



Hochschule München
Fakultät 7 für Informatik und Mathematik

Entwicklung und Evaluierung von Asymmetric Multiprocessing auf einem heterogenen Multiprozessorsystem

Development and evaluation of asymmetric multiprocessing on a heterogeneous multiprocessor system

Masterarbeit

von

David Kauschke

Erstkorrektor
Zweitkorrektor
Betreuer
Tag der Einreichung

Prof. Dr.-Ing. Martin Orehek
Prof. Dr. Max Fischer
Dipl.-Ing. Wolfram Gettert, Mixed Mode GmbH
11. September 2020

Zusammenfassung

Heterogene Multiprozessorsysteme (MPSoC) sind in den letzten Jahren immer beliebter für industrielle Anwendungen aufgrund der hohen Performance, den niedrigeren Kosten und der Energieeffizienz geworden. Besonders die vielfältigen unterschiedlichen integrierten Prozessoren, auf denen unterschiedliche Betriebssysteme laufen, auch als Asymmetric Multiprocessing (AMP) bezeichnet, bringen viele Herausforderungen mit sich. Die zwei größten Herausforderungen sind das Lifecycle-Management (LCM) und die Interprozessorkommunikation (IPC). Diese Arbeit untersucht den Aufbau von heterogenen MPSoCs und den Einsatz von unterschiedlichen Betriebssystemen. Basierend auf den Entscheidungen für die Auswahl der eingesetzten AMP-Architektur, einem heterogenen MPSoC und der Auswahl eines Frameworks als Lösungsansatz für die zwei Herausforderungen erfolgt die Entwicklung eines AMP-Systems. Anschließend wird die Umsetzung des entwickelten AMP-Systems mit dem ausgewählten Framework OpenAMP auf dem ausgewählten MPSoC i.MX8X von NXP mit einem Embedded-Linux auf dem ARM Cortex-A35 und einem FreeRTOS auf dem ARM Cortex-M4 durchgeführt. Für die Evaluierung des umgesetzten AMP-Systems wird auf dem i.MX8X basierend auf den zwei erarbeiteten Anwendungsszenarien Prozessdatenüberwachung und Hardware-in-the-Loop auf Latenzzeiten und Datendurchsatz gemessen. Die Ergebnisse zeigen unter anderem, dass die maximale Latenzzeit vom Linux-Userspace zum FreeRTOS mit dem Einsatz vom RT-Patch bei $628 \mu\text{s}$ und der Datendurchsatz bei $4,08 \text{ MB/s}$ liegt. Aus den Ergebnissen lässt sich ableiten, dass der Einsatz des umgesetzten AMP-Systems für die vorgestellten Anwendungsszenarien und die IPC für weiche Echtzeit geeignet ist.

Abstract

Heterogeneous multiprocessor systems (MPSoC) have become increasingly popular for industrial applications in recent years due to their high performance, lower costs and energy efficiency. Especially the many different integrated processors running different operating systems, also known as asymmetric multiprocessing (AMP), pose many challenges. The two biggest challenges are lifecycle management (LCM) and interprocessor communication (IPC). This thesis investigates the structure of heterogeneous MPSoCs and the use of different operating systems. Based on the decisions for the selection of the used AMP architecture, a heterogeneous MPSoC and the selection of a framework as a solution for the two challenges the development of an AMP system is done. Subsequently, the implementation of the developed AMP system with the selected framework OpenAMP on the selected MPSoC i.MX8X from NXP is carried out with an embedded Linux on the ARM Cortex-A35 and a FreeRTOS on the ARM Cortex-M4. For the evaluation of the implemented AMP system, process data monitoring and hardware-in-the-loop latency and data throughput will be measured on the i.MX8X based on the two application scenarios developed. Among other things, the results show that the maximum latency from Linux user space to FreeRTOS with the use of the RT patch is $628 \mu\text{s}$ and the data throughput is 4.08 MB/s . From the results it can be deduced that the use of the implemented AMP system is suitable for the presented application scenarios and the IPC for soft real-time.

Kauschke, David
(Familienname, Vorname)

München, 11.09.2020
(Ort, Datum)

25.07.1993
(Geburtsdatum)

IG / SS / 2020
(Studiengruppe / WS/SS)

Erklärung

Gemäß § 40 Abs. 1 i. V. m. § 31 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die Masterarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.



(Unterschrift)

Danksagung

Mein Dank gilt zuerst meinem Teamleiter Wolfram Gettert und meinem unternehmensexternen Betreuer Eicke Hecht, mit denen ich Ideen und Lösungsansätze stets diskutieren konnte und der mir bei Problemen im Arbeitsverlauf tatkräftig zur Seite standen.

Für die Möglichkeit, meine Masterarbeit in seinem Unternehmen durchführen zu können, möchte ich dem Geschäftsführer der *Mixed Mode GmbH*, Herr Helmut Süßmuth, herzlich danken.

Mein besonderer Dank gilt auch Herrn Dr. Richard Kölbl, der mir als Lektor bei der Durchsicht und Prüfung der schriftlichen Komponente der Abschlussarbeit helfend zur Seite stand.

Jenseits vom Unternehmen möchte ich Herrn Prof. Dr.-Ing. Martin Orehek für die Diskussionen hinsichtlich möglicher Lösungswege und die tatkräftige Unterstützung als Erstkorrektor und Hochschulbetreuer sowie Herrn Dr. Max Fischer für die Übernahme der Zweitkorrektur meiner Arbeit danken.

Ich möchte weiterhin meiner besseren Hälfte, Ramona Schwendemann, und meiner Familie dafür danken, dass sie mir stets den Rücken frei gehalten haben und für Diskussionen und Fragen jederzeit zur Verfügung standen.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation für heterogene Multiprozessorsysteme	1
1.2 Auftretende Probleme auf heterogenen Multiprozessorsystemen	1
1.3 Ziele der vorliegenden Arbeit	2
1.4 Methodik und Aufbau der vorliegenden Arbeit	2
2 Grundlagen über Multicore- und Multiprozessorsysteme	5
2.1 Definitionen	5
2.2 Multicore-Prozessoren	5
2.3 Multiprozessorsysteme	7
2.3.1 Heterogene Multiprozessorsysteme	8
2.3.2 Prozessorvielfalt auf heterogenen Multiprozessorsystemen	9
2.4 Betriebssysteme auf heterogenen Multiprozessorsystemen	9
2.5 Einsatz von unterschiedlichen Betriebssystemen auf heterogenen Multiprozessorsystemen	11
2.5.1 Symmetric Multiprocessing	11
2.5.2 Asymmetric Multiprocessing	11
2.6 Herausforderungen auf AMP-Systemen	13
2.6.1 Speichermanagement	13
2.6.2 Ressourcenmanagement	14
2.6.3 Systemboot-Management	14
2.6.4 Lifecycle-Management	14
2.6.5 Interprozessorkommunikation	14
2.6.6 Systemsicherheit	14
2.6.7 Debugging	15
2.6.8 Fazit	15
3 Entwicklung eines AMP-Systems auf heterogenen Multiprozessorsystemen	16
3.1 Architekturen für AMP-Systeme	16
3.2 Anforderungen an MMAMP	17
3.2.1 Allgemein	17
3.2.2 Hardware	17
3.2.3 Software	18
3.3 Auswahl des eingesetzten Multiprozessorsystems	18
3.4 Auswahl des eingesetzten Software-Frameworks für LCM und IPC	20
3.4.1 OpenCL, Cilk, openMP und MC API	21
3.4.2 OpenAMP	21
3.4.3 RPMsgLite	22

4 Analyse von OpenAMP	24
4.1 Überblick	24
4.2 Aufbau von RPMsg	26
4.3 Aufbau von RemoteProc	27
4.4 Hardwareabstraktions-Bibliothek	28
4.5 Entscheidung RPMsgLite oder OpenAMP	29
5 Multiprozessor-Architektur des i.MX8X	30
5.1 Subsysteme	31
5.1.1 Application Processor Unit	31
5.1.2 Realtime Processor Unit	31
5.2 System Controller Unit	31
5.2.1 Resource Management Service	32
5.2.2 Power Management Service	33
5.2.3 System Controller Firmware API	33
5.2.4 Messaging Unit	33
5.3 Systembootprozess	33
6 Konzept	36
6.1 Systementwurf	36
6.1.1 Komponente MeasureMan im Userspace	37
6.1.2 Komponente Remoteproc und RPMsg Komponenten im Kernelspace	38
6.1.3 M4 Firmware	38
6.2 Entwurf für das Systemboot-Management	38
7 Umsetzung	40
7.1 Eingesetztes Development-Board Symphony-Board	40
7.2 Übersicht zur Softwarestruktur und Entwicklungsumgebung	41
7.3 Systemboot-Management	42
7.3.1 ATF-Partition	42
7.3.2 Linux-Partition	43
7.3.3 M4-Partition	43
7.4 LCM mit RemoteProc im Kernelspace	43
7.5 IPC mit RPMsg im Kernelspace	45
7.6 Portierung von OpenAMP auf den M4	48
7.6.1 Portierung der Systemumgebung	48
7.6.2 Anpassung der Shared-Memory und Ressourcen-Tabelle	49
7.6.3 Implementierung der M4-Applikation	49
7.7 Userspace Applikation MeasureMan	51
8 Evaluierung	53
8.1 Anwendungsszenarien	53
8.1.1 Prozessdatenüberwachung	53
8.1.2 Hardware-in-the-Loop	54
8.2 Versuchsplanung	55
8.2.1 Ziele der Versuche	56
8.2.2 Randbedingungen der Versuche	56
8.2.3 Vorstellung der Versuche	56
8.2.4 Messwerkzeuge und ihre Ungenauigkeiten	59
8.2.5 Messapplikationen	61

8.3	Versuchsdurchführung und Ergebnisse	61
8.3.1	Hauptversuch 1 - Abhängigkeit der IPI-Latenzzeit zwischen A35 und M4 von der Nachrichtengröße	61
8.3.2	Hauptversuch 2 - Abhängigkeit der IPI-Latenzzeit zwischen A35 und M4 von der Wiederholungsrate	65
8.3.3	Hauptversuch 3 - Abhängigkeit der OpenAMP-Verarbeitungszeit von der Nachrichtengröße	67
8.3.4	Hauptversuch 4 - Abhängigkeit der Latenzzeit des TTY-Devices von der Nachrichtengröße	68
8.3.5	Hauptversuch 5 - Abhängigkeit der Latenzzeit des TTY-Devices von der Wiederholungsrate	78
8.3.6	Hauptversuch 6 - Abhängigkeit des Datendurchsatzes zwischen A35 und M4 von der Systemlast	80
8.4	Diskussion der Messergebnisse	82
8.4.1	Hauptversuche 1 und 2	82
8.4.2	Hauptversuch 3	82
8.4.3	Hauptversuche 4 und 5	82
8.4.4	Hauptversuch 6	83
8.4.5	Stabilität des Lifecycle-Managements	83
9	Résumé und Ausblick	84
9.1	Zusammenfassung und Zielerreichung	84
9.2	Diskussion	85
9.2.1	Entwicklung des MMAMP-Systems	85
9.2.2	Ergebnisse der Evaluierung	86
9.3	Ausblick	87
9.3.1	Optimierungen bei Latenzzeiten und bei Datendurchsatz	87
9.3.2	Erweiterung des Parsens der Firmware	87
9.3.3	Einsatz in der Praxis	87
9.3.4	Ansätze der anderen AMP-Architekturen	87
9.3.5	Entwicklung einer grafischen Benutzeroberfläche	88
9.3.6	Proprietäre Lösung für die IPC	88
10	Anhang	89
10.1	MeasureMan und OpenAMP M4-Applikation	89
10.2	Messungen	91

Glossar

AMBA Advanced Microcontroller Bus Architecture

AMP Asymmetric Multiprocessing

APU Application Processor Unit

ATF ARM Trusted Firmware

BSP Board Support Package

CI Continuous Integration

CMP Chip Multiprocessor

CS Coprocessor System

CSU Configuration and Security Unit

DMA Direct Memory Access

DSP Digital Signal Processor

DUT Device Under Test

ELF Executable and Linkable Format

FPGA Field Programmable Gate Array

GPIO General Purpose Input/Output

GPOS General Purpose Operating Systems

GPU Graphical Processor Unit

HiL Hardware-in-the-Loop

HMI Human-Machine Interface

IP Intellectual Property

IPC Interprozessorkommunikation

IPI Interprocessor-Interrupt

IRQ Interrupt Request

ISA Instruction Set Architecture

ISR Interrupt Service Routine

LA Logic-Analyzer

LCM Lifecycle-Management

LKM Ladbares Kernelmodul

LPS Low Power System

MCA Multicore Assocation

MCAPI Multicore Communications API

MCS Mixed Critical System

MEMF Mentor Embedded Multicore Framework

MMAMP Mixed Mode AMP

MMU Memory Management Unit

MPI Message-Passing-Interface

MPP Massively Parallel Processing

MPSoC Multiprozessorsystem

MPU Memory Protection Unit

MU Messaging Unit

NFS Network File System

NoC Network-On-Chip

OpenAMP Open Asymmetric Multi Processing

OTP One Time Pad

PA Physikalische Adresse

PLL Phasenregelschleifen

PMS Power Management Service

PMU Power Management Unit

POSIX Portable Operating System Interface

RemoteProc Remote Processor Framework

RMS Resource Management Service

RPC Remote Procedure Call

RPMsg Remote Processor Messaging

RPU Real Time Processor Unit

RT Real Time

RTOS Real Time Operating Systems

sAMP Supervised AMP

SCFW System Controller Firmware

SCFW-API System Controller Firmware API

SCPI Standard Commands for Programmable Instruments

SCU System Controller Unit

SECO Security Controller

SIL Safety Integrity Level

SMP Symmetric Multiprocessing

SoC System-on-Chip

SOM System on Module

SPL Second Program Loader

TCM Tightly-Coupled Memory

TFTP Trivial File Transfer Protocol

TTY Teletypewriter

uAMP Unsupervised AMP

UMA Uniform Memory Access

VA Virtuelle Adresse

VPU Video Processing Unit

WCET Worst Case Execution Time

xRDC Extended Resource Domain Controller

Abbildungsverzeichnis

1.1	Aufbau und Methodik der vorliegenden Arbeit.	4
2.1	Aufbau eines Quad-Core-Prozessors.	6
2.2	Aufbau einer MPSoC Architektur.	8
2.3	SMP-fähiges Linux auf einem ARM Cortex-A53.	11
2.4	Aufbau eines möglichen AMP-Systems.	12
3.1	Darstellung des OpenAMP-Frameworks auf einem Linux, RTOS und Bare-Metal.	22
4.1	Darstellung Varianten von OpenAMP.	25
4.2	RPMsg-Aufbau: Unterteilt in drei Schichten gemäß OSI-Modell.	26
4.3	LCM-Ablauf durch die Komponente RemoteProc.	28
5.1	i.MX8X: Systemüberblick.	30
5.2	i.MX8X: System-Boot.	34
5.3	i.MX8X: Standard-Partitionskonfiguration.	35
6.1	MMAMP: Systementwurf des AMP-Systems.	37
6.2	MMAMP: Systemboot-Management.	39
7.1	Eingesetztes Development-Board Symphony-Board von Variscite.	41
7.2	RPMsg-Architektur auf dem eingesetzten Linux.	46
7.3	UML-Sequenzdiagramm für die IPC zwischen A35 und M4 mithilfe von zwei IPIs.	47
7.4	UML-Klassendiagramm für den Aufbau der Ressourcen-Tabelle	49
7.5	UML-Aktivitätsdiagramm für den Ablauf der Initialisierung M4-Firmware.	51
8.1	Systemüberblick zu einem möglichen HiL-System.	55
8.2	Systemüberblick zu den durchzuführenden Versuchen.	57
8.3	Histogramm zu Versuch 1.1: IPI-Latenzzeit von A35 nach M4 mit einer Nachrichtengröße von 496 Bytes	62
8.4	Histogramm zu Versuch 1.2: IPI-Latenzzeit von M4 nach A35 mit einer Nachrichtengröße von 496 Bytes.	64
8.5	Histogramm zu Versuch 2.1: IPI-Latenzzeit von A35 nach M4 mit einer Wiederholungsrate von 1 kHz.	66
8.6	Histogramm zu den Versuch 2.2: IPI-Latenzzeit von M4 nach A35 mit einer Wiederholungsrate von 1 kHz.	67
8.7	Histogramm zu Versuch 4.1, Messung 3: Latenzzeit von TTY-Write mit einer Nachrichtengröße von 496 Bytes.	71
8.8	Histogramm zu Versuch 4.2, Messung 3: Latenzzeit von TTY-Read mit einer Nachrichtengröße von 496 Bytes.	74
8.9	Histogramm zu Versuch 4.3, Messung 3: Latenzzeit von TTY-Read mit einer Nachrichtengröße von 496 Bytes.	76

8.10 Histogramm zu Versuch 4.4, Messung 3: Latenzzeit von TTY-Read mit einer Nachrichtengröße von 496 Bytes.	78
8.11 Versuch 5.1: Durchsatz zwischen Userspace und M4 mit der Systemlast Idle.	81
10.1 Histogramm zu Vorversuch 1: Kernelspace GPIO-Latenzzeit.	91
10.2 Histogramm zu Vorversuch 2: M4 GPIO-Latenzzeit.	92

Tabellenverzeichnis

2.1	Darstellung einer Speicherhierarchie aus Sicht eines Multi-Cores.	7
2.2	Beschreibung einer möglichen Speicheraufteilung eines heterogenen MPSoC.	13
3.1	Bewertung der drei ausgewählten heterogenen MPSoC.	19
7.1	Einsatz und Zugriff auf die eingesetzten Speicherbereiche für den A35 und M4	43
7.2	i.MX8X TCM-Speicherbereich vom M4 innerhalb vom Speicherbereich D aus der Sichtweise vom A35 und M4.	44
7.3	Beschreibung des RPMsg-Buffers.	47
7.4	Aufteilung der Speicherbereiche für die VirtIO-Devices	47
8.1	Auswahl der zur Verfügung gestellten Messwerkzeuge zur Messung der IPI-Latenzzeiten	60
8.2	Überblick zu den Messungenauigkeiten	60
8.3	Parameter zu Versuch 1: IPI-Latenzzeit von A35 zwischen M4 in Abhängigkeit von der Nachrichtengröße.	61
8.4	Ergebnisse zu Versuch 1.1: IPI-Latenzzeit von A35 nach M4 in Abhängigkeit von der Nachrichtengröße.	62
8.5	Ergebnisse zu Versuch 1.2: IPI-Latenzzeit von M4 nach A35 in Abhängigkeit von der Nachrichtengröße.	64
8.6	Parameter zu Versuch 2: IPI-Latenzzeit zwischen A35 und M4 in Abhängigkeit von der Wiederholungsrate.	65
8.7	Ergebnisse zu Versuch 2.1: IPI-Latenzzeit von A35 nach M4 in Abhängigkeit von der Wiederholungsrate.	65
8.8	Ergebnisse zu Versuch 2.2: IPI-Latenzzeit von M4 nach A35 in Abhängigkeit von der Wiederholungsrate.	66
8.9	Parameter zu Versuch 3: OpenAMP Verarbeitungszeit auf dem M4.	68
8.10	Ergebnisse zu Versuch 3: OpenAMP Verarbeitungszeit auf dem M4 in Abhängigkeit von der Nachrichtengröße.	68
8.11	Parameter zu Versuch 4.1 und 4.2: Latenzzeit des TTY-Devices.	69
8.12	Ergebnisse zu Versuch 4.1: Latenzzeit von TTY-Write in Abhängigkeit von der Nachrichtengröße.	70
8.13	Ergebnisse zu Versuch 4.2: Latenzzeit von TTY-Read in Abhängigkeit von der Nachrichtengröße.	73
8.14	Ergebnisse zu Versuch 4.3: Tatsächliche Latenzzeit von TTY-Read.	75
8.15	Parameter zu Versuch 4.3 und 4.4: Latenzzeit des TTY-Devices.	76
8.16	Ergebnisse zu Versuch 4.3: Latenzzeit von TTY-Write.	76
8.17	Ergebnisse zu Versuch 4.4: Latenzzeit von TTY-Read.	77
8.18	Parameter zu Versuch 5.1: Latenzzeit des TTY-Devices.	79
8.19	Ergebnisse zu Versuch 5.1: Latenzzeit von TTY-Write in Abhängigkeit von der Wiederholungsrate.	79

8.20	Ergebnisse zu Versuch 5.2: Latenzzeit von TTY-Read in Abhangigkeit von der Wiederholungsrate.	79
8.21	Parameter zu Versuch 6: Datendurchsatz zwischen A35 und M4.	80
8.22	Ergebnisse zu den Versuchen 6.1 bis 6.4: Datendurchsatze zwischen Userspace sowie Kernelspace und M4 in Abhangigkeit von der Systemlast.	80
9.1	Zusammenfassung der Messergebnisse fur die auftretenden Latenz- und Verarbeitungszeiten bei einer Nachrichtengroe von 496 Bytes.	86
10.1	Ergebnisse zu Vorversuch 1: Kernelspace GPIO-Latenzzeit.	91
10.2	Ergebnisse zu Vorversuch 2: M4 GPIO-Latenzzeit.	92

Kapitel 1

Einleitung

1.1 Motivation für heterogene Multiprozessorsysteme

Heutzutage müssen Embedded-Anwendungen performant, preiswert, sicher und energieeffizient sein. Um diese Anforderungen zu erfüllen, wird immer mehr ein heterogenes Multiprozessorsystem (MPSoC) eingesetzt, da es aus mehreren unterschiedlichen Prozessoren aufgebaut ist, die jeweils für gezielte Anwendungen unabhängig voneinander verwendet werden können und dadurch ein breites Leistungsspektrum von Anwendungsfällen in der Industrie abdecken.

Heterogene MPSoC integrieren beispielsweise Prozessoren der Serie ARM Cortex-A, auf denen performanceintensive Anwendungen ausgeführt werden können und nebenbei noch House-Keeping Aufgaben ausgeführt werden. So können Applikationen zum Datenaustausch in die Cloud über Netzwerk oder die Anzeige von Informationen auf einem Human-Machine Interface (HMI) darauf laufen. Zugleich wird für grafikintensive Anwendungen eine Graphical Processor Unit (GPU) für hohe Beschleunigungen bereitgestellt. Zusätzlich steht beispielsweise auf einem heterogenen MPSoC ein Prozessor der Serie ARM Cortex-M zur Verfügung, der als Coprozessor eingesetzt werden kann und harte Echtzeitanforderungen erfüllt, um echtzeitkritische Anwendungen wie Motorsteuerungen oder das Auslesen von Sensordaten mit hohen Wiederholungsraten bereitzustellen. Die alltäglichen sicherheitsrelevanten Zertifizierungen wie Safety Integrity Level (SIL) können ebenso auf diesem Echtzeitprozessor gewährleistet werden, während auf dem selben Chip ein Prozessor der Serie ARM Cortex-A nicht sicherheitsrelevante Anwendungen bereitstellt. Zusätzlich kann über Sicherheitsmechanismen wie Secure-Boot die Systemsicherheit erhöht werden [1, 2].

Die Integration verschiedener Prozessoren auf einen Chip bringen noch weitere Vorteile mit sich, wie beispielsweise die Reduzierung von Stücklisten für Embedded-Geräte, Minimierung der Entwicklungskosten sowie schnellere Entwicklungszyklen [3, S. 78].

1.2 Auftretende Probleme auf heterogenen Multiprozessorsystemen

Auf den integrierten unterschiedlichen Prozessoren auf einem heterogenen MPSoC werden unterschiedliche Betriebssysteme wie Linux und ein Real Time Operating Systems (RTOS) oder Bare-Metal eingesetzt. Wenn mehrere unterschiedliche Betriebssysteme auf einem heterogenen MPSoC ablaufen, wird dies als Asymmetric Multiprocessing (AMP) bezeichnet. Der Einsatz von AMP-Systemen bringt in der Praxis eine Vielzahl von Herausforderungen mit sich. Typi-

sche Problemstellungen sind die Aufteilung des Speichers für die verwendeten Betriebssysteme und das Festlegen des Shared-Memorys, welcher von beiden Prozessoren zum Austausch von Daten genutzt wird. Diese Probleme können als Interprozessorkommunikation (IPC) zwischen den Prozessoren zusammengefasst werden und repräsentiert die erste Problemgruppe.

Die zweite Problemgruppe stellt das Lifecycle-Management (LCM) dar. Es muss festgelegt werden, welcher Prozessor das Starten, Laden und Herunterfahren von Programmen für andere Prozessoren übernimmt. Wichtig ist hierbei, dass der Coprozessor mit unterschiedlichen Firmwares geladen werden kann, ohne andere Prozessoren zum Absturz zu bringen.

1.3 Ziele der vorliegenden Arbeit

Da derzeit noch kein anerkannter, allgemeingültiger und standardisierter Ansatz für die aufgezeigten Problemgruppen existiert und viele Firmen proprietäre Lösungen entwickeln, ist das Ziel dieser Arbeit, einen möglichst standardisierten Ansatz für die IPC und das LCM auf AMP-basierten MPSoC zu finden. Dieses Hauptziel unterteilt sich in folgende drei Teilziele:

1. Darstellung der relevanten Grundlagen und Herausforderungen auf heterogenen MPSoCs.
2. Ein geeigneter Ansatz für die oben genannten Probleme soll auf einem ausgewählten heterogenen MPSoC umgesetzt werden.
3. Basierend auf zwei erarbeiteten Anwendungsszenarien aus der Industrie soll die umgesetzte Lösung auf relevante Aspekte wie Latenzen oder Datendurchsatz evaluiert werden.

1.4 Methodik und Aufbau der vorliegenden Arbeit

Die Methodik und der Aufbau der Arbeit ist Abbildung 1.1 zu entnehmen und wird im folgenden Abschnitt vorgestellt.

Bevor ein Ansatz für das LCM und IPC ausgewählt werden kann, müssen Begriffe wie Multiprozessorsysteme, Symmetric- und Asymmetric Multiprocessing und Herausforderungen auf heterogenen Multiprozessorsysteme erläutert werden. Aufbauend auf dem erarbeiteten Wissen werden drei mögliche AMP-Architekturen auf heterogenen MPSoCs vorgestellt, die basierend auf einer Literaturrecherche ausgewählt wurden.

Anschließend werden Anforderungen an die ausgewählte AMP-Architektur gestellt, um darauf aufbauend ein AMP-System zu entwickeln. Dazu werden mehrere heterogene MPSoCs und vorhandene Lösungsansätze für die LCM und IPC basierend auf den Anforderungen bewertet und diskutiert. Im Hinblick auf die Umsetzung eines AMP-Systems wird ein heterogener MPSoC und ein Lösungsansatz ausgewählt und darauffolgend vorgestellt.

Mit dem erarbeiteten Wissen zum ausgewählten MPSoC und Lösungsansatz für das LCM und die IPC wird ein Konzept erstellt, das im Anschluss umgesetzt wird.

Um das umgesetzte AMP-System zu evaluieren, werden basierend auf zwei möglichen Anwendungsszenarien aus der Industrie wichtige Aspekte abgeleitet, die das umgesetzte AMP-System gewährleisten muss. Im Anschluss wird die Versuchsplanung beschrieben und die einzelnen

Versuche vorgestellt. Danach folgt die Versuchsdurchführung und die Diskussion der Messergebnisse.

Die Arbeit schließt mit einer Zusammenfassung, Diskussion der Ergebnisse und einem Ausblick ab.

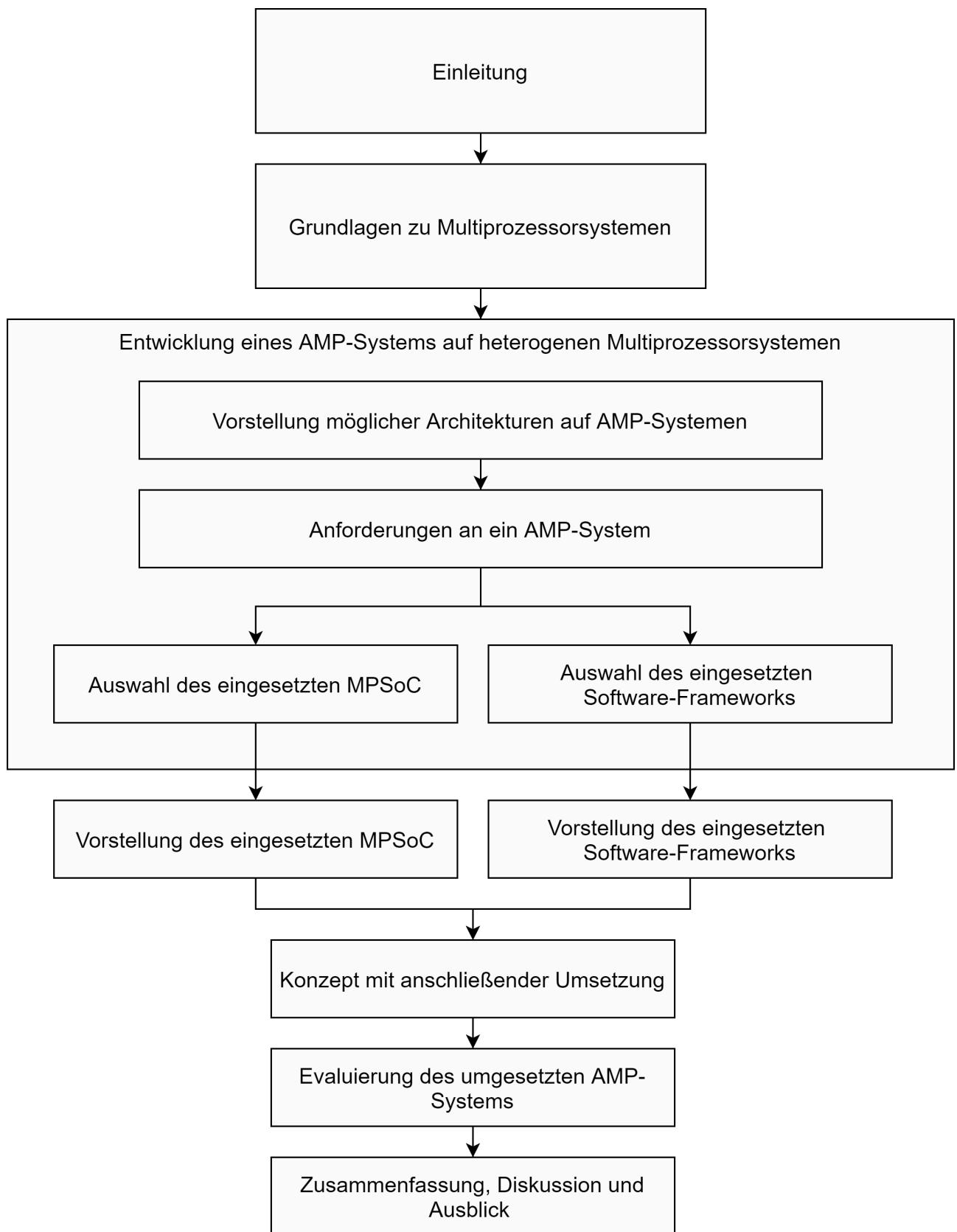


Abbildung 1.1: Aufbau und Methodik der vorliegenden Arbeit.

Kapitel 2

Grundlagen über Multicore- und Multiprozessorsysteme

Dieses Kapitel beschäftigt sich mit den theoretischen Grundlagen von MPSoC, den hierbei auftauchenden Begriffen wie Prozessor, Core und Multi-Core. Im zweiten Teil wird genauer auf die eingesetzten Betriebssysteme auf MPSoC eingegangen und die typischen Herausforderungen vorgestellt, welche beim Einsatz von AMP-Systemen auftreten.

2.1 Definitionen

Da Multiprozessorsysteme und Multicore-Systeme sich erst in den letzten zwei Jahrzehnten verbreitet haben, hat sich bisher noch kein einheitlicher Sprachgebrauch etabliert. Daher sollen grundlegende Begriffe wie Core und Prozessor für den Rahmen der vorliegenden Arbeit klar definiert werden, um eine Verwechslung der Begriffe zu vermeiden und eine klare Definition dieser Begriffe zu benennen [4]:

Definition 1: Ein Prozessor ist eine nicht teilbare Einheit von Rechen- und Hilfseinheiten.

Definition 2: Ein Core ist ein Teil eines Prozessors, der alle notwendigen Komponenten besitzt, die für die Ausführung eines eigenständigen Threads notwendig sind.

Definition 3: Ein Multiprozessorsystem besteht aus mehreren unterschiedlichen Prozessoren, die jeweils aus mehreren Multicores bestehen können und in einem Chip integriert sind.

Definition 4: Wenn mehrere Cores in einem Prozessor integriert sind, wird das hardwareseitig als Chip bezeichnet. In der englischsprachigen Literatur wird dies als Chip Multiprocessor (CMP) bezeichnet. Demzufolge ist ein Multicore-Prozessor ein Prozessor, der mehr als einen Core besitzt und auf dem mehrere Threads gleichzeitig und unabhängig voneinander ausgeführt werden können [5, S. 9f].

2.2 Multicore-Prozessoren

Der Wandel zu Multicore-Prozessoren ist durch die Power Wall entstanden: Für eine höhere Performance muss die Frequenz des Cores erhöht werden, was einen höheren Energieverbrauch erzeugt. Mit einem Multicore-Prozessor kann eine höhere Performance mit einer niedrigeren

Frequenz und einem niedrigeren Energieverbrauch erzielt werden [6, S. 5]. In der Literatur werden verschiedene Multicore-Architekturen vorgestellt und sind in [7] ausführlich behandelt.

Bei einem Multicore-Prozessor teilen sich alle Cores denselben gemeinsamen Speicherbereich (Abb. 2.1). Alle Cores kommunizieren über einen gemeinsam genutzten Bus, was als Uniform Memory Access (UMA) bezeichnet wird. Außerdem teilen sich alle Cores den Zugriff auf gemeinsam genutzte I/O-Geräte über einen Gerätebus. Zusätzlich werden alle Cores eines Prozessors durch ein Betriebssystem verwaltet.

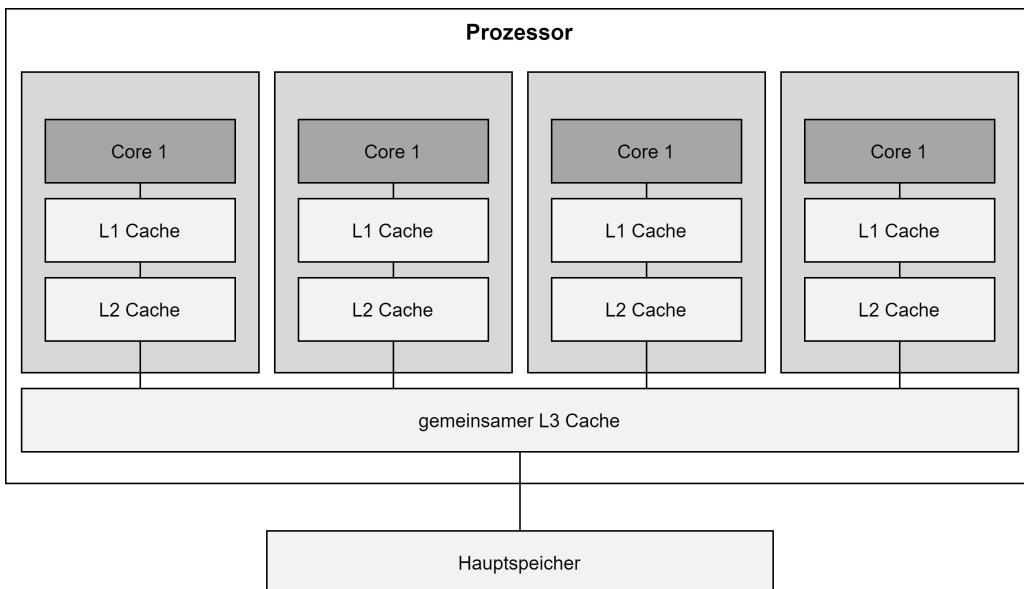


Abbildung 2.1: Aufbau eines Quad-Core-Prozessors. Jeder Core hat Zugriff auf seinen eigenen privaten L1- und L2-Cache. Alle Cores teilen sich einen gemeinsamen L3-Cache und einen gemeinsamen Hauptspeicher (nachgezeichnet nach [8]).

Des Weiteren bestehen Multicore-Prozessoren aus mehreren Cache-Level (Abb. 2.1), die sich in einen privaten L1- und L2-Cache und einem geteilten L3-Cache für alle Cores unterteilen. Diese vorgegebene Anordnung ist nicht fest definiert und hängt von der jeweiligen Multicore-Architektur ab.

Der Grund für die Verwendung der unterschiedlichen Cache-Level und Speicherbereiche sind die verschiedenen Eigenschaften bezüglich Geschwindigkeit sowie Flächen- und Energiebedarf (Tab. 2.1). Dadurch entstehen unterschiedlich eng an einen Core gekoppelte Speichersysteme, die für unterschiedliche Szenarien verwendet und als Speicherhierarchie bezeichnet werden [9, S. 476].

Hierarchie	Speichertyp	Größe	Latenz
CPU-Registerfile	Register	128 B	1
L1	SRAM	32 kB	2-8
L2	SRAM	128 KB	8-32
L3	SRAM	1 MB	32-64
Hauptspeicher	SDRAM	8 GB	128-1024
nicht flücht. Speicher	Flash	256 GB	8000

Tabelle 2.1: Darstellung einer Speicherhierarchie aus Sicht eines Multi-Cores mit Beschreibung der Werte für Speichergöße und Latenz. Die Latenz beschreibt die notwendige Anzahl der Taktraten bei einem CPU-Takt von 1 GHz [10, S.24].

Allerdings führen Multicore-Prozessoren zu neuen Problemen, da immer ein sequenzieller Anteil durch das Amdahls Law vorhanden ist. Somit ist auch eine Trennung von leistungsintensiven und echtzeitkritischen Applikationen nicht möglich, da keine Auslagerung auf separate Prozessoren möglich ist. Aus diesen Gründen stellt das nächste Kapitel MPSoC vor, die unter anderem diese Probleme lösen [11].

2.3 Multiprozessorsysteme

Als System-on-Chip (SoC) werden Systeme bezeichnet, welche aus genau einem Hauptprozessor bestehen und mehrere Subkomponenten wie I/Os, Speicher, und Grafik- und Audiokomponenten beinhalten und gemeinsam in einem Chip verbaut sind (**Definition 5**, [6, S.1-2]).

Ein MPSoC ist aus mindestens zwei oder mehr Prozessoren aufgebaut, die gemeinsam in einem Chip integriert sind. Durch die Vielzahl an möglichen Prozessoren mit ihren eigenen Variationen, existieren viele verschiedene MPSoC Architekturen. Die einzelnen Prozessoren werden anhand ihrer Eigenschaften wie Prozessorarchitektur, Befehlssatz, Taktfrequenz oder Cache-Anordnung klassifiziert [5, 12].

Grundsätzlich werden MPSoCs in zwei Bereiche unterteilt: homogene MPSoCs und heterogene MPSoCs. Homogene MPSoC bestehen aus mehreren Prozessoren mit der gleichen Befehlssatzarchitektur. Sie bieten eine hohe Flexibilität und Skalierbarkeit an, jedoch mit einer geringeren Energieeffizienz. Typische Anwendungsfelder für homogene MPSoCs sind Serversysteme, Spielekonsolen oder Cloud-Computing [6, S. 7][13, 14].

Heterogene MPSoCs hingegen bestehen aus mehreren Prozessoren mit unterschiedlichen Befehlssatzarchitekturen, die für gezielte Anwendungen wie aufwendige Grafikanwendungen oder echtzeitkritische Anwendungen geeignet sind. Durch die Aufteilung der Aufgaben auf verschiedene Prozessoren wird eine höhere Bandbreite von Anforderungen abgedeckt wie ein guter Trade-Off zwischen Energieeffizienz und Performance, Systemsicherheit, Reduzierung der Stückliste oder die Reduzierung von EMV-Störungen [15, S. 6][16–18].

Zusätzlich ist die Kommunikationsstruktur von MPSoC in nachrichtengekoppelte und speichergekoppelte Systeme unterteilt. Nachrichtengekoppelte Systeme wie Network-On-Chip (NoC) haben wenige Abhängigkeiten untereinander und bieten deshalb eine große Flexibilität bei der Entwicklung von Systemen. Die Kommunikation erfolgt über ein Message-Passing-Interface (MPI). Bei einem speichergekoppelten System teilen sich alle Prozessoren einen gemeinsamen

Speicher. Die Kommunikation und Synchronisation erfolgt über gemeinsame Variablen [10, 12].

2.3.1 Heterogene Multiprozessorsysteme

Die Besonderheit von heterogenen MPSoC ist die Ausführung von mehreren Prozessoren mit unterschiedlichen Befehlssatzarchitekturen, die auf deren speziellen Aufgabenbereichen optimiert sind. Die einzelnen eigenständigen Prozessorsysteme verfügen über einen privaten Speicher On-Chip und einen vom Speicher-Controller verwalteten öffentlich geteilten zugänglichen Speicher, auf den alle Prozessoren Zugriff haben [13].

Die Kommunikation zwischen den einzelnen Komponenten auf einem MPSoC erfolgt über einen System- und Peripheriebus (Abb. 2.2). Die Vermittlung zwischen den beiden Bussystemen erfolgt über eine Bridge. Die Übertragungsgeschwindigkeit vom Systembus ist höher als die vom Peripheribus, da an diesem geschwindigkeitskritische Komponenten wie Prozessoren oder Speicher-Controller angeschlossen sind. Bekannte Bussysteme sind der Advanced Microcontroller Bus Architecture (AMBA) von ARM oder der CoreConnect von IBM. Durch die Bussysteme aufgestellten Kommunikationswege können alle Komponenten auf geteilten Speicher und auf Peripheriegeräte zugreifen [12].

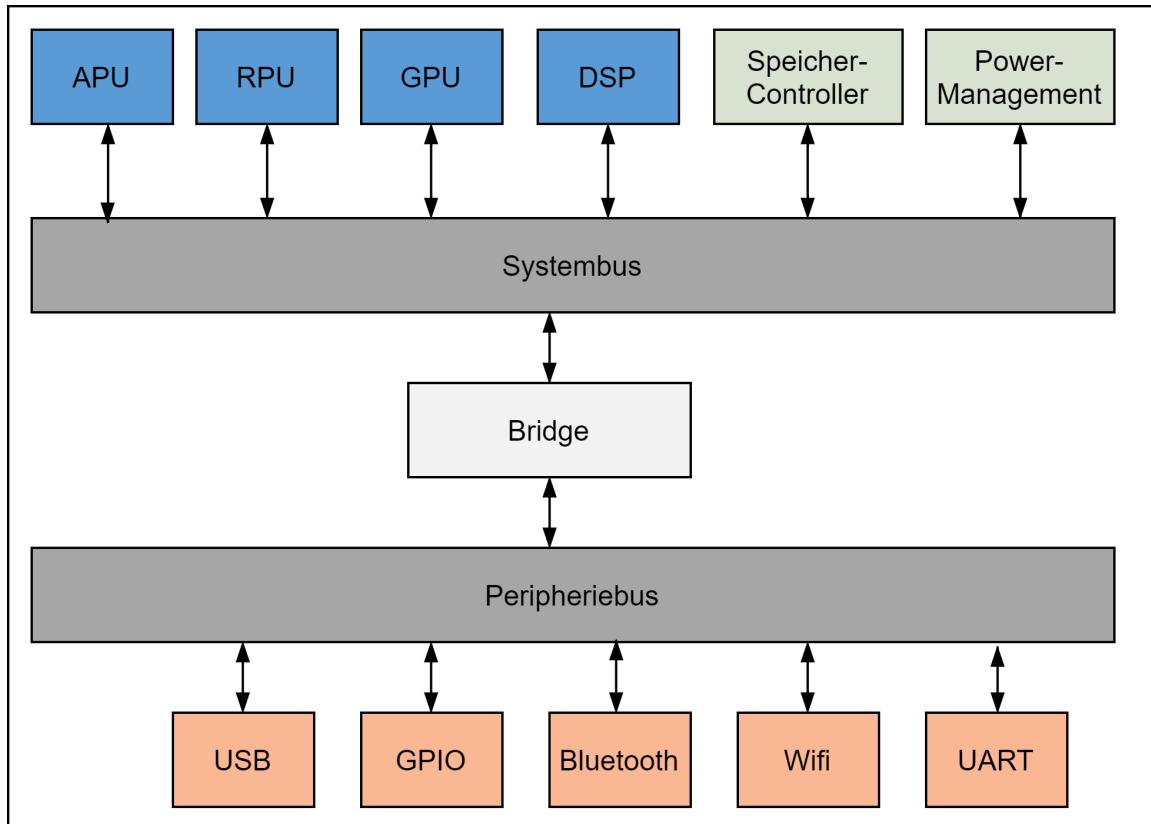


Abbildung 2.2: Aufbau einer MPSoC Architektur. Der Aufbau unterteilt sich in Bussysteme (graue Kästen), an denen Komponenten wie Prozessoren (blaue Kästen), Speicher-Controller und Power-Management (grüne Kästen) oder Peripherieeinheiten (orangene Kästen) angeschlossen sind. Die schwarzen Pfeile stellen die Datenaustauschrichtung dar (umgezeichnet nach [3, S. 175]).

2.3.2 Prozessorvielfalt auf heterogenen Multiprozessorsystemen

Der folgende Abschnitt stellt die wichtigsten Prozessorarten und ihre Anwendungszwecke auf MPSoCs vor, um einen Überblick für die integrierten Komponenten zu geben, der anschließend für die Entwicklung des AMP-Systems notwendig ist [19].

Eine Application Processor Unit (APU) beinhaltet in der Regel Prozessoren aus der ARM Cortex-A Serie, auf denen ein Betriebssystem wie Linux läuft. Das Einsatzgebiet sind nicht echtzeitkritische Anwendungen, wie die Darstellung von Informationen auf Displays, die Kommunikation in die Cloud über ein Netzwerk-Interface oder die Verwaltung der Geschäftslogik über Datenbanken.

Sobald echtzeitfähige Anwendungen notwendig werden, reicht eine APU nicht mehr aus, da sie kein deterministisches Verhalten aufweisen. Für diesen Einsatz wird eine Real Time Processor Unit (RPU) verwendet, die deterministisches Verhalten und die Ausführung von niedrigen Latenzen bei Interrupts und Speicherzugriffen gewährleisten.

Häufig reicht die Performance bei APU in bestimmten Situationen für die Bild- oder Videoverarbeitung nicht aus, weshalb Coprozessoren hierfür verwendet werden. Eine GPU besitzt eine ausgerichtete Hardware für die Beschleunigung von Bildverarbeitungen und ist aus sehr vielen kleinen Cores (Anzahl über 1000) aufgebaut und speziell für die gleichzeitige Ausführung von mehreren Tasks entwickelt worden. Sie führen Funktionen wie Texture Mapping, Image Rotation, Translation oder Shading aus.

Neben der GPU wird noch die Video Processing Unit (VPU) für das Encoding und Decoding von Videos verwendet. Eine industrielle Anwendung mit einer GPU und einer VPU wird in [20] beschrieben.

Sobald kontinuierliche Bearbeitungen von digitalen Signalen zum Einsatz kommen, bietet sich ein Digital Signal Processor (DSP) an, der für aufwendige Signalverarbeitungen z.B. Echounterdrückung und Datenkomprimierung geeignet sind.

Field Programmable Gate Array (FPGA) sind integrierte und veränderbare Schaltkreise, in denen logische Schaltungen aufgebaut werden, die zwar keine klassischen Prozessoren darstellen, dennoch in heterogenen MPSoC aufzufinden sind. Sie kombinieren Flexibilität und Programmierbarkeit von Software mit einer hohen Geschwindigkeit durch die hardwarebasierte Umsetzung. So lassen sich Echtzeitanwendungen über komplexe Algorithmen bis zur digitalen Signalverarbeitung in einem FPGA realisieren. Allerdings führen sie dadurch zu einem hohen Energieverbrauch und einem hohen Kostenfaktor.

2.4 Betriebssysteme auf heterogenen Multiprozessorsystemen

Zur Anfangszeit von Embedded-Systemen war es üblich, eine Firmware zu entwickeln, die direkt auf der Hardware ausgeführt wurde. Hierbei existierte keine Softwareabstraktionsschicht, die für die Abstraktion der Hardware verantwortlich war. Die Art von Programmierung wird als Bare-Metal bezeichnet. Allerdings wurden in den letzten Jahrzehnten die Prozessorsysteme immer umfangreicher an Funktionalitäten. Für die Kapselung dieser Funktionalitäten wurden

Betriebssysteme eingeführt, die eine Abstraktionsebene und viele Vorteile für Entwickler bereitstellen. So bieten Betriebssysteme standardisierte Systemschnittstellen, wie die Schnittstelle Portable Operating System Interface (POSIX) und Driver für sehr viele unterschiedliche Geräte an, die eine deutliche Zeiteinsparung an Entwicklungszeit gewährleisten. Die Ansteuerung eines Displays muss dadurch nicht mehr selbst entwickelt werden, sondern das Betriebssystem stellt dafür bereits Driver bereit. So ist die Verwendung eines Betriebssystems auf vielen unterschiedlichen Hardwareumgebungen kein großes Hindernis. Zusätzlich muss sich der Entwickler nicht mit der Ressourcenverwaltung oder Task-Scheduling auseinandersetzen, da dies Aufgaben eines Betriebssystems sind [3, S. 371-373].

Betriebssysteme werden in drei Kategorien eingeteilt: General Purpose Operating Systems (GPOS), die für die meisten Anwendungen wie PCs, Serverlandschaften und Handys ausreichen. Konkrete Beispiele sind Windows und Linux.

Für Embedded-Systeme sind GPOS nicht geeignet, weil sie zu viel Ressourcen in Anspruch nehmen, die in Embedded-Systemen nur in eingeschränkter Form vorhanden sind. Daher werden häufig maßgeschneiderte Linux Embedded-Betriebssysteme mit Build-Systemen wie Yocto oder PTXDist erstellt (Kategorie zwei), die genau für die geforderten Anwendungsfälle geschnitten sind und nur das notwendigste bereitstellen.

Die letzte Kategorie sind die RTOS, die ein deterministisches Verhalten mit maßgeschneiderten Scheduling-Arten, statischer Speicherzuweisung und die Einfachheit des Betriebssystems sowie der laufenden Prozesse garantieren. Das Ziel eines RTOS ist nicht einen hohen Daten durchsatz zu erreichen oder unbedingt schnelles Handeln, sondern das Einhalten der gesetzten Zeitschränken und rechtzeitig als auch vorhersehbar auf eine bestimmte Aufgabe zu reagieren. Viele Embedded-Anwendungen fordern genau solche Eigenschaften beispielsweise eine Airbagsteuerung im Auto oder eine Fluglagesteuerung. Eingeteilt wird ein RTOS in zwei Arten [21]: **Weiche und harte Echtzeitsysteme**.

Harte Echtzeitsysteme

Bei harten Echtzeitsystemen müssen die gesetzten Zeitschränken zu jedem Zeitpunkt eingehalten werden. Wenn diese Regel verletzt wird, hat das Ergebnis keinen Nutzen für das System. Die hierbei festgelegte Zeit, auch als Worst Case Execution Time (WCET) bezeichnet, muss berechnet werden können sowie beweisbar sein. Da die meisten RTOS-Systeme die Anforderungen nicht erfüllen können, wird die Bare-Metal Programmierung bevorzugt eingesetzt.

Weiche Echtzeitsysteme

Bei einem weichen Echtzeitsystem darf die WCET verletzt werden. Jedoch hat das gelieferte Ergebnis eine gewisse Bedeutung für das System. Es hängt vom Anwendungsfall ab, inwieweit der Nutzen des Ergebnisses unter dem verspäteten Auftreten leidet. Eine RPU kann sowohl harte als auch weiche Echtzeit erreichen. Auf einer APU ist lediglich weiche Echtzeit durch Anpassungen im Linux Kernel möglich. Eine sehr weitverbreitete Möglichkeit ist der Preempt-RT-Patch. Der Patch konzentriert sich darauf, den Linux-Kernel fast vollständig unterbrechbar zu gestalten, da dies ein wichtiges Kriterium für Echtzeit ist.

2.5 Einsatz von unterschiedlichen Betriebssystemen auf heterogenen Multiprozessorsystemen

Auf heterogenen MPSoC können die beschriebenen Betriebssysteme in Kombination eingesetzt werden. Sie werden unterteilt in Symmetric Multiprocessing (SMP) und Asymmetric Multiprocessing (AMP). Beide Möglichkeiten werden in den folgenden Abschnitten näher betrachtet. Sie sind für das allgemeine Verständnis von Betriebssystemen auf heterogenen MPSoC und somit hinsichtlich der Durchführung dieser Arbeit relevant.

2.5.1 Symmetric Multiprocessing

Bei SMP wird genau ein Betriebssystem auf allen vorhandenen Cores eines Prozessors ausgeführt (Abb. 2.3). Alle Cores unterliegen der gleichen Prozessorarchitektur sowie der gleichen Instruction Set Architecture (ISA), teilen sich alle denselben Speicherbereich und teilweise die gleichen Cache-Level (Abb. 2.1) und haben gemeinsamen Zugriff auf I/O-Geräte. Das Betriebssystem verwaltet alle Ressourcen und verteilt die auszuführenden Prozesse auf die zur Verfügung stehenden Cores, die alle gleich behandelt werden. Jeder Prozess kann auf einem beliebigen Core ohne jegliche Änderungen oder Einschränkungen ausgeführt werden. Wenn das Betriebssystem überlastet ist, sind alle auszuführenden Prozesse gleichermaßen davon betroffen, wodurch ein Determinismus auf SMP-Systemen nicht erreicht werden kann. SMP-Systeme nutzen alle Cores der gegebenen Multicore-Architektur, um eine maximale Performance zu erreichen. In der Regel wird ein GPOS wie Linux eingesetzt, das SMP-fähig ist [3, S. 378][22, S. 96][23, S. 7][24].

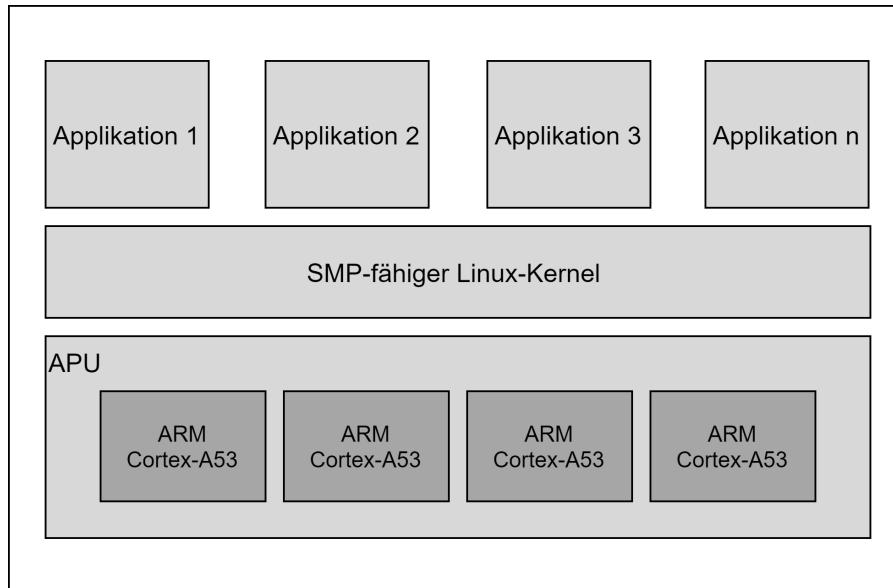


Abbildung 2.3: Darstellung eines SMP-fähigen Linux, das auf einer APU mit einem ARM Cortex-A53 Multicore mit 4 Cores läuft. Es kann eine beliebige Anzahl von Anwendungen auf dem Linux laufen, die sich gemeinsam die Ressourcen der APU teilen (umgezeichnet nach [25, S. 59]).

2.5.2 Asymmetric Multiprocessing

Bei AMP läuft mehr als ein Betriebssystem auf einem oder auf mehreren Prozessoren mit unterschiedlicher ISA unabhängig voneinander. Genau hierfür wird der Einsatz von heterogenen

MPSoC interessant, da sie mehrere unterschiedliche Prozessoren für spezielle Aufgaben wie Multimedia, Echtzeit oder Bildverarbeitung bereitstellen. So kann auf der APU auf allen Cores ein Linux verwendet werden, während auf dem ARM Cortex-M4 ein RTOS läuft (Abb. 2.4).

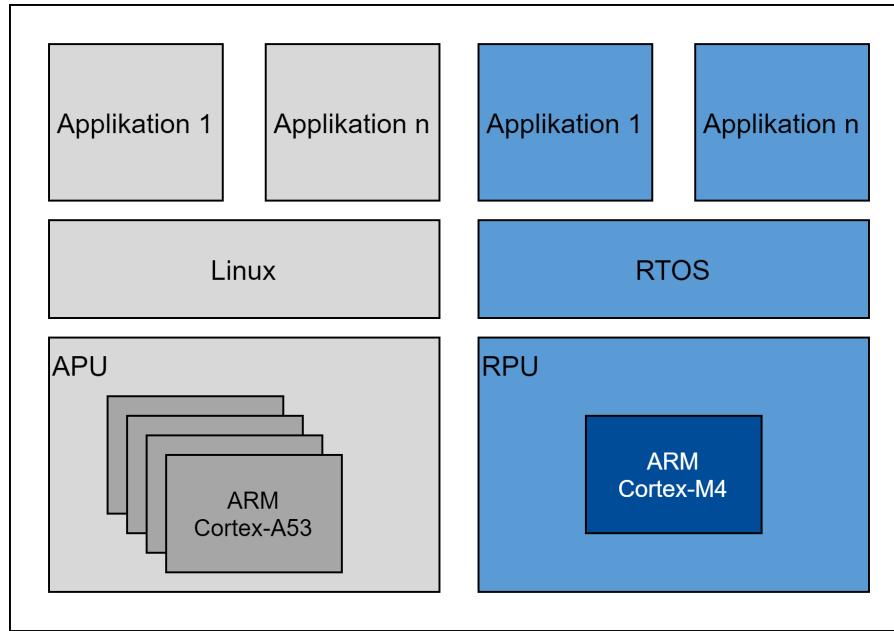


Abbildung 2.4: Aufbau eines möglichen AMP-Systems. Ein Linux verwaltet alle Cores der APU (grauer Bereich), in der ein ARM Cortex-A53 integriert ist. Ein RTOS läuft auf der RPU ab, in der ein ARM Cortex-M4 integriert ist (blauer Bereich).

Alle Betriebssysteme teilen sich den öffentlich zugänglichen Speicher, aber haben zugleich in der Regel noch einen privaten Speicher. Der Zugang auf die Ressourcen wie I/O-Geräte teilen sich auch alle Betriebssysteme. Durch Mechanismen wie IPC, Message-Passing und Shared-Memory, können die Betriebssysteme miteinander kommunizieren und Daten untereinander austauschen. Der Zugriff und die Synchronisation muss hierbei geregelt werden. Häufig werden in der Praxis SMP und AMP miteinander auf MPSoC kombiniert, um den Energieverbrauch zu optimieren und eine bessere Performance zu erreichen [22, S. 44-45].

AMP-Systeme werden in zwei Klassen nach [19, S. 336] unterteilt:

Unsupervised AMP

Ein Unsupervised AMP (uAMP) ist für Anwendungen geeignet, in denen keine strikte Isolierung von Systemressourcen zwischen den beteiligten Betriebssystemen erforderlich ist. Die Betriebssysteme laufen nativ, unabhängig und gleichwertig voneinander auf den jeweiligen Prozessor auf dem System. Es ist keine unabhängige Systemkomponente vorhanden, die den Zugriff der Betriebssysteme auf Systemressourcen regelt. So übernimmt der Anwender die Verantwortung mit dem fehlerfreien Umgang der Systemressourcen.

Supervised AMP

Ein Supervised AMP (sAMP) wird für Anwendungen eingesetzt, in denen eine strikte Trennung von Systemressourcen zwischen den beteiligten Betriebssystemen benötigt wird. Dies wird durch die Virtualisierung von Betriebssystemen ermöglicht, die in Gastbetriebssystemen

ablaufen und von einem Hypervisor verwaltet und definiert werden. Ein bekannter Hypervisor im Embedded-Bereich ist der Xen Hypervisor [26]. Durch einen Hypervisor laufen die Betriebssysteme immer noch voneinander unabhängig ab. Allerdings regelt der Hypervisor den gesamten Zugriff auf das System, wodurch kein Betriebssystem auf Ressourcen zugreifen kann, die von einem anderen Betriebssystem verwendet werden. Mit der Verwendung eines Hypervisors entsteht jedoch eine zusätzliche Komplexität im System. Außerdem kann ein Hypervisor nur auf einem homogenen Prozessor wie einer APU eingesetzt werden und nicht auf einer RPU, da diese nicht die notwendigen Hardwarekomponenten wie eine Memory Management Unit (MMU) und auch kein SMP-fähiges Betriebssystem bereitstellt. Ein weiterer negativer und nicht zu vernachlässigender Faktor ist die Beeinträchtigung der Performance des Systems bei der Verwendung eines Hypervisors.

2.6 Herausforderungen auf AMP-Systemen

Die Verwendung von mehreren Betriebssystemen auf unterschiedlichen Prozessoren und der hierdurch entstehende Synchronisationsaufwand stellt einige Herausforderungen und eine nicht zu unterschätzende Komplexität auf heterogenen MPSoC dar, die in den folgenden Abschnitten näher betrachtet werden.

2.6.1 Speichermanagement

Die Bereitstellung, Zuteilung und Trennung von Speicherbereichen für Subsysteme (z. B. APU, RPU, GPU) ist ein wichtiger Aspekt, welcher beachtet werden muss. Zugleich muss sichergestellt werden, dass die einzelnen Subsysteme keinen Zugriff auf die Speicherbereiche von anderen Subsystemen haben. Dies könnte ansonsten instabile Systemzustände verursachen oder im schlimmsten Fall das gesamte System zu Absturz bringen. In Tabelle 2.2 ist eine mögliche Aufteilung der Subsysteme in Speicherbereiche eines heterogenen MPSoC dargestellt.

Üblicherweise verwenden APUs eine MMU, die konfiguriert werden muss, damit die APU nur auf ihre zugeteilten Speicherbereiche zugreifen darf. Hierzu führt sie Virtuelle Adresse (VA) und Physikalische Adresse (PA) ein. VA sind Adressen aus Sichtweise einer Applikation. PA sind tatsächliche physikalische Adressen, auf denen Komponenten (Tabelle 2.2) abgebildet sind. Die MMU übernimmt hierbei die Adressübersetzung.

Eine RPU besitzt in der Regel keine MMU sondern eine Memory Protection Unit (MPU), die für das Speichermanagement verantwortlich ist.

Komponente	Speicheradresse	Größe	Beschreibung
APU	0x00000 0000	64 MB	Ein SMP-Linux
RPU	0x00400 0000	64 MB	FreeRTOS
GPU	0x00800 0000	64 MB	Grafikeinheit
IO-Geräte	0x00C00 0000	64 MB	USB, Bluetooth, ADC
DDR-Speicherbereich	0x01000 0000	8 GB	Zugriff von allen Subsystemen
Geteilter Speicherbereich	0x21000 0000	64 MB	Geteilter Speicher für mehrere Subsysteme

Tabelle 2.2: Beschreibung einer möglichen Speicheraufteilung eines heterogenen MPSoC.

2.6.2 Ressourcenmanagement

Neben dem Speichermanagement muss noch festgelegt werden, welche Subsysteme auf welche Ressourcen Zugriff haben. So muss definiert werden, ob die APU Zugriff auf ein ausgewähltes IO-Gerät hat oder die RPU darauf zugreifen kann. Wenn beide Subsysteme gleichzeitig auf dasselbe IO-Gerät zugreifen, entsteht ein undefiniertes Verhalten. Daher ist es empfehlenswert, ein Subsystem auf einem MPSoC zu integrieren, welches den Zugriff von unterschiedlichen Subsystemen auf Ressourcen wie IO-Geräte regelt.

2.6.3 Systemboot-Management

Durch die vielen Subsysteme auf heterogenen MPSoC ist eine große Anzahl von Boot-Stages gegeben, die zu einer höheren Systemkomplexität führen. So muss geklärt werden, welches Subsystem die Bestromung und Konfiguration der notwendigen Subsysteme einschließlich der Clocks übernimmt. Außerdem wird ein Boot-Image benötigt, das alle notwendigen Firmwares beinhaltet, um alle gewünschten Subsysteme beim Systemstart in der gewünschten Bootreihenfolge zu starten. In der Regel wird hierfür ein dediziertes Subsystem vom jeweiligen MPSoC bereitgestellt, welches die Verantwortung für die Extraktion des Boot-Images, das Laden der Firmwares und das Starten der Prozessoren übernimmt.

Innerhalb der Subsysteme wie zum Beispiel der APU, muss geklärt werden, welche Bootloader-Stages verwendet werden. Auch das Laden des Device-Tree oder Linux-Kernel muss durchgeführt werden. Weiterhin muss beim Systemboot sichergestellt werden, dass die parallel laufenden Bootprozesse der einzelnen Subsysteme sich nicht gegenseitig blockieren.

2.6.4 Lifecycle-Management

Neben dem Systembootmanagement zur Systemstartzeit ist das LCM zur Laufzeit ein wichtiges Thema, da es sichere Updates zur Laufzeit auf einzelne Subsysteme wie der RPU ermöglicht, ohne das gesamte System updaten zu müssen. Wichtig ist hierbei, dass unterschiedliche Firmwares auf die Subsysteme bzw. Coprozessoren geladen werden können, ohne andere Subsysteme durch diesen Prozess zu beeinflussen. So muss der Coprozessor zuerst gestoppt werden können, bevor er mit einer neuen Firmware erneut gestartet werden kann. Es muss definiert werden, welcher Prozessor der Master und welcher Prozessor die Rolle des Coprozessors übernimmt.

2.6.5 Interprozessorkommunikation

Sobald mehrere Prozessoren innerhalb eines MPSoCs laufen, ist eine IPC unausweichlich, um Daten zwischen ihnen auszutauschen. Dies geschieht in der Regel über Shared-Memory sowie Interprozessor-Interrupt (IPI). Über eine IPI kann ein Prozessor einen anderen Prozessor über neue Informationen im Shared-Memory benachrichtigen. Auf einem MPSoC werden dazu Messaging Units (MUs) verwendet. Der eingesetzte IPI muss bei dem geteilten Interrupt-Controller, welcher als Hardwarekomponente auf dem MPSoC integriert ist, registriert und freigeschalten werden.

2.6.6 Systemsicherheit

Sicherheit wird in Safety und Security unterteilt. Safety betrachtet den Schutz vor Risiken durch unbeabsichtigte Fehler, Störungen oder Ausfälle. Security betrachtet den Schutz vor be-

absichtigen Fehlern. Im Folgenden wird die Sicherheit im Kontext von Security verwendet [27].

Mit der beschriebenen Möglichkeit zur Erhöhung der Systemsicherheit durch sAMP (siehe 2.5.2) können zwar Betriebssysteme isoliert voneinander mit eingeschränkten Zugriff auf Systemressourcen laufen. Allerdings bietet sAMP keinen Mechanismus für Secure Boot, der Funktionen für die Authentifizierung von sicheren Bootimages oder die Signierung von einzelnen Applikationen wie Kernel-Modulen bereitstellt.

Um Secure Boot bereitzustellen bieten die verschiedenen Hersteller unterschiedliche Lösungsansätze an. Ein bekannter Lösungsansatz von ARM ist die ARM TrustZone, welche über eine sogenannte ARM Trusted Firmware (ATF) einen Secure Boot ermöglicht.

2.6.7 Debugging

Während der Plattformentwicklung, Durchführung von Tests oder der allgemeinen Softwarefehlersuche ist Debugging das Mittel zur Lösung. So muss auf MPSoC eine Möglichkeit bestehen, Subsysteme wie Bootloader oder den Linux-Kernel auf einer APU oder ein laufendes RTOS auf einer RPU zu debuggen. In der Regel werden hierfür dedizierte Hardware-Schnittstellen wie eine JTAG-Schnittstelle zur Verfügung gestellt. Über diese Schnittstelle hat der Anwender Zugriff auf die Prozessoren und auf den Systembus, um Peripherieeinheiten oder Speicherbereiche auszulesen oder sich in einen laufenden Prozess einzuhängen.

2.6.8 Fazit

Die letzten Abschnitte haben die Herausforderungen von AMP-Systemen beschrieben und es zeigte sich, dass das Speichermanagement durch eine MMU oder MPU gelöst werden kann. Für das Debugging stehen notwendige dedizierte Hardware-Schnittstellen bereit. Allerdings existiert kein allgemeingültiger Ansatz für das Lösen der Herausforderungen von Ressourcen-, Systemboot-, Lifecycle-Management, der IPC und der Systemsicherheit. Zwar stechen Mechanismen wie die ARM TrustZone hervor, allerdings sind die aufgelisteten Herausforderungen für jeden heterogenen MPSoC einzeln zu betrachten und zu bewerten.

Kapitel 3

Entwicklung eines AMP-Systems auf heterogenen Multiprozessorsystemen

Die Vielzahl an möglichen Prozessoren auf heterogenen MPSoC und die daraus entstehenden Architekturen bieten viele Lösungsansätze für unterschiedliche Szenarien. Nicht nur die eingesetzte Hardware stellt umfangreiche Lösungsansätze bereit, sondern auch die eingesetzten unterschiedlichen Betriebssysteme auf mehreren Prozessoren, die zu den beschriebenen Herausforderungen auf AMP-Systemen führen. Da diese Arbeit den Schwerpunkt auf die softwareseitige Herausforderungen von AMP-Systemen setzt, werden die möglichen hardwareseitigen heterogenen MPSoC-Architekturen nicht näher betrachtet.

Das folgende Kapitel beschreibt drei mögliche Architekturen für AMP-Systeme auf einem heterogenen MPSoC. Anschließend werden Anforderungen für die ausgewählte Architektur im Hinblick auf die auftretenden Probleme auf einem AMP-System definiert. Das zu entwickelnde System wird in dieser Arbeit als Mixed Mode AMP (MMAMP) bezeichnet. Aus den definierten Anforderungen an das MMAMP repräsentieren die letzten beiden Abschnitte die Auswahl der Hardware und die Auswahl des eingesetzten AMP-Frameworks für das LCM und die IPC.

3.1 Architekturen für AMP-Systeme

In der Literatur sind viele Architekturen für AMP-Systeme vertreten, die für unterschiedliche Anwendungszwecke geeignet sind. Die folgenden drei Architekturen wurden bei der Recherche besonders häufig aufgefunden und werden deshalb vorgestellt [15, 28–31]:

1. Mixed Critical System

Ein Mixed Critical System (MCS) ist für Anwendungen mit unterschiedlichen Sicherheitsleveln in der Industrie (Automobil, Luftfahrt, Schienenfahrzeuge) geeignet. Abhängig vom Sicherheitslevel werden auf einer APU Anwendungen ohne Sicherheitslevel ausgeführt, während auf einer RPU Anwendungen mit höheren Sicherheitsleveln ablaufen, die sicherheitskritisch sind. Diese Anwendungen fallen in den Bereich von harter Echtzeit und müssen zertifiziert werden. Eine IPC zwischen den Prozessoren ist nicht erlaubt, da sonst die Anforderungen an sicherheitskritischen Anwendungen verletzt werden. Zusätzlich wird bei MCS oft sAMP über einen Hypervisor für eine höhere Isolierung zwischen Anwendungen auf einer APU eingesetzt.

2. Low Power System

Ein Low Power System (LPS) fordert eine möglichst effiziente Verwaltung der Batterie, um lange Batterielaufzeiten zu garantieren. Die Anwendungen solcher Systeme unterteilen

sich in sehr rechenintensive Aufgaben (Ausführung von Spielen) sowie in kontinuierlich laufende Aufgaben (Spracherkennung). Für rechenintensive und kontinuierliche Aufgaben den gleichen Prozessor einzusetzen, ist für die Erreichung von langen Batterielaufzeiten nicht sinnvoll. Daher werden energieeffiziente Prozessoren für kontinuierliche Aktivitäten eingesetzt, während performante Prozessoren für rechenintensive Anwendungen eingesetzt werden. Diese Architektur wird oft im Consumer-Bereich für Smartphones oder Smartwatches eingesetzt. So kann der kontinuierlich laufende Prozessor bei Bedarf den performanten Prozessor starten und stoppen.

3. Coprocessor System

Ein Coprocessor System (CS) zeichnet sich durch die Auslagerung von gezielten Anwendungen auf andere Prozessoren (z.B. GPU, DSP, RPU) aus. Dadurch werden die ausgewählten Anwendungen auf Prozessoren mit den besten Voraussetzungen ausgelagert, um optimale Bedingungen für die Ausführung zu garantieren. So ist es nicht sinnvoll, eine echtzeitkritische Anwendung auf einer APU ablaufen zu lassen, sondern dafür einen RPU einzusetzen. In der Industrie kann diese Architektur für Prozessdatenüberwachung oder Hardware-in-the-Loop (HiL) eingesetzt werden, in denen eine IPC und das LCM von großer Bedeutung sind (Kap. 8.1).

Das Ziel dieser Arbeit ist es, einen Lösungsansatz für die Probleme LCM und IPC auf AMP-Systemen zu finden (Kap. 1.3). Hierfür ist die Kommunikation zwischen den Prozessoren eine wichtige Voraussetzung. Aufgrund der Vermeidung der IPC wegen hohen sicherheitskritischen Anforderungen wird keine MCS-Architektur eingesetzt. Ein LPS ist auf eine lange Batterielaufzeit ausgelegt und der Schwerpunkt liegt dabei nicht bei einem hohen und kontinuierlichen Datenaustausch zwischen den Prozessoren und wird deswegen ausgeschlossen. Die CS-Architektur setzt ihren Schwerpunkt auf die Auslagerung von Aufgaben. Hierbei ist es wichtig, die Ergebnisse der ausgelagerten Aufgaben zu erhalten, wodurch ein stetiger Datenaustausch zwischen den Prozessoren erforderlich ist. Der Einsatz von LCM kann bei allen drei aufgezeigten Architekturen vielfältig für unterschiedliche Szenarien eingesetzt werden. Wegen den definierten Zielen der Arbeit (Kap. 1.3), einen Lösungsansatz für das LCM und die IPC zu finden, baut das zu entwickelnde System MMAMP auf der CS-Architektur auf, die ihren Schwerpunkt auf eine kontinuierliche Datenkommunikation zwischen den Prozessoren setzt und durch eine LCM flexible Anwendungsszenarien ermöglicht.

3.2 Anforderungen an MMAMP

Dieser Abschnitt beschreibt die Anforderungen an das zu entwickelnde MMAMP AMP-System, welche in allgemeine und hardware- sowie softwareseitige Anforderungen unterteilt werden.

3.2.1 Allgemein

Anforderung A.1: Ausführliche Dokumentation in Form von Application Notes und Foren und die Bereitstellung von BSPs für Linux und eine bestehende Portierung für FreeRTOS.

3.2.2 Hardware

Anforderung H.1: In dem verwendeten MPSoC ist eine APU für performancerelevante und grafikintensive Anwendungen und eine RPU für echtzeitkritische Anwendungen integriert.

Anforderung H.2: Der MPSoC bietet Mechanismen für das Systemboot-, Ressourcen- und Speichermanagement an.

Anforderung H.3: Eine Verfügbarkeit von 15 Jahren wird gewährleistet.

Anforderung H.4: Der MPSoC soll Sicherheitsmechanismen für Secure Boot und für die Validierung von Images bereitstellen.

3.2.3 Software

Anforderung S.1: Auf der APU wird Linux eingesetzt.

Anforderung S.1: Als RTOS soll FreeRTOS auf der RPU eingesetzt werden.

Anforderung S.2: Auf der APU wird uAMP eingesetzt.

Anforderung S.3: Für das LCM und die IPC soll ein standardisierter Ansatz umgesetzt werden, der auf vielen unterschiedlichen Plattformen zum Einsatz kommen kann.

Anforderung S.4: Die APU übernimmt die Aufgaben des LCM.

Anforderung S.5: Echtzeitkritische Aufgaben sollen von der APU an die RPU delegiert werden.

Anforderung S.6: Der Implementierungsaufwand des ausgewählten Ansatzes soll im Rahmen des verfügbaren Zeitraums von 6 Monaten umsetzbar sein.

Anforderung S.7: Der ausgewählte Lösungsansatz soll ein Open-Source Projekt sein.

3.3 Auswahl des eingesetzten Multiprozessorsystems

Für die Auswahl des eingesetzten MPSoCs wurden folgende Plattformen betrachtet:

- STMicroelectronics STM32MP153 [32–34]
- Zynq UltraScale+ MPSoC ZCU104 [35–37]
- Variscite VAR-SOM-MX8X basierend auf dem NXP i.MX8X MPSoC [38, 39]

Die in Tabelle 3.1 genannten Kriterien werden genauer erläutert, um eine Entscheidungsfindung für den ausgewählten MPSoC nachvollziehbar darzustellen.

MPSoC	1. STM32MP153	2. Zynq UltraScale+	3. VAR-SOM-MX8X
Anforderung H.1	-	+	+
Anforderung H.2	-	+	+
Anforderung H.3	-	+	+
Anforderung H.4	0	+	+
Anforderung A.1	+	+	+
<i>Summe</i>	-	+	+

Tabelle 3.1: Bewertung der drei ausgewählten heterogenen MPSoC-Plattformen. Weitere Erläuterungen siehe Text.

Zeichenerklärung: - : Mehr Nach- als Vorteile 0 : Ausgeglichene Lösung + : Mehr Vor- als Nachteile.

- **Anforderung H.1: Performancerelevante APU und echtzeitfähige RPU**

Dieser Unterpunkt beinhaltet die Untersuchung der eingesetzten Prozessoren auf den MPSoC.

1. Auf dem STM32MP153 ist ein DualCore Cortex-A7 mit 800 MHz und ein M4 mit 209 MHz integriert. Aufgrund der schwachen Taktrate der APU kann nicht sichergestellt werden, dass die Leistung für rechenintensive Anwendungen ausreicht.
2. Es wird sowohl eine APU mit einem QuadCore Cortex-A53 mit bis zu 1,5 GHZ als auch eine RPU mit zwei Cortex-R5 bis zu 600 MHz bereitgestellt, welche die Anforderungen erfüllen. Zusätzlich ist ein FPGA integriert. Dieser MPSoC zeigt seine Stärken im Bereich von hochleistungsstarken Anwendungen, für die ein FPGA notwendig ist. Da der Einsatz eines FPGAs keine Anforderung der Arbeit darstellt, ist das Board für den vorgesehenen Einsatz überdimensioniert.
3. Auf dem Variscite VAR-SOM-MX8X ist eine APU mit einem QuadCore Cortex-A35 1,2 GHz und eine RPU mit einem Cortex-M4F 264 MHz integriert. Über die integrierte APU wird ausreichend Leistung für rechenintensive Anwendungen bereitgestellt. Zusätzlich ist eine GPU vorhanden, die mittelfristig für grafikintensive Anwendungen eingesetzt werden kann. Über die integrierte RPU können gezielt Anwendungen ausgelagert werden, welche für Regelschleifen oder für Prozessdatenüberwachungen verwendet werden können.

- **Anforderung H.2: Bereitstellung von Mechanismen für Systemboot-, Ressourcen- und Speichermanagement**

Das Management umfasst alle notwendigen Funktionalitäten, die Herausforderungen von heterogenen MPSoC darstellen.

1. Der STM32MP153 bietet keine zentrale HW-Komponente für die notwendigen Funktionalitäten, was auf die geringe Anzahl der integrierten Prozessoren zurückgeführt werden kann.
2. Die Power Management Unit (PMU) übernimmt die Aufgabe des Systemboots und bietet Mechanismen für das Speicher-, Ressourcen- und Power-Management.
3. Die verbaute System Controller Unit (SCU) stellt die notwendigen Funktionalitäten zur Verfügung.

- **Anforderung H.3: Bereitstellung einer Verfügbarkeit von 15 Jahren**

In der Automobilindustrie und Luftfahrttechnik ist eine Verfügbarkeit von 15 Jahren vorgeschrieben.

1. Bei dem STM32MP153 wird eine Verfügbarkeit von 10 Jahren gewährleistet.
2. Xilinx bietet eine Verfügbarkeit von 15 Jahren.
3. Der VAR-SOM-MX8X MPSoC gewährleistet eine Verfügbarkeit von 15 Jahren.

- **Anforderung H.4: Bereitstellung von Sicherheitsmechanismen**

Zwar wird in dieser Arbeit der Schwerpunkt nicht auf Sicherheit gelegt. Allerdings soll für spätere Arbeiten oder Projekte die ausgewählte Plattform Mechanismen dafür bereitstellen.

1. Der Secure-Boot-Prozess wird ausschließlich für die Plattformen STM32MP153C und STMP32MP153F unterstützt.
2. Die Configuration and Security Unit (CSU) stellt Mechanismen für die Validierung von Images und Secure Boot bereit.
3. Der Security Controller (SECO) bietet Funktionalitäten für die erforderlichen Sicherheitsmechanismen an.

- **Anforderung A.1: Ausführliche Systemdokumentation**

Die vorhandene Dokumentation und bestehenden Betriebssysteme sollen die Einarbeitung in den eingesetzten MPSoC erleichtern.

1. ST stellt für die STM32 Plattform eine umfangreiche Dokumentation zur Verfügung. Die Betriebssysteme Linux und FreeRTOS werden unterstützt.
2. Xilinx stellt ein umfangreiches Dokumentationssystem bereit, das aus einer Vielzahl von Application Notes besteht, die Entwicklern den Einstieg in die Plattform erleichtern. Als Betriebssysteme werden Linux und FreeRTOS unterstützt.
3. Variscite stellt ein umfangreiches Dokumentationssystem zur Verfügung. Es werden Linux und FreeRTOS als Betriebssysteme unterstützt.

Entscheidung für die Auswahl des eingesetzten MPSoCs

Durch die nicht besonders starke APU und der fehlenden Mechanismen für die Anforderung H.2 wird der STM32MP153 in dieser Arbeit nicht eingesetzt. Der Zynq UltraScale+ MPSoC stellt alle notwendigen Funktionalitäten für die Erfüllung der Anforderungen bereit. Besonders der FPGA bietet für hochleistungsstarke Anwendungen umfangreiche Einsatzszenarien durch seine Flexibilität an. Genauso bietet der VAR-SOM-MX8X MPSoC alle notwendigen Funktionalitäten an. Da die Firma Mixed Mode bereits in anderen Projekten Erfahrung mit dem Zynq UltraScale+ gesammelt hat und Wissen auf einer neuen Plattform aufbauen möchte, fällt die Entscheidung auf den VAR-SOM-MX8X MPSoC von Variscite.

3.4 Auswahl des eingesetzten Software-Frameworks für LCM und IPC

Das eingesetzte Software-Framework soll einen möglichst standardisierten Ansatz (*Anforderung S.3*) gewährleisten, da eigene implementierte Lösungen wartungs- und zeitintensiv sind und nur mit hohem Aufwand auf andere Plattformen übertragen werden können. Deshalb wird eine selbstentwickelte Lösung für die IPC und das LCM ausgeschlossen. T. Adiono et. al. stellte

bereits fest, dass MPSoC-Architekturen sehr unterschiedlich sein können und dadurch maßgeschneiderte Lösungen nur ausschließlich für die eingesetzte Architektur verwendbar sind [40].

Für die Auswahl des eingesetzten Software-Frameworks wurden folgende bestehende Lösungen betrachtet:

- OpenCL, Cilk, openMP, Multicore Communications API (MCAPI)
- OpenAMP
- RPMsgLite

3.4.1 OpenCL, Cilk, openMP und MCAPI

In der Literatur sind die Frameworks OpenCL, Cilk, openMP und MCAPI zu finden, wenn nach einer Lösung für LCM und einer IPC gesucht wird. Allerdings setzen sie ihren Schwerpunkt auf die Parallelisierung von Aufgaben verteilt auf mehrere Prozessoren auf MPSoC, um Aufgaben performant und effizient abzuarbeiten. Dies wird auch als Massively Parallel Processing (MPP) bezeichnet [41–44].

Aus diesem Grund werden diese Frameworks nicht näher betrachtet.

3.4.2 OpenAMP

Das Open Asymmetric Multi Processing (OpenAMP) Framework ist durch die Multicore Association standardisiert und wird durch viele Firmen (Xilinx, Mentor Graphics) unterstützt. Es bietet Funktionalitäten für das LCM und für die IPC auf AMP-Systemen an und ist auf unterschiedlichen Betriebssystemen wie Linux, RTOS oder auf Bare-Metal verwendbar. Auf der Linux-Seite werden die bereits existierenden Kernel-Komponenten Remote Processor Framework (RemoteProc) und Remote Processor Messaging (RPMsg) eingesetzt, die seit Version 3.4.x im Mainline-Kernel enthalten sind. RPMsg verwendet VirtIO-Devices, die im späteren Verlauf der Arbeit näher betrachtet werden (Kap. 4.2). RemoteProc übernimmt die Aufgaben des LCM und RPMsg ist für die IPC zuständig. Auf RTOS und Bare-Metal besteht das Framework aus einer hardwareabhängigen Portierungsebene, welche die notwendigen Hardwareschnittstellen und Betriebssystemumgebungen bereitstellt (Abb. 3.1) [46–48].

Der OpenAMP Lösungsansatz bietet eine optimale Lösung, da auf bereits existierenden Mainline-Kernel-Komponenten aufgebaut werden kann, die auf vielen bestehenden Linux-Plattformen enthalten sind. Dadurch kann dieser Ansatz auf viele unterschiedliche Plattformen übertragen werden. Allerdings bringt dieser Ansatz einen hohen Implementierungsaufwand mit sich. Die bisherigen Recherchen zeigten, dass für den eingesetzten MPSoC i.MX8X keine Portierung für die remoteProc-Komponente für das Linux auf dem A35 und keine Portierungsebene für das FreeRTOS auf dem M4 vorhanden ist. Für den RPMsg-Teil existiert auf der M4-Seite bereits eine Lösung von NXP, die als RPMsglite bezeichnet wird. Auf der Linux-Seite steht eine Implementierung für RPMsg bereit. Wegen der fehlenden Portierung der remoteproc-Komponente und Portierungsebene wird der Implementierungsaufwand dieser Lösung als hoch eingestuft.

Neben dem OpenAMP Open-Source-Lösungsansatz bietet die Firma Mentor Graphics die kommerzielle Lösung Mentor Embedded Multicore Framework (MEMF) für OpenAMP an. Da allerdings ein Open-Source-Ansatz verfolgt wird (*Anforderung S.7*), wird das Framework von Mentor Graphics nicht in Betracht gezogen.

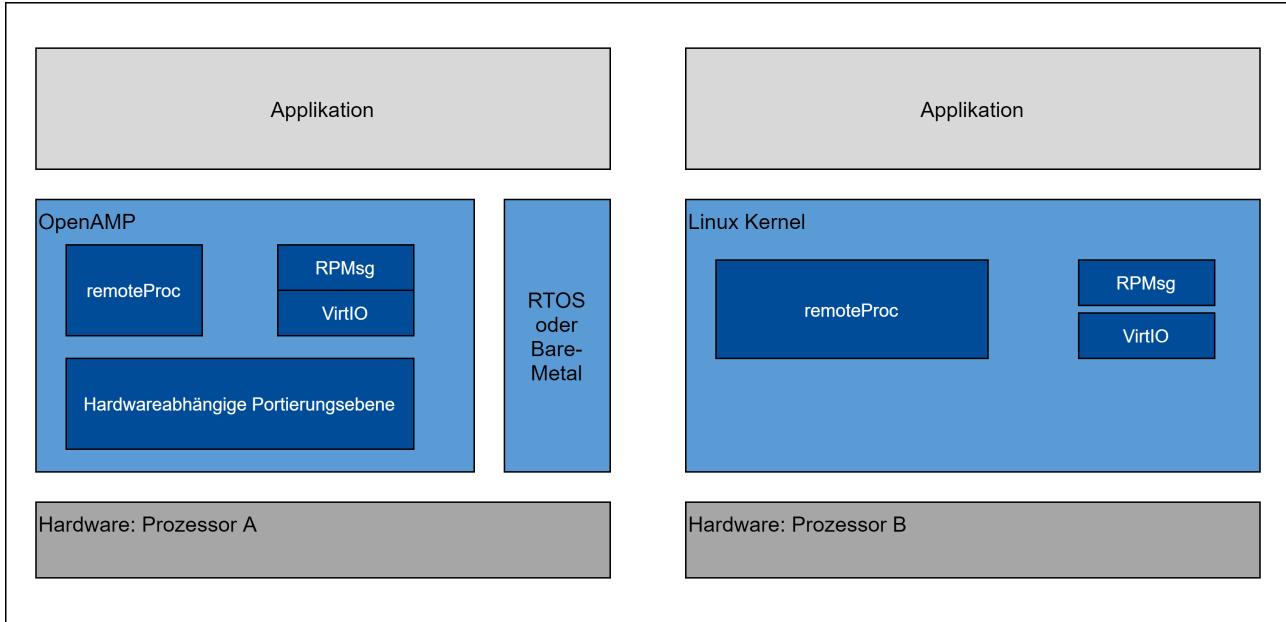


Abbildung 3.1: Darstellung des OpenAMP-Frameworks auf einem Linux, RTOS und Bare-Metal. Die illustrierten Kästchen sind von oben nach unten in die Layer Applikation (hellgrau), Middleware und Betriebssysteme (blau) und Hardware (dunkelgrau) unterteilt. In der linken Hälfte ist der hierarchische Aufbau von OpenAMP auf einem RTOS/Bare-Metal illustriert. Er besteht aus den plattformunabhängigen Komponenten RemoteProc und RPMsg und der hardwareabhängigen Portierungsebene. In der rechten Hälfte ist der Aufbau für Linux abgebildet. Er besteht auch aus den gleichen Komponenten RemoteProc und RPMsg (umgezeichnet nach [45]).

3.4.3 RPMsgLite

Der von NXP entwickelte RPMsgLite-Ansatz setzt seinen Schwerpunkt auf die IPC und setzt dabei wie OpenAMP RPMsg ein. Das LCM wird mit diesem Ansatz nicht unterstützt. Die Besonderheit ist, dass die umfangreiche API von RPMsg vereinfacht wurde, um den benötigten Speicherplatz und die Komplexität zu reduzieren. So setzt RPMsglite seinen Schwerpunkt vor allem auf Prozessoren, die wenig Speicher integriert haben, wie den Cortex M0+. Zusätzlich hat der Anwender die Möglichkeit, RPMsgLite als statische oder dynamische Bibliothek einzusetzen und einen Zero-Copy Mechanismus für das Kopieren der Daten zwischen Applikation und RPMsg-Device zu verwenden. Er minimiert die auftretenden Latenzen für das Kopieren der Nachrichten. Der notwendige Flash-Speicher mit RPMsglite liegt beim Einsatz der statischen Bibliothek bei 2926 Bytes, der notwendige RAM-Speicher bei 352 Bytes. Im Vergleich benötigt OpenAMP 5547 Bytes im Flash und 456 Bytes im RAM, plus zusätzlicher Allokation im dynamischen Speicher [49].

Dieser Ansatz hat den Vorteil, dass er bereits auf dem eingesetzten MPSoC i.MX8X einsatzbereit ist und keinen erhöhten Implementierungsaufwand mit sich bringt. NXP bietet auf Linux bereits die notwendigen Komponenten an, um über RPMsg mit dem M4, der RPMsglite einsetzt, zu kommunizieren. Auf M4-Seite sind ebenso Demo-Applikationen zu finden, die den RPMsglite-Ansatz beinhalten. Der Nachteil dieser Lösung ist die bereits oben erwähnte fehlende Komponente für das LCM.

Die Entscheidung ob OpenAMP, RPMsglite oder eine Kombination aus beiden Lösungsansätzen

3.4. AUSWAHL DES EINGESETZTEN SOFTWARE-FRAMEWORKS FÜR LCM UND IPC

eingesetzt wird, kann mit dem bisherigen erarbeiteten Wissensstand noch nicht endgültig getroffen werden, da noch nicht festgestellt wurde, ob OpenAMP mit RPMsgLite kompatibel ist oder mittelfristig OpenAMP umfangreichere Funktionalitäten bereistellt. Dafür muss der OpenAMP-Ansatz im Detail analysiert werden, was im nächsten Kapitel erfolgt.

Kapitel 4

Analyse von OpenAMP

Dieses Kapitel beschäftigt sich mit der Analyse des OpenAMP-Ansatzes und schließt mit der Entscheidung ab, ob RPMsgLite oder OpenAMP eingesetzt wird.

4.1 Überblick

Das OpenAMP Framework stellt folgende Funktionalitäten zur Verfügung [50]:

- Lifecycle-Operationen für das Starten und Stoppen von Coprozessoren.
- IPC für den Austausch von Nachrichten.
- Proxy-Infrastruktur für den entfernten Zugriff auf Service wie dem Dateisystem. Darüber können Datenbanken, Log- oder Konfigurationsdateien dem Coprozessor zur Verfügung gestellt werden.

Das OpenAMP Framework besteht aus folgenden Komponenten [51]:

- RemoteProc übernimmt das LCM und startet und stoppt den Coprozessor mit unterschiedlichen Firmwares.
- RPMsg ist für die nachrichtenbasierende Kommunikation zwischen mehreren Prozessoren auf einem AMP-System verantwortlich.
- VirtIO stellt die Sicherungsschicht dar, die von RPMsg verwendet wird. Sie implementiert den Austausch der Daten über Shared-Memory und stellt Mechanismen für die Benachrichtigung von neu eingetroffenen Nachrichten bereit.
- Die Hardwareabstraktions-Bibliothek Libmetal abstrahiert die eingesetzte Plattform und repräsentiert die hardwareabhängige Portierungsebene (Abb. 3.1).

Folgende Ansätze bietet OpenAMP an (Abb. 4.1):

1. Auf dem Master-Prozessor kommuniziert eine Linux Userspace-Anwendung mit dem Coprozessor, auf dem OpenAMP auf einem RTOS oder Bare-Metal System läuft. Die Linux-anwendung kommuniziert mit den im Kernelspace laufenden Komponenten RemoteProc und RPMsg.
2. Auf dem Master-Prozessor läuft OpenAMP auf einem RTOS oder Bare-Metal System und kommuniziert mit der auf einem Linux laufenden Userspace-Anwendung.

4.1. ÜBERBLICK

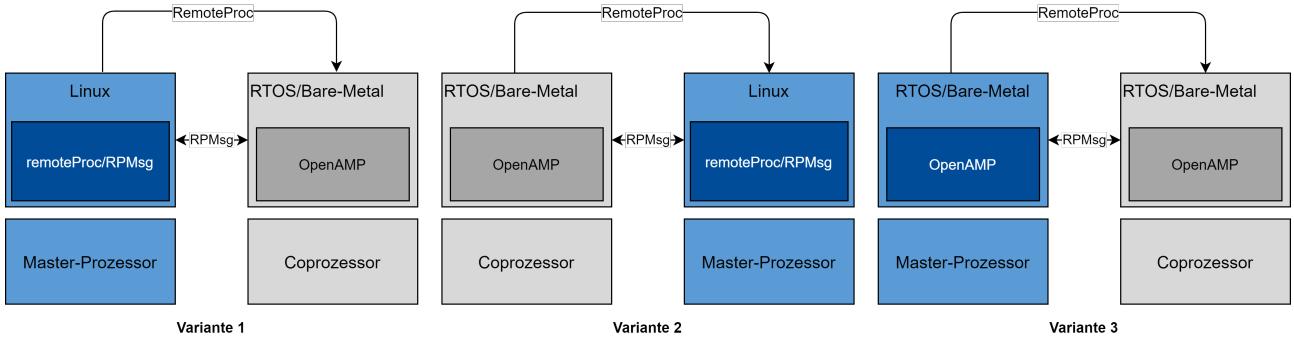


Abbildung 4.1: Darstellung der Varianten von OpenAMP, die in drei mögliche Ansätze unterteilt sind (umgezeichnet nach [51]). Details siehe Text.

3. Auf dem Master-Prozessor und Coprozessor laufen ein RTOS oder Bare-Metal System und sie kommunizieren über das OpenAMP Framework miteinander.

Es ist ersichtlich, dass das OpenAMP Framework auf einem Linux nicht verwendet wird, sondern auf die im Kernel vorhandenen Komponenten zurückgegriffen wird (Abb. 4.1). Auf einem RTOS oder Bare-Metal System wird genau hierfür das OpenAMP Framework eingesetzt, welches die Komponenten RemoteProc und RPMsg für diese Systeme bereitstellt. Der Unterschied zwischen den drei aufgezeigten Ansätzen ist die Festlegung der Rolle des Master-Prozessors.

Seit dem Release v.2016.10 wird zusätzlich die Möglichkeit angeboten, OpenAMP im Linux Userspace zu betreiben. Dies schafft die Möglichkeit zwischen Linux-Prozessen über OpenAMP zu kommunizieren. Da diese Arbeit ihren Schwerpunkt auf AMP-Systeme mit zwei unterschiedlichen Prozessoren setzt, wird diese Möglichkeit nicht näher betrachtet [52].

Das Linux wird bei der Umsetzung des MMAMP-Systems aufgrund der Anforderungen S.4 und S.5 die Rolle des Master-Prozessors übernehmen und Aufgaben an den Cortex-M4 auf dem i.MX8X auslagern.

Vereinfachter Ablauf zwischen Master-Prozessor und Coprozessor über OpenAMP

Folgender Ablauf stellt die Interaktion zwischen dem Master-Prozessor und Coprozessor sowie den Komponenten RemoteProc, RPMsg und OpenAMP dar [53]:

1. Der Master-Prozessor liest die Daten- und Codesektionen aus der Ressourcentabelle, die Teil des Firmware-Images sind, aus und lädt sie in die gewünschte Speicheradresse.
2. Der Master-Prozessor startet den Coprozessor, der mit der Ausführung der geladenen Firmware beginnt. Der Coprozessor initialisiert die notwendigen Ressourcen für die Kommunikation über RPMsg und benachrichtigt den Master-Prozessor über die Fertigstellung.
3. Eine IPC zwischen Master-Prozessor und Coprozessor wird über RPMsg aufgebaut.
4. Sobald der Coprozessor abgeschalten werden soll, benachrichtigt ihn der Master-Prozessor. Der Coprozessor baut darauf hin die angelegten Ressourcen ab.
5. Der Master-Prozessor baut seine angelegten Ressourcen ab.
6. Der Master-Prozessor stoppt den Coprozessor und kann im Anschluss den Coprozessor erneut mit der gleichen oder anderen Firmware starten.

4.2 Aufbau von RPMsg

Über das RPMsg-Protokoll läuft die Kommunikation zwischen Prozessoren auf heterogenen MPSoC über geteilten Speicher ab. Unterteilt wird RPMsg in drei ISO/OSI-Schichten: Transport-, Sicherungs-, und Bitübertragungsschicht (Abb. 4.2).

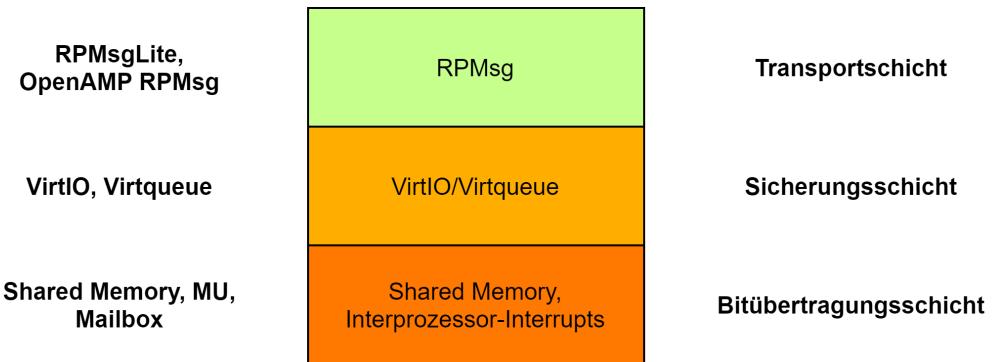


Abbildung 4.2: RPMsg-Aufbau: Unterteilt in drei Schichten gemäß OSI-Modell (umgezeichnet nach [54]). Details siehe Text.

In den folgenden Abschnitten werden die einzelnen Schichten erklärt, welche für das Verständnis der nachfolgenden Kapitel notwendig sind.

Bitübertragungsschicht - Shared-Memory

Der Datenaustausch erfolgt über Shared-Memory, worauf beide Prozessoren zugreifen können, sowie zwei IPIs. Jeder Prozessor benachrichtigt mit einem IPI den anderen Prozessor über den Empfang neuer Daten. Auf Hardwareebene wird die IPC mit den eingesetzten IPIs über eine MU umgesetzt.

Sicherungsschicht - VirtIO

VirtIO repräsentiert die Sicherungsschicht, die aus folgenden Teilkomponenten aufgebaut ist:

- VirtIO: Bereitstellung einer Linux API-Implementierung für die Verwendung der Virtqueue und der Vrings.
- Virtqueue: Ist für die Benachrichtigung von neu bereitstehenden Buffern im Vring über IPIs verantwortlich.
- Vring: Bereitstellung des Buffer-Managements, das aus Ring-Buffern im Shared-Memory besteht.

Jedes VirtIO-Device ist aus zwei unidirektionalen Virtqueues aufgebaut: eine Tx-Virtqueue zum Versenden von Daten vom Master-Prozessor zum Coprozessor und eine Rx-Virtqueue zum Versenden von Daten vom Coprozessor zum Master-Prozessor. Eine Virtqueue enthält einen Vring, der aus einem Bufferdeskriptor, den verfügbaren Ringbuffer und dem verbrauchten Ringbuffer aufgebaut ist. Alle drei sind im Shared-Memory abgelegt, auf den beide Prozessoren Zugriff haben. Durch die Trennung durch von zwei unidirektionale Virtqueues entfällt der Synchronisationsaufwand zwischen Prozessoren. Dieser Mechanismus wird als Single-Writer Single-Reader Circular Buffer bezeichnet. Die genauen Implementierungsdetails werden in dieser Arbeit nicht

behandelt und sind [55, 56] zu entnehmen.

Transportschicht - RPMsg

Die RPMsg-Implementierung erfolgt mit OpenAMP oder RPMsgLite. Dadurch ist es möglich, auf Linux-Seite die bestehenden Kernel-Komponenten RemoteProc und RPMsg einzusetzen und auf einer RPU entweder die Kommunikation über OpenAMP oder RPMsgLite zu gewährleisten.

Eine RPMsg-Nachricht enthält einen im Shared-Memory abgelegten Buffer. Die Adresse, die auf diesen Buffer zeigt, ist im Bufferdeskriptor enthalten. Für die Adressierung des Empfängers und Senders sind jeweils 4 Bytes vorgesehen.

Jeder Prozessor wird auf der RPMsg-Schicht durch ein RPMsg-Device repräsentiert, das einen Kommunikationskanal zwischen Master-Prozessor und Coprozessor darstellt. Deshalb werden RPMsg-Devices auch als **Kanäle** bezeichnet und als Strings repräsentiert.

Darüber angeordnet befinden sich die RPMsg-**Endpunkte**, die es Benutzern erlauben, mehrere Rx-Callbacks mit dem selben Kanal zu verbinden. Jeder Endpunkt besteht aus einer Empfängeradresse und der dazugehörigen Rx-Callback Funktion. Sobald eine Applikation einen Endpunkt mit seiner angegebenen Adresse erstellt, werden alle eingehende Nachrichten, welche als Zieladresse die Empfängeradresse des registrierten Endpunkts angegeben haben, an die registrierte Rx-Callback-Funktion weitergeleitet. Jeder Kanal hat einen Standardendpunkt, der bei der Kanalerstellung erzeugt wird. Für die Benachrichtigung der durchgeföhrten Kanalerstellung muss eine Nameservice-Announcement-Nachricht an den Master-Prozessor gesendet werden. Hierfür wird der Endpunkt *RPMsg_NS_ADDR* mit der Adresse 53 eingesetzt, der auch für den Kanalabbau verwendet wird.

4.3 Aufbau von RemoteProc

Die Komponente RemoteProc übernimmt die Aufgabe des LCM. Hierbei ist sie für das Auslesen der Firmware und das darauffolgende Laden verantwortlich. RemoteProc unterstützt das Format Executable and Linkable Format (ELF), um eine Firmware zu parsen. Die ELF enthält die Informationen über die notwendigen Sektionen für das Anlegen der Interrupts sowie der Text- und Datensegmente. Zusätzlich erwartet RemoteProc eine Ressourcentabelle innerhalb der ELF, welche für die Beschreibung der notwendigen VirtIO-Devices und Vrings verantwortlich ist. Nachdem RemoteProc die Informationen der Ressourcentabelle erfolgreich ausgelesen und die dazu gehörigen VirtIO-Devices erstellt hat, werden die vorhandenen Sektionen in die angegebene Stelle im Adressraum geladen. Anschließend wird der Coprozessor mit Takt versorgt und der Reset ausgelöst (Abb. 4.3). Der Master-Prozessor wartet daraufhin auf eine Nachricht des Coprozessors über die IPC und baut die Verbindung über RPMsg auf.

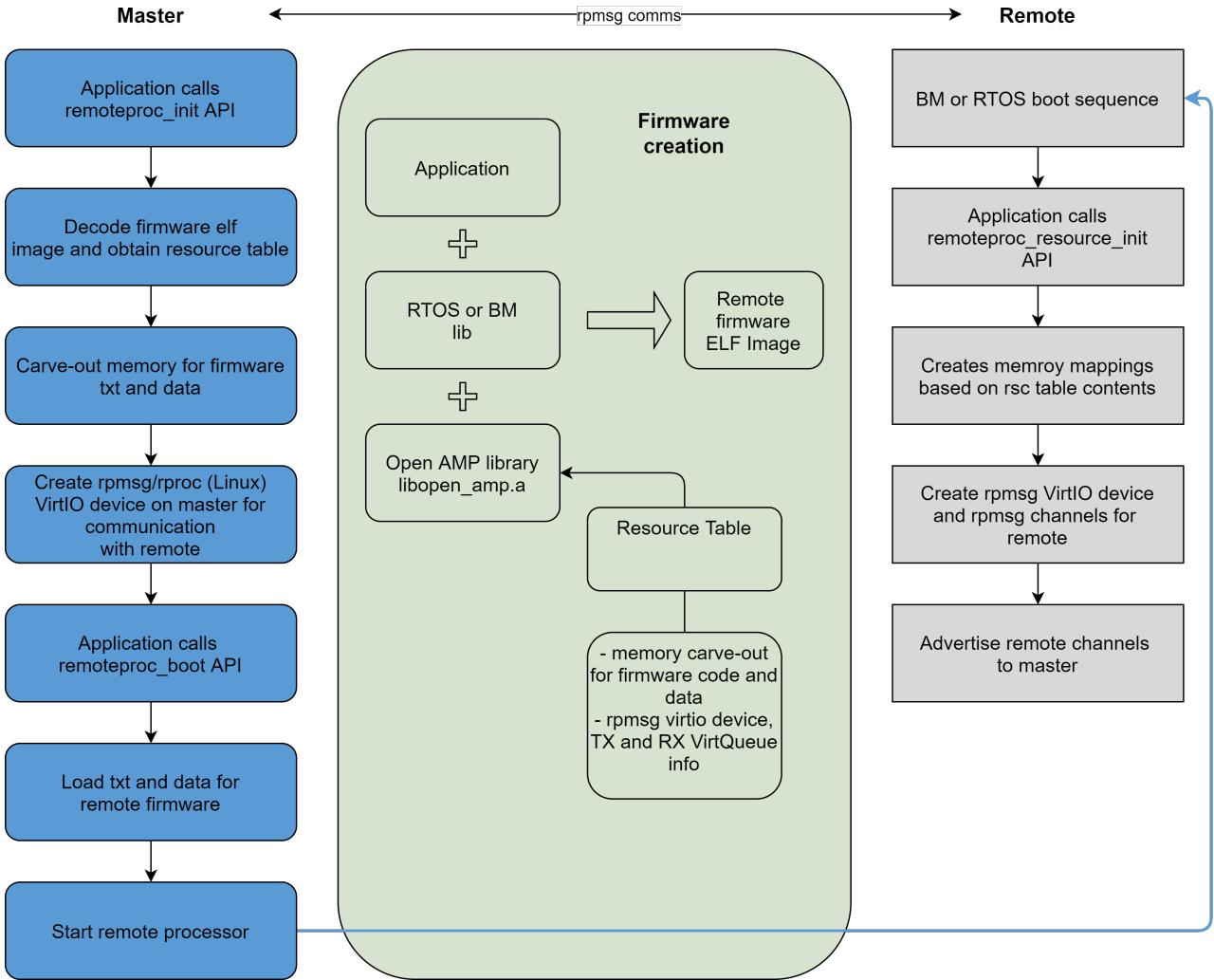


Abbildung 4.3: LCM-Ablauf durch die Komponente RemoteProc. Auf der linken Seite sind die Schritte zum Parsen und Laden der Firmware und der anschließende Start des Coprozessors beschrieben (blau). In der Mitte ist der Aufbau der Firmware illustriert (grün). Sie besteht aus der Applikation, einer optionalen Middleware wie einem RTOS und der dazu gelinkten OpenAMP-Bibliothek. Zusätzlich muss noch eine Ressourcentabelle enthalten sein. Auf der rechten Seite wird die Bootreihenfolge vom Coprozessor beschrieben (grau). Der blaue Pfeil stellt den Start des Coprozessors über dem Master-Prozessor dar. Über den bidirektionalen schwarzen Pfeil wird die IPC zwischen den beiden Prozessoren dargestellt (nachgezeichnet nach [53]). Details siehe Text.

Nachdem das Reset-Signal des Coprozessors ausgelöst wurde, startet er und liest die bereitgestellten Informationen aus der Ressourcentabelle aus. Hierbei werden die notwendigen Speicherbereiche angelegt und die Kommunikation über RPMSg initialisiert. Danach meldet er seinen Kanal beim Master an, der den Kanal bestätigt und die Kommunikation eröffnet (Abb. 4.3).

4.4 Hardwareabstraktions-Bibliothek

Die Hardwareabstraktions-Bibliothek Libmetal stellt eine API zum Zugriff auf Geräte, Interrupts und Speicher zur Verfügung. Sie kann auf unterschiedlichen Systemumgebungen wie Linux, FreeRTOS oder Bare-Metal zum Einsatz kommen. Bei der Portierung auf eine neue Platt-

form muss der hardwareabhängige Teil in die Libmetal integriert werden. Dieser Teil stellt die hardwareabhängige Portierungsebene dar (Abb. 3.1).

4.5 Entscheidung RPMsgLite oder OpenAMP

Durch die Analyse von OpenAMP wurde festgestellt, dass die Transportschicht entweder durch RPMsgLite oder durch RPMsg umgesetzt werden kann und beide zueinander kompatibel sind. Dies hat zur Folge, dass auf dem Coprozessor Cortex-M4 auf dem i.MX8X RPMsgLite und auf dem Master die bereits existierende Komponente RPMsg zum Einsatz kommen könnte. Der Vorteil dieser Lösung ist der nicht erhöhte Implementierungsaufwand (Kap. 3.4.3). Der Nachteil ist, dass über RPMsgLite keine Implementierung für das Parsen der Ressourcentabelle und dadurch keine Möglichkeit zur Bereitstellung von einer Proxy Infrastruktur angeboten wird. Der Schwerpunkt dieser Arbeit liegt zwar nicht in der Anbindung einer Proxy-Infrastruktur, trotzdem bietet OpenAMP mittelfristig dadurch mehr Funktionalitäten. Deshalb wird der Lösungsansatz über OpenAMP weiterverfolgt und die offene Entscheidung aus Kapitel 3.4.3 ist getroffen. Allerdings wird, bevor die Portierung von OpenAMP auf den M4 umgesetzt wird, eine RPMsgLite Demo implementiert, um das Verhalten zu prüfen und einen besseren Einblick in den RPMsg-Ablauf zu erhalten.

Kapitel 5

Multiprozessor-Architektur des i.MX8X

Der auf dem Variscite VAR-SOM-MX8X integrierten MPSoC i.MX8X stellt durch seine vielen Subsysteme eine hohe Komplexität für unterschiedliche Anwendungsfälle dar. Deshalb werden in diesem Kapitel die wichtigsten Komponenten näher betrachtet. Weitere Schwerpunkte liegen auf dem Systemboot-, Speicher- und Ressourcenmanagement. Dies ist erforderlich, um ein Verständnis für die Möglichkeiten und Einschränkungen dieser Architektur zu erlangen, um daraus ein Konzept für das MMAMP-System zu erstellen.

Eine Übersicht zur gesamten Hardwareplattform ist in Abbildung 5.1 dargestellt.

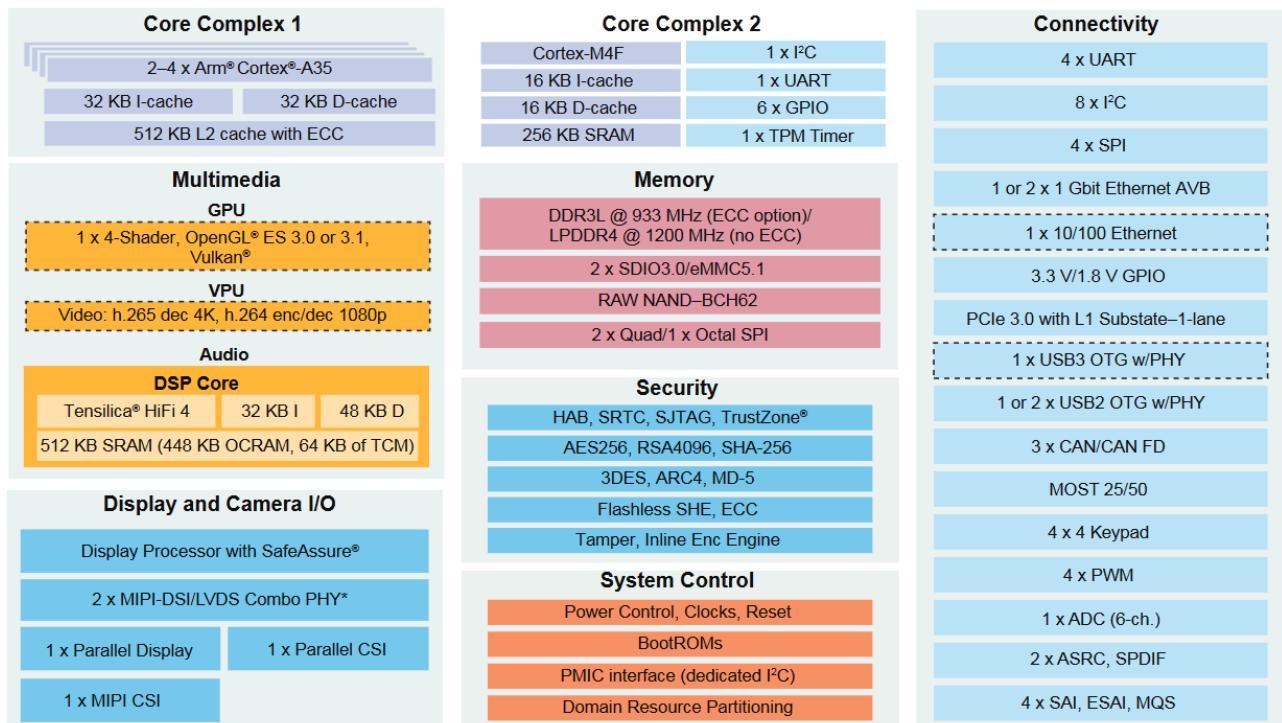


Abbildung 5.1: i.MX8X: Systemüberblick. Eine detaillierte Beschreibung aller vorhandenen Komponenten erfolgt in dieser Arbeit nicht und ist [57] zu entnehmen (übernommen aus [58]).

5.1 Subsysteme

Der MPSoC i.MX8X integriert verschiedene Subsysteme, die miteinander über den AMBA Bus kommunizieren. So ist jeweils ein Quadcore ARM Cortex-A35 für leistungsfähige Applikationen und ein Singlecore ARM Cortex-M4 für echtzeitkritische Applikationen integriert. Für grafische Anwendungen stehen die integrierten Prozessoren GPU oder VPU bereit. Da sich diese Arbeit mit dem AMP zwischen den Prozessoren A35 und M4 beschäftigt, werden die Subsysteme für die grafischen Applikationen in dieser Arbeit nicht näher betrachtet. Das Speichersystem setzt sich aus einem 256 KB On-Chip OCRAM und einem 4 GB Off-Chip LPDDR4 zusammen. Zusätzlich wird ein Speicherbereich mit bis zu 64 GB durch ein eMMC Interface oder durch eine MicroSD-Karte zur Verfügung gestellt. Für die externe Kommunikation stehen Schnittstellen unter anderem für JTAG, USB, GPIO, Bluetooth und CAN bereit (Abb. 5.1). Die SCU übernimmt unter anderem die Verwaltung und den Zugriff auf Peripheriegeräte und besteht aus dem Singlecore Cortex-M4 [57, S. 9-16].

5.1.1 Application Processor Unit

Die APU ist aufgebaut aus einem Quadcore ARM Cortex-A35, von dem jeder Core einen privaten 32 KB L1 Cache besitzt. Alle Cores teilen sich den 512 KB L2 Cache, der einen Fehlerkorrektur zur Verfügung stellt. Über eine MMU wird der virtuelle Adressraum verwaltet, um auf den Off-Chip-Speicher zuzugreifen. Die APU implementiert die 64-Bitarchitektur des ARMv8-A, ist aber dennoch rückwärtskompatibel zur 32-Bitarchitektur des ARMv7-A.

5.1.2 Realtime Processor Unit

Die RPU besteht aus dem Singlecore ARM Cortex-M4, der Zugriff auf seinen privaten On-Chip 256 KB Tightly-Coupled Memory (TCM) besitzt, wodurch schnelle Zugriffszeiten ermöglicht werden. Zusätzlich steht der Off-Chip-Speicher zur Verfügung, um einen Shared-Memory für die IPC bereizustellen. Die RPU implementiert die Architektur ARMv7-M.

5.2 System Controller Unit

Die SCU besteht aus einem ARM Cortex-M4 und ARM Cortex-M0+ für den integrierten SECO. Sie führt eine zusätzliche Abstraktionsschicht für den Zugriff auf die darunterliegende Hardware auf der Plattform i.MX8X ein. Ausschließlich die SCU hat direkten Zugriff auf alle Subkomponenten der gesamten Plattform. Wenn Subsysteme wie die APU oder RPU direkt auf ein Subsystem wie dem DDR-Bereich ohne Berechtigungen zugreifen, wird der Zugriff verweigert und eine Fehlermeldung erzeugt. Die SCU ist für folgende Aufgabenbereiche verantwortlich [59]:

- Systeminitialisierung und Bootprozess
- System Controller Kommunikation
- Power Management Service
- Resource Management Service
- Pad Configuration
- Timer, Interrupts, Security

In den folgenden Kapiteln werden die Begriffe Ressource und Partition erläutert und der Power- und Resource-Management Service näher beschrieben, da sie eine wichtige Voraussetzung für das AMP zwischen der APU und RPU sind. Durch die zur Verfügung gestellte System Controller Firmware API (SCFW-API) erfolgt für die APU und RPU der Zugriff auf die aufgezählten Services. Die restlichen Aufgabenbereiche der SCU sind aus dem System Controller Firmware Porting Guide zu entnehmen [59].

5.2.1 Resource Management Service

Der Resource Management Service (RMS) ist für die Verwaltung von Ressourcen über Partitionen verantwortlich [60].

Ressource

Wie in Kapitel 2.6.2 angesprochen wurde, muss der Lese- und Schreibzugriff auf Speicher- und Peripherieeinheiten geordnet ablaufen, da sonst ein undefiniertes Verhalten erzeugt werden kann. Hierfür wird das Prinzip der Ressource eingeführt. Eine Ressource ist ein Intellectual Property (IP)-Funktionsblock, der wiederum ein Speicher oder eine Peripherieeinheit sein kann. So kann eine Ressource ein GPIO, eine MU oder ein Speicherbereich sein. Jede Ressource hat genau einen Eigentümer, der durch eine Partition dargestellt wird. Die Partition kann den Zugriff auf Ressourcen an andere Partitionen weitergeben oder den Eigentümer der Ressource an eine andere Partition übergeben. Beim Systemstart sind alle Ressourcen der dauerhaft vorhandenen Partition der SCU zugeordnet.

Partition

Im Kontext der i.MX8X-Plattform wird unter Partition nicht die Einteilung einer Festplatte in verschiedene Bereiche definiert, sondern die Zuordnung und Aufteilung von Ressourcen zu einzelnen Partitionen. Partitionen können zur Bootzeit und zur Laufzeit erstellt oder auch gelöscht werden. Jede Partition besitzt bei der Erzeugung eine Parent-Partition, die in der Lage ist, die Child-Partition zu booten, neu zu booten oder auszuschalten. Durch dieses Prinzip werden wichtige Funktionalitäten für das LCM bereitgestellt.

Zur Systemlaufzeit kann der Zugriff auf Ressourcen verändert werden. So ist es möglich, dass mehrere Partitionen auf dieselbe Ressource Zugriff haben, um beispielsweise einen Shared-Memory Bereich zu definieren. Auch die Weitergabe zur Laufzeit einer Ressource an eine andere Partition ist möglich.

Ressourcenzugriffsschutz

Die Steuerung des Ressourcenzugriffsschutzes übernimmt die Softwarekomponente Extended Resource Domain Controller (xRDC), die innerhalb der System Controller Firmware (SCFW) läuft. Sie ist für die tatsächliche Verwaltung der Zugänge, des Speicherzugriffsschutzes und die Isolierung von Peripherieeinheiten verantwortlich.

Fazit

Der RMS ermöglicht eine sichere Umsetzung von mehreren unabhängigen Systemen auf den MPSoC. Hierbei kann in der Praxis eine echtzeitkritische Anwendung isoliert auf einer separaten Partition mit zugewiesenen Ressourcen auf der RPU ablaufen, während auf der APU performance Anwendungen laufen, die keinen Zugriff auf die Ressourcen der RPU haben. Außerdem werden durch die beschriebenen Möglichkeiten gewährleistet, dass die einzelnen Subsysteme nicht auf Ressourcen zugreifen, die ihnen nicht zugeteilt sind. Wichtig ist hierbei, dass der Anwender die Ressourcen gezielt den gewünschten Partitionen zuweist und sicherstellt, dass keine zwei Partitionen auf die gleichen Ressourcen zugreifen. Wenn dies nicht verhindert wird, entsteht undefiniertes Verhalten auf den Systemen und stellt die Systemstabilität und Sicherheit in Frage.

5.2.2 Power Management Service

Der Power Management Service (PMS) ist für die Energieverwaltung, also unter anderem für die Strom-, Takt- und Resetsteuerung verantwortlich. Die Stromsteuerung verwaltet den Leistungszustand der Subsysteme auf dem MPSoC. Die Taktsteuerung verwaltet die Takte der einzelnen Subsysteme. Dazu gehören Taktquellen wie Oszillatoren und Phasenregelschleifen (PLL) sowie Taktteiler, Muxes und Gates. So kann über die SCFW-API der notwendige Takt den einzelnen Ressourcen zugeteilt werden.

5.2.3 System Controller Firmware API

Auf der SCU wird die SCFW ausgeführt, welche die bereits erwähnten Aufgaben ausführt. Die SCFW stellt eine SCFW-API zur Verfügung, über die andere Subsysteme wie die APU oder RPU mit der SCU kommunizieren können. Über die SCFW-API haben die Subsysteme Zugriff auf die oben genannten Bereiche. Die API verwendet Remote Procedure Call (RPC) Aufrufe, um eine IPC zwischen den Prozessoren herzustellen, die auf Hardwareebene durch MUs auf beiden Seiten umgesetzt wird [59].

5.2.4 Messaging Unit

Die integrierten MU ermöglichen eine IPC zwischen zwei Prozessoren auf dem MPSoC. Sie sind für die Steuerung des Datenaustausches verantwortlich. Jede Partition muss mindestens auf eine MU zugreifen können, um mit der SCU zu kommunizieren und darüber den Zugriff auf andere Ressourcen auf dem MPSoC anzufragen.

5.3 Systembootprozess

Der Systembootprozess läuft über die Subsysteme SCU und SECO ab, die ein Bootimage fordern, das aus zwei Containern besteht. Der erste Container enthält das SECO Firmware-Image, das von NXP bereitgestellt und signiert ist. Der zweite Container enthält das SCFW-Image, ein Cortex-M4 Firmware-Image für die RPU und ein Cortex-A35 Image für die APU. Für einen Secureboot müssen die Images im zweiten Container verschlüsselt sein, die während der Startzeit von der SECO Firmware authentifiziert werden. Der SECO ist für die Authentifizierung der Images verantwortlich und autorisiert die Ausführung der Images. Da in dieser Arbeit der Schwerpunkt nicht auf dem Ausführen des Secure-Boots liegt, werden unverschlüsselte Images im Bootimage abgelegt.

5.3. SYSTEMBOOTPROZESS

Der Systembootprozess unterteilt sich in folgende Schritte, wobei einige Details zur Vereinfachung weggelassen werden, die in [57] nachgelesen werden können (Abb. 5.2):

1. Beim Reset startet die SCU basierend auf der Firmware, die im On-Chip SCU ROM abgelegt ist. Gleichzeitig startet der SECO und liest seine Firmware aus dem On-Chip SECO ROM aus. Beide Subsysteme führen die Hardwareinitialisierung aus.
2. Die SCU ROM Firmware liest den Boot-Mode über die Bootpins aus.
3. Die SCU ROM Firmware lädt den ersten Container vom Bootmedium, der immer eine SECO FW enthalten muss, die durch den NXP Schlüssel signiert wurde. Die SECO FW wird von der SCU in den SECO TCM geladen.
4. Die SCU ROM Firmware lädt den zweiten Container vom Bootmedium und kopiert die vorhandenen Images an die im Boot-Image angegebene Speicheradresse.
5. Die SCU ROM Firmware startet die SCFW.

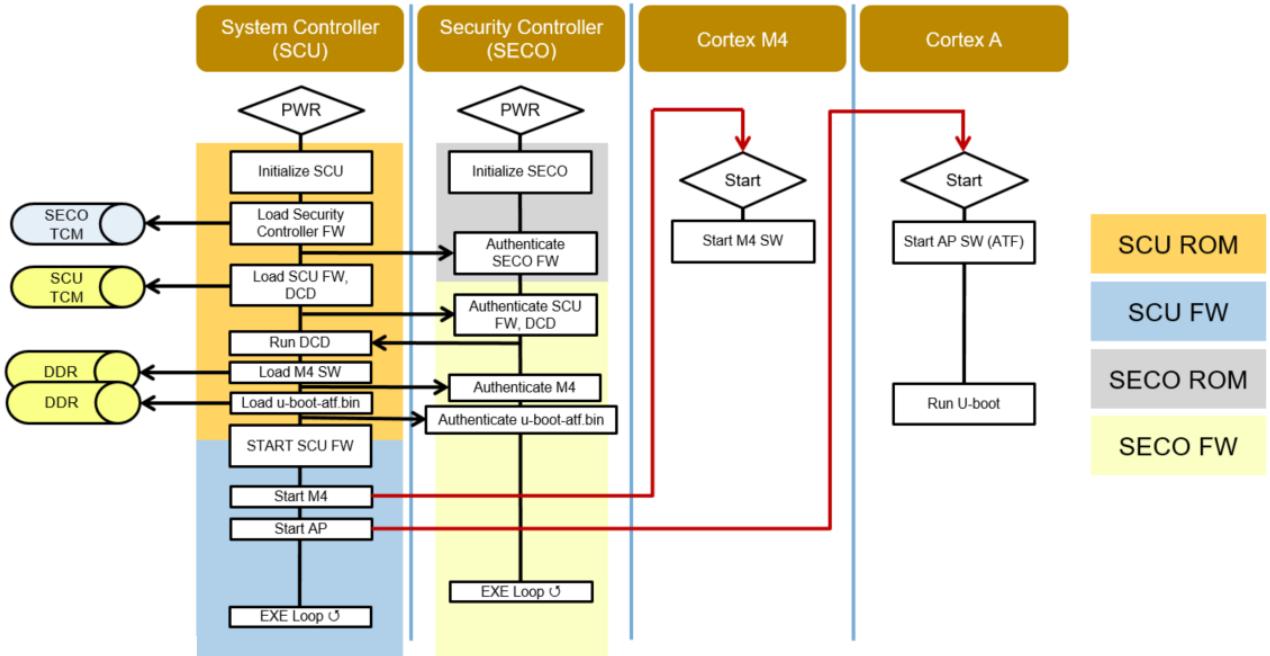


Abbildung 5.2: i.MX8X: System-Boot (übernommen aus [61]). Details siehe Text.

Die SCFW ist für die Erstellung und Zuweisung der Partitionen an sich selbst, an den SECO, Cortex-M4 und Cortex-A35 verantwortlich (Abb. 5.3). Die Erstellung der SCU und SECO Partition erfolgt immer. Abhängig von der Erstellung des Boot-Images werden Partitionen für den Cortex-M4 und für die ATF Partition angelegt. Die ATF, die auch als Boot-Partition bezeichnet wird, erstellt im Anschluss die Partition für das Linux-System. Innerhalb der ATF Partition wird der Second Program Loader (SPL) gestartet, der anschließend die ATF Firmware startet. Die ATF startet wiederum die U-Boot, die innerhalb der Linux-Partition abläuft. Durch den Einsatz der ATF wird eine hohe Systemsicherheit erreicht und zwischen der normalen und sicheren Welt unterschieden (Kap. 2.6.6). In [62] können weitere Details über den Bootprozess mithilfe von ATF entnommen werden.

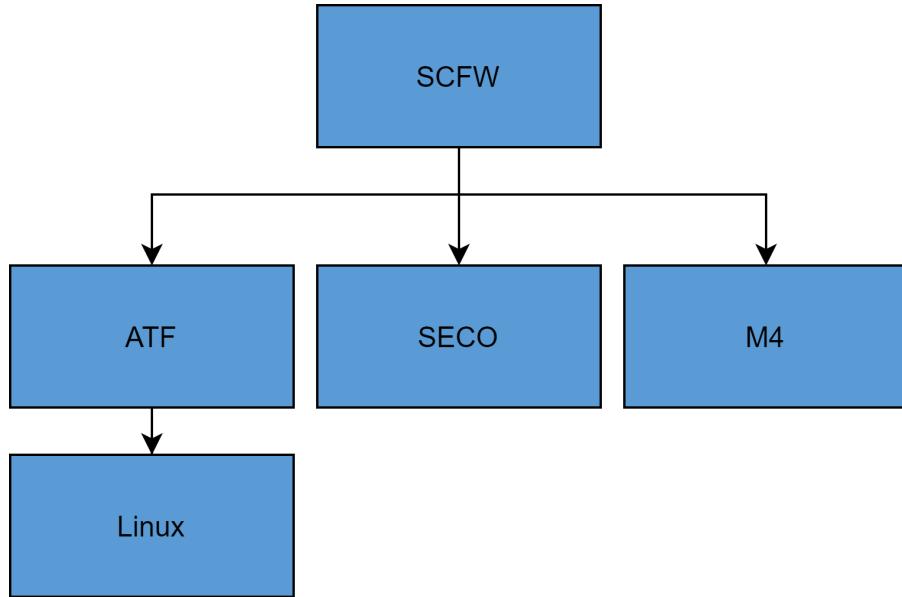


Abbildung 5.3: i.MX8X: Standard-Partitionskonfiguration (Umgezeichnet nach [59]). Details siehe Text.

Durch die flexible Erstellung des Boot-Images und die daraus resultierenden Partitionen entstehen drei mögliche Systemboot-Szenarien [59]:

- 1. Linux startet unabhängig vom M4:** Dieses Szenario ist ein bekannter Anwendungsfall in der Automobilindustrie. Die SCFW erstellt eine Partition für den M4 und für das Linux. Anschließend wird zuerst die M4-Partition, danach erst die Linux-Partition gebootet. Dadurch kann der M4 nach kurzer Zeit (ca. 20ms) bereits mit seiner Ablaufsteuerung starten.
- 2. M4 startet Linux:** Dieses Szenario ist ein bekannter Anwendungsfall für ein Android-Auto. Die SCFW erstellt eine Partition für den M4. Anschließend wird die M4-Partition gebootet. Der M4 erstellt zur Laufzeit eine Partition für das Linux und bootet diese Partition [63].
- 3. Linux startet M4:** In diesem Szenario bootet die SCFW ausschließlich die Linux-Partition, die zur Laufzeit eine Partition für den M4 erstellt und diesen startet.

Die drei vorgestellten Systemboot-Szenarien können auf die bereits vorgestellten Architekturen für AMP-Systeme abgebildet werden (Kap. 3.1). Das erste Boot-Szenario entspricht aufgrund der Trennung von unterschiedlichen sicherheitskritischen Applikationen einem MCS. Das zweite Boot-Szenario für ein Android-Auto kann über das LCM die Ausführung von Linux kontrollieren und das Linux abschalten, wenn keine rechenintensive Aufgaben anstehen und dadurch den Energieverbrauch reduzieren. Dies entspricht dem vorgestellten LPS. Das letzte Boot-Szenario entspricht dem CS, das gezielt Anwendungen auf Coprozessoren wie dem M4 auf dem i.MX8X auslagert und miteinander über eine IPC kommuniziert. Durch die bereits aufgezeigte Argumentation wird für das MMAMP-System die CS-Architektur eingesetzt (Kap. 3.1). Daher wird das dritte Boot-Szenario verwendet.

Kapitel 6

Konzept

Das folgende Kapitel beschreibt den Systementwurf und das Systemboot-Management für das MMAMP-System auf dem MPSoC i.MX8X.

6.1 Systementwurf

Der Systementwurf leitet sich aus den definierten Anforderungen (Kap. 3), dem eingesetzten Framework OpenAMP (Kap. 4) und der beschriebenen Architektur der i.MX8X-Plattform (Kap. 5) ab. Zusätzlich wird im zur Verfügung gestellten Linux-Kernel vom Hersteller NXP des eingesetzten MPSoC die bereits bestehenden Softwarekomponenten eingesetzt, um den Implementierungsaufwand zu reduzieren und eine Wiederverwendbarkeit des Systementwurfs zu gewährleisten. Die Aufteilung des Systementwurfs erfolgt in einzelne Komponenten aufgrund der Trennung von einzelnen Aufgabenbereichen für eine hohe Kohäsion, eine Wiederverwendbarkeit einzelner Komponenten zu gewährleisten und eine verständliche und präzise Übersicht der eingesetzten Komponenten zu erhalten.

Da das Ziel des Systementwurfs die Darstellung der eingesetzten Komponenten und Schnittstellen ist und daher die Struktursicht und einen Grobentwurf darstellt, wird er mithilfe von einem Blockdiagramm dargestellt und auf den Einsatz von UML verzichtet.

Der Systementwurf setzt sich aus drei Hauptkomponenten (weiß, Abb. 6.1) zusammen, die wiederum aus mehreren Komponenten aufgebaut sind. Die ersten beiden Hauptkomponenten befinden sich auf dem Linux, das auf der APU abläuft und in Applikationen im User- und Kernelspace aufgeteilt sind. Die letzte Komponente befindet sich auf der RPU und stellt die Firmware für den M4 dar. Für die Komponenten im Kernelspace und auf der RPU wurde der Aufbau des OpenAMP-Frameworks als Vorbild genommen (Abb. 3.1).

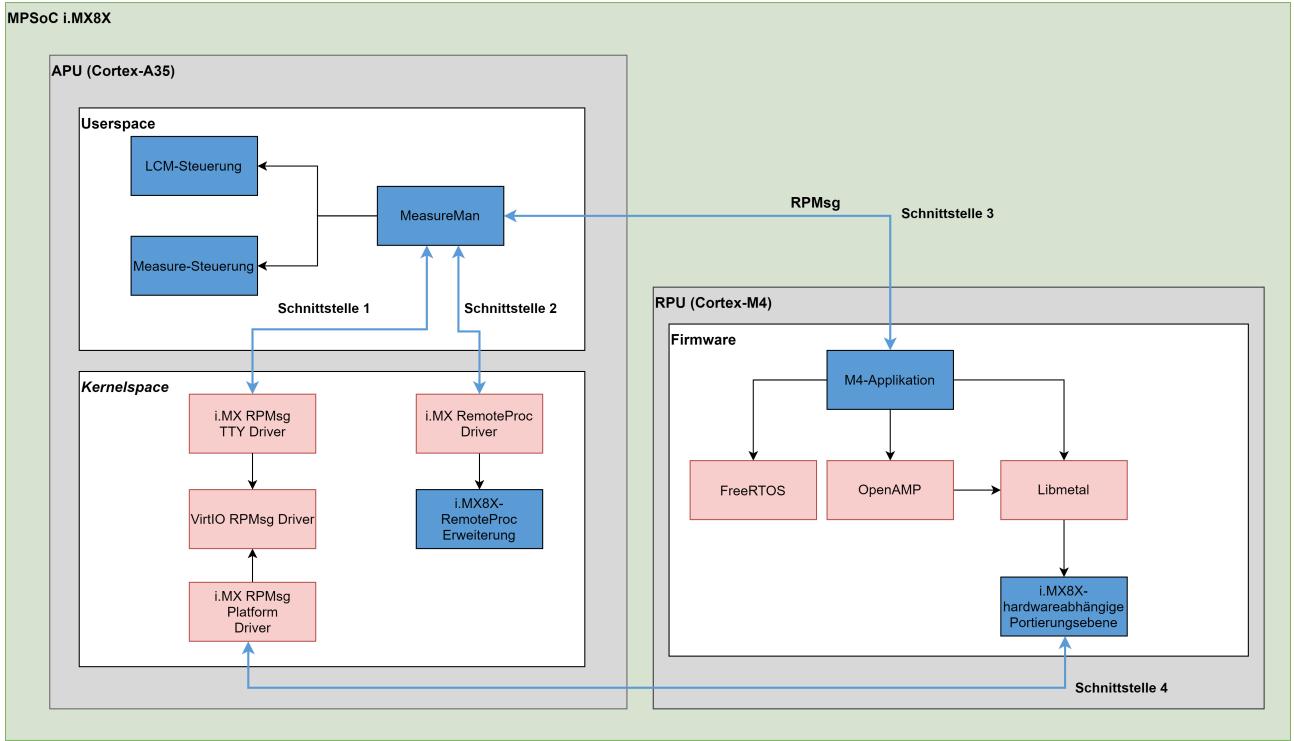


Abbildung 6.1: MMAMP: Systementwurf des AMP-Systems. Auf den integrierten Prozessoren Cortex-A35 und Cortex-M4 (grau) auf dem MPSoC i.MX8X (grün) bauen die darauf aufbauenden Softwaresysteme (weiß) auf. Auf dem Cortex-A35 läuft ein Linux ab, das in den User- und Kernelspace unterteilt wird (weiße Kästchen). Auf dem Cortex-M4 läuft eine Firmware ab, die wieder aus mehreren Komponenten besteht (weiße Kästchen). Innerhalb der weißen Kästchen werden Komponenten hierarchisch in blau und rot dargestellt. Rote Komponenten sind Fremdleistungen und blaue Komponenten sind Eigenleistungen. Die schwarzen Pfeile stellen die Abhängigkeiten zwischen den Komponenten dar. Blaue Pfeile präsentieren die Schnittstellen zwischen den einzelnen Komponenten.

Die folgenden Abschnitte stellen die einzelnen Komponenten und Schnittstellen des Systems näher vor.

6.1.1 Komponente MeasureMan im Userspace

Die Komponente MeasureMan befindet sich im Userspace und hat drei zentrale Aufgaben. Ihre erste Hauptaufgabe ist die Steuerung des LCM und übernimmt daher die Rolle des Master-Prozessors. Sie stellt dem i.MX RemoteProc Driver die zu ladende Firmware für den Cortex-M4 im Dateisystem bereit und informiert über das virtuelle Dateisystem *Sysfs* über eine neu abgelegte Firmware (Schnittstelle 2). Die zweite Hauptaufgabe ist die IPC über RPMsg zwischen Userspace und Coprozessor mithilfe der Schnittstelle 3, die den Kommunikationsfluss zwischen den beiden Prozessoren gewährleistet. Der Datenfluss der ausgetauschten Nachrichten läuft über die Kommunikation mit dem i.MX RPMsg Teletypewriter (TTY) Driver über ein TTY-RPMsg-Interface ab (Schnittstelle 1). Die dritte Hauptaufgabe ist die Steuerung und Ausführung von Messungen, die notwendig für die Durchführung der Evaluation des umgesetzten MMAMP-Systems sind.

6.1.2 Komponente Remoteproc und RPMsg Komponenten im Kernelspace

Die Komponenten im Kernelspace unterteilen sich in zwei Bereiche. Der erste Bereich stellt die Funktionalitäten für die IPC über RPMsg zur Verfügung und besteht aus folgenden Komponenten:

- **i.MX RPMsg TTY Driver** ist für die Kommunikation zwischen User- und Kernelspace verantwortlich und wird als ladefähiges Kernelmodul in den Kernel eingebunden. Er stellt ein serielles TTY-RPMsg-Interface zur Verfügung, welches vom MeasureMan eingesetzt wird, um Nachrichten an den M4 zu senden und Nachrichten vom M4 zu empfangen (Schnittstelle 1).
- **VirtIO RPMsg Driver** übernimmt die Verwaltung der VirtIO-Devices, die aus Virtqueues und VRings aufgebaut sind (Kap. 4).
- **i.MX RPMsg Platform Driver** erstellt die notwendigen VirtIO-Devices und registriert sie beim VirtIO RPMsg Driver. Zusätzlich läuft in diesem Driver die IPC über IPIs und Shared-Memory unter dem Einsatz von zwei MUs ab (Schnittstelle 4).

Der zweite Bereich stellt die Funktionalitäten für das LCM bereit:

- **i.MX RemoteProc Driver** stellt die Funktionalitäten für das LCM zur Verfügung. Nach einer genaueren Analyse dieser Komponente wurde festgestellt, dass er ausschließlich die MPSoC i.MX6SX und i.MX7D unterstützt. Daher kann dieser Driver nicht für die i.MX8X-Plattform eingesetzt werden, da keine Mechanismen für die Partitionen und Ressourcen unterstützen werden. Aus diesem Grund muss dieser Driver erweitert werden, damit er für die i.MX8X-Plattform eingesetzt werden kann.
- **i.MX8X RemoteProc Erweiterung** implementiert und berücksichtigt die i.MX8X Konzepte der Partitionen und Ressourcen. Dadurch kann das LCM über den i.MX RemoteProc Driver auf dem i.MX8X eingesetzt werden. Die Erweiterung wird innerhalb des i.MX RemoteProc Drivers umgesetzt.

6.1.3 M4 Firmware

Für die Entwicklung echtzeitfähiger Anwendungen wird für die RPU eine Applikation basierend auf dem Echtzeitbetriebssystem FreeRTOS eingesetzt, da für den M4 bereits eine Portierung von Variscite vorhanden ist und dadurch der Implementierungsaufwand gering ist. Neben FreeRTOS verwendet die Applikation die OpenAMP-Bibliothek und den dazugehörigen HAL-Layer Libmetal. Da bisher keine Implementierung für OpenAMP für die i.MX8X-Plattform existiert, ist zusätzlich die hardwareabhängige Portierungsebene für den i.MX8X notwendig, die portiert werden muss.

6.2 Entwurf für das Systemboot-Management

Im letzten Abschnitt wurde zwar der Systementwurf mit dem Schwerpunkt der IPC und dem LCM zwischen dem A35 und M4 beschrieben. Allerdings wurde noch kein Entwurf für das

Systemboot-Management für das MMAMP-System vorgestellt, was das LCM auf der i.MX8X-Plattform maßgeblich beeinflusst. Dies wird im folgenden Abschnitt behandelt.

Das Systemboot-Management baut auf dem vorher ausgewählten Bootsszenario 3 (Kap. 5.3) auf. Daher erstellt die SCFW keine Partition für den M4 und erstellt und startet nur die ATF-Partition. Nachdem die ATF-Partition die Linux-Partition gestartet hat, soll im Linux-Userspace die Applikation MeasureMan zur Laufzeit unterschiedliche M4-Firmwares laden und starten können. So ist es notwendig, dass zur Laufzeit die M4-Partition wieder freigegeben und wieder neu angelegt werden kann, um anschließend die neu geladene Firmware auf dem M4 zu starten (Abb. 6.2).

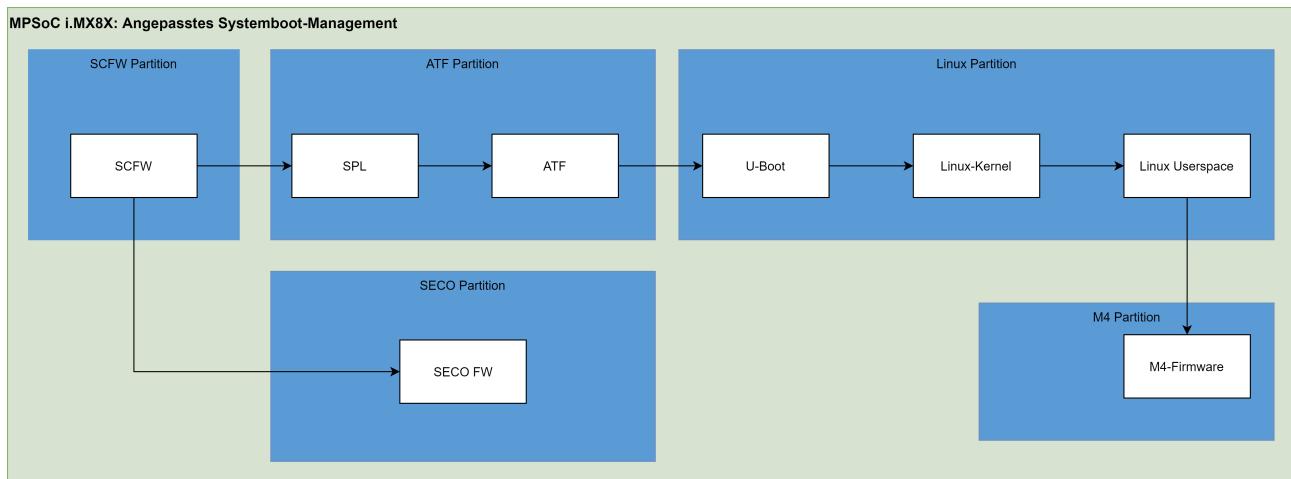


Abbildung 6.2: MMAMP: Systemboot-Management. Die auf dem MPSoC i.MX8X (grün) laufenden Partitionen (blau) sind aus mehreren Komponenten (weiß) aufgebaut, die zum Teil eine andere Komponente laden und starten (schwarze Pfeile). Der Ablauf des Bootprozesses für die SCFW-, SECO- und ATF-Partition ist aus Kapitel 5.3 zu entnehmen. Nachdem die ATF-Komponente die Linux-Partition gestartet hat, wird die U-Boot ausgeführt. Die U-Boot startet den Linux-Kernel gemeinsam mit dem Device-Tree. Nachdem der Kernel seine Initialisierungsphase beendet hat, startet er den ersten Prozess, den Init-Prozess, im Userspace. Daraufhin kann der MeasureMan gestartet werden, der eine M4-Firmware lädt, die M4-Partition erstellt und den M4 bootet.

Durch den aufgezeigten Entwurf für das angepasste Systemboot-Management sind einige Änderungen in den einzelnen Komponenten innerhalb der Partitionen notwendig, die in Kapitel 7.3 näher betrachtet werden.

Kapitel 7

Umsetzung

In diesem Kapitel wird die Umsetzung des definierten Konzepts für das aufgestellte MMAMP-System dargestellt. Beginnend mit dem eingesetzten Development-Board wird anschließend die Übersicht zur Softwarestruktur und Entwicklungsumgebung behandelt, danach die Umsetzung für das Systemboot-Management. Die nächsten zwei Abschnitte umfassen die Umsetzung des LCM und der IPC über RemoteProc und RPMsg im eingesetzten Linux auf dem A35. Daraufhin erfolgt die Portierung von OpenAMP auf den M4. Der letzte Abschnitt stellt die Userspace Applikation MeasureMan vor, welche die umgesetzten Komponenten für die Steuerung des LCMs und die IPC einsetzt.

7.1 Eingesetztes Development-Board Symphony-Board

Für den ausgewählten Variscite VAR-SOM-MX8X basierend auf dem MPSoC i.MX8X von NXP wird zusätzlich von Variscite das Development-Board Symphony-Board für die Umsetzung eingesetzt (Abb. 7.1). Auf dem Development-Board sind die üblichen Schnittstellen wie Ethernet, CAN, GPIO, I2C, SPI und UART sowie Display- und Audioschnittstellen integriert. Folgende Schnittstellen werden in der Umsetzung eingesetzt:

- MicroSD-Schnittstelle für den Einsatz einer MicroSD-Karte als Boot-Medium.
- Ethernet-Schnittstelle für den SSH-Zugang und NFS-Boot.
- JTAG-Schnittstelle für den Einsatz des Segger J-Link GDB Server, um dem M4 unabhängig vom A35 zu debuggen [64].
- GPIO-Schnittstellen für das Toggeln für GPIOs, die später für die Evaluierung benötigt werden (Kap. 8).

Die genaue Zuordnung der Schnittstellen zu den herausgeführten Stifteleisten und dem VAR-SOM-MX8X über eine SOM-Schnittstelle wird nicht erklärt und sind [65] zu entnehmen.

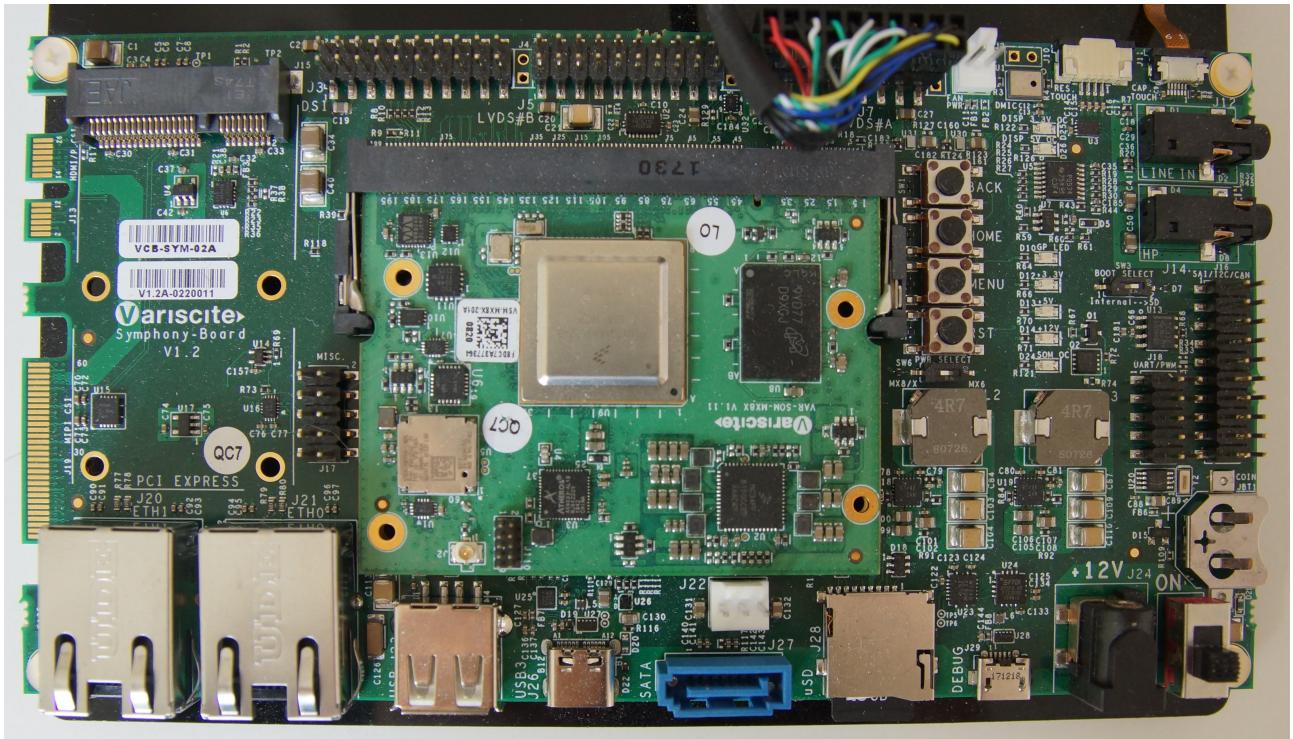


Abbildung 7.1: Eingesetztes Development-Board Symphony-Board von Variscite mit dem aufgesteckten System on Module (SOM) VAR-SOM-MX8X.

7.2 Übersicht zur Softwarestruktur und Entwicklungsumgebung

Die Softwarestruktur auf dem eingesetzten MPSoC i.MX8X besteht aus zwei unterschiedlichen Betriebssystemen auf der APU und RPU. In den folgenden Abschnitten wird die APU als A35 und die RPU als M4 bezeichnet.

Auf dem A35 wird das von NXP zur Verfügung gestellte SDK eingesetzt, das die notwendigen Werkzeuge für das Cross-Kompilieren, Linken und Erstellen von Embedded-Linux Betriebssystemen mit Yocto bereitstellt. Jedoch wird nicht direkt das SDK von NXP eingesetzt, sondern eine modifizierte SDK Variante von Variscite, die an das eingesetzte Entwicklungsboard Symphony-Board angepasst ist. Die zum Zeitpunkt dieser Arbeit aktuelle Version des SDKs ist *sumo-fsl-4.14.98-mx8x-v2.0* mit der Kernel-Version 4.14.98. Über das SDK wird ein Image aus dem Linux-Kernel, der U-Boot, dem RootFS und dem Device-Tree erstellt. Als Basis wird von Yocto das Image *fsl-image-gui* eingesetzt. Beim Bootprozess wird im Normalbetrieb das Kernel-Image aus dem nichtflüchtigen Speicher in den RAM geladen und von dort aus ausgeführt. Allerdings wird während der Entwicklung das Kernel-Image über Trivial File Transfer Protocol (TFTP) in den RAM geladen und das RootFS über Network File System (NFS) gestartet, wodurch die Dauer der Entwicklungszyklen gering gehalten werden kann. Dies erfolgt durch eine einmalige Anpassung der Kernel-Commandline in der U-Boot beim Systemstart.

Für die Entwicklung auf dem M4 wird ein angepasstes MCUXPresso SDK von Variscite mit der Version *mcuxpresso_sdk_2.5.x-var01* eingesetzt, das bereits FreeRTOS-Projekte mit einer vorhandenen Portierung für dem M4 als Projekttemplate bereitstellt und eine ARM GCC Toolchain für die Cross-Kompilierung anbietet.

Für das Booten des MPSoCs muss ein Boot-Image aus den generierten Firmwares für den A35 und M4 erstellt werden. Das Boot-Image wird mit dem NXP Tool *imx-mkimage* erstellt und besteht aus der SCFW-, SECO-, ATF-Firmware, SPL, dem Linux-Betriebssystem und dem dazugehörigen RootFS. Die Firmware für den M4 ist nicht Teil des Boot-Images, da diese über RemoteProc von Linux aus gestartet wird (Abb. 6.1). Die SCFW liest das Boot-Image über die MicroSD-Karte beim Systemboot aus und führt den Systembootprozess durch.

7.3 Systemboot-Management

Der in Kapitel 6.2 vorgestellte Entwurf für das Systemboot-Management wird in diesem Abschnitt implementiert und wird in die notwendigen Änderungen in den jeweiligen Partitionen untergliedert. Der Schwerpunkt liegt hierbei auf den Herausforderungen Ressourcen-, Speicher- und Lifecycle-Management auf AMP-Systemen (Kap. 2.6).

7.3.1 ATF-Partition

Innerhalb der ATF-Partition trennt die ATF-Firmware die normale Welt von der sicheren Welt und behält deswegen wichtige Ressourcen innerhalb der ATF-Partition. Einer der wichtigsten Ressourcen ist die *SC_R_SYSTEM*-Ressource, die als Systemressourc bezeichnet wird. Der Eigentümer dieser Ressource oder Partitionen, die Lese- und Schreibrechte auf diese Ressource haben, können sicherheitsrelevante Funktionen über die SCFW-API aufrufen (Kap. 5.2.1). So können darüber einzelne Partitionen oder der gesamte MPSoC neugestartet werden, die im Bereich des PMS fallen (Kap. 5.2.2). Auch die Durchführung von One Time Pad (OTP)-Verschlüsselungen kann ausschließlich mit Zugriff auf diese Ressource erfolgen.

Aus diesen Gründen überträgt die ATF-Firmware den Zugriff der Systemressource nicht standardgemäß an andere Partitionen, da hierdurch die Stabilität und Sicherheit des Systems gefährdet wird. Jedoch muss die Systemressource an die Linux-Partition übertragen werden, damit das LCM zur Laufzeit für den M4 über seine Partition ablaufen kann. Um trotzdem eine hohe Sicherheit zu gewährleisten, werden ausschließlich innerhalb des RemoteProc-Kernelmoduls Partitionen und Speicherregionen geändert. Die restlichen Ressourcen werden von der ATF-Firmware an die Linux-Partition übergeben, damit die Linux-Partition die Verwaltung auf die Ressourcen wie MUs, GPIOs, Interrupt-Controller übernehmen kann. Diese Ressourcen werden benötigt, damit die Linux-Partition Zugang zu seiner MU hat, um mit der SCFW zu kommunizieren oder über den Interrupt-Controller Interrupts auszulösen oder zu empfangen.

Für die Speicherbereiche müssen ebenso Änderungen innerhalb der ATF-Firmware durchgeführt werden. Bisher erstellt die ATF-Firmware den Speicherbereich A von *0x80020000* bis *0xFFFFFFFF*. Dieser Speicherbereich beinhaltet den Shared-Memory Bereich für die IPC zwischen A35 und M4 (Tab. 7.1). Während der Linux-Laufzeit müssen dem M4 Zugriffsrechte auf den Shared-Memory Bereich C von *0x90000000* bis *0x91FFFFFF* gegeben werden. Dafür müsste zur Laufzeit der Speicherbereich A aufgeteilt werden. Dies kann nicht zur Linux-Laufzeit erfolgen, da sonst der Speicher fragmentiert wird, in dem der Linux-Kernel Speicher angelegt hat und durch die Fragmentierung abstürzen würde. Aus diesem Grund muss in der ATF-Firmware der Speicherbereich A von *0xFFFFFFFF* auf *0x87FFFFFF* reduziert werden. Im Bereich von *0x88000000* und *0x8FFFFFFF* liegt der Offchip-DDR-Speicherbereich B für den M4 ab.

Neben dem Shared-Memory muss der M4 Speicherbereich D von *0x34000000* bis *0x37000000* an den M4 übergeben werden, damit er auf seinen TCM-Bereich und andere On-Chip Geräte Zugriff hat.

Einsatz	Speicherbereiche	Zugriff
A: Linux Speicherbereich	0x80020000 - 0x87FFFFFF	A35
B: M4 DDR Speicherbereich	0x88000000 - 0x8FFFFFFF	M4
C: IPC Shared-Memory	0x90000000 - 0x91FFFFFF	A35, M4
D: M4 Speicherbereich	0x34000000 - 0x37000000	M4

Tabelle 7.1: Einsatz und Zugriff auf die eingesetzten Speicherbereiche für den A35 und M4 (Quelle:[57, S. 22-52]).

Weitere Details zum Systemspeicherabbild des i.MX8X ist [57, S. 22-52] zu entnehmen.

7.3.2 Linux-Partition

Durch den Zugriff auf die Systemressource wird der Speicherbereich C erstellt. Anschließend wird der Zugriff auf den Speicherbereich C für die M4-Partition freigegeben und der Speicherbereich D der M4-Partition übergeben. Die Ressourcen der M4-MU und UART-Schnittstelle werden auch an die M4-Partition weiter übergeben und der Zugriff auf den Interrupt-Controller wird freigegeben. Dies erfolgt innerhalb des i.MX RemoteProc-Drivers.

7.3.3 M4-Partition

Durch die Vorarbeit an den ATF- und Linux-Partitionen hat die M4-Partition Zugriff auf ihre benötigten Speicherbereiche und Ressourcen und muss nur noch über die SCFW-API die Ressourcen anfragen und verwenden.

7.4 LCM mit RemoteProc im Kernelspace

Bevor die Portierung von OpenAMP auf dem M4 umgesetzt wird, erfolgt zuerst auf dem Linux die Umsetzung für das Laden der M4-Firmware über RemoteProc, wodurch später die Entwicklungszyklen bei der OpenAMP-Portierung kürzer gehalten werden, da die Firmware über RemoteProc zur Laufzeit immer wieder neu gestartet werden kann. Dadurch muss nicht bei jedem erneuten Entwicklungszyklus das Boot-Image mit der geänderten M4-Firmware erstellt werden. In Abbildung 6.1 ist dieser Teil durch die Komponenten i.MX RemoteProc Driver und i.MX8X-RemoteProc Erweiterung dargestellt. In Kapitel 3.4.3 wurde bei der Analyse festgestellt, dass der eingesetzte Kernel keine Möglichkeiten für das Laden der M4-Firmware auf dem MPSoC i.MX8X bereitstellt. Daher muss der bestehende RemoteProc-Driver erweitert werden, der als Ladbares Kernelmodul (LKM) in den Kernel dazugeladen wird. Die folgenden Absätze beschreiben die Änderungen, welche notwendig für die Umsetzung des LCM auf dem i.MX8X sind und entsprechen den vorgestellten Schritten zum Laden der Firmware (Abb. 4.3).

Ablage der Firmware

Für die Ablage der M4-Firmware werden zwei Möglichkeiten vom eingesetzten MCUXPressoSDK angeboten. Die erste Möglichkeit ist, die M4-Firmware im DDR-Bereich abzulegen. In diesem Fall sind die Zugriffszeiten länger, da der DDR-Bereich im Off-Chip Speicher liegt (Tab. 2.1). Die zweite Möglichkeit ist, die M4-Firmware im TCM abzulegen, welcher On-Chip auf dem M4 integriert ist. Aufgrund der niedrigeren Speicherzugriffszeiten wird die M4-Firmware in dem vorhandenen TCM des M4 abgelegt.

Repräsentation der Firmware als ELF

Damit der i.MX RemoteProc-Driver die Firmware im Speicher ablegen kann, muss diese zuerst als ELF übergeben, geparst und in den Speicher geladen werden. In diesem Fall repräsentiert die ELF den Aufbau eines ausführbaren Programms und besteht aus einem ELF-Header, der Programmheader-Tabelle und der Sektionsheader-Tabelle. Die Programmheader-Tabelle besteht aus mehreren Einträgen, von dem jeder eine Firmwaresektion repräsentiert und aus einer physikalischen Adresse, Sektions- und Speichergröße aufgebaut ist. Der Aufbau der ELF ist durch das Linkerskript veränderbar. Weitere Details für den Aufbau einer ELF ist [66] zu entnehmen.

Parsen der Firmware als ELF

Aufgrund des zur Verfügung gestellten Linkerskripts durch das MCUXPresso SDK von Variscite, das die M4-Firmware im TCM ablegt, werden keine Änderungen am Linkerskript und somit an der generierten ELF durchgeführt. Beim Parsen der einzelnen Programmheader-Einträgen der ELF ist zu beachten, dass der A35 eine andere Speicheransicht als der M4 hat (Tab. 7.2). Daher muss die im Eintrag gegebene physikalische Adresse aus M4-Speicheransicht zuerst in die A35-Speicheransicht übersetzt werden. Anschließend wird eine Anfrage mit der übersetzten physikalischen Speicheradresse an die MMU gestellt, welche die virtuelle Adresse zu dem angefragten Bereich zurückliefert. Abschließend wird die Firmwaresektion des parsenden Programmheader-Eintrags an der gegebenen virtuellen Adresse abgelegt, die vom M4 bei seiner Ausführung erwartet wird. Damit der RemoteProc-Driver die Firmwaresektion aus der ELF an den Speicheradressen des TCM-Speichers ablegt, die vom M4 erwartet werden, wurden im Driver dafür Anpassungen durchgeführt. Besonders wichtig ist hierbei, dass auch unterschiedlich große Firmwaresektionen geladen werden können. Daher erfolgt eine Adressübersetzung zwischen M4 und A35 Speicheransicht in Abhängigkeit der angegebenen physikalischen Speicheradresse der M4 Firmwaresektion.

Speicherbereich	A35 Speicheransicht	M4 Speicheransicht
TCMU	0x35000000 - 0x3501FFFF	0x20000000 - 0x2001FFFF
TCML	0x34FE0000 - 0x34FFFFFF	0x1FFE0000 - 0x1FFFFFFF

Tabelle 7.2: i.MX8X TCM-Speicherbereich vom M4 innerhalb vom Speicherbereich D aus der Sichtweise vom A35 und M4 [57, S. 22-52].

Erstellung der VirtIO-Devices

Aufgrund der Erstellung der VirtIO-Devices durch den bestehenden i.MX RPMsg Platform Driver beim Starten des Kernels kann auf das Parsen der VirtIO-Devices verzichtet werden.

Erstellung der Partitionen und Zuweisung der Ressourcen und Speicherbereiche

Nach dem Auslesen der Firmwaresektionen aus der ELF erstellt die i.MX8X RemoteProc Komponente eine Partition für den M4 und überträgt die Zugriffsrechte auf die notwendigen Speicherbereiche und Ressourcen (Kap. 7.3). Die Zuteilung erfolgt über die SCFW-API, die von der i.MX8X-Plattform zur Verfügung gestellt wird (Kapitel 5.2.3). Für die Speicherbereiche werden die zwei Bereiche C und D angelegt (Tab. 7.1). Auf dem Speicherbereich D hat ausschließlich der M4 Zugriff. Auf den Shared-Memory Speicherbereich C haben beide Prozessoren Zugriff, da dieser für RPMsg eingesetzt wird und die Vrings und die angelegten Buffers beinhaltet. Damit der M4 über eine MU mit dem A35 und der SCU über die SCFW-API kommunizieren kann und die Rechte auf das Schreiben der UART-Schnittstelle hat, werden ihm die Ressourcen zugeteilt.

Abschließend wird über den PMS der SCFW-API die M4 Partition gestartet, wodurch der Cortex-M4 mit Takt versorgt und der Reset ausgelöst wird.

7.5 IPC mit RPMsg im Kernelspace

Bereits in Kapitel 6.1 wurden die RPMsg-Komponenten i.MX RPMsg TTY Driver, VirtIO RPMsg Driver und der i.MX RPMsg Platform Driver vorgestellt, jedoch wurde noch nicht näher auf die Interaktion zwischen den Drivern und die M4-Kommunikation eingegangen. Dies wird in den folgenden Abschnitten beschrieben, da für die Portierung von OpenAMP auf den M4 (Kap. 7.6) und für die Evaluierung (Kap. 8) ein genaueres Verständnis für die bereits bestehende und eingesetzte RPMsg-Komponente auf dem Linux notwendig ist. Die eingesetzten Komponenten basieren auf dem Device-Driver-Modell, welches vom Linux-Kernel eingesetzt wird [67].

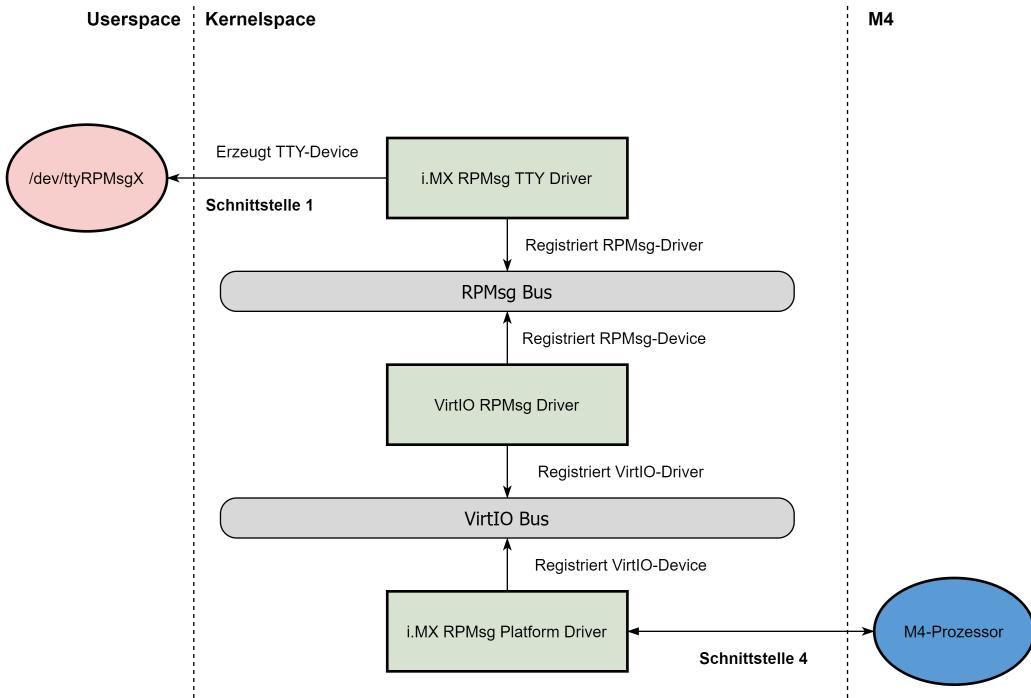


Abbildung 7.2: RPMsg-Architektur auf dem eingesetzten Linux. Die Architektur wird in die drei Bereiche User- sowie Kernelspace und M4 eingeteilt (gestrichelte Linien). Im Userspace steht das TTY-Device (rot) zur Verfügung. Im Kernelspace sind die notwendigen Driver für die RPMsg-Kommunikation (grün) dargestellt, die über Busse (grau) miteinander Informationen austauschen. Der letzte Bereich wird durch den M4-Prozessor (blau) dargestellt (umgezeichnet nach [68]). Details siehe Text.

i.MX RPMsg TTY Driver

Der i.MX RPMsg Driver registriert sich für ein RPMsg-Device mit dem RPMsg-Kanalnamen `rpmsg-openamp-demo-channel`. Wenn ein solches Device gefunden wird, erstellt dieser Driver ein TTY-Device `/dev/ttyRPMsgX`, welches Userspace-Anwendungen für die M4-Kommunikation verwenden können (Schnittstelle 1, Abb. 7.2). Dieser Driver wird als LKM eingebunden und unterstützt die Operationen `open`, `close`, `read` und `write`. Über `open` und `close` kann das Device geöffnet und geschlossen werden. Mit `read` wird auf eine empfangene Nachricht vom M4 gewartet (blockierender Zugriff) und mit `write` wird die übergebene Nachricht an den M4 versendet.

VirtIO RPMsg Driver

Der VirtIO RPMsg Driver registriert sich bei dem VirtIO Bus und wartet auf Registrierungsanfragen von dem VirtIO-Device (Abb. 7.2). Sobald ein VirtIO-Device eine Anfrage schickt, erzeugt der Driver das VirtIO-Device und bildet darauf die im VirtIO-Device enthaltenen Virtqueues und die Buffer-Bereiche im Shared-Memory ab. Besonders wichtig ist beim Shared-Memory die Anzahl und Größe der RPMsg-Buffer (Tab. 7.3). Die angegebenen Eigenschaften für den RPMsg-Buffer müssen später bei der Portierung von OpenAMP auf dem M4 übereinstimmen, da sonst die Kommunikation nicht über ein RPMsg-Device aufgebaut werden kann. Für die Vermeidung von unerwartetem Systemverhalten werden diese Einträge übernommen und nicht modifiziert. Eine weitere wichtige Aufgabe ist die An- und Abmeldung von RPMsg-Kanälen. Sobald eine Nachricht des Typs Nameservice-Announcement empfangen wird, registriert der Driver ein RPMsg-Device beim RPMsg Bus.

Beschreibung der RPMsg-Buffer

RPMsg-Buffer-Größe: 512 Bytes

Anzahl der Buffer: 256

Tabelle 7.3: Beschreibung des RPMsg-Buffers.

i.MX RPMsg Platform Driver

Der i.MX RPMsg Platform Driver wird von NXP zur Verfügung gestellt und ist plattform-abhängig. Über den angelegten Device-Tree Eintrag für die MU registriert dieser Driver die MU, damit darüber eine IPC zwischen A35 und M4 ablaufen kann (Schnittstelle 4). Das Auslösen des IPIs in Richtung des M4 wird über den Interrupt-Vektor 3 und innerhalb der Funktion *imx_rpmsg_notify()* ausgeführt. Für die entgegengesetzte Richtung wird der IPI 2 eingesetzt und die hinterlegte Interrupt Service Routine (ISR) *imx_mu_rpmsg_isr()* aufgerufen (Abb. 7.3).

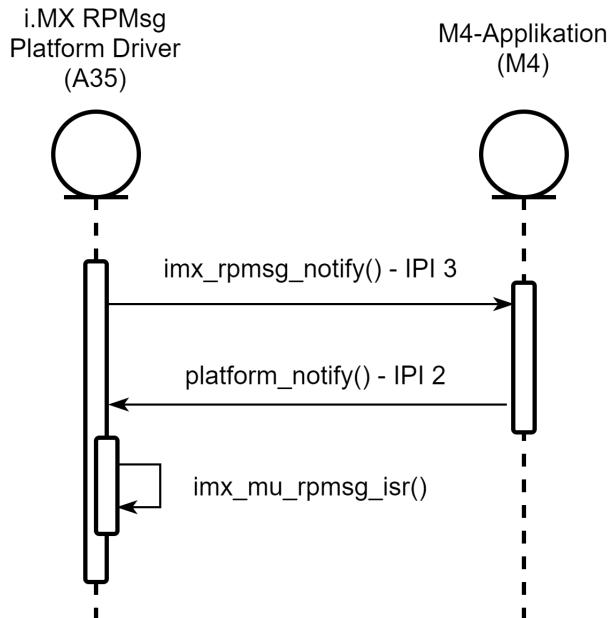


Abbildung 7.3: UML-Sequenzdiagramm für die IPC zwischen A35 und M4 mit zwei IPIs.

Die Registrierung des VirtIO-Devices beim VirtIO Bus übernimmt der Driver und übergibt hierbei die Speicherbereiche für die eingesetzten Virtqueues und dem notwendigen Ringbuffer (Tab. 7.4). Die drei Speicherbereiche werden durch den bestehenden i.MX8X Device-Tree durch *reserved-memory* Knoten beschrieben.

Speicherbereich	Startadresse	Speichergröße
Virtqueue Tx-VRing	0x90010000	32kB
Virtqueue Rx-VRing	0x90018000	32kB
Ringbuffer	0x90400000	29360kB

Tabelle 7.4: Aufteilung der Speicherbereiche für die VirtIO-Devices. Die Vrings und der Ringbuffer repräsentieren hierbei die Sicherungsschicht (Kap. 4.2).

7.6 Portierung von OpenAMP auf den M4

In den letzten beiden Abschnitten wurde die Umsetzung der Komponente RemoteProc auf dem i.MX8X und dadurch die Voraussetzungen für das LCM zur Laufzeit des Linux auf dem A35 erfüllt und die eingesetzte RPMsg-Komponente erklärt. Darauf aufbauend beschäftigt sich dieser Abschnitt mit der Umsetzung und Portierung von OpenAMP auf den M4.

Bei der Analyse von OpenAMP wurde festgestellt, dass die IPC über OpenAMP oder RPMsg-Lite erfolgen kann und NXP für RPMsgLite bereits eine Demoapplikation innerhalb des eingesetzten MCUXPresso SDK bereitstellt (Kap. 4.5). Daher wurde zuerst die vorhandene RPMsg-Lite Demo aufgesetzt und analysiert, um ein tieferes Verständnis auf Implementierungsebene für den RPMsg-Ablauf aufzubauen.

In [69] ist die Portierung von OpenAMP auf eine neue Plattform beschrieben und besteht aus mehreren Schritten. Allerdings bezieht sich die Dokumentation auf den Stand 2018 und ist daher nur begrenzt aussagekräftig, da in dieser Arbeit die aktuelle Version *v2020.01* der Libmetal und OpenAMP eingesetzt wird.

Die Portierung von OpenAMP unterteilt sich in folgende Schritte:

1. Portierung der hardwareabhangigen Systemumgebung für den M4 auf dem i.MX8X MP-SoC in die Libmetal. Die bisher portierten Plattformen beschränken sich auf die von Xilinx entwickelten MPSOC Zync-7000 und Zync Ultrascale.
2. Implementierung des plattformspezifischen RemoteProc-Drivers. Da in dieser Arbeit der A35 das LCM übernimmt, entfällt dieser Teil.
3. Definierung der eingesetzten Shared-Memory-Bereiche und der Ressourcentabelle. Besonders wichtig ist dabei die Übereinstimmung der definierten Shared-Memory-Bereiche mit dem Linuxsystem auf dem A35.
4. Implementierung der M4-Applikation.

Die folgenden Abschnitte stellen wichtige Aspekte der einzelnen Schritte detaillierter vor.

7.6.1 Portierung der Systemumgebung

Bei der Portierung des Hardwarelayers ist es wichtig, den zur Verfügung gestellten Interrupt-Controller innerhalb der Libmetal zu initialisieren, eine Interrupt-Vektortabelle zu erstellen und den zu verwendeten Interrupt-Vektor 3 mit der dazugehörigen ISR zu hinterlegen (Abb. 7.3). Die ISR wird ausgeführt, sobald der A35 eine Interrupt Request (IRQ) bei der darunterliegenden MU auslöst. Der M4 informiert den A35 über die IPI 2 über neue RPMsg-Nachrichten. Für den Zugriff auf die MU, die Initialisierung der IPIs sowie das Empfangen und Senden der IPIs wird bei der Portierung auf die vorhandene FreeRTOS-Softwareschicht zugegriffen (Abb. 6.1).

Neben der Abstraktion der IPIs durch die Libmetal werden zusätzlich die zur Verfügung gestellten Mechanismen für den Zugriff auf Speicherbereiche und den Logging-Mechanismus eingesetzt.

7.6.2 Anpassung der Shared-Memory und Ressourcen-Tabelle

Bei der Erstellung der VirtIO-Devices durch die Verwendung der OpenAMP-Bibliothek ist es besonders wichtig, die Speicherbereiche (Tab. 7.4) für die eingesetzten Vrings in der Ressourcentabelle (Abb. 7.4) zu hinterlegen. Dabei müssen die Speicherbereiche mit der Festlegung der Bereiche auf dem A35-System übereinstimmen.

Die Ressourcen-Tabelle beschreibt die Version der Tabelle, die Anzahl der Tabelleneinträge und die Offsets der RPMsg-Devices in der Tabelle. Im Anschluss folgen die Einträge für das RPMsg-Device und der Vrings. Besonders wichtig ist bei dem RPMsg-Device-Eintrag, dass der Ressourcentyp auf *RSC_VDEV* gesetzt ist, der einem RPMsg-Device entspricht. Ein weiterer Ressourcentyp ist *RSC_CARVEOUT*, der für die Zuweisung einer zusammenhängenden Speicherregion eingesetzt wird. Diese Ressource ist vorteilhaft, wenn die ELF der M4-Firmware aus mehreren Firmwaresektionen besteht, welche auf mehrere Speichertypen auf einem MPSoC verteilt sind. Damit kann der Linux RemoteProc-Driver anhand der angegebenen Speicherregionen in der Ressourcen-Tabelle die dazugehörigen Systemadressen mithilfe einer MMU abrufen und die Speichersektionen an die geforderte Adresse kopieren. Aufgrund der Ablage der M4-Firmware im TCM-Bereich ist die Verwendung von diesem Eintrag nicht notwendig. Ein anderer Ressourcentyp ist *RSC_TRACE* über welchen Dateien dem Coprozessor zur Verfügung gestellt werden. Die beiden Vrings werden durch die Variablen *rpmsg_vring0* und *rpmsg_vring1* repräsentiert und werden mit den angegebenen Vring-Bereichen (Tab. 7.4) und der angegebenen Buffer-Anzahl (Tab. 7.3) gefüllt. Für den Zugriff auf den Shared-Memory Buffer-Bereich wird über die Libmetal eine Speicherregion angelegt, die bei *0x90400000* startet (Tab. 7.4).

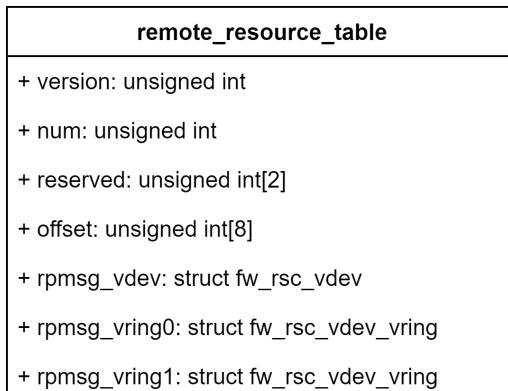


Abbildung 7.4: UML-Klassendiagramm für den Aufbau der Ressourcen-Tabelle, die für die Repräsentation des RPMsg-Devices und der beiden Vrings verantwortlich ist.

7.6.3 Implementierung der M4-Applikation

Die M4-Applikation hat die Aufgabe, mit dem Linux über RPMsg-Nachrichten zu kommunizieren. Dafür sind einige Teilaufgaben notwendig, die in den folgenden Abschnitten genauer beschrieben werden und als UML-Diagramm illustriert sind (Abb. 7.5).

Die erste Teilaufgabe der M4-Applikation ist die Initialisierung aller notwendigen i.MX8X Hardware-Ressourcen. Das beinhaltet die Kommunikation zur SCFW-API über eine MU, die UART-Schnittstelle zum Auslesen der Logausgaben, der Zugriff auf den Interrupt-Controller und schließlich die Initialisierung der MU für die Kommunikation zwischen A35 und M4 (Schritt

1). Nur nach einer erfolgreichen Initialisierung dieser Ressourcen können die folgenden Teilaufgaben durchgeführt werden. Besonders wichtig ist hierbei, dass über den i.MX8X-RemoteProc-Driver die Zugriffsrechte auf die Ressourcen für die M4-Partition erteilt wurden. Ansonsten kann die M4-Applikation nicht darauf zugreifen.

Anschließend werden die Libmetal-Ressourcen initialisiert. Der Libmetal IR-Controller registriert die IPIs und aktiviert sie (Kap. 7.6.1). Im Anschluss erfolgt das Auslesen der Einträge aus der Ressourcentabelle und das Anlegen der Speicherregionen für die Vring- und Bufferbereiche (Schritt 2).

Darauf initialisiert die Firmware das RPMsg-Device mit den hinterlegten Speicherregionen. Im Anschluss darauf kann das RPMsg-Device aufgebaut, der Kanal erzeugt und das dazugehörige Kanal-Announcement über den Kanal-Announcement Endpunkt 53 (Kap. 4.2) an den Master gesendet werden (Schritt 3 und 4). Der erzeugte Endpunkt der M4-Applikation hat die ID 35. Die Applikation ist ab diesem Zeitpunkt bereit Nachrichten über den angelegten Kanal zu verarbeiten.

Zum Prüfen neu empfangener Nachrichten über das RPMsg-Device stehen zwei Lösungs-ansätze zur Auswahl. Empfangene Nachrichten werden direkt in der ISR verarbeitet. Dieser Ansatz wird nicht weiter verfolgt, da ISRs möglichst kurz gehalten werden müssen. Der zweite Ansatz verwendet ein Benachrichtigungsflag, das in der ISR gesetzt wird, wodurch die ISR-Verarbeitungszeit kurz gehalten wird. Die Applikation prüft darüber, ob eine neue RPMsg-Nachricht empfangen wurde. Die Entscheidung fällt auf den zweiten Ansatz. Sobald eine Shutdown-Nachricht empfangen wird (Schritt 6), werden die angelegten Ressourcen (Schritt 1-4) abgebaut.

Ansonsten wird die empfangene RPMsg-Nachricht verarbeitet (Schritt 6.1) und wieder auf neue RPMsg-Nachrichten gewartet (Schritt 5).

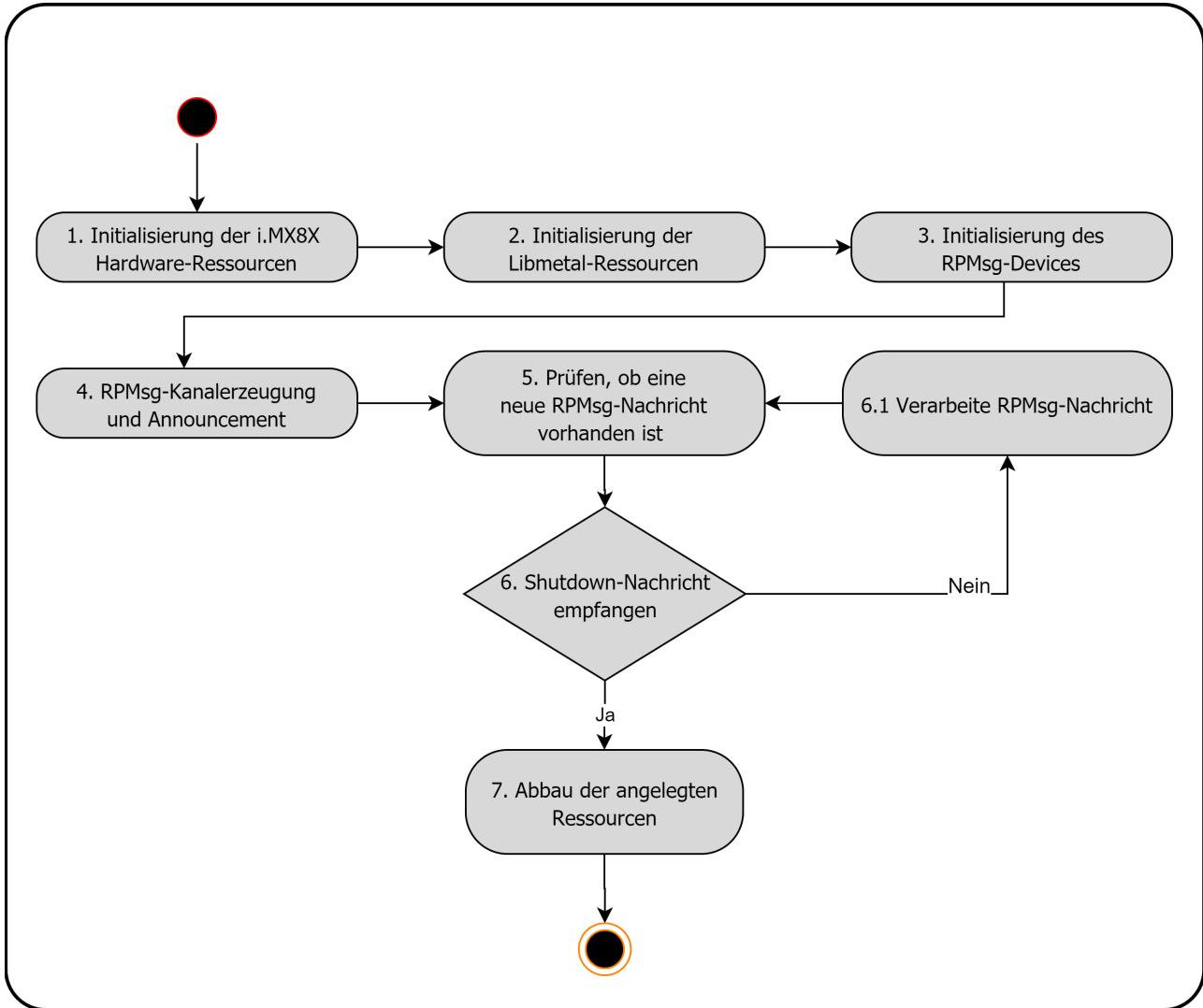


Abbildung 7.5: UML-Aktivitätsdiagramm für den Ablauf der Initialisierung der M4-Firmware. Details siehe Text.

7.7 Userspace Applikation MeasureMan

In den vorhergehenden Abschnitten wurden zwar die Umsetzung im Kernelspace für das LCM und die IPC sowie die Portierung von OpenAMP auf den M4 beschrieben. Es wurde aber nicht auf die notwendige Applikation eingegangen, die im Linux Userspace diese Komponenten verwenden soll. Daher beschreibt folgender Abschnitt die wichtigsten Punkte der umgesetzten Userspace Applikation, welche als MeasureMan bezeichnet wird.

Der MeasureMan ist für die drei folgenden Aufgaben verantwortlich (Kap. 6.1):

- Steuerung des LCM.
- Austausch von RPMsg-Nachrichten über IPC.
- Ausführung von Messungen für die Evaluation.

Da keine dieser Aufgaben eine hohe performante Rechenleistung erfordert und die Übertragung der Nachrichten innerhalb des Kernelspaces und auf M4-Seite ausgeführt werden, wird die Umsetzung der Applikation in der Skriptsprache Python umgesetzt, obwohl der Interpreter eine geringere Ausführungsgeschwindigkeit hat als die Programmiersprache C++. Allerdings ist der

Kosten-Nutzen-Faktor durch die Verwendung von C++ nicht gegeben.

Steuerung des LCM

Das Laden einer M4-Firmware erfolgt durch das Ablegen der Firmware unter dem Pfad `/lib/firmware`. Sobald der MeasureMan die Firmware abgelegt hat, wird das LKM i.MX RemoteProc Driver in den Kernel geladen, welcher anschließend die beschriebenen Abläufe (Kap. 7.4) durchführt und darüber den M4 startet. Der M4 erstellt das RPMsg-Device und informiert den A35 darüber. Im Anschluss lädt der MeasureMan den LKM i.MX RPMsg Driver in den Kernel, der zu dem erstellten RPMsg-Device das entsprechende TTY-Device `/dev/ttyRPMsg30` erstellt.

Für den Wechsel der Firmware benachrichtigt der MeasureMan den M4 über den Abbau des RPMsg-Devices. Im Anschluss entfernt der MeasureMan in entgegengesetzter Richtung die geladenen Kernelmodule. Abschließend wird über das vom RemoteProc-Driver zur Verfügung gestellte `sysfs`-Attribut mit dem Befehl `echo stop > /sys/class/remoteproc/remoteproc0/state` der Coprozessor gestoppt (Schnittstelle 2, Abb. 6.1). Die neu geladene Firmware muss unter dem Pfad `/lib/firmware` mit dem Dateinamen `rproc-imx-rproc-fw` abgelegt werden. Über den Befehl `echo start > /sys/class/remoteproc/remoteproc0/state` wird der Coprozessor mit der neu geladenen Firmware gestartet. Die aufgezählten Befehle werden innerhalb des MeasureMans ausgeführt.

Austausch von RPMsg-Nachrichten über IPC

Der Austausch der RPMsg-Nachrichten erfolgt über das TTY-Device, welches ausführlich in den letzten Abschnitten behandelt wurde. Dafür steht das RPMsg-Device als TTY-Device bereit.

Ausführung von Messungen für die Evaluation

Für die Ausführung der Messungen wurden die einzelnen Hauptversuche innerhalb der MeasureMan-Applikation implementiert und stellen die Measure-Steuerung (Abb. 6.1) dar. Die dafür eingesetzten Werkzeuge und Arten der Hauptversuche werden im nächsten Kapitel erläutert.

Kapitel 8

Evaluierung

Das folgende Kapitel beschäftigt sich mit der Evaluierung des umgesetzten Systems MMAMP mit OpenAMP auf dem MPSoC i.MX8X. Dafür werden zwei mögliche Anwendungsszenarien aus der Praxis vorgestellt, in denen MMAMP verwendet werden kann. Aus den Anwendungsszenarien werden relevante Aspekte identifiziert, die MMAMP gewährleisten muss. Im Anschluss werden basierend auf den relevanten Aspekten die Versuchsplanung beschrieben und die durchzuführenden Versuche vorgestellt. Abschließend folgt die Versuchsdurchführung und eine Diskussion der Messergebnisse.

8.1 Anwendungsszenarien

8.1.1 Prozessdatenüberwachung

Unabhängig ob in der Automobilbranche, beim Sonderanlagenmaschinenbau oder in der Medizintechnik müssen Maschinenanlagen in regelmäßigen Abständen gewartet werden, um Ausfälle zu minimieren und eine längere Lebensdauer zu erzielen. Um diese Prozesse möglichst effizient zu gestalten, hat sich in den letzten Jahren die Instandhaltungsstrategie Predictive Maintenance immer weiter verbreitet. Hierfür werden die Prozessdaten von Maschinen oder Anlagen wie Druck, Vibration, Temperatur oder Energieverbrauch überwacht, protokolliert und ausgewertet. Je nach Zustandsart werden Erfassungen bis in den kHz-Frequenzbereichen wichtig, wie bei der Vibrationsmessung von Elektromotoren, um im Notfall den Motor auszuschalten, um größeren Schaden zu verhindern. Hierfür wird ein Echtzeitcoprozessor benötigt, der die Prozessdaten in einer hohen Wiederholungsrate misst und im Notfall die betroffenen Motoren abschalten kann. Zusätzlich müssen Ausreißer an einen leistungsstarken Prozessor übertragen werden, um weitere Maßnahmen sowie Analysen durchzuführen und die Daten auf einem HMI anzuzeigen [70].

Zum anderen können die aufgezeichneten Prozessdaten für die Qualitätssicherung zur Erreichung einer hohen Produktqualität eingesetzt werden. So werden bei einer Schweißrobotersteuerung Produktionsprozessdaten wie Schweißparameter und Positionsdaten des Werkzeuges ausgelesen. Dies geschieht bis zu 1000 Mal in der Sekunde. Die Daten müssen anschließend auf einen leistungsstarken Prozessor übertragen, ausgewertet und gegebenenfalls Anpassungen an der Robotsteuerung durchgeführt werden [71].

Folgende Aspekte sind daher für die umgesetzte Lösung MMAMP für Predictive Maintenance und für die Qualitätssicherung relevant:

1. Der Datendurchsatz für die Übertragung der Prozessdaten vom M4 zu dem A35.
2. Die Latenzzeiten zwischen M4 und A35, um Ausreißer bei den aufgezeichneten Prozessdaten über ein HMI darzustellen, an eine Steuerungszentrale weiterzugeben oder Anpassungen auf der echtzeitkritischen Steuerung auf dem Coprozessor durchzuführen.
3. Die durch OpenAMP verursachte Verarbeitungszeit für die IPC auf dem M4, die möglichst gering sein soll, damit noch ausreichend Zeit für die Sensor- und Aktuatorenlogik vorhanden ist.

8.1.2 Hardware-in-the-Loop

Für den Test von Fahrzeugdynamiken- und Steuerungen in der Automobilindustrie bieten sich HiL-Systeme an, die während der Entwicklung oder Inbetriebnahme eingesetzt werden, um Entwicklungszyklen zu minimieren und Kosten zu sparen. Ein HiL-System besteht dabei aus einem HiL-Simulator, der eine virtuelle Echtzeitumgebung abbildet und einem Device Under Test (DUT), welches das zu testende Steuergerät ist (Abb. 8.1). Hierbei müssen viele Sensoren und Aktuatoren simuliert und die aufgezeichneten Daten während der Ausführung von Testszenarien ausgewertet werden. Außerdem bietet sich die Integration eines Continuous Integration (CI)-Servers für eine automatisierte Ausführung von verschiedenen Testszenarien mit anschließendem Reporting der Testergebnisse an. Für die HiL-Steuerung, die Auswertung und Anzeige von Daten sowie die Kommunikation zum CI-Server bietet sich der leistungsstarke A35 an, während der echtzeitfähige HiL-Simulator auf dem M4 ausgeführt wird und Sensoren und Aktuatoren simuliert. Zusätzlich übernimmt die HiL-Steuerung das Laden von unterschiedlichen Testszenarien mithilfe von verschiedenen Konfigurationsdateien oder Firmwares [72].

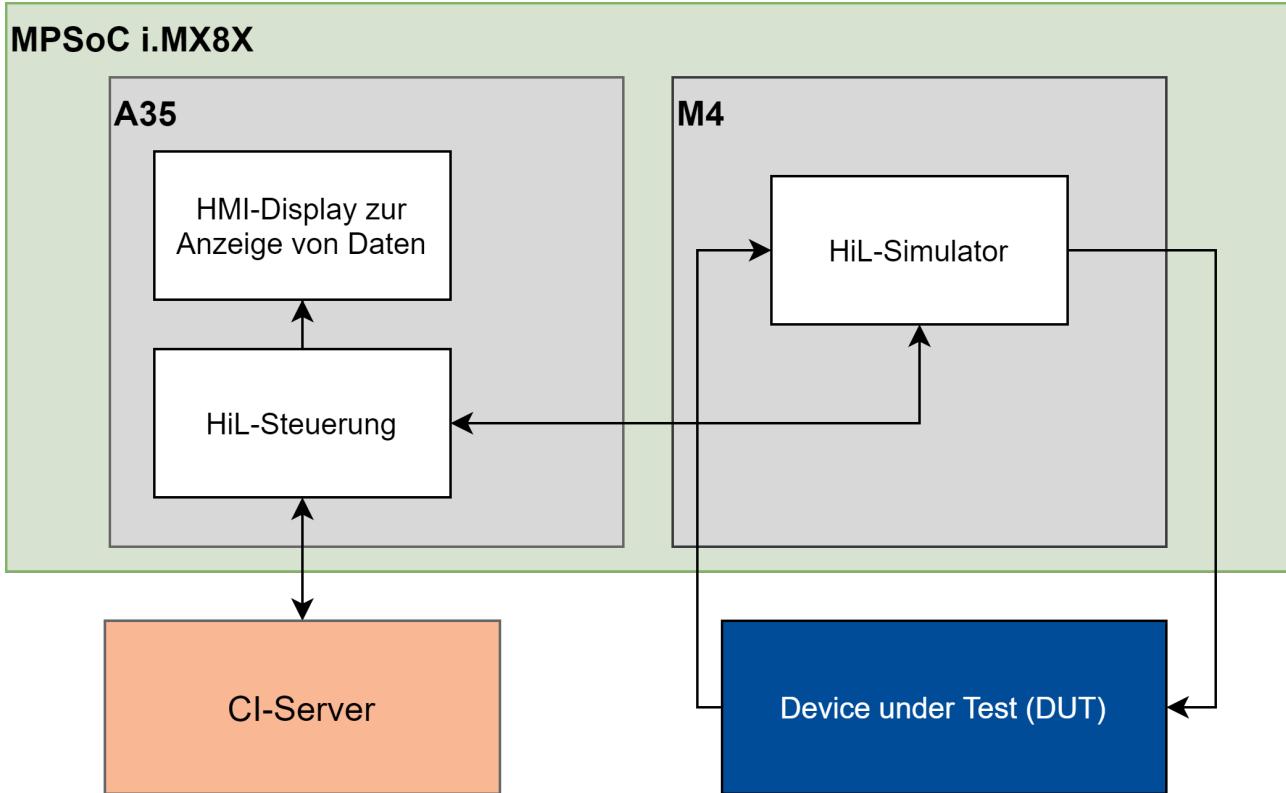


Abbildung 8.1: Systemüberblick zu einem möglichen HiL-System. Auf den integrierten Prozessoren A35 und M4 (grau) auf dem MPSoC i.MX8X (grün) bauen die darauf aufbauenden Softwaresysteme (weiß) auf. Auf dem A35 läuft die HiL-Steuerung, die empfangene Daten vom HiL-Simulator auf dem HMI-Display anzeigt. Zusätzlich ist sie für die Kommunikation zum CI-Server (orange) verantwortlich. Auf dem M4 läuft die HiL-Simulator Firmware, welche die Kommunikation der Sensoren und Aktuatoren zwischen sich und dem DUT (blau) steuert. Die schwarzen Pfeile stellen Kommunikationsrichtungen zwischen den einzelnen Komponenten dar.

Folgende Aspekte sind daher bei der umgesetzten Lösung MMAMP relevant:

- Der Datendurchsatz zwischen dem A35 und M4 für die Übertragung von Sensor- oder Konfigurationsdateien.
- Die Latenzzeiten beim Datenaustausch zwischen A35 und M4.
- Die Möglichkeit unterschiedliche Firmwares auf den HiL-Simulator über die HiL-Steuerung zur Laufzeit zu laden und auszuführen.

8.2 Versuchsplanung

Bevor die Versuchsdurchführung gestartet werden kann, müssen folgende Vorbereitungen berücksichtigt werden:

- Definierung der Ziele der Versuche sowie die aufgespannten Randbedingungen und die Beschreibung der Versuche.
- Eingesetzte Messwerkzeuge und Ungenauigkeiten.

8.2.1 Ziele der Versuche

Ausgehend von den Anwendungsszenarien werden folgende Ziele definiert:

- Messung der Latenzzeiten zwischen den Prozessoren auf den MPSoC.
- Messung des Datendurchsatzes zwischen den Prozessoren.
- Messung der Ausführungszeit der OpenAMP Kommunikation auf dem M4.
- Messung des LCMs.

8.2.2 Randbedingungen der Versuche

Die vorgestellten Versuche, auch als Hauptversuche bezeichnet, werden in mehrere untergeordnete Versuche unterteilt. Jeder Versuch besteht aus einem oder mehreren Experimenten, von dem jedes durch einen fest definierten Parametersatz beschrieben wird. Die mehrfache Ausführung eines Experiments innerhalb eines Versuches hängt vom einzelnen Versuch ab und wird deswegen in den einzelnen Versuchen individuell betrachtet. Die Hauptversuche unterscheiden sich durch die Änderungen des zu untersuchenden Parameters wie der Nachrichtengröße oder der Systemlast.

Auf die mehrfach aufgezeichneten Messungen werden statistische Mittel (arithmetische Mittel, Ausreißer, Streuung) angewendet. Die Anzahl der Stichproben der Messungen hängt vom Versuch ab und wird daher in den nachfolgenden Versuchen einzeln betrachtet und beschrieben.

Alle Versuche werden auf der eingesetzten Hardware-Plattform VAR-SOM-MX8X durchgeführt und das zur Verfügung gestellte Board Support Package (BSP) für das Embedded-Linux (Kap. 7.2) auf dem A35 eingesetzt. Auf dem M4 werden für die Latenzzeit- und Durchsatzmessungen jeweils eine eigene Firmware eingesetzt, die während der Versuchsdurchführung über das LCM gestartet und gestoppt werden kann. Neben der beschriebenen Umgebung wurden keine Änderungen oder Optimierungen auf dem eingesetzten Linux-Betriebssystem auf dem A35 und FreeRTOS auf dem M4 durchgeführt.

8.2.3 Vorstellung der Versuche

Alle nachfolgenden Versuche beziehen sich auf einen definierten Messabschnitt innerhalb des dargestellten Systemüberblicks (Abb. 8.2).

Versuche zur Messung der Latenzzeiten

Die Messungen für die Latenzzeiten unterteilen sich in mehrere Messabschnitte (Abb. 8.2):

- IPI-Latenzzeit vom A35 (i.MX RPMsg Platform Driver) zur aufgerufenen ISR im M4 (Messabschnitt 3 bis 4, Abb. 8.2).
- IPI-Latenzzeit vom M4 zum A35 (i.MX RPMsg Platform Driver) zur aufgerufenen ISR (Messabschnitt 4 bis 3, Abb. 8.2).
- Latenzzeit von der MeasureMan-Anwendung im Userspace bis zum Auslösen des IPI 3 i.MX RPMsg Platform Driver über die TTY-Write Schnittstelle (Messabschnitt 1 bis 3, Abb. 8.2).

- Latenzzeit vom i.MX RPMsg Platform Driver bis zur MeasureMan-Anwendung über die TTY-Read Schnittstelle (Messabschnitt 3 bis 1, Abb. 8.2). Besonders wichtig ist hierbei zu erwähnen, dass bei dieser Messung die M4-Verarbeitungszeit und die IPI-Latenzzeit beinhaltet ist.
- Die durch den OpenAMP verursachte Verarbeitungszeit auf dem M4 (Messabschnitt 4, Abb. 8.2)

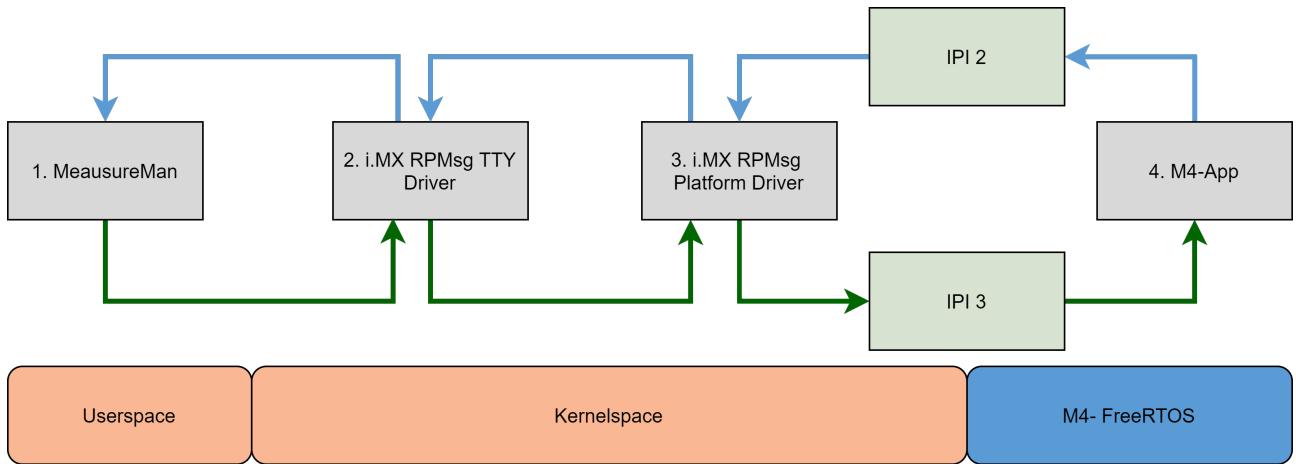


Abbildung 8.2: Systemüberblick zu den durchzuführenden Versuchen, welcher für die Versuche die betroffenen Komponenten darstellt. Er unterteilt sich in vier Komponenten, wovon der MeasureMan, der i.MX RPMsg TTY Driver und der i.MX RPMsg Platform Driver auf dem A35 ablaufen und die M4 Applikation auf dem M4 (grau). Beide Prozessoren kommunizieren über die zwei IPIs 2 und 3 miteinander (grün). Die grünen Pfeile stellen die Kommunikationsrichtung vom Linux, die sich in die Bereiche User- und Kernelspace (orange) unterteilen, bis zum FreeRTOS auf dem M4 (blau) dar. Die blauen Pfeile stellen die entgegengesetzte Kommunikationsrichtung dar.

Bei den Latenzzeitmessungen werden Zusatzfaktoren wie eine Systemlast nicht berücksichtigt, da das Ziel möglichst kurze Latenzzeiten sind und ein belastetes System die Latenzzeiten verschlechtern würde.

Die Ergebnisse der Latenzmessungen werden in Histogrammen dargestellt, um die Häufigkeitsverteilung der Latenzzeiten zu erkennen. Ein Histogramm stellt die absolute Häufigkeit von Intervallen dar (z.B. die Anzahl der Messungen zwischen $10 \mu s$ und $20 \mu s$). Diese Intervalle werden in der Statistik als Klassen bezeichnet. Durch die getätigten Testmessungen wurden die Messgenauigkeit, die Breite und die Anzahl der Klassen definiert. Für die Messgenauigkeit wird $1 \mu s$ verwendet, da die auftretenden Latenzzeiten bei den Testmessungen zwischen $3 \mu s$ und $1000 \mu s$ liegen und daher die definierte Messgenauigkeit ausreicht. Dies führt zwar zu einem Genauigkeitsverlust, bei denen die Messwerte zwischen $1 \mu s$ und $1000 \mu s$ liegen, allerdings hat diese Genauigkeit für die vorgestellten Anwendungsszenarien keine Relevanz. Für die Klassenbreite werden $5 \mu s$ gewählt, um einen hohen Genauigkeitsverlust zu vermeiden. Zwar führt diese Klassenbreite trotzdem zu einem minimalen Genauigkeitsverlust, der aufgrund den erwarteten Latenzzeiten im Mikro- und Millisekundenbereich vernachlässigbar ist und keine auch keine Relevanz für die vorgestellten Anwendungsszenarien hat. Für die Anzahl der Klassen werden alle auftretenden Messungen eingeschlossen, damit dem Histogramm, die maximal auftretende Latenzzeit entnommen werden kann.

Die Latenzzeitmessungen werden in Abhängigkeit der Nachrichtengröße und der Wiederholungsrate der übertragenen Nachrichten durchgeführt.

Versuche zur Messung des Datendurchsatzes

Bei der Messung des Datendurchsatzes wird gemessen, wie viele Nachrichtenpakete in welcher Zeit von der Userspace-Applikation MeasureMan bis zur M4-Applikation übertragen werden können (Messabschnitt 1 bis 4, Abb. 8.2). Außerdem wird der Datendurchsatz in die entgegengesetzte Richtung gemessen (Messabschnitt 4 bis 1, Abb. 8.2).

Die Datendurchsatzmessungen werden mit dem Zusatzfaktor der Systemlast durchgeführt, um die Stabilität des Datendurchsatzes unter erschwerten Bedingungen zu bewerten. Hierbei wird eine Systemlast durch die Auslastung des CPUs erzeugt. Neben der Auslastung des CPUs können zusätzliche Lasten auf I/O-Geräte, Speicher oder den Scheduler erzeugt werden. Allerdings fokussiert sich diese Arbeit auf die Betrachtung der Systemlast durch die Auslastung des CPUs. Hierbei wird der Datendurchsatz im Idle-Zustand mit einer Systemlast von 25%, 50% und 75% gemessen. Die Systemlast wird mit dem Linuxwerkzeug *stress-ng* erzeugt.

Die Ergebnisse werden in einem Weg-Zeit-Diagramm dargestellt, aus dem die Datenübertragungsgeschwindigkeit abgeleitet wird.

Alle Experimente werden mit der maximal zulässigen RPMsg-Nachrichtengröße von 496 Bytes durchgeführt, damit die Anzahl der Interrupts minimiert und dadurch der Datendurchsatz maximiert wird.

Versuche zur Messung des Lifecycle-Managements

Für die Bewertung des LCMs werden die vorgestellten Versuche für die Messung der Latenzzeiten und des Datendurchsatzes in zwei separate Firmwares implementiert, die zur Laufzeit für die Ausführung der Messung gestartet und gestoppt werden müssen (Kap. 8.2.2). Hierbei wird beobachtet, ob die wiederholende Initialisierung der IPC über ein RPMsg-Device stabil abläuft oder Fehler erzeugt werden (Abb. 7.5). Daher wird kein separater Versuch für die Bewertung des LCM durchgeführt sondern während der Ausführung der anderen Versuche das Verhalten beobachtet und in der anschließenden Diskussion die Ergebnisse dazu dargestellt (Kap. 8.4).

Überblick zu den Hauptversuchen

Die beschriebenen Versuche werden zu den folgenden Hauptversuchen zusammengefasst:

- **Hauptversuch 1:** Abhängigkeit der IPI-Latenzzeit zwischen A35 und M4 von der Nachrichtengröße
- **Hauptversuch 2:** Abhängigkeit der IPI-Latenzzeit zwischen A35 und M4 von der Wiederholungsrate
- **Hauptversuch 3:** Abhängigkeit der OpenAMP-Verarbeitungszeit auf dem M4 von der Nachrichtengröße
- **Hauptversuch 4:** Abhängigkeit der Latenzzeit zwischen TTY-Device und M4 von der Nachrichtengröße

- **Hauptversuch 5:** Abhängigkeit der Latenzzeit zwischen TTY-Device und M4 von der Wiederholungsrate
- **Hauptversuch 6:** Abhängigkeit des Datendurchsatzes zwischen A35 und M4 von der Systemlast

8.2.4 Messwerkzeuge und ihre Ungenauigkeiten

Die folgenden Teilabschnitte beschreiben die ausgewählten Methoden zur Messung der Latenzzeit und des Datendurchsatzes. Die Latenzzeitmessungen unterteilen sich in die Messung der IPI und der TTY-Latenzzeiten auf dem Linux-System. Abschließend folgt die Diskussion der auftretenden Ungenauigkeiten durch die ausgewählten Methoden.

IPI-Latenzzeitmessungen

Für die Zeitmessung der IPIs muss die vergangene Zeit zwischen Auslösung des Interrupts 3 auf Linux-Seite und der Ankunft in der ISR auf M4-Seite (Schritt 3 zu 4, Abb. 8.2) sowie in entgegengesetzter Richtung gemessen werden. Hierfür bietet sich der Einsatz von zwei General Purpose Input/Output (GPIO)s an, welche die Zeitmarken für das Auslösung und der Ankunft der IPIs repräsentieren. Ein anderer Lösungsansatz ist der Einsatz eines unabhängigen Timerbausteins auf dem MPSoC. Aufgrund der einfacheren und schnelleren Umsetzung wird der Ansatz über das Schalten von zwei GPIOs weiter verfolgt. Bevor in der Funktion *imx_rpmsg_notify()* der IPI 3 ausgelöst wird, wird der GPIO 1 auf High geschaltet (Abb. 7.3). Sobald die ISR auf M4-Seite aufgerufen wird, wird der GPIO 2 auf High geschaltet. Aus dieser zeitlichen Differenz kann die IPI-Latenzzeit vom A35 nach M4 bestimmt werden. Das gleiche Vorgehen wird für die Messung der IPI-Latenzzeit vom M4 zum A35 durchgeführt. Durch die Verwendung der GPIOs als Zeitmarken zwischen den beiden Prozessoren entstehen zusätzliche Latenzzeiten für das Schalten des GPIOs. Durch Testmessungen wurde verifiziert, dass die durchschnittliche Zeit zum Schalten des GPIOs auf A35-Seite in der ISR und auf M4-Seite unter $1 \mu\text{s}$ liegt (Anhang 10.2 Messungen, Vorversuch 1 und 2).

Damit der M4-Prozessor Zugriff auf den eingesetzten GPIO hat, muss über die SCFW-API die Ressource über den RemoteProc-Driver an den M4 weitergegeben werden.

Für die Messung der GPIOs, welche die IPI-Latenzzeiten repräsentieren, wurden folgende Messwerkzeuge in Betracht gezogen (Tab. 8.1). Saleae bietet eine Software an, welche die aufgenommenen Messdaten als CSV-Datei exportieren kann. Mit dem Oszilloskop von Agilent wird das Auslesen der aufgenommenen Messdaten über die Standard Commands for Programmable Instruments (SCPI)-Schnittstelle angeboten. Aufgrund der einfachen Bedienung und schnellen Exportierung von Messdaten, wird der Logic-Analyzer (LA) von Saleae eingesetzt. Die Messung von den zwei GPIOs erfolgt über zwei eingesetzten Kanäle, wodurch eine Ungenauigkeit von 80 ns erzeugt wird.

Messwerkzeug	Bandbreite [MHz]	Abtastrate [MS/s]	Genauigkeit [ns]
Oszilloskop MSO7034B	350	2000	9
Logic Pro 8	100	500	40

Tabelle 8.1: Auswahl der zur Verfügung gestellten Messwerkzeuge zur Messung der IPI-Latenzzeiten. Die Genauigkeit bezieht sich auf die maximale Genauigkeit, die bei der Verwendung von genau einem Kanal erzielt wird. Neben der Genauigkeit ist die Bandbreite und die Abtastrate der Messwerkzeuge beschrieben.

RPMsg-Latenzzeiten auf dem Linux-System

Für die Messung der RPMsg-Latenzzeit auf dem Linux-System wird die Ausführungszeit der Systemaufrufe *write* und *read* gemessen, die vom i.MX RPMsg TTY Driver zur Verfügung gestellt werden (Abb. 7.5). Um die Latenzzeiten der verwendeten Systemaufrufe zu messen, wird das Linux-Werkzeug *Ftrace* eingesetzt. *Ftrace* bietet Möglichkeiten für die Messung von Latenzzeiten innerhalb des Linux-Kernels an [73]. Dadurch kann die Latenzzeit der Systemaufrufe *write* und *read* gemessen werden. Die dabei auftretende Ungenauigkeit ist durch die eingesetzte Linux-Systemzeit begrenzt, die bei 125 ns liegt.

Eine weitere Ungenauigkeit stellt die eingesetzte Hardware-Plattform dar, die Verzögerungen aufweist, wenn über den Systembus auf den DDR-Off-Chip-Bereich zugegriffen wird. Dieser kann wiederum nur über den DDR-Controller erreicht werden, der ebenso eine Verzögerung erzeugt. Der DDR-Controller hat eine Taktrate von 800 MHz, der wiederum 10 Taktzyklen für einen Zugriff benötigt und sich somit auf 15 ns begrenzt.

Diskussion der Ungenauigkeiten bei Latenzzeitmessungen

In den vorhergehenden Abschnitten wurden die Ungenauigkeiten aufgezeigt, die durch die eingesetzte Methode entstehen. Sie werden in Tabelle 8.2 zusammengefasst.

Bereich	Messungenauigkeit
Kernel GPIO-Zugriff	1 μ s
M4 GPIO-Zugriff	0,1 μ s
Logic-Analyzer	0,080 μ s
Ftrace	0,125 μ s
DDR-Controller	0,015 μ s
Geschätzter Messfehler	1,32 μ s
Geschätzter aufgerundeter Messfehler	2 μ s

Tabelle 8.2: Überblick zu den Messungenauigkeiten.

Durch die aufgezeichneten Ungenauigkeiten wird der Messfehler auf 2 μ s geschätzt, was zu keiner signifikanten Messabweichung in einer der durchzuführenden Versuche führt (Tab. 8.2). Zwar führt der geschätzte Messfehler zu Abweichungen, die zu schlechteren Messergebnissen führen. Das bedeutet im Umkehrschluss, dass falls die gemessenen Werte ausreichend gut für die dargestellten Anwendungsszenarien sind, so sind auch die tatsächlichen Werte gut genug.

Datendurchsatzmessungen

Die Durchsatzmessung zwischen den beiden Prozessoren erfolgt durch Zählen der empfangenen Pakete in einem bestimmten Zeitraum. So muss bei der Auswertung ausschließlich die Paketanzahl innerhalb des gemessenen Zeitraums entnommen werden. Der gemessene Zeitraum wird auf 60 Sekunden festgelegt, wodurch Ungenauigkeiten im Mikrosekundenbereich vernachlässigbar sind.

8.2.5 Messapplikationen

Für die Durchführung der definierten Versuche (Kap. 8.2.3) wird die bereits vorgestellte Applikation **MeasureMan** eingesetzt, welche für die Ausführung der Messungen verantwortlich ist und die passende Firmware auf dem M4 lädt und ausführt (Kap. 7.7).

Auf dem Host-System wird die Applikation **MeasureExporter** für die Auswertung der aufgenommenen Messungen eingesetzt, die daraus statistische Größen und Diagramme generiert.

8.3 Versuchsdurchführung und Ergebnisse

8.3.1 Hauptversuch 1 - Abhängigkeit der IPI-Latenzzeit zwischen A35 und M4 von der Nachrichtengröße

Beschreibung zu den Versuchen 1.1 und 1.2

Die beiden ersten Versuche 1.1 und 1.2 bewerten, wie sich die Interrupt-Latenzzeit zwischen A35 und M4 mit ansteigender Nachrichtengröße verändert. Daher stellt die Nachrichtengröße in beiden Versuchen den freien Parameter dar (Tab. 8.3). Die Wiederholungsrate ist maximal, was bedeutet, dass so viele Nachrichten wie möglich zwischen den Prozessoren ausgetauscht werden.

Erwartungen zu Versuch 1.1 und 1.2

Wenn die Nachrichtengröße erhöht wird, so wird als Ergebnis erwartet, dass die IPI-Latenzzeit konstant bleibt, da das Kopieren der Nachrichten in den Shared-Memory vor Auslösung des Interrupts durchgeführt wird und daher keine Auswirkungen auf die IPI-Latenzzeit haben sollte.

Versuchsaufbau zu Versuch 1.1 und 1.2

Versuch 1	IPI-Latenzzeit zwischen A35 nach M4
Freier Parameter	Nachrichtengröße
Wiederholungsrate [kHz]	Maximal
Anzahl der Stichproben	2.500.000
Messungen pro Experiment	1
Untersuchte Nachrichtengrößen [Byte]	1, 4, 100, 496 $\hat{=}$ 4 Experimente

Tabelle 8.3: Parameter zu Versuch 1: IPI-Latenzzeit von A35 zwischen M4 in Abhängigkeit von der Nachrichtengröße.

Ergebnisse zu Versuch 1.1

Versuch 1.1, Nachrichtengröße [Byte], A35 nach M4	1	4	100	496
Max. [μ s]	781	80	129	809
Min. [μ s]	3	3	3	3
Arith. Mittel [μ s]	3	3	4	4
Standardabweichung [μ s]	1	0	1	1
99 %-Quantil [μ s]	4	4	4	4
99,9 %-Quantil [μ s]	4	4	4	4

Tabelle 8.4: Ergebnisse zu Versuch 1.1: IPI-Latenzzeit von A35 nach M4 in Abhängigkeit von der Nachrichtengröße.

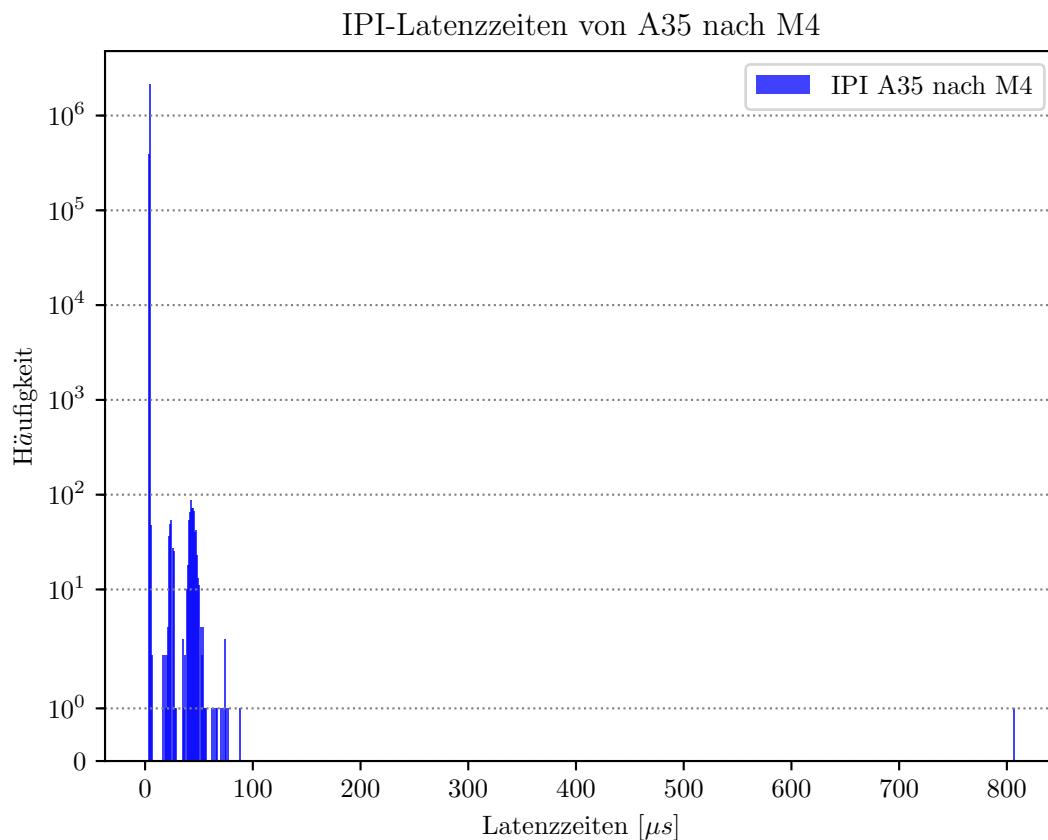


Abbildung 8.3: Histogramm zu Versuch 1.1: IPI-Latenzzeit von A35 nach M4 mit einer Nachrichtengröße von 496 Bytes. Auf der X-Achse sind die Klassen für die Latenzzeiten aufgetragen, wohingegen auf der Y-Achse die Häufigkeit der einzelnen Klassen logarithmisch aufgetragen ist. Diese Auftragung gilt für alle nachfolgenden Histogramme.

Auswertung zu Versuch 1.1

Die IPIs erzeugen eine durchschnittliche Latenz von 3 bis 4 μ s mit einer geringen Standardabweichung von 0 bis 1 μ s unabhängig von der eingesetzten Nachrichtengröße (Tab. 8.4). Daher kann die aufgestellte Erwartung bestätigt werden. Allerdings ist festzustellen, dass unabhängig von der Nachrichtengröße Latenzzeiten von 781 und 809 μ s auftreten, was zwei Größenordnungen

mehr sind (Abb. 8.3). Ebenso ist eine dünne Häufigkeit von Latenzzeiten zwischen 3 und 100 μs zu erkennen. Das 99,9 %-Quantil liegt bei allen Experimenten bei unter 4 μs .

Die Ursache hierfür könnte an dem Parameter der Wiederholungsrate liegen, der in diesem Versuch maximiert wurde. Dadurch werden übertragene Nachrichten vom M4 sofort wieder an den A35 zurückgeschickt, der im Anschluss direkt wieder eine neue Nachricht an den M4 überträgt. Dieser Zusammenhang zwischen gesendeten Nachrichten in Abhängigkeit der Wiederholungsrate wird noch genauer untersucht (Kap. 8.3.2).

Ergebnisse zu Versuch 1.2

Versuch 1.2, Nachrichtengröße [Byte], M4 nach A35	1	4	100	496
Max. [μs]	6413	1155	668	581
Min. [μs]	3	3	4	4
Arith. Mittel [μs]	4	4	5	5
Standardabweichung [μs]	8	7	2	2
99 %-Quantil [μs]	5	5	8	10
99,9 %-Quantil [μs]	9	9	10	11

Tabelle 8.5: Ergebnisse zu Versuch 1.2: IPI-Latenzzeit von M4 nach A35 in Abhängigkeit von der Nachrichtengröße.

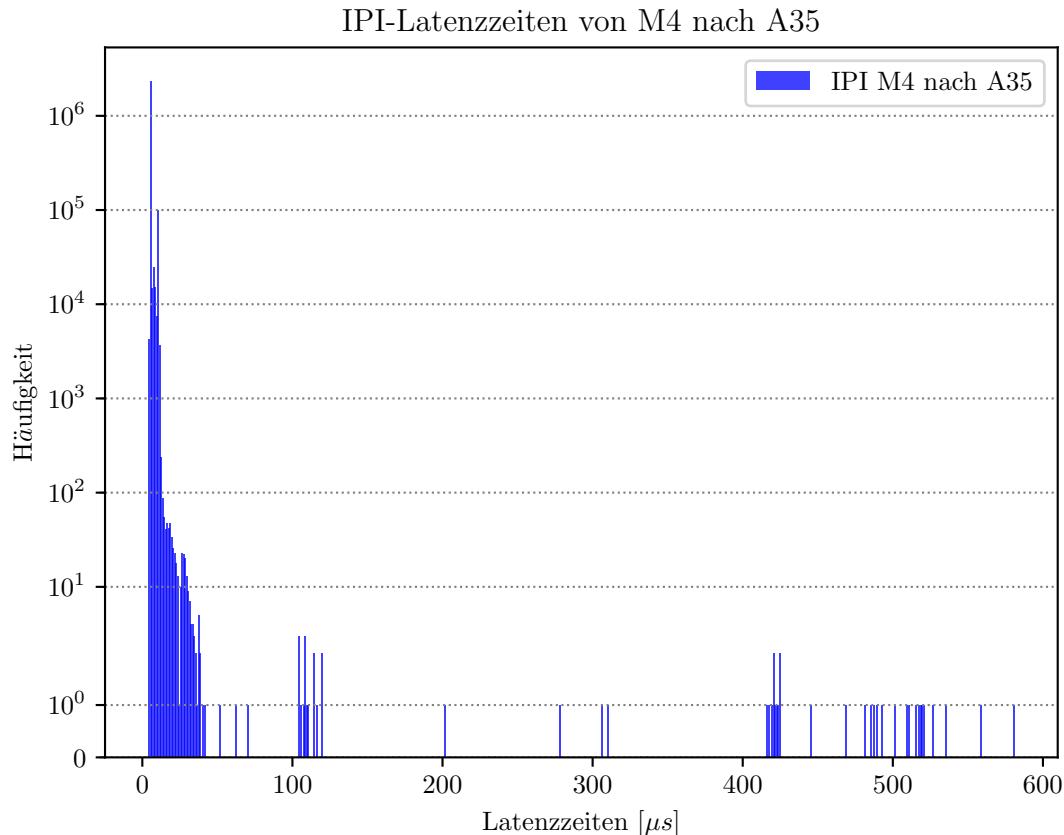


Abbildung 8.4: Histogramm zu Versuch 1.2: IPI-Latenzzeit von M4 nach A35 mit einer Nachrichtengröße von 496 Bytes.

Auswertung zu Versuch 1.2

Die IPIs erzeugen wieder eine durchschnittliche Latenz von 3 bis 4 μs mit einer höheren Standardabweichung von 2 bis 7 μs unabhängig von der eingesetzten Nachrichtengröße (Tab. 8.5), wodurch die Erwartung wieder füllt wird. Allerdings sind diesmal die Ausreißer eine weitere Größenordnung höher, wovon der maximale bei 6413 μs liegt. Hervorzuheben ist der maximal auftretende Ausreißer von 6413 μs und die maximale Standardabweichung von 8 μs bei der

minimalen Nachrichtengröße von 1 Byte. Das 99,9 %-Quantil liegt bei allen Experimenten bei unter 11 μs .

8.3.2 Hauptversuch 2 - Abhängigkeit der IPI-Latenzzeit zwischen A35 und M4 von der Wiederholungsrate

Beschreibung zu den Versuchen 2.1 und 2.2

Die Versuche 2.1 und 2.2 prüfen die Abhängig der IPI-Latenzzeit von der eingesetzten Wiederholungsrate. Somit stellt die Wiederholungsrate den freien Parameter für die Versuche dar (Tab. 8.6). Alle nachfolgenden Experimente werden mit einer Nachrichtengröße von 496 Bytes durchgeführt.

Versuchsaufbau zu Versuch 2.1 und 2.2

Versuch 2	IPI-Latenzzeit zwischen A35 nach M4
Freier Parameter	Wiederholungsrate
Nachrichtengröße [Byte]	496
Anzahl der Stichproben	100.000
Messungen pro Experiment	1
Untersuchte Wiederholungsrate [kHz]	0.1, 1, 10 $\hat{=}$ 3 Experimente

Tabelle 8.6: Parameter zu Versuch 2: IPI-Latenzzeit zwischen A35 und M4 in Abhängigkeit von der Wiederholungsrate.

Ergebnisse zu Versuch 2.1

Versuch 2.1, Wiederholungsrate [kHz], A35 nach M4	0.1	1	10
Max. [μs]	50	54	715
Min. [μs]	3	3	3
Arith. Mittel [μs]	4	4	4
Standardabweichung [μs]	1	1	2
99 %-Quantil [μs]	4	4	4
99,9 %-Quantil [μs]	4	4	4

Tabelle 8.7: Ergebnisse zu Versuch 2.1: IPI-Latenzzeit von A35 nach M4 in Abhängigkeit von der Wiederholungsrate.

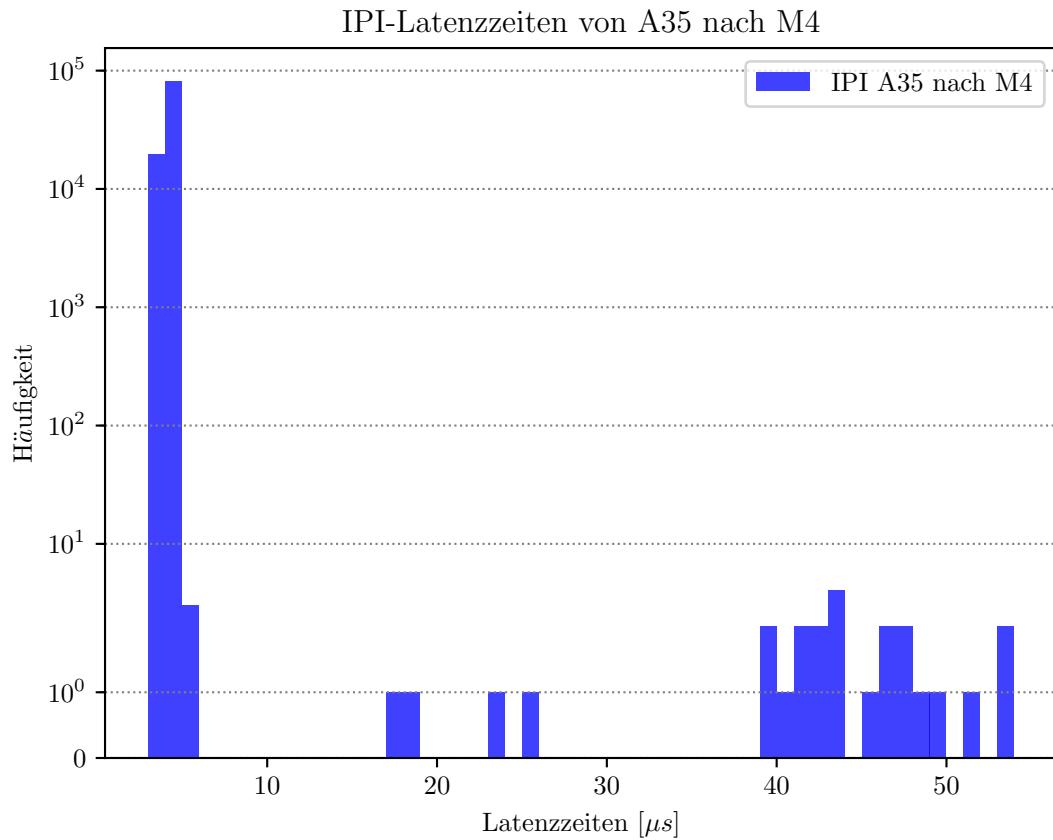


Abbildung 8.5: Histogramm zu Versuch 2.1: IPI-Latzenzeit von A35 nach M4 mit einer Wiederholungsrate von 1 kHz.

Auswertung zu Versuch 2.1

Die IPIs erzeugen eine durchschnittliche Latenz von $4 \mu\text{s}$ mit einer geringen Standardabweichung von 1 bis $2 \mu\text{s}$ unabhängig von der verwendeten Wiederholungsrate. (Tab. 8.7). Das 99,9 %-Quantil liegt bei allen durchgeföhrten Experimenten bei $4 \mu\text{s}$. Bei einer Wiederholungsrate von 10 kHz tritt die maximale Latenzzeit von $715 \mu\text{s}$ auf. Unabhängig von der Wiederholungsrate treten allerdings mehrere Ausreißer zwischen 3 und $54 \mu\text{s}$ auf (Abb. 8.5).

Ergebnisse zu Versuch 2.2

Versuch 2.2, Wiederholungsrate [kHz], M4 nach A35	0.1	1	10
Max. [μs]	553	543	525
Min. [μs]	4	4	4
Arith. Mittel [μs]	5	5	5
Standardabweichung [μs]	7	3	3
99 %-Quantil [μs]	8	8	10
99,9 %-Quantil [μs]	11	12	11

Tabelle 8.8: Ergebnisse zu Versuch 2.2: IPI-Latzenzeit von M4 nach A35 in Abhängigkeit von der Wiederholungsrate.

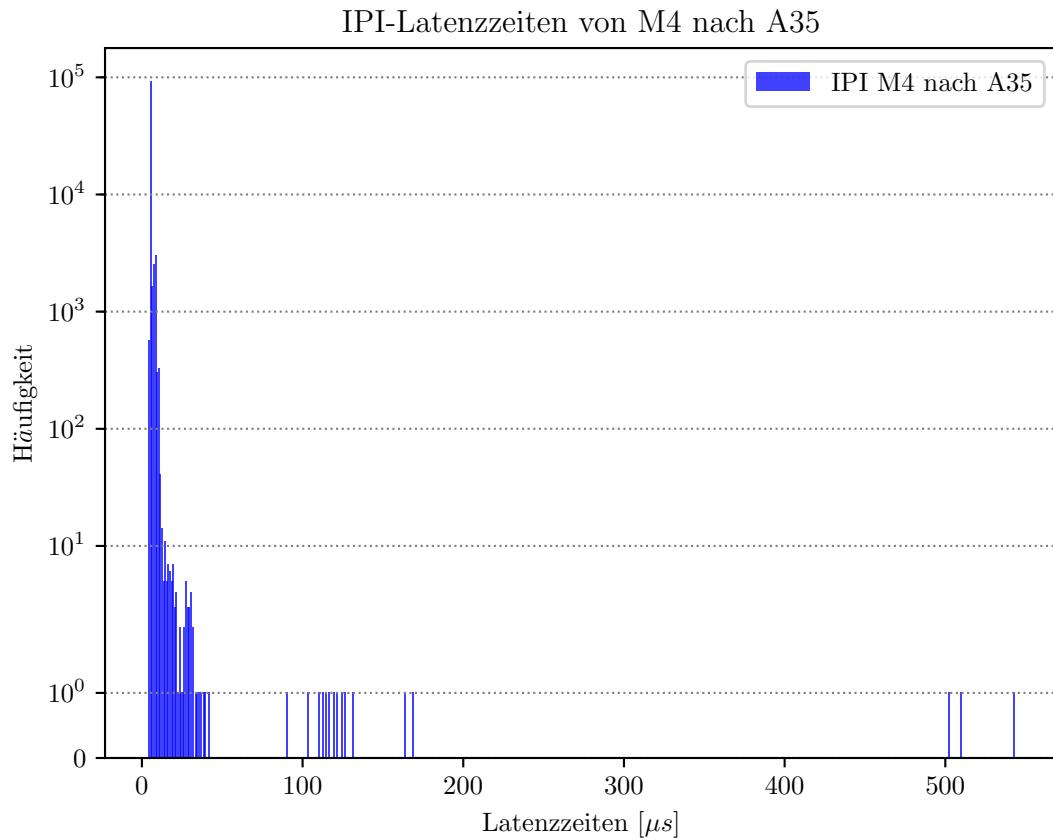


Abbildung 8.6: Histogramm zu den Versuch 2.2: IPI-Latenzzeit von M4 nach A35 mit einer Wiederholungsrate von 1 kHz.

Auswertung zu Versuch 2.2

Die IPIs erzeugen eine durchschnittliche Latenz von 5 μs mit einer geringen Standardabweichung von 3 bis 7 μs unabhängig von der verwendeten Wiederholungsrate (Tab. 8.7). Das 99,9 %-Quantil liegt bei allen durchgeföhrten Experimenten zwischen 11 bis 12 μs . Die maximalen Latenzzeiten liegen bei allen Experimenten zwischen 525 und 553 μs .

8.3.3 Hauptversuch 3 - Abhängigkeit der OpenAMP-Verarbeitungszeit von der Nachrichtengröße

Beschreibung zu Versuch 3

Der Versuch 3 prüft die durch OpenAMP entstehende Verarbeitungszeit auf dem M4 für die IPC zwischen A35 und M4 in Abhängigkeit von der Nachrichtengröße (Tab. 8.9).

Erwartungen zu Versuch 3

Mit zunehmender Nachrichtengröße erhöht sich die Verarbeitungszeit auf dem M4, da mehr Daten aus dem Shared-Memory-Bereich in den M4 On-Chip kopiert werden und dadurch mehr Taktzyklen auf dem MPSoC anfallen.

Versuchsaufbau zu Versuch 3

Versuch 3	OpenAMP Verarbeitungszeit
Freier Parameter	Nachrichtengröße
Wiederholungsrate [kHz]	Maximal
Anzahl der Stichproben	2.500.000
Messungen pro Experiment	1
Untersuchte Nachrichtengröße [Byte]	1, 4, 100, 496 $\hat{=} 4$ Experimente

Tabelle 8.9: Parameter zu Versuch 3: OpenAMP Verarbeitungszeit auf dem M4.

Ergebnisse zu Versuch 3

Versuch 3, Nachrichtengröße [Byte]	1	4	100	496
Max. [μ s]	26	29	103	365
Min. [μ s]	18	19	52	190
Arith. Mittel [μ s]	23	23	57	198
Standardabweichung [μ s]	1	1	1	1
99 %-Quantil [μ s]	23	24	59	199
99,9 %-Quantil [μ s]	24	25	59	200

Tabelle 8.10: Ergebnisse zu Versuch 3: OpenAMP Verarbeitungszeit auf dem M4 in Abhängigkeit von der Nachrichtengröße.

Auswertung zu Versuch 3

Mit zunehmender Nachrichtengröße steigt die Verarbeitungszeit an, was die aufgestellte Erwartung bestätigt. Die minimale Verarbeitungszeit bei einem Byte liegt bei 18 und die maximale bei 26 μ s. Die Verarbeitungszeiten bei der maximalen Nachrichtengröße von 496 Byte sind im Vergleich deutlich höher, welche zwischen 190 und 365 μ s liegen. Der Mittelwert liegt bei 198 μ s, die Standardabweichung bei 1 μ s und das 99,9 %-Quantil bei 200 μ s. Auffällig ist die große zeitliche Differenz zwischen der minimalen und maximalen Verarbeitungszeit, die 175 μ s beträgt. Die Ursache dafür kann möglicherweise auf den Systembus des MPSoCs und den Zugang auf dem Shared-Memory über den DDR-Controller zurückgeführt werden.

8.3.4 Hauptversuch 4 - Abhängigkeit der Latenzzeit des TTY-Devices von der Nachrichtengröße

Beschreibung zu den Versuchen 4.1 und 4.2

Im Versuch 4 wird die Latenzzeit vom TTY-Device in Abhängigkeit der Nachrichtengröße gemessen. Zusätzlich wird die Systemauslastung während den Messungen aufgezeichnet, um einen Zusammenhang zwischen Systemauslastung und steigender Nachrichtengröße darzustellen. Die Parameter sind Tabelle 8.11 zu entnehmen.

Erwartungen zu Versuch 4.1 und 4.2

Während bei den Hauptversuchen 1 und 2 die aktive Verarbeitungszeit im Linux minimal ist und sich ausschließlich auf Abschnitt 3 begrenzt (Abb. 8.2), ist in den folgenden Versuchen die Verarbeitungszeit innerhalb des Linux deutlich länger, wodurch deutlich höhere Ausreißer zu erwarten sind.

Versuchsaufbau zu Versuch 4.1 und 4.2

Versuch 4.1 und 4.2	Latenzzeit des TTY-Devices
Freier Parameter	Nachrichtengröße
Wiederholungsrate [kHz]	Maximal
Anzahl der Stichproben	2.500.000
Messungen pro Experiment	3
Untersuchte Nachrichtengrößen [Byte]	1, 4, 100, 496 $\hat{=}$ 4 Experimente

Tabelle 8.11: Parameter zu Versuch 4.1 und 4.2: Latenzzeit des TTY-Devices.

Probleme bei der Durchführung von Versuch 4.1 und 4.2

Bereits bei der wiederholten Ausführung der Messungen bei der gleichen Nachrichtengröße sind unterschiedliche Mittelwerte und Standardabweichungen festzustellen, die bei einer Stichprobenanzahl von 100.000 Messungen nicht reproduzierbar sind. Aus diesem Grund wird die Stichprobenanzahl auf 2.500.000 Messungen erhöht, was zu Messausführungsduer von bis zu 120 Minuten führt. Außerdem werden 3 Messungen pro Experiment für jede Nachrichtengröße durchgeführt, um repräsentative Ergebnisse zu erhalten und eine bessere Einschätzung für die Latenzzeiten zu interpretieren (Tab. 8.11). In den nachfolgenden Histogrammen wird die Klassenanzahl bis zu $2000 \mu\text{s}$ abgebildet, da sonst die hohen Ausreißer zu einer sehr feinen Auflösung der einzelnen Klassen führt und dadurch die Streuung um den Mittelwert nicht deutlich erkennbar ist.

Ergebnisse zu Versuch 4.1

Versuch 4.1, TTY-Write	1	2	3
1 Byte			
Max. [μs]	6078	11959	1468
Min. [μs]	7	7	7
Arith. Mittel [μs]	9	9	9
Standardabweichung [μs]	11	10	7
99 %-Quantil [μs]	11	11	11
99,9 %-Quantil [μs]	48	47	44
Systemauslastung [%]	52	51	52
4 Byte			
Max. [μs]	1486	755	1548
Min. [μs]	7	7	7
Arith. Mittel [μs]	9	9	9
Standardabweichung [μs]	3	3	4
99 %-Quantil [μs]	11	11	11
99,9 %-Quantil [μs]	43	44	44
Systemauslastung [%]	51	51	50
100 Byte			
Max. [μs]	1265	11682	125396
Min. [μs]	7	7	7
Arith. Mittel [μs]	10	10	10
Standardabweichung [μs]	10	17	67
99 %-Quantil [μs]	12	12	12
99,9 %-Quantil [μs]	136	158	121
Systemauslastung [%]	39	40	40
496 Byte			
Max. [μs]	1707	13878	1278
Min. [μs]	7	7	7
Arith. Mittel [μs]	12	12	12
Standardabweichung [μs]	5	10	6
99 %-Quantil [μs]	14	14	14
99,9 %-Quantil [μs]	61	56	60
Systemauslastung [%]	21	20	19

Tabelle 8.12: Ergebnisse zu Versuch 4.1: Latenzzeit von TTY-Write in Abhängigkeit von der Nachrichtengröße.

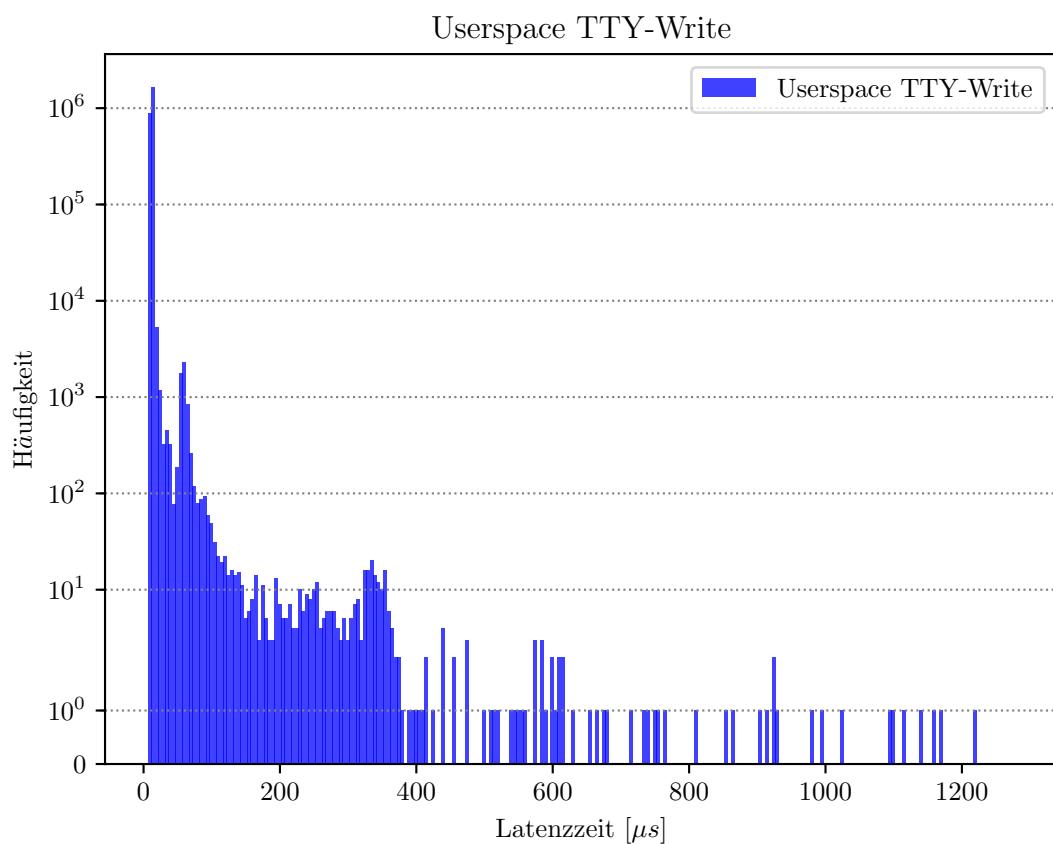


Abbildung 8.7: Histogramm zu Versuch 4.1, Messung 3: Latenzzeit von TTY-Write mit einer Nachrichtengröße von 496 Bytes.

Auswertung zu Versuch 4.1

Der Mittelwert von der TTY-Write Latenzzeit liegt unabhängig von der Nachrichtengröße zwischen 9 und 12 μs . Ebenso treten unabhängig von der Nachrichtengröße Ausreißer von bis zu 125396 μs bei 100 Bytes auf, der den hohen Ausreißer bei der Standardabweichung mit 67 μs bei der dritten Messung erzeugt. So können bei keiner Nachrichtengröße die Ausreißer unterbunden werden, was die aufgestellte Erwartung untermauert. Die Systemauslastung sinkt von 52 % auf bis zu 20 % bei zunehmender Nachrichtengröße. Durch die Erhöhung der Messungen auf 3 pro Nachrichtengröße und die Stichprobenanzahl auf 2.500.000 entstehen reproduzierbare Ergebnisse, was am fast nicht abweichenden Mittelwert und an einer niedrigen Standardabweichung erkennbar ist. Die Standardabweichung bei 496 Byte ist mit bis zu 10 μs gering und der Schwerpunkt aller Latenzzeiten liegt bei 12 μs (Abb. 8.8).

Ergebnisse zu Versuch 4.2

Versuch 4.2, TTY-Read	1	2	3
1 Byte			
Max. [μs]	19755	125085	50404
Min. [μs]	3	3	3
Arith. Mittel [μs]	54	53	53
Standardabweichung [μs]	61	97	41
99 %-Quantil [μs]	101	92	85
99,9 %-Quantil [μs]	848	786	456
Systemauslastung [%]	52	51	52
4 Byte			
Max. [μs]	15967	59730	18840
Min. [μs]	3	3	3
Arith. Mittel [μs]	54	53	53
Standardabweichung [μs]	20	47	23
99 %-Quantil [μs]	81	80	80
99,9 %-Quantil [μs]	110	114	115
Systemauslastung [%]	51	51	50
100 Byte			
Max. [μs]	16647	13001	125396
Min. [μs]	3	3	3
Arith. Mittel [μs]	93	96	97
Standardabweichung [μs]	59	63	98
99 %-Quantil [μs]	138	164	206
99,9 %-Quantil [μs]	729	891	900
Systemauslastung [%]	39	40	40
496 Byte			
Max. [μs]	21168	21563	125472
Min. [μs]	4	4	4
Arith. Mittel [μs]	248	248	254
Standardabweichung [μs]	45	50	126
99 %-Quantil [μs]	295	288	276
99,9 %-Quantil [μs]	861	787	1222
Systemauslastung [%]	21	20	19

Tabelle 8.13: Ergebnisse zu Versuch 4.2: Latenzzeit von TTY-Read in Abhängigkeit von der Nachrichtengröße.

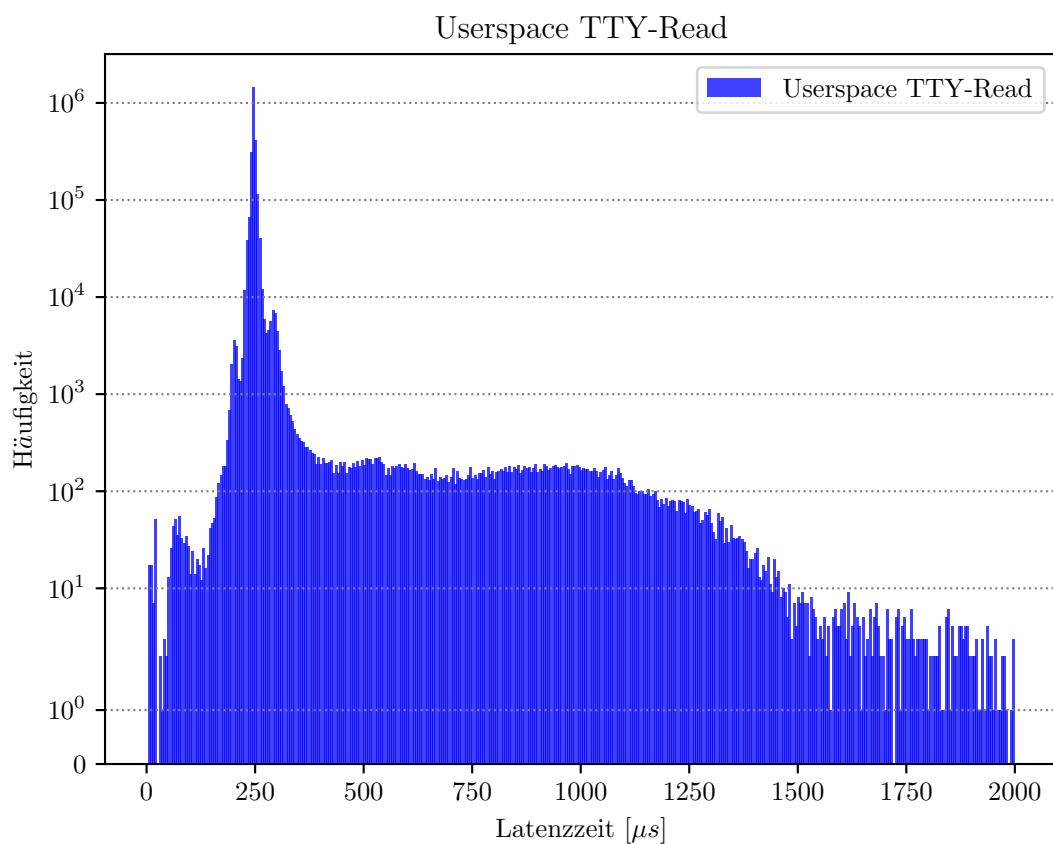


Abbildung 8.8: Histogramm zu Versuch 4.2, Messung 3: Latenzzeit von TTY-Read mit einer Nachrichtengröße von 496 Bytes.

Auswertung zu Versuch 4.2

Der Mittelwert von der TTY-Read Latenzzeit steigt mit zunehmender Nachrichtengröße von 53 bis auf 254 μs bei 496 Byte an. Wieder entstehen unabhängig von der Nachrichtengröße bis zu 125396 μs bei 100 Bytes. Besonders auffällig bei allen Messungen ist die minimale Latenzzeit bei 3 und 4 μs . Sie ist zwar nur im niedrigen zweistelligen Bereich vertreten (Abb. 8.8, 496 Bytes, was dennoch vernachlässigbar ist. Die Ursache für die einzelnen sehr niedrigen Latenzzeiten ist unbekannt. Bei der Nachrichtengröße von 1 Byte beim 99,9 %-Quantil entsteht eine starke Schwankung zwischen 456 und 848 μs und bei 496 Byte zwischen 787 und 1222 μs (Tab. 8.13). Der Schwerpunkt aller Latenzzeiten bei 496 Bytes liegt bei 254 μs und lässt stark in beide Richtungen nach (Abb. 8.8).

Es ist anzumerken, dass die TTY-Read Latenzzeiten die IPI Latenzzeiten und die OpenAMP-Verarbeitungszeiten enthält (Kap. 8.2.3). Aufgrund der Aufnahme der Messdaten zu unterschiedlichen Zeitpunkten zwischen den Hauptversuchen können die Messdaten nicht miteinander verknüpft werden. Um dennoch eine Einschätzung für die tatsächliche TTY-Read Latenzzeit zu erhalten, wird in Tab. 8.14 das arithmetische Mittel abzüglich der Latenzzeiten aus Tab. 8.4, Tab. 8.5 und Tab. 8.10 dargestellt.

Nachrichtengröße [Byte]	Arith. Mittel der tatsächlichen TTY-Read [μs]
1	23
4	24
100	31
496	47

Tabelle 8.14: Ergebnisse zu Versuch 4.3: Tatsächliche Latenzzeit von TTY-Read.

Durch die tatsächliche TTY-Read Latenzzeit ist eine sehr niedrige Zeit bis zu 47 μs festzustellen (Tab. 8.14). Dennoch treten Ausreißer im Millisekundenbereich auf. Um die Ausreißer zu reduzieren, wird in den folgenden Versuchen die Messungen mit dem Real Time (RT)-Patch durchgeführt.

Versuchsaufbau zu Versuch 4.3 und 4.4

Für die Versuche 4.3 und 4.4 wird der RT-Patch in den bisher eingesetzten Kernel integriert, um damit Verbesserungen zu erzielen. Durch den RT-Patch wird der Kernel an deutlich mehr Stellen unterbrechbar, wodurch geringere Latenzzeiten ermöglicht werden, weil schneller auf eingehende Ereignisse wie eine ISR reagiert werden kann. Dadurch kann das Verhalten des Kernels genauer vorhergesagt werden. Der eingesetzte RT-Patch hat die Version *patch-4.14.103-rt55.patch*. Zusätzlich wird die ausgeführte Messung über den MeasureMan mit der Echtzeitpriorität von 99 gestartet, um andere Tasks unterbrechen und verdrängen zu können.

Die folgenden Versuche untersuchen ausschließlich die Latenzzeiten mit einer Nachrichtengröße von 496 Bytes mit einer Anzahl von 3 Messungen pro Experiment (Tab. 8.15).

Ergebnisse zu Versuch 4.3

Versuch 4.3 und 4.4	Latenzzeit des TTY-Devices
Freier Parameter	-
Wiederholungsrate [kHz]	Maximal
Anzahl der Stichproben	2.500.000
Messungen pro Experiment	3
Untersuchte Nachrichtengrößen [Byte]	496 $\hat{=}$ 1 Experiment

Tabelle 8.15: Parameter zu Versuch 4.3 und 4.4: Latenzzeit des TTY-Devices.

Versuch 4.3, TTY-Write	1	2	3
496 Byte			
Max. [μ s]	628	624	623
Min. [μ s]	10	10	10
Arith. Mittel [μ s]	15	15	15
Standardabweichung [μ s]	5	5	5
99 %-Quantil [μ s]	19	18	19
99,9 %-Quantil [μ s]	50	50	50
Systemauslastung [%]	20	20	21

Tabelle 8.16: Ergebnisse zu Versuch 4.3: Latenzzeit von TTY-Write.

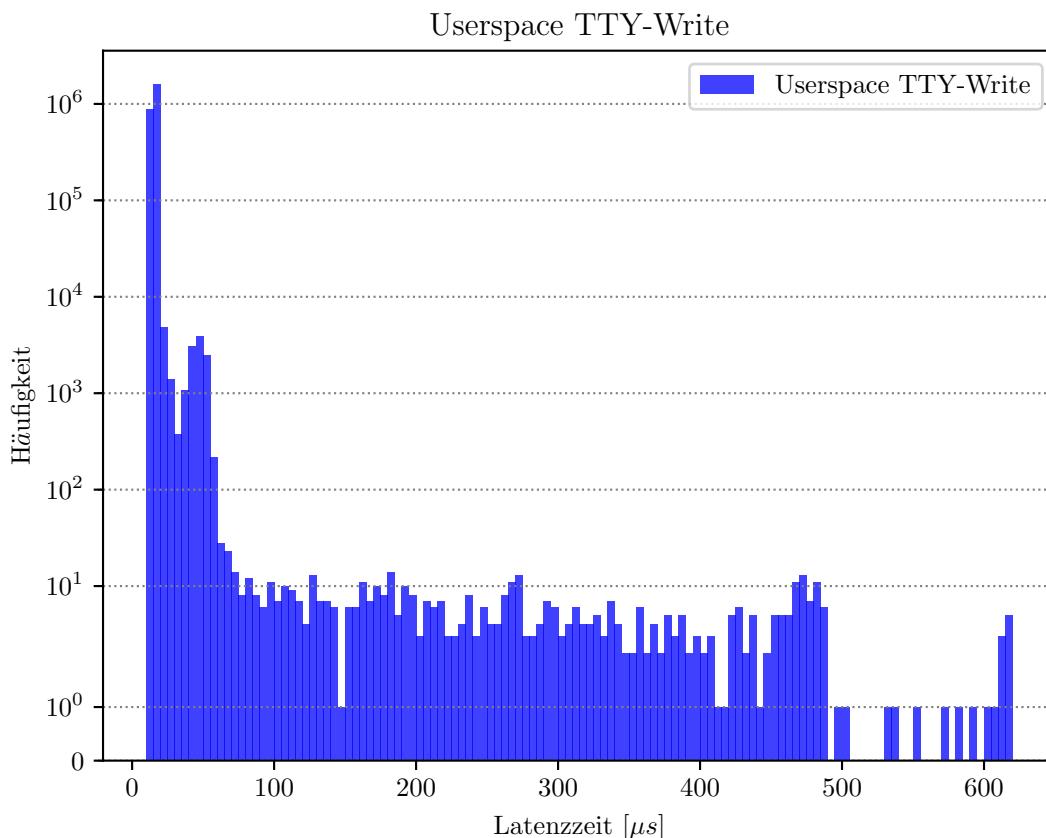


Abbildung 8.9: Histogramm zu Versuch 4.3, Messung 3: Latenzzeit von TTY-Read mit einer Nachrichtengröße von 496 Bytes.

Auswertung zu Versuch 4.3

Im Vergleich zu Versuch 4.1 (Tab. 8.13) haben sich die Ausreißer deutlich reduziert und liegen nun maximal bei 628 anstatt bei 13878 μs (Tab. 8.15). Auch an den Werten wie dem arithmetischen Mittel, der Standardabweichung und dem 99,9 %-Quantil ist eine positive Tendenz zwischen den einzelnen Messungen zu erkennen. So liegt das 99,9 %-Quantil immer bei 50 μs und die Standardabweichung bei 5 μs . Im Vergleich zu der Abbildung 8.7 sind in Abbildung 8.9 mehr Zeiten bis zu 500 μs vertreten. Der Versuch zeigt deutlich eine Verbesserung der Latenzzeiten durch den Einsatz des RT-Patches.

Ergebnisse zu Versuch 4.4

Versuch 4.4, TTY-Read	1	2	3
496 Byte			
Max. [μs]	55291	55194	45028
Min. [μs]	4	4	3
Arith. Mittel [μs]	278	278	277
Standardabweichung [μs]	108	105	107
99 %-Quantil [μs]	744	511	428
99,9 %-Quantil [μs]	947	942	949
Systemauslastung [%]	20	20	21

Tabelle 8.17: Ergebnisse zu Versuch 4.4: Latenzzeit von TTY-Read.

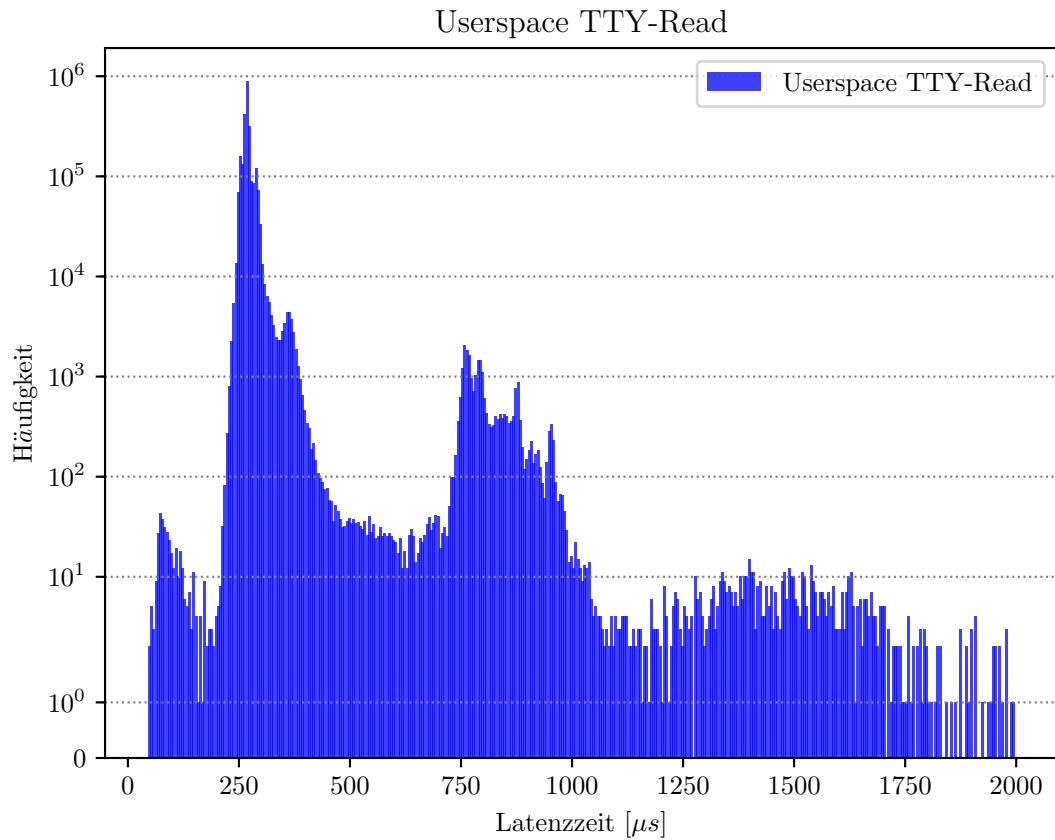


Abbildung 8.10: Histogramm zu Versuch 4.4, Messung 3: Latenzzeit von TTY-Read mit einer Nachrichtengröße von 496 Bytes.

Auswertung zu Versuch 4.4

Im Vergleich zu Versuch 3.2 reduzieren sich in diesem Versuch wieder die Ausreißer von 125472 auf 55194 μ s (Tab. 8.17). Die Standardabweichung und das 99,9 %-Quantil sind annähernd über die 3 Experimente konstant. Der arithmetische Mittelwert liegt bei 277 μ s, was auch in Abbildung 8.10 erkennbar ist. Auch in diesem Versuch ist deutlich eine Verbesserung durch den Einsatz des RT-Patches zu erkennen. Allerdings fällt auch auf, dass immer noch hohe Ausreißer vorhanden sind, was darauf hindeutet, dass für niedrigere Latenzzeiten Optimierungen im Linux-Kernel notwendig sind.

8.3.5 Hauptversuch 5 - Abhängigkeit der Latenzzeit des TTY-Devices von der Wiederholungsrate

Beschreibung zu den Versuchen 5.1 und 5.2

Aufgrund der besseren Ergebnisse mit dem RT-Patch wird die Wiederholungsrate des TTY-Devices ausschließlich mit dem RT-Patch und einer Nachrichtengröße von 496 Bytes ausgeführt (Tab. 8.18).

Versuchsaufbau

Versuch 5.1 und 5.2	Latenzzeit des TTY-Devices
Freier Parameter	Wiederholungsrate
Nachrichtengröße [Byte]	496
Anzahl der Stichproben	100.000
Messungen pro Experiment	1
Untersuchte Wiederholungsrate [kHz]	0.1, 1, 10 $\hat{=}$ 3 Experimente

Tabelle 8.18: Parameter zu Versuch 5.1: Latenzzeit des TTY-Devices.

Ergebnisse zu Versuch 5.1

Versuch 5.1, Wiederholungsrate [kHz], TTY-Write	0.1	1	10
Max. [μ s]	560	91	73
Min. [μ s]	13	11	11
Arith. Mittel [μ s]	17	16	16
Standardabweichung [μ s]	4	3	2
99 %-Quantil [μ s]	26	20	19
99,9 %-Quantil [μ s]	50	50	47

Tabelle 8.19: Ergebnisse zu Versuch 5.1: Latenzzeit von TTY-Write in Abhängigkeit von der Wiederholungsrate.

Auswertung zu Versuch 5.1

Die durchschnittlichen TTY-Write Latenzzeiten liegen unabhängig von der Wiederholungsrate bei bis zu 17μ s mit einer niedrigen Standardabweichung von bis 4μ s. Der größte Ausreißer liegt bei 560μ s bei einer Wiederholungsrate von 0,1 kHz.

Ergebnisse zu Versuch 5.2

Versuch 5.2, Wiederholungsrate [kHz], TTY-Read	0.1	1	10
Max. [μ s]	58914	56936	56420
Min. [μ s]	4	4	4
Arith. Mittel [μ s]	294	295	288
Standardabweichung [μ s]	210	189	188
99 %-Quantil [μ s]	366	356	363
99,9 %-Quantil [μ s]	862	394	802

Tabelle 8.20: Ergebnisse zu Versuch 5.2: Latenzzeit von TTY-Read in Abhängigkeit von der Wiederholungsrate.

Auswertung zu Versuch 5.2

Die durchschnittlichen TTY-Read Latenzzeiten liegen von 288 bis 295μ s mit einer hohen Standardabweichung von bis zu 210μ s. Im Vergleich zum vorherigen Versuch 4 (Tab. 8.17) ist die Standardabweichung deutlich gestiegen. Hohe Ausreißer im Mikrosekundenbereich sind weiterhin von bis zu 58914μ s vorhanden, unabhängig von der Wiederholungsrate.

8.3.6 Hauptversuch 6 - Abhängigkeit des Datendurchsatzes zwischen A35 und M4 von der Systemlast

Beschreibung zu den Versuchen 6.1 - 6.4

Der Hauptversuch 6 prüft den Datendurchsatz zwischen den beiden Prozessoren in Abhängigkeit der Systemlast. Bei den ersten beiden Versuchen 6.1 und 6.2 wird der Datendurchsatz vom User-space bis M4 und entgegengesetzt gemessen. Die letzten beiden Versuche 6.3 und 6.5 betrachten den Datendurchsatz zwischen Kernelspace und M4. Alle Experimente werden mit einer Nachrichtengröße von 496 Bytes durchgeführt (Tab. 8.21).

Versuchsaufbau zu den Versuchen 6.1 bis 6.4

Versuch 6	Datendurchsatz zwischen A35 und M4
Freier Parameter	Systemlast
Nachrichtengröße [Byte]	496
Messdauer [s]	60
Messungen pro Experiment	1
Untersuchte Systemlast [%]	Idle, 25, 50, 75 $\hat{=}$ 4 Experimente

Tabelle 8.21: Parameter zu Versuch 6: Datendurchsatz zwischen A35 und M4.

Ergebnisse zu den Versuchen 6.1 bis 6.4

Versuch 6, Systemlast [%]	Idle	25	50	75
Versuch 6.1 Userspace nach M4				
Anzahl der Pakete	510957	507072	494953	483884
Geschwindigkeit [MB/s]	4.03	3.99	3.90	3.81
Versuch 6.2 Kernelspace nach M4				
Anzahl der Pakete	517835	507944	509249	508564
Geschwindigkeit [MB/s]	4.08	4.00	4.01	4.01
Versuch 6.3 M4 nach Userspace				
Anzahl der Pakete	1542470	1423675	1354205	1319260
Geschwindigkeit [MB/s]	12.16	11.21	10.66	10.39
Versuch 6.4 M4 nach Kernelspace				
Anzahl der Pakete	1541871	1538116	1535455	1530595
Geschwindigkeit [MB/s]	12.16	12.13	12.11	12.07

Tabelle 8.22: Ergebnisse zu den Versuchen 6.1 bis 6.4: Datendurchsätze zwischen Userspace sowie Kernelspace und M4 in Abhängigkeit von der Systemlast. Pro Versuch wird die Anzahl der übertragenen Pakete und die daraus abgeleitete Geschwindigkeit beschrieben.

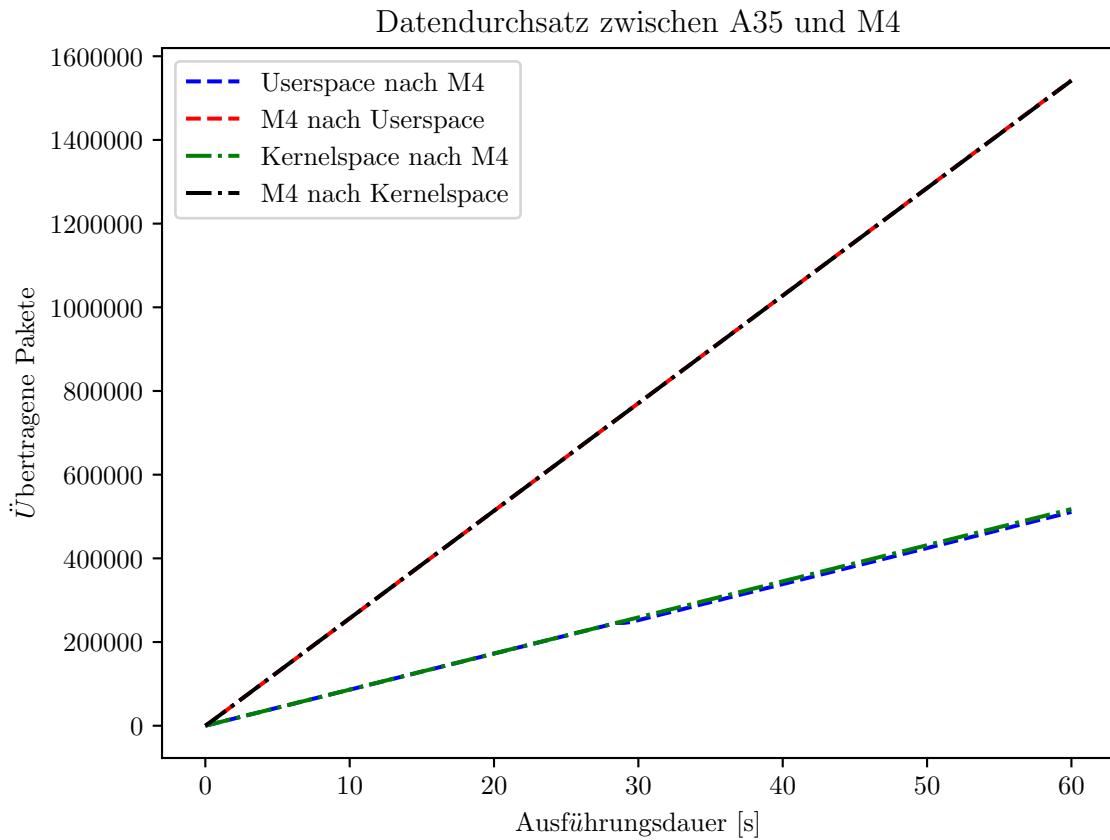


Abbildung 8.11: Versuch 5.1: Durchsatz zwischen Userspace und M4 mit der Systemlast Idle.

Auswertung zu den Versuchen 6.1 bis 6.4

Der Datendurchsatz zwischen beiden Prozessoren steigt unabhängig davon, ob vom Userspace oder vom Kernelspace die Messung ausgeführt wird, linear an (Abb. 8.11). Hervorzuheben ist der deutlich höhere Datendurchsatz in der Richtung von M4 zum A35, der bei 12,16 MB/s liegt (rot und schwarz, Abb. 8.11). Der niedrigere Datendurchsatz in Richtung zum M4 liegt bei 4,03 vom Userspace aus und bei 4,08 MB/s vom Kernelspace aus (blau und grün, Abb. 8.11). Sobald eine Systemlast anliegt, ist eine Reduzierung beim Datendurchsatz von bis zu 1,77 MB/s (Versuch 6.3) zu erkennen. Im Vergleich zwischen User- und Kernelspace reduziert sich der Datendurchsatz gegenüber dem Kernelspace schneller, sobald eine Systemlast anliegt (Tab. 8.22). Bereits ohne Systemlast ist eine Reduzierung des Datendurchsatzes von Userspace in Richtung M4 zu erkennen, die an der Anzahl der übertragenen Pakete und an der kurzen Reduzierung der übertragenen Pakete nach einer Ausführungszeit von 30 Sekunden (Abb. 8.11) zu erkennen sind (Versuch 6.1: 510957 Pakete, Versuch 6.2: 517835 Pakete). Die Ursache dafür kann der Linux-Scheduler sein, der für eine kurze Zeit den Userspace-Task blockiert, wodurch keine Pakete in diesem Zeitraum übertragen werden können.

8.4 Diskussion der Messergebnisse

8.4.1 Hauptversuche 1 und 2

Unabhängig von der Wiederholungsrate oder Nachrichtengröße treten Ausreißer bei den IPIs zwischen den Prozessoren A35 und M4 auf, die bis in den einstelligen Millisekundenbereich mit $6413 \mu\text{s}$ übergehen. Die Ursache kann dafür nicht durch den M4 verursacht werden, da dieser eine maximale Interrupt-Latenzzeit von 12 Taktzyklen zulässt [74]. Durch die mehrfach ausgeführten Messungen sind die auftretenden Ausreißer reproduzierbar, wodurch ein Messfehler bei der Messmethodik ausgeschlossen werden kann. Daher kann daraus abgeleitet werden, dass die Ausreißer hauptsächlich durch das Linux erzeugt werden. Die Ursache ist der Scheduler im Linux-Kernel, der nicht unterbrechbare Abschnitte hat. So kann während der Nachrichtenübertragung eine andere höhere priorisierte Aufgabe im Kernel abgearbeitet werden wie eine andere höher priorisierte ISR oder ein höhere priorisierter Task, die nicht unterbrechbar sind. Erst wenn ein passender Zeitslot für das betroffene Kernelmodul (i.MX RPMsg Platform Driver) frei ist, kann der IPI ausgelöst werden oder die ISR abgearbeitet werden. Dadurch können die auftretenden hohen IPI-Latenzzeiten erzeugt werden.

8.4.2 Hauptversuch 3

Durch eine zunehmende Nachrichtengröße wurden längere Verarbeitungszeiten durch OpenAMP auf dem M4 festgestellt, die bei einer Nachrichtengröße von 496 Bytes durchschnittlich bei $198 \mu\text{s}$ liegen. Allerdings liegt die maximale Verarbeitungszeit bei $365 \mu\text{s}$. Dies kann durch das Kopieren der Nachricht aus dem VirtIO-Buffer, der sich im Shared-Memory Bereich befindet, in den Applikations-Buffer mit verursacht werden. Um diese hohen Verarbeitungszeiten zu minimieren, würde es sich anbieten, RPMsgLite einzusetzen, das einen Zero-Copy Mechanismus zur Verfügung stellt, wodurch kein Kopiervorgang zwischen den VirtIO-Buffer und Applikations-Buffer stattfindet [49]. Eine andere Möglichkeit ist der Einsatz des OCRAM-Speicherbereichs als Shared-Memory. Dadurch wird der Zugriff auf den externen Speicher vermieden. Allerdings ist der OCRAM-Speicherbereich auf 256 KB begrenzt, wodurch einige Anpassungen in der RPMsg-Implementierung notwendig wären, da RPMsg bisher mit einem größeren Shared-Memory Bereich eingesetzt wird (Tab. 7.4). Eine weiterer Lösungsansatz ist der Einsatz einer Direct Memory Access (DMA), die den Kopiervorgang zwischen den beiden Speicherbereichen übernimmt und dadurch den M4 entlastet.

Eine andere Ursache für den hohen Ausreißer kann möglicherweise auf ein Hardwareproblem, wie durch den vorhandenen Systembus des MPSoCs, zurückgeführt werden. Eine zusätzliche Ursache kann die Speicherausrichtung sein, wodurch für einen Datenzugriff mehrere Speicherzugriffe notwendig sein können und daher kein konstanter Zugriff auf Speicher garantiert werden kann.

8.4.3 Hauptversuche 4 und 5

Durch die Hauptversuche 4 und 5 konnte wieder festgestellt werden, dass unabhängig von der Wiederholungsrate oder der Nachrichtengröße Ausreißer auftreten. Durch die durchgeföhrten Optimierungen durch den RT-Patch und der Task-Echtzeitpriorität von 99 konnten die maximalen Latenzzeiten bei 496 Bytes bei TTY-Write von 125396 auf $628 \mu\text{s}$ und bei TTY-Read von 125472 auf $55914 \mu\text{s}$ reduziert werden. Zwar ist eine Verbesserung erkennbar, allerdings ist es wünschenswert, diese Ausreißer weiter zu reduzieren.

Folgende weitere Optimierungen bieten sich für die Reduzierung der maximalen Latenzzeiten an:

- Analyse der Ursache für die hohen Ausreißer mithilfe von Werkzeugen wie TraceCompass oder TraceShark.
- Optimierung des RT-Patches im Linux-Kernel durch Anpassung der Konfigurationsparameter mithilfe von *menuconfig*.
- Minimierung des eingesetzten Yocto-Images.

Da eine weitere Auseinandersetzung mit den möglichen Maßnahmen den Rahmen der Arbeit sprengen würde und die Durchführung der Optimierungsmaßnahmen nicht Ziel der Arbeit ist, wurden keine weiteren Optimierungen durchgeführt.

8.4.4 Hauptversuch 6

Die Datenübertragung Richtung A35 ist deutlich höher als in Richtung M4 und beträgt 12,16 MB/s. Im Vergleich liegt die maximale Datenübertragung Richtung M4 bei 4,08 MB/s. Für einen höheren Datendurchsatz bietet sich der Einsatz einer DMA sowie die Erhöhung der RPMsg Buffergröße an, die bisher auf 496 Bytes begrenzt ist. Sobald eine Systemlast anliegt, reduziert sich der Datendurchsatz zwischen den beiden Prozessoren. Um im Falle einer auftretenden Systemlast einen höheren Datendurchsatz zu erzielen, ist die Abarbeitung der Datenübertragung innerhalb des Kernelspaces gegenüber dem Userspace vorzuziehen.

8.4.5 Stabilität des Lifecycle-Managements

Während der Durchführung der sechs Hauptversuche konnte die IPC über OpenAMP erfolgreich auf- und abgebaut werden, wenn ein Firmware-Wechsel durchgeführt wurde. Im Anhang können dazu die aufgenommenen Applikationslogmeldungen eingesehen werden (Anhang 10.1).

Kapitel 9

Résumé und Ausblick

In den folgenden Abschnitten wird die vorliegende Arbeit zusammengefasst, es werden zentrale Punkte dieser Arbeit diskutiert und es wird ein Ausblick gegeben.

9.1 Zusammenfassung und Zielerreichung

Die vorliegende Arbeit stellt die wesentlichen Möglichkeiten von laufenden Betriebssystemen für AMP-Systeme auf heterogenen Multiprozessorsystemen vor. So kann auf einem ARM Cortex-A35 ein Betriebssystem als sAMP oder uAMP laufen, während auf einem ARM Cortex-M4 ein Echtzeitbetriebssystem wie ein FreeRTOS oder Bare-Metal läuft. Durch den AMP-System Ansatz entstehen Herausforderungen wie Speicher-, Ressourcen oder Systemboot-Management sowie die Systemsicherheit und das Debugging. Auch die beiden Problemgruppen LCM und IPC wurden näher betrachtet. Es wurde festgestellt, dass für auftretende Herausforderungen kein plattformübergreifender Lösungsansatz besteht und für jeden heterogenen MPSoC einzeln betrachtet werden muss. Das erste Teilziel konnte durch die Darstellung der relevanten Grundlagen und Herausforderungen auf heterogenen MPSoC erfolgreich erreicht werden.

Anschließend wurden die drei Architekturen für AMP-Systeme MCS, LPS und CS vorgestellt. Für das zu entwickelnde AMP-System MMAMP wurde die CS-Architektur ausgewählt, da sie ihren Schwerpunkt auf die LCM und das IPC setzt. Für die Erreichung des Hauptziels wurden die Software-Frameworks OpenCL, Cilk, openMP, MC API, RPMsgLite und OpenAMP betrachtet. Das OpenAMP Framework, das von der Multicore Association (MCA) standardisiert ist und durch die Firmen (Xilinx, Mentor) unterstützt wird, wurde aufgrund der Bereitstellung von Funktionalitäten für das LCM und die IPC ausgewählt. Die vorhandenen plattformunabhängigen Komponenten aus dem Linux-Kernel, die von OpenAMP eingesetzt werden, sind ein großer Vorteil für dieses Framework. Durch die Auswahl von OpenAMP wurde ein standardisierter Ansatz für die beiden Problemgruppen gefunden, der auf unterschiedlichen heterogenen MPSoC eingesetzt werden kann. Hierdurch wurde das Haupziel der Arbeit erreicht.

Im Anschluss erfolgte die Umsetzung der Entwicklung des AMP-Systems MMAMP auf dem ausgewählten heterogenen MPSoC i.MX8X von NXP. Hierbei wurde ein Konzept erstellt, das in einen Systementwurf für die eingesetzten Komponenten und einen Entwurf für das Systemboot-Management unterteilt wurde. Im eingesetzten Linux-Kernel auf dem ARM Cortex-A35 wurden Anpassungen im RemoteProc-Driver erfolgreich durchgeführt, um das LCM auf der i.MX8X Plattform einsetzen zu können. Zugleich wurde auf dem eingesetzten FreeRTOS auf dem ARM Cortex-M4 das OpenAMP Framework erfolgreich portiert. Hierdurch wurde das zweite Teilziel erreicht.

Für die Evaluierung des umgesetzten AMP-Systems wurden die zwei Anwendungsszenarien Prozessdatenüberwachung und HiL vorgestellt und die relevanten Aspekte wie Latenzzeiten, Datendurchsatz und die OpenAMP Verarbeitungszeit abgeleitet. Im Anschluss wurde die Versuchsplanung erstellt, welche für die Versuchsdurchführung einen Logic-Analyzer, *Ftrace* und das Toggeln von GPIOs einsetzt. Durch die auftretenden Messungenauigkeiten wurde der gesamte Messfehler auf $2 \mu\text{s}$ geschätzt und stellt durch die auftretenden Messergebnisse im mehrstelligen Mikrosekundenbereich keine signifikante Messabweichung dar. Anschließend erfolgte die Durchführung der sechs Hauptversuche, welche die IPI-Latenzzeit, TTY-Device-Latenzzeit, OpenAMP Verarbeitungszeit und den Datendurchsatz zwischen den Prozessoren ARM Cortex-A35 und ARM Cortex-M4 untersuchen. Die Durchführung der Hauptversuche erfolgte durch den Einsatz von zwei unterschiedlichen Firmwares, die zur Laufzeit über das LCM erfolgreich ausgewechselt wurden. Durch die erfolgreiche Evaluierung der umgesetzten Lösung wurde das dritte Teilziel erreicht. Weitere Ergebnisse der Evaluierung werden im nachfolgenden Abschnitt näher erläutert.

Zusammenfassend wurden alle gesetzten Ziele der Arbeit (Kap. 1.3) erreicht.

9.2 Diskussion

9.2.1 Entwicklung des MMAMP-Systems

Die Entscheidung bei der Entwicklung des MMAMP-Systems auf der Coprocessor Systems-Architektur aufzubauen, erwies sich aufgrund der Möglichkeit für die Auslagerung von gezielten Anwendungen auf Coprozessoren und die daraus notwendige Bereitstellung des LCM und der IPC als richtig. Der Einsatz von OpenAMP auf dem ausgewählten heterogenen MPSoC zeigte sich allerdings im Hinblick auf den hohen Implementierungsaufwand als Nachteil, da das eingesetzte Embedded-Linux für den MPSoC i.MX8X keinen plattformabhängigen RemoteProc-Driver zur Verfügung stellte.

Außerdem führte die hohe Komplexität des MPSoCs i.MX8X zu einer erhöhten Einarbeitungszeit durch die vielen unterschiedlichen integrierten Subsysteme. So bringt die vorhandene SCU hardwareabhängige Funktionalitäten für die Steuerung des Systemboot-, Speicher- und Ressourcen-Management mit, die auf Ressourcen und Partitionen aufbauen. Dadurch wird unabhängig vom eingesetzten MPSoC eine hohe Einarbeitungszeit gefordert, da jede Plattform ihre Besonderheiten bereitstellt und die Hersteller keinen gemeinsamen Ansatz verfolgen.

Bei der Portierung von OpenAMP auf dem ARM Cortex-M4 hat sich durch die Trennung des plattformabhängigen Teils durch die hardwareabhängige Portierungsebene innerhalb der Libmetal-Bibliothek als großen Vorteil erwiesen. Allerdings wurde in dieser Arbeit nur ein Anwendungsfall umgesetzt und das gesamte Potential von OpenAMP nicht ausgenutzt. Daher hätte der Einsatz von RPMsgLite für die Evaluierung der IPC ausgereicht. Doch wegen der fehlenden Funktionalitäten für das LCM wurde OpenAMP dennoch eingesetzt und nicht RPMsglite. Für Anwendungen für die ausschließlich eine IPC zwischen Prozessoren ohne LCM gefordert ist, ist RPMsgLite die bessere Wahl. Jedoch muss dabei beachtet werden, dass diese Lösung ausschließlich für die von NXP zur Verfügung gestellten MPSoCs eingesetzt werden kann.

Als Alternative zu dem i.MX8X hätten sich die MPSoCs von Xilinx wie der MPSoC Zynq UltraScale+ angeboten. Auf dieser Plattform ist bereits die Portierung von OpenAMP für den Einsatz von unterschiedlichen Anwendungsfällen erfolgt. Allerdings wurde sich gegen den MPSoC von Xilinx entschieden aufgrund der erläuterten Argumente (Kap. 3.3) [75].

9.2.2 Ergebnisse der Evaluierung

Die aufgestellte Messmethodik hat sich aufgrund der aufschlussreichen Ergebnisse über die Latenzzeiten und dem Datendurchsatz zwischen den beiden Prozessoren bewährt. Der hierbei auftretende gesamte Messfehler war ebenso vernachlässigbar (Kap. 8.2.4). Allerdings stellte die Durchführung der Messungen einen zeitlich hohen Aufwand dar, da sie nicht automatisiert wurde und die Messungen der einzelnen Hauptversuche daher separat durchgeführt werden mussten. Dennoch kann die Vorgehensweise der Messmethodik auf andere MPSoC übertragen werden. Eine Optimierung durch eine komplette Messautomatisierung ist jedoch empfehlenswert. In Tabelle 9.1 sind die wichtigsten Ergebnisse mit einer Nachrichtengröße von 496 Bytes zusammengefasst. Die durchgeföhrten Versuche zeigen einen Überblick über die Latenzzeiten und den Datendurchsatz zwischen den beiden Prozessoren. Durch den Einsatz des RT-Patches konnten die maximalen Latenzzeiten für das TTY-Device deutlich reduziert werden. Jedoch konnten hohe Ausreißer nicht verhindert werden.

Nachrichtengröße: 496 Bytes	TTY- Write	TTY- Read	IPI- Latenzzeit nach M4	IPI- Latenzzeit nach A35	OpenAMP- Verarbeitungs- zeit
Max. [μ s]	628	55291	809	581	365
Min. [μ s]	10	4	3	4	190
Arith. Mittel [μ s]	15	278	4	5	198
Standardabweichung [μ s]	5	108	1	2	1
99,9%-Quantil [μ s]	50	942	4	11	200

Tabelle 9.1: Zusammenfassung der Messergebnisse für die auftretenden Latenz- und Verarbeitungszeiten bei einer Nachrichtengröße von 496 Bytes.

Bei der Prozessdatenüberwachung für den Einsatz von Predictive Maintenance führen zu spät ankommende Prozessdaten nicht zu einem Ausfall, denn im schlechtesten Fall wird ein Datensatz zu einem späteren Zeitpunkt empfangen, was sehr selten vorkommt. Ein Ankommen aller Nachrichten ist garantiert und ein Verlust einzelner Nachrichten kann ausgeschlossen werden. Unabhängig von der verspäteten Nachrichtenübertragung kann der M4 die Motorsteuerung fortsetzen und bei Notfällen die Motoren abschalten und dennoch Ausreißer an den anderen Prozessor übertragen, der die Daten über ein HMI darstellt. Aus diesen Gründen ist festzustellen, dass die entstehenden Ausreißer vernachlässigbar sind und das MMAMP-System für den Einsatz von Predictive Maintenance sowie für die Qualitätssicherung geeignet ist.

Der Einsatz für HiL-Szenarien ist mit dem MMAMP-System umsetzbar. Die hohe Datenübertragung von bis zu 12,16 MB/s ermöglicht eine schnelle Übertragung von Konfigurations- oder Kalibrierungsdateien. Auch der Wechsel von Firmwares über das LCM zwischen der Ausführung von Tests kann problemlos durchgeführt werden.

Für andere Anwendungsszenarien können Kunden eine Einschätzung dazu treffen, ob die Veruchsergebnisse ihre Anforderungen erfüllen, wenn OpenAMP für die IPC und das LCM ein-

gesetzt wird. Allerdings sind die aufgezeichneten Ergebnisse nur begrenzt aussagekräftig, nur für den eingesetzten MPSoC i.MX8X gültig und nicht auf andere MPSoC übertragbar, da die Latenzzeiten von der vorhandenen Hardwarearchitektur und dessen integrierten Prozessoren abhängen.

Für den Einsatz von harter Echtzeit ist die IPC des MMAMP-Systems nicht geeignet. Der Einsatz für weiche Echtzeit ist hingegen plausibel, wenn die aufgezeigten Ausreißer (Tab. 9.1) tolerierbar sind.

9.3 Ausblick

9.3.1 Optimierungen bei Latenzzeiten und bei Datendurchsatz

Die Ergebnisse der Versuche zeigten hohe Ausreißer bei den Latenzzeiten, für die ausschließlich Hypothesen für die Ursachen abgeleitet wurden. Die Ursachen wurden im Rahmen dieser Arbeit nicht näher betrachtet und nur noch Ansätze für die Untersuchung vorgeschlagen (Kap. 8.4.3). Hierfür sollten weitere Untersuchungen durchgeführt werden.

Ein weiterer Aspekt, der im Rahmen der Arbeit nicht untersucht wurde, ist die Optimierung des Datendurchsatzes durch den Einsatz einer DMA oder die Erhöhung der RPMsg Buffergröße, die bisher auf 512 Bytes begrenzt ist.

9.3.2 Erweiterung des Parsens der Firmware

Die bisher eingesetzten M4-Firmwares, die über das LCM durch RemoteProc geladen werden, beschränken sich auf den TCM-Speicherbereich, der auf 256 KB begrenzt ist. Der portierte OpenAMP Softwarestack benötigt davon insgesamt bis zu 130 KB, wodurch für die Applikation nicht mehr viel Speicherplatz zur Verfügung steht. Daher ist es sinnvoll, das Parsen der ELF-Firmware innerhalb des i.MX8X-RemoteProc-Drivers zu erweitern, damit Teile der Firmware in den externen DDR-Speicher ausgelagert werden können.

9.3.3 Einsatz in der Praxis

Bisher wurden zwar die Anwendungsszenarien Prozessdatenüberwachung und HiL vorgestellt, allerdings wurde die umgesetzte Lösung noch nicht in der Praxis erprobt. Eine tatsächliche Verwendung des Systems in der Praxis, um den erfolgreichen Einsatz zu evaluieren, ist eine weitere Aufgabe, die umgesetzt werden kann.

9.3.4 Ansätze der anderen AMP-Architekturen

Die zwei anderen Möglichkeiten der AMP-Architekturen LPS und MCS wurden zwar in dieser Arbeit erklärt, jedoch wurde keiner dieser Ansätze umgesetzt. Daher ist ein möglicher Schritt, eine dieser Architekturen auf einem heterogenen MPSoC anzuwenden. Besonders spannend ist hierbei der Einsatz von sAMP auf einem MCS-System, um hohe sicherheitskritische Anforderungen zu erfüllen. Bei dem Einsatz der LPS-Architektur ist die Höhe des Energieverbrauchs interessant und der Einsatz von OpenAMP mit dem ARM Cortex-M4 als Master-Prozessor, der zur Laufzeit den ARM Cortex-A35 für rechenintensive Aufgaben steuern kann.

9.3.5 Entwicklung einer grafischen Benutzeroberfläche

Die bisher entwickelten Pythonanwendungen MeasureMan und DataExporter werden ausschließlich über die Konsole bedient und bieten keine grafische Benutzeroberfläche. Daher ist die Entwicklung einer grafischen Benutzeroberfläche von Vorteil, um die Bedienbarkeit einfacher zu gestalten, eine bessere Übersicht über das laufende System zu erhalten und die Messergebnisse nach Ausführung der Versuche darzustellen.

9.3.6 Proprietäre Lösung für die IPC

Alternativ sind auch völlig andere Optimierungsansätze denkbar, die nicht den standardisierten Ansatz OpenAMP einsetzen, sondern eine proprietäre Lösung verwenden, wodurch der Datendurchsatz deutlich höher ausfallen kann, allerdings die Portierbarkeit auf unterschiedliche Plattformen eingeschränkt ist.

Kapitel 10

Anhang

10.1 MeasureMan und OpenAMP M4-Applikation

Linux Logausgaben für den Wechsel der Firmware über den MeasureMan

```
1 // 1. Start firmware latency_app_imxcm4.elf
2 root@imx8qxp-var-som:~# python3.5 /home/root/measureman/measureman.py setup_openamp default
3 Start measureman application
4 Init ImxOpenAMPController
5 Number of arguments: 3 arguments.
6 Argument List: [/home/root/measureman/measureman.py, setup_openamp, default]
7 Setup default openamp stack
8 Start system with type default
9 Serial port /dev/ttyRPMSG30 opened
10 Speed set to 115200
11 // 2. Load firmware throughput_app_imxcm4.elf to /lib/firmware
12 root@imx8qxp-var-som:~# python3.5 /home/root/measureman/measureman.py
13 load_firmware throughput_app_imxcm4.elf
14 Start measureman application
15 Init ImxOpenAMPController
16 Number of arguments: 3 arguments.
17 Argument List: [/home/root/measureman/measureman.py, load_firmware,
18 throughput_app_imxcm4.elf]
19 load_firmware: throughput_app_imxcm4.elf
20 // 3. Reload M4 with new firmware throughput_app_imxcm4.elf
21 root@imx8qxp-var-som:~# python3.5 /home/root/measureman/measureman.py reload_openamp default
22 Start measureman application
23 Init ImxOpenAMPController
24 Number of arguments: 3 arguments.
25 Argument List: [/home/root/measureman/measureman.py, reload_openamp, default]
26 Reload openamp stack
27 Stop system
28 Rproc ist running -> stop rproc
29 Start system with type default
30 Serial port /dev/ttyRPMSG30 opened
31 Speed set to 115200
```

FreeRTOS Logausgaben für den Wechsel der Firmware auf dem M4

```
1 **** Latency App ... 0.0.0.7 ****
2 Memreg 2 0x34000000 -- 0x0
3 Memreg 8 0x88000000 -- 0x0
4 metal: info: platform.c:267 platform initialized
5 metal: info: platform_info.c:185 init_system - success
6 metal: info: rsc_table.c:80 get_resource_table rsc_table size: 1x
```

```

7 metal: info: sys.c:77 no address mapping. return raw virtual adress
8 metal: info: device.h:156 index: 0 , num_regions 1
9 metal: info: remoteproc_init success
10 metal: info: sys.c:77 no address mapping. return raw virtual adress
11 metal: info: remoteproc.h:551 remoteproc_add_mem finished
12 metal: info: platform_info.c:123 rsc_table memory regions created - 20000200
13 metal: info: sys.c:77 no address mapping. return raw virtual adress
14 metal: info: remoteproc.h:551 remoteproc_add_mem finished
15 metal: info: platform_info.c:131 SHARED_MEM_PA vring memory regions created - 90010000
16 metal: info: sys.c:77 no address mapping. return raw virtual adress
17 metal: info: remoteproc.h:551 remoteproc_add_mem finished
18 metal: info: platform_info.c:142 SHARED_BUFFER_MEM_PA buffer memory regions created - 90400000
19 metal: info: rsc_table_parser.c:40 rsc table version: 1
20 metal: info: rsc_table_parser.c:89 rsc table handling finished
21 metal: info: platform_info.c:153 Initialize remoteproc successfully.
22 metal: info: openamp_controller.c:17 Platform intialized.
23 metal: info: platform_info.c:231 creating remoteproc virtio
24 metal: info: platform_info.c:239 initializing virtio device
25 metal: info: platform_info.c:273 initializing rpmsg device
26 metal: info: latency_app.c:177 Successfully created rpmsg endpoint.
27 metal: info: latency_app.c:195 openamp communication finished.
28 metal: info: platform.c:56 platform_deinit_interrupt for interrupt id: 3, isr_counter: 1
29
30 **** Latency App ... finish ****
31
32 **** Throughput App ... 0.0.0.7 ****
33 Memreg 2 0x34000000 -- 0x0
34 Memreg 8 0x88000000 -- 0x0
35 metal: info: platform.c:267 platform initialized
36 metal: info: platform_info.c:185 init_system - success
37 metal: info: rsc_table.c:80 get_resource_table rsc_table size: 1x
38 metal: info: sys.c:77 no address mapping. return raw virtual adress
39 metal: info: device.h:156 index: 0 , num_regions 1
40 metal: info: remoteproc_init success
41 metal: info: sys.c:77 no address mapping. return raw virtual adress
42 metal: info: remoteproc.h:551 remoteproc_add_mem finished
43 metal: info: platform_info.c:123 rsc_table memory regions created - 20000200
44 metal: info: sys.c:77 no address mapping. return raw virtual adress
45 metal: info: remoteproc.h:551 remoteproc_add_mem finished
46 metal: info: platform_info.c:131 SHARED_MEM_PA vring memory regions created - 90010000
47 metal: info: sys.c:77 no address mapping. return raw virtual adress
48 metal: info: remoteproc.h:551 remoteproc_add_mem finished
49 metal: info: platform_info.c:142 SHARED_BUFFER_MEM_PA buffer memory regions created - 90400000
50 metal: info: rsc_table_parser.c:40 rsc table version: 1
51 metal: info: rsc_table_parser.c:89 rsc table handling finished
52 metal: info: platform_info.c:153 Initialize remoteproc successfully.
53 metal: info: openamp_controller.c:17 Platform intialized.
54 metal: info: platform_info.c:231 creating remoteproc virtio
55 metal: info: platform_info.c:239 initializing virtio device
56 metal: info: platform_info.c:273 initializing rpmsg device
57 metal: info: throughput_app.c:181 Successfully created rpmsg endpoint.

```

10.2 Messungen

Vorversuch 1 - Messung der Kernelspace GPIO-Latenzzeit

Vorversuch 1, Kernelspace GPIO-Latenzzeit		1
Max. [μs]		20
Min. [μs]		0
Arith. Mittel [μs]		0
Standardabweichung [μs]		0
99%-Quantil [μs]		0
99,9%-Quantil [μs]		1

Tabelle 10.1: Ergebnisse zu Vorversuch 1: Kernelspace GPIO-Latenzzeit. Experiment mit einer Messung durchgeführt.

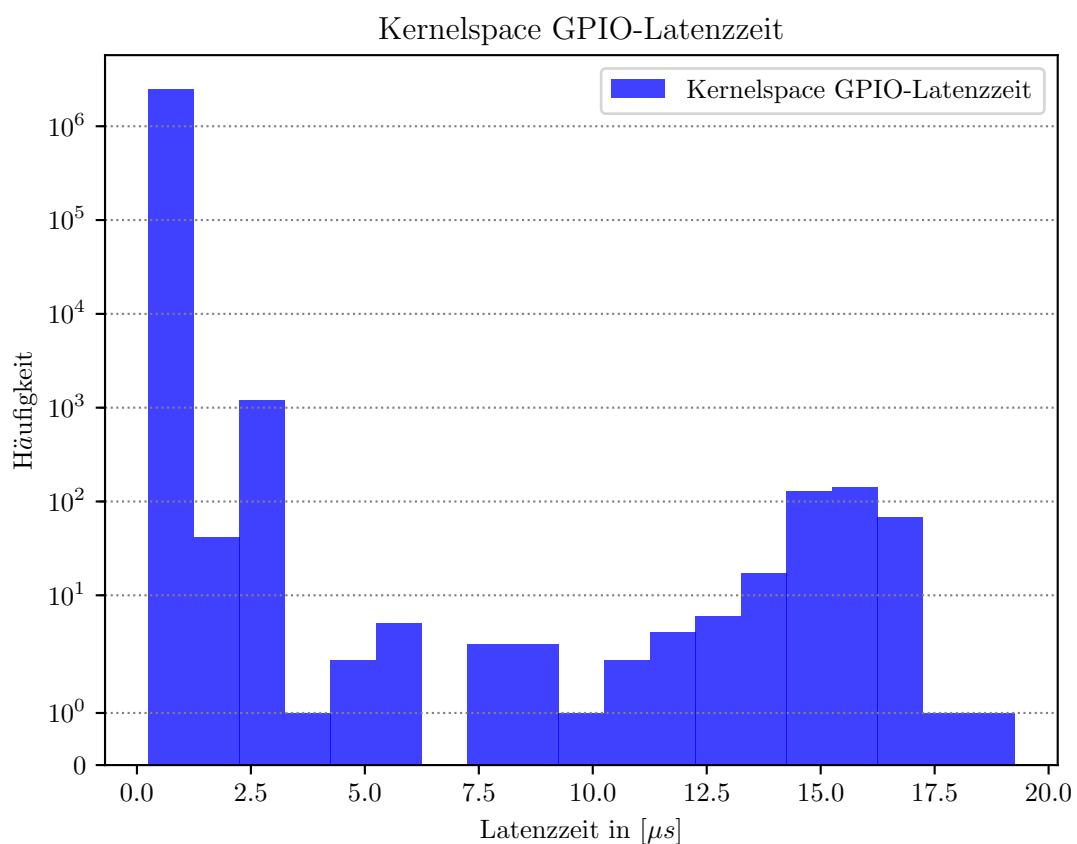


Abbildung 10.1: Histogramm zu Vorversuch 1: Kernelspace GPIO-Latenzzeit.

Vorversuch 2 - Messung der M4 GPIO-Latenzzeit

Vorversuch 2, M4 GPIO-Latenzzeit		1
Max. [μs]		4.6
Min. [μs]		0
Arith. Mittel [μs]		0
Standardabweichung [μs]		0
99%-Quantil [μs]		0
99,9%-Quantil [μs]		0

Tabelle 10.2: Ergebnisse zu Vorversuch 2: M4 GPIO-Latenzzeit. Experiment mit einer Messung durchgeführt.

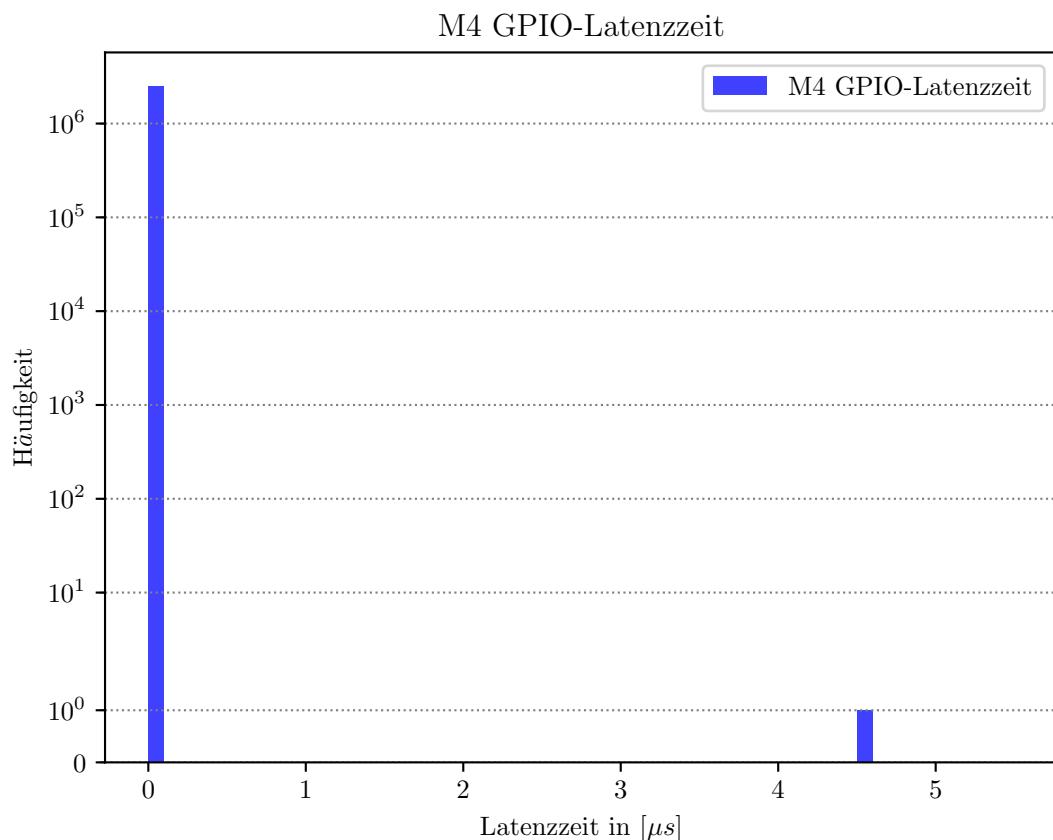


Abbildung 10.2: Histogramm zu Vorversuch 2: M4 GPIO-Latenzzeit.

Literaturverzeichnis

- [1] Storm Frank und Silica Avnet. *OpenAMP - Endlich! - ESE 2017 — MicroConsult*. 2017. URL: <https://www.microconsult.de/1714-0-OpenAMP---Endlich---ESE-2017.html> (besucht am 24.08.2020).
- [2] Pat Stilwell u. a. *i.MX 8 Product Overview*. 2018.
- [3] Louise Crockett u. a. *The Zynq Book*. Strathclyde Academic Media, 2014. URL: <http://www.zynqbook.com/>.
- [4] Philipp Wendler. *Multicore-Architekturen*. 2009.
- [5] Sabri Pllana und Fatos Xhafa. *Programming Multicore and Many-Core Computing Systems*. John Wiley & Sons, 2017.
- [6] Michael Hübner und Jürgen Becker. *Multiprocessor System-on-Chip*. Springer Science & Business Media, 2011.
- [7] Rakesh Kumar, Dean M. Tullsen und Norman P. Jouppi. “Core architecture optimization for heterogeneous chip multiprocessors”. In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT 2006*. June 2014 (2006), S. 23–32.
- [8] Urs Gleim und Tobias Schüle. *Multicore-Software*. dpunkt.verlag, 2011.
- [9] Heinrich Klar und Tobias Noll. *Integrierte Digitale Schaltungen*. 2015.
- [10] Gregor Sievers. “Entwurfsraumexploration eng gekoppelter paralleler Rechnerarchitekturen”. Doktorarbeit. Technischen Fakultät der Universität Bielefeld, 2016.
- [11] Mark D. Hill und Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* 50.6 (2008), S. 33–38.
- [12] Mpsoc Technology u. a. “Multiprocessor System-on-Chip”. In: *IEEE Transactions on Multi-Scale Computing Systems* 27.10 (2008), S. 1701–1713.
- [13] Katalin Popovici und Wolf Marilyn. *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. Springer Science & Business Media, 2010.
- [14] Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran und Rupesh Nasre. “CHO-AMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors”. In: *IEEE Transactions on Multi-Scale Computing Systems* 4.2 (2018), S. 163–176.
- [15] Youssef Zaki. “An Embedded Multi-Core Platform for Mixed-Criticality Systems”. Masterarbeit. KTH Royal Institute of Technology School of Information und Communication Technology Stockholm, Sweden, 2016.
- [16] Vishal Gupta u. a. “The forgotten ‘Uncore’: On the energy-efficiency of heterogeneous cores”. In: *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012 September 2016* (2019), S. 367–372.
- [17] Alexandra Fedorova u. a. “Maximizing power efficiency with asymmetric multicore systems”. In: *Communications of the ACM* 52.12 (2009), S. 48–57.

- [18] Sunao Torii u. a. "Asymmetric multi-processing mobile application processor MP211". In: *NEC Journal of Advanced Technology* 2.3 (2005), S. 204–210.
- [19] Louise Crockett u. a. *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Strathclyde Academic Media, 2019.
- [20] *Android Camera - Variscite Wiki*. URL: https://variwiki.com/index.php?title=Android_Camera (besucht am 24.06.2020).
- [21] Jürgen Quade. *Harte und weiche Echtzeitsysteme Material zur Vorlesung Echtzeitsysteme an der Hochschule Niederrhein*. 2011.
- [22] Bryon Moyer. *Real World Multicore Embedded Systems*. Newnes, 2013, S. 1–10.
- [23] Steffen Rempp. "Multithreading für eine Bildverarbeitungs-Pipeline unter Linux auf einem ARM-basierten MPSoC". Masterarbeit. Hochschule für Angewandte Wissenschaften Hamburg, 2013.
- [24] TenAsys. "Asymmetric Multi-Processing (AMP) Systems vs . Symmetric Multi-Processing (SMP) Systems". 2016.
- [25] Xilinx. *Zynq UltraScale+ MPSoC Software Developer Guide*. 2019. URL: https://www.xilinx.com/support/documentation/user_guides/ug1137-zynq-ultrascaling-mpsoc-swdev.pdf.
- [26] *Home - Xen Project*. URL: <https://xenproject.org/> (besucht am 14.08.2020).
- [27] Uwe Schneider. *Taschenbuch der Informatik* -. München; Wien: Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2012.
- [28] Pedro Ignacio Martos, Universidad De Buenos Aires und Universidad De La Plata. "Architectural Patterns for Asymmetric Multiprocessing Devices on Embedded Systems". In: (2016), S. 1–13.
- [29] Mustafa O E Aboelhassan, Ondrej Bartik und Marek Novak. "Embedded Multi-Core Systems for Mixed-Critical Applications with RPMsg Protocol Based on Xilinx ZYNC-7000". In: November (2017), S. 24–26.
- [30] Felix Baum. *Heterogeneous Multicore OpenAMP*. 2016.
- [31] Songchun Fan und Benjamin C. Lee. "Evaluating asymmetric multiprocessing for mobile applications". In: *ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software* (2016), S. 235–244.
- [32] *STM32MPU - Platform boot*. URL: https://wiki.st.com/stm32mpu/wiki/Category:Platform_boot (besucht am 25.06.2020).
- [33] *STM32MPU - Wikipedia*. URL: https://wiki.st.com/stm32mpu/wiki/Main_Page (besucht am 25.06.2020).
- [34] *STM32MP153C/F*. URL: <https://www.st.com/resource/en/datasheet/stm32mp153c.pdf>.
- [35] *ZCU104 Evaluation Board User Guide*. 2018. URL: www.xilinx.com.
- [36] "Zynq Power Management Framework User Guide". User Guide. 2016. URL: www.xilinx.com.
- [37] *Zynq UltraScale+ MPSoC - Xilinx Wiki - Confluence*. URL: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/444006775/Zynq+UltraScale+MPSoC> (besucht am 25.06.2020).

- [38] Variscite. *VAR-SOM-MX8X - Variscite Wiki*. URL: <http://variwiki.com/index.php?title=VAR-SOM-MX8X> (besucht am 25.06.2020).
- [39] Variscite. “VAR-SOM-MX8X Datasheet”. Datasheet. 2019.
- [40] Trio Adiono u. a. “An inter-processor communication (IPC) data sharing architecture in heterogeneous MPSoC for OFDMA”. In: *Journal of ICT Research and Applications* 12.1 (2018), S. 70–86.
- [41] Suyang Zhu u. a. “Exploring task parallelism for heterogeneous systems using multicore task management API Paper”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10104 LNCS (2017), S. 697–708.
- [42] Lauri Matilainen u. a. “Multicore Communications API (MCAP) implementation on an FPGA multiprocessor”. In: *Proceedings - 2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2011* July (2011), S. 286–293.
- [43] Peng Sun, Sunita Chandrasekaran und Barbara Chapman. “OpenMP-MCA: Leveraging Multiprocessor Embedded Systems Using Industry Standards”. In: *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015* May (2015), S. 679–688.
- [44] Jerry L Potter. *The Massively Parallel Processor*. Cambridge, MA, USA: MIT Press, 1985.
- [45] Warren Kurisu. *Implementierung eines Multicore-Frameworks — All-Electronics*. 2017. URL: <https://www.all-electronics.de/multicore-framework/> (besucht am 26.06.2020).
- [46] *Multicore Framework - Mentor Graphics*. URL: <https://www.mentor.com/embedded-software/multicore> (besucht am 26.06.2020).
- [47] *OpenAMP/open-amp*. URL: <https://github.com/OpenAMP/open-amp> (besucht am 26.06.2020).
- [48] *Linux_3.4 - Linux Kernel Newbies*. URL: https://kernelnewbies.org/Linux_3.4 (besucht am 18.08.2020).
- [49] NXP. *NXPmicro/rpmsg-lite: RPMsg implementation for small MCUs*. URL: <https://github.com/NXPmicro/rpmsg-lite> (besucht am 26.06.2020).
- [50] *OpenAMP Overview · OpenAMP/open-amp Wiki*. URL: <https://github.com/OpenAMP/open-amp/wiki/OpenAMP-Overview> (besucht am 26.06.2020).
- [51] *OpenAMP Components and Capabilities · OpenAMP/open-amp Wiki*. URL: <https://github.com/OpenAMP/open-amp/wiki/OpenAMP-Components-and-Capabilities> (besucht am 26.06.2020).
- [52] *Release OpenAMP v2016.10 Release · OpenAMP/open-amp*. URL: <https://github.com/OpenAMP/open-amp/releases/tag/v2016.10> (besucht am 26.06.2020).
- [53] *OpenAMP Life Cycle Management · OpenAMP/open-amp Wiki*. URL: <https://github.com/OpenAMP/open-amp/wiki/OpenAMP-Life-Cycle-Management> (besucht am 26.06.2020).
- [54] *RPMsg Messaging Protocol · OpenAMP/open-amp Wiki*. URL: <https://github.com/OpenAMP/open-amp/wiki/RPMsg-Messaging-Protocol> (besucht am 29.06.2020).

- [55] Red Hat und Red Hat. *Virtual I / O Device (VIRTIO) Version 1 . 0.* 2015.
- [56] *Remote Processor Messaging (rpmsg) Framework.* URL: <https://www.kernel.org/doc/Documentation/rpmsg.txt> (besucht am 28.07.2020).
- [57] NXP Semiconductors. *i.MX 8DualX/8DualXPlus/8QuadXPlus Applications Processor Reference Manual.* 2019.
- [58] NXP. *i.MX 8X Family of Applications Processors.* Techn. Ber. 2020. URL: www.nxp.com/productlongevity.
- [59] NXP Semiconductors. *System Controller Firmware Porting Guide.* 2019.
- [60] Manuel Rodriguez. *Introduction to the i.MX8 System Controller Unit and System Controller Firmware.* 2018.
- [61] NXP Semiconductors. *Secure Boot on i.MX 8 and i.MX 8X Families using AHAB.* 2019.
- [62] NXP Semiconductors. *i . MX Porting Guide.* 2020.
- [63] *Android Auto — Android.* URL: https://www.android.com/intl/de_de/auto/ (besucht am 18.08.2020).
- [64] NXP Semiconductors. *Getting Started with MCUXpresso SDK for i.MX 8QUadXPlus.* 2019.
- [65] *Symphony-Board Datasheet Symphony-Board Datasheet.* 2020.
- [66] *elf(5) - Linux manual page.* URL: <https://man7.org/linux/man-pages/man5/elf.5.html> (besucht am 31.07.2020).
- [67] C Hallinan. *Embedded Linux Primer Second Edition.* 2011.
- [68] *Asymmetric Multiprocessing: RPMsg – Linux/Android — Kynetics.* URL: https://www.kynetics.com/docs/2018/Linux_rpmsg_char_driver/ (besucht am 28.07.2020).
- [69] *OpenAMP Porting GuideLine · OpenAMP/open-amp Wiki · GitHub.* URL: <https://github.com/OpenAMP/open-amp/wiki/OpenAMP-Porting-GuideLine> (besucht am 12.02.2020).
- [70] *6 Beispiele, wie man mit Industrie 4.0 Geld sparen kann.* URL: <https://www.produktion.de/trends-innovationen/id-6-beispiele-wie-man-mit-industrie-4-0-geld-sparen-kann-295.html> (besucht am 14.08.2020).
- [71] *Woraus die Smart Factory besteht und was aktuell dazu kommt.* URL: <https://www.produktion.de/trends-innovationen/woraus-die-smart-factory-bestehst-und-was-aktuell-dazu-kommt-118.html> (besucht am 18.07.2020).
- [72] *HIL-Simulation (Hardware-in-the-Loop) - MATLAB & Simulink.* URL: <https://de.mathworks.com/discovery/hardware-in-the-loop-hil.html> (besucht am 02.08.2020).
- [73] Red Hat Inc. *ftrace - Function Tracer.* 2008. URL: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt> (besucht am 13.07.2020).
- [74] ARM. *ARM Cortex M-4 Technical Reference Manual.* 2015.
- [75] *OpenAMP 2020.1 - Xilinx Wiki - Confluence.* URL: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/441942867/OpenAMP+2020.1> (besucht am 03.08.2020).