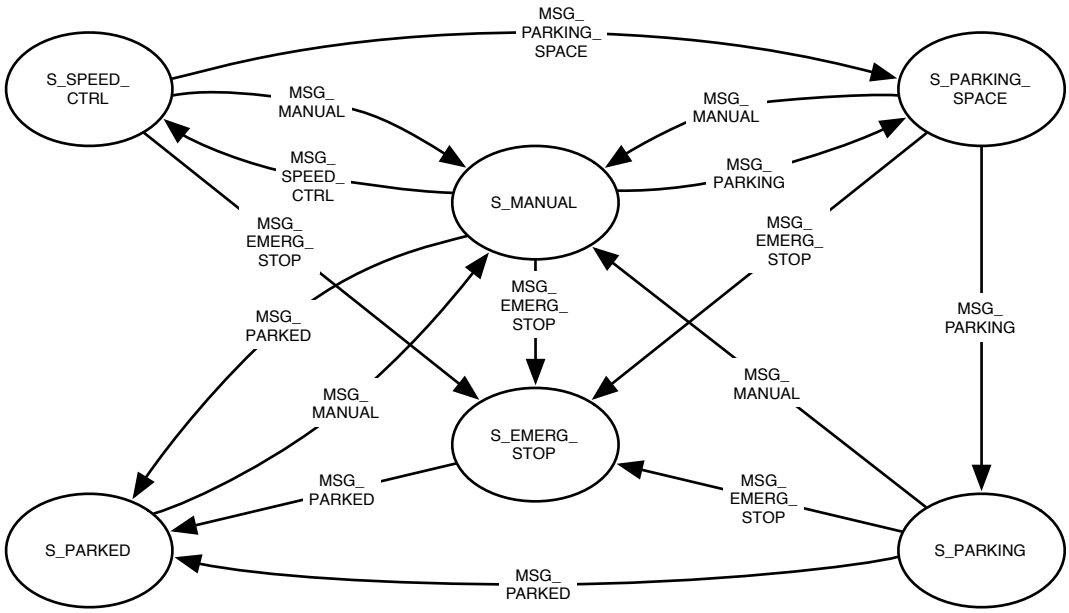
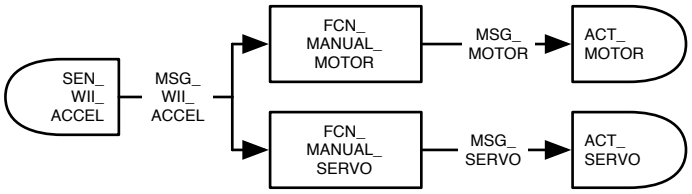


Abbildung 5.6: Zustandsgraph für die Aktoren ACT_MOTOR und ACT_SERVO.

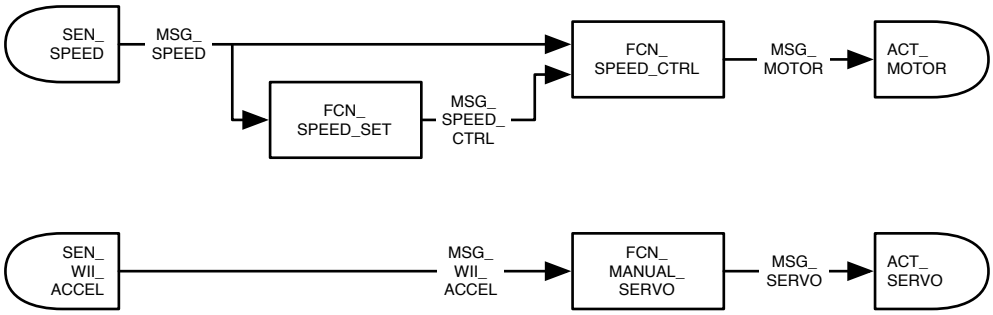
Der dazugehörige Zustandsübergangsgraph ist in Abbildung 5.6 dargestellt. Ausgehend von dem Zustand S_PARKED, in dem die beiden Aktoren ausgeschaltet sind, wird über die Nachricht MSG_MANUAL in den manuell gesteuerten Betrieb des Modell-Fahrzeugs gewechselt. Der Zustand S_SPEED_CTRL wird mit Hilfe der Zustandsnachricht MSG_SPEED_CTRL eingenommen, in dem die Leistung des Motors von dem Geschwindigkeitsregler kontrolliert wird. Über die Nachricht MSG_PARKING_SPACE wird der erste Teil der Einpark-Automatik gestartet. Sobald die Parklücke gefunden wurde, wird mit dem Einparken in die Lücke begonnen, was in dem Fahrzeug-Zustand S_PARKING geschieht. Die übrigen Zustandswechsel durch die entsprechenden Zustandsnachrichten können aus der Abbildung 5.6 entnommen werden.

Für jeden dieser Zustände existieren Funktionsketten für die Steuerung der beiden Aktoren ACT_MOTOR und ACT_SERVO. Dabei gilt für alle Funktionsketten, dass die beiden Aktoren mit einer Frequenz von 20 Hz gesteuert werden, d. h. dass die Funktionsketten alle 50 ms durchlaufen werden. Die Funktionskette für den Zustand S_MANUAL ist in Abbildung 5.7a dargestellt. Als Sensor wird der Beschleunigungssensor der Nintendo Wii Remote genutzt, aus der die Motor-Leistung und der Lenk-Einschlag in dem Funktionsblock FCN_MANUAL_MOTOR bzw. FCN_MANUAL_SERVO berechnet wird. Anschließend werden die Steuerungsnachrichten MSG_MOTOR und MSG_SERVO an den Motor bzw. den Lenk-Servo geschickt.

Als nächste Funktionskette wird die Geschwindigkeitsregelung beschrieben, die graphisch in Abbildung 5.7b dargestellt ist. Diese sorgt dafür, dass die eingestellte Soll-Geschwindigkeit von dem Modell-Fahrzeug gehalten wird. Dafür wird der Geschwindigkeitssensor SEN_SPEED, der Funktionsblock FCN_SPEED_CTRL benötigt, der die Nachrichten für den Motor generiert. Dabei wird der Lenk-Servo immer noch über den



(a) Funktionskette für das manuelle Fahren im Zustand S_MANUAL.



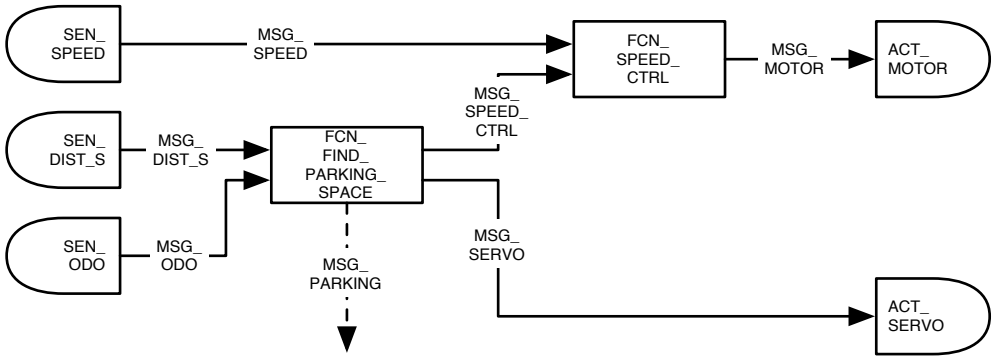
(b) Funktionskette für die geschwindigkeitsgeregelte Fahrt im Zustand S_SPEED_CTRL.

Abbildung 5.7: Funktionsketten für die Aktoren ACT_MOTOR und ACT_SERVO in den Fahrzeug-Zuständen S_MANUAL und S_SPEED_CTRL.

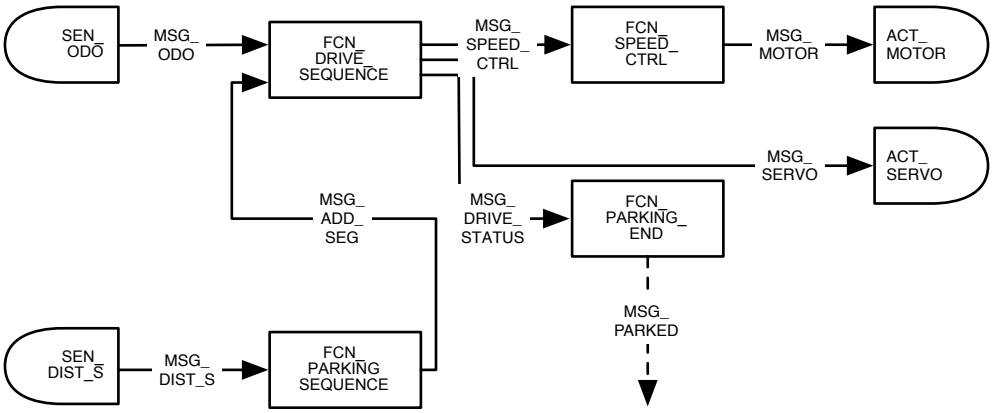
Funktionsblock FCN_MANUAL_SERVO gesteuert, der die Beschleunigungswerte der Nintendo Wii Remote als Eingabe heranzieht.

Das automatische Einparken ist in zwei Teilschritte gegliedert, die in den Fahrzeug-Zuständen S_PARKING_SPACE und S_PARKING ausgeführt werden. Zuerst wird nach einer passenden Parklücke im Zustand S_PARKING_SPACE gesucht, wofür die Funktionskette in Abbildung 5.8a zuständig ist. Die Suche nach der Parklücke übernimmt dabei der Funktionsblock FCN_FIND_PARKING_SPACE, der den Abstand von dem seitlichen Distanz-Sensors und die gefahrene Strecke durch das Hodometer als Eingaben erhält. Als Ausgabe legt er die Geschwindigkeit fest, mit der das Auto geradeaus vorwärts fahren soll. Sobald eine passende Parklücke gefunden wurde, sendet er die Zustandsnachricht MSG_PARKING und löst somit den Zustandswechsel in den Zustand S_PARKING aus.

Im Zustand S_PARKING wird die Fahrsequenz-Regelung mit der vordefinierten Strecke für das Einparken aktiviert. Die dafür geltende Funktionskette ist in Abbildung 5.8b zu sehen. Aus dem aktuell gemessenen Abstand zum seitlichen Objekt berechnet der Funktionsblock FCN_PARKING_SEQUENCE die für das Einparken notwendigen Segmente, die an den Funktionsblock FCN_DRIVE_SEQUENCE übergeben werden. Aus der Segment-Liste der Fahrsequenz-Regelung wird die Geschwindigkeit eingestellt, die der Funktionsblock FCN_SPEED_CTRL einregeln soll. Zusätzlich wird für das Abfahren der Segmente das Hodometer genutzt, damit das Erreichen eines Segment-Endes erkannt wird. Neben den Steuer-Nachrichten für den Servo und die Geschwindigkeitsregelung sendet der Funktionsblock FCN_DRIVE_SEQUENCE noch Status-Nachrichten über den aktuellen Zustand der Fahrsequenz, so dass der Funktionsblock FCN_PARKING_END



(a) Funktionskette für die Suche nach einer Parklücke im Zustand S_PARKING_SPACE.



(b) Funktionskette für das automatische Einparken im Zustand S_PARKING.

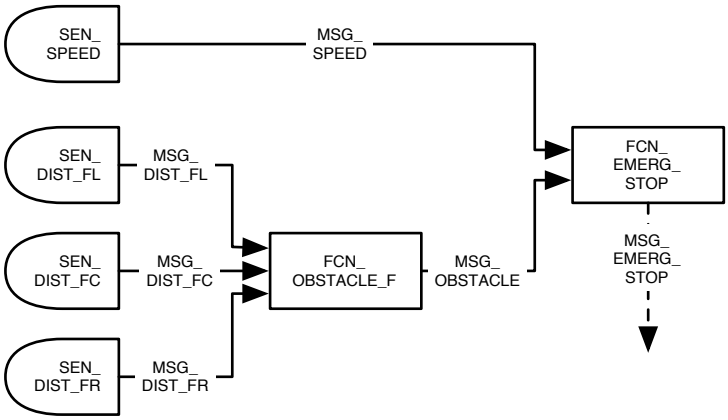
Abbildung 5.8: Funktionsketten für die Aktoren ACT_MOTOR und ACT_SERVO in den Fahrzeug-Zuständen S_PARKING_SPACE und S_PARKING.

erkennen kann, ob das Einparken abgeschlossen ist. Sobald dies der Fall ist, sendet dieser die Nachricht MSG_PARKED, die den Parkzustand des Modell-Fahrzeugs auslöst.

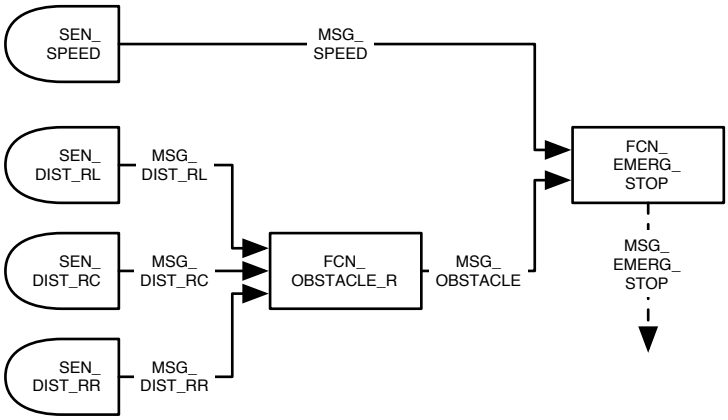
Aus den beschriebenen Funktionsketten für den Motor und den Lenk-Servo in den Zustände S_MANUAL, S_SPEED_CTRL, S_PARKING_SPACE und S_PARKING wird ersichtlich, wie die Funktionsketten für den Motor und den Lenk-Servo grundsätzlich aufgebaut sind. Für den Zustand S_PARKED existieren keine Funktionsketten für den Motor oder den Lenk-Servo, da die beiden Aktoren ausgeschaltet sind, womit auch keine weiteren Sensoren oder Funktionsblöcke für die beiden Aktoren benötigt werden.

Hindernis-Erkennung und Nothalt

Eine weitere funktionale Anforderung an das Modell-Fahrzeug ist die Erkennung von Hindernissen in Fahrtrichtung. Dafür sind sowohl vorne als auch hinten drei Abstandssensoren an der Karosserie verbaut, die jeweils die Entfernung zum nächstgelegenen Objekt liefern. Die Abbildung 5.9a zeigt die Funktionskette für die Hindernis-Erkennung in der Vorwärtsfahrt, in der der Funktionsblock FCN_EMERG_STOP den Nothalt auslöst.



(a) Funktionskette für den Nothalt in der Fahrt nach vorne und im Zustand S_EMERG_ON.

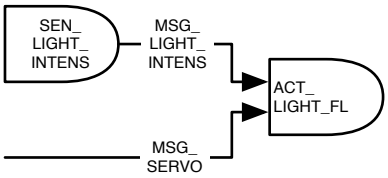


(b) Funktionskette für den Nothalt in der Fahrt nach hinten und im Zustand S_EMERG_ON.

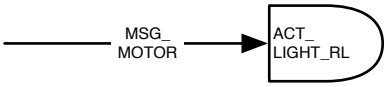
Abbildung 5.9: Funktionsketten für den Nothalt.

Die drei Abstandssensoren liefern jeweils die gemessene Entfernung zum nächstgelegenen Objekt. Die drei Messwerte werden anschließend dazu genutzt, um festzustellen, ob ein Hindernis bis zu einem bestimmten Abstand im Weg liegt. Ebenso verhält es sich mit der Hindernis-Erkennung in der Rückwärtsfahrt, deren Funktionskette in Abbildung 5.9b dargestellt ist. Dabei sind die Funktionsketten nur in der entsprechenden Fahrtrichtung und im Zustand S_EMERG_ON aktiv, der über die Nachricht MSG_EMERG_ON eingeschaltet werden kann. Dabei wird der Lenkeinschlag für die Hindernis-Erkennung nicht verwendet, um z. B. über das Berechnen der Fahrkurve zu entscheiden, ob an dem erkannten Objekt vorbeigefahren wird.

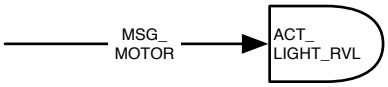
Im Zustand S_EMERG_OFF sind beide Funktionsketten nicht aktiv, da die Nothalt-Funktion ausgeschaltet ist, womit die Abstandssensoren nicht benötigt und abgeschaltet werden können.



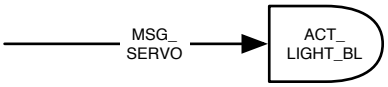
(a) Funktionskette für die Frontlichter im aktiven Zustand S_LIGHT_ON.



(b) Die stets aktive Funktionskette für die Rücklichter.



(c) Die stets aktive Funktionskette für die Rückfahrlichter.



(d) Funktionskette für die Blinker im Zustand S_BLINKER_ON.

Abbildung 5.10: Funktionsketten für die Lichter des Modell-Fahrzeugs.

Lichtsteuerung

Der logische Aktor ACT_LIGHT_FL steuert die Frontlichter des Modell-Fahrzeugs, für die es zwei Fahrzeug-Zustände S_LIGHT_ON und S_LIGHT_OFF gibt. Im aktiven Zustand S_LIGHT_ON wird die Funktionskette in Abbildung 5.10a für die Frontlichter ausgeführt. Um die Helligkeit der Frontlichter zu steuern, wird der Helligkeitssensor SEN_LIGHT_INTENS benötigt, dessen Messwerte direkt vom Aktor ACT_LIGHT_FL verarbeitet werden. Im inaktiven Zustand S_LIGHT_OFF sind die Frontlichter ausgeschaltet und somit wird keine Funktionskette ausgeführt.

Die Funktionskette zur Steuerung des logischen Aktors ACT_LIGHT_RL für die Rücklichter, die ebenfalls als Bremslichter dienen, ist immer aktiv. Die dazugehörige Funktionskette ist in Abbildung 5.10b zu sehen. Dabei wird die Nachricht MSG_MOTOR dazu genutzt, um einen Bremsvorgang zu erkennen. Auf dieselbe Weise verhält es sich mit dem logischen Aktor ACT_LIGHT_RVL für die Rückfahrlichter, die automatisch eingeschaltet werden, wenn das Modell-Fahrzeug rückwärts fährt. Somit wird für die Rücklichter nur die Nachricht MSG_MOTOR benötigt, um die Rückfahrlichter zu steuern, wie es in Abbildung 5.10c dargestellt ist.

Die Blinker lassen sich unabhängig von den übrigen Lichter ein- bzw. ausschalten, wofür die Fahrzeug-Zustände S_BLINKER_ON und S_BLINKER_OFF eingeführt werden. Die Funktionskette für die Blinker im Zustand S_BLINKER_ON ist in Abbildung 5.10d dargestellt. Im Fahrzeug-Zustand S_BLINKER_OFF sind die Blinker deaktiviert, wofür auch keine weiteren Sensoren oder Funktionsblöcke benötigt werden. Bei den Blinkern ist zu beachten, dass sowohl die Blinker links als auch die Blinker rechts von der einen Aktor-Komponente ACT_LIGHT_BL gesteuert werden. Welche Blinker gerade aktiv sind, wird als Parameter in der Nachricht MSG_BLINKER_ON mitgeteilt. Die Nachricht MSG_SERVO dient dazu, den Blinker automatisch zu deaktivieren, sobald das Fahrzeug wieder geradeaus fährt.

Batterie-Überwachung

Die Funktionskette für die Batterie-Überwachung ist in Abbildung 5.11 dargestellt. Diese ist zu jedem Zeitpunkt aktiv und überwacht die Batterie-Spannung. Sobald diese unter eine Grenze fällt, wird die Nachricht MSG_BATT_LOW gesendet.

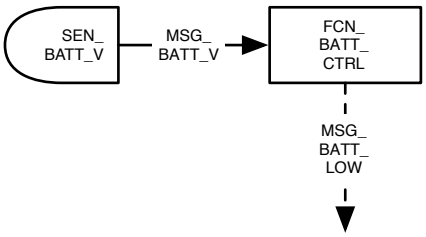


Abbildung 5.11: Funktionskette für die Batterie-Überwachung.

5.3 Software-Architektur der PLASA-Plattform

Nachdem die logische Architektur der PLASA-Plattform, in der die Anwendungsfälle mit Hilfe von logischen Komponenten beschrieben werden, vorgestellt wurde, erfolgt als nächster Schritt die Abbildung der logischen Komponenten auf Software-Komponenten, die auf den ECUs der PLASA-Plattform ausgeführt werden. In diesem Kapitel liegt der Fokus vor allem auf den TinyOS-Software-Komponenten, die auf dem PSB, DAB und LSB laufen. TinyOS wurde als Betriebssystem für diese ECUs aufgrund der beschränkten Hardware-Ressourcen ausgewählt und weil zwei der TinyOS-Entwicklungsziele ebenfalls auf die PLASA-Plattform zutreffen – erstens das erhöhte Kommunikationsaufkommen und zweitens die Notwendigkeit für ein effizientes DPM. Trotzdem sind einige Erweiterungen in TinyOS notwendig, um automotive Anwendungen darauf auszuführen. Dennoch bietet TinyOS eine Grundlage, auf die die PLASA-Plattform gut aufsetzen kann.

Die Abbildung 5.12 gibt eine Übersicht über die Erweiterungen und zusätzlichen TinyOS-Komponenten, die für die Realisierung der Anwendungsfälle für die Fortbewegung des Modell-Fahrzeugs notwendig sind. Die Komponenten aus der logischen Architektur werden in den sogenannten Task-Komponenten[1] gekapselt, die in der Applikationsschicht enthalten sind.

Für die Logik in den Task-Komponenten sowie für das Auslesen der Sensoren und die Ansteuerung der Aktoren benötigen die Task-Komponenten weitere TinyOS-Komponenten, die in den tiefer gelegenen Schichten realisiert sind. Aufbauend auf den HIL-Schnittstellen, die im Abschnitt 3.3.4 vorgestellt wurden, sind in den Schichten *COM*, *Motion* und *DPM* die Komponenten für die Kommunikation zwischen den ECUs der PLASA-Plattform, für die Fortbewegung des Modell-Fahrzeugs bzw. für das DPM der PLASA-Plattform enthalten.

[1] Diese sind nicht mit den TinyOS-Tasks für den synchronen Ablauf und dem Ausführungsmodell zu verwechseln.

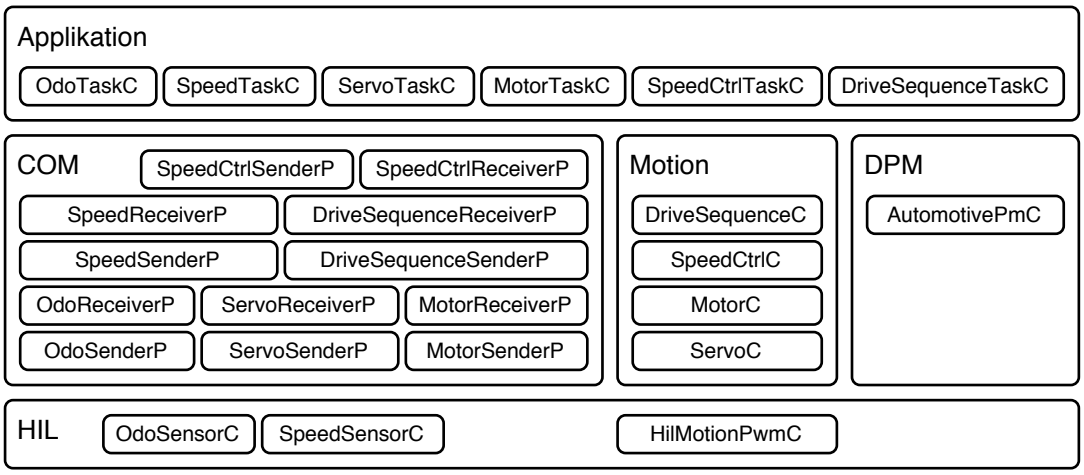


Abbildung 5.12: Die Software-Komponenten der PLASA-Plattform in TinyOS.

Die Schicht *COM* besteht aus den sogenannten Sender- und Receiver-Komponenten. Diese sorgen dafür, dass die spezifischen Nachrichten der PLASA-Plattform erzeugt und verschickt bzw. aus den empfangenen Nachrichten die entsprechenden Schnittstellen aufgerufen werden.

In der Schicht *Motion* ist die Fortbewegung des Modell-Fahrzeugs realisiert. Darunter fällt die Komponente `MotorC`, die die Logik des Fahrten-Reglers abstrahiert und eine Schnittstelle für die Ansteuerung des Motors bereitstellt. Die Komponente `ServoC` ist für die Steuerung des Lenk-Servos zuständig. Für die Geschwindigkeitsregelung und die Fahrsequenz-Regelung sind ebenso die Komponenten `SpeedCtrlC` und `DriveSequenceC` in dieser Schicht enthalten.

In der Schicht *DPM* befindet sich die Komponente `AutomotivePmC`, die den automotiven PM unter TinyOS realisiert und zwischen der Applikationsschicht und der HIL-Schicht liegt. Dieser verbindet die Tasks mit den Kommunikations- und Timer-Komponenten und informiert die Software-Tasks über Ereignisse, ob diese gerade gestartet oder gestoppt sind. Der PM steuert ebenfalls die Hardware-PMCs über die HIL-Schicht auf der ECU.

Auf dem Gumstix kommt dagegen ein embedded Linux zum Einsatz, das mit Hilfe des OpenEmbedded Build-Frameworks [115] erstellt wird. Im weiteren Verlauf dieses Kapitels wird auf das DPM auf dem Gumstix nicht weiter eingegangen. A. Barthels beschreibt in seiner Dissertation, wie ein Power-Management für Automotive Systeme auf einer Linux-Plattform umgesetzt und in einer Test-Umgebung mit mehreren Gumstix eingesetzt werden kann [27].

Im weiteren Verlauf dieses Abschnitts wird zuerst dargelegt, wie die logische Architektur der PLASA-Plattform auf TinyOS umgesetzt wird. Dabei wird gezeigt, wie Sensoren, Funktionsblöcke und Aktoren aus der logischen Architektur in TinyOS als Software-Tasks realisiert werden. Anhand von ausgewählten Beispielen wird dies näher beschrieben und dargelegt, wie die Software-Tasks als PMCs an den PM angebunden werden. Im Anschluss daran wird detaillierter auf die Komponente `AutomotivePmC` ein-

gegangen, die den PM unter TinyOS realisiert. Im darauf folgenden Abschnitt werden die automotive-spezifischen Erweiterungen für Sensoren, Regelungen oder Aktoren unter TinyOS erläutert, die zum Teil bis zur HPL-Schicht beschrieben werden.

5.3.1 Abbildung der logischen Architektur

Für die logische Architektur der PLASA-Plattform wurden drei unterschiedliche Komponenten verwendet: die Sensoren, die Funktionsblöcke und die Aktoren. Der Unterschied zwischen den Komponenten ist, ob sie Daten-Quellen, datenverarbeitende Komponenten oder Daten-Senken sind. Für die Software-Architektur ist es an dieser Stelle wichtig, ein möglichst einheitliches Konzept für die drei Komponenten der logischen Architektur bereitzustellen, um die Umsetzung der logischen Architektur in der Software möglichst generisch zu realisieren.

Somit werden die Task-Komponenten unter TinyOS eingeführt, die genau eine Komponente aus der logischen Architektur kapseln. Die Schnittstelle zu den Task-Komponenten besteht aus drei Teilbereichen. Der erste und wichtigste Bereich ist die Kommunikationsschnittstelle, über die ein Task Nachrichten von anderen Tasks empfangen bzw. Nachrichten an andere Tasks versenden kann. Der zweite Bereich ist die Schnittstelle zu den Timern, mit denen ein Task periodisch aufgerufen werden kann. In dem dritten Bereich sind die lokalen Benachrichtigung über Task-Ereignisse enthalten, die ein Task von dem PM der ECU erhält, worunter das Starten oder das Beenden eines Tasks fällt. Als nächstes werden die Schnittstellen aus den drei unterschiedlichen Bereichen näher beschrieben, wobei mit der Kommunikationsschnittstelle begonnen wird. Im Anschluss wird die Schnittstelle der Timer erläutert, bevor mit den Task-Ereignissen abgeschlossen wird. Am Ende des Abschnitts werden Beispiele für die Task-Komponenten vorgestellt, um das Konzept der Abbildung von der logischen Architektur auf die Software-Architektur zu verdeutlichen.

Kommunikationsschnittstellen

Bevor die Schnittstelle für das Senden und Empfangen von Nachrichten vorgestellt wird, wird ein Datentyp eingeführt, der die Nutz-Daten einer jeden Nachricht aus der logischen Architektur aufnehmen kann. Dafür wird eine Union definiert, die alle existierenden Nutz-Daten der Nachrichten aus der PLASA-Plattform enthalten kann. Diese Union ist notwendig, da sie als Argument für alle Kommunikationsschnittstellen verwendet wird. In Quelltext 5.1 ist der Datentyp `plasa_msg_t` für die Repräsentation einer Nachricht dargestellt. Dies entspricht dem Vorgehen für die AM-Pakete aus TinyOS, das in TEP 111 beschrieben ist [96] und bereits in Abschnitt 3.3.4 im Detail vorgestellt wurde.

Die logischen Nachrichten der PLASA-Plattform sind Broadcasts, die, sobald diese versendet wurden, allen empfangenden Komponenten zur Verfügung stehen. Aus diesem Grund wird in der PLASA-Plattform für alle Nachrichten ein zentraler Puffer auf jeder ECU angelegt, der von jeder Nachricht die zuletzt gesendete speichert. Um die letzte Nachricht eines Typs zu lesen, nutzt ein Task die Schnittstelle `Get`. Mit Hilfe der Schnittstelle `Set` kann ein Task eine Nachricht in den zentralen Puffer schreiben, was

```
1  typedef union
2  {
3    plasa_msg_speed_t msg_speed;
4    plasa_msg_odo_t msg_odo;
5    plasa_msg_servo_t msg_servo;
6    plasa_msg_motor_t msg_motor;
7    plasa_msg_speed_ctrl_t msg_speed_ctrl;
8    plasa_msg_add_seg_t msg_add_seg;
9    plasa_msg_drive_status_t msg_drive_status;
10   plasa_msg_batt_v_t msg_batt_v;
11   plasa_msg_sensor_value_t msg_sensor_value;
12   plasa_msg_parking_t msg_parking;
13   plasa_msg_blinker_on_t msg_blinker_on;
14   plasa_msg_lights_on_t msg_light_on;
15 } plasa_msg_t;
```

Quelltext 5.1: Die Datenstruktur `plasa_msg_t` für die Aufnahme aller Nutz-Daten aus den Nachrichten der PLASA-Plattform.

einem Senden der Nachricht entspricht. Beide Schnittstellen sind in TEP 114 definiert und bereits in Abschnitt 3.3.4 vorgestellt worden. Als Datentyp für den Parameter wird die Union `plasa_msg_t` verwendet, die die Nutz-Daten einer jeden Nachricht aufnehmen kann.

Sobald eine Nachricht über die Schnittstelle `Set` in den zentralen Puffer geschrieben wurde, kümmert sich dieser um die Benachrichtigung der anderen Tasks, die auf die Nachricht warten. Die Tasks werden dann über die Schnittstelle `Notify` über den Empfang einer Nachricht von einem bestimmten Typ informiert. Dies kann der Task als Ereignis nutzen, um je nach seiner Programm-Logik Berechnungen durchzuführen oder Aktoren zu steuern. Auf den zentralen Puffer und dessen Funktionsweise sowie die Verlinkung zwischen dem zentralen Puffer und den Tasks wird im nächsten Abschnitt detaillierter eingegangen.

Die Komponenten in der logischen Architektur empfangen und versenden Nachrichten. Dabei können Sensoren aufgrund ihrer Eigenschaften als Daten-Quellen keine Nachrichten empfangen und Aktoren als Daten-Senken keine Nachrichten senden. Dies entspricht auf der Ebene der Software-Komponenten, dass ein Task im Falle eines Sensors keine Nachrichten über die Schnittstelle `Get` liest. Im Falle eines Aktors werden keine Nachrichten mit Hilfe der Schnittstelle `Set` geschrieben.

Timer-Schnittstellen

Falls Tasks periodisch aufgerufen werden sollen, müssen sie über den Ablauf eines periodischen Timers benachrichtigt werden. Dies geschieht ebenfalls wie bei dem Empfang von Nachrichten über die Schnittstelle `Notify`. Der Grund, weshalb hierfür nicht die Schnittstelle `Timer` verwendet wird, ist, dass der Task nicht selbstständig die Periode festlegt, sondern von einer zentralen Timer-Komponente, in der das Wissen über die Funktionsketten hinterlegt ist. Denn die Timer-Periode kann vom Fahrzeug-Zustand abhängig sein, je nachdem welcher Aktor und dessen Funktionskette gerade aktiv ge-

schaltet ist. Diese zentrale Timer-Komponente auf jeder ECU kümmert sich um die Timer und deren zustandsabhängiger Konfiguration. Auf diese zentrale Timer-Komponente wird wie auf den zentralen Puffer für die Nachrichten im nächsten Abschnitt eingegangen.

Vor allem für Sensoren sind Timer als Quelle essenziell, da sie als erstes Glied einer Funktionskette keine Nachrichten empfangen und somit nur über den Ablauf eines Timers benachrichtigt werden. Sobald sie die Benachrichtigung über den Ablauf des Timers erhalten haben, erfassen die Sensoren ihre Messwerte und generieren daraus eine Nachricht, die sie im System verteilen. Dies führt im Anschluss dazu, dass die Funktionsblöcke diese Nachrichten empfangen und weiterverarbeiten und wiederum Nachrichten versenden. Am Ende der Kette stehen die Aktoren, die die Nachrichten konsumieren und Einfluss auf den physikalischen Prozess nehmen.

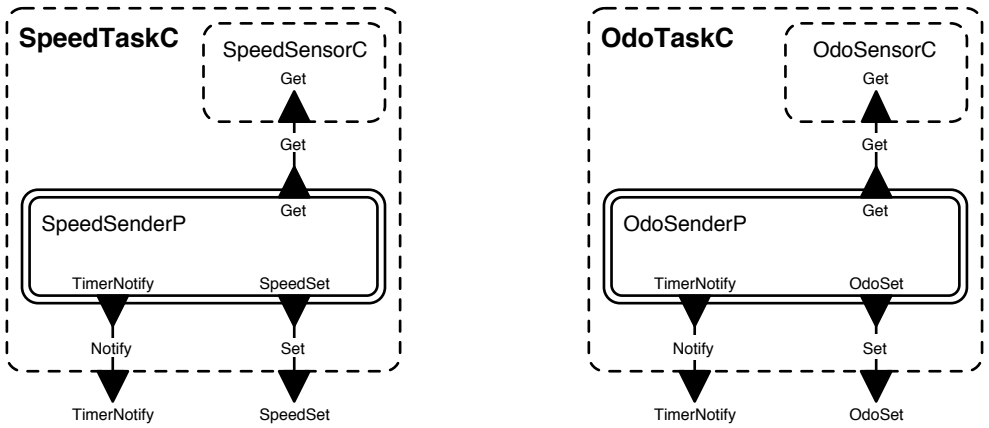
Schnittstelle für die Task-Ereignisse

Jede Komponente aus der logischen Architektur ist mindestens einer Funktionskette zugeordnet. Da jede Kette eine Aktiv-Bedingung hat, sind auch die Komponenten entweder aktiv oder werden nicht benötigt und können somit ausgeschaltet werden. Somit müssen auch die Task-Komponenten in der Software-Architektur informiert werden, wenn sie gestartet oder gestoppt werden. Der Grund dafür ist, dass Tasks interne Zustände haben können, die ebenfalls bei einem Starten oder Beenden eines Tasks verändert werden müssen. Um den Tasks das Starten oder das Beenden mitzuteilen, wird die Schnittstelle `TaskEvent` verwendet, die in Quelltext 5.2 abgebildet ist. Für die Aktivierung eines Tasks erhält diese das Ereignis `started`. Falls er beendet wurde, wird dies dem Task über das Ereignis `stopped` mitgeteilt. Dies erlaubt es dem Task, sich bei einer Aktivierung neu zu initialisieren oder bei einer Deaktivierung noch Aufräum-Arbeiten zu erledigen. So ist dies beispielsweise bei der Geschwindigkeitsregelung der Fall, dass bei Aktivierung der Regelung die internen Zustände des Reglers an die momentane Fahrsituation anpassen kann. Dies ist notwendig, da bei Fehlen dieser Benachrichtigungen der Task nicht unterscheiden kann, ob er zwischen den eingehenden Nachrichten zwischenzeitlich ausgeschaltet war.

```
1 interface TaskEvent
2 {
3   event void started();
4   event void stopped();
5 }
```

Quelltext 5.2: Die Schnittstelle `TaskEvent` für das Benachrichtigen der Tasks.

Alternativ könnte der Task auch auf alle anderen Zustandsnachrichten hören, die ihn deaktivieren können. Allerdings muss dann der Task die logische Architektur kennen. Ebenfalls kommt es dann zu Problemen, wenn die logische Architektur um weitere Komponenten erweitert wird. Dadurch müssen alle Tasks erweitert werden, damit sie die Nachrichten über die Deaktivierung entsprechend verarbeiten können. Um die Tasks möglichst wiederverwendbar und erweiterbar zu gestalten, ist die Benachrichtigung über



(a) Der Sensor SEN_SPEED als Task in der Konfiguration `SpeedTaskC`.

(b) Der Sensor SEN_ODO als Task in der Konfiguration `OdoTaskC`.

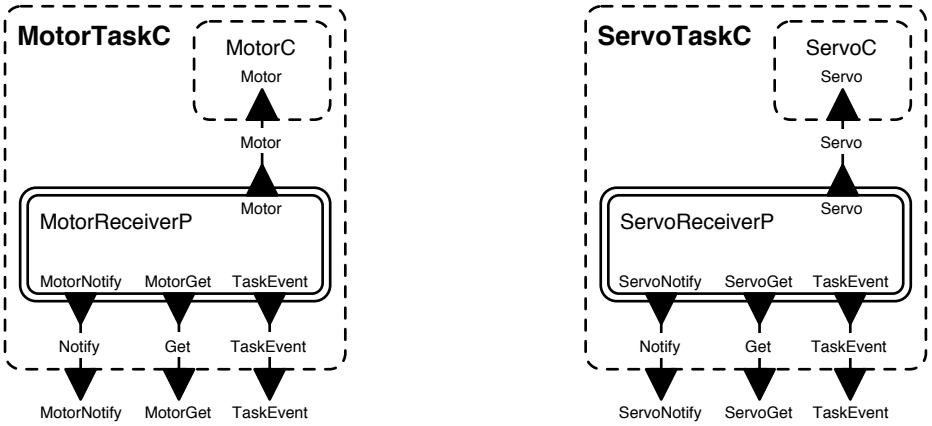
Abbildung 5.13: Die Umsetzung der Sensoren SEN_SPEED und SEN_ODO.

die Schnittstelle `TaskEvent` notwendig. Auf diese Weise wird die Durchsetzung der Fahrzeug-Zustände einer zentralen Komponente überlassen, das über die Schnittstelle `TaskEvent` die einzelnen Tasks über deren Aktivierung bzw. Deaktivierung informiert. Dies entspricht ebenfalls dem Prinzip der Komponenten-basierten Betriebssysteme, dass die Komponenten so wenig wie möglich Kontext-Informationen enthalten sollen. Somit wird über die Schnittstelle `TaskEvent` erreicht, dass eine zentrale Komponente, die die logische Architektur kennt, über eine generische Schnittstelle die Task-Komponenten informiert und steuert.

Beispiele für Task-Komponenten in der PLASA-Plattform

Nachdem die Schnittstellen zwischen den zentralen Komponenten und den Tasks beschrieben wurden, werden im Anschluss verschiedene Tasks aus dem Bereich der Sensoren, der Aktoren und der Funktionsblöcke, die in der PLASA-Plattform Anwendung finden, beispielhaft vorgestellt. Um dieses Vorgehen zu verdeutlichen, werden im Folgenden zwei Sensor-Komponenten aus der logischen Architektur in TinyOS-Komponenten mit der zuvor beschriebenen Methodik abgebildet. Als Beispiel werden die logischen Sensoren SEN_SPEED und SEN_ODO aus dem Abschnitt 5.2.1 herangezogen. Für die weiteren Sensoren wie z. B. für die Abstandsmessung wird dieses Konzept entsprechend angewendet.

Die Sensoren werden über die Schnittstelle `Notify` benachrichtigt, wenn diese die Messung durchführen sollen und den gemessenen Wert über den Bus versenden. In der Abbildung 5.13a ist die Konfiguration `SpeedTaskC` für den Geschwindigkeitssensor SEN_SPEED aus der logischen Architektur dargestellt. Diese Konfiguration enthält die zwei Komponenten `SpeedSenderP` und `SpeedSensorC`. Die generische Konfiguration `SpeedSenderP` wandelt den Sensor-Wert, den sie über die Schnittstelle `Get` von der Konfiguration `SpeedSensorC` erhält, in eine MSG_SPEED Nachricht um und versendet diese mit Hilfe der Schnittstelle `Set`. Die Realisierung der Konfiguration `SpeedSensorC`



(a) Der Aktor ACT_MOTOR als Task in der Konfiguration `MotorTaskC`.

(b) Der Aktor ACT_SERVO als Task in der Konfiguration `ServoTaskC`.

Abbildung 5.14: Umsetzung der Aktoren ACT_MOTOR und ACT_SERVO.

wird im Abschnitt 5.3.3 im Detail vorgestellt.

Als zweites Beispiel ist der Sensor SEN_ODO in Abbildung 5.13b dargestellt. Die Sensoren unterscheiden sich konzeptionell nicht im Aufbau. Unterschiedlich ist nur, dass die Komponente `OdoSenderP` eine MSG_ODO Nachricht aus dem Wert, den sie von der Konfiguration `OdoSensorC` erhält, erstellt. Wiederum existiert die Schnittstelle `Notify`, das den Sensor-Task zum Messen des Werts veranlasst und anschließend den Wert über die Schnittstelle `Set` versendet.

Da Sensoren in der logischen Architektur Daten-Quellen sind, implementieren die dazugehörigen Task-Komponenten niemals die Schnittstelle `Get`. Anders verhält es sich mit den Aktoren aus der logischen Architektur. Diese Task-Komponenten benutzen niemals die Schnittstelle `Set`, sondern nur die Schnittstelle `Get`. Für die Aktor-Komponente ACT_MOTOR wird die Konfiguration `MotorTaskC` eingeführt, die in Abbildung 5.14a dargestellt ist. Diese Konfiguration enthält nur zwei Komponenten: `MotorReceiverP` und `MotorC`. Die generische Konfiguration `MotorReceiverP` sorgt für die Interpretation der Nachricht MSG_MOTOR und wandelt diese in die Befehle der Schnittstelle `Motor` um. In diesem Task werden die Befehle an die Komponente `MotorC` weitergeleitet, die später in Abschnitt 5.3.3 über die automotiven Software-Anteile in der PLASA-Plattform näher beschrieben wird.

Für die Steuerung des Lenk-Servos wird der logische Aktor ACT_SERVO auf der Komponente `ServoTaskC` abgebildet, die in Abbildung 5.14b dargestellt ist. Wiederum existiert eine generische Konfiguration für die Umwandlung der Nachricht MSG_SERVO in die Befehle der Schnittstelle `Servo`. Die Steuerung des Servos übernimmt im Anschluss die Komponente `ServoC`, die ebenfalls in Abschnitt 5.3.3 detaillierter beschrieben wird.

Die dazwischen liegenden Funktionsblöcke verwenden sowohl die Schnittstelle `Notify` und `Get` als auch die Schnittstelle `Set`, um sowohl Nachrichten zu empfangen als auch Nachrichten zu versenden. Als Beispiel ist der Funktionsblock FCN_SPEED_CTRL als TinyOS-Komponente `SpeedCtrlTaskC` in Abbildung 5.15 dargestellt. Die Konfigurati-

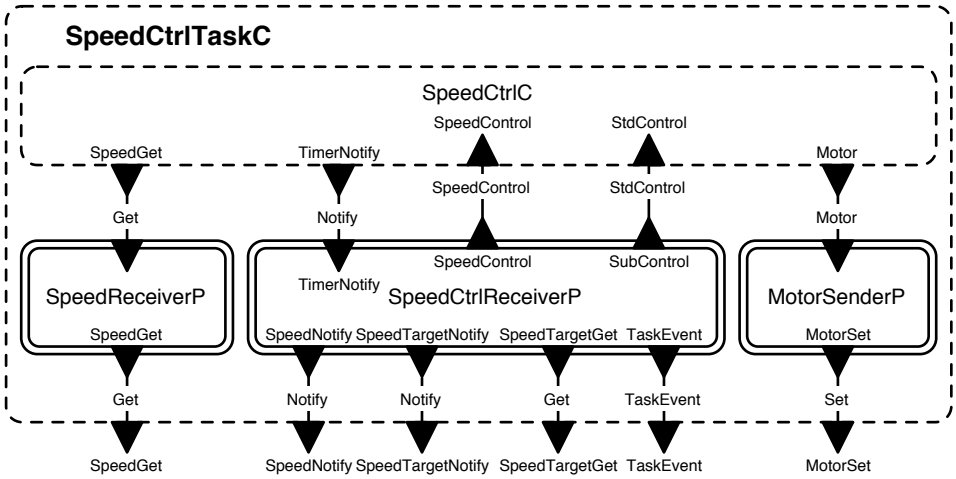


Abbildung 5.15: Umsetzung des Funktionsblocks FCN_SPEED_CTRL als Konfiguration `SpeedCtrlTaskC`.

on `SpeedCtrlTaskC` enthält die Komponente `SpeedCtrlReceiverP`, die die Nachrichten für den Geschwindigkeitsregler empfängt und über die Schnittstelle `SpeedControl` an die Komponente `SpeedCtrlC` weiterleitet. Die Komponente `SpeedCtrlC` implementiert den Geschwindigkeitsregler und wird in Abschnitt 5.3.3 über die Software-Architektur der automotiven Software-Anteile in der PLASA-Plattform detailliert vorgestellt. Neben den Befehlen aus der Schnittstelle `SpeedControl` benötigt der Geschwindigkeitsregler auch die momentane Geschwindigkeit des Modell-Fahrzeugs, die er über die Nachricht MSG_SPEED erhält, die zuvor durch die Komponente `SpeedReceiverP` interpretiert wird. Für das Versenden der MSG_MOTOR Nachricht implementiert die Konfiguration `SpeedCtrlTaskC` die Komponente `MotorSenderP`, die aus den Befehlen der Schnittstelle `Motor` Nachrichten vom Typ MSG_MOTOR erzeugt und anschließend über die Schnittstelle `MotorSet` versendet.

Als ein weiteres Beispiel für die Abbildung der Funktionsblöcke wird die logische Komponente FCN_DRIVE_SEQUENCE herangezogen, die in der TinyOS-Komponente `DriveSequenceTaskC` realisiert wird. In Abbildung 5.16 ist deren Aufbau, der dem der Konfiguration `SpeedCtrlTaskC` sehr ähnlich ist, dargestellt. Um die Nachrichten MSG_ODO und MSG_SPEED auszupacken und die enthaltenen Sensor-Werte über die Schnittstelle `Get` bereitzustellen, enthält die Konfiguration `DriveSequenceTaskC` die Subkomponenten `OdoReceiverP` und `SpeedReceiverP`. Für das Hinzufügen eines Segments in die Fahrsequenz-Regelung wird die Nachricht MSG_ADD_SEG von der Komponente `DriveSequenceReceiverP` empfangen, die im Anschluss die zentrale Komponente `DriveSequenceC` über die Schnittstelle `DriveSequence` steuert. Diese realisiert die Fahrsequenz-Regelung, die mit Hilfe der Sensor-Werte und des Regel-Timers die Steuer-Befehle für die Geschwindigkeitsregelung und des Lenk-Servos generiert. Über die Schnittstellen `Servo` und `SpeedCtrl` werden die Steuer-Befehle an die generischen Komponenten `ServoSenderP` bzw. `SpeedSenderP` weitergeleitet, die wiederum die Nachrichten MSG_SERVO bzw. MSG_SPEED_CTRL generieren und über die Schnittstellen

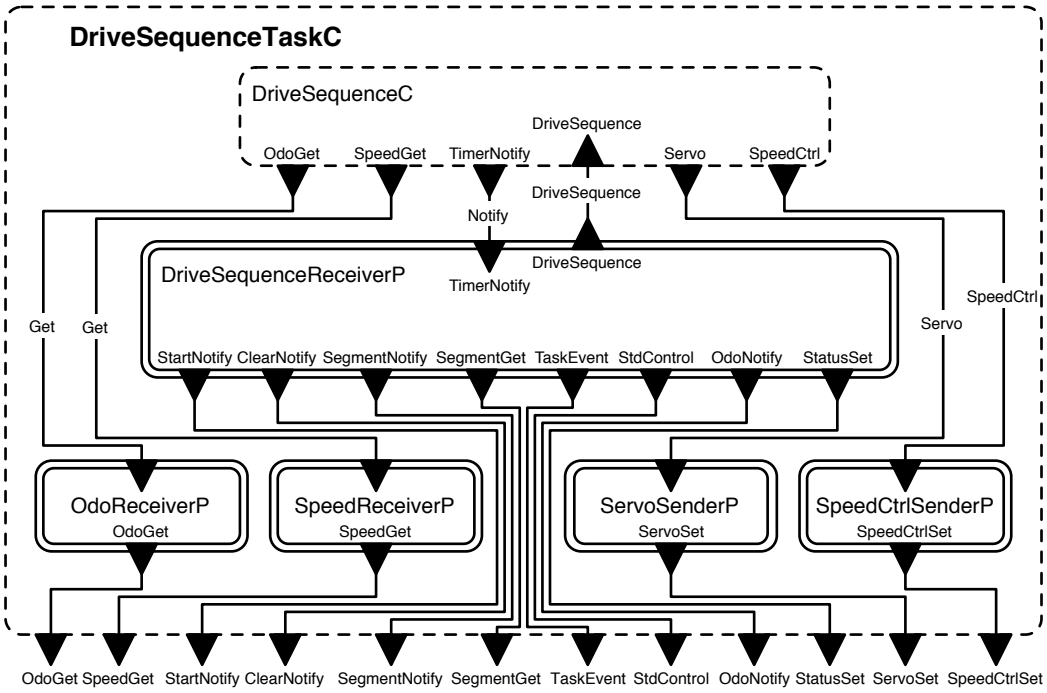


Abbildung 5.16: Umsetzung des Funktionsblocks FCN_DRIVE_SEQUENCE als Konfiguration `DriveSequenceTaskC`.

`ServoSet` bzw. `SpeedCtrlSet` im System verteilen.

5.3.2 Umsetzung des PMs in der PLASA-Plattform

In Kapitel 4 wurden die Anforderungen sowie die hierarchische Architektur eines DPMs für Automotive Systeme sowie der Aufbau eines PMs auf ECU-Ebene mit dessen verschiedenen Komponenten vorgestellt. Wie der PM in der PLASA-Plattform unter TinyOS realisiert ist, wird nun im Laufe dieses Abschnitts erläutert. Dazu gehört auch die Anbindung der PMCs an den PM auf ECU-Ebene, der in einem DPM für Automotive Systeme neben den Hardware-PMCs auch die Software-PMCs steuert. Die Software-PMCs entsprechen den Software-Tasks, die im vorhergehenden Abschnitt vorgestellt wurden. Im weiteren Verlauf dieses Abschnitts wird zuerst erläutert, wie die Software- und Hardware-PMCs an den PM in TinyOS angebunden sind, bevor detaillierter auf den PM eingegangen wird. Dabei werden die in TinyOS vorhandenen Power-Management Mechanismen wiederverwendet und nur bezüglich des automotiven Anteils erweitert. Somit kommen die in Abschnitt 3.3.4 vorgestellten Power-Management Schnittstellen von TinyOS wieder zum Einsatz.

Anbindung der Software-Tasks an den PM

Das DPM für Automotive Systeme ist dafür verantwortlich, dass nur die Software-Tasks ausgeführt werden, die in dem aktuellen Power-Modus der ECU aktiv sind. Dies wird

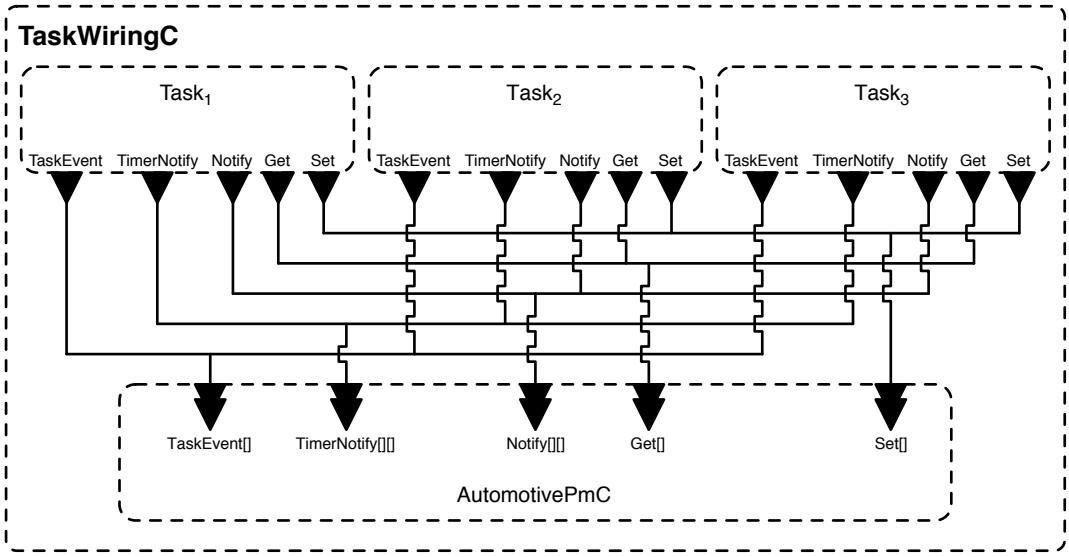


Abbildung 5.17: Anbindung der Software-Tasks an die Komponente `AutomotivePmC` in der Konfiguration `TaskWiringC`.

in der TinyOS-Implementierung dadurch erreicht, dass Bus-Nachrichten und Timer-Benachrichtigungen nur an die im aktuellen Power-Modus aktiven Software-Tasks über die Schnittstelle `Notify` weitergeleitet werden. Nicht aktive Tasks erhalten die Benachrichtigungen nicht, womit sie ihre Funktion auch nicht ausführen können [109]. Somit werden alle Software-Tasks über die Konfiguration `TaskWiringC` mit der Komponente `AutomotivePmC` verbunden, die den PM auf ECU-Ebene für Automotive Systeme realisiert. Dabei verbindet diese Konfiguration alle Software-Tasks auf einer ECU mit der Komponente `AutomotivePmC`, was in der Abbildung 5.17 dargestellt ist.

Im Folgenden wird beschrieben, wie die Software-Tasks mit den bereits vorgestellten Schnittstellen an den PM in TinyOS angebunden werden, so dass abhängig von dem aktuellen Power-Modus der ECU nur die aktiven Software-Tasks über Bus-Nachrichten oder Timer-Signale benachrichtigt werden. In der Konfiguration `TaskWiringC` werden die Schnittstellen `Notify` der Software-Tasks an die zweifach parametrisierten Schnittstellen `Notify` der Komponente `AutomotivePmC` verbunden. Damit wird angegeben, in welchem lokalen Power-Modus der ECU ein Task welche Nachricht empfängt. Falls eine Nachricht msg_1 im Power-Modus pm_1 empfangen wird, werden alle Tasks, die mit der Schnittstelle `Notify[`pm_1`][`msg_1`]` verbunden sind, mit dem Ereignis `notify` benachrichtigt.

Wie bei der Benachrichtigung der aktiven Software-Tasks über Bus-Nachrichten werden sie auch über den gleichen Mechanismus über Timer-Ereignisse informiert. Da die Software-Tasks in den unterschiedlichen Power-Modi der ECU unterschiedliche Perioden haben können, muss hier ebenfalls zwischen den Power-Modi unterschieden werden, was wiederum durch eine zwei dimensionale, parametrisierte Schnittstelle zu den Tasks realisiert wird. Der erste Parameter ist wie bei den Bus-Nachrichten der Power-Modus der ECU, wobei der zweite Parameter der gewünschte Timer ist. Falls ein Software-Task

im Power-Modus pm_2 von dem Timer $timer_1$ periodisch benachrichtigt werden soll, so wird dessen Timer-Schnittstelle mit der Schnittstelle `TimerNotify[`pm_2`][`$timer_1$`]` der Komponente `AutomotivePmC` verbunden.

Für das Lesen der letzten Bus-Nachricht eines Typs und das Senden einer Bus-Nachricht werden die Software-Tasks jeweils über die Schnittstellen `Get` bzw. `Set` an die einfach parametrisierten Schnittstellen `Get` bzw. `Set` der Komponente `AutomotivePmC` verbunden. Hierfür wird nicht die Unterscheidung des aktuellen Power-Modus der ECU benötigt, da das Lesen und das Senden einer Bus-Nachricht zustandsunabhängig erfolgt. Hierbei wird über den Parameter in den Schnittstellen `Get` und `Set` der Komponente `AutomotivePmC` der Typ der Bus-Nachricht festgelegt. Somit ist für die Verbindungen in der Konfiguration `TaskWiringC` wichtig, dass die Software-Tasks mit dem richtigen Index an die parametrisierten Schnittstellen der Komponente `AutomotivePmC` angeschlossen sind, damit diese auch die richtigen Bus-Nachrichten lesen und senden. Der Vorteil, die parametrisierten Schnittstellen zu nutzen, liegt darin, dass die Schnittstellen der Komponente `AutomotivePmC` generisch für alle ECUs sind und die Verbindung der Software-Tasks mit dem PM nur in der ECU-spezifischen Konfiguration `TaskWiringC` erfolgt.

Die Task-Ereignisse werden über die Schnittstelle `TaskEvent` an die Software-Tasks versendet. Hier besitzt die Komponente `AutomotivePmC` wiederum eine einfach parametrisierte Schnittstelle `TaskEvent`, die als Parameter den Software-Task enthält. Somit besitzt jeder Software-Task genau einen Index in der Schnittstelle, über den er seine Ereignisse erhält.

Zur Veranschaulichung, wie die Software-Tasks mit der Komponente `AutomotivePmC` verbunden werden, wird der Software-Task `MotorTaskC` herangezogen, der im vorherigen Abschnitt bereits eingeführt wurde. Ein Ausschnitt aus der Konfiguration `TaskWiringC` ist in Quelltext 5.3 dargestellt, die die Komponente `MotorTaskC` mit der Konfiguration `AutomotivePmC` verbindet. Aus der logischen Architektur der PLASA-Plattform ist ersichtlich, dass der Aktor ACT_MOTOR in den Fahrzeug-Zuständen S_MANUAL, S_EMERG_STOP, S_SPEED_CTRL, S_PARKING_SPACE und S_PARKING aktiv ist und nur die Nachricht MSG_MOTOR benötigt. Deshalb ist der entsprechende Software-Task `MotorTaskC`, der den logischen Aktor ACT_MOTOR realisiert, über die parametrisierten Schnittstelle `Get` der Komponente `AutomotivePmC` mit dem Index MSG_MOTOR verbunden. Die Schnittstelle `MotorNotify` des Software-Tasks `MotorTaskC` hängt an der zweifach parametrisierten Schnittstelle `Notify`, wobei als erster Index jeweils die fünf verschiedenen Power-Modi und als zweiter Index immer der für die Nachricht MSG_MOTOR verwendet wird. Hervorzuheben ist an dieser Stelle, dass der erste Parameter der Power-Modus der ECU ist und nicht der Fahrzeug-Zustand aus der logischen Architektur. Jedoch findet hier eine 1-zu-1 Abbildung des Fahrzeug-Zustands für den Motor auf den Power-Modus der ECU statt, so dass aus dem Fahrzeug-Zustand S_MANUAL der Power-Modus PMS_MANUAL wird. Entsprechend werden die weiteren Fahrzeug-Zustände auf die Power-Modi abgebildet. Zusätzlich besteht die Verbindung zwischen der Schnittstelle `TaskEvent` mit dem Index TASK_MOTOR auf der Seite der Komponente `AutomotivePmC` und der Schnittstelle `TaskEvent` auf der Seite der Komponente `MotorTaskC`, um die Komponente `MotorTaskC` über die Task-Ereignisse zu benachrichtigen.

```
1  configuration TaskWiringC {}
2  implementation
3  {
4
5  components AutomotivePmC;
6
7  // ...
8
9  components MotorTaskC;
10 AutomotivePmC.Notify[PMS_MANUAL, MSG_MOTOR] <- MotorTaskC.MotorNotify;
11 AutomotivePmC.Notify[PMS_EMERG_STOP, MSG_MOTOR] <- MotorTaskC.MotorNotify;
12 AutomotivePmC.Notify[PMS_SPEED_CTRL, MSG_MOTOR] <- MotorTaskC.MotorNotify;
13 AutomotivePmC.Notify[PMS_PARKING_SPACE, MSG_MOTOR] <- MotorTaskC.
       MotorNotify;
14 AutomotivePmC.Notify[PMS_PARKING, MSG_MOTOR] <- MotorTaskC.MotorNotify;
15
16 AutomotivePmC.Get[MSG_MOTOR] <- MotorTaskC.MotorGet;
17
18 AutomotivePmC.TaskEvent[TASK_MOTOR] <- MotorTaskC.TaskEvent;
19
20 // ...
21
22 }
```

Quelltext 5.3: Anbindung des `MotorTaskC` an die Komponente `AutomotivePmC` in der Konfiguration `TaskWiringC`.

Auf dieselbe Art und Weise werden die übrigen Software-Tasks mit der Komponente `AutomotivePmC` verbunden. Vor allem die Verbindungen für die Schnittstelle `Notify` ist sehr aufwendig und fehleranfällig, womit sich eine automatische Generierung der Konfiguration `TaskWiringC` anbietet, die aus den Informationen der logischen Architektur die Verbindungen in TinyOS generiert. In der PLASA-Plattform ist somit ein Werkzeug entstanden, dass durch Annotationen im nesC Quelltext, die die Informationen aus der logischen Architektur der PLASA-Plattform enthalten, die nötigen Verbindungen liest und daraus automatisiert die Konfiguration `TaskWiringC` generiert [104].

Falls ein neuer Task in die Konfiguration einer ECU aufgenommen werden soll, so muss dieser als neue Teilkomponente in der Konfiguration `TaskWiringC` eingetragen werden. Zusätzlich sind die Verbindungen zu der Komponente `AutomotivePmC` festzulegen, wobei wiederum die logische Architektur betrachtet wird. Der Vorteil dieser Methode beim Hinzufügen eines Tasks ist, dass die vorhandenen Tasks nicht verändert werden müssen. Ebenfalls führt das Entfernen eines Tasks durch das Löschen der Task-Komponente mit seinen Verknüpfungen aus der Konfiguration `TaskWiringC` zu keiner Veränderung der übrigen Software-Tasks.

Allerdings ist über die vorgestellte Methode keine Prüfung der logischen Architektur vorhanden. Denn es wird nicht mit Hilfe der Standard-TinyOS-Werkzeuge überprüft, ob alle Tasks einer Funktionskette in der Konfiguration enthalten sind. Ebenfalls wird nicht untersucht, ob auch alle Tasks in dem jeweiligen Power-Modus die Nachrichten auch über die Schnittstelle `Notify` erhalten. Solche Überprüfungen müssen über ein

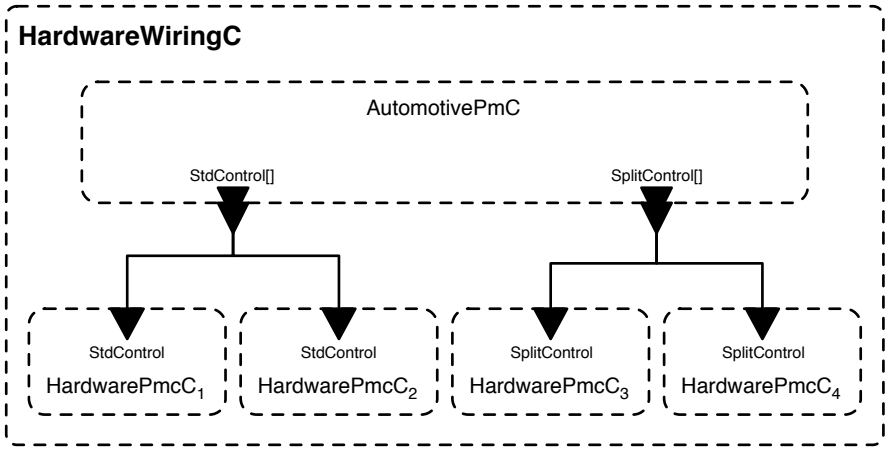


Abbildung 5.18: Anbindung der Hardware-PMCs an die Komponente `AutomotivePmC` in der Konfiguration `HardwareWiringC`.

weiteres Werkzeug realisiert werden.

Anbindung der Hardware-PMCs an den PM

Da TinyOS bereits von Grund auf für ein energieeffizientes Verhalten ausgelegt ist, eignen sich die verwendeten Mechanismen auch für ein DPM für Automotive Systeme. Die TEPs, die für das Power-Management zuständig sind, sollen somit auch in der PLASA-Plattform wiederverwendet werden. Darunter fällt das TEP 115 für nicht-virtualisierte Geräte [88] und das TEP 112 für das Power-Management für Mikrocontroller [147].

Im TEP 115 werden die Schnittstellen `StdControl` und `SplitControl` definiert, über die Geräte mit dem Befehl `start` ein- und mit dem Befehl `stop` ausgeschaltet werden können. Diese Schnittstelle wird für das explizite Power-Management genutzt und kann somit auch für das DPM für Automotive Systeme verwendet werden, um die Hardware-PMCs, wie Sensoren, Aktoren oder Teile des Mikrocontrollers zu steuern. Aus diesem Grund nutzt die Konfiguration `AutomotivePmC` die beiden Schnittstellen, um die Hardware-PMCs ein- und auszuschalten. Um möglichst für alle ECUs eine generische Schnittstelle zu schaffen, werden die Schnittstellen parametrisiert von der Komponente `AutomotivePmC` genutzt. Jedes Gerät, das von dem PM gesteuert werden soll, erhält einen bestimmten Index in der parametrisierten Schnittstelle. Die Verbindungen zwischen der Komponente `AutomotivePmC` und den Hardware-PMCs erfolgen in der Konfiguration `HardwareWiringC`, die in Abbildung 5.18 dargestellt ist. Die Komponente `AutomotivePmC` enthält das Wissen, in welchem Power-Modus der ECU welche Hardware-PMCs ein- oder ausgeschaltet sind. Bei einem Wechsel des lokalen Power-Modus wird der Zustand der PMCs zwischen dem alten und dem neuen Power-Modus verglichen. Muss ein PMC eingeschaltet werden, wird der Befehl `start` aufgerufen. Falls ein PMC ausgeschaltet werden muss, wird dies mit dem Befehl `stop` erreicht.

Wie bei den Software-Tasks sind die Schnittstellen zur Steuerung der Hardware-PMCs durch die Komponente `AutomotivePmC` generisch. Nur über die ECU-spezifischen Kon-

figuration `HardwareWiringC` werden alle auf einer ECU vorhandenen Hardware-PMCs an den generischen PM angebunden.

In TEP 112 wird das Power-Management des Mikrocontrollers für TinyOS festgelegt, das ebenfalls für das DPM der PLASA-Plattform genutzt wird. Das TinyOS Konzept basiert darauf, dass, wenn kein Task mehr ausgeführt werden muss, der Mikrocontroller in den Schlaf-Zustand überführt wird. Dabei findet allerdings vorher noch eine Überprüfung statt, in welchen der verschiedenen Power-Modi der Mikrocontroller versetzt werden kann. Wie im Abschnitt 4.3.1 beschrieben wurde, besitzen die Mikrocontroller unterschiedliche Power-Modi, die von einem *Idle* bis zu einem *Power-Down* reichen. Je mehr Komponenten des Mikrocontrollers ausgeschaltet sind, um ein desto niedrigerer Schlaf-Zustand kann eingenommen und somit mehr Energie eingespart werden. Nach TEP 112 wird vor jedem Schlafen-Legen des Prozessors überprüft, welche Teile noch aktiv sind und daraus wird der dazu passende niedrigste Power-Modus des Prozessors berechnet. Dies setzt allerdings voraus, dass alle benötigten Komponenten des Mikrocontrollers ebenfalls als PMCs an die Komponente `AutomotivePmC` verbunden sind und damit von dem DPM gesteuert werden. Somit wird erreicht, dass die Komponente `AutomotivePmC` die Hardware-PMCs des Mikrocontrollers steuert und im Anschluss das generische Konzept von TinyOS verwendet wird, um den Mikrocontroller entsprechend dem Power-Modus der ECU schlafen zu legen.

PM auf ECU-Ebene unter TinyOS

Nachdem vorgestellt wurde, wie die Software-PMCs in Form von Software-Tasks und die Hardware-PMCs an den PM in TinyOS angebunden werden, wird im Folgenden der Aufbau der Konfiguration `AutomotivePmC` und somit, wie der PM auf ECU-Ebene in TinyOS realisiert ist, beschrieben. In Abbildung 5.19 ist die Konfiguration `AutomotivePmC` dargestellt, die das DPM für die PLASA-Plattform implementiert.

Das Modul `AutomotivePmP`, das aus den Bus-Nachrichten den Power-Modus der ECU bestimmt, ist die zentrale Komponente der Konfiguration `AutomotivePmC`. Sie erhält wie die Software-Tasks über die Schnittstelle `Notify` die Bus-Nachrichten, die nicht nur die Nachrichten der lokalen Software-Tasks, sondern auch die Nachrichten der auf anderen ECUs laufenden Software-Tasks enthalten. Aufgrund dieser Nachrichten entscheidet die Komponente `AutomotivePmP`, welchen lokalen Power-Modus die ECU einnimmt. Diesen aktuellen Power-Modus verteilt sie wiederum über die Schnittstelle `ModeNotify` an die übrigen Teilkomponenten der Konfiguration `AutomotivePmC`. Über die Schnittstelle `Get` können die übrigen Komponenten zu jeder Zeit den aktuellen Power-Modus der ECU abfragen und diesen für die Ausführung ihrer Aufgaben nutzen.

Die Komponente `AutomotivePmP` bestimmt den lokalen Power-Modus durch einen deterministischen, endlichen Automaten, der als Eingabe den aktuellen Zustand und die empfangene Bus-Nachricht erhält und daraus den neuen lokalen Power-Modus der ECU berechnet. Dabei wird nicht zwischen einer Zustandsnachricht oder einer Daten-Nachricht aus der logischen Architektur unterschieden. Es werden beide Arten von Nachrichten für den Übergang des Power-Modus benutzt. Unter TinyOS sind die Power-Modi durchgängig durchnummeriert. Die Zustandsübergänge zwischen den Power-Modi werden mit Hilfe einer zweidimensionalen Tabelle festgelegt, wobei die erste Dimension die

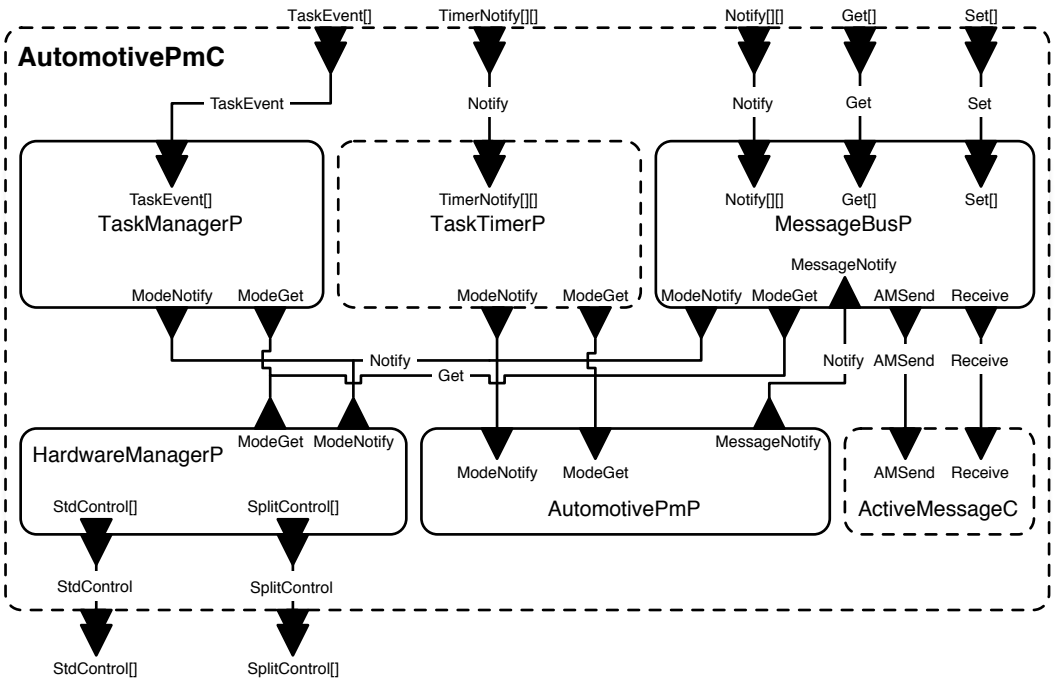


Abbildung 5.19: Der PM auf ECU-Ebene in der Konfiguration `AutomotivePmC`.

Power-Modi und die zweite Dimension die eingegangene Nachricht enthält. Als Eintrag in der Tabelle ist der neue Power-Modus abgelegt. Jedoch sind auch weitere Implementierungen vorstellbar, um aus den Bus-Nachrichten den lokalen Power-Modus der ECU zu bestimmen, z. B. dass die Historie bei der Zustandsberechnung mit betrachtet wird, wie es A. Barthels *et al.* vorschlagen [29]. Ebenfalls wird durch diese Teilung der Konfiguration `AutomotivePmC` ersichtlich, dass die Bestimmung des lokalen Power-Modus leicht ausgetauscht werden kann, da die Zustandsberechnung in der Komponente `AutomotivePmP` gekapselt ist.

Die Komponente `AutomotivePmC` bietet für die Ausführung der Software-Task die Schnittstellen `TaskEvent`, `TimerNotify`, `Notify`, `Get` und `Set` an, die sich in die Teilaufgaben Task-Benachrichtigung, Timer-Benachrichtigung und Bus-Nachrichten unterteilen lassen. Dafür enthält die Komponente `AutomotivePmC` entsprechend die Teilkomponenten `TaskManagerP`, `TaskTimerP` und `MessageBusP`, die diese Teilaufgaben realisieren. Dabei bedienen sie sich alle des lokalen Power-Modus der ECU, der durch die Komponente `AutomotivePmP` zentral bereitgestellt wird. Im Folgenden werden diese drei Komponenten detaillierter vorgestellt.

Die Komponente `TaskManagerP` informiert die an die Komponente `AutomotivePmC` verbundenen Software-Tasks über die Schnittstelle `TaskEvent`, ob diese wegen eines Wechsel des Power-Modus gestartet oder gestoppt wurden. Als ECU-spezifische Konfiguration enthält die Komponente `TaskManagerP` eine Tabelle, in der für jeden Power-Modus der ECU die aktiven Software-Tasks angegeben sind. Bei Wechsel des lokalen Power-Modus überprüft sie, ob sich der Zustand der Software-Tasks geändert hat und benachrichtigt entsprechend die Software-Tasks über die Schnittstelle `TaskEvent`.

Für die periodische Timer-Benachrichtigungen der Software-Tasks sorgt die Teilkomponente `TaskTimerP` über die Schnittstelle `TimerNotify`. Dabei enthält die Komponente `TaskTimerP` selbst die generische Konfiguration `TimerMilliC`, die die TinyOS Standard-Komponente für Timer ist. Über eine ECU-spezifische Tabelle erhält sie die Information, wie die Periode der verschiedenen Timer ist. Sobald ein Timer abläuft, werden die in dem lokalen Power-Modus aktiven Software-Tasks über die Schnittstelle `TimerNotify` darüber benachrichtigt.

Für die Zustellung der Bus-Nachrichten an die Software-Tasks kümmert sich die Teilkomponente `MessageBusP`, die über die Schnittstellen `Notify`, `Get` und `Set` mit den Software-Tasks in Verbindung steht. Dabei übernimmt die Komponente `MessageBusP` die Aufgabe des zentralen Nachrichten-Puffers, in dem jede Nachricht genau einmal gespeichert werden kann und immer die jeweils letzte Nachricht eines Typs über die Schnittstelle `Get` zur Verfügung stellt. Grundsätzlich wird jede Bus-Nachricht, die entweder lokal gesendet oder über den Bus empfangen wurde, zuerst in einem Empfangspuffer in der Komponente `MessageBusP` gespeichert und anschließend in einem eigenen TinyOS-Task abgearbeitet, bevor die nächste Nachricht behandelt wird. Die Komponente `MessageBusP` nutzt die TinyOS Standard-Komponente `ActiveMessageC`, um Nachrichten, die von anderen ECUs über den Bus gesendet werden, mit Hilfe der Schnittstelle `Receive` zu empfangen. Falls ein Software-Task eine Nachricht senden möchte, steht ihm die Schnittstelle `Set` zur Verfügung. Falls die Nachricht über den externen Bus gesendet werden muss, nutzt die Komponente `MessageBusP` die Komponente `ActiveMessageC`, die die Schnittstellen `AMSend` für das Senden von Bus-Nachrichten beinhaltet, wie es im TEP 116 spezifiziert ist [97]. Als ECU-spezifische Konfiguration enthält die Komponente `MessageBusP`, ob eine Nachricht in einem bestimmten lokalen Power-Modus über den externen Bus gesendet werden muss oder nicht.

Für die Steuerung der Hardware-PMCs ist die Komponente `HardwareManagerP` verantwortlich, die die Schnittstellen `StdControl` und `SplitControl` nutzt. Diese nutzt eine ECU-spezifische Tabelle, die für jeden Power-Modus der ECU einen Eintrag enthält, der den Zustand einer jeden Hardware-PMC beschreibt. Bei einem Wechsel des lokalen Power-Modus, überprüft die Komponente `HardwareManagerP`, ob sich eine Änderung des Zustands ergeben hat, und ändert den Zustand der Hardware-PMCs entsprechend über die Schnittstellen `StdControl` und `SplitControl`.

Alle Teilkomponenten der Konfiguration `AutomotivePmC` besitzen eine ECU-spezifische Konfiguration in Form von Tabellen, die als Arrays in nesC-Code für jede ECU unterschiedlich ausfallen. Diese Information ist die PM-Wissensbasis, die mit Hilfe der logischen Architektur zur Entwicklungszeit des Systems erzeugt wird. Für die PLASA-Plattform sind diese hauptsächlich durch Ingenieursleistung erstellt worden, wobei teilweise Werkzeuge diesen Prozess unterstützt haben, wie bei der Generierung der Konfiguration `TaskWiringC`.

Beim Empfangen einer Nachricht wird diese in der Komponente `AutomotivePmC` in einem eigenen TinyOS-Task wie folgt abgearbeitet. Über diesen bereitgestellten Mechanismus werden alle Nachrichten synchron hintereinander bearbeitet, d. h. dass alle Software-Tasks die empfangene Nachricht in demselben Power-Modus der ECU behandeln können. Zuerst wird die Nachricht aus dem Empfangspuffer geholt und in den zentralen Nachrichten-Puffer der Komponente `MessageBusP` eingetragen. Anschließend

wird die Nachricht über die Schnittstelle `Notify` an die Komponente `AutomotivePmP` weitergeleitet, die daraufhin überprüft, ob ein Wechsel des Power-Modus durchgeführt werden muss. Falls kein Wechsel des Power-Modus erfolgt, werden alle Software-Tasks, die in dem aktuellen Power-Modus aktiv und an der Nachricht interessiert sind, über den Empfang der Nachricht über die Schnittstelle `Notify` informiert. Falls während der Task-Ausführung neue Nachrichten versendet werden, werden diese wiederum in dem Empfangspuffer gespeichert. Sobald alle Software-Tasks ausgeführt worden sind, wird die nächste Nachricht im Empfangspuffer in einem neuen Aufruf des TinyOS-Tasks abgearbeitet.

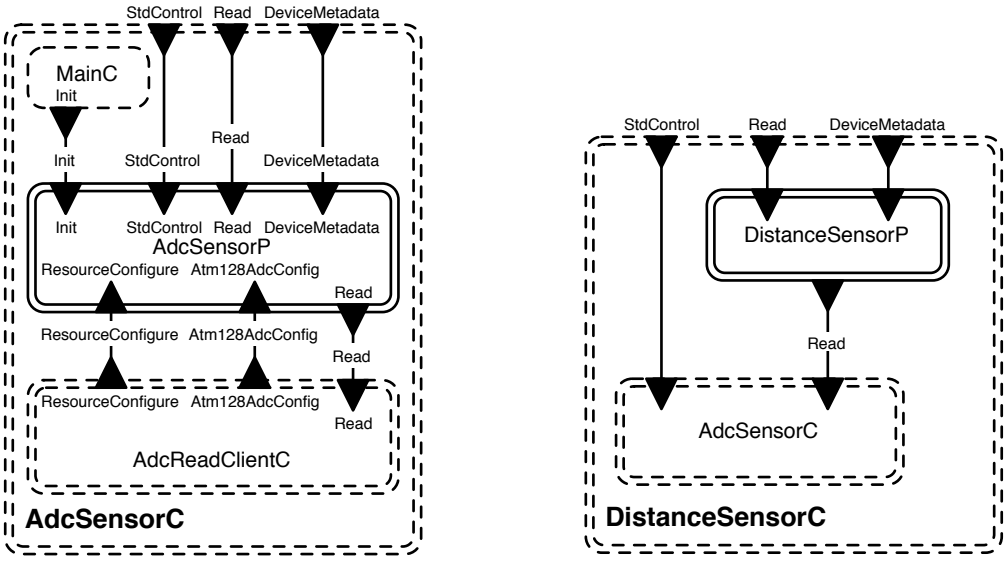
Falls ein Wechsel des Power-Modus durch die Komponente `AutomotivePmP` festgestellt wird, werden zuerst alle Teilkomponenten der Konfiguration `AutomotivePmC` über den Wechsel informiert. Die Teilkomponente `TaskManager` stoppt daraufhin die Software-Tasks, die im neuen Power-Modus nicht mehr aktiv sind. Anschließend werden die Software-Tasks gestartet, die in dem neuen Power-Modus aktiv sind. Ebenfalls aktiviert bzw. deaktiviert die Teilkomponente `HardwareManagerP` die Hardware-PMCs entsprechend dem neuen Power-Modus der ECU. Sobald alle Hardware- und Software-PMCs den entsprechenden Zustand eingenommen haben, werden alle aktiven Software-Tasks über den Empfang der Nachricht informiert und ausgeführt. Der weitere Ablauf ist derselbe, wie wenn kein Wechsel des Power-Modus stattgefunden hätte.

5.3.3 Software-Architektur der automotiven Software-Anteile in der PLASA-Plattform

Viele der Software-Tasks realisieren automotive Funktionen wie das Auslesen von den Abstands- und Geschwindigkeitssensoren oder das Ansteuern der Aktoren wie den Lenk-Servo oder den Fahrtenregler des Motors. Wie sich diese automotiven Funktionen in TinyOS-Komponenten umsetzen lassen wird im Folgenden beschrieben. Als erstes werden die Erweiterungen für die Sensoren im Anschluss die Erweiterungen für die Aktoren und am Schluss die Realisierung der Geschwindigkeits- und Fahrsequenz-Regelungen vorgestellt.

Erweiterungen für die Sensoren

In Abschnitt 2.3.3 sind die Sensoren und deren Messwert-Erfassung beschrieben. In der PLASA-Plattform sind zwei unterschiedliche Arten von Sensoren enthalten. Die einen sind analoge Sensoren, deren Ausgangsspannung von der gemessenen Größe abhängt. Darunter fallen die Distanzsensoren, der Batteriesensor und der Helligkeitssensor. Diese Sensoren sind an die verschiedenen Kanäle des ADC im Mikrocontroller verbunden. Die zweite Art der Sensoren liefern einen Takt abhängig von der gemessenen Größe. Diese Sensoren sind mit dem ICP (engl. *input capture pin*) des Mikrocontrollers verbunden. Ein Timer des Mikrocontrollers kann die Periode des Taktes messen. Auf diese Weise wird die Geschwindigkeit des Modell-Fahrzeugs ermittelt. Im Folgenden wird die Software-Architektur beider Arten von Sensoren beschrieben, wobei beide Sensoren die Schnittstellen `Read` oder `Get` besitzen, die im TEP 109 für Sensoren und Sensor Bords [53] beschrieben sind und bereits in Abschnitt 3.3.4 erläutert wurden.



(a) Die Komponente `AdcSensorC` für das Auslesen beliebiger Analog-Sensoren.

(b) Die Komponente `DistanceSensorC` für die Realisierung von Distanz-Sensoren.

Abbildung 5.20: Die Komponenten `AdcSensorC` und `DistanceSensorC` für die Messwert-Erfassung über einen ADC.

Der Unterschied zwischen den Sensoren ist, dass bei den Sensoren, die an den ADC angeschlossen sind, die Schnittstelle `Read` verwendet wird. Dies bedeutet, dass das Lesen des Sensors zuerst mit `read` angestoßen werden muss. Anschließend braucht der ADC eine Zeit bis er die Analog-Digital-Wandlung abgeschlossen hat. Sobald der Wert vorliegt, wird dies mit dem Ereignis `readDone`, das den gemessenen Wert enthält, signalisiert.

Im Folgenden wird nun auf die Software-Architektur aus den beiden unterschiedlichen Messwert-Erfassungen eingegangen. Dabei werden zuerst die Distanz-Sensoren vorgestellt, die über den ADC ihren Sensor-Wert ermitteln. Im Anschluss wird die Software-Architektur des Hodometers und des Geschwindigkeitssensors beschrieben, die ihre Werte über einen Takt eines angeschlossenen Hall-Sensors berechnen, dessen Funktionsweise in der Hardware-Architektur in Abschnitt 5.4.1 im Detail erläutert wird.

Da die Distanzsensoren an den ADC des ATmega1284P angeschlossen sind, werden auch die schon vorhandenen TinyOS-Komponenten für den ADC für die Messwert-Erfassung verwendet. Die Komponente `AdcReadClientC` steht für die Analog-Digital-Wandlung bereit, wie es in TEP 101 spezifiziert ist [67]. Die Komponente `AdcSensorC` nutzt die Komponente `AdcReadClientC` für die Analog-Digital-Wandlung und erweitert diese um das Ein- und Ausschalten der am ADC angeschlossenen externen Sensoren. Die Komponente ist in Abbildung 5.20a dargestellt. Sie kann die Pins des Mikrocontrollers steuern, mit denen die externen Sensoren ein- und ausgeschaltet werden. Die Komponente `AdcSensorP` ist neben dem Umrechnen der gewandelten Werte auch für die Konfiguration des ADCs über die Schnittstelle `Atm128AdcConfig` zuständig.

Aufbauend auf der generischen Konfiguration `AdcSensorC` für beliebige analoge Sensoren implementiert die Konfiguration `DistanceSensorC`, die in Abbildung 5.20b dar-

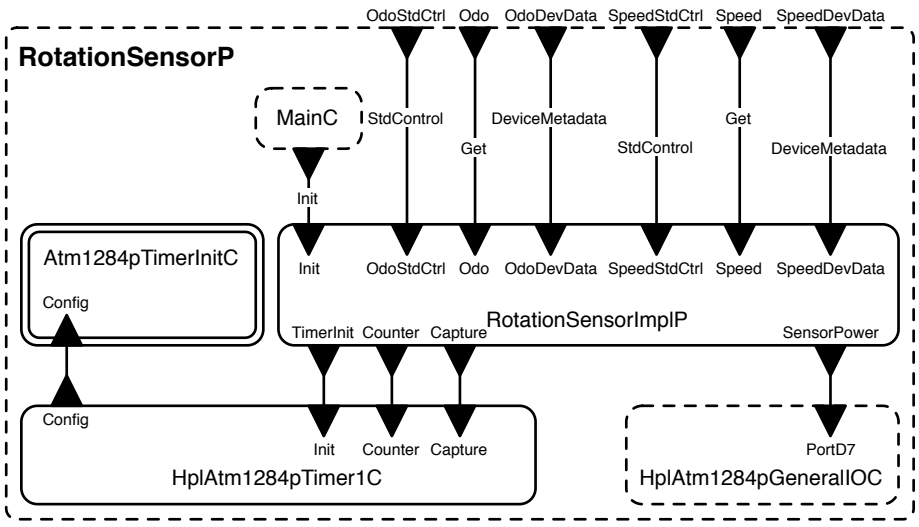


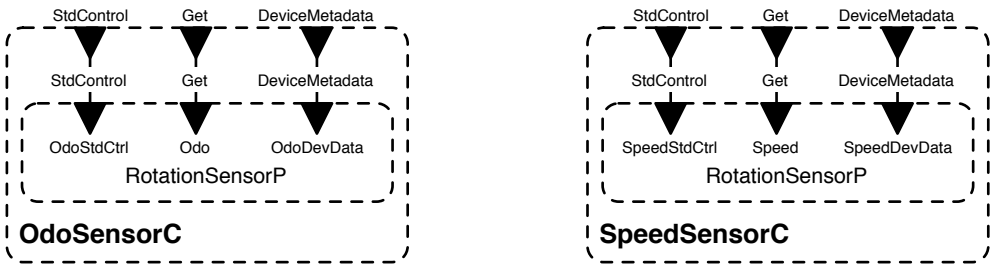
Abbildung 5.21: Die Komponente `RotationSensorP` als Sensor für die Kurbelumdrehung.

gestellt ist, einen Distanz-Sensor. Als Schnittstelle bietet sie `Read` an, wie es in TEP 109 spezifiziert ist. Die Konfiguration enthält das generische Modul `DistanceSensorP`, das aus der ermittelten analogen Spannung die Distanz des erfassten Objektes berechnet.

Auf die selbe Art und Weise sind weitere analoge Sensoren wie u. a. der Helligkeitssensor aufgebaut, der eine eigene Konfiguration `LightIntensSensorC` hat, die wiederum die Komponente `AdcSensorC` für die Analog-Digitalwandlung nutzt und über die Schnittstelle `Read` ausgelesen werden kann.

Nachdem die Sensoren beschrieben wurden, die den ADC für die Messwert-Erfassung nutzen, wird nun die Software-Architektur der Sensoren erläutert, die über ein Taktsignal ihren Messwert an den Mikrocontroller liefern. Als Beispiel sollen an dieser Stelle das Hodometer und der Geschwindigkeitssensor vorgestellt werden, die beide einen Hall-Sensor nutzen, um die Kurbelumdrehungen zu messen.

Für die Vermessung der Kurbelumdrehung ist die Komponente `RotationSensorP` zuständig, die sowohl die Geschwindigkeit des Modell-Fahrzeugs berechnet als auch die gefahrene Strecke ermittelt. Die Konfiguration `RotationSensorP` ist in Abbildung 5.21 dargestellt. Das Prinzip des Sensors basiert darauf, die Zeit zwischen zwei Flanken des Taktsignals zu messen. Dafür bietet der Mikrocontroller einen Timer an, der im Modul `HplAtm1284pTimer1C` repräsentiert wird. Über die Schnittstelle `HplAtm1284pCapture` wird der Capture-Mechanismus des ATmega1284P im Timer konfiguriert. Dieser speichert den Counter des Timers bei einer Taktflanke und in einem Register ab und informiert die Software per Interrupt. Das Modul `RotationSensorImplP` liest den Registerwert aus und berechnet auf Grund der Konfiguration des Timers und mit dem Register-Wert der vorherigen Takt-Flanke die benötigte Zeit zwischen zwei Takt-Flanken aus. Daraus wird die Umdrehungsgeschwindigkeit im Anschluss berechnet. Für das Hodometer reicht es aus die Takt-Flanken zu zählen, da ein Takt eine festgelegte Umdrehung der Kurbel bedeutet.



(a) Die Komponente `OdoSensorC` als Hodometer.

(b) Die Komponente `SpeedSensorC` als Geschwindigkeitssensor.

Abbildung 5.22: Komponenten für das Hodometers und den Geschwindigkeitssensor.

Der genutzte Timer wird über die Komponente `Atm1284pTimerInitC` konfiguriert. Die Komponente `HplAtm1284pGeneralIOC`, die die GPIO-Pin des Mikrocontrollers steuert, wird benutzt, um die externen Sensoren ein- und auszuschalten.

Auf der Konfiguration `RotationSensorP` baut das Hodometer auf und ist in der Konfiguration `OdoSensorC`, die in Abbildung 5.22a dargestellt ist, implementiert. Da die gefahrene Strecke über die Kurbel-Umdrehungen ermittelt wird, wird der Sensor-Wert aus der Komponente `RotationSensorP` nur weiter nach außen geleitet. Für den Geschwindigkeitssensor verhält es sich ähnlich, womit die Konfiguration `SpeedSensorC` den Sensor-Wert für die aktuelle Geschwindigkeit des Modell-Fahrzeugs von der Konfiguration `RotationSensorP` nach außen leitet, wie es in Abbildung 5.22b dargestellt ist.

Erweiterungen für die Aktoren

Die elektromechanischen Komponenten der PLASA-Plattform wie der Lenk-Servo oder der Fahrten-Regler des Motors werden über PWM-Signale gesteuert, wie später im Detail in Abschnitt 5.4.1 beschrieben wird. Dabei ist die Länge des High-Pegels des Signals ausschlaggebend, welchen Lenk-Winkel der Servo und welche Stromstärke der Fahrten-Regler an den Motor abgibt. Die Periode des PWM-Signals nimmt dabei keinen Einfluss auf die Steuerung dieser beiden Aktoren. Die Schnittstelle `MotionPwm`, die in Quelltext 5.4 abgebildet ist, wird auf der HIL-Ebene in TinyOS eingeführt, um genau diesen Sachverhalt gerecht zu werden. Diese besitzt die zwei Befehle `getDuty` und

```
1  interface MotionPwm<frequency_tag, pwm_size>
2  {
3    async command pwm_size getDuty();
4    async command void setDuty(pwm_size val);
5
6    async event void periodCompleted();
7  }
```

Quelltext 5.4: Die Schnittstelle `MotionPwm` für das Erzeugen eines PWM-Signals zur Steuerung eines Lenk-Servos oder eines Fahrten-Reglers.

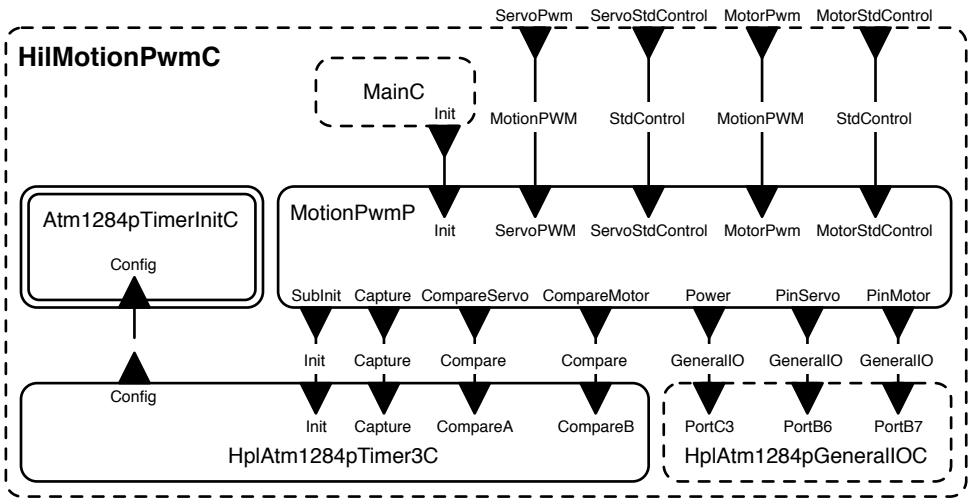


Abbildung 5.23: Die Konfiguration `HilMotionPwmC` für die Steuerung eines Lenk-Servos und eines Fahrten-Reglers.

`setDuty` mit denen jeweils die Dauer des High-Pegels abgefragt bzw. gesetzt werden kann. Über die HAA von TinyOS wird der Timer des Mikrocontrollers konfiguriert, damit dieser die PWM-Signale für die Steuerung des Lenk-Servos und des Motors erzeugt. Für eine erweiterte Steuerung des PWM-Signals ist das Ereignis `periodCompleted` in die Schnittstelle hinzugefügt worden, um die übergeordneten, Plattform-unabhängigen Komponenten darüber zu informieren, dass gerade eine Periode des PWM-Signals abgeschlossen wurde. Dies bietet die Möglichkeit einer Perioden-abhängigen Steuerung des Lenk-Servos bzw. des Motors.

Die Schnittstelle `MotionPwm` bietet keinen Befehl zur Änderung der Periode an, da dies nicht für die Steuerung der beiden Aktoren benötigt wird. Die Periode wird somit selbstständig in der Plattform-abhängigen Schicht festgelegt. Damit wird dem Hardware-abhängigen Teil die Festlegung der Periode überlassen und muss nicht durch die Komponenten in der höheren Schicht konfiguriert werden. Auf dieser Schnittstelle aufbauend können die höheren Software-Komponenten sowohl für die Steuerung des Lenk-Servos als auch für den Fahrten-Regler realisiert werden.

Die Plattform-unabhängige Schnittstelle `MotionPwm` wird nun von der Komponente `HilMotionPwmC` implementiert und bietet sowohl für den Lenk-Servo als auch für den Motor die Schnittstelle `MotionPwmC` an. Um den Servo oder den Motor einzeln ein- bzw. ausschalten zu können, wird zusätzlich die Schnittstelle `StdControl` einzeln für beide Aktoren angeboten. In Abbildung 5.23 ist die Konfiguration `HilMotionPwmC` zusammen mit der Implementierung auf dem DAB abgebildet.

Die Komponente `HplAtm1284pTimer3C` ist aus der HPL-Schicht des ATmega1284Ps und repräsentiert den Timer 3, der für die Generierung des PWM-Signals für den Lenk-Servo und des Fahrten-Reglers verantwortlich ist. Für die Steuerung der einzelnen Pins des ATmega1284Ps ist die Komponente `HplAtm1284pGeneralIOC` zuständig, über die festgelegt werden kann, ob ein Pin Ein- oder Ausgang ist und welchen Pegel dieser als Ausgang einnehmen kann. Über diese Komponente werden die Pins, über die das

PWM-Signal an den Lenk-Servo oder den Motor geht, als Ausgänge geschaltet.

Aufbauend auf dieser Plattform-unabhängigen Schnittstelle `MotionPwm`, die die Komponente `HilMotionPwmC` anbietet, werden nun die Komponenten für die Steuerung des Lenk-Servos und des Motors implementiert. Die Steuerung des Servos übernimmt dabei die Komponente `ServoC`, die als Eingabe einen Einschlag erhält und dazu das PWM-Signal berechnet. Dafür werden Konfigurationen verwendet, da sich je nach Servo die dafür notwendigen PWM-Signale unterscheiden. Für die Steuerung des Motors ist entsprechend die Komponente `MotorC` zuständig, die die passenden PWM-Signale für den Motor berechnet.

```
1 interface Servo
2 {
3   command error_t left(uint8_t val);
4   command error_t right(uint8_t val);
5 }
```

Quelltext 5.5: Die Schnittstelle `Servo` für das Steuern eines Servos.

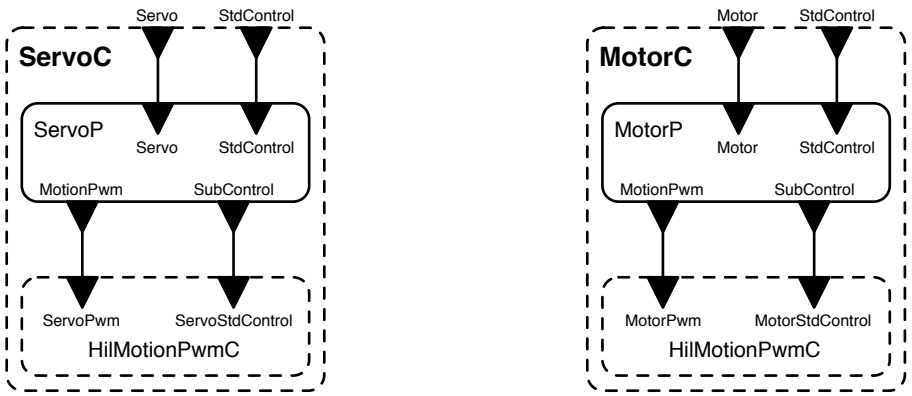
Die Software-Architektur für das Steuern des Lenk-Servos ist in Abbildung 5.24a dargestellt. Die Komponente `ServoC` bietet die Schnittstelle `Servo` an, die im Quelltext 5.5 abgebildet ist. Der Befehl `left` sorgt für einen Einschlag des Lenk-Servos nach links, während der Befehl `right` den Servo nach rechts lenkt. Als Parameter wird die Stärke des Einschlags angegeben, wobei 0 kein Einschlag und 255 ein voller Einschlag nach links bzw. rechts bedeutet.

Die Konfiguration `ServoC` bietet die Schnittstelle `Servo` und `StdControl` an, mit denen der Lenk-Servo gesteuert und ein- bzw. ausgeschaltet werden kann. Die Konfiguration enthält die beiden Komponenten `HilMotionPwmC`, die zuvor beschrieben wurde, und das Modul `ServoP`, das aus dem Servo-Befehl für den Einschlag die Dauer des High-Pegels im PWM-Signal berechnet. Der berechnete Wert wird anschließend über die Schnittstelle `MotionPwm` an die Komponente `HilMotionPwmC` weitergeleitet. In dem Modul `ServoP` ist dabei die spezifische Konfiguration für den eingebauten Lenk-Servo hinterlegt, mit Hilfe derer das für den verbauten Lenk-Servo spezifische PWM-Signal berechnet wird.

```
1 interface Motor
2 {
3   command error_t forward(uint16_t val);
4   command error_t backward(uint16_t val);
5   command error_t brake(uint16_t val);
6 }
```

Quelltext 5.6: Die Schnittstelle `Motor` für das Steuern eines Motors.

Ähnlich verhält es sich mit der Steuerung des Motors. Dafür wurde die Schnittstelle `Motor` eingeführt, die in Quelltext 5.6 dargestellt ist. Hier gibt es die drei Befehle `forward`, `backward` und `brake`, die die drei Bewegungsarten des Motors steuern. Mit `forward` wird vorwärtsgefahren, mit `backward` rückwärts und mit `brake` wird gebremst. Als Parameter wird jeweils die Leistung für die jeweilige Bewegung in einem 16-bit Wert



(a) Die Komponente `ServoC` für die Steuerung eines Lenk-Servos.

(b) Die Komponente `MotorC` für die Steuerung eines Motors.

Abbildung 5.24: Ansteuerung des Lenk-Servos und des Motors unter TinyOS.

übergeben. Dabei bedeutet 0 keine Bewegung bis 32767, was einer vollen Leistung in der Vor- bzw. Rückwärtsfahrt oder einer Vollbremsung entspricht.

Die Software-Architektur für den Motor ist konzeptuell gleich aufgebaut wie die Architektur des Servos. Die Konfiguration `MotorC`, die in Abbildung 5.24b abgebildet ist, berechnet die Dauer des High-Pegels im PWM-Signal. Die Komponente bietet die Schnittstelle `Motor` und die Schnittstelle `StdControl` an, mit der die Bewegung des Motors gesteuert bzw. der Motor ein- und ausgeschaltet werden kann. Das Modul `MotorP` enthält dabei die Konfiguration des Fahrten-Reglers, aus der das spezifische PWM-Signal für den im Modell-Fahrzeug verbauten Fahrten-Regler berechnet wird.

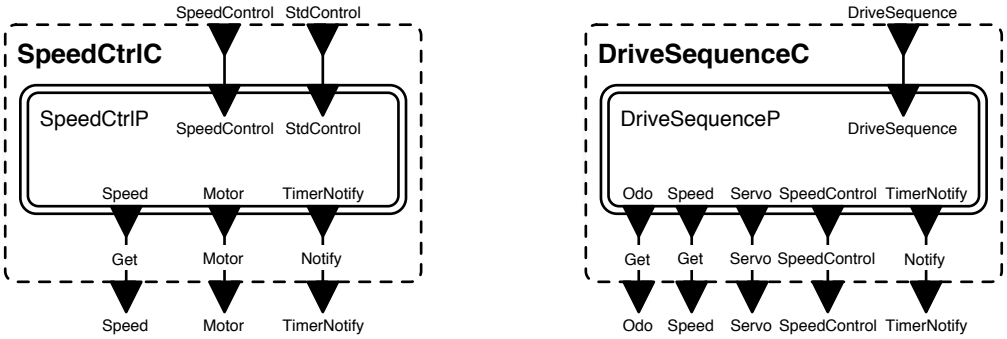
Erweiterungen für die Regelungen

Im Folgenden werden zwei Regelungen für Automotive Systeme exemplarisch beschrieben – zum einen der Geschwindigkeitsregler und zum anderen die Fahrsequenz-Regelung.

Der Geschwindigkeitsregler ist in der Komponente `SpeedCtrlC` realisiert, die über die Schnittstelle `SpeedControl` gesteuert wird. Diese ist in Quelltext 5.7 abgebildet und besitzt die beiden Befehle `forward` und `backward`, mit denen die gewünschte Richtung angegeben wird. Als Parameter wird die einzustellende Geschwindigkeit und die Beschleunigung übergeben, mit der der Regler die Geschwindigkeit anfahren soll. Die Komponente `SpeedCtrlC`, die in Abbildung 5.25a dargestellt ist, hat zusätzlich die Schnitt-

```
1 interface SpeedControl
2 {
3   command error_t forward(int16_t speed, uint8_t accel);
4   command error_t backward(int16_t speed, uint8_t accel);
5 }
```

Quelltext 5.7: Die Schnittstelle `SpeedControl` für das Steuern eines Geschwindigkeitsreglers.



(a) Geschwindigkeitsregler in der Komponente `SpeedCtrlC`.

(b) Fahrsequenz-Regler in der Komponente `DriveSequenceC`.

Abbildung 5.25: Umsetzung des Geschwindigkeitsreglers und der Fahrsequenz-Regelung in TinyOS.

stelle `StdControl`, mit dem der Regler gestartet oder gestoppt werden kann. Um die gewünschte Geschwindigkeit zu regeln, benötigt der Geschwindigkeitsregler die aktuelle Geschwindigkeit, die er über die Schnittstelle `Get` erfragen kann. Über die Schnittstelle `Motor` wird die errechnete Leistung an den Motor übergeben. Der Regler ist in der PLASA-Plattform als PID-Regler (engl. *proportional–integral–derivative controller*) realisiert, der über die Schnittstelle `Notify` periodisch angestoßen wird [121].

Aufbauend auf dem Geschwindigkeitsregler wird eine Fahrsequenz-Regelung implementiert, die eine Strecke aus einzelnen Segmenten abfährt. Die Segmente bestehen dabei jeweils aus einem Einschlagswinkel, einer Geschwindigkeit und einer Beschleunigung. Die Segmente werden dann kontinuierlich abgefahren, womit nach dem vollständigem Abfahren eines Segments im Anschluss sofort das nächste Segment beginnt. Der Regler ist in der TinyOS Komponente `DriveSequenceC` realisiert, die in Abbildung 5.25b graphisch dargestellt ist. Über die Schnittstelle `DriveSequence`, die in Quelltext 5.8 abgebildet ist, wird der Fahrsequenz-Regler gesteuert. Die Befehle `start`, `stop` starten bzw. stoppen die Fahrsequenz-Regelung. Für die Verwaltung der Fahrsequenz-Segmente dienen die Befehle `clear`, `skipSeqment` und `addSegment`, die für das Löschen aller Segmente, für das Überspringen eines Segments bzw. für das Hinzufügen eines Segments

```
1  interface DriveSequence
2  {
3    command error_t start();
4    command error_t stop();
5    command error_t clear();
6    command error_t skipSegment();
7    command error_t addSegment(plasa_msg_drive_segment_t *segment);
8    event void segmentFinished(plasa_msg_drive_status_t *status);
9  }
```

Quelltext 5.8: Die Schnittstelle `DriveSequence` zur Konfiguration der Fahrsequenz-Regelung.

am Ende der Segment-Liste zuständig sind. Über das Ereignis `segmentFinished` wird signalisiert, wenn ein Segment vollständig abgefahren ist [121].

Als Eingangsgrößen für die Steuerung benötigt der Regler die aktuell gefahrene Strecke und die Geschwindigkeit, die er über die Schnittstelle `Get` mit den Namen `Odo` bzw. `Speed` erhält. Für das Steuern der Geschwindigkeit nutzt er die Schnittstelle `SpeedControl`, die bereits bei der Geschwindigkeitsregelung erläutert wurde. Für die Steuerung des Lenk-Servos wird die Schnittstelle `Servo` verwendet. Die Regelung wird periodisch über die Schnittstelle `Notify` angestoßen, wobei überprüft wird, ob die aktuell gefahrene Strecke sich noch innerhalb des Segments befindet oder ob das aktuelle Segment bereits abgefahren wurde. Falls das aktuelle Segment zu Ende ist, wird das nächste Segment in der Liste geholt und dessen Lenkeinschlag und Geschwindigkeit angesteuert.

5.4 Hardware-Architektur der PLASA-Plattform

Nachdem in den vorhergehenden Abschnitten die Anwendungsfälle, die logische und die Software-Architektur der PLASA-Plattform beschrieben wurde, wird nun die Hardware-Architektur der PLASA-Plattform vorgestellt. Dabei orientiert sich diese an einem Automotiven System, wie es in Abschnitt 2.3 eingeführt wurde. Jedoch werden folgende Bereiche der automotiven Hardware-Architektur in dem PLASA-Modell wie folgt abstrahiert:

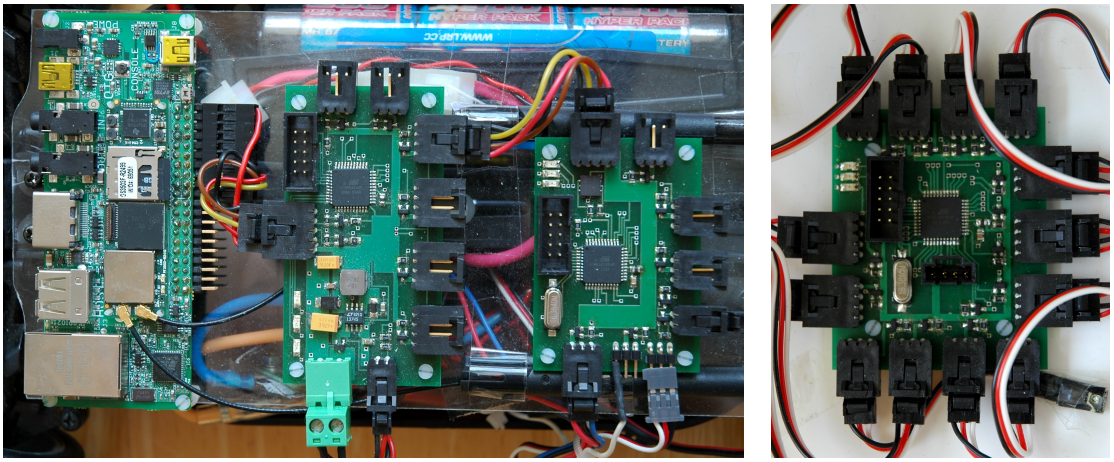
Domänen. Die PLASA-Plattform besitzt nur eine gemeinsame Domäne, in der alle ECUs an einem gemeinsamen Bus-System angeschlossen sind und somit ohne zusätzlichem Gateway kommunizieren können.

Anzahl der ECUs. Die Anzahl der ECUs beschränkt sich auf vier Stück. Somit steht nur ein sehr kleines Automotives System mit der PLASA-Plattform zur Verfügung. Jedoch lassen sich die Konzepte auf ein reales Automotives System mit ca. 80 ECUs, die mehreren Teilnetze zugeordnet sind, übertragen.

Bus-System. Der in dem Modell-Fahrzeug verbaute I^2C-Bus ist kein Feldbus [122], der für die Vernetzung einzelner ECUs eingesetzt wird. Jedoch verhält sich der I^2C-Bus ähnlich zum CAN-Bus, der in realen Automotiven Systemen zum Einsatz kommt, und kann somit in der PLASA-Plattform als zentrales Bus-System eingesetzt werden.

Mikrocontroller. Bei den Mikrocontrollern sind ebenso keine automotive-fähigen Varianten in der PLASA-Plattform verbaut. Wie allerdings in Abschnitt 2.3.1 über die automotive-fähigen Mikrocontroller bereits dargestellt wurde, unterscheiden sich die automotive-fähigen 8-Bit Mikrocontroller von den in der PLASA-Plattform verbauten Mikrocontrollern nur unwesentlich.

Die PLASA-Plattform besteht aus vier ECUs, die über den I^2C-Bus miteinander vernetzt sind. Dabei werden zwei unterschiedliche Klassen von ECUs eingesetzt. Die eine Klasse besteht aus der Gumstix-Plattform mit einem 600 MHz getaktetem OMAP3530



(a) Großaufnahme des Gumstix, PSBs und DABs (von links nach rechts).

(b) Großaufnahme des LSBs.

Abbildung 5.26: Großaufnahmen der im Modell-Fahrzeug verbauten ECUs.

von Texas Instruments (ein ARM Cortex A8 Prozessor). Die andere Klasse von ECUs besteht aus 8-Bit Mikrocontrollern der Firma Atmel. Somit bildet die Plattform ein heterogenes, verteiltes System, wie es bis auf die zuvor genannten Abstraktionen in der automotiven Domäne zu finden ist. In Abbildung 5.26a ist von links nach rechts der Gumstix, das PSB und das DAB dargestellt. Die Abbildung 5.26b zeigt das LSBs, das auf der Unterseite des Daches angebracht ist, in der Großansicht.

Eine schematische Darstellung der Plattform ist in Abbildung 5.27 zu sehen. Die ECUs sind mittels des I^2C-Bus miteinander verbunden und werden aus einem 5 V Bordnetz, das vom PSB aus der Batterie-Spannung generiert wird, mit Strom versorgt. Am DAB hängen sowohl der Fahrten-Regler, der den Motor steuert, als auch der Lenk-Servo. Für die Messung der Geschwindigkeit ist am DAB ein Rotationssensor angeschlossen, der die Rotationen der Kurbelwelle misst. Die genaue Funktionsweise des Sensors wird im nächsten Abschnitt erklärt, wenn die elektromechanischen Komponenten der PLASA-Plattform beschrieben werden. Am LSB sind die Abstandssensoren, die an der Karosserie des Fahrzeugs verteilt angebracht sind, der Helligkeitssensor und zusätzlich die Fahrzeug-Lichter angeschlossen.

Das PSB, DAB und LSB sind speziell für die PLASA-Plattform entwickelt worden, um den Hardware-Anforderungen für das DPM für Automotive Systeme gerecht zu werden. So müssen die an die ECUs angeschlossenen Sensoren und Aktoren hardwaremäßig ein- und ausgeschaltet werden. Auf die konkrete Realisierung der einzelnen ECUs wird in Abschnitt 5.4.2 eingegangen. Der Gumstix ist mit dem OMAP3530-Prozessor eine ARM-Plattform, die es als Prototypen-Bord zu kaufen gibt [62], und die eine drahtlose Anbindung über Bluetooth und WLAN bietet.

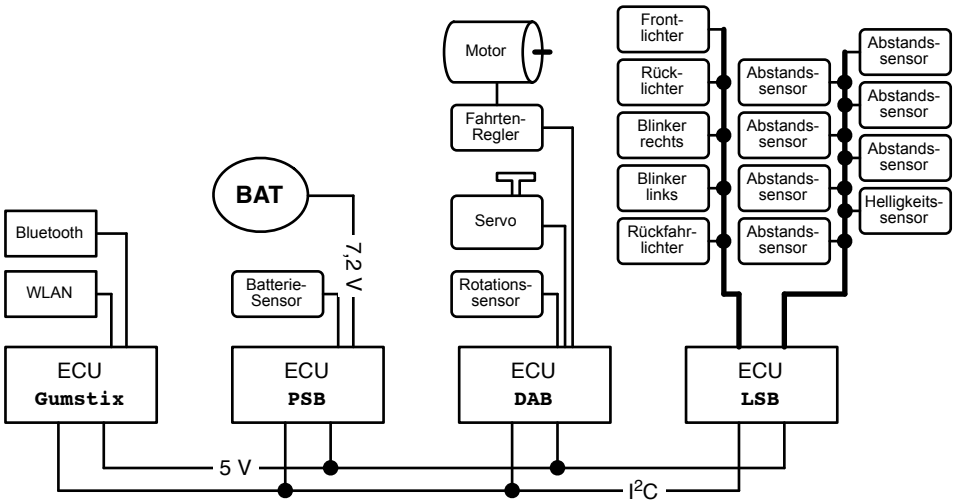


Abbildung 5.27: Die PLASA-Plattform mit ihren ECUs und den verbauten Sensoren bzw. Aktoren.

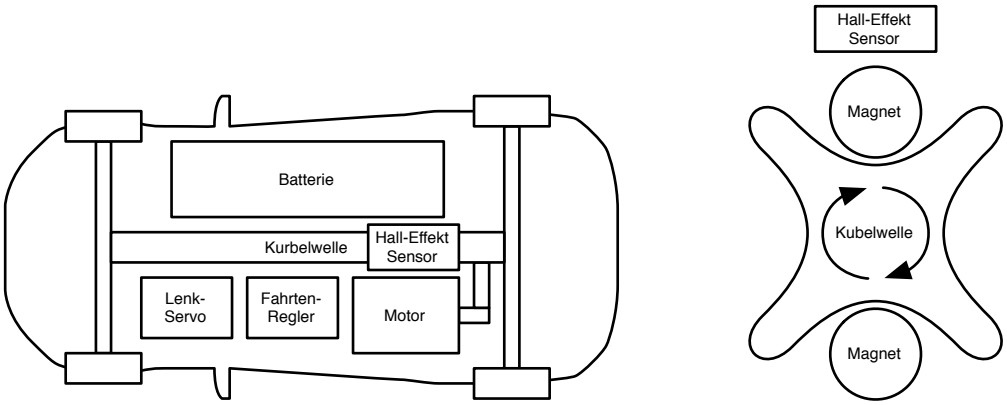
5.4.1 Elektromechanische Komponenten der PLASA-Plattform

Bevor auf die einzelnen ECUs der PLASA-Plattform eingegangen wird, werden die elektromechanischen Komponenten des Modell-Fahrzeugs für das weitere Verständnis der Hardware-Architektur im Folgenden kurz vorgestellt.

In Abbildung 5.28a sind die elektromechanischen Komponenten des Modell-Fahrzeugs abgebildet. Darunter fallen der Lenk-Servo, der Fahrten-Regler mit Motor, die Welle für den Vierrad-Antrieb sowie eine 7,2 V Batterie mit 4200 mAh, die das gesamte Modell-Fahrzeug inklusive der ECUs, des Lenk-Servos und des Motors mit Strom versorgt. Die elektromechanischen Komponenten in dem Modell-Fahrzeug sind Standard-Komponenten, die in RC-Car verbaut werden.

Für die Lenkung des Modell-Fahrzeugs kommt ein digitaler Servo zum Einsatz, der ein PWM-Signal als Eingabe für den Lenk-Einschlag erhält. Dabei ist nur die Dauer des High-Pegels, die zwischen 0,5 ms und 2,5 ms bei einer Periode von ca. 20 ms liegt, im PWM-Signal für den Einschlag-Winkel des Servos ausschlaggebend. Die Perioden-Dauer nimmt keinen Einfluss auf den Lenk-Einschlag und kann somit innerhalb des Toleranzbereichs, der sich zwischen den verschiedenen Servos unterscheiden kann, variiert werden. Die Lenk-Richtung und die Größe des Lenk-Einschlags bei einer bestimmten Dauer des High-Pegels ist nicht über die verschiedenen Servos normiert und ist somit von dem verbauten Lenk-Servo abhängig. Dies führt dazu, dass jeder Lenk-Servo vorab kalibriert werden muss, bevor der Servo im Modell-Fahrzeug eingesetzt werden kann.

Als Motor kommt ein Poison Truck Puller zum Einsatz, der sich durch seine geringe maximale Umdrehungsgeschwindigkeit von 6000 Umdrehungen/min auszeichnet. Im Gegensatz zum Standard-Motor, der eine maximale Umdrehungsgeschwindigkeit von 15000 Umdrehungen/min erreicht, dreht der Poison Truck Puller nur ca. 40 % so schnell. Für die PLASA-Plattform wurde dieser Motor gewählt, um im niedrigeren Geschwindigkeitsbereich kleinere Geschwindigkeitsstufen zu erhalten. Dies ergibt sich



(a) Die Übersicht über die elektromechanischen Komponenten des Modell-Fahrzeugs.

(b) Eine schematische Darstellung des Hall-Effekt-Sensors.

Abbildung 5.28: Elektromechanische Komponenten im Modell-Fahrzeug.

daraus, dass der Fahrten-Regler diskrete Stufen für Motor-Leistung hat und somit sich diese Stufen auf dem gesamten Geschwindigkeitsbereich des Motors verteilen. Da beim Truck Puller die Höchstgeschwindigkeit niedriger ausfällt, sind ebenfalls die Geschwindigkeitsstufen kleiner, was eine präzisere Steuerung des Modell-Fahrzeugs im unteren Geschwindigkeitssegment erlaubt. Besonders für das autonome Einparken ist die präzise Steuerung im langsamen Geschwindigkeitsbereich von Vorteil.

Das Modell verfügt über einen Vierrad-Antrieb, der über eine Kurbelwelle alle Räder des Fahrzeugs antreibt. Diese Kurbelwelle verläuft in der Mitte des Modells und wird ebenfalls für die Messung der Geschwindigkeit des Fahrzeugs genutzt. Dabei wird die Rotationsgeschwindigkeit der Kurbelwelle über den Hamlin 55100 Hall-Effekt-Sensor [64] und zwei Magnete, die gegenüber auf der Welle angebracht sind, ermittelt. Die mechanische Konstruktion ist schematisch in Abbildung 5.28b verdeutlicht. Jedes Mal, wenn der Magnet an dem Hall-Sensor vorbeiläuft, wird an der Signal-Leitung eine fallende Flanke erzeugt. Sobald der Magnet an dem Hall-Sensor vorbeigelaufen ist, erfolgt wieder eine steigende Flanke. Auf diese Weise kann der Mikrocontroller, an dem der Sensor angeschlossen ist, das Vorbeilaufen der Magnete am Hall-Sensor erkennen und die vergangene Zeit messen. Da die Übersetzung zwischen Kurbelwelle und den Rädern des Modell-Fahrzeugs konstant ist, kann mit Hilfe der Strecke, die bei einer Kurbel-Umdrehung zurückgelegt wird, die Geschwindigkeit des Modell-Fahrzeugs berechnet werden.

5.4.2 ECUs der PLASA-Plattform

Im Folgenden wird auf die ECUs der PLASA-Plattform eingegangen und deren Aufgaben beschrieben. Wie bereits in der Übersicht über die PLASA-Plattform vorgestellt wurde, sind vier ECUs im Modell-Fahrzeug verbaut. In Tabelle 5.7 ist eine Übersicht über die vier verbauten ECUs mit deren Rechen-, Speicher- und Kommunikationsfähigkeiten gegeben.

Anhand dieser Tabelle ist erkennbar, dass zwei unterschiedliche Klassen von Mikro-

Komponente	Gumstix	PSB	DAB	LSB
Mikrocontroller	TI OMAP3530	Atmel ATmega644P	Atmel ATmega1284P	Atmel ATmega1284P
Taktrate	600 MHz	8 MHz	16 MHz	16 MHz
Arbeitsspeicher	256 MB DDR RAM	4K Bytes SRAM	16K Bytes SRAM	16K Bytes SRAM
Persistenter Speicher	256 MB NAND Flash 2 GB MicroSD Karte	64K Bytes Flash 2K Bytes EEPROM	128K Bytes Flash 4K Bytes EEPROM	128K Bytes Flash 4K Bytes EEPROM
Kommunikation	Ethernet WLAN Bluetooth I^2C-Bus SPI-Bus	I^2C-Bus SPI-Bus	I^2C-Bus SPI-Bus	I^2C-Bus SPI-Bus

Tabelle 5.7: Hardware-Ausstattung der ECUs in der PLASA-Plattform.

controllern in der PLASA-Plattform eingesetzt werden. Während auf dem Gumstix der OMAP3530 Applikationsprozessor [153] verbaut ist, ist auf dem PSB, DAB und LSB ein 8-Bit Mikrocontroller enthalten. Auf dem PSB ist der kleinere ATmega644P [8] verbaut, der sich durch seinen geringeren persistenten und flüchtigen Speicher unterscheidet. Auf dem DAB und LSB wird der ATmega1284P [7] eingesetzt, um mehr Programmspeicher und einen zusätzlichen Timer zur Steuerung der elektromagnetischen Komponenten zu besitzen.

Als Fahrzeug-Bus wird der I^2C-Bus im 400 kHz High-Speed Mode verwendet, um die vier ECUs miteinander zu verbinden. Normalerweise wird der I^2C-Bus mit nur einem Master und mehreren Slaves betrieben. Allerdings kann in dieser Kommunikationsart nur der Master die Kommunikation initiieren und der Slave antwortet auf die Anfragen des Masters. Jedoch muss in der PLASA-Plattform jede ECU gleichberechtigt die Kommunikation starten können. Somit wird der I^2C-Bus im Multi-Master Modus betrieben, womit jede ECU als Master auf dem I^2C-Bus agiert und somit die Kommunikation starten kann.

Der Einsatz des I^2C-Busses auf der Plattform lässt sich wie folgt begründen. Die Atmel Mikrocontroller besitzen standardmäßig einen I^2C-Bus Controller auf dem Chip, so dass keine zusätzlichen Bus-Controller und Transceiver eingesetzt werden müssen. Damit werden die Kosten und die Größe der ECUs geringer. Andererseits verhält sich der I^2C-Bus im Multi-Master Modus ähnlich zum CAN-Bus, der üblicherweise in der automotiven Domäne verbaut wird. Die Geschwindigkeit des Busses ist ebenso vergleichbar.

Im Folgenden wird nun detaillierter auf die verschiedenen ECUs der PLASA-Plattform eingegangen, wobei die angeschlossenen Sensoren und die verbauten IC-Komponenten beschrieben werden. Zuerst wird der Aufbau der drei ECUs mit dem Atmel Mikrocontroller erklärt, die vorrangig für die Steuerung der elektromechanischen Komponenten zuständig sind. Im Anschluss daran erfolgt eine kurze Beschreibung des Gumstix, der

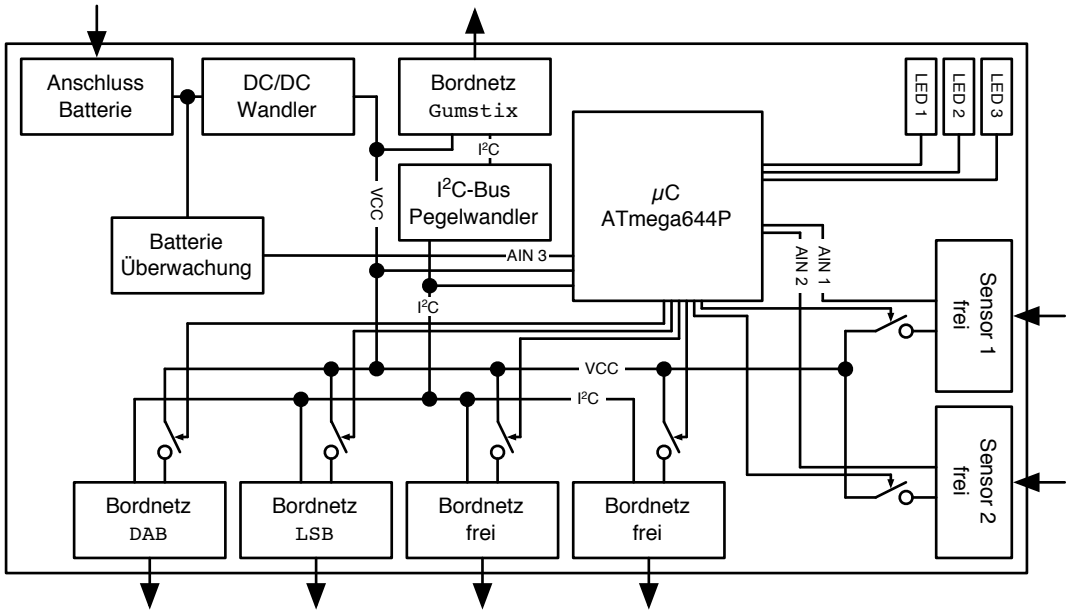


Abbildung 5.29: Schematische Übersicht über das PSB.

mit Hilfe der Bluetooth Kommunikation die Verbindung zur Nintendo Wii Remote herstellt.

Das Power Supply Board

Als erste ECU wird das PSB beschrieben. Wie der Name schon andeutet, übernimmt das PSB die Batterie-Überwachung und die Generierung der 5 V Bordnetz-Spannung, mit der die PLASA-Plattform versorgt wird. Dafür besitzt das PSB einen DC/DC-Wandler, der aus der Batterie-Spannung von 7,2 V die benötigten 5 V erzeugt, aus der alle weiteren ECUs versorgt werden. Schematisch ist der Aufbau des PSB in Abbildung 5.29 zu sehen. Die Stromversorgung erhält das PSB von der 7,2 V Batterie. Diese wandelt der DC/DC-Wandler LT1347 [102] von Linear Technology in die 5 V Bordnetz-Spannung um. Die 5 V Spannung steht anschließend für den verbauten Mikrocontroller ATmega644P und die angeschlossenen ECUs zur Verfügung.

Der OMAP3530 und die Atmel-Mikrocontroller besitzen unterschiedliche I/O-Spannungen. Während die Atmel-Mikrocontroller mit einer I/O-Spannung von 5 V arbeiten, verwendet der OMAP3530 eine I/O-Spannung von 1,8 V. Aus diesem Grund muss eine Pegelwandlung für den I^2C-Bus vorgenommen werden, was ebenfalls das PSB übernimmt. Die Pegelwandlung erfolgt durch eine Schaltung mit zwei MOSFETs (engl. *metal oxide semiconductor field-effect transistor*), wie sie im I^2C-Bus-Standard beschrieben wird [122].

Das PSB besitzt für die weiteren ECUs der PLASA-Plattform vier Anschlüsse, an denen das PSB die 5 V Bordnetz-Spannung und den I^2C-Bus im 5 V I/O-Pegel zur Verfügung stellt. Die Besonderheit an diesen Anschlüssen ist, dass der ATmega644P die

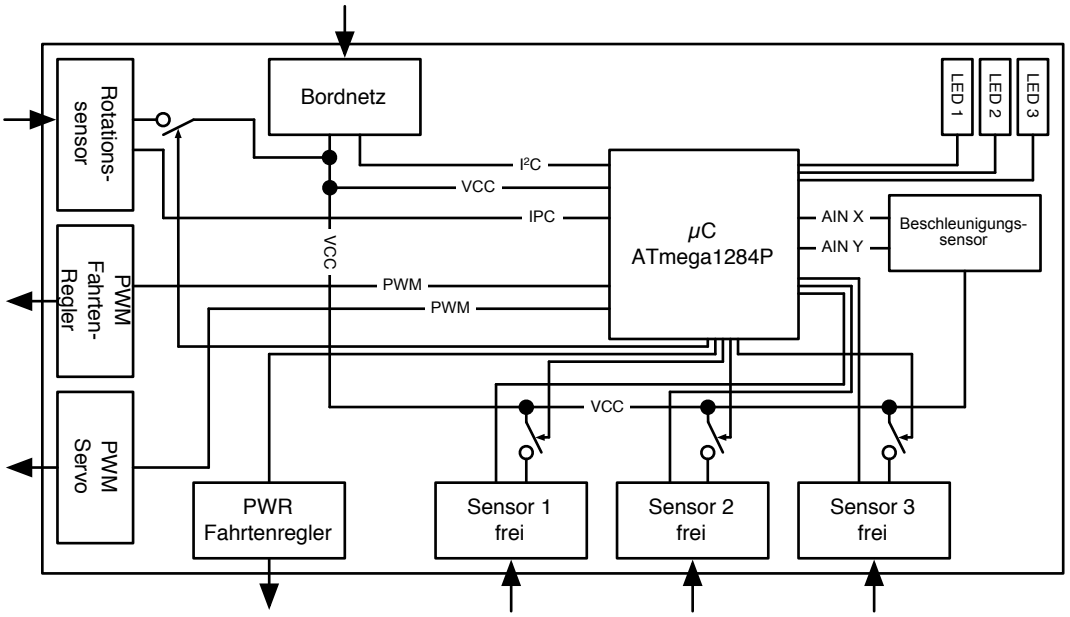


Abbildung 5.30: Schematische Übersicht über das DAB.

Spannungsversorgung jeder einzelnen Anschluss-Stelle über MOSFETs schalten kann. Somit ist es möglich, ECUs in der PLASA-Plattform ohne ein eigenes Power-Management ein- oder auszuschalten. An diesen Anschlüssen ist zum einen das DAB und zum anderen das LSB angeschlossen.

Für die Überwachung der Batterie wird ihre Spannung an einem der ADC-Eingänge des ATmega644Ps gemessen. Da die Batterie-Spannung 7,2 V beträgt und der ATmega644P auf dem PSB nur eine Betriebsspannung und Referenz-Spannung für den ADC von 5 V besitzt, wird ein Spannungsteiler davorgeschaltet, der die Batterie-Spannung halbiert. Damit kann das PSB die Spannung der Batterie überwachen und gegebenenfalls per Software-Steuerung darauf reagieren.

Das PSB bietet zusätzlich noch zwei Anschlüsse für analoge Sensoren, deren Spannungsausgänge an die ADC-Eingänge des ATmega644Ps angeschlossen sind. Dabei kann der Mikrocontroller per MOSFET jeden Sensor individuell ein- und ausschalten.

Für die Statusanzeige des PSB sind drei LEDs mit dem ATmega644P verbunden. Diese können individuell per Software gesteuert werden und können somit für die Anzeige bestimmter Zustände unter anderem für die Anzeige einer niedrigen Batteriespannung genutzt werden.

Das Drive Assistant Board

Als nächste ECU wird das DAB näher beschrieben, das als Hauptaufgabe die Steuerung des Fahrten-Reglers und des Lenk-Servors übernimmt. In Abbildung 5.30 ist eine schematische Darstellung des DABs zu sehen. Als Mikrocontroller wird der ATmega1284P eingesetzt, der über die 5 V Bordnetz-Spannung vom PSB versorgt wird und an dem

Fahrzeug-Bus angeschlossen ist.

Auf dem DAB ist der Beschleunigungssensor MXA2300 von MEMSIC verbaut, der die Beschleunigung in X- und Y-Richtung auf dem Chip misst [108]. Die Beschleunigungswerte werden vom MXA2300 in analoge Spannungswerte umgewandelt, die vom ADC des ATmega1284Ps digitalisiert werden. Zusätzlich sind wie im PSB auch im DAB drei LEDs für die Zustandsanzeige verbaut, die über die Software-Steuerung frei programmiert werden können.

Wie bei den elektromechanischen Komponenten beschrieben wurde, wird sowohl der Fahrten-Regler als auch der Lenk-Servo über PWM-Signale gesteuert. Beide PWM-Signale werden vom ATmega1284P erzeugt und direkt an die jeweiligen Anschlüsse geleitet. Der im PLASA-Modell verbaute Fahrten-Regler lässt sich über einen Schalter ein- bzw. ausschalten. Über den PWR-Anschluss ist das DAB mit dem Schalter des Fahrten-Reglers verbunden und kann diesen per MOSFET ein- und ausschalten.

Der Hall-Effekt-Sensor, der für die Erkennung einer Rotation der Kurbelwelle benötigt wird, ist ebenfalls mit dem DAB verbunden. Der Sensor erfasst das Vorbeilaufen der Magneten an der Kurbelwelle und zeigt dies durch eine Flanke im Ausgangssignal an. Dieses Signal ist am Eingang ICP des Mikrocontroller angeschlossen. Über diesen Eingang wird im Mikrocontroller ein Interrupt ausgelöst, auf den der aktuelle Timer-Wert in einem Register gespeichert wird. Per Software-Steuerung wird im Anschluss der Timer-Wert ausgelesen und für die Berechnung der Geschwindigkeit verwendet. Der Rotationssensor kann per MOSFET-Steuerung vom ATmega1284P ein- und ausgeschaltet werden.

Das DAB bietet für drei weitere analoge Sensoren Anschlüsse, deren Ausgangssignale an die ADC-Eingänge des ATmega1284P verbunden sind. Alle drei Sensor-Anschlüsse können einzeln per MOSFET-Steuerung ein- und ausgeschaltet werden. In der bestehenden Konfiguration der PLASA-Plattform sind diese Anschlüsse nicht belegt und bieten somit eine Erweiterungsmöglichkeit der Plattform an.

Das Light and Sensor Board

Das LSB ist in der Karosserie des Modell-Fahrzeugs verbaut und ist für die Steuerung des Lichts und für das Auswerten der acht angeschlossenen Sensoren verantwortlich. Es sind sieben Abstandssensoren, die den Abstand zu anderen Objekten messen und außen an der Karosserie des Modell-Fahrzeugs befestigt sind. Daneben erfasst ein Helligkeitssensor die Helligkeit der Umgebung. Eine schematische Übersicht über das LSB ist in Abbildung 5.31 zu sehen.

Der ATmega1284P Mikrocontroller wird von der 5 V Bordnetz-Spannung versorgt, die er über den Bordnetz-Anschluss vom PSB erhält. Ebenfalls ist er über den Bordnetz-Anschluss mit dem I^2C-Bus verbunden. Wie schon beim PSB und DAB sind auch im LSB drei LEDs für die Statusanzeige verbaut, die der ATmega1284P für die Statusanzeige schalten kann.

Die Sensoren sind am LSB über acht Anschlüsse verbunden, die der ATmega1284P einzeln über eine MOSFET-Schaltung ein- und ausschalten kann. Die Ausgangsleitungen der Sensoren hängen jeweils an den acht ADC-Eingängen des ATmega1284P. Somit werden die Messwerte der analogen Sensoren vom ATmega1284P digitalisiert und anschließend von der Software-Steuerung weiter verarbeitet. An den acht Sensor-Anschlüssen

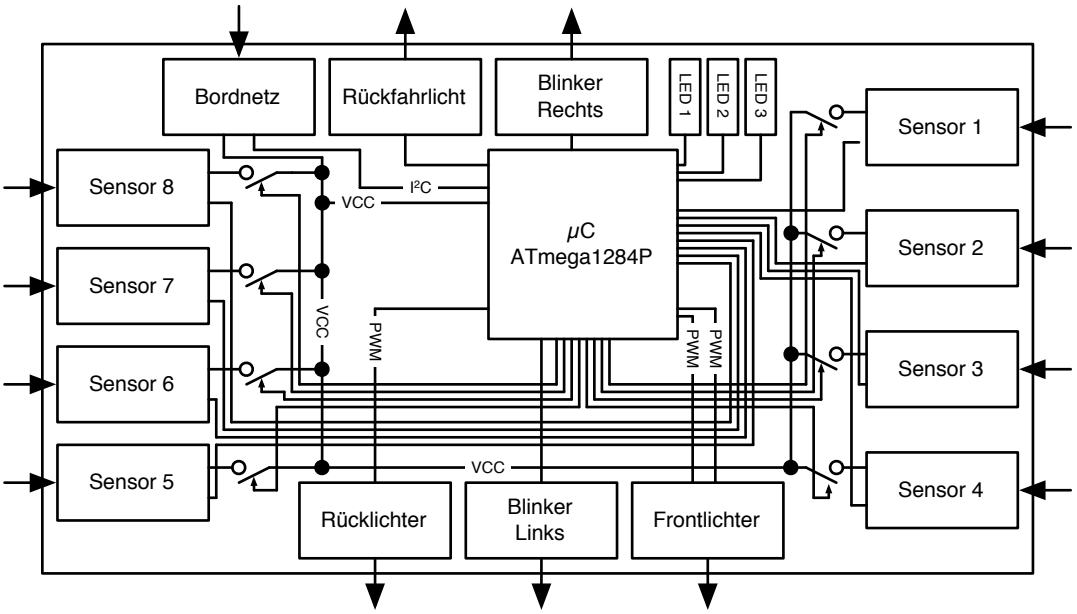


Abbildung 5.31: Schematische Übersicht über das LSB.

sind die sieben Abstandssensoren angeschlossen, die außen an der Karosserie angebracht sind. Am achten Sensor-Anschluss ist der Helligkeitssensor TSL257 [151] verbunden.

Neben der Auswertung der analogen Sensoren steuert das LSB die Lichter des Modell-Fahrzeugs. Es sind fünf verschiedene Scheinwerfer in das Modell-Fahrzeug integriert, die vom ATmega1284P unterschiedlich gesteuert werden. Die beiden Frontscheinwerfer können individuell über zwei PWM-Signale vom Mikrocontroller aus kontrolliert werden, womit beide Frontscheinwerfer unterschiedliche Leuchtstärken einnehmen können. Die Rücklichter werden ebenfalls über ein PWM-Signal gesteuert. Allerdings werden dabei beide Lichter mit dem selben PWM-Signal kontrolliert. Für die Blinker und das Rückfahrlicht werden keine PWM-Signale verwendet, sondern die Lichter können nur ein- und ausgeschaltet werden, womit diese dann nur in einer Helligkeit leuchten können.

Der Gumstix

Neben dem PSB, dem DAB und dem LSB, die sich um die Sensor-Auswertung und die Aktor-Steuerung kümmern, ist in der PLASA-Plattform noch der Gumstix verbaut, der die sogenannte Head-Unit in der Infotainment-Domäne nachstellt, wie es in der Beschreibung der E/E-Architektur in Abschnitt 2.2.1 dargestellt ist.

Der Gumstix wird von der 5 V Bordnetz-Spannung vom PSB versorgt und ist ebenfalls mit dem I^2C-Bus verbunden. Mit Hilfe der I/O-Pegelwandlung auf dem PSB kann er direkt mit dem PSB, DAB und LSB über den I^2C-Bus kommunizieren. Auf dem Gumstix ist ein 600 MHz getakteter OMAP3530 Prozessor von Texas Instruments verbaut. Als flüchtiger Speicher ist der Gumstix mit 256 MB DDR RAM ausgestattet und besitzt zusätzlich 256 MB persistenten Flash Speicher für Programme und Daten. Zusätzlich ist

der Gumstix mit einem MicroSD Karten-Slot ausgestattet, was erlaubt, den persistenten Speicher mit einer MicroSD-Karte zu erweitern. Der Gumstix ist für mobile Embedded Systeme konzipiert worden. Jedoch ist der OMAP3530 mit der Klasse von automotiven Prozessoren, die in Head-Units heutiger Fahrzeuge verbaut werden, vergleichbar.

Auf dem Gumstix ist ebenfalls der TPS65950 Power-Management Chip von Texas Instruments verbaut, der passend zum OMAP3530 dessen Power-Management übernimmt, worunter auch das sogenannte *Dynamic Voltage and Frequency Scaling* fällt.

Für die drahtlose Kommunikation ist der Wi2Wi W2CBW003C WiFi/BT Transceiver verbaut, der sowohl WLAN IEEE-802.11 b/g [82] als auch Bluetooth 2.0+EDR [31] unterstützt. Dies bietet die Möglichkeit, dass das Fahrzeug drahtlos mit anderen Geräten oder Rechnern verbunden werden kann. Über eine WLAN-Verbindung können andere Rechner mit dem Gumstix eine Verbindung aufbauen, um darüber Informationen auszutauschen oder das Auto steuern. Vor allem die Bluetooth-Schnittstelle wird dazu genutzt, die Nintendo Wii Remote als Sensor für die PLASA-Plattform zu verwenden und für die Steuerung des Modell-Fahrzeugs einzusetzen.

5.4.3 Deployment und Hardware/Software Codesign der PLASA-Plattform

In den vorhergehenden Abschnitten wurde die logische Architektur, die Software-Architektur und die Hardware-Architektur der PLASA-Plattform vorgestellt. Um die Blöcke der logischen Architektur auf der Hardware auszuführen, wurde das Konzept der Tasks im Abschnitt 5.3.1 vorgestellt. Die Tasks sind Software-Komponenten, die die Funktionalität der Blöcke aus der logischen Architektur kapseln. Nun müssen die Komponenten der logischen Architektur in Form der ausführbaren Software-Komponenten den verschiedenen ECUs zugewiesen werden. Durch die Integration der Software-Komponenten auf die einzelnen ECUs ergibt sich die technische Architektur der PLASA-Plattform.

Dabei wurde bei der Verteilung der logischen Architektur wie folgt vorgegangen. Da Sensoren und Aktoren drahtgebunden an eine bestimmte ECU angeschlossen sind, werden die Sensor- und Aktor-Komponenten aus der logischen Architektur immer auf die ECU partitioniert, an der Sensor oder Aktor auch physikalisch angeschlossen sind.

Somit sind die ECUs für die Sensoren und Aktoren schon festgelegt und es müssen noch die Funktionsblöcke der logischen Architektur den ECUs zugewiesen werden. Hier erfolgt die Partitionierung der Blöcke primär nach deren Zugehörigkeit der Funktionen zu Sensoren oder Aktoren. Somit erfolgt der erste verarbeitende Schritt der Sensor-Daten auf der ECU, an der der Sensor auch angeschlossen ist. Oder der letzte Verarbeitungsschritt der Aktor-Steuerung wird auf der ECU durchgeführt, an der der Aktor angeschlossen ist. Das Ergebnis dieser Partitionierung ist die Reduzierung der Bus-Last, da die Vorverarbeitung der Sensor-Daten meist zu einer Reduzierung der Nachrichtengröße führt. Dies gilt ebenfalls für die Aktor-Steuerung, wenn der regelnde Funktionsblock auf der ECU ausgeführt wird, an der der Aktor auch angeschlossen ist.

Somit ist die Partitionierung der logischen Architektur auf die Hardware-Architektur durch reine Ingenieursleistung erfolgt und nicht durch eine mathematische Optimierung nach energetischen Gesichtspunkten. In der PLASA-Plattform wurden ebenfalls keine Alternativen für eine Software-Partitionierung untersucht. Somit wurde das DPM für Automotive Systeme nur mit einer Verteilung untersucht, die zusammenfassend in Ta-

ECU	Sensor	Funktionsblock	Aktor
Gumstix	SEN_WII_BUTTONS SEN_WII_ACCEL	FCN_CAR_STATE FCN_MANUAL_MOTOR FCN_MANUAL_SERVO	
PSB	SEN_BATT_V	FCN_BATT_CTRL	
DAB	SEN_ODO SEN_SPEED	FCN_SPEED_SET FCN_SPEED_CTRL FCN_DRIVE_SEQUENCE	ACT_MOTOR ACT_SERVO
LSB	SEN_DIST_FL SEN_DIST_FC SEN_DIST_FR SEN_DIST_RL SEN_DIST_RC SEN_DIST_RR SEN_DIST_S SEN_LIGHT_INTENS	FCN_FIND_PARKING_SPACE FCN_PARKING_SEQUENCE FCN_PARKING_END FCN_OBSTACLE_F FCN_OBSTACLE_R FCN_EMERG_STOP	ACT_LIGHT_FL ACT_LIGHT_RL ACT_LIGHT_BL ACT_LIGHT_RVL

Tabelle 5.8: Das Deployment der logischen Komponenten auf die ECUs der PLASA-Plattform.

belle 5.8 dargestellt ist. Jedoch ist es aufgrund der Plattform-unabhängigen Gestaltung der TinyOS-Komponenten möglich, die Software-Komponenten auf den ECUs anders zu verteilen, um weitere Patitionierungsmöglichkeiten zu erhalten und diese nach energetischen Gesichtspunkten zu untersuchen.

Nach der hier vorgestellten Partitionierung der logischen Architektur für die PLASA-Plattform erhalten die ECUs folgende Aufgaben. Der Gumstix erhält die Aufgabe einer Head-Unit, die das zentrale Steuergerät in dem Fahrzeug darstellt. Er ist die Schnittstellen zur Außenwelt und steuert die übrigen ECUs im Fahrzeug. In der PLASA-Plattform verbindet er sich mit der Nintendo Wii Remote, nimmt deren Sensor-Werte auf und steuert damit die Funktionen des Modell-Fahrzeugs über Nachrichten. Die drei weiteren ECUs haben die Aufgabe die Sensor-Daten zu ermitteln und die angeschlossenen Aktoren zu steuern. So überwacht das PSB die Batterie und versendet bei zu niedriger Batterie-Spannung die entsprechende Nachricht. Das DAB ist hauptsächlich für die beiden Aktoren für die Fortbewegung zuständig. So steuert es den Lenk-Servo und den Fahrten-Regler des Motors an. Auch die dazugehörigen Funktionen wie Geschwindigkeitsregler oder Fahrsequenz-Regelung übernimmt das DAB. Die dafür notwendigen Sensor-Werte für die Geschwindigkeit oder das Hodometer sind ebenfalls auf dem DAB partitioniert. Das LSB ist für die Sensoren an der Karosserie, wie die Abstandssensoren und den Helligkeit-Sensor zuständig. Aus den berechneten Werten für die Abstände werden Funktionen für das automatische Einparken und für den Nothalt ausgeführt. Ebenfalls wird die Fahrsequenz für das Einparken auf dem LSB berechnet und die einzelnen Segmente werden an das DAB gesendet, das im Anschluss diese über die Fahrsequenz-Regelung abfährt. Eine weitere Funktion des LSBs ist die Steuerung der angeschlossenen Lichter. Über die gemessene Helligkeit der Umgebung wird die Helligkeit der LEDs berechnet und diese über PWM-Signale gesteuert.

5.5 Zusammenfassung und Bewertung der PLASA-Plattform

Die PLASA-Plattform wurde mit Hilfe der verschiedenen Abstraktionsebenen, die in Abschnitt 2.2.2 beschrieben sind, nach den Methoden des Software-Engineerings für Automotive Systeme entwickelt. Während die gesamten Anwendungsfälle zu Beginn des Kapitels kurz informell beschrieben sind, liegt ein Fokus in der Beschreibung der logischen Architektur, wobei hier exemplarisch die Aktoren zur Steuerung und Fortbewegung des Modell-Fahrzeugs herangezogen wurden. Die logische Architektur als eine der Abstraktionsebenen bildet die Basis für das DPM der PLASA-Plattform, weshalb sie so ausführlich für die PLASA-Plattform definiert wurde. Bei der Abstraktionsebene der Software-Architektur steht TinyOS im Vordergrund, das auf dem PSB, DAB und LSB zum Einsatz kommt. Hier wird erläutert, wie die automotiven Funktionen wie die Steuerung des Modell-Fahrzeugs in TinyOS realisiert und wie das DPM in der PLASA-Plattform umgesetzt ist. In der Abstraktionsebene der Hardware-Architektur werden der Gumstix, das PSB, das DAB und das LSB, die über den I^2C-Bus als Fahrzeug-Bus miteinander verbunden sind, als ECUs der PLASA-Plattform mit ihren elektromagnetischen Komponenten beschrieben.

In der Software-Architektur der PLASA-Plattform wird erläutert, wie die logische Architektur auf die Software-Architektur unter TinyOS abgebildet wird. Hierfür werden die Komponenten aus der logischen Architektur auf sogenannte Software-Tasks abgebildet, die als PMCs vom PM auf ECU-Ebene unter TinyOS gesteuert werden. Ein weiterer Fokus liegt auf dem PM auf ECU-Ebene, wie er in der TinyOS-Architektur verankert ist und wie er die TinyOS-Konzepte für das Power-Management nutzt, um damit die Hardware- und Software-PMCs zu steuern. Der PM leitet dabei aus den Nachrichten, die zwischen den Software-Tasks auch über ECU-Grenzen hinweg versendet werden, den Fahrzeug-Zustand ab und entscheidet auf dieser Basis, welchen Power-Modus die PMCs einnehmen sollen. Mit diesem Konzept wurde ein DPM für die PLASA-Plattform umgesetzt, das allerdings nicht hierarchisch organisiert ist, wie es in Kapitel 4 über das DPM für Automotive Systeme erklärt wurde. Der Grund ist, dass die PLASA-Plattform nur aus einer Domäne mit vier ECUs besteht. Allerdings hat dies für das Konzept des PMs auf ECU-Ebene aus folgendem Grund keine Auswirkung: Da der PM auf ECU-Ebene auf Nachrichten, die zwischen den Software-Tasks versendet werden, hört und daraus die Power-Modi der PMCs bestimmt, würde bei der Einführung eines PMs auf Domänen-Ebene keine grundsätzliche Änderungen im PM auf ECU-Ebene ergeben. Denn der PM auf Domänen-Ebene würde als Software-Task konzipiert werden, der wie jeder andere Software-Task Nachrichten für die Änderung des Fahrzeug-Zustands versenden würde. Damit der PM auf ECU-Ebene diese Nachrichten berücksichtigt, müsste nur die PM-Wissensbasis angepasst werden, damit die Nachrichten des PMs auf Domänen-Ebene in die Bestimmung des Fahrzeug-Zustands lokal miteinfließen kann.

Während der Realisierung der PLASA-Plattform und des PMs auf ECU-Ebene hat sich gezeigt, dass sich TinyOS als Software-Basis für das PSB, DAB und LSB und die Untersuchung des DPMs für Automotive Systeme bewährt hat, obwohl es kein Betriebssystem für Automotive Systeme ist, sondern aus der benachbarten Domäne der WSNs stammt. Folgenden Gründe tragen u. a. dazu bei:

Kleines Software-System. TinyOS ist für Mikrocontroller mit wenigen KBs an Programm- und Datenspeicher ausgelegt und bietet dadurch ein kleines und überschaubares Software-System als Grundlage für die Untersuchung eines DPMs an, in dem die Software-PMCs durch eine eigens definierte Logik gesteuert werden können. Im Vergleich zu einem Linux-System, in dem viele Tasks zu einem Zeitpunkt für die Funktionserfüllung laufen, können in TinyOS alle Software-Tasks kontrolliert werden und besitzen nur wenige Abhängigkeiten zu bereits existierenden Software-Komponenten, die sich unter Umständen nicht kontrollieren lassen.

Komponenten-basiertes Betriebssystem. TinyOS ist als Komponenten-basiertes Betriebssystem leicht um Schnittstellen und Komponenten, die eine neue Funktionalität mitbringen, erweiterbar. Deshalb konnten die fehlenden automotiven Funktionen ohne Probleme in das System aufgenommen werden. Die in TinyOS definierte HAA ermöglichte es, bereits in TinyOS vorhandene Software-Komponenten wieder zu verwenden. Darunter fällt u. a. für das Auslesen der analogen Sensoren die Bedienung des ADCs oder für die Steuerung des Lenk-Servos und des Fahrten-Reglers die Timer-Steuerung zur Generierung der PWM-Signale.

Eignung für Mikrocontroller. Da für die automotiven Funktionen wie die Steuerung des Lenk-Servos oder des Fahrten-Reglers Mikrocontroller benötigt werden, hat sich TinyOS mit seinen Konzepten und Prinzipien sehr bewährt. Zusätzlich bietet TinyOS bereits Unterstützung für die in der PLASA-Plattform verwendeten Mikrocontroller ATmega644P und ATmega1284P an.

Konzepte für niedrigen Energiebedarf. Im TinyOS-Design wird bereits auf einen niedrigen Energiebedarf geachtet und dessen Power-Management ist von Grund auf dafür ausgelegt. Das Konzept, bei dem die Hardware-PMCs in den niedrigsten möglichen Power-Modus geschaltet werden, sobald kein Task mehr lauffähig ist, hat sich als eine solide Grundlage für die Umsetzung des PMs auf ECU-Ebene für die PLASA-Plattform erwiesen und konnte unverändert wiederverwendet werden.

Ähnlichkeit mit automotiven Software-Architekturen. Ein Vergleich der Software-Architektur und der vorhandenen Software-Komponenten mit der des automotiven Software-Standards AUTOSAR zeigt, dass sich TinyOS in seiner grundsätzlichen Architektur nur wenig unterscheidet. Dadurch können die aus der PLASA-Plattform erlangten Erkenntnisse leichter auf ein Software-System für Automotive Systeme übertragen werden, was im Ausblick in Abschnitt 7.3 noch vertieft wird.

Obwohl TinyOS eine bewährte Basis für die PLASA-Plattform bietet, ist es allerdings nur für weiche Echtzeit-Systeme ausgelegt, was für die Domäne der WSNs ausreicht. Die Realisierung in der PLASA-Plattform zeigt, dass damit auch ein Automotives System betrieben werden kann. Allerdings ist TinyOS für sicherheitskritische Systeme mit harten Echtzeit-Anforderungen ohne zusätzliche Erweiterungen keine Alternative, da es bisher keine Konzepte für die Einhaltung von harten Deadlines bietet. Da TinyOS als Komponenten-basiertes Betriebssystem leicht erweiterbar ist, muss noch untersucht werden, ob durch die Einführung von zusätzlichen Software-Komponenten die Erfüllung von harten Echtzeit-Anforderungen in TinyOS möglich ist.

Kapitel 6

Messungen an der PLASA-Plattform

Nachdem im vorherigen Kapitel die PLASA-Plattform mit deren DPM für Automotive Systeme beschrieben wurde, werden in diesem Kapitel die Auswirkungen des DPMs auf den Stromverbrauch der ECUs dargestellt. Dafür wird die PLASA-Plattform um ein Mess-System erweitert, das den Stromverbrauch des DABs, des LSBs und des Gumstix misst. Die Messungen dienen zur Auswertung und zum Nachweis, dass die Konzepte des DPMs in der PLASA-Plattform korrekt funktionieren und dass diese im Stromverbrauch der ECUs beobachtbar sind. Dabei steht der Nachweis über das korrekte Funktionieren des DPMs in der PLASA-Plattform im Vordergrund und nicht wie viel Energie im Detail damit eingespart werden kann. Aus diesem Grund wird auf eine Analyse der Messgenauigkeit verzichtet und nur der Aufbau des Mess-Systems mit den Hardware-Komponenten vorgestellt.

Um den Stromverbrauch der ECUs in der PLASA-Plattform während der Fahrt des Modell-Fahrzeugs aufzuzeichnen, wird das Mess-System komplett in das Modell-Fahrzeug integriert. Der Aufbau des Mess-Systems wird im folgenden Abschnitt erläutert, wobei zuerst der Hardware-Aufbau des Mess-Systems mit den Strom-Sensoren und der Mess-Platine beschrieben wird. Im Anschluss werden die einzelnen Schritte für die Messwert-Erfassung und deren Auswertung dargestellt.

Im darauffolgenden Abschnitt 6.2 werden die Ergebnisse der Messungen aus der PLASA-Plattform vorgestellt. Zuerst werden Referenz-Messungen präsentiert, die am DAB und LSB durchgeführt wurden. Darunter fallen die Vermessungen der verschiedenen Power-Modi der Mikrocontroller und der externen Verbrauchen, wie die Abstandssensoren oder die Beleuchtung des Modell-Fahrzeugs. Im Anschluss wird der Stromverbrauch der ECUs während der Anwendungsfälle der PLASA-Plattform, die in Abschnitt 5.1 beschrieben wurden, gezeigt. Um die während der Fahrmanöver durchgeführten Strommessungen besser interpretieren zu können, helfen die zuvor durchgeführten Referenz-Messungen.

6.1 Aufbau des Mess-Systems

Bevor jedoch auf die einzelnen Messergebnisse eingegangen wird, wird im Folgenden der Aufbau des Mess-Systems beschrieben. Neben dem Einbau der Strom-Sensoren und der Mess-Hardware in die PLASA-Plattform ist ein Software-System entstanden, das die Messwerte, die während der Fahrmanöver erfasst werden, in einer Datenbank speichert. Diese Aufzeichnungen sind die Grundlage für die weiteren Auswertungen, die im Anschluss auf einem zweiten Desktop-System offline durchgeführt werden.

Bei der Konzeption des Mess-Systems wurde darauf geachtet, dass das gesamte Mess-System in das Modell-Fahrzeug integriert werden kann. Damit ist es möglich, den Stromverbrauch der ECUs im fahrendem Betrieb zu untersuchen und das DPM im Modell-Fahrzeug unter realen Bedingungen und nicht in einer simulierten Umgebung zu vermessen.

Um die Messergebnisse nicht zu verfälschen, wurde das Mess-System so ausgelegt, dass es so wenig wie nötig mit dem Fahrzeug-Bordnetz verbunden ist. Neben den Strom-Sensoren, die am Strom-Eingang der ECUs hängen, wird das Mess-System ebenfalls aus der Fahrzeug-Batterie versorgt, um nicht eine zweite Batterie im Modell-Fahrzeug zu verbauen. Des Weiteren ist die Mess-Platine an den Fahrzeug-Bus angeschlossen, um die Nachrichten auf dem Fahrzeug-Bus zu empfangen und aufzuzeichnen. Damit können die Nachrichten bei der späteren Auswertung mitberücksichtigt werden.

Durch diese geringe Vernetzung des Mess-Systems mit der PLASA-Plattform, kann das Mess-System bei Bedarf leicht aus dem Modell-Fahrzeug entfernt werden, um die PLASA-Plattform als Demonstrator für weitere Modell-Versuche ohne eine Messung des Stromverbrauchs zu nutzen.

Im nächsten Abschnitt wird im Detail auf die Hardware-Erweiterungen des Mess-Systems eingegangen, die für die Messung des Stromverbrauchs an den einzelnen ECUs benötigt werden. Darauffolgend werden die notwendigen Software-Erweiterungen beschrieben, die für die Speicherung der Messwerte in einer Datenbank und deren Auswertung notwendig sind.

6.1.1 Hardware-Aufbau des Mess-Systems

Der Hardware-Aufbau des Mess-Systems für die PLASA-Plattform ist in einer studentischen Arbeit entstanden [165], in der das Konzept des Mess-Systems entwickelt und im Anschluss realisiert wurde. Um den Stromverbrauch der einzelnen ECUs zu messen, müssen Strom-Sensoren in die Versorgungsleitungen der ECUs integriert werden. Deren Messwerte werden in der Mess-Platine mit Hilfe eines ADCs digitalisiert und aufgezeichnet.

Abbildung 6.1 gibt eine Übersicht über die notwendigen Erweiterungen des Systems gegenüber der ursprünglichen PLASA-Plattform, die in Abschnitt 5.4 vorgestellt wurde. An dem DAB, dem LSB und dem Gumstix sind zusätzliche Strom-Sensoren installiert, die den Durchfluss-Strom zu der jeweiligen ECU messen. Der Strom wird hierbei in eine Spannung konvertiert, die ein ADC auf der Mess-Platine digitalisiert. Um den Fahrzeug-Zustand zu den Messergebnissen der Strom-Sensoren zu speichern, ist die Mess-Platine ebenfalls am Fahrzeug-Bus angeschlossen, damit die Nachrichten auf dem Bus verfolgt

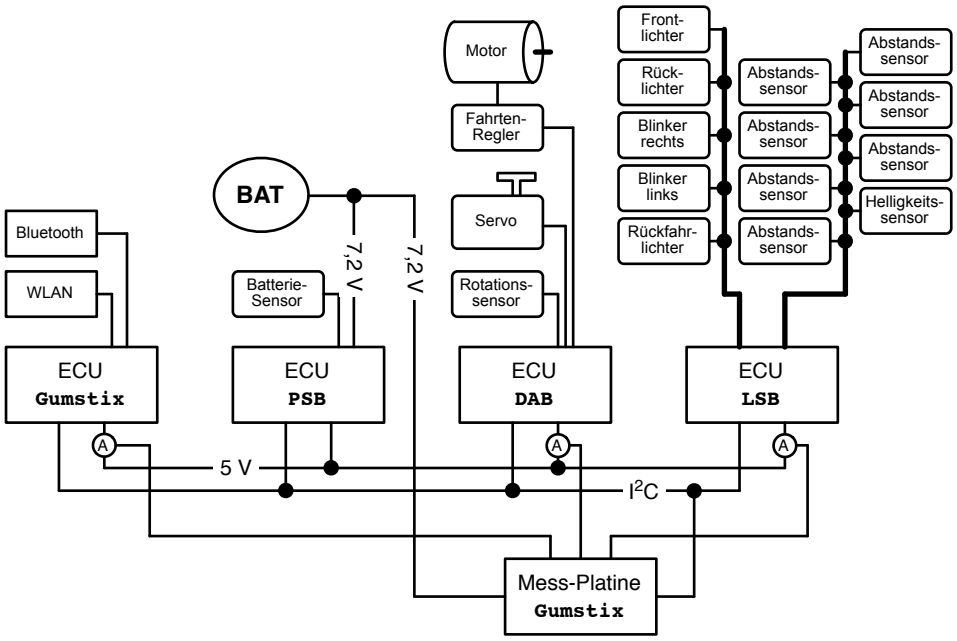


Abbildung 6.1: Die PLASA-Plattform mit dem integrierten Mess-System.

werden können.

Die Strom-Sensoren sind durch sogenannte Shunt-Widerstände gelöst. An diesem fällt nach dem Ohm'schen Gesetz je nach Stromstärke eine Spannung ab, die im Anschluss auf der Mess-Platine verstärkt und von einem ADC digitalisiert wird. Bei den Shunt-Widerständen liegt das Problem, dass die dort abfallende Spannung nicht zu hoch sein darf, da ansonsten die jeweilige ECU nicht mit der benötigten Versorgungsspannung versorgt wird. Im Gegensatz dazu fällt bei einem zu niedrig gewählten Shunt-Widerstand nur eine geringe Spannung ab, was eine genaue Messung erschwert. Somit muss der Shunt-Widerstand passend zu der fließenden Stromstärke gewählt werden.

Jedoch schwanken die Stromstärken bei der Untersuchung eines DPMs sehr stark, da sich die Strom-Aufnahmen der Mikrocontroller in ihren verschiedenen Power-Modi sehr stark unterscheiden. Als Beispiel unterscheidet sich die Strom-Aufnahme des Atmel ATmega1284P Mikrocontroller nach dessen Spezifikation bis zu einem Faktor 1000 zwischen dem aktiven Zustand und dem Power-Modus *Power-Down* [7]. Somit muss die Wahl des Shunt-Widerstands gut auf die jeweilige ECU und deren Mikrocontroller abgestimmt sein. In Tabelle 6.1 sind die Parameter mit den Widerstandsgrößen für das DAB, das LSB und den Gumstix zusammengefasst. Dabei wurden die Stromstärken, die jede ECU benötigt, aus den Datenblätter der jeweiligen Komponenten entnommen und der Gesamtverbrauch für die Bestimmung des Shunt-Widerstands geschätzt. Je nach geschätzter Stromstärke wird zusammen mit dem Widerstand die abfallende Spannung ΔV berechnet.

Aus dem Widerstandssortiment stehen 0,15 Ω, 0,22 Ω oder 1 Ω Widerstände zur Auswahl. Es wird eine Spannungstoleranz beim DAB und beim LSB von 100 mV und beim Gumstix von 150 mV angenommen. Somit lässt sich aus der Tabelle der für jede ECU am

ECU	Strom		Shunt-Widerstand	ΔV	
	min	max		min	max
			1 Ω	0,1 mV	100 mV
DAB	0,1 mA	100 mA	**0,22 Ω**	**0,022 mV**	**22 mV**
			0,15 Ω	0,015 mV	15 mV
			1 Ω	0,1 mV	400 mV
LSB	0,1 mA	400 mA	**0,22 Ω**	**0,022 mV**	**88 mV**
			0,15 Ω	0,015 mV	60 mV
			1 Ω	100 mV	1000 mV
Gumstix	100 mA	1000 mA	0,22 Ω	22 mV	220 mV
			0,15 Ω	**15 mV**	**150 mV**

Tabelle 6.1: Bestimmung der Shunt-Widerstände für die Strom-Messung an den ECUs der PLASA-Plattform.

besten passender Shunt-Widerstand ermitteln, der in der Tabelle 6.1 fett gekennzeichnet ist und in dem Mess-System der PLASA-Plattform verwendet wird. Aus der Tabelle wird ebenfalls ersichtlich, dass für die verschiedenen ECUs aufgrund der verschiedenen Stromaufnahmen unterschiedliche Shunt-Widerstände eingesetzt werden.

Die Mess-Platine, an der die Shunt-Widerstände angeschlossen werden, besitzt folgenden Aufbau, der in Abbildung 6.2 übersichtlich dargestellt ist. Als Komponente ist der DC/DC-Spannungswandler LT1767 [103], der aus der Batterie-Spannung des Modell-Fahrzeugs von 7,2 V die benötigte Versorgungsspannung der Mess-Platine von 5 V erzeugt, verbaut. Daraus wird auch ein zusätzlicher Gumstix versorgt, der über einen Anschluss-Konnektor mit der Mess-Platine verbunden ist. Die Eingangssignale von den Shunt-Widerständen werden von dem Operationsverstärker AD8630 [5] um den Faktor 16,5 verstärkt und in den ADC AD7682 [4] geleitet. Der ADC wird über den SPI-Bus

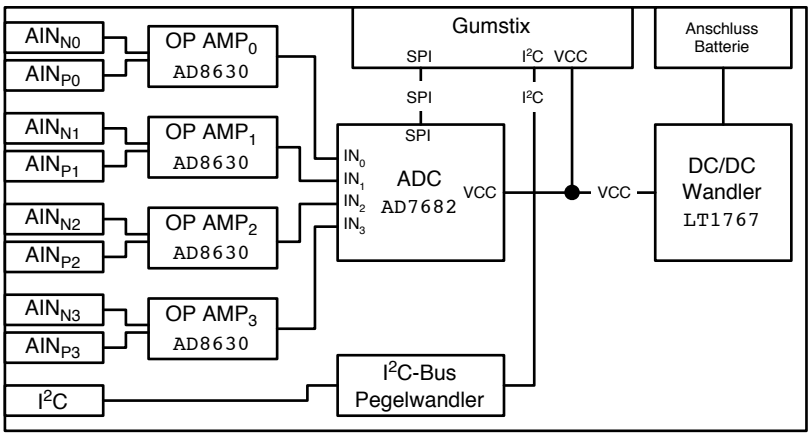


Abbildung 6.2: Schematische Übersicht über die Mess-Platine.

vom Gumstix gesteuert. Somit ist der Gumstix für das Auslesen des ADCs und das Speichern der Messwerte in einer Datenbank verantwortlich, was im nächsten Abschnitt im Detail beschrieben wird. Zusätzlich werden die Bus-Nachrichten der PLASA-Plattform über den Pegelwandler für den I^2C-Bus in den Gumstix der Mess-Platine geleitet.

6.1.2 Mess-Software

Die Mess-Software ist für die Aufzeichnung und die Aufbereitung der Messergebnisse verantwortlich. Dabei unterteilt sich der gesamte Prozess der Messung mit deren Auswertung in mehrere Teilschritte, die in Abbildung 6.3 dargestellt sind. Der Prozess lässt sich in zwei Teilabschnitte gliedern. Zuerst erfolg die Messwert-Erfassung im Modell-Fahrzeug, in dem die Messungen an den Strom-Sensoren durchgeführt und zusammen mit den Fahrzeug-Nachrichten in einer Datenbank abgespeichert werden. Im zweiten Schritt erfolgt die Mess-Auswertung der zuvor gesammelten Daten. Dazu wird die Datenbank auf einem Desktop-Rechner für eine schnellere und einfachere Auswertung übertragen. Die Auswertung teilt sich dabei wieder in zwei Schritte auf. Zuerst werden nur die benötigten Daten für die konkrete Auswertung aus der Datenbank entnommen und in einer Daten-Auswahl gespeichert. Mit diesen ausgewählten Messergebnissen werden im zweiten Schritt die Diagramme erzeugt, die im Abschnitt 6.2 als Messergebnisse abgebildet sind.

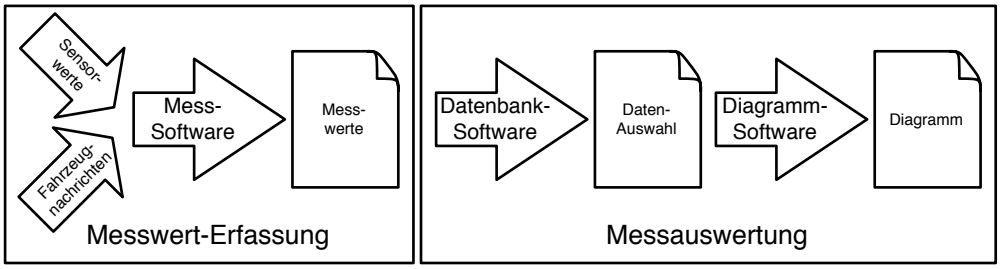


Abbildung 6.3: Der Ablauf der Messauswertung.

Zuerst müssen die Messwerte zentral zusammen mit den Fahrzeug-Nachrichten in einer Datenbank abgelegt werden, während das Modell-Fahrzeug die verschiedenen Anwendungsfälle ausführt. Dafür läuft auf dem Gumstix, an dem die Mess-Platine angeschlossen ist, eine spezielle Mess-Software, die in zwei studentischen Arbeiten entwickelt wurde. Die Ansteuerung des ADCs wurde zusammen mit der Konzeption der Mess-Platine durchgeführt [165]. Darauf aufbauend wurde die Mess-Software um die Erfassung der Fahrzeug-Nachrichten in einer weiteren studentischen Arbeit erweitert [120].

Für die Steuerung des ADCs AD7682 müssen spezifische Kommandos über den SPI-Bus gesendet werden. Die Messergebnisse des ADCs werden ebenfalls über den SPI-Bus an den Gumstix zurückgeschickt. Dafür nutzt die Mess-Software die Treiber-Komponente für den SPI-Bus aus dem Linux-Kernel, über den sie die Kommandos an den AD7682 schickt. Die darauf hin empfangenen Messwerte werden zusammen mit den Zeitstempeln in der Datenbank abgelegt.

Neben den Messwerten speichert die Mess-Software auch die Bus-Nachrichten, die

über den Fahrzeug-Bus gesendet werden. Dafür nutzt die Mess-Software ebenfalls eine im Linux-Kernel enthaltene Treiber-Komponente, um die I^2C-Bus-Nachrichten zu empfangen, die wiederum mit einem Zeitstempel in der Datenbank abgelegt werden.

Als Datenbank kommt die Quell-offene SQL-Datenbank SQLite [142] zum Einsatz, die als Bibliothek in die Mess-Software integriert ist. SQLite bietet für diesen hier benötigten Anwendungsfall den Vorteil, dass die Datenbank in einer gewöhnlichen Datei gespeichert wird. Dies ermöglicht eine Übertragung der Datei auf einen weiteren Rechner, von dem aus die Datenbank ausgelesen und ausgewertet werden kann. Neben der Implementierung von SQLite als Bibliothek existiert ebenfalls eine Implementierung der Datenbank als Terminal-Programm, das für die Auswertung der Messergebnisse genutzt wird.

Die Erzeugung der Diagramme und die Auswertung der aufgezeichneten Daten erfolgt in zwei Schritten. Als erstes werden die gewünschten Messwerte mit Hilfe von SQL-Anfragen aus der Datenbank extrahiert. Dabei ist es möglich, die Messwerte aus verschiedenen Aufzeichnungsläufen auszuwählen und mit Hilfe des SQLite Terminal-Programms in einer CSV-Datei (engl. *comma-separated values*) abzulegen, die die Eingabe-Werte für das Zeichnen der Diagramme enthält. Zum Zeichnen der Diagramme wird gnuplot [58] genutzt, das die zuvor generierten CSV-Dateien als Eingabe für die Graphen verarbeitet. Über Konfigurationsdateien lassen sich mehrere CSV-Dateien als Eingabe auswählen und die Parameter für die Diagramme setzen. Daneben werden zusätzliche Skripts verwendet, um den Stromverbrauch in mehrere Abschnitte zu unterteilen und für jeden Abschnitt einzeln den Durchschnittsverbrauch zu berechnen.

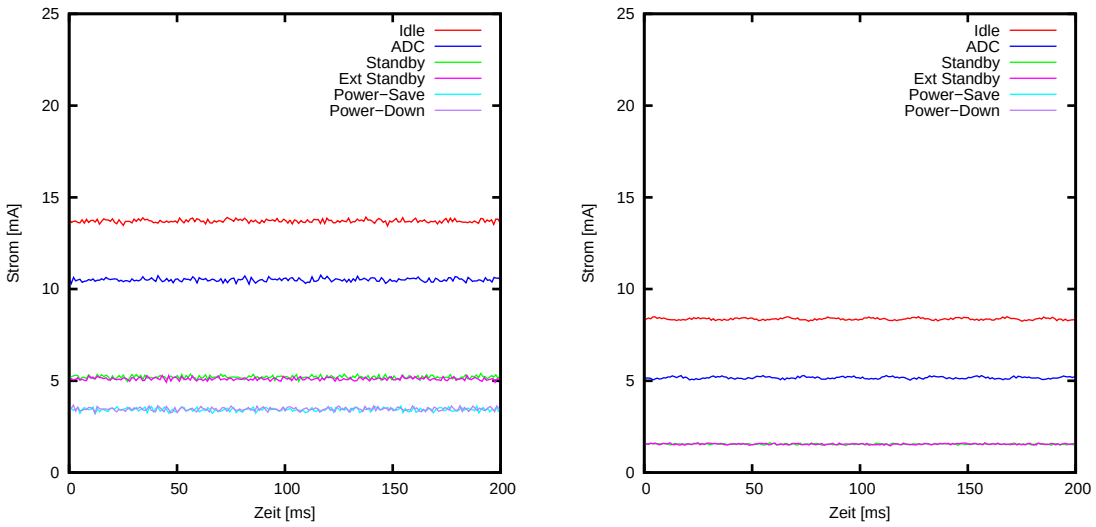
6.2 Messergebnisse aus der PLASA-Plattform

Nachdem die Messwert-Erfassung und die Erzeugung der Diagramme beschrieben wurde, werden im Folgenden die Messergebnisse aus der PLASA-Plattform vorgestellt. Dabei werden als Erstes Referenz-Messungen von dem DAB und dem LSB gezeigt, die notwendig sind, um den Stromverbrauch während der Anwendungsfälle richtig zu interpretieren. Im Anschluss daran werden die Messergebnisse während der Anwendungsfälle beschrieben.

6.2.1 Referenz-Messungen

Bei den Referenz-Messungen wird das DAB und das LSB einzeln ohne eine Integration in das Modell-Fahrzeug vermessen. Dabei werden die ECUs mit einer Test-Applikation programmiert, die den Mikrocontroller in den entsprechenden Power-Modus fährt oder die Peripherie entsprechend der gewünschten Messung schaltet. Somit lässt sich der Stromverbrauch der entsprechenden Hardware-Komponenten ermitteln. Im Fokus der Messungen steht der Mikrocontroller ATmega1284P mit seinen verschiedenen Power-Modi, die LEDs des Modell-Fahrzeugs sowie die Abstandssensoren. Diese Referenz-Messungen sind wichtig, um im Anschluss die Stromverbrauchskurven während der Anwendungsfälle der PLASA-Plattform besser interpretieren zu können.

Als Erstes wird der unterschiedliche Stromverbrauch des ATmega1284P in seinen verschiedenen Power-Modi ermittelt. Diese Messungen wurden sowohl am DAB als auch am



(a) Referenz-Messung am DAB.

(b) Referenz-Messung am LSB.

Abbildung 6.4: Stromverbrauch des ATmega1284P in den verschiedenen Power-Modi.

LSB durchgeführt. Für diese Testzwecke wurde der Mikrocontroller mit einer Applikation programmiert, die den Mikrocontroller jeweils die Power-Modi *Idle*, *ADC*, *Standby*, *Extended Standby*, *Power-Save* und *Power-Down* einnehmen und in diesen Modi verharren lässt. Die Power-Modi des ATmega1284P wurden in Abschnitt 4.3.1 bei der Beschreibung als Hardware-PMC vorgestellt. Die restlichen Hardware-Komponenten sind sofern möglich ausgeschaltet.

Die Ergebnisse der Messungen für das DAB sind in Abbildung 6.4a dargestellt. Im Modus *Power-Save* und im Modus *Power-Down* des ATmega1284P hat das DAB einen durchschnittlichen Stromverbrauch von 3,4 mA bzw. von 3,5 mA. Wenn der ATmega1284P den Power-Modus *Extended Standby* bzw. *Standby* einnimmt, steigert sich dieser auf 5,1 mA bzw. auf 5,2 mA. Sobald der ATmega1284P im Power-Modus *ADC* betrieben wird, wächst der Stromverbrauch auf das doppelte an und bewegt sich durchschnittlich bei 10,6 mA. Falls sich der ATmega1284P im *Idle* Modus schlafen legt, benötigt das DAB durchschnittlich 13,7 mA. Im aktiven Modus hat das DAB je nach Applikation einen höheren Stromverbrauch, was mit den Anwendungsfällen im nächsten Abschnitt beschrieben wird.

Die Ergebnisse am LSB, in denen der ATmega1284P die unterschiedlichen Power-Modi einnimmt, sind in Abbildung 6.4b zu sehen. Auffällig ist, dass der Stromverbrauch des LSBs unter die Messgenauigkeit des Mess-Systems fällt, wenn sich der ATmega1284P im *Power-Down* oder im *Power-Save* Modus schlafen legt. Im Power-Modus *Extended Standby* und *Standby* benötigt das LSB durchschnittlich 1,6 mA. Der durchschnittliche Stromverbrauch steigert sich auf 5,2 mA, sobald der Power-Modus *ADC* eingenommen wird. Legt sich der ATmega1284P im Power-Modus *Idle* schlafen, verbraucht das LSB durchschnittlich 8,4 mA.

Vergleicht man den Stromverbrauch der beiden ECUs, so ist auffällig, dass der durch-

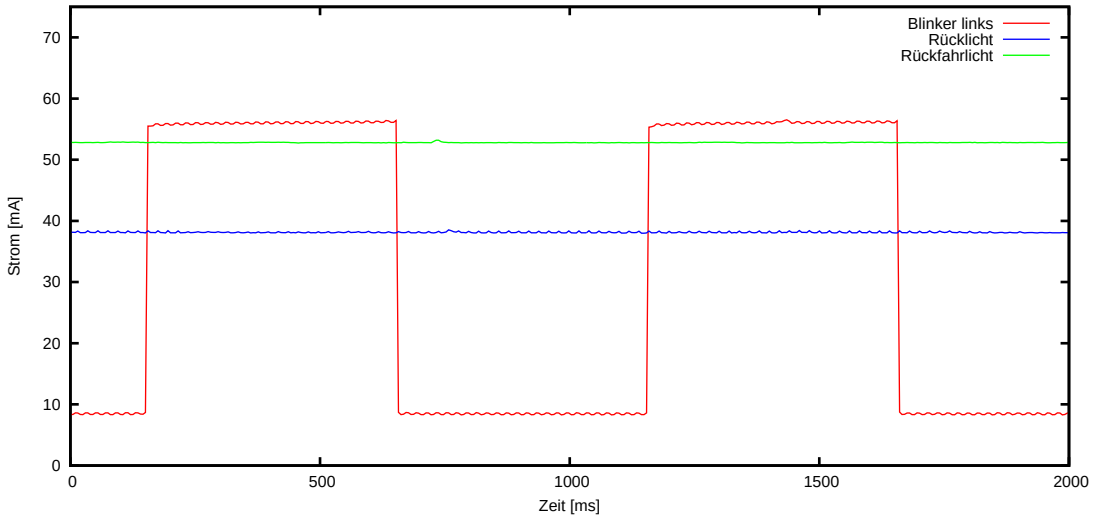


Abbildung 6.5: Stromverbrauch der Fahrzeug-Beleuchtung am LSB.

schnittliche Grundverbrauch des DABs im Bereich von 3,4 mA höher ist als der Stromverbrauch des LSBs, wenn sich der ATmega1284P in den Power-Modi *Power-Down* oder *Power-Save* befindet. Der Grund dafür ist, dass der Beschleunigungssensor MXA2300 auf dem DAB nicht ausgeschaltet werden kann, und somit immer Strom verbraucht. Dieser beläuft sich nach dem Datenblatt auf 4,8 mA bei einer Versorgungsspannung von 3 V [108]. Somit lässt sich der erhöhte Stromverbrauch des DABs gegenüber dem LSB auf den Beschleunigungssensor zurückführen.

Die Messungen des Stromverbrauchs am DAB und LSB in den verschiedenen Power-Modi zeigen die Potenziale für die Energieeinsparung auf, wenn der niedrigste Power-Modus gewählt wird. Dennoch ist es nicht immer möglich, die Modi *Power-Save* oder *Power-Down* zu wählen, da die in diesen Power-Modi ausgeschalteten Hardware-Komponenten für die Fahrzeug-Funktion benötigt werden. Ein Beispiel wäre im Falle des DABs die Ansteuerung des Fahrten-Reglers und des Lenk-Servos. Dafür wird die Erzeugung des PWM-Signals benötigt, die über den Timer 3 laufen. Dafür muss die Clock clk_{IO} eingeschaltet sein, die die Timer für ihre Funktion benötigen. Jedoch ist die Clock clk_{IO} nur im Power-Modus *Idle* aktiv, womit der Mikrocontroller auf dem DAB nicht die Power-Modi *Standby* oder *Power-Down* einnehmen kann, wenn das Modell-Fahrzeug in Bewegung ist.

Neben dem Mikrocontroller ATmega1284P sind am LSB weitere Hardware-Komponenten wie die LEDs und die Abstandssensoren angeschlossen, deren Stromverbrauch ebenfalls ermittelt wird. Um deren Stromverbrauch einschätzen zu können, werden die Hardware-Komponenten einzeln ein- und ausgeschaltet. Die Differenz des Stromverbrauchs am LSB ergibt somit den Strombedarf der einzelnen Hardware-Komponenten.

Zuerst werden die Blinker, die Rücklichter und die Rückfahrlichter des Modell-Fahrzeugs vermessen, deren Ergebnis in Abbildung 6.5 zu sehen ist. Die drei Beleuchtungen sind jeweils hintereinander eingeschaltet worden und der Stromverbrauch am LSB wurde dabei gemessen. Bei den Blinkern ist der ausgeschaltete Zustand ebenfalls zu

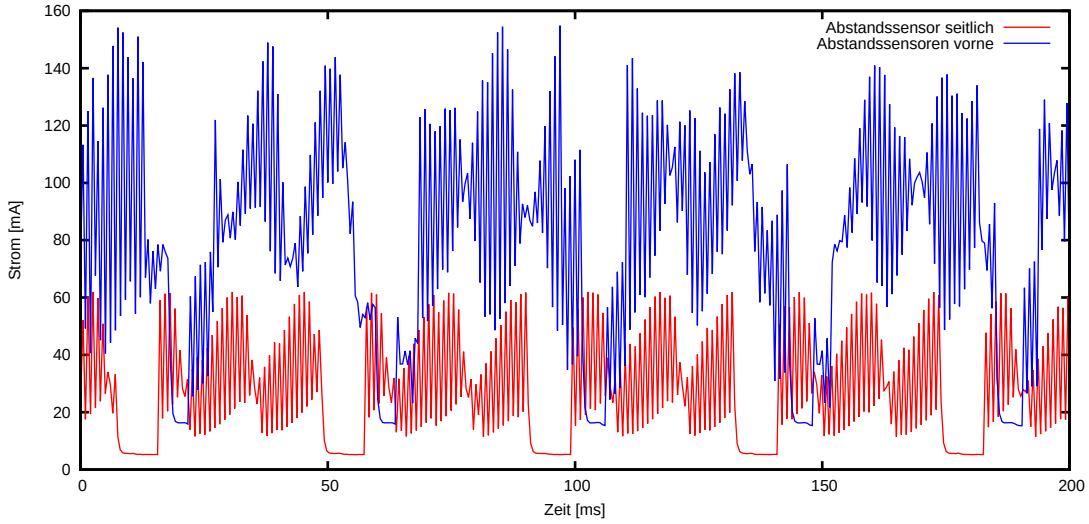


Abbildung 6.6: Stromverbrauch der Abstandssensoren am LSB.

sehen, bei dem sich der Stromverbrauch durchschnittlich bei 8,5 mA befindet, was dem Stromverbrauch des ATmega1284P im Power-Modus *Idle* im LSB entspricht. Bei eingeschalteten Blink-Lichtern ist der gesamte Stromverbrauch des LSBs bei durchschnittlichen 56,0 mA. Somit lässt sich der reine Stromverbrauch der Blinker-LEDs auf die Differenz von ca. 47,5 mA beziffern. Wenn die Rücklichter alleine aufleuchten, ist der Stromverbrauch des LSBs bei 38,1 mA. Gleich verhält es sich mit den Rückfahrlichtern, bei deren Aufleuchten das LSB durchschnittlich 52,8 mA benötigt.

Neben den LEDs sind auch die Abstandssensoren am LSB vermessen worden. Deren Stromverbrauch ist in Abbildung 6.6 dargestellt, wofür zwei Messungen durchgeführt worden sind. Zuerst wurde nur der seitlich angebrachte Abstandssensor eingeschaltet. Auffällig ist hierbei, dass dieser keinen konstanten Stromverbrauch hat, sondern dieser zwischen 5,2 mA und maximal 61,9 mA schwankt. Für eine Dauer von ca. 7,5 ms bleibt der Stromverbrauch des LSBs konstant bei ca. 5,4 mA, bevor der Stromverbrauch wieder ansteigt und für ca. 34,2 ms zu schwanken beginnt. In der zweiten Messung wurden die drei Abstandssensoren vermessen, die vorne am Modell-Fahrzeug angebracht sind. Falls wie in dieser Messung mehrere Abstandssensoren parallel betrieben werden, überlagern sich, wie zu erwarten, die Schwankung und der Stromverbrauch des LSBs verdreifacht sich.

6.2.2 Messergebnisse während der Anwendungsfälle

Nachdem die Referenz-Messungen der Hardware-Komponenten dargestellt wurden, wird im Folgenden der Stromverbrauch des DABs und des LSBs während einiger, ausgesuchter Anwendungsfälle der PLASA-Plattform vorgestellt.

Zuerst wird der Stromverbrauch am DAB im Park-Zustand, während des manuellen Fahrens und des Fahrens mit aktivierter Geschwindigkeitsregelung gemessen. Während der drei Anwendungsfälle befindet sich das Modell-Fahrzeug entsprechend in den Fahr-

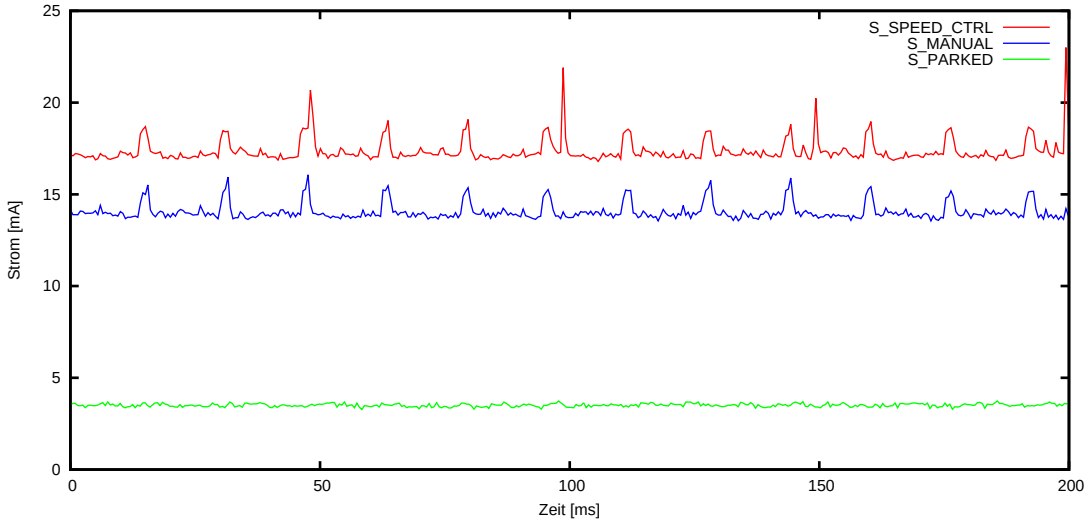


Abbildung 6.7: Stromverbrauch des DABs in den drei Fahrzeug-Zuständen S_PARKED, S_MANUAL und S_SPEED_CTRL.

zeug-Zuständen S_PARKED, S_MANUAL oder S_SPEED_CTRL. Der Stromverbrauch des DABs in diesen drei Fahrzeug-Zuständen ist in Abbildung 6.7 dargestellt.

Im Zustand S_PARKED benötigt das DAB durchschnittlich 3,5 mA und ist ständig auf demselben Niveau. Dies entspricht, wie es in der Referenz-Messung im vorherigen Abschnitt gezeigt wurde, dem Stromverbrauch des DABs, wenn der ATmega1284P im Power-Modus *Power-Down* läuft. Damit schaltet der PM auf dem DAB alle nicht benötigten Verbraucher ab und legt den ATmega1284P im Power-Modus *Power-Down* schlafen. Dies entspricht ebenfalls nach der logischen Architektur der PLASA-Plattform dem gewünschten Verhalten, da im Zustand S_PARKED, keine Aktoren aktiv sein sollten, die nach dem Deployment dem DAB zugeordnet sind.

Im Fahrzeug-Zustand S_MANUAL, in dem das Modell-Fahrzeug manuell mit Hilfe der Nintendo Wii Remote gesteuert wird, steigt der Stromverbrauch des DABs auf eine untere Grenze von ca. 14 mA. Dies entspricht dem Stromverbrauch des DABs, wenn sich der ATmega1284P im Power-Modus *Idle* befindet. Dies ist im Fahrzeug-Zustand S_MANUAL notwendig, da für die Generierung der PWM-Signale die Clock clk_{IO} benötigt wird, die nur im Power-Modus *Idle* noch aktiv ist. Auffällig sind die Stromspitzen, bei denen der Stromverbrauch für 2 ms alle 14,5 ms auf ca. 15,5 mA ansteigt. Dies ist auf die Generierung der PWM-Signale zurückzuführen, die für die Ansteuerung des Fahrten-Reglers und des Lenk-Servos benötigt werden.

Im dritten Fahrzeug-Zustand S_SPEED_CTRL, in dem das Modell-Fahrzeug mit Hilfe eines Geschwindigkeitsregler die Geschwindigkeit konstant hält, erhöht sich der Grund-Verbrauch des DABs um weitere 2,2 mA auf durchschnittlich 17,2 mA. Für die Steigerung des Stromverbrauchs ist der Hall-Sensor verantwortlich, der für die Geschwindigkeitsregelung vom PM im DAB eingeschaltet wird, um die Geschwindigkeit des Modell-Fahrzeugs messen zu können. Nach dem Datenblatt des Hall-Sensors benötigt dieser zwischen 1,6 mA und 5,2 mA [64], was der hier gemessenen Steigerung des Stromver-

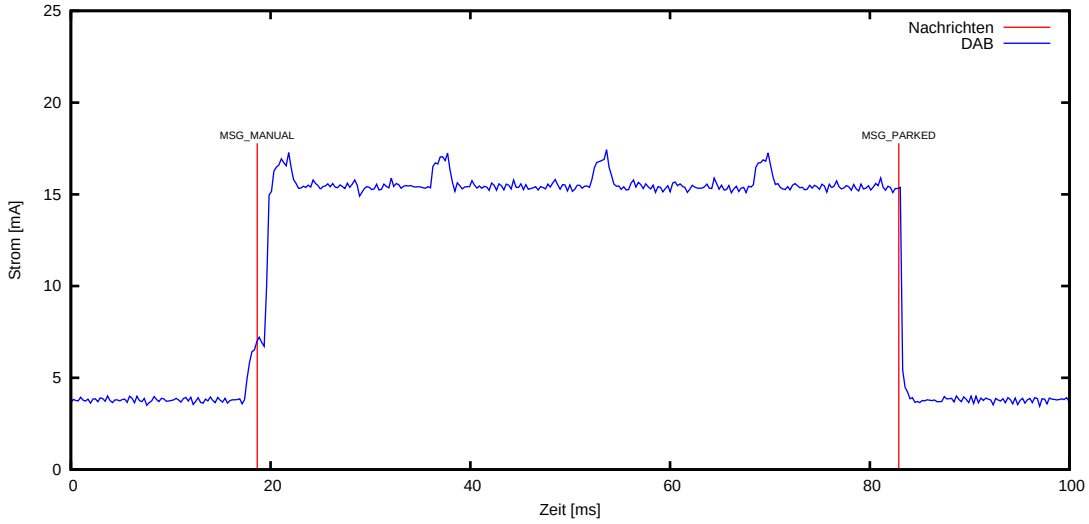


Abbildung 6.8: Stromverbrauch des DABs während eines Wechsels des Zustands.

brauchs entspricht.

Ebenfalls sind wieder Spitzen im Stromverbrauch in den Messungen des DABs zu sehen. Hier allerdings sind zwei Spitzen unterschiedlicher Periode und unterschiedlicher Höhe sichtbar. Die eine Periode ist schon aus dem Zustand S_MANUAL bekannt, bei der der Stromverbrauch des DABs um 1,5 mA alle 14,5 ms für 2 ms ansteigt. Diese lässt sich wie im Fahrzeug-Zustand S_MANUAL auf die Erzeugung des PWM-Signals zurückführen. Die zweite Periode hat eine Periodenlänge von 50 ms mit einer Erhöhung des Stromverbrauchs auf ca. 23 mA. Diese Steigerung wird durch den Geschwindigkeitsregler verursacht, der vom PM in dem Fahrzeug-Zustand S_SPEED_CTRL aktiviert wird. Dieser misst alle 50 ms die aktuelle Geschwindigkeit und regelt die Motor-Leistung, um die gewünschte Soll-Geschwindigkeit zu halten.

Dieser Vergleich der Stromverbräuche in den drei Zuständen des DABs zeigt, dass der PM im DAB sowohl die Hardware-PMCs als auch die Software-PMCs korrekt schaltet. Durch den Wechsel von dem Zustand S_PARKED auf S_MANUAL werden die Software-Tasks für die logischen Funktionsblöcke ACT_SERVO und ACT_MOTOR eingeschaltet, die die PWM-Signale für den Lenk-Servo bzw. den Fahrten-Regler generieren. Des Weiteren wird der Hall-Sensor und der Software-Task für den logischen Funktionsblock FCN_SPEED_CTRL nur im Fahrzeug-Zustand S_SPEED_CTRL aktiviert, in dem die Geschwindigkeitsregelung eingeschaltet ist.

In der nächsten Messung wird der Stromverbrauch des DABs bei einem Wechsel des Zustands mit Hilfe zweier Nachrichten verfolgt. Dies soll zeigen, wie der PM im DAB auf Nachrichten reagiert und die PMCs in den entsprechenden Zustand schaltet, was in Abbildung 6.8 gezeigt wird. Das DAB befindet sich im Zustand S_PARKED und erhält die Nachricht MSG_MANUAL. Daraufhin wechselt das DAB in den Zustand S_MANUAL, was am Stromverbrauch des DABs erkannt wird. Alle 14,5 ms erhöht sich der Stromverbrauch periodisch, was sich auf die Erzeugung des PWM-Signals für den Lenk-Servo und den Fahrten-Regler zurückführen lässt, wie es bereits zuvor in Abbildung 6.7 ge-

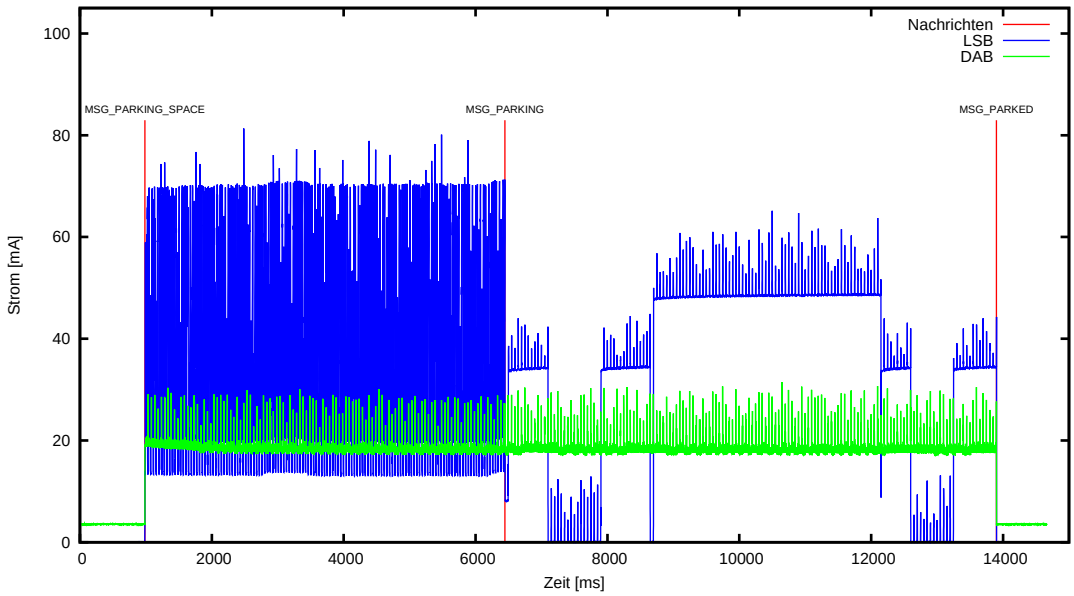


Abbildung 6.9: Stromverbrauch des LSBs und des DABs während des Einparkens.

zeigt wurde. Sobald die Nachricht MSG_PARKED auf dem Bus gesendet wird, wechselt das DAB in den Zustand S_PARKED und der Stromverbrauch springt auf das Niveau zurück, wenn sich der ATmega1284P im Power-Modus *Power-Down* befindet.

Als letzte Messung wird der Stromverbrauch des DABs und des LSBs während eines Einpark-Manövers vorgestellt und erläutert, wie sich dies mit den Entscheidungen des DPMs in der PLASA-Plattform erklären lässt. Im Abschnitt 5.1 ist beschrieben, dass das Einparken in zwei Schritten erfolgt. Im ersten Schritt wird die Parklücke gesucht und im zweiten Schritt wird mit Hilfe der Fahrsequenz-Regelung in die Parklücke hinein manövriert. Der Stromverbrauch des DABs und des LSBs während des Einparkens ist in der Abbildung 6.9 dargestellt. Durch die drei Nachrichten MSG_PARKING_SPACE, MSG_PARKING und MSG_PARKED wird der Einpark-Vorgang in die zwei Teilschritte geteilt. Das Modell-Fahrzeug befindet sich zuerst im Zustand S_PARKED und beginnt mit der Nachricht MSG_PARKING_SPACE einzuparken. Der erste Teilschritt wird zwischen den Nachrichten MSG_PARKING_SPACE und MSG_PARKING durchgeführt, worauf der zweite Teilschritt zwischen den Nachrichten MSG_PARKING und MSG_PARKED erfolgt. Im Anschluss befindet sich das Modell-Fahrzeug wieder im Zustand S_PARKED.

Der Stromverbrauch des DABs liegt vor der Nachricht MSG_PARKING_SPACE und nach der Nachricht MSG_PARKED bei den erwarteten Werten, die auftreten, wenn der Atmel im Power-Modus *Power-Down* läuft. Dagegen steigt der Stromverbrauch des DABs während des Einparkens auf den des Zustands S_SPEED_CTRL. Dabei gibt es im zweiten Teilschritt des Einparkens keine Veränderung, da die Geschwindigkeitsregelung ebenfalls für die Fahrsequenz-Regelung benötigt wird. Während dieser Phase zeigt der Stromverbrauch des DABs den typischen Verlauf, der bei einer aktiven Geschwindigkeits- und Fahrsequenz-Regelung auftritt, wobei sich alle 50 ms der Strom-

Nr.	Abschnitt		ØI	Beschreibung
	von	bis		
1.	6498 ms	7099 ms	34.3 mA	Bremsen.
2.	7099 ms	7903 ms	0.4 mA	kurzes Vorwärtsfahren für ein optimales Einparken.
3.	7903 ms	8649 ms	34.5 mA	Bremsen.
4.	8649 ms	12152 ms	48.8 mA	Rückwärts in die Parklücke einparken.
5.	12152 ms	12599 ms	34.3 mA	Bremsen.
6.	12599 ms	13250 ms	0.4 mA	kurzes Vorwärtsfahren in die Mitte der Lücke.
7.	13250 ms	13902 ms	34.5 mA	Bremsen.

Tabelle 6.2: Abschnitte im zweiten Teilschritt des Einparkens.

bedarf für die Berechnungen der Regler erhöht.

Dagegen zeigt der Stromverbrauch des LSBs deutliche Unterschiede zwischen den verschiedenen Teilschritten des Einparkens. Im ersten Teilschritt wird die Parklücke gesucht, wofür der seitlichen Abstandssensor aktiviert wird, der für die starken Schwankungen im Stromverbrauch des LSBs verantwortlich ist. Diese Schwankungen sind in dem Bereich des Distanzsensors, wie es in der Referenz-Messung des seitlichen Distanz-Sensors in Abbildung 6.6 gezeigt wurde. Sobald die Nachricht MSG_PARKING versendet wird, ist die Parklücke gefunden und das Einpark-Manöver beginnt. Die starken Schwankungen im Stromverbrauch des LSBs, die durch den Distanz-Sensor hervorgerufen wurden, sind nicht mehr vorhanden. Damit wurde dieser vom PM im LSB korrekt ausgeschaltet, da er für den zweiten Schritt des Einparkens nicht benötigt wird.

Der Stromverbrauch des LSBs im zweiten Schritt des Einparkens unterteilt sich in sieben verschiedene Abschnitte, die mit Hilfe der Fahrsequenz-Regelung abgefahren werden. Diese sind in Tabelle 6.2 näher beschrieben, wobei zusätzlich der Start- und End-Zeitpunkt und der durchschnittliche Stromverbrauch des Abschnitts angegeben sind. Der Stromverbrauch des LSBs während der Brems-Abschnitte 1, 3, 5 und 7 gleichen sich, wobei der Grundverbrauch bei ca. 34,5 mA liegt. Dies entspricht dem Stromverbrauch der Bremsleuchte, wie in Abbildung 6.5 dargestellt ist. Bei den Abschnitten 2 und 6 mit einer Vorwärtsfahrt hat das LSB keinen Grundverbrauch, da keine zusätzlichen Scheinwerfer eingeschaltet sind. In Abschnitt 4, bei dem das Modell-Fahrzeug rückwärts fährt, sind die Rückfahrlichter aktiv, was mit den durchschnittlich gemessenen 48,8 mA dem Verbrauch der Rücklichter entspricht (siehe ebenfalls Abbildung 6.5).

Während des gesamten zweiten Teilschritts des Einparkens sind alle 50 ms eine Erhöhung des Stromverbrauchs am LSB zu sehen. Dieser wird durch den Empfang der Nachrichten verursacht, die vom DAB verschickt werden. Denn alle 50 ms versendet die Geschwindigkeitsregelung die Nachricht MSG_MOTOR, auf die das LSB nach der logischen Architektur hört, um die angeschlossenen Lichter zu schalten. Damit wird der Atmel auf dem LSB aus dem Schlafzustand geweckt, der im Anschluss die Nachricht verarbeitet. Diese Verarbeitung ist für den Anstieg des Stromverbrauchs verantwortlich.

Die in diesem Abschnitt gezeigten Messungen des Stromverbrauchs am DAB und LSB zeigen, dass das DPM in der PLASA-Plattform korrekt funktioniert. Sowohl die Hardware-PMCs als auch die Software-PMCs werden durch den PM der jeweiligen ECU geschaltet, was sich teilweise deutlich am Stromverbrauch der ECUs bemerkbar macht.

6.3 Bewertung des Mess-Systems aus der PLASA-Plattform

Das für die PLASA-Plattform entwickelte Mess-System, das den Stromverbrauch des Gumstixs, des DABs und des LSBs misst, sorgt für den Nachweis, dass das DPM in der PLASA-Plattform korrekt funktioniert. Das Mess-System wird komplett in das Modell-Fahrzeug integriert und misst während der Ausführung der Anwendungsfälle den Stromverbrauch der drei ECUs. Dabei ermittelt die entwickelte Mess-Platine den Stromverbrauch und schreibt die gemessenen Werte in eine Datei-basierte Datenbank. Zusätzlich werden die auf dem Fahrzeug-Bus versendeten Nachrichten protokolliert und neben den Messwerten in der Datenbank gespeichert. Die Auswertung der Messungen erfolgt offline auf einem Desktop-System in einem nachgelagerten Schritt, indem die Datei-basierte Datenbank aus dem Fahrzeug kopiert und auf dem Desktop-System ausgewertet wird. Dies ermöglicht bei Archivierung der Datenbank-Dateien auch eine Auswertung bestimmter Fahrten zu einem späteren Zeitpunkt.

Durch Referenz-Messungen wurde der Stromverbrauch in definierten Fahrzeug-Zuständen ermittelt, die dafür genutzt wurden, um das korrekte Verhalten des DPMs anhand des Stromverbrauchs während der Ausführung der Anwendungsfälle nachzuweisen. Da der Fokus bei der Umsetzung des PMs auf TinyOS liegt, stehen auch die Messergebnisse des DABs und LSBs im Vordergrund. Es wurde durch den Stromverbrauch des DABs und des LSBs gezeigt, dass aufgrund der auf dem Fahrzeug-Bus versendeten Nachrichten der PM den Power-Modus der PMCs an den jeweiligen Fahrzeug-Zustand anpasst und sich damit der Stromverbrauch der ECUs ändert. Besonders im Anwendungsfall des Einparkens, das ca. 13 s dauert, sind die Auswirkungen des DPMs am Stromverbrauch in der PLASA-Plattform deutlich beobachtbar.

Das Mess-System liefert den Nachweis, dass das DPM der PLASA-Plattform korrekt funktioniert. Allerdings ist keine detaillierte Analyse durchgeführt worden, wie viel Energie durch das DPM in der PLASA-Plattform eingespart wird. Der Grund dafür ist die fehlende Analyse der Messgenauigkeit, was für den Nachweis der Energieeinsparung unbedingt notwendig ist.

Kapitel 7

Zusammenfassung und Ausblick

Zum Abschluss dieser Arbeit werden die Inhalte des DPMs für Automotive Systeme noch einmal zusammenfassend dargestellt und Erweiterungsmöglichkeiten der PLASA-Plattform vorgestellt. Am Ende wird noch in einem Ausblick diskutiert, wie das DPM, das in die PLASA-Plattform integriert wurde, auch in realen Automotiven Systemen zum Einsatz kommen könnte und welche Herausforderungen dort noch zu meistern sind.

7.1 Zusammenfassung

Ein Automotives System besteht aus bis zu 80 ECUs, die über mehrere Bus-Systeme miteinander verbunden sind. Somit ist ein Automotives System ein stark verteiltes System, in dem ein DPM für Automotive Systeme ebenfalls verteilt auf den ECUs arbeiten muss. Dies bedeutet, dass die PMs auf den einzelnen ECUs nicht nur auf Basis der Informationen, die sie lokal auf ihrer ECU erhalten, ihre Entscheidungen treffen, sondern dass sie das ganze Automotive System mit dessen Zustand betrachten müssen. Um dies zu gewährleisten, wird das DPM für Automotive Systeme hierarchisch in mehreren Abstraktionsebenen unterteilt. Dafür gibt es einen PM auf System-Ebene, der das DPM für das gesamte Automotive System regelt, auf das der Wunsch des Fahrers großen Einfluss nimmt. Die Regel-Befehle werden dann auf die verschiedenen Domänen des Automotiven Systems verteilt, worauf die PMs auf Domänen-Ebene ihre Entscheidungen treffen. Auf der nächst tieferen Abstraktionsstufe arbeiten die PMs auf ECU-Ebene, worauf der Fokus dieser Arbeit liegt. Es wurde untersucht, wie ein PM auf ECU-Ebene die Zustände des Automotiven Systems ohne erhöhten, zusätzlichen Aufwand aus den Bus-Nachrichten erhält und darauf basierend seine Entscheidungen trifft.

Um Erkenntnisse über ein DPM für Automotive Systeme zu gewinnen, wurde ein DPM prototypisch implementiert und in die PLASA-Plattform integriert, die aus dem Lehrbetrieb am Fachgebiet für Betriebssysteme im Institut für Informatik der Technischen Universität München entstanden ist. Die PLASA-Plattform ist ein kleines Automotives System in Form eines Modell-Fahrzeugs, für das Anwendungsfälle wie u. a. ein Geschwindigkeitsregler, eine Nothalt-Funktion oder das automatische Einparken

als Grundlage für die Untersuchungen definiert wurden. Aus den definierten Anwendungsfällen ist eine logische Architektur der PLASA-Plattform spezifiziert worden, auf die sich das integrierte DPM stützt.

Drei der vier ECUs in der PLASA-Plattform laufen auf der Basis von dem Komponenten-basierten Betriebssystem TinyOS, das für das Anwendungsgebiet der WSNs konzipiert wurde. Dadurch sind in der PLASA-Plattform Grundlagen des Power-Managements aus dem Bereich der WSNs enthalten. Da ein energieeffizientes Power-Management eines der Entwicklungsprinzipien von TinyOS ist, wurden die Konzepte des Power-Managements ebenfalls in der PLASA-Plattform wiederverwendet. Allerdings musste TinyOS um automotive Software-Komponenten erweitert werden, um die zu untersuchenden Anwendungsfälle der PLASA-Plattform zu realisieren. Im Rückblick zeigt sich, dass sich TinyOS als Software-Basis für das PSB, das DAB und das LSB bewährt hat, obwohl es grundsätzlich nicht für harte Echtzeit-Systeme ausgelegt ist. Jedoch bewältigt TinyOS die weichen Echtzeit-Bedingungen der PLASA-Plattform hervorragend, wie es mit den Anwendungsfällen gezeigt werden konnte. Zusätzlich konnten die bereits vorhandenen Power-Management-Konzepte in TinyOS für den PM auf ECU-Ebene wiederverwendet werden. Durch die Komponenten-basierte Architektur von TinyOS ließen sich die zusätzlich benötigten, automotiven Funktionen leicht integrieren, wofür bereits in TinyOS existierende Komponenten u. a. für die Timer- oder ADC-Steuerung sogar in der PLASA-Plattform wiederverwendet werden konnten.

Um ein DPM auf ECU-Ebene zu realisieren, ist in jeder ECU ein PM vorhanden, der die lokalen PMCs der ECU nach dem Zustand des gesamten Automotiven Systems steuert. In dieser Arbeit wurde gezeigt, wie sich ein solcher PM in das Software-System einer ECU eingliedern lässt. Dabei steuert der PM auf ECU-Ebene in der PLASA-Plattform nicht nur Hardware-PMCs, sondern auch die Software-PMCs in Form von Software-Tasks, die sich aus der logischen Architektur der PLASA-Plattform ableiten.

Entscheidend für das DPM auf ECU-Ebene ist die PM-Logik mit der PM-Wissensbasis, die die Strategie des DPMs enthält und entscheidet, welchen Power-Modus die ECU abhängig von dem Zustand des gesamten Automotiven Systems einnimmt. Die Entscheidungen fallen aufgrund der Nachrichten, die im System zwischen den Funktionsblöcken der logischen Architektur versendet werden. Die Schwierigkeit liegt hierbei nun darin, aus der logischen Architektur die PM-Logik einer jeden ECU im System zu bestimmen. Für die PLASA-Plattform wurde die PM-Logik durch Ingenieursleistung erstellt, wofür sich allerdings für zukünftige Untersuchungen die Unterstützung durch CASE-Werkzeuge anbieten würde.

Für den Nachweis, dass das DPM in der PLASA-Plattform korrekt funktioniert und ein energieeffizientes Power-Management ermöglicht, ist der Stromverbrauch von zwei der vier ECUs während der Anwendungsfälle aufgezeichnet und ausgewertet worden. Anhand von zuvor erstellten Referenz-Messungen konnte der während des Ablaufs der Anwendungsfälle gemessene Stromverbrauch den einzelnen Aktoren und Sensoren zugeordnet und deren Zustand je nach Aktivität bzw. Inaktivität sichtbar gemacht werden. So konnte belegt werden, dass die PMs auf den ECUs ihre PMCs bei Zustandsänderungen im Automotiven System, die sie durch das Mithören der Nachrichten erkennen, korrekt ein- und ausschalten. Die PMs auf den ECUs stellen dabei immer den niedrigsten Energieverbrauch der ECUs ein, da sie alle für den aktuellen Fahrzeug-Zustand nicht

benötigten PMCs abschalten. Das Ein- bzw. Ausschalten der PMCs ist besonders im Anwendungsfall des Einparkens beobachtbar und zeigt das Einsparungspotenzial hinsichtlich des Stromverbrauchs mit Hilfe eines energieeffizienten Power-Managements.

7.2 Erweiterungsmöglichkeiten der PLASA-Plattform

In dieser Arbeit wurde ein DPM für Automotive Systeme prototypisch für die PLASA-Plattform umgesetzt und in diese integriert. Aus der Umsetzung können Erkenntnisse gezogen werden, wie man das DPM für Automotive Systeme noch erweitern bzw. durch Generierungsschritte zwischen den Abstraktionsebenen noch verbessern kann.

Für die PLASA-Plattform wurden die Zustandsautomaten der PM-Logik in den verschiedenen ECUs durch Ingenieursleistung erstellt. Dabei wurde die logische Architektur der PLASA-Plattform als Grundlage für die Zustandsautomaten herangezogen, aber nicht in einem CASE-Werkzeug verwendet. Ein nächster Schritt wäre die automatische Generierung der Zustandsautomaten für die PM-Logik aus der logischen Architektur. Dafür wäre zusätzlich eine formale Beschreibung der Partitionierung für die Software-Komponenten auf die ECUs notwendig. Des Weiteren ist ebenfalls denkbar, dass die logische Architektur um zusätzliche Attribute erweitert wird, um eine vollständige Generierung der Zustandsautomaten zu ermöglichen. In der bisherigen Form sind die Funktionsketten nur eine informelle Beschreibung des Informations- und Nachrichtenflusses in der PLASA-Plattform, aus der die Software-Tasks in TinyOS abgebildet werden. Mit Hilfe von CASE-Werkzeugen würde die PM-Logik flexibler und weniger anfällig für Fehler sein und somit zu einem stabilerem DPM für Automotive Systeme führen.

Zusätzlich ist die Zuordnung der Software-Tasks auf die ECUs in der PLASA-Plattform vorgegeben und wirkt sich stark auf die Zustandsautomaten der PM-Logik aus. Falls Software-Tasks beliebig auf ECUs verteilt werden sollen, so müssen die Zustandsautomaten flexibel und schnell an die neue Zuordnung angepasst werden. Dafür ist ebenfalls die Generierung der Zustandsautomaten durch CASE-Werkzeuge hilfreich. Nur so kann erreicht werden, dass eine Änderung der Zuordnung schnell und ohne Fehler auf die PLASA-Plattform aufgespielt und auf ihren Stromverbrauch untersucht werden kann.

Eine weitere Möglichkeit für den Einsatz von CASE-Werkzeugen ist die automatische Generierung der Software-Task in TinyOS. Bis jetzt werden nur die Verbindungen zwischen den Software-Tasks und dem PM generiert. Allerdings müssen die Schnittstellen in den Software-Tasks immer ordnungsgemäß kommentiert werden, damit die Verbindungen korrekt erzeugt werden. Dies ist ebenfalls eine mögliche Fehlerquelle, so dass eine automatische Generierung der Software-Task zusammen mit der Verbindung zum PM erhebliche Vorteile bieten könnte.

Neben den CASE-Werkzeugen, die automatisch aus der logischen Architektur die Software-Tasks generieren und den PM konfigurieren, könnten weitere DPM-Strategien in der PLASA-Plattform untersucht und vermessen werden. Der PM mit seiner PM-Logik könnte durch eine andere Implementierung ausgetauscht werden, die z. B. nicht nur die Nachrichten der logischen Architektur als Eingangssignal für eine Zustandsänderung des Automaten, der den Power-Modus der ECU bestimmt, heranziehen. So könnten auch weitere Informationen herangezogen oder Annahmen getroffen werden, wie der

Erwartungswert einer Zustandsänderung im System, um eine Zustandsänderung auszulösen, wie es bereits für das DPM von mobile Geräte untersucht wurde [30].

Die PLASA-Plattform bildet ebenfalls eine Basis, um weitere automotive Anwendungsfälle zu implementieren und diese im Bereich des DPMs und nach energetischen Gesichtspunkten zu untersuchen. So können sich weitere Fahrzeug-Funktionen wie die adaptive Geschwindigkeitsregelung, bei der die aktuelle Geschwindigkeit einem voranfahrenden Fahrzeug angeglichen wird, realisieren lassen. An dieser Stelle sind der PLASA-Plattform keine Grenzen gesetzt, sofern sich die benötigte Sensoren und gegebenenfalls die notwendigen Aktoren in das Modell-Fahrzeug integrieren lassen.

Um die Energieeinsparung durch das DPM in der PLASA-Plattform während der Ausführung der Anwendungsfälle genau zu ermitteln, muss zuvor die Messgenauigkeit des vorhandenen Mess-System bestimmt werden, um damit belastbare Messergebnisse zu erhalten. Falls die Messgenauigkeit nicht ausreichend ist, muss das Mess-System durch ein genaueres ersetzt werden. Eventuell sollte hierfür auch der Aufbau des Mess-Systems überdacht werden, um z. B. noch mehr Messpunkte aufzunehmen, um nicht nur den Stromverbrauch der ECUs, sondern auch der einzelnen Sensoren oder der Aktoren wie den Lenk-Servo oder den Motor zu messen. Damit wäre eine viel genauere Analyse des Energieverbrauchs der PLASA-Plattform möglich. Dadurch ließen sich mit weiteren Mess-Reihen die Energieeinsparungen durch ein energieeffizientes DPM in der PLASA-Plattform genau ermitteln.

Ob daraus allerdings die Energieeinsparungen realer Fahrzeuge abgeleitet werden könnte, ist wissenschaftlich nicht gesichert. Dagegen spricht vor allem, dass die PLASA-Plattform nur ein Modell eines realen Automotiven Systems ist. Ein reales Automotives System mit seine Domänen-Struktur ist um ein vielfaches größer als die PLASA-Plattform. In diesem sind nicht nur ca. 20-mal so viele ECUs verbaut, sondern auch die Anzahl der Kundenfunktionen wäre um ein Vielfaches größer. Damit die Energieeinsparungen in realen Automotiven Systemen untersucht werden können, müssten die in dieser Arbeit vorgestellten Konzepte in reale Automotive Systeme integriert werden, womit sich im Folgenden der Ausblick dieser Arbeit befasst.

7.3 Ausblick

Neben den bereits vorgestellten Erweiterungsmöglichkeiten der PLASA-Plattform stellt sich noch als Ausblick die Frage, wie die aus der PLASA-Plattform erlangten Erkenntnisse über dessen DPM auf reale Automotive Systeme übertragen werden können. Eine Möglichkeit ist die Einbettung des DPMs für Automotive Systeme in den AUTOSAR-Standard, der detaillierter in Abschnitt 2.4.2 vorgestellt wurde. Dieser bietet bereits mehrere Konzepte für ein Power-Management zur Energieeinsparung an, was zu Beginn des Kapitels 4 erläutert wurde. Dieser Weg wird im Folgenden nur kurz skizziert und müsste in einem System mit AUTOSAR-gesteuerten ECUs weiter untersucht werden.

Eine Möglichkeit wäre hierbei, dass der PM des DPMs für Automotive Systeme als SW-C außerhalb der AUTOSAR-BSW läuft und die Schnittstellen des *ECU State Managers* und des *BSW Mode Manager* nutzt, um seine Befehle an das System abzusetzen. Dies hätte den Vorteil, dass der AUTOSAR-Standard nicht um BSW-Module erweitert

oder weitere AUTOSAR-Schnittstellen definiert werden müssten. Allerdings müsste sich zeigen, ob die in AUTOSAR vorhandenen Schnittstellen des *ECU State Managers* und des *BSW Mode Managers* ausreichen, um alle notwendigen Befehle des PMs zur Steuerung seiner PMCs abzubilden [10].

Damit allerdings der PM seine Entscheidungen treffen kann, müssen ihm die Fahrzeug-Zustandsänderungen über die relevanten Nachrichten mitgeteilt werden. Damit dem PM alle benötigten Nachrichten zugestellt werden, ist er in der PLASA-Plattform tief im System verankert. Vor allem Nachrichten anderer ECUs, die einen Wechsel des Fahrzeug-Zustandes herbeiführen, müssten bei Bedarf an den PM über den Nachrichten-Bus gesendet werden, damit er seine PMCs auf der ECU korrekt schalten kann. Dies ist in dieser Form nicht im AUTOSAR-Standard realisierbar, wenn der PM als SW-C umgesetzt ist. Allerdings könnte ein PM als SW-C über den VFB alle benötigte Nachrichten über Ports empfangen. Hierfür ist wie in der PLASA-Plattform das Wissen notwendig, welche Nachrichten für den PM relevant sind. Dieses Wissen müsste wieder aus einer logischen Architektur des gesamten Automotiven Systems hervorgehen und der PM müsste für jede relevante Nachricht einen entsprechenden Port am VFB besitzen. Über die Generierung der RTE und der Kommunikationsmodule in der AUTOSAR-BSW würde der PM auch die Nachrichten von SW-Cs erhalten, die auf anderen ECUs laufen.

Zusätzlich schaltet der PM in der PLASA-Plattform die benötigten Software-PMCs aktiv und leitet Nachrichten nur an aktive Software-PMCs weiter, damit sie ihre Funktion erfüllen können. Dies ist auf einer AUTOSAR-ECU nicht ohne weiteres umsetzbar, wenn der PM als SW-C läuft. Denn dieser kann zwar einen Änderung des Modes beim *BSW Mode Manager* veranlassen, der anschließend die Mode-Änderung an die übrigen SW-Cs weiterreicht. Allerdings sind die SW-Cs selbst dafür verantwortlich, dass sie auch die Mode-Änderung vornehmen. Somit würde es kein zentrales Zustandsmanagement der SW-Cs im PM geben, sondern ein dezentrales Management, in dem jede SW-C selbst den aktuellen Zustand steuert. Dies führt dann allerdings dazu, dass die SW-Cs in allen Zuständen ihre benötigten Nachrichten erhalten und nicht nur in ihrem aktiven Zustand. Somit würden sie die Nachrichten immer empfangen und anschließend abhängig von ihrem Zustand die bereits empfangenen Nachrichten ignorieren oder bearbeiten. Dies hat zur Folge, dass Nachrichten von SW-Cs, die auf einer anderen ECU laufen, immer über den Bus gesendet werden und nicht wie in der PLASA-Plattform abhängig davon, ob in dem aktuellen Fahrzeug-Zustand die Nachrichten von einer anderen SW-C benötigt werden oder nicht.

Der AUTOSAR-Standard bietet somit Mechanismen an, um ein DPM für Automotive Systeme zu realisieren und eine systemweite Strategie umzusetzen. Dennoch legt der AUTOSAR-Standard nicht die Definition der systemweiten Fahrzeug-Zustände fest, die angepasst an jedes Automotive System definiert werden müssten. Aus diesem Grund muss der OEM die Strategie für das DPM des gesamten Automotiven Systems definieren, das die verschiedenen Tier 1 in den SW-Cs auf den ECUs realisieren müssten. Dafür ist, wie es anhand der PLASA-Plattform gezeigt wurde, eine logische Architektur des gesamten Automotiven Systems notwendig, in der die Funktionalität des Automotiven Systems mit bis zu ca. 2000 Kundenfunktionen abstrahiert werden müssten. Die logische Architektur müsste detailliert genug sein, um ein DPM für das gesamte Automotive Sys-

tem übergreifend umzusetzen. An dieser Stelle ist eine enge Zusammenarbeit zwischen dem OEM und seinem Tier 1 notwendig, da der OEM für die Integration des DPMs in das Gesamtsystem verantwortlich ist und der Tier 1 das Wissen über die ECUs mit den enthaltenen SW-Cs und den PMCs bereitstellt. An dieser Stelle müsste die logische Architektur detailliert genug sein, um daraus die PM-Wissensbasis für jede ECU zu generieren, die anschließend der PM mit seiner PM-Logik auf der jeweiligen ECU befolgen muss.

Ein wesentlicher Punkt, der in realen Automotiven Systemen mit den dort verbauten ECUs und Bus-Systemen untersucht werden müsste, ist das Echtzeit-Verhalten eines DPMs und dessen Einhalten der dort herrschenden Echtzeit-Bedingungen. Für viele der sicherheitsrelevanten Funktionen, die in einem Automotiven System enthalten sind, ist dies ein wesentlicher Punkt, der unter keinen Umständen außer Acht gelassen werden darf. Daraus ergibt sich, dass die PMs mit deren PM-Logik die Echtzeit-Anforderungen beachten und um Eigenschaften und Wissen erweitert werden müssten, um die Echtzeit-Bedingungen garantiert einzuhalten.

Es bleibt zu wünschen, dass die Erkenntnisse aus dieser Arbeit zu energieeffizienteren Automotiven Systemen führen. Durch die Globalisierung und Modernisierung der Welt wird der Bedarf nach der individuellen Mobilität nicht zurückgehen, sondern eher noch steigen. Die Automobile, die einen großen Beitrag zur individuellen Mobilität leisten, würden durch energieeffizientere Automotive Systeme ihren CO_2-Ausstoß weiter absenken. Dies würde einen zusätzlichen Beitrag liefern, um die Erderwärmung weiter zu verringern und das 2 °C-Ziel der internationalen Klima-Politik, wie zu Beginn erläutert, zu erreichen.

Literatur

[1] R. Achatz, K. Beetz, M. Broy, H. Dämbkes, W. Damm, K. Grimm und P. Liggesmeyer. *Nationale Roadmap Embedded Systems*. Techn. Ber. ZVEI - Zentralverband Elektrotechnik- und Elektronikindustrie e.V., Dez. 2009.

[2] D. Aiello. *The Future of AUTOSAR*. Juni 2014.

[3] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt und J. Larus. „Deconstructing process isolation“. In: *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*. San Jose, California: ACM Press, 2006, S. 1–10. ISBN: 1-59593-578-9. DOI: 10.1145/1178597.1178599.

[4] Analog Devices, Inc. *AD7682/AD7689: 16-Bit, 4-Channel/8-Channel, 250 kSPS PulSAR ADC*. 2008.

[5] Analog Devices, Inc. *AD8628/AD8629/AD8630: Zero-Drift, Single-Supply, Rail-to-Rail Input/Output Operational Amplifier*. 2010.

[6] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*. Apr. 2008.

[7] Atmel Corp. *8-bit Atmel Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash — ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, and ATmega1284P*. Apr. 2013.

[8] Atmel Corp. *8-bit Atmel Microcontroller with 16K/32K/64K Bytes In-System Programmable Flash — ATmega164P/V, ATmega324P/V, and ATmega644P/V*. Feb. 2013.

[9] Atmel Corp. *8-bit AVR Instruction Set*. Juli 2010.

[10] AUTOSAR. *Guide to Modemanagement*. Release 4.2.2. 2015.

[11] AUTOSAR. *Layered Software Architecture*. Release 4.2.2. 2015.

[12] AUTOSAR. *List of Basic Software Modules*. Release 4.2.2. 2015.

[13] AUTOSAR. *Methodology*. Release 4.2.2. 2015.

[14] AUTOSAR. *Specification of Basic Software Mode Manager*. Release 4.2.2. 2015.

[15] AUTOSAR. *Specification of CAN Driver*. Release 4.2.2. 2015.

[16] AUTOSAR. *Specification of CAN Interface*. Release 4.2.2. 2015.

[17] AUTOSAR. *Specification of CAN Network Management*. Release 4.2.2. 2015.

[18] AUTOSAR. *Specification of CAN State Manager*. Release 4.2.2. 2015.

[19] AUTOSAR. *Specification of CAN Transceiver Driver*. Release 4.2.2. 2015.

[20] AUTOSAR. *Specification of CAN Transport Layer*. Release 4.2.2. 2015.

[21] AUTOSAR. *Specification of ECU State Manager*. Release 4.2.2. 2015.

[22] AUTOSAR. *Specification of Network Management Interface*. Release 4.2.2. 2015.

[23] AUTOSAR. *Specification of Operating System*. Release 4.2.2. 2015.

[24] AUTOSAR. *Specification of RTE*. Release 4.2.2. 2015.

[25] AUTOSAR. „The Worldwide Automotive Standard for E/E Systems“. In: *ATZ-extra* (Sep. 2013), S. 5–12. DOI: 10.1007/s40111-013-0003-5.

[26] AUTOSAR. *Virtual Functional Bus*. Release 4.2.2. 2015.

[27] A. Barthels. „Energy-aware Computing in Embedded Systems and its Operating System Support“. Diss. Technische Universität München, 2014.

[28] A. Barthels, J. Fröschl und U. Baumgarten. „An Architecture for Power Management in Automotive Systems“. In: *Architecture of Computing Systems - ARCS*. 25th International Conference. 2012.

[29] A. Barthels, F. Ruf, G. Walla, J. Fröschl, H.-U. Michel und U. Baumgarten. „A model for sequence based power management in cyber physical systems“. In: *Proceedings of the First international conference on Information and communication on technology for the fight against global warming*. ICT-GLOW'11. Toulouse, France: Springer-Verlag, 2011, S. 87–101. ISBN: 978-3-642-23446-0.

[30] L. Benini, A. Bogliolo und G. D. Micheli. „A survey of design techniques for system-level dynamic power management“. In: *IEEE Transactions on VLSI Systems* 8 (2000), S. 299–316.

[31] Bluetooth SIG. *Specification of the Bluetooth System 2.0 + EDR*. Nov. 2004.

[32] D. Bovet und M. Cesati. *Understanding the Linux Kernel, Second Edition*. Hrsg. von A. Oram. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. ISBN: 0596002130.

[33] M. Broy. „Automotive software and systems engineering“. In: *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*. Juli 2005, S. 143–149. DOI: 10.1109/MEMCOD.2005.1487905.

[34] M. Broy, I. Kruger, A. Pretschner und C. Salzmann. „Engineering Automotive Software“. In: *Proceedings of the IEEE* 95.2 (Feb. 2007), S. 356–373. ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.888386.

[35] M. Broy. „Challenges in automotive software engineering“. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM, 2006, S. 33–42. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134292.

[36] A. Chou, J. Yang, B. Chelf, S. Hallem und D. Engler. „An empirical study of operating systems errors“. In: *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. Banff, Alberta, Canada: ACM Press, 2001, S. 73–88. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502042.

[37] J. Cook, I. Kolmanovsky, D. McNamara, E. Nelson und K. Prasad. „Control, Computing and Communications: Technologies for the Twenty-First Century Model T“. In: *Proceedings of the IEEE* 95.2 (Feb. 2007), S. 334–355. ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.888384.

[38] J. Corbet, A. Rubini und G. Kroah-Hartman. *LINUX device drivers*. 3rd. Sebastopol, CA, USA: O'Reilly Media, Feb. 2005. ISBN: 0-596-00008-1.

[39] P. Cuenot, D. J. Chen, S. Gerard, H. Lonn, M.-O. Reiser, D. Servat, C.-J. Sjostedt, R. Kolagari, M. Torngren und M. Weber. „Managing Complexity of Automotive Electronics Using the EAST-ADL“. In: *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*. Juli 2007, S. 353–358. DOI: 10.1109/ICECCS.2007.28.

[40] V. Debruyne, F. Simonot-Lion und Y. Trinquet. „EAST-ADL — An Architecture Description Language“. In: *Architecture Description Languages*. Hrsg. von P. Dissaux, M. Filali-Amine, P. Michel und F. Vernadat. Bd. 176. IFIP International Federation for Information Processing. Springer Boston, 2005, S. 181–195. ISBN: 978-0-387-24589-8. DOI: 10.1007/0-387-24590-1_12.

[41] R. Dörfel, M. Haunreiter und U. Baumgarten. „Das RTOS Symobi: Erfüllung der Anforderungen in eingebetteten Systemen“. In: *GI/ITG KuVS Fachgespräch Systemsoftware und Energiebewusste Systeme*. Okt. 2007.

[42] F. Dötzer. „Security Concepts for Robust and Highly Mobile Ad-hoc Networks“. Diss. Technische Universität München, 2008.

[43] A. Dunkels, B. Gronvall und T. Voigt. „Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors“. In: *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Washington, DC, USA: IEEE Computer Society, 2004, S. 455–462. ISBN: 0-7695-2260-2. DOI: 10.1109/LCN.2004.38.

[44] Ecma International. *Common Language Infrastructure (CLI): Partition I: Concepts and Architecture*. ISO/IEC 23271, Juni 2012.

[45] Ecma International. *Common Language Infrastructure (CLI): Partition III: CIL Instruction Set*. ISO/IEC 23271, Juni 2012.

[46] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus und S. Levi. „Language support for fast and reliable message-based communication in singularity OS“. In: *EuroSys '06: Proceedings of the 2006 EuroSys conference*. Leuven, Belgium: ACM Press, 2006, S. 177–190. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217953.

[47] J.-P. Fassino, J.-B. Stefani, J. L. Lawall und G. Muller. „Think: A Software Framework for Component-based Operating System Kernels.“ In: *USENIX Annual Technical Conference, General Track*. 2002, S. 73–86.

[48] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley und J. Haskins Jr. „A survey of configurable, component-based operating systems for embedded applications“. In: *Micro, IEEE* 21.3 (2001), S. 54–68.

[49] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa und K. Lange. „AUTOSAR – A Worldwide Standard is on the Road“. In: *14th International VDI Congress Electronic Systems for Vehicles*. 2009.

[50] E. Gabber, C. Small, J. L. Bruno, J. C. Brustoloni und A. Silberschatz. „The Pebble Component-Based Operating System.“ In: *USENIX Annual Technical Conference, General Track*. 1999, S. 267–282.

[51] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer und D. Culler. „The nesC language: A holistic approach to networked embedded systems“. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. PLDI '03. San Diego, California, USA: ACM, 2003, S. 1–11. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781133.

[52] D. Gay, P. Levis, D. Culler und E. Brewer. *nesC 1.2 Language Reference Manual*. Techn. Ber. Aug. 2005.

[53] D. Gay, P. Levis, W. Hong, J. Polastre und G. Tolle. *TEP 109: Sensors and Sensor Boards*. TinyOS Core Working Group.

[54] GENIVI Alliance. *GENIVI Compliant Products*. URL: http://genivi.org/compliant-products.

[55] GENIVI Alliance. *GENIVI Homepage*. URL: http://www.genivi.org.

[56] GENIVI Alliance. *Reference Architecture*. Okt. 2015.

[57] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss und P. Levis. „Collection Tree Protocol“. In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09)*. Berkeley, CA, USA, Nov. 2009.

[58] Gnuplot. *Gnuplot Homepage*. URL: http://www.gnuplot.info.

[59] Google. *Andriod Homepage*. URL: http://www.android.com/.

[60] A. Grzemba. *MOST: The Automotive Multimedia Network*. Electronics library. Franzis, 2008. ISBN: 9783772353161.

[61] A. Grzemba und H.-C. von der Wense. *LIN-Bus: Systeme, Protokolle, Tests von LIN-Systemen, Tools, Hardware, Applikationen*. Franzis Verlag, 2005.

[62] Gumstix, Inc. *Homepage*. URL: http://www.gumstix.com.

[63] W. Haberl. „Code Generation and System Integration of Distributed Automotive Applications“. Diss. Technische Universität München, 2011.

[64] Hamlin. *55100 Mini Flange Mount Hall Effect Sensor Features and Benefits*. März 2003.

[65] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz und D. Culler. „Flexible hardware abstraction for wireless sensor networks“. In: *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*. Jan. 2005, S. 145–157. DOI: 10.1109/EWSN.2005.1462006.

[66] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, D. Culler und D. Gay. *TEP 2: Hardware Abstraction Architecture*. TinyOS Core Working Group.

[67] J.-H. Hauer, P. Levis, V. Handziski und D. Gay. *TEP 101: Analog-to-Digital Converters (ADCs)*. TinyOS Core Working Group.

[68] M. Haunreiter und U. Baumgarten. „An embeddable RTOS“. In: *Embedded Systems Europe* 9.67 (Juni 2005), S. 27–29.

[69] S. Heath. *Embedded Systems Design*. 2nd. Newton, MA, USA: Butterworth-Heinemann, 2002. ISBN: 0750655461.

[70] G. Heiser. „Hypervisors for consumer electronics“. In: *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*. CCNC'09. Las Vegas, NV, USA: IEEE Press, 2009, S. 614–618. ISBN: 978-1-4244-2308-8.

[71] G. Heiser und B. Leslie. „The OKL4 microvisor: convergence point of microkernels and hypervisors“. In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. APSys '10. New Delhi, India: ACM, 2010, S. 19–24. ISBN: 978-1-4503-0195-4. DOI: 10.1145/1851276.1851282.

[72] J. Helander und A. Forin. „MMLite: A Highly Componentized System Architecture“. In: *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. EW 8. Sintra, Portugal: ACM, 1998, S. 96–103. DOI: 10.1145/319195.319210.

[73] J. N. Herder, H. Bos, B. Gras, P. Homburg und A. S. Tanenbaum. „MINIX 3: A Highly Reliable, Self-repairing Operating System“. In: *SIGOPS Oper. Syst. Rev.* 40.3 (Juli 2006), S. 80–89. ISSN: 0163-5980. DOI: 10.1145/1151374.1151391.

[74] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd. und Toshiba Corp. *Advanced Configuration and Power Interface Specifications*. Dez. 2011.

[75] D. Hildebrand. „An Architectural Overview of QNX“. In: *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. Berkeley, CA, USA: USENIX Association, 1992, S. 113–126. ISBN: 1-880446-42-1.

[76] M. Hillenbrand und K. Müller-Glaser. „An Approach to Supply Simulations of the Functional Environment of ECUs for Hardware-in-the-Loop Test Systems Based on EE-architectures Conform to AUTOSAR“. In: *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*. Juni 2009, S. 188–195. DOI: 10.1109/RSP.2009.14.

[77] P. C. Hruschka und C. Rupp. *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML*. Hanser, 2002.

[78] G. C. Hunt und J. R. Larus. „Singularity: rethinking the software stack“. In: *SIGOPS Oper. Syst. Rev.* 41.2 (2007), S. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424.

[79] G. C. Hunt, J. R. Larus, D. Tarditi und T. Wobber. „Broad New OS Research: Challenges and Opportunities.“ In: *HotOS*. 2005.

[80] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi und T. Wobber. „Sealing OS processes to improve dependability and safety“. In: *EuroSys '07: Proceedings of the 2007 conference on EuroSys*. Lisbon, Portugal: ACM Press, 2007, S. 341–354. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273032.

[81] IEEE. *IEEE 1003.1-2008 Standard for Information Technology - Portable Operating System Interface (POSIX(R))*. 2008.

[82] IEEE. „IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications“. In: *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)* (Juni 2007), S. 1–1076. DOI: 10.1109/IEEESTD.2007.373646.

[83] Intel Corp. und Microsoft Corp. *Advanced Power Management (APM): BIOS Interface Specification*. Feb. 1996.

[84] ISO. *ISO 11898-1:2003 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. 2003.

[85] ISO. *ISO 11898-2:2003 Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*. 2003.

[86] ISO. *ISO 11898-3:2006 Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface*. 2006.

[87] ISO. *ISO 11898-4:2004 Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication*. 2004.

[88] K. Klues, V. Handziski, J.-H. Hauer und P. Levis. *TEP 115: Power Management of Non-Virtualised Devices*. TinyOS Core Working Group.

[89] H. Kopetz, R. Obermaisser, C. El Salloum und B. Huber. „Automotive Software Development for a Multi-Core System-on-a-Chip“. In: *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS '07. Fourth International Workshop on*. Mai 2007, S. 2. DOI: 10.1109/SEAS.2007.2.

[90] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. ISBN: 0792398947.

[91] T. Landahl. „Entwicklung von automotive Anwendungen unter TinyOS und Evaluierung ihrer Auswirkungen auf den Energieverbrauch“. Bachelorarbeit. Technische Universität München, Apr. 2012.

[92] J. Larus und G. Hunt. „The Singularity system“. In: *Commun. ACM* 53 (8 Aug. 2010), S. 72–79. ISSN: 0001-0782. DOI: 10.1145/1787234.1787253.

[93] E. Lee. „Cyber Physical Systems: Design Challenges“. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. Mai 2008, S. 363–369. DOI: 10.1109/ISORC.2008.25.

[94] E. Lee. „What's ahead for embedded software?“ In: *Computer* 33.9 (Sep. 2000), S. 18–26. ISSN: 0018-9162. DOI: 10.1109/2.868693.

[95] G. Leen und D. Heffernan. „Expanding automotive electronic systems“. In: *Computer* 35.1 (Jan. 2002), S. 88–93. ISSN: 0018-9162. DOI: 10.1109/2.976923.

[96] P. Levis. *TEP 111: message_t*. TinyOS Core Working Group.

[97] P. Levis. *TEP 116: Packet Protocols*. TinyOS Core Working Group.

[98] P. Levis und D. Gay. *TinyOS Programming*. New York, NY, USA: Cambridge University Press, 2009. ISBN: 0521896061, 9780521896061.

[99] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer und D. Culler. „TinyOS: An operating system for sensor networks“. In: *Ambient Intelligence*. Springer Verlag, 2004.

[100] P. Levis und C. Sharp. *TEP 106: Schedulers and Tasks*. TinyOS Core Working Group.

[101] J. Liedtke. „On micro-kernel construction“. In: *SIGOPS Oper. Syst. Rev.* 29.5 (1995), S. 237–250. ISSN: 0163-5980. DOI: 10.1145/224057.224075.

[102] Linear Technology Corp. *LT1374: 4.5A, 500kHz Step-Down Switching Regulator*. 1998.

[103] Linear Technology Corp. *LT1767: Monolithic 1.5A, 1.25MHz Step-Down Switching Regulators*. 1999.

[104] P. Lowack, M. Gielesberger und S. Schenk. „Konzeption und Entwicklung einer automotive Applikation mit Anbindung an ein Power Management“. Interdisziplinäres Projekt. Technische Universität München, Apr. 2012.

[105] Maemo. *Homepage*. URL: http://maemo.org.

[106] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk und J. Anderson. „Wireless sensor networks for habitat monitoring“. In: *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. Atlanta, Georgia, USA: ACM, 2002, S. 88–97. ISBN: 1-58113-589-0. DOI: 10.1145/570738.570751.

[107] P. Marwedel. *Embedded System Design*. Bd. 1. Springer, 2003.

[108] MEMSIC, Inc. *MXA2300 - Ultra Low Cost, ± 1.5 g Dual Axis Accelerometer with Absolute Outputs*. Feb. 2007.

[109] A. Messdaghi und M. Schanzenbach. „Konzeption und Entwicklung eines Power Managements für die Energie-optimierte Ausführung von Applikationen unter TinyOS“. Interdisziplinäres Projekt. Technische Universität München, Apr. 2012.

[110] Miray Software AG. *Symobi Homepage*. URL: http://www.symobi.de.

[111] Moblin. *Homepage*. URL: http://www.moblin.com.

[112] N. Navet, A. Monot, B. Bavoux und F. Simonot-Lion. „Multi-source and multi-core automotive ECUs - OS protection mechanisms and scheduling“. In: *IEEE International Symposium on Industrial Electronics (ISIE)*. Juli 2010, S. 3734–3741. DOI: 10.1109/ISIE.2010.5637677.

[113] N. Navet, Y. Song, F. Simonot-Lion und C. Wilwert. „Trends in Automotive Communication Systems“. In: *Proceedings of the IEEE* 93.6 (Juni 2005), S. 1204–1223. ISSN: 0018-9219. DOI: 10.1109/JPROC.2005.849725.

[114] N. Navet und F. Simonot-Lion. *In-vehicle communication networks-a historical perspective and review*. Techn. Ber. University of Luxembourg, Aug. 2013.

[115] OpenEmbedded e. V. *Homepage*. URL: http://www.openembedded.org.

[116] OSEK/VDX. *OSEK/VDX Communication*. Version 3.0.3. Juli 2004.

[117] OSEK/VDX. *OSEK/VDX Network Management*. Version 2.5.3. Juli 2004.

[118] OSEK/VDX. *OSEK/VDX Operating System*. Version 2.2.3. Feb. 2005.

[119] OSEK/VDX Portal. *Homepage*. URL: http://portal.osek-vdx.org.

[120] F. Otto. „Erweiterung eines Systems zur Messung des Energieverbrauchs einer Automotive-Plattform“. Interdisziplinäres Projekt. Technische Universität München, Sep. 2012.

[121] F. Otto. „Konzeption und Implementierung einer Geschwindigkeits- und Fahrsequenzregelung unter TinyOS“. Bachelorarbeit. Technische Universität München, Aug. 2010.

[122] Philips Semiconductors. *The I2C-Bus Specification*. Jan. 2001. URL: http://www.nxp.com/documents/other/39340011.pdf.

[123] A. Pretschner, M. Broy, I. H. Kruger und T. Stauner. „Software Engineering for Automotive Systems: A Roadmap“. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, S. 55–71. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.22.

[124] A. O. S. Project. *Dalvik Technical Information*. URL: https://source.android.com/devices/tech/dalvik/.

[125] A. Pughat und V. Sharma. „A review on stochastic approach for dynamic power management in wireless sensor networks“. In: *Human-centric Computing and Information Sciences* 5.1 (2015), S. 1–14.

[126] R. Rajkumar, I. Lee, L. Sha und J. Stankovic. „Cyber-physical systems: The next computing revolution“. In: *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. Juni 2010, S. 731–736.

[127] F. Riemenschneider. „NVIDIAs Tegra-3-Prozessor kommt in zukünftigen Infotainment-Systemen und digitalen Kombiinstrumenten zum Einsatz“. In: *elektroniknet.de Automotive* (Jan. 2012).

[128] M. E. Russinovich, D. A. Solomon und J. Allchin. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Bd. 4. Microsoft Press Redmond, 2005.

[129] M. Scheer. „A Communication Protocol for the Propagation of Power States in a Distributed Automotive System: Design, Implementation, Evaluation“. Diplomarbeit. Technische Universität München, Aug. 2011.

[130] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak und R. Ernst. „System Level Performance Analysis for Real-Time Automotive Multicore and Network Architectures“. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.7 (Juli 2009), S. 979–992. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2013286.

[131] C. Schmutzler. „Hardwaregestützte Energieoptimierung von Elektrik/Elektronik-Architekturen durch adaptive Abschaltung von verteilten, eingebetteten Systemen“. Diss. Karlsruher Institut für Technologie, 2012.

[132] C. Schmutzler, A. Kruger, F. Schuster und M. Simons. „Energy efficiency in automotive networks: Assessment and concepts“. In: *International Conference on High Performance Computing and Simulation (HPCS)*. Juli 2010, S. 232–240. DOI: 10.1109/HPCS.2010.5547131.

[133] C. Schmutzler, A. Lakhtel, M. Simons und J. Becker. „Increasing energy efficiency of automotive E/E-architectures with Intelligent Communication Controllers for FlexRay“. In: *International Symposium on System on Chip (SoC)*. Nov. 2011, S. 92–95. DOI: 10.1109/ISSOC.2011.6089228.

[134] W. Schröder-Preikschat. „Automotive Betriebssysteme“. German. In: *Eingebettete Systeme*. Hrsg. von P. Holleczek und B. Vogel-Heuser. Informatik aktuell. Springer Berlin Heidelberg, 2004, S. 92–101. ISBN: 978-3-540-23424-1. DOI: 10.1007/978-3-642-18594-6_10.

[135] J. Shapiro und N. Hardy. „EROS: a principle-driven operating system from the ground up“. In: *Software, IEEE* 19.1 (2002), S. 26–33. ISSN: 0740-7459.

[136] C. Sharp, M. Turon und D. Gay. *TEP 102: Timers*. TinyOS Core Working Group.

[137] H.-J. Siegert und U. Baumgarten. *Betriebssysteme - Eine Einführung*. 6. Oldenbourg Verlag, 2006.

[138] G. Smethurst. *Changing the In-Vehicle Infotainment Landscape*. Techn. Ber. GENIVI Alliance, 2010.

[139] J. Smith und R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[140] S. Solomon, D. Qin, M. Manning, Z. Chen, M. Marquis, K. B. Averyt, M. Tignor und H. L. M. (eds.) „IPCC, 2007: Summary for Policymakers“. In: *Climate Change 2007: The Physical Science Basis, Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge, United Kingdom und New York, NY, USA: Cambridge University Press, 2007.

[141] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt und S. Levis. „Solving the starting problem: device drivers as self-describing artifacts“. In: *EuroSys '06: Proceedings of the 2006 EuroSys conference*. Leuven, Belgium: ACM Press, 2006, S. 45–57. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217941.

[142] SQLite. *Homepage*. URL: http://www.sqlite.org.

[143] J. Stankovic. „Misconceptions about real-time computing: a serious problem for next-generation systems“. In: *Computer* 21.10 (Okt. 1988), S. 10–19. ISSN: 0018-9162. DOI: 10.1109/2.7053.

[144] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey und B. Ellis. „VEST: an aspect-based composition tool for real-time systems“. In: *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*. Mai 2003, S. 58–69. DOI: 10.1109/RTTAS.2003.1203037.

[145] J. A. Stankovic. „VEST—A toolset for constructing and analyzing component based embedded systems“. In: *Embedded Software*. Springer. 2001, S. 390–402.

[146] M. M. Swift, B. N. Bershad und H. M. Levy. „Improving the reliability of commodity operating systems“. In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. Bolton Landing, NY, USA: ACM Press, 2003, S. 207–222. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945466.

[147] R. Szewczyk, P. Levis, M. Turon, L. Nachman, P. Buonadonna und V. Handziski. *TEP 112: Microcontroller Power Management*. TinyOS Core Working Group.

[148] A. S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780136006633.

[149] A. S. Tanenbaum, J. N. Herder und H. Bos. „Can We Make Operating Systems Reliable and Secure?“ In: *Computer* 39.5 (Mai 2006), S. 44–51.

[150] A. S. Tanenbaum und M. van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[151] Texas Advanced Optoelectronic Solutions. Inc. *TSL257 - High-Sensitivity Light-to-Voltage Converter*. Sep. 2007.

[152] Texas Instruments, Inc. *DRA74x OMAP Automotive Applications Processors Technical Brief*. Jan. 2013.

[153] Texas Instruments, Inc. *OMAP35x Applications Processor - Technical Reference Manual*. Sep. 2008.

[154] The European Parliament and the Council of the European Union. *Regulation No. 715/2007*. Juni 2007.

[155] TinyOS Community. *TinyOS Documentation Wiki: TEPs*. URL: http://tinyos.stanford.edu/tinyos-wiki/index.php/TEPs.

[156] Tizen. *Homepage*. URL: https://www.tizen.org.

[157] G. Tolle, P. Levis und D. Gay. *TEP 114: SIDs: Source and Sink Independent Drivers*. TinyOS Core Working Group.

[158] M. Triki, Y. Wang, A. Ammari und M. Pedram. „Hierarchical power management of a system with autonomously power-managed components using reinforcement learning“. In: *Integration, the VLSI Journal* 48 (2015), S. 10–20.

[159] G. Walla, A. Barthels, F. Ruf, R. Dörfel, H.-U. Michel, J. Fröschl, O. Sirch, U. Baumgarten, H.-G. Herzog und A. Herkersdorf. „Framework and Model for the Evaluation of Energy Efficiency of Partitioning Alternatives, Elektrik/Elektronik in Hybrid- und Elektrofahrzeugen und elektrisches Energiemanagement“. In: *Haus der Technik*. Apr. 2012, S. 151–158.

[160] G. Walla, A. Barthels, F. Ruf, R. Dörfel, H.-U. Michel, J. Fröschl, O. Sirch, U. Baumgarten, H.-G. Herzog, W. Stechele und A. Herkersdorf. „Aspects of Function Partitioning in Respect to Power Management“. In: *2nd International Energy Efficient Vehicles Conference (EEVC)*. Juni 2012, S. 25–35.

[161] G. Walla, A. Enger, A. Barthels, H.-U. Michel und A. Herkersdorf. „An Automotive Specific MILP Model Targeting Power-Aware Function Partitioning“. en. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. Juli 2014.

[162] G. Walla, D. Gabriel, A. Barthels, F. Ruf, H.-U. Michel und A. Herkersdorf. „ITE-Sim: A simulator and power evaluation framework for electric/electronic architectures“. In: *Vehicle Power and Propulsion Conference (VPPC), 2012 IEEE*. IEEE. 2012, S. 869–874.

[163] G. Walla, Z. Molotnikov, H.-U. Michel, W. Stechele, A. Barthels und A. Herkersdorf. „A Design Space Exploration Framework For Automotive Embedded Systems And Their Power Management“. In: *ECMS*. 2013, S. 228–234.

[164] H. Wallentowitz und K. Reif. *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen*. 2. Wiesbaden: Vieweg + Teubner, 2011.

[165] B. Wiesmüller. „Konzeption und Entwicklung eines Systems zur Messung des Energieverbrauchs einer automotive Plattform“. Interdisziplinäres Projekt. Technische Universität München, Juli 2011.

[166] M. Wolf. *High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 0124105114, 9780124105119.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Glossar

ACC Adaptive Cruise Control. 52

ACPI Advanced Configuration and Power Interface. 10, 104

ADC Analog-to-Digital Converter. 25, 31, 87

APM Advanced Power Management. 104

ASOS Application-Specific Operating System. 71

AUTOSAR Automotive Open System Architecture. 8, 23, 36

BIT Baseline Integration Team. 52

BSP Buffered Serial Port. 30

BSW Basic Software. 40

CAN Controller Area Network. 30

CASE Computer-aided Software Engineering. 45

CE Consumer Electronics. 50

CIL Common Intermediate Language. 68

CLI Common Language Infrastructure. 68

CLR Common Language Runtime. 68

CPS Cyber-Physical System. 3

CSMA/CA Carrier Sense Multiple Access/Collision Avoidance. 30

CSV Comma-Separated Values. 174

CTP Collection Tree Protocol. 90

DAB Drive Assistant Board. 113, 162

DLT Diagnostic Log and Trace. 47

DPM Dynamic Power Management. 11, 98

DSP Digital Signal Processor. 28

E/E-Architektur Elektrik/Elektronik Architektur. 14, 17

EAST-ADL Electronics Architecture and Software Technology - Architecture Description Language. 23

ECU Electronic Control Unit. 4, 24

EG Expert Group. 49

FCFS First come, first served. 81

GC Garbage Collector. 68

GPOS General-Purpose Operating System. 71

HAA Hardware Abstraction Architecture. 82

HAL Hardware Abstraction Layer. 64

HAL Hardware Adaptation Layer. 83

HIL Hardware Interface Layer. 83

HIP Hardware-Isolated Process. 69

HMI Human Machine Interface. 47

HPL Hardware Presentation Layer. 82

I^{2}C Inter-Integrated Circuit. 26, 30, 156, 160

ICOM Intelligent Communication Controller. 98

ICP Input Capture Pin. 148, 163

IDL Interface Description Language. 51

IPC Inter-Process Communication. 51, 65

ISP Image Signal Processor. 29

ISR Interrupt Service Routine. 81

IVI In-Vehicle Infotainment. 46

JTAG Joint Test Action Group. 26

LCD Liquid Crystal Display. 29

LED Light-Emitting Diode. 114

LIN Local Interconnect Network. 31

LSB Light and Sensor Board. 113, 163

MBP Manifest-Based Program. 69, 70

MCAL Microcontroller Abstraction Layer. 41

MDD Model-Driven Development. 23

MILP Mixed-Integer Linear Programming. 103

MMU Memory Management Unit. 26, 28

MOSFET Metal Oxide Semiconductor Field-Effect Transistor. 161

MOST Media Oriented Systems Transport. 31

MSIL Microsoft Intermediate Language. 68

NFC Near Field Communication. 50

OBD On-board Diagnostics. 19

OEM Original Equipment Manufacturer. 14

OSEK Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug. 33

OSEK COM OSEK/VDX Communication. 34

OSEK NM OSEK/VDX Network Management. 34

OSEK OS OSEK/VDX Operating System. 34

OSPM OS-based Power Management. 98

OSS Open-Source Software. 46, 62

PDA Personal Digital Assistant. 56

PID-Regler Proportional–Integral–Derivative Controller. 155

PLASA Platform for Automotive Systems and Applications. 8, 113

PM Power Manager. 11, 98, 100, 108

PMC Power Manageable Component. 100, 104

PMO Program Management Office. 49

PSB Power Supply Board. 113, 161

PSM Power State Machine. 100, 104

PWM Pulse-Width Modulation. 27, 33, 158, 164

RC-Car Radio Controlled Car. 158

RTE Runtime Environment. 39

SAR Successive Approximation Register. 28

SAT System Architecture Team. 51

SIP Software-Isolated Process. 69

SoC System on a Chip. 25

SPI Serial Peripheral Interface. 26, 30

SWC Software Component. 37

TCB Trusted Computing Base. 68

TDMA Time Division Multiple Access. 31

TEP TinyOS Enhancement Proposal. 73, 83

TT-CAN Time-triggered Controller Area Network. 30

UART Universal Asynchronous Receiver Transmitter. 29

UML Unified Modeling Language. 23

USART Universal Synchronous/Asynchronous Receiver Transmitter. 26

USB Universal Serial Bus. 29

USB-OTG Universal Serial Bus On-The-Go. 29

VDX Vehicle Distributed Executive. 33

VEST Virginia Embedded Systems Toolset. 72

VFB Virtual Functional Bus. 37

WSN Wireless Sensor Network. 2, 71, 73

ZGW Zentrales Gateway. 17

ZVEI Zentralverband Elektrotechnik- und Elektronikindustrie e. V. 2