

Multicore-Entwicklungsplattform für den Automobilbereich

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Angewandte Informatik

der Universität Augsburg



Universität
Augsburg
University

eingereicht von

Dipl.-Inf. Christian Bradatsch

MULTICORE-ENTWICKLUNGSPLATTFORM FÜR DEN AUTOMOBILBEREICH

Christian Bradatsch

Erstgutachter: Prof. Dr. Theo Ungerer

Zweitgutachter: Prof. Dr. Bernhard Bauer

Tag der mündlichen Prüfung: 23. Mai 2016

Kurzfassung

Heutzutage sind eingebettete Echtzeitsysteme allgegenwärtig, beispielsweise im Automobilsektor, in der Luftfahrt, in industriellen Kontrollsystemen oder in der Heimautomatisierung. Infolge des kontinuierlich steigenden Anspruchs an die Funktionalität, nimmt die Komplexität von Software weiterhin zu. Diese Entwicklung stellt immer höhere Anforderungen an die Rechenleistung eingebetteter Systeme. Nach der Leistungssteigerung durch Multicore-Prozessoren im Desktop- und Serverbereich haben Multicore-Systeme mittlerweile Einzug in die Domäne von eingebetteten Systemen gehalten. Allerdings besitzen eingebettete Systeme zumeist zusätzliche Anforderungen aufgrund ihrer Arbeitsumgebung und des Einsatzzwecks, u. a. Bedingungen an das Zeitverhalten von Softwareanwendungen.

Die vorliegende Arbeit hat zum Ziel, eine Entwicklungsplattform bereitzustellen, die eine Evaluierung von Techniken bezüglich der Kommunikation in eingebetteten Multicore-Systemen unter Berücksichtigung von Echtzeitanforderungen ermöglicht. Hierfür wurden eine mehrschichtige Softwarearchitektur und eine Kernel-Bibliothek entworfen, die im Rahmen des parMERASA Projekts exemplarisch im Automobilsektor, Luftfahrtbereich und in der Baumaschinenbranche eingesetzt wurde. Dabei wurde großes Augenmerk auf die Kommunikations- und Synchronisationsmechanismen gelegt, die eine obere Zeitschranke für Analysen der Programmausführungszeit erlauben. Zudem wurden Betriebssystemdienste, die zwischen Kernen in Multicore-Prozessoren vermitteln und speziell im Automobilbereich vorkommen, anhand zweier grundlegend verschiedener Techniken umgesetzt. Da diese Dienste zur Kommunikation und Synchronisation zwischen Prozessorkernen benötigt werden, lassen sich auf diese Weise Vor- und Nachteile beider Methoden im Bezug auf den Kommunikationsmehraufwand beurteilen.

Darüber hinaus werden verschiedene Echtzeitprogrammiermodelle kurz angerissen und im Besonderen auf die Vor- und Nachteile des Logical Execution Time (LET) Modells eingegangen. Ein Nachteil dabei ist, dass die Aktualität der Daten zwischen kommunizierenden Programmteilen merklich niedriger ist als bei anderen Modellen. Das letzte Hauptkapitel dieser Arbeit befasst sich indirekt mit der Erhaltung des Datenflusses bei der Portierung von Industrieanwendungen von Singlecore- auf Multicore-Systeme unter Verwendung des LET Modells. Im Speziellen wird eine Methode vorgestellt, die den Nachteil des hohen Datenalters mindert.

Abstract

Nowadays, embedded real-time control systems interacting with their environment are omnipresent, for example in automotive and avionic domains, in industrial control systems, or home automation. Due to continuously increasing demands on functionality, the complexity of software rises. This development demands for increasing computing power in embedded systems. After performance increase through multi-core processors in the desktop and server area, multi-core processors are present in the embedded system domain in the meantime. However, embedded systems have further requirements due to constraints on their environment and purpose. These are, amongst other things, constraints on the timing behaviour of applications.

The intention of this thesis is to provide a platform, which allows the evaluation of techniques concerning the communication in embedded multi-core systems in consideration of real-time requirements. Therefore, a layered software architecture in combination with a kernel library were developed. The concept was exemplarily approved within the parMERASA project by applying the architecture in the automotive, avionic, and construction machinery domain. The focus was on the communication and synchronisation mechanisms, which allow deriving an upper bound for timing analysis. Further, two fundamentally different techniques for operating system services were implemented, which are used for communication between different cores in a multi-core processor in the automotive domain. Since these services are required for communication and synchronisation between cores, it is possible to assess the advantages and disadvantages of both methods by means of the communication overhead.

Furthermore, different real-time programming models are shortly explained. Especially the advantages and disadvantages of the Logical Execution Time (LET) model are

highlighted. One disadvantage of the LET model compared to the other models is that the freshness of data between communicating tasks is obviously higher. The last main chapter addresses indirectly the preservation of data flow when porting industrial applications from single- to multi-core systems. In the specific case a method is presented, which reduces the disadvantage of the higher data age in the LET model.

Danksagung

An dieser Stelle möchte ich mich in erster Linie bei meinem Doktorvater Prof. Dr. Theo Ungerer für die hervorragende Betreuung dieser Dissertation bedanken. Ich möchte mich auch dafür bedanken, dass er mir eine Promotion ermöglicht hat. Ferner bin ich dankbar für die Auswahl und Mitarbeit in verschiedenen Projekten, durch die ich umfangreiche Erfahrungen sammeln konnte, im Speziellen im Automobilbereich, und viele nette Kollegen und Partner kennen lernen durfte. Dazu trug auch die Möglichkeit zur Teilnahme an bilateralen Gesprächen, Tagungen und Konferenzen bei.

Ebenso möchte ich meinen Dank Prof. Dr. Bernhard Bauer für die Zweitbegutachtung aussprechen.

Mein besonderer Dank gilt meinem Kollegen Dr. Florian Kluge, der mir bei zahlreichen Veröffentlichungen und Problemen mit Rat zur Seite stand und mich bei meinen Simulationen mit seinem Timing Models Scheduling Simulation Framework maßgeblich unterstützt hat.

Ich möchte mich auch bei all meinen Kollegen am Lehrstuhl herzlichst bedanken. Sei es für die konstruktive Kritik, die gemeinsamen Dienstreisen und Lehrstuhlseminare oder die unterhaltsamen Mittagessen, insbesondere am Fast-Food-Friday.

Zu guter Letzt danke ich meinem Vater, der mich während meines Studiums und meiner Promotion fortwährend unterstützt hat.

Augsburg, Im März 2016

Christian Bradatsch

Inhaltsverzeichnis

1	Einleitung	19
1.1	Aufbau und Ziele der Arbeit	20
2	Grundlagen	23
2.1	Eingebettete Echtzeitsysteme	23
2.1.1	Multicore-Systeme	26
2.2	Echtzeitbetriebssysteme	27
2.2.1	Prozesse	28
2.2.2	Threads	30
2.2.3	Scheduling	31
2.2.4	Speicherverwaltung	34
2.2.5	Interprozesskommunikation	35
2.2.6	Systemaufrufe	42
2.3	Betriebssysteme aus dem Automobilumfeld	43
2.3.1	OSEK/VDX	43
2.3.2	AUTOSAR	45
3	Generische Systemarchitektur für eingebettete Echtzeitsysteme	51
3.1	Zielsetzung und Überblick	51
3.2	Systemarchitektur	52
3.2.1	Anforderungen	53
3.2.2	Spezifikation	69
3.2.3	Fallstudien	76
3.2.4	Angrenzende Arbeiten	86
3.3	ACMC-OS	93
4	Untersuchung von Cross-Core Service Calls im Automobil-OS	99
4.1	OS-Dienste	100
4.2	Modell	101

4.3	Betrachtung des Zeitverhaltens	105
4.4	Implementierung	108
4.5	Evaluierung	110
4.6	Angrenzende Arbeiten	115
4.7	Diskussion	118
5	Verwendung von Echtzeit-Programmiermodellen im Automobilbereich	121
5.1	Task-Modell	122
5.2	Grundlagen	124
5.2.1	Datenalter	127
5.3	Erweitertes LET Modell	129
5.4	Wiederherstellung der Softwarekombinierbarkeit	131
5.5	Evaluierung	136
5.5.1	Zufällig generierte Task-Sets	137
5.5.2	Automobil-Benchmark	139
5.6	Angrenzende Arbeiten	141
5.7	Diskurs	144
6	Zusammenfassung und Ausblick	147
6.1	Zusammenfassung	147
6.2	Ausblick	149
	Literaturverzeichnis	151

Abbildungsverzeichnis

2.1	AUTOSAR Systemarchitektur	47
2.2	Detaillierte Ansicht der AUTOSAR Systemarchitektur	48
3.1	Allgemeiner Aufbau der parMERASA Systemarchitektur	53
3.2	Beispiel einer ARINC Systemarchitektur	58
3.3	Systemarchitektur einer exemplarischen Baumaschinenanwendung	62
3.4	Domänenübergreifender Systemarchitekturaufbau	64
3.5	Domänenübergreifende Systemarchitektur mit zweigeteilter RTE	71
3.6	AUTOSAR Systemarchitektur mit Kernel-Bibliothek	78
3.7	ARINC 653 Systemarchitektur mit Kernel-Bibliothek	81
3.8	ESX-3XL-BIOS Systemarchitektur mit Kernel-Bibliothek	85
3.9	Systemarchitektur nach Popovici (2009)	92
3.10	ACMC-OS Systemarchitektur	96
4.1	Generelle OS-Dienst Abarbeitungssequenz	100
4.2	Vergleich zweier Zugriffsmethoden bei einem Cross-Core Service Call auf <i>OS Data</i> in einer <i>Shared Memory</i> Architektur	102
4.3	Abarbeitungssequenz eines Cross-Core Dienstaufrufs beim nachrichtenba- sierten Verfahren	103
4.4	Abarbeitungssequenz eines Cross-Core Dienstaufrufs beim Lock-basierten Verfahren	104
4.5	Sender-/Receiver-Kommunikation zwischen Kernen	111
5.1	Typische Zeitparameter von Echtzeit-Tasks (Buttazzo 2011)	123
5.2	Echtzeitprogrammierung im Scheduled-Modell	125
5.3	Echtzeitprogrammierung im Timed-Modell	126
5.4	Ausführungszeitmodell mit Logical Execution Time (LET)	126
5.5	Exemplarische Ausführung zweier Tasks auf einem Singlecore-Prozessor bei Verwendung des BET Modells	127

5.6	Exemplarische Ausführung zweier Tasks auf einem Singlecore-Prozessor bei Verwendung des LET Modells	128
5.7	Exemplarische Ausführung zweier Tasks auf einem Singlecore-Prozessor bei Verwendung des LET-RTA Modells	130
5.8	Exemplarische Task-Ausführung in einer Multitasking-Umgebung mit harmonischen Task-Perioden und Kommunikation mit fast-to-slow Rate .	132
5.9	Exemplarische Task-Ausführung in einer Multitasking-Umgebung mit harmonischen Task-Perioden und Kommunikation mit slow-to-fast Rate .	134
5.10	Exemplarische Task-Ausführung in einer Multitasking-Umgebung mit harmonischen Task-Perioden bei Verwendung des LET-RTA Modells . . .	135
5.11	Mittelwerte der durchschnittlichen und maximalen Datenalter der interagierenden Tasks aller Task-Sets	138
5.12	Vergleich des Datenalters bei Verwendung des BET Modells und LET-RTA Modells	139
5.13	Mittelwerte der durchschnittlichen Datenalter aller interagierender Tasks aus dem Automobil-Benchmark	140

Tabellenverzeichnis

2.1	Unterteilung des Prozesskontextes in Hardware- und Systemkontext . . .	29
2.2	Vergleich der Komponenten unter Betriebssystemverwaltung je Prozess bzw. Thread (Tanenbaum 2009)	30
3.1	Gegenüberstellung der domänenspezifischen Anforderungen an die jeweilige Softwarearchitektur	65
3.2	Abhängigkeiten der RTE Basisdienste von anderen RTE Basis- und Kernel-Diensten	77
4.1	OS-Dienst-Ausführungszeiten in Prozessortakten bei einer Dualcore-Konfiguration	112
4.2	OS-Dienst-Ausführungszeiten in Prozessortakten bei einer Quadcore-Konfiguration	113
4.3	Mehraufwand der Quadcore- im Gegensatz zur Dualcore-Konfiguration .	114
5.1	Task-Perioden (T_i) und Worst-Case Response Times (WCRTs) (C_i) in Millisekunden des Automobil-Benchmarks	139
5.2	Logische Task-Ausführungszeiten in Millisekunden des Automobil-Benchmarks bei Verwendung des LET (T_i), Logical Execution Time (LET-RTA) (R_i) und LET-RTA Modells mit verlängerter Antwortzeit ($R_i + \Delta_i$) . . .	141

Abkürzungsverzeichnis

ABS	Antiblockiersystem
ACMC-OS	Automotive Compatible Multi-Core Operating System
ADV	Advance
APEX	APplication/EXecutive
API	Application Programming Interface
ARINC	Aeronautical Radio Incorporated
AUTOSAR	AUTomotive Open System ARchitecture
BET	Bounded Execution Time
BIOS	Basic Input Output System
BSP	Board Support Package
BSW	Basic Software
BswM	BSW Mode Manager
CDD	Complex Device Driver
COM	Communication
DRAM	Dynamic Random Access Memory
E-TDL	Extended Timing Definition Language
E/A	Eingabe und Ausgabe
E/E	Elektrik/Elektronik
ECU	Electronic Control Unit
EcuM	ECU State Manager
EDF	Earliest Deadline First

ESP	Elektronische Stabilitätskontrolle
FIFO	First In – First Out
FPP	Fixed-Priority Preemptive
HAL	Hardware Abstraction Layer
ICC	Inter-core Communication
ID	Identifier
IMA	Integrated Modular Avionics
IOC	Inter-OS-Application Communicator
IPC	Inter-process Communication
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LET	Logical Execution Time
LRU	Line Replaceable Unit
MCAL	Microcontroller Abstraction Layer
MCU	Micro Controller Unit
MMU	Memory Management Unit
MPCP	Multiprocessor Priority Ceiling Protocol
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
MPU	Memory Protection Unit
Mutex	Mutual Exclusion
NM	Network Management
NoC	Network-on-Chip
OEM	Original Equipment Manufacturer
OS	Operating System

OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
PCP	Priority Ceiling Protocol
POSIX	Portable Operating System Interface
RE	Runnable Entity
RMS	Rate Monotonic Scheduling
RTA	Response Time Analysis
RTE	Runtime Environment
SPS	Speicherprogrammierbare Steuerung
SW-C	Software Component
TDL	Timing Definition Language
TLB	Translation Lookaside Buffer
VDX	Vehicle Distributed eXecutive
VFB	Virtual Functional Bus
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

1

Einleitung

In den letzten Jahren ist die Anzahl der Kerne pro Prozessor kontinuierlich gestiegen. Multicore-Prozessoren sind heutzutage Standard im Desktop-Bereich und auch im Bereich der eingebetteten Systeme weit verbreitet. In eingebetteten Systemen steigt der Bedarf nach mehr Rechenleistung kontinuierlich, so z. B. im Luftfahrtbereich und in der Baumaschinenbranche. Dies gilt ebenso für den Automobilssektor, wo immer ausgefeiltere Fahrerassistenzsysteme Einzug nehmen und hochautomatisiertes bzw. autonomes Fahren im Vormarsch sind. Solche Systeme verlangen nach hoher Rechenleistung verbunden mit beschränktem Energieverbrauch und harschen Betriebsbedingungen. Neben der Anforderung nach mehr Rechenleistung zur Berechnung komplexer Aufgaben ist ein weiterer Grund die Konsolidierung vieler Einzelsysteme in einige zentrale elektronische Steuergeräte.

Im Zuge der Verbreitung von Multicore-Prozessoren wurden in verschiedenen Industriezweigen entsprechende Änderungen und Erweiterungen an etablierten Standards und Normen vorgenommen. Seit der Veröffentlichung der *AUTomotive Open System Architecture (AUTOSAR)* Release 4.0 Operating System Spezifikation (AUTOSAR 2011) fand die Unterstützung von Multicore-System Einzug in den Automobilstandard. Ebenso wurden Bemühungen im Luftfahrtbereich (Cullmann et al. 2010; Nowotsch et al. 2012) als auch im Industriebereich, etwa in der Baumaschinenbranche (Gerdes et al. 2011), unternommen, Multicore-Plattformen nutzbar zu machen.

Aufgrund der wachsenden Anforderungen an eingebettete Systeme, steigt auch zunehmend die Komplexität der Software. Zudem ist die Entwicklung von Software für eingebettete Systeme sehr stark Plattform abhängig. Darüber hinaus besitzen solche Systeme zusätzlich nicht-funktionale Anforderungen, u. a. Echtzeitbedingungen. Um dem Programmierer die Entwicklung zu erleichtern, werden schon seit langem Echtzeitbetriebssysteme verwendet. Diese gilt es für die Unterstützung von Multicore-Systemen zu erweitern und dabei die besonderen Eigenschaften von eingebetteten Echtzeitsystemen zu berücksichtigen.

1.1 Aufbau und Ziele der Arbeit

Ziel der Arbeit ist es, eine Evaluierungsplattform zu entwickeln, die den Anforderungen von parallelisierten bzw. parallel ausführbaren Anwendungen aus dem Automobilbereich gerecht wird. Besonderes Augenmerk wird hierbei auf eine effiziente Kommunikation und Synchronisation zwischen den Kernen eines Prozessors seitens des *Operating Systems* (OSs) ‘Betriebssystems’ gelegt. Zudem wird darauf geachtet, dass Betriebssystemkomponenten so umgesetzt sind, dass obere Zeitschranken bei einer zeitlichen Analyse des Programmablaufs abgeleitet werden können. Dies gilt insbesondere für Synchronisationsmechanismen, die nicht nur vom Betriebssystemkernel selbst verwendet werden, sondern auch Anwendungen über das Application Programming Interface (API) des Echtzeitbetriebssystems bereitgestellt werden. Dabei wird auf eine weitestgehende Kompatibilität zur OSEK OS (OSEK/VDX 2005) und AUTOSAR OS Spezifikation (AUTOSAR 2015b) geachtet.

Da die Entwicklungsplattform ihren Ursprung im parMERASA Projekt (Ungerer et al. 2013) fand, beruhen verschiedene Designentscheidungen auf grundlegenden Projektanforderungen. Das parMERASA Projekt hatte zum Ziel, Multicore-Plattformen für parallelisierte Anwendungen mit harten Echtzeitbedingungen zugänglich zu machen. Dabei war eine Herausforderung, Zeitanalysen für parallele Echtzeitprogramme zu ermöglichen. Eine weitere Anforderung bestand darin, Anwendungen aus den drei verschiedenen Industriezweigen, Automobilsektor, Luftfahrt und Baumaschinenbranche, auf derselben Zielplattform ausführen zu können. In Kapitel 3 werden die Anforderungen der einzelnen Industriezweige herausgearbeitet und die daraus resultierende System- und Softwarearchitektur der Entwicklungsplattform erläutert.

Aufgrund der Bestrebung, eine möglichst große Kompatibilität der API zum AUTOSAR Standard zu gewährleisten, wurden neben nicht konformen Erweiterungen zur Spezifikation im Speziellen auch Untersuchungen zu standardkonformen Techniken im Bereich der Inter-Core-Kommunikation durchgeführt. Kapitel 4 beschäftigt sich mit zwei Implementierungsvarianten von Systemaufrufen, die im Zusammenspiel mit der Kommunikation zwischen Kernen gemäß dem AUTOSAR Standard eine wichtige Rolle spielen. Als Systemaufrufe werden Aufrufe von Diensten bezeichnet, die vom Betriebssystem über dessen API zur Verfügung gestellt werden. Bei den gegenübergestellten Techniken handelt es sich um Systemaufrufe, die Betriebssystemdienste anfordern, welche von Betriebssystemkernels bereitgestellt werden, die auf anderen Kernen ausgeführt werden. Dabei kommt einerseits ein Lock- und andererseits ein nachrichtenbasierter Mechanismus zum Informationsaustausch zum Einsatz.

Da viele Industrieanwendungen über eine langjährige Zeit der (Weiter-)Entwicklung verfügen und es Ziel ist, diese Codebasis auf Multicore-Plattformen zu transferieren, sind Portierbarkeit und Zeit- und Wert-Determinismus bei Anwendungen von großem Belang. Hochlevel-Sprachen unterstützen den Programmierer bei der Umsetzung von funktionalen Anforderungen. Darüber hinaus gibt es verschiedene Programmiermodelle, die dem Entwickler bei nicht-funktionalen Erfordernissen behilflich sind. Dazu zählt das Logical Execution Time (LET) Modell, das die Zeitpunkte von Ein- und Ausgaben so anordnet, dass diese zeit- und wert-deterministisch sind. Allerdings bringt dieses Modell den Nachteil mit sich, dass das Alter von Daten dadurch erhöht wird. Dies kann wiederum negative Auswirkungen auf Latenzen mit Echtzeitbedingungen haben. In Kapitel 5 wird ein Ansatz zur Reduktion des Datenalters bei Verwendung des LET Programmiermodells vorgestellt. Somit ermöglicht die Entwicklungsplattform, den Datenfluss einer Anwendung zeitlich verfolgen zu können und somit einen Teil der Softwareentwicklung für Multicore-Systeme zu vereinfachen.

Kapitel 2 beschreibt die Grundlagen, die dem Verständnis der vorliegenden Arbeit dienen. Dabei wird eine Einführung in eingebettete Echtzeitsysteme gegeben und auf die zusätzlichen Anforderungen bei eingebetteten Multicore-Systemen eingegangen. Des Weiteren werden elementare Begrifflichkeiten und Mechanismen im Zusammenhang mit Echtzeitbetriebssystemen erläutert, wie etwa die Abgrenzung von Prozessen, Threads und Tasks. Den Schluss des Grundlagenkapitels bilden die Automobilstandards *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK)* und *AUTomotive*

Open System ARchitecture (AUTOSAR) und deren Bezug zu Echtzeitbetriebssystemen in Kombination mit Multicore-Prozessor-Unterstützung.

Kapitel 6 schließt die Arbeit mit einer kurzen Zusammenfassung und einem Ausblick auf weiterführende Forschungsziele ab.

2

Grundlagen

Das folgende Kapitel erläutert Begriffe und Techniken, die dem Verständnis der Arbeit dienen. Es bietet einen Abriss, der im Wesentlichen aus den Inhalten folgender Literatur besteht: *Parallelrechner und parallele Programmierung* (Ungerer 1997), *Echtzeitsysteme* (Wörn et al. 2005), *Eingebettete Systeme* (Marwedel 2007), *Mikrocontroller und Mikroprozessoren* (Brinkschulte et al. 2010) und *Moderne Betriebssysteme* (Tanenbaum 2009).

2.1 Eingebettete Echtzeitsysteme

Bei *eingebetteten Systemen* handelt es sich um informationsverarbeitende Systeme, die in einem technischen Umfeld integriert sind. Dies können beispielsweise Computer-Systeme in Fahrzeugen, Produktionsanlagen oder auch in ‘Weißer Ware’ sein. Dabei sind die Systeme für den Benutzer zumeist nicht sichtbar und eine direkte Benutzerschnittstelle ist größtenteils nicht vorhanden. Stattdessen interagieren eingebettete Systeme über Aktuatoren und Sensoren mit ihrem Umfeld. Ihre Aufgabe besteht darin, die kontinuierlich einfließenden Informationen zu verarbeiten und entsprechend darauf zu reagieren, indem die angebotenen Aktuatoren gestellt werden. Häufig werden eingebettete Systeme auch für Steuerungs- und Regelungsaufgaben in *sicherheitskritischen Systemen* eingesetzt,

wie z. B. in Flugzeugen oder Atomkraftwerken. Aufgrund der engen Interaktion mit ihrer Umgebung und den teilweise harschen Betriebsbedingungen, müssen eingebettete Systeme bestimmte Eigenschaften erfüllen:

- *Zuverlässigkeit* gibt die Wahrscheinlichkeit an, dass kein Fehler in einem bestimmten Zeitabschnitt auftritt.
- *Wartbarkeit* bezeichnet die Wahrscheinlichkeit, ein System nach einem Fehler in einen funktionierenden Zustand zu bringen.
- *Verfügbarkeit* beschreibt die Zeit, in der ein System verfügbar, sprich lauffähig ist. Eingebettete Systeme können u. U. nicht ohne weiteres für Reparaturzwecke heruntergefahren werden bzw. sind dafür schlecht zugänglich.
- *Sicherheit* beschreibt die korrekte Funktion in einem sicherheitskritischen System und durchzuführende Maßnahmen im Fehlerfall, um das System in einen sicheren Zustand zu überführen.
- *Integrität* bezeichnet die Echtheit von Daten in einem System und welche Eigenschaften ein System besitzt, um unautorisierte Veränderungen an System und Daten zu verhindern.

Neben diesen Eigenschaften müssen eingebettete Systeme auch *effizient* sein, aufgrund von beengten Verhältnissen oder geringer zur Verfügung stehender elektrischer Energie, um nur zwei Beispiele zu nennen. Infolge dessen werden in vielen Bereichen Mikrocontroller eingesetzt, da diese einerseits auf Energiesparsamkeit ausgelegt sind und auf der anderen Seite bereits viele Peripheriefunktionen integrieren, wodurch die Anzahl zusätzlicher elektronischer Bauteile verringert werden kann. Dies hat allerdings auch zur Folge, dass sowohl die Rechenleistung als auch der nutzbare Programm- und Datenspeicher begrenzt sind.

Ein weiterer wichtiger Gesichtspunkt in eingebetteten Systemen ist der Begriff *Echtzeit*, der besagt, dass Aufgaben innerhalb einer vorgegebenen Zeitspanne erledigt sein müssen, um ihre korrekte Funktion zu gewährleisten. Dabei spielt die Schnelligkeit der Programmausführung eine untergeordnete Rolle. Wichtig hierbei ist, dass die Aufgabe garantiert bis zu einem vorgegebenen Zeitpunkt fertiggestellt ist. Daraus ergeben sich

für *eingebettete Echtzeitsysteme* weitere Anforderungen, die sie erfüllen müssen. Dabei handelt es sich um folgende Eigenschaften:

- *Rechtzeitigkeit* bedeutet, dass Ergebnisse bzw. Ausgabedaten rechtzeitig, bis zu einem bestimmten Zeitpunkt zur Verfügung stehen. Diese Zeitpunkte werden durch Zeitschranken, auch Deadlines genannt, vorgegeben.
- *Gleichzeitigkeit* beschreibt die Eigenschaft, mehrere Ereignisse parallel zu bearbeiten und für jede Aktion die Ausgabedaten rechtzeitig bereitzustellen. Dabei wird zwischen *echt-paralleler* und *quasi-paralleler* Ausführung unterscheiden. Bei *echt-paralleler* Ausführung werden alle Aktionen zeitgleich von mehreren Ausführungseinheiten bearbeitet, wohingegen sich bei *quasi-paralleler* Ausführung alle Aufgaben eine Ausführungseinheit teilen und anteilig Zeitscheiben an der Rechenzeit zugeteilt bekommen. Durch den schnellen Wechsel der Aufgaben entsteht der Eindruck, dass diese parallel ausgeführt werden.

Je nach Strenge der einzuhaltenden Zeitbedingungen kann zwischen weichen, festen und harten Echtzeitschranken unterschieden werden.

- Bei *weicher Echtzeit* ist ein gelegentliches Überschreiten der Zeitschranke tolerierbar. Hierbei handelt es sich eher um eine Richtlinie als um eine strikte Notwendigkeit.
- Bei *fester Echtzeit* ist ein Ergebnis nach Überschreiten der Zeitschranke wertlos. Die Folgen sind jedoch nicht unmittelbar katastrophal.
- Bei *harter Echtzeit* muss die Zeitschranke unter allen Umständen eingehalten werden, ansonsten kann das System Schaden nehmen.

Für feste und vor allem für harte Echtzeitsysteme ist es unabdingbar, die maximale Laufzeit oder eine obere Schranke für die Laufzeit eines Programms oder Programmteils auf einer bestimmten Hardwareplattform ermitteln zu können. Die Laufzeit eines Programms hängt einerseits von seinen Eingabewerten ab und zum anderen beeinflussen verschiedene Hardwarearchitekturmerkmale, wie etwa komplexe Out-of-Order Pipelines, Caches, oder andere spekulative Komponenten, die Laufzeit, die im ungünstigsten, anzunehmenden Fall eintreten kann (Wilhelm et al. 2008). Hierbei spricht man von der *Worst-Case Execution Time (WCET)* oder der ‘Laufzeit im schlimmsten Fall’. Mit Hilfe der WCET kann festgestellt werden, ob die Ausführung einer Aufgabe auf einem harten Echtzeitsystem

mit Sicherheit vor der festgelegten Zeitschranke beendet ist. In der Praxis ist es mit zunehmender Komplexität eines Systems schwer bis überhaupt nicht möglich, die WCET eines Programms zu bestimmen. Daher werden Annäherungen oder obere Schranken an die tatsächliche WCET durch *Laufzeitanalysen* mit Hilfe von Softwarewerkzeugen ermittelt.

2.1.1 Multicore-Systeme

Wie bereits erwähnt, ist eine Voraussetzung von eingebetteten Systemen, dass diese effizient sind. Dazu zählt unter anderem auch Energiesparsamkeit. Daher ist es wichtig, dass die Rechenleistung pro elektrischer Leistung (Watt) sehr hoch ist, um Aufgaben mit möglichst geringem Energieverbrauch zu bewerkstelligen. Um die Rechenleistung bei Universalprozessoren zu erhöhen, wurden in der Vergangenheit verschiedene Verfahren angewandt: Zum einen durch Erhöhung der Taktfrequenz, und zum anderen durch Anpassung der Mikroarchitektur, wie etwa mehrstufige Cache-Hierarchien oder durch Hinzufügen weiterer spekulativer Techniken in die Prozessorpipeline. Beide Verfahren führen allerdings zu einer Verringerung der Rechenleistung pro Watt, wenn man die Verkleinerung der Strukturgröße beim Herstellungsprozess außer Acht lässt. Daher sind diese Techniken für eingebettete Systeme eher kontraproduktiv. Eine andere Möglichkeit die Rechenleistung zu erhöhen besteht darin, weitere Rechen- bzw. Prozessorkerne hinzuzufügen. Mittlerweile (Stand 2015) sind solche Multicore-Prozessoren in eingebetteten Systemen weit verbreitet. Für eingebettete Echtzeitsysteme ergeben sich beim Einsatz von Multicore-Prozessoren weitere Herausforderungen. Im Gegensatz zu Singlecore-Prozessoren teilen sich in Multicore-Systemen die einzelnen Kerne einen gemeinsamen Hauptspeicher und Peripheriegeräte. Der Zugriff auf Speicher und Geräte erfolgt hierbei meist über einen gemeinsamen System- bzw. Peripheriebus. Um gleichzeitige Zugriffsanfragen auf den Bus zu entscheiden, wird ein Busarbiter eingesetzt. Dieser ermöglicht einen geordneten Zugriff auf die einzelnen Geräte. Intern arbeitet der Arbiter mit einem bestimmten (Prioritäts-)Schema, um den Zugriff zu regeln. Je nach Verfahren kann dies enorme Auswirkungen auf die Bestimmung der WCET haben. Sofern die Arbitrierungslogik nicht fair ist, kann nicht ermittelt werden, zu welchem Zeitpunkt bzw. in welcher Zeitspanne ein Prozessorkern Zugriff auf den Bus bekommt. Somit ist eine obere Schranke nur schwer oder überhaupt nicht bestimmbar. Selbst bei fairen Arbitrierungsmechanismen kann

nicht vorhergesagt werden, ob beispielsweise ein Speicherzugriff sofort gewährt wird. Dies ist abhängig davon, ob der entsprechende Kern gerade den Zugriff auf den Bus erhält, oder ob dieser im ungünstigsten Fall warten muss, bis alle anderen Kerne ihre Buszugriffe beendet haben. Ein weiteres Problem bei Zugriffen auf DRAM Speicher ist, dass die Zugriffslatenz aufgrund der notwendigen Wiederauffrischung der Speicherzellen nicht konstant ist. In (Ungerer et al. 2010) wird u. a. eine Lösung für Bus- und Speichercontroller vorgestellt, um engere Schranken bei der Laufzeitanalyse zu erhalten. Ein weiterer Gesichtspunkt bei Multicore-Systemen ist die Notwendigkeit, Prozesse oder Programmsequenzen zu synchronisieren, um somit den Programmablauf im gesamten System zu steuern. Hierfür werden Synchronisationsmechanismen verwendet, welche atomare Instruktionen voraussetzen. Diese Instruktionen garantieren eine unterbrechungsfreie Ausführung von Anweisungen. Ein Beispiel hierfür sind sogenannte read-modify-write Befehle, die einen Wert von einer Speicherstelle lesen, ihn verändern und anschließend an dieselbe Speicherstelle zurückschreiben. Bezüglich der Zeitanalyse ergeben sich zwei Herausforderungen. Zum einen blockieren viele atomare Instruktionen den Bus für mehrere Takte, wodurch eine sichere obere Zeitschranke überschätzt wird. Zum anderen müssen die softwareseitigen Synchronisationsmechanismen fair sein, so dass jeder Teilnehmer in einer bestimmten Zeitspanne garantiert einmal den Zugriff erhält. In (Gerdes et al. 2012b) werden Techniken für zeit-analysierbare Synchronisationsmechanismen vorgestellt. Die Schwierigkeit bei Multicore-Systemen mit Echtzeitanforderungen liegt darin, dass aufgrund dessen, dass die Ressourcen geteilt sind, eine obere Zeitschranke teils stärker überschätzt wird als bei Singlecore-Plattformen.

2.2 Echtzeitbetriebssysteme

Um die Komplexität der Echtzeitprogrammierung zu handhaben, werden in eingebetteten Systemen *Echtzeitbetriebssysteme* eingesetzt, die neben den Standardaufgaben eines Betriebssystems noch zwei weitere grundlegende Aufgaben übernehmen. Dabei handelt es sich einerseits um die Einhaltung der Rechtzeitigkeit und Gleichzeitigkeit und andererseits um die Wahrung der Verfügbarkeit. Nachfolgend werden die einzelnen Betriebssystemaufgaben und die Anpassungen für Echtzeitbetriebssysteme beschrieben.

2.2.1 Prozesse

Bei einem Prozess handelt es sich um ein in der Ausführung befindliches Programm. Da ein Programm aus Anweisungen besteht, die nacheinander abgearbeitet werden, spricht man auch von sequentiellen Prozessen. Zweck des Betriebssystems ist es, mehrere gleichzeitig zu bearbeitende Aufgaben bzw. Programme zu koordinieren und auszuführen. Hierfür wird jedem Programm nur ein kleines Intervall an der Rechenzeit gewährt und das Betriebssystem wechselt ständig zwischen den Prozessen, indem es den laufenden Prozess unterbricht und einen anderen Prozess startet oder fortführt. Die Prozesse werden hierbei nebenläufig ausgeführt. Da sich meist mehrere parallele Prozesse einen Prozessor teilen müssen, spricht man auch von Quasi-Parallelität. Aufgrund der Möglichkeit Prozesse zu wechseln, besitzt ein Prozess unterschiedliche Zustände:

- Bereit (*ready*):
Ein Prozess besitzt alle benötigten Ressourcen zur Ausführung außer dem Prozesskern.
- Aktiv, laufend (*running*):
Ein Prozess wird ausgeführt und die Rechenzeit ist ihm exklusiv zugeteilt.
- Blockiert, wartend (*blocked, waiting*):
Ein Prozess wurde unterbrochen und wartet zur Fortsetzung seiner Ausführung auf ein Ereignis.
- Ruhend (*dormant, suspended*):
Ein Prozess ist im System vorhanden, allerdings ist er noch nicht zur Ausführung freigegeben.

Der Zustand *ruhend* kommt meist nur in eingebetteten (Echtzeit-)Systemen vor. In Standardbetriebssystemen wird ein Prozess stattdessen auf Anfrage erzeugt und nach Beendigung zerstört, d. h., der benötigte Speicher für den Prozess wird dynamisch zugewiesen und am Ende freigegeben. In vielen Echtzeitsystemen hingegen wird der Speicher für Prozesse und andere Betriebssystemelemente statisch reserviert (siehe Abschnitt 2.2.4). Deshalb wird ein Prozess nach seiner Terminierung in den Zustand ruhend versetzt. Ein weiterer Grund ist, dass Prozesse in Echtzeitsystemen zyklisch ausgeführt werden. Zwischen zwei Ausführungen ist ein Prozess ebenfalls ruhend.

Um Zustandswechsel durchführen zu können, verwaltet das Betriebssystem eine *Prozessstabelle*, in der für jeden Prozess Daten in einem *Prozesskontrollblock* gespeichert werden. Diese Informationen sind nötig, um einen Prozess in demselben Kontext an dem Befehl fortzusetzen, an der dieser unterbrochen wurde. Dabei bezeichnet ein *Prozesskontext* alle benötigten Daten zur Verwaltung eines Prozesses, die im Prozesskontrollblock abgelegt sind. Der Prozesskontext lässt sich weiter in den *Hardware-* und *Systemkontext* unterteilen. Tabelle 2.1 listet die Informationen auf, die zum Prozesskontext zählen.

Tabelle 2.1: Unterteilung des Prozesskontextes in Hardware- und Systemkontext

Systemkontext	Hardwarekontext
<ul style="list-style-type: none"> • Prozess-ID • Prozesszustand • Scheduling-Parameter • Signale und Ereignisse • Alarme • Dateideskriptoren 	<ul style="list-style-type: none"> • Prozessorregister (u. a. allgemeine Register, Befehlszähler, Stapelzeiger, Statusregister) • Speicherverwaltungsregister

Der Vorgang, bei dem von einem Prozess zu einem Anderen gewechselt wird, nennt man *Kontextwechsel*. Ein Kontextwechsel wird entweder durch eine *Exception* ‘Ausnahme’ oder einen *Interrupt Request (IRQ)* ‘Unterbrechung’ ausgelöst. Beispielsweise löst ein Zeitgeber eine Unterbrechung aus, um einen Prozess nach einem bestimmten Zeitintervall in den Zustand wartend zu versetzen und einen anderen Prozess fortzuführen. Ein Beispiel für eine Ausnahme ist ein *System Call* ‘Systemaufruf’, der den laufenden Prozess unterbricht, um zu einem Betriebssystemprozess zu wechseln. Bei einem Kontextwechsel wird der Prozesskontext des aktiven Prozesses gesichert, durch einen *Scheduler* ‘Planer’ der neue zu aktivierende Prozess ermittelt, der Prozesskontext des neuen Prozesses wiederhergestellt und die Ausführung des neuen Prozesses an seinem zuletzt ausgeführten Befehl fortgeführt. Falls der neue Prozess zum ersten Mal ausgeführt wird, wird mit der Abarbeitung des ersten Befehls begonnen.

2.2.2 Threads

Ein Nachteil von Prozessen ist, dass Kontextwechsel teuer sind, bezogen auf die dafür benötigte Anzahl an Prozessortakten. Dies liegt darin begründet, dass bei einem Kontextwechsel der Cache teilweise oder ganz zurückgeschrieben und das *Mapping* ‘Abbildung’ der Speicherverwaltungsregister geändert werden müssen. Diesen Nachteil adressiert das *Thread*-Konzept. Dabei ist jeder Thread genau einem Prozess zugewiesen und Teil von diesem. Ferner teilt sich ein Thread denselben Adressraum mit allen anderen Threads des Prozesses. Der Adressraum gibt an, welche Adressen einem Prozess zur Verfügung stehen und somit auch dessen Threads. Threads werden auch als *leichtgewichtige Prozesse* bezeichnet, da sie sich Betriebsmittel und den Adressraum teilen und dadurch ein Kontextwechsel schneller vonstattengeht. Vor allem durch den gemeinsam genutzten Adressraum werden teure Translation Lookaside Buffer (TLB) und Cache Flushes vermieden. Tabelle 2.2 stellt die Elemente gegenüber, die sich alle Threads teilen und jene, die Thread-eigen sind. Eines der wichtigsten Merkmale ist, dass ein Thread seinen

Tabelle 2.2: Vergleich der Komponenten unter Betriebssystemverwaltung je Prozess bzw. Thread (Tanenbaum 2009)

Elemente pro Prozess	Elemente pro Thread
<ul style="list-style-type: none"> • Prozess-ID • Adressraum (Code-/Daten-Segment, Heap-Speicher) • Dateideskriptoren 	<ul style="list-style-type: none"> • Thread-ID • Prozessorregister • Stack-Speicher • Zustand • Scheduling-Parameter • Signale und Ereignisse

eigenen Stack-Speicher besitzt. Sonstige Speichersegmente sind allen Threads gemein. In Analogie zum Prozesskontrollblock werden die Thread-eigenen Elemente in einem *Thread-Kontrollblock* verwaltet. Jeder Thread hat seinen eigenen Zustand und kann analog zu Prozessen einen der vier Zustände annehmen. Ebenso besitzt jeder Thread seine separaten Scheduling-Parameter und Signal- und Ereignismasken. Der *Portable Operating System*

Interface (POSIX) Standard (IEEE Computer Society 2008) etwa definiert, dass jeder Prozess im Grunde mindestens einen (Haupt-)Thread beinhaltet. Sofern der Prozess nur aus einem Thread besteht, sind Prozess-ID und Thread-ID identisch. Andernfalls besitzt jeder Thread eine eindeutige Thread-ID und alle Threads eines Prozesses dieselbe Prozess-ID.

2.2.3 Scheduling

Ein Betriebssystem, das eine nebenläufige Abarbeitung mehrerer Aufgaben unterstützt, wird auch als multitaskingfähig bezeichnet. In einem Multitasking-Betriebssystem konkurriert oft eine Menge von Aufgaben, auch *Tasks* genannt, gleichzeitig um die Rechenzeit des Prozessors. Dabei ist es in einem Betriebssystem Aufgabe des *Schedulers* zu entscheiden, welcher Task als nächstes ausgeführt wird. Zur Planung des Task-Ablaufs verwendet der Scheduler einen Scheduling-Algorithmus. Allgemein formuliert stellen Tasks Einheiten dar, die ein Programmcodestück in einem bestimmten *Umgebungskontext* ausführen. Dabei kann ein Task in seiner Ausführung auch beliebig oft unterbrochen werden. Der Umgebungskontext eines Tasks muss die entsprechenden Ressourcen zur Ausführung des Programmcodes bereitstellen. Hierbei handelt es sich in der Regel um einen *Prozess* oder *Thread*.

Ein Scheduling-Verfahren versucht den Ablauf hinsichtlich bestimmter Kriterien zu optimieren, z. B. Fairness, Durchsatz, Effizienz etc. Während Fairness bei allen Systemen gefordert ist, sind bei Echtzeitsystemen die Einhaltung von Deadlines und zeitliche Vorhersagbarkeit die Hauptkriterien. Bei harten Echtzeitsystemen bedeutet dies, dass das Verhalten jedes Tasks schon vorher bekannt ist und somit garantiert werden kann, dass jeder Task seine Deadline einhält. Dabei ist es Aufgabe des Schedulers, beim Auftreten von *Ereignissen* die Tasks so zu planen, dass alle Deadlines eingehalten werden. Ereignisse können wie folgt kategorisiert werden:

- Periodisch:
Ein Ereignis tritt in regelmäßigen Zeitabständen auf.

- Sporadisch:
Zwischen dem unregelmäßigen Eintreten eines Ereignisses kann ein minimales Zeitintervall und somit die Auftrittshäufigkeit bestimmt werden.
- Aperiodisch:
Ein Ereignis tritt zu einem unvorhersehbaren Zeitpunkt auf.

Ein weiteres Merkmal von Tasks in eingebetteten Systemen ist, dass deren Laufzeit im Allgemeinen nur einige Millisekunden beträgt, diese dafür jedoch wiederholt (periodisch) ausgeführt werden. Des Weiteren können Scheduling-Verfahren nach folgenden Merkmalen gruppiert werden:

- Kooperativ/präemptiv:
Bei Ersterem entscheidet der Prozess selbst, wann er seine Ressourcen freigibt und somit die Ausführung eines anderen Prozesses ermöglicht. Bei präemptiven Scheduling-Verfahren können einem Prozess seine Ressourcen noch vor Beendigung entzogen werden, indem dieser unterbrochen und in den Zustand *bereit* versetzt wird. Zu einem späteren Zeitpunkt wird die Ausführung des unterbrochenen Prozesses fortgesetzt.
- Statisch/dynamisch:
Beim statischen Verfahren wird im Vorfeld eine Zuordnungstabelle berechnet, in deren Reihenfolge die Tasks ausgeführt werden. Beim dynamischen Scheduling-Verfahren wird zur Laufzeit entschieden, welcher Task als nächstes ausgeführt wird.

Bei den dynamischen Scheduling-Verfahren erfolgt die Auswahl des nächsten Tasks zudem anhand von Prioritäten, die wie folgt festgelegt werden:

- Statische Prioritäten:
Die Prioritäten werden im Vorfeld festgelegt und zur Laufzeit nicht verändert.
- Dynamische Prioritäten:
Die Prioritäten werden zur Laufzeit vom Scheduler anhand von bestimmten Task-Kriterien berechnet.

- Keine Prioritäten:

Die Zuteilung der Tasks erfolgt ohne Prioritäten. Stattdessen werden andere Aspekte herangezogen, z. B., Reihenfolge und Ablaufbereitschaft beim Round-Robin-Algorithmus.

Im Folgenden werden die Unterscheidungsmerkmale von Scheduling-Algorithmen anhand dreier, häufig in Echtzeitbetriebssystemen eingesetzter Verfahren beschrieben.

Round-Robin Scheduling Hierbei werden die für den Ablauf bereiten Tasks in einer Liste angeordnet und *rundherum* der Reihe nach ausgeführt. Tasks, die nicht bereit sind, werden aus der Liste entfernt und können zu einem beliebigen Zeitpunkt wieder eingefügt werden. Das Verfahren ist *dynamisch* und *nicht prioritätsbasiert*, da die Wahl des nächsten Tasks abhängig ist von den aktuell bereiten Tasks und der Listenreihenfolge. Zudem wird jedem Task ein Zeitintervall zugewiesen, auch Quantum genannt, in dem dieser ausgeführt werden kann. Sofern ein Task am Ende seines Quantums die Abarbeitung seiner Aufgabe noch nicht abgeschlossen hat, wird dieser unterbrochen (*präemptiv*) und der nächste Task in der Liste wird ausgeführt.

Fixed-Priority Preemptive Scheduling Beim *FPP Scheduling* handelt es sich um ein *präemptives, dynamisches* Scheduling-Verfahren mit *statischen Prioritäten*. In Echtzeitbetriebssystemen werden die Prioritäten zumeist nach dem *rate-monotonic* 'raten-monotonen' Verfahren den Tasks zugewiesen. Dabei werden die Prioritäten umgekehrt proportional zur Task-Periode vergeben. Im Allgemeinen spricht man bei FPP Scheduling mit raten-monotoner Vergabe der Prioritäten von *Rate Monotonic Scheduling (RMS)*. Beim Aufruf des Schedulers ermittelt dieser den Task mit der aktuell höchsten Priorität und veranlasst dessen Ausführung.

Earliest Deadline First Scheduling Beim *EDF Scheduling* werden beim Aufruf des Schedulers die *Task-Prioritäten dynamisch* anhand des Intervalls vom aktuellen Zeitpunkt bis zur jeweiligen Task-Deadline ermittelt. Der Task mit der *frühesten Deadline*, bzw. der das kürzeste Intervall besitzt, wird *dynamisch* als nächstes (*zuerst*) zur Ausführung ausgewählt. Ferner ist das Verfahren *präemptiv*, da der laufende Task jederzeit unterbrochen werden kann, um einen Task mit einer früheren Deadline auszuführen.

2.2.4 Speicherverwaltung

Zur Aufgabe eines Standardbetriebssystems gehören die Verwaltung des Adressraums und Speichers. Dabei ist der physikalische Speicher auf Adressen im physikalischen Adressraum abgebildet. Bei einem 32-Bit-Prozessor umfasst der physikalische Adressraum 2^{32} Byte. Dabei steht der Adressraum nicht ausschließlich für Speicher zur Verfügung. Ebenso werden Peripheriegeräte über Adressen bzw. Adressbereiche angesprochen. Das Betriebssystem weist jedem Prozess seinen eigenen Adressraum zu. Unter Umständen können sich Adressräume verschiedener Prozesse auch überschneiden. Die Abbildung eines Prozessadressraums auf den physikalischen Adressraum kann verschieden sein. Dies hängt davon ab, ob ein Prozessor eine *Memory Protection Unit (MPU)* ‘Speicherschutzseinheit’ oder *Memory Management Unit (MMU)* ‘Speicherverwaltungseinheit’ besitzt und wie die jeweilige Einheit konfiguriert ist.

Im einfachsten Fall besitzt ein Prozessor weder eine MPU noch MMU. Hierbei entspricht der zugewiesene Prozessadressraum den Adressen im physikalischen Adressraum. Sofern ein Prozess auf Adressen zugreift, die ihm nicht zugewiesen wurden, erfährt das Betriebssystem nichts davon. Somit kann ein Prozess die Daten eines anderen Prozesses unbemerkt manipulieren. Dieser Ansatz ist für eingebettete Systeme mit sicherheitskritischen Anforderungen nicht tragbar. An dieser Stelle greift die MPU ein. Hierbei können verschiedenen Adressbereichen unterschiedliche Zugriffsrechte vergeben werden. Bei einer Zugriffsverletzung löst die MPU eine Ausnahme aus und das Betriebssystem kann entsprechend darauf reagieren. Einen Schritt weiter geht die MMU. Neben der Vergabe von Zugriffsrechten ermöglicht die MMU virtuelle Adressierung. Hierbei weist das Betriebssystem jedem Prozess einen virtuellen Adressraum zu. Die MMU bildet alle Adressen von jedem virtuellen Adressraum auf physikalische Adressen ab. Somit können verschiedene Prozesse jeweils den gleichen virtuellen Adressraum nutzen, da die virtuellen Adressen auf unterschiedliche physikalische Adressen abgebildet werden. Ferner ist es möglich, einen linearen virtuellen Adressraum auf mehrere fragmentierte physikalische Adressbereiche abzubilden, z. B.: Einem Prozess ist der lineare Adressraum $0x0000 - 0x1FFF$ zugewiesen. Die MMU bildet die virtuellen Adressen $0x0000 - 0x0FFF$ bzw. $0x1000 - 0x1FFF$ auf die physikalischen Adressen $0x4000 - 0x4FFF$ und $0x7000 - 0x7FFF$ ab.

Ein weiterer Vorteil der virtuellen Adressierung besteht darin, dass zu Beginn nicht für jeden virtuellen Adressbereich eines Prozesses ein physikalischer Adressbereich bzw. Speicherbereich zugewiesen sein muss. Dadurch kann der Speicher für einen Prozess zur Laufzeit vergrößert werden, beispielsweise wenn ein Prozess Heap-Speicher reserviert. Zur schnelleren Übersetzung von virtuellen zu physikalischen Adressen ist eine MMU meist mit einem Translation Lookaside Buffer (TLB) kombiniert.

Sofern keine MPU oder MMU vorhanden sind oder diese vom Betriebssystem nicht verwendet werden, kann eine Unterscheidung zwischen Prozessen und Threads vernachlässigt werden. Dies liegt darin begründet, da der Vorteil des schnelleren Kontextwechsels von Threads nicht mehr zum Tragen kommt. Da keine MMU vorhanden ist, entfallen die TLB Flushs. Ebenso müssen keine Cache-Zeilen invalidiert oder der Cache zurückgeschrieben werden, da der physikalisch adressierte Speicher zwischengepuffert wird und die Zuordnung von Prozessadressraum zu physikalischem Adressraum eindeutig ist.

In eingebetteten Echtzeitbetriebssystemen hingegen wird die Verwendung von Heap-Speicher wenn möglich vermieden, da einerseits nicht garantiert werden kann, dass genügend freier Speicher zur Verfügung steht und andererseits die Zeit, bis der angeforderte Speicher reserviert ist, nur schwer abschätzbar ist. Daher wird sämtlich benötigter Speicher aller Prozesse statisch reserviert. Aus Kompatibilitätsgründen kann ein Echtzeitbetriebssystem auch eine Heap-Speicherverwaltung zur Verfügung stellen. Allerdings muss die Größe des benötigten Heap-Speichers jedes Prozesses schon vor Laufzeit bekannt sein. Sobald Heap-Speicher von einem Prozess reserviert wurde, wird dieser bis zum Herunterfahren des Systems nicht mehr freigegeben. Des Weiteren besitzt ein TLB nur eine begrenzte Anzahl an Einträgen. Wird eine virtuelle Adresse angefragt, die nicht im TLB vorhanden ist, so löst dieser eine Ausnahme aus. Da im Allgemeinen schwer vorhersagbar ist, zu welchem Zeitpunkt auf welche Adresse zugegriffen wird, ist eine zeitliche Abschätzung aufgrund von TLB Verfehlungen schwierig.

2.2.5 Interprozesskommunikation

Inter-process Communication (IPC) ‘Interprozesskommunikation’ bezeichnet den Informations- bzw. Datenaustausch zwischen nebenläufigen Prozessen oder Threads. Dies geht mit der *Prozesssynchronisation* einher, die die zeitliche Abfolge von Prozessen koordiniert.

Dabei kann zwischen zwei Programmiermodellen unterschieden werden, *Message Passing* ‘Nachrichtenaustausch’ und *Shared Variables* ‘gemeinsame Variablen’. Häufig wird *Shared Variable Programming* als *Shared Memory Programming* bezeichnet. *Shared Memory* beschreibt jedoch ein Architektur- und kein Programmiermodell. Im Folgenden wird die Kommunikation zwischen Prozessen erläutert. Dies gilt gleichermaßen für Threads.

2.2.5.1 Kommunikation über gemeinsame Variablen

Bei der Kommunikation über gemeinsame Variablen arbeiten beliebig viele Prozesse bzw. Threads auf *denselben Daten*. Im Allgemeinen wird diese Art des Programmierens über gemeinsamen Speicher realisiert. Hierbei haben die Kerne über Bussysteme oder gemeinsame Caches direkten Zugriff auf den Hauptspeicher, wie dies bei gängigen Multi-core-Prozessoren anzutreffen ist. Ferner ist der Datenaustausch zwischen Threads bei der Verwendung von gemeinsamen Speicher im Vergleich zu dem zwischen Prozessen weniger aufwendig, da sich Threads einen Adressraum teilen. Eine andere Variante sind Systeme mit *Distributed Shared Memory* ‘verteilter gemeinsamen Speicher’. Dabei wird der direkte Zugriff von den einzelnen Systemen auf den gemeinsamen Speicher durch andere Hardwareübertragungstechniken ‘simuliert’.

Der Zugriff auf gemeinsame Variablen erfolgt *lesend* und *schreibend*. Dabei können mehrere Prozesse die Daten beliebig verändern, was zu Dateninkonsistenz führen kann. Des Weiteren können Prozesse bei echt-parallelen Abläufen auf Multicore-Systemen nicht deterministische Ergebnisse liefern, je nachdem, welcher Prozess zuerst Zugriff auf die gemeinsamen Daten bekommt. Man spricht hierbei auch von *Race Conditions*, da der Zustand vom Wettrennen um den Datenzugriff der beteiligten Prozesse und dessen Ausgang abhängt. Aus diesem Grund gibt es Möglichkeiten, den Datenzugriff per *Mutual Exclusion (Mutex)* ‘wechselseitiger Ausschluss’ zu synchronisieren. Dabei ist es immer nur einem Prozess gleichzeitig gestattet, auf den gemeinsamen Daten zu arbeiten. Das Codefragment, welches exklusiven Zugriff auf die gemeinsamen Variablen benötigt, wird auch als *kritischer Abschnitt* oder *kritischer Bereich* titulierte. Ein kritischer Abschnitt sichert zu, dass die Speicherinhalte bzw. Zustände bei (Peripherie-)Registern der Adressen, auf die in diesem Bereich zugegriffen wird, von keinem anderen Prozess oder Interrupt Service Routines (ISRs) modifiziert werden. Es gibt verschiedene Möglichkeiten, die

Sicherung kritischer Bereiche zu realisieren. Nachfolgend sind unterschiedliche Techniken erläutert, die bevorzugt in eingebetteten Systemen zur Anwendung kommen.

Deaktivierung von IRQs In Singlecore-Systemen besteht die Möglichkeit, wechselseitigen Ausschluss durch Deaktivierung von IRQs zu erzielen. Dabei deaktiviert der laufende Prozess beim Eintritt in den kritischen Bereich die IRQs. Dadurch ist es nicht mehr möglich, dass ein anderer Prozess durch den Scheduler bevorzugt wird, da keine Unterbrechung durch Ereignisse mehr stattfinden kann, die einen Aufruf des Schedulers zur Folge hätten. Beim Austritt aus dem kritischen Bereich aktiviert der jeweilige Prozess die IRQs. Ein Nachteil dabei ist, dass sämtliche IRQs deaktiviert sind und die Kontrolle darüber der jeweilige Benutzerprozess besitzt. Daher gibt es Mechanismen, die nur einen Teil der IRQs deaktivieren. Systemrelevante IRQs bleiben unter Kontrolle des Betriebssystems. Dabei muss sichergestellt sein, dass die entsprechenden ISRs die Daten des kritischen Bereichs nicht verändern. Die Option, kritische Abschnitte durch Deaktivierung von IRQs zu schützen, ist allerdings nur bei Singlecore-Systemen praktikabel. Bei einem Dualcore-Prozessor beispielsweise hindert dies einen Prozess des zweiten Kerns nicht daran, die Daten zeitgleich zu modifizieren, während der Prozess des ersten Kerns sich in einem kritischen Abschnitt befindet, der lediglich die IRQs des ersten Kerns deaktiviert. Für echt-parallele Prozesse existieren daher andere Mechanismen, um wechselseitigen Ausschluss zu realisieren.

Lock-Variablen Eine Möglichkeit einen kritischen Bereich zu sichern, besteht darin, ihn mit einem *Lock* ‘Schloss’ zu versehen. Derjenige Prozess, der den Lock akquiriert, hat exklusiven Zugriff auf die Daten. Alle anderen Prozesse müssen solange warten, bis der Lock wieder freigegeben wird. Ob ein Lock frei oder zugeteilt ist, wird durch eine Lock-Variable signalisiert. Dabei symbolisiert der Wert 0 frei und 1 belegt. Solange der Lock vergeben ist, lesen die Prozesse, die ebenfalls den Lock akquirieren möchten, wiederholt den Status der Lock-Variable, um die Freigabe des Locks zu registrieren. Dieses Verfahren wird *busy-waiting* ‘aktiv wartend’ genannt. Bei der Umsetzung ergibt sich jedoch folgendes Problem, das an einem kurzen Beispiel erläutert wird: Initial ist der Lock frei und die Lock-Variable $L = 0$. Prozess *A* liest L und erhält den Wert Null. Somit ist aus Sicht von *A* der Lock frei und setzt $L = 1$. Allerdings liest zwischen dem Lese- und Schreibvorgang der Lock-Variable durch Prozess *A* Prozess *B* ebenfalls L und erhält den Wert 0. Somit ist auch aus Sicht von *B* der Lock frei. Im Endeffekt betreten beide

Prozesse gleichzeitig den kritischen Bereich. Um diesen Effekt zu verhindern, muss die Lock-Variable ohne Unterbrechung gelesen und je nach Zustand gesetzt werden. Hierfür bedient man sich sogenannter *atomarer Instruktionen*. Ein Beispiel hierfür ist die Test-and-Set Instruktion, die die angegebene Speicherstelle liest und überprüft, ob der Wert gleich Null ist. Ist dies der Fall, setzt sie den Wert der Speicherstelle auf Eins. Andernfalls bleibt der aktuelle Wert bestehen. Eine typische und einfache Implementierung eines aktiv wartenden Locks mit Hilfe der Test-and-Set Instruktion ist der Spin-Lock. Aus Sicht von Echtzeitanforderungen allerdings hat der Spin-Lock den Nachteil, dass die Eintrittsreihenfolge der wartenden Prozesse zum kritischen Bereich nicht deterministisch ist. Im schlimmsten Fall wäre es denkbar, dass ein Prozess niemals Zutritt zum Lock erhält, da zu jeder Zeit ein anderer Prozess auf den Lock wartet und bevorzugt den Lock zugeteilt bekommt. Dieser Missstand wird durch Ticket- und Queuing-Lock Implementierungen behoben. Beide Mechanismen sind fair, d. h., jeder Prozess erhält in der Reihenfolge Zutritt zum Lock, in welcher sie sich in der Warteschlange einreihen. Eine weitreichende Betrachtung von fairen Mechanismen zum wechselseitigen Ausschluss ist in (Gerdes 2013) beschrieben. Ein genereller Nachteil dieser Locking-Mechanismen besteht darin, dass die wartenden Prozesse den jeweiligen Prozessorkern belegen bzw. blockieren, ohne dass währenddessen sinnvolle Aufgaben bearbeitet werden können.

Semaphore Das Konzept eines *Semaphors* wurde erstmals von Dijkstra (1968) vorgestellt. Ähnlich einem Lock, regelt ein Semaphor den Zugriff auf eine Ressource. Während bei einem Lock nur ein Prozess gleichzeitig den kritischen Abschnitt betreten kann, kann ein Semaphor eine festgelegte Anzahl an Ressourcen verwalten. Ein binärer Semaphor ist ähnlich einem Lock, mit dem Unterschied, dass das Semaphor nicht zwangsweise von dem Prozess freigegeben werden muss, der den Semaphor akquiriert hat. Ein Semaphor ist eine Datenstruktur, die aus einer positiven Integer-Variablen und zwei Operationen besteht, die als P für *passieren* und V für *verlassen* bezeichnet werden. Die Variable bildet den Semaphorzähler S , der mit der Anzahl der zur Verfügung stehenden Ressourcen initiiert wird. Die Operation P prüft, ob $S > 0$ ist. Fällt die Überprüfung positiv aus, wird S um Eins dekrementiert und der Prozess darf *passieren*, um die Ressource zu verwenden. Andernfalls wird der Prozess in den Zustand ruhend versetzt bzw. schlafen gelegt. Die Operation V inkrementiert S um Eins und der entsprechende Prozess gibt die Ressource frei bzw. verlässt den kritischen Bereich. Falls ruhende Prozesse existieren, wird je nach Algorithmus ein bestimmter Prozess aufgeweckt. Das zweite Unterscheidungsmerkmal zum

Lock besteht darin, dass falls die Ressource bzw. der kritische Abschnitt nicht verfügbar ist, der anfragende Prozess in den Ruhezustand versetzt wird und die Rechenzeit somit einem anderen Prozess zur Verfügung steht. Bei der Verwendung in Echtzeitsystemen ist darauf zu achten, dass der Algorithmus zum Aufwecken der wartenden Prozesse fair ist (vgl. vorherigen Abschnitt). Nur dadurch kann eine obere Zeitschranke ermittelt werden, wie lange ein Prozess im ungünstigsten Fall warten muss, um einen Semaphor zu passieren.

Barrieren Die bisher erläuterten Techniken dienen dazu, dass nur eine festgelegte Anzahl an Prozessen gleichzeitig einen kritischen Abschnitt betreten können. In diesem Zusammenhang bedeutet Synchronisation, dass eine zeitliche Reihenfolge eingehalten und dadurch ein wahlloser Zugriff auf den zu schützenden Bereich unterbunden wird. Eine andere Art der Synchronisation stellen *Barrieren* dar. Hierbei wird sichergestellt, dass sich eine Gruppe von Prozessen an einem bestimmten Punkt, der Barriere, abgleichen. Das bedeutet, dass jeder Prozess an der Barriere warten muss, bis alle beteiligten Prozesse die Barriere erreicht haben. Erst danach ist es allen Prozessen erlaubt, ihre Programmausführung fortzusetzen. Dabei kann das Ausharren auf den letzten, ankommenden Prozess auch hier aktiv wartend oder non-busy-waiting erfolgen, indem bei letzterem die Prozesse in den Ruhezustand versetzt werden. Bei der Realisierung von Barrieren müssen bestimmte Problematiken beachtet werden. Zum einen kann eine augenscheinlich korrekte Implementierung zu Deadlocks führen und zum anderen können je nach Umsetzung massive Buszugriffskonflikte auftreten, die die Ausführungszeit einer Barrierenoperation drastisch erhöhen. Hennessey und Patterson (2011, S. I-14) schildern die beiden Problemstellungen und erläutern mögliche Lösungen. Um die zeitliche Überschätzung von Barrieren gering zu halten, wird eine Realisierung mit Hilfe einer atomaren *Fetch-and-Increment* Instruktion empfohlen. Bei dieser Instruktion wird der Wert aus der angegebenen Speicherstelle gelesen, um Eins erhöht und an dieselbe Speicherstelle zurückgeschrieben.

2.2.5.2 Kommunikation über Nachrichten

Bei der Kommunikation über Nachrichten tauschen sich in der Regel zwei Prozesse miteinander aus. Hierbei *sendet* der eine Prozess eine Nachricht an einen anderen Prozess, der diese *empfängt*. Im Gegensatz zur Kommunikation über gemeinsame Variablen steht

dem Empfänger nach der Übermittlung eine *Kopie der Daten* bereit. Hierzu werden die Daten von dem Datenbereich des Senders in den Datenbereich des Empfängers transferiert. Als Übertragungsmedium kann beispielsweise ein Netzwerk in einem verteilten System fungieren. In Multicore-Systemen kann allerdings auch ein gemeinsamer Speicher zur Übertragung der Daten vom Sender zum Empfänger dienen.

Sender-Receiver-Modell Beim ‘Sender-Empfänger-Modell’ erfolgt der Nachrichtenaustausch unidirektional vom Sender zum Empfänger. Im geringsten Umfang enthält eine Nachricht die eigentlichen Daten, den Datentyp, die Quelle und ein Ziel. Die Sendeoperation muss einen eindeutigen Empfänger angeben, wohingegen die Empfangsoperation eine Nachricht von nur einem bestimmten als auch von einem beliebigen Sender in Empfang nehmen kann. Darüber hinaus besteht die Möglichkeit, per Multi- oder Broadcast eine Nachricht an mehrere oder alle Empfänger zu senden. Dabei erhält jeder Empfänger eine separate Kopie der Nachricht. Da eine Nachricht erst empfangen werden kann, nachdem diese versendet wurde, entsteht eine Sequentialisierung des Programmablaufs zwischen Sender und Empfänger. Dies führt somit zu einer *impliziten* Synchronisation, sprich einer Koordination der zeitlichen Abfolge zwischen den beteiligten Prozessen.

Beim Nachrichtenaustausch kann zwischen synchroner und asynchroner Übertragung unterschieden werden. Bei *synchroner* Datenübertragung werden Sender und Empfänger solange blockiert, bis die Nachricht übermittelt und vollständig empfangen wurde. Eine andere Möglichkeit besteht darin, dass der Sendeprozess seine Ausführung direkt fortsetzt, ohne auf den Empfang der Nachricht zu warten. Hierfür ist jedoch ein Puffer nötig, um die Nachricht zwischenspeichern. Ebenso kann ein Prozess unmittelbar nach einer Empfangsoperation fortgeführt werden, auch wenn keine Nachricht vorhanden ist. Der Prozess wird in diesem Fall vom Betriebssystem benachrichtigt, sobald eine Nachricht eintrifft. Hierbei spricht man von einer nicht-blockierenden Sende- bzw. Empfangsoperation. Sofern einer der beiden miteinander kommunizierenden Prozesse eine nicht-blockierende Operation verwendet, wird dies als *asynchrone* Datenübertragung bezeichnet. Im Allgemeinen kann man festhalten, dass beim synchronen Nachrichtenaustausch die Datenübertragung *ungepuffert* stattfindet, da die Daten vom Datenbereich des Senders über das Übertragungsmedium direkt in den Datenbereich des Empfängers kopiert werden. Beim asynchronen Nachrichtenaustausch hingegen werden die Daten *gepuffert*, damit

der Sende- bzw. Empfangsprozess seine Ausführung fortsetzen kann, ohne auf seinen Kommunikationspartner warten zu müssen.

Beim Sender-Receiver-Modell mit gepufferter Datenübertragung wird zudem zwischen zwei Übertragungsmodi unterschieden. In der Regel reiht sich jede Nachricht im Puffer in einer Warteschlange ein und wird in gleicher Reihenfolge vom Empfänger herausgenommen. Dabei wird keine Nachricht verworfen. Sollte der Puffer voll sein, wird der Sender daran gehindert, eine weitere Nachricht zu senden. Dieser Übertragungsmodus wird *Queuing Mode* genannt. Die zweite Möglichkeit besteht darin, Nachrichten im Puffer zu überschreiben, sobald eine neue Nachricht vom Sender eintrifft. Diese Nachrichten enthalten zumeist Daten, die regelmäßig aktualisiert werden. Beim Lesen der Nachricht enthält der Empfänger somit immer die neuesten Daten. Bei ARINC (Aeronautical Radio, Inc. 2010) wird dieser Modus als *Sampling Mode* bezeichnet, kann allerdings je nach Domäne auch mit einem anderen Namen bezeichnet werden.

Bei einem Nachrichtenaustausch muss bei jeder Sende- und Empfangsoperation ein Ziel bzw. eine Quelle, von welcher Nachrichten entgegengenommen werden, angegeben werden. Dabei gibt es verschiedene Verfahren, um die Zieladressierung bzw. Identifikation einer Nachricht umzusetzen. Ein möglicher Weg ist, Quelle und Ziel durch eindeutige Identifikatoren zu benennen. Allerdings müssen hierfür die Identifikatoren zur Entwicklungszeit feststehen. Um auch dynamisch die Zuweisung von Quelle und Ziel zu gestatten, kann man sich sogenannter *Ports* ‘Pforten’ und *Channels* ‘Kanäle’ bedienen. Pforten sind entweder Sende- oder Empfangspforten und stehen zur Designzeit fest. Eine Sendeoperation gibt daher als Ziel eine Sendepforte an und eine Empfangsoperation als Quelle dementsprechend eine Empfangspforte. Der Weg einer Nachricht wird durch einen Kanal bestimmt, der eine Sende- mit einer Empfangspforte verknüpft. Diese Zuweisung kann zur Laufzeit erfolgen.

Client-Server-Modell Das Sender-Receiver-Modell beschreibt den Informationsaustausch von Daten. Im Vergleich dazu ermöglicht das *Client-Server-Modell*, Aufgaben bzw. Dienste bei einem Server in Auftrag zu geben. Dabei sendet der Client einen *Request* ‘Anfrage’ an den Server, der diese nach Bearbeitung mit einem *Reply* ‘Antwort’ an den Client quittiert. Hierfür sind zwei Nachrichten nötig: Der Client sendet eine Nachricht an den Server, die die Art von Dienst und die entsprechenden Daten enthält. Nach Erledigung des Dienstes, sendet der Server eine Nachricht an den Client, die einen

Rückgabestatus und je nach Dienstleistung berechnete Werte zurückliefert. Diese Art der Dienstleistung wird auch *Remote Procedure Call* ‘Fernprozeduraufruf’ betitelt. Wie beim Sender-Receiver- kann auch beim Client-Server-Modell zwischen *synchronen* und *asynchronen* Fernprozeduraufrufen unterschieden werden. Bei einem synchronen Aufruf wartet bzw. blockiert der Client solange, bis der Server eine Antwort liefert. Im Gegensatz dazu setzt der Client nach einem asynchronen Fernprozeduraufruf unmittelbar seine Programmausführung fort. Der Client wird vom Betriebssystem benachrichtigt, sobald eine Antwort vom Server vorliegt. Da einem Server mehrere Anfragen von verschiedenen Clients vorliegen können, ist es sinnvoll, einen Fernprozeduraufruf asynchron durchzuführen, da sonst der Client solange warten muss, bis alle Anfragen, die vor seiner eigenen eingegangenen sind, bearbeitet worden sind.

2.2.6 Systemaufrufe

Die Funktionalitäten, die ein Betriebssystem Anwendungen bereitstellt, werden durch das entsprechende Application Programming Interface (API) des Betriebssystems abstrahiert. Jede Funktionalität wird durch ein oder mehrere Funktionen umgesetzt. Der Aufruf von Betriebssystemfunktionen erfolgt dabei durch *Systemaufrufe*. Der Grund hierfür liegt in der Zugriffsrechteverwaltung begründet. Im Allgemeinen unterstützt jeder 32-Bit-Mikrocontroller mindestens zwei *Privilegierungsstufen*, einen privilegierten Kernel-Modus und einen unprivilegierten Benutzer-Modus. Im Bereich der Hardwarevirtualisierung sind noch weitere Modi üblich. Wenn der Prozessor im *Kernel-Modus* betrieben wird, ist uneingeschränkter Zugriff auf alle Prozessorregister und Speicheradressen gegeben. Im *Benutzer-Modus* hingegen ist der Zugriff soweit eingeschränkt, dass der laufende Prozess seine Privilegierungsstufe nicht verlassen kann, um eine Stufe mit höherer Berechtigung zu erlangen. In Verbindung mit einer MPU oder MMU kann auch der Zugriff auf Adressen bzw. Adressbereiche eingeschränkt werden (siehe Abschnitt 2.2.4).

Eine Anwendung wird üblicherweise im Benutzer-Modus ausgeführt, um eine Korruption anderer Anwendungen und des Betriebssystems zu unterbinden. Das hat den Vorteil, dass sich ein Fehler in der einen Anwendung nicht auf eine andere Anwendung weiter propagieren kann. Dies ist eine Voraussetzung, um *Freedom of Interference* ‘Rückwirkungsfreiheit’ in einem System zu gewährleisten. Das Betriebssystem dagegen wird im

Kernel-Modus betrieben und als ‘sicher’ erachtet. Das bedeutet, dass von einer korrekten Funktionsweise des Betriebssystems ausgegangen wird, das durch einen genormten Entwicklungsprozess, wie er etwa in IEC 61508 (Deutsches Institut für Normung e.V. 2011) oder ISO 26262 (Internationale Organisation für Normung 2011) definiert ist, zugesichert wird. Da das Betriebssystem die (Hardware-)Ressourcen des Systems verwaltet, werden die meisten Betriebssystemfunktionen im privilegierten Kernel-Modus ausgeführt. Wenn eine Anwendung eine Betriebssystemfunktion aufruft, erfolgt daher ein Wechsel vom Benutzer- in den Kernel-Modus. Der Betriebssystemkernel überprüft zunächst, ob das Anwendungsprogramm berechtigt ist, den angeforderten Betriebssystemdienst in Anspruch zu nehmen. Wenn die Anfrage gestattet wird, führt der Kernel die entsprechende Betriebssystemfunktion durch.

Der Wechsel vom Benutzer- in den Kernel-Modus wird durch einen Kontextwechsel vollzogen, der durch einen Systemaufruf initiiert wird. Technisch gesehen geht ein Systemaufruf ähnlich einer Ausnahme bzw. einem IRQ vonstatten (siehe Abschnitt 2.2.1). Bei einem Systemaufruf werden verschiedene Parameter übergeben. Dabei kennzeichnet ein Parameter, welche Betriebssystemfunktion durch den Aufruf angefordert wird. Im Gegensatz zu einem normalen Funktionsaufruf wird die Funktion nicht durch den Funktionsnamen bestimmt. Nach dem Kontextwechsel wird eine Systemaufrufroutine ausgeführt, die anhand der übergebenen Parameter und einer Systemaufruftabelle zur entsprechenden Betriebssystemfunktion springt und diese ausführt. Nach deren Abarbeitung erfolgt ein erneuter Kontextwechsel und ein Wechsel vom Kernel- in den Benutzermodus wird durchgeführt. Dabei werden eventuelle Rückgabeparameter an die Anwendung übergeben und die Programmausführung wird fortgesetzt.

2.3 Betriebssysteme aus dem Automobilumfeld

2.3.1 OSEK/VDX

OSEK/VDX steht für *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive* (OSEK/VDX 2005) und ist ein industrieller

Automobilstandard für verteilte *Electronic Control Units (ECUs)* ‘elektronische Steuergeräte’ in Fahrzeugen. Häufig wird der Einfachheit halber nur die Abkürzung OSEK verwendet, so auch in diesem Dokument. OSEK spezifiziert drei Bereiche:

- OSEK Operating System (OS):
Echtzeitbetriebssystem für eingebettete Systeme bzw. ECUs.
- OSEK Communication (COM):
Kommunikation innerhalb und zwischen ECUs.
- OSEK Network Management (NM):
Netzwerkverwaltung mit standardisierten Schnittstellen zur Konfigurationsbestimmung und Überwachung von Kommunikationsnetzwerken.

Dabei spezifiziert OSEK Schnittstellen, die die Anwendungssoftware weitestgehend von der jeweiligen Hardwareplattform und dem Netzwerk entkoppeln. Die konkrete Implementierung der spezifizierten Funktionalitäten ist hingegen offen. Im Folgenden wird näher auf die Merkmale von OSEK OS eingegangen.

Die meisten Automobilanwendungen haben strenge Echtzeitanforderungen. OSEK OS berücksichtigt diese Anforderungen und zählt somit zur Kategorie der Echtzeitbetriebssysteme (siehe Abschnitt 2.2). Wie jedes Betriebssystem stellt es verschiedene Betriebssystemdienste über eine API der Anwendungssoftware zur Verfügung. Die Betriebssystemdienste sind in die Bereiche *Task Management*, *Interrupt Handling*, *Resource Management*, *Event Control*, *Alarms*, *Operating System Execution Control* und *Hook Routines* unterteilt. Jeder Bereich stellt verschiedene *Service Calls* ‘Systemaufrufe’ bereit, die eine standardisierte Schnittstelle zur Anwendungsebene bilden und deren Funktionen wohldefiniert sind. Dadurch werden die Anwendungen von der konkreten Hardwareplattform abstrahiert. Ferner soll damit eine möglichst hohe Portabilität der Anwendungssoftware und Wiederverwendbarkeit erreicht werden.

Da die Palette der eingesetzten Mikrocontroller im Automobilbereich je nach Einsatzzweck von 8-Bit bis 32-Bit-Controllern variiert, spielt die Skalierbarkeit von OSEK OS eine wichtige Rolle. Zu diesem Zweck sind in OSEK OS vier sogenannte *Conformance Classes* definiert, namentlich BCC1, BCC2, ECC1 und ECC2 (Basic/Extended Conformance Class). Sie unterscheiden sich in der Anzahl der Tasks je Priorität, den unterstützten Task-Typen und der Möglichkeit zur Mehrfachaktivierung von Tasks. Je nach Klasse

sind die Anforderungen an Prozessor und Speicher unterschiedlich hoch, angefangen mit BCC1, die die geringsten Hardwareanforderungen benötigt. Somit ist auch der Einsatz auf leistungsschwächeren ECUs möglich.

Bei den möglichen Task-Typen wird zwischen *Basic Tasks* und *Extended Tasks* unterschieden, was sich in der Namensgebung der Conformance Classes widerspiegelt. Basic Tasks unterstützen die Task-Zustände bereit, laufend und ruhend, während Extended Tasks zusätzlich noch den Zustand wartend erlauben (siehe Abschnitt 2.2.1). Unabhängig vom Task-Typ verwendet OSEK OS das RMS-Verfahren. Aufgrund der strengen Zeitanforderungen von Automobilanwendungen wird auf eine dynamische Erzeugung von Betriebssystemobjekten, wie beispielsweise Tasks, Ressourcen, Alarme etc., verzichtet. Die Konfiguration dieser Objekte geschieht statisch zur Entwicklungszeit.

2.3.2 AUTOSAR

Bei der AUTomotive Open System ARchitecture (AUTOSAR 2016) handelt es sich um einen Zusammenschluss verschiedener Original Equipment Manufacturers (OEMs) und erstrangiger Zulieferer aus dem Automobilssektor, um einen offenen de-facto Standard für Elektrik/Elektronik (E/E) Architekturen im Automobilssegment zu etablieren. Ziele sind:

- Standardisierung von Basisfunktionalitäten
- Skalierbarkeit hinsichtlich verschiedener Fahrzeug- und Plattformvarianten
- Unterstützung verschiedener Anwendungsbereiche
- Entwicklung von hochverfügbaren Systemen
- Portierbarkeit von Software
- Einbettung von Modulen anderer Hersteller
- Wartbarkeit während des gesamten Produktlebenszyklus (Software Updates/Upgrades)

Diese Zielsetzungen werden durch eine modulare, in mehreren Ebenen gegliederte Softwarearchitektur, standardisierte Schnittstellen, eine Runtime Environment (RTE) und die Standardisierung der Austauschformate und Systemkonfiguration erreicht. Dabei werden bestimmte OSEK Spezifikationen in AUTOSAR wiederverwendet. Dies betrifft in erster Linie das AUTOSAR OS, das abwärtskompatibel zur OSEK OS Spezifikation ist. Das OSEK NM unterscheidet sich hingegen deutlich vom Netzwerkmanagement in AUTOSAR. Analog zu OSEK sind bei AUTOSAR Schnittstellen und Austauschformate standardisiert, die konkrete Umsetzung der Basissoftware und Werkzeuge ist dagegen offen.

AUTOSAR ermöglicht die Entwicklung nach einer Top-down-Methode. Hierzu wird eine Anwendung in *Software Components (SW-Cs)* gekapselt, die nach außen eindeutige Schnittstellen besitzen, die über ein Austauschformat namens *Software Component Description* standardisiert sind. Über diese Schnittstellen ist eine Kommunikation der einzelnen SW-Cs möglich. Dies gilt auch für SW-Cs unterschiedlicher Anwendungen, die auf verschiedenen ECUs platziert sind. Dabei ist es unerheblich, wie das darunterliegende Kommunikationsnetzwerk zwischen den ECUs beschaffen ist. Durch den *Virtual Functional Bus (VFB)* wird das Kommunikationsnetzwerk technologieunabhängig abstrahiert. Dadurch ist schon in einer frühen Design- und Entwicklungsphase eine Systemsichtweise gestattet. Das Aufschlüsseln der SW-Cs auf die einzelnen ECUs unter Einhaltung aller Beschränkungen geschieht anhand von System- und ECU-Konfigurationsbeschreibungen. Die konkrete Realisierung des VFB auf jeder ECU erfolgt durch die *AUTOSAR Runtime Environment (RTE)*. Die Kommunikation zwischen den SW-Cs ist für die RTE transparent. Sie erfolgt durch API Aufrufe an die RTE, die die Aufrufe wiederum an die *Basic Software (BSW)* delegiert. Die BSW umfasst standardisierte Softwaremodule, auch BSW Module genannt, die die Grundfunktionalität einer jeden ECU bilden und deren Dienste durch die RTE den Anwendungen bzw. SW-Cs zur Verfügung gestellt werden. Dabei übernimmt die BSW selbst keine funktionalen Aufgaben.

Abbildung 2.1 zeigt den Aufbau der AUTOSAR Systemarchitektur und die Unterteilung in die einzelnen Ebenen. Die AUTOSAR Softwarearchitektur ist in einen *Application Layer* ‘Anwendungsebene’, die *AUTOSAR Runtime Environment* und die *BSW* untergliedert. Die BSW ist zudem in den *Services Layer*, *ECU Abstraction Layer* und *Microcontroller Abstraction Layer (MCAL)* unterteilt. Daneben existieren noch die *Complex Device Drivers (CDDs)*. Der MCAL beinhaltet Treiber für den direkten Zugriff auf

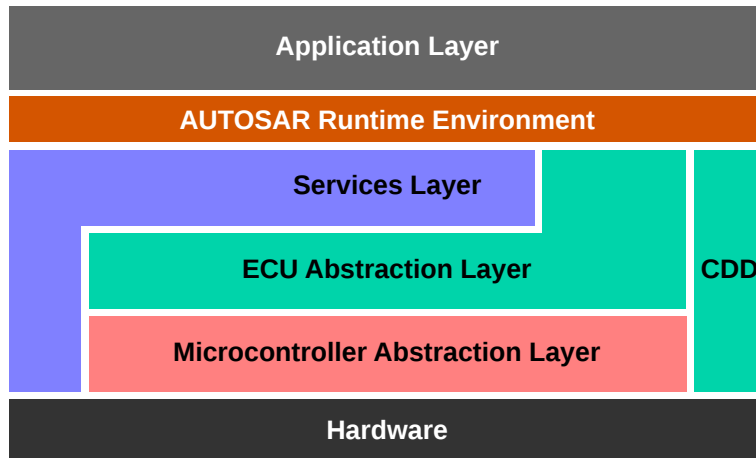


Abbildung 2.1: AUTOSAR Systemarchitektur

den Mikrocontroller und dessen interne Peripheriegeräte. Er entkoppelt höhere Softwareebenen von den spezifischen Merkmalen des Mikrocontrollers. Darauf aufsetzend bietet der ECU Abstraction Layer eine einheitliche Schnittstelle für Peripheriegeräte, unabhängig davon, ob ein Gerät intern oder extern über einen Erweiterungsbus angeschlossen ist. Die CDDs sitzen direkt zwischen Mikrocontroller und RTE. Sie integrieren Gerätetreiber, die nicht in AUTOSAR spezifiziert sind bzw. aus Migrationsgründen oder die besonders strenge Zeitanforderungen besitzen (AUTOSAR 2015a). Der Services Layer bildet die höchste Ebene und bietet verschiedenste Dienste an, die vorwiegend von Anwendungen über die RTE genutzt werden. Aber auch andere BSW Module nutzen dessen Dienste.

Abbildung 2.2 stellt einen detaillierten Überblick der AUTOSAR Softwarearchitektur mit einer Untergliederung in einzelne Funktionsbereiche dar. Hierbei sind u. a. die drei Dienstkategorien des Services Layer erkennbar, die *System*, *Memory* und *Communication Services* anbieten. Neben einer ECU Zustandsverwaltung, Verwaltung verschiedener Betriebsmodi, Diagnosefunktionen und weiteren Diensten beinhalten die System Services auch ein *OS Modul*, das Betriebssystemfunktionalitäten bereitstellt. In die AUTOSAR OS Spezifikation wurde die OSEK OS Spezifikation integriert und der Funktionsumfang des OSEK OS Moduls deutlich erweitert. Eine wichtige Neuerung gegenüber OSEK OS ist, dass AUTOSAR OS nun Speicherschutz durch die Unterstützung von MPUs bzw. MMUs anbietet. Neben der Unterstützung der Conformance Classes (siehe Abschnitt 2.3.1) unterscheidet AUTOSAR noch zwischen vier *Scalability Classes* (AUTOSAR 2015b,

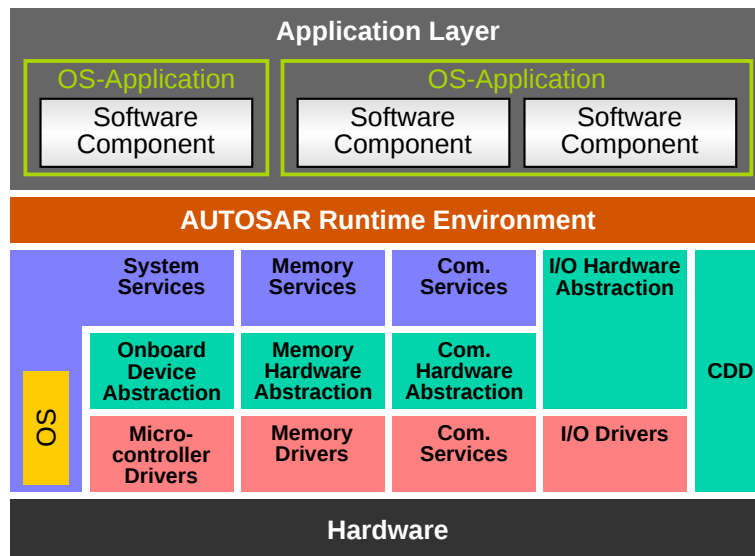


Abbildung 2.2: Detaillierte Ansicht der AUTOSAR Systemarchitektur

S. 123). Diese dienen der Skalierbarkeit des Betriebssystems, um dieses entsprechend den Anwendungsanforderungen anzupassen und das volle Potenzial des jeweiligen Mikrocontrollers auszuschöpfen. Beispielsweise wird Speicherschutz erst ab den Scalability Classes 3 und 4 unterstützt. Der Funktionsumfang des OSEK OSs ist in allen Scalability Classes gegeben.

In Abbildung 2.2 ist ebenfalls zu erkennen, dass eine Anwendung aus mindestens einer oder mehreren *SW-Cs* besteht. Ferner ist jeder Task einer *SW-C* mindestens einer *OS-Application* zugeordnet. Eine *OS-Application* bildet einen Container, dem verschiedene Betriebssystemobjekte, wie Tasks, ISRs, Ressourcen, Events, Alarmer etc., zugeordnet sind. Jedes Betriebssystemobjekt muss einer *OS-Application* zugewiesen sein. Des Weiteren wird zwischen *trusted* ‘vertrauenswürdigen’ und *non-trusted OS-Applications* differenziert. Trusted *OS-Applications* haben uneingeschränkten Zugriff auf Code- und Datenspeicherbereiche von Tasks, ISRs und anderen *OS-Applications*, während der Zugriff bei *non-trusted OS-Applications* auf Tasks und ISRs der eigenen *OS-Application* beschränkt ist. *OS-Applications* bilden somit Speicherschutzdomänen, ähnlich dem Speicherschutz bei Prozessen (vgl. Abschnitt 2.2.1). Der Datenaustausch zwischen *SW-Cs* findet über festgelegte Kommunikationsports statt. Die Kommunikation kann entweder nach dem Client-Server- oder Sender-Receiver-Modell erfolgen.

2.3.2.1 Multicore-Prozessor-Unterstützung

Seit der Veröffentlichung der AUTOSAR Release 4.0 OS Spezifikation (AUTOSAR 2011) existiert eine Unterstützung für Multicore-Systeme. Um einen Überblick darüber zu geben, wie die Multicore-Prozessor-Unterstützung in das AUTOSAR Modell bezüglich der Anwendungsebene, der RTE und dem OS integriert ist, werden nachfolgend einige Begriffe und Zusammenhänge erläutert. Aus Sicht der Softwaremodellierung besteht eine Anwendung aus ein oder mehreren SW-Cs, die über Kommunikationsports miteinander verbunden sind. Eine SW-C kann auch als Komposition dienen, die wiederum aus weiteren SW-Cs besteht. Sofern eine SW-C nicht weiter unterteilt werden kann, wird sie als *Atomic SW-C* bezeichnet. Eine Atomic SW-C wiederum umfasst ein oder mehrere *Runnable Entitys (REs)*, wobei eine RE eine Sequenz von Instruktionen ist und als eine C-Funktion angesehen werden kann. Des Weiteren ist jede SW-C einer *Partition* zugewiesen. Partitionen werden für Speicherschutz und bei Multicore-Systemen benötigt.

Aus Betriebssystemensicht wird eine Partition durch eine OS-Application realisiert. Eine OS-Application ist vollständig einem Prozessorkern zugewiesen, wohingegen einem Kern mehrere OS-Applications zugeordnet sein können. Daraus folgt, dass pro Kern mindestens eine OS-Application vorhanden ist. Eine Ausnahme bilden Singlecore-Systeme mit Scalability Class 1 oder 2, da hierfür nicht zwingend eine OS-Application existieren muss. Wie bereits erläutert, muss jedes Betriebssystemobjekt einer OS-Application zugewiesen sein, somit auch Tasks. Ein Task bildet die kleinste ausführbare Einheit, die vom Betriebssystem planbar ist. Ferner beinhaltet ein Task-Körper REs und *Glue-Code* ‘Verbindungscode’, der die REs verbindet und keinerlei Funktionalität innehat. Jede RE einer SW-C ist unabhängig voneinander planbar und wird in dem Kontext eines Tasks ausgeführt. Die Zuweisung von REs zu Tasks und somit zu einer *OS-Application* wird allerdings durch die Zuordnung einer RE zu einer SW-C, die wiederum einer *Partition* zugehört, beschränkt. Somit bilden einerseits REs und andererseits Partitionen und OS-Applications die Verbindung zwischen Softwaremodellierungs- und Betriebssystemensicht.

Die Kommunikation zwischen verschiedenen SW-Cs findet nach dem Client-Server- bzw. Sender-Receiver-Modell statt. Je nach Zuordnung der miteinander kommunizierenden SW-Cs zu Partitionen und der daraus resultierenden Platzierung auf Prozessorkernen oder ECUs, wird zwischen den folgenden Kommunikationsmustern unterschieden (AUTOSAR 2015c, S. 143):

- Kommunikation innerhalb einer Atomic SW-C
- Intra-Partition-Kommunikation zwischen SW-Cs
- Inter-Partition-Kommunikation
- Inter-ECU-Kommunikation

Bei Multicore-Systemen spielt die Inter-Partition-Kommunikation eine besondere Rolle, da diese für die Kommunikation zwischen Prozessorkernen bzw. Partitionen/OS-Applications verwendet werden muss. Für den Datenaustausch zwischen OS-Applications ist der *Inter-OS-Application Communicator (IOC)* verantwortlich. Je nach Konfiguration steigt der zeitliche Mehraufwand durch die Verwendung des IOCs. Beispielsweise bedeutet der Datenaustausch zwischen trusted OS-Applications einen höheren Rechenaufwand als bei non-trusted OS-Applications, da hierbei Kontextwechsel aufgrund des Speicherschutzes nötig sind. Eine Beispielkommunikation nach dem Sender-Receiver-Modell wird in Abschnitt 4.5 erläutert und anhand von Abbildung 4.5 gezeigt.

3

Generische Systemarchitektur für eingebettete Echtzeitsysteme

3.1 Zielsetzung und Überblick

Die Abkürzung ACMC-OS steht für *Automotive Compatible Multi-Core Operating System*. Die Idee dahinter ist, eine Evaluierungsplattform zu entwickeln, die den Anforderungen von parallelisierten bzw. parallel ausführbaren Anwendungen aus dem Automobilbereich an eine solche Plattform und deren zugehörigen Systemarchitektur gerecht wird. Dabei wird besonderes Augenmerk auf die effiziente Kommunikation zwischen den Kernen eines Prozessors seitens des Betriebssystems gelegt. Ein wichtiger Punkt hierbei ist auch eine möglichst einfache Analysierbarkeit der Software bezüglich der *WCET*. Sämtliche von ACMC-OS zur Verfügung gestellten API Funktionen sind dahingehend optimiert. Um eine problemlose zeitliche Analyse zu ermöglichen, muss auch die Implementierung der Anwendungssoftware gewissen Anforderungen genügen. Bei der zugrunde liegenden Hardware liegt der Fokus auf typischen Multicore-Prozessoren mit zwei bis vier Kernen, die Anwendung im automobilen Umfeld finden (z.b. Freescale Qorivva, Infineon Aurix), wobei auch zukünftige Prozessoren mit bis zu acht Kernen für eingebettete Systeme berücksichtigt werden. Da die Bereitstellung einer Infrastruktur zur Ausführung paralleler

Anwendungen zentraler Bestandteil ist, konzentriert sich ACMC-OS überwiegend auf Komponenten, die zur Programmausführung, genauer gesagt Betriebssystemmechanismen, und Kommunikation bzw. Synchronisation benötigt werden.

Die Betriebssystemmechanismen sind gemäß der OSEK OS (OSEK/VDX 2005) und AUTOSAR OS Spezifikationen (AUTOSAR 2015b) implementiert, wodurch eine Kompatibilität zur AUTOSAR API weitestgehend gewährleistet ist. Die Kommunikations- und Synchronisationsmechanismen sind in Anlehnung an die in der AUTOSAR RTE Spezifikation (AUTOSAR 2015c) beschriebenen Paradigmen implementiert. Darüber hinaus wurden die spezifizierten Kommunikations- und Synchronisationsmechanismen erweitert bzw. um weitere Mechanismen ergänzt, um möglichst kleine zeitliche Abschätzungen bei WCET Analysen zu erzielen.

ACMC-OS fand seinen Ursprung im parMERASA Projekt (Ungerer et al. 2013). Daher basieren grundlegende Designentscheidungen von ACMC-OS auf Anforderungsanalysen, die im Rahmen des parMERASA Projekts durchgeführt wurden. Das Projekt hatte zum Ziel, parallelisierte Anwendungen mit harten Echtzeitbedingungen auf Multicore-Plattformen auszuführen und zudem deren Analysierbarkeit bezüglich ihrer WCET zu garantieren. Eine weitere Anforderung bestand darin, Anwendungen aus den drei verschiedenen Industriezweigen, *Automobilsektor*, *Luftfahrt* und *Baumaschinenbranche*, auf der gleichen Zielplattform ausführen zu können. Um die zeitliche Analyse zu vereinfachen, wurden die Anwendungen, die Systemsoftware und die Hardwareplattform hinsichtlich ihrer WCET optimiert. Im Folgenden werden die im parMERASA Projekt getroffenen Designentscheidungen erläutert, die Einfluss auf die Architektur von ACMC-OS hatten.

3.2 Systemarchitektur

Eine Systemarchitektur beschreibt sowohl die Bestandteile der Hardware als auch der Software. Abbildung 3.1 zeigt den allgemeinen Aufbau der Systemarchitektur im Falle von parMERASA. Zwischen der *Hardware* und der eigentlichen *Anwendung* liegt ein *domänenspezifischer Software-Stack*, der je nach industrieller Domäne verschiedene Schnittstellen und Funktionalitäten für die Anwendungsebene bereitstellt und unterschiedlich gegliedert sein kann. Eine Softwarearchitektur umfasst sowohl den domänenspezifischen Software-



Abbildung 3.1: Allgemeiner Aufbau der parMERASA Systemarchitektur

Stack als auch die Anwendungsebene (Bass et al. 2012, S. 30). Jede der drei Domänen legt für deren Anwendungen eine eigene Softwarearchitektur zugrunde. Die Anwendungen interagieren daher mit dem für ihre Domäne spezifischen Software-Stack, der als Mittler zwischen der Anwendungssoftware und der Hardware fungiert und forthin auch als Systemsoftware bezeichnet wird (Nehmer et al. 2001, S. 1–2). Die Systemsoftware umfasst dabei klassische Betriebssystemfunktionalitäten als auch eine auf die jeweilige Domäne zugeschnittene *Runtime Environment* ‘Laufzeitumgebung’. Über standardisierte Schnittstellen haben Anwendungen Zugriff auf die bereitgestellten Funktionen der Systemsoftware. Um die Entwicklungskosten gering zu halten, wurde auf eine möglichst hohe Wiederverwendbarkeit von Software geachtet. Dazu wurden Gemeinsamkeiten und Unterschiede in den Softwarearchitekturen identifiziert, die im folgenden Abschnitt genauer betrachtet werden.

3.2.1 Anforderungen

Die allgemeinen Anforderungen an die Systemarchitektur im parMERASA Projekt bestanden in (1) einer möglichst guten Auslastung der Multicore-Plattform durch die jeweils parallelisierte Industrieapplikation, (2) der zeitlichen Analysierbarkeit der kompletten Systemarchitektur und (3) der Ausführung der verschiedenen Anwendungen aus dem jeweiligen Industriezweig auf einer einheitlichen Plattform. Zusätzlich hat jede Domäne noch ihre eigenen Anforderungen, die berücksichtigt werden müssen. Bei der Anforderungsanalyse liegt das Hauptaugenmerk auf Funktionalitäten, die für das Betriebssystem relevant sind. Darüber hinaus benötigte Funktionalitäten sind Teil der jeweiligen domänenspezifischen Runtime Environment (RTE).

Im parMERASA Projekt wurde für jede Industriedomäne eine bereits vorhandene Beispielanwendung, die auf Basis einer Singlecore-Plattform entwickelt wurde, auf die parMERASA Multicore-Plattform adaptiert. Im Folgenden werden daher zuerst die ursprünglichen Anforderungen der jeweiligen Anwendung bzw. domänenspezifischen RTE dargelegt. Nachfolgend werden unterschiedlichen Anforderungen verglichen und die zusätzlichen Anforderungen an die RTE erläutert, um die jeweilige Beispielanwendung auf einer Multicore-Plattform betreiben zu können.

3.2.1.1 Automobilssektor

Die meisten Softwareanwendungen in einem Kraftfahrzeug, die die Bereiche Karosserie und Komfort, Antriebsstrang oder Fahrgestell adressieren, sind konform zur OSEK Spezifikation, seit 2008 meist sogar zur AUTOSAR Spezifikation (BMW Group 2008). In Kapitel 2, Abschnitt 2.3.1 und 2.3.2 wurden die beiden Standards bereits ausführlich behandelt.

Scheduling: Die Ablaufplanung von Tasks erfolgt nach einem prioritätsbasierten, präemptiven Scheduling-Verfahren. Die Prioritäten werden hierbei fest, nach dem RMS-Verfahren, vergeben. Verschiedene Ereignisse können zu einem Rescheduling und je nach Systemzustand meistens zu einem Task-Wechsel führen. Um Tasks zeitgesteuert aktivieren zu können, bietet AUTOSAR Alarme und Schedule Tables an. Beide Mechanismen lösen zu bestimmten Zeitpunkten ein Ereignis aus, dass eine Aktivierung eines oder mehrerer Tasks und ein anschließendes Rescheduling zur Folge hat. Im Zuge der AUTOSAR Release 4.0 OS Spezifikation (AUTOSAR 2011) wurde die Funktionalität des Schedulers um eine Unterstützung von Multicore-Prozessoren erweitert. Dabei findet die Ablaufplanung pro Prozessorkern lokal statt. Eine Beeinflussung der Ablaufplanung auf einem anderen Kern als dem eigenen kann durch einige, wenige API Aufrufe erfolgen. Ungeachtet dessen existiert keine Instanz, die eine globale Sicht auf die Ablaufplanung besitzt.

Schutz: Zu den Schutzvorkehrungen zählen die zeitliche und räumliche Trennung, Absicherung von Dienstaufrufen und Fehlerbehandlung bei Schutzverletzungen. Voraussetzung

für eine räumliche Trennung ist, dass die Hardware eine MPU oder MMU bereitstellt. Ferner muss die Anwendung in ein oder mehrere OS-Applications unterteilt sein, da diese als Abgrenzung der einzelnen Schutzbereiche dienen. Es kann der Zugriff auf Code- und Datensegmente und den Stack-Speicherbereich von Tasks bzw. ISRs der Kategorie 2 gesichert werden. Falls hardwareseitig keine MPU oder MMU vorhanden sind, besteht die Möglichkeit des Stack-Monitorings. Dabei kann ein Über- bzw. Unterlauf des Stack-Speicherbereiches des aktiven Tasks bzw. ISRs bei einem Task-Wechsel erkannt werden und adäquate Maßnahmen ergriffen werden. Im Gegensatz zum Speicherschutz per MPU oder MMU entfällt eine Absicherung des Stack-Speicherbereichs, der somit überschrieben werden kann. Ferner findet keine zeitnahe Überprüfung wie beim Speicherschutz statt, sondern erst bei einem Task-Wechsel.

Zeitliche Trennung wird auf Task-Ebene realisiert. Dabei kann jedem Task ein maximales Zeitbudget zugewiesen werden. Die Überprüfung findet nach einem Rescheduling-Ereignis im Scheduler statt. Bei der Absicherung von Dienstaufrufen wird überprüft, ob der Aufruf eines Systemdienstes aus dem aktuellen Ausführungskontext erlaubt ist. Beispielsweise führt der Aufruf von `TerminateTask()` innerhalb einer ISR der Kategorie 2 dazu, dass dieser nicht ausgeführt und eine Fehlerbehandlungsroutine aufgerufen wird. Die Fehlerbehandlung greift bei Fehlern, die sich innerhalb von Objekten ereignen, die von einer OS-Application verwaltet werden. Im Fehlerfall kann in der Fehlerbehandlungsroutine die Anwendungssoftware entscheiden, ob auf Task-, OS-Application- oder Systemebene eine Terminierung oder ein Neustart erfolgen soll.

Kommunikation: Auf Anwendungsebene sind die beiden grundlegenden Kommunikationsparadigmen Client-Server und Sender-Receiver vorgesehen. Während das Client-Server-Modell eine bidirektionale Kommunikation darstellt, wird bei Letzterem nur in eine Richtung, vom Sender zum Receiver, kommuniziert. Um eine Empfangsbestätigung zu erhalten, muss eine zweite, separate Verbindung hergestellt werden, wobei der ursprüngliche Receiver nun als Sender agiert. Des Weiteren wird bei der Sender-Receiver-Kommunikation zwischen einer Data- und Event-Semantik unterschieden. Bei der Data-Semantik, die auch als *last-is-best* Semantik bezeichnet wird, sind nur die zuletzt empfangenen Werte von Interesse. Im Gegensatz dazu ist bei der Event-Semantik die gesamte Abfolge der empfangenen Nachrichten von Relevanz.

Synchronisation: Einfache Kommunikation zwischen Anwendungen erfolgt mit Hilfe von Events. Diese können verwendet werden, um binäre Informationen zwischen Tasks zu kommunizieren. Des Weiteren bietet das OS Modul eine Ressourcen-Verwaltung an, um einen wechselseitigen Ausschluss beim Zugriff auf geteilte Ressourcen von quasi-parallelen Tasks zu garantieren. Zudem verhindert das Modell der Ressourcen-Verwaltung eine Prioritätsinversion durch das *Immediate Priority Ceiling Protocol* (Burns et al. 2001). Um bestimmte Codeabschnitte ohne Unterbrechung durch den Scheduler ausführen zu können, besteht die Möglichkeit, dass ein Task die Ressource RESOURCE_SCHEDULER akquiriert. Solange der Task diese Ressource hält, kann dieser nicht unterbrochen werden mit Ausnahme von ISRs der Kategorie 1. Für einen synchronisierten Zugriff auf gemeinsame Ressourcen von unterschiedlichen Kernen ist ausschließlich ein wechselseitiger Ausschluss nach dem Spin-Lock Verfahren verantwortlich.

Eingabe und Ausgabe: Die meisten Automobilanwendungen interagieren mit ihrer physikalischen Umgebung durch Sensoren und Aktuatoren, die über Einheiten zur Eingabe und Ausgabe (E/A) des elektronischen Steuergeräts angebunden sind. Dabei müssen einige Anwendungen innerhalb weniger Millisekunden auf externe Ereignisse reagieren, z. B. das Antiblockiersystem (ABS) oder die Elektronische Stabilitätskontrolle (ESP). Aufgrund dieser Tatsache ist eine geringe Latenz zwischen dem Eintritt eines Ereignisses an einer E/A-Einheit und der Softwarekomponente, die die entsprechende Funktionalität umsetzt, erforderlich.

3.2.1.2 Luftfahrt

Das *Integrated Modular Avionics (IMA)* Konzept der dritten Generation (RTCA, Inc. 2005; Eveleens 2006) kam in der zivilen Luftfahrt erstmals im Airbus A380 zum Einsatz. Seitdem werden die meisten elektronischen Systeme in der Luftfahrt entsprechend dem IMA 3G Konzept entwickelt. Davor wurden Systeme üblicherweise nach dem Prinzip der sogenannten *Federated Avionics Architecture*, sprich verbundene Avionik-Architektur, entwickelt. Dabei hat jedes System seinen eigenen Rechner bzw. Line Replaceable Unit (LRU), der ausschließlich zur Erbringung einer bestimmten Funktionalität zuständig ist. Die einzelnen Systeme sind lose miteinander gekoppelt und daher nur bedingt voneinander abhängig. Die Funktionssicherheit und Verfügbarkeit wird anhand der einzelnen

LRUs bemessen. Das Hinzufügen weiterer Systemfunktionen ist oft mit einem größeren Redesign der kompletten Architektur, Integrationstests und Verifikationsprozessen verbunden, was wiederum hohe Kosten nach sich zieht (Watkins et al. 2007; Ekman 2010). Weitere Nachteile sind ein erhöhter Energieverbrauch, da jede LRU ihre eigene Stromversorgung besitzt, und die Zunahme des Gesamtvolumens sowie des Gewichts. IMA adressiert diese Nachteile, indem es ein Netzwerk an gemeinsam nutzbaren Recheneinheiten, Kommunikations- und E/A-Ressourcen bereitstellt. Im Gegensatz zur Federated Architecture teilen sich die Anwendungen bei IMA die vorhandenen Ressourcen und IMA erlaubt die unabhängige Ausführung dieser Anwendungen. Darüber hinaus unterstützt IMA auch die simultane Verwendung von mehreren Avionikfunktionen mit unterschiedlichen Kritikalitätsstufen (Garside et al. 2007). Um dies zu ermöglichen, führt IMA das Partitionierungskonzept ein, d.h. die einzelnen Anwendungen werden funktional voneinander getrennt, um eine Fehlerausbreitung zu unterbinden.

Daneben gibt es weitere Standards (Aeronautical Radio, Inc. 1997), die in engem Zusammenhang zu IMA stehen. Dazu zählen im Allgemeinen die Aeronautical Radio Incorporated (ARINC) Standards. ARINC 653 beispielsweise befasst sich mit der von IMA geforderten Partitionierung aus Softwaresicht und nennt die dafür notwendigen Anforderungen an das zugrunde liegende Betriebssystem. Dessen Aufgabe ist es, die für die Partitionierung benötigten Partitionen zu verwalten. Jede Avionikanwendung ist üblicherweise einer Partition zugeteilt. Große Anwendungen können sich auch über mehrere Partitionen erstrecken. Neben der simultanen Ausführung verschiedener Anwendungen bzw. Partitionen innerhalb einer IMA Architektur, ist auch die Verlagerung einer Partition von einer LRU zu einer Anderen zur Laufzeit möglich.

In erster Linie spezifiziert ARINC 653 die APplication/EXecutive (APEX) API (Aeronautical Radio, Inc. 2010), die eine einheitliche Programmierschnittstelle zwischen Anwendungssoftware und Betriebssystem in einem Avionik-Rechnersystem zur Verfügung stellt. Abbildung 3.2 zeigt u. a. die APEX Schnittstelle in einer exemplarischen Systemarchitektur, im Avionikbereich auch *Integrated Module* oder kurz *Module* genannt, gemäß der ARINC 653 Spezifikation.

Die *Anwendungssoftware einer Partition* hat über die *APEX* Schnittstelle und deren bereitgestellten Dienste Zugang zu systemrelevanten Funktionen, wie z. B. Steuerung

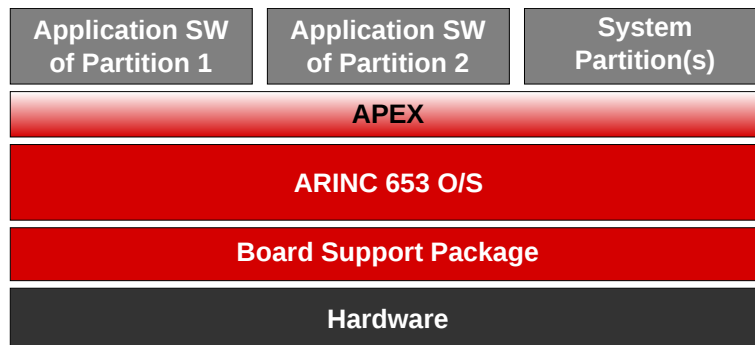


Abbildung 3.2: Beispiel einer ARINC Systemarchitektur

des Scheduling, der Kommunikation und Abfrage von Statusinformationen. *Systempartitionen* benötigen zudem Funktionalitäten, die nicht über die APEX API abgedeckt werden. Sie übernehmen zumeist Funktionen, die speziell auf die jeweilige Hardware zugeschnitten sind, z. B. Hardwaretreiber für bestimmte Kommunikationsgeräte, und sind im Gegensatz zu Anwendungspartitionen nicht ohne Weiteres zwischen unterschiedlichen LRUs portierbar. Während APEX die Schnittstelle zwischen Anwendungsebene und Betriebssystem (*ARINC 653 O/S*) bildet, abstrahiert das *Board Support Package* zwischen Betriebssystem und *Hardware* durch die Bereitstellung hardware-spezifischer Treiber. Das Betriebssystem verwaltet allein die Partitionen und die Kommunikation untereinander. Ebenso bietet es über die APEX API Dienste zur Prozess- und Kommunikationssteuerung innerhalb von Partitionen an, deren Ausführung unter der Obhut des Betriebssystems stehen. Allerdings hat die Anwendungssoftware hierbei die Möglichkeit, den Prozess- und Kommunikationsablauf zu steuern. Somit ist das Betriebssystem für den Betriebsablauf auf zwei Ebenen zuständig, der Partitions- und Prozessebene. Für beide Ebenen bietet das Betriebssystem entsprechende Dienste an, deren Aufgabenbereich sogar identisch sein kann, deren Funktionalität sich hingegen je nach Ebene unterscheidet, so etwa Partitions- kontra Prozess-Scheduling.

Schutz: Das Partitionierungskonzept von IMA bzw. ARINC wird realisiert, indem sämtliche Anwendungen bzw. Partitionen eines Avionikrechnersystems sowohl räumlich als auch zeitlich voneinander separiert werden. Somit wird auch die nebenläufige Ausführung von Anwendungen unterschiedlicher Kritikalitätsstufen ermöglicht. Die räumliche Trennung bzw. der Speicherschutz verhindern, dass eine Anwendung den Speicherbereich

einer anderen Anwendung überschreiben kann und dadurch Daten oder Code verändert. Die zeitliche Separation teilt die zur Verfügung stehende Rechenzeit auf die einzelnen Partitionen auf und unterbindet das Überschreiten des vorgegebenen Zeitbudgets der jeweils aktiven Partition.

Scheduling: Die zeitliche Partitionierung ist durch ein festes, zyklisches Scheduling-Verfahren realisiert. Dabei wird jeder Partition ein Zeitfenster zugeteilt. Solange das Zeitfenster einer Partition aktiv ist, hat die entsprechende Anwendung exklusiven Zugriff auf die Prozessorrechenzeit. Um das Scheduling zu vereinfachen, sind die Partitionsfenster in einem *Major Time Frame* 'Hauptzeitfenster', angeordnet, das sich über die gesamte Laufzeit periodisch wiederholt. Die Partitionsfenster werden anhand ihres jeweiligen Offsets plus dem aktuellen Startzeitpunkt des Hauptzeitfensters aktiviert und sind für die Dauer der Ausführung aktiv. Jeder Partition ist mindestens ein Prozess zugewiesen und jede aktive Partition führt genau einen Prozess gleichzeitig aus. Bei Partitionen mit mehreren Prozessen teilen sich diese einen gemeinsamen Adressraum und die zur Verfügung gestellten Ressourcen. Um mehreren konkurrierenden Prozessen einer Partition Zugriff auf die Rechenzeit zu gewähren, findet ein Scheduling auf Prozessebene statt, welches unabhängig vom Partitions-Scheduling abläuft. Dabei kommt ein prioritätsbasiertes, präemptives Scheduling-Verfahren zum Einsatz.

Kommunikation: Neben der strikten Trennung von Partitionen muss das Betriebssystem auch den Datenaustausch zwischen Partitionen abwickeln. Dabei findet die Kommunikation zwischen Partitionen durch Nachrichten über so genannte (Kommunikations-) Kanäle statt, wobei Kanäle auch zwischen Partitionen unterschiedlicher LRUs bestehen können. Jeder Kanal hat einen Sender und einen oder mehrere Empfänger, über den Uni-, Multi- und Broadcast-Nachrichten übertragen werden können. Empfänger einer Nachricht ist stets eine Partition und kein Prozess innerhalb einer Partition. Auf Anwendungsebene werden Nachrichten nur als gesamte Einheit verarbeitet, unvollständige Nachrichten oder Teile einer Nachricht werden verworfen bzw. ignoriert. In den darunter liegenden Ebenen können Nachrichten beliebig unterteilt und weiterverarbeitet werden, solange die Nachricht im ursprünglichen Zustand beim Empfänger ankommt. Für die Übertragung von Nachrichten gibt es zwei Modi, Sampling- und Queuing-Modus. Im Sampling-Modus transportieren Nachrichten üblicherweise identische Daten, allerdings mit aktualisierten Werten. Eine neu eingetroffene Nachricht beim Empfänger überschreibt den vorhandenen

Datenwert unabhängig davon, ob der alte Wert gelesen wurde oder nicht. Daher kann der Sender zu jeder Zeit eine neue Nachricht schicken. Im Gegensatz zum Sampling-Modus ist beim Queuing-Modus jede Nachrichteninstanz von Interesse und darf nicht überschrieben werden oder sollte unbeabsichtigt verloren gehen. Um mehrere Nachrichten zu speichern und somit ein Überschreiben zu verhindern, werden diese in einem First In – First Out (FIFO)-Puffer abgelegt, dessen maximale Kapazität an Nachrichten je nach Anwendung entsprechend konfiguriert wird.

Für die Kommunikation zwischen Prozessen innerhalb von Partitionen stehen ebenfalls zwei verschiedene Mechanismen zur Verfügung. Dabei handelt es sich um Buffers ‘Puffer’, und Blackboards. Die Kommunikation findet dabei nur zwischen Prozessen innerhalb derselben Partition und nicht partitionsübergreifend statt. Puffer besitzen FIFO-Queues und speichern jede Nachrichteninstanz, analog zum Queuing-Modus. Die Arbeitsweise von Blackboards ist ähnlich der vom Sampling-Modus bei der Kommunikation zwischen Partitionen. Jeweils die zuletzt gesendete Nachricht ist für alle anderen Teilnehmer lesbar, bis diese durch eine neue Nachricht überschrieben oder gelöscht wird.

Synchronisation: Semaphore und Events dienen der Prozesssynchronisation. Wie bei der Prozesskommunikation ist die Anwendbarkeit auf Prozesse innerhalb derselben Partition beschränkt. Semaphore regeln den kontrollierten Zugriff auf gemeinsam verwendete Partitionsressourcen durch gegenseitigen Ausschluss. Events werden dazu verwendet, um Prozessen das Auftreten von Ereignissen zu signalisieren. Sie dienen dazu, binäre Informationen zu übertragen. Wartet ein Prozess auf ein bestimmtes Ereignis und wird dessen Bedingung erfüllt, so kann der Prozess entsprechend darauf reagieren.

Eingabe und Ausgabe: Eingebettete Avioniksysteme müssen eine Vielzahl an Anforderungen erfüllen, u. a. dass sie sicher, echtzeitfähig, deterministisch und vorhersagbar konzipiert sind (Boniol 2013). Dies gilt auch für die Anbindung von E/A-Komponenten an Avionikrechnersysteme, wie beispielsweise Kommunikationsbusse oder Sensoren. Da Partitionen nach einem festen Zeitschema geplant werden, ist es von Vorteil, wenn E/A-Anfragen zu dem Zeitpunkt eintreffen, in dem das Partitionsfenster der entsprechenden Partition aktiv ist. Daher ist es lohnend, wenn E/A-Anfragen zeitlich vorhersagbar sind. Falls dies nicht möglich ist, können Anfragen u. a. gepuffert oder der Status aktiv vom entsprechenden Treiber abgefragt werden (Müller et al. 2014).

3.2.1.3 Baumaschinenbranche

Die Baumaschinenanwendung, die im Rahmen des parMERASA Projekts Verwendung fand (Gerdes et al. 2011), läuft auf einer ESX-3XL Steuerung, einem elektronischen Steuergerät auf Basis eines 32-Bit-Mikrocontrollers. Im Gegensatz zu den beiden vorhergehend besprochenen Domänen gibt es keine standardisierte Systemarchitektur. Stattdessen abstrahiert eine eigenentwickelte Middleware des Baumaschinenherstellers zusammen mit dem ESX-3XL-BIOS die Hardware von der Anwendungssoftware. Das ESX-3XL-BIOS bietet minimale Betriebssystemfunktionalitäten und einen *Hardware Abstraction Layer* ‘Hardwareabstraktionsschicht’, für die Programmiersprache C (Sensor-Technik Wiedemann GmbH 2011).

Eine andere und weit verbreitete Möglichkeit sind modulare Steuerungen auf Basis von Industrie-PCs oder speicherprogrammierbare Steuerungen (SPSen). Industrie-PCs ähneln IBM-kompatiblen PCs mit dem Unterschied, dass sie den besonderen Anforderungen (Sicherheit, Robustheit, etc.) im industriellen Umfeld genügen. Außerdem sind Industrie-PCs kompatibel mit Standardsoftware, wodurch ein sehr großer Fundus an Softwarelösungen zur Verfügung steht. Dies gilt auch für die Wahl des Betriebssystems, wobei aufgrund von eventuellen Echtzeitanforderungen spezialisierte Betriebssysteme oder spezielle Anpassungen von Standard-Betriebssystemen zum Einsatz kommen, z. B. QNX Neutrino RTOS (QNX Software Systems Ltd. 2014) oder ein Real-Time Linux.

Je nach Einsatzzweck können auch speicherprogrammierbare Steuerungen zur Anwendung kommen. Eine SPS hat im einfachsten Fall Eingänge, Ausgänge und einen programmierbaren Speicher zur Ausführung benutzerorientierter Steuerungsanweisungen (Firmware) (Deutsches Institut für Normung e.V. 2004). Prinzipiell ist eine SPS ein speziell auf industrielle Steuerungsaufgaben zugeschnittener Computer. Dabei unterscheidet man drei verschiedene Arten von SPSen: Hardware-SPS, Slot-SPS und Soft-SPS. Eine Hardware-SPS besteht zumeist aus mehreren Modulen, die über einen Bus gekoppelt sind. Das zentrale Modul beinhaltet die Recheneinheit mit zugehöriger Firmware und wird um E/A-Module ergänzt. Bei der Slot-SPS läuft die SPS auf einer autarken PC-Einsteckkarte mit dedizierter Hardware und eigenem Betriebssystem. Der PC dient als Stromquelle und der Datenaustausch zwischen SPS und PC ist problemlos möglich. Im Gegensatz dazu wird eine Soft-SPS auf einem PC oder Industrie-PC mit entsprechenden Eingabe- und Ausgabe-Einsteckkarten unter einem Standard-Betriebssystem ausgeführt (Seitz 2012).

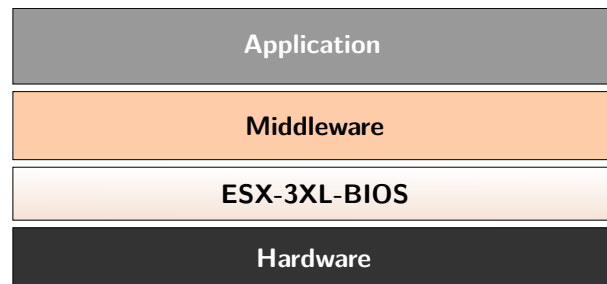


Abbildung 3.3: Systemarchitektur einer exemplarischen Baumaschinenanwendung

Neben der Programmierung in C unterstützt die ESX-3XL Steuerung auch die nach EN 61131-3 (Deutsches Institut für Normung e.V. 2003) normierten Programmiersprachen für speicherprogrammierbare Systeme. Bei der ESX-3XL Steuerung handelt es sich im Grunde um eine Hardware-SPS mit der Möglichkeit, die Anwendungssoftware alternativ in der Hochsprache C zu programmieren. Soft-SPSen und Industrie-PCs bauen auf PC ähnlicher Hardware auf und unterscheiden sich dagegen deutlich von der hier betrachteten Baumaschinensteuerung. Aufgrund des unterschiedlichen Aufbaus zu Industrie-PCs und dem Umstand, dass die Anwendung in der Hochsprache C und nicht in einer Programmiersprache für speicherprogrammierbare Systeme programmiert ist und mangels frei verfügbarer Betriebssysteme von SPSen, werden diese Systeme nachfolgend nicht weiter berücksichtigt.

Abbildung 3.3 zeigt die Unterteilung der Systemarchitektur der Baumaschinenanwendung. Das *ESX-3XL-BIOS* abstrahiert von der *Hardware* und umfasst diverse Treiber und einen minimalen Betriebssystemkern. Dieser beinhaltet u. a. einen Scheduler. Auf dem Basic Input Output System (BIOS) aufsetzend befindet sich die *Middleware*, die den Funktionsumfang des BIOS um ein erweitertes Scheduling-Verfahren ergänzt und eine anwendungsspezifische Ansteuerung von E/A-Geräten ermöglicht. Die *Application* ‘Anwendung’ besteht zumeist aus mehreren Tasks und setzt auf der Middleware API auf.

Scheduling: Die Ablaufplanung der Tasks einer Anwendung erfolgt zweistufig, wobei die Tasks in erster Instanz vom BIOS anhand ihrer Prioritäten geplant werden. Hierfür kommt ein prioritätsbasiertes, präemptives Scheduling-Verfahren zum Einsatz. In zweiter Instanz übernimmt die Middleware die Planung und setzt hierbei auf ein Round-Robin Scheduling.

Hierfür werden Tasks gleicher Priorität zu Gruppen zusammengefasst und in einer Ringliste angeordnet. Diese werden entsprechend der Reihenfolge in der Liste nacheinander eingeplant. Es wird jeweils der Task an der aktuellen Listenposition ausgeführt, sofern dieser die derzeitig höchste Priorität aller Tasks außerhalb der Gruppe besitzt.

Schutz: Das ESX-3XL-BIOS bietet im Gegensatz zu AUTOSAR und ARINC 653 optional nur die Absicherung vor unerlaubtem Zugriff auf den Stack-Speicherbereich und nicht für die Code- und Datensegmente eines Tasks an. Ferner wird eine Überwachung des zeitlich und logisch korrekten Programmflusses unterstützt.

Synchronisation: Zur Synchronisation zwischen Tasks stellt das ESX-3XL-BIOS Funktionen zur Absicherung von kritischen Sektionen und zur Verwaltung von Events bereit. Die kritischen Sektionen dienen zur Absicherung von Code-Bereichen, deren Ausführung nicht unterbrochen werden darf oder auf die von verschiedenen quasi-parallelen Tasks simultan zugegriffen wird. Durch diesen Mechanismus wird eine Aussetzung der Task-Ausführung durch höher priorisierte Tasks oder bestimmte IRQs unterbunden. Ferner bietet das BIOS, analog zu AUTOSAR, die Verwendung von Events an. Es werden bis zu 32 Event-Flags unterstützt. Da im parMERASA Projekt ein Hauptaugenmerk die Ausführung von parallelisierten Anwendungen auf mehreren Kernen ist, wurde die ESX-3XL-BIOS API um einen Lock-Mechanismus erweitert. Im Gegensatz zu kritischen Sektionen gewährt der Lock-Mechanismus einen wechselseitigen Ausschluss auf geteilte Ressourcen, auf denen ein gleichzeitiger Zugriff von echt-parallelen Tasks erfolgt, sprich von Tasks, die auf unterschiedlichen Prozessorkernen ausgeführt werden.

Kommunikation: Zur Kommunikation zwischen verschiedenen Tasks verwendet die Anwendung globale Variablen. Ein dedizierter Kommunikationsmechanismus ist nicht vorgesehen. Lese- und Schreibzugriffe auf Daten in Datenbuswortbreite werden nicht explizit gesichert, da der Zugriff hierbei atomar erfolgt. Bei größeren Daten oder Datenstrukturen wird der simultane Zugriff von verschiedenen quasi-parallelen Tasks durch kritische Sektionen gesichert. Dieses Konzept wird beim Austausch von Daten zwischen echt-parallelen Tasks beibehalten, allerdings wird zur Wahrung der Datenkonsistenz der Lock-Mechanismus verwendet.

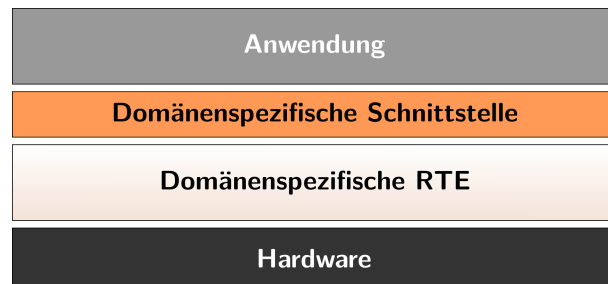


Abbildung 3.4: Domänenübergreifender Systemarchitekturaufbau

Eingabe und Ausgabe: Ähnlich dem Automobilsektor, interagiert die Anwendung mit ihrem Umfeld weitestgehend durch Aktuatoren und Sensoren. Dabei ist es wichtig, dass die Kommunikation mit E/A-Geräten mit möglichst geringer Latenz vonstattengeht, um umgehend auf externe Ereignisse reagieren zu können, wie etwa die Überschreitung zulässiger Betriebsparameter (Geschwindigkeit, Gewicht), die beispielsweise ein Umkippen der Maschine oder die Gefährdung von Personen nach sich ziehen würde.

3.2.1.4 Vergleich

Alle drei vorgestellten Systemarchitekturen besitzen einen ähnlichen Aufbau, der sich im Allgemeinen wie in Abbildung 3.4 domänenübergreifend darstellen lässt. Jede Systemarchitektur basiert auf einer für ihre Domäne typischen Hardwareplattform. Aufbauend auf der *Hardware* folgt die Softwarearchitektur mit der *Anwendung* und dem domänenspezifischen Software-Stack (vgl. Abschnitt 3.2), der wiederum in eine *domänenspezifische Laufzeitumgebung* und *Schnittstelle* untergliedert ist. Im Falle von AUTOSAR sind dies die BSW und die AUTOSAR RTE, bei ARINC das Board Support Package (BSP) und ARINC 653 O/S und bei der Baumaschinenanwendung das ESX-3XL-BIOS und die Middleware. Eng damit verbunden sind die domänenspezifischen Schnittstellen, dies sind AUTOSAR Interface, APEX API und Middleware API.

Tabelle 3.1 fasst nochmals die charakteristischen Merkmale der jeweiligen Softwarearchitektur bezüglich ihrer Funktionalität zusammen. Hierbei wurden ausschließlich für das Betriebssystem relevante Funktionalitäten betrachtet. Zudem sei angemerkt, dass sich die Anforderungen auf die zu diesem Zeitpunkt (Stand Juni 2012) spezifizierten Domänen-

Tabelle 3.1: Gegenüberstellung der domänenspezifischen Anforderungen an die jeweilige Softwarearchitektur

Domäne Funktion	Automobilsektor (AUTOSAR)	Luftfahrt (ARINC 653)	Baumaschinen- branche (ESX-3XL-BIOS)
Scheduling	festes präemptives Prioritäts- Verfahren	festes zyklisches + präemptives Prioritäts- Verfahren	präemptives Prioritäts- + Rundumlauf- Verfahren
Schutz	OS-Application	Partition	Task-Stack
Kommunikation	Client-Server, Sender-Receiver	Nachrichten, Puffer, Blackboards	(globale Variablen)
Synchronisation	Events, Ressourcen, Scheduler als Ressource, Spin-Locks	Events, Semaphore, Preemption Locking, (Locks)	Events, kritische Sektionen, (Locks)
Ein-/Ausgabe	geringe Latenz	zeitliche Vor- hersagbarkeit	geringe Latenz

Standards stützen. Des Weiteren beziehen sich ARINC 653 (Aeronautical Radio, Inc. 2010) und ESX-3XL-BIOS (Sensor-Technik Wiedemann GmbH 2011) auf Anwendungen, die nebenläufig, sprich quasi-parallel, auf einem Singlecore-Prozessor ausgeführt werden. Es gibt jedoch Untersuchungen bezüglich der Eignung von Multicore-Prozessoren in Avioniksystemen (Löfwenmark et al. 2014). Im Automobilbereich wurde mit AUTOSAR Release 4.0 (AUTOSAR 2011) die Unterstützung von Multicore-Prozessoren eingeführt. Somit ist auch eine echt-parallele Ausführung von Tasks auf mehreren Kernen möglich. Die Funktionalitäten bei AUTOSAR in Tabelle 3.1 decken daher sowohl quasi- als auch echt-parallele Ausführung ab. Bei ARINC 653 und ESX-3XL-BIOS sind dagegen die Anforderungen für Singlecore-Plattformen angegeben. Im Folgenden wird erläutert, inwiefern die Funktionalitäten für eine echt-parallele Ausführung abgeändert oder erweitert werden müssen. Ferner stellen die geklammerten Merkmale Erweiterungen dar, die im Zuge des parMERASA Projekts ergänzt wurden, um eine parallele Ausführung zu ermöglichen.

Scheduling: Aus Tabelle 3.1 wird ersichtlich, dass zwar alle drei Softwarearchitekturen ein prioritätsbasiertes, präemptives Scheduling-Verfahren verwenden, bei zwei Domänen allerdings in Kombination mit einem zweiten Scheduling-Verfahren. Zudem unterscheiden sich die Reihenfolgen der Scheduling-Verfahren. Unter AUTOSAR und ESX-3XL-BIOS wird primär prioritätsbasiert geplant, wohingegen bei ARINC 653 an erster Stelle ein zyklisches Scheduling-Verfahren steht und erst an zweiter Stelle das Prioritätsverfahren. Im Falle einer Multicore-Plattform besitzt bei AUTOSAR jeder Kern seinen eigenen, unabhängigen Scheduler, d. h. es findet kein globales Scheduling statt.

Für die Baumaschinenanwendung wurde das zweistufige Scheduling-Verfahren für Multicore-Plattformen erweitert und analog zu AUTOSAR ein eigenständiger Scheduler je Kern vorgesehen. Da die exemplarische Avionikanwendung aus nur einer Partition besteht und diese deshalb durchgehend aktiv ist, ist eine Synchronisation der Partitionen auf verschiedenen Kernen nicht vonnöten. Somit genügt ein lokaler Scheduler je Prozessorkern mit gegenseitiger Abstimmung der Prozess-Scheduler, beispielsweise durch Events. Bei anderen Avionikanwendungen und der Verwendung von mehreren Partitionen muss u. U. ein komplexeres Verfahren zur Synchronisation der Scheduler auf unterschiedlichen Kernen eingesetzt werden.

Schutz: Ein weiterer Punkt ist die strikte Trennung verschiedener Anwendungen bzw. Anwendungsbereiche voneinander. Hier nehmen Avioniksysteme eine Sonderrolle ein, da sie nach höchsten Sicherheitsintegritätslevels zertifiziert werden müssen. Auch im Automobilssektor und in der Baumaschinenbranche müssen elektronische Steuerungen unterschiedlichste Sicherheitsstandards erfüllen, z. B. die Norm ISO 26262 (Internationale Organisation für Normung 2011). Um dies zu erreichen, ist eine seit langem bestehende Anforderung das Konzept der sogenannten *Freedom of Interference* ‘Rückwirkungsfreiheit’. Die Umsetzung erfolgt dabei u. a. durch eine zeitliche und räumliche Trennung der zu schützenden Bereiche. Diese Aufgabe übernimmt bei ARINC 653 der Scheduler, indem er den Partitionen strikte Zeitfenster zuweist und die Zugriffsrechte auf die einzelnen Speicherbereiche verwaltet. Bei AUTOSAR können die Speichersegmente und der Stack-Speicher von Tasks verschiedener OS-Applications voneinander geschützt werden. Zudem ist eine Überwachung der Zeitbudgets von Tasks möglich. Das ESX-3XL-BIOS bietet dagegen nur eine Überwachung der Task-Stacks an. Alle drei Softwarearchitekturen erlauben zudem die Überwachung von zeitlicher und logischer Korrektheit des Programmflusses.

Da die AUTOSAR Spezifikation nur eine relativ grobkörnige Parallelisierung auf der Ebene von SW-Cs gestattet, wurde diese Vorschrift umgangen, um auch eine feingranulare Parallelisierung von Anwendungsteilen zu ermöglichen (vgl. Abschnitt 2.3.2). Dazu wurde die Definition der OS-Application dahingehend erweitert, dass diese auch mehr als einen Prozessorkern umfassen darf. Ähnlich sieht es für die Avionikanwendung aus. Hier darf einer Partition nun ebenfalls mehr als ein Kern zugewiesen sein. Somit haben echt-parallele Prozesse derselben Partition Zugriff auf einen gemeinsamen Speicherbereich. Für die Parallelisierung der Baumaschinenanwendung ist keine Änderung am Stack-Monitoring erforderlich.

Synchronisation: Zur Synchronisation bieten alle drei Softwarearchitekturen Events an. Ein Prozess bzw. Task kann durch Aufruf einer *Wait-Event* Operation auf ein bestimmtes Ereignis von einem anderen Prozess respektive Task warten. Dabei wird die aufrufende Einheit schlafen gelegt, bis das entsprechende Ereignis eintritt und der Scheduler übergibt zwischenzeitlich einer anderen betriebsbereiten Einheit die Prozessorzeit. Des Weiteren verwendet AUTOSAR das Ressourcen-Modell für quasi-parallele Tasks auf demselben Kern, das einen wechselseitigen Ausschluss garantiert und zudem eine Prioritätsinversion durch das *Immediate Priority Ceiling Protocol* vermeidet. Bei ARINC geschieht dies durch Semaphore. Im Gegensatz zum Ressourcen-Modell bei AUTOSAR, bei dem jede Ressource nur einmal vorhanden ist, erlaubt ein Semaphor auch die Verwaltung beschränkter Ressourcen mit einer Anzahl größer als Eins. Allerdings bieten Semaphore von sich aus keinen Schutz vor Prioritätsinversion. Bei beiden Mechanismen kann der aktive Task bzw. Prozess während des Zugriffs auf geteilte Ressourcen von höher priorisierten Tasks respektive Prozessen unterbrochen werden. Dies kann bei allen drei Softwarearchitekturen verhindert werden. Durch den ‘Scheduler als Ressource’ Mechanismus, *Preemption Locking* und kritische Sektionen wird ein Eingreifen des Schedulers und insofern ein Task-/Prozesswechsel innerhalb eines bestimmten Codeabschnitts unterbunden. Mit Hilfe von kritischen Sektionen kann somit auch beim ESX-3XL-BIOS ein wechselseitiger Ausschluss ermöglicht werden, da hierfür kein expliziter Mechanismus vorhanden ist.

Bei AUTOSAR können Events nicht nur für nebenläufige Tasks, sondern auch für Tasks auf unterschiedlichen Kernen verwendet werden. Das ESX-3XL-BIOS wurde derartig angepasst, dass Benachrichtigungen von Events ebenfalls an Tasks auf anderen Kernen geschickt werden können. Beim erweiterten ARINC 653 unterstützt eine Partition echt-

parallele Prozesse. Im Zuge dessen wurde der Mechanismus von Events für Multicore-Prozessoren entsprechend angepasst. Dies gilt auch für das ESX-3XL-BIOS. Die kurzzeitige Sperrung des Schedulers und die Mechanismen zum wechselseitigen Ausschluss sind auch weiterhin nur für die Verwendung von quasi-parallelen Tasks/Prozessen gedacht. Für die Synchronisation von echt-parallelen Tasks ist bei AUTOSAR explizit der Spin-Lock Mechanismus verantwortlich. Dabei wartet der jeweilige Task aktiv (*busy-waiting*) solange, bis der Zugriff auf den Lock gewährt wird. Zur besseren zeitlichen Abschätzung wurde der Spin-Lock durch einen Ticket-Lock Mechanismus ausgetauscht. Die Semantik dagegen ist unverändert. Sowohl ARINC 653 als auch das ESX-3XL-BIOS bieten kein entsprechendes Verfahren an. Aus diesem Grund wurden diese um einen zeitlich abschätzbaren Mechanismus zum wechselseitigen Ausschluss ergänzt. Dabei handelt es sich ebenfalls um das Ticket-Lock Verfahren. Auf anderen Multicore-Plattformen, bei denen der Ticket-Lock Mechanismus nicht realisiert werden kann, können auch andere Verfahren verwendet werden, sofern diese zeitlich abschätzbar sind.

Kommunikation: Die Sender-Receiver ebenso wie die Client-Server-Kommunikation sind gleichermaßen für quasi- und echt-parallele Tasks angedacht. Dabei kann die Sender-Receiver-Kommunikation sowohl *busy-waiting* ‘aktiv wartend’, als auch *non-busy-waiting* erfolgen. Bei der synchronen Client-Server-Kommunikation wartet der aufrufende Task aktiv auf die Antwort vom Server, bei der asynchronen Variante kann der Task andere Aufgaben abarbeiten oder schlafen gelegt werden. Ähnlich verhält es sich bei ARINC 653 mit der Nachrichten-Kommunikation zwischen Partitionen. Im Sampling-Modus liebt der Empfänger die zuletzt gesendete Nachricht. Falls keine Nachricht vorhanden ist, wird ein Fehler zurückgegeben und der Aufruf beendet. Im Queuing-Modus wird in solch einem Fall der empfangende Prozess schlafen gelegt, bis die entsprechende Nachricht eintrifft. Bei Puffern und Blackboards erfolgt die Kommunikation *non-busy-waiting*, sprich wenn keine Nachricht vorhanden ist, wird der beteiligte Prozess gleichfalls schlafen gelegt. Das ESX-3XL-BIOS bietet kein explizites Kommunikationsmodell. Die Baumaschinenanwendung verwendet zum Austausch von Nachrichten globale Variablen. Bei Datenstrukturen wird der Zugriff durch kritische Sektionen geschützt.

Diese Möglichkeit des Nachrichtenaustauschs wurde bei der Erweiterung auf Multicore-Plattformen beibehalten. Lediglich der Zugriffsschutz auf Datenstrukturen erfolgt bei echt-parallelen Tasks durch einen Lock-Mechanismus. Bei ARINC 653 sind für die Nach-

richtenkommunikation keine speziellen Änderungen erforderlich. Die kommunizierenden Partitionen können ohnehin unterschiedlichen LRUs zugewiesen sein. Die einzig neu hinzugekommene Konstellation besteht darin, dass sich die beteiligten Partitionen auf unterschiedlichen Kernen derselben LRU befinden können. Puffer und Blackboards wurden dahingehend erweitert, dass der Informationsaustausch zwar weiterhin Prozessen innerhalb ein und derselben Partition vorbehalten ist, die Prozesse allerdings unterschiedlichen Kernen zugeordnet sein dürfen.

Eingabe und Ausgabe: Bei der Eingabe und Ausgabe sind sowohl im Automobilbereich als auch bei der Baumaschinenbranche geringe Latenzen gefordert. Das bedeutet im Umkehrschluss, dass die Latenz zwischen Eintritt eines Ereignisses und Beginn der ISR möglichst gering ist. Um dies zu erreichen, sind kurze Kontextwechselzeiten seitens des Schedulers gefragt. Bei ARINC hingegen ist die zeitliche Vorhersagbarkeit von E/A-Anfragen interessant, weshalb IRQs nach Möglichkeit vermieden werden. Das parMERASA Projekt hatte zum Ziel, eingebettete Anwendungen zu parallelisieren und diese auf einer Multicore-Plattform auszuführen und nicht mehrere Anwendungen parallel abzuarbeiten. Somit kann ein simultaner Zugriff auf Peripheriegeräte ausgeschlossen werden, da die Anwendung selbst für den exklusiven Zugriff auf E/A-Einheiten zuständig ist.

3.2.2 Spezifikation

Ein Ziel des parMERASA Projekts bestand darin, eine gemeinsame Systemarchitektur für die Beispielanwendungen aus dem Automobilbereich, der Luftfahrt und der Baumaschinenbranche zu definieren. Abschnitt 3.2.1 beleuchtet Gemeinsamkeiten und Unterschiede in Bezug auf die etablierten Systemarchitekturen der drei verschiedenen Industriezweige. Die Aufbauten der drei Systemarchitekturen sind sich prinzipiell sehr ähnlich (vgl. Abbildung 3.4), allerdings unterscheiden sie sich hinsichtlich der Anwendungsschnittstellen und den Anforderungen an die jeweiligen Funktionalitäten der RTEs, insbesondere die der Scheduling-Funktionalität (vgl. Tabelle 3.1). Aufgrund dieser Unterschiede hätte eine Implementierung einer gemeinsamen Systemsoftware bzw. der Betriebssystemfunktionalitäten für alle drei Domänen zur Folge gehabt, dass mehrere Scheduling-Verfahren und die verschiedenen Schutzmechanismen in einen Betriebssystemkern hätten integriert werden

müssen. Dies hätte zu Kompromissen und letztendlich zu Einbußen bei der Performance geführt.

Aus diesen Gründen wurde ein zweistufiger Ansatz gewählt. Abbildung 3.5 verdeutlicht diesen. Dabei werden Teile der ursprünglichen RTEs in eine sogenannte *Kernel Library* ‘Kernel-Bibliothek’ (Bradatsch et al. 2013)) ausgegliedert und die übrigen Funktionalitäten verbleiben in den geringfügig *reduzierten, domänenspezifischen RTEs*. Die Kernel-Bibliothek enthält grundlegende Funktionalitäten eines Betriebssystemkerns, die auch Bestandteil eines vollständigen Echtzeitbetriebssystems sind. Hierbei handelt es sich um Funktionen bzw. *Dienste*, die von allen RTEs benötigt werden und nicht von einer domänenspezifischen Implementierung abhängig sind. Dadurch ist die Realisierung einer einheitlichen Schnittstelle zwischen Kernel-Bibliothek und RTEs möglich.

Sowohl die Kernel-Bibliothek als auch Teile der RTEs sind in vier Kategorien unterteilt. In Anlehnung an die domänenspezifischen Anforderungen aus Tabelle 3.1 sind dies bei den RTE Betriebssystemfunktionalitäten die Bereiche *Scheduling, Schutz, Kommunikation und Synchronisation* und *Eingabe und Ausgabe*. Diese sogenannten RTE Basisdienste umfassen überwiegend Funktionen, die von allen drei Softwarearchitekturen benötigt werden, sich in ihrer Implementierung indessen unterscheiden. Entsprechend den RTE Basisdiensten sind auch die Kernel-Dienste in vier Kategorien unterteilt. Dabei handelt es sich im einzelnen um *Kontextverwaltung, Speicherverwaltung, Synchronisationsmechanismen* und *Ausnahmebehandlung*. Jeder RTE Basisdienst nutzt Funktionen eines oder mehrerer Kernel-Dienste. Neben den RTE Basisdiensten gibt es noch weitere domänenspezifische RTE Dienste, die auf den Basisdiensten aufsetzen, wie etwa ein Dienst zur Ansteuerung von Feldbussen, der hierfür die E/A-Dienste verwendet.

Ein weiterer Aspekt der Kernel-Bibliothek ist die Möglichkeit zur flexiblen Unterstützung von Singlecore- und Multicore-Plattformen. Die beiden domänenspezifischen Standards ARINC 653 und ESX-3XL-BIOS definieren Anforderungen für Singlecore-Plattformen mit quasi-paralleler Ausführung. Im Rahmen des parMERASA Projekts wurden die Beispielanwendungen dieser beiden Domänen auf eine Multicore-Plattform portiert. Dabei wurden nicht spezifizierte Erweiterungen in die jeweilige Systemsoftware integriert. Mit Hilfe der Kernel-Bibliothek ist es möglich, sowohl die Singlecore- als auch Multicore-Implementierung gleichermaßen zu unterstützen. Da die Dienste der Kernel-Bibliothek keine gegenseitigen Abhängigkeiten besitzen, können weitere Dienste, die zur Unterstützung von

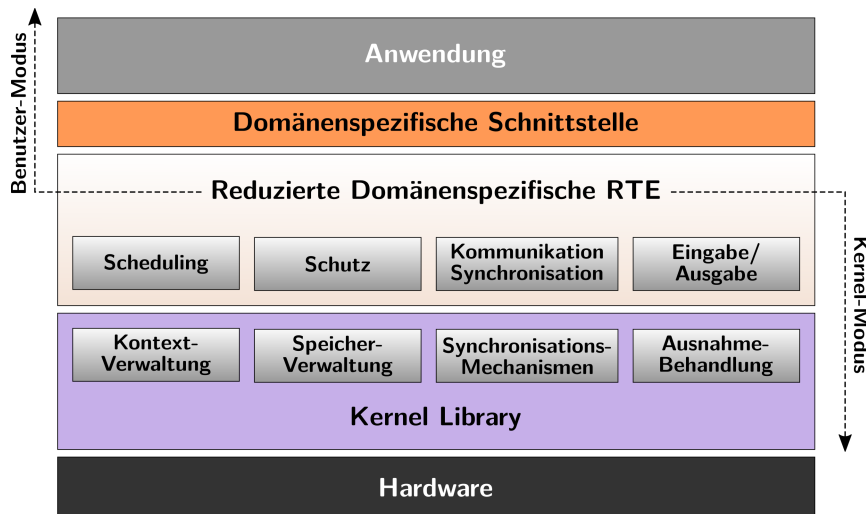


Abbildung 3.5: Domänenübergreifende Systemarchitektur mit zweigeteilter RTE

mehreren Kernen benötigt werden, problemlos hinzugefügt werden. Anders verhält es sich bei den RTE Basisdiensten. Durch die enge Verflechtung des Scheduling-Dienstes und des Synchronisations- und Kommunikationsdienstes erfordert eine Erweiterung für Multicore-Prozessor-Unterstützung einige Änderungen an den Basisdiensten und verursacht somit teilweise Inkompatibilitäten zum jeweiligen Standard.

Des Weiteren stellt jede Softwarearchitektur Schutzmechanismen bzw. zeitliche und räumliche Separation bereit. Um eine räumliche Trennung bzw. Speicherschutz zu ermöglichen, muss dies einerseits die Hardware in Form von einer MPU oder MMU unterstützen und andererseits die Codeausführung in unterschiedlichen Privilegierungsstufen erfolgen. Die meisten Mikrocontroller unterstützen zwei Privilegierungsstufen, einen *privilegierten Kernel-Modus* und einen *unprivilegierten Benutzer-Modus*, wobei Ersterer uneingeschränkten Zugriff auf die Hardware und Prozessorregister ermöglicht.

3.2.2.1 Kernel Dienste

Die Kernel-Dienste, welche in der Kernel-Bibliothek zusammengefasst sind, bilden die gemeinsame Basis für alle domänenspezifischen RTEs. Die Kernel-Bibliothek stellt grundlegende Teile der Betriebssystemkernfunktionalitäten zur Verfügung, die für die Imple-

mentierung der drei RTEs nötig sind. Da der Betriebssystemkern zentraler Bestandteil des Gesamtsystems ist und als unterste Softwareschicht den Zugang zur Hardware verwaltet, wird dieser im Kernel-Modus mit der höchsten Berechtigungsstufe ausgeführt. Somit werden auch sämtliche Dienste der Kernel-Bibliothek im Kernel-Modus betrieben. Im Folgenden werden die Dienste der einzelnen Kategorien genauer beschrieben.

Kontextverwaltung: Eine zentrale Aufgabe der Kontextverwaltung ist, den Hardwarekontext des aktiven Prozesses zu sichern und unmittelbar im Anschluss den Kontext eines anderen Prozesses zu laden (vgl. Abschnitt 2.2.1). Hierfür findet ein sogenannter Kontextwechsel statt. Die Kontextverwaltung stellt Funktionen bereit, die einen Hardwarekontext sowohl initialisieren als auch durch einen anderen Kontext austauschen kann. Da die Größe eines Hardwarekontextes für eine gegebene Instruction Set Architecture (ISA) ‘Befehlssatzarchitektur’ feststeht, besitzt ein Kontextwechsel eine konstante Ausführungszeit, welche eine zeitliche Analyse wiederum vereinfacht. Die Sicherung eines Task- bzw. Prozesskontextes, der neben dem Hardwarekontext noch weitere Parameter beinhaltet, wie z. B. Prozess ID oder Priorität, ist nicht Aufgabe der Kontextverwaltung. Dies fällt in den Aufgabenbereich des Scheduling-Dienstes.

Speicherverwaltung: Die Speicherverwaltung dient dazu, auf unterster Ebene die geforderten Schutzmechanismen der jeweiligen domänenspezifischen Systemsoftware in Bezug auf die räumliche Separation bzw. den Speicherschutz umzusetzen. Dafür werden entsprechende Speicherzugriffsrechte und Speicherzuordnungen gesetzt. Dies wird – je nach Hardware – über spezielle MPU oder MMU Register konfiguriert. Um harte Echtzeitfähigkeit zu gewährleisten, müssen nichtdeterministische Ausnahmen aufgrund von Speicherschutzverletzungen und TLB Fehlzugriffen unterbunden werden. Da es sich hierbei um Anwendungen handelt, die echtzeitfähigen, teilweise sogar sicherheitskritischen Anforderungen genügen müssen, sind die Speicheranforderungen der jeweiligen Anwendung eindeutig spezifiziert. Somit kann eine dynamische Speicherzuteilung abgeschlossen werden, die einer einfacheren zeitlichen Abschätzung zugute kommt. Zudem ist zu beachten, dass der eingeplante Speicher des jeweils aktiven Prozesses bzw. Tasks reserviert ist.

Synchronisationsmechanismen: Die Synchronisationsmechanismen stellen Funktionen bereit, die eine load-modify-store Semantik besitzen, und darauf aufbauend einfache

Softwaresynchronisationsmechanismen. Je nach RTE dienen die load-modify-store Funktionen auch als Basis für komplexere Softwaresynchronisationsmechanismen. Auf Seite der in der Kernel-Bibliothek enthaltenen Softwaresynchronisationsmechanismen steht ein hart echtzeitfähiger, aktiv wartender Spin-Lock zur Verfügung. Im parMERASA Projekt wurde dieser als Ticket-Lock implementiert. Zusätzlich umfasst die Kernel-Bibliothek eine aktiv wartende Barriere. Beide Mechanismen wurden in früheren Arbeiten (Gerdes et al. 2012a; Gerdes et al. 2012b) auf echtzeitfähigen Multicore-Prozessoren untersucht. Des Weiteren muss über die Speicherverwaltung sicher gestellt sein, dass die beteiligten Prozesse bzw. Tasks Zugriff auf die entsprechenden Speicherbereiche haben.

Ausnahmebehandlung: Da jede ISA auf unterschiedliche Weise IRQs behandelt und auch die meisten Hardwareplattformen verschiedene Ausnahme- und Unterbrechungsquellen unterstützen, umfasst die Ausnahmebehandlung lediglich einen Systemzeitgeber, Systemaufrufe und eine grundlegende IRQ Behandlung von externen E/A-Geräten. Um – je nach Domäne und konkreter Anwendung – die Anforderungen an Speicherschutz und zeitliche Separation zu ermöglichen, wurde ein zweistufiger Ansatz gewählt. Zunächst behandelt eine kompakte Interrupt Service Routine (ISR) die Unterbrechung. Dabei werden die notwendigen Register und Flags gesetzt und der dazugehörige E/A-Dienst in der RTE aktiviert. Im zweiten Schritt, wird der entsprechende E/A-Dienst unter der Planung des Schedulers ausgeführt. Zeitintensive oder umfangreiche Berechnungen werden in dem jeweiligen E/A-Dienst behandelt. Um auf die Speicherbereiche zugreifen zu können, auf denen die jeweiligen E/A-Geräte abgebildet sind, müssen die Adressbereiche mittels der Speicherverwaltung entsprechend konfiguriert sein. Zudem umfasst die Ausnahmebehandlung rudimentäre Funktionen zum Setzen und Auslesen des Systemzeitgebers, da alle drei domänenspezifischen RTEs eine Echtzeituhr voraussetzen.

3.2.2.2 RTE Dienste

Bei dem im parMERASA Projekt verwendeten und in Abbildung 3.5 dargestellten Ansatz besteht die Systemsoftware aus den RTE Basisdiensten und den domänenspezifischen RTE Diensten. Ein Spezifikum der Basisdienste ist, dass sie Bestandteil einer jeden domänenspezifischen Softwarearchitektur sind. Je nach Industriezweig unterscheiden sie sich jedoch in ihren bereitgestellten Funktionalitäten. Zudem greifen die RTE Basisdienste

zur Bereitstellung ihrer Funktionalitäten auf Funktionen der Kernel-Dienste und/oder anderer RTE Basisdienste zu. Nachfolgend werden die einzelnen RTE Basisdienste und ihre jeweiligen Abhängigkeiten zu anderen Diensten erläutert.

Scheduling: Der Scheduler plant die zeitliche Abfolge der Anwendungsprozesse bzw. Tasks. Je nach Industriezweig ist gemäß der jeweiligen Spezifikation das Verfahren zur Ablaufplanung verschieden. Wie aus Tabelle 3.1 ersichtlich, unterstützen ARINC 653 und ESX-3XL-BIOS sogar ein zweistufiges Scheduling-Verfahren. Einen Teilaufgabenbereich des Scheduling-Dienstes deckt der Dispatcher ab. Zu dessen Aufgaben gehören der Task-Wechsel und die Überführung der dabei beteiligten Tasks von einem Task-Zustand in einen Anderen. Hierfür bedient er sich der Funktionen der *Kontextverwaltung* und der *Speicherverwaltung*. Letzteres wird zum Festlegen der Speicherzugriffsrechte des jeweiligen Prozesses (ARINC 653) bzw. Tasks (AUTOSAR) benötigt. Mit dem Scheduling-Dienst einhergehend findet auch die zeitliche Separation von Prozessen bzw. Tasks statt. Dazu wird bei einem Task-Wechsel das jeweilige Zeitbudget des Tasks festgelegt. Der Scheduling-Dienst verwendet zudem Funktionen der *Ausnahmebehandlung* zur Anpassung und zum Auslesen des Systemzeitgebers, um die Ablaufplanung zu steuern. Des Weiteren spielt der Scheduler eine maßgebliche Rolle für die effiziente Ausführung von parallelisierten Anwendungen. Der Scheduler kann entweder kernlokal oder global agieren, wobei bei beiden Varianten nochmals zwischen dynamischem und statischem Scheduling unterschieden wird.

Schutz: Dieser Dienst kann aus zwei Sichtweisen betrachtet werden. 1.) Funktionen zum Steuern von Schutzmechanismen und 2.) Privilegierungsstufen von Prozessen bzw. Tasks. Zu Punkt eins zählen beispielsweise Funktionen zum Überwachen des logischen und zeitlichen Programmablaufes. Diese können wiederum – je nach Implementierung – Gebrauch von einem Watchdog-Zeitgeber machen, der zu den *Eingabe und Ausgabe* Diensten zählt. Der zweite Punkt betrifft die räumliche Separation. Diese wird über die Speicherverwaltung gesteuert. Allerdings ist die räumliche Trennung auch als Teil des Schutzkonzepts der jeweiligen Domäne zu erachten. Sowohl bei den RTE Basisdiensten als auch bei den übrigen RTE Diensten kommt es auf die jeweilige Domäne, den Dienst und die konkreten Anwendungsanforderungen an, ob ein bestimmter Dienst im Kernel- oder Benutzer-Modus ausgeführt wird. Da die RTE Basisdienste überwiegend Betriebssystemkernfunktionen innehaben, werden diese zumeist im Kernel-Modus be-

trieben. Je nach Bedarf entfällt der Benutzer-Modus vollständig und die Anwendung wird im Kernel-Modus ohne Speicherschutzmechanismen realisiert, so z. B. bei Automobilanwendungen, bei denen AUTOSAR OS mit Scalability Class 1 konfiguriert ist. Bei ARINC hingegen werden sämtliche Anwendungssoftware-Partitionen im Benutzer-Modus ausgeführt, um dem Partitionierungskonzept von IMA gerecht zu werden und somit die Rückwirkungsfreiheit (siehe Abschnitt 3.2.1) zu gewährleisten.

Kommunikation: Der Kommunikationsdienst ist zuständig für den Datenaustausch sowohl innerhalb von Anwendungen, z. B. per Puffer oder Blackboards, als auch zwischen Anwendungsteilen, die unterschiedlichen Schutzbereichen, wie etwa OS-Applications, zugeordnet sind. Dabei haben u. a. die zugrunde liegenden Mechanismen der bereitgestellten Kommunikationsfunktionen Einfluss auf die Übertragungsgeschwindigkeit zwischen Kernen, die wiederum Auswirkungen auf die Leistungsfähigkeit paralleler Anwendungen hat. Seitens der vorgegebenen Softwarearchitektur des AUTOSAR Standards ist die Kommunikation zwischen unterschiedlichen Kernen durch den Inter-OS-Application Communicator (IOC) auf Sender-Receiver-Kommunikation beschränkt. Ein direkter Zugriff auf einen gemeinsamen Speicher wird nicht unterstützt (vgl. Abschnitt 2.3.2). Aufgrund des zusätzlichen Verwaltungsaufwands bezüglich der Sender-Receiver-Kommunikation gegenüber direkten Speicherzugriffen verringert sich der Datendurchsatz. Aber auch die jeweilige Handhabung zum Schutz von Prozessen bzw. Tasks hat Auswirkungen auf den Implementierungsoverhead und somit auf die Übertragungsleistung. Ein weiteres Merkmal ist, dass viele Kommunikationsparadigmen den Abbruch einer Verbindung nach einer zuvor festgelegten Zeitüberschreitung unterstützen. Dafür werden Funktionalitäten des *Schutzdienstes* verwendet.

Synchronisation: Zweck des Synchronisationsdienstes ist die Bereitstellung von thread-sicheren Zugriffsverfahren auf geteilte Systemressourcen, d. h. Zugangskontrolle zu Ressourcen, die von mehreren Prozessen bzw. Tasks simultan angefordert werden können. Dabei muss zwischen Synchronisierung von quasi- und echt-parallelen Tasks unterschieden werden (vgl. Abschnitt 2.1). Des Weiteren umfasst der Synchronisationsdienst komplexere, non-busy-waiting Softwaresynchronisationsmethoden, die einerseits Gebrauch von den *Synchronisationsmechanismen* der Kernel-Bibliothek machen und andererseits eine Interaktion mit dem *Scheduling*-Dienst voraussetzen, um Tasks bei Bedarf schlafen legen zu können. Ebenso unterstützen einige Synchronisationsmechanismen die Überprüfung

von Zeitüberschreitungen. Wie beim Kommunikationsdienst hat die konkrete Realisierung der Synchronisationsfunktionen Auswirkungen auf die Performance von parallelen Anwendungen. Die möglichen Umsetzungen hängen jedoch auch stark von der verwendeten Hardwareplattform ab.

Eingabe und Ausgabe: Der RTE Basisdienst zur Steuerung der E/A-Geräte unterscheidet sich insofern von den anderen Basisdiensten, dass die angebotenen Funktionalitäten abhängig sind von der Domäne und der zugrunde liegenden Hardwareplattform. Größtenteils können die bereitgestellten Funktionen auch nur hardwareabhängig implementiert werden. Somit dient der E/A-Dienst auch als Hardwareabstraktionsschicht für die darauf aufsetzenden RTE Dienste. Eine Funktionalität, die alle drei RTEs benötigen, ist die Konfiguration von Zeitgebern. Neben der Funktion zur Verwaltung des Systemzeitgebers in den Kernel-Diensten, stellt der E/A-Dienst eine Schnittstelle zur Administration eines Watchdogs zur Verfügung.

Der Kernel-Bibliothek-Ansatz verlagert domänenspezifische Funktionalitäten, die Betriebssystemkernfunktionen innehaben, in die jeweiligen RTE Basisdienste und die Kernel-Bibliothek beinhaltet nur Funktionen, die von allen drei betrachteten RTEs verwendet werden. Dadurch wird u. a. eine kompakte Größe der Kernel-Bibliothek ermöglicht. Daneben sind die Funktionen der Kernel-Bibliothek so implementiert, dass sie eine möglichst gute zeitliche Abschätzung erlauben. Insbesondere bei den Synchronisationsfunktionen können, je nach verwendetem Mechanismus, signifikante Überschätzungen bei einer Laufzeitanalyse zum Vorschein treten, oder im ungünstigsten Fall lässt sich keine obere Zeitschranke abschätzen, da der eingesetzte Mechanismus nicht fair ist (Gerdes 2013, S. 10). Somit wird durch den Einsatz der Kernel-Bibliothek die Laufzeitanalyse erleichtert. Abschließend fasst Tabelle 3.2 nochmals die Abhängigkeiten der RTE Basisdienste voneinander und zu den Kernel-Diensten zusammen.

3.2.3 Fallstudien

Im vorangegangenen Abschnitt wurde der Kernel-Bibliothek-Ansatz vorgestellt. Um die Machbarkeit zu demonstrieren und einen Konzeptnachweis vorzulegen, wird im Folgenden für alle drei Industriebereiche ein Fallbeispiel anhand der Integration der Kernel-Bibliothek in die für die jeweilige Domäne typischen Systemarchitektur präsentiert.

Tabelle 3.2: Abhängigkeiten der RTE Basisdienste von anderen RTE Basis- und Kernel-Diensten

RTE Basisdienste				
	Scheduling	Schutz	Komm. & Sync.	E/A
RTE Basisdienste				
Scheduling			×	
Schutz				
Komm. & Sync.				
E/A		×		
Kernel Dienste				
Kontext-Verwaltung	×			
Speicher-Verwaltung	×	×		
Synchronisation			×	
Ausnahme-Behandlung	×			×

3.2.3.1 Automobilssektor

Abbildung 3.6 zeigt die AUTOSAR Systemarchitektur mit der Erweiterung um den Kernel-Bibliothek-Ansatz. Um die Berührungspunkte der Kernel-Bibliothek mit der BSW aus Abbildung 2.2 genauer darstellen zu können, ist die BSW untergliedert in ihre drei Ebenen *Services Layer*, *ECU Abstraction Layer* und *MCAL* dargestellt. Daneben gibt es noch die *CDD*. Die Kernel-Bibliothek abstrahiert von der konkreten Hardwareplattform und offeriert ihre Dienste dem MCAL, ECU Abstraction Layer und Services Layer, wobei hier ausschließlich das *OS Modul* die zur Verfügung gestellten Dienste verwendet. Somit interagiert das OS Modul nicht mehr direkt mit der Hardware und eine Portierung auf eine andere ISA wird dadurch vereinfacht. Der MCAL greift sowohl indirekt über die Kernel-Bibliothek als auch weiterhin direkt auf die Hardware zu. Dies liegt darin begründet, dass die Kernel-Dienste nur Betriebssystemkernfunktionalitäten bereitstellen. Sämtliche darüber hinausgehenden Spezifika des Mikrocontrollers bzw. der Peripherie werden vom MCAL abstrahiert. Im Vergleich zur ursprünglichen AUTOSAR Softwarearchitektur aus Abbildung 2.1, bei welcher der ECU Abstraction Layer nur durch den MCAL auf die darunter liegende Hardwareplattform zugreifen kann, hat dieser nun direkten Zugriff auf die sich unterhalb des MCALs befindliche Ebene, die Kernel-Bibliothek. Grund hierfür ist,

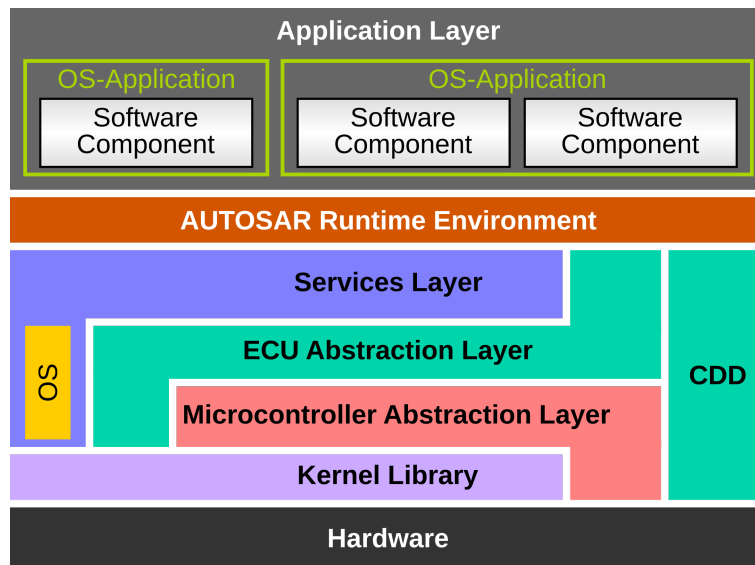


Abbildung 3.6: AUTOSAR Systemarchitektur mit Kernel-Bibliothek

dass die Kernel-Bibliothek Teilaufgaben des MCAL übernimmt und so gesehen Dienste auf derselben Ebene wie der MCAL anbietet. Die CDDs sind meist hardwareabhängig und/oder besitzen strikte Zeitanforderungen, weshalb der Einzug einer zusätzlichen Ebene kontraproduktiv wäre. Aus diesen Gründen wird der direkte Zugriff der CDDs auf die Hardwareplattform beibehalten.

Sowohl die RTE Basisdienste als auch die domänenspezifischen RTE Dienste sind Bestandteil einer jeden domänenspezifischen Systemsoftware und befinden sich bei der AUTOSAR Softwarearchitektur in der BSW bzw. in einer der drei Ebenen. Die genaue Verankerung des jeweiligen RTE Basisdienstes in der BSW wird nachfolgend erläutert.

Scheduling-Dienst: Dieser RTE Basisdienst ist eine Teilkomponente des OS Moduls und folglich im Services Layer verankert. Er wird von anderen OS Service Aufrufen verwendet, wie etwa die Beendigung des aktuell ausgeführten Tasks durch `TerminateTask()`, oder bei Ereignissen, die ein Rescheduling zur Folge haben, wie etwa der Ablauf eines Zeitgebers. Für die zeitgesteuerte Aktivierung von Tasks sieht der AUTOSAR Standard Alarme und Schedule Tables vor. Die Auswertung erfolgt periodisch beim Auslösen des Systemzeitgeber-IRQs. Hierfür wird der Systemzeitgeber mit Hilfe von Funktionen der Ausnahmebehandlung entsprechend konfiguriert. Des Weiteren werden die Dienste der

Kontextverwaltung benötigt, um den Wechsel des Prozessorkontextes im Zuge eines Task-Wechsels durchzuführen. Dabei verändert die Speicherverwaltung auch die Zugriffsrechte, je nach konfigurierter Speicherschutzstufe, indem die Register der MPU oder MMU entsprechend gesetzt werden. Durch die Verwendung der Kernel-Dienste zur Konfiguration der MPU oder MMU sowie zur Durchführung der Kontextwechsel und bedingt durch deren konstante Ausführungszeit ist eine zeitliche Abschätzung möglich.

Schutzdienst: Dieser Dienst ist im Services Layer angesiedelt und Bestandteil des Watchdog Manager Moduls. Er ermöglicht weiteren RTE Diensten des Services Layers, wie beispielsweise dem OS Modul, die Überprüfung logischer und zeitlicher Korrektheit des Programmflusses. Hierzu nutzt der Schutzdienst Funktionen des E/A-Dienstes, um den Watchdog zu konfigurieren. Dabei werden die Anfragen des E/A-Dienstes durch den ECU Abstraction Layer zum Eingabe und Ausgabe Dienst im MCAL weitergeleitet.

Kommunikationsdienst: In der AUTOSAR Softwarearchitektur ist der Kommunikationsdienst bei genauer Betrachtung kein persistenter Dienst. Vielmehr werden dessen Funktionsblöcke bei der Generierung der applikationsspezifischen AUTOSAR Runtime Environment in API Aufrufe eingebunden. Dazu zählen die zur Sender-Receiver- und Client-Server-Kommunikation gehörenden API Aufrufe. Sowohl das Sender-Receiver als auch das Client-Server-Kommunikationsparadigma unterstützen blockierende als auch nicht-blockierende API Aufrufe auf Empfangsseite. Beide Kommunikationsparadigmen erlauben bei blockierenden API Aufrufen die Festlegung von Zeitlimits. Im Falle von blockierenden Aufrufen interagiert der Kommunikationsdienst mit dem OS Modul und infolgedessen mit dem Scheduling-Dienst, um einerseits den jeweiligen Task schlafen zu legen und bei Eintritt des erwarteten Ereignisses aufwecken zu lassen und um andererseits die Zeitlimits der Tasks zu überwachen. Zur Verwaltung von intern genutzten Puffern für den Datenaustausch zwischen Sender- und Empfängerseite werden je nach Implementierung meist Verfahren zum wechselseitigen Ausschluss verwendet. Hierfür bieten die Synchronisationsmechanismen der Kernel-Bibliothek entsprechende Funktionen an. Zudem stellen diese unteilbare read-modify-write Funktionen bereit, womit sich Zeigeroperationen zur Pufferverwaltung, die atomar ausgeführt werden müssen, umsetzen lassen.

Synchronisationsdienst: Das OS Modul, welches zum Services Layer zählt, stellt entsprechend der AUTOSAR OS Spezifikation einen Spin-Lock Mechanismus zur Verfügung, der Teil des Synchronisationsdienstes ist. Der Spin-Lock Mechanismus ist für den wechselseitigen Ausschluss zwischen echt-parallelen Tasks gedacht, d. h., die auf unterschiedlichen Kernen ausgeführt werden. Ferner ist er aktiv wartend und aufgrund dessen ist keine Interaktion mit dem Scheduling-Dienst nötig. Zur Realisierung der Funktionalität verwendet der Spin-Lock Mechanismus einen der in der Kernel-Bibliothek enthaltenen Synchronisationsmechanismen. Im parMERASA Projekt handelt es sich hierbei um eine hart echtzeitfähige Ticket-Lock Implementierung. Aufgrund der Verwendung von Synchronisationsmechanismen aus der Kernel-Bibliothek für die Kommunikations- und Synchronisationsdienste können nicht faire Mechanismen ausgeschlossen werden und eine obere Schranke für zeitliche Abschätzungen kann ermittelt werden.

Eingabe- und Ausgabedienst: Aufgrund seiner Aufgabe, der Steuerung von E/A-Geräten und der damit einhergehende Nähe zur Hardwareebene, befindet sich der E/A-Dienst im MCAL (vgl. Abbildung 3.6). Da die angebotenen Funktionalitäten sehr stark von der Zielplattform und ihrer Peripherie abhängig sind, greifen die meisten Funktionen des Dienstes direkt, sprich ohne Nutzung der Kernel-Dienste, auf die Hardware zu. Im Rahmen des parMERASA Projekts zählen zum E/A-Dienst die gemeinsam von allen RTEs verwendeten Komponenten. Hierzu zählen Zeitgeber, Watchdog und CAN Treiber. Die äquivalenten Bezeichnungen der AUTOSAR Module im MCAL sind GPT, Watchdog und CAN Driver. Dies erspart eine Re-Implementierung durch Softwarewiederverwendung.

3.2.3.2 Luftfahrt

Abbildung 3.7 zeigt die ARINC 653 Systemarchitektur mit integrierter Kernel-Bibliothek. Das *ARINC 653 O/S* kommuniziert sowohl über die *Kernel-Library* als auch dem *BSP* mit der *Hardware*, da beide Teilebenen verschiedene Bereiche der Hardware abstrahieren. In der ursprünglichen ARINC 653 Systemarchitektur (vgl. Abbildung 3.2) hat das BSP auch die Funktionalitäten der Kernel-Dienste inne. Der Unterschied zwischen Kernel-Bibliothek und BSP besteht darin, dass die Kernel-Dienste die Hardwaremerkmale von Mikrocontrollern bzw. Mikroprozessoren abstrahieren, die zur Ausführung eines Betriebssystemkerns elementar sind. Sämtliche weiteren hardwarespezifischen Treiber sind im BSP enthalten.

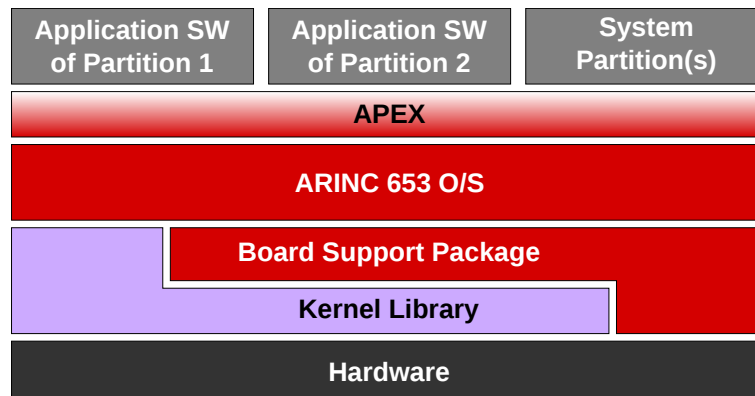


Abbildung 3.7: ARINC 653 Systemarchitektur mit Kernel-Bibliothek

Darüber hinaus verwenden Treiber des BSPs Funktionen der Kernel-Bibliothek, weshalb sich die beiden Ebenen in der erweiterten ARINC 653 Systemarchitektur überlappen. Die RTE Basisdienste sind bei der erweiterten ARINC 653 Systemarchitektur ausnahmslos Bestandteil des ARINC 653 O/Ss.

Scheduling-Dienst: Für die zeitliche Ablaufplanung der Partitionen als auch der Prozesse innerhalb einer Partition ist der Scheduling-Dienst zuständig. Ihm obliegt gleichermaßen die Planung des festen zyklischen wie auch des prioritätsbasierten Scheduling-Verfahrens zwischen Partitionen bzw. Prozessen eines Moduls. Die von der APEX API bereitgestellten Dienste in den Bereichen Partition bzw. Process Management Services greifen auf die Funktionalitäten des Scheduling-Dienstes zurück. Daneben führen partitionsinterne Ereignisse, die ein Rescheduling zur Folge haben, zum Aufruf des Schedulers. Bei der Prozessplanung werden periodische ebenso wie aperiodische Prozesse unterstützt. Zur Steuerung periodischer Prozesse und zur zyklischen Aktivierung von Partitionen wird der Systemzeitgeber über den Ausnahmebehandlungsdienst entsprechend konfiguriert. Darüber hinaus wird bei einem Prozesswechsel der Hardwarekontext mit Hilfe der Dienste der Kontextverwaltung ausgetauscht. Da das IMA Partitionierungskonzept eine vollständige Trennung der Speicherbereiche zwischen allen Partitionen fordert, werden im Zuge eines Wechsels des Hardwarekontextes auch die Speicherzugriffsrechte angepasst. Hierfür wird Gebrauch von den Funktionalitäten des Kernel-Dienstes für die Speicherverwaltung gemacht.

Schutzdienst: Er ist Teil des Health Monitors, der für die Meldung und Behandlung von Anwendungs-, Betriebssystem- und Hardwarefehlern zuständig und in ARINC 653 spezifiziert ist. Auf Prozessebene ist ein spezieller Fehlerbehandlungsprozess dafür zuständig. Neben Fehlern, die von Anwendungsprozessen selbst berichtet werden, meldet der E/A-Dienst Fehler, die durch Ausnahmen hervorgerufen werden. Dazu zählen u. a. Speicherzugriffsverletzungen oder die Division durch Null. Des Weiteren ist es möglich, dass eine Anwendung innerhalb einer Partition so fehlschlägt, dass der Fehler nicht richtig bzw. gar nicht durch den Fehlerbehandlungsprozess gemeldet wird. Eine Möglichkeit solche Anwendungsfehler zu erkennen, ist der Einsatz eines Watchdogs, dessen Funktionalität der Schutzdienst bereitstellt. Dadurch kann die logisch und zeitlich korrekte Ausführung einer Anwendung verfolgt werden.

Kommunikationsdienst: Dieser Dienst zählt zu den Inter- und Intra-Partition Communication Services innerhalb von ARINC 653 und stellt Funktionen zum Austausch von Nachrichten zwischen Partitionen und Prozessen bereit. Während die Kommunikation zwischen Partitionen sowohl zwischen Partitionen auf demselben Kern als auch zwischen Partitionen auf unterschiedlichen Kernen erfolgen kann, findet die Kommunikation zwischen Prozessen ausschließlich innerhalb einer Partition statt und somit auf ein und demselben Kern. Wie in Abschnitt 3.2.1 bereits erläutert, untergliedert sich die Kommunikation zwischen Partitionen in einen Sampling- und Queuing-Modus. Bei ersterem Modus handelt es sich bei den von der APEX API bereitgestellten Funktionen ausschließlich um nicht-blockierende Aufrufe. Zur Verwaltung von internen Puffern bei der Kommunikation zwischen unterschiedlichen Kernen greift der Kommunikationsdienst auf Synchronisationsmechanismen der Kernel-Bibliothek zurück. Des Weiteren sind beim Queuing-Modus die API Aufrufe zum Senden wie auch zum Empfangen von Nachrichten blockierend. Je nach Systemzustand kann es deshalb vorkommen, dass der aktive Prozess schlafen gelegt wird und ein Rescheduling erfolgt. In diesem Fall interagiert der Kommunikationsdienst mit dem Scheduling-Dienst.

Ein ähnliches Verhalten ergibt sich für Puffer und Blackboards bei der Prozesskommunikation. Beide Kommunikationsarten benötigen eine Interaktion mit dem Prozess-Scheduler bzw. Scheduling-Dienst. Da Puffer und Blackboards nach ARINC 653 für den Datenaustausch zwischen Prozessen innerhalb derselben Partition gedacht sind, werden keine Mechanismen zur Synchronisation zwischen unterschiedlichen Kernen benötigt. Im

Rahmen des parMERASA Projekts wurde diese Beschränkung jedoch aufgehoben, um feingranulare Parallelisierung auf Anwendungsebene zu ermöglichen. Daher greift der Kommunikationsdienst in diesem Fall ebenso auf die Synchronisationsmechanismen der Kernel-Bibliothek zurück. Weitere sich daraus ergebende Änderungen bzw. Abweichungen von der ARINC 653 Spezifikation werden im Rahmen dieser Arbeit nicht weiter thematisiert.

Synchronisationsdienst: Gleichmaßen verhält es sich bei den von ARINC 653 spezifizierten Intra-Partition Communication Services zur Prozesssynchronisation. Im Einzelnen sind dies Semaphore und Events, die ebenfalls zur Synchronisation von Prozessen auf demselben Kern vorgesehen sind. Wie auch beim Kommunikationsdienst wurde die Funktionalität bei beiden Synchronisationsarten zur Verwendung auf verschiedenen Kernen erweitert. Somit verwendet der Synchronisationsdienst Synchronisationsmechanismen der Kernel-Bibliothek. Darüber hinaus benötigt der Dienst analog zum Kommunikationsdienst ebenfalls den Scheduling-Dienst, da Semaphore und Events je nach Systemzustand ein Rescheduling anstoßen.

Eingabe- und Ausgabedienst: Die Funktionalitäten dieses RTE Basisdienstes beschränken sich auf eine einheitliche Schnittstelle zur Konfiguration von Zeitgebern und Watchdogs. Wie bereits bei der Fallstudie zur Automobilbranche erwähnt, sind die meisten Funktionalitäten im Bereich der Peripherie stark von der zugrunde liegenden Zielplattform abhängig. Daher erfolgt die Ansteuerung weiterer Peripheriegeräte über die Kernel-Dienste, zumeist jedoch direkt über das BSP. Komplexe Treiber-Stacks, wie z. B. Netzwerkprotokolle für Real-time Ethernet, werden zudem oft in Systempartitionen realisiert und sind nicht Bestandteil des ARINC 653 O/Ss, um das O/S möglichst generisch zu halten und die Portierbarkeit zu gewährleisten. Das Fehlermanagement von komplexen Treiber-Stacks hingegen fällt in das Ressort des ARINC 653 O/Ss und somit besteht auch eine Wechselbeziehung mit dem Schutzdienst.

3.2.3.3 Baumaschinenbranche

In Abbildung 3.8 ist die um die Kernel-Bibliothek erweiterte ESX-3XL-BIOS Systemarchitektur zu sehen. Analog zur ARINC 653 Systemarchitektur (vgl. Abbildung 3.7)

überlappen sich die *Kernel-Bibliothek* und das *ESX-3XL-BIOS*, da beide Teilebenen verschiedene Bereiche der *Hardware* abstrahieren. Im Gegensatz zum BSP bei der ARINC 653 Systemarchitektur, welches zur ausschließlichen Abstraktion der Hardwareplattform dient, besitzt das *ESX-3XL-BIOS* einen größeren Funktionsumfang. Dieser umfasst neben einer Hardwareabstraktionsschicht und Systemtreibern für diverse Peripherie auch einen minimalen Betriebssystemkern. Aus diesem Grund sind die RTE Basisdienste in der BIOS Ebene verankert. Die *Middleware* greift über das *ESX-3XL-BIOS* auf hardwarenahe Funktionen zu. Dabei benötigt das *ESX-3XL-BIOS* teilweise Dienste der Kernel-Bibliothek. Lediglich die Kernel-Dienste zur Synchronisation werden von der *Middleware* direkt über die Kernel-Bibliothek angesprochen.

Scheduling-Dienst: Er ist für die zeitliche Ablaufplanung der Tasks verantwortlich. In erster Instanz geschieht die Planung nach einem prioritätsbasierten, präemptiven Scheduling-Verfahren. Tasks gleicher Priorität können in Gruppen zusammengefasst werden, die nach einem Rundlauf-Verfahren geplant werden. Dabei verwenden die Funktionen zur Task-Verwaltung der *ESX-3XL-BIOS* API den Scheduling-Dienst, während das Rundlauf-Scheduling-Verfahren Bestandteil der *Middleware* ist und auf die Task-Funktionen des BIOS aufsetzt. Da die erweiterte *ESX-3XL-BIOS* Softwarearchitektur darüber hinaus Multicore-Prozessoren unterstützt, läuft auf jedem Prozessorkern ein separater, unabhängiger Scheduler. Ferner werden auch Tasks unterstützt, die periodisch wiederkehrend und/oder zeitgesteuert nach dem Systemstart plus einem bestimmten Offset aktiviert werden. Hierfür wird der Systemzeitgeber entsprechend über den RTE Basisdienst zur Ausnahmebehandlung konfiguriert. Daneben benötigt der Scheduling-Dienst den Speicherverwaltungsdienst, um bei einem Task-Wechsel den Prozessorkontext auszutauschen. Das *ESX-3XL-BIOS* erlaubt außerdem die Überwachung von Stacks beliebiger Tasks. Um einen Über- bzw. Unterlauf von Task-Stacks zu verhindern, können im Zuge eines Task-Wechsels die entsprechenden Zugriffsrechte für den Stack-Speicherbereich des als nächstes auszuführenden Tasks gesetzt werden.

Schutzdienst: Neben dem Stack-Monitoring bietet das *ESX-3XL-BIOS* auch eine Überwachung des Programmflusses durch einen Watchdog Zeitgeber. Dieser wird mit Hilfe des E/A-Dienstes konfiguriert. Als Reaktion auf eine Fehlererkennung im Programmfluss wird ein Reset oder die Abschaltung des Systems angeboten.

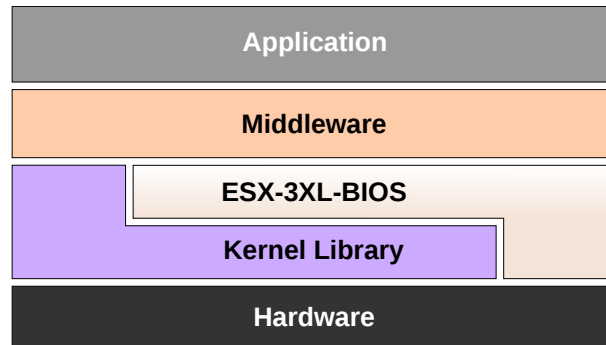


Abbildung 3.8: ESX-3XL-BIOS Systemarchitektur mit Kernel-Bibliothek

Kommunikationsdienst: Bei der ursprünglichen ESX-3XL-BIOS Laufzeitumgebung, die alleinig Singlecore-Plattformen unterstützt, findet der Datenaustausch über globale Variablen statt. Derselbe Mechanismus wird auch zur Kommunikation zwischen Kernen auf Multicore-Plattformen verwendet. Allerdings wird hierbei zur Wahrung der Konsistenz beim Datenaustausch der Ticket-Lock Synchronisationsmechanismus der Kernel-Bibliothek benutzt.

Synchronisationsdienst: Dieser stellt Funktionen zur Sicherung von kritischen Sektionen, zur Verwaltung von Events und Lock-Mechanismen zur Verfügung. Dazu nutzt der Dienst Synchronisationsmechanismen der Kernel-Dienste, im parMERASA Projekt im Speziellen die Ticket-Lock Implementierung der Kernel-Bibliothek. Des Weiteren bietet der Synchronisationsdienst die Nutzung von Events an. Gegenüber der ursprünglichen Implementierung im ESX-3XL-BIOS wurde die Funktionalität um eine Multicore-Prozessor Unterstützung erweitert. Dadurch können Events auch zwischen verschiedenen Kernen signalisiert werden.

Eingabe- und Ausgabedienst: In Analogie zu AUTOSAR stellt dieser Funktionen für Zeitgeber, Watchdog und CAN Treiber zu Verfügung. Darüber hinausgehend verwendete Peripheriegeräte werden ebenfalls durch Dienste des ESX-3XL-BIOS abstrahiert. Diese zählen allerdings nicht zu den RTE Basisdiensten, sondern zu den domänenspezifischen RTE Diensten. Ungeachtet dessen verwendet die überwiegende Anzahl der E/A-Gerätetreiber die grundlegenden Interrupt-Behandlungsroutinen des Kernel-Dienstes zur Ausnahmebehandlung. Ebenso wird der Hardware-Watchdog darüber konfiguriert.

3.2.4 Angrenzende Arbeiten

Der Kernel-Bibliothek Ansatz greift einen kleinen Teil des Mikrokern-Konzepts auf. Eine bekannte Implementierung dieses Konzepts ist der L4-Mikrokern (Liedtke 1993; Liedtke 1995). Ein Mikrokern beinhaltet nur diejenigen Komponenten, die zur Implementierung eines Betriebssystems Bestandteil des Kerns sein müssen. Sämtliche Betriebssystemfunktionen, die bei einer Implementierung außerhalb des Kerns die geforderte Funktionalität an ein System nicht erfüllen, sind Bestandteil des Mikrokerns. Hierzu zählen *Address Space Management* 'Adressbereichsverwaltung', *Thread Management* 'Prozessverwaltung', und *IPC* 'Interprozesskommunikation'. Alle darüber hinausgehenden Funktionalitäten werden im Benutzer-Modus ausgeführt, entweder als eigenständige Prozesse, sogenannte Server, mit denen die Anwendungen kommunizieren, oder eingebunden als Bibliothek in die jeweilige Anwendung. Die Adressbereichsverwaltung weist jeder virtuellen Seite eine physikalische Seite zu. Die Adressumsetzung geschieht dabei hardwaregestützt in Form von TLBs und Seitentabellen. Da grundsätzlich alle Programme außerhalb des Kerns 'böse' sein könnten, muss die Adressbereichsverwaltung Bestandteil des vertrauenswürdigen Kerns sein, da ansonsten jede Anwendung den Speicher einer beliebigen anderen Anwendung manipulieren könnte. Die eigentliche Speicherverwaltung wiederum obliegt einem Anwenderprogramm, das initial über den gesamten Adressraum verfügt und auf Anfrage Speicherbereiche zuweist. Jeder Prozess besitzt einen Prozesskontext, zu dem zumindest ein Stack- und Befehlszeiger zählen und ein Speicherkontext, der besagt, auf welchen Adressbereichen dieser arbeitet. Eine willkürliche Änderung der Adressbereiche könnte eine Manipulation an Speicherinhalten beliebiger Prozesse nicht ausschließen und dadurch die Isolation und den Schutz von Anwendungen nicht garantieren. Aus diesem Grund ist die Prozessverwaltung Bestandteil des Kerns. Ähnlich verhält es sich mit der IPC, die eine Kommunikation per Nachrichten zwischen Adressbereichen ermöglicht. Einige Operationen der Adressbereichsverwaltung benötigen selbst den IPC Mechanismus und somit gehört dieser zwangsläufig zum Kern. Hardwareunterbrechungen werden auch durch IPC Nachrichten signalisiert. Dabei wird bei einem IRQ in der zugehörigen ISR eine Nachricht zu dem entsprechend bearbeitenden Prozess gesendet. Dadurch ist die eigentliche ISR relativ kurz und der zuständige Prozess kann unter die Kontrolle vom Betriebssystem gestellt werden. Der Kernel-Bibliothek Ansatz geht insofern einen ähnlichen Weg, dass in der ISR nur die Register zum Zurücksetzen des IRQs geschrieben werden und der dazugehörige Task aktiviert wird. Allerdings wird der Task nicht zwingend

im Benutzer-Modus ausgeführt. Dies hängt von den jeweiligen Systemanforderungen ab. Somit ist ein Zugriff auf alle benötigten Register auch ohne einen Systemaufruf möglich. Dies bedeutet eine geringere Latenz beim Zugriff und ist beispielsweise bei den AUTOSAR CDDs nötig. Ansonsten liegt der Fokus beim Mikrokern in der Absicherung des Kerns. Liedtke (1995) beschreibt Techniken zur Performancesteigerung von Mikrokerns und konstatiert zudem, dass dies nicht vereinbar ist mit einer hohen Portabilität des Mikrokerns selbst. Das steht im Widerspruch zum Kernel-Bibliothek Ansatz, der u. a. eine möglichst hohe Portabilität zum Ziel hat. Der größte Unterschied besteht darin, dass der L4 Mikrokern einen vollständigen, wenn auch kleinen Kern realisiert, wohingegen die Kernel-Bibliothek eine Art Hardware Abstraction Layer (HAL) darstellt.

Einen weiteren Ansatz in der Betriebssystementwicklung präsentiert das *Library OS*. Dabei ist die Schnittstelle zwischen Kern und dem darauf aufsetzenden Betriebssystem und den Anwendungen noch näher in Richtung Hardware verschoben, als dies beim Mikrokern der Fall ist. Das Library OS Modell kann somit als konsequente Fortführung des Mikrokern Ansatzes verstanden werden. Dabei werden Betriebssystemaufgaben zusammen mit einer Anwendung im Benutzer-Modus ausgeführt und dieser in Form einer Bibliothek zur Verfügung gestellt. Im Zusammenhang mit Library OS werden in der Literatur oft die Arbeiten zum Cache Kern (Cheriton et al. 1994) und Exokern (Engler et al. 1995) zitiert, die beide den Ansatz aufgreifen, das Betriebssystem als Bibliothek zusammen mit einer Anwendung zu binden. Einer der oft genannten Vorteile dabei ist eine bessere Performance gegenüber anderen Kerns, da jede Anwendung direkte Kontrolle über die verfügbaren Hardwareressourcen hat und die Betriebssystembibliothek individuell auf die jeweilige Anwendung angepasst werden kann. Beispielsweise kann eine Anwendung ein speziell auf ihre Anforderungen zugeschnittenes Dateisystem realisieren, ohne auf ein darunterliegendes Kern-Dateisystem aufbauen zu müssen. Dies wird dadurch ermöglicht, dass die einzige Aufgabe des Kerns in der Absicherung und der Mehrfachausnutzung von Hardwareressourcen besteht. Alle darüber hinausgehenden Aufgaben obliegen dem Library OS bzw. der Anwendung. Die Maxime beim Exokern lautet daher, die Absicherung von der Verwaltung der Ressourcen zu trennen (Engler et al. 1995). Wie auch beim Mikrokern ist die Portabilität hier umso mehr eingeschränkt, da die Betriebssystembibliothek direkten Zugriff auf die jeweilige Hardwareplattform besitzt. Auch wenn die eigentliche Anwendung durch eine API von der Bibliothek gekapselt

ist, so muss die Bibliothek selbst an das entsprechende Zielsystem angepasst werden. Des Weiteren liegt das Hauptaugenmerk auf der Separation verschiedener Anwendungen voneinander und nicht auf der parallelen Ausführung einer Anwendung. Aufgrund des Aufbaus kann der Library OS Ansatz auch als vertikal strukturiertes Betriebssystem angesehen werden. Im Gegensatz zu klassischen Betriebssystemen bietet das Library OS keine durchgängige Abstraktionsschicht zwischen Anwendungen und Hardware an. Vielmehr sind die Betriebssystemaufgaben in zwei Bereiche unterteilt, dem Kernel und der Bibliothek. Darüber hinaus besteht auch die Möglichkeit, dass die Anwendungen direkt mit dem Kernel kommunizieren.

Die Kernel-Bibliothek hingegen bildet zwar jeweils zusammen mit dem MCAL, BSP, bzw. ESX-3XL-BIOS eine horizontale Abstraktionsebene, unterteilt diese allerdings ebenso vertikal in zwei Teilbereiche, wobei der domänenspezifische Teil in gewissem Umfang direkten Zugriff auf die Hardwareplattform besitzt. Insgesamt lässt sich eine gewisse Parallelität zum Library OS erkennen. Die RTE Basisdienste sind an das jeweilige domänenspezifische Betriebssystem angepasst, ähnlich der Anwendungsbibliothek beim Library OS. Allerdings geben die Basisdienste bestimmte Dienste vor, die in allen drei Softwarearchitekturen zur Anwendung kommen, um Synergieeffekte zu nutzen. Die Kernel-Dienste ermöglichen zusammen mit der jeweils domänenspezifischen Abstraktionsschicht eine vereinfachte Portabilität des Betriebssystems und der Anwendung auf eine andere Plattform. Jedoch bilden sie keinen eigenständigen, in sich geschlossenen und abgesicherten Kernel, sondern mit den anderen Betriebssystemkomponenten einen monolithischen Kernel, der an die jeweils ursprüngliche Softwarearchitektur angelehnt ist. Hier bietet das Library OS eine bessere Separation bzw. Absicherung aufgrund der kompakteren Kernel-Basis.

Neben den genannten Arbeiten zur Optimierung von Kernels gibt es Ansätze mit Fokus auf Multi-/Manycore-Systeme. Corey (Boyd-Wickizer et al. 2008) ist ein Betriebssystem für Manycore-Prozessoren mit der Devise, dass Anwendungen selbst die Kontrolle darüber haben sollten, welche Daten geteilt werden, um auf Manycore-Architekturen zu skalieren. Per se sollte auf alle Datenstrukturen nur von jeweils einem Kern zugegriffen werden, außer die Anwendung verlangt explizit, dass es sich um geteilte Daten handelt. Hierfür führt Corey drei Betriebssystemabstraktionen ein, im Einzelnen *Address Ranges*, *Kernel Cores* und *Shares*, um Anwendungen direkte Kontrolle über geteilte Kernel-Ressourcen zwischen Kernen zu geben. *Address Ranges* erlauben der Anwendung die Kontrolle darüber, welche Adressbereiche privat respektive geteilt sind. Dadurch können Seitenfehler aufgrund von

geteilten Seiten und TLB Shootdowns minimiert werden. Der Kernel-Bibliothek Ansatz hingegen zielt auf eingebettete Echtzeitsysteme ab, die eine einfache Speicherhierarchie besitzen. Es wird davon ausgegangen, dass dem komplett verwendeten, virtuellen Speicher auch gleich viel physikalischer Speicher zur Verfügung steht und somit echte Seitenfehler nicht auftreten können. Kernel Cores ermöglichen Anwendungen, Kernel-Funktionen explizit als Server auf bestimmten Kernen auszuführen, um Zugriffskonflikte auf die dafür benötigten, geteilten Daten zu verringern. Beispielsweise kann ein bestimmter Kern für die Interaktion mit dem CAN Gerät verantwortlich sein und anstatt dass alle Kerne Daten des Treibers manipulieren, kommunizieren diese mit dem relevanten Server-Kern. Bei Shares handelt es sich um Nachschlagetabellen für Kernel-Objekte, welche privat oder geteilt sind. Statt das Betriebssystem entscheiden zu lassen, welche Objekte über welche Kerne geteilt sind, übernimmt diese Aufgabe die Anwendung selbst. Dadurch können Shares gezielter durch die jeweilige Anwendung erstellt werden und die gemeinsame Nutzung von Objekten, die nicht geteilt werden müssen, unterbunden werden. Da die Kernel-Bibliothek lediglich Dienste zur Verfügung stellt, die von den jeweiligen RTEs zur Realisierung des Betriebssystemkerns verwendet werden, kann das Konzept der Kernel Cores bzw. Shares in den Kernel der entsprechenden RTE einfließen. Vor allem die Auslagerung von Gerätetreibern auf explizite Kerne kann eine bessere zeitliche Abschätzung ermöglichen.

Einen im weitesten Sinne ähnlichen Ansatz geht Baumann et al. (Baumann et al. 2009) mit einer neuen Betriebssystemstruktur, namens *Multikernel*. Dabei wird der Prozessor als ein Netzwerk bestehend aus einzelnen Kernen angesehen, vergleichbar mit einem verteilten System. Die Interprozesskommunikation wird hierbei per Nachrichtenübermittlung realisiert und somit von der zugrunde liegenden Hardwarearchitektur entkoppelt. Ferner wird davon ausgegangen, dass kein gemeinsamer Speicher für das Betriebssystem existiert, mit Ausnahme von Speicherbereichen zum Austausch von Nachrichten zwischen verschiedenen Kernen. Der Multikernel hat drei Designprinzipien zum Ziel: sämtliche Interprozesskommunikation explizit erfolgen zu lassen, das Betriebssystem von der Hardware zu entkoppeln und Daten zu replizieren anstatt zu teilen. Durch den zielgerichteten Informationsaustausch per Nachrichten sind nur noch die jeweils relevanten Kerne involviert, ähnlich dem Konzept bei Corey, wo ebenfalls die (Kernel-) Daten nur zwischen den erforderlichen Kernen geteilt sind. Bei beiden Ansätzen müssen dadurch weniger Cache-Zeilen kohärent gehalten werden, das wiederum die Belastung des Interconnects senkt.

Da beim Multikernel keinerlei gemeinsame Daten vorliegen, müssen Kernel-Objekte redundant auf allen involvierten Kernen existent sein. Bei Corey dagegen werden Objekte, auf die gemeinsam zugegriffen wird, auf ein Minimum reduziert, allerdings nicht komplett ausgeschlossen. Ein weiterer Punkt ist die Entkopplung des Betriebssystems von der Hardwarearchitektur. Die einzigen Bestandteile, die in Abhängigkeit zur Hardware implementiert werden müssen, sind der Nachrichtentransportmechanismus und die Schnittstelle zur CPU und Peripherie. Barrelfish ist eine Realisierung des Multikernel Konzepts. Hierbei sind *CPU Drivers* ausschließlich für einige wenige, kernlokale Aufgaben zuständig und werden im privilegierten Kernel-Modus ausgeführt, angelehnt an das Exokernel Modell, während *Monitore* im Benutzer-Modus alle weiteren Betriebssystemaufgaben koordinieren, u. a. die Kommunikation zwischen Kernen.

Wie im vorherigen Abschnitt bereits erwähnt, stellt die Kernel-Bibliothek verschiedene Dienste zur Verfügung, jedoch keinen eigenständigen Kernel. Somit lässt sich eine Betriebssystemstruktur nach dem Multikernel Ansatz mit Hilfe der Kernel-Dienste realisieren. Am Beispiel von Barrelfish wären die Kernel-Dienste Teil der CPU Drivers und die RTE Basisdienste Bestandteil der Monitore. Die konkrete Integration des Multikernel Konzepts in die jeweilige RTE ist davon abhängig, ob sich dies mit der domänenspezifischen Softwarearchitektur vereinen lässt. Die AUTOSAR Systemarchitektur etwa spezifiziert auf Anwendungsebene und nicht wie beim Multikernel auf Betriebssystemebene einen vergleichbaren Ansatz. In einem KFZ beispielsweise stellen sämtliche miteinander verbundenen ECUs ein Netzwerk von Recheneinheiten dar, die über unterschiedliche Bussysteme gekoppelt sind. Die AUTOSAR RTE stellt den Anwendungen über definierte Schnittstellen einen Fundus an Diensten zur Verfügung, unabhängig davon, von welchen ECUs diese bereitgestellt werden. Meist handelt es sich dabei um Dienste zur Interaktion mit Peripheriegeräten. Per Nachrichtenkommunikation delegiert die AUTOSAR RTE die Anfragen zu den entsprechenden ECUs bzw. Serverprozessen. Dieses Verfahren ist vergleichbar mit den Kernel Cores bei Corey. Klassische Betriebssystemaufgaben werden hingegen weiterhin pro ECU gekapselt. Eine gemeinsame Sicht auf den Adressraum über alle ECUs ist nicht möglich. In Analogie zu den Multikernel Designprinzipien abstrahiert AUTOSAR die Anwendungen von einem Netzwerk aus verteilten ECUs und sämtliche Kommunikation zwischen ECUs erfolgt ausschließlich durch Nachrichten.

Eine gänzlich andere Zielsetzung verfolgt der *Separation Kernel*, dessen Konzept von Rushby (1981) eingeführt wurde. Der Separation Kernel zielt darauf ab, die Konstruktion

eines rundum sicheren Computersystems zu ermöglichen und dieses so zu realisieren, dass es möglichst einfach verifiziert werden kann. Ausgangspunkt seiner Überlegungen war die Tatsache, dass in einem verteilten System nicht eine einzelne Instanz, beispielsweise ein Kernel, für die Sicherheit zuständig ist, sondern die einzelnen Funktionen auf physikalisch getrennten Subsystemen verteilt sind, die lediglich über eingeschränkte Kanäle kommunizieren. Somit ist es Aufgabe der einzelnen Subsysteme, die Absicherung für ihre jeweiligen kritischen Funktion nach einer systemweiten Sicherheitsrichtlinie zu garantieren. Übertragen auf den Separation Kernel bedeutet dies, dass der Kernel durch Virtualisierung von geteilten Ressourcen – in Form von Partitionen – eines einzelnen Systems eine Umgebung bereitstellt, die nicht von einem echten verteilten System unterscheidbar ist. Dies wird durch eine physikalische Separation aller Partitionen und der Bereitstellung von expliziten Kommunikationskanälen zwischen diesen erreicht. Der Unterschied zu einem Hypervisor besteht darin, dass ein Separation Kernel in erster Linie für die Bereitstellung von Kommunikationskanälen zwischen Partitionen verantwortlich ist, wohingegen ein Hypervisor primär virtuelle Maschinen zur Verfügung stellt. Des Weiteren ist der Kernel die einzige Komponente, die im privilegierten Modus ausgeführt wird. Zudem ist dem Kernel nichts über die Sicherheitsrichtlinie des Systems bekannt. Diese Aufgabe obliegt den kritischen Funktionen, die innerhalb von Partitionen mit beschränkten Rechten ausgeführt werden. Dadurch ist der Separation Kernel komplett vom Zweck und der Ausrichtung des Systems entkoppelt. Durch die Auslagerung der kritischen Funktionen außerhalb des Kernels, ist dessen Codebasis relativ kompakt, wodurch dieser einfacher zu verifizieren ist. Im Vergleich zu den anderen vorgestellten Kernel Konzepten geht es beim Separation Kernel weniger um technische Detaillösungen als um den Entwurf und die Verifikation von sicheren Systemen. Ungeachtet dessen haben der L4 Kernel und der Separation Kernel beide zum Ziel, möglichst wenig Funktionen im Kernel zu integrieren, die im privilegierten Modus ausgeführt werden und somit ein potenzielles Sicherheitsrisiko darstellen. Der Grund für die Erwähnung des Separation Kernels im Zusammenhang mit der Kernel-Bibliothek besteht darin, dass der ARINC 653 Standard diese Prinzipien als Basis für das IMA Partitionierungskonzept verwendet (Masmano et al. 2010) und im Rahmen des parMERASA Projekts die Kernel-Bibliothek Bestandteil der erweiterten ARINC Softwarearchitektur ist (vgl. Abbildung 3.7). Wie in Abschnitt 3.2.3 bereits gezeigt, ist der ARINC 653 Standard und somit das Konzept des Separation Kernels mit dem Kernel-Bibliothek Ansatz vereinbar.

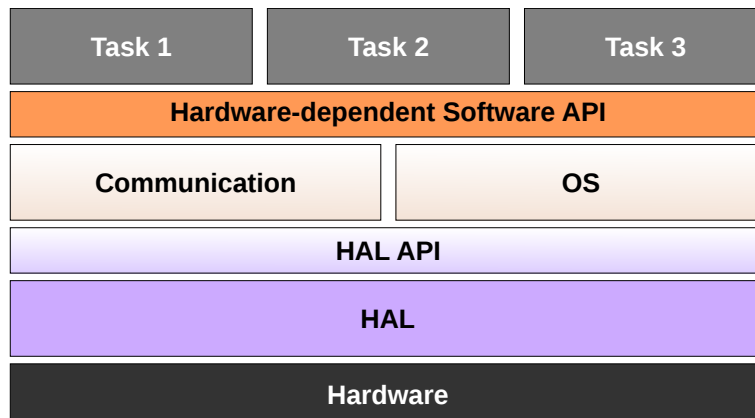


Abbildung 3.9: Systemarchitektur nach Popovici (2009)

Popovici und Jerraya (2009) präsentieren eine Abstraktion der Hardwareplattform von der Software durch eine mehrschichtige Organisation des Software-Stacks, forthin als Hardware Abstraction Layer (HAL) Ansatz bezeichnet. Bei den betrachteten Zielplattformen handelt es sich um heterogene *Multiprocessor System-on-Chips (MPSoCs)*, die verschiedene Prozessoren mit spezifischen Funktionen für die jeweils anvisierte Domäne auf einem Ein-Chip-System vereinen. Das Hauptaugenmerk liegt dabei auf der Wiederverwendbarkeit von Software über unterschiedliche MPSoC Plattformen hinweg. Hierfür wird der Software-Stack in mehrere Ebenen gegliedert, wobei die unterste Ebene der HAL bildet (vgl. Abbildung 3.9), der für eine möglichst reibungslose Portierung verantwortlich ist. Beim HAL Ansatz wird davon ausgegangen, dass sich Software generell in eine Anwendungsebene, in Abbildung 3.9 durch *Task 1 – 3* repräsentiert, und eine hardwareabhängige Softwareschicht unterteilen lässt, die die *Hardware-dependent Software API* bis einschließlich dem *HAL* umfasst. Um eine Softwareportierung zu erleichtern, ist die hardwareabhängige Schicht in drei Komponenten untergliedert: *OS*, *Communication* und *HAL*. Die Hardware-dependent Software API entkoppelt die Anwendung von der hardwareabhängigen Softwareschicht. Das Echtzeitbetriebssystem übernimmt dabei charakteristische Betriebssystemaufgaben, während die Kommunikationskomponente Dienste zur Interaktion von Tasks auf unterschiedlichen Prozessorkernen bereitstellt. Die HAL API stellt eine definierte Schnittstelle zum HAL zur Verfügung. Sie ist meist für ein bestimmtes Betriebssystem spezifiziert und bietet den Vorteil, dass somit durch Austausch des HALs eine einfache Portierung des Betriebssystems an verschiedene ISAs

möglich ist, vergleichbar mit dem BSP bei ARINC. Sowohl der HAL als auch der Kernel-Library Ansatz abstrahieren von der zugrunde liegenden Hardware, um eine von der Hardware unabhängige Implementierung zu ermöglichen. Bei Ersterem liegt der Fokus auf heterogenen MPSoC Plattformen, wohingegen die Kernel-Library die Unterstützung zur Ausführung von parallelisierten Anwendungen auf homogenen Multi-/Manycore- Architekturen zum Ziel hat. Trotzdem weisen beide Ansätze eine deutliche Übereinstimmung auf. Die Hardware-dependent Software API entspricht der domänenspezifischen Schnittstelle aus Abbildung 3.5 und die Kommunikations- und Betriebssystemkomponente der domänenspezifischen RTE. Genauer betrachtet, kommt die Kommunikationskomponente dem Kommunikations- und Synchronisationsdienst der Kernel-Library gleich. Der HAL und die HAL API sind funktional vergleichbar mit der Kernel-Bibliothek und deren API, die allerdings nicht explizit dargestellt ist. Im Unterschied zum HAL Ansatz hat die Kernel-Library zum Ziel, Gemeinsamkeiten der domänenspezifischen RTEs in Form von RTE Basisdiensten zu bündeln und für sämtliche zur Verfügung gestellten Dienste eine zeitliche obere Schranke abzuschätzen. Letzteres wird erreicht, indem bei der Umsetzung der RTE Basisdienste bestimmte Kriterien eingehalten werden. Beim HAL Ansatz bildet die HAL API eine durchgängige, horizontale Schnittstelle, die jedoch in den meisten Fällen für genau ein Betriebssystem zugeschnitten ist. Dagegen abstrahiert die Kernel-Bibliothek nur gemeinsam verwendete Teile der Hardware. Zugriff auf domänen- oder plattformspezifische Hardware erfolgt durch die jeweilige RTE. Dadurch kann die Kernel-Bibliothek universell eingesetzt werden.

3.3 ACMC-OS

Die Idee hinter ACMC-OS ist, eine Evaluierungsplattform für parallelisierte und parallel ausführbare Anwendungen zu schaffen, indem die AUTOSAR Softwarearchitektur auf ein minimales Set an Modulen und Funktionalitäten reduziert wird. Um diese Ziele zu erreichen, beinhaltet ACMC-OS nur grundlegende Funktionalitäten, die zur Ausführung von Anwendungssoftware benötigt werden. Ausgeschlossen hiervon sind zudem sämtliche Module zur Ansteuerung von Peripheriegeräten. Falls ein entsprechendes E/A-Modul unabdingbar ist, so kann dieses aus externen Programmquellen eingebunden oder durch Emulation ersetzt werden. Des Weiteren stellt ACMC-OS erweiterte Kommunikations-

und Synchronisationsfunktionen bereit, um eine effiziente Ausführung paralleler Anwendungen zu ermöglichen. Die Grundlage für ACMC-OS bildet die in Abbildung 3.5 gezeigte Softwarearchitektur. Dabei greift die Umsetzung auf die RTE Basisdienste und Kernel-Dienste zurück.

Für jede Anwendung definiert eine Reihe von Konfigurationsdateien, welche Module benötigt werden und die entsprechenden Modulkonfigurationsoptionen. Mittels dieser Konfiguration wird der AUTOSAR Software-Stack anwendungsspezifisch angepasst. In einem BSW-Modul können zwei Arten von Funktionen vorkommen:

1. Normale Funktionen mit gleich bleibenden Namen und Implementierung.
2. Funktionen mit parametrierbaren Namen und unterschiedlichen Implementierungen je Funktionsname.

Die Standardfunktionen besitzen einen definierten Funktionsprototypen und innerhalb einer ACMC-OS Version eine gleichbleibende Implementierung. Die zweite Art von Funktionen wird vom RTE Generator anhand der Anwendungskonfiguration erstellt. Beispielsweise führt die RTE API Funktion `Rte_Read_<p>_... (...)` einen expliziten Lesevorgang bei einer Sender-Receiver-Kommunikation mit Data-Semantik aus, wobei `<p>` den jeweiligen Port der Sender-Receiver-Schnittstelle kennzeichnet. Die konkrete Implementierung der Funktion und deren Komplexität hängt u. a. davon ab, ob die Datenübertragung zwischen geschützten Speicherbereichen stattfindet oder nicht.

Die Kernaufgaben im ACMC-OS übernimmt das OS-Modul, das Teile der in der AUTOSAR OS und OSEK Spezifikation (AUTOSAR 2015b; OSEK/VDX 2005) definierten Funktionalitäten zur Verfügung stellt. Es ist für die Bereitstellung und Verwaltung der Ausführungskontexte von Anwendungen und Systemdiensten verantwortlich, die aus Tasks und ISRs bestehen. Das OS-Modul kann grob in die folgenden Systemdienstgruppen untergliedert werden: Das *Task-Management* ist für die Ablaufplanung und die Task-/ISR-Ausführung verantwortlich. Das *Ressourcenmanagement* ist für die Zugriffssteuerung auf gemeinsam genutzte Ressourcen zuständig. *Events* kommunizieren binäre Informationen zwischen Tasks. *Zähler*, *Alarmer* und *Zeitplantabellen* unterstützen ereignis- und zeitgesteuerte Aktionen. *Schutzdomänen* ermöglichen Speicherschutz, Überwachung von Zeitlimits und Einhaltung von Zugriffsrechten bei Systemdiensten. Für die Kommunikation zwischen OS-Applications und somit auch zwischen Kernen

eines Multicore-Prozessors wird der Inter-OS-Application Communicator (IOC) benötigt. Dessen Funktionen werden zumeist von der AUTOSAR RTE aufgerufen, die für die Kommunikationsabstraktion verantwortlich ist. Alle weiteren OS-Dienste können in die allgemeine *Betriebssystemausführungssteuerung* zusammengefasst werden. Einige dieser OS-Dienste verwenden u. a. die Dienste der Kernel-Bibliothek (siehe Abschnitt 3.2.2).

Eine weitere zentrale Rolle kommt der AUTOSAR RTE zu, die als Mittler zwischen der BSW und der Anwendung dient. Die RTE abstrahiert von der Kommunikation, indem sie der Anwendungsebene die gleichen Schnittstellen und Dienste zur Verfügung stellt, egal ob der Datenaustausch zwischen Modulen, die auf demselben Kern oder verschiedenen Kernen bei einem Multicore-Prozessor, oder zwischen unterschiedlichen ECUs stattfindet. Hier beschränkt sich APMC-OS auf die Kommunikation innerhalb einer Single- oder Multicore ECU. Die konkrete Implementierung der AUTOSAR RTE pro ECU unterscheidet sich somit je nach Anforderung der Anwendung und den Kommunikationssystemen der jeweiligen Hardwareplattform. Die AUTOSAR RTE wird mit Hilfe von Konfigurationsdateien ECU-spezifisch generiert. Neben den in (AUTOSAR 2015b; AUTOSAR 2015c) spezifizierten Kommunikationsparadigmen unterstützt APMC-OS bei der Generierung der RTE weitere Kommunikations- und Ausführungsmodelle, um die Auswirkungen auf parallelisierte Anwendungen evaluieren zu können.

Um für eine parallelisierte Anwendung, die OSEK-konform konzipiert wurde, mittels APMC-OS eine minimale Ausführungsplattform bereitzustellen, genügen das OS-Modul und der RTE Generator. Im Rahmen des parMERASA Projekts wurde bei der Bereitstellung einer Ausführungsplattform für ein Motorsteuerungssystem so verfahren. Dabei wurden die benötigten Funktionalitäten des OS-Moduls in APMC-OS umgesetzt und der RTE Generator entsprechend ausgelegt. Um auch AUTOSAR-konforme Anwendungen zu unterstützen, müsste APMC-OS noch um weitere Module ergänzt werden, die im Folgenden beschrieben werden.

Neben den parametrierbaren Funktionen der RTE gibt es noch Standardfunktionen, die zum Starten und Stoppen der RTE benötigt werden. Diese Funktionen werden ausschließlich von anderen BSW-Modulen aufgerufen. Das *Micro Controller Unit (MCU)* Modul initialisiert MCU-spezifische Hardware und liefert Statusinformationen zurück. Es ist Teil des MCAL und abstrahiert neben der Kernel-Bibliothek von der Hardwareplattform.

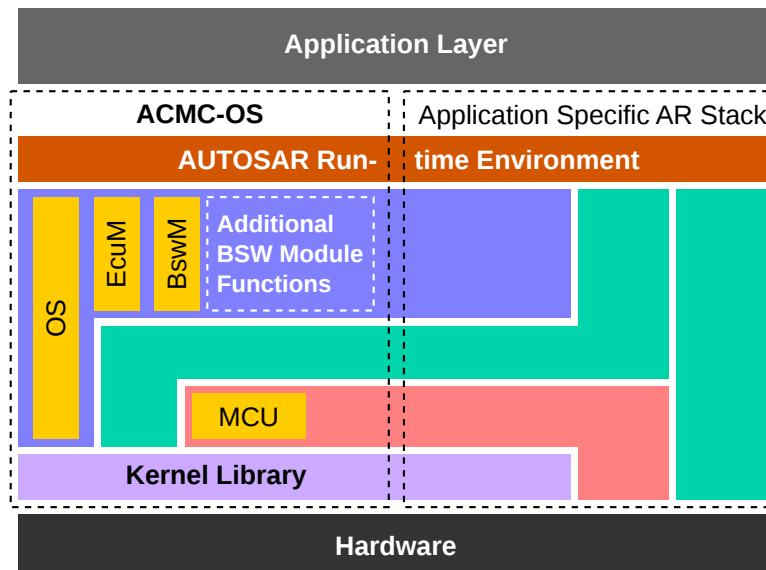


Abbildung 3.10: ACMC-OS Systemarchitektur

Während die Kernel-Dienste den Zugriff auf allgemeine Funktionseinheiten eines Mikrocontrollers unterstützen, ermöglicht das MCU Modul auch die Ansteuerung spezieller Merkmale eines Mikrocontrollers. Da die von der Kernel-Bibliothek bereitgestellten Funktionalitäten den Anforderungen des Motorsteuerungssystems genügen, wurde das MCU Modul weggelassen. Der *ECU State Manager (EcuM)* initialisiert und de-initialisiert u. a. das Betriebssystem, den Ablaufplaner und den BSW Mode Manager neben weiteren BSW Modulen. Eine Aufgabe ist die Steuerung des in mehreren Zyklen unterteilten Startvorgangs und das Herunterfahren der ECU. Für OSEK-konforme Anwendungen ist der Startprozess des OS-Moduls ausreichend. Gemäß der Spezifikation ist der ECU State Manager auch für die Schlafmodi verantwortlich. Da dies keine notwendige Eigenschaft für die Ausführung von Anwendungen ist, ist diese Funktionalität nicht Bestandteil von ACMC-OS. Für Anwendungen, die diese Funktionalität benötigen, können die fehlenden Funktionen hinzugefügt werden. Der *BSW Mode Manager (BswM)* ist für die Steuerung und Arbitrierung von Betriebsmodi verantwortlich. Ein Betriebsmodus beschreibt die aktuelle Betriebsart einer ECU. Je Modus können pro Modul intern unterschiedliche Zustandsmaschinen aktiv sein. Beispielsweise können die unterschiedlichen Zustände des ECU State Managers, wie etwa **STARTUP**, **RUN**, **SLEEP**, als Betriebsmodi realisiert sein und durch den BSW Mode Manager gesteuert werden.

Abbildung 3.10 grenzt APMC-OS innerhalb der um die Kernel-Bibliothek erweiterten AUTOSAR Systemarchitektur ab. Neben der Kernel-Bibliothek sind die OS, EcuM, BswM, MCU und RTE Module fester Bestandteil von APMC-OS. Um den unterschiedlichen Anforderungen von Anwendungen gerecht zu werden, kann APMC-OS um einen *Application Specific AR Stack* ‘anwendungsspezifischen AUTOSAR Stack’ erweitert werden, der beliebige AUTOSAR Module einschließen kann. Da auch bei den im APMC-OS enthaltenen Modulen gegenüber der jeweiligen Spezifikation nicht alle Funktionen enthalten sind, können *Additional BSW Module Functions* ‘zusätzliche BSW Modulfunktionen’ ergänzt werden. Entgegen der Abbildung hat das MCU Modul auch bei Nichtvorhandensein eines anwendungsspezifischen AUTOSAR Stacks direkten Zugriff auf die Hardware. APMC-OS bietet eine kompakte Evaluierungsplattform für parallelisierte Anwendungen aus dem Automobilumfeld mit der Möglichkeit zur anwendungsspezifischen Erweiterung an Modulen. Durch die Verwendung der Kernel-Bibliothek und den zugehörigen RTE Basisdiensten kann für die Dienste des OS Moduls eine zeitliche Obergrenze ermittelt werden. Dies ist vor allem bei Multi-/Manycore-Plattformen von Vorteil, da hier nicht faire Synchronisationsmechanismen zu zeitlichen Überschätzungen führen können, wie etwa der in (AUTOSAR 2015b) spezifizierte Spin-Lock Mechanismus.

4

Untersuchung von Cross-Core Service Calls im Automobil-OS

Seit der AUTOSAR Release 4.0 OS Spezifikation (AUTOSAR 2011) ist die Unterstützung von Multicore-Prozessoren ein fester Bestandteil der Betriebssystemspezifikation. Die AUTOSAR OS Spezifikation (AUTOSAR 2015b) beschreibt das allgemeine Multicore-Konzept, das Application Programming Interface (API) und die Funktionalitäten der OS-Dienste und deren Erweiterungen für eine Multicore-Unterstützung. Außerdem werden die erforderlichen Erweiterungen für die OS-Dienste erläutert, die von der OSEK-Spezifikation (OSEK/VDX 2005) übernommen wurden. Insgesamt sind bei AUTOSAR 19 OS-Dienste spezifiziert, die für eine Verwendung auf Multicore-Prozessoren erweitert wurden, und 8 neue OS-Dienste, die speziell für Multicore-Prozessoren eingeführt wurden. Des Weiteren wird der Inter-OS-Application Communicator (IOC) eingeführt, der für die Kommunikation zwischen OS-Applications zuständig ist. Da bei einem Multicore-Prozessor pro Kern mindestens eine OS-Application vorhanden sein muss, wird der IOC zur Kommunikation über Kerngrenzen hinweg benötigt. Die OS-Dienste bilden für viele Anwendungen und Module der Basic Software (BSW) die Grundlage für die Kommunikation zwischen Prozessorkernen. In (Bradatsch et al. 2014a) werden zur Optimierung des Datenaustauschs zwischen Kernen die *Cross-Core Service Calls* ‘Cross-Core Dienstaufrufe’ genauer betrachtet.

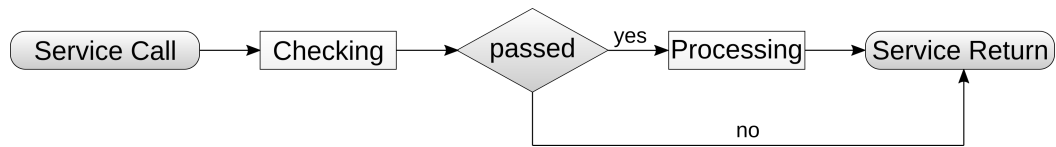


Abbildung 4.1: Generelle OS-Dienst Abarbeitungssequenz: Nach dem Dienstaufwurf folgt eine Überprüfungsphase, in der verschiedene Bedingungen kontrolliert werden. Je nach Auswertung findet danach die eigentliche Abarbeitung des Dienstes statt. Abschließend kehrt der Dienst mit einem Fehlerstatus zurück.

4.1 OS-Dienste

Ein *OS Service* ‘OS-Dienst’ ist ein System- bzw. Funktionsaufruf. Die OS-Dienste bilden sozusagen die API des Betriebssystems. Ein OS-Dienst hat folgende Eigenschaften: Er kann Eingabe- und Ausgabeparameter besitzen und liefert einen Fehlerstatus und in Abhängigkeit vom OS-Dienst einen Rückgabewert zurück. Fast alle OS-Dienste besitzen mindestens einen Eingabewert. Dabei handelt es sich zumeist um einen *Object Identifier (ID)* ‘Objekt-ID’, z. B. eine TaskID, die das OS-Objekt kennzeichnet, auf das sich der OS-Dienst bezieht. Ein OS-Objekt besitzt spezifische statische und dynamische Eigenschaften, je nach Objekttyp. Tasks, ISRs, Events, Alarme etc. sind Beispiele für OS-Objekttypen. Bei der Priorität eines Task-Objekttyps etwa handelt es sich um eine statische Eigenschaft, während der aktuelle Zustand, sprich *laufend*, *gestoppt* oder *bereit*, eine dynamische Eigenschaft beschreibt. Die Summe aller Eigenschaften von OS-Objekten wird forthin als *OS- bzw. Betriebssystemobjekt*daten betitelt. Abbildung 4.1 stellt die allgemeine Struktur eines OS-Dienstes dar. Der Programmablauf eines OS-Dienstes beginnt mit dem *Service Call*. Im Anschluss wird ein *Checking* ‘Überprüfung’ der Vorbedingungen durchgeführt, beispielsweise, ob sich die übergebenen Eingabeparameter in einem gültigen Wertebereich befinden. Falls die Überprüfung fehlschlägt, wird der Dienst mit einem *Return* ‘Rücksprung’ beendet. Andernfalls findet das *Processing* ‘Abarbeitung’ der entsprechenden Aufgaben statt und der Dienst kehrt im Anschluss zurück. Beim Beenden wird ein entsprechender Fehlerstatus zurückgeliefert. Dabei wird die Programmausführung solange verzögert, bis der Dienst beendet ist, d. h., der Dienstaufwurf ist synchron. Während einige OS-Dienste lediglich ein paar Statusinformationen abfragen, nehmen andere Dienste größere Änderungen am Systemstatus vor und führen u. U. zusätzlich ein Rescheduling nach Beenden des Dienstes durch. Mit Hilfe des OS-Objekt-ID-Parameters kann der

Prozessorkern identifiziert werden, auf dem sich das OS-Objekt befindet, auf das sich der Dienst bezieht. Einige OS-Dienste erfordern keine Objekt-ID. Dies bedeutet, dass der Dienst auf demselben Kern ausgeführt wird, auf dem er gestartet wurde. Der Fehlerstatus wird an den Dienst auf dem Kern gesendet, auf dem dieser aufgerufen wurde, unabhängig davon, ob die Aktion auf einem anderen oder demselben Kern durchgeführt wurde. Aufgrund der Tatsache, dass das Betriebssystem vollständig statisch konfiguriert ist, sind zur Zeit der Kompilierung sämtliche OS-Objekte und wo diese im Speicher liegen – und damit auch auf welchem Kern – bekannt. Dadurch muss keine dynamische Auflösung von IDs zur Laufzeit stattfinden, das der Komplexität der OS-Dienste zu Gute kommt.

4.2 Modell

Die AUTOSAR OS Spezifikation lässt die konkrete Implementierung der OS-Dienste offen. Für Multicore-Systeme werden im Folgenden zwei grundlegende Implementierungsansätze für Cross-Core Dienstaufrufe gegenübergestellt. Jeder Dienstaufruf liest während der Überprüfungsphase OS-Objektdaten und liest und/oder verändert OS-Objektdaten im Verlauf der Abarbeitungsphase. Abbildung 4.2 kontrastiert die beiden Zugriffsmethoden auf OS-Objektdaten, bei denen der Dienstaufruf und die Daten auf unterschiedlichen Kernen stattfindet bzw. liegen. In der Zeichnung werden die OS-Objektdaten als *OS Data* bezeichnet. Voraussetzung für die beiden Ansätze ist, dass alle Kerne über einen *Shared Memory* ‘geteilten Speicher’ miteinander verbunden sind. Auf jedem Prozessorkern wird ein separater *Kernel* ‘Betriebssystemkern’ ausgeführt, der seine eigenen OS-Objektdaten verwaltet. Sämtliche OS-Objektdaten, die für Cross-Core Dienstaufrufe benötigt werden, sind in dem geteilten Speicher abgelegt. Abbildung 4.2a illustriert einen *nachrichtenbasierten* Ansatz. Dabei wird der *Target Core* ‘Zielkern’ per Nachricht über einen Dienstaufruf vom *Source Core* ‘Quellkern’ informiert und der Dienst im Anschluss auf dem Zielkern ausgeführt. Da jeder Cross-Core Dienstaufruf auf dem jeweiligen Zielkern ausgeführt wird, besteht keine Notwendigkeit, auf die OS-Daten eines anderen Kernels bzw. Kerns zuzugreifen. Aus diesem Grund ist es für den nachrichtenbasierten Ansatz unerheblich, ob die OS-Daten in einem geteilten Speicherbereich liegen. Es wird jedoch von geteiltem Speicher ausgegangen, zwecks der besseren Vergleichbarkeit der beiden Ansätze und dem Umstand, dass zur Übertragung von Nachrichten ebenfalls geteilter

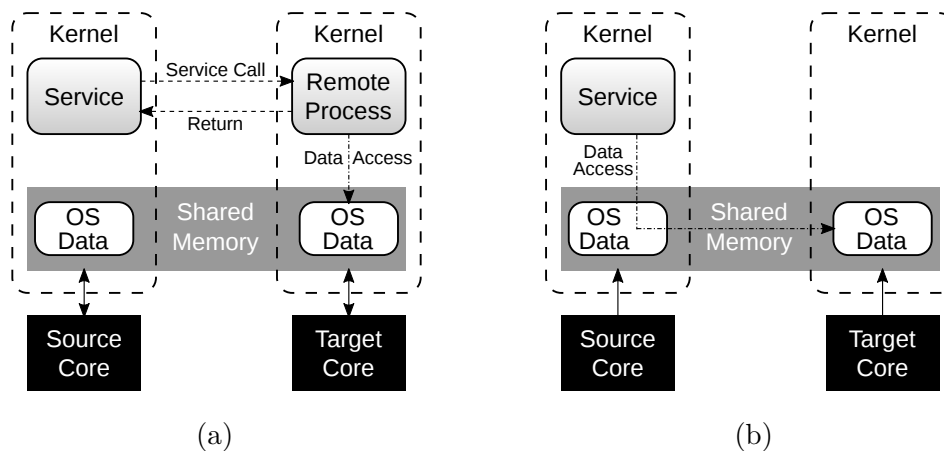


Abbildung 4.2: Vergleich zweier Zugriffsmethoden bei einem Cross-Core Service Call auf *OS Data* in einer *Shared Memory* Architektur: (a) Der *Service* stellt einen *Request* an den *Remote-Process* des *Kernel* auf dem *Target Core*. Der *Remote-Process* bearbeitet den *Request*, indem dieser auf die *OS Data* zugreift. Im Anschluss bestätigt der *Remote-Process* die Bearbeitung per *Reply* an den *Service* auf dem *Source Core*. (b) Der *Service* auf dem *Source Core* greift bei der Abarbeitung direkt auf die *OS Data* des *Kernel* vom *Target Core* zu, indem sich dieser einen exklusiven Zugriff auf die *OS Data* sichert.

Speicher verwendet wird. Abbildung 4.2b zeigt einen *Lock-basierten* Ansatz. In diesem Fall finden sowohl der Dienstaufruf als auch die Abarbeitung auf dem Zielkern statt. Um die relevanten OS-Objektdaten lesen bzw. verändern zu können, ist ein direkter Zugriff auf diese erforderlich. Hierfür ist es unabdingbar, dass die OS-Objektdaten im geteilten Speicher gehalten werden. Ferner ist es notwendig, dass der Zugriff durch wechselseitigen Ausschluss gesichert ist, um die Datenkonsistenz sicherzustellen.

Nachrichtenbasierter Ansatz: Abbildung 4.3 stellt den zeitlichen Verlauf eines OS-Dienstaufrufs gemäß dem nachrichtenbasierten Verfahren dar. Auf dem Quellkern ruft *Task 1* einen OS-Dienst auf, dessen ID auf ein OS-Objekt verweist, welches von dem Kernel auf dem Zielkern verwaltet wird. Vom Quellkern wird ein *Request* ‘Anfrage’ an den Zielkern gesendet, die eine Betriebssystemdienst-ID, beispielsweise die ID für den `ActivateTask(...)` Dienst, Betriebssystemobjekt-ID und optional weitere Parameter besitzt. Zum Zeitpunkt des Aufrufs führt der Zielkern *Task 2* aus. Dieser wird unmittelbar nach Ankunft der Nachricht unterbrochen und die Vorbedingungen des Dienstes werden

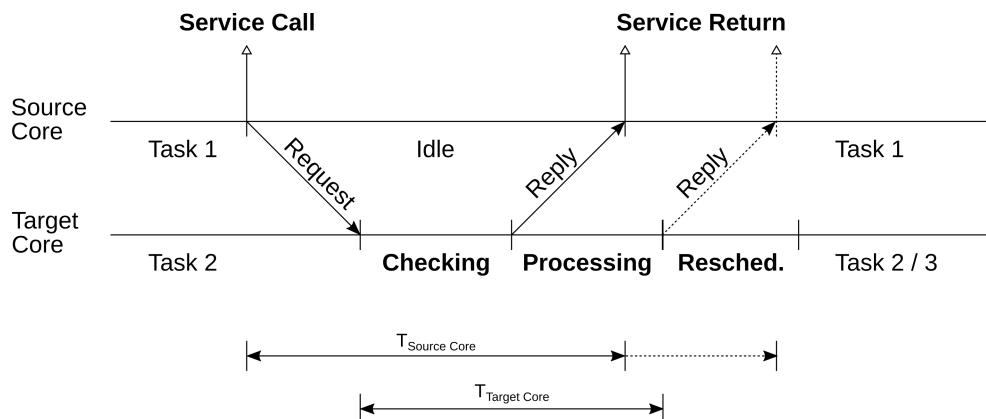


Abbildung 4.3: Abarbeitungssequenz eines Cross-Core Dienstaufrufs beim nachrichtenbasierten Verfahren: Task 1 initiiert einen Dienstaufruf, der eine Anfrage an den Zielkern sendet. Dabei wird Task 2 unterbrochen und der Zielkern beginnt mit der Überprüfungs- und Bearbeitungsphase. Je nach Dienstaufruf erfolgt eine Antwort vor oder nach der Bearbeitungsphase mit anschließender Dienstrückkehr und Wiederaufnahme der Ausführung von Task 1. Abhängig vom Dienst, erfolgt ein Rescheduling auf dem Zielkern und die Ausführung von Task 2 wird fortgesetzt oder ein höher priorisierter Task 3 wird gestartet.

überprüft. Je nach OS-Dienst wird ein *Reply* ‘Antwort’ nach der Überprüfungs- oder Bearbeitungsphase zurückgesendet. Bei Diensten, die neben dem Fehlerstatus weitere Werte zurückgeben, wird die Antwort erst nach der Bearbeitungsphase gesendet, da die entsprechenden Werte erst nach dieser Phase verfügbar sind. Dabei handelt es sich fast ausschließlich um *Get... (...)* Dienstaufrufe. Dienste, die einzig einen Fehlerstatus zurückliefern, antworten direkt nach der Überprüfungsphase, da der Dienst zu diesem Zeitpunkt nur noch mit einem abnormalen Fehler terminieren kann. Auf dem Quellkern wird der Dienstaufruf anschließend beendet und die Ausführung von Task 1 wird fortgeführt. Sollte der Fehlerstatus einen Fehler signalisieren, so ist dessen Behandlung die Aufgabe von Task 1. Je nach Systemzustand des Kernels auf dem Zielkern und der Art des OS-Dienstes kann ein Rescheduling stattfinden. Daher wird entweder die Ausführung von Task 2 fortgesetzt oder ein neuer Task wird eingewechselt, in Abbildung 4.3 beispielhaft als *Task 3* betitelt.

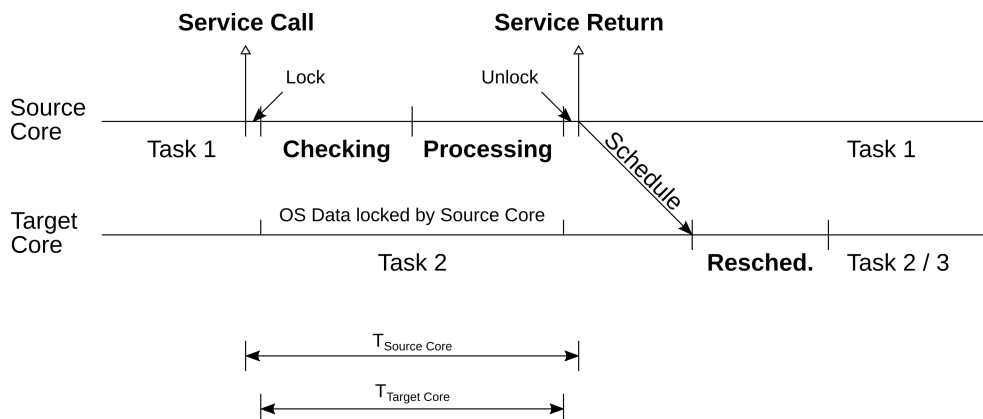


Abbildung 4.4: Abarbeitungssequenz eines Cross-Core Dienstaufrufs beim Lock-basierten Verfahren: Task 1 initiiert einen Dienstaufwurf, woraufhin die eigenen und die OS-Objektdaten auf dem Zielkern von Zugriffen anderer Kerne gesperrt sind. Anschließend findet die Überprüfungs- und Bearbeitungsphase statt und der Zugriff auf die Daten wird per Freigabe gestattet. In Abhängigkeit vom OS-Dienst sendet der Quellkern eine Scheduling-Nachricht an den Zielkern gefolgt von der Dienstrückkehr. Auf dem Zielkern wird unterdessen Task 2 ausgeführt. Je nachdem, ob eine Scheduling-Nachricht eintrifft, findet ein Rescheduling statt und die Ausführung von Task 2 wird fortgesetzt oder ein höher priorisierter Task 3 wird gestartet.

Lock-basierter Ansatz: Bei diesem Verfahren wird direkt auf die OS-Objektdaten auf dem Zielkern zugegriffen. Um Datenkonsistenz zu gewährleisten, wird der Zugriff per wechselseitigem Ausschluss gesichert. Abbildung 4.4 veranschaulicht den Lock-basierten Ansatz. *Task 1* ruft einen OS-Dienst auf, dessen OS-Objektdaten zum Kernel auf dem Zielkern gehören. Der Dienst versucht den exklusiven Zugriff auf die OS-Objektdaten zu erhalten. Nachdem dem Dienst der Zugriff gewährt wird, überprüft dieser die Vorbedingungen. Sofern alle Bedingungen gegeben sind, beginnt die Abarbeitung des Dienstes. Dabei werden die OS-Objektdaten des Kernels auf dem Zielkern gelesen bzw. verändert. Falls der Dienst kein Rescheduling erfordert, wird der exklusive Zugriff freigegeben und der Dienst beendet. Anschließend wird Task 1 fortgesetzt. In der Zeitspanne des exklusiven Zugriffs, in Abbildung 4.4 vom *Lock* ‘Sperre’ bis zum *Unlock* ‘Freigabe’, kann kein anderer OS-Dienst auf die OS-Objektdaten des Kernels vom Zielkern zugreifen, selbst dann nicht, wenn es sich um einen lokalen OS-Dienstaufwurf handelt. Sofern dies nicht nötig ist, kann während der gesamten Laufzeit des Dienstes die Ausführung auf

dem Zielkern ungehindert fortgeführt werden. Andernfalls wird die Ausführung eines anderen OS-Dienstes bis zur Freigabe verzögert. Für den Fall, dass der auf dem Quellkern gestartete Dienst ein Rescheduling voraussetzt, überprüft dieser, ob ein höher priorisierter Task als Task 2 zur Ausführung bereitsteht. Falls dem so ist, gibt der OS-Dienst den exklusiven Zugriff frei und sendet eine *Schedule*-Nachricht an den Zielkern, um ein Rescheduling anzustoßen. Dabei wird Task 2 unterbrochen und der Kernel auf dem Zielkern überprüft, ob ein Task-Wechsel ansteht. Je nach Ergebnis wird Task 2 fortgesetzt oder ein anderer Task wird eingewechselt. Die zweifache Überprüfung, ob ein Task-Wechsel anliegt, ist notwendig, da sich der Systemzustand aufgrund anderer OS-Dienstaufrufe zwischen Freigabe und Empfang der *Schedule*-Nachricht geändert haben könnte.

Beim nachrichtenbasierten Ansatz wird der Zielkern bei jedem OS-Dienstaufruf unterbrochen, der ein OS-Objekt auf diesem referenziert. Im Gegensatz dazu findet beim Lock-basierten Ansatz nur eine Unterbrechung auf dem Zielkern statt, wenn der OS-Dienst ein Rescheduling erfordert. Beiden Ansätzen ist gemein, dass während der Ausführung eines Dienstes nicht auf die OS-Objektdaten des Kernels auf dem Zielkern von einem anderen Dienst zugegriffen werden kann. Dies wird dadurch verhindert, dass bei dem einen Ansatz der OS-Dienst auf dem Zielkern ausgeführt wird und bei dem Anderen der Zugriff auf die OS-Objektdaten durch wechselseitigen Ausschluss unterbunden wird.

4.3 Betrachtung des Zeitverhaltens

Anwendungen aus dem Automobilumfeld müssen meist Echtzeitanforderungen erfüllen. Da sämtliche AUTOSAR Dienste Bestandteil einer Anwendung sein können, müssen auch diese den zeitlichen Anforderungen genügen. Daher ist das Zeitverhalten eines Dienstes für ein System von Bedeutung. Die Einhaltung von Echtzeitanforderungen erfordert, dass für die Ausführungszeit von Diensten eine sichere Zeitschranke berechnet werden kann. Des Weiteren müssen Interferenzen zwischen verschiedenen Tasks zeitlich eingrenzbar sein, wenn sie nicht vollständig ausgeschlossen werden können. Ein derartiges Verhalten muss auch auf Multicore-Prozessoren gewährleistet werden.

In Bezug auf die Ausführung eines Dienstes gilt folgender Sachverhalt: Die OS-Dienste werden längst in heutigen Implementierungen für Singlecore-Plattformen eingesetzt, wo

sie bereits zeitliche Vorhersagbarkeit erfüllen. Dies bedeutet, dass für die Überprüfungs- und Bearbeitungsphase auf einem Singlecore-Prozessor obere Zeitschranken abgeleitet werden können. Da das Kernstück der Implementierung ebenso Verwendung bei der Multicore-Variante der OS-Dienste findet, bleibt die zeitliche Berechenbarkeit bestehen. Die Migration auf Multicore-Prozessoren führt einige zusätzliche Mechanismen ein, die weitere Untersuchungen erfordern. Insbesondere müssen Konstellationen untersucht werden, die bei einer Singlecore-Plattform nicht zu Tage treten, jedoch höchstwahrscheinlich bei der Umsetzung auf eine Multicore-Plattform.

Beim nachrichtenbasierten Verfahren gilt es zuerst zu untersuchen, inwiefern der Empfang einer Nachricht beim Zielkern verzögert werden kann, nachdem diese vom Quellkern versendet wurde. Im ungünstigsten Fall würden Nachrichten, die fast zeitgleich von anderen OS-Dienstaufrufen an denselben Zielkern gesendet wurden, vor der Nachricht des betrachteten OS-Dienstes bearbeitet werden. In solch einem Szenario kann die Bearbeitung des OS-Dienstes verzögert werden, wobei die Wartedauer beschränkt werden kann. Dienstauf-rufe werden synchron ausgeführt, d. h., dass ein Task, der einen ausstehenden Dienstaufruf hat, blockiert wird, bis der Aufruf zurückkehrt. Somit kann ein Task nicht mehr als einen Dienstaufruf gleichzeitig ausstehend haben. Da das System statisch konfiguriert ist, ist die Anzahl aller Tasks zum Integrationszeitpunkt bekannt, wodurch eine obere Grenze für alle Dienstauf-rufe gegeben werden kann, die zur Bearbeitung durch den Kernel des Zielkerns anstehen können. Dieser Grenzwert kann noch durch die maximale Anzahl an echt-parallelen Tasks eingeengt werden, das identisch ist mit der zur Verfügung stehenden Anzahl an Prozessorkernen zur Task-Ausführung. Die Nachrichten der Dienstauf-rufe werden entsprechend der Eingangsreihenfolge bearbeitet. Eine grobe Abschätzung für die Wartezeit zwischen dem Dienstaufruf und dessen Bearbeitungsbeginn im ungünstigsten Fall wäre die Anzahl der Prozessorkerne abzüglich dem betrachteten Dienst auf dem Quellkern multipliziert mit der zeitlichen Abschätzung für den komplexesten Dienst. Nach Bearbeitungsbeginn des Dienstes wird dieser ohne Unterbrechung auf dem Zielkern ausgeführt.

Beim Lock-basierten Ansatz kann dem OS-Dienst ebenso eine Wartedauer widerfahren, bis der exklusive Zugriff auf den geteilten Speicher und somit auf die OS-Objektdaten gewährt wird. Um zeitliche Schranken ermitteln zu können, wird ein fairer Mechanismus zum wechselseitigen Ausschluss benötigt. In diesem Zusammenhang ist solch ein Mechanismus wie folgt definiert: “Wenn mehr als ein Task um den Eintritt in einen kritischen Abschnitt

konkurrieren, muss eine deterministische Zugriffsreihenfolge (z. B. first-come, first-served ...) gewährleistet sein, die es ermöglicht, für jeden Task eine bestimmbare Wartezeit für den Eintritt (und Austritt) in den kritischen Abschnitt zu ermitteln" (Gerdes 2013, S. 10). Aufgrund dieser Anforderung gelten für das Lock-basierte Verfahren dieselben Umstände bezüglich der Wartezeit zwischen dem Dienstaufruf und dessen Bearbeitungsbeginn wie für den nachrichtenbasierten Ansatz im vorherigen Abschnitt.

Darüber hinaus müssen die Auswirkungen auf den Task betrachtet werden, der auf dem Zielkern ausgeführt wird – Task 2 in den Abbildungen 4.3 und 4.4. Beim nachrichtenbasierten Ansatz wird Task 2 jedes mal unterbrochen, sobald ein OS-Dienst aufgerufen wird, der sich auf ein OS-Objekt bezieht, das vom selben Kernel wie Task 2 verwaltet wird. Eine Analyse des Systems und seines Verhaltens kann Informationen zur Auftrittsratesolcher Unterbrechungen liefern. Durch Verwendung bekannter Techniken, wie Event Streams (Gresser 1993), ist es möglich, solche Unterbrechungen in der Planbarkeitsanalyse zu berücksichtigen und insofern ein vorhersagbares Zeitverhalten zu erreichen. Im Falle des Lock-basierten Ansatzes (vgl. Abbildung 4.4) wird Task 2 nur unterbrochen, wenn der Dienst ein Rescheduling erfordert. Dieses Verhalten unterscheidet sich nicht von dem, das bei Implementierungen für Singlecore-Plattformen beobachtet werden kann. Auch hier wird der momentan aktive Task bei einem Rescheduling ausgesetzt. Solch ein Verhalten wird bereits durch gewöhnliche Planbarkeitsanalysen abgedeckt, z. B. nach dem Rate Monotonic Verfahren (Liu et al. 1973). Ferner kann Task 2 eine zusätzliche Blockierung widerfahren, wenn dieser einen (lokalen) Dienstaufruf ausführt, während ein anderer Task den exklusiven Zugriff auf die OS-Objektdaten hält. Wie bereits zuvor erwähnt, ist die Wartezeit durch die maximale Anzahl an echt-parallelen Tasks, die gleichzeitig auf den Zugriff des kritischen Abschnitts warten, nach oben beschränkt.

Obwohl beide Ansätze hinsichtlich Echtzeitbedingungen realisierbar sind, scheint der Lock-basierte Ansatz unter folgenden Gesichtspunkten gewinnbringender zu sein: Beim nachrichtenbasierten Verfahren wird die Task-Ausführung auf dem Zielkern auf jeden Fall ausgesetzt und Rechenzeit geht dadurch verloren. Im Gegensatz dazu wird beim Lock-basierten Ansatz der Zielkern nur unterbrochen, wenn der OS-Dienst ein Rescheduling erfordert. Solange kein lokaler OS-Dienstaufruf stattfindet, kann die Ausführung auf dem Zielkern ungehindert ablaufen.

4.4 Implementierung

Für die Realisierung muss die zugrunde liegende Plattform gewisse Voraussetzungen erfüllen. Das nachrichtenbasierte Verfahren benötigt einen synchronen Mechanismus zur *Inter-core Communication (ICC)* ‘Kommunikation zwischen Prozessorkernen’ und der Lock-basierte Ansatz einen fairen Mechanismus für wechselseitigen Ausschluss. Als Plattform wurde der Manycore-Simulator aus dem parMERASA Projekt gewählt (Ungerer et al. 2013), der die genannten Anforderungen erfüllt und in Kapitel 4.5 genauer beschrieben ist. Die ICC erfolgt über ein optimiertes *Message Passing Interface (MPI)* ‘Schnittstelle für Nachrichtenaustausch’. Die Schnittstelle unterstützt eine angepasste Sendeoperation

```
void MessageSend(void *buf, int size, int coreID, int serviceID);
```

und Empfangsoperation

```
void MessageRecv(void *buf, int size, int coreID, int serviceID); .
```

Beide Operationen besitzen folgende Parameter: Zeiger auf den Sende-/Empfangspuffer `buf`, Puffer- bzw. Nachrichtengröße `size`, ID des Quell-/Zielkerns `coreID` und ID des Dienstaufrufs `serviceID`. Die `coreID` bestimmt bei `MessageSend` den Zielkern, wohin gegen sie bei `MessageRecv` den Quellkern spezifiziert, von dem Nachrichten bearbeitet werden. Anhand der `serviceID` können die verschiedenen OS-Dienste unterschieden werden. Nur wenn die `serviceID` auf Sender- und Empfängerseite gleich sind, wird die Nachricht akzeptiert. Damit ein Aufruf von `MessageRecv` auch Nachrichten von einem beliebigen Quellkern und/oder Dienst annimmt, können anstatt konkreter IDs auch `ANY_CORE` bzw. `ANY_SERVICE` als Parameter übergeben werden. Größe und Inhalt einer Nachricht hängen vom jeweiligen OS-Dienst ab. Fast alle OS-Dienste erfordern eine Betriebssystemobjekt-ID und bestimmte Übergabeparameter.

Nachrichtenbasierter Ansatz: Beim nachrichtenbasierten Verfahren wird jeder Cross-Core Dienstaufruf auf dem Zielkern ausgeführt. Zu Beginn überprüft jeder Dienst, ob die übergebene Betriebssystemobjekt-ID gültig ist. Anhand dieser ID wird der Zielkern bestimmt. Wenn es sich um ein lokales OS-Objekt handelt, wird der Dienst wie bei der Singlecore-Implementierung abgearbeitet. Andernfalls wird per `MessageSend`

eine Nachricht mit der `coreID` des Zielkerns, der `serviceID` des entsprechenden Dienstes, der Größe der Nachricht und einem Zeiger auf den Nachrichteninhalte, der weitere optionale Parameter des Dienstes enthalten kann, gesendet. Der Zielkern wird durch einen Inter-Core-IRQ benachrichtigt. Anschließend wartet der Quellkern auf die Antwort mit dem Fehlerstatus und, je nach OS-Dienst, auch den Rückgabewerten. Hierfür wird `MessageRecv` mit der `coreID` des Zielkerns und der `serviceID` von `ReturnStatus` aufgerufen. Bei `ReturnStatus` handelt es sich nicht um einen OS-Dienst im eigentlichen Sinne. Allerdings bekommt dieser eine eigene `serviceID` zugewiesen, um die Nachricht von den übrigen Betriebssystemdienst-IDs unterscheiden und über dieselbe Nachrichtenschnittstelle versenden zu können. Nach Erhalt dieser Nachricht beendet sich der Dienst und gibt den Fehlerstatus an den aufrufenden Task zurück. Seitens des Zielkerns ist eine Interrupt Service Routine (ISR) für den Empfang und die Auswertung der Nachricht zuständig. Um keine Nachricht zu verwerfen, wird `MessageRecv` mit den Parametern `ANY_CORE` und `ANY_SERVICE` aufgerufen. Nach dem Empfang einer Nachricht wertet die ISR die `serviceID` aus und führt die Überprüfungs- und Bearbeitungsphase durch. Hierfür werden die in der Nachricht gekapselten Parameter einbezogen. Je nach OS-Dienst erfolgt die Antwort vor oder nach der Bearbeitungsphase. Zu diesem Zweck wird `MessageSend` mit der `coreID` des Quellkerns, der `serviceID` von `ReturnStatus` und dem Fehlerstatus bzw. den Rückgabewerten als Nachrichteninhalte aufgerufen. Je nach OS-Dienst wird zum Schluss entweder die Ausführung des zuvor unterbrochenen Tasks fortgesetzt oder es findet ein Rescheduling statt und der höchst priorisierte Task wird eingewechselt.

Lock-basierter Ansatz: Dieses Verfahren beginnt insofern auf die gleiche Weise wie der nachrichtenbasierte Ansatz, dass die Betriebssystemobjekt-ID auf Gültigkeit überprüft wird. Wenn die ID gültig ist, versucht der Quellkern exklusiven Zugriff auf die OS-Objektdaten des Zielkerns zu erlangen. Hierbei müssen alle OS-Objektdaten des Kerns für andere OS-Dienstaufrufe gesperrt sein, da nicht nur das relevante OS-Objekt, beispielsweise ein Task-Objekt, allein von Bedeutung ist, sondern einige Dienste an dem zentralen OS-Objekt Änderungen vornehmen müssen. Da nicht vorhergesehen werden kann, ob ein OS-Dienst, der das zentrale OS-Objekt verändern muss, aufgerufen wird, muss dieses grundsätzlich durch wechselseitigen Ausschluss geschützt sein. Nachdem der exklusive Zugriff stattgegeben wurde, findet die Überprüfungs- und Bearbeitungsphase statt. Fordert der OS-Dienst ein Rescheduling, wird überprüft, ob ein Task-Wechsel

erfolgen soll. Ist ein Task-Wechsel erforderlich, wird eine Nachricht per `MessageSend` mit der `coreID` des Zielkerns und der `serviceID` für *Schedule* gesendet. Analog zur `serviceID` von `ReturnStatus` handelt es sich hierbei um keinen echten OS-Dienst. Zur Differenzierung von anderen Diensten wird *Schedule* eine eigene ID zugewiesen. Danach wird ein Inter-Core-IRQ an den Zielkern ausgelöst, um diesen über die Nachricht zu informieren. Anschließend wird der kritische Abschnitt freigegeben. Der Dienstaufruf wird beendet und gibt den Fehlerstatus an den aufrufenden Task zurück. Wie schon beim nachrichtenbasierten Verfahren erläutert, bearbeitet eine ISR ankommende Nachrichten. Nach Auswertung der `serviceID` der Nachricht, wird ein Rescheduling auf dem Zielkern angestoßen.

4.5 Evaluierung

Für die Evaluierungen wurde der `ActivateTask(...)` OS-Dienstaufruf gewählt. *ActivateTask*, *ChainTask* und *SetEvent* sind die einzigen Cross-Core Dienstaufträge, die ein Rescheduling auf dem Zielkern veranlassen. Zudem ist das Einfügen des zu aktivierenden Tasks in die entsprechende Task-Liste erforderlich. Aus diesen Gründen handelt es sich hierbei mit um eine der komplexesten Dienstaufträge. Der Ablauf beim nachrichtenbasierten Verfahren entspricht der nicht gestrichelten Sequenz aus Abbildung 4.3, d.h. die Antwort wird direkt nach der Überprüfungsphase gesendet. Da die Abläufe der Dienste fast identisch sind, wird im Folgenden nur `ActivateTask` betrachtet. Ein Kriterium, weshalb `ActivateTask` für die Evaluierung ausgewählt wurde, ist seine Komplexität. Der andere Grund liegt in der Tatsache begründet, dass er auf Anwendungsebene (ETAS Group Automotive LifeCycle Solutions 2009) sowohl für die Client-Server- als auch Sender-Receiver-Kommunikation zwischen OS-Applications (AUTOSAR 2015b) und somit auch zwischen unterschiedlichen Prozessorkernen verwendet wird. Die zweite Sequenz aus Abbildung 4.3 mit der Antwort nach der Bearbeitungsphase wird nicht weiter untersucht. Hierbei handelt es sich ausschließlich um `Get...(...)` Aufrufe, die neben dem Fehlerstatus einen weiteren Wert zurückgeben. Da hier in der Bearbeitungsphase lediglich ein Wert ausgelesen wird, verlängert dies die gesamte Sequenz nur minimal. Im Gegensatz dazu beeinflusst die Bearbeitungsphase von `ActivateTask` den zeitlichen Ablauf des Dienstes deutlich.

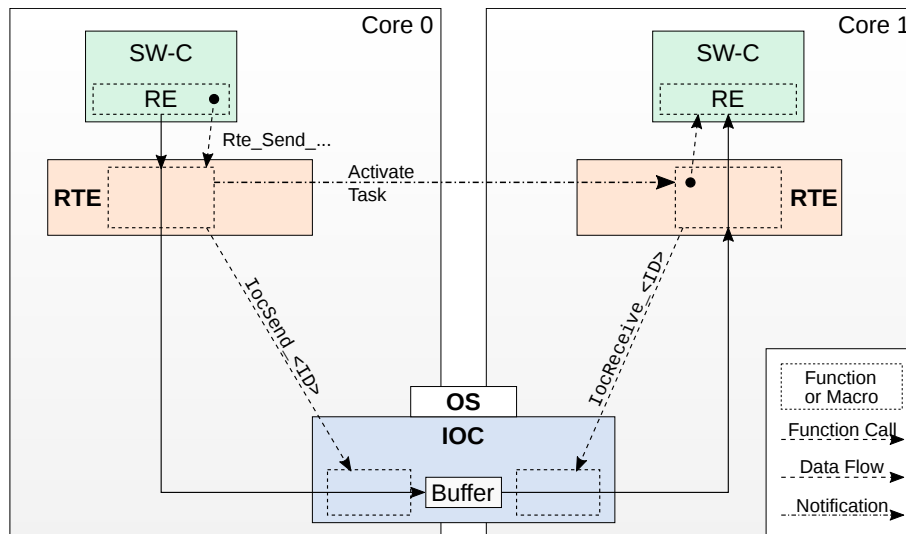


Abbildung 4.5: Sender-/Receiver-Kommunikation zwischen Kernen: Die RE der SW-C auf Core 0 initiiert einen `Rte_Send_...` Aufruf. Die RTE übergibt die Daten per `IocSend_<ID>` Aufruf an den IOC und aktiviert per `ActivateTask` einen Task auf Core 1. Dieser liest die Daten per `IocReceive_<ID>` Aufruf aus dem Buffer und benachrichtigt die RE der SW-C auf Core 1 über den Erhalt der Nachricht (AUTOSAR 2015b).

Abbildung 4.5 zeigt ein exemplarisches Sender-Receiver-Kommunikationsszenario zwischen zwei SW-Cs, die zu verschiedenen Kernen gehören. Für die Kommunikation zwischen OS-Applications bzw. Prozessorkernen wird zudem der Inter-OS-Application Communicator (IOC) verwendet. Diese Art von Mechanismus wird nicht nur für die Sender-Receiver-Kommunikation mit Benachrichtigung verwendet, sondern auch für die Client-Server-Kommunikation. In dem Beispiel aus Abbildung 4.5 sendet die RE der SW-C von Core 0 'Kern 0' Daten zur RE der SW-C auf Core 1 'Kern 1'. Dazu ruft die RE der SW-C von Kern 0 die Funktion `RTE_Send_...` auf. Die RTE ist dafür verantwortlich, dass die Daten von der RE auf Kern 0 zur RE auf Kern 1 transportiert werden. Für die Anwendung ist die Datenübermittlung transparent, d. h. sie hat keinerlei Kenntnis darüber, ob die SW-C, für die die Daten bestimmt sind, zur selben OS-Application gehören und falls nicht, ob die OS-Applications demselben Prozessorkern zugeteilt sind. Da sich in dem skizzierten Beispiel die SW-Cs auf verschiedenen Kernen befinden, wird die Kommunikation von Seiten der RTE per `IOC` abgewickelt. Dazu ruft die RTE die Funktion `IocSend_<IocId>(...)` auf, wodurch die Daten in einen geteilten `Buffer` 'Puf-

Tabelle 4.1: OS-Dienst-Ausführungszeiten in Prozessortakten bei einer Dualcore-Konfiguration

Zeit \ Verfahren	nachrichtenbasiert	Lock-basiert	Mehraufwand
$t_{\text{Source Core}}$	41513	12499	232 %
$t_{\text{Target Core}}$	18728	8550	119 %

fer' geschrieben werden. Anschließend ruft die RTE ActivateTask auf, um einen Lese-Task auf Kern 1 zu starten, der die Daten per `IocReceive_<IocId>(...)` Funktionsaufruf aus dem Puffer liest. Der Lese-Task wiederum informiert die RE über den Erhalt der Daten und leitet diese zur RE. Ohne dem Aufruf von ActivateTask, müsste die RE auf Kern 1 periodisch überprüfen, ob neue Daten im Puffer vorliegen.

Um beide Verfahren zu vergleichen, werden die Zeiten, wie in Abbildung 4.3 bzw. 4.4 dargestellt, gemessen. Bei beiden Ansätzen wird die Dauer des Dienstaufrufs auf dem Quellkern $t_{\text{Source Core}}$ und dem Zielkern $t_{\text{Target Core}}$ ermittelt. Bei $t_{\text{Target Core}}$ handelt es sich beim nachrichtenbasierten Ansatz um die Laufzeit der ISR auf dem Zielkern und bei dem anderen Verfahren um die Zeitspanne, währenddessen der Zielkern keinen Zugriff auf die OS-Objektdaten hat. Für die gemessenen Werte wurde ActivateTask jeweils 10 mal ausgeführt und die Ergebnisse wurden gemittelt.

Als Plattform wurde der Simulator aus dem parMERASA Projekt gewählt, der auf SocLib (SOCLIB Consortium 2013) basiert. Bei SocLib handelt es sich um eine Plattform zum Bau virtueller Prototypen von Multiprocessor System-on-Chips (MPSoCs). SocLib ist Taktzyklen-genau und bietet einen globalen Adressraum. Der parMERASA Simulator modelliert einen MPSoC, der aus mehreren Clustern besteht, die durch ein Network-on-Chip (NoC) untereinander verbunden sind. Jedes Cluster besteht aus bis zu acht Kernen und verfügt über einen separaten geteilten Speicher. Jeder Kern besitzt einen Level 1 Instruktions- und Datencache, wobei Letzterer kohärent ausgelegt ist (Pyka et al. 2014). Allerdings muss die Kohärenz explizit durch entsprechende Funktionsaufrufe gehandhabt werden. Zur Benachrichtigung können zwischen Prozessorkernen Inter-Core-IRQs ausgelöst werden. Als Softwarebasis kommt die in Kapitel 3 vorgestellte erweiterte Softwarearchitektur mit ACMC-OS zum Einsatz. Bei dem fairen Mechanismus zum wechselseitigen Ausschluss der Kernel-Bibliothek handelt es sich um eine *Ticket-Lock* Implementierung, die auf einer *fetch-and-add* Instruktion beruht. Für die Evaluierungen

Tabelle 4.2: OS-Dienst-Ausführungszeiten in Prozessortakten bei einer Quadcore-Konfiguration

Zeit \ Verfahren	nachrichtenbasiert	Lock-basiert	Mehraufwand
$t_{\text{Source Core}}$	56726	18104	213 %
$t_{\text{Target Core}}$	32156	10024	221 %

wurde ein einzelnes Cluster verwendet, da dieses einen geteilten Speicher verwendet und gängigen Prozessoren für eingebettete Systeme im Automobilbereich, z. B. Qorivva MPC5643L, gleich kommt.

Für das erste Testszenario wurde eine Dualcore-Konfiguration gewählt. Der Ablauf eines Dienstaufrufs ist dabei identisch zu den Sequenzen aus Abbildung 4.3 bzw. 4.4. In beiden Fällen ruft Task 1 den Dienstaufruf ActivateTask auf, um einen Task auf dem Zielkern zu aktivieren. Es wird davon ausgegangen, dass Task 2 zu diesem Zeitpunkt keinen OS-Dienst ausführt. Dies würde bei beiden Ansätzen die Dauer des Dienstes auf Task 1 gleichermaßen um eine gewisse Zeitspanne verzögern. Sowohl beim Lock- als auch nachrichtenbasierten Ansatz würde die Bearbeitungsphase erst beginnen, wenn der Dienstaufruf von Task 2 beendet wäre. Ferner spielt das Rescheduling für die Messungen keine Rolle, da der eigentliche OS-Dienst beendet ist, bevor ein Rescheduling stattfindet. Zudem ist dies die Aufgabe des Schedulers und nicht des OS-Dienstes. Außerdem erfordern nur einige wenige OS-Dienste ein Rescheduling.

Die Ergebnisse sind in Tabelle 4.1 aufgeführt. Die Zeiten sind in Prozessortakten angegeben und die letzte Spalte verdeutlicht den Mehraufwand des nachrichtenbasierten gegenüber des Lock-basierten Verfahrens. Wie aus Tabelle 4.1 zu erkennen ist, liegen die Ausführungszeiten des nachrichtenbasierten Ansatzes deutlich über denen des Lock-basierten Ansatzes. Dies ist darauf zurückzuführen, dass der Aufwand für den Nachrichtenaustausch augenscheinlich höher ist, als für den Mechanismus zum wechselseitigen Ausschluss. Darüber hinaus kann es bei der Dualcore-Konfiguration keinen weiteren Kern geben, der zeitgleich eine Anfrage an den Zielkern stellt. Somit wird dem Quellkern der Zugriff auf den kritischen Abschnitt unmittelbar gestattet.

Bei dem zweiten Testszenario kam eine Quadcore-Konfiguration zum Einsatz. Dabei versuchen drei unterschiedliche Quellkerne gleichzeitig einen Task auf dem Zielkern zu

Tabelle 4.3: Mehraufwand der Quadcore- im Gegensatz zur Dualcore-Konfiguration

Zeit \ Verfahren	nachrichtenbasiert	Lock-basiert
$t_{\text{Source Core}}$	37 %	45 %
$t_{\text{Target Core}}$	72 %	17 %

aktivieren. Im Falle des nachrichtenbasierten Ansatzes bedeutet dies, dass die Anfragen aller Quellkerne fast zeitgleich am Zielkern eintreffen. Gemessen wurden die Zeiten der zuerst eintreffenden Anfrage. Somit sind die Ergebnisse vergleichbar mit denen der Dualcore-Konfiguration. Ferner verlängern sich die Ausführungszeiten der nachfolgenden OS-Dienstaufrufe um die Überprüfungs- bzw. Bearbeitungsphase auf dem Zielkern. Gleiches gilt für den Lock-basierten Ansatz. Auch hier wurden die Zeiten von dem OS-Dienst gemessen, der zuerst den Zugriff auf die OS-Objektdateien erhalten hat. Die Zeiten für die nachfolgenden OS-Dienstaufrufe verlängern sich auch hier entsprechend der Überprüfungs- und Bearbeitungsphase. Die Ergebnisse aus Tabelle 4.2 bescheinigen auch hier dem Lock-basierten Ansatz einen Vorteil gegenüber dem nachrichtenbasierten Ansatz. Wie schon bei der Dualcore-Konfiguration ist dies auf den Mehraufwand beim Nachrichtenaustausch zurückzuführen.

Tabelle 4.3 setzt den Mehraufwand bei der Quadcore-Konfiguration im Vergleich zur Dualcore-Konfiguration in Relation. Dabei wird ersichtlich, dass sich der Mehraufwand beim Lock-basierten Verfahren auf dem Zielkern um ca. 17%, wohingegen auf dem Quellkern um 45% erhöht. Eine grundlegende Zunahme ist erhöhten Zugriffskonflikten auf den gemeinsamen Bus zuzuschreiben. Der höhere Mehraufwand auf dem Zielkern im Kontrast zum Quellkern liegt darin begründet, dass bei der Zeitmessung auf dem Quellkern der Ein- und Austritt zum kritischen Abschnitt mit einbegriffen sind und hierbei drei Kerne um den Zugang zum kritischen Abschnitt buhlen. Beim nachrichtenbasierten Verfahren ist der Mehraufwand auf dem Zielkern fast doppelt so hoch wie auf dem Quellkern. In absoluten Prozessortakten betrachtet, benötigt die Quadcore-Konfiguration allerdings auf beiden Kernen ca. 14000 Takte länger. Dies ist dem Mehraufwand beim Nachrichtenaustausch geschuldet, der sich sowohl bei $t_{\text{Source Core}}$ als auch bei $t_{\text{Target Core}}$ übereinstimmend niederschlägt.

4.6 Angrenzende Arbeiten

Die beiden beschriebenen Verfahren entsprechen zum Teil dem *Multiprocessor Priority Ceiling Protocol (MPCP)* (Rajkumar et al. 1988). Das MPCP erweitert das PCP (Sha et al. 1990), welches u. a. im OSEK Standard spezifiziert ist, um globale kritische Abschnitte. Darüber hinaus vermeidet es Verklemmungen und ermöglicht, eine obere Grenze für die Dauer von Blockierungen anzugeben. Beim MPCP wird ebenso wie bei OSEK bzw. AUTOSAR davon ausgegangen, dass Tasks statisch an den jeweiligen Prozessorkern gebunden sind und Jobs nicht dynamisch zwischen Kernen verschoben werden und dass Rate Monotonic Scheduling (RMS) verwendet wird. Ferner wird vorausgesetzt, dass innerhalb globaler kritischer Abschnitte keine lokalen kritischen Abschnitte ineinander geschachtelt sind und umgekehrt. Nicht kritischer Anwendungscode und lokale kritische Abschnitte eines Tasks werden grundsätzlich auf dem jeweiligen Prozessorkern, auch *Anwendungsprozessorkern* genannt, ausgeführt, an welchen der Task gebunden ist. Alle globalen kritischen Abschnitte hingegen, die einem globalen Semaphor zugeordnet sind, werden auf demselben Prozessorkern, dem *Synchronisationsprozessorkern*, ausgeführt. Neben der Synchronisierung globaler Semaphore können auf diesem Kern auch Anwendungstasks ausgeführt werden. In dem MPCP Modell wird die Ausführung eines globalen kritischen Abschnitts als Migration des korrespondierenden Jobs auf den Synchronisationsprozessorkern betrachtet. In Wirklichkeit wird auf dem Synchronisationsprozessorkern ein entsprechender Prozess mit derselben Priorität gestartet, der den globalen kritischen Abschnitt ausführt. Währenddessen kann der Anwendungsprozessorkern anderen Aufgaben nachkommen. Um die eingangs genannten Eigenschaften zu erfüllen, müssen bestimmte Bedingungen erfüllt sein. Im Folgenden wird die höchste Priorität aller Tasks, die in dem gesamten betrachteten System vorkommen, als P_H bezeichnet. Nach (Rajkumar et al. 1988) wird einem globalen Semaphor S eine Prioritätsobergrenze zugewiesen, die höher ist als P_H . Wenn S_i und S_j zwei globale Semaphore sind, deren jeweils höchstpriorer Task die Priorität P_i bzw. P_j hat und $P_i > P_j$, so gilt, dass die Prioritätsobergrenze von S_i größer ist als die von S_j . Dadurch kann ein Job, der einen höherpriorigen globalen Semaphor beansprucht, einen Job, der einen niederpriorigen globalen Semaphor hält, unterbrechen.

Im Falle der Cross-Core Dienstaufrufe sieht der Zusammenhang zum MPCP wie folgt aus.

Jeder OS-Dienst wird als Kernel-Prozess ausgeführt. Dieser Kernel-Prozess besitzt eine höhere Priorität als der höchstprioritäre Task des jeweiligen Kernels. Zudem haben sämtliche dieser Kernel-Prozesse dieselbe Priorität inne, d. h., ihre Priorität ist höher als der höchstprioritäre Task des Systems. Aus Sicht des MPCPs handelt es sich bei allen OS-Diensten bzw. jenen Kernel-Prozessen um Tasks mit gleicher und höchster Priorität im System. Dies bedeutet, dass unterschiedliche globale Semaphore dieselbe Prioritätsobergrenze besitzen und sich gegenseitig nicht unterbrechen können. Der Zugriff auf die OS-Objektdaten des Kernels eines jeden Prozessorkerns bildet einen separaten globalen kritischen Abschnitt, dessen Synchronisationsprozessorkern der Prozessorkern ist, auf dem die jeweiligen OS-Objektdaten liegen. Beim nachrichtenbasierten Ansatz wird die Anfrage des OS-Dienstaufrufs an den Zielkern gesendet, der zugleich als Synchronisationsprozessorkern fungiert. Somit wird der kritische globale Abschnitt auf dem Synchronisationsprozessorkern ausgeführt, analog zum MPCP. Da die Priorität des Prozesses auf dem Zielkern höher ist als die Prioritäten aller Tasks und identisch ist mit den Prioritäten aller anderen OS-Dienste, wird der Dienst durch keinen anderen Task oder OS-Dienst unterbrochen. Im Gegensatz zum MPCP kann der Quellkern bzw. Anwendungsprozessorkern unterdessen keine anderen Aufgaben erledigen, da der OS-Dienst synchron ist und den Quellkern bis zur Beendigung des Dienstes blockiert. Dadurch widerfährt dem Quellkern dieselbe Dauer der Blockierung wie dem Zielkern plus die Zeit für den Nachrichtenversand (vgl. Abbildung 4.3). Beim Lock-basierten Ansatz wird der globale kritische Abschnitt auf dem jeweiligen Quellkern bzw. Anwendungsprozessorkern ausgeführt. Im Vergleich zum MPCP gibt es keinen Synchronisationsprozessorkern. Da der OS-Dienst auf dem Quellkern nicht präemptiv ist und auch der Zugriff auf die eigenen OS-Objektdaten geschützt ist, wird eine Verklemmung vermieden. Analog zum nachrichtenbasierten Ansatz ist die Dauer der Blockierung des Quellkerns abhängig von der Sperrdauer der OS-Objektdaten des Kernels auf dem Zielkern.

Ein weiterer Forschungsansatz, der ein ähnliches Themengebiet behandelt, wird in (Böhm et al. 2010) beschrieben. Der Fokus liegt hierbei in der Exploration von AUTOSAR Systemsoftware bezüglich Multicore-Systemen. Ein weiteres Kriterium ist dabei eine möglichst einfache Portierung von *Legacy Code* 'bewährten Code' auf solche Systeme. Dabei liegt das Hauptaugenmerk auf der Untersuchung von Systemdienstaufrufen, sprich Dienstaufrufe an die Basic Software, zwischen Kernen. Dabei werden verschiedene Ansätze gegenübergestellt, im Detail sind dies:

- **Partitioning:**
Hierbei werden mehrere Systeme auf verschiedenen Kernen instantiiert und die Kommunikation findet über dedizierte Kanäle statt, vergleichbar mit dem IMA Konzept (vgl. Kapitel 3.2.1).
- **Big Kernel Lock:**
Ein einziger Semaphor regelt den Zugriff von Anwendungen auf den Kernel, im Falle von AUTOSAR auf das OS und die BSW.
- **AUTOSAR Ansatz:**
Systemaufrufe aus dem Legacy Code werden alleinig auf dem Hauptkern ausgeführt. Alle Aufrufe von anderen Kernen werden an den Hauptkern delegiert.
- **Fine Grained Locks:**
Die Systemdienste werden auf dem Kern ausgeführt, auf dem sie aufgerufen wurden. Das Level, auf welchem der wechselseitige Ausschluss geschieht, kann dabei grobkörnig auf Modulebene bis hin zur Neuentwicklung und feinkörnigen Semaphore sein.
- **Non-Blocking Synchronization:**
Hierbei findet quasi eine Neuentwicklung der Systemmodule unter Verwendung von nicht blockierenden Synchronisationsmechanismen statt, soweit dies im Detail umsetzbar ist.

In (Böhm et al. 2011) werden der Big Kernel Lock und der AUTOSAR Ansatz genauer betrachtet. Dazu wurde auf jedem Kern eine SW-C platziert. Die Ergebnisse zeigen, dass der Big Kernel Lock Ansatz eine bessere Systemperformance aufweist als der AUTOSAR Ansatz. Bei beiden Verfahren wird allerdings kein oder nur ein sehr geringer Leistungszuwachs gegenüber Singlecore-Systemen erreicht. Im Vergleich zur Untersuchung der Cross-Core Dienstaufrufe betrachten die dabei angeführten Ansätze BSW-Systemaufrufe, die in einer höheren Ebene angesiedelt sind als die OS-Dienstaufrufe. Je nach BSW-Systemaufruf verwenden diese selbst OS-Dienstaufrufe. Ferner finden die OS-Dienstaufrufe nicht nur Verwendung in der BSW, sondern auch für die Kommunikation auf Anwendungsebene.

4.7 Diskussion

In diesem Kapitel wurden zwei unterschiedliche Umsetzungen für Cross-Core Dienstauf-rufe auf einer AUTOSAR Multicore-Plattform miteinander verglichen. Die Ergebnisse zeigen, dass der Lock-basierte Ansatz auf der modellierten Multicore-Plattform besser abschneidet als das nachrichtenbasierte Verfahren. Ein Grund dafür ist der Mehraufwand für den Nachrichtenaustausch. Hierfür ließen sich noch weitere Techniken untersuchen, beispielsweise Lock-Free Synchronisationsmechanismen, die den Gewinn des Lock- gegenüber dem nachrichtenbasierten Ansatz u. U. schrumpfen ließen. Des Weiteren spielen die Speicherzugriffszeiten eine entscheidende Rolle. Im parMERASA Simulator sind die Zugriffszeiten von allen Kernen auf den Speicher identisch. Bei Multicore-Prozessoren mit unterschiedlichen Zugriffszeiten wäre ein Vorteil für den nachrichtenbasierten Ansatz denkbar. Angenommen jeder Kern besitzt einen geteilten Speicher, dessen Zugriffszeiten vom näher liegenden lokalen Kern eindeutig kürzer sind als von allen anderen Kernen. Somit würden sich die Ausführungszeiten beim Lock-basierten Verfahren durch die Speicherzugriffe auf den Zielkern erhöhen. Im Vergleich dazu benötigt der nachrichtenbasierte Ansatz nur wenige Zugriffe auf den geteilten Speicher, um seine Anfrage und Antwort zu übermitteln. Je nach Verhältnis der Anzahl an Speicherzugriffen zwischen Lock- und nachrichtenbasiertem Ansatz und der Speicherzugriffszeiten, kann das nachrichtenbasierte Verfahren besser abschneiden.

Die beiden vorgestellten Ansätze stellen zwei komplett gegensätzliche Realisierungen dar. Denkbar wären auch Lösungen, die beide Verfahren kombinieren. Ein weiterer Gesichtspunkt betrifft die *Freedom of Interference* ‘Rückwirkungsfreiheit’. Während beim nachrichtenbasierten Ansatz der Kernel des Zielkerns selbst die Änderungen im Laufe der Überprüfungs- und Bearbeitungsphase an den eigenen OS-Objektdaten vornimmt, greift beim Lock-basierten Ansatz der Kernel des Quellkerns in die OS-Objektdaten des Kerns vom Zielkern ein. Somit kann keine Rückwirkungsfreiheit zwischen den ansonsten unabhängigen Kernels mehr gewährleistet werden. Im parMERASA Projekt ging es in erster Linie darum, bei einer einzelnen Anwendung eine möglichst hohe Parallelität zu erreichen und diese auf mehrere Kerne zu verteilen. Daher kann davon ausgegangen werden, dass ein Fehler, ausgelöst durch einen beliebigen Kernel auf einem Kern, sich in einem Fehlverhalten der gesamten Anwendung manifestiert. Somit würde trotz Rückwirkungsfreiheit zwischen den Kernels ein Fehler in einem Kernel zu einem inkorrekten Verhalten der

Anwendung führen. Trotz alledem erschwert der Lock-basierte Ansatz die Sicherstellung der korrekten Arbeitsweise des Systems, da einzelne Programmteile, die auf verschiedenen Kernen ausgeführt werden, nicht isoliert betrachtet werden können. Ein Lösungsansatz war die Kombination beider Verfahren. Dabei wurde die Überprüfungsphase nach dem Lock-basierten Ansatz durchgeführt und die Bearbeitungsphase nach dem nachrichtenbasierten Ansatz. Der Vorteil besteht darin, dass der Zielkern nur unterbrochen wird, wenn alle Vorbedingungen erfüllt sind. Der Kernel des Quellkerns muss dazu nur lesend auf die OS-Objektdaten des Zielkerns zugreifen und nimmt keine Veränderungen vor. Dies gilt auch für alle `Get...(...)` Dienstaufrufe, da diese nur Lesezugriff benötigen. Allerdings lagen die Testergebnisse über denen des rein nachrichtenbasierten Ansatzes. Dies war darauf zurückzuführen, dass sowohl der Mehraufwand für den Nachrichtenaustausch als auch für den Mechanismus zum wechselseitigen Ausschluss die Ausführungszeiten erhöhten. Abschließend kann festgestellt werden, dass der Lock-basierte Ansatz je nach Plattform und Speicherzugriffszeiten einen nicht zu vernachlässigenden Leistungsgewinn bringen kann, dies allerdings auf Kosten der Rückwirkungsfreiheit einhergeht.

5

Verwendung von Echtzeit- Programmiermodellen im Automobilbereich

Die Aufgabe von Echtzeitsystemen ist es, vorgegebene Echtzeitanforderungen zu erfüllen. Die Anforderungen bestehen darin, dass das Ergebnis einer Berechnung nicht nur logisch korrekt ist, sondern auch zu einem bestimmten Zeitpunkt, sprich *rechtzeitig*, zur Verfügung steht. Zudem muss die Rechtzeitigkeit für mehrere nebenläufige Aufgaben *gleichzeitig* gewährleistet sein. Ein weiterer Punkt ist die Reaktion auf asynchrone Ereignisse innerhalb einer definierten Zeitspanne. Die meisten Echtzeitbetriebssysteme und Entwicklungswerkzeuge unterstützen das *Bounded Execution Time (BET)* Modell (Kirsch et al. 2007), welches auf Nebenläufigkeit und Einhaltung von Zeitbedingungen ausgerichtet ist. Die korrekte zeitliche Ausführung von nebenläufigen Aufgaben wird mit Hilfe von Echtzeitplanungsverfahren sichergestellt. Zwei der am häufigsten verwendeten Verfahren sind Earliest Deadline First (EDF) und Rate Monotonic Scheduling (RMS). Ein Nachteil des BET Modells ist, dass innerhalb der Periode eines Tasks Beginn und Ende einer Ausführung je Job schwanken können und somit auch die Eingabe- und Ausgabezeitpunkte. Dieses Problem adressiert das *Logical Execution Time (LET)* Modell (Henzinger et al. 2001). Hierbei sind die Zeitpunkte zum Lesen und Schreiben von Daten

zu Beginn bzw. am Ende der Periode und daher deterministisch. Somit findet auch die Kommunikation zwischen Tasks zu definierten Zeitpunkten statt. Dieser Umstand ermöglicht es, eine explizite Absicherung der Kommunikation durch wechselseitigen Ausschluss zu umgehen, indem die Datenkonsistenz anhand einer zeitlich vorhersagbaren Kommunikation sichergestellt wird. Nachteilig beim LET Modell ist, dass es sich bei den Daten, die zwischen Tasks ausgetauscht werden, u. U. nicht um die aktuellsten Werte handelt. Dies liegt darin begründet, dass aufgrund der früheren Eingabe- bzw. späteren Ausgabezeitpunkte im Vergleich zum BET Modell die aktuellen Werte noch nicht geschrieben sind und somit Werte der vorherigen Jobausführung konsumiert werden. Dies hat wiederum Auswirkungen auf die Ende-zu-Ende Zeiten von *Event Chains* ‘Wirkketten’, die aus einer Aneinanderreihung von verschiedenen Tasks bzw. Runnable Entitys (REs) und Kommunikationsphasen bestehen. Eine Möglichkeit das Datenalter zu minimieren ist in (Bradatsch et al. 2016) beschrieben und wird in diesem Kapitel genauer beleuchtet.

5.1 Task-Modell

Das in (Bradatsch et al. 2016) beschriebene Verfahren betrachtet synchrone Task-Sets, die aus periodischen und/oder sporadischen Echtzeit-Tasks mit einer minimalen *Inter Arrival Time* ‘Zwischenankunftszeit’ bestehen. Da in vielen eingebetteten Systemen und größtenteils in Systemen aus dem Automobilumfeld ein Prioritäts-Scheduling nach RMS zum Einsatz kommt, wird von impliziten Deadlines ausgegangen, sprich die Deadline entspricht der Periode. Abbildung 5.1 veranschaulicht einige typische Zeitparameter von Echtzeit-Tasks (Buttazzo 2011). Ein Task τ_i wird durch ein Tupel $\tau_i = (C_i, T_i)$ mit Worst-Case Execution Time (WCET) C_i und Periode T_i repräsentiert. Die relative Deadline eines Tasks ist gleich seiner Periode, d. h., dass jeder Job vor der nächsten Aktivierung desselben Tasks beendet sein muss. Alle Tasks in einem Task-Set $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_n\}$ erzeugen ihren ersten Job $\tau_{i,0}$ zum Zeitpunkt $t = 0$. Jobs $\tau_{i,k}$ mit $k \geq 0$ werden periodisch zu den *Release Times* $r_{i,k} = k \cdot T_i$ ‘Freigabezeitpunkte’ erzeugt und müssen ihre Ausführung bis zur absoluten Deadline $d_{i,k} = (k + 1)T_i$ beendet haben. Im Folgenden wird davon ausgegangen, dass das Task-Set \mathcal{T} planbar ist. Die *Start Time* $s_{i,k}$ ‘Startzeit’ eines Jobs $\tau_{i,k}$ ist der Zeitpunkt innerhalb des Intervalls $[r_{i,k}, d_{i,k}]$, in dem $\tau_{i,k}$ zum ersten Mal ausgeführt wird. Entsprechend liegt die *Completion Time* $c_{i,k}$ ‘Fertig-

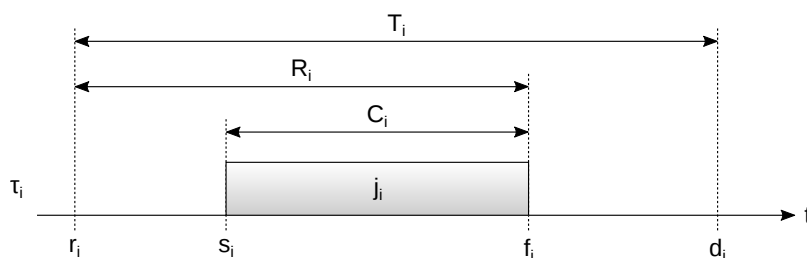


Abbildung 5.1: Typische Zeitparameter von Echtzeit-Tasks (Buttazzo 2011)

stellungszeitpunkt' eines Jobs $\tau_{i,k}$ zwischen $s_{i,k} < f_{i,k} \leq d_{i,k}$. Die *Response Time* $R_{i,k}$ 'Antwortzeit' eines Jobs ist die Zeit vom Freigabezeitpunkt bis zum Fertigstellungszeitpunkt, das bedeutet $R_{i,k} = f_{i,k} - r_{i,k}$. Im Zuge dessen ist die *Worst-Case Response Time (WCRT)* R_i 'Antwortzeit im schlimmsten Fall' eines Tasks τ_i das Maximum aller Antwortzeiten von dessen Jobs, sprich $R_i = \max_{k \geq 0} R_{i,k}$. Da davon ausgegangen wird, dass \mathcal{T} planbar ist, gilt $R_i \leq T_i$. Des Weiteren wird das kleinste gemeinsame Vielfache (kgV) aller Task-Perioden eines Task-Sets als Hyperperiode $T = \text{kgV}(T_1, T_2, \dots, T_n)$ bezeichnet. Ein Sonderfall dabei sind Task-Sets mit harmonischen Perioden, d. h. $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$, $\forall i, j \leq n, i < j, T_i < T_j, \exists k \in \mathbb{N} : kT_i = T_j$.

Neben den bisher definierten, allgemeinen Zeitparametern für Echtzeit-Tasks müssen im Zusammenhang mit dem Datenalter noch weitere Gesichtspunkte berücksichtigt werden. So lässt sich ein Task-Set \mathcal{T} in zwei Gruppen aufteilen, den *Producer Tasks* \mathcal{P} 'Erzeuger-Tasks' und *Consumer Tasks* \mathcal{C} 'Verbraucher-Tasks', für die $\mathcal{T} = \mathcal{P} \cup \mathcal{C}$ gilt. Ferner besteht eine Abhängigkeit zwischen Tasks der beiden Gruppen. Jeder Verbraucher-Task konsumiert Daten von mindestens einem Erzeuger-Task und maximal der Anzahl an Erzeuger-Tasks in einem Task-Set. Analog dazu werden die Daten eines Erzeuger-Tasks von mindestens einem Verbraucher-Task gelesen. Das Datenalter ist definiert als die Zeit zwischen dem Schreiben der Daten von einem Erzeuger-Job $\tau_{i,k}$ zum Zeitpunkt $t_{i,k}$ und dem Lesen der Daten von einem Verbraucher-Job $\tau_{j,l}$ zum Zeitpunkt $t_{j,l}$.

Definition 1. Das *Datenalter* $A_{i,k,j,l}$ quantifiziert die Aktualität der Daten von einer Kommunikation zwischen dem k -ten Job $\tau_{i,k}$ eines Erzeuger-Tasks τ_i und dem l -ten Job $\tau_{j,l}$ eines Verbraucher-Tasks τ_j wie folgt:

$$A_{i,k,j,l} = s_{j,l} - c_{i,k} \text{ mit } i \neq j; k, l \geq 0$$

Die tatsächlichen Indizes k und l hängen von der Kommunikationssemantik ab. Zudem ist das Lesen von Daten zum Zeitpunkt $t_{j,l}$ möglich, sofern Daten vorliegen, d. h. $t_{i,k} \leq t_{j,l}$. Andernfalls wird davon ausgegangen, dass die Daten mit Standardwerten initialisiert sind.

5.2 Grundlagen

Die Intention von eingebetteten Systemen ist es, auf Eingaben von ihrer Umgebung mit entsprechenden Ausgaben zu reagieren. Dabei werden Softwareprozesse durch Ereignisse von physikalischen Prozessen freigegeben, anschließend findet eine Verarbeitung im Softwareprozess statt und zum Schluss erhält der physikalische Prozess eine Rückmeldung. Ein Softwareprozess wird in sogenannter *Soft-Time* ‘Soft-Zeit’ ausgeführt, wohingegen der physikalische Prozess in *Real-Time* ‘Echtzeit’ vonstattengeht (Kirsch 2002). Die Berührungs- bzw. Synchronisationspunkte der beiden Zeitdomänen sind das Ereignis vom physikalischen Prozess und die Antwort vom Softwareprozess. Während Echtzeit kontinuierlich ist, trifft dies auf Soft-Zeit nicht zu. Ein Softwareprozess kann aufgrund verschiedenster Ereignisse unterbrochen werden, sei es durch IRQs oder Warten auf Freigabe eines Semaphor, um nur zwei Beispiele zu nennen. Daher hängt der Zusammenhang zwischen Soft- und Echtzeit von verschiedenen Faktoren ab, wie z. B. Hardwareleistung, Auslastung, Planungsverfahren etc. (Kirsch 2002). Abbildung 5.2 verdeutlicht den Zusammenhang zwischen Soft- und Echtzeit.

In einer physikalischen Umgebung laufen verschiedene Prozesse parallel ab. Daher ist eine Anforderung an die Echtzeitprogrammierung, ebenfalls mehrere Softwareprozesse nebenläufig auszuführen und darüber hinaus Softwareprozesse im Nachhinein verändern oder hinzufügen zu können und dabei Gleichzeitigkeit und Rechtzeitigkeit aller Softwareprozesse zu gewährleisten. Aufgrund dieser Anforderungen sind verschiedene Echtzeitprogrammiermodelle entstanden. Im Folgenden werden das Scheduled- und Timed-Modell erläutert. Weitere Programmiermodelle sind u. a. in (Kirsch 2002) nachzulesen.

Abbildung 5.2 veranschaulicht das Scheduled-Modell. Die Soft-Zeit umfasst hierbei die Zeit für eine geplante Aufgabe, vom Eintritt des Ereignisses bzw. dem Freigabezeitpunkt bis zur Antwort bzw. dem Fertigstellungszeitpunkt. Aufgrund von Unterbrechungen,

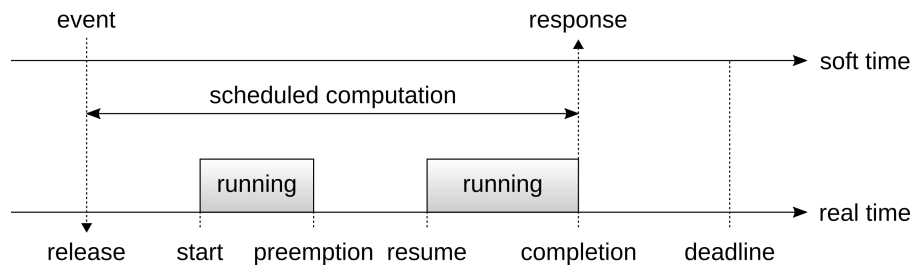


Abbildung 5.2: Echtzeitprogrammierung im Scheduled-Modell. Die benötigte Soft-Zeit zur Ausführung eines Tasks (Event – Response) ist kleiner gleich der eingeplanten Echtzeit (Release – Deadline).

variierenden Startzeitpunkten, etc. kann die Soft-Zeit schwanken. Die gleichzeitige Ausführung unterschiedlicher Tasks steuert ein Scheduler. Ferner signalisiert die Deadline die Schranke für die Rechtzeitigkeit und macht aus dem Scheduled-Modell ein Echtzeitprogrammiermodell. Durch eine Planbarkeitsanalyse, die eine WCET Analyse voraussetzt, wird sichergestellt, dass jeder Task seine Deadline einhält. Das bedeutet, solange jeder Task seine Aufgabe vor bzw. mit seiner Deadline fertigstellt, ist die Programmabarbeitung korrekt. Daraus folgt, dass die Soft-Zeit *kleiner gleich* der Echtzeit sein muss. Man spricht auch von Bounded Execution Time (BET) Programmierung, da die Zeitspanne zur Fertigstellung eines Tasks durch die Deadline *bounded* ‘begrenzt’ ist.

Beim Timed-Modell schließt die Soft-Zeit die Zeitspanne vom Eintritt des Ereignisses bzw. dem Freigabezeitpunkt bis zur Antwort bzw. der Deadline ein (siehe Abbildung 5.3). Somit ist die Soft-Zeit *gleich* der Echtzeit und fix, d. h., sie ist unabhängig von der tatsächlichen Task-Ausführungszeit. Anders ausgedrückt, besitzt ein Task eine Logical Execution Time (LET) ‘logische Ausführungszeit’. Diese erstreckt sich von einem *Release Event* ‘Freigabeereignis’ bis zu einem *Termination Event* ‘Beendigungsereignis’. Abbildung 5.3 und Abbildung 5.4 veranschaulichen die Verknüpfung zwischen logischer, Soft- und Echtzeit. Die logische Ausführung eines Tasks beginnt somit mit dem Freigabezeitpunkt und endet mit der Deadline. Zwischen diesen beiden Zeitpunkten wird der Task als *aktiv* angesehen. Daher entspricht die logische Ausführungszeit eines Tasks der Task-Periode. Die tatsächliche Task-Ausführung findet irgendwo dazwischen statt. Zudem werden sämtliche Eingaben zu Beginn, sprich beim Freigabeereignis, gelesen und alle Ausgaben zum Beendigungsereignis geschrieben, auch wenn der Task mit seiner Ausführung früher fertig ist. Da hierdurch die Datentransfers zu vorhersagbaren Zeitpunkten erfolgen, ist die

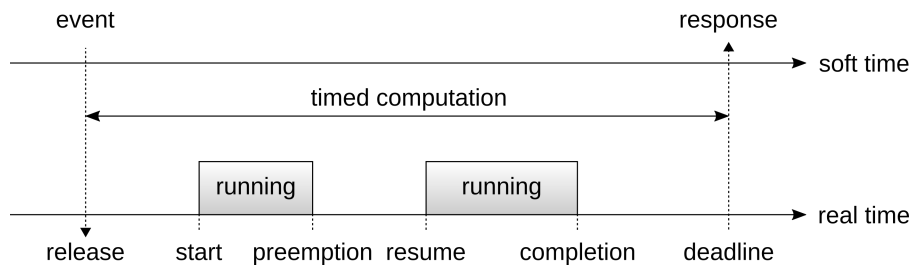


Abbildung 5.3: Echtzeitprogrammierung im Timed-Modell. Die benötigte Soft-Zeit zur Ausführung eines Tasks (Event – Response) ist gleich der eingeplanten Echtzeit (Release – Deadline).

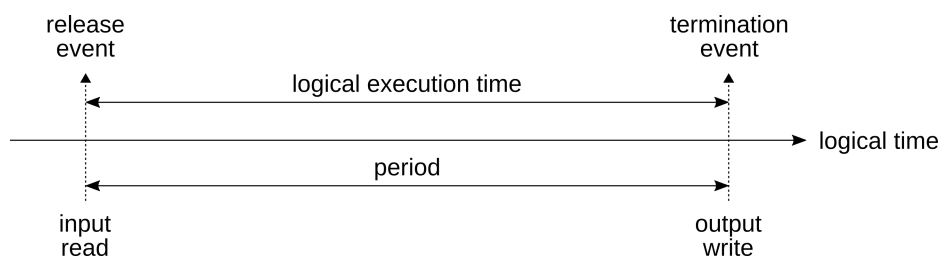


Abbildung 5.4: Ausführungszeitmodell mit Logical Execution Time (LET). Die Eingabedaten werden zu Beginn der Periode, nach dem Freigabezeitpunkt, gelesen und am Ende der Periode, vor der Deadline, geschrieben. Die logische Ausführungszeit entspricht der Task-Periode und somit ist die logische Zeit gleich der Soft- und Echtzeit (vgl. Abbildung 5.3).

LET Programmierung *zeit-deterministisch*. Solange der aktuelle Job eines Tasks logisch aktiv ist, sind nur die Ausgaben der vorherigen Jobausführung für alle anderen Tasks sichtbar. Erst mit dem Beendigungsereignis des aktuellen Jobs werden die Ausgabedaten aktualisiert. Im Gegensatz zum BET Modell ist das LET Modell *wert-deterministisch*. Beim LET Modell werden bei einem identischen Eingabeset die gleichen Ausgaben erzeugt, da sich die Eingaben, während der Job logisch aktiv ist, nicht verändern. Dem entgegen können beim BET Modell während der Task-Ausführungszeit Werte durch andere Tasks oder ISRs modifiziert werden und unterschiedliche Jobausführungen mit identischen Eingabesets zu differierenden Ausgaben führen. Während beim BET Modell die Task-Ausführungszeit u. a. von der zugrunde liegenden Hardwareplattform abhängt, ist dies beim LET Modell nicht der Fall. Solange alle Tasks eines Task-Sets ihre Eingabe, Verarbeitung und Ausgabe innerhalb der jeweiligen LET beenden, ist die Programmausführung wert- und zeit-deterministisch und unabhängig von der verwendeten

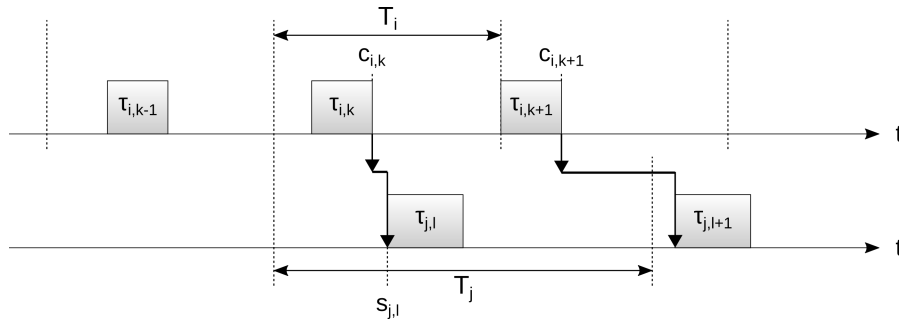


Abbildung 5.5: Exemplarische Ausführung zweier Tasks auf einem Singlecore-Prozessor. Die hervorgehobenen Pfeile markieren die beobachtbare Datenausgabe von Task τ_1 bzw. die Dateneingabe in Task τ_2 bei Verwendung des *BET Modells*.

Hardware. Um dies sicherzustellen, müssen analog zum BET Modell für die vorgesehene Hardwareplattform die WCETs der einzelnen Tasks bestimmbar sein.

5.2.1 Datenalter

Im vorherigen Abschnitt 5.1 wurde im Zuge des Task-Modells bereits Definition 1 für das Datenalter gegeben. Um das Datenalter bestimmen zu können, müssen für das BET und LET Modell unterschiedliche Voraussetzungen erfüllt sein. Im Falle der BET Programmierung wird die Eingabe eines Jobs $\tau_{i,k}$ zum Startzeitpunkt $s_{i,k}$ gelesen und die Ausgabe zum Fertigstellungszeitpunkt $c_{i,k}$ geschrieben. Abbildung 5.5 zeigt den Datenfluss zwischen zwei Tasks mit Perioden T_i und T_j im BET Modell. Erzeuger-Task τ_i schreibt Daten und Verbraucher-Task τ_j liest Daten. Die hervorgehobenen Linien und Pfeile kennzeichnen den Datenfluss zwischen Eingabe und Ausgabe.

Annahme 1. Um im BET Modell das Datenalter $A_{i,k,j,l}$ (vgl. Definition 1) zwischen dem Freigabezeitpunkt $c_{i,k}$ des zuletzt erzeugten Ausgabewertes des Erzeugerjobs $\tau_{i,k}$ und dem Startzeitpunkt $s_{j,l}$, an dem der Wert von Verbraucherjob $\tau_{j,l}$ gelesen wird, zu quantifizieren, muss $k(l)$ so gewählt werden, dass gilt:

$$c_{i,k} \leq s_{j,l} < c_{i,k+1}$$

Abbildung 5.6 zeigt den gleichen Auszug aus dem Task-Set wie in Abbildung 5.5, mit dem Unterschied, dass hierbei das BET Modell zugrunde liegt. Die Ausgabedaten jedes

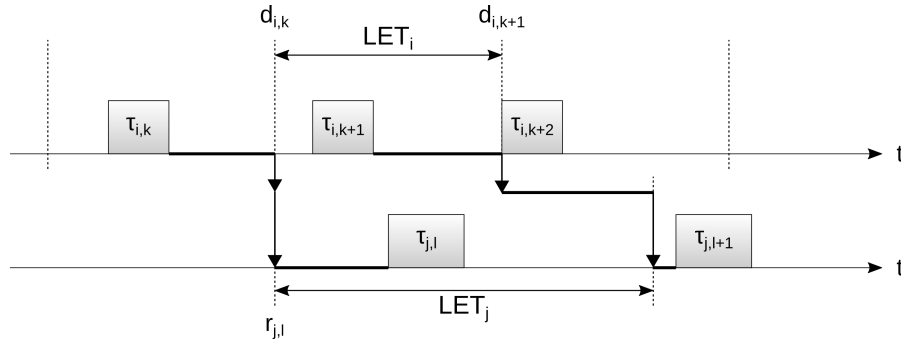


Abbildung 5.6: Exemplarische Ausführung zweier Tasks auf einem Singlecore-Prozessor. Die hervorgehobenen Pfeile markieren die beobachtbare Datenausgabe von Task τ_i bzw. die Dateneingabe in Task τ_j bei Verwendung des *LET Modells*.

Jobs von Erzeuger-Task τ_i sind jeweils am Ende der Task-Periode bzw. logischen Ausführungszeit LET_i von τ_i verfügbar und die Eingabedaten jedes Jobs von Verbraucher-Task τ_j werden zu Beginn dessen Periode gelesen. Im Beispiel von Abbildung 5.6 fallen der Schreibzeitpunkt von $\tau_{i,n-1}$ und der Lesezeitpunkt von $\tau_{j,n}$ zusammen, wohingegen $\tau_{j,n+1}$ die Daten von $\tau_{i,n}$ liest, da der Schreibzeitpunkt von $\tau_{i,n+1}$ an dessen Periodenende ist und somit hinter dem Lesezeitpunkt von $\tau_{j,n+1}$ liegt.

Annahme 2. Im LET Modell ist das Datenalter $A_{i,k,j,l}$ die Differenz zwischen dem Fertigstellungszeitpunkt $c_{i,k}$, an dem die Daten von Job $\tau_{i,k}$ tatsächlich erzeugt wurden, und dem Startzeitpunkt $s_{j,l}$, an dem die Daten von Job $\tau_{j,l}$ effektiv verbraucht werden. Um $A_{i,k,j,l}$ zu bestimmen, muss $k(l)$ so gewählt werden, dass der Freigabezeitpunkt $r_{j,l}$ von $\tau_{j,l}$ größer gleich der Deadline $d_{i,k}$ von $\tau_{i,k}$ und kleiner der Deadline $d_{i,k+1}$ des nachfolgenden Jobs $\tau_{i,k} + 1$ ist:

$$d_{i,k} \leq r_{j,l} < d_{i,k+1}$$

Im Beispiel von Abbildung 5.6 ist das Datenalter im Falle des LET Modells größer verglichen zum BET Modell. Das Datenalter zwischen Tasks eines Task-Sets, welches das LET Modell verwendet, ist offensichtlich im besten Fall gleich groß zu Task-Sets, denen das BET Modell zugrunde liegt. Grund dafür ist, dass beim BET Modell die Daten von den Erzeugerjobs unmittelbar bereitgestellt und direkt von den entsprechenden Verbraucherjobs gelesen werden. Eine Verringerung des Datenalters ist hierbei nur möglich, wenn sich Änderungen an den zeitlichen Parametern des Task-Sets ergeben. Das LET Modell verzögert die Ausgabe der Daten jedoch, wodurch sich das Datenalter erhöht.

Somit besitzt das BET Modell die geringsten Datenalter von den hier betrachteten Ausführungszeitmodellen.

5.3 Erweitertes LET Modell

Wie Eingangs bereits erwähnt, bringt die LET Programmierung in den meisten Fällen ein höheres Datenalter mit sich. Ziel des *erweiterten LET Modells* ist es, das Datenalter zu minimieren und zugleich das deterministische Verhalten zu bewahren. Hierfür wird das Beendigungsereignis des Erzeugerjobs so weit wie möglich an dessen Fertigstellungszeitpunkt angenähert. Als Konsequenz daraus sind die logische Ausführungszeit und Periode eines Tasks nicht mehr identisch. Der interessante Punkt hierbei ist, wie weit das jeweilige Beendigungsereignis an den Fertigstellungszeitpunkt eines jeden Tasks angenähert werden kann, um dennoch das deterministische Verhalten des LET Modells garantieren zu können. Im LET Modell wird die Datenausgabe eines Jobs erst beim Beendigungsereignis bzw. wenn die logische Ausführungszeit verstrichen ist für alle anderen Jobs sichtbar. Daher gibt es keine Schwankung bzw. Jitter bezüglich des Datenausgabezeitpunktes. Für das erweiterte LET Modell wurde deshalb eine neue logische Ausführungszeit gewählt, an deren Ende jeder Job eines Tasks seinen Fertigstellungszeitpunkt mit Sicherheit erreicht hat. Diese Anforderung erfüllt die Worst-Case Response Time (WCRT) eines Tasks. Somit ist die neue logische Ausführungszeit LET_i eines Tasks τ_i gleich der WCRT R_i . Um die WCRTs aller Tasks eines Task-Sets zu bestimmen, wird für jeden Task eine *Response Time Analysis (RTA)* ‘Antwortzeitanalyse’ durchgeführt (Audsley et al. 1991; Audsley et al. 1993). Aus diesem Grund wird das erweiterte LET Modell forthin als LET-RTA Modell bezeichnet.

Abbildung 5.7 zeigt das Task-Set aus Abbildung 5.6 unter Verwendung des LET-RTA Modells. Job $\tau_{i,k}$ wird hierbei als der Job mit der schlechtesten Antwortzeit R_i aller Jobs von Erzeuger-Task τ_i betrachtet. Daraus ergibt sich, dass $LET_i = R_i$. Die Periode T_i bleibt gleich. Aufgrund des nach vorne verlagerten Beendigungsereignisses des Erzeugerjobs $\tau_{i,k+2}$ zum Zeitpunkt $r_{i,k+2} + R_i$ liest Verbraucherjob $\tau_{j,l+1}$ die Daten von derselben Jobinstanz wie bei der BET Programmierung (vgl. Abbildung 5.5) und erhält somit aktuellere Daten im Vergleich zur Verwendung des LET Modells (vgl. Abbildung 5.6). Verbraucherjob $\tau_{j,l}$ kann jedoch nicht von dem nach vorne verlagerten Beendigungsereignis

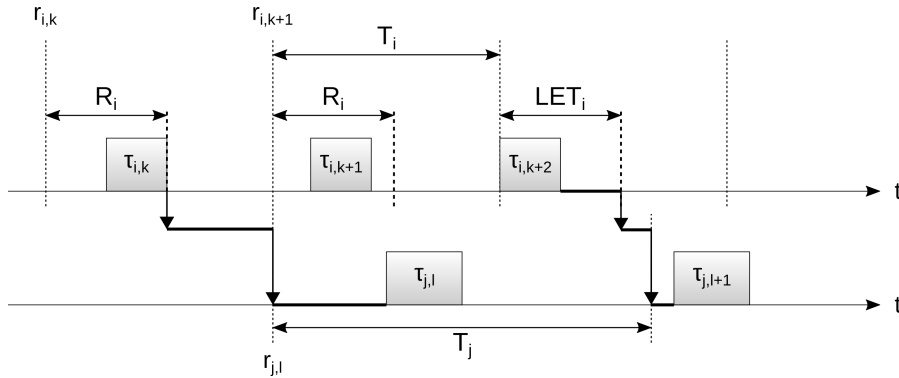


Abbildung 5.7: Exemplarische Ausführung zweier Tasks auf einem Singlecore-Prozessor. Die hervorgehobenen Pfeile markieren die beobachtbare Datenausgabe von Erzeuger-Task τ_i bzw. Dateneingabe zum Verbraucher-Task τ_j bei Verwendung des LET-RTA Modells. Die logische Ausführungszeit LET_i entspricht hierbei der WCRT R_i .

des Erzeugerjobs $\tau_{i,k+1}$ profitieren und bekommt die Daten von demselben Erzeugerjob $\tau_{i,k}$ wie bei der LET Programmierung. Somit lässt sich allgemein festhalten, dass durch den Einsatz des LET-RTA Modells das Datenalter zwischen beliebigen Erzeuger- und Verbraucherjobs verringert werden kann.

Annahme 3. Im LET-RTA Modell ist das Datenalter $A_{i,k,j,l}$ das Intervall zwischen dem Fertigstellungszeitpunkt $c_{i,k}$, an dem die Daten von Erzeugerjob $\tau_{i,k}$ tatsächlich generiert werden, und dem Startzeitpunkt $s_{j,l}$, an dem die Daten von Verbraucherjob $\tau_{j,l}$ effektiv gelesen werden. Um $A_{i,k,j,l}$ zu bestimmen, muss $k(l)$ so gewählt werden, dass der Freigabezeitpunkt $r_{j,l}$ von $\tau_{j,l}$ größer gleich der Antwortzeit $r_{i,k} + R_i$ von $\tau_{i,k}$ und kleiner der Antwortzeit $r_{i,k+1} + R_i$ des nachfolgenden Jobs $\tau_{i,k} + 1$ ist:

$$r_{i,k} + R_i \leq r_{j,l} < r_{i,k+1} + R_i$$

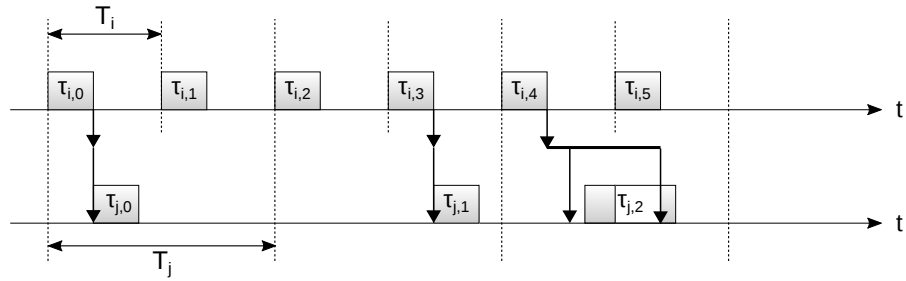
Ein Nachteil des LET-RTA Modells ist, dass bei jeder Änderung von Task-Parametern eines Task-Sets durch Softwaremodifikationen die WCRT neu bestimmt werden muss und somit auch die logische Ausführungszeit. Beim LET Modell hingegen ändern Modifikationen an der Software die logische Ausführungszeit nicht, da diese identisch mit der Periode ist. Allerdings muss sichergestellt werden, dass das Task-Set weiterhin planbar ist. Sofern diese Voraussetzung erfüllt ist, können Softwarekomponenten geändert, hinzugefügt oder

weggelassen werden. Man spricht hierbei auch von einer verbesserten *Softwarekombinierbarkeit* bei Verwendung des LET Modells. Im Gegensatz dazu können beim LET-RTA Modell Softwarekomponenten lediglich weggelassen werden, da sich dadurch die WCRT nicht vergrößert und somit die logische Ausführungszeit beibehalten werden kann.

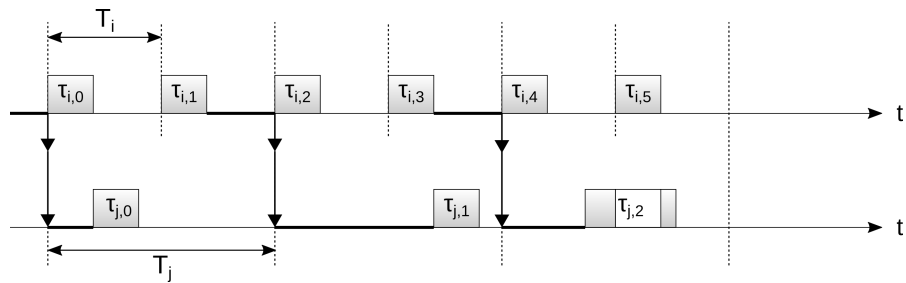
5.4 Wiederherstellung der Softwarekombinierbarkeit

Die bisherigen Betrachtungen gelten für beliebige Task-Sets. In eingebetteten Systemen kommen bisweilen auch Task-Sets mit harmonischen Task-Perioden vor, d. h., jede Periode ist ein ganzzahlig Vielfaches von der nächst kleineren Periode (siehe Abschnitt 5.1). Aufgrund der harmonischen Anordnung besitzen solche Task-Sets besondere Eigenschaften, die der Softwarekombinierbarkeit entgegen kommen. In Bezug auf den Datenaustausch zwischen Tasks mit unterschiedlichen, jedoch harmonischen Perioden kann eine Unterscheidung zwischen Kommunikation mit einer *fast-to-slow* und *slow-to-fast* Rate vorgenommen werden.

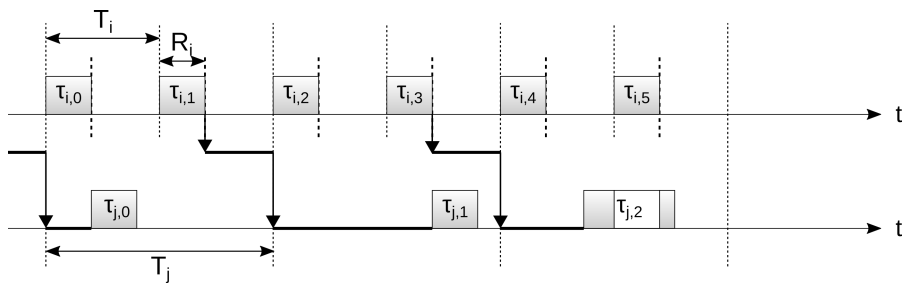
Für die folgenden Betrachtungen wird von Task-Sets mit FPP Scheduling und ratenmonotoner Zuweisung der Prioritäten ausgegangen. Eine Kommunikation mit einem fast-to-slow Verhältnis liegt vor, wenn ein höher priorisierter Erzeuger-Task Daten an einen niedriger priorisierten Verbraucher-Task übermittelt. Abbildung 5.8a stellt solch eine Kommunikation zwischen einem höher priorisierter Erzeuger-Task τ_i und einem niedriger priorisierten Verbraucher-Task τ_j unter Annahme des BET Modells dar. Da τ_i häufiger als τ_j ausgeführt wird, kann ein Verbraucherjob $\tau_{j,l}$ die Daten von einem beliebigen Erzeugerjob $\tau_{i,k}$ aus dem Intervall der Periode T_j erhalten ($\tau_{i,0} \rightarrow \tau_{j,0}$ bzw. $\tau_{i,3} \rightarrow \tau_{j,1}$). Ebenso kann es passieren, dass ein Verbraucherjob $\tau_{j,l}$ von einem Erzeugerjob $\tau_{i,k}$ unterbrochen wird und die Daten von $\tau_{j,l}$ zwischenzeitlich aktualisiert werden ($\tau_{i,4} \rightarrow \tau_{j,2}$, $\tau_{i,5} \rightarrow \tau_{j,2}$). Somit ändern sich die Eingabedaten während der Jobausführung ($\tau_{j,2}$) des niedriger priorisierten Tasks τ_j , das zu unvorhersehbaren Ergebnissen führt. Bei demselben Task-Set mit LET Programmierung erhält Verbraucherjob $\tau_{j,l}$ die Daten von dem Erzeugerjob $\tau_{i,k}$, der als letztes vor der Periode beendet wurde, in der $\tau_{j,l}$ ausgeführt wird (siehe Abbildung 5.8b: $\tau_{i,1} \rightarrow \tau_{j,1}$ bzw. $\tau_{i,3} \rightarrow \tau_{j,2}$). Da beim LET Modell die Eingabedaten zu Beginn der Periode gelesen werden und, solange der Verbraucherjob logisch aktiv ist, nicht verändert werden können, hat eine Aktualisierung der Daten des



(a)



(b)



(c)

Abbildung 5.8: Exemplarische Task-Ausführung in einer Multitasking-Umgebung mit harmonischen Task-Perioden. Die hervorgehobenen Pfeile symbolisieren den Datenfluss einer fast-to-slow Transition von einem höher priorisierten Task τ_i zu einem niedriger priorisierten Task τ_j unter Verwendung des (a) BET, (b) LET bzw. (c) LET-RTA Modells.

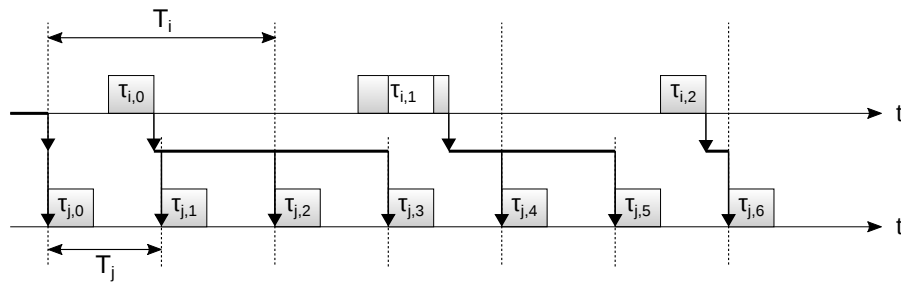
Erzeugerjobs keinen Effekt auf die Eingabedaten. Allerdings erhöht sich das Datenalter (vgl. Abbildung 5.8a: $\tau_{i,3} \rightarrow \tau_{j,1}$ mit Abbildung 5.8b: $\tau_{i,1} \rightarrow \tau_{j,1}$). Bezüglich des Datenalters und des Wert-Determinismus verhält sich das LET-RTA Modell identisch zum LET Modell (vgl. Abbildung 5.8b mit Abbildung 5.8c).

Theorem 1. In einem harmonischen Task-Set ist für sämtliche Kommunikation mit fast-to-slow Rate das Datenalter im LET-RTA Modell gleich dem im LET Modell.

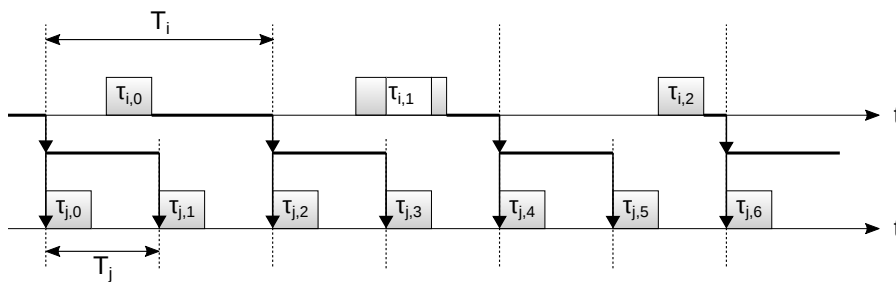
Beweis. Für alle Verbraucherjobs $\tau_{j,l}$ mit $l \neq 0$ existiert ein Wert k , so dass $\tau_{j,l}$ unter Berücksichtigung von Annahme 3 Daten von einem Erzeugerjob $\tau_{i,k}$ liest. Aufgrund der Tatsache, dass die Perioden T_i und T_j harmonisch sind und Erzeuger-Task τ_i eine höhere Aktivierungsrate als Verbraucher-Task τ_j hat, wird τ_i innerhalb der Periode T_j mindestens einmal aktiviert. Dies impliziert, dass der Freigabezeitpunkt $r_{j,l}$ gleich der Deadline $d_{i,k}$ ist. Somit ist auch Annahme 2 erfüllt. \square

In einem Task-Set mit harmonischen Task-Perioden, dem das LET-RTA Modell zugrunde liegt, und in dem jeder Erzeuger-Task τ_i Daten nur an Verbraucher-Tasks τ_j mit $T_i > T_j$ übermittelt, wird die logische Ausführungszeit LET_i gleich der Periode T_i gesetzt, da das Datenalter gleich bleibt.

Im Gegensatz dazu liegt eine Kommunikation mit slow-to-fast Verhältnis vor, wenn ein niedriger priorisierter Task Daten einem höher priorisierten Task bereitstellt. Abbildung 5.9a zeigt solch eine Kommunikation bei Verwendung der BET Programmierung zwischen einem niedriger priorisierten Erzeuger-Task τ_i und einem höher priorisierten Verbraucher-Task τ_j . Dabei ist deutlich erkennbar, dass je nach Fertigstellungszeitpunkt von Erzeugerjob $\tau_{i,k}$ die Daten von unterschiedlich vielen Verbraucherjobs $\tau_{j,l}$ gelesen werden. Daher ist ohne den genauen Ablaufplan nicht vorhersagbar, von welchem Erzeugerjob die Daten gelesen werden. Anders verhält es sich bei Verwendung des LET Modells. Hier lesen die Verbraucherjobs, die innerhalb der Periode von τ_i liegen, die Daten des letzten Erzeugerjobs aus der vorherigen Periode von τ_i (siehe Abbildung 5.9b: $\tau_{i,0} \rightarrow \tau_{j,2}$ und $\tau_{j,3}, \tau_{i,1} \rightarrow \tau_{j,4}$ und $\tau_{j,5}$, etc.). Beim Vergleich von Abbildung 5.9a mit Abbildung 5.9b kommt allerdings auch zum Vorschein, dass das Alter der Daten, die von $\tau_{j,1}$ gelesen werden, beim BET Modell kleiner ist als im LET Modell. Gleichermaßen wie bei der LET Programmierung verhält es sich bei der Verwendung des LET-RTA Modells, wie Abbildung 5.10a verdeutlicht. Aufgrund der schwankenden Fertigstellungszeitpunkte des



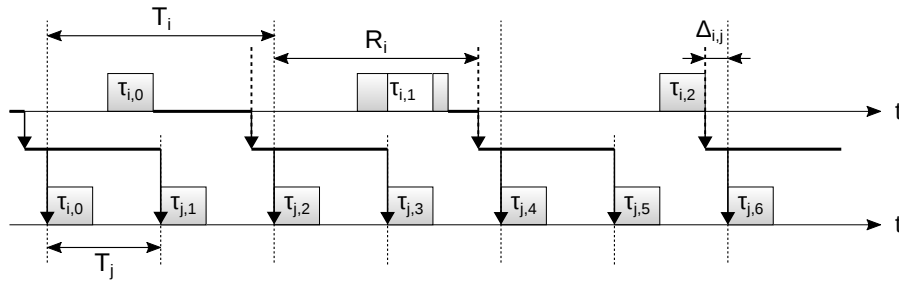
(a)



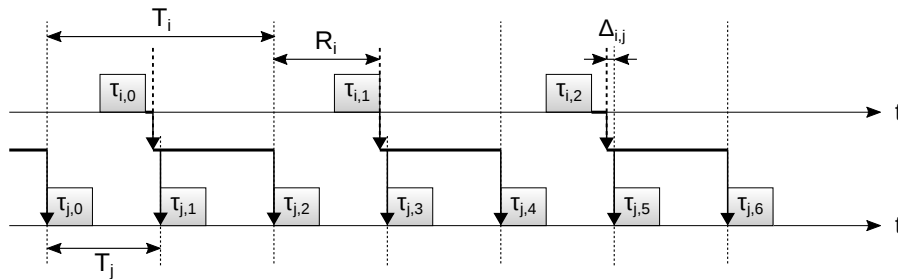
(b)

Abbildung 5.9: Exemplarische Task-Ausführung in einer Multitasking-Umgebung mit harmonischen Task-Perioden. Die hervorgehobenen Pfeile symbolisieren den Datenfluss einer slow-to-fast Transition von einem niedriger priorisierten Task τ_i zu einem höher priorisierten Task τ_j unter Verwendung des (a) BET bzw. (b) LET Modells.

Erzeuger-Tasks τ_i und der daraus resultierenden WCRT R_i , kann das Datenalter nicht minimiert werden. Anders verhält es sich, wenn R_i kleiner gleich der Periode T_j ist, wie Abbildung 5.10b zeigt. In diesem Fall profitieren $\tau_{j,1}$, $\tau_{j,3}$, $\tau_{j,5}$, etc. von einem geringeren Datenalter. Allerdings muss auch angemerkt werden, dass $\tau_{j,1}$ usw. die Daten von einer anderen, sprich der nachfolgenden Jobinstanz von Erzeuger-Task τ_i lesen. Abbildung 5.10 zeigt zudem eine Lücke $\Delta_{i,j}$ zwischen der WCRT R_i des Erzeuger-Tasks τ_i und dem Freigabezeitpunkt r_j des Verbraucher-Tasks τ_j , der unmittelbar nach τ_i freigegeben wird, auf. Für das Datenalter ist es unerheblich, ob der Erzeugerjob die Daten zum Zeitpunkt $r_{i,k} + R_i$ oder $r_{i,k} + R_i + \Delta_{i,j}$ schreibt, da sie der Verbraucherjob erst zum Zeitpunkt $r_{i,k} + R_i + \Delta_{i,j}$ liest. Die Idee ist, das Beendigungsereignis und somit das Ende der logischen Ausführungszeit auf diesen Zeitpunkt zu verlegen. Dazu wird das kürzeste $\Delta_{i,j}$ zwischen einem Erzeuger-Task τ_i und all seinen Verbraucher-Tasks τ_j ermittelt. Da



(a)



(b)

Abbildung 5.10: Exemplarische Task-Ausführung desselben Task-Sets wie in Abbildung 5.9 in einer Multitasking-Umgebung mit harmonischen Task-Perioden. (a) stellt die Ausführung desselben Task-Sets wie in Abbildung 5.9 dar mit dem Unterschied, dass hierbei das LET-RTA Modell zur Anwendung kommt. (b) zeigt das Task-Set in leicht abgewandelter Form mit kürzerer WCRT R_i und dem daraus resultierenden verminderten Datenalter.

nur Task-Sets mit FPP Scheduling betrachtet werden, genügt es, den Verbraucher-Task mit der höchsten Priorität zu untersuchen, da dieser vor allen anderen Verbraucher-Tasks ausgeführt wird. Die logische Ausführungszeit LET_i eines Erzeuger-Tasks τ_i wird so gewählt, dass gilt: $LET_i = R_i + \Delta_{i,j}$ mit $\Delta_{i,j} = r_{j,l} - (r_{i,k} + R_i)$ und l so, dass $r_{j,l-1} < (r_{i,k} + R_i) \leq r_{j,l}$. Daraus ergibt sich, dass $LET_i = r_{j,l} - r_{i,k}$.

Der Vorteil durch die zeitliche Verschiebung des Beendigungsereignisses nach hinten ist, dass Modifizierungen an der Software und damit Änderungen an den Task-Ausführungszeiten u. U. keine Auswirkungen auf die Zeitpunkte der Datenein- und -ausgabe bzw. das Datenalter haben. Dies erhöht einerseits die Softwarekombinierbarkeit und erleichtert andererseits die Testbarkeit, da die Datenein- und -ausgabe zeit- und wert-

deterministisch gegenüber Programmänderungen bleibt. Ungeachtet dessen muss nach einer Softwaremodifikation erneut eine Antwortzeitanalyse durchgeführt werden.

5.5 Evaluierung

Um zu zeigen, dass es möglich ist, das Datenalter mittels des LET-RTA Modells zu verringern und in gewissem Umfang an das Datenalter unter Verwendung des BET Modells anzuknüpfen, wurden die Datenalter aller drei Ausführungszeitmodelle anhand von Simulationen untersucht. Als Plattform für die Simulationen diente *tms-sim*, das ein C++ Framework für die Simulation und Evaluierung von Scheduling-Algorithmen bietet (Kluge 2014).

Wie in Abschnitt 5.1 bereits erwähnt, werden Task-Sets vorausgesetzt, die aus den zwei Task-Gruppen bestehen. Jedem Verbraucher-Task ist mindestens ein Erzeuger-Task zugeordnet. Jedes mal wenn ein Verbraucherjob freigegeben wird, liest dieser am Anfang seiner Ausführung die Daten, die von den vorigen Jobinstanzen der Erzeuger-Tasks geschrieben wurden und speichert die dazugehörigen Datenalter. Der Task-Set-Generator basiert auf dem ursprünglichen Generator, der in (Kluge et al. 2015) Verwendung fand, wurde jedoch insofern erweitert, dass er die Abhängigkeiten zwischen Erzeuger- und Verbraucher-Tasks erstellt. Für jedes Task-Set wird eine Auslastung U festgelegt, die in einem vorgegebenen Intervall $U \pm d_U$ liegt. U repräsentiert die angestrebte Auslastung und d_U die maximale Abweichung von U . Um aussagekräftige Ergebnisse zu erhalten, ist es ausreichend, die Task-Sets eine Hyperperiode lang zu simulieren. Sofern die Task-Perioden rein zufällig gewählt werden, kann die entsprechende Hyperperiode sehr lang werden, was wiederum die Simulationszeit drastisch erhöht. Aus diesem Grund wurde der Ansatz von Goossens und Macq (Goossens et al. 2001) im Simulator implementiert, um die Hyperperiode der Task-Sets zu limitieren. Somit können Simulationen in annehmbarer Zeit durchgeführt werden. Dabei wird angestrebt, die Perioden aller Tasks in einem Task-Set vom Generator so wählen zu lassen, dass ihr kleinstes gemeinsames Vielfaches beschränkt ist. Task-Sets, deren Auslastung außerhalb von $U \pm d_U$ liegt, werden sofort verworfen. Die Planbarkeit eines Task-Sets wird durch eine RTA sichergestellt. Sollte der Planbarkeitstest fehlschlagen, wird das entsprechende Task-Set ebenfalls verworfen. Neben den Voraussetzungen bei der Task-Set-Generierung existiert zusätzlich zwischen

zwei beliebigen Tasks aus den beiden verschiedenen Task-Gruppen eine Abhängigkeit. Jeder Verbraucher-Task verarbeitet die Daten von mindestens einem Erzeuger-Task und maximal von der Anzahl an Erzeuger-Tasks im Task-Set. Die Daten, die von einem Erzeuger-Task bereitgestellt werden, werden wiederum von mindestens einem Verbraucher-Task gelesen.

Aus den mitprotokollierten Datenalter einer Kommunikation zwischen einem Erzeuger-Task τ_i und Verbraucher-Task τ_j lassen sich folgende Kennzahlen ableiten:

- Das Minimum gibt das kürzeste, vorkommende Datenalter zwischen zwei Jobs von τ_i und τ_j an.
- Analog dazu gibt das Maximum das längste vorkommende Datenalter zwischen zwei Jobs von τ_i und τ_j an.
- Der Durchschnitt berechnet sich aus den Datenalter aller Jobs von τ_j mit den interagierenden Jobs von τ_i innerhalb einer Hyperperiode.

In erster Linie sind der Durchschnitt und das Maximum von Interesse, da diese Kennzahlen die durchschnittliche und maximale Latenz angeben und im Rahmen von eingebetteten Echtzeitsystemen von Bedeutung sind.

5.5.1 Zufällig generierte Task-Sets

Für die Evaluierung wurden 100 Task-Sets untersucht, die den Prozessor fast vollständig belegten und eine Auslastung von $U = 0,9$ hatten, wobei davon $U_p = 0,3$ auf die Erzeuger- und $U_c = 0,6$ auf die Verbraucher-Tasks fielen. Die Anzahl der Erzeuger- und Verbraucher-Tasks wurde gleichmäßig zu $n_p = n_c = 5$ verteilt. Dies spiegelt ein kleines bis mittelgroßes Task-Set in einem eingebetteten System wieder. Die Task-Perioden wurden aus dem Intervall $[7, 1664863200]$ gewählt.

Für jedes Task-Set werden das durchschnittliche und maximale Datenalter bei Verwendung des BET, LET und LET-RTA Modells zwischen zugehörigen Erzeuger- und Verbraucher-Tasks aufgezeichnet. Abbildung 5.11 vergleicht die Mittelwerte der durchschnittlichen und maximalen Datenalter von den interagierenden Erzeuger- und Verbraucher-Tasks aller 100 Task-Sets. Die Ergebnisse sind in simulierten *Zeitschritten* angegeben. Die

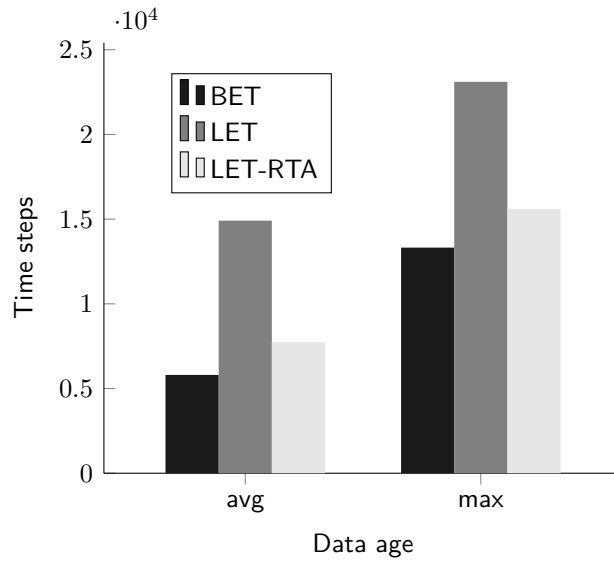


Abbildung 5.11: Mittelwerte der durchschnittlichen und maximalen Datenalter der interagierenden Tasks aller Task-Sets

gewonnenen Werte für die durchschnittlichen und maximalen Datenalter beim BET, LET und LET-RTA Modell betragen 5804, 14912 und 7731 bzw. 13319, 23109 und 15594. Die Ergebnisse bestätigen, dass das durchschnittliche bzw. maximale Datenalter mittels des LET-RTA Modells gegenüber dem LET Modell um $\approx 48,2\%$ respektive $\approx 32,5\%$ gesenkt werden konnte. Zugleich ist das durchschnittliche bzw. maximale Datenalter beim LET-RTA Modell um $\approx 33,2\%$ respektive $\approx 17,1\%$ schlechter als beim BET Modell. Wie aus Abbildung 5.11 ersichtlich, konnten sowohl das durchschnittliche als auch maximale Datenalter fast auf die Werte bei Verwendung des BET Modells verringert werden.

Dennoch existieren in fast jedem Task-Set Ausreißer, bei denen das Datenalter zwischen Erzeuger- und Verbraucher-Task beim LET-RTA Modell fast so hoch ist wie beim LET Modell. Abbildung 5.12 veranschaulicht diesen Sachverhalt. Bei Verwendung des BET Modells erhält Verbraucherjob $\tau_{j,0}$ die Daten von Erzeugerjob $\tau_{i,4}$. Im Gegensatz dazu bekommt $\tau_{j,0}$ bei Benutzung des LET-RTA Modells die Daten von Erzeugerjob $\tau_{i,0}$, da dessen Daten zu Beginn der Periode T_j bereitstehen. Da die Periode T_j des Verbraucher-Tasks relativ lang ist und somit nach raten-monotoner Verteilung eine niedrige Priorität erhält, beginnt dessen Ausführung sehr spät, die Daten werden jedoch zu Beginn von T_j gelesen. Dadurch ist das Datenalter beim LET-RTA Modell ($A_{\text{LET-RTA}}$) deutlich höher als beim BET Modell (A_{BET}).

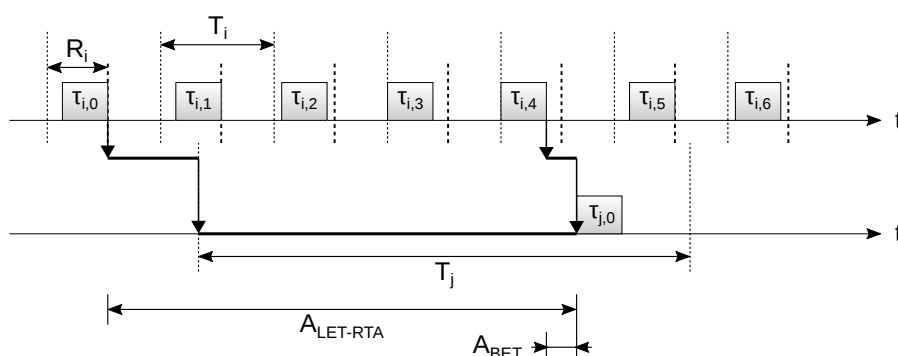


Abbildung 5.12: Vergleich des Datenalters bei Verwendung des BET Modells (A_{BET}) und LET-RTA Modells ($A_{LET-RTA}$) bei einem Task-Set mit kurzer Erzeuger-Task-Periode T_i und langer Verbraucher-Task-Periode T_j

5.5.2 Automobil-Benchmark

Um die Ergebnisse, die durch die Simulation von zufällig generierten Task-Sets gewonnen wurden, zu verifizieren, wurde das LET-RTA Modell auf einen praxisnahen Automobil-Benchmark angewandt, der von realen Motormanagementsystemen abgeleitet ist. In (Kramer et al. 2015) werden die Charakteristika des Benchmarks genau beschrieben, dessen Vorgaben das gewählte Task-Set entspricht. Tabelle 5.1 zeigt die relevanten Parameter des Task-Sets. Die Task-Perioden sind hierbei pseudoharmonisch, d. h., dass eine Division der meisten Perioden durch jede kleinere Periode jeweils eine natürliche Zahl ergibt. Allerdings existieren auch Perioden, die nicht durch alle kleineren Perioden teilbar sind, so z. B., der 5 ms bzw. 50 ms Task durch den 2 ms bzw. 20 ms Task. Die Auslastung des Task-Sets beträgt 82,36 %. Für den Kurbelwellen-synchronen Task hat die Periode eine Länge von 4 ms. Dies entspricht einem vierzylindrigen Motor mit maximal 7500 U/Min. Die Kommunikationsmatrix, die die Abhängigkeiten der einzelnen Tasks bezüglich des Datenaustauschs beschreibt, ist bis auf eine Ausnahme identisch mit der in (Kramer et al. 2015) dargestellten Matrix. Der Unterschied besteht darin, dass der Kurbelwellen-synchrone Task mit keinem anderen Task Daten austauscht. Dies liegt

Tabelle 5.1: Task-Perioden (T_i) und WCRTs (C_i) in Millisekunden des Automobil-Benchmarks

T_i	1	2	5	10	20	50	100	200	1000	4
C_i	0,150	0,084	0,220	2,522	2,184	0,526	2,106	0,026	0,018	0,778

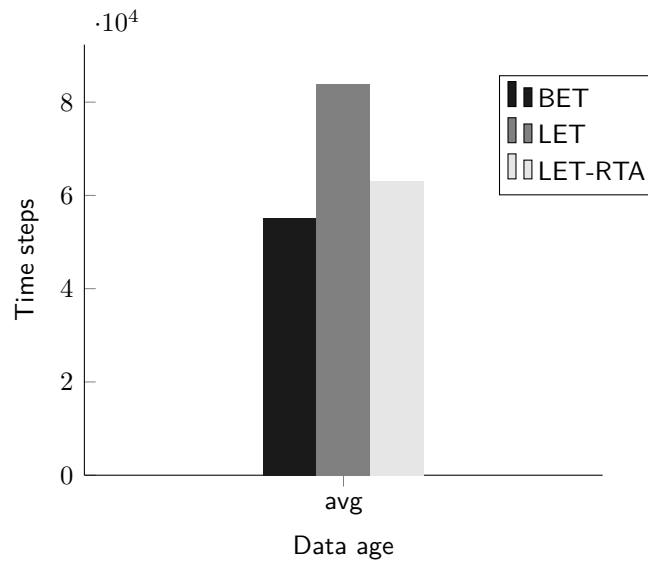


Abbildung 5.13: Mittelwerte der durchschnittlichen Datenalter aller interagierender Tasks aus dem Automobil-Benchmark

darin begründet, dass die Periode des kurbelwellensynchronen Tasks je nach Anzahl an Umdrehungen pro Minute variiert und in diesem Fall die Daten nach dem BET Prinzip, sprich sofort nach Verfügbarkeit, verarbeitet werden. Um die Unterbrechungen aller Tasks durch den Kurbelwellen-synchronen Task zu berücksichtigen, ist dieser trotzdem im Task-Set enthalten.

Die Simulationsergebnisse sind in Abbildung 5.13 dargestellt. Die durchschnittlichen Datenalter aller Tasks des Automobil-Benchmarks betragen bei Verwendung des BET, LET und LET-RTA Modells 55182, 83931 und 63089 Zeitschritte. Bei Verwendung des LET-RTA Modells anstatt des LET Modells ist das Datenalter um $\approx 24,8\%$ geringer. Im Vergleich zum BET Modell ist es allerdings um $\approx 14,3\%$ schlechter.

Aufgrund der pseudoharmonischen Task-Perioden wurden ebenso die Zwischenzeiten $\Delta_{i,j}$ zwischen den Erzeuger-Tasks τ_i und Verbraucher-Tasks τ_j ermittelt (siehe Abschnitt 5.4). Hierfür werden für einen Erzeuger-Task τ_i und allen Verbraucher-Tasks τ_j , denen τ_i Daten bereitstellt, alle $\Delta_{i,j}$ berechnet und das kleinste $\Delta_{i,j}$ als Zwischenzeit Δ_i des Erzeuger-Tasks τ_i bestimmt. Dadurch ist sichergestellt, dass alle τ_j von derselben Jobinstanz ihre Daten erhalten, wie ohne einer Anpassung mit Δ_i . Tabelle 5.2 vergleicht die logischen Ausführungszeiten eines jeden Tasks unter Verwendung des LET Modells ($LET_i = T_i$),

des LET-RTA Modells ($LET - RTA_i = R_i$) und des LET-RTA Modells mit verlängerter Antwortzeit ($LET_i = R_i + \Delta_i$). Aus Tabelle 5.2 ist ersichtlich, dass für die 1 ms und 2 ms Tasks die logische Ausführungszeit beim LET-RTA Modell gleich der beim LET Modell gesetzt werden kann, ohne dass sich das durchschnittliche Datenalter verändert. Für alle anderen Tasks kann die logische Ausführungszeit ebenfalls verlängert werden, ohne Auswirkungen auf das Datenalter zu haben. Dadurch kann die Softwarekombinierbarkeit geringfügig gesteigert werden.

5.6 Angrenzende Arbeiten

In Abschnitt 5.3 wurden bereits die Unterschiede zwischen dem LET-RTA Modell und dem ursprünglichen LET Modell aus Giotto differenziert (Henzinger et al. 2001). Beide Modelle teilen die gemeinsame Idee der logische Ausführungszeit. Während Giotto eine vollständige Programmiersprache inklusive Compiler und Laufzeitumgebung beschreibt, konzentriert sich das LET-RTA Modell auf eine wert- und zeit-deterministische Datenein- und -ausgabe und die Aktualität der Daten. Eine Erweiterung von Giotto ist *xGiotto*, das nicht nur Tasks unterstützt, die zu bestimmten Zeitpunkten freigegeben werden, sondern auch durch Ereignisse ausgelöst werden. Das LET-RTA Modell erlaubt zwar sporadische, per Ereignis ausgelöste Tasks, für die eine minimale Zwischenankunftszeit bestimmbar ist, unterstützt jedoch keine Analyse des Datenalters für diese Tasks.

Ein weiterer Nachfolger von Giotto ist ein Komponentenmodell für Echtzeitsysteme, genannt *Timing Definition Language (TDL)* (Farcas et al. 2005). Giotto ist in erster Linie ein abstraktes mathematisches Modell. Die TDL erweitert dieses Modell um den Begriff *Module* ‘Modul’ und ein Komponentenmodell. Module sind Giotto Programme,

Tabelle 5.2: Logische Task-Ausführungszeiten in Millisekunden des Automobil-Benchmarks bei Verwendung des LET (T_i), LET-RTA (R_i) und LET-RTA Modells mit verlängerter Antwortzeit ($R_i + \Delta_i$)

T_i	1	2	5	10	20	50	100	200	1000
R_i	0,928	1,162	1,382	5,670	9,400	9,926	18,320	18,346	18,364
$R_i + \Delta_i$	1	2	2	6	10	10	20	20	20

die exportiert und wiederum in anderen Modulen verwendet werden können. Somit ist u. a. eine einfache Mehrfachverwendung von Modulen möglich und die Kombinierbarkeit von Modulen wird ebenfalls erleichtert. Ferner bleibt die logische Ausführungszeit eines Moduls erhalten. Auf TDL aufsetzend existiert eine weiterführende Sprache, die in (Kloda et al. 2014a) erläutert ist und den Namen *Extended Timing Definition Language (E-TDL)* trägt. Im Vergleich zu TDL wird hierbei die logische Ausführungszeit von der Periode entkoppelt. Das bedeutet, dass das Freigabe- und Beendigungsereignis nicht länger an Start bzw. Ende einer Periode gebunden sind. Stattdessen wird das Freigabeereignis um einen Zeitabstand ϕ verschoben, das ein späteres Lesen von Daten erlaubt. Analog dazu werden Ausgabedaten zum Beendigungsereignis geschrieben, welches vor oder zur Deadline sein kann. In (Kloda et al. 2014b) wird die Anwendbarkeit der E-TDL für Task-Sets mit EDF Scheduling demonstriert. Die E-TDL hat mit dem LET-RTA Modell gemein, dass bei beiden Ansätzen die logische Ausführungszeit ungleich der Periode ist und somit von dieser entkoppelt wird. Die E-TDL geht noch einen Schritt weiter und entkoppelt darüber hinaus das Freigabeereignis vom Periodenbeginn. Die Intention dahinter ist, *explizite* Zeiträume vor und nachdem ein Task logisch aktiv ist zu schaffen, um zeitintensive Kommunikationsaufgaben mit Aktuatoren, Sensoren oder anderen Tasks zu bewerkstelligen. Die Aktualität der Daten zwischen kommunizierenden Tasks steht hierbei nicht im Fokus. Zudem ist die E-TDL momentan auf Task-Sets mit EDF Scheduling beschränkt.

In (Kirsch et al. 2012) geben die Autoren einen Überblick über angrenzende und weiterführende Arbeiten zu Giotto. Unter anderem wird ein weiteres Echtzeitprogrammiermodell erläutert, mit dem Namen *Synchronous Model* (Kirsch et al. 2007), oder auch *Synchronous Reactive Programming* (Halbwachs 1993; Potop-Butucaru et al. 2005). Dabei wird angenommen, dass sämtliche Berechnungsaufgaben inklusive Kommunikation keine Zeit benötigen. Die Soft-Zeit ist hierbei gleich Null (vgl. Abschnitt 5.2). Analog zu den anderen Echtzeitprogrammiermodellen werden die Berechnungen periodisch durch *Events* ‘Ereignisse’ angestoßen. Eine physikalische Ausführung eines synchronen Programms ist korrekt, wenn die Eingabedaten unmittelbar nach einem Ereignis gelesen werden und die Berechnung und Ausgabe der Daten vor dem Eintreffen des nächsten Ereignisses beendet sind. Da die Daten unmittelbar nach der Berechnung ausgegeben werden, unterliegen diese einem gewissen Jitter, je nach Ausführungszeit der Jobinstanz. Jedoch ist im synchronen Modell inbegriffen, dass sämtliche Ausgaben eines Berechnungsschrittes vor dem

nachfolgenden Ereignis erledigt sind und somit trotz zeitlicher Variation die Ausgaben wert-deterministisch sind. Die Herausforderung liegt hierbei beim Compiler, dass dieser nicht nur den Code generiert, sondern auch die Rechtzeitigkeit der Ausgaben in jedem Schritt prüft. Hierfür muss die WCET des Programmcodes ermittelt werden.

Ein weiterer Ansatz, genannt OASIS Modell (Aussaguès et al. 1998; David et al. 1998), beschäftigt sich ebenfalls mit deterministischen und vorhersagbaren Verhalten in sicherheitskritischen, eingebetteten Systemen. Ähnlich zu Giotto beschreibt OASIS ein formales Modell in Kombination mit einer Spracherweiterung für C, einem entsprechenden Compiler und einer Laufzeitumgebung. Dabei basiert OASIS auf einem zeitgetriebenen Programmiermodell. Die Berechnung bzw. Ausführung eines Tasks, bei OASIS *Agent* genannt, findet zwischen einem frühesten Startzeitpunkt und einem spätesten Endzeitpunkt statt und unterscheidet sich dadurch vom Synchronen Modell. Vielmehr zählt OASIS zur Kategorie der Timed-Modelle. Jeder Agent ω besitzt seine eigene Echtzeituhr H_ω , die die Zeitpunkte beobachtbarer Ein- und Ausgaben von ω beschreibt. Die System-Echtzeituhr H_Ω wiederum vereint alle H_ω s. Da bei den H_ω s nur beobachtbare Ereignisse enthalten sind, kann das Intervall zwischen aufeinander folgenden Ereignissen variieren. Aus diesem Grund wird für H_Ω ein festes Intervall für zwei aufeinander folgende Zeitpunkte so festgelegt, dass alle H_ω s wie folgt ausgedrückt werden können: $H_\omega = K_\omega \cdot H_\Omega + \Delta_\omega$ mit einem konstanten Faktor K_ω und einem Offset Δ_ω . Mit Hilfe der *Advance (ADV)* Instruktion können die beobachtbaren Ereignisse einer Echtzeituhr H_ω beeinflusst werden. Die Instruktion $ADV(k)$ besagt zum einen, dass der Code vor dieser Instruktion bis zum relativen Zeitpunkt k fertig abgearbeitet sein muss und zum anderen, dass der nachfolgende Code erst ab Zeitpunkt k mit der Ausführung beginnen kann. Ferner bedeutet der Aufruf der ADV Instruktion ein beobachtbares Ereignis zu diesem Zeitpunkt. Das heißt, dass zu diesem Zeitpunkt von dem entsprechenden Agenten Eingaben gelesen und Ausgaben geschrieben werden. Dies entspricht von der Idee dem LET Modell. Allerdings gibt es einen entscheidenden Unterschied. Im Allgemeinen gibt ein Task fixe Jobs frei, spricht mit einer festen Ausführungszeit und relativen Deadline. Bei OASIS dagegen ist ein Agent ein Job, der sich dynamisch ändert, z. B.:

```
Processing_A; ADV(4); Processing_B; ADV(6); Processing_A; ...
```

Hierbei hat der Agent (Task) für die Ausführung von *Processing_A* 4 Zeiteinheiten und im Anschluss für *Processing_B* 6 Zeiteinheiten zeit. Danach wiederholt sich die Ausführungs-

reihenfolge. Mit dem OASIS Modell ist es möglich, das LET Modell nachzubilden, jedoch nicht umgekehrt. Die ADV Instruktion bildet den Freigabezeitpunkt und die Deadline eines Agenten, bei deren Zeitpunkt die Ein- und Ausgaben gelesen bzw. geschrieben werden. Beim LET Modell werden die Eingaben ebenfalls zum Freigabezeitpunkt, dem Beginn der Periode, gelesen und die Ausgaben zur Deadline bzw. dem Periodenende geschrieben. Neben der impliziten Kommunikation zwischen Agenten durch Ein- und Ausgaben besteht die Möglichkeit, explizite Nachrichten zu senden. Eine Besonderheit dabei ist, dass der Sender einen Zeitpunkt angeben kann, ab dem der Inhalt der Nachricht für den Empfänger erst sichtbar bzw. lesbar ist. Das OASIS Modell wurde später noch für die Verwendung von Multicore-Prozessoren angepasst (Aussaguès et al. 2009) und mit PharOS wurden auch Untersuchungen für die Eignung im Automobilbereich durchgeführt (Aussaguès et al. 2010). Im Rahmen des parMERASA Projekts wurde bei der Parallelisierung der Motorsteuerungssoftware ebenfalls das OASIS Konzept der ADV Instruktion aufgegriffen. Darüber hinaus wurde die ADV Instruktion erweitert, um einen synchronen Start von parallelisierten ISRs zu ermöglichen (Bradatsch et al. 2014b).

5.7 Diskurs

In diesem Kapitel wurde das Logical Execution Time (LET) Modell erläutert und gegenüber anderen Programmiermodellen abgegrenzt. Die Vorteile für eingebettete Systeme bei Anwendung der LET Programmierung liegen in erster Linie in einem zeit- und wertdeterministischem Verhalten, wodurch die Portabilität, Testbarkeit und Softwarekombinierbarkeit erhöht werden. Des Weiteren findet die Kommunikation zwischen Tasks zu definierten Zeitpunkten statt, wodurch bei Manycore-Plattformen eine explizite Synchronisation durch wechselseitigen Ausschluss entfallen kann. Nachteilig dagegen ist die Tatsache, dass der Zeitdeterminismus eine Verzögerung der Ausgabedaten zur Folge hat. Dadurch verschlechtert sich die Aktualität der Daten. Dies kann u. a. negative Auswirkungen auf die Regelgüte bei digitalen Reglern haben. Im Automobilbereich können sich die Ende-zu-Ende Latenzen von Wirkketten erhöhen und dadurch Echtzeitbedingungen verletzt werden. Hier setzt das LET-RTA Modell an, indem es den Zeitpunkt zur Bereitstellung der Daten vorverlegt. Um weiterhin vorhersagbare Datenausgabezeitpunkte zu garantieren, wird eine neue logische Ausführungszeit definiert, in der ein Task logisch

aktiv ist. Zur Ermittlung der logische Ausführungszeit wird eine Antwortzeitanalyse durchgeführt. Anhand der Simulationsergebnisse, sowohl bei den zufällig generierten Task-Sets als auch beim praxisnahen Automobil-Benchmark, ist zu erkennen, dass das Datenalter von miteinander kommunizierenden Tasks durch das LET-RTA Modell reduziert werden konnte.

Das LET Modell erlaubt den Austausch der Hardwareplattform, sofern die Rechenleistung ausreicht, um alle Task-Berechnungen innerhalb der logische Ausführungszeit durchzuführen. Da das LET-RTA Modell die logische Ausführungszeit minimiert, erfordern minimale Änderungen an Softwarekomponenten oder am Task-Set eine Neuberechnung der WCRT. Für Task-Sets mit (pseudo)harmonischen Task-Perioden wurde eine Möglichkeit aufgezeigt, die logische Ausführungszeit beim LET-RTA Modell auszuweiten, ohne das Datenalter zu erhöhen. Somit führen kleine Modifikationen an der Software nicht zwangsweise zu einer Neuberechnung der WCRT. Dadurch wird die Softwarekombinierbarkeit verbessert und die Testbarkeit erleichtert. Um das Datenalter noch weiter zu reduzieren, insbesondere bei den Ausreißern eines Task-Sets (siehe Abbildung 5.12), wäre eine Verschiebung des Freigabeereignisses bzw. Lesezeitpunktes der Daten zu erwägen.

6

Zusammenfassung und Ausblick

Dieses Kapitel fasst die entwickelten Modelle und umgesetzten Techniken dieser Arbeit nochmals zusammen. Im Anschluss folgt ein kurzer Ausblick auf weiterführende Forschungsthemen, die im Rahmen dieser Arbeit nicht näher betrachtet wurden.

6.1 Zusammenfassung

Multicore-Prozessoren sind heutzutage nicht nur auf Desktop- und Server-Systeme beschränkt, sondern ebenso in eingebetteten Systemen etabliert. Die meisten Industriezweige haben bereits Ansätze und Lösungen für die Unterstützung von Multicore-Plattformen. Dennoch existieren zu lösende Herausforderungen in diesem Bereich, da viele eingebettete Systeme im industriellen Umfeld im Gegensatz zu Universalcomputern neben funktionalen auch nicht-funktionale Anforderungen besitzen, beispielsweise Echtzeitbedingungen. Erschwerend kommt noch die wachsende Softwarekomplexität aufgrund zunehmenden Funktionsumfangs hinzu. Eine weitere Bestrebung ist die Portierung von Software, die über viele Jahre kontinuierlich gewachsen ist und ständig weiterentwickelt wurde, auf Multicore-Plattformen.

Ziel der Arbeit ist es, hierfür eine Entwicklungsplattform zur Verfügung zu stellen, die es ermöglicht, parallelisierte und parallel ausführbare Anwendungen aus dem Automobilsektor zu evaluieren. Dies wurde mit dem *Automotive Compatible Multi-Core Operating System (ACMC-OS)* umgesetzt. Ferner unterstützt die Plattform die Einhaltung von Echtzeitbedingungen, im Besonderen im Bereich der Kommunikation und Synchronisation bei Multicore-Prozessoren. Ein weiterer Punkt ist die Unterstützung von Zeit- und Wert-Determinismus. Hierdurch können Eingaben und Ausgaben von Daten verschiedener Tasks bei der Entwicklung und Fehlersuche leichter nachverfolgt werden. Somit ist es auch möglich, den Datenfluss von Anwendungen, die ursprünglich für Singlecore-Systeme entwickelt wurden, auf Multicore-Plattformen im Überblick zu behalten.

Die Realisierung von ACMC-OS basiert auf einer Kernel-Bibliothek, die im Rahmen des parMERASA Projekts entwickelt wurde und neben dem Automobilbereich auch Einsatz in Betriebssystemen aus dem Luftfahrtbereich und der Baumaschinenbranche fand. Durch die mehrlagige Gliederung der Softwarearchitektur ist ein Austausch von Betriebssystemkomponenten zu Evaluationszwecken und eine Anpassung an zukünftige Technologien gewährleistet. Ferner ist es möglich, Teile der Betriebssystemsoftware für andere Industriedomänen nutzbar zu machen.

Bei der Untersuchung der Cross-Core Service Calls, die im AUTOSAR OS Standard spezifiziert sind, wurden zwei Techniken gegenübergestellt, die bei der Kommunikation zwischen Prozessorkernen Verwendung finden. Dabei handelt es sich um zwei gänzlich verschiedene Kommunikationsmodelle, einem Lock- und nachrichtenbasierten Ansatz. Der Vergleich zeigt die Leistungsunterschiede bei der Kommunikation auf Multicore-Plattformen auf, je nach eingesetzter Methode. Die Entwicklungsplattform bietet somit die Möglichkeit, vom Austausch grundlegender Synchronisationsmechanismen über gesamte Kommunikationsparadigmen die Inter-Core-Kommunikation zu untersuchen.

Um die Softwareentwicklung bezüglich des Zeitverhaltens und der Portabilität zu vereinfachen, kann man sich neben anderen Echtzeitprogrammiermodellen des Logical Execution Time (LET) Modells bedienen. Dieses Modell ermöglicht es, die Testbarkeit zu erleichtern und den Datenfluss innerhalb eines Programms besser verfolgen zu können, indem es vorhersagbare Zeitpunkte beim Lesen und Schreiben von Daten erlaubt. Somit ist auch die Kommunikation zwischen Prozessorkernen, die auf dem Austausch von Daten beruht, besser nachvollziehbar. Ein Nachteil dabei ist, dass die Aktualität der ausgetauschten

Daten abnimmt. Dies kann zu Verletzungen von Echtzeitbedingungen führen. Im letzten Hautkapitel dieser Arbeit wird eine Methode vorgestellt, die diesen Nachteil adressiert und das Datenalter je nach Anwendung deutlich verringern kann. Dabei wurde anhand von zufällig generierten Task-Sets gezeigt, dass das Datenalter mit der vorgestellten Methode durchschnittlich um $\approx 33\%$ gegenüber dem LET Modell gesenkt werden konnte.

6.2 Ausblick

In den Kapiteln 4 und 5 wurden bereits anknüpfende Forschungsfelder zum jeweiligen Thema angesprochen. In diesem Kapitel soll nochmals ein kurzer Überblick darüber gegeben werden.

Für die Untersuchung des Lock- und nachrichtenbasierten Kommunikationsmodells wurde eine Multicore-Plattform mit geteilter Speicherarchitektur zugrunde gelegt, wie sie in aktuellen Multicore-Mikrocontrollern für den Automobilbereich üblich sind. Eine weitere Möglichkeit wäre die Erforschung dieser Kommunikationsmodelle auf einer nachrichtenbasierten Hardwarearchitektur, bei welcher die Prozessorkerne keinen gemeinsamen Speicher zum Datenaustausch besitzen, sondern über ein Kommunikationsnetzwerk miteinander verbunden sind. Ein Beispiel für solch eine Architektur ist die Tiler TILE-Gx Prozessorfamilie (Tiler Corporation 2015). Ein weiterer Gesichtspunkt ist die Betrachtung der Inter-Core-Kommunikation unter der Prämisse von funktionaler Sicherheit. Hierbei ergeben sich andere Schwerpunkte bei der Umsetzung, um eine Rückwirkungsfreiheit zu garantieren.

Bei den Ausführungsmodellen und dem Ansatz zur Reduktion des Datenalters wurden die Ausgabezeitpunkte der Daten der einzelnen Tasks betrachtet. Durch das Vorziehen der Ausgabezeitpunkte konnte das Datenalter je nach Task-Set signifikant verringert werden. Ebenso bestünde die Möglichkeit, eine zeitliche Verschiebung der Eingabezeitpunkte zu untersuchen. Des Weiteren wäre eine genaue Analyse des zusätzlichen Aufwands für das Schreiben der Daten am Ende der Periode (LET Modell) und beim vorgezogenen Ausgabezeitpunkt (erweitertes LET Modell) denkbar. Diese Betrachtung könnte im weiteren Kontext die daraus resultierenden Kommunikationszeiten zwischen Kernen eines Multicore-Prozessors umfassen.

Literaturverzeichnis

- Aeronautical Radio, Inc., 1997. *ARINC Report 651-1: Design Guidance for Integrated Modular Avionics*.
- Aeronautical Radio, Inc., 2010. *ARINC Specification 653P1-3: Avionics Application Software Standard Interface, Part 1, Required Services*.
- Audsley, N. C., A. Burns, M. F. Richardson und A. J. Wellings, 1991. „Hard Real-Time Scheduling: The Deadline-Monotonic Approach“. In: *Workshop on Real-Time Operating Systems and Software (RTOS)*, S. 133–137.
- Audsley, N., A. Burns, M. Richardson, K. Tindell und A. J. Wellings, 1993. „Applying new scheduling theory to static priority pre-emptive scheduling“. In: *Software Engineering Journal* 8.5, S. 284–292. DOI: 10.1049/sej.1993.0034.
- Aussaguès, Christophe, Damien Chabrol, Vincent David, Didier Roux, Natalia Willey, Arnaud Tournadre und Marc Graniou, 2010. „PharOS, a Multicore OS Ready for Safety-Related Automotive Systems: Results and Future Prospects“. In: *Embedded Real-Time Software and Systems (ERTS²)*.
- Aussaguès, Christophe und Vincent David, 1998. „A Method and a Technique to Model and Ensure Timeliness in Safety Critical Real-Time Systems“. In: *International Conference on Engineering of Complex Computer Systems (ICECCS)*, S. 2–12. DOI: 10.1109/ICECCS.1998.706651.
- Aussaguès, Christophe, Emmanuel Ohayon, K. Brifault und Quang Dinh, 2009. „Using Multi-Core Architectures to Execute High Performance-Oriented Real-Time Applications“. In: *Parallel Computing (PARCO)*, S. 677–684.
- AUTOSAR, 2011. *Specification of Operating System*. Version 4.0.3.
- AUTOSAR, 2015a. *Layered Software Architecture*. Version 4.2.2.
- AUTOSAR, 2015b. *Specification of Operating System*. Version 4.2.2.
- AUTOSAR, 2015c. *Specification of RTE*. Version 4.2.2.

- AUTOSAR, 2016. *AUTOSAR*. URL: <http://www.autosar.org/> (besucht am 01. 02. 2016).
- Bass, Len, Paul Clementsl und Rick Kazman, 2012. *Software Architecture in Practice*. 3. Aufl. Boston, MA, USA: Addison-Wesley Professional. ISBN: 978-0-321-81573-6.
- Baumann, Andrew, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach und Akhilesh Singhanian, 2009. „The Multikernel: A New OS Architecture for Scalable Multicore Systems“. In: *Symposium on Operating Systems Principles (SOSP)*. SOSP '09, S. 29–44. DOI: 10.1145/1629575.1629579.
- BMW Group, 2008. *AUTOSAR erstmals in Serie – im neuen BMW 7er*. URL: <https://www.press.bmwgroup.com/deutschland/pressDetail.html?title=autosar-erstmals-in-serie-im-neuen-bmw-7er> (besucht am 17. 11. 2008).
- Böhm, Niko, Daniel Lohmann und Wolfgang Schröder-Preikschat, 2010. „Multi-Core Processors in the Automotive Domain: An AUTOSAR Case Study“. In: *Work-in-Progress Session of Euromicro Conference on Real-Time Systems (ECRTS)*, S. 25–28.
- Böhm, Niko, Daniel Lohmann und Wolfgang Schröder-Preikschat, 2011. „A Comparison of Pragmatic Multi-Core Adaptations of the AUTOSAR System“. In: *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, S. 16–22.
- Boniol, Frédéric, 2013. „New Challenges for Future Avionic Architectures“. English. In: *Modeling Approaches and Algorithms for Advanced Computer Applications*. Hrsg. von Abdelmalek Amine, Ait Mohamed Otmane und Ladjel Bellatreche. Bd. 488. Studies in Computational Intelligence, S. 1–1. DOI: 10.1007/978-3-319-00560-7_1.
- Boyd-Wickizer, Silas, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang und Zheng Zhang, 2008. „Corey: An Operating System for Many Cores“. In: *Operating Systems Design and Implementation (OSDI)*. OSDI'08, S. 43–57.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2013. „A Cross-Domain System Architecture for Embedded Hard Real-Time Many-Core Systems“. In: *Embedded and Ubiquitous Computing (EUC)*, S. 2034–2041. DOI: 10.1109/HPCC.and.EUC.2013.293.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2014a. „Comparison of Service Call Implementations in an AUTOSAR Multi-core OS“. In: *Industrial Embedded Systems (SIES)*, S. 199–205. DOI: 10.1109/SIES.2014.6871204.

- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2014b. „Synchronous Execution of a Parallelised Interrupt Handler“. In: *Work-in-Progress Session of Real-Time Embedded Technology and Applications Symposium (RTAS)*, S. 9–10.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2016. „Data Age Diminution in the Logical Execution Time Model“. In: *Accepted for publication at Architecture of Computer Systems (ARCS)*.
- Brinkschulte, Uwe und Theo Ungerer, 2010. *Mikrocontroller und Mikroprozessoren*. 3. Aufl. eXamen.press. Springer-Verlag Berlin Heidelberg. DOI: 10.1007/978-3-642-05398-6.
- Burns, Alan und Andrew J. Wellings, 2001. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. 3. Aufl. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201729881.
- Buttazzo, Giorgio C., 2011. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3. Aufl. Bd. 24. Real-Time Systems Series. Springer US. DOI: 10.1007/978-1-4614-0676-1.
- Cheriton, David R. und Kenneth J. Duda, 1994. „A Caching Model of Operating System Kernel Functionality“. In: *Operating Systems Design and Implementation (OSDI)*. OSDI '94.
- Cullmann, Christoph, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet und Reinhard Wilhelm, 2010. „Predictability Considerations in the Design of Multi-Core Embedded Systems“. In: *Embedded Real Time Software and Systems (ERTS²)*, S. 36–42.
- David, Vincent, Jean Delcoigne, Evelyne Leret, Alain Ourghanlian, Philippe Hilsenkopf und Philippe Paris, 1998. „Safety Properties Ensured by the OASIS Model for Safety Critical Real-Time Systems“. In: *International Conference on Computer Safety, Reliability and Security*. SAFECOMP '98, S. 45–59. ISBN: 3-540-65110-1.
- Deutsches Institut für Normung e.V., 2003. *DIN EN 61131-3:2003-12: Speicherprogrammierbare Steuerungen - Teil 3: Programmiersprachen (IEC 61131-3:2003); Deutsche Fassung EN 61131-3:2003*. EN 61131-3:2003-12. Berlin: Beuth.
- Deutsches Institut für Normung e.V., 2004. *DIN EN 61131-1:2004-03: Speicherprogrammierbare Steuerungen - Teil 1: Allgemeine Informationen (IEC 61131-1:2003); Deutsche Fassung EN 61131-1:2003*. EN 61131-1:2004-03. Berlin: Beuth.
- Deutsches Institut für Normung e.V., 2011. *DIN EN 61508:2011-02; VDE 0803:2011-02: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer*

- elektronischer Systeme (IEC 61508:2010); Deutsche Fassung EN 61508:2010*. EN 61508-1:2010-02. Berlin: Beuth.
- Dijkstra, Edsger W., 1968. „Cooperating Sequential Processes. From Semaphores to Remote Procedure Calls“. In: *The Origin of Concurrent Programming*. Hrsg. von Per Brinch Hansen. Springer New York, S. 65–138. DOI: 10.1007/978-1-4757-3472-0_2.
- Ekman, Mats, 2010. „Avionic Architectures – Trends and challenges“. In: *ICES-INCOS: Architecting Embedded Systems*. presentation. Saab.
- Engler, Dawson R., M. Frans Kaashoek und James O’Toole Jr., 1995. „Exokernel: An Operating System Architecture for Application-Level Resource Management“. In: *Symposium on Operating Systems Principles (SOSP)*. SOSP ’95, S. 251–266. DOI: 10.1145/224056.224076.
- ETAS Group Automotive LifeCycle Solutions, 2009. *Multi-core Automotive ECUs: Software and Hardware Implications*. URL: <http://www.etas.com/>.
- Eveleens, René L. C., 2006. „Integrated Modular Avionics Development Guidance and Certification Considerations“. In: *Mission Systems Engineering*. NATO Science und Technology Organization, pages. ISBN: 978-92-837-0053-1.
- Farcas, Emilia, Claudiu Farcas, Wolfgang Pree und Josef Templ, 2005. „Transparent Distribution of Real-time Components Based on Logical Execution Time“. In: *ACM SIGPLAN Notices* 40.7, S. 31–39. DOI: 10.1145/1070891.1065915.
- Garside, Richard und F. Joe Pighetti, 2007. „Integrating Modular Avionics: A new role emerges“. In: *Digital Avionics Systems Conference (DASC)*, pages. DOI: 10.1109/DASC.2007.4391843.
- Gerdes, Mike, 2013. „Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores“. Diss. Augsburg: University of Augsburg.
- Gerdes, Mike, Florian Kluge, Theo Ungerer und Christine Rochange, 2012a. „The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs“. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, S. 88–97. DOI: 10.1109/RTCSA.2012.11.
- Gerdes, Mike, Florian Kluge, Theo Ungerer, Christine Rochange und Pascal Sainrat, 2012b. „Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications“. In: *Design, Automation Test in Europe Conference Exhibition (DATE)*, S. 671–676. DOI: 10.1109/DATE.2012.6176555.
- Gerdes, Mike, Julian Wolf, Irakli Guliashvili, Theo Ungerer, Michael Houston, Guillem Bernat, Stefan Schnitzler und Hans Regler, 2011. „Large Drilling Machine Control

- Code – Parallelisation and WCET Speedup“. In: *Industrial Embedded Systems (SIES)*, S. 91–94. DOI: 10.1109/SIES.2011.5953688.
- Goossens, Joel und Christophe Macq, 2001. „Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation“. In: *RTS Embedded Systems*, S. 133–148.
- Gresser, Klaus, 1993. „An Event Model for Deadline Verification of Hard Real-Time Systems“. In: *Workshop on Real-Time Systems*, S. 118–123. DOI: 10.1109/EMWRT.1993.639067.
- Halbwachs, Nicolas, 1993. *Synchronous Programming of Reactive Systems*. 1. Aufl. Bd. 215. The Springer International Series in Engineering and Computer Science. Springer US. DOI: 10.1007/978-1-4757-2231-4.
- Hennessy, John L. und David A. Patterson, 2011. *Computer Architecture: A Quantitative Approach*. English. 5. Aufl. San Francisco, CA, USA: Morgan Kaufmann Publishers. ISBN: 978-0-12383-872-8.
- Henzinger, Thomas A., Benjamin Horowitz und Christoph Meyer Kirsch, 2001. „Giotto: A Time-Triggered Language for Embedded Programming“. In: *Embedded Software*. Hrsg. von Thomas A. Henzinger und Christoph Meyer Kirsch. Bd. 2211. Lecture Notes in Computer Science, S. 166–184. DOI: 10.1007/3-540-45449-7_12.
- IEEE Computer Society, 2008. „IEEE Standard 1003.1-2008: IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))“. In: *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell*, S. 1–3874. DOI: 10.1109/IEEESTD.2008.7394902.
- Internationale Organisation für Normung, 2011. *ISO 26262:2011-11; Straßenfahrzeuge - Funktionale Sicherheit*. German. ISO 26262-1:2011-11. Berlin: Beuth.
- Kirsch, Christoph M., 2002. „Principles of Real-Time Programming“. English. In: *Embedded Software*. Hrsg. von Alberto Sangiovanni-Vincentelli und Joseph Sifakis. Bd. 2491. Lecture Notes in Computer Science, S. 61–75. DOI: 10.1007/3-540-45828-X_6.
- Kirsch, Christoph M. und Raja Sengupta, 2007. „The Evolution of Real-Time Programming“. In: *Handbook of Real-Time and Embedded Systems*. Hrsg. von Insup Lee, Joseph Y-T. Leung und Sang H. Son. Chapman & Hall/CRC Computer and Information Science Series, pages. DOI: 10.1201/9781420011746.
- Kirsch, Christoph M. und Ana Sokolova, 2012. „The Logical Execution Time Paradigm“. In: *Advances in Real-Time Systems*. Hrsg. von Samarjit Chakraborty und Jörg Eberspächer, S. 103–120. DOI: 10.1007/978-3-642-24349-3_5.

- Kloda, Tomasz, Bruno D'Ausbourg und Luca Santinelli, 2014a. „Towards a More Flexible Timing Definition Language“. In: *Workshop on Quantitative Aspects of Programming Languages and Systems (ETAPS)*.
- Kloda, Tomasz, Bruno d'Ausbourg und Luca Santinelli, 2014b. „EDF Schedulability Analysis for an Extended Timing Definition Language“. In: *Symposium on Industrial Embedded Systems (SIES)*, S. 30–40. DOI: 10.1109/SIES.2014.6871183.
- Kluge, Florian, 2014. *tms-sim – Timing Models Scheduling Simulation Framework*. Techn. Ber. 2014-07. Universität Augsburg. DOI: 10.13140/2.1.1251.2321.
- Kluge, Florian, Markus Neuerburg und Theo Ungerer, 2015. „Utility-Based Scheduling of (m,k)-Firm Real-Time Task Sets“. English. In: *Architecture of Computing Systems (ARCS)*. Hrsg. von Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen und Uwe Brinkschulte. Bd. 9017. Lecture Notes in Computer Science, S. 201–211. DOI: 10.1007/978-3-319-16086-3_16.
- Kramer, Simon, Dirk Ziegenbein und Arne Hamann, 2015. „Real World Automotive Benchmarks For Free“. In: *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.
- Liedtke, Jochen, 1993. „Improving IPC by Kernel Design“. In: *Symposium on Operating Systems Principles (SOSP)*. SOSP '93, S. 175–188. DOI: 10.1145/168619.168633.
- Liedtke, Jochen, 1995. „On μ -Kernel Construction“. In: *Symposium on Operating Systems Principles (SOSP)*. SOSP '95, S. 237–250. DOI: 10.1145/224056.224075.
- Liu, C. L. und James W. Layland, 1973. „Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment“. In: *Journal of the ACM* 20.1, S. 46–61. DOI: 10.1145/321738.321743.
- Löfwenmark, Andreas und Simin Nadjm-Tehrani, 2014. „Challenges in Future Avionic Systems on Multi-Core Platforms“. In: *International Symposium on Software Reliability Engineering Workshops*. ISSREW '14, S. 115–119. DOI: 10.1109/ISSREW.2014.70.
- Marwedel, Peter, 2007. *Eingebettete Systeme*. 1. Aufl. eXamen.press. Springer-Verlag Berlin Heidelberg. DOI: 10.1007/978-3-540-34049-2.
- Masmano, M, Y Valiente, P Balbastre, I Ripoll, A Crespo und J J. Metge, 2010. „LithOS: a ARINC-653 guest operating for XtratuM“. In: *Real-Time Linux Workshop (RTLWS)*.
- Müller, Kevin, Georg Sigl, Benoit Triquet und Michael Paulitsch, 2014. „On MILS I/O Sharing Targeting Avionic Systems“. In: *Dependable Computing Conference (EDCC)*, S. 182–193. DOI: 10.1109/EDCC.2014.35.

- Nehmer, Jürgen und Peter Sturm, 2001. *Systemsoftware: Grundlagen moderner Betriebssysteme*. 2. Aufl. dpunkt-Verlag. ISBN: 978-3-89864-115-9.
- Nowotsch, Jan und Michael Paulitsch, 2012. „Leveraging Multi-Core Computing Architectures in Avionics“. In: *Dependable Computing Conference (EDCC)*, S. 132–143. DOI: 10.1109/EDCC.2012.27.
- OSEK/VDX, 2005. *Operating System*. Version 2.2.3.
- Popovici, Katalin und Ahmed Jerraya, 2009. „Hardware Abstraction Layer“. English. In: *Hardware-dependent Software*. Hrsg. von Wolfgang Ecker, Wolfgang Müller und Rainer Dömer, S. 67–94. DOI: 10.1007/978-1-4020-9436-1_4.
- Potop-Butucaru, Dumitru, Robert de Simone und Jean-Pierre Talpin, 2005. „The Synchronous Hypothesis and Synchronous Languages“. In: *Embedded Systems Handbook*. Hrsg. von Richard Zurawski, pages. DOI: 10.1201/9781420038163.ch8.
- Pyka, Arthur, Mathias Rohde und Sascha Uhrig, 2014. „A Real-time Capable Coherent Data Cache for Multicores“. In: *Concurrency and Computation: Practice and Experience* 26.6, S. 1342–1354. DOI: 10.1002/cpe.3172.
- QNX Software Systems Ltd., 2014. *QNX Neutrino RTOS 6.6*. Software. URL: <http://qnx.com>.
- Rajkumar, Rangunathan, Lui Sha und John P. Lehoczky, 1988. „Real-Time Synchronization Protocols for Multiprocessors“. In: *Real-Time Systems Symposium (RTSS)*, S. 259–269. DOI: 10.1109/REAL.1988.51121.
- RTCA, Inc., 2005. *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. DO-297.
- Rushby, John M., 1981. „Design and Verification of Secure Systems“. In: *Symposium on Operating Systems Principles (SOSP)*. SOSP '81, S. 12–21. DOI: 10.1145/800216.806586.
- Seitz, Matthias, 2012. *Speicherprogrammierbare Steuerungen für die Fabrik- und Prozessautomation: Strukturierte und objektorientierte SPS-Programmierung, Motion Control, Sicherheit, vertikale Integration*. 3. Aufl. Carl Hanser Verlag GmbH Co KG. ISBN: 978-3-446-43325-0.
- Sensor-Technik Wiedemann GmbH, 2011. *ESX-3XL User Manual*. Version V1.03r2. Kaufbeuren.
- Sha, Lui, Rangunathan Rajkumar und John P. Lehoczky, 1990. „Priority Inheritance Protocols: An Approach to Real-Time Synchronization“. In: *IEEE Transactions on Computers* 39.9, S. 1175–1185. DOI: 10.1109/12.57058.

- SOCLIB Consortium, 2013. *Projet SOCLIB: Plate-forme de modélisation et de simulation de systèmes intégrés sur puce (the SOCLIB project: An integrated system-on-chip modelling and simulation platform)*. Techn. Ber. CNRS.
- Tanenbaum, Andrew S., 2009. *Moderne Betriebssysteme*. 3. Aufl. Pearson Education Deutschland GmbH. ISBN: 978-3-8273-7342-7.
- Tilera Corporation, 2015. *TILE-Gx8072 Specification Brief*. URL: <http://www.tilera.com/> (besucht am 11.03.2015).
- Ungerer, Theo, 1997. *Parallelrechner und parallele Programmierung*. Spektrum Akademischer Verlag. ISBN: 978-3-8274-0231-8.
- Ungerer, Theo, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, João Fernandes, Pavel G. Zaykov, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Ian Broster, Nick Lay, David George, Eduardo Quiñones, Milos Panic, Jaume Abella, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde und Arthur Pyka, 2013. „parMERASA – Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability“. In: *Euromicro Conference on Digital System Design (DSD)*, S. 363–370. DOI: 10.1109/DSD.2013.46.
- Ungerer, Theo, Francisco J. Carzola, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaß und Jörg Mische, 2010. „Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability“. In: *IEEE Micro* 30.5, S. 66–75. DOI: 10.1109/MM.2010.78.
- Watkins, Christopher B. und Randy Walter, 2007. „Transitioning from federated avionics architectures to Integrated Modular Avionics“. In: *Digital Avionics Systems Conference (DASC)*, pages. DOI: 10.1109/DASC.2007.4391842.
- Wilhelm, Reinhard, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat und Per Stenström, 2008. „The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools“. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3, 36:1–36:53. DOI: 10.1145/1347375.1347389.
- Wörn, Heinz und Uwe Brinkschulte, 2005. *Echtzeitsysteme*. 1. Aufl. eXamen.press. Springer-Verlag Berlin Heidelberg. DOI: 10.1007/b139050.

Veröffentlichungen als Erstautor

- Bradatsch, Christian und Florian Kluge, 2013a. *parMERASA Multi-core RTOS Kernel*. Techn. Ber. 2013-02. University of Augsburg.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2013b. „A Cross-Domain System Architecture for Embedded Hard Real-Time Many-Core Systems“. In: *Embedded and Ubiquitous Computing (EUC)*, S. 2034–2041. DOI: 10.1109/HPCC.and.EUC.2013.293.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2014a. „Comparison of Service Call Implementations in an AUTOSAR Multi-core OS“. In: *Industrial Embedded Systems (SIES)*, S. 199–205. DOI: 10.1109/SIES.2014.6871204.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2014b. „Synchronous Execution of a Parallelised Interrupt Handler“. In: *Work-in-Progress Session of Real-Time Embedded Technology and Applications Symposium (RTAS)*, S. 9–10.
- Bradatsch, Christian, Florian Kluge und Theo Ungerer, 2016. „Data Age Diminution in the Logical Execution Time Model“. In: *Accepted for publication at Architecture of Computer Systems (ARCS)*.
- Bradatsch, Christian, Sebastian Schlingmann, Sascha Uhrig und Theo Ungerer, 2011a. „MANJAC - Ein Many-Core-Emulator auf Multi-FPGA-Basis“. In: *Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS)*. Bd. 28. PARS Mitteilungen GI, S. 48–57.
- Bradatsch, Christian, Theo Ungerer, Rafael Zalman und Andre Lajtkep, 2011b. „Towards Runtime Testing in Automotive Embedded Systems“. In: *Symposium on Industrial Embedded Systems (SIES)*, S. 55–58. DOI: 10.1109/SIES.2011.5953679.

Weitere Veröffentlichungen

Ungerer, Theo, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel G. Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernandez, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde und Arthur Pyka, 2015. „Experiences and Results of Parallelisation of Industrial Hard Real-time Applications for the parMERASA Multi-core“. In: *Workshop on High-performance and Real-time Embedded Systems (HiRES)*.

Ungerer, Theo, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, João Fernandes, Pavel G. Zaykov, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Ian Broster, Nick Lay, David George, Eduardo Quiñones, Milos Panic, Jaume Abella, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde und Arthur Pyka, 2013. „parMERASA – Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability“. In: *Euromicro Conference on Digital System Design (DSD)*, S. 363–370. DOI: 10.1109/DSD.2013.46.