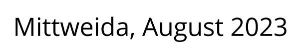
HOCHSCHULE
MITTWEIDA
University of Applied Sciences

BACHELORARBEIT

Herr
Moritz Marko Pöhlandt

Voruntersuchungen zum Einsatz des Zephyr RTOS für eine bestehende Gerätesteuerung



Mittweida, August 2023

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Voruntersuchungen zum Einsatz des Zephyr RTOS für eine bestehende Gerätesteuerung

Autor:
Moritz Marko Pöhlandt

Studiengang:
Angewandte Informatik

Seminargruppe:
IF20wS1-B

Erstprüfer:
Prof. Dr.-Ing. Thomas Beierlein

Zweitprüfer:
Jens Römer, Dipl.-Ing.

Einreichung:
Mittweida, 25.08.2023

Verteidigung/Bewertung:
Mittweida, 2023

Faculty of **Applied Computer Sciences and Biosciences**

BACHELOR THESIS

Preeliminary investigation into the use of the Zephyr RTOS for an existing device control system

Author:
Moritz Marko Pöhlandt

Course of Study:
Applied Computer Science

Seminar Group:
IF20wS1-B

First Examiner:
Prof. Dr.-Ing. Thomas Beierlein

Second Examiner:
Jens Römer, Dipl.-Ing.

Submission:
Mittweida, 25.08.2023

Defense/Evaluation:
Mittweida, 2023

Bibliografische Beschreibung

Pöhlandt, Moritz Marko:
Voruntersuchungen zum Einsatz des Zephyr RTOS für eine bestehende Gerätesteuerung. – 2023. – 51 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2023.

Referat

Die Anforderungen an Software in einer Gerätesteuerung werden zunehmend aufwändiger. Neben leistungsstarken Prozessoren mit teilweise mehreren Kernen und erheblich mehr Speicherplatz als noch vor ein paar Jahren, können neue Softwarekonzepte für Mikrocontroller verwendet werden. Dazu zählen unter anderem Echtzeitbetriebssysteme, welche nicht nur klassische Betriebssystemkonzepte, wie beispielsweise Threads einführen, sondern auch eine Abstraktion der zugrundeliegenden Hardware enthalten. Das Echtzeitbetriebssystem Zephyr enthält zum Beispiel eine große Sammlung an fertigen Bibliotheken, womit Steuerungssoftware für Geräte entwickelt werden kann. Das Ziel der vorliegenden Arbeit ist es, zu beantworten, welche Vor- und Nachteile der Einsatz des Echtzeitbetriebssystems Zephyr gegenüber einer Gerätesteuerung ohne ein unterliegendes System mit sich bringt. Dabei liegt der Fokus auf der Implementierung von Steuerungssoftware der Hardware mittels verschiedenen Ansätzen sowie der Kommunikation zwischen mehreren Kernen eines Mikrocontrollers. Um diese Fragen zu beantworten, wurden verschiedene Programme entwickelt und miteinander verglichen. Umfängliche Recherchen zu dem Thema haben ebenfalls dazu beigetragen, eine aussagekräftige Einschätzung bezüglich des Einsatzes von Zephyr für eine Gerätesteuerung zu treffen. Eine pauschale Empfehlung für das Zephyr Echtzeitbetriebssystem ist zum jetzigen Zeitpunkt nicht sinnvoll, da dies auf mehreren Faktoren beruht. Deshalb kommt die vorliegende Arbeit zu dem Ergebnis, dass der Einsatz für jedes System individuell geprüft werden sollte. Trotzdem bietet Zephyr viele Vorteile, welche die Entwicklung von Software vereinfachen kann.

Inhaltsverzeichnis

Inhaltsverzeichnis I

Abbildungsverzeichnis III

Tabellenverzeichnis V

Quelltextverzeichnis VII

Abkürzungsverzeichnis IX

1 Einleitung 1

2 Grundlagen 3
2.1 Eingebettete Systeme . . . 3
2.2 Bare Metal Programmierung . . . 3
2.3 Das Echtzeitbetriebssystem Zephyr . . . 4
2.3.1 Architektur . . . 5
2.3.2 Wichtige Betriebssystemkonzepte in Zephyr . . . 5
2.3.3 Verwaltungs- und Erstellungskonzepte . . . 8
2.4 Vorstellung des Prozessors i.MX RT1170 . . . 8

3 Zephyr Konzepte zur Hardwaresteuerung 9
3.1 Devicetree zur Abstraktion der Hardware . . . 9
3.1.1 Syntax und Struktur . . . 9
3.1.2 Bindings . . . 12
3.1.3 Das Konfigurationssystem Kconfig . . . 12
3.1.4 Projektkonfiguration mittels Kconfig . . . 13
3.2 Zephyr APIs . . . 13
3.2.1 Devicetree APIs . . . 13
3.2.2 Gerätetreibermodell . . . 14
3.2.3 SPI API zur Steuerung eines DAC's . . . 14

4 Steuerung von Hardwarekomponenten in einem eingebetteten System 19
4.1 Gegenüberstellung Zephyr RTOS zur Bare-Metal Umgebung . . . 19
4.2 Ansätze zur Integration von bestehenden Treibern in Zephyr . . . 19
4.2.1 Direkte Steuerung über der Register . . . 20
4.2.2 Verwendung herstellerspezifischer Treiber . . . 22
4.2.3 Gerätesteuerung mittels der Zephyr API . . . 23
4.2.4 Vergleich der Implementierungsmöglichkeiten . . . 25

5 Datenaustausch zwischen mehreren Kernen eines Mikrocontrollers 29
5.1 Anforderungen an die IPC . . . 29
5.2 IPC-Mechanismen im Embedded Bereich . . . 29
5.3 Dualcore Boot in Zephyr als Vorbereitung der IPC . . . 31
5.4 Umsetzung der IPC in Zephyr . . . 32
5.4.1 Messaging Unit zum Austausch von Informationen . . . 32

5.4.2 Zephyr IPM API . . . 34
5.4.2.1 Zeitmessung . . . 34
5.4.3 OpenAMP . . . 36
5.4.4 Vergleich der aufgeführten Möglichkeiten . . . 38

6 Firmware Update durch das Zephyr RTOS 41
6.1 Grundlagen MCUboot Bootloader . . . 41
6.2 Beispielablauf des Firmwareupdates mittels MCUboot . . . 43

7 Fazit 49
7.1 Zusammenfassung . . . 49
7.2 Ausblick . . . 50

Glossar 51

Literaturverzeichnis 53

Eidesstattliche Erklärung 55

Abbildungsverzeichnis

2.1 Softwarearchitektur des Zephyr RTOS [eigene Abbildung] 6
2.2 Zephyr Threadzustände [eigene Abbildung] 7
2.3 SMP Initialisierungsprozess einer Dualcore CPU mittels Zephyr [eigene Abbildung] 8

3.1 SPI Verbindung zwischen Master und zwei Slaves [eigene Abbildung] 15
3.2 DAC Rampenmessung mittels Oszilloskop [eigene Abbildung] 18

4.1 Zephyr API GPIO Pinwechsel [eigene Abbildung] 27

5.1 Auftreten einer Race Condition beim Shared Memory Zugriff [eigene Abbildung] 30
5.2 Block Diagramm der Messaging Unit [eigene Abbildung] 33
5.3 Ablaufdiagramm IPM Kommunikation [eigene Abbildung] 35
5.4 Ablaufdiagramm der Interprozessor-Kommunikation mittels Mailbox [eigene Abbildung] 35
5.5 Headerstruktur RPMsg [eigene Abbildung] 37

6.1 Image Manifest [eigene Abbildung] 43

Tabellenverzeichnis

3.1 ausgewählte Properties in Zephyr 10
3.2 Grundlegende Datentypen der Properties in Zephyr 11

Quelltextverzeichnis

3.1 Aufbau der Nodehierarchie im Devicetree 10
3.2 Devicetreeeintrag einer UART Node 11
3.3 Ausschnitt der Binding-Datei: spi-device [30] 12
3.4 Datenstruktur eines Gerätes in Zephyr [29] 14
3.5 Pinmultiplexing für LPSPI3 15
3.6 Definition von LPSPI3 im Devicetree 16
3.7 Programmausschnitt SPI DAC Steuerung 17

4.1 Programm zur Steuerung einer LED durch Register mittels Zephyr 21
4.2 Programm zur Steuerung einer LED mithilfe des NXP Treibers in der Zephyr Umgebung 22
4.3 Programm zur Steuerung einer LED durch die Zephyr API 24

5.1 Programmausschnitt der Interrupt Routine auf der CM7 Seite 36

6.1 Partitionierung i.MX RT1170 im Devicetree 42
6.2 Ausgabe MCUboot bei korrekter Implementierung 44
6.3 Ausgabe MCUboot bei korrekter Implementierung mit signierten Image 45
6.4 Overlay Datei zur USB Integration für den Update-Handler 45

Abkürzungsverzeichnis

API . . . . Application Programming Interface

ARM . . . . Advanced RISC Machines

CM4 . . . . Arm Cortex®-M4 Kern

CM7 . . . . Arm Cortex®-M7 Kern

CS . . . . Chip Select

DAC . . . . Digital Analog Converter

DFU . . . . Direct Firmware Update

IPC . . . . Interprocessor-Communication

IPM . . . . Interprozessor Mailbox

ISR . . . . Interrupt Service Routine

LCM . . . . Life Cycle Management

LPSPI . . . . Low Power SPI

MCU . . . . Micro Controller Unit

MISO . . . . Master Input Slave Output

MOSI . . . . Master Output Slave Input

MPU . . . . Microprozessor Unit

MU . . . . Messaging Unit

OS . . . . Operating System

RAM . . . . Random-Access Memory

RTOS . . . . Realtime Operating System

SCLK . . . . Serial Clock

SHM . . . . Shared Memory

SPI . . . . Serial Peripheral Interface

1 Einleitung

Während sich Betriebssysteme in der Desktopumgebung schon seit langen durchgesetzt haben, verzichten viele Gerätesteuerungen im eingebetteten Bereich noch immer auf ein solches System. Echtzeitanforderungen konnten in der Vergangenheit durch ein Betriebssystem nicht für eingebettete Geräte garantiert werden. Zudem waren viele Betriebssysteme für die meisten Mikrocontroller nicht geeignet, da Ressourcen, wie der Speicher oder die Rechenleistung nicht ausreichend vorhanden waren. Durch das Aufkommen neuer Mikrocontroller mit wesentlich mehr Performance und fortgeschrittenen Entwicklungen für sogenannte Echtzeitbetriebssysteme wie das Zephyr Realtime Operating System (RTOS), entfalten sich vollkommen neue Möglichkeiten in der hardwarenahen Programmierung. Neue abstrakte Konzepte die mit den Systemen einhergehen, können die Entwicklung von Software erleichtern. Dazu gehört unter anderem die Abstraktion von Hardwarekomponenten, eine Sammlung von Programmierschnittstellen (engl. Application Programming Interfaces (APIs)) und Konzepte wie Threads, die aus Desktop Betriebssystemen schon bekannt sind.

Zephyr ist Open Source und laut den GitHub *Stars*, welche einen groben Überblick über die Popularität eines Projektes geben, zur jetzigen Zeit eines der beliebtesten Echtzeitbetriebssysteme. [23] Es ist für viele Einsatzgebiete gedacht und kann sowohl auf Mikrocontrollern mit wenigen Kilobyte Flash-Speicher als auch mit geringen Speicherbedarf im Random-Access Memory (RAM) auskommen. Trotzdem bleibt es flexibel und legt einen großen Wert auf Sicherheit. Somit kann jeder Entwickler das Betriebssystem weiterentwickeln, um dieses auch in Zukunft mit neuer Hardware und Softwarekomponenten zu nutzen. Des Weiteren gibt es viele namhafte Hersteller wie NXP Semiconductors, Infineon oder auch Nordic Semiconductor, die das Projekt aktiv voranbringen. Damit ist Zephyr eine neue Lösung zur Nutzung eines RTOS auf einem eingebetteten System. Diese gilt es mit der Bare Metal Variante zu vergleichen und mögliche Einsatzgebiete herauszuarbeiten.

Eine Portierung von Software von einem System auf ein anderes kann viele Probleme mit sich bringen. Falls ein Umstieg von einer Bare Metal Umgebung auf das Zephyr RTOS geplant ist, gilt es einige Punkte zu berücksichtigen. Diese werden anhand der Integration von bestehenden Treibern in der vorliegenden Arbeit vorgestellt sowie mögliche Lösungsansätze genauer betrachtet.

Des Weiteren wird sich die vorliegende Arbeit mit Zephyr und den Mechanismen zur Steuerung von Hardwareeinheiten beschäftigen. Dabei werden unterschiedliche Ansätze vorgestellt und miteinander verglichen. Besondere Aufmerksamkeit gilt den Zephyr Konzepten zur Implementierung von Treibern. Anschließend werden Möglichkeiten zum Datenaustausch in einem Mehrkern-Mikrocontroller untersucht. Es folgt ein Vergleichen zwischen den bestehende Mechanismen mit denen von Zephyr. Viele Produkte erfordern heutzutage eine Aktualisierung der Software. Dies kann ebenfalls mit Zephyr abgedeckt werden. Als Vorbereitung für Projekte mit diesem System wird das Firmwareupdate vorgestellt werden.

2 Grundlagen

Zum Verständnis der vorliegenden Arbeit werden zu Beginn einige wichtige Themen, Methoden und Konzepte erläutert. Diese dienen als grundlegendes Wissen zum Nachvollziehen der vorgestellten Untersuchungen. Damit beginnend wird der Begriff von eingebetteten Systemen aufgeführt. Anschließend folgt das Konzept der Bare Metal Programmierung und dessen Bedeutung im Bereich der Mikrocontroller. Das Echtzeit Betriebssystem Zephyr ist eine Alternative zu der Bare Metal Programmierung und spielt eine große Rolle für diese Arbeit. Deshalb erfolgt auch hier noch eine nähere Beschreibung der wichtigsten Konzepte, die das Betriebssystem (engl. Operating System (OS)) mitbringt. Zuletzt wird die Prozessorfamilie *i.MX RT1170* von NXP Semiconductors vorgestellt.

2.1 Eingebettete Systeme

Bei den eingebetteten Systemen (engl. embedded Systems) handelt es sich um Geräte, die Informationen verarbeiten können. Diese verfügen meist über eine eingebaute Microprozessor Unit (MPU), welche von dem eigentlichen Gerät umschlossen ist. Angeschlossene Peripheriekomponenten wie Sensoren oder Aktoren können mit der Umgebung interagieren. Derartige Systeme unterliegen häufig starken Einschränkungen wie Platz-, Energie- und Speicherverbrauch. Zudem spielt oft die Kostenoptimierung eine große Rolle. Produkte, die zur Kategorie der eingebettete Systeme gehören, sind zum Beispiel technische Haushaltsgegenstände wie Toaster und Waschmaschinen. Weitere computergestützte Systeme im eingebetteten Bereich kommen zum Beispiel in Fahrzeugen, der Telekommunikation sowie in Industrieanlagen vor. Im Gegensatz zu klassischen PCs sind diese Geräte meist für eine Aufgabe spezialisiert. [12] [13] [5] [10]

2.2 Bare Metal Programmierung

Determinismus ist eine der wichtigsten Eigenschaften eines Programms für einen Mikrocontroller. Mit Determinismus ist gemeint, dass das System vorhersehbar reagiert und die Anforderungen erfüllt. Darunter fällt unter anderem die Echtzeitfähigkeit der Programme, welche garantiert, dass Deadlines und Antwortzeiten eingehalten werden. Für eine saubere Umsetzung der Echtzeitanforderungen müssen bestimmte Faktoren beachtet werden. Dazu zählt des Weiteren eine Programmiersprache, welche hoch performant, ressourcensparend und zuverlässig ist. In der hardwarenahen Programmierung wurde lange Zeit in Assembler programmiert, mittlerweile hat sich größtenteils C durchgesetzt. In dieser Sprache hat der Entwickler durch eine geringe Abstraktion volle Kontrolle über die Ansteuerung der Hardware. Trotzdem ist die Softwareerstellung wesentlich übersichtlicher und einfacher als mittels Assembler. Für objektorientierte Softwareerstellung wird häufig C++ verwendet. [4]

Um diesen Anforderungen gerecht zu werden, wurden Programme direkt für einen speziellen Mikrocontroller geschrieben. Durch den Zugriff auf die Register können die Komponenten angesteuert werden. Dabei kann nicht nur die Echtzeitfähigkeit gewährleistet, sondern auch

die Ressourcenbelastung auf ein Minimum reduziert werden. Damit das Programm dauerhaft aktiv ist, wird meistens eine Endlosschleife in der Main-Methode verwendet. Diese Art, Programme ohne ein zugrundeliegendes Softwaresystem zu erstellen, wird auch Bare Metal Programming, zu deutsch „bloßes Metall Programmierung“ genannt.

Eine besondere Rolle bei der Softwareentwicklung in einen Bare Metal System stellt die Programmierung von Treibern dar. In vielen Fällen werden diese in einer Bibliothek mit einheitlichen API Aufrufen für meist mehrere Prozessoren zusammengefasst. Dabei sind die Register von Peripheriekomponenten mit bestimmten Namen definiert. Diese werden durch Funktionen, welche bestimmte Steuerabläufe enthalten, ergänzt. Eine wichtige Grundlage dazu bilden die Handbücher der Chiphersteller. Diese enthalten die Beschreibungen zur korrekten Implementation von Steuerungssoftware für die dafür vorgesehenen Prozessoren. Für diese Arbeit wurde zur Programmierung des i.MX RT1170 Prozessors ebenfalls das Handbuch[1] zur Hilfe genommen. Mit dieser Herangehensweise kann eine Abstraktionsschicht geschaffen werden, auf der weitere Steuerungsmechanismen aufbauen. Gerade in Projekten die viel Steuerungssoftware enthalten und mehrere Produkte bedienen sollen, kann die Entwicklung beschleunigt und die Pflege der Software vereinfacht werden. Auf der anderen Seite können durch eine Eigenentwicklung dieser Treiber mit schlechter Konzeption, mangelhaften Umsetzung, fehlender Dokumentation oder unzureichender Softwaretests unerwartete Probleme auftreten. Trotzdem gibt es viele Anwendungsfälle wie zum Beispiel UART, Serial Peripheral Interface (SPI) oder auch einen ADC. Prozessor interne Funktionen wie die Interprocessor-Communication (IPC) für einen Mehrkernprozessor kann ebenfalls durch Treiber vereinfacht genutzt werden. Anmerkend dazu muss erwähnt werden, dass viele Systeme wie Echtzeitbetriebssysteme auf diese Treiber aufbauen, gerade wenn diese durch die Hardwarehersteller vorgegeben sind.

Programme, welche für eingebettete Systeme bestimmt sind, müssen meist auf einem Host-PC kompiliert werden. Der Host-PC ist in diesem Fall ein Computer, mit dem die Software entwickelt wird. In der Regel wird dafür ein PC mit dem Betriebssystem Linux oder Windows genutzt. Mit einem sogenannten Cross-Compiler kann dann ein Programm für ein anderes Zielsystem erstellt werden. Um beispielsweise ein Programm auf einer x86 Architektur für Advanced RISC Machines (ARM) zu bauen, wird der *gcc-arm-linux-gnueabihf* Compiler benötigt.

2.3 Das Echtzeitbetriebssystem Zephyr

Große und komplexe Programme können in einer Bare Metal Umgebung schnell unstrukturiert und unübersichtlich werden. Hierfür eignet sich der Einsatz eines Echtzeitbetriebssystems wie Zephyr[2], welches für den Bereich der eingebetteten Systeme optimiert wurde. So sind die minimalen Anforderungen zur Ausführung von Applikationen mit Zephyr rund 8 KB Flash-Speicher und 5 KB RAM. Ein weiterer Vorteil eines RTOS liegt in der quasi Parallelisierung von Softwarebausteinen durch mehrere Prozesse oder Threads. Diese werden in der Sektion 2.3.2 genauer vorgestellt. Ein modularer Aufbau sorgt dafür, dass das Betriebssystem für weitere Anwendungsbereiche skaliert werden kann. In vielen Beispielen reichen

[1] i.MX RT1170 Handbuch: https://www.nxp.com/webapp/Download?colCode=IMXRT1170RM
[2] Offizielle Zephyr Webseite: `https://zephyrproject.org/`

Konfigurationseinträge, um neue Bibliotheken in das Projekt einzubinden. Zudem bringt das RTOS viele Konzepte und Tools mit, welche sich Entwickler zur Applikationsentwicklung zu nutze machen können. Die Wichtigsten davon werden folgend angerissen. [23]

2.3.1 Architektur

Das Zephyr Betriebssystem besteht aus mehreren Ebenen. Diese sind systematisch aufgebaut und enthalten spezifische Elemente und Features. Wie der Grafik in Abbildung 2.1 zu entnehmen ist, befindet sich auf der obersten Schicht die Applikation. Diese wird für ein Produkt entwickelt und dient als Softwareschnittstelle zu den Kunden. Mit Hilfe der High-Level API können Entwickler bei der Softwareerstellung auf Zephyr-Schnittstellen der Konnektivität zugreifen. Diese sind stark an den Linux Standards wie POSIX angelehnt. Die einzelnen Protokolle wie Bluetooth-Low-Energy, Ethernet, CAN, usw. sind in diesem Bereich angeordnet. Anschließend folgen die Low-Level APIs. Zu diesen gehören grundlegende Betriebssystem-Features wie UART, GPIO, Inter-Prozessor Kommunikation und Timing. Der darunterliegende Kernel enthält weitere wichtige Aufgaben. So zum Beispiel das Verwalten von Threads oder das Speichermanagement. Die herstellerspezifischen Prozessoren und Sensoren werden in der Hardware-Abstraktionsschicht zusammengefasst und für darüber liegenden Schichten einheitlich bereitgestellt. Zuletzt folgt die Schicht der Hardwareplattformen. Hierzu zählen sämtliche Gerätekomponenten, welche durch Zephyr unterstützt werden. Dazu gehört unter anderem der i.MX RT1170 Prozessor von NXP. [23]

2.3.2 Wichtige Betriebssystemkonzepte in Zephyr

Dieses Kapitel soll aufzeigen, welche Konzepte Zephyr mit sich bringt und wie diese in dem RTOS umgesetzt werden. Dabei ist zu beachten, dass alle folgenden Schemen nicht nur auf das Zephyr RTOS zutreffen, sondern auch in vielen anderen Betriebssystemen zum Einsatz kommen. Teilweise sind jedoch Zephyr eigene Implementierungsansätze enthalten, welche kurz angerissen werden.

Threads

Eines der wichtigsten Konzepte in einem Betriebssystem sind Threads. Mit diesen können Aufgaben oder Funktionen unabhängig und simultan voneinander ausgeführt werden. Entweder geschieht dies durch einen Zeitmultiplex auf einem Kern oder parallel auf mehreren Kernen. Zudem kann die Software dadurch abstrakter gestaltet und daraus folgend verständlicher aufgebaut werden. Bei der Erstellung eines Threads wird diesem ein eigener Stack, Befehlszähler und Registersatz zugeordnet. Andere Ressourcen müssen untereinander aufgeteilt werden. Dazu zählt unter anderem der Speicher oder Peripherieeinheiten.

Mit dem Zephyr RTOS ist es möglich, neben weiteren Echtzeitbetriebssystemen, Threads in einer Mikroprozessorumgebung zu nutzen. Dabei ist der Thread ein Kernel-Objekt, welcher beliebig oft erstellt werden kann. Jedoch wird die Anzahl durch den vorhandenen Speicher begrenzt. Besonders vorteilhaft ist die Nutzung bei unabhängigen Aufgaben oder für komplexe oder zeitintensive Interrupt Service Routinen. Um die Zeitanforderungen der Aufgaben zu gewährleisten, können Prioritäten vergeben werden. So sind Zephyr-Threads in zwei Gruppen

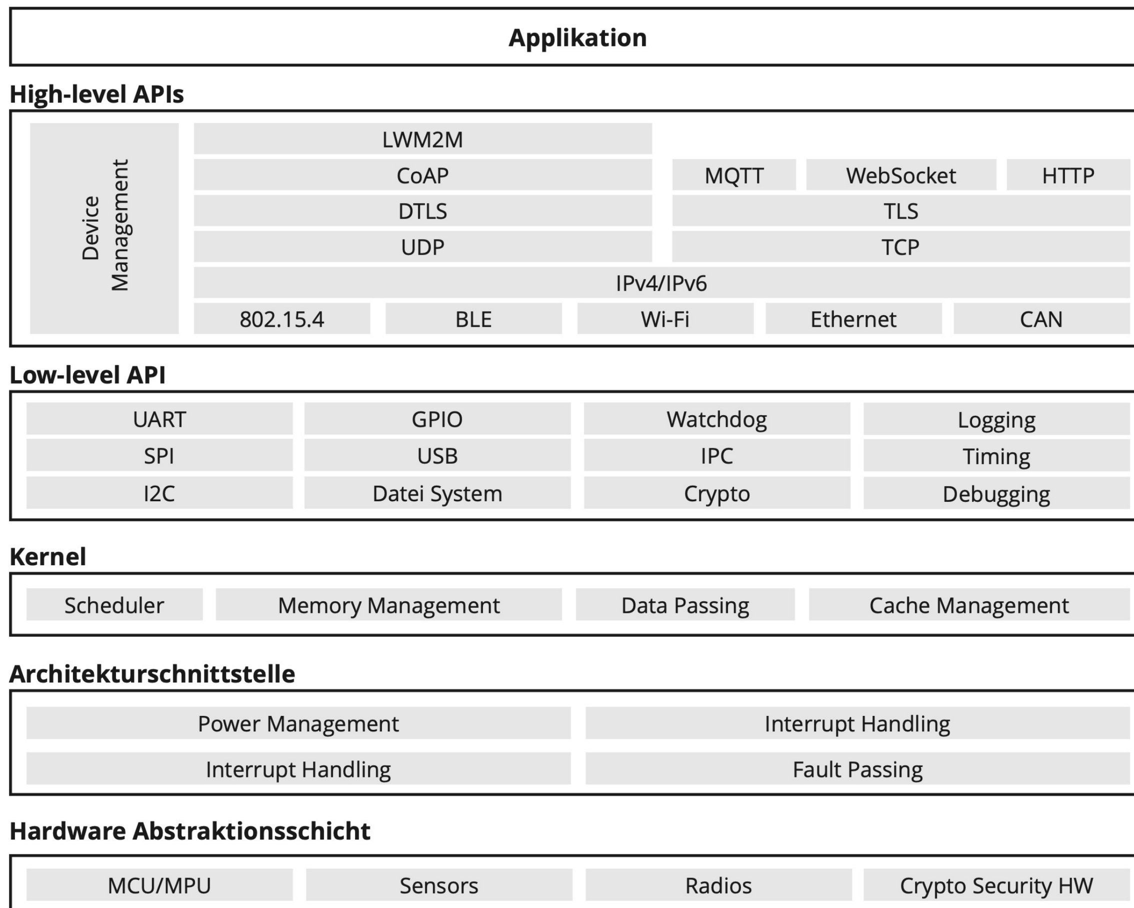


Abbildung 2.1: Softwarearchitektur des Zephyr RTOS [eigene Abbildung]

unterteilt. Zum einen gibt es die *preemptiven* Threads. Diese werden durch den Scheduler des Betriebssystems unterbrochen oder freigegeben. Bei der Erstellung der Threads muss eine Priorität, ein Zahlenwert beginnend bei 0 und einer eigens konfigurierten positiven Ganzzahl, festgelegt werden. Je größer die Zahl ist, desto geringer ist die Priorität des Threads. Zum anderen gibt es neben den preemptiven Threads noch die *kooperativen* Threads. Diese haben eine negative Ganzzahl als Priorität, wobei die negativsten Zahlen priorisiert werden. Ein aktiver kooperativer Thread kann durch den Scheduler nicht unterbrochen werden und muss sich selber wieder freigeben. [22]

Die folgende Abbildung 2.2 veranschaulicht die Zustände der Threads innerhalb von Zephyr. Durch den entsprechenden Systemaufruf kann ein Thread erstellt werden. Dieser ist anschließend *Bereit* und kann vom Scheduler Rechenzeit zugewiesen bekommen. Damit würde er *Aktiv* werden. Ein *Aktiv* laufender Thread kann ebenfalls durch den Scheduler unterbrochen werden und somit in den Zustand *Ausgesetzt* gelangen. Des Weiteren kann der Thread in den Zustand *Wartend* übergehen, wenn zum Beispiel auf Daten gewartet wird. Durch einen Interrupt kann ein aktiver Thread ebenfalls unterbrochen werden und wieder in den Zustand *Bereit* eintreten. Sobald der Thread seine Aufgabe erledigt hat, wird dieser durch das Betriebssystem beendet. Des Weiteren ist es möglich, dass dieser durch den entsprechenden Systemaufruf von außerhalb *Beendet* wird. [22]

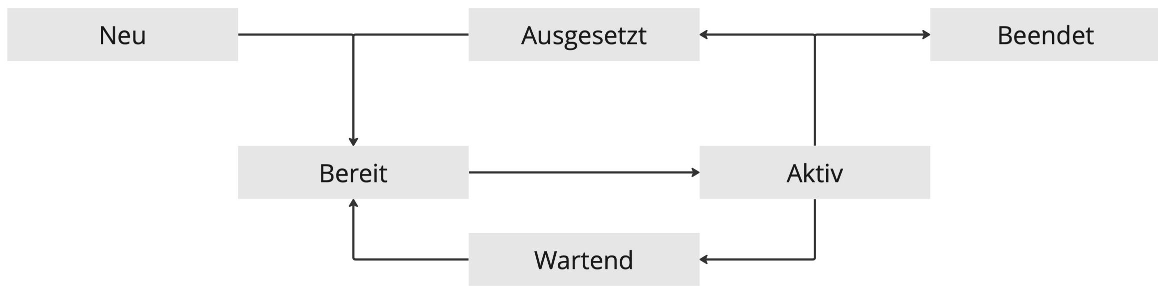


Abbildung 2.2: Zephyr Threadzustände [eigene Abbildung]

Thread-Synchronisation durch Semaphore und Mutexe

Wie in Abschnitt 2.3.2 unter Threads schon beschrieben wurde, gibt es Ressourcen, welche zwischen den Threads aufgeteilt werden müssen. Um den exklusiven Zugriff auf diese meist begrenzten Ressourcen zu gewährleisten, können in Zephyr Synchronisationsmechanismen wie Semaphore oder Mutexe genutzt werden.

Ein *Semaphore* besteht im Wesentlichen aus einem Zähler (engl. Counter) und einem Limit. Der Zähler gibt an, wie viele Ressourcen aktuell verfügbar sind, während das Limit zeigt, wie viele Ressourcen insgesamt verfügbar sein können. Der Zähler wird verwendet, um den Zustand des Semaphors zu verfolgen und zu aktualisieren. [18]

Ein *Mutex* wird oft als binärer Semaphore betrachtet, da er nur zwei Zustände haben kann. Entweder ist dieser gesperrt (engl. locked) oder entsperrt (engl. unlocked). [20]

Insgesamt sind Mutexe und Semaphoren wichtige Werkzeuge für die Synchronisation und den geschützten Zugriff auf Ressourcen in einer Multi-Thread-Umgebung. Der Hauptunterschied besteht darin, dass ein Mutex den exklusiven Zugriff auf eine Ressource gewährleistet, während ein Semaphore den Zugriff auf eine begrenzte Anzahl von Ressourcen steuert.

Interrupts

Eine Interrupt Service Routine (ISR) ist eine Funktion, welche durch entweder ein Hardware- oder Softwareereignis aufgerufen wird. Diese hat normalerweise eine höhere Priorisierung als der aktuelle Thread und wird somit direkt aufgerufen. Diese sollte möglichst wenig Programmlogik enthalten, damit das Hauptprogramm nicht länger als notwendig blockiert wird.

Das Zephyr RTOS verwaltet automatisch die Interrupts sowie ISRs und bietet eine vereinfachte Schnittstelle zur Erstellung dieser. Dadurch kann die Entwicklung von Systemen, welche Interrupts nutzen beschleunigt und vereinfacht werden. [17]

Symmetrische Mehrprozessorumgebung

Eine symmetrische Mehrprozessorumgebung ist ein System, welches einen Prozessor mit mehreren Kernen besitzt. Zudem muss das Betriebssystem in der Lage sein, diese zu nutzen, um Threads parallel auszuführen. Beispielsweise können auf einem Dualcore Chip zwei

Threads simultan ausgeführt werden. Die Abbildung 2.3 veranschaulicht den Ablauf eines Dualcore-Prozessors, welcher nach dem Start einen Thread auf der jeweiligen CPU ausführt. Damit sind insgesamt zwei Threads gleichzeitig aktiv. [21] [6]

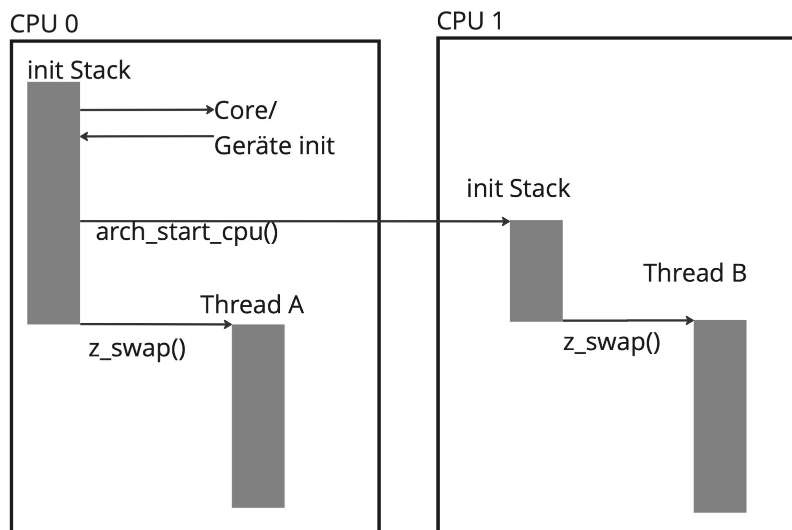


Abbildung 2.3: SMP Initialisierungsprozess einer Dualcore CPU mittels Zephyr [eigene Abbildung]

2.3.3 Verwaltungs- und Erstellungskonzepte

Zephyr verwendet ein eigens entwickeltes Verwaltungstool mit dem Namen West. Damit können mehrere Git-Repositories in einem einzigen Verzeichnis mit nur einer Datei verwaltet werden. Diese Manifest Datei ist in dem Projekt unter dem Namen west.yml zu finden. Eigene Repositories können in die Manifest Datei mit aufgenommen und verwaltet werden. Zudem ist es möglich, mittels des West Meta Tools Programme zu kompilieren, flashen und debuggen. [28]

2.4 Vorstellung des Prozessors i.MX RT1170

Die von NXP entwickelte Prozessorserie i.MX RT1170[3] ist eine ARM basierende MPU. Diese besteht aus zwei Kernen, welche zum einen der leistungsstarke Arm Cortex®-M7 Kern (CM7) mit bis zu 1 GHz Taktrate ist. Mit dem Arm Cortex®-M4 Kern (CM4) ist zum anderen ein energiesparender weiterer Kern mit bis zu 400 MHz Taktrate implementiert. Der i.MX RT1170 verfügt über insgesamt 2 MB On-Chip-RAM, der teilweise selbst konfiguriert werden kann. Es sind zudem einige Schnittstellen zum Anschluss von Peripheriegeräten wie WLAN, Bluetooth®, GPS, Displays und Kamerasensoren vorhanden. [11] Die angefertigten Programme und Programmausschnitte für die vorliegende Arbeit wurden für spezielle Platinen mit dem vorgestellten Prozessor entwickelt. Die Nutzung von anderen Prozessoren bedarf möglicherweise weiteren Anpassungen.

[3]Produktseite des i.MX RT1170: `https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1170-first-ghz-crossover-mcu-with-arm-cortex-m7-and-cortex-m4-cores:i.MX-RT1170`

3 Zephyr Konzepte zur Hardwaresteuerung

3.1 Devicetree zur Abstraktion der Hardware

Im Zephyr-Projekt ist der Devicetree, in vielen APIs abgekürzt mit *dt*, ein wichtiger Bestandteil des Systems. Dieser ist eine strukturierte, hierarchisch aufgebaute Datenbeschreibung, die die Hardware-Konfiguration eines eingebetteten Systems definiert. Es handelt sich um eine plattformunabhängige, geräteorientierte Darstellung, die die Hardwarekomponenten und deren Konfiguration beschreibt. [19]

Die in dem Devicetree enthaltene Abstraktionsebene ermöglicht es, denselben Zephyr Quellcode auf verschiedenen Plattformen laufen zu lassen, ohne diese zu ändern. Durch die enthaltenen Informationen über Prozessoren, Speicher, Peripheriegeräte und weitere Hardwarekomponenten wird eine große Plattformunabhängigkeit gewährleistet. Damit eine maximale Performance zur Laufzeit des Prozessors garantiert werden kann, wird der Devicetree während der Kompilierungszeit in das Zephyr-Image integriert.

Ein Projekt, welches das Konzept des Devicetrees nutzt, enthält meist mehrere Dateien, welche den finalen Devicetree erstellen. So wird für eine Applikation, die für eine bestimmte Platine gebaut werden soll, eine sogenannte Board-Beschreibung benötigt. Die Dateien in diesem Ordner beschreiben und konfigurieren die spezielle Hardware. Diese sind mit der Dateiendung *.dts* und *.dtsi* gekennzeichnet. Dabei steht dts für „devicetree source“ und dtsi für „devicetree source include“. Die .dtsi Dateien werden beim bauen eines Projektes in die .dts Datei eingebunden.[19] Der entsprechende West-Befehl zum Kompilieren einer Applikation für eine entsprechende Platine lautet:

```
$ west build -b  board
```

Dabei ist *board* die entsprechend konfigurierte .dts Datei. Wird für ein Projekt eine erweiterte Hardwarekonfiguration oder Änderung der bestehenden Beschreibung benötigt, kann eine *Overlay*-Datei in dem Projektpfad ergänzt werden. Wie der Name *Overlay* (dt. Überlagerung) schon vermuten lässt, werden Hardwarekonfigurationen ergänzt oder bestehende Einträge verändert. Wird die Overlay Datei *app.overlay* genannt, verwendet Zephyr diese bei jeder Kompilierung für eine Architektur automatisch mit. Wenn der Name *Boardname.overlay* verwendet wird, und diese Datei sich in dem lokalen *boards* Ordner befindet, wird diese nur beim Bauen für die entsprechende Platine verwendet. Anderenfalls können eine oder mehrere Overlay Datein mit der CMake Variablen *DTC_OVERLAY_FILE* beim Compilieren der Applikation mittels West hinzugefügt werden. Beispielsweise kann ein Befehl folgendermaßen aussehen:

```
$ west build -b mimxrt1170_evk_cm7 -- -DDTC_OVERLAY_FILE=sample.overlay
```

3.1.1 Syntax und Struktur

Ein vollständiger Devicetree besteht meist aus mehreren Parametern. Dazu zählen die *Nodes*, *Properties*, *Unit Adressen* sowie die speziellen Nodes *Aliase* und *Chosen*. Folgend werden die zuvor aufgezählten Strukturelemente des Devicetrees genauer erläutert.

Nodes

Im Kontext des Devicetrees ist eine Node eine grundlegende Einheit zur Beschreibung einer Hardwarekomponente oder eines Subsystems in einem eingebetteten System. Eine Node repräsentiert eine einzelne Komponente wie einen Prozessor, ein Peripheriegerät, einen Speicher oder eine Schnittstelle. Jede Node enthält Konfigurationsinformationen und Eigenschaften die benötigt werden, um die entsprechende Hardwarekomponente im System zu initialisieren und zu konfigurieren.

Der Aufbau der Nodes erfolgt hierarchisch. So beginnt der vollständige Devicetree mit der sogenannten *root-Node*, die durch ein */* dargestellt wird. Anschließend folgen *a-nodes*, welche Kindelemente (engl. childnodes) der root-Node sind. Zuletzt gibt es noch die *a-sub-nodes*. Diese sind childnodes der a-nodes. In dem Beispiel des Quelltextes 3.1 wird der grobe Aufbau veranschaulicht.

```
/ {
 a-node {
  subnode_nodelabel: a-sub-node {
   foo = <3>;
  };
 };
};
```

Quelltext 3.1: Aufbau der Nodehierarchie im Devicetree

Properties

Devicetree Properties sind spezifische Attribute, welche aus einem Schlüssel und einem Wert bestehen. Attribute sind in den Nodes enthalten und beschreiben oder konfigurieren diese. Die wichtigsten Properties, welche häufig Anwendung finden, sind in der folgenden Tabelle 3.1 aufgeführt.

Name	Beschreibung
compatible	Name der Hardware, welche durch die Node repräsentiert wird
reg	Information zur Adressierung der Komponente
status	Beschreibt, ob die Node aktiviert ist oder nicht
interrupts	Informationen zu Interrupts, welche durch das Gerät ausgelöst werden

Tabelle 3.1: ausgewählte Properties in Zephyr

Jedes Attribut benötigt einen speziellen Datentyp, welcher einen Wert enthält. Die wichtigsten sind in der Tabelle 3.2 erläutert. Das Devicetree-Besipiel 3.2 veranschaulicht den Einsatz der Properties.

Name	Beschreibung	Beispiel
string	in doppelten Hochkommas	name = "hello, world";
int	in spitzen Klammern	wert = <2>;
boolean	für Wahr kein Wert, für falsch /delete-property/	wahre_eigenschaft;
array	in spitzen Klammern mit Leerzeichen zwischen den Werten	werte = <0x143 14 0>
uint8-array	in Hexadezimalwerten ohne 0x zwischen eckigen Klammern	array = [01 1b 2f];
string-array	Separiert durch Kommas	strings = "hello", "world", "!";
phandle	in spitzen Klammern	node = <&my_node>;
phandles	in spitzen Klammern mit Leerzeichen dazwischen	nodes = <&my_node0 &node1 &node2>

Tabelle 3.2: Grundlegende Datentypen der Properties in Zephyr

```
uart@40020000 {
 compatible = "vendor,serial-uart";
 reg = <0x40020000 0x1000>;
 interrupts = <0 4 0>;
 current-speed = <115200>;
};
```

Quelltext 3.2: Devicetreeeintrag einer UART Node

Unit Adressen

Der Begriff Unit Adressen bezeichnet die numerischen Werte, die unter anderen in der *reg* Property einer Node verwendet werden, um die Adresse und die Größe einer Hardwareeinheit oder einer Peripheriekomponente zu definieren.

Die *reg* Property besteht normalerweise aus einem Array von Ganzzahlen, die die Registeradressen und Größen in einem eingebetteten System darstellen. Die Anzahl der Elemente im Array und deren Bedeutung können je nach spezifischer Hardware variieren. In dem Devicetree Beispiel 3.2 ist unter dem Propertyeintrag *reg* der Einsatz von einer Unit Adresse zu erkennen.

Alias und Chosen Nodes

Alias-Nodes werden verwendet, um alternative Namen für bestimmte Hardwarekomponenten oder Peripheriegeräte anzugeben. Diese Aliasse können in anderen Teilen des Devicetrees oder in der Konfigurationsdatei verwendet werden, um auf die Hardwarekomponenten zu verweisen, ohne die ursprüngliche vollständige Node-Bezeichnung zu verwenden.

Die *Chosen*-Nodes werden im Devicetree verwendet, um bestimmte Einstellungen, Optionen oder Konfigurationen festzulegen, die von dem Bootloader oder vom Betriebssystem während des Startvorgangs berücksichtigt werden sollen. Diese Nodes enthalten in der Regel Eigenschaften, die den Startvorgang beeinflussen oder spezifische Konfigurationen für das Betriebssystem.

3.1.2 Bindings

Durch die Devicetree *Bindings* wird die Struktur von Nodes für ein spezielles Gerät definiert. So enthält die Devicetree YAML Datei spezifische vordefinierte Merkmale einer Node. Die Property *compatible* erlaubt es die Binding Datei einzubinden. Wenn in der Binding-Datei eine Eigenschaft mit dem Keyword *required: true* enthalten ist, muss diese unbedingt in der Node des Devicetrees definiert werden. Einfacher ausgedrückt sind Bindings eine Art Blaupause zu Beschreibung von Treibern oder Peripheriekomponenten. Viele Hersteller stellen *Devicetree Bindings* für ihre Produkte bereit. So können Entwickler diese leichter integrieren und nutzen.

Das Quelltextbeispiel 3.3 veranschaulicht den groben Aufbau und die Konfiguration einer Binding-Datei am Beispiel des SPI Protokolls in Zephyr. In Zeile 1 des Beispiels werden weitere Binding Dateien eingebunden. Der Eintrag *on-bus* wird in dem Fall genutzt, um zu zeigen, dass der SPI Bus verwendet wird. Danach folgen die Eigenschaften, wie der *reg-*, *spi-max-frequency* Eintrag. Diese müssen beim benutzen des Bindigs in dem Devicetree verwendet werden, da diese den Parameter *required: true* haben.

```
1  include: [base.yaml, power.yaml]
2
3  on-bus: spi
4
5  properties:
6  reg:
7  required: true
8  spi-max-frequency:
9  type: int
10 required: true
11 description: Maximum clock frequency of devices SPI interface in Hz
```

Quelltext 3.3: Ausschnitt der Binding-Datei: spi-device [30]

Zusammengefasst kann gesagt werden, dass die Verwendung des Device Trees die Portabilität von Zephyr auf verschiedene Hardwareplattformen erleichtert und die Wiederverwendung von Treibercode fördert, indem Gerätespezifikationen von der eigentlichen Anwendungslogik getrennt werden. Dadurch wird die Entwicklung von Zephyr-basierten Anwendungen für eine breite Palette von eingebetteten Systemen erleichtert.

3.1.3 Das Konfigurationssystem Kconfig

Genau wie in dem Linux Kernel besitzt Zephyr ein ähnliches Konfigurationssystem, welches auch Kconfig genannt wird. Dieses ermöglicht es die Subsysteme zum Zeitpunkt der Erstellung zu konfigurieren. Dabei kann Software an plattformspezifische Anforderungen angepasst werden, ohne den Quellcode aktiv zu verändern. Die Einträge zur Konfiguration können in den Kconfig-Dateien festgelegt werden. Dabei ist eine Gruppierung zu beachten, welche sämtliche Optionen übersichtlich darstellt.

Während des Build-Prozesses entsteht aus der Kconfig-Datei eine *autoconf.h* Datei. Diese enthält Makros, welche bei der Verwendung in dem Programm mit kompiliert werden. [16]

3.1.4 Projektkonfiguration mittels Kconfig

Die Projektkonfiguration erfolgt standardmäßig in der *prj.conf* Datei eines Projektordners. Diese überschreibt Standardwerte der vordefinierten Kconfig-Optionen. Dadurch kann zum Beispiel das Verhalten von Treibern auf die Applikation angepasst werden. Größtenteils enthält die Datei Aktivierungen von bestimmten Funktionen oder Makros, welche anschließend in dem Projekt verwendet werden können. [15]

3.2 Zephyr APIs

Die Zephyr APIs sind eine Sammlung von Programmierschnittstellen, die von Zephyr für eingebettete Systeme bereitgestellt werden. Diese APIs ermöglichen Entwicklern die Interaktion mit verschiedenen Systemkomponenten. Nachfolgend wird der Aufbau und die Funktionsweise erläutert.

3.2.1 Devicetree APIs

Damit auf die Einträge des Devicetrees zugegriffen werden kann, muss die Makro-basierte Devicetree API genutzt werden. Zephyr bietet unterschiedliche Zugriffsmöglichkeiten an, um die zugrundeliegende Hardware mithilfe der Devicetree API zu steuern. Diese werden in der Zephyr Dokumentation in die *Generic APIs*, *Instance-based APIs* und *Hardware specific APIs* unterteilt. Da die Generic API die meist verwendete von allen ist, wird diese nachfolgend näher betrachtet.

Generic APIs

Durch die Generic APIs können verschiedene Aktionen mit dem Devicetree ausgeführt werden. Unter anderem kann auf bestimmte Nodes und deren Eigenschaften zugegriffen werden. Damit ein Gerät, welches im Devicetree definiert ist, angesteuert werden kann, benötigt es zuvor einen sogenannten *Identifier*. Für die *root-Node* ist dieser zum Beispiel *DT_ROOT*. Mithilfe der Makros *DT_PATH*, *DT_NODELABEL*, *DT_ALIAS* und *DT_INST* kann ein Identifier für eine spezielle Node erstellt werden. Die wichtigsten Makros, um auf die Properties zuzugreifen, sind *DT_PROP* und *DT_PROP_LEN*. Eine genauere Beschreibung dieser, mit den entsprechenden Parametern, ist ebenfalls in der Dokumentation von Zephyr zu finden. Beispiele zur Nutzung dieser Makros sind folgend aufgeführt. Diese beziehen sich auf den Devicetree-Ausschnitt des Quelltextes 3.1 und fragen den Wert zur Property *foo* ab. Der Rückgabewert von den zwei unten aufgeführten Beispielen ist der Integer Wert *3*.

```
DT_PROP(DT_PATH(a-node, subnode_nodelabel), foo)
```

```
DT_PROP(DT_NODELABEL(subnode_nodelabel), foo)
```

3.2.2 Gerätetreibermodell

Nachdem geklärt wurde, wie auf die im Devicetree definierten Einträge vom Quellcode aus zugegriffen werden kann, folgt nun die Steuerung und Konfiguration dieser mithilfe der in Zephyr integrierten Treiber. Durch das *Device Driver Model* werden in Zephyr Bibliotheken für die Boards bereitgestellt. Für jede Art von Gerätesteuerung, wie z.B einer ADC Steuerung, UART oder SPI Schnittstelle, gibt es eine bestimmte API. Diese sollten falls möglich Interrupt basiert implementiert sein. Low Level Aufrufe wie I2C oder SPI sind jedoch synchron gedacht und damit blockierend.

In Zephyr gibt es eine einheitliche Datenstruktur mit dem Namen *device*. Diese enthält gerätebeschreibende Einträge, wie zum Beispiel den Namen des Devices, Konfigurationseinstellungen, die zugehörige API Struktur sowie ein Datenfeld, welches dynamische Werte im RAM speichert. Der genaue Aufbau ist in dem Quelltext 3.4 veranschaulicht.

```
1 struct device {
2  const char *name;
3  const void *config;
4  const void *api;
5  void * const data;
6 };
```

Quelltext 3.4: Datenstruktur eines Gerätes in Zephyr [29]

Diese Struktur kann durch das Makro *DEVICE_DEFINE* erstellt werden. Ist die Struktur für ein bestimmtes Gerät schon vorhanden, kann der Name einer bestimmten Node mit *DEVICE_DT_-NAME* erhalten werden. Bei der Verwendung von *DEVICE_GET* wird ein Pointer auf die ganze Struktur zurückgegeben. Gibt es für ein entsprechendes Gerät noch keine Deklaration, kann dieses mit dem Makro *DEVICE_DECLARE* erstellt werden.

Wie schon erläutert gibt es für viele Schnittstellen eine bestehende API. Anhand der SPI API soll gezeigt werden, wie diese implementiert und zur Steuerung eines externen DAC genutzt wird.

3.2.3 SPI API zur Steuerung eines DAC's

Das Serial Peripheral Interface (dt. Serielle Peripherieschnittstelle), ermöglicht die Kommunikation zwischen mehreren unterschiedlichen Mikrocontrollern oder SPI-Geräten in synchroner Art und Weise. Für SPI werden unterschiedliche Verbindungen zwischen den Komponenten benötigt. So wird zwischen einer Datenleitung und einer Anbindung für die Taktfrequenz (engl. Clock) getrennt. SPI funktioniert im Master-Slave Prinzip.

Die Abbildung 3.1 veranschaulicht einen SPI Bus mit einem Die Abbildung 3.1 veranschaulicht einen SPI Bus mit einem Master und zwei Slaves. Dabei spiegelt *Slave1* eine weitere Micro Controller Unit (MCU) und *Slave2* ein Peripheriegerät, wie zum Beispiel ein externer DAC, wieder. Dabei ist die schwarz markierte Verbindung das Taktsignal (engl. Serial Clock) zwischen den Busteilnehmern. Das Taktsignal geht vom Master aus und bestimmt wie schnell Daten

ausgetauscht werden können. Die darunter liegenden, lila dargestellten Datenleitungen sind zum eigentlichen Austausch von Daten gedacht. Zwischen dem Master und dem *Slave 1* ist diese Verbindung zwischen Master Input Slave Output (MISO) und Master Output Slave Input (MOSI). Im Gegensatz dazu ist *Slave 2* über die Anschlüsse *DIN* sowie *DOUT* an die Datenleitungen angeschlossen. Zuletzt folgen die Verbindungen für das sogenannte Chip Select (CS). Dieses wird durch die grün gefärbten Verbindungen dargestellt. Sobald Daten zwischen dem Master und den Busteilnehmern ausgetauscht werden sollen, muss das CS vom Master gegen Masse gezogen werden. Ansonsten ist das CS High Aktiv. Mit jedem Takt wird ein Bit vom Master zum Slave und eines vom Slave zum Master transportiert.

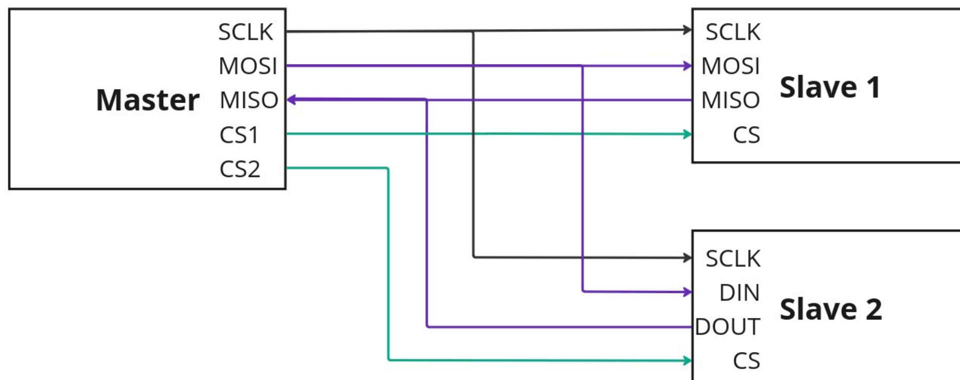


Abbildung 3.1: SPI Verbindung zwischen Master und zwei Slaves [eigene Abbildung]

Viele Prozessoren, wie beispielsweise die i.MX RT1170 Baureihe von NXP, besitzen Schnittstellen zur Kommunikation über SPI. Bei der Verwendung eines i.MX RT1170 Prozessors kann die NXP eigene SPI Schnittstelle Low Power SPI (LPSPI) verwendet werden. Ausschnitte aus einem eigens entwickelten Programm wurden verwendet, um das Prinzip und den Ablauf zu veranschaulichen.

Konfiguration der Pins für LPSPI

Da LPSPI standardmäßig nicht definiert ist, muss der Devicetree um diese Komponente ergänzt werden. Wird LPSPI nur für ein Projekt benutzt, bietet es sich an, dieses in einer Overlay Datei zu implementieren. Folgender Quelltext 3.5 ist ein Auszug aus der Overlay Datei zur Konfiguration der entsprechenden Pins. Dabei werden die Pins für Serial Clock (SCLK), MOSI und MISO als Array der Property *pinmux* übergeben. Zu beachten ist, dass kein CS mit in *pinmux* eingetragen wird. Dieses wird im späteren Verlauf einem anderen Eintrag zugewiesen. Sämtliche Konfigurationen werden unter dem Namen *pinmux_lpspi3_default* gespeichert.

```
1  &pinctrl {
2   pinmux_lpspi3_default: pinmux_lpspi3_default {
3    group0 {
4     pinmux =
5     <&iomuxc_gpio_emc_b2_04_lpspi3_sck>,
6     <&iomuxc_gpio_emc_b2_07_lpspi3_sdi>,
7     <&iomuxc_gpio_disp_b1_06_lpspi3_sdo>;
8     bias-pull-up;
9     slew-rate = "fast";
```

```
    };
  };
};
```

Quelltext 3.5: Pinmultiplexing für LPSPI3

Anschließend folgt die Zuweisung zu der *lpspi3* Node im Quelltext 3.6 unter der *pinctrl-0* Property. Zwingend notwendig ist die Statuszuweisung mit dem String „okay“, ansonsten bleibt *lpspi3* inaktiv. In Zeile 4 des Quelltextes 3.6 ist zu erkennen, dass ein CS Pin der Property *cs-gpios* zugewiesen ist. Weitere spezifische Einstellungen sind zum Beispiel *transfer-delay* und *sck-pcs-delay*. Als letzter Eintrag ist ein Slave-Gerät mit dem Namen *spidev* angegeben. Dieses hat die Unit Adresse 0 und ist somit mit dem ersten Eintrag der *cs-gpios* verknüpft. Das heißt, wenn GPIO8 Pin 15 von High auf Low gewechselt wird, kann der Datenaustausch mit dem *spidev* erfolgen. Dabei setzt Zephyr automatisch das CS sobald eine Übertragung durch die API ausgelöst wird. Damit das *spidev* als Slave definiert wird, muss der *compatible* Eintrag „vnd,spi-device“verwendet werden. Falls ein Produkt mit zugehörigem Binding-Eintrag als Slave über SPI angeschlossen wird, ist dieser Eintrag, anstatt „vnd,spi-device“, zu nehmen.

```
&lpspi3 {
  status = "okay";
  transfer-delay = <250>;
  cs-gpios = <&gpio8 15 GPIO_ACTIVE_LOW>;
  pinctrl-0 = <&pinmux_lpspi3_default>;
  pinctrl-names = "default";
  transfer-delay = <4000>;
  sck-pcs-delay = <2000>;
  pcs-sck-delay = <2000>;
  clock-frequency = <1000000>;

  spidev: spi-dev@0 {
    compatible = "vnd,spi-device";
    reg = <0x0>;
    spi-max-frequency = <1000000>;
  };
};
```

Quelltext 3.6: Definition von LPSPI3 im Devicetree

Steuerung mithilfe der SPI API

Nachdem *lpspi3* im Devicetree konfiguriert und aktiviert wurde, kann diese nun mit der SPI API verwendet werden. Der Programmausschnitt des Quelltextes 3.7 veranschaulicht die Vorgehensweise zur Steuerung eines externen DAC[4] mithilfe von SPI. So wird in Zeile 1 mit dem Makro *SPI_DT_SPEC_GET* auf die im Devicetree definierte Node *lpspi3* zugegriffen. Des Weiteren wurden folgende Operationen festgelegt:

- *SPI_OP_MODE_MASTER* setzt den Operationsmodus auf Masterbetrieb.

[4]Handbuch DAC8562: [9]

- *SPI_TRANSFER_MSB* überträgt das Most Significant Bit (MSB) zuerst.
- *SPI_WORD_SET(word_size)* gibt die Größe eines zu übertragenden Datenworts an.
- *SPI_LINES_SINGLE* legt die Übertragung über eine Datenleitung fest.
- *SPI_MODE_CPOL* legt fest, ob die SCLK-Leitung im Ruhezustand auf einen hohen oder niedrigen Pegel gehalten wird.

Mit dem letzten Parameter kann eine Verzögerung (engl. delay) an das Makro übergeben werden, welches in diesem Beispiel *2* beträgt. Der Rückgabewert ist eine Struktur bestehend aus einer Device-Struktur, welche in dem Quelltext 3.4 dargelegt wurde sowie einem Konfigurationseintrag mit dem Name *spi_config*. Dieser enthält die Frequenz des Busses, Flags, die Anzahl der Slaves und eine CS Struktur. Die Prüfung zur erfolgreichen Initialisierung erfolgt in Zeile 3. Anschließend wird ein Buffer definiert, welcher Werte, die an den Slave gesendet werden sollen, enthält. In Zeile 8 wird der Buffer angelegt und in Zeile 9 ist das Array mit den Daten zu erkennen. Die Struktur *spi_buf_set* hingegen speichert mehrere *spi_buf* Strukturen. In Zeile 17 wird mit dem Befehl *spi_write_dt* schlussendlich auf den Bus geschrieben. Dazu wird die initialisierte *spi_dt_spec* Struktur sowie die *spi_buf_set* Struktur übergeben. [**193833**]

Als Digital Analog Converter (DAC) wurde der 16-Bit DAC8562[5] von Texas Instruments verwendet. Der DAC besitzt zwei Ausgänge, die jeweils durch einen eigenen DAC angesteuert werden. Diese werden mit DAC-A und DAC-B gekennzeichnet. Über SPI ist dieser mit dem Prozessor verbunden. Die in dem Internal Reference datasheet [9] des DAC856x dargestellte *Command Matrix* zeigt die Bitmuster zur Steuerung. Um eine individuelle Ausgangsspannung vom DAC-A zu erhalten muss an diesen der entsprechende Befehl und Datenwert gesendet werden. Aus der genannten *Command Matrix* kann der Bit-Befehl *011* mit der Adresse *000* zur Beschreibung des DAC-A Registers mit einem anschließenden Update entnommen werden. Die Nachricht ist in dem Programmausschnitt 3.7 in Zeile 9 zu finden. Der erste Datenwert wird mit *1* festgelegt. Damit eine Rampe erzeugt wird, muss der Datenwert inkrementiert werden. Nach einem Überlauf des Datenregisters wird dieser Wert wieder auf *0* gesetzt. Es ist eine Rampe wie in der Abbildung 3.2 zu beobachten. Die Umsetzung der Rampensteuerung des DAC wird nicht von dem Quelltextausschnitt abgedeckt, kann jedoch mit der zuvor erwähnten Beschreibung umgesetzt werden.

```
1  static const struct spi_dt_spec spec = SPI_DT_SPEC_GET(DT_NODELABEL(
     spidev), SPI_OP_MODE_MASTER | SPI_TRANSFER_MSB | SPI_WORD_SET(8) |
     SPI_LINES_SINGLE | SPI_MODE_CPOL, 2);
2
3  if (!spi_is_ready_dt(&spec)) {
4   printk("spi not ready, aborting test");
5   return;
6  }
7
8  struct spi_buf txDAC = {
9   .buf = {0x28, 0x00, 0x01},
10  .len = 3
11 };
12 struct spi_buf_set txADCBufs = {
```

[5] Produktwebseite des DAC8562: https://www.ti.com/product/de-de/DAC8562

```
13   .buffers = &txDAC,
14   .count = 1
15 };
16
17 int res = spi_write_dt(&spec, &txBufs);
```

Quelltext 3.7: Programmausschnitt SPI DAC Steuerung

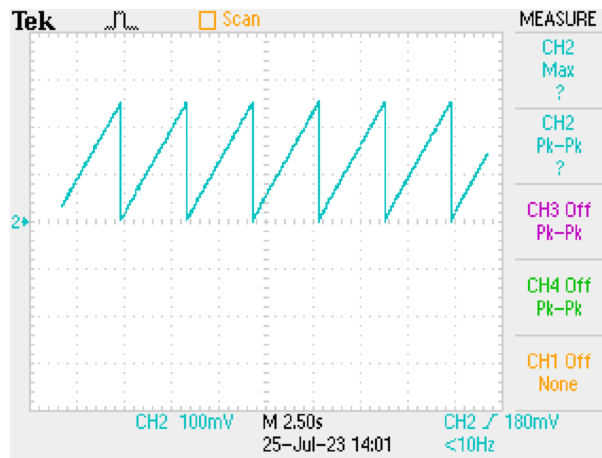


Abbildung 3.2: DAC Rampenmessung mittels Oszilloskop [eigene Abbildung]

4 Steuerung von Hardwarekomponenten in einem eingebetteten System

4.1 Gegenüberstellung Zephyr RTOS zur Bare-Metal Umgebung

Sowohl eine mit Zephyr geschriebene Applikation als auch ein Bare Metal Programm sind für den Einsatz in Mikrocontrollern mit limitierten Ressourcen gedacht. So ist es nicht verwunderlich, dass sowohl Zephyr als auch ein Bare Metal Programm größtenteils in der Programmiersprache C geschrieben wird. Diese bietet den Vorteil, Speicheradressen direkt zu schreiben und lesen. Zudem ist die Hochsprache C im Gegensatz zu Assembler wesentlich übersichtlicher und bietet viele Strukturen und Abstraktionselemente. Im Gegensatz zu abstrakteren Programmiersprachen wie Java, kann ein Programm, welches in C geschrieben wurde, sehr performant und ressourcensparend ausgeführt werden. Neben C wird auch die Sprache C++ in Zephyr unterstützt und kann meist problemlos verwendet werden. Zu beachten ist, dass Features wie C++ Threads nicht unterstützt werden. Das kann sich jedoch in zukünftigen Zephyr Versionen ändern.

Eine weitere Gemeinsamkeit beider Ansätze ist die Verwendung von herstellerspezifischen Hardware-Abstraktions-Ebenen, kurz *HAL*. Diese Bibliotheken bestehen aus Treibern und APIs, die zur Steuerung der dafür bstimmten Hardware gedacht sind. Somit kann auch in einer Bare-Metal Umgebung eine gewisse Abstraktion erreicht werden. Zephyr hingegen nutzt die HALs der Hersteller als Middleware zwischen den eigenen APIs und der vorhandenen Hardware. Ein Beispiel dafür ist der HAL von NXP. Dieser enthält unter anderem Treiber für GPIO oder Serial Peripheral Interface (SPI), die direkt von Zephyr wiederverwendet werden.

Trotzdem unterscheidet sich Zephyr von einer Bare Metal Umgebung in vielen Punkten. Neue Konzepte wie Threads, Semaphore, viele integrierte Bibliotheken und Treiber können die Softwareerstellung wesentlich vereinfachen. Dazu gehört unter anderem der Devicetree, auf welchen in dem Kapitel 3.1 genauer eingegangen wurde. Neben dem Hardwaresupport für viele Plattformen, wie die von NXP Semiconductors, Nordic Semiconductor, Infineon oder Intel, unterstützt das RTOS im Bereich Konnektivität viele Protokolle und Treiber. In Abschnitt 2.1 sind einige Beispiele unter Konnektivität zu finden. Durch eine einheitliche API können diese verhältnismäßig einfach implementiert werden. Zudem ist die API von Zephyr meist gut getestet und erprobt. Dadurch ist die Wahrscheinlichkeit für Fehler wesentlich geringer als eine Eigenimplementation.

4.2 Ansätze zur Integration von bestehenden Treibern in Zephyr

Das Zephyr RTOS unterstützt verschiedene Implementationsansätze zu Steuerung von Hardwarekomponenten. Viele Programmteile die Treiber oder Schnittstellen von Altprojekten enthalten, welche in Bare Metal programmiert wurden, können in Zephyr wiederverwendet werden. Als Grundlage dafür soll die Registersteuerung in Zephyr erläutert werden. Da Projekte auch auf bestehenden Herstellertreibern, wie denen von NXP aufbauen können, soll dieses

Verfahren ebenfalls dargestellt werden. Der Vorteil liegt darin, dass diese Treiber schon in Zephyr integriert sind und somit nicht zusätzlich eingebunden werden müssen. Zuletzt folgt noch die Hardwaresteuerung mittels der Zephyr API. Diese ist für das System optimiert und kann meist direkt genutzt werden.

4.2.1 Direkte Steuerung über der Register

Viele Projekte, die in C oder C++ entwickelt wurden, beruhen auf eigens entwickelten Treibern. Diese sind meist speziell für einen oder wenige Prozessortypen konzipiert.

Bei der Umstellung auf das Zephyr RTOS können bestehende Bibliotheken wiederverwendet werden. Wird beispielsweise derselbe Prozessor wie in bereits bestehenden Projekten genutzt, bietet sich diese Möglichkeit an. Dadurch kann neuer Implementierungsaufwand verhindert und Software schnell mithilfe des Betriebssystems erstellt werden. Zephyreigene Konzepte, wie unter anderem Threads, können in die Projekte integriert werden. Das folgende Beispiel zur Steuerung einer LED veranschaulicht, wie ein Projekt ohne vordefinierte Treiber mit diesem Ansatz erstellt werden kann.

LED Steuerung im Bare-Metal Ansatz

Anhand des Quelltextes 4.1 soll gezeigt werden, wie mithilfe von Adressbeschreibungen eine einfache Gerätesteuerung, welche eine LED des Entwicklerboards blinken lässt, implementiert wird. Dafür wurde das i.MX RT1170 Evaluation Board[6] von NXP verwendet. Für die korrekte Umsetzung wurde das Prozessor-Handbuch [11] genutzt.

Damit Kernelobjekte wie *sleep* verwendet werden können, wurde der Kernel Header des Zephyr Betriebssystems eingebunden. Falls keine Zephyr eigenen Methoden benötigt werden, ist das Inkludieren des Headers nicht notwendig.

Anschließend folgt die *Main-Methode*, welche nach der Initialisierung des Betriebssystem aufgerufen wird. Damit die LED angesteuert werden kann, muss der entsprechende Pin des Ports freigeschaltet werden. Aus den Designfiles [25] des i.MX RT1170 Entwicklerboard ist zu entnehmen, dass die LED über Pin 4 auf GPIO9 angeschlossen ist. Das Handbuch [11] des Prozessors enthält alle Register mit der jeweiligen Aufteilung und deren Beschreibung. So besitzt jedes Register mehrere Ausgänge, die frei konfiguriert werden können. Dieser Vorgang wird auch Multiplexing genannt. Damit der benötigte Ausgang freigeschaltet werden kann, müssen die entsprechenden Bits in das *MUX_MODE* Register geschrieben werden. Dieses enthält standardmäßig für die i.MX RT1170 Baureihe den Binärwert *0101* und hat somit den Multiplexer Modus, kurz Mux Modus, 5. Da in dem Register *SW_MUX_CTL_PAD_GPIO_AD_04* der Mux Modus *GPIO9_IO03* an zehnter Stelle steht, muss diese Auswahl in das Register geschrieben werden. Hexadezimal ist der Wert 10 ein *A*, weshalb dieser in Zeile 7 in das Register geschrieben wurde. Zu beachten ist, dass Pin 4 der dritte Anschluss ist, da die Zählung mit 0 und nicht mit 1 beginnt. [11]

[6] MIMXRT1170 Entwicklungsboard: https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/i-mx-rt1170-evaluation-kit:MIMXRT1170-EVK

Die Ansteuerung des Pins erfolgt in dem GPIO9 Register mit der Adresse *40C6_4000* Hex. Dafür muss als erstes der Pin als Ausgang definiert werden. Dieser besitzt einen Abstand von 4 Hex zu der Basisadresse von GPIO9. Aus dem Handbuch kann entnommen werden, dass eine *1* an die Stelle des Pins geschrieben werden muss. Dann ist dieser als Ausgang definiert. Der Vorgang dafür ist in Zeile 10 und 11 der Abbildung 4.1 zu erkennen. [11]

Um den Pin High zu setzten, muss eine *1* inTag das *DR_SET* Register gesetzt werden. Dieses hat einen sogenannten Offset von *84* Hex zu der Basisadresse von GPIO9. Damit ergibt sich die Adresse *0x40C64084*. Um die Spannung wieder auf Low zu schalten muss eine *1* in das *DR_-CLEAR* Register geschrieben werden. Dies hat die Adresse *0x40C64088*. Die Definition dieser Pointer sind in Zeile 14 und 17 beschrieben.[11]

Anschließend folgt eine Schleife, die unendlich lange wiederholt wird. Diese setzt das Bit in dem DR_SET Register, wodurch die LED anfängt zu leuchten. Danach wird in der Schleife eine halbe Sekunde gewartet, bis in Zeile 22 das Bit für Pin 4 gesetzt wird. Damit schaltet sich die LED wieder aus. Nun wird wieder eine halbe Sekunde gewartet bis sich die Schleife wiederholt.

```
#include <zephyr/kernel.h>

void main(void)
{
 // schaltet GPIO9 Pin 4 frei
 volatile uint32_t* GPIO9_IO04 = (volatile uint32_t*) 0x400E811C;
 *GPIO9_IO04 = 0xA;

 // setzt GPIO9 als Ausgang
 volatile uint32_t* GPIO9_DIRECTION = (volatile uint32_t*) 0x40C64004;
 *GPIO9_DIRECTION |= (1<<3);

 // Adresse PIN LED Set GPIO9
 volatile uint32_t* GPIO9_PIN_SET = (volatile uint32_t*) 0x40C64084;

 // Adresse PIN LED Clear GPIO9
 volatile uint32_t* GPIO9_PIN_CLEAR = (volatile uint32_t*) 0x40C64088;

 while (1) {
  *GPIO9_PIN_SET = 8;     // setzt PIN 3 auf High
  k_msleep(500);
  *GPIO9_PIN_CLEAR = 8;   // setzt PIN 3 auf Low
  k_msleep(500);
 };
}
```

Quelltext 4.1: Programm zur Steuerung einer LED durch Register mittels Zephyr

Die in dem Programmausschnitt gezeigten Befehle können in Funktionen ausgegliedert werden. Somit ist die Umsetzung von eigenen Bibliotheken problemlos möglich. Diese können dann in einem Projekt mit Zephyr verwendet werden.

4.2.2 Verwendung herstellerspezifischer Treiber

Eine weitere Möglichkeit Treiber in Zephyr zu integrieren und zu nutzen ist, die Verwendung von bereits entwickelten Treibern der entsprechenden Prozessorhersteller. So können Projekte, die schon mit den gerade erwähnten Treibern entwickelt wurden, direkt in Zephyr integriert und genutzt werden. Meistens besitzen diese Bibliotheken schon eine Abstraktionsschicht die mehrere Prozessortypen des gleichen Herstellers, durch eine einheitliche API zusammenfasst. Mithilfe dieser kann relativ schnell Software für das entsprechende System entwickelt werden. Zudem ist der Umstieg auf einen Prozessor, welcher die selbe API unterstützt wie ein schon verwendeter, einfacher umzusetzen.

Ein wesentlicher Vorteil der herstellerspezifischen Treiber liegt darin, dass diese meist sehr gut dokumentiert und getestet sind. Außerdem decken diese im besten Fall schon sämtliche Funktionalitäten des Prozessors ab. Dadurch dass die Treiber meistens in einer Hochsprache z.B. C geschrieben sind, bleiben diese leicht zu verstehen und können bei Bedarf gepatcht werden.

LED Steuerung am Beispiel eines NXP Treibers

Für dieses Beispiel wurden Treiber von NXP verwendet, um die verbaute LED auf dem Entwicklerboard MIMXRT1170 blinken zu lassen. Da Zephyr mit seiner API auf die Treiber der Prozessorhersteller aufbaut, sind diese fest in das RTOS integriert. Dadurch müssen diese nicht mit in das Projektverzeichnis aufgenommen werden und können direkt inkludiert werden. Das Einbinden der Bibliotheken wird in Zeile 1 bis 5 des Quelltext 4.2 vorgenommen.

Die Pin-Initialisierung sowie das Multiplexing wird in der Funktion BOARD_InitPins durchgeführt. Die Methode enthält eine Struktur vom Typ *gpio_pin_config_t*, welche in diesem Fall den Pin als Ausgang definiert und keine Interrupts vorsieht. Daraufhin wird GPIO9 mit dem Pin der LED und der zuvor erwähnten Struktur durch die Treiber-Funktion *GPIO_PinInit* aktiviert. Das Multiplexing erfolgt durch die darunterliegende Methode in Zeile 22. Dieser wird der Name des Pins und der Mux Modus übergeben. Die Board_InitPins Funktion wird dann in der *Main-Methode* in Zeile 27 aufgerufen. Anschließend folgt die endlos While-Schleife, die zuerst den Ausgang auf High stellt und anschließend auf low. Zwischen jedem Wechsel wartet der Prozessor eine halbe Sekunde.

```
1 #include <zephyr/kernel.h>
2 #include "fsl_iomuxc.h"
3 #include "fsl_common.h"
4 #include "fsl_gpio.h"
5 #include "fsl_clock.h"
6
7 #define LED_GPIO     GPIO9
```

```
#define LED_GPIO_PIN (3U)

// Initialisierung des LED GPIO Pins
void BOARD_InitPins(void) {

 // Pin Konfiguration
 gpio_pin_config_t gpio9_pinM13_config = {
  .direction = kGPIO_DigitalOutput,
  .outputLogic = 0U,
  .interruptMode = kGPIO_NoIntmode
 };

 // schaltet GPIO9 Pin 4 frei
 GPIO_PinInit(LED_GPIO, LED_GPIO_PIN, &gpio9_pinM13_config);
 IOMUXC_SetPinMux(IOMUXC_GPIO_AD_04_GPIO9_IO03, 0U);
}

int main(void)
{
 BOARD_InitPins();

 while (1)
 {
  GPIO_PinWrite(LED_GPIO, LED_GPIO_PIN, 0U);
  k_msleep(500);
  GPIO_PinWrite(LED_GPIO, LED_GPIO_PIN, 1U);
  k_msleep(500);
 }
}
```

Quelltext 4.2: Programm zur Steuerung einer LED mithilfe des NXP Treibers in der Zephyr Umgebung

4.2.3 Gerätesteuerung mittels der Zephyr API

Die Zephyr API bietet eine umfangreiche Sammlung von Treibern und Middleware-Komponenten, die den Umgang mit verschiedenen Peripheriegeräten erleichtern. Dazu zählen unter anderem die *Analog-to-Digital Converter API*, die *Bluetooth APIs*, die *Multi-Channel Inter-Processor Mailbox API* und viele weitere. In der Abbildung 2.1 sind die meisten Treiber durch die markierten Hard- und Softwarekomponenten aufgeführt. Diese können dann mithilfe der entsprechenden API angesteuert werden.

Um die Zephyr API korrekt zu nutzen und zu implementieren, müssen mehrere Konzepte verstanden werden. Dazu gehört beispielsweise der Devicetree, die Projektkonfiguration, sogenannte Bindings sowie die KConfig Datei, welche in Kapitel 3 genauer erläutert wurden. Während des Kompilierens wird aus diesen Dateien die Applikation für die entsprechende Zielarchitektur gebaut. Durch geringfügige Anpassungen oder Ergänzungen können sämtliche Prozessoren, welche Zephyr unterstützt, mithilfe der APIs gleich angesteuert werden.

Dadurch muss die eigentliche Code-Basis mit der Gerätelogik nicht verändert werden. Jedoch ist zu beachten, dass die benötigten Komponenten und Eigenschaften auf den jeweiligen Prozessoren vorhanden sein müssen.

Beispielprogramm zur LED Steuerung in Zephyr

Wie die beiden zuvor beschriebenen Programme in Quelltext 4.1 und 4.2 steuert auch diese Applikation das Blinkverhalten der LED des Entwicklerboards. Der Quellcode ist an dem Zephyr-Beispiel[7] orientiert.

In diesem Projekt wird zu dem Kernel noch der Treiber für die GPIO Funktionalität eingebunden. Dieser stellt eine API bereit, um GPIO Pins zu konfigurieren und diese anschließend zu steuern.

Zephyr nutzt das Konzept des Devicetrees, weshalb keine Aktivierung und kein Multiplexing im Code erfolgen muss. Die meisten Pins sind in der jeweiligen Boarddatei[8] schon vorkonfiguriert. In der Devicetree-Datei des i.MX RT1170 *mimxrt1170_evk.dtsi* ist festgelegt, dass *led0* die grüne LED mit GPIO9 und Pin 4 ist. Die LEDs sind in dem Abschnitt von Zeile 20 bis 31 definiert. Mehr Informationen zu dem Devicetree und dessen Funktionsweise kann unter 3.1 nachgelesen werden.

Durch das Macro *GPIO_DT_SPEC_GET* kann auf den Devicetree Eintrag zugegriffen werden. Dieser wird in die Struktur *gpio_dt_spec* aufgelöst und unter der Variable *led* gespeichert.

In der Main-Methode wird anschließend geprüft, ob dieser GPIO Kanal bereit ist. In Zeile 11 ist die entsprechende API Funktion zu sehen. Anschließend wird der Pin in Zeile 15 als Ausgang gesetzt. Nachdem geprüft wurde ob die Konfiguration erfolgreich war, beginnt die Endlosschleife. Diese schaltet mit der API Funktion *gpio_pin_toogle_dt* den Zustand der LED mit jedem Durchlauf auf den jeweils anderen Zustand um.

```
1  #include <zephyr/kernel.h>
2  #include <zephyr/drivers/gpio.h>
3
4  static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(led0, gpios);
5
6  int main(void)
7  {
8
9   if (!gpio_is_ready_dt(&led)) {
10   return 0;
11  }
12
13  int ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
```

[7] Zephyr Blinky Beispielprojekt: https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/basic/blinky

[8] Boarddatei i.MX RT1170 EVK: https://github.com/zephyrproject-rtos/zephyr/blob/main/boards/arm/mimxrt1170_evk

```
14   if (ret < 0) {
15    return 0;
16   }
17
18   while (1) {
19    ret = gpio_pin_toggle_dt(&led);
20    k_msleep(500);
21   }
22   return 0;
23  }
```

Quelltext 4.3: Programm zur Steuerung einer LED durch die Zephyr API

4.2.4 Vergleich der Implementierungsmöglichkeiten

In den vorangegangen Abschnitten wurden drei Konzepte vorgestellt. Im Folgenden findet ein Vergleich dieser Konzepte statt. Es werden die Vor- und Nachteile aufgeführt und zudem wird ein Überblick über die unterschiedlichen Verwendungsmöglichkeiten dargelegt.

Ein Vorteil der drei unterschiedlichen Herangehensweisen ist, dass alle drei komplett durch das Zephyr RTOS unterstützt und mit diesem kompiliert, sowie auf dem Zielsystem ausgeführt werden können. Dadurch müssen sich Softwareentwickler nicht auf eines dieser Konzepte festlegen und können bei ihrer Auswahl flexibel bleiben. Je nach Gegebenheiten kann zwischen einer dieser Möglichkeiten entschieden oder eine Kombination aus mehren gewählt werden.

Ein wesentlicher Vergleichspunkt zwischen den Programmen ist die Lesbarkeit. Dabei fällt auf, dass das Bare-Metal Programm mit den definierten Pointer, Bitverschiebungen und das Beschreiben der Register schwieriger nachzuvollziehen ist. Es ist zwar sinnvoll, sogenannte *defines* für die benötigten Adressen und Funktionen für das Beschreiben dieser anzulegen, jedoch stellt diese Arbeit zusätzlichen Implementierungsaufwand dar. Zudem müssen sehr gute Kenntnisse für den entsprechenden Mikrocontroller vorhanden sein, z.B. wenn für komplexere Komponenten wie der Messaging Unit eines Multicore Prozessors Software geschrieben wird. Im Vergleich dazu ist das Programm mit den Treibern von NXP wesentlich einfacher zu interpretieren. Da keine Registeradressen, sondern nur die abstrakteren Definitionen davon verwendet werden, ist direkt erkenntlich, was das Programm steuert. Zudem können komplexere Operationen und Konfigurationen übersichtlicher dargestellt werden. Ein Beispiel dafür ist die Struktur zur Konfiguration des LED Pins in Zeile 14 bis 18 des Quelltext 4.2. Um die Treiber richtig nutzen zu können, braucht es Erfahrungen mit der API oder etwas Einarbeitungszeit. Da diese sich meist stark an der zugrundeliegenden Hardware orientieren, braucht es wie in dem Bare-Metal Ansatz Kenntnisse der vorhandenen Komponenten. Zudem beinhalten die Treiber von NXP bis jetzt keine Konnektivitätsprotokolle, wie zum Beispiel Bluetooth. Diese müssen eigens implementiert werden. Die vermeintlich einfachste Art und Weise, die Hardware eines Prozessors zu steuern, ist die mit Zephyr. Das Betriebssystem baut auf den Herstellertreibern auf und stellt eine eigene API bereit. Diese wird durch verschiedene Protokolle und Funktionalitäten ergänzt. Durch die Ausgliederung der Pin-Konfiguration in den Devicetree, ist das Programm wesentlich kleiner und verständlicher.

Neben der Lesbarkeit der Programme spielt die Portierung auf andere Microcontroller ebenfalls eine wichtige Rolle. Da der Ansatz in Bare Metal nur für ein System konzipiert ist, kann ein Programm ohne Anpassungen nicht für ein anderes System gebaut bzw. korrekt ausgeführt werden. Das gleiche gilt meistens auch für die Treiber der Chiphersteller. Es ist zwar möglich mehrere Prozessoren des gleichen Herstellers anzusteuern, jedoch muss eine neue Pin-Konfiguration festgelegt werden. Dies kann wie in der Funktion *BOARD_InitPins* in der Zeile 11 des Quelltextes 4.2 erfolgen. Je nach gewähltem Board muss die entsprechende Funktion aufgerufen werden. Beim Ausgliedern der Methode in eine eigene Datei kann diese beim Bauen des Binaries für das entsprechende Board durch den Kompiler hinzugefügt werden. Dadurch können auch mehrere Prozessoren desselben Herstellers unterstützt werden. Zephyr bietet mit dem Devicetree eine einfache und vielfältige Möglichkeit an, Projekte für verschiedene Prozessoren und Boards zu erstellen. So wird der eigentliche Quellcode nicht verändert. Die Konfiguration erfolgt nur in dem zuvor erwähnten Devicetree. In einem Projekt, das für mehrere Prozessoren erstellt werden soll, wird pro Board eine sogenannte *Overlay*-Datei ergänzt. Diese enthält die genauen Konfigurationen und wird beim Kompilieren für das entsprechende Board ausgewählt.

Die folgende Abbildung 4.1 zeigt die Pegelausgaben des Quelltextes 4.1 ohne die *k_msleep* Kernelfunktion. Da alle Messungen die gleichen Ergebnisse zeigten, wurde exemplarisch nur die eine Abbildung dargestellt. So ist der GPIO Pin-Wechsel bei allen Programmen deutlich zu erkennen. Die Programme wurden alle mit Zephyr für das MIMXRT1170 Evaluation Kit gebaut. Der dargestellte GPIO Pin steuert die grüne LED *D6*. Die Periode ist bei allen drei Möglichkeiten in etwa 180ns. Ein Pin-Wechsel von High auf Low oder von Low auf High dauert somit rund 90ns. Daraus ist abzuleiten, dass alle Programme gleich schnell ausgeführt werden und durch die Treiber oder das Betriebssystem keine Verzögerung entsteht. Zu beachten ist jedoch, dass die LED über einen anderen Bus angesteuert wird. Dieser hat eine Taktfrequenz von 10 MHz. Da das Programm auf dem Core M7 mit einer Taktfrequenz von 1 GHz ausgeführt wird, sollten zwischen den Programmen auch keine Performanceunterschiede auftreten.

Zusammengefasst kann gesagt werden, dass mithilfe der Zephyr API am schnellsten und einfachsten Software für Treiber entwickelt werden kann. Zudem ist die API durch den Devicetree flexibel, was die Portierung für andere Prozessoren und Platinen angeht. Sofern der Devicetree für eine Platine schon konfiguriert ist, sind Kenntnisse zu der Hardware nicht notwendig. Es reicht aus, die API sinnvoll einsetzen zu können.

Das Konzept des Devicetrees und weiterer Konfigurationsdateien in Zephyr können am Anfang sehr verwirrend sein. Wenn Software für eine bestimmte Plattform schon in einer Bare Metal Umgebung oder mithilfe von Treibern entwickelt wurde, kann diese gleich verwendet werden. Somit können Projekte schnell mit bestehenden Gerätetreibern oder Bibliotheken umgesetzt werden. Jedoch sind diese Projekte nicht sehr flexibel. Gerade wenn es um die Portierung für neue Prozessoren oder Boards geht.

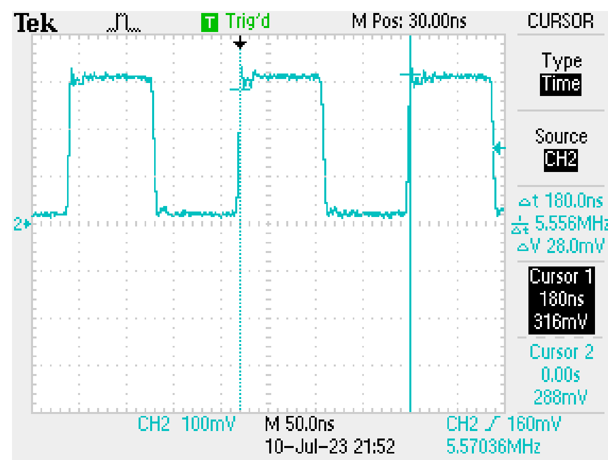


Abbildung 4.1: Zephyr API GPIO Pinwechsel [eigene Abbildung]

5 Datenaustausch zwischen mehreren Kernen eines Mikrocontrollers

Ein Großteil an Prozessoren besitzt mehrere Kerne, um die Gesamtleistung des Systems zu steigern. Damit die Kerne miteinander kommunizieren und Zustände austauschen können, wird das Konzept der Interprocessor-Communication (IPC) genutzt. Diese setzt die Verwendung von einer der folgend vorgestellten Architekturen voraus. Zum einen, wie bereits in Kapitel 2.3.2 erläutert, die symmetrische Prozessorverarbeitung zum anderen auch die asymmetrische Verarbeitung. Bei der asymmetrischen Variante laufen auf jedem Prozessorkern eigene Programme. Diese können ein Betriebssystem wie Linux oder ein RTOS wie Zephyr nutzen. Eine Bare-Metal Umgebung ist ebenfalls möglich. Für diese Arbeit wurde der Fokus mehr auf die AMP-Architekturen (Asymetric Multiprocessing) gelegt. In den folgenden Abschnitten wird die Umsetzung diverser Herangehensweisen erläutert. [1]

5.1 Anforderungen an die IPC

Die meisten eingebetteten Systeme besitzen Limitierungen, wie zum Beispiel die Speichergröße oder die Rechenleistung. Damit es trotzdem zu einer reibungslosen Integration von IPC-Mechanismen kommt, müssen spezielle Anforderungen berücksichtigt werden. Diese sind folgend aufgeführt:

- Eine **Ressourcenbeschränkung** ist fast in jedem System gegeben. Gerade Mikrocontroller haben meist wenig Speicher und Rechenleistung. IPC-Mechanismen sollten diese effizient nutzen.
- **Echtzeitverhalten** entscheidet, ob eine Reaktion auf ein bestimmtes Ereignis mit möglichst geringer Latenz erfolgt. Des Weiteren sollten die Routinen ohne Verzögerung ausgeführt werden.
- **Synchronisationen** zur Verriegelung von geteilten Ressourcen sollten ebenfalls durch die IPC abgedeckt werden. Falls dies nicht geschieht, kann es zu Dateninkonsistenz oder Verklemmungen kommen.
- **Flexibilität** wird benötigt, um verschiedene Kommunikationsprotokolle zwischen den Softwarekomponenten zu unterstützen.
- Eine **einfache Handhabung** des IPC-Mechanismus erlaubt es, die Entwicklung und das Debugging zu erleichtern.

5.2 IPC-Mechanismen im Embedded Bereich

Im Embedded-Bereich gibt es verschiedene IPC-Mechanismen, die verwendet werden, um die Kommunikation und den Datenaustausch zwischen verschiedenen Rechenkernen zu ermöglichen. Die gängigsten IPC-Mechanismen im Embedded-Bereich sind nachfolgend aufgezählt. Es ist zu beachten, dass diese sowohl auf Softwareebene zwischen Threads oder Prozessen als auch auf Hardware Ebene zwischen physisch getrennten Rechenkernen zur Anwendung kommen.

- **Nachrichtenwarteschlangen** (engl. Message Queues): Diese asynchrone Herangehensweise wird genutzt, um zwischen Threads oder Prozessorkernen Daten auszutauschen. [27]
- **Geteilter Speicher** (engl. Shared Memory): diese Methodik erlaubt es, dass mehreren Threads oder Prozesse auf den gleichen Bereich im Speicher zugreifen. Um Datenkorruption, welches die Beschädigung oder Veränderung von Daten beschreibt, vorzubeugen, sind zudem Synchronisationsmechanismen notwendig. [27]
- **Semaphore**: Diese sind globale Zähler, die entweder null oder positive Ganzzahlen annehmen können. Sie werden dazu genutzt, den exklusiven Zugriff auf Ressourcen abzusichern oder asynchrone Abläufe zu koordinieren. Beim Initialisieren eines Semaphors wird der Zähler auf die maximale Anzahl der verfügbaren Ressourcen gesetzt. Bei jeder Anfrage wird geprüft ob der Semaphor nicht null ist und somit noch reserviert werden kann. Das geschieht durch die Dekrementierung des Zählers um die Zahl eins. Die Freigabe der Ressource erfolgt durch eine Inkrementieren um die Zahl eins. Diese Synchronisationsmechanismus ist dafür verantwortlich, sogenannte *Race Conditions* zu verhindern. Eine Race Condition ist ein unerwünschter Zustand der entsteht, wenn mehrere Operationen gleichzeitig ausgeführt werden. In Sektion 5.2 ist das Prinzip genauer erläutert. [27]
- **Mutexe**: Diese sind binäre Semaphore, die entweder geschlossen oder geöffnet sind. Die Funktionsweise ist mit einem Schloss zu vergleichen, welches einen kritischen Abschnitt, wie den geteilten Speicher, absichert.
- **Event Interrupts**: Mithilfe von Event Interrupts können Signale von dem einen Prozessor zu dem anderen übertragen werden.

Race Condition

Die Abbildung 5.1 veranschaulicht das Prinzip der *Race Conditions* bei einem Shared Memory Zugriff. So enthält der Shared Memory Bereich in der ersten Zeile den String „Hello, World!". Dieser soll in der zweiten Zeile von Kern A einen neuen Wert „Lorem Ipsum"zugewiesen bekommen. In Zeile drei wurde der erste Teil „Lorem"in den Speicher geschrieben. Bevor „Ipsum"von Kern A geschrieben wird, liest Kern B aus den Speicher und enthält den korrupten Datenwert „Lorem, World!".

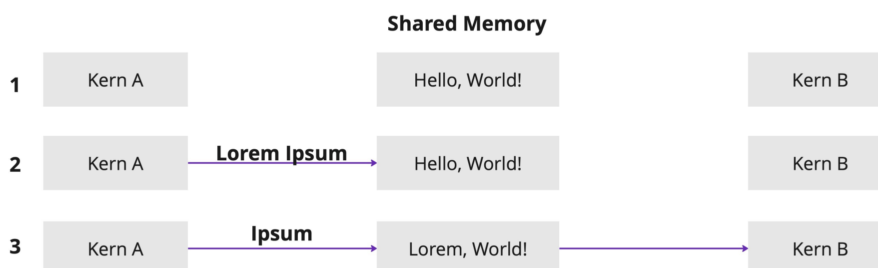


Abbildung 5.1: Auftreten einer Race Condition beim Shared Memory Zugriff [eigene Abbildung]

5.3 Dualcore Boot in Zephyr als Vorbereitung der IPC

Als Vorbereitung zur IPC müssen zwei Applikationen auf einem Dualcore System gebootet werden. Dabei wird der Ansatz der asymmetrischen Programmverarbeitung wie in Kapitel 5 beschrieben, verwendet. Damit zwei voneinander getrennte Applikationen auf jeweils einen Kern geflasht werden können, kann bei der Prozessorfamilie i.MX RT1170 zwischen zwei Verfahren gewählt werden. Bei beiden Möglichkeiten wird davon ausgegangen, dass der Master der CM7 und der Remote der CM4 ist. Die erste Herangehensweise verwendet keine Zephyr spezifischen Konzepte.

Manueller Dualcore Boot

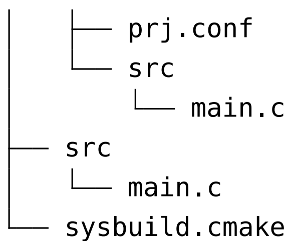
Das Programmimage des CM4 wird von der Binärdatei zu einer Hex Datei konvertiert und anschließend in das Programm des CM7 eingebunden. Zur Umwandlung kann das Python Skript *file2hex.py*[9] verwendet werden. Anschließend wird das Image beim Starten des Master-Kerns in den Speicherbereich *0x20200000* geladen. Diese Adresse ist ein Alias für den Bereich *0x1FFE0000* auf dem CM4. Damit kein Cache-Fehler auftritt, muss dieser vorher noch gelöscht werden. Außerdem muss die Vektortabelle gesetzt werden, bevor der CM4 mit dem Programm gestartet wird. [2]

Zephyr Dualcore Boot

In Zephyr wird ein Projekt zur Nutzung eines Dualcore Prozessors verwendet. Dieses enthält die Ordnerstruktur für das Remote Projekt, welches auf dem CM4 läuft und das, was auf dem CM7 ausgeführt wird. Dabei ist das Remote Projekt in dem Master Projekt, welches alle Dateien einschließt, eingebunden. Des Weiteren enthält das Projekt noch Konfigurationen für den Dualcore Boot. Diese sind mit dem Namen *sysbuild* gekennzeichnet. Der folgende Ausschnitt zeigt die Ordnerstruktur für ein beispielhaftes Dualcore Projekt in Zephyr.

```
├── CMakeLists.txt
├── Kconfig
├── Kconfig.sysbuild
├── app.overlay
├── boards
│   ├── mimxrt1170_evk_cm7.conf
│   ├── mimxrt1170_evk_cm7.overlay
├── prj.conf
├── remote
│   ├── CMakeLists.txt
│   ├── boards
│   │   ├── mimxrt1170_evk_cm4.conf
│   │   ├── mimxrt1170_evk_cm4.overlay
```

[9] file2hex.py Datei: https://github.com/zephyrproject-rtos/zephyr/blob/main/scripts/build/file2hex.py



Damit ein solches Projekt erstellt werden kann, darf der Paramter –*sysbuild* beim Kompilieren mit *West* nicht fehlen. Beim Flashen werden beide Applikationen in den Prozessor geladen und anschließend gestartet.

5.4 Umsetzung der IPC in Zephyr

Es gibt mehrere Möglichkeiten den Datenaustausch zwischen mehreren Kernen in Zephyr zu gestalten. Dabei wird in der vorliegenden Arbeit ein besonderer Fokus auf die Messaging Unit des i.MX RT1170 und deren Integration gelegt. Des Weiteren wird erwähnt, welche weiteren Mechanismen mithilfe der Komponente umgesetzt werden können. Das nachfolgende Kapitel beschreibt den grundlegenden Aufbau und die Funktionsweise der Messaging Unit.

5.4.1 Messaging Unit zum Austausch von Informationen

Die Messaging Unit (MU) ist eine Komponente des i.MX RT1170 und erlaubt die direkte Kommunikation zwischen den beiden Kernen. Es können Daten, Statusinformationen sowie Kontrollinformationen ausgetauscht werden. Zudem ist es möglich, Interrupts auf dem jeweils anderen Kern auszulösen. Somit ist diese Unit eine der wichtigsten im Bereich IPC auf dem Prozessor i.MX RT1170 geht. [11, S. 2731 f.]

Die Abbildung 5.2 veranschaulicht den Aufbau der MU. Die Komponente ist über die Peripherie-Busse jeweils mit dem CM7 und CM4 verbunden. Daten werden über das TX- und RX-Register übertragen, wobei die Bytes in das TX-Register geschrieben und aus dem RX-Register gelesen werden. Des Weiteren wird über das sogenannte Statusregister signalisiert, dass neue Daten im RX-Register vorliegen. Durch die Konfiguration der Controlregister kann dann ein Interrupt in dem jeweils anderen Kern ausgelöst werden.

Es gibt mehrere Verfahren, um Daten mithilfe der MU auszutauschen. Diese können für unterschiedliche Einsatzgebiete verwendet werden.

- Das **Non-Blocked Polling** ist eine Form des Datenaustausches, bei der ein Prozessor Daten in das TX-Register der MU schreibt. Auf der anderen Seite kann der zweite Prozessor mithilfe des RX-Registers diese Daten wieder auslesen. Es können mit dieser Methode maximal ein bis vier Datenwörter (engl. Words) gleichzeitig übertragen werden. Die fehlende Prüfung von Statusinformationen kann zu Datenverlust oder doppelten Lesen von Daten führen. Diese Möglichkeit des Datenaustausches bietet sich an, um kurze Nachrichten von 1 bis 4 Words möglichst schnell auszutauschen. [11, S. 2736]

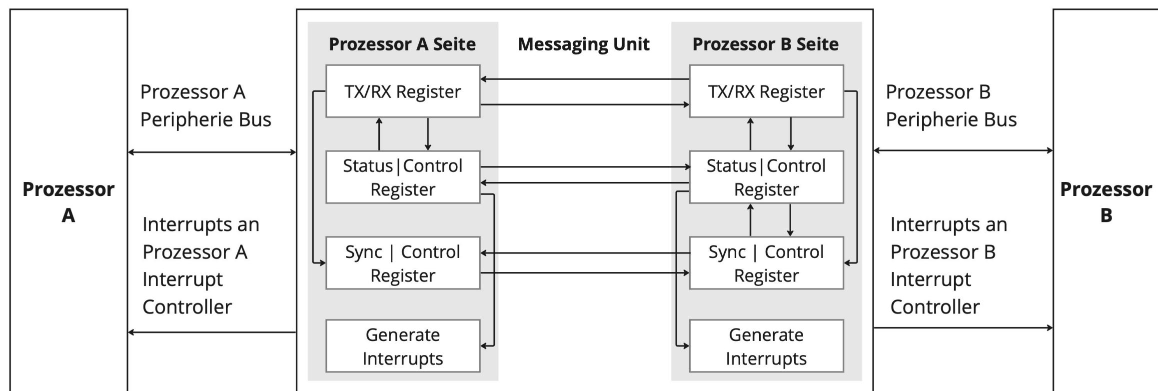


Abbildung 5.2: Block Diagramm der Messaging Unit [eigene Abbildung]

- Das **Blocking Polling** verhält sich ähnlich zu dem Non-Blocking Polling. Auch hier schreibt ein Prozessor Daten in das TX-Register. Durch das Schreiben der Daten ändert sich in dem Statusregister das *TX Empty Bit* (TEn) und das *Receiver Full Bit* (RFn) wird gesetzt. Das Lesen der Daten bewirkt, dass das *TEn* erneut gesetzt und das *RFn* zurückgesetzt wird. Mit dem Prüfen der Bits vor jedem Lese- und Schreibvorgang kann garantiert werden, dass Daten sicher von einem zum anderen Prozessor übertragen werden. Diese Möglichkeit des Datenaustausches bietet sich an, um kurze Nachrichten von 1 bis 4 Words auszutauschen, welche vollständig vorliegen sollen. [11, S. 2740 f.]
- Mit **Event Interrupts** können Signale von dem einen Prozessor zu dem anderen übertragen werden. Dafür muss das *General Interrupt Request Bit* an der Stelle n mit $n = 0, 1, 2, 3$ des Empfänger-Prozessors gesetzt werden. Wenn der Signal auslösende Prozessor sein *General Interrupt Request Bit* an der Stelle n setzt, ändert sich das *General Interrupt Pending Bit* an der Stelle n. Sobald dieses geändert wurde, wird ein Interrupt ausgelöst. Um den Interrupt wieder zu deaktivieren, muss die Empfängerseite eine *1* in das *GIP-Register* schreiben. [11, S. 2737]
- Durch das Verwenden der Kombination aus dem **Shared Memory** mit **Event Interrupts** wird die MU genutzt, um den exklusiven Zugriff auf den Shared Memory (SHM) Bereich dem anderen Prozessor mitzuteilen. Dabei müssen beide Seiten die Größe und Position des Speichers wissen. Wenn Prozessor A den exklusiven Zugriff haben möchte, sendet er ein Event Interrupt an den Prozessor B. Falls dieser nicht auf den geteilten Speicher zugreift, kann er mit einem weiteren Interrupt dem anderen Prozessor den Zugriff bestätigen. Dies ermöglicht eine Übertragung von großen, statischen Datenframes in relativ kurzer Zeit. [11, S. 2743 f.]
- **Shared Memory** mit **Blocked Polling** kann genutzt werden, um eine Übertragung von großen Datenframes zu ermöglichen. Zudem kann ein *Spinlock* oder *Round Robin System* eingebunden werden. Dadurch wird der Datenaustausch beschleunigt. Falls die Datenmenge noch unbekannt ist und unter Umständen länger vorgehalten werden muss, ist diese Herangehensweise empfehlenswert. Wenn Prozessor A Zugriff auf den SHM Bereich möchte, sendet dieser einen Interrupt an Prozessor B. Zudem sendet er über ein Transmit-Register den Ort und die Länge des Datenblocks im SHM. A wartet auf einen Interrupt von B, um fortzufahren. B wartet auf einen Interrupt seines Receiver-Registers und liest die Daten nach einem Interrupt aus. Wenn B die Zugriffsanfrage von

A akzeptiert, sendet dieser eine Nachricht an A. Nachdem B einen Interrupt ausgelöst hat, setzt A fort und hat Zugriff auf den SHM Bereich. Dies ermöglicht eine Übertragung von großen Datenframes. [11, S. 2743 f.]

5.4.2 Zephyr IPM API

Die Zephyr Interprozessor Mailbox (IPM) API nutzt eine Mailbox, um Nachrichten zu übertragen und Ereignisse zwischen Kernen eines Prozessors zu steuern. Die Mailbox besteht aus Kanälen, die für die Kommunikation von Nachrichten und Befehlen skalierbar sind. Beim genaueren Betrachten fällt auf, dass die Mailbox ein Alias für eine Komponente, wie der MU auf dem i.MX RT1170, ist. Damit hat sie die gleichen Eigenschaften und Beschränkungen wie die verbaute Komponente. Die grundlegenden Konzepte zur Funktionsweise wurden bereits in dem Kapitel 5.4.1 erläutert. Mithilfe der IPM API können nun Nachrichten durch einfache Funktionsaufrufe zwischen den Kernen ausgetauscht werden. Die wichtigsten Methoden dafür sind zum einen die *ipm_send* und *ipm_register_callback* Methode. Durch die *ipm_send* Funktion werden Daten an die Applikation des anderen Kerns gesendet. Durch das Übergeben einer Wartezeit kann ein neuer Datenwert mit einer Verzögerung gesendet werden. Das Empfangen von Nachrichten erfolgt interruptbasiert über eine Funktion, welche durch die *ipm_register_callback* initialisiert wurde. Diese wird ausgeführt, sobald eine Nachricht in der Mailbox vorliegt. Die Abbildung 5.3 zeigt grafisch den Ablauf zur Kommunikation zwischen zwei Applikationen auf einem Dualcore Prozessor. Für den i.MX RT1170 kann Applikation A zum Beispiel auf dem CM7 und Applikation B auf dem CM4 ausgeführt werden. Bei beiden Programmen muss vorerst die *callback* Methode initialisiert werden. Anschließend folgt die Aktivierung der IPM. In diesem Fall sendet Applikation A eine Nachricht durch die *ipm_send* Funktion. Das hat zur Folge, dass ein Interrupt in der Applikation B auftritt und die entsprechende Routine startet. Enthält die Routine wie in dem Beispiel ein weiteres *send*, so führt das zu einen Interrupt auf der Seite des Prozessors A. Nachdem eine Interrupt-Routine abgeschlossen ist, wird das Programm fortgesetzt.

Es ist zu beachten, dass keine anderen Aufgaben auf dem Kern ausgeführt werden, während der Interrupt durch die Callback-Methode bearbeitet wird. Das heißt, dass alle Threads erst Rechenzeit bekommen, wenn der Datenaustausch sowie die Verarbeitung abgeschlossen ist. Besonders kritisch kann dieses Verhalten bei großen Datenmengen sein, da hier das Programm nur noch mit dem Verarbeiten von Nachrichten des anderen Kerns beschäftigt ist.

5.4.2.1 Zeitmessung

Die Dauer für einen vollständigen Datenaustausch, der aus dem Senden und dem Empfangen einer Nachricht auf dem selben Kern besteht, wurde mit den Zephyr System Clock APIs getestet. Der Programmablauf ist ausschnittsweise aus der Abbildung 5.4 zu entnehmen und beruht auf dem Ping-Pong-Ablauf aus Abbildung 5.3. So beginnt die Kommunikation mit dem ersten Senden eines Zählers beginnend bei 0. Die Messung startet in dem lila markierten Rechteck der Interrupt Routine auf der CM7 Seite. Diese wurde durch vorherige Kommunikation durch den CM4 aufgerufen. Anschließend folgt in dieser Funktion ein Aufruf zum Senden

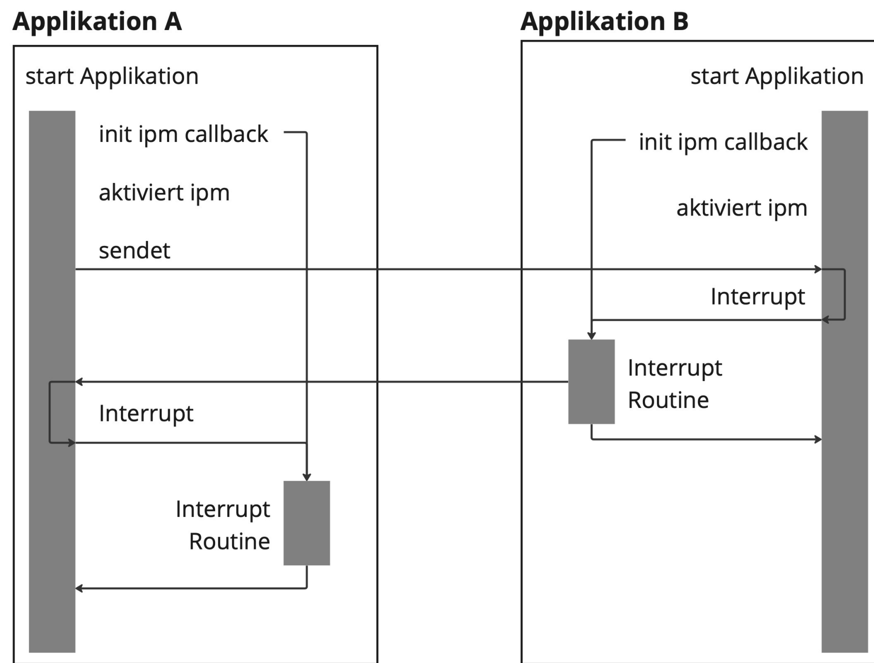


Abbildung 5.3: Ablaufdiagramm IPM Kommunikation [eigene Abbildung]

von Daten an die Seite der Applikation B. Bei dieser wird ebenfalls eine Interrupt Routine ausgeführt. Diese nimmt die eintreffenden Daten und sendet diese wieder an die Applikation A. Sobald die Routine in A erneut gestartet wird, folgt wieder eine Messung.

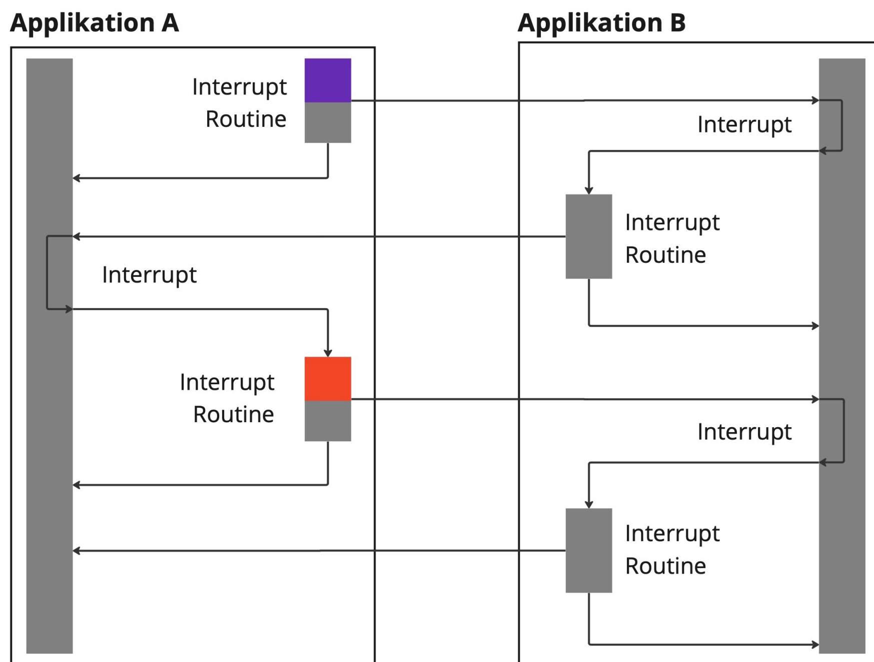


Abbildung 5.4: Ablaufdiagramm der Interprozessor-Kommunikation mittels Mailbox [eigene Abbildung]

Das Beispiel im Quelltext 5.1 veranschaulicht die Interrupt-Routine der Applikation A. In Zeile 4 wird mit der Methode *k_cycle_get_32()* die Anzahl der Takte seit Prozessorbeginn festgestellt. Anschließend folgt die Zuweisung des übertragenen Datenwerts an die Variable *gcounter*,

welche anschließend noch mit der Zahl eins inkrementiert wird. Solange der *gcounter* Wert kleiner als 100 ist, wird der Wert zurück an die parallel laufende Applikation gesendet. Es folgt noch eine Zuweisung des Messwertes aus Zeile 4 in die globale Variable *old_timer*. Daraufhin wird die Routine beendet. Um die korrekte Zeit zu ermitteln, spielen in dem Ping-Pong-System nur die letzten zwei Interruptaufrufe auf der CM7 Seite eine Rolle. Diese setzen die globalen Variablen *new_timer* und *old_timer* zum letzten mal. Die Differenz zwischen den beiden ergibt die Prozessorcyclen zwischen den zwei Interruptroutinen des CM7. Danach werden die Ergebnisse über das Terminal ausgegeben. Für das folgende Beispiel wurden *4.295* Zyklen für die Übertragung von den Daten gemessen. Bei einer Taktrate von 1 Gigahertz, welches der CM7 hat, kommt es zu einer Zeit von rund 4,3 Mikrosekunden. Der Vorteil dieser Messmethode besteht darin, dass keine rechenintensiven bzw. zeitmanipulierenden Ausgabemethoden während der Messung ausgeführt werden.

```
1  void ping_ipm_callback(const struct device *dev, void *context,
2  uint32_t id, volatile void *data)
3  {
4   new_timer = k_cycle_get_32();
5   gcounter = *(int *)data;
6   gcounter++;
7   if (gcounter < 100) {
8    ipm_send(dev, 0, 0, &gcounter, 4);
9    old_timer = new_timer;
10  }
11 }
```

Quelltext 5.1: Programmausschnitt der Interrupt Routine auf der CM7 Seite

5.4.3 OpenAMP

Das von der Multicore Association bereitgestellte OpenAMP [26] ist eine Softwarekomponente zur Entwicklung von IPC-Mechanismen in einer Mehrkernumgebung. Neben den IPC-Mechanismen wird auch eine Life Cycle Management (LCM)-Funktion angeboten. Als Abstraktionsschicht wird die *Libmetal-Bibliothek* verwendet. OpenAMP ist dafür verantwortlich, Speicher zuzuordnen, Interrupts zu behandeln und auf virtuelle Geräte zuzugreifen. Ein Beispiel zur Integration von OpenAMP in einem Zephyr Projekt kann auf der GitHub-Seite[10] von Zephyr gefunden werden.

Remoteproc

Durch Remoteproc wird das LCM auf einem System, welches mit OpenAMP genutzt wird, ermöglicht. Es erfolgt eine Zuweisung eines Masters und eines oder mehrerer Slaves oder sogenannter *Remotes* für jeweils einen Prozessorkern. Der Master kann durch das Verfahren Software-Images für die Remotes laden und anschließend starten. Dafür muss nur ein Programmimage in dem *ELF* Format vorliegen. In einer Ressourcentabelle sind notwendige

[10] OpenAMP-Beispiel in Zephyr: https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/subsys/ipc/openamp

Einträge, wie Speicherbedarf des Remoteimages, geteilter Speicher sowie eine VirtIO Gerätestruktur, vorhanden. Durch diese Struktur können später Daten mittels *vrings* zwischen den Kernen ausgetauscht werden. [24]

RPMsg

Diese Komponente von OpenAMP erlaubt die IPC zwischen verschiedenen Programmen. Dabei wird das Datenformat durch RPMsg definiert und als *Header* jeder Nachricht vorangestellt. Dieser enthält Informationen zu den Quell- und Zieladressen sowie der Größe des Payloads. Abbildung 5.5 zeigt den Aufbau einer RPmsg. [3] [24]

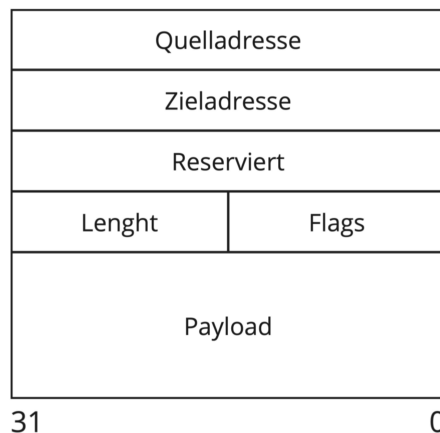


Abbildung 5.5: Headerstruktur RPMsg [eigene Abbildung]

Das RPMsg Framework besteht aus mehreren Komponenten. Diese sind nachfolgend aufgeführt und beschrieben:

- **Kanal** (engl. *Channel*) ist ein bidirektionaler Link zwischen dem Master und dem Remote Kern, der durch einen Namen von dem RPMsg Framework identifiziert wird.[3]
- **Endpunkt** (engl. *Endpoint*) ist eine logische Verbindung. Diese liegt über dem RPMsg Kanal und hat eine eigene Quell- und Zieladresse. Jeder RPMsg Endpunkt hat eine Callback-Funktion, welche aufgerufen wird, wenn Daten eingetroffen sind. Diese Daten werden dann in der Callback-Routine verarbeitet.[3]
- **VirtIO** wird genutzt, um den Datenaustausch über den geteilten Speicher zu abstrahieren. Dies geschieht durch die Erstellung virtueller Geräte in dem gemeinsamen Speicher sowie APIs zum Lesen und Schreiben der Gerätespeicher. Die Kommunikationsabstraktion wird durch die sogenannten *Virtqueues* definiert. Diese enthalten *vrings*, die jeweils einen Puffer im geteilten Speicher darstellen, welcher eine Liste aus Pointern enthält. Diese Pointer zeigen dann auf die Puffer mit Daten, welche zwischen dem Master und Remote ausgetauscht werden sollen.[3]

Zeitmessung

Ähnlich zu dem IPM-Mechanismus wurde auch hier eine Zeitmessung durchgeführt. Dafür wurde die Funktion *ping_ipm_callback*, in dem Zephyr Beispiel *OpenAMP*-Projekt[11], angepasst. Sobald die Funktion aufgerufen wird, werden die Prozessorzyklen seit Start mit der Methode *k_cycle_get_32()* ermittelt. Nachdem die *receive_message* Methode erfolgreich bearbeitet wurde, folgt eine weitere Messung der Prozessorzyklen. Zum Schluss wird eine Differenz zwischen den beiden gebildet. Diese ergab für das Beispiel rund *3.134.860* Zyklen, was bei dem CM7 in etwa *3,13* Millisekunden an Übertragungsdauer sind.

5.4.4 Vergleich der aufgeführten Möglichkeiten

Neben den drei vorgestellten Verfahren zum Datenaustausch gibt es noch einige weitere Herangehensweisen. So zum Beispiel mit den Hardwaresemaphoren oder dem Direct Memory Access (DMA) Controller. Diese wurden im Rahmen der vorliegenden Arbeit jedoch nicht näher behandelt.

Während der Bare Metal Ansatz zur Programmierung der MU die meisten Umsetzungsmöglichkeiten und Anpassungen bietet, ist es einer der kompliziertesten. Ein großer Vorteil ist, dass je nach Anforderung an das fertige System ein optimaler IPC-Mechanismus verwendet werden kann. So können zum Beispiel für größere Datenframes der Shared Memory oder für Signale Interrupts verwendet werden. Auf der anderen Seite ist die Steuerung über die Register der MU sehr komplex und benötigt ein gutes Verständnis. Zudem kann eine Eigenimplementierung unübersichtlich und fehleranfälliger sein. Außerdem ist der Pflegeaufwand der Software wesentlich zeitintensiver.

Mithilfe der Zephyr IPM API können die gerade angesprochenen Nachteile größtenteils gelöst werden. Die Bibliothek stellt eine einheitliche API bereit, die zum einen gut dokumentiert und leicht zu verwenden ist. Es reichen drei API-Funktionen aus, um eine erfolgreiche Kommunikation zwischen den Applikationen durchzuführen. Des Weiteren wird die API durch die Zephyr Foundation stets weiterentwickelt und für neue Hardware- und Softwareanforderungen angepasst. Zudem können weitere Konzepte entwickelt werden, die diese API als Grundlage nutzen. Beispielsweise kann ein gemeinsamer Speicher implementiert werden. Der Zugriff wird dann durch die IPM Bibliothek verwaltet. Zu beachten ist, dass kein Thread aktiv ist, solange die Callback-Funktion ausgeführt wird. So kann bei einer dauerhaften Kommunikation die eigentliche Arbeit des Prozessors nicht fortgesetzt werden.

Die vielseitigste Bibliothek ist wahrscheinlich die OpenAMP Bibliothek. Diese nutzt mit *Libmetal* eine eigene Bibliothek zur Hardwareabstraktion. Neben dem Einsatz in einer vollständigen Bare Metal Umgebung oder einem RTOS wie Zephyr, kann OpenAMP auch auf einem Linux System genutzt werden. Dadurch kann eine Kommunikation relativ einfach zwischen unterschiedlichen Systemen auf verschiedenen Kernen eines Prozessors umgesetzt werden. Mit

[11] Zephyr OpenAMP Projekt: `https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/subsys/ipc/openamp`

Blick auf das Beispielprojekt in Zephyr fällt auf, dass die Implementierung trotzdem aufwändig und kompliziert ist. Im Gegensatz zu der IPM API kann bei der Benutzung von OpenAMP keine Verklemmung durch dauernd ausgeführte Interrupts auftreten. Jedoch ist diese Variante auch die langsamste bei einer Datenübertragung von wenigen Bytes.

6 Firmware Update durch das Zephyr RTOS

Die Aktualisierung von Software ist ein wichtiger Bestandteil in der Mikrocontroller Programmierung. Es bietet nicht nur die Möglichkeit neue Firmwarereleases auf eingebetteten Systemen einzuspielen, sondern auch Fehlerbehebungen (engl. Bugfixes) durchzuführen. Kundengeräte, die Softwareupdates unterstützen, müssen bei einem Fehlerfall nicht zwingend ersetzt oder repariert werden. Stattdessen reicht eine Softwareanpassung, die entweder durch einen Vertriebsmitarbeiter oder den Kunden selbst eingespielt werden kann, um den Fehler zu beheben.

Zephyr bietet mit dem sogenannten Direct Firmware Update (DFU) ebenfalls die Möglichkeit Softwareupdates durchzuführen. Um ein Firmwareupdate erfolgreich einzuspielen, müssen entsprechende Schritte beachtet werden. Dazu gehört die Partitionierung des nichtflüchtigen Speichers, ein Bootloader sowie Anpassungen für die Anwendungsprogramme. Diese werden in dem folgenden Kapitel näher ausgeführt.

6.1 Grundlagen MCUboot Bootloader

MCUboot ist ein Open Source Bootloader, welcher in 32 Bit Mikrocontrollern Anwendung findet. Die Hauptaufgabe besteht darin, eine sichere Infrastruktur für das Booten und Updaten von Programmen zu ermöglichen. Dabei spielt die zugrundeliegende Hardware keine Rolle, da spezifische Betriebssystem-Methoden verwendet werden. Neben dem Einsatz in Zephyr[12] kann MCUboot in anderen Echtzeitbetriebssystemen wie Apache Mynewt[13], Apache NuttX[14], RIOT[15] und einigen weiteren verwendet werden. [14]

Für die Einrichtung des DFU hat der Bootloader eine zentrale Bedeutung. Er ist fest in Zephyr integriert und wird als eigenständige Zephyr-Applikation gebaut. Dabei ist er in dem Projektordner von Zephyr unter *bootloader* zu finden. Zudem kann MCUboot mit dem Meta Tool *West* genutzt werden, um diesen zu kompilieren und anschließend auf den Prozessor zu flashen. Viele moderne Micro Controller Units (MCUs) enthalten den Flash-Speicher direkt auf dem Chip. Jedoch ist es ebenfalls möglich, einen oder mehrere externe Speicher an die MCU anzuschließen. [8]

Partitionierung des Programmspeichers für MCUboot

Das DFU erfordert eine Partitionierung, welche den Speicher in mehrere virtuelle Sektionen unterteilt. Diese können verschiedene Aufgaben erfüllen. Beispielsweise kann eine Partition zur Speicherung des Hauptprogramms oder als reiner Datenspeicher genutzt werden. Die erste Partition sollte jedoch immer den Bootloader enthalten. Des Weiteren sollten mindestens zwei Partitionen für Applikationen erstellt werden. Eine sogenannte Scratch Partition

[12]Zephyr Bootloader: https://github.com/mcu-tools/mcuboot/tree/main/boot/zephyr
[13]Apache Mynewt Webseite: https://mynewt.apache.org/
[14]apache NuttX Webseite: https://nuttx.apache.org/
[15]RIOT OS Webseite: https://www.riot-os.org/

ist ebenfalls notwendig. Diese wird von MCUboot verwendet, um zwischen den Applikationen zu wechseln. Der Quelltext 6.1 ist ein Auszug aus der *mimxrt1170_evk.dtsi*[16] Datei. Der verwendete externe Speicher wurde über *flexspi* mit dem Prozessor verbunden und kann ab der Adresse *0x30000000* genutzt werden. Er besitzt eine Speichergröße von 16 MB.

Wie zuvor ausgeführt wurde, enthält die erste Partition den Bootloader. Diese wurde in Zeile 7 des Quelltextes 6.1 definiert. In Zeile 8 ist durch das *reg* Keyword die Startadresse und Größe der Partition angegeben. So beginnt der Bootloader in dem Speicherbereich *0x30000000* auf der ersten Adresse und endet mit einer Größe von *128* KB auf der Adresse *0x000019999*. Anschließend folgen die beiden Programmpartitionen. Diese sind mit slot0_-partition in Zeile 10 und slot1_partition in Zeile 14 angegeben. Diese Speicherbereiche sollten jeweils eine Binärdatei enthalten. Der erste Slot wird meistens Primärpartition und der zweite Slot Sekundärpartition genannt. Die *scratch_partition* in Zeile 18 ist eine vom Bootloader verwendete Partition. Diese besitzt eine Größe von 128 KB und wird verwendet, wenn die Programme zwischen *slot0* und *slot1* ausgetauscht werden. Zuletzt folgt noch die *storage_-partition* in Zeile 22, folgend auch Datenpartition genannt. Diese beginnt auf der Adresse *0x00641000* und hat eine Größe von *1.856* Kb. Die Partition kann genutzt werden, um Daten dauerhaft zu speichern. Es ist möglich, die Partitionen auf verschiedene und unabhängige physische Speicher aufzuteilen. Gerade eine Trennung zwischen den programmspezifischen Partitionen und der Datenpartition kann in vielen Fällen sinnvoll sein.

Zahlreiche weitere Entwicklerboards haben diesen Partitionierungseintrag des Quelltextes standardmäßig in dem Devicetree enthalten. Falls ein eigenes Board zum Bauen von Programmen angelegt wurde, kann in dem Devicetree die Anpassung durchgeführt werden.

```
1  partitions {
2   compatible = "fixed-partitions";
3   #address-cells = <1>;
4   #size-cells = <1>;
5
6   boot_partition: partition@0 {
7    label = "mcuboot";
8    reg = <0x00000000 DT_SIZE_K(128)>;
9   };
10  slot0_partition: partition@20000 {
11   label = "image-0";
12   reg = <0x00020000 0x301000>;
13  };
14  slot1_partition: partition@321000 {
15   label = "image-1";
16   reg = <0x00321000 0x300000>;
17  };
18  scratch_partition: partition@621000 {
19   label = "image-scratch";
20   reg = <0x00621000 DT_SIZE_K(128)>;
```

[16] Quelle mimxrt1170_evk.dtsi Datei: https://github.com/zephyrproject-rtos/zephyr/blob/main/boards/arm/mimxrt1170_evk/mimxrt1170_evk.dtsi

```
21    };
22    storage_partition: partition@641000 {
23     label = "storage";
24     reg = <0x00641000 DT_SIZE_K(1856)>;
25    };
26   };
```

Quelltext 6.1: Partitionierung i.MX RT1170 im Devicetree

Aufbau des Images für MCUboot

Jedes Programm, welches in Kombination mit dem Bootloader funktioniert hat ein spezielles, durch den Bootloader vorgegebenes Format. Dieses besteht aus dem Header, Code, TLV Manifest, Padding und Update Status. Diese Bereiche benutzt der Bootloader, um beispielsweise die Signatur zu prüfen. Die Abbildung 6.1 zeigt die Unterteilung für jeden Image Slot durch den Bootloader.[7]

Der Header ist der erste Teil des Programms und enthält Informationen zu dem Image selbst. Dazu gehört zum Beispiel die Größe und die Version des Images. Des Weiteren wird der Eintrag *Magic* genutzt, um das Image zu identifizieren und die korrekte Struktur zu prüfen. Daraus kann der Bootloader entnehmen, dass das geladene Image ein gültiges Firmware-Image ist. Das sogenannte TLV (Type-Lenght-Value) Manifest ist ein Bereich, welches Meta-Daten zu dem Programm enthält. Diese sind in einer Struktur enthalten, welche aus einem Tag zur Identifikation der Länge und dem zugehörigen Datenwert bestehen. [7]

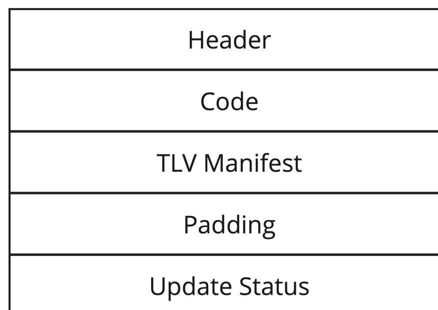


Abbildung 6.1: Image Manifest [eigene Abbildung]

6.2 Beispielablauf des Firmwareupdates mittels MCUboot

Installation des MCUboot Bootloaders

Der Ablauf zur Nutzung des Bootloaders folgt immer dem gleichen Konzept. Dafür muss dieser zuerst kompiliert werden. Der folgende Bash-Befehl kann zum Bauen der Applikation für das NXP Entwicklerboard genutzt werden.

```
$ west build -p -b ucpbaseboard\_cm7
```

Nachdem dieser in den Flash-Speicher geladen wurde, kann er durch einen Reset automatisch gestartet werden. Falls alles korrekt ausgeführt wurde, sollten über die serielle Ausgabe folgenden Informationen aus dem Quelltextbeispiel 6.2 ausgegeben werden. Aus der ersten Zeile geht hervor, dass die Zephyr Version 3.4.0 für den Bootloader verwendet wurde. Die Warnung, abgekürzt durch den Buchstabe *W* in Zeile 6 sowie der Fehler in Zeile 8, sagen aus, dass noch kein bootfähiges Programm in der *slot0* Partition enthalten ist. Wenn eine Applikation vorhanden ist, jedoch nicht signiert, tritt zudem die Fehlermeldung in Zeile 7 auf.

```
1 *** Booting Zephyr OS build zephyr-v3.4.0-1-gd8e8ee017607 ***
2 I: Starting bootloader
3 I: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
4 I: Secondary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0
    x3
5 I: Boot source: none
6 W: Failed reading image headers; Image=0
7 E: Image in the primary slot is not valid!
8 E: Unable to find bootable image
```

Quelltext 6.2: Ausgabe MCUboot bei korrekter Implementierung

Ein Programm kann über zwei Methoden signiert werden. Zum einen kann durch den folgenden Befehl ein schon bestehendes Binary signiert werden. Dafür muss der Schlüssel als Pfad dem Befehl angehangen werden. Der Befehl veranschaulicht das Signieren mittels *imgtool* in der *West* Umgebung.

```
$ west sign -t imgtool -- --key ~/zephyrproject/bootloader/mcuboot/root-
    rsa-2048.pem
```

In dem oberen Beispiel wurde der *root-rsa-2048* verwendet. Der Befehl ist in dem Projektordner auszuführen und erstellt eine *zephyr.signed.bin* in dem *build/zephyr* Ordner. Des Weiteren kann der Schlüssel als Konfigurationseintrag in der *prj.conf* ergänzt werden. Der Eintrag dafür ist anschließend zu erkennen. Der Pfad muss einen Schlüssel aus dem *mcuboot* Ordner enthalten.

```
CONFIG_MCUBOOT_SIGNATURE_KEY_FILE="~/zephyrproject/bootloader/mcuboot/root
    -rsa-2048.pem"
```

Beim Flashen muss nun diese signierte Version des Images verwendet werden. Der folgende Terminal Befehl zeigt, wie mittels *west* das signierte Image in die primäre Partition geflasht wird.

```
$ west flash --file build/zephyr/zephyr.signed.bin
```

Die Ausgabe über die serielle Schnittstelle wird in dem Terminal-Ausschnitt des Quelltextes 6.3 dargestellt. Es ist, gerade im Vergleich zu dem Ausschnitt im Quelltext 6.2, zudem zu erkennen, dass ein bootfähiges Image vorliegt, welches in Zeile 8 vom Bootloader gestartet wird und in Zeile 9 anfängt zu booten. Damit eine Verbindung zwischen einem PC und dem Board aufgebaut werden kann, muss ein sogenannter Update-Handler implementiert werden.

```
*** Booting Zephyr OS build zephyr-v3.4.0-1-gd8e8ee017607 ***
I: Starting bootloader
I: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
I: Secondary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0
  x3
I: Boot source: none
I: Swap type: none
I: Bootloader chainload address offset: 0x20000
I: Jumping to the first image slot
*** Booting Zephyr OS build zephyr-v3.4.0-1-gd8e8ee017607 ***
```

Quelltext 6.3: Ausgabe MCUboot bei korrekter Implementierung mit signierten Image

Integration des Update-Handlers in ein Programm

Ein Update-Handler ist eine Komponente oder ein Programm, das für die Verwaltung und Durchführung von Updates in einem System verantwortlich ist. Zephyr stellt ein Beispielprogramm mit dem Namen *smp_svr*[17] bereit, welches den Update-Handler implementiert.

Für die nahtlose Integration muss in der *prj.conf* Datei des Projektes der Eintrag *CONFIG_-MCUMGR=y* enthalten sein. Der MCUmgr ist ein Tool, welches eine Kommunikation zwischen einem Host-PC und dem eingebetteten System erlaubt. Eine detailliertere Ausführung folgt in Kapitel 6.2. Weitere wichtige Eintrage können in dem bereits erwähnt smp_svr Beispiel nachgelesen werden.

Das Programm muss noch die entsprechende Schnittstelle, in diesem Fall USB, aktivieren, damit der Update-Handler eine Verbindung zu dem angeschlossenen PC aufbauen kann. Der folgende Quelltext 6.4 zeigt einen exemplarischen Devicetreeeintrag.

```
 / {
 chosen {
  zephyr,uart-mcumgr = &cdc_acm_uart0;
 };
};

&zephyr_udc0 {
 cdc_acm_uart0: cdc_acm_uart0 {
  compatible = "zephyr,cdc-acm-uart";
 };
};
```

Quelltext 6.4: Overlay Datei zur USB Integration für den Update-Handler

[17] smp_svr Beispielprojekt: https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/subsys/mgmt/mcumgr/smp_svr

Image Management mittels MCUmgr

Das MCUmgr-Tool ist ein Befehlszeilenwerkzeug, das als Teil des Zephyr-Projects entwickelt wurde. Es ist ein Kommunikationsprotokoll, welches das Management von Mikrocontrollern ermöglicht, die das Zephyr RTOS ausführen. Es bietet eine Schnittstelle zur Interaktion mit der MCU-Management-Funktionalität, einschließlich der Möglichkeit Daten zu lesen, schreiben und konfigurieren. Des Weiteren können Firmware-Updates durchgeführt werden.

Das Tool wird auf einem Desktop-PC ausgeführt und muss vor der ersten Benutzung eingerichtet werden. Die Schritte zur Installation sind auf der Zephyr Webseite[18] zu finden. Damit eine Verbindung zu dem Mikrocontroller hergestellt werden kann, muss eine fertige Applikation mit dem Update-Handler auf diesem laufen. Um ein neues Programm auf den Prozessor zu laden, wird der anschließende Befehl verwendet. Dafür wird der Pfad des signierten Images übergeben.

```
$ ./mcumgr -c acm0 image upload /path/to/project/build/zephyr/zephyr.
    signed.bin
```

Wenn die Übertragung erfolgreich war, sollten zwei Images auf den jeweiligen Slots liegen. Mit dem folgendem Befehl können Informationen zu den Images auf der MCU ausgelesen werden. Es ist zu erkennen, dass zwei Images auf dem jeweiligen Slot liegen, wobei das Programm auf dem primären Slot das Flag *active confirmed* hat und somit gerade ausgeführt wird. Neben dem Eintrag für die Image-Version ist der Hashwert von großer Bedeutung. Dieser wird benötigt, wenn ein Wechsel zwischen den Programmen erfolgen soll.

```
$ ./mcumgr -c acm0 image list
  Images:
  image=0 slot=0
  version: 0.0.1
  bootable: true
  flags: active confirmed
  hash: 98aaef029e9873bc2800f03bce839263ac1836333fcb9384d5c892f192
  image=0 slot=1
  version: 1.0.0
  bootable: true
  flags:
  hash: c892f192b987cef3ec8addee00f03bce8393fae46a2c52684d5cea4c98
  Split status: N/A (0)
```

Es gibt zwei Varianten ein Programmwechsel durchzuführen. Zum einen kann das Binary auf dem sekundären Slot zum einmaligen Ausführung auf den primären Slot kopiert werden. Sobald ein *Reset* erfolgt, wird es wieder auf den sekundären Slot kopiert. Dieses Verfahren kann zum Testen von Anwendungen genutzt werden. Der Befehl benötigt zur Ausführung den Hashwert des Programms. Dieser kann mit dem oben stehenden Bash-Befehl ermittelt werden.

[18] Zephyr Webseite MCUmgr Tool: https://docs.zephyrproject.org/latest/services/device_mgmt/mcumgr.html

```
$ ./mcumgr -c acm0 image test <hash>
```

Die zweite Variante eignet sich für den dauerhaften Einsatz der Applikation. So wird mit dem folgendem Befehl ein Image in die erste Bootpartition geladen und bleibt bis zu einem weiteren manuell ausgeführten Wechsel in dieser Partition. Es wird ebenfalls der Hashwert des Programms benötigt.

```
$ ./mcumgr -c acm0 image confirm <hash>
```

7 Fazit

7.1 Zusammenfassung

Das Echtzeitbetriebssystem Zephyr ist eine immer populärer werdende Alternative, um Programme für Mikrocontroller zu entwickeln. Gerade die Steuerung und Implementierung von Treibern kann mit den bereitstehenden Konzepten, wie dem Devicetree und den vorhanden APIs, gut umgesetzt werden. Trotzdem besteht weiterhin die Möglichkeit, bestehende Softwarekomponenten zur Hardwaresteuerung zu implementieren, welche nicht auf Zephyr Bibliotheken aufbauen. Diese Möglichkeit ist zu empfehlen, wenn schon Software für ein bestehendes Produkt entwickelt wurde und dieses mit Zephyr-Konzepten erweitert werden soll. Ansonsten sollte Entwicklungsaufwand in die Neuimplementierung mittels des Devicetrees und den schon vorhandenen APIs investiert werden. Dadurch wird nicht nur sichergestellt, dass eine Anpassung der Software für weitere Platinen erfolgt, sondern auch, dass diese APIs weiterentwickelt werden. Durch den Support der Hersteller für das Zephyr RTOS kann davon ausgegangen werden, dass die Anpassung für neue Prozessorserien weiterhin gewährleistet wird. Fehlen projektnotwendige Softwarekomponenten, können diese entweder in das offizielle Zephyr-Project oder auf einen Fork dessen implementiert werden.

In dem Zusammenhang mit den Treibern wurden in der vorliegenden Arbeit die IPC Möglichkeiten mit Zephyr untersucht. Dabei wurde ein Vergleich zwischen einem Bare-Metal Ansatz, der Zephyr IPM Bibliothek und dem OpenAMP Modul näher betrachtet. Die Messaging Unit des i.MX RT1170 hat bei diesen Verfahren eine tragende Rolle gespielt, da sie speziell für die Kommunikation zwischen dem CM7 und CM4 gedacht ist. Somit verwenden alle IPC Mechanismen diese, mit Ausnahme des Abstraktionslevels zur Programmierung. Die Arbeit kommt zu dem Ergebnis, dass eine direkte Ansteuerung, ohne jegliche Abstraktion, sehr individuell gestaltet und an das jeweilige Projekt optimal angepasst werden kann. Jedoch erfordert diese Herangehensweise großen Entwicklungsaufwand und kann gerade in der Anfangsphase fehleranfällig programmiert sein. Deshalb sollte eine der beiden anderen Lösungen gewählt werden. Dabei ist die Zephyr IPM API die einfachere umsetzbarere Variante von beiden. Mit wenigen Zeilen Code kann eine erfolgreiche Interrupt-basierte Kommunikation stattfinden. Zudem kann das Verfahren mit dem Einsatz eines Shared Memories erweitert werden. Dafür wird die Bibliothek genutzt, um den Zugriff auf den geteilten Speicher zu managen. Sollen auf einem Mehrkernprozessor unterschiedliche Systeme, wie eine Linux Distribution, ein Echtzeitbetriebssystem wie Zephyr oder nur eine Gerätesteuerung ohne OS zum Einsatz kommen, ist die Verwendung der OpenAMP Bibliothek von Vorteil. Diese kann in alle Systeme eingebunden werden und einheitlich zwischen diesen Daten austauschen.

Zuletzt wurde noch das Firmwareupdate in Zephyr aufgegriffen. Da viele Unternehmen, welche Produkte mit Software ausliefern, diese ebenfalls aktualisieren wollen, ist das DFU ein wichtiger zu untersuchender Bestandteil. So müssen gewisse Grundvoraussetzungen für dieses getroffen werden. Zephyr verwendet den Bootloader MCUboot, der eine eigene Applikation ist und bei jedem Start des Mikrochips zuerst ausgeführt wird. Die Hauptaufgaben des Bootloaders liegen darin, die Signatur der Images zu Prüfen und diese zwischen den Partitionen auszutauschen. Partitionen sind in dem Fall mehrere Segmente, die Applikationen

enthalten können. Das Programm, welches in der Partition nach dem Bootloader kommt, wird anschließend ausgeführt. Mithilfe des MCUmgr-Tools kann eine Verbindung zu dem Prozessor hergestellt werden und eine Verwaltung der Images erfolgen. Zudem können mit dem Tool neue Applikationen in die MPU geladen und anschließend aktiviert werden.

7.2 Ausblick

Im Vergleich zu anderen Betriebssystemen ist Zephyr noch relativ neu und in einigen Breichen noch nicht komplett ausgereift. Deshalb gibt es noch viele Stellen an denen weiterentwickelt wird. Es ist nicht garantiert, dass die bisherigen APIs in ihrer jetzigen Form bestehen bleiben. Für die vorliegende Arbeit wurden alle Programme in der Zephyr Version 3.3.0 oder 3.4.0 umgesetzt. Zukünftig können jedoch auch neue Konzepte oder Herangehensweisen in das RTOS implementiert werden. Dies gilt es zu berücksichtigen. Des Weiteren wurde in der vorliegenden Arbeit nur SPI in Kombination mit einem Analog Digital Converter vorgestellt. Jedoch gibt es auch Zephyr spezifische Ansätze zur Steuerung anderer Treiber wie I2C oder PWM. Vertiefende Forschungsarbeiten in diesem Themenbereich sollten sich mit der Implementierung sowie mit den Vor- und Nachteilen der APIs in Zephyr beschäftigen.

Eine genauere Untersuchung der Performance zur Steuerung von GPIO Pins zwischen den Ansätzen aus Kapitel 4 ist weiterhin zu prüfen. Hierfür wären zum Beispiel die Pins von GPIO2 oder GPIO3 geeignet, da direkt mit der Taktfrequenz des Rechenkerns verbunden.

Während die vorgestellten IPC-Verfahren einen großen Nutzungsbereich abdecken, gibt es noch weitere Möglichkeiten des Datenaustausches zwischen zwei oder mehreren Kernen einer MPU. Beispielsweise könnten Untersuchungen zu den Hardwaresemaphoren oder dem Direct-Memory-Access Controller des Prozessors durchgeführt werden. Diese sollten zudem mit den in dieser Arbeit aufgeführten Mechanismen verglichen werden. Es bietet sich an, Zeitmessungen durchzuführen und herauszuarbeiten, für welchen Anwendungsfall der jeweilige Datenaustauschmechanismus von Vorteil ist. Die Anwendungsfälle können sich beispielsweise auf die zeitlichen Anforderungen oder die Größe der Daten beziehen. Für diese Versuche ist es empfehlenswert, mit dem gleichen, im Rahmen der vorliegenden Arbeit verwendeten Mikrocontroller zu arbeiten. Damit können die Ergebnisse sinnvoll mit denen der vorliegenden Arbeit verglichen werden.

Glossar

Flash-Speicher nichtflüchtige Speichereinheit ohne Erhaltungs-Energieverbrauch.

Interrupt kurzfristige Unterbrechung des Programms zur Ausführung von kurzen, zeitkritischen Routinen.

Middleware Software, die sich zwischen einem Betriebssystem und der Anwendung befindet.

Scheduler regelt die zeitliche Ausführung von Prozessen und Threads in einem Betriebssystem.

Tag (dt. Schlagwort) dient zur Identifikation und Kategorisierung von Elementen.

YAML maschinenlesbare Sprache zur Datenserialisierung, die häufig für Konfigurationseinträge verwendet wird.

Literaturverzeichnis

[1] Sara Alonso u. a. „Evaluating the OpenAMP framework in real-time embedded SoC platforms“. In: *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*. 2021, S. 1–6. DOI: 10.1109/DCIS53048.2021.9666157.

[2] *AN13264. i.MX RT1170 Dual Core Application*. NXP Semiconductos. URL: https://www.nxp.com/docs/en/application-note/AN13264.pdf (besucht am 16. 08. 2023).

[3] Etsam Anjum und Jeffrey Hancock. „Introduction to OpenAMP Library. An Open Source Standard and APIs for Asymmetric Multiprocessing (AMP) Systems“. In: URL: https://www.openampproject.org/docs/whitepapers/Introduction_to_OpenAMPlib_v1.1a.pdf (besucht am 16. 08. 2023).

[4] *Bare-metal vs RTOS programming*. URL: https://academy.nordicsemi.com/topic/bare-metal-vs-rtos-programming/ (besucht am 16. 08. 2023).

[5] Christian Bradatsch. „Multicore-Entwicklungsplattform für den Automobilbereich“. Fakultät für Angewandte Informatik der Universität Augsburg. URL: https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/deliver/index/docId/4210/file/Dissertation_Bradatsch.pdf (besucht am 16. 08. 2023).

[6] Alexey Brodkin, Hrsg. *Multicore Application Development with Zephyr RTOS*. Linux Piter 2019. synopsis. URL: https://ostconf.com/system/attachments/files/000/001/688/original/Brodkin_Linux_Piter_presentation.pdf?1570035188 (besucht am 16. 08. 2023).

[7] David Brown. „MCUboot: Multi-Image“. In: Linaro. URL: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/MCUboot-Multi-Image-Support-David-Brown-Linaro-Ltd.pdf (besucht am 16. 08. 2023).

[8] *Building and using MCUboot with Zephyr. Secure boot for 32-bit Microcontrollers!* URL: https://docs.mcuboot.com/readme-zephyr.html (besucht am 16. 08. 2023).

[9] *DACxx6x Dual 16-, 14-, 12-Bit, Low-Power, Buffered, Voltage-Output DACs With 2.5-V, 4-PPM/°C Internal Reference*. Texas Instruments Incorporated.

[10] *Eingebettetes System*. Wikipedia. URL: https://de.wikipedia.org/wiki/Eingebettetes_System (besucht am 16. 08. 2023).

[11] *i.MX RT1170 Processor Reference Manual*. Version 2. NXP Semiconductors. NXP Semiconductors. URL: https://www.nxp.com/webapp/Download?colCode=IMXRT1170RM.

[12] P. Marwedel. „Embedded software: how to make it efficient?“ In: *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*. 2002, S. 201–207. DOI: 10.1109/DSD.2002.1115370.

[13] P. Marwedel. „Embedded Systems Foundations of Cyber-Physical Systems“. In: *Embedded System Design*. 2011, S. XXI–400. DOI: 10.1007/978-94-007-0257-8.

[14] *MCUboot*. URL: https://docs.mcuboot.com (besucht am 22. 08. 2023).

[15] Zephyr Project members und individual contributors. *Application Development*. URL: https://docs.zephyrproject.org/latest/develop/application/index.html (besucht am 16. 08. 2023).

[16] Zephyr Project members und individual contributors. *Configuration System (Kconfig)*. URL: https://docs.zephyrproject.org/latest/build/kconfig/index.html.

[17] Zephyr Project members und individual contributors. *Interrupts*. URL: https://docs.zephyrproject.org/latest/kernel/services/interrupts.html (besucht am 16. 08. 2023).

[18] Zephyr Project members und individual contributors. *Mutexes*. URL: https://docs.zephyrproject.org/latest/kernel/services/synchronization/mutexes.html (besucht am 16. 08. 2023).

[19] Zephyr Project members und individual contributors. *Scope and purpose for Zephyr Devicetree*. 16. Aug. 2023. URL: https://docs.zephyrproject.org/latest/build/dts/intro-scope-purpose.html (besucht am 16. 08. 2023).

[20] Zephyr Project members und individual contributors. *Semaphores*. URL: https://docs.zephyrproject.org/latest/kernel/services/synchronization/semaphores.html (besucht am 16. 08. 2023).

[21] Zephyr Project members und individual contributors. *Symmetric Multiprocessing*. URL: https://docs.zephyrproject.org/latest/kernel/services/smp/smp.html (besucht am 16. 08. 2023).

[22] Zephyr Project members und individual contributors. *Threads*. URL: https://docs.zephyrproject.org/latest/kernel/services/threads/index.html (besucht am 16. 08. 2023).

[23] Zephyr Project members und individual contributors. *Zephyr Project Overview. A proven RTOS ecosystem, by developers, for developers*. URL: https://zephyrproject.org/wp-content/uploads/sites/38/2023/05/Zephyr-Overview.pdf (besucht am 16. 08. 2023).

[24] Dan Milea. „Hypervisor-less virtio: Assembling Multi-OS systems using standards-based protocols for intra-SoC connectivity and device sharing“. In: Wind River. URL: https://static.sched.com/hosted_files/zephyr2022/43/Zephyr_DevSummit22_DMilea_v1.1.pdf (besucht am 16. 08. 2023).

[25] *MIMXRT1170-EVK Design files*. Design Files. Version 3. NXP Semiconductors. URL: https://www.nxp.com/webapp/Download?colCode=MIMXRT1170-EVK-DESIGNFILES (besucht am 16. 08. 2023).

[26] *OpenAMP Github Repository*. URL: https://github.com/OpenAMP/open-amp.

[27] Sri Manikanta Palakollu. „Practical System Programming with C“. In: DOI: 10.1007/978-1-4842-6321-1_6.

[28] *West (Zephyr's meta-tool)*. URL: https://docs.zephyrproject.org/latest/develop/west/index.html (besucht am 16. 08. 2023).

[29] *Zephyr Project Documentation. Release 3.0.0*. Zephyr Foundation. 21. Feb. 2022. URL: https://docs.zephyrproject.org/3.0.0/zephyr.pdf (besucht am 19. 08. 2023).

[30] *Zephyr SPI Binding Datei*. URL: https://github.com/zephyrproject-rtos/zephyr/blob/main/dts/bindings/spi/spi-device.yaml (besucht am 15. 08. 2023).

Eidesstattliche Erklärung

Hiermit versichere ich – Moritz Marko Pöhlandt – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 22. August 2023

Ort, Datum

Moritz Marko Pöhlandt