Smart Mechatronics

Hochschule Bochum
Bochum University
of Applied Sciences

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Engineering (B.Eng.)

Machbarkeitsstudie zu der Mikroprozessor-Plattform STM32MP1

Autor: Philip Geuchen
philip.geuchen@hs-bochum.de
Matrikelnummer: 016200549

Erstgutachter: Prof. Dr.-Ing. Arno Bergmann
Zweitgutachter: Dipl.-Math. Kai Morwinski

Abgabedatum: 13.02.2020

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Abschlussarbeit:
»Machbarkeitsstudie zu der Mikroprozessor-Plattform STM32MP1«

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift: *Ort, Datum*:

Danksagung

Ein großes Dankeschön gilt all jenen Personen, die mich bei der Bearbeitung dieser Bachelorarbeit unterstützt und motiviert haben. Ganz besonders danken möchte ich Herrn Prof. Dr.-Ing. Arno Bergmann und Herrn Dipl.-Math. Kai Morwinski, für Ihre fachliche sowie persönliche Unterstützung.
Des Weiteren möchte ich der Smart Mechatronics GmbH danken, die mir ermöglicht hat, dieses interessante Thema zu bearbeiten.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADA	Adressbus des Datenspeichers
API	Application-Programming-Interface
APR	Adressbus des Programmspeichers
ARM	Acorn RISC Machines
BIOS	Basic Input Output System
CISC	Complex Instruction Set Computer
CoPro	Coprocessor
COM	Communication
CPU	Central Processing Unit
DA	Datenspeicher
DC	Duty Cycle
DDA	Datenbus des Datenspeichers
DDR	Double Data Rate
DPR	Datenbus des Programmspeichers
DMA	Direct Memory Access
DT	Devicetree
FPU	Floating Point Unit
FR	Forward/Reverse

FreeRTOS	Free Real-Time Operating System
Fw	Firmware
GDB	GNU Debugger
GIC	Generic Interrupt Controller
GPIO	General-Purpose Input/Outputs
GNU	GNU's Not Unix
HDMI	High Definition Multimedia Interface
HSEM	Hardware Semaphore
IDE	Integrated Development Environment
IWDG	Independent Watchdog
IPC	Inter-Process Communication
IPCC	Inter-Processor Communication Controller
LAN	Local Area Network
lsb	least significant bit
MCA	Multicore Association
MDMA	Memory Access Controller
msb	most significant bit
mmc	Multimedia Card
MPU	Microprocessing Unit
NVIC	Nested Vectored Interrupt Controller
OTG	On-The-Go
OpenAMP	Open Asymmetric Multi Processing

OP-TEE Open Source Trusted Execution Environment

PC Personal Computer

PCI Peripheral Component Interconnect

PM Prozessmanagement

PR Programmspeicher

Protobuf Google Protocol Buffers

PWM Pulsweitenmodulation

RAM Random-Access Memory

RCC Reset and Clock Controller

RemoteProc Remote Processor

RISC Reduced Instruction Set Computer

ROS Robot Operating System

RPMsg Remote Processor Messaging

RPROC Remote Processor

RTOS Real Time Operating System

SD Secure Digital Memory

SDK Software Development Kit

STM STMicroelectronics

SWOT Strengths Waknesses Opportunities Threats

SYS System

TA Trusted Application

TCP Transmission Control Protocol

TF-A	Trusted Firmware-for ARM A-Profile architectures
TTY	Teletype
UART	Universal Asynchronous Receiver Transmitter
U-Boot	Universal Boot Loader
UDP	User Datagram Protocol
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
WWDG	System Window Watchdog

Symbolverzeichnis

Symbol	Bedeutung	Einheit
bit	Bit	bit
B	Byte	8 bit
$Counter\ Period$	Zählerperiode	
f	Frequenz	Hz
f_{PWM}	PWM Frequenz	Hz
$f_{Prozessortakt}$	Prozessor-Taktfrequenz	Hz
$f_{timetick}$	inkrementier Taktfrequenz des Timers	Hz
n	Anzahl	
$Prescaler$	Vorteiler	
T_{PWM}	Periodendauer	s

1 Einleitung

Wir erleben derzeit ein explosionsartiges Wachstum bei der Entwicklung und dem Einsatz von eingebetteten Systemen und persönlichen mobilen Computersystemen, die eine zunehmende Unterstützung mehrerer Standards erfordern. [1, 2]

Aufgrund steigender Anforderungen an eingebettete Systeme entwickeln die Hersteller fortlaufend neue Konzepte zur Umsetzung. STMicroelectronics kombiniert auf der Mikroprozessor-Plattform „STM32MP1“ die Eigenschaften von Mikroprozessoren und Mikrocontrollern. [3, 4, 5]

Diese Arbeit bewertet die Einsatzmöglichkeit der Mikroprozessor-Plattform „STM32MP157C-DK2“ im Verhältnis zu dem Implementierungsaufwand am Beispiel des Einsatzes in einem Miniaturfahrzeug.

Das in dem Entwicklungsprojekt entstandene Miniaturfahrzeug trägt den Namen „BumbleBee“. In Abbildung 1.1 ist das Miniaturfahrzeug mit der integrierten Mikroprozessor-Plattform „STM32MP157C-DK2“ zu sehen. [6]

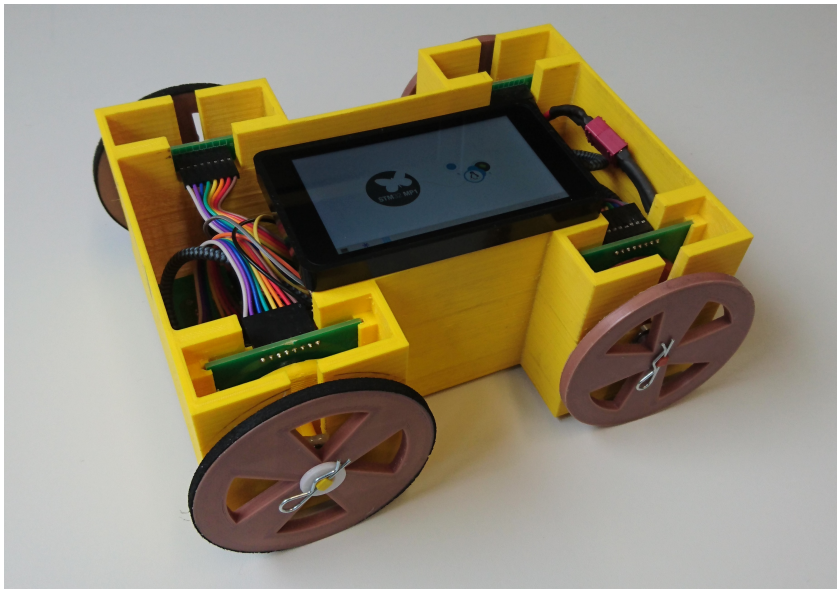

Abbildung 1.1: Miniaturfahrzeug „BumbleBee“

1.1 Motivation

Eingebettete Systeme werden heute in der industriellen Steuerung, in der Luft- und Raumfahrt, der Automobilindustrie, der Netzwerkkommunikation, dem Gesundheitswesen und der Unterhaltungselektronik eingesetzt. [7]
Dabei müssen eingebettete Systeme eine Vielzahl von Anforderungen erfüllen. [3]
Die Mikroprozessor-Plattform „STM32MP1“ vereint die Eigenschaften eines Embedded Linux Betriebssystems mit der eines Echtzeitfähigen Systems. [8]
Embedded Linux ist ein Kern-basiertes, voll speichergeschütztes, Multitasking-fähiges Multiprozess-Betriebssystem, das unterschiedliche Computer-Hardware (X86, Alpha, Sparc, MIPS, PPC, ARM, NEC, MOTOROLA usw.) unterstützt. Der Linux Quellcode des Betriebssystems ist frei zugänglich und kann auf die eigenen Bedürfnisse und das eigene System angepasst werden. Ein wirtschaftlicher Vorteil von Embedded-Linux ist die Vielzahl von frei verfügbaren Anwendungen. [7]

2 Grundlagen

2.1 Machbarkeit

Begriffsklärung: Machbarkeit

Als Machbarkeitsstudie wird die Überprüfung der Durchführbarkeit eines Lösungsansatzes für ein Projekt bezeichnet. Machbarkeitsstudien werden gerade dann erhoben, wenn es Zweifel an der Durchführbarkeit eines Projektes gibt. [9]
Der Projektmanagementprozess der DIN 69901-2:2009-01 [10, S. 24] enthält den Prozess D.8.3 namens „(Mindeststandard) Machbarkeit bewerten“.

Für diesen Prozess ist entscheidend, dass es einen Vorgängerprozess gibt. In dem Vorgängerprozess wird der Vertragsinhalt mit dem Kunden festgelegt [10]. Der Kunde ist die „Smart Mechatronics GmbH“. Der Vertragsinhalt wird im Kapitel 3 beschrieben.

Machbarkeit bewerten

Zweck und Hintergrund

Zweck und Hintergrund der Bewertung der Machbarkeit eines Projektes ist eine Entscheidung über das weitere Vorgehen im Projekt. Die Bewertung findet auf der Basis der eigenen Stärken und Schwächen statt. Dabei werden Chancen und Risiken abgewägt. [10]

Prozessbeschreibung

Um die Machbarkeit zu prüfen, werden alle gesammelten Informationen ausgewertet und mit den Projektzielen verglichen. Dabei ist entscheidend, ob die Projektziele mit den zur Verfügung stehenden Mitteln in der vorgegebenen Zeit umgesetzt werden können. [10]
In der DIN 69901-2:2009-01 [10] ist festgehalten, welche *Inputs*, welche Prozessmanagement (PM)-Methoden und welche *Outputs* die Machbarkeit bewerten.

- ***Input***:
  Der *Input* ist bei diesem Projekt das Projektziel und das Projektumfeld.
- **Prozessmanagement-Methode**:
  Als PM-Methode wird die *Strengths Waknesses Opportunities Threats (SWOT)* aufgelistet.
- ***Output***:
  Der *Output* ist die Bewertung der Machbarkeit.

2.2 Die Mikroprozessor-Plattform STM32MP1

Technische Spezifikationen [4]:

- *Cores*
  - 32 bit dual-core ARM-Cortex-A7 bis zu 650MHz
  - 32 bit Floating Point Unit (FPU)/Microprocessing Unit (MPU) ARM-Cortex-M4 bis zu 209MHz

  4 GB DDR3L, 16 bit, 533MHz
- Bis zu 29 *Timer* und 3 *Watchdog*
- Bis zu 37 Kommunikations Peripherien
- 6 analoge Peripherien
- Bis zu 176 interrupt-fähige General-Purpose Input/Outputs (GPIO)

2.3 Mikrocontroller

Mikrocontroller haben Ähnlichkeiten mit Mikroprozessoren. Der grundlegende Unterschied besteht darin, dass Mikrocontroller im Gegensatz zu Mikroprozessoren keine *North-* und keine *Southbridge* benötigen. [5]
Die Schnittstellen zu den Peripherien sind bereits im Mikrocontroller integriert. Mikrocontroller haben beispielsweise verschiedene unabhängige integrierte Schnittstellen, *Timer*, Arbeitsspeicher, Flashspeicher, ...

In Abbildung 2.1 ist die Implementierung des Cortex-M4 Prozessors zu erkennen.

Figure 1. STM32 Cortex-M4 implementation

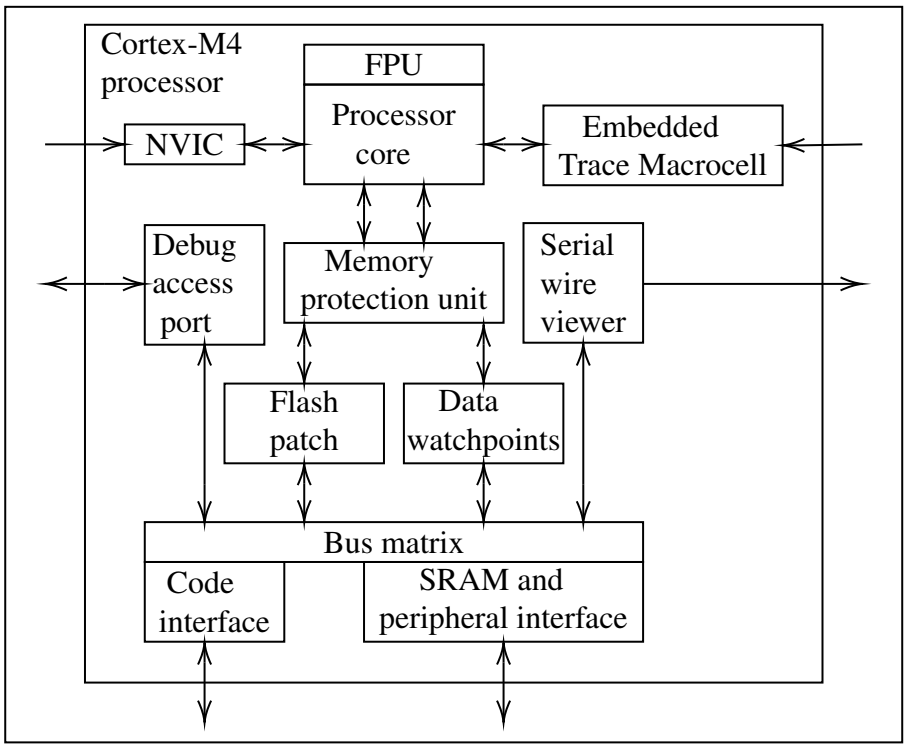

Abbildung 2.1: STM32 Cortex-M4 Implementierung[11, S. 13]

Der Cortex-M4-Prozessor ist ein 32 bit-Prozessor, der für den Mikrocontrollermarkt entwickelt wurde. Der Cortex-M4-Prozessor basiert auf einem Prozessorkern mit einer dreistufigen Harvard-*Pipeline*-Architektur [12] (Abschnitt 2.11), wodurch sich dieser für Embedded-Anwendungen eignen soll. [11]

Auf dem Mikrocontroller läuft ein *Bare-Metal-System*.

Der Begriff *Bare-Metal*-System wird in den meisten Fällen für Computersysteme genutzt, die ohne Betriebssystem genutzt werden. [5, S. 20]

Bare-Metal-Systeme verfügen oft nicht über die Ressourcen, um ein vollständiges Betriebssystem auszuführen. Vor allem Mikrocontroller-Steuerungen werden als *Bare-Metal*-Systeme bezeichnet. *Bare-Metal*-Systeme sind in der Regel für ganz bestimmte Aufgaben, wie zum Beispiel die Steuerung einer Waschmaschine oder eines Kaffeevollautomaten, konstruiert. [5]

Die Anwendung des Mikrocontroller wird über einen *Bootstrap Loader* in den Speicher des Mikrocontrollers geladen. [13]

2.4 Mikroprozessor

Mikroprozessoren sind üblicherweise hochintegrierte Halbleiterbausteine mit mehreren Milliarden Transistoren, die sich in einem Gehäuse befinden[5, S. 21].
Man unterscheidet Mikroprozessoren nach zwei Architekturen:

1. Reduced Instruction Set Computer (RISC) = Reduced Instruction Computer
2. Complex Instruction Set Computer (CISC) = Complex Instruction Set Computer

Beide Architekturen sind im Stande, schnelle binäre Rechenoperationen durchzuführen [5]. Für den Einsatz von Mikroprozessoren sind weitere Komponenten erforderlich [5]. In der Literatur werden diese Komponenten in *Northbridge* und *Southbridge* unterschieden.

1. Die *Northbridge* ist die Anbindung an die »schnelle« externe Hardware.
   Beispiele:
   Random-Access Memory (RAM)
   Grafikkarten
2. Die *Southbridge* ist die Anbindung an die »langsame« externe Hardware.
   Beispiele:
   Peripheral Component Interconnect (PCI)-Bus
   Basic Input Output System (BIOS)
   externe Schnittstellen, zum Beispiel Universal Serial Bus (USB)

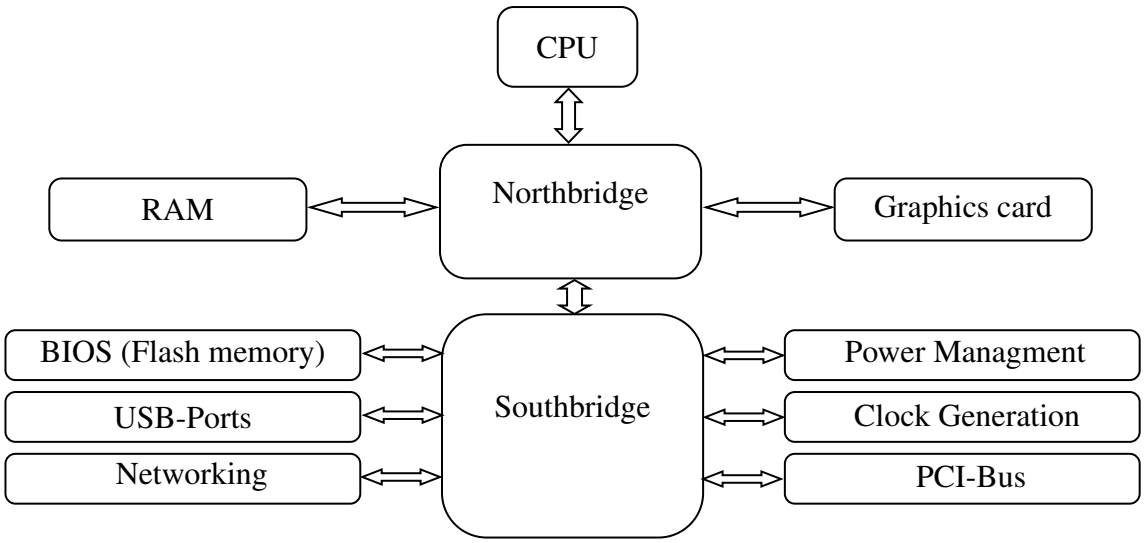

Abbildung 2.2: Veranschaulichung von *Northbridge* und *Southbridge*.

Auf der Central Processing Unit (CPU) der MPU (dual Cortex-A7) kann ein *Embedded* Betriebssystem ausführt werden. Der Hersteller STMicroelectronics bietet für die Nuzung der MPU das auf die Plattform zugeschnittene *Embedded* Betriebssystem *OpenSTLinux*. [14]

2.5 Das *Embedded* Betriebssystem OpenSTLinux

Als *Embedded* Betriebssysteme werden solche bezeichnet, die speziell auf die Ressourcen der System-Hardware angepasst wurden [5]. Auf vielen Minicomputern wie dem Raspberry Pi [15], dem BeagleBone Black [16] und dem Cubietruck [17] wird *Embedded* Linux [18] bevorzugt eingesetzt[5]. Der Quelltext von *Embedded* Linux ist frei verfügbar, was begründet, warum dieses *Embedded* Betriebssystem bevorzugt eingesetzt wird [5]. Es gibt aber noch eine große Anzahl von anderen Embedded Betriebssystemen, wie zum Beispiel Windows CE [19], Emdedix [20], eCos [21] oder VxWorks [22], wovon Emdedix, eCos und VxWorks zu den sogenannten echtzeitfähigen Betriebssystemen (Real Time Operating System (RTOS)) gehören. *Embedded* Linux gehört nicht zu den echtzeitfähigen Betriebssystemen. [5]
Da das auf dem Mikroprozessor laufende Betriebssystem kein echtzeitfähiges Betriebssystem ist, können Aufgaben, die Echtzeit benötigen, auf das echtzeitfähige Bare-

Metal-System ausgelagert werden. [8]

2.6 Linux

Im folgenden Unterkapitel wird ein Überblick zu der Architektur einer Linux-Distribution gegeben.

Linux ist ein sehr komplexes Betriebssystem. Um die Komplexität zu überblicken, wird der Kernel (Betriebssystemkern) in *Layer* (Schichten) unterteilt [5]. Diese *Layer* enthalten wiederum viele verschiedene Programme. Der schematische Aufbau des Linux-Systems ist in Abbildung 2.3 zu sehen [5].

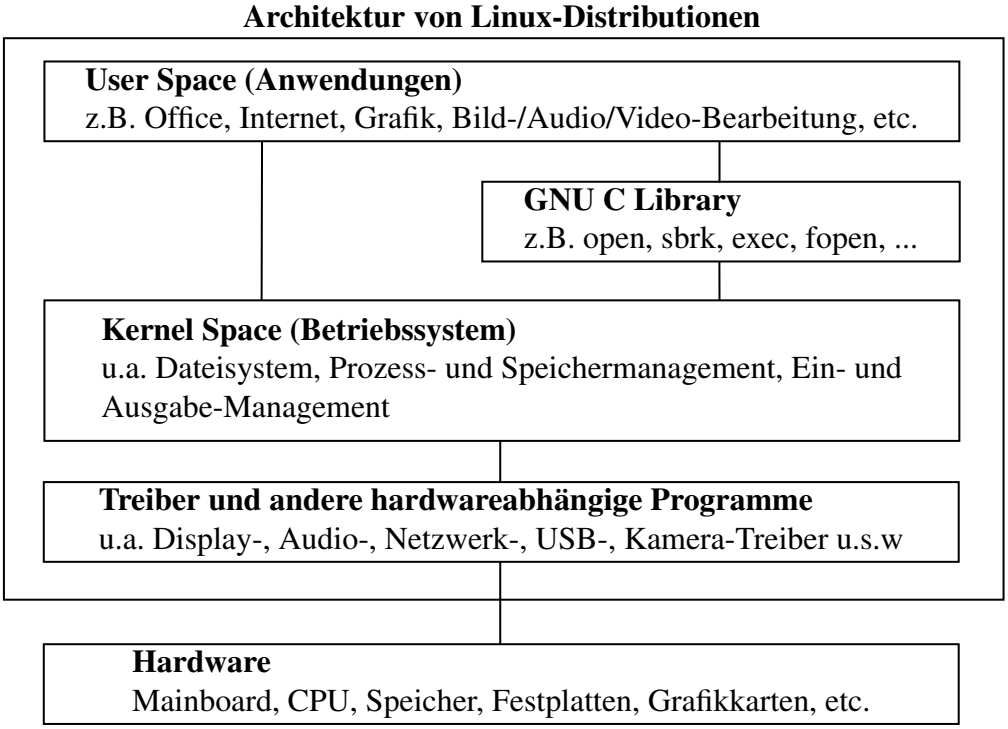

Abbildung 2.3: Architektur von Linux-Distributionen [5, S. 23]

Das Betriebssystem OpenSTLinux wird über den *Open-Source Bootloader* U-Boot geladen. [23]

2.7 U-Boot

Das Universal Boot Loader (U-Boot) steht für *The Universal Bootloader*. U-Boot ist ein *Bootloader* für eingebettete *Boards*. Die Entwicklung von U-Boot ist eng mit der Entwicklung von Linux verbunden. Teile des U-Boot-Quellcodes stammen aus dem Linux-Devicetree (DT). Bei der Entwicklung des U-Boot wurde darauf geachtet, dass insbesondere das Booten von Linux-*Images* möglich ist. [24]
U-Boot lädt und prüft die *Images* von Kernel DT und RAM-*disk*, darauffolgend übergibt U-Boot die Kontrolle an den Linux-Kernel oder die Zielanwendung. [23]

2.8 *Devicetree*

Der DT ist ein Konstrukt zur Beschreibung der Systemhardware [25]. Ein DT hat eine Baumdatenstruktur mit Knoten. Diese Knoten beschreiben die in dem System enthaltenen Geräte. Jeder Knoten hat Eigenschafts-/Wertpaare, die die Eigenschaften der dargestellten Geräte beschreiben. Jeder Knoten hat einen übergeordneten Knoten mit Ausnahme des Wurzelknotens, der keinen übergeordneten Knoten hat. Ein Bootprogramm lädt den DT in den Speicher des Client-Programms und übergibt einen Zeiger auf den DT an den Client [25]. Das Beispiel eines DT ist in Abbildung 2.4 dargestellt.

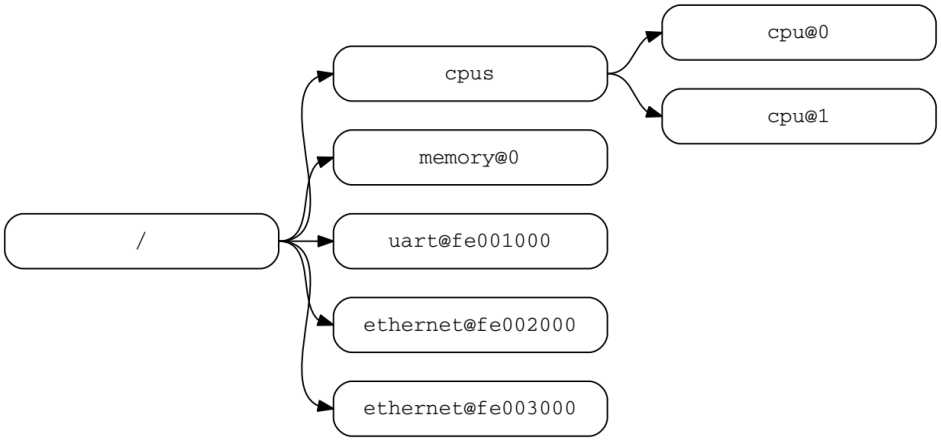

Abbildung 2.4: Beispiel DT [25, S. 8]

2.9 Verzeichnisstruktur von Linux/OpenSTLinux

Die Abbildung 2.5 zeigt die Struktur des Wurzelverzeichnis der OpenSTLinux Distribution.

```
root@stm32mp1:~# ls /
bin   dev  home  lost+found  mnt  proc  sbin  tmp  var
boot  etc  lib   media       opt  run   sys   usr  vendor
```

Abbildung 2.5: Wurzelverzeichnisstruktur

- **/bin**
  In dem Verzeichnis /bin befinden sich alle Dienstprogramme. Diese stehen allen Benutzern in der Shell (Terminal) zur Verfügung. [5]
- **/boot**
  In dem Verzeichnis /boot befinden sich alle Dateien, die zum Booten des Systems zwingend erforderlich sind. [5]
  Zum Beispiel das Linux-*Image* und die *Device Tree Blobs*. [26]
- **/dev**
  In dem Verzeichnis /dev befinden sich Gerätedateien. Geräte werden in Linux wie Dateien behandelt. [5]
- **/etc**
  In dem Verzeichnis /etc sind Dateien enthalten, die zur Konfiguration von Programmen dienen. [5]
- **/home**
  In dem Verzeichnis /home wird für jeden *User* ein privater Bereich angelegt. [5]
- **/lib**
  In dem Verzeichnis /lib sind wichtige Funktionsbibilioteken. [5]
- **/lost+found**
  In dem Verzeichnis /lost+found werden Dateien vom System gespeichert, die aus einem beliebigen Grund *corrupt* (beschädigt) sind. Können diese Daten von dem System nicht mehr repariert werden oder lassen diese sich nicht mehr zuordnen, werden sie vom System in das Verzeichnis /lost+found geschrieben. [5]

- **/mnt**
  In dem Verzeichnis /mnt werden Beschreibungen gespeichert, an welcher Stelle temporär genutzte Geräte oder Festplattenpartitionen im Dateisystem (*mount*) eingehängt
  wurden. [5]
- **/opt**
  In dem Verzeichnis /opt werden zusätzliche Programme gespeichert, die nicht zwingend für Linux erforderlich sind. [5]
- **/proc**
  Das Verzeichnis /proc wird als Schnittstelle zum Kernel genutzt. Jeder Prozess hat über dieses Verzeichnis die Möglichkeit, mit dem Kernel zu kommunizieren. [5]
- **/run**
  In dem Verzeichnis /run werden Daten gespeichert, die für den Betrieb des Linux-Systems erforderlich sind. [5]
- **/sbin**
  Das Verzeichnis /sbin stellt Programme zur Verfügung, die nur vom Systemadministrator ausgeführt werden können (*user: root*). [5]
- **/sys**
  Das Verzeichnis /sys ist ein virtuelles Verzeichnis, das dazu dient, eine Schnittstelle zwischen System und Kernel herzustellen. [5]
- **/tmp**
  Das Verzeichnis /tmp speichert temporäre Daten. Es wird beim Neustarten des Linux-Systems geleert. [5]
- **/usr**
  Das Verzeichnis /usr wird für *Unix System Resources* verwendet. [5]
- **/var**
  Das Verzeichnis /var beinhaltet Daten, die sich häufig ändern. Zum Beispiel Log-Dateien, die den Zustand des Systems dokumentieren. [5]
- **/vendor**
  Das Verzeichnis /vendor hat Ähnlichkeiten mit dem /lib Verzeichnis, mit dem Unterschied, dass in /vendor *Third-Party-Framework*-Bibliotheken gespeichert sind. [5]

2.10 Definition *Embedded Systems*

Eingebettete Systeme sind informationsverarbeitende Systeme, die in größeren Produkten eingebettet sind [3].
Eingebettete Systeme sind also zum Beispiel in Autos, Zügen, Flugzeugen und Telekommunikations- oder Fertigungsanlagen integriert. Eingebettete Systeme haben Eigenschaften wie Echtzeitbeschränkungen, Anforderungen an die Zuverlässigkeit und an die Effizienz. Oft sind eingebettete Systeme mit der Physikalische Umgebung durch Sensoren und Aktoren verbunden. [3]

Da viele eingebettete Systeme direkte Einwirkung auf die Umwelt haben, müssen diese Systeme folgende Anforderungen erfüllen [3].

Anforderungen [3]:

1. Zuverlässigkeit: Zuverlässigkeit ist die Wahrscheinlichkeit, dass ein System nicht ausfallen wird. [3]
2. Wartbarkeit: Die Wartbarkeit ist die Wahrscheinlichkeit, dass ein ausgefallenes System innerhalb eines bestimmten Zeitrahmens repariert werden kann. [3]
3. Verfügbarkeit: Verfügbarkeit ist die Wahrscheinlichkeit, dass das System verfügbar ist. [3]

   Die Zuverlässigkeit und die Wartbarkeit müssen hoch sein, damit eine hohe Verfügbarkeit erreicht wird.
4. Sicherheit: Dieser Begriff beschreibt die Eigenschaft, dass ein ausgefallenes System keinen Schaden anrichtet. [3]
5. *Security*: Dieser Begriff beschreibt die Eigenschaft, dass vertraulichen Daten eine sichere und zuverlässige Kommunikation gewährleistet wird. [3]

Eingebettete Systeme müssen in mehreren Hinsichten effizient sein [3].

1. Energieeffizienz
2. Code-Größe
3. Laufzeiteffizienz
4. Gewicht
5. Kosten

Viele eingebettete Systeme erfüllen Echtzeitbedingungen. Nicht abgeschlossene Berechnungen in festen Zeitrahmen können dafür sorgen, dass dem Benutzer Schaden

zugefügt wird oder dass das System gravierende Datenverluste erleidet [3]. Eine Zeitbeschränkung wird dann als hart bezeichnet, wenn es zu einer Katastrophe kommen kann, wenn diese Zeitbeschränkung nicht eingehalten wird [27]. Alle anderen Zeitbeschränkungen werden als weich bezeichnet.

Eingebettete Systeme werden als Hybridsystem bezeichnet, wenn das System analoge und digitale Komponenten enthält [3]. In analogen Komponenten herrschen kontinuierliche Signalwerte in kontinuierlicher Zeit. In digitalen Komponenten herrscht zu diskreter Zeit ein diskreter Signalwert. [3]

Für gewöhnlich sind eingebettete Systeme reaktive Systeme. Sie können wie folgt definiert werden:
Ein reaktives System ist ein System, das in ständiger Interaktion mit seiner Umgebung ist und einen von dieser Umgebung bestimmten Schritt ausführt. [28]

2.11 Harvard-Architektur

Um die Vorteile und Eigenschaften der Harvard-Architektur zu beschreiben, ist ein Einstieg über das „Von Neumann“ Rechenmodell und die „Von Neumann“-Architektur sinnvoll.
Das „Von Neumann“ Rechenmodell (Abbildung 2.6) besteht, wie in [12] beschrieben, aus den folgenden Komponenten.

- Eingabe-Einheit
- Operationswerk (Rechenwerk)
- Speicher
- Steuerwerk (Leitwerk)
- Verbindungsleitungen
- Ausgabe-Einheit

Eingabe- und Ausgabe-Einheit werden für die Kommunikation des Rechners mit der Außenwelt genutzt. Diese Einheiten sind dazu da, um die Kommunikation zwischen Mensch und Maschine zu ermöglichen. Das Operationswerk führt logische und arithmetische Operationen durch. Es wird auch Rechenwerk genannt. Der Speicher wird unterteilt in Haupt- und Arbeitsspeicher. Der Speicher enthält die Informationen

für die Steuerung des Prozessors. Das Steuerwerk ist für die Programmablaufsteuerung zuständig. Es wird auch Leitwerk genannt. Das Steuerwerk steuert das ganze Mikroprozessor-System mit Steuersignalen. Die Verbindungsleitungen sorgen für die Vernetzung der einzelnen Komponenten. [12]

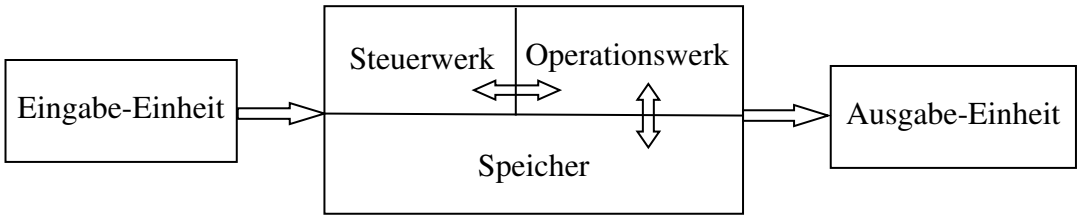

Abbildung 2.6: Rechenmodell nach „Von Neumann“

Die Erweiterung des Rechenmodells nach „Von Neumann“ wird „Von Neumann Struktur“ genannt (Abbildung 2.7). Die Verbindungen der Einzelkomponenten werden in Bussystemen gebündelt. Dabei werden Daten-, Adress- und Steuerbus getrennt. Der Prozessor adressiert über den Adressbus Speicherplatz auf dem Hauptspeicher und den Ein- und Ausgabeeinheiten, auf die zugegriffen werden soll. Der Datenbus wird für die Datenübertragung vom und zum Speicher oder von und zu den Ein- und Ausgabeeinheiten verwendet. Der Steuerbus dient dem Übertragen von Steuersignalen, die den Programmablauf steuern. [12]

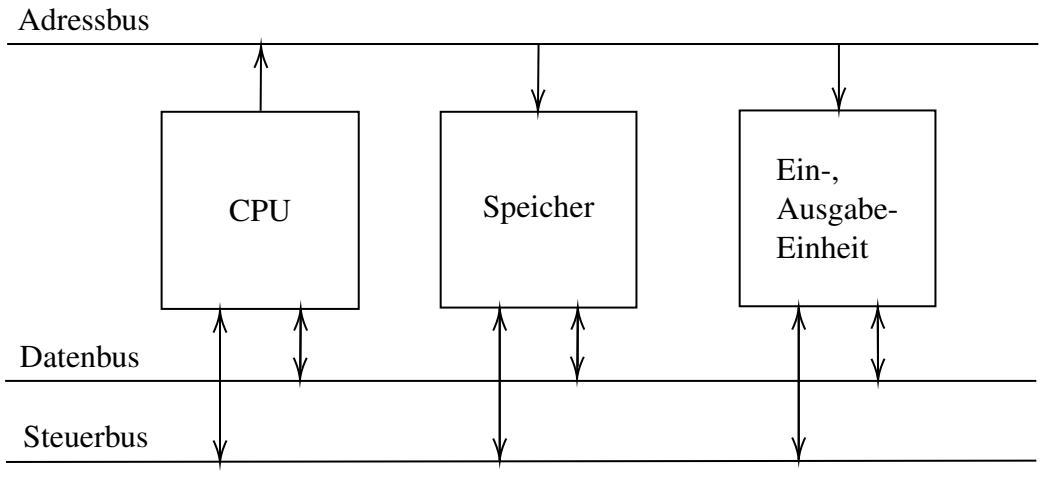

Abbildung 2.7: Struktur des „Von Neumann Rechners“

Für die Bearbeitung eines Befehles sind mehrere streng sequenzielle ablaufende Schritte notwendig [12]:

1. Der Prozessor adressiert den Adressbus.
2. Der Prozessor liest ein Bitmuster aus dem Datenbus, das einen Befehl repräsentiert.
3. Der Prozessor entschlüsselt das Bitmuster und führt alle erforderlichen Teiloperationen aus (zum Beispiel arithmetische Operationen oder das Zwischenspeichern von Daten in internen Registern).

Daten und Programmcode werden aus dem gleichen Speicher über den Datenbus transportiert. Die Inhalte von jeder Speicherzelle sind als Befehl, Daten oder Adressen interpretierbar. Die „Von-Neumann-Architektur“ hat den Nachteil, dass es zu einem Engpass auf dem Datenbus kommt [12]. Dort werden Befehle und Daten von und zum Arbeitsspeicher nacheinander gesendet. Dieser Engpass wird als „Von-Neumann-Flaschenhals“ bezeichnet. [12]

Dieser Flaschenhals-Engpass, wird durch das Nutzen von getrennten Datenbussystemen bei der Harvard-Architektur (Abbildung 2.8) vermieden [12]. Dadurch können bei der Harvard-Architektur Programm- und Datenspeicher parallel zueinander arbeiten.

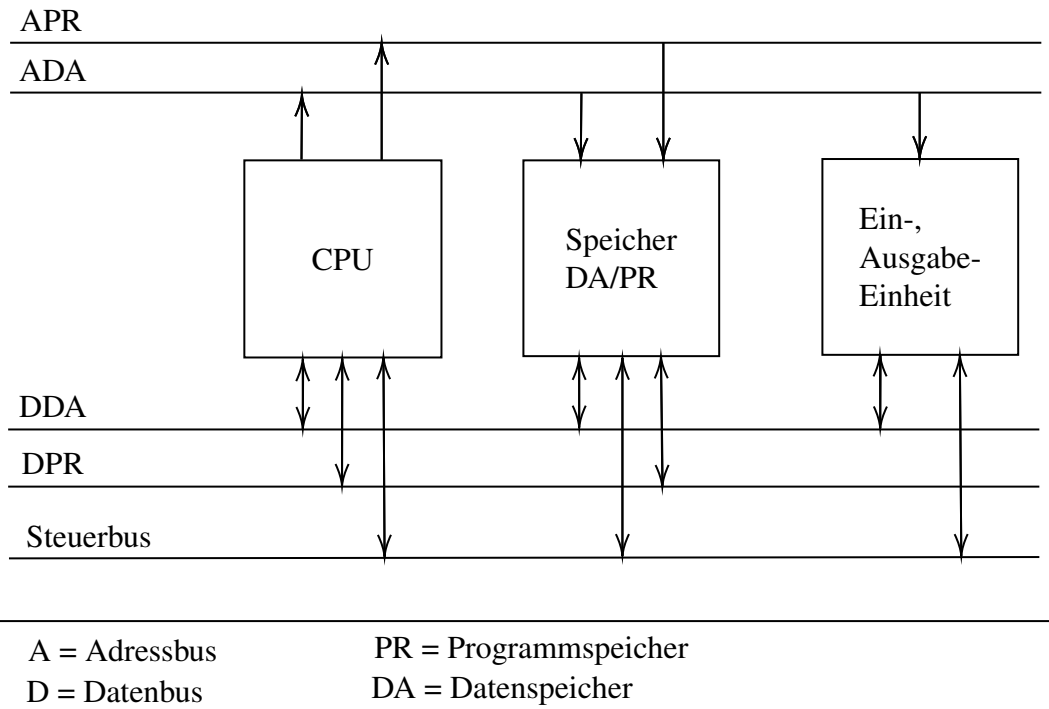

Abbildung 2.8: Rechnermodell mit Harvard-Struktur

Bei der Harvard-Struktur existieren getrennte Speicher für Programme und Daten [12]. Der Adressbus des Programmspeichers (APR) dient dazu, Programmspeicher (PR) zu adressieren. Der Adressbus des Datenspeichers (ADA) ist dafür da, Datenspeicher (DA) zu adressieren. Der Datenbus des Datenspeichers (DDA) verbindet die CPU mit dem DA. Der Datenbus des Programmspeichers (DPR) verbindet die CPU mit dem PR. Der Steuerbus bleibt so, wie er schon bei der „Von-Neumann-Struktur“ war. Das Trennen der Adressbusse ist eine notwendige Voraussetzung, damit Daten und Programmcode zeitlich getrennt voneinander bearbeitet werden können [12].

2.12 Interprozessor-Kommunikation

Die Kommunikation zwischen den Prozessoren basiert auf dem Remote Processor Messaging (RPMsg)-*Framework* und den *Mailbox-Mechanismen* [29].

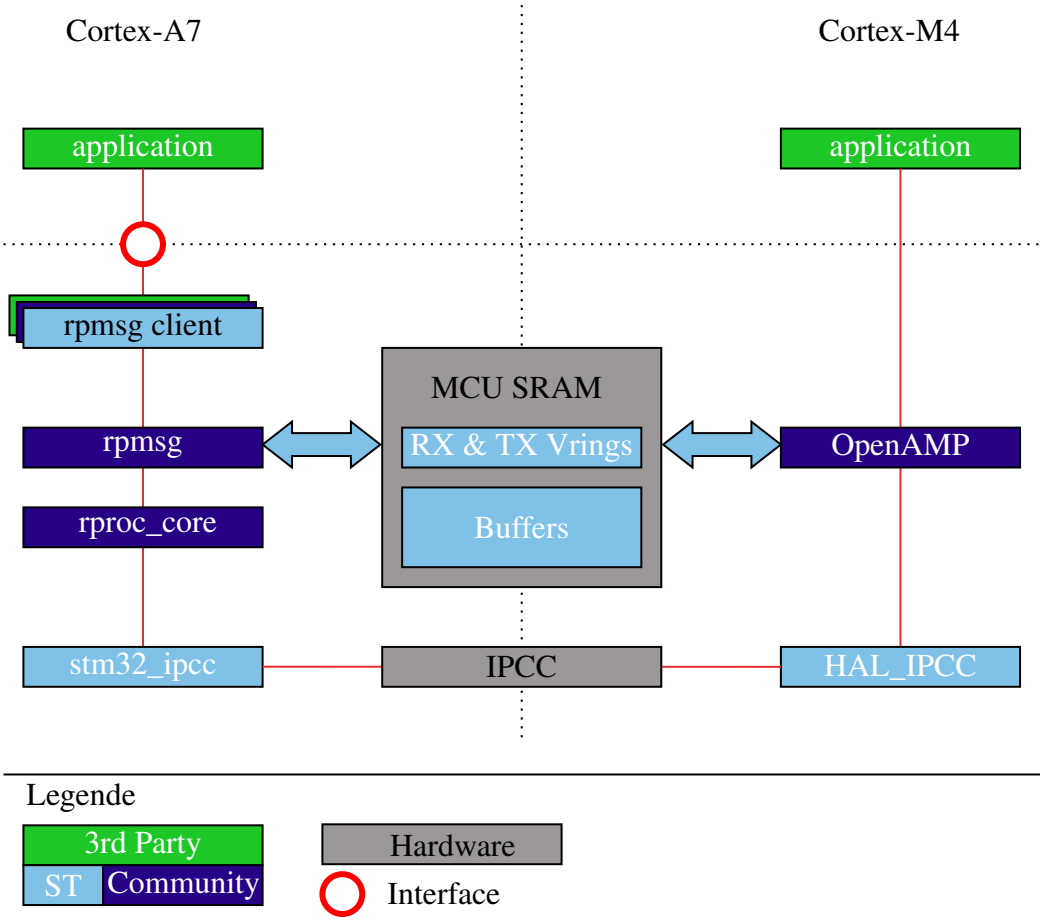

Abbildung 2.9: Struktur der Interprozessor-Kommunikation [30]

Auf dem Cortex-A7: [29]

- Das Remote Processor (RemoteProc)-Framework ist für die Aktivierung des Inter-Process Communication (IPC) auf Linux-Seite zuständig, basierend auf den in der *Firmware Resource Table* verfügbaren Informationen.
- Der RPMsg-Dienst wird durch das RPMsg-Framework implementiert.
- Der *Mailbox*-Dienst wird durch den *Mailbox*-Treiber stm32_ipcc implementiert.

Auf dem Cortex-M4: [29]

- Der RPMsg-Dienst wird durch die Open Asymmetric Multi Processing (OpenAMP)-Bibliothek implementiert.
- Der *Mailbox*-Dienst wird durch den HAL_IPCC-Treiber implementiert.

2.13 Das *Framework* OpenAMP für *Asynchronous Multiprocessing*

Bei verteilten Systemen mit mehreren Prozessoren treten Schwierigkeiten, wie die Verteilung des Speichers, das Festlegen eines *Shared-Memorys* zum Austausch von Informationen zwischen den Prozessoren, die Verteilung von System-Ressourcen, die Verteilung und das Managen von Interruptroutinen, so wie viele andere Probleme auf [31]. Um diesen Problemen entgegenwirken zu können, wird auf das *Open Source Framework* OpenAMP [32] zurückgegriffen.

OpenAMP ist eine Entwicklung der Multicore Association (MCA) [33] mit dem Ziel, die Interaktion von Betriebssystemen innerhalb eines breiten Spektrums komplexer homogener und heterogener Architekturen und asymmetrische *Multiprocessing*-Anwendungen zu ermöglichen, um die durch die *Multicore*-Konfiguration gebotene Parallelität zu nutzen. Diese MCA-Arbeitsgruppe konzentriert sich auf die Standardisierung der Application-Programming-Interface (API)s, die Bereitstellung einer detaillierten Dokumentation für die Spezifikation und die Erweiterung der Funktionalität von OpenAMP. [33]

2.14 Pin-Konfiguration mit STM32CubeMX

STM32CubeMX ist ein grafisches Software-Konfigurationswerkzeug [34], das die Generierung von C-Initialisierungscodes durch einen grafischen Assistenten ermöglicht. In STM32CubeMX [34] können *Pinout*-, *Clock*-, und Projekt-Einstellungen gewählt werden. Aus diesen Einstellungen wird dann ein Integrated Development Environment (IDE) spezifisches Projekt mit den gewählten Einstellungen generiert. STM32CubeMX reduziert durch das Generieren von IDE spezifischen Projekten, in dem die *Pin*-Konfiguration, die *Driver*-Konfiguration, die *Middleware*-Konfiguration und die Hardware-Initialisierung enthalten sind, den Zeitaufwand und die Kosten für die Entwicklung [34]. Für MPUs der Serie STM32MP1 können auch DT-Dateien generiert werden [34].

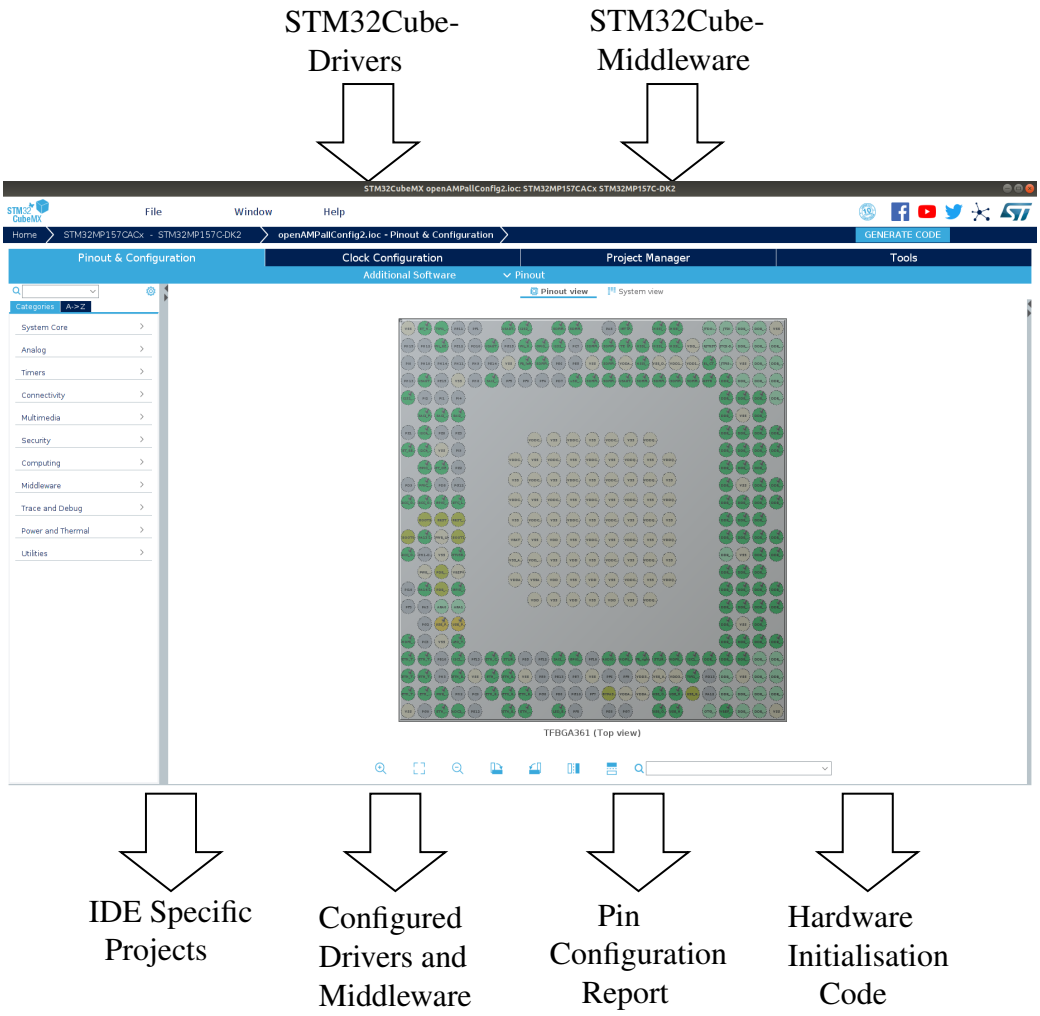

Abbildung 2.10: STM32CubeMX

2.15 Robot Operating System

Das Robot Operating System (ROS) ist ein System, das Entwicklern von Roboteranwendungen Bibliotheken und Werkzeuge zur Verfügung stellt. ROS beinhaltet Visualisierungen, Bibliotheken, Hardware Abstraktion, Gerätetreiber, Nachrichtenvermittlung, Paketverwaltung und andere Komponenten. [35]

Im Kontext dieser Bachelorarbeit wird ROS für die Kommunikation und Parameter-Übergabe im lokalen Netzwerk genutzt. Ein ROS-Netzwerk ist so organisiert, dass ein Master mit *n* Knoten verbunden ist (Abbildung 2.11). Die Knoten veröffentlichen und abonnieren Topics. Daten können durch vordefinierte und selbst erstellte Nachrichtentypen übertragen werden. [36]

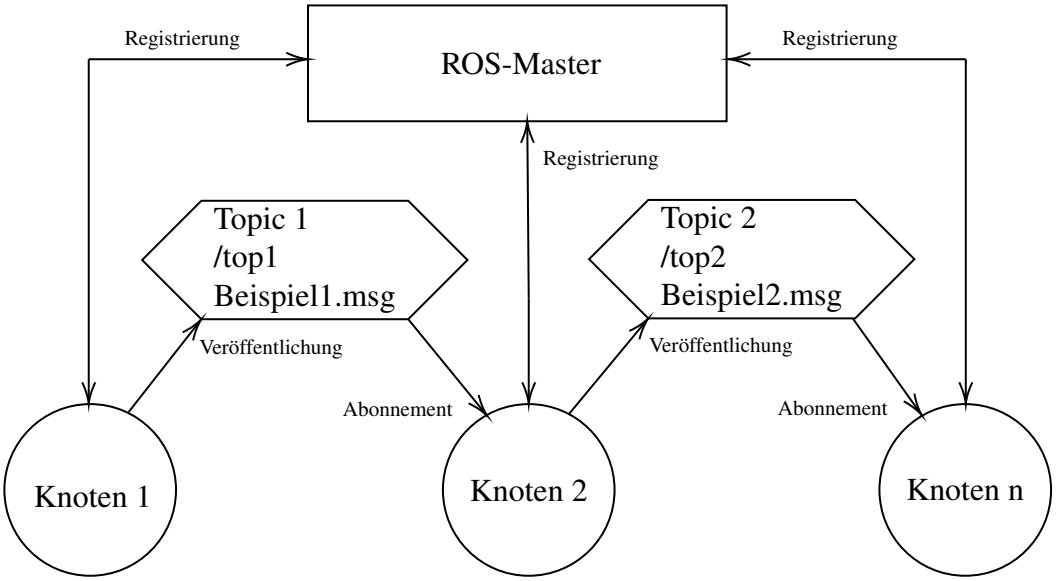

Abbildung 2.11: Beispiel eines ROS-Netzwerkes

3 Systemkonzipierung

In das Miniaturfahrzeug BumbleBee wird die Mikroprozessor-Plattform STM32MP157C-DK2 der Firma STMicroelectronics integriert (Abbildung 1.1). Die Mikrocontroller-Plattform enthält zwei ARM Cortex-A7 Prozessoren und einen ARM-Cortex-M4 Prozessor. [8]

Das Miniaturfahrzeug BumbleBee stammt aus einem Entwicklungsprojekt. Der in dem Entwicklungsprojekt verwendete Arduino Uno R3 wird durch die Mikroprozessor-Plattform STM32MP157C-DK2 ersetzt. Abbildung 3.1 zeigt auf der linken Seite die MPU von der Unterseite und auf der rechten Seite die Motorcontroller-Platine des BumbleBees von der Oberseite sowie die Signalverbindungen zwischen den beiden Komponenten. [6]

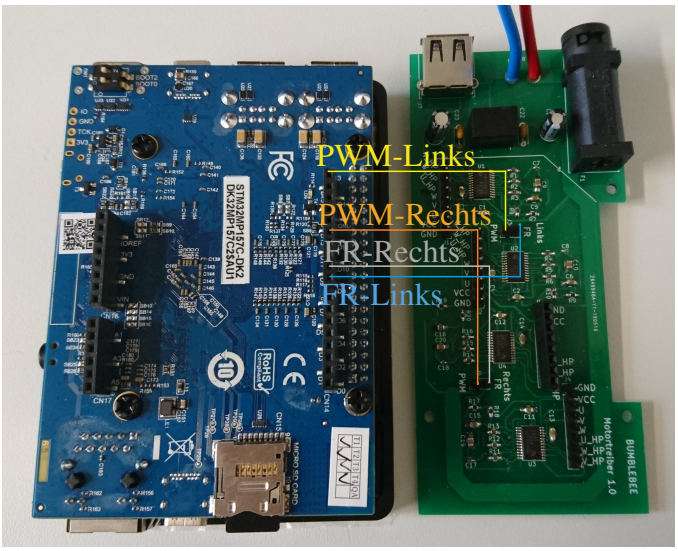

Abbildung 3.1: Unterseite der MPU mit Signalverbindungen zu der Motorcontroller-Platine des BumbleBee

3.1 Anforderungen

Die Anforderungen, die an die BumbleBee-Firmware gestellt werden, werden aus dem Projektauftrag Punkt A.1.1 und dem aus dem Projektauftrag entstandenem Lastenheft Punkt A.1.2 abgeleitet. Im Folgenden werden diese Anforderungen zusammengefasst [37, 38]. Das Festlegen von Vertragsinhalten ist nach DIN 69901-2:2009-01 [10, S. 24], wie in Kapitel 2.1 beschrieben, der Prozess, der vor dem Prozess D.8.3 der »(Mindeststandard) „Machbarkeit bewerten“ stattfinden muss.

Synchronisation zwischen Linux- und Real-Time-Core (Anforderung 01)
Es ist eine Kommunikation zwischen dem Linux Betriebssystem (Cortex-A7) und dem *Bare-Metal*-System (Cortex-M4) möglich.

Autoload des A7-*Executable* und der M4- *Firmware* (Anforderung 02)
Nach dem Booten der MPU werden die Programme (Cortex-A7: *Executable*, Cortex-M4: *Firmware*) zur Steuerung des BumbleBees automatisch ausgeführt.

Betrieb von Bumblebee bei WLAN-Anbindung (Anforderung 03)
Geräte, die mit dem Wireless Local Area Network (WLAN)-Hotspot der MPU verbunden sind, können den BumbleBee steuern.

Erprobung des Flash-Vorgangs per WLAN (Anforderung 04)
Die *Firmware* des Cortex-M4 Prozessors wird per WLAN auf das System geflasht und geladen.

Matlab Simulink Code Generierung (Anforderung 05)
Es wird erprobt, Code, der in Matlab mit dem Simulink Coder generiert wurde, für die Architektur des Cortex-A7 Prozessors *cross* zu kompilieren.

3.2 Umfeldmodell

Das Umfeldmodell wird nach [39, S. 91f.] erstellt. Das Umfeldmodell (Abbildung 3.2) zeigt, welche Kommunikationsschnittstellen implementiert werden.

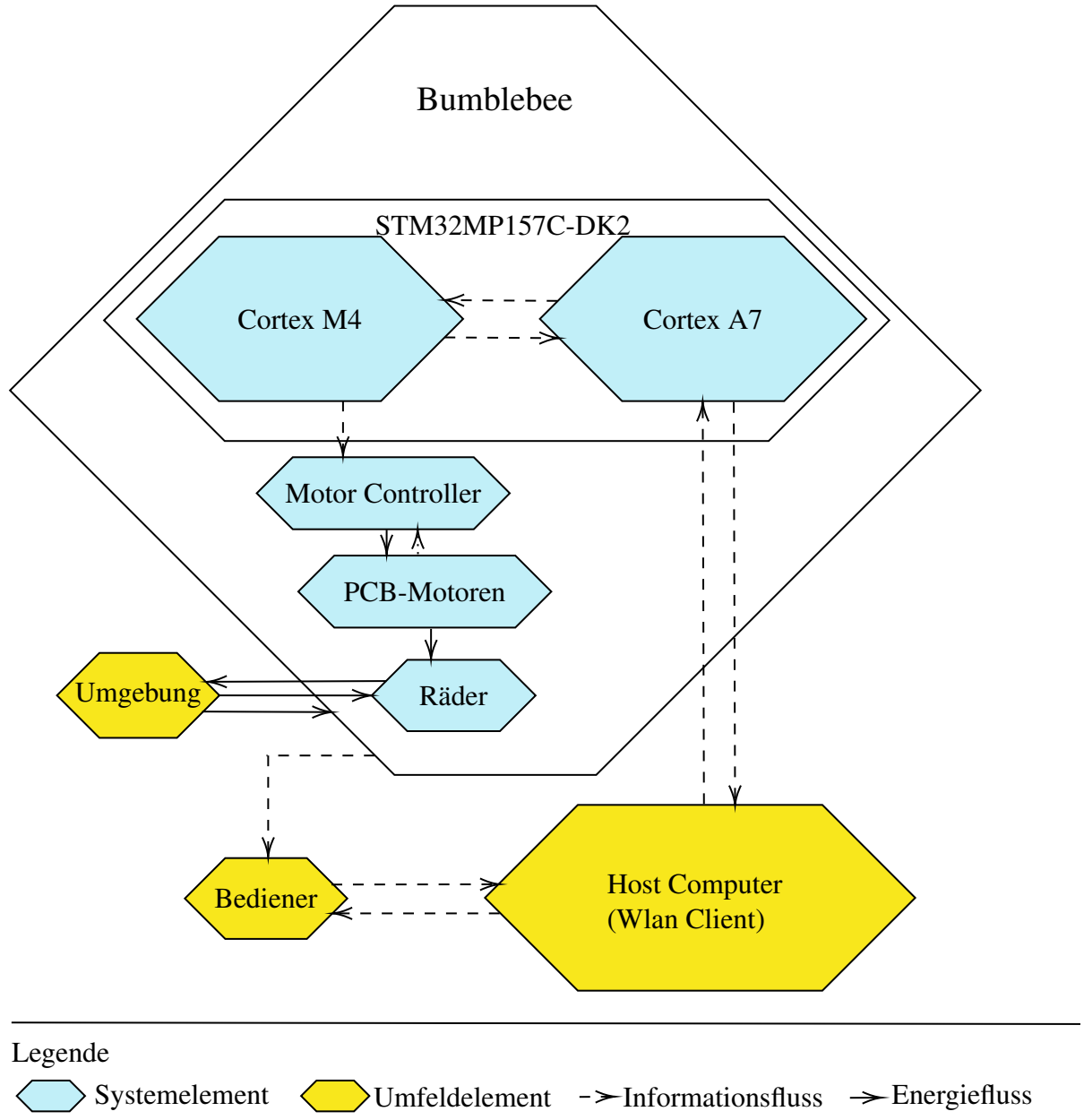

Abbildung 3.2: Umfeldmodell des Bumblebees

3.3 Kommunikationsschnittstellen

Die Kommunikationsschnittstellen können anhand des Umfeldmodelles (Abbildung 3.2) erkannt werden. Die Betrachtung der Schnittstellen geht von dem Umfeldelement Bediener aus.

- Bediener stellt Parameter am Host-Computer ein
- Host-Computer kommuniziert mit der MPU
- Cortex-A7 Prozessor kommuniziert mit dem Cortex-M4 Prozessor
- Cortex-M4 Prozessor stellt physikalische Größen, die die Motor-Controller steuern
- Hall-Sensoren der PCB-Motoren geben Signalwerte an die Motor-Controller (dieser Vorgang wurde in [6] dokumentiert)
- Bediener beobachtet, wie sich der BumbleBee verhält

4 Software Implementierung

Das folgende Kapitel beschreibt die Umsetzung der in Kapitel 3 aufgestellten Anforderungen. Eine Gesamtdokumentation des Codes wurde mit Doxygen [40] erstellt und befindet sich unter Punkt A.1.4.

4.1 Auswahl und Installation der Linux-Distribution

STMicroelectronics bietet verschiedene Möglichkeiten, eine Linux Distribution auf der Mikroprozessor-Plattform STM32MP1 zu installieren. Diese Möglichkeiten gliedern sich in drei verschiedene *Software Packages*, die in dem Dokument „Installation der *Software Packages*“ Punkt A.1.3 genauer beschrieben werden. Es muss zwischen *Starter Package*, *Developer Package* und *Distribution Package* gewählt werden. [41]
Um eine Auswahl des *Software Package* [41] zu treffen, werden Basiskriterien aufgestellt. Das *embedded Software Package* soll:

- Kriterium 1:
  wenn möglich, die Nutzung des *Framwork* OpenEmbedded [42, 43] umgehen.
- Kriterium 2:
  die Nutzung von STM32CubeMX [34] zulassen.
- Kriterium 3:
  eine IDE enthalten.
- Kriterium 4:
  es möglich machen, Veränderungen am DT vorzunehmen.

Erfüllt ein *embedded Software Package* ein Kriterium nicht, scheidet es aus.

Kriterien:	1	2	3	4
Starter Package				
Developer Package				
Distribution Package				

Tabelle 4.1: Kriterien für die Auswahl des *embedded Software Package*

Durch das Aufstellen und das Begutachten der Kriterien wird das *Developer Package* ausgewählt. Die Installation des *Developer Package* ist in dem Dokument „Installation der *Software Packages*“ Punkt A.1.3, beschrieben.

4.2 Konfiguration des Cortex-M4 Prozessors

Für die Cortex-M4-seitige Konfiguration der MPU wird das *Tool* STM32CubeMX verwendet. Das STM32MP157C-DK2 *Board* ist im Vergleich zu den Mikrocontrollern der Serie STM32F4 aufwendiger zu konfigurieren. Das liegt daran, dass die STM32MP157C-DK2 zwei Recheneinheiten besitzt, die sich den Zugriff auf Peripherien teilen. Aus diesem Grund müssen zusätzliche *„System Core“* und *„Middleware“* Einstellungen getroffen werden, die bei der Konfiguration eines STM32 Mikrocontrollers mit einer Recheneinheit nicht gemacht werden müssten. Die *„System Core“* Einstellungen beinhalten die Unterpunkte Double Data Rate (DDR), Direct Memory Access (DMA), Generic Interrupt Controller (GIC), GPIO, Hardware Semaphore (HSEM), Inter-Processor Communication Controller (IPCC), Independent Watchdog (IWDG)1, IWDG2, Memory Access Controller (MDMA), Nested Vectored Interrupt Controller (NVIC), Reset and Clock Controller (RCC), System (SYS), System Window Watchdog (WWDG)1. Die Einstellung *„Middleware“* beinhaltet die Unterpunkte Free Real-Time Operating System (FreeRTOS) und OpenAMP. Um nicht die gesamte Konfiguration der MPU händisch einstellen zu müssen, wird eine von STM32CubeMX mitgelieferte Standard-Konfiguration als Grundlage der Parametrierung verwendet.

Pfad zu der Standard-Konfiguration aus dem Installationsverzeichnis von STM32CubeMX:

```
1 PC $> ~/STM32CubeMX/db/plugins/boardmanager/boards/M10_Discovery_STM
    32MP157C-DK2_STM32MP157CAC_Board_AllConfig.ioc
```

Die Standard-Konfiguration des STM32MP157C-DK2 wird in STM32CubeMX geladen. Die Standard-Konfiguration wird in den *workspace* gespeichert. Aufbauend auf der Standard-Konfiguration werden jetzt die Einstellungen für die BumbleBee Anwendungen gemacht. In der Tabelle 4.2 sind die für die Ansteuerung der Motorcontroller benötigten Pins aufgelistet.

PIN - Name	Eingangs-Spannung		max. Eingangs-Strom	Eingangs-Frequenz
	Low Pegel	High Pegel		
FR - Links	0 V	5 V	10 mA	-
FR - Rechts	0 V	5 V	10 mA	-
PWM - Links	0 V	5 V	20 mA	15 kHz - 100 kHz
PWM - Rechts	0V	5 V	20 mA	15 kHz - 100 kHz

Tabelle 4.2: Eingangscharakteristik der BumbleBee-Platine [6, 44]

Die Ansteuerung des BumbleBees, erfordert wie in Tabelle 4.2 zusehen ist, vier Eingänge. Die Platine des BumbleBees hat zwei Motorcontroller, die jeweils einen Pulsweitenmodulation (PWM)-Eingang und einen Forward/Reverse (FR)-Eingang besitzen. Der Duty Cycle (DC) der PWM gibt die Geschwindigkeit der Motoren vor. Der FR-Eingang bestimmt die Drehrichtung der Motoren [44, 6].

Um die FR-Eingänge anzusteuern, werden zwei GPIO-Pins, wie in Abbildung 4.1 zu sehen ist, konfiguriert. Die Konfiguration kann unter *„System Core“*, unter dem Reiter GPIO durchgeführt werden. Ausgewählt werden die GPIO-Pins PG3 und PH6, da sich diese GPIO-Pins auf der „CN13:Arduino“ (Arduino *Connector*) befinden. Aus dem Datenblatt der Motorcontroller [44] und dem *Reference Manual* der MPU [8] wird entnommen, dass die GPIO-Pins als *„Output Push Pull“* und *„No pull-up and no pull-down“* konfiguriert werden müssen.

Pin Name	Signal on Pin	GPIO mode	GPIO Pull-up/...	Maximum outp...	User Label	Modified
PA6	n/a	Input mode	No pull-up and...	n/a	ETH_MDINT [...	☑
PA10	n/a	Output Push Pull	No pull-up and...	Low	HDMI_NRST [...	☑
PG3	n/a	Output Push Pull	No pull-up and...	High	FR_Links	☑
PG9	n/a	Output Push Pull	No pull-up and...	Low	AUDIO_RST [...	☑
PH4	n/a	Output Push Pull	No pull-up and...	Low	WL_REG_ON...	☑
PH5	n/a	Input mode	No pull-up and...	n/a	BT_HOST_W...	☑
PH6	n/a	Output Push Pull	No pull-up and...	High	FR_Rechts	☑
PH7	n/a	Output Push Pull	No pull-up and...	Low	LED_Y [LD7_...	☑

Abbildung 4.1: STM32CubeMX Konfiguration der GPIO-Pins

Für den geordneten Zugriff des Cortex-M4 auf gemeinsame Ressourcen, wird HSEM aktiviert. Dies wird, wie in Abbildung 4.2 zu sehen, unter dem Unterpunkt HSEM eingestellt.

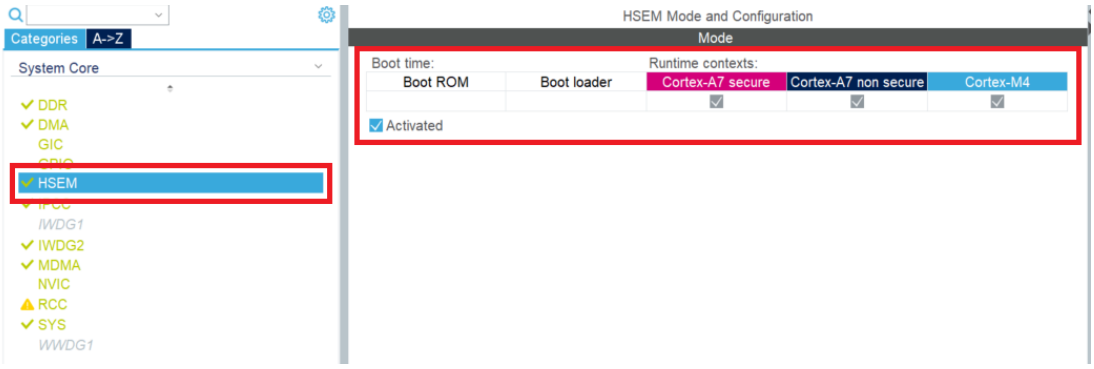

Abbildung 4.2: STM32CubeMX Konfiguration der HSEM

Für die IPC muss, wie in Abbildung 4.3 zu sehen, der Unterpunkt IPCC konfiguriert werden. Die *Interrupts* „IPCC RX1 *occupied interrupt*“ und „IPCC TX1 *free interrupt*“ werden angewählt, um die Nachrichten der IPC über die Interrupt-Routinen senden und empfangen zu können.

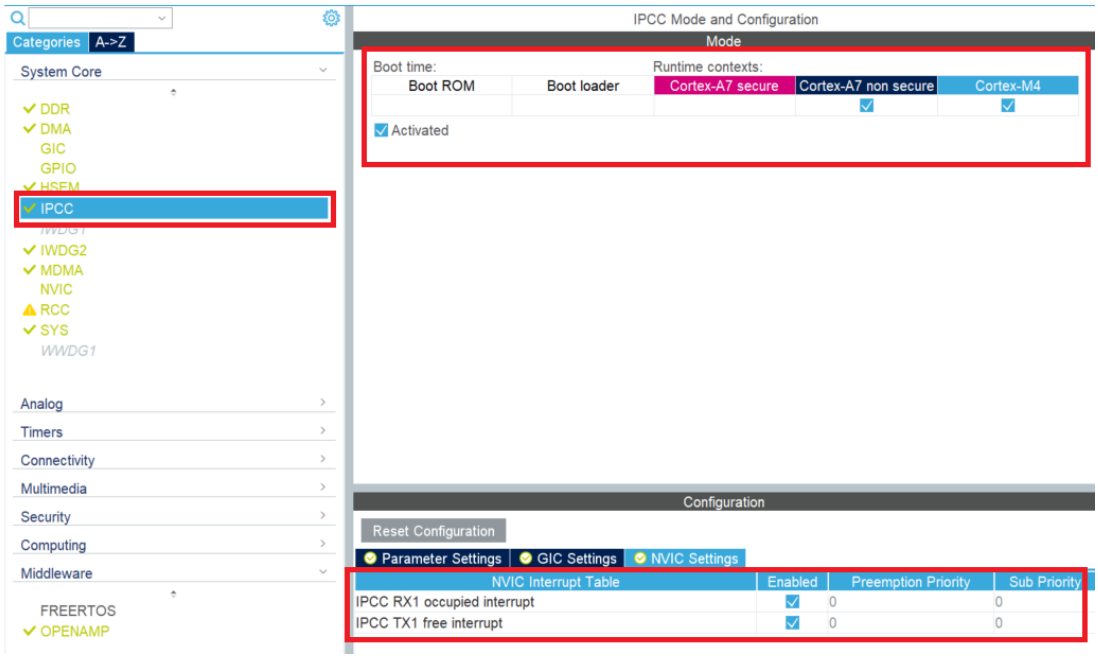

Abbildung 4.3: STM32CubeMX Konfiguration des IPCC

Die Middleware OpenAMP wird unter Middleware im Unterpunkt OpenAMP ausgewählt. In dem unteren linken roten Rahmen der Abbildung 4.4 wird die Startadresse des gemeinsamen Speichers sowie die Größe des gemeinsamen Speichers angezeigt.

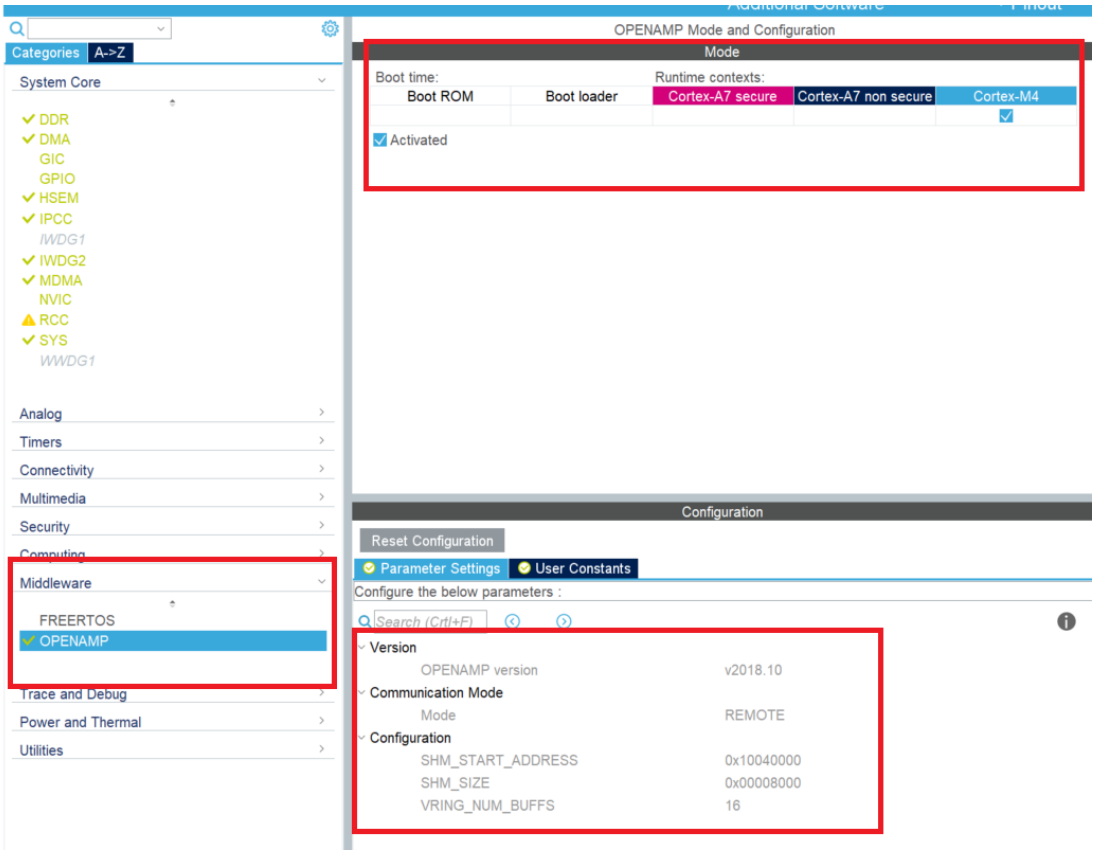

Abbildung 4.4: STM32CubeMX Konfiguration der Middleware OpenAMP

Um die PWM-Eingänge der Motorcontroller anzusteuern, werden zwei PWM-*Output*-Pins konfiguriert. Die Frequenz der PWM muss zwischen 15 kHz und 100 kHz liegen [44]. Um eine PWM-Frequenz zu wählen, wird der Mittelwert aus dem minimalen Wert 15 kHz und dem maximalen Wert 100 kHz gebildet.

$$f_{PWM_{ausgewählt}} = \frac{f_{PWM_{min}} + f_{PWM_{max}}}{2}$$

$$f_{PWM_{ausgewählt}} = \frac{15\,\text{kHz} + 100\,\text{kHz}}{2} = 57{,}5\,\text{kHz}$$

Für die Berechnung der *Timer*-Frequenz ($f_{timetick}$) gilt nach [8] folgende Formel:

$$f_{timetick} = \frac{f_{Prozessortakt}}{Prescaler + 1}$$

Die Frequenz $f_{timetick}$ ist die Frequenz, mit der der *Timer* das Zählregister inkrementiert. Der *Timer* inkrementiert so lange, bis der Wert des Zählregisters den Wert der *Counter Period* überschritten hat. Um $f_{timetick}$ zu errechnen, wird geprüft ob ein *Prescaler* erforderlich ist. Dafür muss bekannt sein, wie groß der maximale Wert der *Counter Period* ist. Als PWM-*Timer* wird der *Timer* 1 ausgewählt, da dieser *Timer* zwei GPIO-Pin-Ausgänge auf dem Anschluss „CN13:Arduino“ (Arduino *Connector*) hat. *Timer* 1 ist ein 16 bit *Timer* [8]. Der maximale Wert der *Counter Period* eines 16 bit *Timer* beträgt 65535.

$$Counter\ Period\ max. = 2^{16} - 1 = 65535$$

Mit dem maximalen *Counter Period* Wert und dem $f_{Prozessortakt}$ wird errechnet, ob ein *Prescaler* erforderlich ist.

$$f_{min.\ (mit\ max.\ Counter\ Period\ Wert)} = \frac{f_{Prozessortakt}}{Counter\ Period_{max.} + 1}$$

Der Prozessortakt $f_{Prozessortakt}$ beträgt 208,877 93 MHz [8].

$$f_{PWM\ min.\ (mit\ max.\ Counter\ Period\ Wert)} = 3{,}187\,224\,274\,\text{kHz}$$

Da die Freuenz $f_{PWM\ min.\ (mit\ max.\ Counter\ Period\ Wert)}$ kleiner ist als die ausgewählte Frequenz $f_{PWM_{ausgewählt}}$, wird als *Prescaler* 0 verwendet.

$$f_{PWM\ min.\ (mit\ max.\ Counter\ Period\ Wert)} < f_{PWM_{ausgewählt}}$$

Um die Auswahl des *Prescaler* deutlich zu machen, wird diese nochmals im Zeitbereich dargestellt. Das bedeutet, dass die ausgewählte Periodendauer $T_{PWM_{ausgewählt}}$ kleiner ist als die mit einem *Prescaler* von 0 maximal mögliche Periodendauer $T_{PWM\ max.\ (mit\ max.\ Counter\ Period\ Wert)}$.

$$T_{PWM\ max.\ (mit\ max.\ Counter\ Period\ Wert)} > T_{PWM_{ausgewählt}}$$

Mit dem ausgewählten *Prescaler* von 0 und der Frequenz des Prozessortaktes $f_{Prozessortakt}$ wird die Frequenz $f_{timetick}$ errechnet.

$$f_{timetick} = \frac{f_{Prozessortakt}}{Prescaler + 1} = f_{Prozessortakt} = 208{,}877\,93\,\text{MHz}$$

Der *Counter Period* Wert wird ermittelt.

$$Counter\ Period = \frac{f_{timetick}}{f_{PWM_{ausgewählt}}} - 1 = 3631{,}659652 \approx 3632$$

Es wird überprüft, welche PWM-Frequenz eingestellt wurde.

$$f_{PWM_{Kontrolle}} = \frac{f_{timetick}}{Counter\ Period + 1} = 57{,}494\,613\,27\,\text{kHz}$$

Der *Timer Channel* 3 hat als GPIO-Ausgang den Pin PE13 und der *Timer Channel* 4 hat als GPIO-Ausgang den Pin PA11. Die Pulsweite wird im Programm des Cortex-M4 vorgegeben. Die Konfigurationsparameter, die in STM32CubeMX gesetzt werden, sind in Abbildung 4.5 zu sehen.

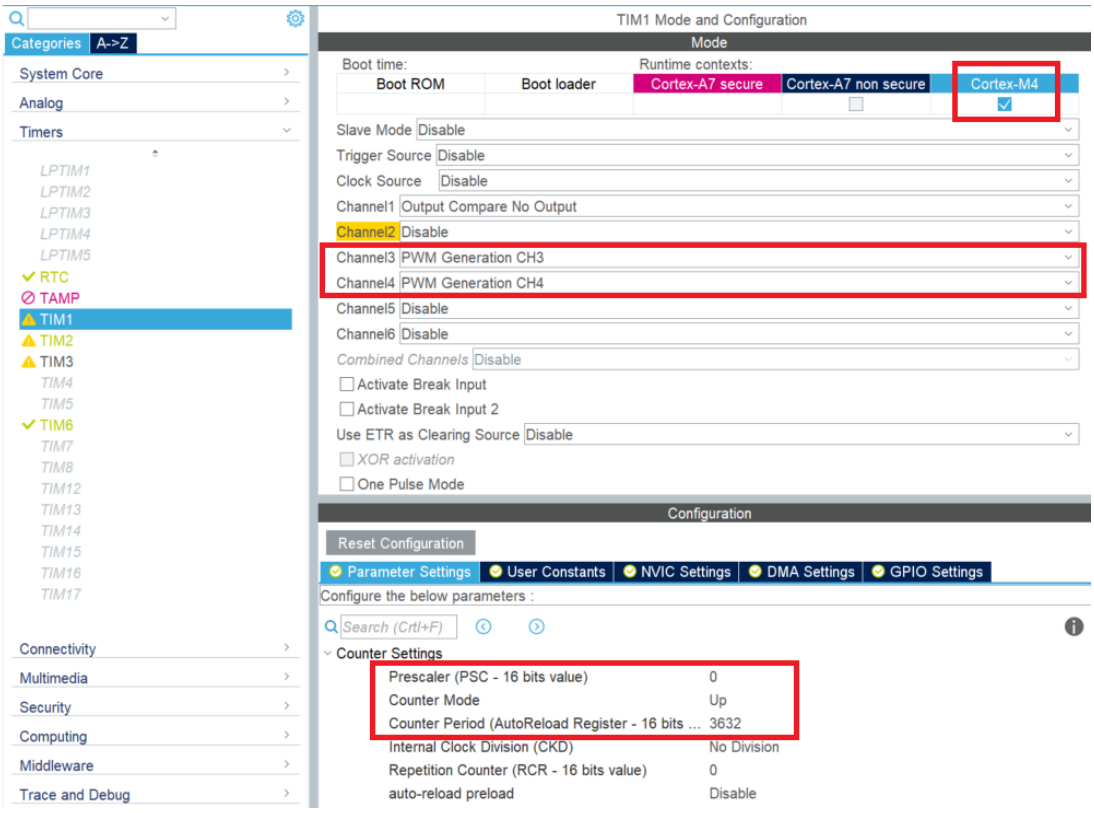

Abbildung 4.5: STM32CubeMX Konfiguration des *Timer* 1

Die Tabelle 4.3 zeigt die Pinbelegung zwischen der MPU und der Platine des BumbleBees.

Pins der MPU	Pins der BumbleBee Platine
PG3	FR-links
PH6	FR-rechts
PA11	PWM-links
PE13	PWM-rechts

Tabelle 4.3: Pinbelegung

In den Projekt Manager Einstellungen wird unter *Project* die „*Toolchain* /IDE“ ausgewählt. Dies ist in Abbildung 4.6 zu erkennen [34].

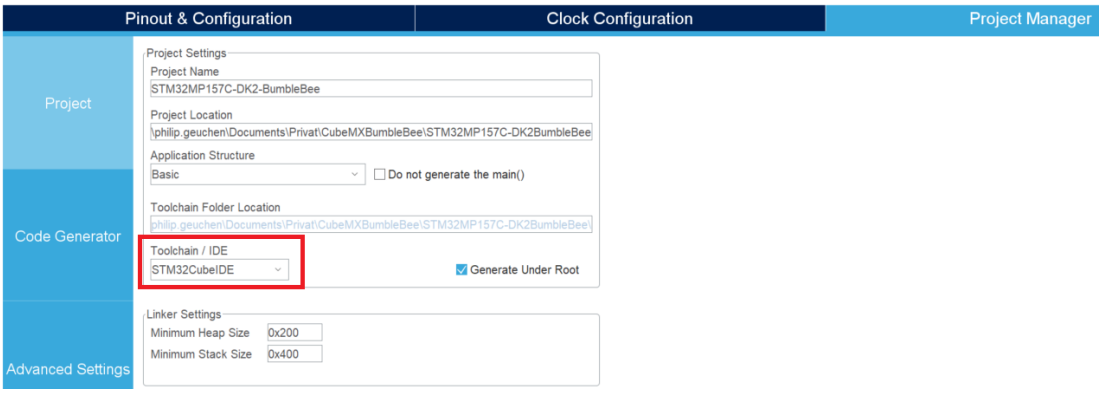

Abbildung 4.6: STM32CubeMX Konfiguration der *Toolchain*

Unter „Code Generator“ wird der Haken „*Keep User Code when re-generating*“ gesetzt [34].

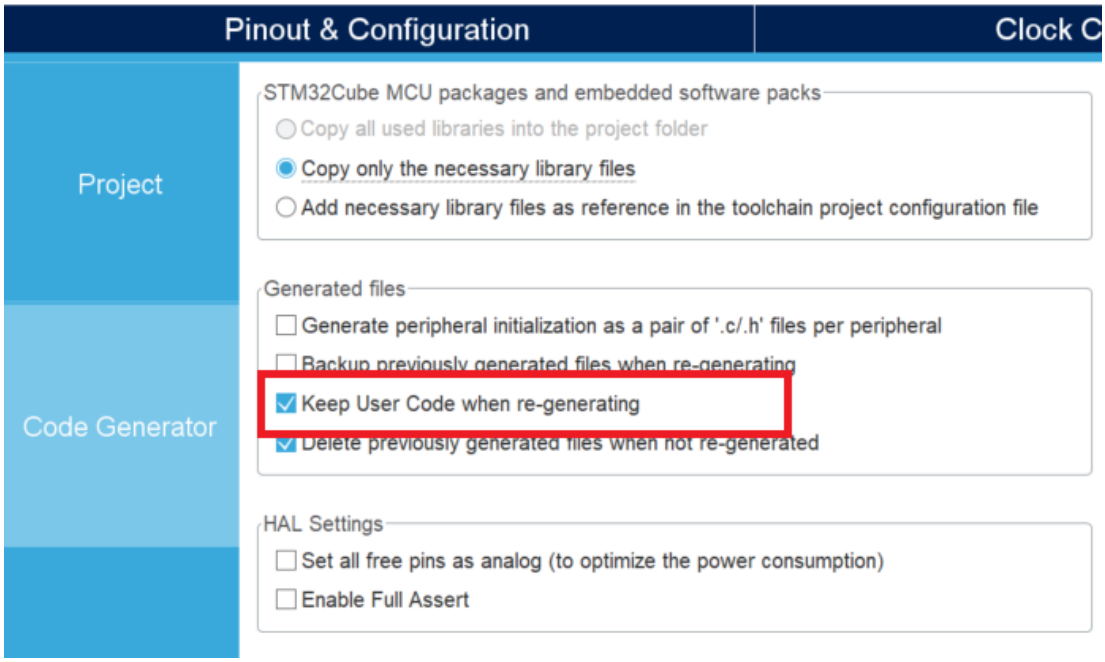

Abbildung 4.7: STM32CubeMX Konfiguration „*Keep User Code when re-generating*“

Der Code-Generierungsprozess wird mit dem Button „*GENERATE CODE*“ (Abbildung 4.8) gestartet [34].

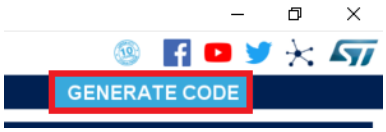

Abbildung 4.8: STM32CubeMX starten des Code-Generierungsprozesses

Das generierte Projekt wird durch das Klicken auf „*Open Project*“ geöffnet (Abbildung 4.9) [34].

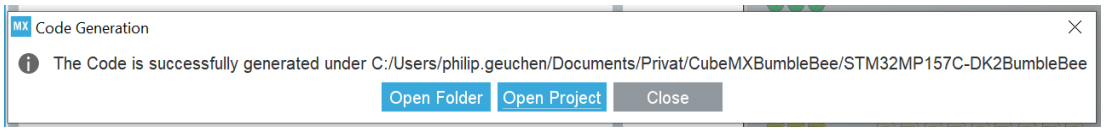

Abbildung 4.9: STM32CubeMX: Öffnen des Projektes in der STM32CubeIDE

4.3 Hardware Semaphore

Bei der MPU sind die Cortex-A7 Prozessoren und der Cortex-M4 Prozessor mit der „MLAHB: ARM 32 bit multi-AHB bus matrix (209 MHz)“ verbunden (Abbildung 4.10). Die Cortex A-7 Prozessoren sind über den „AXIM: ARN 64 bit AXI interconnect 266 MHz“ asynchron mit dem MLAHB verbunden. Der Cortex-M4 Prozessor ist über den D-Bus, den I-Bus und den S-Bus mit dem MLAHB verbunden (Abbildung 4.10). Um vom Cortex-M4 Prozessor während der Laufzeit der Cortex-A7 Prozessoren einen GPIO Pin zu schalten, ist es wichtig, dass beide Prozessoren nicht gleichzeitig auf einen GPIO-Pin zugreifen können. [4]

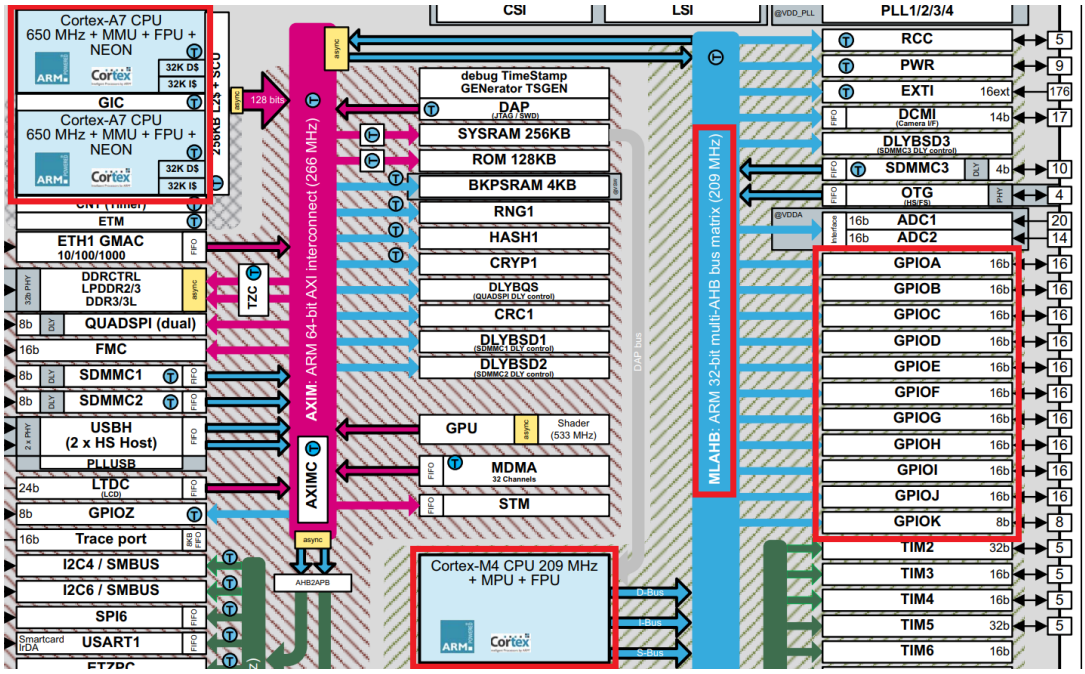

Abbildung 4.10: Ausschnitt des Prozessorbus der MPU [4, S. 19]

Für den Zugriff auf die GPIO-Ports werden aus diesem Grund HSEM genutzt. HSEM können in Computersystemen mit mindestens zwei Prozessoren, die über Busse Zugriff auf gemeinsame Ressourcen haben, eingesetzt werden, um ein gleichzeitiges Zugreifen auf eine Speicherstelle zu verhindern [45]. Die HSEM werden durch die Funktionen Periph_Lock und Periph_Unlock aufgerufen. Durch die Funktion Periph_Lock, der der Pointer auf das Makro des GPIO-Ports und

die *Timeout* Zeit in ms übergeben wird, wird der Zugriff für die Cortex-A7 Prozessoren gesperrt. Ist der GPIO-Port gesperrt, kann der Cortex-M4 Prozessor auf den GPIO-Port zugreifen. Die Funktion Periph_Lock sperrt den GPIO-Port mit der Funktion HAL_HSEM_FastTake. HAL_HSEM_FastTake wird der HSEM Index des GPIO-Ports übergeben. Wenn die Funktion HAL_HSEM_FastTake den GPIO-Port vor dem Ablauf der *Timeout* Zeit sperren kann, gibt die Funktion Periph_Lock das Makro LOCK_RESOURCE_STATUS_OK zurück. Wenn die Funktion HAL_HSEM_FastTake den GPIO-Port nicht vor dem Ablauf der *Timeout* Zeit sperren kann, gibt die Funktion Periph_Lock das Makro LOCK_RESOURCE_STATUS_Timeout zurück. Tritt ein Fehler bei dem Aufruf der Funktion HAL_HSEM_FastTake auf, dann gibt die Funktion Periph_Lock das Makro LOCK_RESOURCE_STATUS_ERROR zurück. Die Funktion Periph_Lock wird durch das Makro PERIPH_LOCK aufgerufen. Dem Makro PERIPH_LOCK wird der GPIO-Port übergeben. Das Makro PERIPH_LOCK übergibt den GPIO-Port und eine konstante *Timeout* Zeit von 100 ms. Die Funktion Periph_Unlock entsperrt den von Periph_Lock gesperrten GPIO-Port. Der Funktion Periph_Unlock wird der Pointer auf das Makro des GPIO-Ports übergeben. In der Funktion Periph_Unlock wird durch den Aufruf der Funktion HAL_HSEM_Release die Sperrung des GPIO-Ports wieder aufgehoben. Die Funktion Periph_Unlock wird durch das Makro PERIPH_UNLOCK aufgerufen. Dem Makro PERIPH_UNLOCK wird der GPIO-Port übergeben. Das Makro PERIPH_UNLOCK übergibt den GPIO-Port an die Funktion Periph_Unlock. Soll an einer beliebigen Stelle im Programm ein GPIO-Pin geschaltet werden, ist dies mit dem zusätzlichen Aufrufen der Makros PERIPH_LOCK und PERIPH_UNLOCK möglich. Bevor der GPIO-Pin geschaltet wird, wird das Makro PERIPH_LOCK aufgerufen. Der GPIO-Port wird für den Zugriff der Cortex-A7 Prozessoren gesperrt. Ist der Zugriff für die Cortex-A7 Prozessoren gesperrt, kann der Cortex-M4 Prozessor die GPIO-Pins des gesperrten GPIO-Ports schalten. Der GPIO-Pin wird durch den Aufruf der Funktion HAL_GPIO_WritePin geschaltet. Der Funktion HAL_GPIO_WritePin werden das Makro des GPIO-Ports, das Makro des GPIO-Pins und der Pin Status übergeben. Anschließend wird das Makro PERIPH_UNLOCK aufgerufen, das den GPIO-Port entsperrt. [46]

4.4 Interprozessor Kommunikation

Um einen Datentransfer zwischen dem Cortex-M4 Prozessor und den Cortex-A7 Prozessoren zu ermöglichen, gibt es einen geteilten Speicher. Abbildung 4.11 zeigt die Anordnung der Prozessoren und der Speicher. Rot eingerahmt sind die Dual Cortex-A7 CPU, der Cortex-M4 Prozessor und die Speicherbereiche SRAM1, SRAM2, SRAM3 und SRAM4. Die Speicherbereiche SRAM1- 4 sind zwei 128 k Byte und zwei 64 k Byte große statische RAMs. SRAM1- 4 wird vom Cortex-M4 für das Ausführen von Code, Datenverwaltung und Datenspeicherung für den Austausch mit den Cortex-A7 Prozessoren genutzt [8].

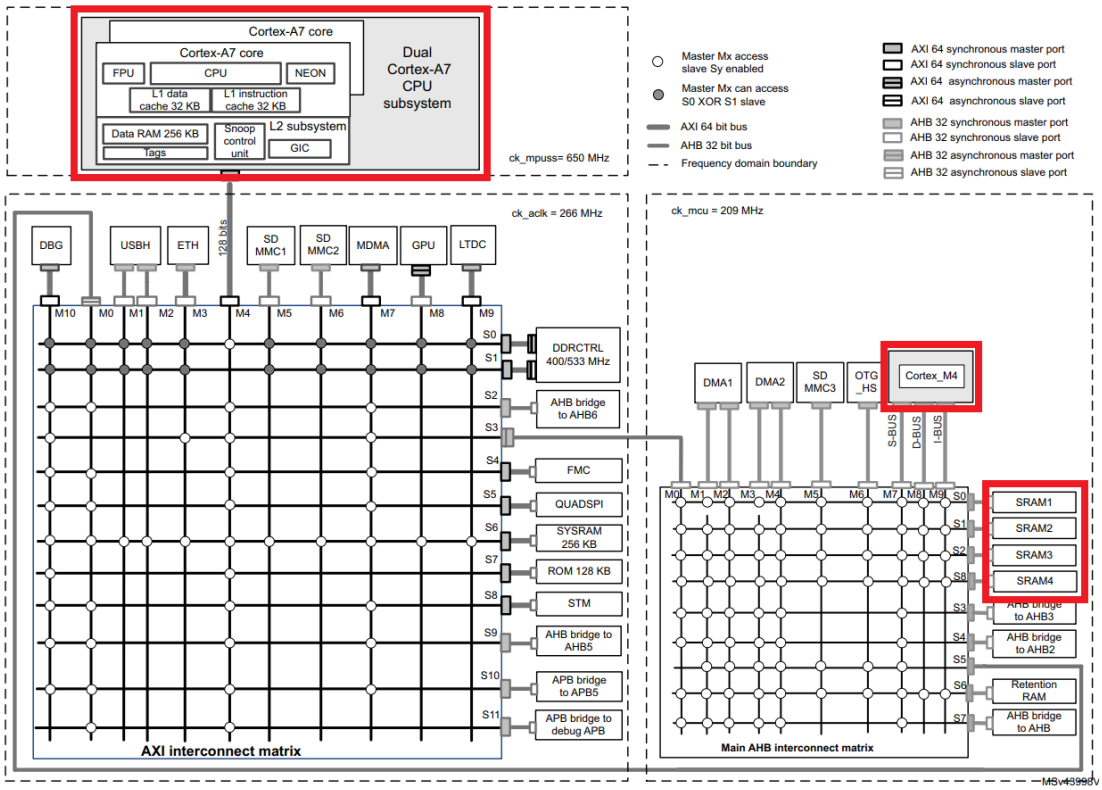

Abbildung 4.11: Bus *Interconnect* der MPU [8, S. 121]

Der gemeinsame Speicher wird durch die Middleware OpenAMP initialisiert. [32] Die Middleware OpenAMP wird durch das Generieren von Quellcode mit STM32CubeMX in den Quellcode für den Cortex-M4 Prozessor eingebunden.

In den Initialisierungsschritten der Main Funktion, ist die Funktion

MX_OPENAMP_Init zu finden. In der Funktion MX_OPENAMP_Init wird die Funktion OPENAMP_shmem_init aufgerufen. Die Funktion OPENAMP_shmem_init initialisiert den gemeinsamen Speicher. OpenAMP baut bei der Initialisierung des gemeinsamen Speichers auf libmetal [47] auf. Die genauere Beziehung von OpenAMP und libmetal kann unter [48] nachgelesen werden. Bei der Initialisierung des gemeinsamen Speichers werden zwei Geräte und zwei Kanäle mit jeweils einem Empfangspuffer angelegt. Dies ist in Abbildung 4.12 dargestellt. Die Cortex-A7 CPU wird dabei als Master CPU definiert. Die Cortex-M4 CPU wird als Slave definiert. Die initialisierten Empfangspuffer sind Ringspeicher mit definierten 512 Byte.

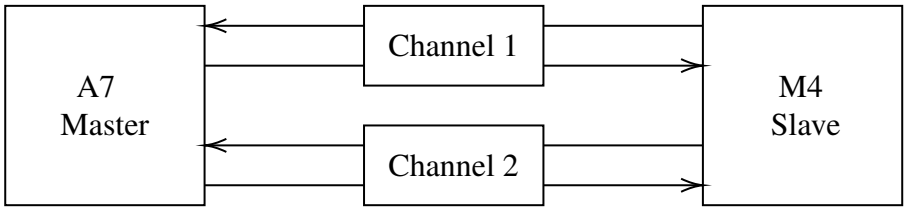

Abbildung 4.12: Virtuelle UART-Schnittstelle zwischen dem Cortex-A7 und dem Cortex-M4

Um eine Nachricht von dem Cortex-M4 Prozessor zu der Cortex-A7 CPU zu senden wird, ein virtuelles UART-Gerät angelegt und durch einen an den Cortex-A7 angeschlossenen RPMsg-Kanal definiert. Dies wird durch die Funktion VIRT_UART_Init, der als Übergabe Parameter der Pointer auf die Struktur des VIRT_UART_HandleTypeDef übergeben wird, getan. Nachdem das virtuelle UART-Gerät angelegt ist, kann mit der Funktion VIRT_UART_Transmit eine Nachricht gesendet werden. Der Funktion VIRT_UART_Transmit werden der Pointer auf die Struktur des VIRT_UART_HandleTypeDef, der Pointer auf das Daten-Array, das übertragen werden soll, und die Anzahl der Array-Elemente übergeben. [49]

Um eine Nachricht von der Cortex-A7 CPU am Cortex-M4 Prozessor zu empfangen, wird die Registrierung des Rückrufs für den Nachrichtenempfang durch den RPMsg-Kanal durchgeführt. Dies wird durch die Funktion VIRT_UART_RegisterCallback getan. Die Nachricht wird in die Funktion Virt_UART0_RXCpltCallback in ein externes Array kopiert und die Größe der Nachricht wird in eine Variable geschrieben. [49]

Zum Testen der M4-Firmware können aus dem Linux Terminal (Cortex-A7) Nachrichten an den Cortex-M4 Prozessor gesendet werden.

```
1 Board $>  stty -onlcr -echo -F /dev/ttyRPMSG0
```

Terminaleinstellungen ändern

```
Board $>  cat /dev/ttyRPMSG0 &
```

Nachrichten des Kanals ttyRPMSG0 anzeigen

```
Board $>  echo "Test Nachricht" >/dev/ttyRPMSG0
```

schreibt eine Nachricht in den ttyRPMSG0 Kanal

Der Aufruf der Funktion VIRT_UART_RegisterCallback wird im Programm durch die Interrupt-Routine IPCC_RX1_IRQHandler ausgelöst. Das Interrupt setzt eine *Flag*, die in der Funktion controllIrFlags dafür sorgt, dass folgender Funktionsablauf (Abbildung 4.13) gestartet wird. Der rote Kasten in der Abbildung 4.13 signalisiert, dass die Funktionen, die sich in diesem Kasten befinden, zum *Framework* OpenAMP gehören.

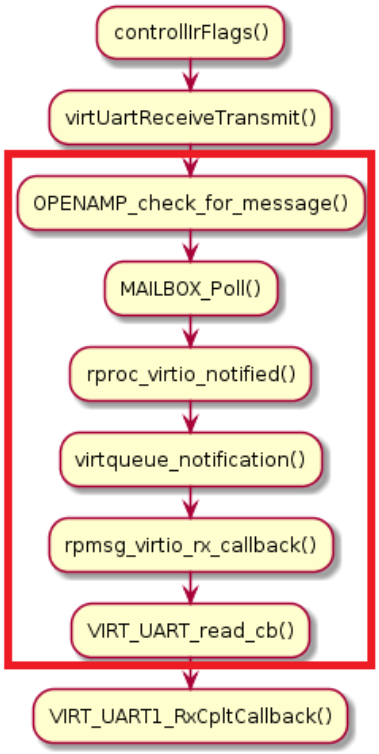

Abbildung 4.13: Funktionsablauf nach dem Abruf des IPC-Interrupt (Cortex-M4)

Implementierung eines Protokolles für die Interprozessor Kommunikation

Um bei der IPC ein definiertes Nachrichtenformat einzurichten, wird ein Protokollformat implementiert. Bei der Auswahl des Nachrichten-Protokolles für die IPC wurden

verschiedene Protokolle verglichen: „MQTT“ [50], Apache Kafka“ [51] und „Protocol Buffers“ [52].
Für den Vergleich der Protokolle wurden zwei Kriterien aufgestellt:

- Kriterium 1: Das Protokoll muss in C implementierbar sein.
- Kriterium 2: Die Nutzung des Protokolles muss benutzerfreundlich sein.

Um zu testen, ob die Nutzung das Protokolles benutzerfreundlich ist, wurde versucht ein einfaches Beispielprotokoll zu finden und dieses innerhalb von 2 h auf dem Host-Computer zu implementieren.

Kriterien:	1	2
MQTT		
Apache Kafka		
Protocol Buffers		

Tabelle 4.4: Kriterien zu der Auswahl des embedded Software Package

Nur ein Beispiel des Protokoll-Format „Protocol Buffers“ konnte im zeitlichen Rahmen von 2 h implementiert werden.

Google Protocol Buffers

Google Protocol Buffers (Protobuf) ist ein Serialisierungsformat, das Nachrichten nach einem Muster serialisiert [53].
Um die Protokoll-Buffer-Kodierung [54] zu verstehen, müssen zunächst Varints verstanden werden. Varints sind Methoden zur Serialisierung von ganzen Zahlen mit einem oder mehreren Byte. Jedes Byte in einem Varint, mit Ausnahme des letzten Bytes, hat das höchstwertige Bit most significant bit (msb) gesetzt. Dies zeigt an, dass noch weitere Bytes kommen werden. Die unteren 7 bits jedes Bytes werden verwendet, um die Zweierkomplement-Darstellung der Zahl in Gruppen von 7 bits zu speichern, wobei die niederwertigste Gruppe zuerst gesetzt wird [55]. Hier ist zum Beispiel die Zahl 1. Es ist ein einzelnes Byte, also ist das msb nicht gesetzt:

```
1 0000 0001
```

Hier ist zum Beispiel die Zahl 300:

```
1  1010 1100 0000 0010
2 > 010 1100  000 0010
```

Zuerst entfallen alle msbs, da diese nur anzeigen, ob das Ende der Zahl erreicht ist.

Dann werden die Gruppen von 7 Bits getauscht, da die Variants Zahlen mit der niedrigsten Gruppe zuerst gespeichert wurde.

```
1 000 0010  010 1100
2 >  000 0010 ++ 010 1100
3 >  100101100
4 >  256 + 32 + 8 + 4 = 300
```

Eine Protokoll-Buffer-Nachricht besteht aus einer Reihe von Schlüssel-Werte-Paaren. Die binäre Version einer Nachricht verwendet nur die Nummer des Feldes als Schlüssel. Der Name und der deklarierte Typ für jedes Feld kann nur auf der Dekodierungsseite durch Bezugnahme auf die Definition des Nachrichtentyps aus der .proto-Datei, bestimmt werden. [55]

Wenn eine Nachricht kodiert wird, werden die Schlüssel und Werte zu einem Byte-*Stream* verkettet. Wenn die Nachricht dekodiert wird, muss der Parser in der Lage sein, Felder zu überspringen, die er nicht erkennt [55]. Auf diese Weise können neue Felder zu einer Nachricht hinzugefügt werden, ohne dass alte Programme, die diese nicht kennen, unterbrochen werden. Zu diesem Zweck besteht der Schlüssel für jedes Paar in einer Nachricht im *Wire Type* eigentlich aus zwei Werten: die Feldnummer aus Ihrer .proto-Datei plus einem *Wire Type*, der gerade genug Informationen liefert, um die Länge des folgenden Wertes zu finden. In den meisten Sprachimplementierungen wird dieser Schlüssel als *Tag* bezeichnet. [55]

Die verfügbaren *Wire Types* sind wie folgt definiert:

Nanopb

Nanopb [55] bezeichnet die C Implementierung für einen Encoder und einen Decoder des Protokollformates Protobuf.

Das Protokoll, das erzeugt werden soll, wird zu Anfang in einer *.proto Datei definiert. Aus dieser *.proto Datei werden mit dem Protobuf-Compiler („protoc -o“) *.pb Dateien. Die *.pb Dateien sind *„File Descriptor Sets“*. Das Python-Script „na-

Typ	Bedeutung	verwendet für
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64 bit	fixed64, sfixed64, double
2	längenbegrenzt	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32 bit	fixed32, sfixed32, float

Tabelle 4.5: *Wire Types* des Protobuf Formats [55]

nopb_generator.py“ erzeugt auf der Basis der *„File Descriptor Sets“* C-Code. Aus jedem *„File Descriptor Sets“* wird eine *Header*- und eine Quellcodedatei erzeugt [53]. In Abbildung 4.14 ist zu erkennen, dass sechs Nachrichten definiert wurden. Die Nachrichten werden jeweils in Structs übersetzt. Die ersten fünf Nachrichten sind Steuerbefehle. Die sechste Nachricht enthält die ersten fünf Nachrichten. Das aus der *.proto Datei generierte Protokoll hat die in Abbildung 4.14 abgebildete Struktur.

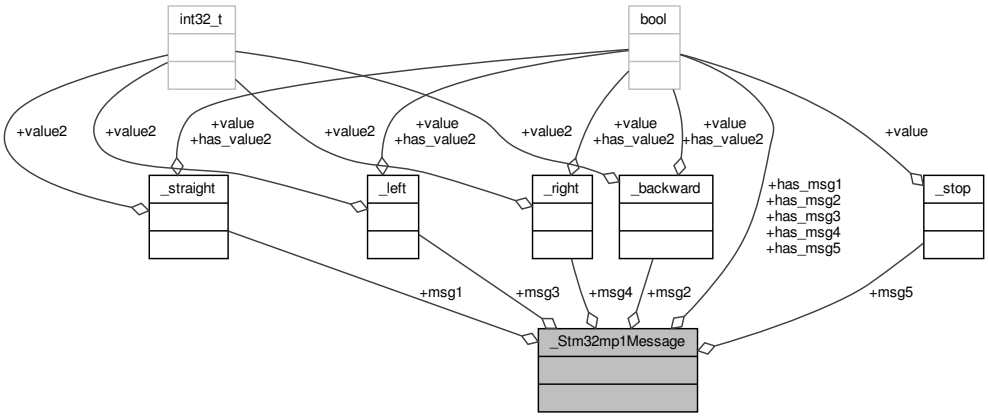

Abbildung 4.14: Struktur der Protobuf Nachricht

Cortex-A7 seitig wird die Funktion encode aus der Funktion convert_ros_to_protobuf aufgerufen (Abbildung 4.15).

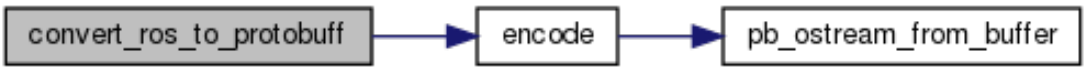

Abbildung 4.15: Verschlüsseln der Protobuf Nachricht auf der Seite des Cortex-A7

Cortex-M4 seitig wird die Funktion encode nach dem Interrupt IPCC_RX1 aus der Funktion virtUartReceiveTransmit aufgerufen.

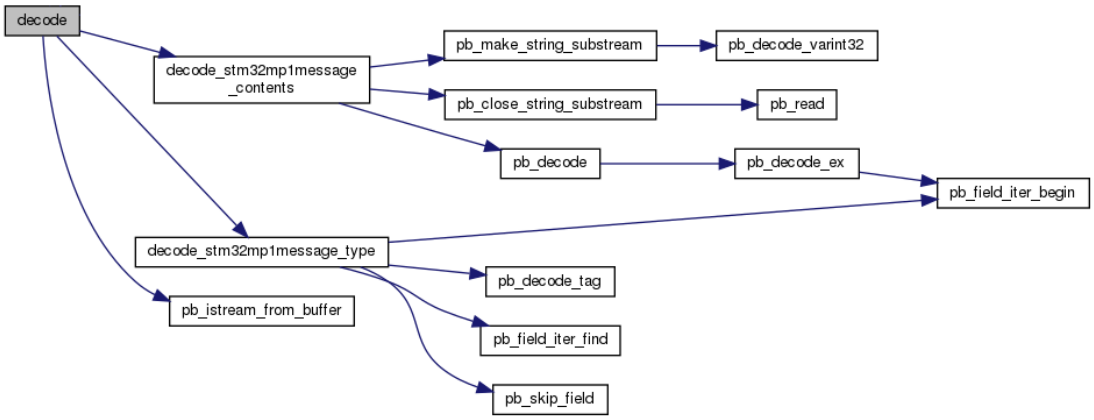

Abbildung 4.16: Entschlüsseln der Protobuf Nachricht auf der Seite des Cortex-M4

4.5 Cortex-A7 *Executable*

Die Hauptaufgabe des Cortex-A7 *Executable* ist es, die ROS-Bus-Nachrichten an den Coprozessor zu senden.
Das *Executable* des Cortex-A7 startet den Coprozessor (Cortex-M4) und initialisiert zwei *Threads* Abbildung 4.17. Das Starten des Coprozessors ist unter Abschnitt 4.7 beschrieben. Der *Thread* thread_copro dient dazu, IPCC-Nachrichten zu empfangen. Der *Thread* thread_ROS dient dazu, ROS-Bus-Nachrichten zu empfangen. Wird von

dem ROS *Thread* eine Nachricht empfangen, wird eine *Flag* gesetzt, die in der Funktion ros_Handler bearbeitet wird (Abbildung 4.18).

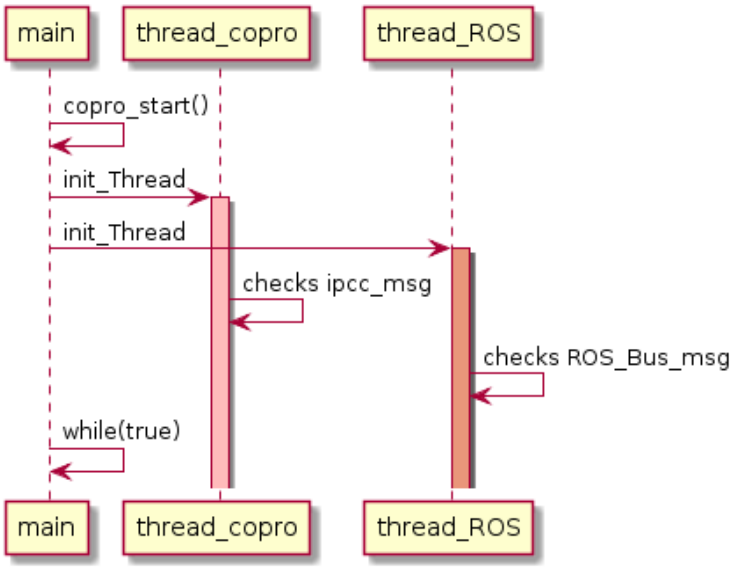

Abbildung 4.17: *Thread*-Initialisierung Cortex-A7 *Executable*

In Abbildung 4.18 ist die *Main while*-Schleife des Cortex-A7 *Executable* abgebildet. Zusehen ist, dass, wenn die rosFlag gesetzt wurde, die Funktionen convert_ros_to_protobuff(), encode(), copro_send_nanopb(), copro_writeTtyRpmsg(), und write() ausgeführt werden. Anschließend wird das rosFlag wieder auf 0 gesetzt. Die Funktion write() schreibt den vorher kodierten Byte-*Stream* in den Teletype (TTY) Kanal ttyRPMSG0 (vgl. Abschnitt 4.4).

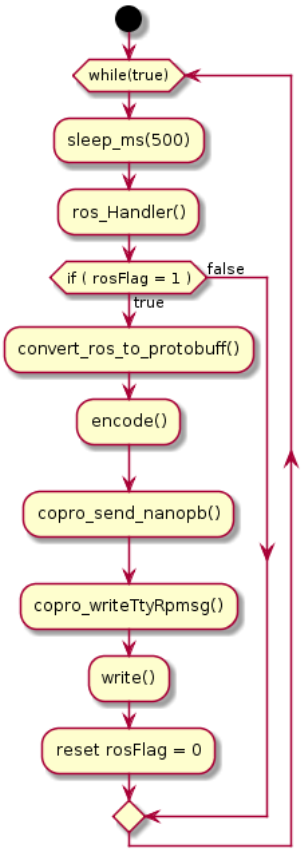

Abbildung 4.18: *Main while* des Cortex-A7 Programm

4.6 Cortex-M4 *Firmware*

Die *Firmware* auf dem Cortex-M4 führt eine *while*-Schleife aus (Abbildung 4.19). In der *while*-Schleife wird in der Funktion controllIrFlag die *Interrupt-Flag* des IPCC-*Interrupt* IPCC_RX1_IRQHandler abgefragt. Wird die *Flag* von dem IPCC-*Interrupt* gesetzt, wird die Funktion virtUartReceiveTransmit aufgerufen (Abbildung 4.20).

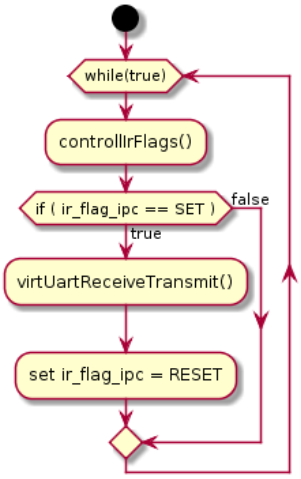

Abbildung 4.19: *Main while* der Cortex-M4 *Firmware*

Abbildung 4.20: Aufruf der Funktion virtUartReceiveTransmit

Die Funktion virtUartReceiveTransmit führt folgende, in Abbildung 4.21 zu erkennende Schritte durch.

1. Liest die IPCC-Nachricht (OpenAMP-*Framework*).
2. Dekodiert die IPCC-Nachricht (Protobuf).
3. Führt den Befehl der IPCC-Nachricht aus (setzt PWM-Frequenz und GPIO-*Pins*).
4. Sendet eine Fehler-Nachricht an den Cortex-A7 Prozessor, wenn die IPCC-Nachricht nicht dekodiert werden konnte.

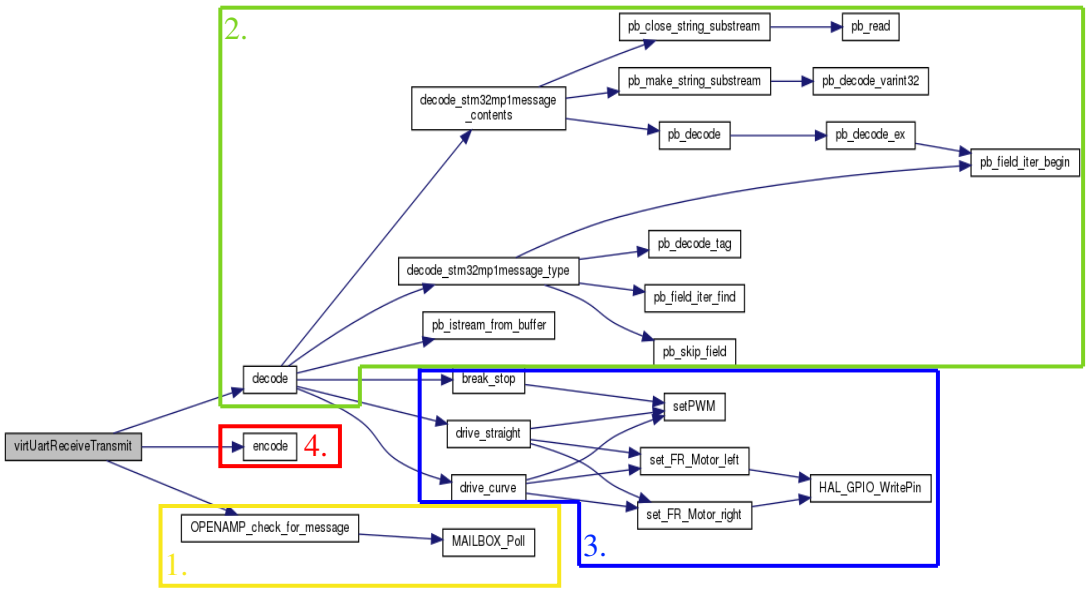

Abbildung 4.21: *Firmware* des Cortex-M4

4.7 *Autoload* und Autostart der *Firmware* auf dem Cortex-M4 Prozessor

Das Remote Processor (RPROC)-Framework (Abbildung 4.22) ermöglicht es, verschiedene Remote-Prozessoren zu steuern (Einschalten, Laden der *Firmware* und Ausschalten), zu debuggen und zu überwachen. [56]

remote proc:
Lädt die *Firmware* in den Speicher des *Remote*-Prozessors, parst den *Firmware Ressource table*, steuert die Ausführung des *Remote*-Prozessors (Start, Stop) und stellt einen Dienst bereit, der die *Remote-Firmware* überwacht und/oder debugget. [56]

stm32_rproc:
Dies ist der Plattformtreiber des *Remote*-Prozessors. Der Treiber registriert herstellerspezifische Funktionen (*Callback*) für das RPROC-*Framework*, handelt die Pattform-Ressourcen wie Register, *Watchdogs*, *Reset*, Uhr und Speicher und leitet die Benachrichtigungen über das *Mailbox-Framework* weiter. [56]

Die noch nicht definierten Begriffe IWDG und SYSCFG sind die Namen von internen Peripherien. [8]

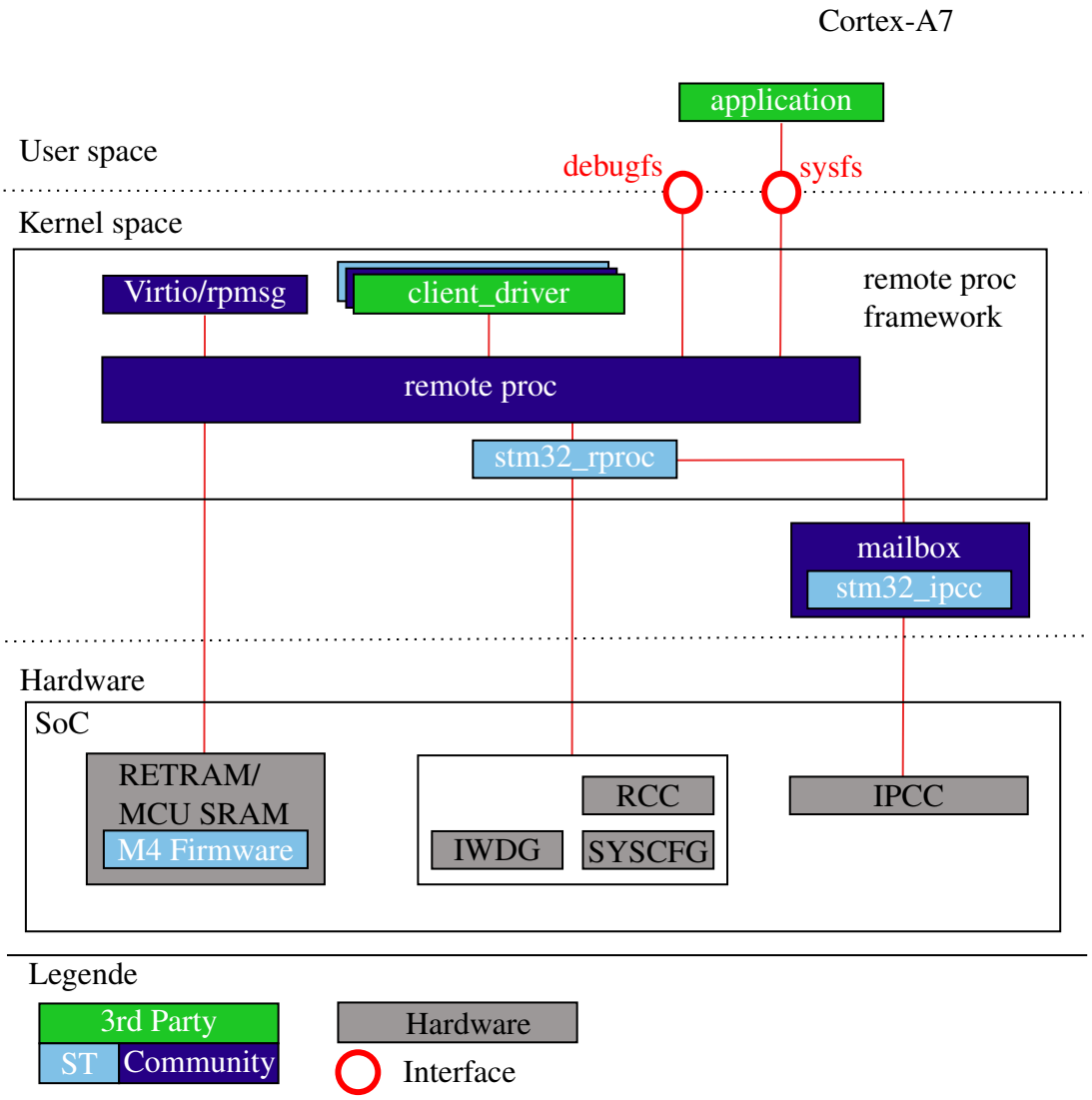

Abbildung 4.22: Das Remote-Processor-Framework [56]

Um die Firmware vom Cortex-A7 in den Cortex-M4 zu laden, wird die *Firmware* in das Verzeichnis /lib/firmware kopiert und mit dem RPROC-Framework in den Speicher des Cortex-M4 Prozessor geladen. [56]

```
1 Board $> cp <Firmware-Name> /lib/firmware
2 Board $> echo -n <Firmware-Name> > /sys/class/remoteproc/remoteproc
    0/firmware
```

Gestartet wird die geladene *Firmware* mit dem folgenden Befehl [56]:

```
Board $> echo -n start > /sys/class/remoteproc/remoteproc0/state
```

Gestoppt wird die gestartete *Firmware* mit dem folgenden Befehl [56]:

```
Board $> echo -n stop > /sys/class/remoteproc/remoteproc0/state
```

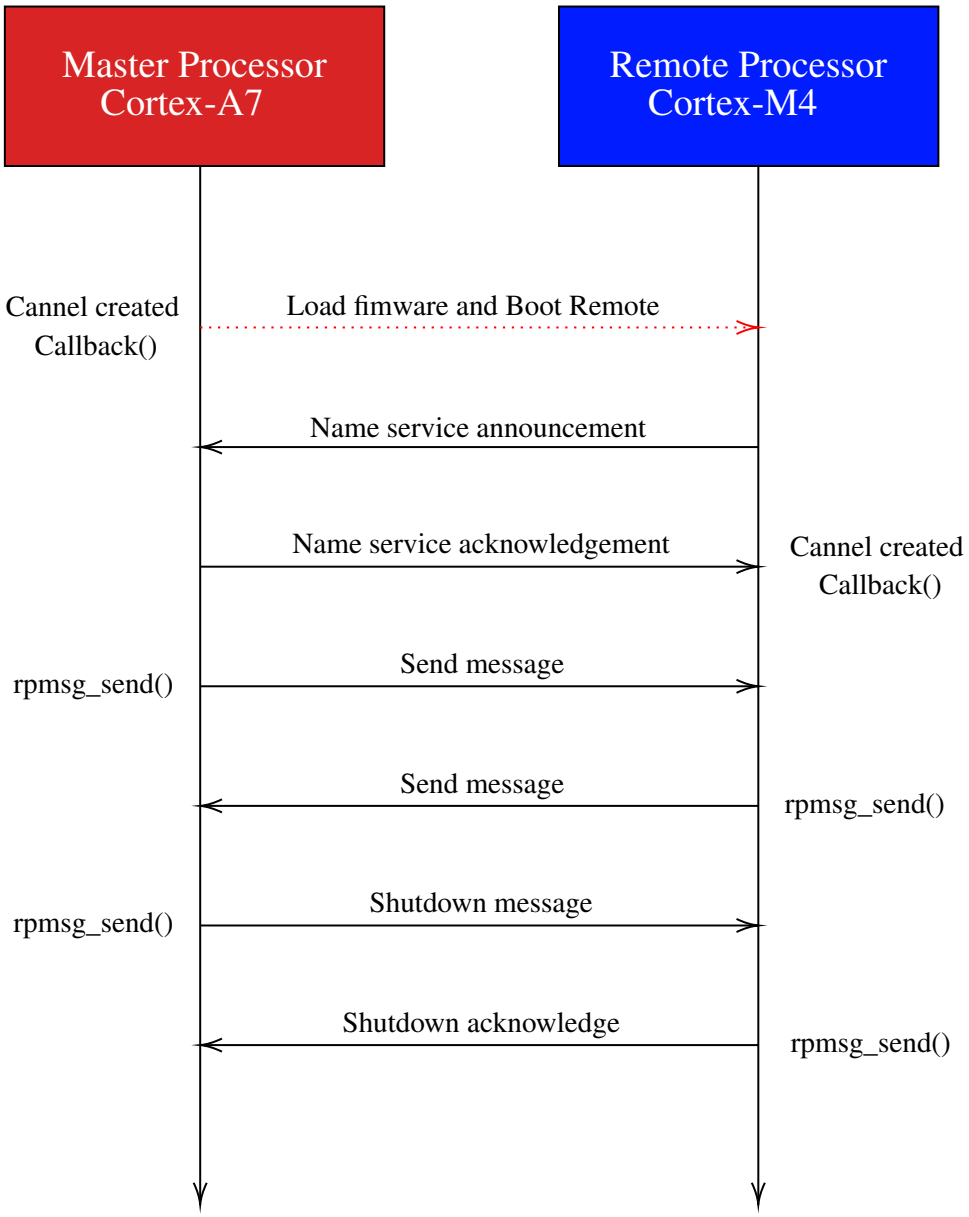

Abbildung 4.23: Laden, Starten und Stoppen der *Firmware* des *Remote Processor* [48, S. 24]

Um die *Firmware* des Cortex-M4 Prozessors nach dem Boot-Vorgang zu starten, wird ein Shellskript genutzt. Das Shellskript wird auf dem Cortex-A7 Prozessor ausgeführt. In dem Shellskript wird die BumbleBee-Anwendung gestartet, die im Initialisierungsschritt die *Firmware* des Cortex-M4 Prozessors lädt. Das Shellskript, das nach dem Boot-Vorgang gestartet wird, ist dabei vom *Developer Package* vorgegeben. Der Ablauf des Ladens der Cortex-M4 *Firmware* läuft, wie in Abbildung 4.24 zu erkennen ist ab.

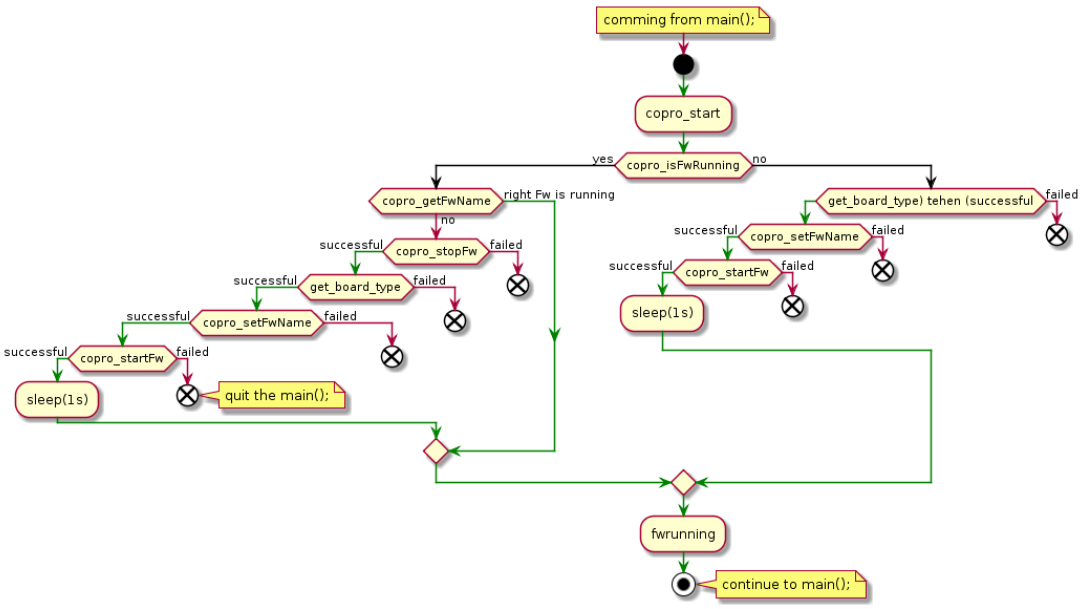

Abbildung 4.24: Starten des Cortex-M4 Prozessors aus dem Programm des Cortex-A7 Prozessors

4.8 Flash-Vorgang per WLAN

Um auf der MPU einen WLAN-*Hotspot* zu erzeugen, wird ein Shellskript genutzt, das als Beispiel im *Starter* und im *Developer Package* enthalten ist. Nach der Ausführung des Shellskripts wird der *Hotspot* von der MPU gestartet. Der Host-Computer wird mit dem Hotspot der MPU verbunden. *Firmware*-Dateien können mit dem Befehl scp vom Host-Computer an die MPU gesendet werden.

```
1 PC $> scp <Firmware Name> root@<WLAN IP-Adresse der MPU>:/usr/local/
    projects/<Pojekt Name>/lib/firmware/
```

Die *Firmware* kann wie in Abschnitt 4.7 geladen und gestartet werden.

4.9 Robot Operating System

Als Kommunikations-Schnittstelle zwischen dem Host-Computer und dem BumbleBee wird das ROS gewählt. ROS eignet sich für die Implementierung auf Roboterystemen mit mehreren Verarbeitungseinheiten [57]. Die Entscheidung, ROS zu verwenden, wird durch den erfolgreichen Einsatz von ROS in der vorherigen Masterarbeit „Entwicklung und Verifikation eines autonomen Logistik Fahrzeuges“ [58] und der Bachelorarbeit „Implementierung einer Schlupfregelung per Model-Based Design sowie einer SLAM-Kartografierung für ein autonomes Logistik-Fahrzeug“ [59] unterstützt.
Das ROS basiert, wie schon in Abschnitt 2.15 Abbildung 2.11 dargestellt, auf einem Master Slave System. Auf dem Master wird ein Parameterserver ausgeführt, der den Knoten, den Austausch von Daten, insbesondere von *Topics*, ermöglicht. Dieser Datenaustausch wird Standardgemäß über Transmission Control Protocol (TCP) und User Datagram Protocol (UDP) Protokolle durchgeführt. [58]
Ein weiterer Vorteil von ROS ist die in Matlab Simulink verfügbare „*ROS Toolchain*“. Diese ermöglicht das Verbinden von Simulink-Blöcken zu ROS-Netzwerken, den live Zugriff auf ROS-Nachrichten und das Generieren von C++-Code mit dem Simulink Coder. [60, 61]
Durch die Entscheidung, ROS auf der MPU zu installieren, ist es nicht mehr möglich, die Nutzung des *Framework OpenEmbedded* auszuschließen. Das bedeutet, dass die in Abschnitt 4.1 getroffene Entscheidung, das *Developer Package* zu nutzen, zurück gezogen wird und ab diesem Punkt mit dem *Distributon Package* weitergearbeitet wird. In Abschnitt 4.1 wurde das *Distribution Package* ausgeschlossen, weil es die Nutzung des *Buildframework OpenEmbedded* enthält. Bei der Installation von Paketen auf der MPU ist es nicht möglich, *OpenEmbedded* zu umgehen. Da die Entscheidung getroffen wird, das ROS auf der MPU zu installieren, ist es nötig, das *Buildframework OpenEmbedded* zu nutzen.

OpenEmbedded

OpenEmbedded ist ein *Buildframework* zum Bauen von kompletten Linux-Distributionen für eingebettete Linux Systeme. [42]
Um die Nutzung von *OpenEmbedded* beschreiben zu können, werden die drei Schlüsselwörter *„BitBake“*, *„recipes“* und *„layers“* beschrieben.

- ***BitBake***
  BitBake ist ein *Build-Tool*, dass den Fokus auf Distributionen und Pakete für das *cross compiling* von eingebetteten Linux Systeme gerichtet hat. *BitBake* ist von dem Paketmanagementsystem der Gentoo Linux-Distribution abgeleitet. *BitBake* war eine Zeit lang ein fester Bestandteil des *OpenEmbedded*-Projektes, bis es als eigenständiges distributionsunabhängiges Werkzeug ausgegliedert wurde. Die Wartung von *BitBake* wird vom Yocto- und vom *OpenEmbedded*-Projekt übernommen. [42, 62]
- ***recipes***
  BitBake-recipes geben vor, wie ein Paket gebaut wird. In den *recipes* sind alle Paketabhängigkeiten, Quellcodeorte, Konfigurations-, Kompilierungs-, Bau-, und Installationsanweisungen. Zusätzlich werden die Metadaten für das jeweilige Paket in Standardvariablen gespeichert. Während des *Build*-Prozesses werden die *recipes* dazu verwendet, Abhängigkeiten zu verfolgen, Pakete *cross* zu kompilieren und diese so zu paketieren, dass diese für die Installation auf dem Zielgerät geeignet sind. [42, 62]
- ***layers***
  Eine *layer* ist eine Sammlung von *recipes* und/oder Konfigurationen, die auf dem *OpenEmbedded Core* verwendet werden können. Typischerweise sind die *layer* nach Themen organisiert. [42, 62]

OpenEmbedded Layer for ROS Applications

Die *OpenEmbedded Layer for ROS Applications*[1] ist online verfügbar. In der *layer* ist eine ROS-Indigo Version für eingebettete Geräte enthalten. [63] Diese ROS-Version wird als *Open-Embedded-Layer* zu der ST-Standard Distribution hinzugefügt. In dem Dokument „Installation der *Software Packages*“ Punkt A.1.3, das sich im Anhang befindet, ist beschrieben, wie das *Distribution Package* installiert wird und wie die ST-Standard-Distribution *„st-image-weston“* gebaut wird. Aufbauend auf dieser Distribu-

[1] `https://github.com/ros/meta-ros`

tion wird jetzt ein *workspace* erstellt, in dem ROS-Indigo durch *recipes* heruntergeladen, für die MPU *cross*-kompiliert und in die ST-Standard-Distribution integriert wird. Für diesen Vorgang werden die *Build-Tools „BitBake“* und *„devtool“* verwendet.

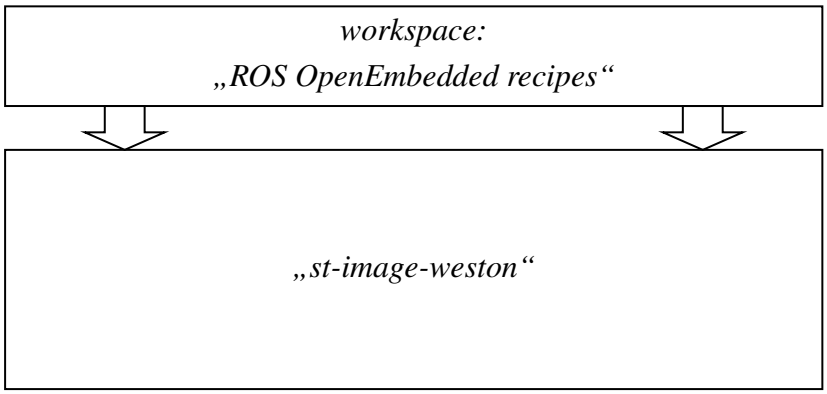

Abbildung 4.25: Hinzufügen der ROS *OpenEmbedded recipes* zu der ST-Standart-Distribution *„st-image-weston“*

Um ROS in die ST-Standard-Distribution zu integrieren, wird das „meta-ros“ *Repository* auf den Host-Computer in das Verzeichnis der *layer* geklont.

```
1 PC $> cd ~/STM32MPU_workspace/STM32MP15-Ecosystem-v1.1.0/
    Distribution-Package/openstlinux-4.19-thud-mp1-19-10-09/layers
2 PC $> git clone https://github.com/ros/meta-ros.git
```

Die *OpenEmbedded-Build*-Umgebung wird durch das Ausführen des *Setup-Skript* für die *OpenEmbedded*-Umgebung initialisiert.

```
1 PC $> cd  ~/STM32MPU_workspace/STM32MP15-Ecosystem-v1.1.0/
    Distribution-Package/openstlinux-4.19-thud-mp1-19-10-09/build-
    openstlinuxweston-stm32mp1
2 PC $> DISTRO=openstlinux-weston MACHINE=stm32mp1 source layers/meta-
    st/scripts/envsetup.sh
```

Mit dem Befehl

```
1 PC $> bitbake-layers show-layers
```

können alle *layer* angezeigt werden.

Die „meta-ros“ *layer* werden mit dem Befehl

```
1 PC $> bitbake-layers add-layer ../layers/meta-ros/
```

zu der *layer*-Liste hinzugefügt. Nach dem Hinzufügen der „meta-ros“ *layer* können die in ihr enthaltenen *recipes* mit dem Befehl

```
PC $> devtool find-recipe core-image-ros-roscore
```

durchsucht und mit dem Befehl

```
PC $> devtool add <source path of the recipe>
```

zum *workspace* hinzugefügt werden.

Für die BumbleBee Anwendung wurden die *recipes* „core-image-ros-roscore.bb“, „geometry-msgs.bb“ und „ros-sdk-test.bb“ mit dem *Build-Tool* „devtool add“ zum *workspace* hinzugefügt.

Nach dem Hinzufügen der ROS *recipes* in den *workspace* wird die ST-Standard-Distribution gebaut. In dieser Distribution sind nun auch die *cross*-kompilierten ROS-Inhalte enthalten. Nach dem Bau der Distribution wird auf der Basis der veränderten *„st-image-weston“* Distribution ein Software Development Kit (SDK) erzeugt. Die Beschreibung, wie das SDK zu erstellen ist, ist unter Punkt A.1.3 beschrieben. Die gebaute Distribution wird wie in Punkt A.1.3 beschrieben auf der MPU installiert. Nachdem die Distribution auf der MPU installiert wurde und die MPU gebootet hat, werden die ROS Umgebungsvariablen hinzugefügt.

```
Board $> export ROS_ROOT=/opt/ros/indigo
Board $> export PATH=$PATH:/opt/ros/indigo/bin
Board $> export LD_LIBRARY_PATH=/opt/ros/indigo/lib
Board $> export PYTHONPATH=/opt/ros/indigo/lib/python2.7/site-
    packages
Board $> export ROS_MASTER_URI=http://localhost:11311
Board $> export CMAKE_PREFIX_PATH=/opt/ros/indigo
Board $> touch /opt/ros/indigo/.catkin
```

Nachdem die Umgebungsvariablen hinzugefügt wurden, können ROS-Befehle auf der MPU ausgeführt werden. Dies ermöglicht das Ausführen eines ROS-Masters.

```
Board $> roscore
```

Das ROS-Netzwerk in Matlab Simulink

In Matlab Simulink wird das ROS-Netzwerk (Abbildung 4.26 und Abbildung 4.27) aufgebaut und getestet. Abbildung 4.26 zeigt das Simulink-Modell, das auf dem Host-Computer ausgeführt wird. Es beinhaltet drei *Slider*, die die Sollwertvorgaben für den linken Antrieb, den rechten Antrieb und für die Zeit, die die Bewegung ausgeführt werden soll, einstellen. Die Werte werden über den Nachrichten-Typ *geometry_msg/Point* unter dem *Topic „my_topic“* veröffentlicht. Durch das Betätigen des Schalters werden alle Werte auf null gesetzt. Das ermöglicht es, den BumbleBee zu stoppen.

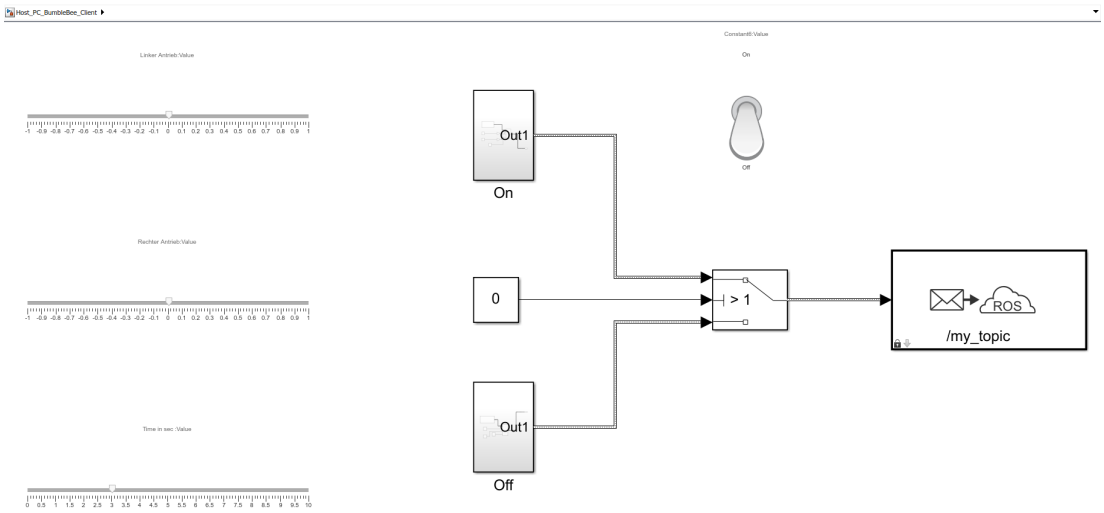

Abbildung 4.26: ROS-Knoten des Host-PC in der BumbleBee-Anwendung

Abbildung 4.27 zeigt das Simulink-Modell, aus dem mit der Hilfe des Simulink Coder C++ Code generiert wird. Das Simulink-Modell abonniert das *Topic „my_topic“* und schlüsselt die Nachricht *geometry_msg/Point* in drei Ausgänge auf.

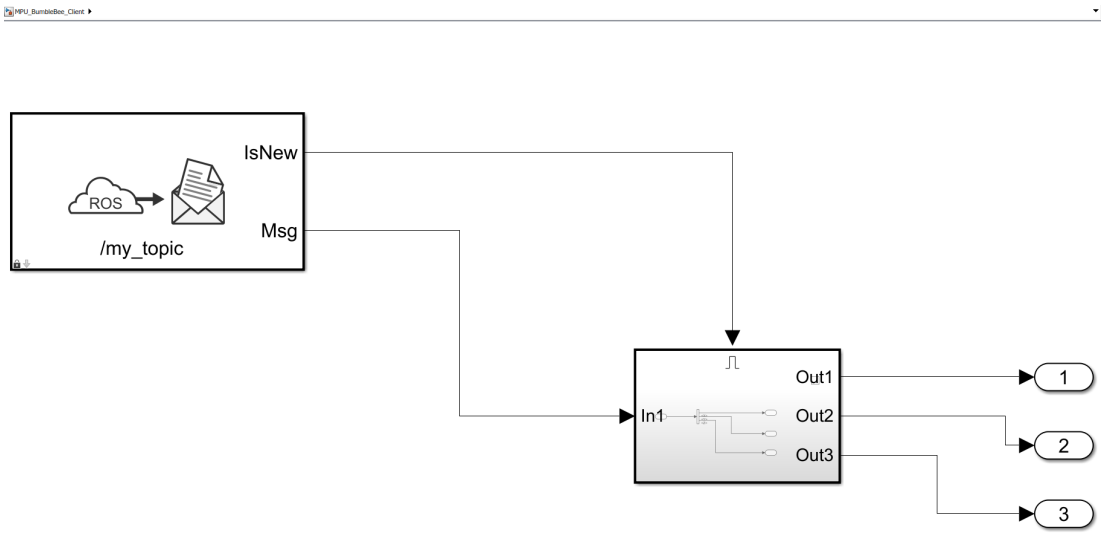

Abbildung 4.27: ROS-Knoten der MPU in der BumbleBee-Anwendung

Um aus dem Simulink-Modell (Abbildung 4.27) C++ Code zu generieren, müssen die folgenden Einstellungen im Simulink Coder gemacht werden: der *Solver-Type* wird auf *Fixed-step* und der *Solver* wird auf „ode3 (Bogacki-Shampine)“ gestellt [64].

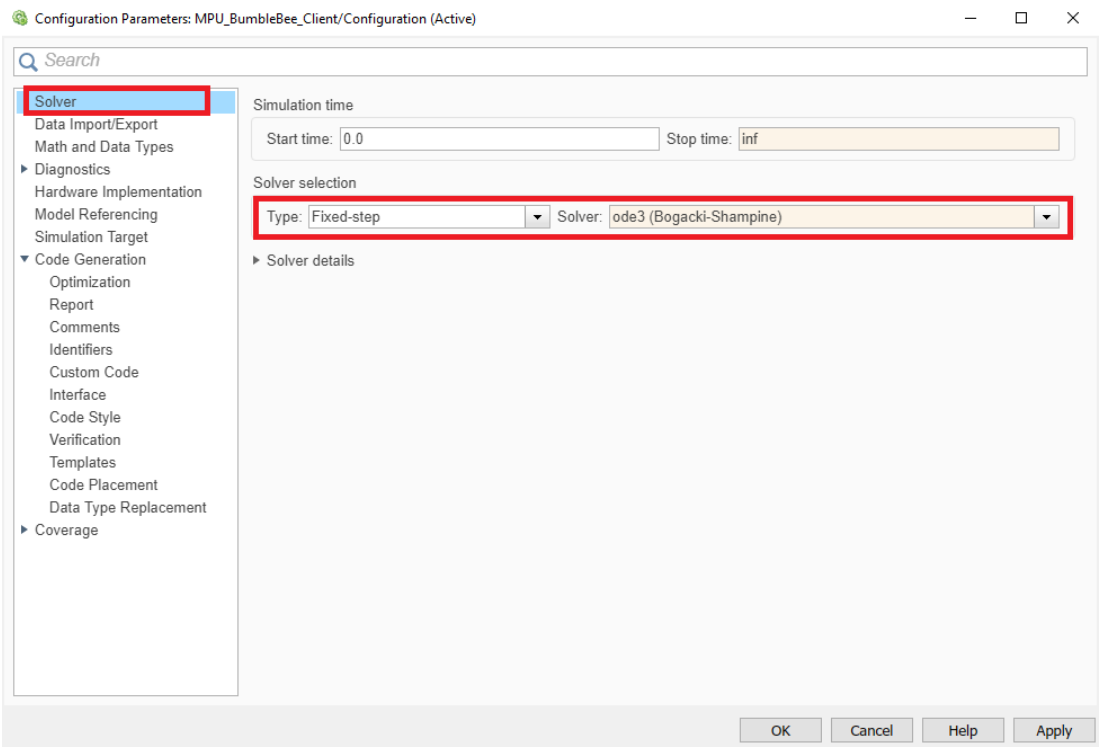

Abbildung 4.28: Simulink Coder *Solver* Einstellung

Die Einstellung des *Hardware board* wird auf *Roboter Operating System (ROS)* gestellt. Die Einstellung des *Device vendor* wird auf *ARM Compatible* gestellt. Die Einstellung des *Device type* wird auf *ARM Cortex* gestellt. [64]

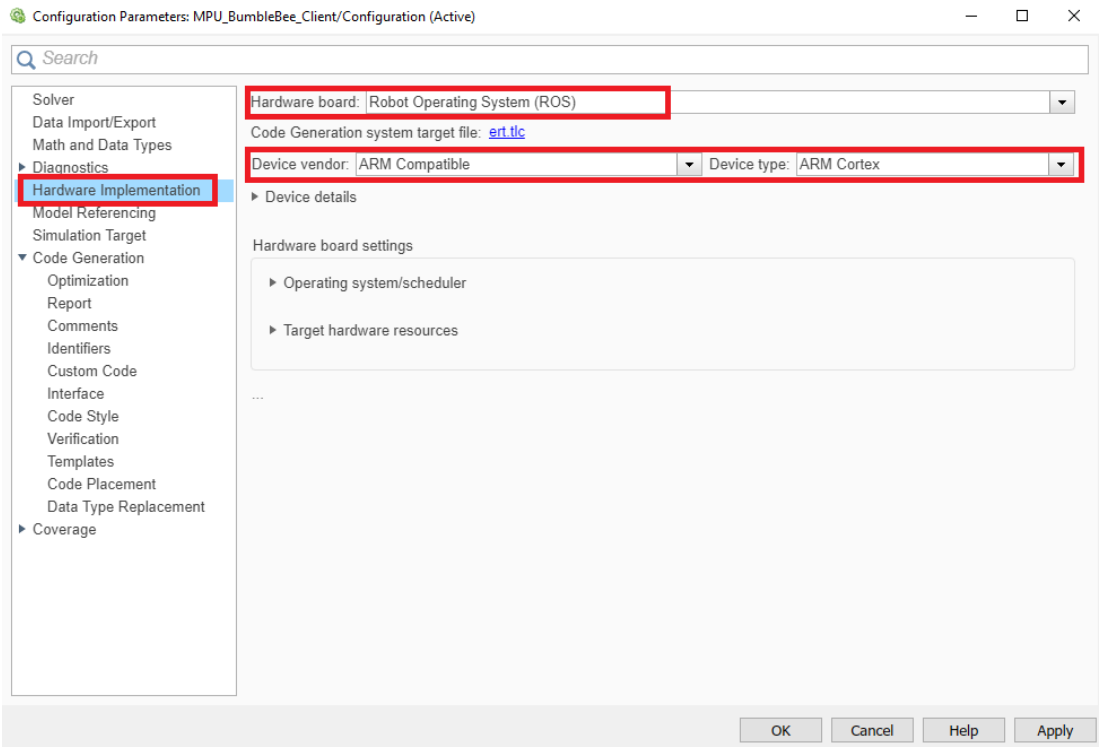

Abbildung 4.29: Simulink Coder *Hardware Implementation* Einstellung

Die Einstellung *Generate code only* wird gewählt. Die *Toolchain* Catkin wird ausgewählt. Die *Toolchain* Catkin ist das in ROS enthaltene *Build-System*. Catkin vereint CMake-Makros und Python-Skripte, um zusätzliche Funktionen zu dem Arbeitsablauf von CMake zu ergänzen. Das ermöglicht eine automatische Paketfindungs-Infrastruktur und die Unterstützung für das *Cross Compiling*. [65]

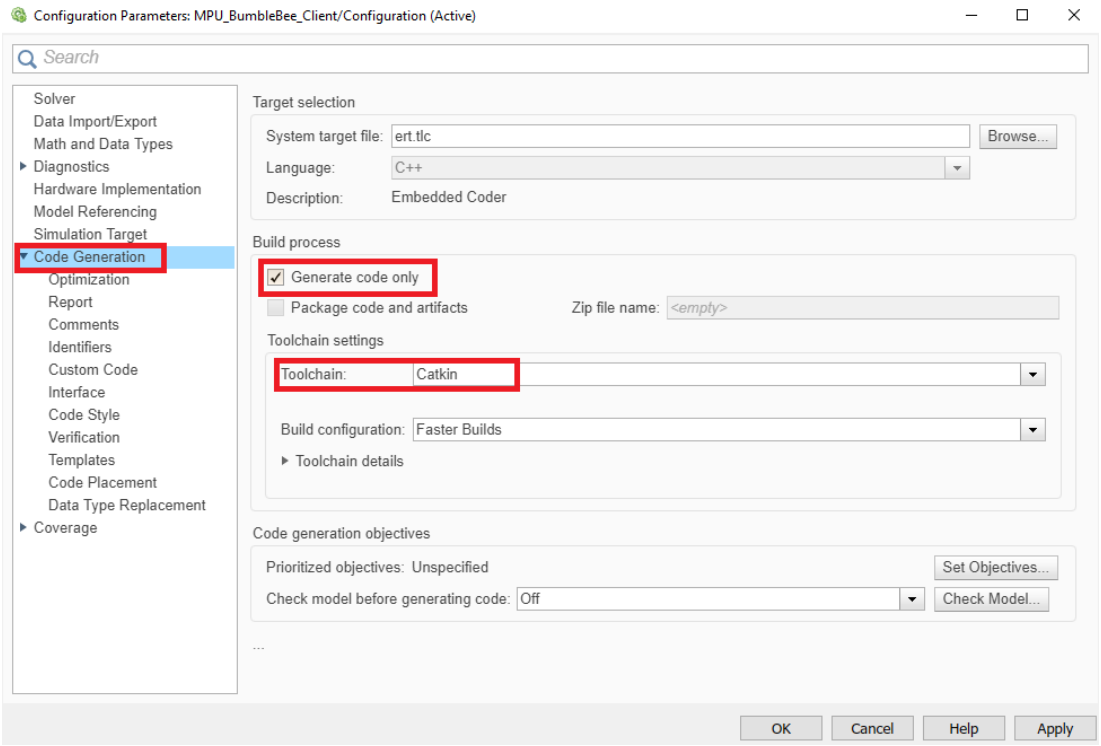

Abbildung 4.30: Simulink Coder *Code Generation* Einstellung

Nachdem die Einstellungen wie in den Abbildung 4.28, Abbildung 4.29 und Abbildung 4.30 gemacht wurden, wird mit der Codegenerierung gestartet.

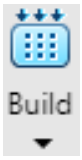

Abbildung 4.31: *Button* der den Prozess der Codegenerierung startet

Das Resultat der Codegenerierung ist ein Ordner, in dem sich ROS-Quellcode-Dateien und eine CMakeLists.txt Datei befinden. Um das Projekt für die Architektur der MPU

cross zu kompilieren, wird das *Environment-Setup-Skript* des mit *BitBake* gebauten SDK aufgerufen und die für den *Build*-Vorgang benötigten *Path*-Variablen exportiert.

Aufrufen des SDK *Environment-Setup-Skript*:

```
PC $> source /opt/st/stm32mp1/2.6-snapshot/environment-setup-cortexa
    7t2hf-neon-vfpv4-openstlinux_weston-linux-gnueabi
```

Exportieren der in der SDK enthaltenen ROS-*Path*-Variablen:

```
PC $> export ROS_ROOT=/opt/st/stm32mp1/2.6-snapshot/sysroots/cortexa
    7t2hf-neon-vfpv4-openstlinux_weston-linux-gnueabi/opt/ros/indigo
PC $> export PATH=$PATH:/opt/st/stm32mp1/2.6-snapshot/sysroots/
    cortexa7t2hf-neon-vfpv4-openstlinux_weston-linux-gnueabi/opt/ros/
    indigo/bin
PC $> export LD_LIBRARY_PATH=/opt/st/stm32mp1/2.6-snapshot/sysroots/
    cortexa7t2hf-neon-vfpv4-openstlinux_weston-linux-gnueabi/opt/ros/
    indigo/lib
PC $> export PYTHONPATH=/opt/st/stm32mp1/2.6-snapshot/sysroots/
    cortexa7t2hf-neon-vfpv4-openstlinux_weston-linux-gnueabi/opt/ros/
    indigo/lib/python2.7/site-packages
PC $> export CMAKE_PREFIX_PATH=/home/philip/catkin_ws/devel:/opt/st/
    stm32mp1/2.6-snapshot/sysroots/cortexa7t2hf-neon-vfpv4-
    openstlinux_weston-linux-gnueabi/opt/ros/indigo
PC $> export ROS_PACKAGE_PATH=/opt/st/stm32mp1/2.6-snapshot/sysroots
    /cortexa7t2hf-neon-vfpv4-openstlinux_weston-linux-gnueabi/opt/ros
    /indigo/share
```

Mit der Hilfe der CMakeLists.txt Datei, der Catkin *Toolchain* und der SDK wird der generierte Code *cross* kompiliert.

```
PC $> cd <generated code dir>
PC $> mkdir build
PC $> cd build
PC $> cmake ..
PC $> make
```

5 Verifikation

Die im Lastenheft Punkt A.1.2 festgehaltenen zu erreichenden Ziele werden anhand des Verifikationsplans (Tabelle 5.1) geprüft.
Im Verifikationsplan wurde dokumentiert, mit welchen Hilfsmitteln die geforderten Ziele überprüft werden.

Nr./ID	Nichttechnischer Titel	Verifikation der Anforderung	Hilfsmittel
ANF_01	Kommunikationder Prozessoren	Es wird geprüft, ob eine char String vom Cortex-A7 zum Cortex-M4 gesendet werden kann. Es wird geprüft, ob eine char String vom Cortex-M4 zum Cortex-A7 gesendet werden kann. Ergebnis: Die String wird in beide Richtungen fehlerfrei übertragen. **Test bestanden.**	MPU, Computer
ANF_02	Starten des Programmes nach Boot des MPU	Es wird geprüft, ob das Programm des Cortex-A7 und die Firmware des Cortex-M4 nach dem Booten automatisch gestartet werden. Ergebnis: Das Programm und die Firmware werden nach dem Boot-Vorgang automatisch gestartet. Es gibt während des Bootens einen Fehler. Die PWM-Pins sind während den Boot-Vorgang auf dem *High Level*. **Test nicht bestanden.**	MPU, Computer, Oszilloskop

Nr./ID	Nichttechnischer Titel	Verifikation der Anforderung	Hilfsmittel
ANF_03	Steuerung durch Geräte, die im WLAN Hotspot eingeloggt sind	Es wird geprüft, ob Geräte, die mit dem WLAN-Hotspot der MPU verbunden sind, den BumbleBee steuern können. Ergebnis: BumbleBee kann von Geräten, die mit dem WLAN-Hotspot verbunden sind und auf den ROS oder die Simulink ROS *Toolbox* installiert ist, gesteuert werden. **Test bestanden.**	BumbleBee, WLAN fähiger Computer Simulink, ROS Toolbox
ANF_04	Flash-Vorgang per WLAN	Es wird geprüft, ob die *Firmware* des Cortex-M4 über WLAN geflashet werden kann. Ergebnis: Die *Firmware* des Cortex-M4 kann über WLAN geflashet werden. **Test bestanden.**	MPU, Computer
ANF_05	Matlab Simulink Code Generierung	Es wird geprüft, ob Code, der in Matlab Simulink generiert wurde für die MPU *cross*-kompiliert werden kann. Ergebnis: Code, der in Matlab Simulink generiert wurde kann für den Cortex-A7 *cross*-kompiliert werden **Test bestanden.**	MPU, Computer, Mathlab Simulink, ROS Toolbox

Tabelle 5.1: Verifikationsplan

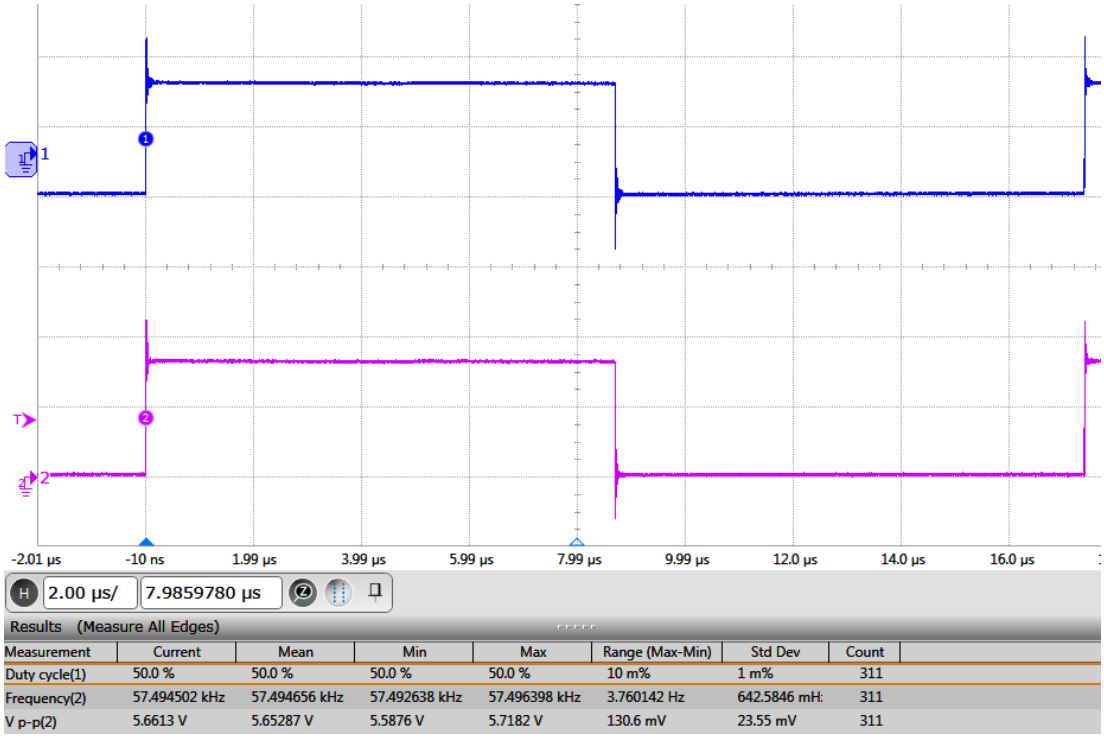

Abbildung 5.1: Oszilloskop-Aufnahme der PWM bei einem DC von 50 %

6 Fazit und Ausblick

Die vorliegende Arbeit befasst sich mit der Implementierung der Mikroprozessor-Plattform „STM32MP157C-DK2“ in das Miniaturfahrzeug „BumbleBee“. Es wurde manuell und modellbasiert Software entwickelt. Es wurde eine Interprozessor Kommunikation implementiert und erprobt. Das automatische Laden der Cortex-M4 *Firmware* nach dem Boot der Plattform wurde implementiert. Die Steuerung des Miniaturfahrzeugs durch einen WLAN-Client wurde durch modellbasierte Entwicklung implementiert. Die Erprobung des Flash-Vorgangs der Cortex-M4 *Firmware* per WLAN wurde erprobt. Die modellbasierte Entwicklung von Quellcode für den Cortex-A7 Prozessor wurde mit der Hilfe des Matlab Simulink *Coder* und der ROS *Toolbox* erprobt. Hinzu kam das Bauen einer Linux-Distribution und einer SDK mit dem *Build-Framework OpenEmbedded*. Dies war nötig, um das ROS auf der Mikroprozessor-Plattform zu installieren und um generierten Code *cross* zu kompilieren. Bei der modellbasierten Entwicklung mit der ROS-Toolbox in Matlab Simulink stellte sich heraus, dass die gemeinsame Nutzung von verschiedenen *Build*-Umgebungen während des Kompilierens zu einem erhöhten Aufwand führen kann. Bei dem automatischen Laden der Cortex-M4 *Firmware* wurde nicht bedacht, dass die verwendeten GPIO-Pins sich während des Boot-Vorgangs auf den *High*-Pegeln befinden können. Das hat zur Folge, dass die Motoren des BumbleBees während des Boot-Vorgangs in unterschiedliche Richtungen drehen. Nach dem Booten des Linux-Betriebssystemes und dem automatischen Start der *Firmware* werden die Motoren gestoppt.

Es wurde in einem einmaligen Versuch erprobt, ob die Cortex-M4 Firmware vor dem Booten des Betriebssystemes über das *bootcmd* [66] des *Bootloaders* U-Boot geladen werden kann. Dieser Versuch scheiterte.

In einem nachfolgenden Projekt könnte der automatische Start der *Firmware* vor dem Booten des Betriebssystems erprobt werden.

6.1 Auflistung von Stärken, Schwächen, Chancen und Risiken

Aufgelistet werden Erfahrungen, die bei der Inbetriebnahme und der Implementierung von Software und Firmware während der Arbeitsphase am BumbleBee Projekt mit der Mikroprozessor-Plattform „STM32MP1“, gemacht wurden.

Stärken

- Verursacht keine Lizenzkosten, da der Code des Kernels auf *General Public License v2.0* basiert [67]
- Inbetriebnahme durch ausführliche Dokumentation und Code-Beispiele des Herstellers möglich [68]
- Netzwerkverbindungen können schnell hergestellt werden [69]
- Schnelles Erstellen der Pin-Konfiguration und des Device Tree durch das Generieren mit STM32CubeMX [34]
- Viel freie Software verfügbar, wie zum Beispiel das verwendete meta-ros *Package* [67, 63]
- Kombination von Netzwerkanbindung durch Linux-Betriebssystem und Echtzeitfähigkeit durch den Coprozessor [8]

Schwächen

- Handhabung durch Windows Betriebssystem auf dem Host-Computer sehr erschwert bis nicht möglich
- Erhöhter Aufwand bei dem Zusammenführen von mehreren *Build-Environments* (Abschnitt 4.9)
- Erhöhter Einarbeitungsaufwand

Chancen

- Kostengünstige Entwicklung komplexer Systeme aufbauend auf der Basis von *Open Source* Software Komponenten [67]
- Durch Embedded-Linux besteht die Möglichkeit, Applikationen auf dieser Plattform zu entwickeln und diese auf angepasste Embedded-Linux-Systeme zu portieren [7]
- Es besteht die Möglichkeit, eine Android Distribution auf der Plattform zu installieren. [70]

Risiken

- Bei *Bugs* von *Open Source* Software ist man auf die Hilfe der *Community* angewiesen

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] R. Berezdivin, R. Breinig und R. Topp. »Next-generation wireless communications concepts and technologies«. In: *IEEE Communications Magazine* 40.3 (März 2002), S. 108–116. ISSN: 1558-1896. DOI: 10.1109/35.989768.

[2] T. Lewis. »Information appliances: gadget Netopia«. In: *Computer* 31.1 (Jan. 1998), S. 59–68. ISSN: 1558-0814. DOI: 10.1109/2.641978.

[3] Peter Marwedel. *Embedded System Design*. 1. Aufl. Kluwer Academic Publishers, 2003. ISBN: 9781402076909.

[4] STMicroelectronics. *STM32MP157C Datasheet*. https://www.st.com/resource/en/datasheet/stm32mp157c.pdf. (Besucht am 26. 12. 2019).

[5] Ralf Jesse. *Embedded Linux*. 1. Aufl. mitp Verlags GmbH & Co. KG, 2016. ISBN: 978-3-95845-061-5.

[6] Justus Wiedau und Tim Remmert und Martin Müller und Jesper Dietrich. »Entwicklung eines Miniaturfahrzeuges auf PCB-Ebene BumbleBee«. Entwicklungsprojekt. Hochschule Bochum - Bochum University of Applied Sciences, 2019.

[7] J. Hu und G. Zhang. »Research Transplanting Method of Embedded Linux Kernel Based on ARM Platform«. In: *2010 International Conference of Information Science and Management Engineering*. Bd. 2. Aug. 2010, S. 35–38. DOI: 10.1109/ISME.2010.191.

[8] STMicroelectronics. *RM0436 Reference manual STM32MP157*. https://www.st.com/resource/en/reference_manual/DM00327659.pdf. (Besucht am 16. 12. 2019).

[9] Nancy Itzeck. »Machbarkeitsstudie für ausgewählte E-Mobility-Konzepte für Deutschland 2020«. Bachelorarbeit. https://www.grin.com/document/175235: Hochschule für Technik und Wirtschaft Berlin, 2011. ISBN: 978-3-640-96471-0. (Besucht am 05. 01. 2020).

[10] DIN Deutsches Institut für Normung e.V. »Projektmanagement – Projektmanagementsysteme – Teil 2: Prozesse, Prozessmodell«. In: DIN 69901-2 (Jan. 2009).

[11] STMicroelectronics. »STM32 Cortex®-M4 MCUs and MPUs programming manual«. In: (Jan. 2019), S. 13.

[12] Dieter Wecker. *Prozessorentwurf: von der Planung bis zum Prototyp*. De Gruyter Oldenbourg Studium. De Gruyter Oldenbourg, 2015. ISBN: 9783110402964.

[13] Joachim Schröder und Tilo Gockel und Rüdiger Dillmann. »Embedded Linux«. In: *Embedded Linux Das Praxisbuch*. 1. Aufl. Springer International Publishing, 2009. ISBN: 978-3-540-78620-7. DOI: 10.1007/978-3-540-78620-7.

[14] STMicroelectronics. *Data brief Discovery kits with STM32MP157 MPUs*. https://www.st.com/resource/en/data_brief/stm32mp157c-dk2.pdf. (Besucht am 16. 01. 2020).

[15] RASPBERRY PI FOUNDATION. *Raspberry Pi 1 Modell A+ 512MB RAM*. https://www.raspberrypi.org/. (Besucht am 16. 01. 2020).

[16] BeagleBoard.org Foundation. *BeagleBone Black*. https://beagleboard.org/black/. (Besucht am 16. 01. 2020).

[17] Inc. CubieTech Limited. *cubietruck*. http://cubieboard.org/tag/cubietruck/. (Besucht am 16. 01. 2020).

[18] Linux Foundation. *Main Page Embedded Linux Wiki*. https://elinux.org/Main_Page. (Besucht am 16. 01. 2020).

[19] Microsoft. *Windows Embedded CE 6.0 R3*. https://www.microsoft.com/en-us/download/details.aspx?id=14226. (Besucht am 16. 01. 2020).

[20] SPiiD GmbH. *TECHNISCHE INNOVATION IST UNSER ANTRIEB*. http://www.embedix.com/. (Besucht am 16. 01. 2020).

[21] eCos. *ecos Home Page*. http://ecos.sourceware.org/. (Besucht am 16. 01. 2020).

[22] VxWorks Embedded Group. *VxWorks*. https://www.vxworks.net/. (Besucht am 16. 01. 2020).

[23] STMicroelectronics. *U-Boot overview*. https://wiki.st.com/stm32mpu/wiki/U-Boot_overview. (Besucht am 05. 02. 2020).

[24] Wolfgang Denk. *README u-boot*. https://gitlab.denx.de/u-boot/u-boot/raw/HEAD/README. (Besucht am 27. 01. 2020).

[25] DeviceTree c/o Linaro. *Devicetree Specification Release v0.2*. https://github.com/devicetree-org/devicetree-specification/releases/download/v0.2/devicetree-specification-v0.2.pdf. (Besucht am 27. 01. 2020).

[26] Thomas Petazzoni. *Device Tree for Dummies*. https://bootlin.com/pub/conferences/2013/elce/petazzoni-device-tree-dummies/petazzoni-device-tree-dummies.pdf. (Besucht am 16. 01. 2020).

[27] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1. Aufl. USA: Kluwer Academic Publishers, 1997. ISBN: 0792398947.

[28] Jean-Michel Berge, Oz Levia und Jacques Roulliard. *High-Level System Modeling: Specification Languages*. USA: Kluwer Academic Publishers, 1995. ISBN: 0792396324.

[29] STMicroelectronics. *Coprocessor management overview*. https://wiki.st.com/stm32mpu/wiki/Coprocessor_management_overview. (Besucht am 25. 12. 2019).

[30] STMicroelectronics. *Coprocessor management overview Picture*. https://wiki.st.com/stm32mpu/nsfr_img_auth.php/3/3e/Copro-sw-ipc-overview.png. (Besucht am 25. 12. 2019).

[31] Frank Storm. »OpenAMP – Ein Open Source Framework für asymmetrisches Multiprocessing«. In: *embedded-software.engineer Fachwissen für Software-Professionals* (Okt. 2018), S. 1. (Besucht am 17. 12. 2019).

[32] Multicore Association. *Open-source software framework for developing AMP systems application software*. https://github.com/OpenAMP/open-amp/. (Besucht am 22. 12. 2019).

[33] Multicore Association. *OPEN ASYMMETRIC MULTI PROCESSING (OpenAMP)*. https://www.multicore-association.org/workgroup/oamp.php. (Besucht am 15. 12. 2019).

[34] STMicroelectronics. *User manualSTM32CubeMX for STM32 configuration and initialization C code generation*. https://www.st.com/content/ccc/resource/technical/document/user_manual/10/c5/1a/43/3a/70/43/7d/DM00104712.pdf/files/DM00104712.pdf/jcr:content/translations/en.DM00104712.pdf. (Besucht am 26. 01. 2020).

[35] Fabian Sacilotto. *ROS.org*. http://wiki.ros.org/de. (Besucht am 28. 01. 2020).

[36] Tully Foote. *ROS.org Concepts*. http://wiki.ros.org/de/ROS/Concepts. (Besucht am 28. 01. 2020).

[37] Arno Bergmann und Kai Morwinski und Philip Geuchen. *Projektauftrag "Bumblebee"*. 2019-11-19.

[38] Philip Geuchen. *Lastenheft Version 1.0*. 2019-12-15.

[39] Jürgen Gausemeier, G. Lanza und U. Lindemann. »Produkte und Produktionssysteme integrativ konzipieren: Modellbildung und Analyse in der frühen Phase der Produktentstehung«. In: 2012.

[40] Dimitri van Heesch. *Doxygen Generate documentation from source code*. http://www.doxygen.nl/. (Besucht am 10. 02. 2020).

[41] STMicroelectronics. *STM32MP1 Package Decision*. https://wiki.st.com/stm32mpu/wiki/Which_STM32MPU_Embedded_Software_Package_better_suits_your_needs. (Besucht am 15. 12. 2019).

[42] STMicroelectronics. *STOpenEmbedded*. https://wiki.st.com/stm32mpu/wiki/OpenEmbedded. (Besucht am 15. 12. 2019).

[43] Openembedded.org. *OpenEmbedded*. https://www.openembedded.org/wiki/Main_Page. (Besucht am 04. 02. 2020).

[44] Texas Instruments. *Application Note: „DRV10970 Three-Phase Brushless DC Motor Driver (Rev. A)“*. http://www.ti.com/document-viewer/DRV10970/datasheet/important_notice#ImpNotice001. (Besucht am 31. 01. 2020).

[45] Asael Dror for Chips und Inc. Technologies. *Hardware Semaphores in a Multi-Processor Environment*. US Patent 5,276,886. Jan. 1994. URL: https://patentimages.storage.googleapis.com/e4/b0/2b/bcf4467cf6bd04/US5276886.pdf.

[46] STMicroelectronics. *Description of STM32H7 HAL and low-layer drivers*. https://www.st.com/content/ccc/resource/technical/document/user_manual/group0/40/ee/88/53/f6/1e/4c/87/DM00392525/files/DM00392525.pdf/jcr:content/translations/en.DM00392525.pdf. (Besucht am 26. 12. 2019).

[47] Arnopo. *libmetal*. https://github.com/OpenAMP/libmetal. (Besucht am 30. 01. 2020).

[48] XILINX. *Libmetal and OpenAMP User Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1186-zynq-openamp-gsg.pdf, Mai 2019. (Besucht am 30. 01. 2020).

[49] Mentor Graphics Corporation. *OpenAMP Framework User Reference*. (Besucht am 15. 12. 2019).

[50] MQTT. *MQTT*. https://mqtt.org/. (Besucht am 03. 02. 2020).

[51] Apache Software Foundation. *kafka*. https://kafka.apache.org/. (Besucht am 03. 02. 2020).

[52] Google Developers. *Protocol Buffers*. https://developers.google.com/protocol-buffers. (Besucht am 02. 02. 2020).

[53] Hauke Mönck. »Entwicklung einer WLAN Schnittstelle mit Google Protobuf für humanoide Roboter«. Bachelorarbeit. Freie Universität Berlin, 2014.

[54] Google Developers. *Encoding Protocol Buffers*. https://developers.google.com/protocol-buffers/docs/encoding?hl=de. (Besucht am 25. 12. 2019).

[55] Aimonen Petteri. *Nanopb - protocol buffers with small code size*. http://koti.kapsi.fi/jpa/nanopb/. (Besucht am 28. 12. 2019).

[56] STMicroelectronics. *Linux remoteproc framework overview*. https://wiki.st.com/stm32mpu/wiki/Linux_remoteproc_framework_overview#Kernel_configuration. (Besucht am 02. 02. 2020).

[57] Uwe Jahn und Carsten Wolff und Peter Schulz. »Concepts of a Modular System Architecture for Distributed Robotic Systems«. In: *Computers 2019* (März 2019). (Besucht am 04. 02. 2020).

[58] Dennis Hotze und Dominik Eickmann. »Entwicklung und Verifikation eines autonomen Logistik-Fahrzeugs«. Masterthesis. Hochschule Bochum - Bochum University of Applied Sciences, 2018.

[59] Giuliano Montorio und Hannes Dittmann. »Implementierung einer Schlupfregelung per Model-Based Design sowie einer SLAM-Kartografierung für ein autonomes Logistik-Fahrzeug«. Bachelorarbeit. Hochschule Bochum - Bochum University of Applied Sciences, 2019.

[60] Inc. The MathWorks. *ROS Toolbox*. https://de.mathworks.com/products/ros.html. (Besucht am 04. 02. 2020).

[61] Inc. The MathWorks. *Simulink Coder*. https://de.mathworks.com/products/simulink-coder.html?s_tid=srchtitle. (Besucht am 04. 02. 2020).

[62] Chris Larson Richard Purdie und Phil Blundell. *BitBake User Manual*. https://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html. (Besucht am 28. 01. 2020).

[63] Lukas Bulwahn. *OpenEmbedded Layer for ROS Applications*. https://github.com/ros/meta-ros. (Besucht am 04. 02. 2020).

[64] Inc. The MathWorks. *Simulink User's Guide R2018b*. https://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf. (Besucht am 07. 02. 2020).

[65] Open Source Robotics Foundation. *Catkin Conceptual Overview*. http://wiki.ros.org/catkin/conceptual_overview. (Besucht am 04. 02. 2020).

[66] STMicroelectronics. *How to start the coprocessor from the bootloader*. https://wiki.st.com/stm32mpu/wiki/How_to_start_the_coprocessor_from_the_bootloader#Automatic_start. (Besucht am 08. 02. 2020).

[67] STMicroelectronics. *OpenSTLinux licenses - v1.1.0*. https://wiki.st.com/stm32mpu/wiki/OpenSTLinux_licenses_-_v1.1.0. (Besucht am 10. 02. 2020).

[68] STMicroelectronics. *Development zone*. https://wiki.st.com/stm32mpu/wiki/Development_zone. (Besucht am 10. 02. 2020).

[69] STMicroelectronics. *How to configure a wlan interface on hotspot mode*. https://wiki.st.com/stm32mpu/wiki/How_to_configure_a_wlan_interface_on_hotspot_mode. (Besucht am 10. 02. 2020).

[70] STMicroelectronics. *STM32MPU distribution for Android*. https://wiki.st.com/stm32mpu/wiki/STM32MPU_distribution_for_Android. (Besucht am 09. 02. 2020).

A Anhang

A.1 Inhalt Daten-CD

1 Projektauftrag BumbleBee

2 Lastenheft Version 1.0

3 Installation der *Software Packages*

4 Doxygen generierte Dokumentation der Programme

- 1 Cortex-A7 Doxygen generierte Dokumentation
- 2 Cortex-M4 Doxygen generierte Dokumentation

5 BumbleBee Programm Code

- 1 BumbleBee Cortex-A7 Programmcode
- 2 BumbleBee Cortex-M4 Programmcode

6 Simulink-Modelle

- 1 ROS-BumbleBee
- 2 ROS-Host-Computer