

Einführung	5
Vorwort	6
Abstract	7
Deutsch	7
Italiano	7
English	8
Motivation	9
Name	9
Warum Rust?	9
Eigenständigkeitserklärung	10
Lizenz	11
Hardware	13
Blockschaltbild	15
Prozessor	16
Schematic	17
Prozessor	17
Spannungsversorgung	19
SD-Karte	22
Boot Pins	24
Platine	26
Software	35
Github Repository	36
Der Arm Cortex A8	36
Die CPU	38
CPU Modi	38
Register	40
Coprocessor 15	42
Vektoren	42
Die MMU	43
Page Tables	43
Level 1	44
Level 2	45
Translation Lookaside Buffer	46
Booten	47
Kompilation	47
Linker Script	48
Cargo	49
Build Script	49
Startup	51
Stack Setup	52
Cache Setup	53
Exceptions Setup	54
AM335 Hardware	55

Interrupts	56
Aktivierung	56
Handler	57
Clocks	59
Pinmuxxing	60
GPIO	61
Initialisierung	61
Pin Modus	61
Pin Schreiben	62
Pin Lesen	62
Interrupts	62
Initialisierung	62
Registration	63
Handling	64
I2C	65
Initialisierung	66
Operation	67
Transmit	67
Interrupts	67
XRDY	69
XDR	69
ARDY	70
NACK	71
Timer	72
Initialisierung	73
Interrupts	74
Modularität	74
Kernel	76
Sysclock	77
Syscalls	78
Handler	79
Rust Handler	81
Signatur	81
Logik	82
Syscall Enum	83
Multitasking	85
Preemptive Multitasking	86
Cooperative Multitasking	89
Task Wiederherstellung	91
Scheduling	92
Paging	96
Initialisierung	97
L1 Page Table	98
L2 Page Table	99

Erstellen eines L2 Page Tables	100
Anwendung	102
Hinzufügen von Tasks	103
User Binary	106
Strukturierung	107
Heap	108
Kernel	109
Libfenix	110
Projektmanagement	113
Projekttantrag	115
Pflichtenheft	117
Gantt-Diagramm	119
Kostenberechnung	123
Beraterstunden	124
Arbeitsstunden	124
Platine	125
BOM	126
Gesamtausgaben	127
Abbildungsverzeichnis	129
Fazit	133

Einführung

Vorwort

Sehr geehrter Leser,

Mein Name ist Felix Salcher und ich besuche derzeit die 5. Klasse der Technologischen Fachoberschule „Oskar von Miller“ in Meran mit dem Schwerpunkt Elektronik und Elektrotechnik.

Diese Dokumentation beschreibt mein Maturaprojekt, im Rahmen dessen ich selbstständig ein eigenes Betriebssystem entwickelt habe. Das Projekt sah vor, sowohl eine Hardware- als auch eine Softwarekomponente zu realisieren und die Ergebnisse anschließend zu präsentieren. Für die Umsetzung standen uns wöchentlich fünf Unterrichtsstunden zur Verfügung. Da mein Projekt jedoch sehr umfangreich war, investierte ich den Großteil der Arbeit zusätzlich in meiner Freizeit.

Während der Entwicklung konnte ich wertvolle Erfahrungen sammeln, insbesondere hinsichtlich der Funktionsweise von Betriebssystemen und der Architektur der CPU. Zudem hat das Projekt meine Begeisterung für hardwarenahe Programmierung geweckt – ein Bereich, in dem ich zuvor nur wenig Erfahrung hatte, da ich hauptsächlich im Serverbereich programmierte. Ich bin überzeugt, dass mir dieses Projekt wertvolle Erfahrungen für meinen weiteren Karriereweg mitgegeben hat.

Ich hoffe, dass diese Dokumentation einen kleinen Einblick in die Welt der Betriebssysteme vermittelt und vielleicht auch Ihr Interesse für dieses faszinierende Thema weckt.

Abstract

Deutsch

Das Ziel meines Maturaprojekts war die Entwicklung eines minimalistischen Kernels für einen Mikrocontroller. Dabei durfte kein externer Code verwendet werden, sodass die gesamte Logik von Grund auf selbst implementiert werden musste. Zusätzlich wurde eine eigene Leiterplatte entworfen, die mit einem ARM-Cortex-Prozessor bestückt ist.

Der Kernel erfüllt zentrale Aufgaben eines Betriebssystems wie Multitasking, bei dem mehrere Prozesse parallel ausgeführt werden können. Jeder Task läuft dabei im User Mode, erhält einen fest zugewiesenen Speicherbereich im RAM und ist so davor geschützt, auf andere Speicheradressen – insbesondere die des Kernels – zuzugreifen. Dieses Verfahren der Speicherisolierung wird als Paging bezeichnet.

Die Kommunikation zwischen Kernel und Tasks erfolgt über Syscalls. Darüber hinaus stellt der Kernel grundlegende Schnittstellen zur Steuerung von Peripherie bereit, beispielsweise für GPIO-Pins oder den I2C-Bus.

Italiano

L'obiettivo del mio progetto di maturità è stato lo sviluppo di un kernel minimalista per un microcontrollore. Per realizzarlo non sono state utilizzate librerie esterne: tutto il codice è stato scritto manualmente. Inoltre, è stata progettata una scheda elettronica dedicata con un processore ARM Cortex.

Il kernel implementa funzioni fondamentali di un sistema operativo, tra cui il multitasking, che consente l'esecuzione concorrente di più task. Ogni task viene eseguito in modalità utente (User Mode) e dispone di una propria regione di memoria RAM, senza la possibilità di accedere a spazi non autorizzati, garantendo così la protezione del kernel. Questa tecnica è conosciuta come paginazione (paging).

La comunicazione tra il kernel e i task avviene tramite chiamate di sistema (syscall). Inoltre, il kernel fornisce interfacce per la gestione delle periferiche principali, come GPIO e il bus I²C.

English

The goal of my final project was to develop a minimalist kernel for a microcontroller. No external libraries were used; instead, all code was written from scratch. To support the implementation, a custom board with an ARM Cortex processor was designed.

The kernel implements essential operating system functions, including multitasking, allowing multiple tasks to run concurrently. Each task is executed in User Mode and is assigned a dedicated block of memory, preventing unauthorized access to other memory regions and ensuring kernel protection. This mechanism is known as paging.

Communication between the kernel and user-mode tasks is implemented through system calls (syscalls). In addition, the kernel provides interfaces for controlling peripherals such as GPIO and the I²C bus, enabling interaction with external hardware.

Motivation

Ich habe mich für die Entwicklung eines eigenen Betriebssystems als Maturaprojekt entschieden, da mich dieses Themengebiet schon seit längerem fasziniert. Durch die eigenständige Umsetzung wollte ich nicht nur mein Wissen im Bereich der Softwareentwicklung erweitern, sondern auch ein tieferes Verständnis für den Aufbau und die Funktionsweise von Prozessoren erlangen. Ein solches Projekt erwies sich dafür als besonders geeignet, da es sowohl theoretische als auch praktische Aspekte der Informatik vereint.

Name

Bei der Namenswahl orientierte ich mich an Linux, das nach seinem Entwickler Linus Torvalds benannt wurde. Daher entschied ich mich, mein Projekt Fenix zu nennen. Ursprünglich trug es den Titel MicroOS, dieser erschien mir jedoch zu generisch, weshalb ich mich schließlich entschloss, das Projekt umzubenennen. In einigen Dokumenten findet sich daher noch der alte Name.

Als Logo wählte ich einen Phönix, da er in seiner englischen Aussprache mit dem Namen meines Projekts entspricht.

Warum Rust?

Da wir in der Schule nur C++ sowie Assembly verwenden, mag Rust als Programmiersprache für dieses Projekt unüblich erscheinen. Ich habe Rust gewählt, da ich gerne immer neue Sprachen ausprobieren. So habe ich bei meinem Füla-Projekt 2023/24 Golang als Programmiersprache verwendet, um diese zu lernen. Rust hat hierbei einige Features, die es im Vergleich zu C-basierten Sprachen wie C++ oder C sehr herausheben.

Hier die meiner Meinung nach wichtigsten Features von Rust für dieses Projekt:

- **Memory Safety:** Das wahrscheinlich wichtigste Feature von Rust ist die garantierter Memory Safety. Dabei verwendet Rust den “Borrow Checker”, dieser überprüft den geschriebenen Code, ob die statischen Regeln eingehalten werden, durch die der Compiler dann Memory Safety garantiert. Probleme, die dadurch eliminiert werden, sind u.a.: Null Pointer, use after free, Dangling Pointer und Race Conditions.
- **Error Handling:** In vielen C-basierten Sprachen wie C++ wird Try Catch zum Überprüfen von Fehlern benutzt. Dies ist jedoch nicht optimal, da das Überprüfen dort optional ist. Bei Rust hingegen muss jeder Fehler überprüft werden, bevor der Wert verwendet werden kann.
- **Type System:** Das Type System in Rust ist ein weiterer Punkt für die Verwendung von Rust. Besonders Enums in Rust sind ein sehr großer Vorteil und werden oft verwendet. Dabei ist das *match* Statement in Rust sehr sinnvoll, im Gegenzug zu dem *switch* Statement kann dieses einen Wert direkt zurückgeben, dabei ist vor allem das “exhaustive matching” ein sehr großer Vorteil, bei dem alle Möglichkeiten überprüft werden müssen.
- **Tooling:** Im Gegensatz zum C Ecosystem, mit einem guten Dutzend an verschiedenen Tools zur Kompilation, LSP, Formatter, Compiler usw. hat Rust alle Tools, welche verwendet werden müssen, bereits integriert, was das Entwickeln von Software deutlich einfacher macht.

Eigenständigkeitserklärung

Hiermit versichere ich, Felix Salcher, Schüler der Technologischen Fachoberschule "Oskar von Miller" mit Schwerpunkt Elektronik und Elektrotechnik in Meran, dass ich die vorliegende Maturaarbeit mit dem Titel „Fenix“ selbstständig und ohne unerlaubte Hilfe Dritter angefertigt habe. Sämtliche Ideen, Daten und Textpassagen, die ich wörtlich oder sinngemäß aus Büchern, Zeitschriften, Internetquellen oder persönlichen Gesprächen übernommen habe, sind im Text als solche gekennzeichnet und vollständig im Literatur- und Quellenverzeichnis aufgeführt. Ich versichere ferner, dass ich neben den im Vorwort genannten Personen und Hilfsmitteln keinerlei weitere Unterstützung in Anspruch genommen habe und dass ich die Arbeit in dieser oder einer ähnlichen Form keiner anderen Prüfungsbehörde vorgelegt habe.

Meran, den 19. Mai 2025

Projektleiter, Felix Salcher

Fachlehrperson: De Tomaso Martin

Fachlehrperson: Ivan Huber

Direktor: David Augscheller

Lizenz

Die Lizenzierung dieses Projekts (einschließlich Quellcode und Dokumentation) erfolgt unter der GNU General Public License (GPL) Version 3.0. Diese Lizenz gewährt allen Nutzern weitreichende Rechte, wie zum Beispiel die Freiheit:

- Das Projekt für jeden Zweck zu nutzen.
- Die Funktionsweise zu untersuchen und an eigene Bedürfnisse anzupassen.
- Änderungen vorzunehmen und Verbesserungen zu entwickeln.
- Kopien des Originals oder der Modifikationen weiterzuverbreiten.

Als zentrale Bedingung (Copyleft) verlangt die GPL jedoch, dass jegliche Weitergabe des Projekts oder davon abgeleiteter Werke unter den gleichen Lizenzbedingungen erfolgen muss. Dies gewährleistet den dauerhaften Charakter als freie Software.

Die gesamte Lizenz kann im Github Repository unter folgendem Link aufgerufen werden:
<https://github.com/salfel/fenix/blob/master/LICENSE>

Hardware

Am Anfang des Projektes wollte ich noch den AM62 Prozessor von TI verwenden. Nach einigen Wochen der Entwicklung dieses Prozessors beschloss ich jedoch, den weitaus einfacheren AM335 zu verwenden. Dies lag vor allem daran, dass ich bei der Entwicklung des AM62 die gesamte Platine von "scratch" entwickelte, also alles selbst recherchieren und designen musste. Da ich keine Erfahrung in der Entwicklung von solch komplexen Projekten hatte, musste ich den AM62 nach einigen Wochen schließlich aufgeben, da dessen Entwicklung zu viel Zeit brauchte.

Stattdessen entschloss ich mich, den AM335 zu verwenden, da bereits das *Beaglebone Black*, welches auf diesem Prozessor basiert, entwickelt wurde. Auf dessen Design wurde meine Platine aufgebaut, wobei nur die Wahl der Bauteile der des *Beaglebone* entspricht, das Routen der Platine wurde von mir selbst durchgeführt.

Dabei ist zu beachten, dass die Platine sehr minimalistisch entwickelt wurde, um Entwicklungszeit zu sparen und vor allem um einen Prototypen zu entwickeln, auf welchem das Betriebssystem ausgeführt werden kann.

Blockschatzbild

In der folgenden Abbildung ist das Blockschatzbild der Platine dargestellt.

Der PMIC wird hierbei durch USB mit Strom versorgt. Dieser stellt anschließend die nötigen Spannungen bereit. Unter anderem kann er auch über I2C programmiert werden und speist auch die Mikro-SD-Karte mit 3.3 V.

Die Mikro-SD-Karte verbindet insgesamt vier Datenleitungen mit dem Prozessor. Unter anderem führen noch der zu verwendende Clock der Mikro-SD-Karte und die CMD-Leitung, an welcher der Prozessor die Befehle an die Mikro-SD-Karte sendet.

Zudem sind hier noch die Boot Pins und der Pin Header aufgezeichnet. Die Boot Pins haben die Aufgabe, dem Prozessor das Gerät mitzuteilen, welches er zum Booten verwenden soll, da er verschiedene

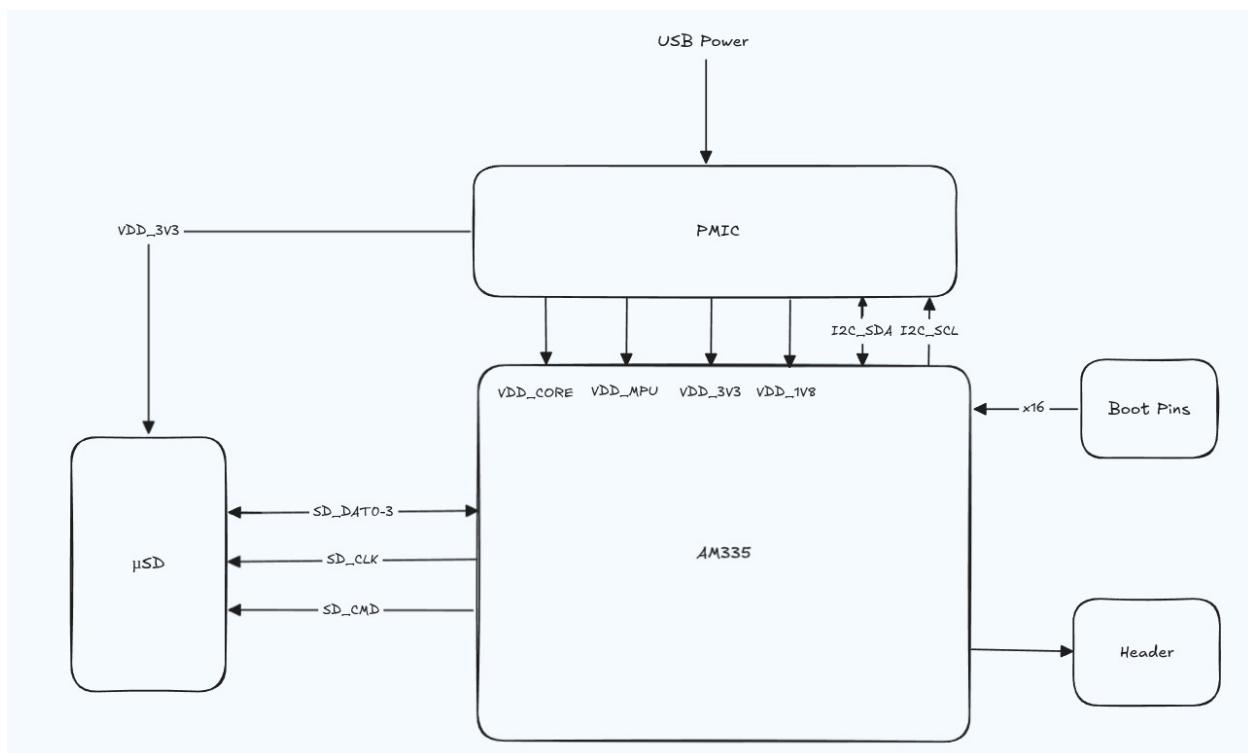


Abbildung 1. Blockschatzbild der Platine

Prozessor

Als Prozessor des Projektes wurde ein Prozessor der AM335 Familie von Texas Instruments verwendet. Dieser basiert auf der ARM Cortex-A8 Architektur und hat eine Taktfrequenz von bis zu 1 GHz.

Der AM335 hat eine breite Palette an Anwendungen und kann unter anderem in der Industrieautomation, Medizintechnik oder Bedienelementen verwendet werden. Der große Vorteil des AM335 besteht in der guten Balance von Rechenleistung, Kosteneffizienz, Leistungseffizienz und seinen vielen Peripherien, darunter UART, I2C, USB, CAN und vielen mehr, wodurch er besonders für industrielle Anwendungen sehr attraktiv ist.

Der AM335 kann unter anderem nicht nur für industrielle Anwendungen verwendet werden. Er hat hierbei genug Leistung, um Betriebssysteme wie Linux auszuführen. Durch Boards wie das Beaglebone Black, welches den AM335 verwendet, kann er unter anderem auch als Desktop-Prozessor verwendet werden. Durch die Implementierung von HDMI oder LCD kann ein Bildschirm angesteuert werden, einige der Prozessoren der AM335-Familie besitzen außerdem noch eine GPU, um das Anzeigen von GUIs zu optimieren.

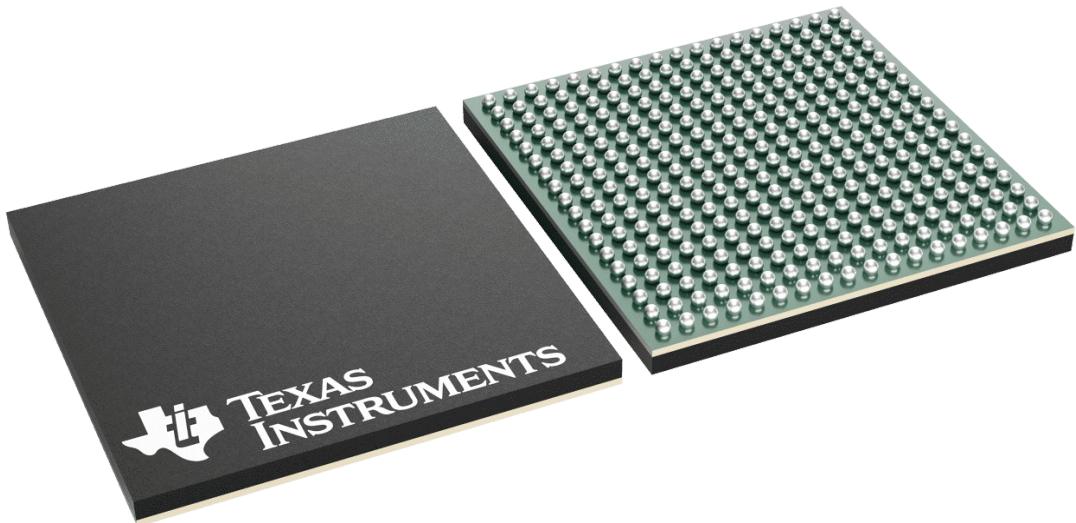


Abbildung 2. AM335 Prozessor <https://www.ti.com/product/AM3358>

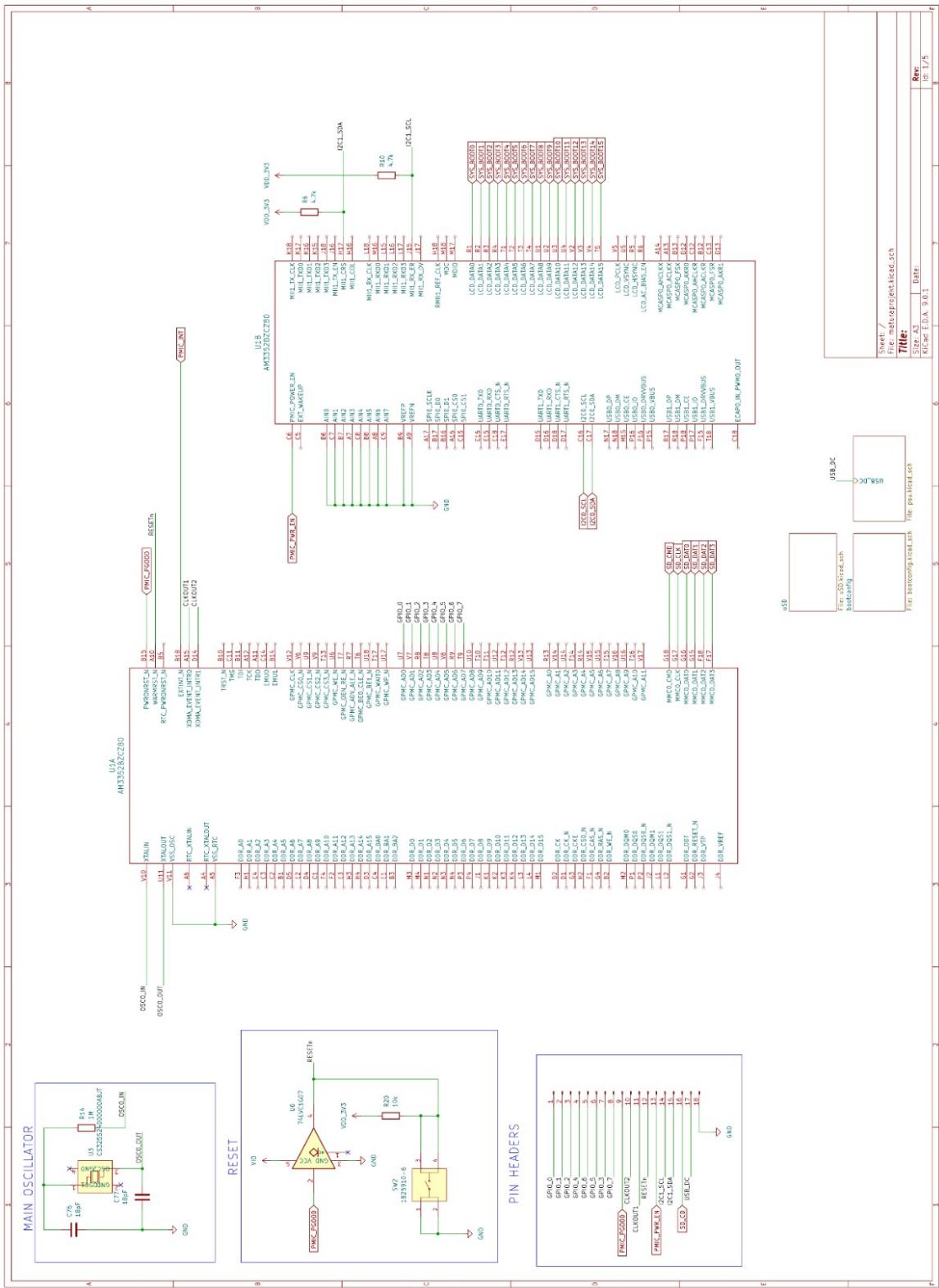
Schematic

Prozessor

Im Projekt wurde ein Prozessor der Familie AM335 von Texas Instruments verwendet. Dieser basiert auf einem ARM Cortex-A8 und hat eine Taktfrequenz von bis zu 1 GHz. Er beinhaltet eine Menge von Schnittstellen und Peripherien, darunter unter anderem USB, UART, I2C, Ethernet, CAN und GPIO.

In der folgenden Abbildung kann man das Schematic Sheet des Prozessors erkennen. Darin enthalten sind:

- Der Quarz, welcher dem Prozessor eine konstante Taktfrequenz von 25MHz liefert, ist nicht die primäre Taktfrequenz des Prozessors.
- Einen Reset Taster, um den Prozessor zurückzusetzen.
- Einen Pin Header um verschiedene Signale debuggen zu können.
- Leitungen, welche den Prozessor mit verschiedenen Peripherien verbinden, wie zum Beispiel I2C und GPIO.



Spannungsversorgung

Als PMIC wurde der *TPS65217* verwendet. Dieser ist der vorgeschlagene PMIC für den AM335, es kann jedoch auch ein anderer verwendet werden. Doch der *TPS65217* ist bei Weitem die beste Option, da dessen Kompatibilität mit dem AM335 bereits vielfach getestet wurde. Er enthält unter anderem 3 Step Down Converter und 4 LDOs. Dabei kann zudem noch eine Batterie an den PMIC angeschlossen werden, welcher anschließend das Laden und Entladen dieser verwaltet. und bietet zudem Mechanismen zum Versorgen der Spannung über eine externe Spannungsversorgung. Außerdem können bestimmte Mechanismen des *TPS65217* über I2C programmiert werden. Um die Stabilität der Spannung zu kontrollieren, bietet der PMIC zudem 2 Signale, durch welche der Prozessor oder externe Bauteile bestimmte Aktionen ausführen können, wenn die Versorgung nicht mehr stabil ist.

Die erforderlichen Signale werden anschließend mit dem Prozessor verbunden, wobei eine Menge an Stützkondensatoren für die verschiedenen Spannungen nötig sind, um die Spannung bei Einbrüchen möglichst stabil zu halten.

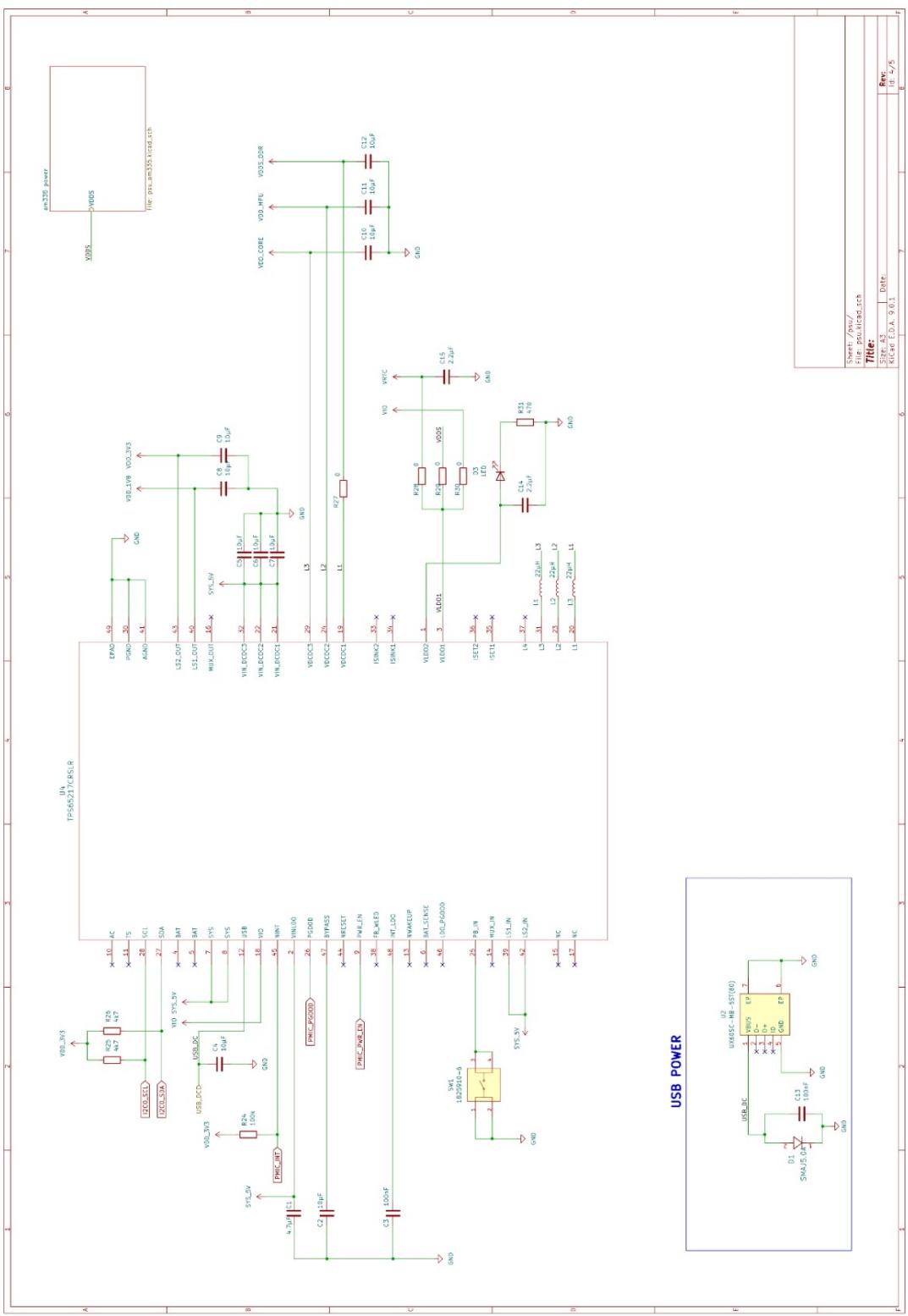


Abbildung 4. PSU Schematic

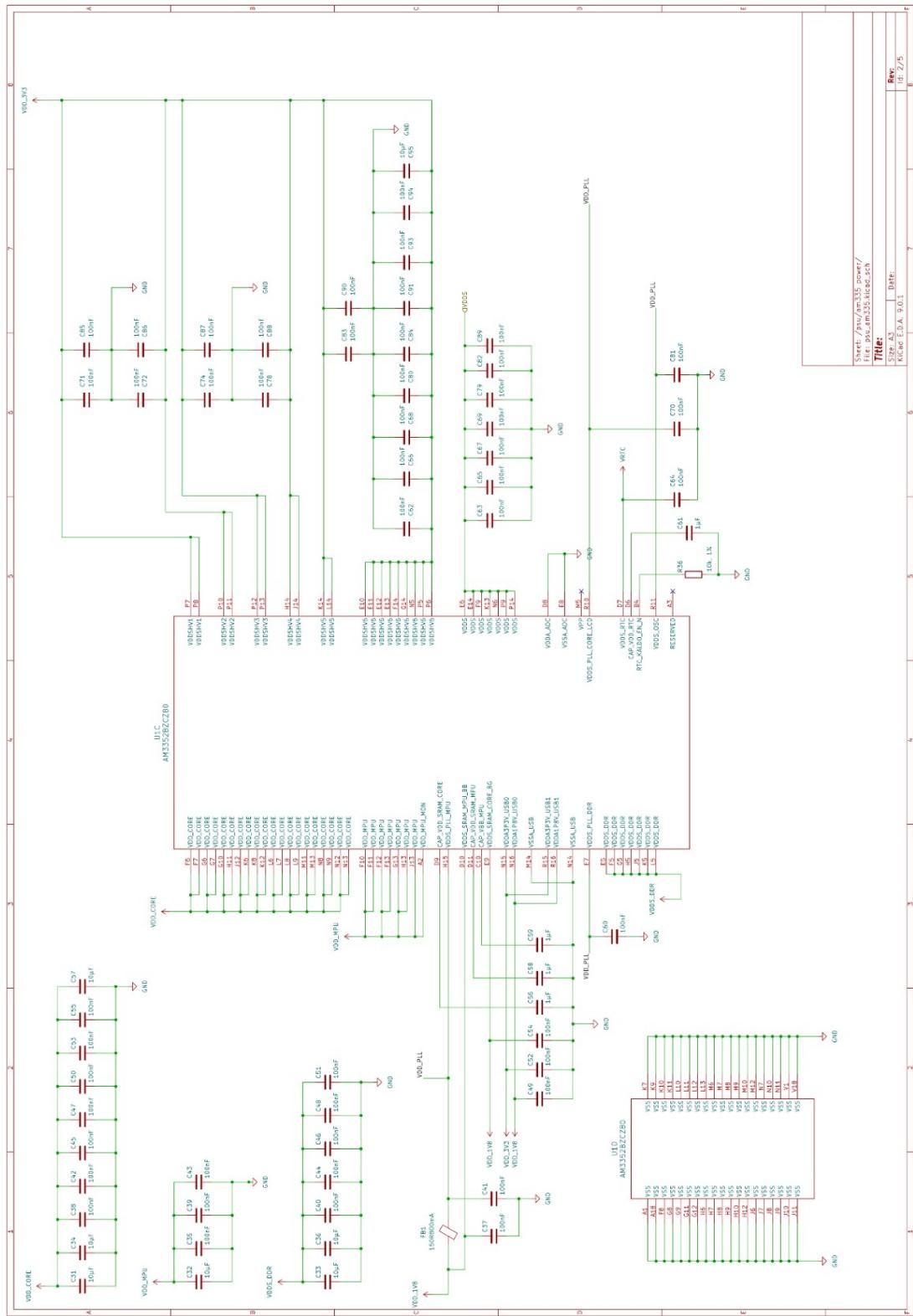


Abbildung 5. PSU AM335 Schematic

SD-Karte

Der AM335 bietet eine Vielzahl an Möglichkeiten für das Booten des Prozessors an, darunter sind unter anderem USB, UART, Ethernet, Flash oder über eine Micro-SD-Karte. In diesem Projekt wurde die Micro-SD-Karte für das Booten des Kernels verwendet, da dies die einfachste Variante zu realisieren ist. Da der Prozessor nicht wissen kann, über welche Schnittstelle er booten soll, muss ihm dies über eine Konfiguration von Widerständen als Eingänge mitgeteilt werden.

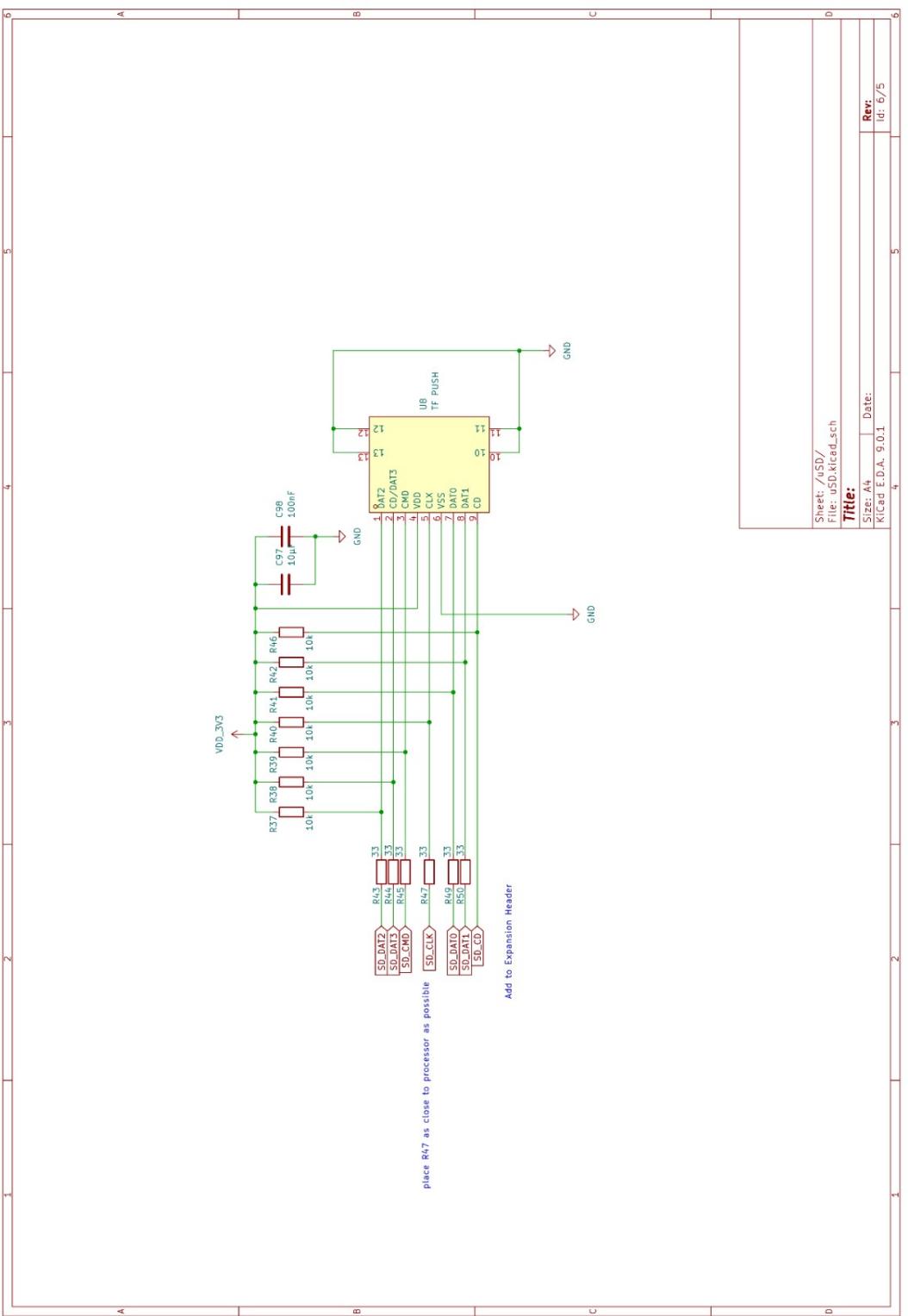


Abbildung 6. SD-Karte Schematic

Boot Pins

Damit der Prozessor die SD-Karte zum Booten verwendet, müssen die Boot Pins richtig gesetzt werden. Dabei können neben der Boot-Sequenz auch unter anderem die gewünschte Frequenz des Quarz ausgewählt werden. Die anderen Bits sind für die folgende Boot-Sequenz nicht relevant und werden demnach auf 0 gesetzt.

Es können insgesamt 31 verschiedene Boot Sequenzen ausgewählt werden. Wenn die erste Peripherie hierbei scheitert, den Prozessor zu booten, wird das Booten über die nächste Peripherie versucht. Ein Vorteil dieser Methode ist unter anderem, dass somit zum Beispiel über einen Tastendruck verschiedene der Prozessor über verschiedene Peripherien gebootet werden kann, wie es beim Beaglebone Black der Fall ist, wo durch einen Tastendruck der Prozessor über die SD-Karte gebootet werden kann, ansonsten wird er durch die Flash gebootet.

SYSBOOT[15:14] 1	SYSBOOT[13:12] 1	SYSBOOT[11:10] 1	SYSBOOT[9]	SYSBOOT[8]	SYSBOOT[7:6]	SYSBOOT[5]	SYSBOOT[4:0]	Boot Sequence			
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)	Don't care for ROM code	0 = CLKOUT1 disabled 1 = CLKOUT1 enabled	11100b	MMC1	MMC0	UART0	USB0			

Abbildung 7. Verwendete Boot Konfiguration TRM Seite 5035

Das Setzen der Boot Pins wird hierbei durch eine Reihe an Widerständen realisiert, welche je nach Pin entweder auf Masse oder 3.3V angeschlossen werden.

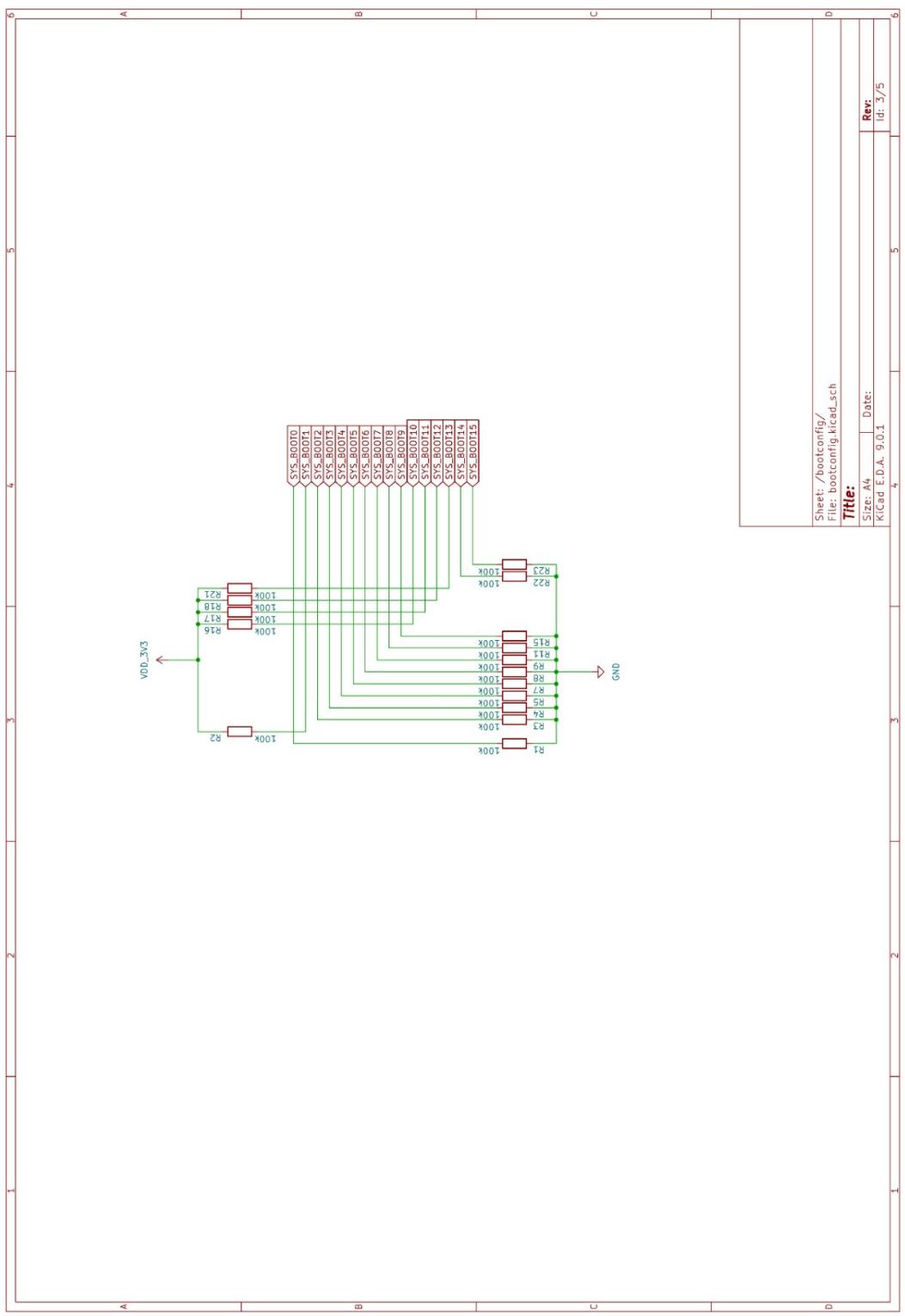


Abbildung 8. Boot Pins Schematic

Platine

Da die Pins für die Spannungsversorgung des AM335 alle auf sehr kleinem Raum liegen und zudem möglichst große Leitungen bzw. Polygone aufgrund der Hitzebildung benötigen, werden bei der Platine hier insgesamt 6 Layer verwendet. Dabei werden der zweite für Masse und der vierte Layer für die Power Signale verwendet.

In den folgenden Seiten ist die Platine abgebildet. Das erste Bild kombiniert hierbei alle Layer, wobei jedoch die Masse und Power ausgenommen sind, um die anderen Layer nicht zu verdecken. Die weiteren Abbildungen sind anschließend die verschiedenen Layer einzeln abgebildet, von Layer 1 bis Layer 6.

Wie man erkennen kann, werden eine Menge Polygone für den Prozessor benötigt, um die verschiedenen Spannungsversorgungen zu verbinden. Dabei werden auf dem Top Layer, auf welchem der Prozessor angebracht ist, die verschiedenen Pins mit den anderen Layern verbunden. Auf dem Bottom Layer sind außerdem die nötigen Stützkondensatoren platziert. Diese haben hierbei ein Package von 0402, da sehr viele von ihnen in einem sehr kleinen Raum nötig sind.

Die Platine bestellte ich anschließend bei dem Hersteller JLCPCB. Dabei entschloss ich mich unter anderem auch dafür, die Platine vorne und hinten bestücken zu lassen. Dies liegt daran, dass die Pins des Prozessors nicht von einem Menschen gelötet werden können und die Stützkondensatoren klein und auf engem Raum platziert sind, was das Löten sehr erschwert. Aufgrund dieser Entscheidung schoss der Preis der Platine jedoch in die Höhe, vor allem deshalb, weil auf beiden Seiten bestückt werden musste, weshalb ein anderes Verfahren verwendet werden musste.

Leider konnte die Platine nicht mehr getestet werden, da JLC bei der Bestückung der Platine einige Pins des PMIC zusammengelötet hat. Da dieser Fehler zuerst nicht bemerkt wurde, wurde der PMIC kurzgeschlossen und funktionierte nicht mehr. Da die Platine insgesamt 140€ gekostet hatte, beschloss ich, die Platine nicht mehr erneut zu bestellen, um nicht erneut so viel Geld auszugeben.

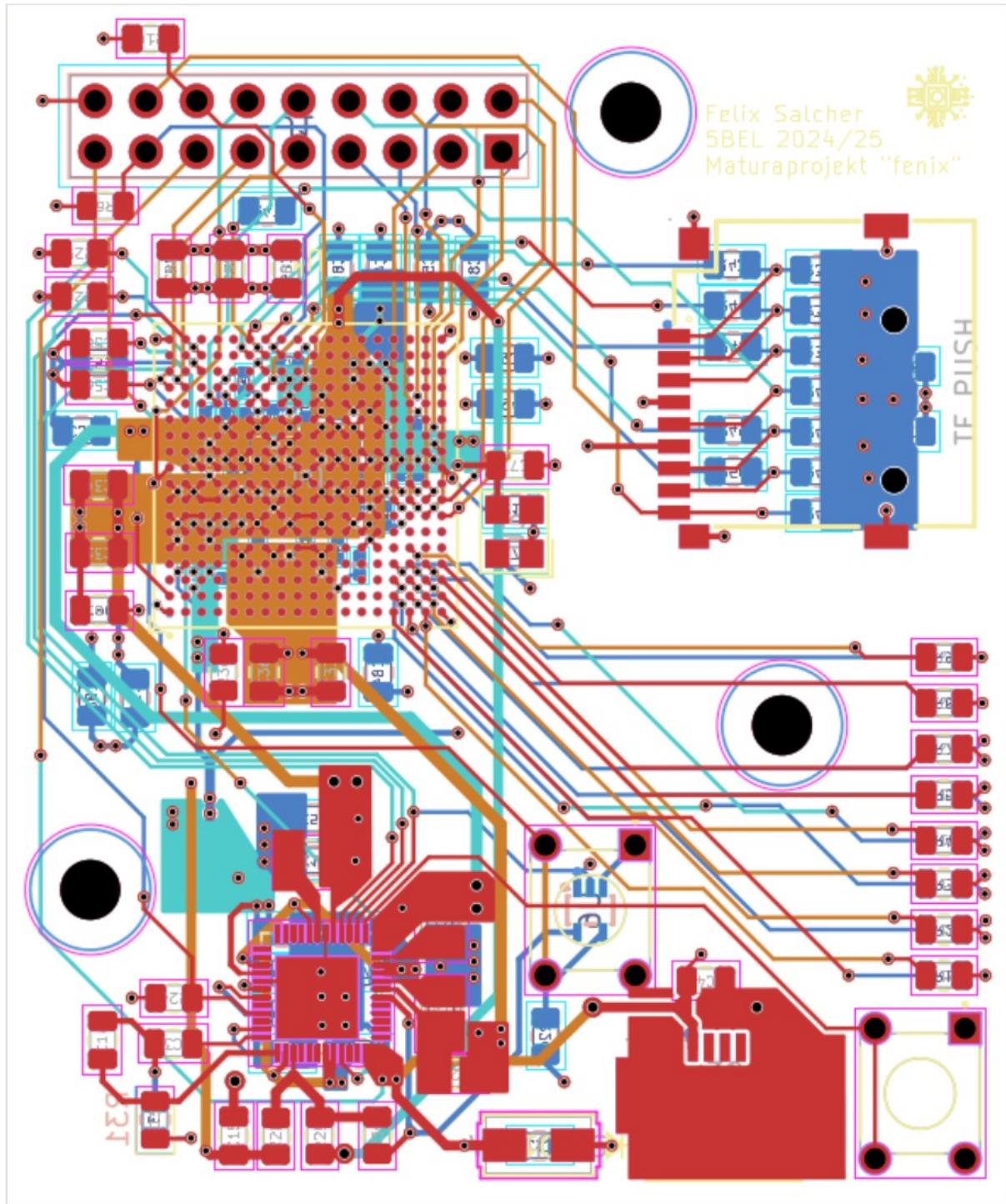


Abbildung 9. Gesamte Platine

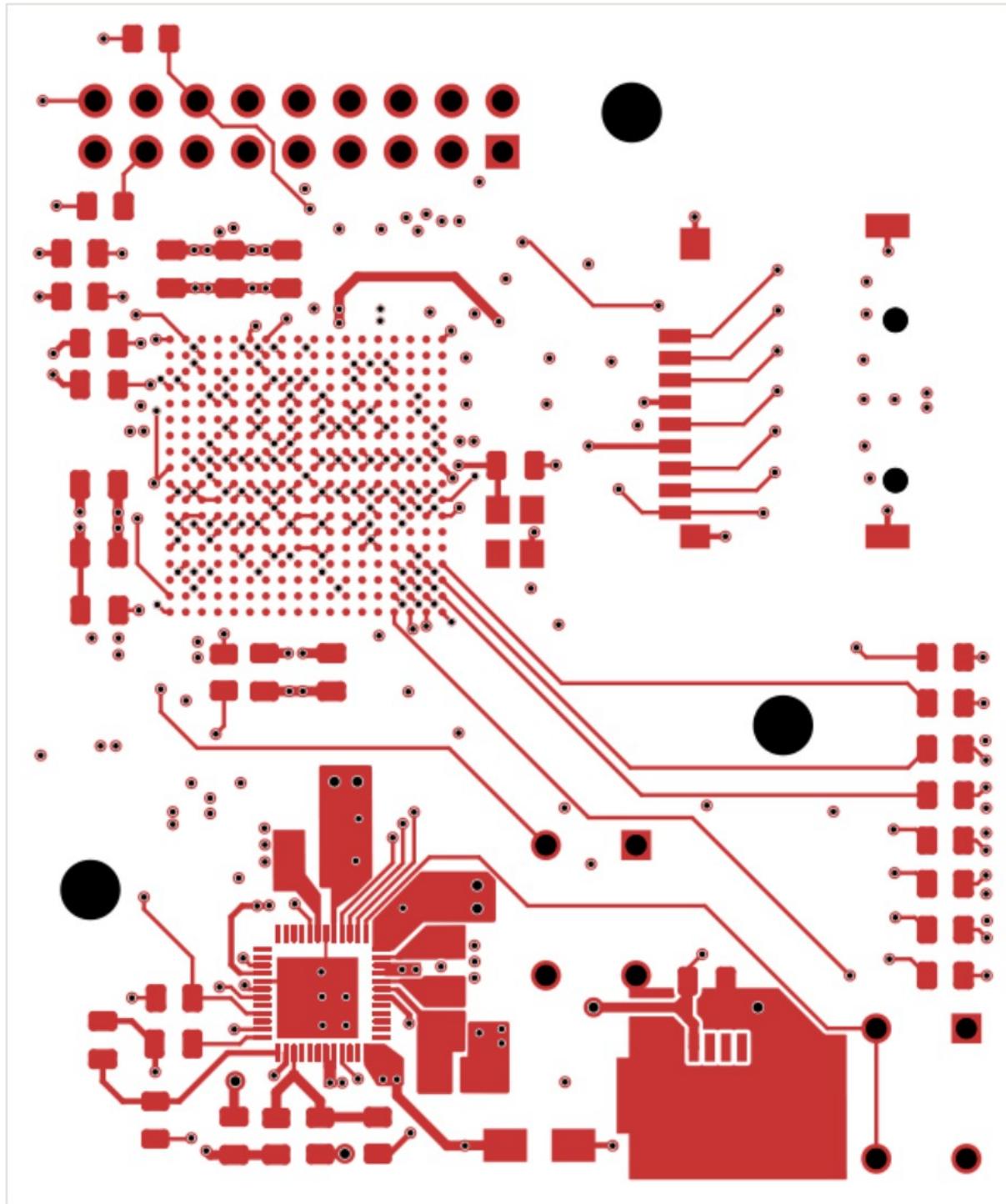


Abbildung 10. Platine Top Layer

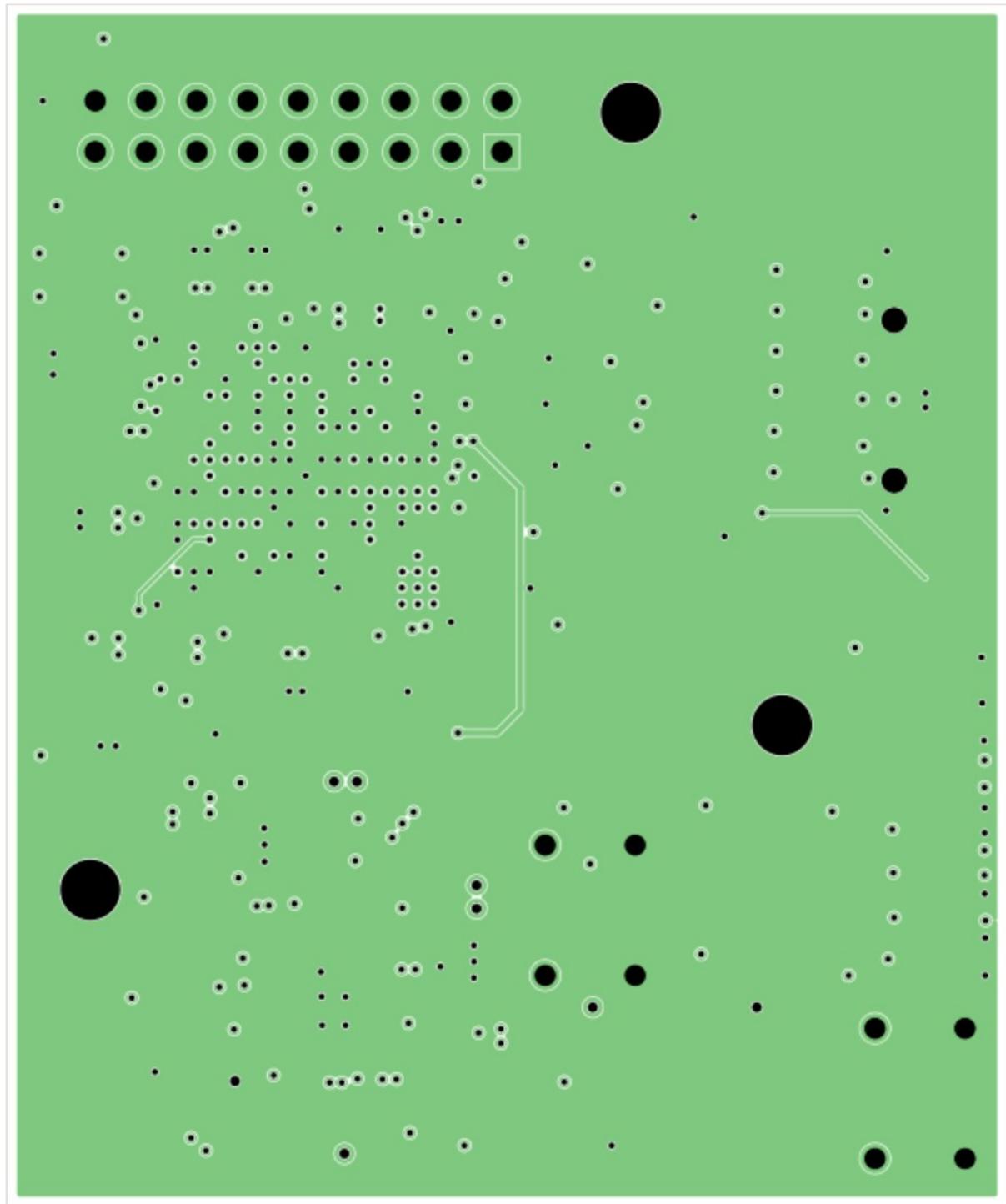


Abbildung 11. Platine GND Layer

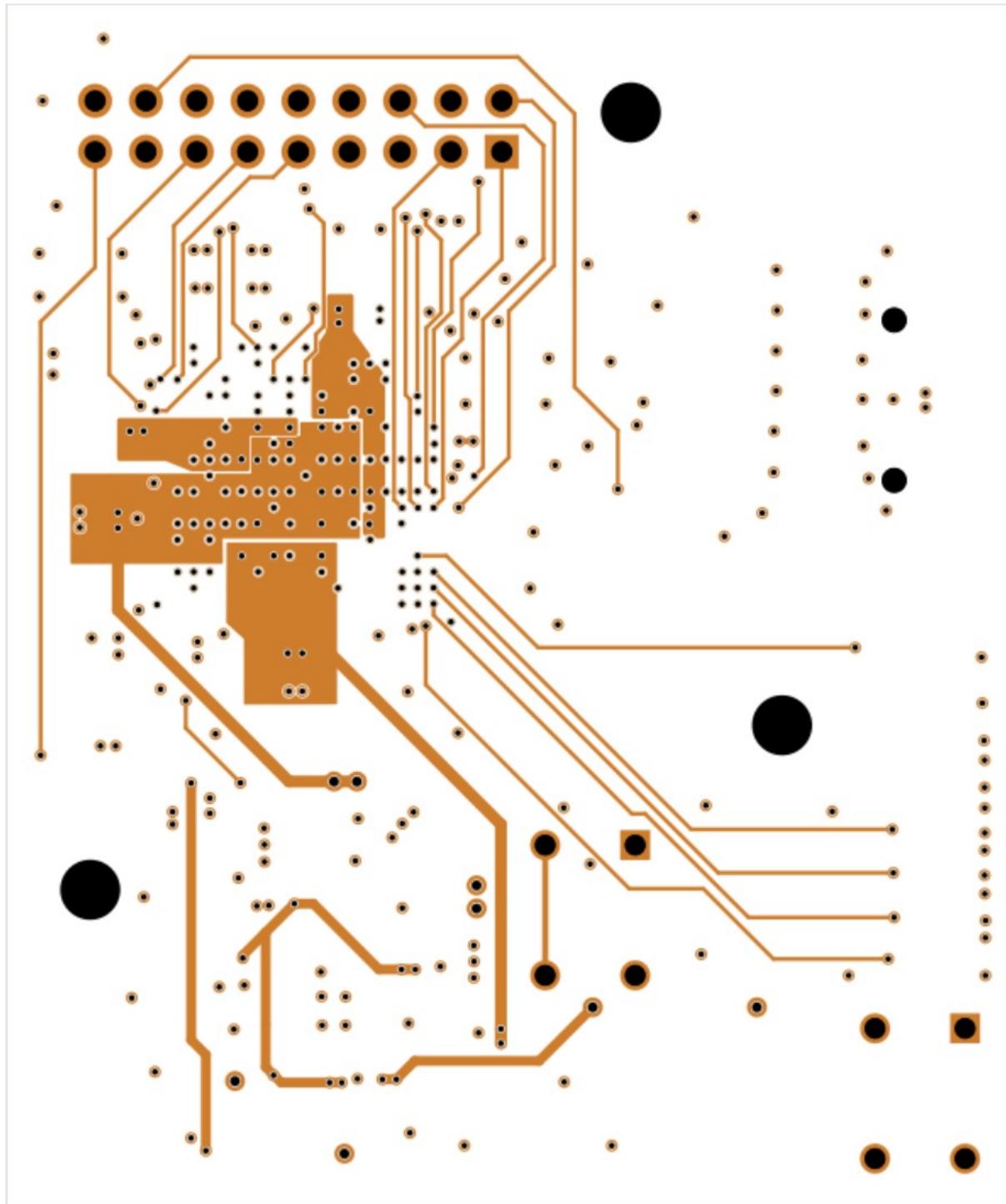


Abbildung 12. Platine Layer 3

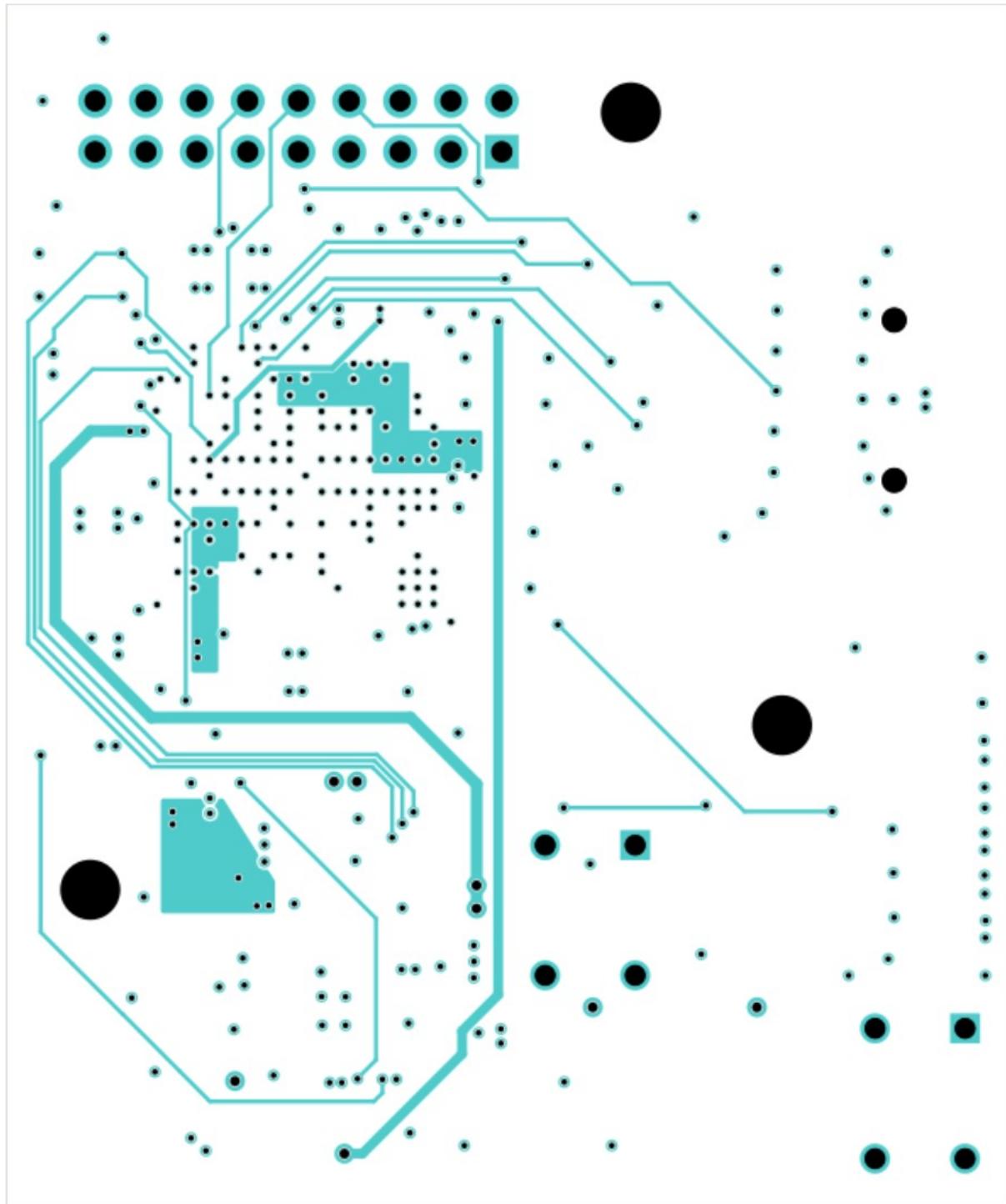


Abbildung 13. Platine Layer 4

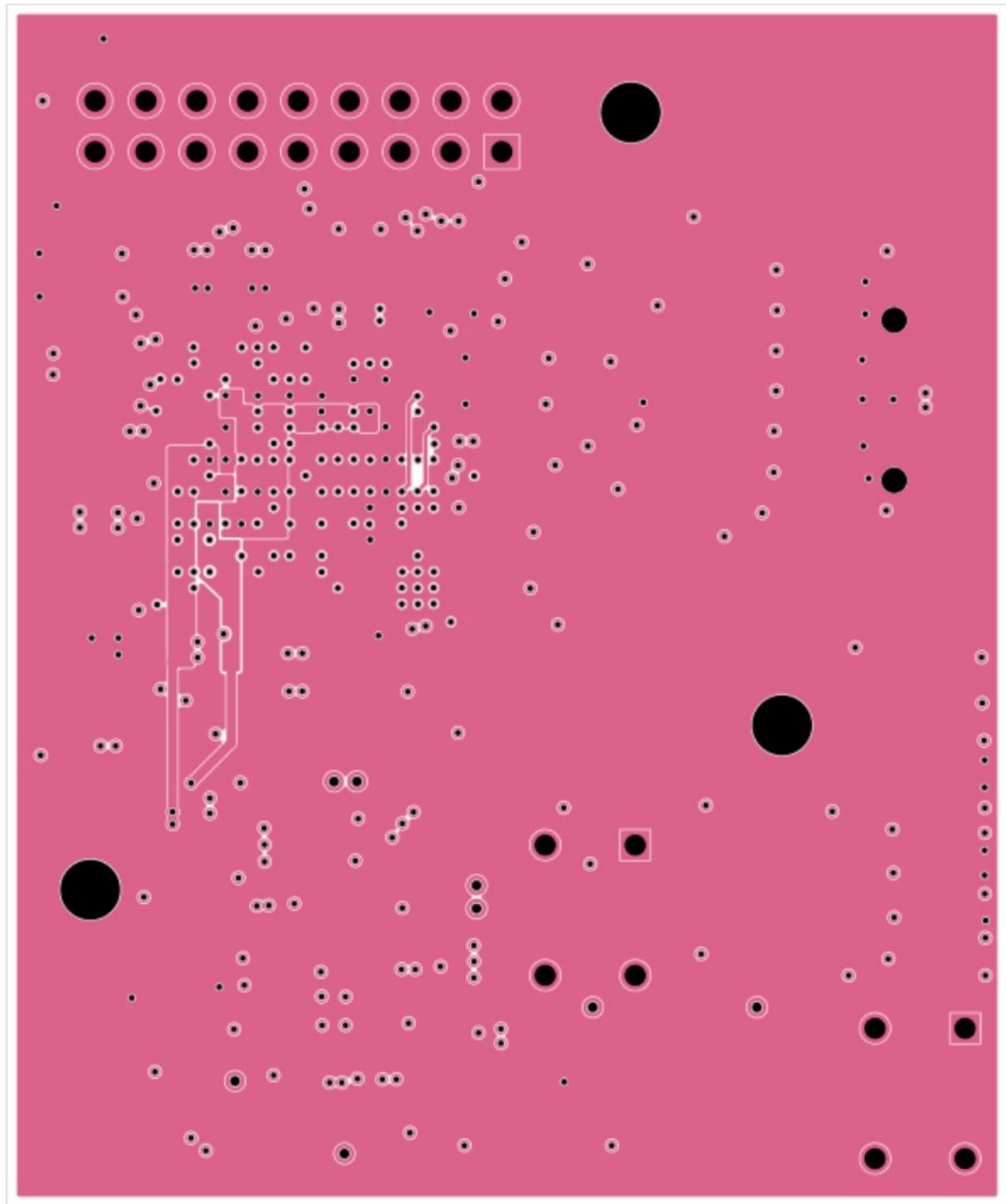


Abbildung 14. Platine POWER Layer

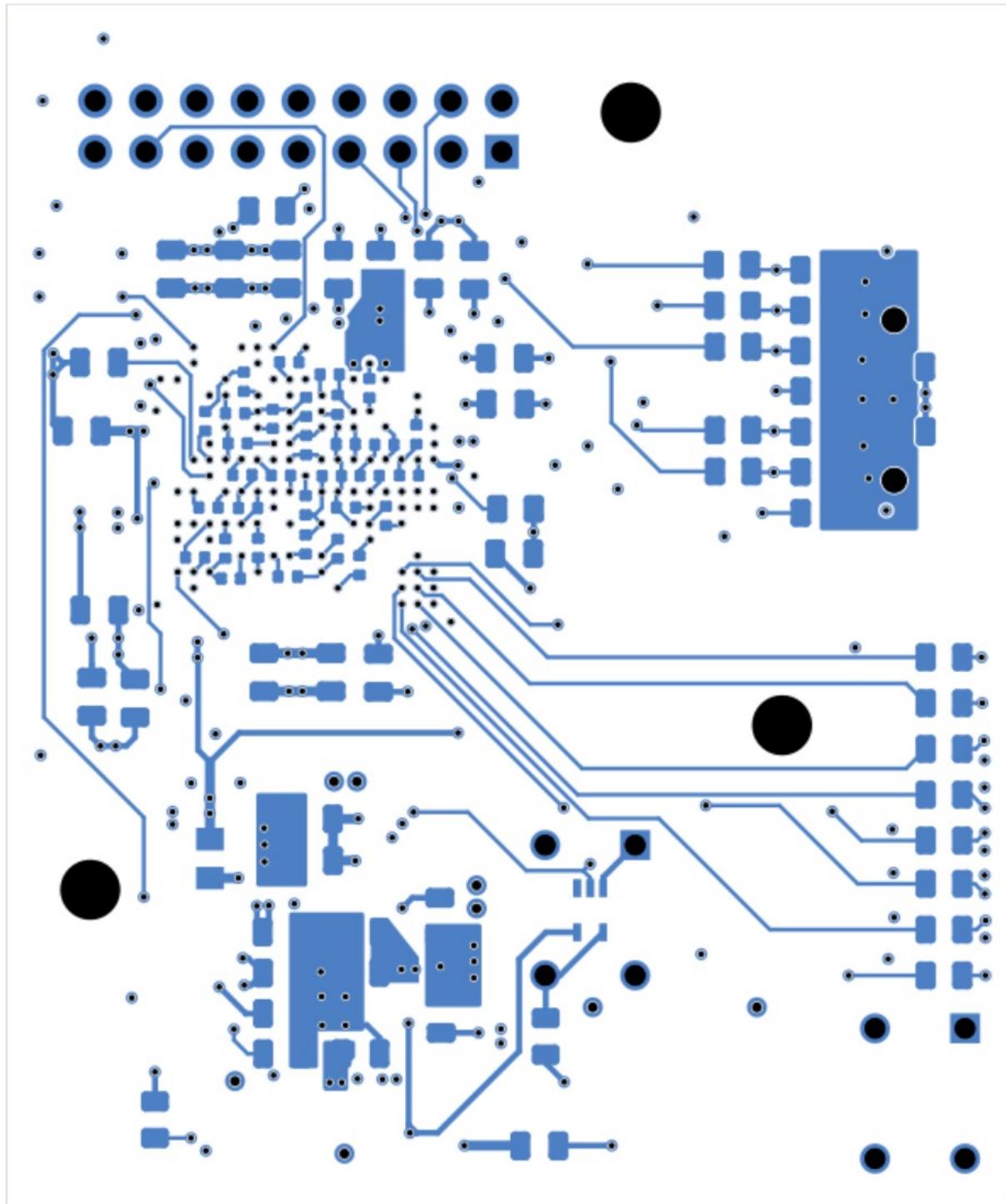


Abbildung 15. Platine Bottom Layer

Software

Github Repository

Die gesamte Software des Projektes ist in einem Online-Repository gespeichert. Es kann über folgenden link aufgerufen werden. <https://github.com/salfel/fenix>



Abbildung 16. QR Code Github Repository

Der Arm Cortex A8

In dem folgenden Abschnitt möchte ich dem Leser gerne den Arm Cortex Controller nahelegen, da dies für die weiteren Kapitel bezüglich der Software notwendig ist, um diese besser verstehen zu können. Der verwendete Controller ist hierbei der AM335 von TI, welcher auf einem 32-bit Arm Cortex A8 basiert. Dieser implementiert neben den üblichen Features des Arm Cortex A8 zudem noch einige Peripherien wie z.B. GPIO, I2C und viele weitere.

Der Arm Cortex A8 ist hierbei Teil der Armv7a Architektur. Er implementiert hierbei Version 7 der Arm-Architektur und gehört der “Applications” Familie an. Diese ist die schnellste Variante der Armv7 Familien, neben den “Microcontroller” (Armv7-M) und “Real-Time” (Armv7-R) Controllern. Features welche der Arm Cortex A8 implementiert sind unter anderem:

- **L1 und L2 Caches:**

Der Arm Cortex besitzt jeweils 32 KB von L1 Data und Instruction Caches. Der Data Cache hat hierbei die Aufgabe, die verschiedenen Adressübersetzungen der MMU zu cachen, damit diese die Adressen nicht bei jedem Zugriff übersetzen muss. Der Instruction Cache hat die Aufgabe, die verschiedenen Instructions zu cachen, damit diese nicht bei jeder Instruction neu decodiert werden müssen. Der L2 Cache hat die Aufgabe, einen weiteren Instruction und Data

Cache bereitzustellen, falls der L1 Cache verfehlt wird, weiteren Cache bereitzustellen, in welchem die Daten dann möglicherweise vorhanden sind.

- **Verschiedene CPU Modi:**

Der Prozessor enthält insgesamt 9 CPU Modi, welche jeweils verschiedene Aufgaben erfüllen. Mehr dazu jedoch in einem späteren Kapitel.

- **Memory Management Unit (MMU):**

Die MMU, welche Teil eines jeden Desktop/Laptop Prozessors ist, dient dazu, bestimmte Regionen von RAM nur für bestimmte Prozesse freizugeben. Mehr Dazu in einem späteren Abschnitt

Die CPU

CPU Modi

Der Arm Cortex A8 hat insgesamt 9 CPU Modi, welche jeweils verschiedene Aufgaben haben. Die meisten davon werden durch sog. Exceptions ausgelöst. In der untenstehenden Abbildung kann man alle CPU Modi des Prozessor sehen.

Mode	Encoding	Function	Security state	Privilege level
User (USR)	10000	Unprivileged mode in which most applications run	Both	PL0
FIQ	10001	Entered on an FIQ interrupt exception	Both	PL1
IRQ	10010	Entered on an IRQ interrupt exception	Both	PL1
Supervisor (SVC)	10011	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	PL1
Monitor (MON)	10110	Implemented with Security Extensions. See Chapter 21	Secure only	PL1
Abort (ABT)	10111	Entered on a memory access exception	Both	PL1
Hyp (HYP)	11010	Implemented with Virtualization Extensions. See Chapter 22	Non-secure	PL2
Undef (UND)	11011	Entered when an undefined instruction executed	Both	PL1
System (SYS)	11111	Privileged mode, sharing the register view with User mode	Both	PL1

Abbildung 17. ARMV7a CPU Modi

<https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Registers>

Wie man erkennen kann, werden der Monitor- und Hypervisor-Modus nur mit bestimmten Extensions des Prozessors implementiert und sind im AM335 nicht vorhanden.

- **User:** Der Benutzer Modus ist bei den meisten Betriebssystemen für die Ausführung von sog. User Code zuständig. Darin sind normalerweise die Applikationen enthalten, die ein Benutzer auf seinem Betriebssystem herunterlädt. Der User Mode ist der einzige CPU Modus welcher nicht privilegiert ist. D.h. dass er nicht den Prozessor-Modus wechseln kann, dies muss durch einen SVC durchgeführt werden. Zudem kann der Prozessor im User Mode nicht auf jede beliebige RAM-Adresse zugreifen, mehr dazu jedoch später.
- **System:** Der System Modus besitzt die gleichen Register wie der User Modus, ist jedoch privilegiert. Er wird normalerweise nur für den Übergang von Supervisor auf User Modus verwendet, um die Register des User Modus zu setzen, ohne in einen unprivilegierten Modus zu wechseln.
- **Supervisor:** Dies ist der Modus, in welchem der Prozessor die Ausführung startet. Er ist zudem der Modus, in dem der Kernel durchgeführt wird. Wenn der Prozessor sich in einem nicht privilegierten Modus befindet, kann über eine “Supervisor Call Instruction (SVC)” in den Supervisor Modus gewechselt werden.
- **IRQ:** Der Prozessor wechselt in den IRQ Modus, wenn ein Interrupt ausgeführt wird.

- **FIQ:** Der FIQ (Fast Interrupt Request) Modus oder entspricht dem IRQ Modus, der Unterschied ist jedoch, dass der FIQ Modus nur für jene Interrupts gedacht ist, die unbedingt mit der kürzesten Latenz durchgeführt werden müssen. Er kann hierbei auch normale Interrupts überschreiben.
- **Abort:** Der Prozessor wechselt in den Abort-Modus, wenn entweder eine Data Abort oder Prefetch Abort Exception ausgelöst wird. Ein Data-Abort wird ausgelöst, wenn der Prozessor eine Adresse fordert, auf welche er keine Berechtigungen hat oder nicht existiert. Ein Prefetch-Abort wird ausgelöst, wenn der Prozessor eine Instruction dekodieren möchte, diese jedoch nicht von der RAM gelesen werden kann, was aus denselben Gründen wie bei einem Data-Abort ausgelöst werden kann.
- **Undefined:** Der Undefined Modus wird ausgewählt, wenn der Prozessor eine *Undefined Exception* auslöst. Diese wird ausgelöst, wenn eine Instruction nicht dekodiert werden kann, bzw. wenn sie ungültig ist.

Register

In der untenstehenden Abbildung kann man die verschiedenen Register des Arm Cortex erkennen. R0-R12 sind hierbei sog. "General Purpose Register", d.h. sie können für alles verwendet werden.

Der "Arm Procedure Call Standard" (AAPCS) definiert hierbei R0-R3 und R12 als "Scratch Registers". D.h. Jede Funktion, welche durchgeführt wird, muss diese nicht auf den ursprünglichen Zustand zurückversetzen. Sie werden vielmehr dazu verwendet, um Parameter und Rückgabewerte an andere Funktionen zu geben. Die Register R4-R11 werden hierbei "Callee-Saved" Register genannt, dies bedeutet, dass wenn eine Funktion eines dieser Register verwendet wird, es den Wert des Registers wieder auf den ursprünglichen Wert zurücksetzt.

Der AAPCS garantiert hierbei, dass jede Funktion isoliert voneinander geschrieben werden kann und garantiert damit, dass keine Register unvorhergesehen beschädigt werden.

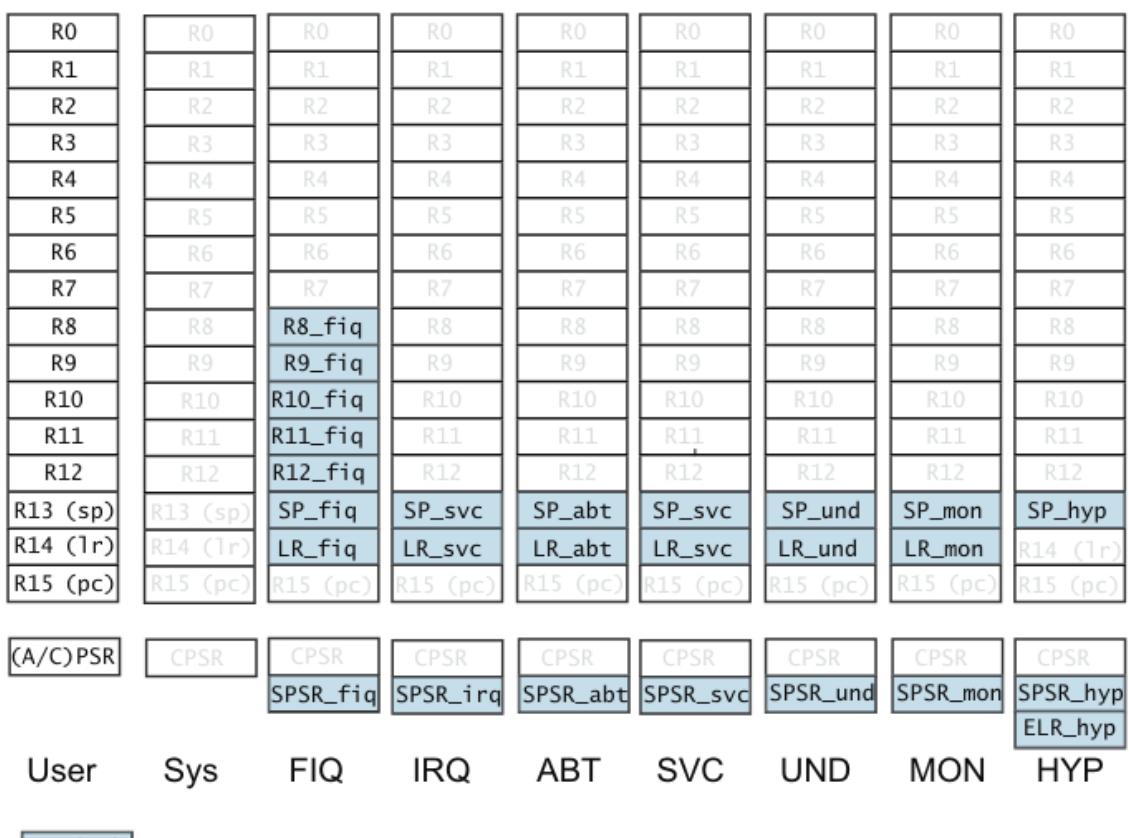


Abbildung 18. ARMV7a Register

<https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Registers/Registers>

Das **Stack Pointer Register (sp)** hat die Aufgabe, die Position des Stack Pointers zu speichern. Wie man in der untenstehenden Abbildung erkennen kann, ist dieser bei den meisten Modi “banked”, d.h. dass jeder Modus seinen eigenen Stack hat, um nicht den Stack anderer Modi, besonders bei Exceptions, zu beschädigen.

Das **Link Register (lr)** hat die Aufgabe, die Rückführadresse einer Funktion zu speichern, diese zeigt dann meist auf die nächste Instruction, nach der Function Call Instruktion. Das Link Register ist zudem wie der Stack Pointer banked, da bei einer Exception, die Adresse, welche die Adresse, welche die Exception ausgelöst hat und auf welche nach Abschluss der Exception gesprungen werden muss, im Link Register gespeichert ist, und diese nicht das Link Register des vorherigen Modus überschreiben sollte.

Das **Program Counter Register (pc)** speichert die Adresse der aktuell ausgeführten Instruktion.

Das **Current Program Status Register (CPSR)** speichert unter anderem den aktuellen Prozessor Modus, aber auch Flags, welche zum Branching verwendet werden, ob Interrupts aktiviert sind oder andere Dinge wie der aktuelle Execution State des Prozessors, worauf ich jedoch nicht weiter eingehen möchte.

Das **Saved Program Status Register (SPSR)** speichert bei Eintritt einer Exception das CPSR des Prozessors vor Eintritt der Exception. Dies ist vor allem deshalb wichtig, um den CPU Modus und die Branching Flags wieder auf den vorherigen Stand zurückzuversetzen.

Coprocessor 15

Die Armv7 Architektur beinhaltet zudem den sog. "Coprocessor 15", durch welchen man auf die "System Control Registers" zugreifen kann, also jene Register mit der man bestimmte Aspekte der CPU steuern kann, unter anderem kann man damit den Cache, die MMU, Exceptions und weitere Aspekte des Controllers kontrollieren

Vektoren

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisors	Monitor
0x0	0xFFFF0000	Not used	Reset	Reset	Not used
0x4	0xFFFF0004	UNDEFINED instruction	UNDEFINED instruction	UNDEFINED instruction from Hyp mode.	Not used
0x8	0xFFFF0008	Supervisor Call	Supervisor Call	Secure Monitor Call	Secure Monitor Call
0xC	0xFFFF000C	Prefetch Abort	Prefetch Abort	Prefetch Abort from Hyp mode.	Prefetch Abort
0x10	0xFFFF0010	Data Abort	Data Abort	Data Abort from Hyp mode,	Data Abort
0x14	0xFFFF0014	Not used	Not used	Hyp mode entry	Not used
0x18	0xFFFF0018	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

Abbildung 19. ARMV7a Exception Vektor

Die Vektoren des Arm Cortex A8 sind standardmäßig an der Adresse `0x00000000` können aber auch konfiguriert werden, dass sie als sog. hohe Vektoren, wobei sie dann bei `0xFFFF0000` liegen. Dies ist jedoch für den AM335 nicht praktikabel, da er an diesen Adressen keine RAM hat, die er ansprechen kann. Man kann jedoch durch den Coprocessor die Adresse der Vektoren auf ein beliebige Adresse umschreiben

Die MMU

Die Memory Management Unit oder kurz MMU ist eine Komponente eines jeden modernen Chips, welcher ein “General Purpose” Betriebssystem wie Linux, Mac oder FreeBSD verwendet. Die Hauptaufgabe der MMU ist, bestimmte Regionen von RAM vor unprivilegiertem Zugriff zu schützen, zum Beispiel sollte keine Benutzer Applikation in der Lage sein, den Speicher des Kernels oder anderen Prozessen auszulesen. Zudem ist die MMU in der Lage, bestimmte Regionen der RAM “umzumappen”. D.h. Wenn der Prozessor eine bestimmte Adresse der RAM fordert, z.B. `0x402f0400` übersetzt die MMU diese Adresse und spricht je nach Konfiguration eine andere physische RAM-Adresse an, z.B. `0x00000000`. Dieses Konzept wird “Virtual Memory” genannt und ist der Grundbaustein vieler Betriebssysteme. Dies ist unter anderem sehr praktisch, wenn das Memory Layout eines Prozessors nicht sehr effizient ist für eine bestimmte Anwendung, mit dem “Ummappen” der RAM kann dieses Problem überwunden werden.

Die Übersetzungen der MMU werden direkt auf der Hardware durchgeführt, was die Operationen sehr schnell macht. Meist werden die Übersetzungen zudem noch im L1 oder L2 Cache gespeichert, was zu noch schnelleren Übersetzungen führt.

Auf kleineren Mikrocontrollern wie jenen der Armv7-M Architektur wird nur eine sog. MPU (Memory Protection Unit) verwendet, welche Berechtigungen für verschiedene Regionen setzen kann, kann jedoch nicht Virtual Memory implementieren.

Page Tables

Das gesamte Memory wird bei einer MMU gewöhnlich in sog. Pages unterteilt, hierbei hat jede “Page” eigene Berechtigungen, so kann jedem Prozess im Kernel eine Page zugewiesen werden, auf welche nur dieser Prozess Zugriff hat. Beim Übersetzen werden hierbei nur die oberen 12 bzw. 20 Bits verwendet, um die Adresse zu übersetzen. Die LSBs werden hierbei nicht verändert.

Level 1

Beim Arm Cortex wird das Memory gewöhnlich in 4096 verschiedene Bereiche unterteilt , jede mit einer Größe von 1MB.

Um die Übersetzung durchzuführen, nimmt sie hierbei die 12 MSBs und verwendet diese als Offset für den Page Table, um die richtige Page Table Entry zu finden.

Der Prozessor stellt unter anderem die Möglichkeit bereit, zwei verschiedenen L1 Page Tables zu verwenden, einen für höhere Adressräume und den anderen für die niedrigeren. Dies wird meist deshalb verwendet, um verschiedene Memory Layouts für den Kernel und die User Prozesse zu haben. Bei meinem Kernel wurde jedoch nur ein L1 Page Table verwendet.

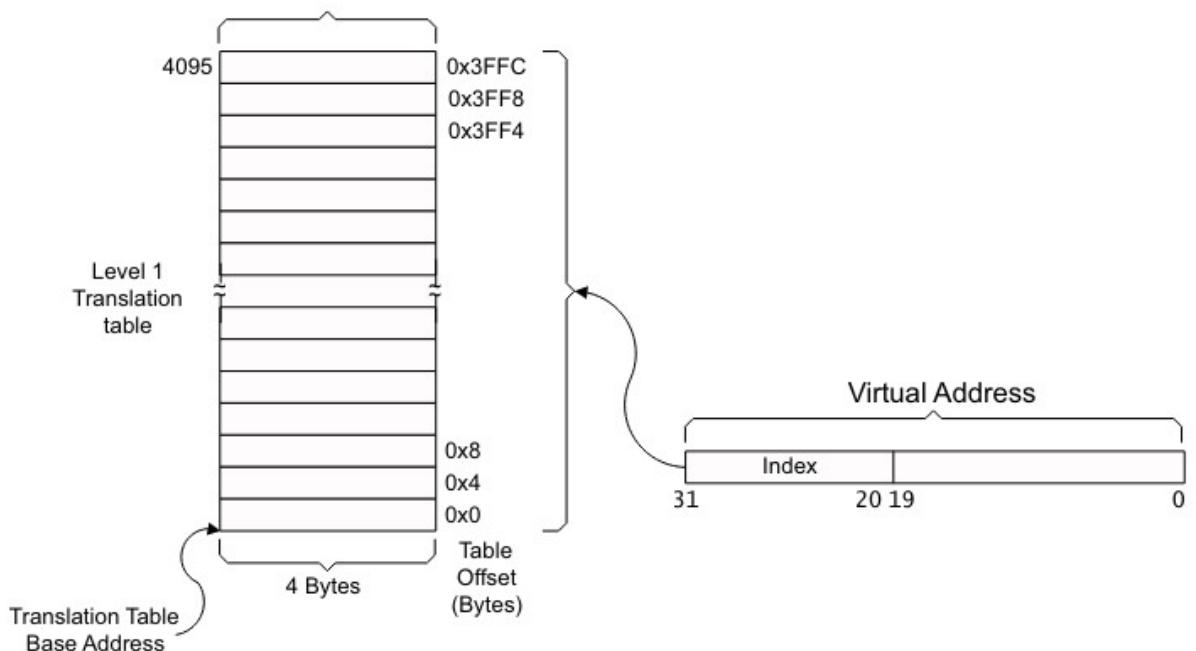


Abbildung 20. ARMV7a L1 Translation Table

<https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/First-level-address-transl ation>

Es wird hierbei zwischen vier verschiedenen Page Table Entries unterschieden.

- **Fault:** Löst eine Data Abort Exception aus
- **Pointer:** Diese ist ein Pointer zu einem weiteren Translation Table, mehr dazu später
- **Section:** Übersetzt die oberen 12 Bits in weitere 12 Bits.
- **Supersection:** Supersektionen übersetzen nicht wie normale Sektions Regionen von 1 MB, sondern Regionen von 16 MB, dies wird jedoch seltener verwendet.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Fault	Ignored																												0	0				
Pointer to 2 nd level page table	Level 2 Descriptor Base Address																												SBZ	0	1			
Section	Section Base Address																												X	N	C	B	1	0
Supersection	Supersection Base Address																												X	N	C	B	1	0

Abbildung 21. ARMV7a L1 Page Table Entry

<https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/First-level-address-transl-ation>

Level 2

Wenn ein Page Table ein Pointer zu einem Level 2 Page Table ist, wird der Pointer im Page Table eingesetzt. Der Level 2 Page Table muss hierbei auf 1 KB “aligned” sein, d.h. dass die 10 LSBs alle null sein müssen und umfasst 256 Page Table Entries. Unter anderem kann man den Page Table in Sektionen von 4 KB oder 64 KB unterteilen.

Wie bei Level 1 Page Table Entries unterscheidet man zwischen verschiedenen Typen von Page Table Entries:

- **Fault:** Löst eine Data Abort Exception aus
- **Large page:** Übersetzt nur die oberen 16 Bits, dadurch haben “Large Pages” eine Größe von 64 KB. Diese müssen jedoch 16x wiederholt werden.
- **Small page:** Übersetzt die 20 MSBs, was zu einer Page Größe von 4 KB führt.

	31		15	14	12	11	10	9	8	6	5	4	3	2	1	0				
Fault	Ignored														0	0				
Large page	Large Page Base Address														SBZ	AP	C	B	0	1
Small page	Small Page Base Address														X	N	C	B	1	X

Abbildung 22. ARMV7a L2 Page Table Entry

<https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/Level-2-translation-tables>

In den obenstehenden Abbildungen kann man eine Reihe von anderen Bits und Abschnitten der verschiedenen Page Table Entries sehen, diese sind vor allem Permission und Memory Ordering Attribute. Ich möchte auf diese hier jedoch nicht näher eingehen.

Translation Lookaside Buffer

Der Translation Lookaside ist ein Cache der zuletzt ausgeführten Übersetzungen. Durch den TLB muss somit nicht bei jeder Anfrage von Memory die MMU einen Translation Table Walk machen, sondern kann die erforderlichen Werte einfach aus dem Cache nehmen. Da sich die meisten der Variablen zudem meist in derselben Region von RAM befinden und diese meist gecacht ist, kann man davon ausgehen, dass bei jeder Anfrage von Memory der Cache getroffen wird, was die Schnelligkeit von RAM-Zugriffen deutlich verbessert.

Dabei werden die Virtuelle und Physische Adresse sowie die Memory Attribute dort gespeichert. Zudem wird dort auch die ASID gespeichert, unter welcher man die aktuelle Prozess ID verstehen kann. Dadurch können mehrere Pagetranslations mit derselben virtuellen Adresse gespeichert werden, denn der TLB gibt in diesem Fall nur die Pagetranslation, welche dieselbe ASID besitzt, welche zu diesem Zeitpunkt im *Coprocessor 15* gespeichert ist. Dadurch muss bei jedem Context Switch nicht jedes Mal der TLB invalidiert werden.

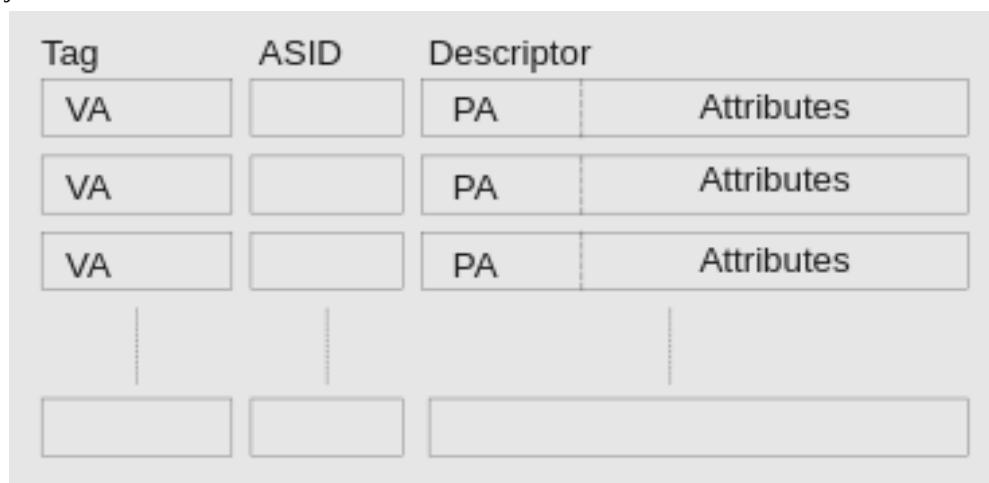


Abbildung 23. ARMV7a Translation Lookaside Buffer

<https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/The-Translation-Lookaside-Buffer>

Booten

Der AM335 wird nicht wie wir bei anderen Mikrocontrollern wie dem Arduino Nano oder dem ESP mit USB gebootet, sondern mit einer MicroSD Karte, welche mit dem Kernel beschrieben werden muss.

Der AM335 benötigt hierbei einen “Table of Contents” und einen “GP Header” vor dem Code. Der “Table of Contents” hat hierbei die Aufgabe, generelle Informationen über das Image bereitzustellen, während der Header die Größe des Images und den Standort im Memory festlegt.

Der folgende Command in Linux wurde verwendet um das Image auf die SD-Karte zu flashen:

```
○ ○ ○  
1 sudo dd if=./out/rom.img of=/dev/sda oflag=direct bs=4M status=progress
```

Komplilation

Wenn man ein Binary für einen Controller komplizieren möchte, der jedoch auf keinem Betriebssystem wie Linux oder Mac läuft, muss ein Linker zu Hilfe genommen werden. Denn bei einer Komplilation für z.B. Linux ist der Linker direkt in den Compiler integriert, da der Compiler bereits weiß, wo er den Code platzieren soll. Wenn man aber für sog. “Bare Metal” programmiert, weiß der Compiler nicht, wo er den Code platzieren soll. Deshalb muss ein Linker Script bei der Compilation hinzugefügt werden, damit der Compiler weiß, wo er den Code platzieren soll.

Notiz: Es wird hier von Code platzieren gesprochen, damit ist jedoch nur gemeint, welche Adressen der Compiler die verschiedenen Instruktionen kennzeichnet, um Branching an die richtige Adresse zu ermöglichen.

Linker Script

In dem folgenden Abschnitt möchte ich gerne das Linker Script etwas näher erklären.

In der **MEMORY** Sektion wird das Memory Layout des Prozessors beschrieben. **RWX** bedeutet, dass diese Region beschreibbar, lesbar und ausführbar ist, während **ORIGIN** die Startadresse im Memory beschreibt. **Entry** beschreibt hier das Symbol, an welchem der Controller startet.

In dem **SECTIONS** Abschnitt wird später beschrieben, wo die verschiedenen Symbole platziert werden. Hier eine kurze kurze Beschreibung der Symbole:

- **Text:** Hier wird der gesamte Code platziert. Hier wurde das `.text._start` symbol vor all den übrigen symbolen platziert um zu garantieren, dass das `_start` symbol ganz am anfang ist
- **BSS:** Hier werden alle globalen Variablen gespeichert, welche noch nicht initialisiert wurden
- **Data:** Hier werden alle globalen Variablen gespeichert, welche mit einem Wert initialisiert werden.



Linker.ld

```
1 MEMORY {  
2     sram (rwx) : ORIGIN = 0x402F0400, LENGTH = 0xFFFF  
3 }  
4  
5 ENTRY(_start)  
6 SECTIONS {  
7     .text : {  
8         *(.text._start)  
9         *(.text*)  
10    } > sram  
11  
12    .bss : {  
13        *(.bss*)  
14    } > sram  
15  
16    .data : {  
17        *(.data*)  
18    } > sram  
19  
20    ...  
21  
22    _end = .;  
23 }
```

Cargo

Cargo ist der Package Manager in Rust. Dieser muss für die Kompilation für den AM335 angepasst werden. Untenstehend kann man die Konfigurationsdatei von Cargo für die Kompilation sehen.

Zu Beginn wird das “target” gesetzt, dies beschreibt die Architektur und die CPU des Controllers, damit Cargo weiß, welche Features eines Controllers er verwenden kann. Z.B. kann er so Floats in den Code einbinden, wenn er weiß, dass dieser Controller eine FPU besitzt.

Anschließend wird der Linker gesetzt, dieser ist Teil der *arm-none-eabi* Toolchain, in welcher weitere Tools wie Assembler, Compiler und weiteres vorhanden sind. In den *rustflags* werden anschließend die Linker Argumente gesetzt. Diese beinhalten unter anderem die Position des Linker Scripts, aber auch die verschiedene “object files”, zu welchen einige Assembly Dateien konvertiert worden sind.

Notiz: Object Files sind Assembly Dateien, welche bereits kompiliert wurden, welche jedoch noch nicht wissen, in welcher Region von Memory sie platziert werden, sie müssen noch in den Linker eingebaut werden

```
○ ○ ○ .cargo/config.toml

1 [build]
2 target = "armv7a-none-eabi"
3
4 [target.armv7a-none-eabi]
5 linker = "arm-none-eabi-ld"
6 rustflags = [
7     "-C", "link-arg=-Tkernel/boot/linker.ld",
8     "-C", "link-arg=kernel/out/setup.o",
9     "-C", "link-arg=kernel/out/exceptions.o",
10    "-C", "link-arg=kernel/out/interrupts.o",
11    "-C", "link-arg=kernel/out/software_interrupts.o",
12    "-C", "link-arg=kernel/out/kernel.o",
13 ]
```

Build Script

Das Ganze wird anschließend in ein GNU Makefile eingebaut, um den Build Prozess zu automatisieren. Hier werden zuerst die Assembly Files kompiliert, dann wird das gesamte Projekt mit Cargo kompiliert und anschließend mit dem TOC und dem Header kombiniert, welches man anschließend auf die SD-Karte flashen kann.



Makefile

```
1 build:
2 arm-none-eabi-gcc -mcpu=cortex-a8 -c src/asm/setup.S -o out/setup.o
3 arm-none-eabi-gcc -mcpu=cortex-a8 -c src/asm/interrupts.S -o out/interrupts.o
4 arm-none-eabi-gcc -mcpu=cortex-a8 -c src/asm/exceptions.S -o out/exceptions.o
5 arm-none-eabi-gcc -mcpu=cortex-a8 -c src/asm/software_interrupts.S -o out/software_interrupts.o
6 arm-none-eabi-gcc -mcpu=cortex-a8 -c src/asm/kernel.S -o out/kernel.o
7 cargo build --release
8 cp ../target/armv7a-none-eabi/release/kernel out/kernel.elf
9 arm-none-eabi-objdump -d out/kernel.elf > out/kernel.dump
10 arm-none-eabi-nm out/kernel.elf > out/kernel.map
11 arm-none-eabi-objcopy out/kernel.elf -O binary out/boot.bin
12 cat boot/toc.bin boot/header.bin out/boot.bin > out/rom.img
13
```

Startup

Bevor der Kernel die Peripherie oder anderes initialisiert, müssen vor allem der Sack, die Caches und die Exceptions initialisiert werden. Hierbei beginnt der Kernel mit der *setup_modes* Routine den Stack für die jeweiligen Modi. Anschließend werden die Caches initialisiert, gefolgt mit dem Initialisieren der Vektoren, in welchen die Handler der verschiedenen Exceptions vorhanden sind.

```
○ ○ ○                                     src/main.rs

1 pub fn _start() {
2     unsafe {
3         setup_modes();
4         setup_caches();
5         setup_exceptions();
6     }
7
8     ...
}
```

Stack Setup

Zu Beginn wird der Stack Pointer des IRQ Modus gesetzt. Mittels der *msr* Instruction kann ein Register in das *CPSR* Register geschrieben werden. Das Suffix des *CPSR* bedeutet hierbei, dass nur die Interrupt Flags und der Modus überschrieben werden können. Der Stack Pointer wird dann mit der Adresse es *irq_stack_end* Symbols überschrieben, welches im Linker Script definiert ist. Dasselbe wird anschließend mit dem Supervisor Modus gemacht. Zu Ende werden dann Interrupts aktiviert, der Prozessor bleibt hierbei im Supervisor Modus. Abschließend springt der Prozessor zum Inhalt des LR-Registers.

Dem aufmerksamen Beobachter wird hier auffallen, dass der Stack Pointer des User Modus nicht gesetzt wird. Dies wird immer vor einem Task Switch durchgeführt, wodurch es hier überflüssig ist.

○ ○ ○

src/asm/setup.S

```
1 setup_modes:
2     @ IRQ mode
3     mov r0, #0xD2
4     msr cpsr_c, r0
5     ldr sp, =irq_stack_end
6
7     @ Supervisor mode
8     mov r0, #0xD3
9     msr cpsr_c, r0
10    ldr sp, =stack_end
11
12    @ Enable IRQ
13    mov r0, #0x53
14    msr cpsr_c, r0
15
16    bx lr
17
```

Cache Setup

Um die verschiedenen Caches zu aktivieren, muss der L1 Cache zu Beginn deaktiviert werden. Anschließend werden die L1 Caches sowie die Data Caches invalidiert, um sie komplett zu entleeren. Anschließend werden sowohl Branch Prediction als auch D-Side Prefetching aktiviert.

Armv7 Chips enthalten sogenannte Branch Prediction Units (BPU), die parallel zur CPU laufen. Da die CPU nicht wissen kann, welche Möglichkeit von beiden sie bei einem *If/else* z.B. nimmt, kann sie nicht Instructions und Daten cachen. Die BPU kann hierbei erraten, welche von beiden Möglichkeiten am wahrscheinlichsten ist und die Instructions und Daten prefetchen.

Beim D-Side Prefetching greift der CPU eine Hardware-Komponente unter die Arme, wenn gewisse Muster beim Zugriff von Daten erkennbar sind. Wenn z.B. die CPU ein Array iteriert, kann diese Hardware-Komponente dies erkennen und die Daten bereits in den Cache laden, um einen RAM-Roundtrip zu verhindern. Da der Cache deutlich näher an der CPU ist, kann dies zu einer großen Performance Verbesserung führen.

Wie man erkennen kann, wird zu Beginn der Funktion das Link Register auf den Stack gepusht und anschließend in das PC-Register gepoppt. Dies ist nötig, da bei einer *bl* Instruction (Branch with Link) das LR überschrieben wird. Dementsprechend muss dieses auf dem Stack gespeichert werden und wird abschließend an das PC-Register transferiert.

Die Implementation der jeweiligen Funktionen hier bleibt dem Leser erspart, da dies nicht sehr wichtig ist.



src/asm/setup.S

```
1 setup_caches:  
2     push {lr}  
3  
4     bl disable_l1_caches  
5     bl invalidate_l1_caches  
6     bl invalidate_data_caches  
7     bl branch_prediction_enable  
8     bl enable_d_side_prefetch  
9  
10    pop {pc}
```

Exceptions Setup

Das Setup der Vektoren ist sehr wichtig, da sie durch die Handler von z.B. Interrupts und Software Interrupts definiert werden, ohne die das Kernel nicht funktionieren könnte. Die Vektoren müssen hierbei um 5 Bits “aligned” sein. Um die Vektoren zu initialisieren, muss deren Adresse in das *r0* Register geladen werden und anschließend in das “Vector Base Address Register” (VBAR) geschrieben werden, welches sich im Coprocessor 15 befindet. Abschließend wird eine sog. “Data Synchronization Barrier” ausgeführt, welche garantiert, dass alle Daten Zugriffe abgeschlossen sind, bevor die Ausführung des Programms wieder startet.

○ ○ ○

src/asm/exceptions.S

```
1 .align 5
2 vectors:
3   b _start
4   b undefined_handler
5   b handle_swi
6   b fetch_abort_handler
7   b data_abort_handler
8   b .
9   b irq_handler
10  b .
11
12 setup_exceptions:
13  ldr r0, =vectors
14  mcr p15, 0, r0, c12, c0, 0
15  dsb
16
17  bx lr
```

AM335 Hardware

In dem folgenden Kapitel möchte ich gerne auf die Hardware des AM335 zu sprechen kommen. Darunter sind hier die verschiedenen Implementationen der AM335-spezifischen Module zu verstehen. Darunter sind unter anderem auch Module wie Timer und Interrupts, da deren Implementation jedoch spezifisch für den AM335 ist, sind sie im selben Kapitel wie die typische Peripherie wie GPIO oder I2C gegliedert.

Interrupts

Der AM335 implementiert nicht wie viele weitere Prozessoren den *ARM General Interrupt Controller (GIC)*. Stattdessen wurde hier ein eigener Interrupt Controller für den AM335 entwickelt. Dieser hat insgesamt 128 verschiedene Interrupts, welche von den verschiedenen Modulen des AM335 ausgelöst werden können. Unter anderem hat jeder Interrupt eine Priorität, welche einen Wert von 0-127 haben kann.

Aktivierung

Um einen Interrupt zu aktivieren, muss zu Beginn der Modus und die Priorität in das *INTC_ILR* geschrieben werden. Anschließend wird das Mask-Bit des entsprechenden Interrupts geclear, wodurch der Interrupt aktiviert wird.

```
○ ○ ○                                     src/interrupts.rs

1 pub fn enable_interrupt(interrupt: Interrupt, mode: Mode, priority: u8) {
2     let interrupt_number = interrupt as u32;
3
4     let addr = INTC + INTC_ILR + (4 * interrupt_number);
5     let enable_fiq = match mode {
6         Mode::IRQ => 0,
7         Mode::FIQ => 1,
8     };
9     let bank = match InterruptBank::new(interrupt_number) {
10        Some(bank) => bank,
11        None => return,
12    };
13
14     write_addr(addr, enable_fiq | (priority << 2) as u32);
15     set_bit(INTC + bank.get_mir() + MIR_CLR_OFFSET, interrupt_number % 32);
16 }
```

Handler

Der Interrupt Handler wird, wie bereits [hier](#) beschrieben, über die Vektoren gesetzt.

Zu Beginn des Interrupt Handlers wird vom Link Register 4 abgezählt, da dieses beim Aufrufen des Handlers das Link Register auf 4 Adressen weiter setzt, als die nächste Instruction. Dementsprechend müssen von ihm 4 abgezählt werden. Anschließend werden die gesamten Register des vorherigen Tasks auf dem Stack gespeichert. !Note: Da das Link Register gebanked ist, kann das Link Register des Interrupt Modus problemlos überschrieben werden.

Anschließend wird das *SPSR*, in welchem das *CPSR* des vorherigen Task gespeichert ist, auf dem Stack gespeichert. Anschließend wird der Interrupt Handler des Rust Codes aufgerufen, welcher unter anderem dafür sorgt, dass der richtige Interrupt Handler ausgeführt wird. Anschließend wird eine sogenannte *Data Instruction Barrier (dsb)* eingefügt, welche dafür sorgen soll, dass alle Memory Operationen abgeschlossen werden. Abschließend werden wieder das *SPSR* und die Register wiederhergestellt. Hierbei ist wichtig zu bemerken, dass wenn bei der *ldmfd* Instruction ein *^* als Suffix angehängt wird und ein *pc* in der Liste der Register vorhanden ist, außerdem noch das *SPSR* in das *CPSR* kopiert wird und somit das *CPSR* der vorherigen Task wieder zurückgesetzt wird.



src/asm/interrupts.S

```
1  irq_handler:
2      sub lr, lr, #4
3      stmdf sp!, {r0-r12, lr}
4
5      mrs r11, spsr
6      push {r11}
7
8      bl handle_interrupt
9
10     mov r0, #0
11     mcr p15, #0, r0, c7, c10, #4
12
13     pop {r11}
14     msr spsr, r11
15
16     ldmfd sp!, {r0-r12, pc}^
```

Die Aufgabe des Rust Handlers ist unter anderem die aktuelle Interrupt Nummer auszulesen, den entsprechenden Interrupt auszuführen und abschließend den Interrupt zu clearen, um dem Prozessor zu signalisieren, dass der Interrupt erfolgreich abgeschlossen wurde.

○ ○ ○

src/interrupts.rs

```
1 static mut INTERRUPT_HANDLERS: &mut [fn(); 128] = &mut [noop; 128];
2
3 #[no_mangle]
4 fn handle_interrupt() {
5     let interrupt = current();
6     execute(interrupt);
7     clear();
8 }
```

Clocks

Jede Peripherie in dem AM335 Controller benötigt einen Clock für die interne Logik des Moduls. Dieser wird “Interface Clock” genannt. Dieser Clock muss explizit aktiviert werden, da die Clocks auch Leistung verbrauchen und sind demnach standardmäßig deaktiviert.

Hier unterscheidet man unter anderem zwischen der WKUP und der PER Domain.

Die WKUP Domain beinhaltet pro Peripherie meist ein Modul, welches davon gespeist wird. Diese Module werden vor allem dann verwendet, wenn der Controller in den Low-Power Modus wechselt, und diese den Controller durch gewisse Signale wieder aufwecken können. Diese Module wurden hier jedoch nicht verwendet, da der Low-Power Modus nicht verwendet wurde.

Die PER Domain beinhaltet alle übrige Peripherie des Controllers.

Die verschiedenen Clocks können aktiviert werden, indem man *0x2* in das jeweilige “Clock Module” Register schreibt, um den Clock zu aktivieren. Dazu wurde für das Kernel ein Modul geschrieben, durch welches jeder Clock in einer einfachen API aktiviert werden kann.

Mit der folgenden Routine kann das Kernel den Clock für ein spezifisches Modul aktivieren:

```
○ ○ ○
src/peripherals/gpio.rs

1 clock::enable(clock::FuncClock::Gpio1);
```

Pinmuxxing

Da der Prozessor sehr viele verschiedene Peripherien implementiert hat, kann er nicht einen Pin mit nur einer Funktionalität für jede Peripherie bereitstellen. Daher implementiert der AM335 Pinmuxxing für bis zu 8 verschiedene Funktionen eines einzigen Pins. Durch das *Control Module* kann für jeden Pin eine von bis zu 8 verschiedenen Funktionen ausgewählt werden, wobei dabei noch definiert werden kann, ob der Input eines Pins aktiviert werden sollte. Außerdem kann man dabei noch definieren, ob ein bestimmter Pin einen Pullup-, Pulldown-Widerstand oder keinen Widerstand am Ausgang anliegen sollte.



src/pinmux.rs

```
1 pub fn set_pin_mode(offset: u32, mode: u32, input_enable: bool, pull_resistor:  
    PullResistor) {  
2     let control_module = CONTROL_MODULE_BASE + offset;  
3  
4     write_addr(  
5         control_module,  
6         mode | (pull_resistor.to_mask() << 3) | ((input_enable as u32) << 5),  
7     );  
8 }
```

GPIO

Initialisierung

Um das GPIO Modul zu verwenden, werden zwei Clocks benötigt. Der erste ist hierbei der Interface Clock, dieser ist wie bei anderen Modulen nötig, um die Register ansteuern zu können. Der zweite ist der Debounce Clock, welcher automatisch für das Debouncing sorgt und einen 32KHz Clock benötigt.

Das GPIO Modul beinhaltet insgesamt ein Modul in der WKUP Domain (GPIO0) und drei Module in der PER Domain (GPIO1-3). Da das Modul mehr als ein Modul enthält, unterteilt man die GPIO Pins in vier verschiedene “GPIO Banks”. Jede dieser Banks beinhaltet insgesamt 32 verschiedene Pins.

Pin Modus

Wie bei anderen Microcontrollern muss zu Beginn der Pin-Modus des Moduls gesetzt werden. Dies wird durch das *GPIO_OE* Register ermöglicht, wobei ein gesetztes Bit einem Input entspricht und ein nicht gesetztes Bit einem Output entspricht. Wichtig zu beachten ist hierbei, dass wie bei anderen GPIO-Registern das n-te Bit für den n-ten Pin der GPIO-Bank zuständig ist.

```
○ ○ ○                                     src/peripherals/gpio.rs

1 pub fn pin_mode((pin, bank): GpioPin, mode: GpioMode) {
2     match mode {
3         GpioMode::Input => {
4             set_bit(bank as u32 + GPIO_OE, pin);
5         }
6         GpioMode::Output => {
7             clear_bit(bank as u32 + GPIO_OE, pin);
8         }
9     }
10 }
```

Pin Schreiben

Um einen Pin zu beschreiben, dessen Richtung als Output eingestellt wurde, muss man das jeweilige Bit in dem *GPIO_DATAOUT* Register setzen.

```
○ ○ ○                                     src/peripherals/gpio.rs

1 pub fn write((pin, bank): GpioPin, value: bool) {
2     if value {
3         set_bit(bank as u32 + GPIO_DATAOUT, pin);
4     } else {
5         clear_bit(bank as u32 + GPIO_DATAOUT, pin);
6     }
7 }
```

Pin Lesen

Um einen GPIO Pin auszulesen, muss das jeweilige Bit vom *GPIO_DATAIN* Register ausgelesen werden.

```
○ ○ ○                                     src/peripherals/gpio.rs

1 pub fn read((pin, bank): GpioPin) → bool {
2     read_bit(bank as u32 + GPIO_DATAIN, pin)
3 }
```

Interrupts

Initialisierung

Jedes GPIO Modul bietet je zwei Interrupt Linien. Dies hat den Grund, verschiedenen Pins verschiedene Prioritäten zuzuordnen. In dieser Implementation wurde der Einfachheit halber aber nur eine Interrupt Linie verwendet. Mit den folgenden Routinen werden die Interrupts für die GPIO Bank 1 aktiviert und der Handler dem Interrupt zugewiesen.

```
○ ○ ○                                     src/peripherals/gpio.rs

1 interrupts::enable_interrupt(Interrupt::GPIOINT1A, Mode::IRQ, 1);
2 interrupts::register_handler(handle_interrupts, Interrupt::GPIOINT1A);
```

Registration

Um einen Handler für einen spezifischen GPIO Pin zu registrieren, muss der Benutzer spezifizieren, ab welchem Event der Interrupt getriggert wird. Dies kann eines oder beide von steigender oder fallender Flanke sein. Der Code setzt hier den Interrupt Handler im Array der GPIO Handler und setzt das Bit im *GPIO_IRQSTATUS_SET0*, welches die Interrupts für den Pin aktiviert. Abschließend werden die jeweiligen Bits der entsprechenden Flanken gesetzt.

○ ○ ○

src/peripherals/gpio.rs

```
1 pub fn register_interrupt(pin: u32, interrupt: GpioInterrupt, handler: fn()) {
2     unsafe {
3         GPIO_INTERRUPT_HANDLERS[pin as usize] = handler;
4     }
5
6     set_bit(GPIO01 + GPIO_IRQSTATUS_SET0, pin);
7
8     match interrupt {
9         GpioInterrupt::Rising => set_bit(GPIO01 + GPIO_RISINGDETECT, pin),
10        GpioInterrupt::Falling => set_bit(GPIO01 + GPIO_FALLINGDETECT, pin),
11        GpioInterrupt::Change => {
12            set_bit(GPIO01 + GPIO_RISINGDETECT, pin);
13            set_bit(GPIO01 + GPIO_FALLINGDETECT, pin)
14        }
15    }
16 }
```

Handling

Der Interrupt Handler hat die Aufgabe, den korrekten Handler beim Auslösen eines Interrupts auszuwählen und diesen auszuführen. Zu Beginn liest der Handler die Nummer des ausgelösten Interrupts aus. Dabei wird das entsprechende Bit gesetzt, mittels der “Count Leading Zeros” (clz) Instruction wird ermittelt, welches Bit gesetzt ist und anhand dieser dann der entsprechende Handler ausgeführt. Abschließend wird das entsprechende Bit in dem *GPIO_IRQSTATUS_0* gesetzt, welches den Interrupt als abgeschlossen kennzeichnet.

Dieses Register ist hierbei ein sog. “Write to Clear” Register.

```
○ ○ ○                                     src/peripherals/gpio.rs

1 fn handle_interrupts() {
2     let irq_raw = read_addr(GPIO1 + GPIO_IRQSTATUS_RAW_0);
3     let number = irq_raw.trailing_zeros();
4
5     unsafe {
6         GPIO_INTERRUPT_HANDLERS[number as usize]();
7     }
8
9     write_addr(GPIO1 + GPIO_IRQSTATUS_0, 1 << number);
10 }
```

I2C

Damit der Leser die weiteren Seiten bezüglich der I2C Implementation besser verstehen kann, möchte ich in diesem Abschnitt die Funktionsweise des I2C Moduls erklären, weil ein Verständnis dieser ohne die Erklärung sehr schwer sein kann. Ich möchte mich hierbei der Einfachheit halber nur auf die Transmit Funktionsweise beschränken, dieselben Prinzipien gelten jedoch ebenso für die Receive Funktionsweise. Es gilt hier die Annahme, dass der Leser einigermaßen mit dem I2C Protokoll vertraut ist.

Um zu vermeiden, dass die Software jedes Bit der Nachricht einzeln und manuell senden muss, ist auf dem AM335 eine 32-Bit FIFO für jeweils Rx und Tx implementiert. Diese hat den Vorteil, dass sie im Vorfeld von der Software gefüllt werden kann und ohne deren Zutun ausgegeben werden kann. Wenn die FIFO einen bestimmten Threshold erreicht, kann die FIFO durch einen Interrupt an den Controller signalisiert werden, welcher sie anschließend wieder befüllt bzw. entleert.

Der Threshold wird bei solchen FIFOs meist in der Mitte gesetzt, um Interrupts nicht zu oft auszulösen aber auch um zu garantieren, dass die FIFO leer ist, bevor sie wieder gefüllt wird weil die Software zu langsam ist.

Die folgende Abbildung beschreibt hierbei den gesamten Prozess. Zu Beginn wird ein Interrupt ausgelöst, welcher das Ziel hat, die FIFO zu füllen. Diese werden dann anschließend nach und nach gesendet und dementsprechend hat die FIFO immer weniger Werte. Wenn die Anzahl der Werte nun kleiner als TXTRSH, also dem Threshold wird, wird ein neuer Interrupt ausgelöst, der die FIFO weiter befüllt. Dieser Prozess wird immer weiter ausgeführt, bis alle Daten gesendet wurden.

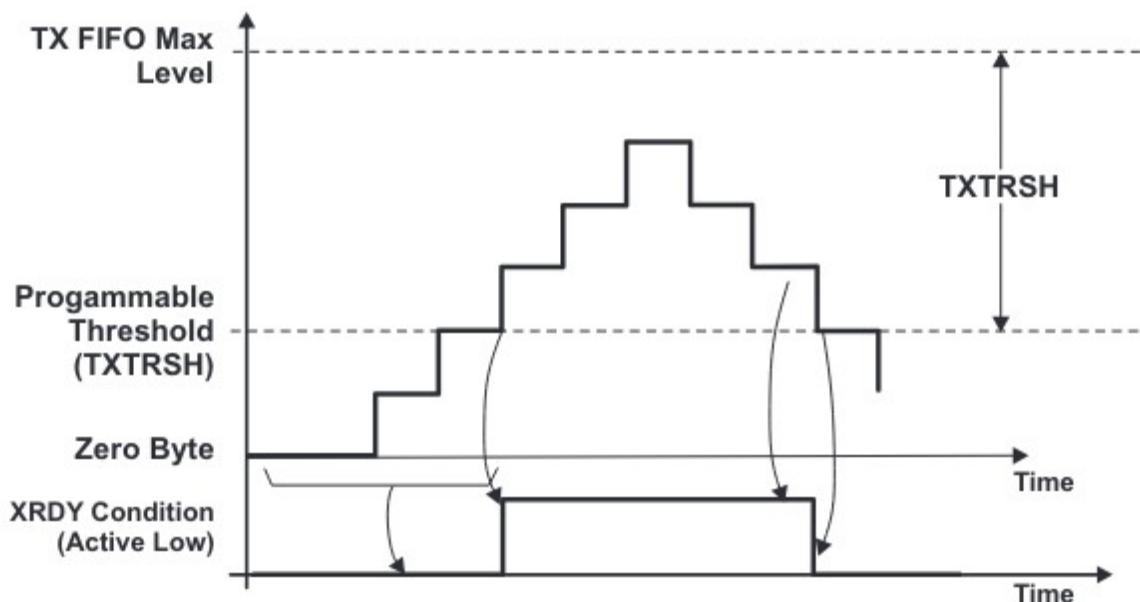


Abbildung 24. AM335 I2C TXTRSH TRM Seite 4595

Initialisierung

Der AM335 beinhaltet insgesamt 3 I2C Module. Eines davon befindet sich in der WKUP Domain, die beiden anderen in der PER Domain. Bei der Implementation des I2C Moduls wurde hier das I2C2 Modul verwendet. Der Clock dieses Moduls kann ähnlich wie beim GPIO Modul aktiviert werden.

○ ○ ○

src/peripherals/i2c.rs

```
1 clock::enable(FuncClock::I2C2);
```

Anschließend werden die Interrupts für das I2C Modul aktiviert, da dieses mit Interrupt arbeitet.

○ ○ ○

src/peripherals/i2c.rs

```
1 interrupts::enable_interrupt(Interrupt::I2C2INT, Mode::IRQ, 2);
2 interrupts::register_handler(irq_handler, Interrupt::I2C2INT);
```

Um das I2C Modul anschließend zu initialisieren, muss zunächst das Modul in einen Software-Reset Modus gewechselt werden. Anschließend werden die Clock Register des I2C Moduls initialisiert, um einen Clock Frequenz von 100kHz im Master Modus zu generieren. Anschließend wird die eigene Adresse des gesetzt, für den Fall, dass das Modul als Slave angesprochen wird. Nun kann das Modul aktiviert werden und wartet anschließend bis der Reset abgeschlossen ist. Abschließend wird noch der Threshold auf 16 gesetzt.

○ ○ ○

src/peripherals/i2c.rs

```
1 self.soft_reset();
2 self.init_clocks();
3 self.set_own_address();
4 self.enable();
5 self.wait_reset();
6
7 self.setup_threshold();
```

Operation

Die Implementation für das Ausführen einer bestimmten I2C Operation wurde hier die *hal_embedded* Crate (Library) für das I2C Modul implementiert. Diese wird von vielen verschiedenen Libraries verwendet. Durch die Implementation des *I2C* Traits können diese somit mit meinem Kernel verwendet werden. Die Implementation von *Receive* wurde bisher nicht implementiert, sollte jedoch kein großes Problem darstellen, ich hatte jedoch zunächst andere Prioritäten.

Transmit

Zu Beginn wird hierbei der Modus ausgewählt, in diesem Fall wäre das Master Transmitter. Anschließend wird die Slave Adresse gesetzt und abschließend werden der interne Buffer sowie die FIFO geleert. Abschließend wird noch gewartet, bis der Bus frei ist, denn wenn jemand den Bus bereits beansprucht hat, sollte nicht versucht werden, den Bus selbst zu beanspruchen.

```
○ ○ ○          src/peripherals/i2c.rs

1 for i in 0 .. buffer.len() {
2     self.transmit_buffer.push(buffer[i]);
3 }
4
5 self.set_count(buffer.len() as u32);
6 self.ready = false;
7
8 self.start();
9
10 self.enable_interrupts(I2cMode::Transmitter);
11 self.wait_ready();
12 self.disable_interrupts(I2cMode::Transmitter);
13
14 self.stop();
```

Anschließend werden nun die Daten des *buffer* Parameters in den internen Transmit Buffer kopiert. Zudem wird das *I2C_CNT* Register mit der Länge des Buffers gesetzt, um dem Prozessor mitzuteilen, wie viele Bytes gesendet werden sollen. Dieses Register wird bei jedem Sender dekrementiert und sobald sie null erreicht, wird die Operation als abgeschlossen erklärt. Anschließend wird die *self.ready* Variable auf false gesetzt, mehr dazu jedoch später. Später wird das Start-Bit gesetzt und daraufhin die nötigen Interrupts für den Transmitter-Modus aktiviert. Der Prozessor wartet anschließend, bis die interne *self.ready* Variable durch die Interrupts auf true gesetzt wird und deaktiviert wieder die Interrupts für den Transmitter-Modus. Zum Schluss sendet das Modul das Stop-Bit, woraufhin die Ausführung des Programms fortgesetzt wird.

Interrupts

Ich möchte in diesem Abschnitt gerne nur die Funktionsweise der Transmit Interrupts darlegen. Zudem ist zu beachten, dass dies nur eine minimale Implementation des I2C Moduls ist, und diese besonders in der Fehlerbehandlung verbessert werden kann.

```
1 impl I2cMode {
2     fn interrupts(&self) -> &[I2cInterrupt] {
3         match self {
4             I2cMode::Transmitter => &[
5                 I2cInterrupt::XRDY,
6                 I2cInterrupt::XDR,
7                 I2cInterrupt::ARDY,
8                 I2cInterrupt::NACK,
9             ],
10            I2cMode::Receiver => &[
11                I2cInterrupt::RRDY,
12                I2cInterrupt::RDR,
13                I2cInterrupt::ARDY,
14                I2cInterrupt::NACK,
15            ],
16        }
17    }
18 }
```

Der folgende Code bestimmt, welche Interrupts bei welcher Operation aktiviert werden. Hierbei möchte ich wie bereits gesagt nur auf jene des Transmitter Modus eingehen, jene des Receiver Modus sind konzeptuell jedoch ähnlich gegenüber jenen des Transmitter Modus.

XRDY

Der XRDY Interrupt wird ausgelöst, wenn die zu sendenden Daten unter dem TXTRSH liegen und sie ein Faktor von TXTRSH sind.

○ ○ ○

src/peripherals/i2c.rs

```
1 if value & I2cInterrupt::XRDY as u32 ≠ 0 {
2     for _ in 0..TRANSMIT_THRESHOLD {
3         self.write_data();
4     }
5
6     write_addr(self.base() + I2C_IRQSTATUS, I2cInterrupt::XRDY as u32);
7     return;
8 }
```

XDR

Der XDR Interrupt ist Teil des sogenannten “Draining Feature” des I2C Core. Dieses wird ausgelöst, wenn die Länge der Übrigen zu sendenden Daten unter dem TXTRSH liegt. Dies hat den Vorteil, dass die Software nicht bei jedem XRDY Interrupt überprüfen muss, wie viele Daten sie überhaupt noch senden kann. Dies wird somit auf nur einen Interrupt minimiert.

○ ○ ○

src/peripherals/i2c.rs

```
1 if value & I2cInterrupt::XDR as u32 ≠ 0 {
2     for _ in 0..self.transmit_bytes_available() {
3         self.write_data();
4     }
5
6     write_addr(self.base() + I2C_IRQSTATUS, I2cInterrupt::XDR as u32);
7     return;
8 }
```

ARDY

Der ARDY Interrupt wird ausgelöst, wenn alle Daten gesendet wurden und die Operation als abgeschlossen gilt. Dabei setzt der Interrupt die interne `self.ready` Variable auf `true`, was zur Folge hat, dass der Prozessor die `wait_ready` Funktion verlässt und die Interrupts für den Transmitter Modus wieder deaktiviert, wie [hier](#) beschrieben wird.

○ ○ ○

src/peripherals/i2c.rs

```
1 if value & I2cInterrupt::ARDY as u32 != 0 {
2     self.ready = true;
3
4     write_addr(self.base() + I2C_IRQSTATUS, I2cInterrupt::ARDY as u32);
5     return;
6 }
```

NACK

Wenn das Acknowledgement Bit nicht vom Slave gesendet wird, wird ein *NACK* Interrupt ausgelöst. Dieser setzt hier die *self.error* Variable.

○ ○ ○

src/peripherals/i2c.rs

```
1 if value & I2cInterrupt::NACK as u32 != 0 {
2     self.error = Some(I2cError::Nack);
3     self.ready = true;
4
5     write_addr(self.base() + I2C_IRQSTATUS, I2cInterrupt::NACK as u32);
6 }
```

Nachdem die Transmit Operation wie [hier](#) beschrieben abgeschlossen ist, wird in diesem Abschnitt überprüft, ob die *self.error* Variable *true* ist. Wenn sie wahr ist, wird die Operation gestoppt und der Fehler zurückgegeben.

○ ○ ○

src/peripherals/i2c.rs

```
1 if let Some(error) = self.error {
2     self.stop();
3     self.disable();
4
5     self.error = None;
6
7     return Err(error);
8 }
```

Timer

Der AM335 enthält insgesamt 7 *DMTimer* Module und ein *DMTimer_1ms* Modul. Durch das letztere Modul kann eine höhere Genauigkeit erreicht werden, das Modul ist jedoch nicht implementiert worden, da es keine hohe Priorität hatte.

Bei der folgenden Implementation wurde der sogenannte *Auto Reload* Modus verwendet. Dabei startet der Timer mit einem Startwert und inkrementiert den Counter in einem definierten Intervall. Sobald der Counter das Maximum (`0xFFFF_FFFF`) erreicht, wird der Counter wieder auf den Wert des *Load Registers* gesetzt, dieses entspricht normalerweise dem Startwert. Jedesmal, wenn der Prozessor den Counter Wert zurücksetzen muss, wird ein Interrupt ausgelöst.

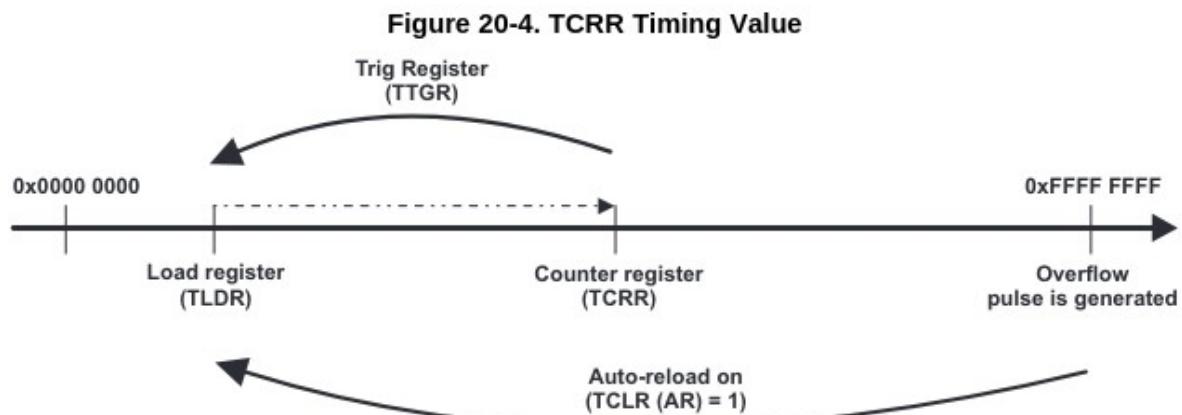


Abbildung 25. AM335 Timer TCRR Register TRM Seite 4442

Durch die richtige Definition des Startwertes/Load Register kann der Programmierer somit in einem beliebigen Intervall einen Interrupt auslösen. Der Clock kann hierbei angepasst werden. Bei der folgenden Implementation wurde nur der 32kHz Clock verwendet, es könnte jedoch ein beliebiger Clock verwendet werden, welcher jedoch über die Hardware an den TCLKIN Pin angelegt werden müsste. Als Prescaler wurde der Einfachheit halber hier ein Wert von 1 verwendet. Bei der untenstehenden Tabelle ist es wichtig zu beachten, dass der 32KHz Clock nicht exakt einen Intervall von 31.25µs produziert, sondern einen Intervall von ca. 30.5µs. Bei meinem Projekt ist dies jedoch nicht unbedingt wichtig, weshalb hier der normale *DMTimer* verwendet wurde und nicht der genauere *DMTimer_1ms*.

Clock	Prescaler	Resolution	Interrupt Period Range
32.768 KHz	1 (min)	31.25 us	31.25 us to ~36h 35m
	256 (max)	8 ms	8 ms to ~391d 22h 48m
25 MHz	1 (min)	40 ns	40 ns to ~171.8s
	256 (max)	10.24 us	~20.5 us to ~24h 32m

Abbildung 26. AM335 Timer Intervall Reichweite

Initialisierung

Zu Beginn muss wie bei jeder Peripherie der Clock aktiviert werden.



src/internals/timer.rs

```
1 self.timer.clock().enable()
```

Anschließend werden das *Timer Counter Register* und das *Timer Load Register* mit dem Reload Wert. Dieser Wert kann unter anderem mit der folgenden Formel berechnet werden.

$(0xFFFF\ 0xFFFF - TLDR + 1) \times \text{timer Clock period} \times \text{Clock Divider (PS)}$. Wenn man nun einen Intervall von z.B. 1 ms haben möchte, muss man für das *TLDR* Register hier den Wert *0xFFFFFE0* einsetzen.



src/internals/timer.rs

```
1 write_addr(self.timer.address() + TIMER_LOAD, self.reload);  
2 write_addr(self.timer.address() + TIMER_COUNTER, self.reload);
```

Anschließend werden die Interrupts für den Auto-Reload Modus aktiviert, gefolgt von dem Registrieren des Handlers des Interrupts.



src/internals/timer.rs

```
1 self.irq_enable();  
2  
3 interrupts::register_handler(Self::handle_timer_irq, self.timer.interrupt());  
4 interrupts::enable_interrupt(self.timer.interrupt(), interrupts::Mode::IRQ, 0);
```

Interrupts

Zu Beginn des Interrupt Handlers wird der aktive Interrupt ausgelesen. Anschließend wird durch die Interrupt Nummer der aktive *DMTimer* ausgewählt. Von diesem werden dann das Interrupt-Flag zurückgesetzt und der Handler des Timers ausgeführt.



src/internals/timer.rs

```
1 fn handle_timer_irq() {
2     let interrupt = interrupts::current();
3
4     if let Some(interrupt) = interrupt {
5         let timer = get_timer(interrupt);
6
7         if let Some(timer) = timer {
8             timer.irq_acknowledge();
9             (timer.handler)();
10        }
11    }
12 }
```

Modularität

Bei der Implementation des Timer Moduls wurde zudem auf die Modularität des Systems Wert gelegt, wodurch es möglich ist, 6 verschiedene Timer mit je verschiedenen Intervallen zu verwenden.



src/internals/timer.rs

```
1 pub enum DmTimer {
2     Timer2,
3     Timer3,
4     Timer4,
5     Timer5,
6     Timer6,
7     Timer7,
8 }
```

Dies wurde dadurch realisiert, dass ein globales Array mit all den verschiedenen Timern gespeichert wird. Mit Hilfe der *register_timer* Funktion kann ein Timer dem Array hinzugefügt werden, wobei zu bemerken ist, dass er dabei initialisiert wird. Mithilfe der *get_timer* Funktion kann über den Interrupt der aktuell ausgeführte *DMTimer* zurückgegeben werden.

○ ○ ○

src/internals/timer.rs

```
1 static mut TIMERS: &mut [Option<Timer>; 6] = &mut [const { None }; 6];
2
3 pub fn register_timer(dm_timer: DmTimer, reload: u32, handler: fn()) {
4     let timer = Timer::new(dm_timer, reload, handler);
5     unsafe { TIMERS[dm_timer as usize] = Some(timer) }
6 }
7
8 pub fn get_timer(interrupt: Interrupt) → &'static Option<Timer> {
9     let dm_timer = match DmTimer::try_new(interrupt) {
10         Some(dm_timer) ⇒ dm_timer,
11         None ⇒ return &None,
12     };
13
14     unsafe { &TIMERS[dm_timer as usize] }
15 }
```

Kernel

In dem Kernel Kapitel möchte ich gerne die Struktur des Kernels beschreiben, sowie die Kernel-spezifischen Features, wie z.B. Multitasking und Paging erklären. Dazu zählen unter anderem der Aufbau der Codebase und das Hinzufügen von externen Tasks, die anschließend gleichzeitig ausgeführt werden. Unter anderem wird in der folgenden Beschreibung des Kernels auch die Implementierung von User und Kernel Space beschrieben, die verhindert, dass User Code das Kernel nicht auf Kernel Funktionen direkt zugreifen kann.

Sysclock

Der folgende Code ist die Implementation des Sysclock Moduls. Dieses ist dafür zuständig, die Zeit des Systems zu speichern. Dabei initialisiert er Timer2 mit einem Wert von `0xFFFF_FFE0`, was einen Intervall von 1 ms garantiert. Im Interrupt Handler wird der Sysclock um 1 inkrementiert. Anschließend wird alle 10 ms die `yield_task` routine ausgeführt, mehr dazu jedoch später.

Durch die `millis` Routine kann anschließend der Wert des Sysclocks ausgelesen werden.



src/internals/sysclock.rs

```
1 pub fn initialize() {
2     timer::register_timer(DmTimer::Timer2, 0xFFFF_FFE0, interrupt_handler);
3 }
4
5 static mut SYS_CLOCK: u32 = 0;
6
7 fn interrupt_handler() {
8     unsafe { SYS_CLOCK += 1 };
9
10    if unsafe { SYS_CLOCK } % 10 == 0 {
11        unsafe { yield_task() };
12    }
13 }
14
15 pub fn millis() → u32 {
16     unsafe { SYS_CLOCK }
17 }
```

Syccalls

Syccalls, Software Interrupts oder auch *Supervisor Calls (SVC)* in der ARM-Architektur werden dazu verwendet, privilegierte Aktionen in einem nicht privilegierten Modus durchzuführen. Da der Prozessor in einem nicht privilegierten Modus nicht den Modus selbst wechseln kann, werden Syccalls benötigt.

Dabei können dem Syscall verschiedene Parameter gegeben werden, durch welche er die entsprechende Aktion auswählt und anschließend eine gewünschte Aktion ausführt oder einen bestimmten Wert wieder zurückgibt.

Durch die folgenden Instructions kann z.B. der Syscall Nummer 0 mit dem ersten Parameter als Wert 5 ausgeführt werden. Die Syscall Nummer wird hier dazu verwendet, die Art des Syccalls auszuwählen, während die weiteren Register als Parameter verwendet werden.

```
○ ○ ○  
1 mov r0, #5  
2 svc #0x0
```

Handler

Der Handler der Software Interrupts ist ähnlich aufgebaut wie jener der Interrupts. Es gibt hier jedoch einige wichtige Unterschiede. Zum Ersten zeigt das Link Register im Gegensatz zum Interrupt direkt auf die nächste Instruction. Zudem nimmt der Syscall Handler die Register *r0-r3* als Parameter auf und kann zudem einen Wert im *r0* Register zurückgeben.

Um den ersten Teil des Handlers zu verstehen, muss man wissen, dass wenn ein *Supervisor Call* durchgeführt, wie in der obenstehenden Abbildung, muss ein Immediate angehängt werden, welches bis zu 24 bit lang sein kann und in der Instruction codiert sein wird.

Der Handler nimmt dabei den Wert der vorherigen Instruction, also jener, die den SVC ausgelöst hat und löscht die 8 MSBs, sodass nur noch die Syscall Nummer übrig bleibt.

Anschließend werden die entsprechenden Register auf dem Stack gespeichert, welcher dann in das *r1* Register gespeichert wird. Anschließend werden noch 8 Bytes Platz im Stack Pointer geschaffen, um die Rückgabewerte speichern zu können.

```
○ ○ ○

1 handle_swi:
2     stmfd sp!, {r0-r12, lr}
3
4     sub lr, lr, #4
5     ldr r12, [lr]
6     bic r12, r12, #0xFF000000
7
8     push {r0-r3, r12}
9     mov r1, sp
10
11    sub sp, #8
12    mov r0, sp
13
14    bl swi_handler
15
16    ldr r0, [sp, #4]
17    ldrb r1, [sp]
18    add sp, #8
19
20    add sp, sp, #20
21
22    cmp r1, #0x1
23    beq exit
24
25    str r0, [sp]
26
27    ldmfd sp!, {r0-r12, pc}^
```

Dann wird der Rust Software Interrupt Handler ausgeführt, dessen Aufgabe es ist, die entsprechenden Aktionen auszuführen und anschließend, falls nötig, einen Rückgabewert zurückzugeben.

Nachdem der Rust Software Interrupt Handler ausgeführt wurde, werden die beiden Rückgabewerte in *r0* und *r1* gespeichert und der Stackpointer um 8 erhöht, um diese vom Stack zu trennen. Anschließend wird der Stackpointer um 20 angehoben, um den Platz der Eingangsparameter auf dem Stack wieder frei zu machen. Wenn das *r1* Register 1, also true entspricht, wird die *exit* Funktion ausgeführt. Abschließend wird das auf dem Stack gespeicherte *r0* Register mit dem Rückgabewert des Syscalls überschrieben.

Die *exit* Funktion wurde hier hinzugefügt, um es möglich zu machen, nach einem *SVC* weiterhin im Supervisor Modus zu bleiben, also die Ausführung der aktuellen Task zu stoppen. Dabei wird der Modus auf Supervisor gewechselt, sowie der Stackpointer auf 56 angehoben, da die Register *r0-r12* sowie das *pc* nicht vom Stack gepoppt wurden. Abschließend springt der Prozessor in die *kernel_loop* Funktion, welche anschließen die nächste Task auswählt und ausführt, mehr dazu jedoch später.

```
○ ○ ○  
1 exit:  
2     msr cpsr_c, #0x53  
3     add sp, sp, #56  
4  
5     b kernel_loop
```

Rust Handler

Signatur

Die folgende Signatur wird für den Rust Syscall Handler verwendet. Dabei wird wie bereits zuvor besprochen der Stack Pointer für den Rückgabewert in *r0* gespeichert und die Argumente werden in einem Stack Pointer in *r1* übergeben.

```
○ ○ ○                                     src/kernel.rs

1 extern "C" fn swi_handler(frame: &TrapFrame) → SyscallReturn {
```

In den untenstehenden Abbildungen kann man die erwähnten Datenstrukturen erkennen. So speichert das *TrapFrame* Struct die verschiedenen Register in seinen Feldern. Das *SyscallReturn* Struct hingegen speichert, ob der Handler den User Modus verlassen soll und zudem eine der verschiedenen Rückgabewerte.

```
○ ○ ○                                     src/kernel.rs

1 #[repr(C)]
2 struct TrapFrame {
3     r0: u32,
4     r1: u32,
5     r2: u32,
6     r3: u32,
7     r12: u32,
8 }
9
10 #[repr(C)]
11 struct SyscallReturn {
12     exit: bool,
13     value: SyscallReturnValue,
14 }
15
16 #[repr(C)]
17 pub union SyscallReturnValue {
18     pub millis: u32,
19     pub gpio_read: bool,
20     pub i2c_write: I2cError,
21     pub alloc: *mut u8,
22     pub none: (),
23 }
```

Logik

Im `swi_handler` wird anschließend das *Trapframe* in ein Syscall Enum konvertiert. Anschließend wird die entsprechende Logik des Syscalls ausgeführt und von jedem dieser Handler der Syscalls ein *SyscallReturn* Objekt zurückgegeben.

○ ○ ○

src/kernel.rs

```
1 extern "C" fn swi_handler(frame: &TrapFrame) → SyscallReturn {
2     let syscall: Syscall = match frame.try_into() {
3         Ok(syscall) ⇒ syscall,
4         Err(_) ⇒ panic!("invalid syscall"),
5     };
6
7     match syscall {
8         Syscall::Exit ⇒ {
9             ...
10            SyscallReturn::exit()
11        ...
12    }
```

Syscall Enum

Das Syscall Enum wird im Kernel verwendet, um die verschiedenen Parameter der Syscalls zu speichern. Ein weiterer Vorteil ist zudem, dass durch die Verwendung dieses Enums der Benutzer nicht mehr manuell Assembly schreiben muss, um den Syscall auszuführen, dies kann automatisch vom Syscall durchgeführt werden.

○ ○ ○

src/kernel.rs

```
1 pub enum Syscall<'a> {
2     Exit,
3     Yield {
4         sp: u32,
5         pc: u32,
6         until: Option<u32>,
7     },
8     Millis,
9     GpioRead {
10        pin: GpioPin,
11    },
12     GpioWrite {
13        pin: GpioPin,
14        value: bool,
15    },
16     I2cWrite {
17        address: u8,
18        data: &'a [u8],
19    },
20     Panic,
21     Alloc {
22        layout: Layout,
23    },
24     Dealloc {
25        ptr: *mut u8,
26        layout: Layout,
27    },
28 }
29
```

Durch die folgende Funktion kann der Syscall ausgeführt werden, wobei er zudem noch die verschiedenen Rückgabewerte speichert und diese ebenfalls zurückgibt.

○ ○ ○

src/kernel.rs

```
1 impl Syscall<'_> {
2     pub fn call(self) -> Option<SyscallReturnValue> {
3         match self {
4             Syscall::Exit => unsafe {
5                 asm!("svc 0x0", options(noreturn));
6             },
7             Syscall::Yield { sp, pc, until } => unsafe {
8                 asm!("svc 0x1", in("r0") sp, in("r1") pc, in("r2")
9                     until.unwrap_or(0), options(noreturn));
10            },
11            ...
```

Multitasking

Da der AM335 nur einen Core hat, muss der Benutzer, wenn er mehrere Tasks gleichzeitig ausführen möchte, jedem Task eine gewisse Zeit zuweisen, bevor er einen anderen Task ausführt. Dieser Prozess, welcher auch “CPU Scheduling” genannt wird, wurde bei meinem Betriebssystem in Anwendung gebracht.

Bei der Implementation des CPU Scheduling wurde jedem Prozess eine Ausführungszeit von 10ms zugewiesen. Möglich wird das ganze dadurch, dass jedem Task ein eigener Stack zugewiesen wird und auf diesem anschließend das *CPSR* und die Register dieser Task gespeichert werden.

Das Kernel implementiert hierbei *Cooperative* und *Preemptive Multitasking*. Bei *Preemptive Multitasking* wird durch die CPU in einem bestimmten Zeitraum der Task gewechselt, dies wird meist durch Timer und Interrupts realisiert. Bei *Cooperative Multitasking* gibt der aktive Task jedoch bereitwillig die Ausführung weiter, dies kann zum Beispiel bei einem *Sleep* der Fall sein oder wenn der Task aktuell auf die Abschluss eines Interrupts wartet.

Preemptive Multitasking

In dem Interrupt Handler des Sysclocks wird jede 10 ms die *yield_task* Routine aufgerufen, welche das *yielded* Flag setzt. Wenn dieses Flag am Ende des Interrupt Handlers gesetzt ist, werden die Register und das *CPSR* nicht wie üblicherweise zurückgesetzt, sondern werden auf dem Stack der vorherigen Task gespeichert, woraufhin die nächste Task ausgeführt wird.

```
○ ○ ○                                     src/kernel.rs

1 fn interrupt_handler() {
2     unsafe { SYS_CLOCK += 1 };
3
4     if unsafe { SYS_CLOCK } % 10 == 0 {
5         unsafe { yield_task() };
6     }
7 }
```

Am Schluss des Interrupt Handler Codes befindet sich folgender Abschnitt. Dabei wird im oberen Abschnitt überprüft, ob der vorherige Modus Supervisor war, das Kernel also keinen Task gerade ausführte. In diesem Fall werden die Register wie üblich wiederhergestellt und der Modus gewechselt. Wenn dies nicht der Fall ist, wird das *yielded* Flag ausgelesen, wenn dieses gesetzt ist, wird der Kontext der vorherigen Task gespeichert, andernfalls werden die Register wie üblich wiederhergestellt.

```
○ ○ ○                                     src/asm/interrupts.S

1      pop {r11}
2      msr spsr, r11
3      and r11, r11, #0b11111
4      cmp r11, #0b10011
5      beq return_interrupt
6
7      ldr r0, yielded
8      cmp r0, #0x1
9      beq store_context
10
11 return_interrupt:
12      ldmfd sp!, {r0-r12, pc}^
```

Die *store_context* Routine hat die Aufgabe, die Register der Task auf dem Stack der Task zu speichern, sodass sie wiederhergestellt werden können. Zu Beginn werden die Register hier vom Stack gepoppt, wichtig ist dabei zu beachten, dass der Suffix [^] nicht angehängt wurde, weshalb der

Modus nicht gewechselt wird. Wichtig zu nennen ist hierbei auch, dass einige Register wie das Link Register und das *SPSR* in temporären Variablen gespeichert werden, da die Register gebankt sind und somit nicht von anderen Modi verwendet werden können. Anschließend wird der Modus zum System Modus gewechselt, welcher die Register mit dem User Modus teilt, woraufhin die Register der vorherigen Task auf dem Stack der Task gespeichert werden, ebenso wie das *SPSR* genauer gesagt das *CPSR* der vorherigen Task.

Abschließend wird ein Software Interrupt initiiert, welcher die Aufgabe hat, den Stack Pointer der Task sowie die nächste Instruction der Adresse, welche im Link Register gespeichert war, als Parameter dem Syscall mitzugeben.

```
○ ○ ○                                     src/asm/interrupts.S

1  store_context:
2  ldmfd sp!, {r0-r12, lr}
3
4  str lr, next_pc
5
6  push {r0}
7  mrs r0, spsr
8  str r0, temp_spsr
9  pop {r0}
10
11 msr cpsr_c, #0xDF
12 stmfd sp!, {r0-r12, lr}
13
14 mov r0, #0x0
15 str r0, yielded
16
17 ldr r0, temp_spsr
18 push {r0}
19
20 mov r0, sp
21 ldr r1, next_pc
22 mov r2, #0x0
23 svc #0x1
24
```

In dem Handler des Syscalls wird anschließend der aktuelle Task ausgewählt und der Stack Pointer und die Rückföhadrresse dort gespeichert. Zudem wird der Status der Task auf *Stored* gesetzt, was dem Scheduler signalisiert, dass er zudem die Register wiederherstellen muss. Abschließend ein *exit* zurückgegeben, wodurch der aktuelle Code nicht weiter ausgeführt wird sondern wieder in den *kernel_loop* springt.

○ ○ ○

src/kernel.rs

```
1 Syscall::Yield {
2     sp,
3     pc,
4     until: None,
5 } => {
6     let scheduler = scheduler();
7     if let Some(task) = scheduler.current() {
8         task.context.pc = pc;
9         task.context.sp = sp;
10        task.state = TaskState::Stored;
11    }
12
13    scheduler.cycle();
14
15    SyscallReturn::exit()
16 }
17
```

Cooperative Multitasking

Beim *Cooperative Multitasking* gibt der Prozessor freiwillig die Ausführung auf, bis eine bestimmte Aktion ausgeführt wurde. Dieser Prozess wird auch *yielding* genannt. Bei diesem Projekt wurde *yielding* bislang nur für die *sleep* Routine verwendet, könnte jedoch weiter ausgebaut werden, wie z.B. bis eine I2C Operation abgeschlossen ist.

Um die aktuelle Task zu *yielden* muss man folgende Funktion aufrufen, wobei sie ein Argument hat, welches die Zeit ist, bis zu welcher die Exekution der Task aufgeschoben werden soll. Dabei werden zuerst die jeweiligen Register auf dem Stack Pointer gespeichert, ebenfalls wie das *CPSR* und anschließend über die verschiedenen Register als Parameter für den Syscall hinzugefügt.

src/sysclock.rs

```
1 yield_task:  
2     stmfd sp!, {{r0-r12, lr}  
3  
4     mov r2, r0  
5  
6     mrs r0, cpsr  
7     push {r0}  
8  
9     mov r0, sp  
10    mov r1, lr  
11    mov r2, r2  
12    svc #0x1
```

src/sysclock.rs

```
1 fn yield_task(ms: u32);
```

Der folgende Code ist der Handler des Yield Syscalls, wenn dieser vom Prozess selbst initiiert wurde. Dabei speichert er wie bei *Preemptive Multitasking* den Stack Pointer und die Adresse der nächsten Instruktion, setzt den Status aber auf Waiting und fügt dabei noch die Zeit ein, bis zu welcher die Exekution der Task aufgeschoben werden soll.

○ ○ ○

src/kernel.rs

```
1 Syscall::Yield {
2     sp,
3     pc,
4     until: Some(until),
5 } => {
6     let scheduler = scheduler();
7     if let Some(task) = scheduler.current() {
8         task.context.pc = pc;
9         task.context.sp = sp;
10        task.state = TaskState::Waiting { until };
11    }
12
13    scheduler.cycle();
14
15    SyscallReturn::exit()
16 }
```

Task Wiederherstellung

Wenn der nächste Prozess ausgewählt wurde, und dieser bereits zuvor ausgeführt wurde, müssen dessen *CPSR* und Register auf den ursprünglichen Zustand wiederhergestellt werden. Dabei wird zu Beginn das *CPSR* vom Stack gepoppt und anschließend in das *SPSR* geschrieben, welches am Ende für das *CPSR* eingesetzt wird. Hierbei gilt zu beachten, dass hier vom Register *r0* also vom Stack der Task gepoppt wird. Anschließend werden die Adressen, an denen die Exekution der Task wieder starten soll, auf dem Stack gespeichert, um in das *pc* Register geschrieben zu werden. Dann wird der Stack Pointer im System Modus gesetzt und die Register vom Stack gepoppt. Anschließend wird der Modus wieder zum Supervisor Modus gewechselt und anschließend das auf dem Stack des Supervisor Modus gespeicherte *pc* gepoppt und in den User Modus gewechselt, da dieser in dem *SPSR* gespeichert ist.

○ ○ ○

src/asm/interrupts.S

```
1 restore_context:
2     ldmfd r0!, {r2}
3     msr spsr, r2
4
5     push {r1}
6
7     msr cpsr_c, #0x0DF
8     mov sp, r0
9     pop {r0-r12, lr}
10    msr cpsr_c, #0xD3
11
12    ldmfd sp!, {pc}^
```

○ ○ ○

src/tasks.rs

```
1 fn restore_context(sp: u32, pc: u32) → !;
```

Scheduling

Wie bereits erwähnt wurde als Scheduling Technik hier das sog. Round-Robin Scheduling verwendet. Dabei hat keiner der Prozesse eine Priorität wie bei vielen Betriebssystemen üblich ist, stattdessen wird nach der Ausführung eines Prozesses der nächste Prozess ausgeführt und immer so weiter, bis der Scheduler am letzten Prozess angelangt ist, ab welchem er wieder von vorne beginnt. Dieses Verfahren ist nicht optimal für die meisten Systeme, die zwischen verschiedenen Prioritäten unterscheiden müssen, es wurde jedoch implementiert, da es eine der einfachsten der Scheduling Methoden ist.

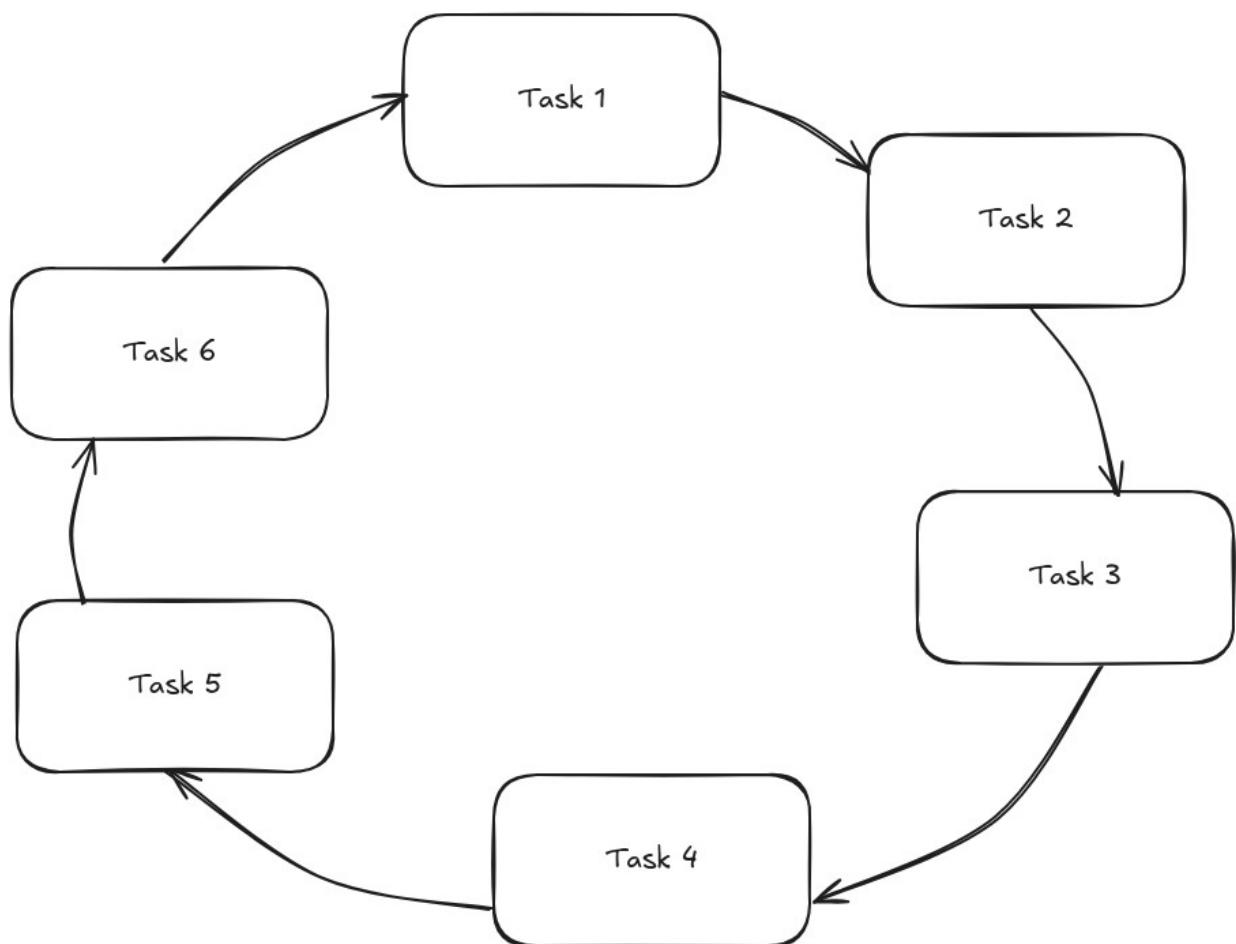


Abbildung 27. Round Robin Scheduling

Der folgende Code ist zuständig für das Ausführen der nächsten ausführbaren Task. Zu Beginn bekommt sie die nächste Task, welche ausgeführt werden soll. Anschließend wird diese nun als aktuelle Task gesetzt und ausgeführt. Wenn der Status der Task *Ready* ist, bedeutet das, dass sie noch nicht ausgeführt wurde und dementsprechend keine Register auf deren Stack gespeichert sind. Wenn der Status jedoch *Stored* ist, wurde die Task bereits schon ausgeführt, wodurch ihre Register auf dem Stack gespeichert sind und wieder gepoppt werden müssen.

```
○ ○ ○          src/internals/tasks.rs

1 pub fn switch(&mut self) {
2     let next_task_id = match self.next_task() {
3         Some(task) => task.id,
4         None => return,
5     };
6
7     self.current_index = Some(next_task_id);
8
9     let task = self.task_mut(next_task_id);
10
11    match task.state {
12        TaskState::Ready => {
13            task.state = TaskState::Running;
14            unsafe {
15                switch_context(task.context.sp, task.context.pc);
16            }
17        }
18        TaskState::Stored => {
19            task.state = TaskState::Running;
20            unsafe {
21                restore_context(task.context.sp, task.context.pc);
22            }
23        }
24        _ => {}
25    }
26}
27
```

Die `switch_context` Funktion speichert so zu Beginn den Stack Pointer im System Modus und wechselt daraufhin wieder in den Supervisor Modus, woraufhin das `SPSR` mit dem User Modus gesetzt wird und abschließend wird das `r1` Register in das `pc` Register verschoben, wobei der `s` Suffix der `mov` Instruction denselben Effekt hat wie der `^` Suffix bei der `ldmfd` Instruction, nämlich dass das `SPSR` in das `CPSR` kopiert wird.

```
○ ○ ○                                     src/asm/kernel.S

1  switch_context:
2      msr cpsr_c, #0x0DF
3      mov sp, r0
4      msr cpsr_c, #0xD3
5
6      mov r2, #0x50
7      msr spsr_c, r2
8
9      movs pc, r1
```

Die `next_task` Funktion hat die Aufgabe, die nächste ausführbare Task zu finden. Dafür iteriert sie über alle Tasks, wenn eine Task ausführbar ist, gibt sie diese zurück. Wenn der Index dann wieder den ursprünglichen Index erreicht, wird die Suche abgeschlossen und ein `None` zurückgegeben.

```
○ ○ ○                                     src/internals/tasks.rs

1  fn next_task(&mut self) -> Option<&mut Task> {
2      let initial_index = self.current_index.unwrap_or(0);
3      let mut index = initial_index;
4
5      loop {
6          let current_task = self.task_mut(index);
7          if current_task.executable() {
8              return Some(current_task);
9          }
10
11         index = (index + 1) % MAX_TASKS;
12         if index == initial_index {
13             break;
14         }
15     }
16
17     None
18 }
```

Die *kernel_loop* Funktion hat die Aufgabe immer wieder den Prozess zu wechseln und wird unter anderem dann aufgerufen, wenn ein Prozess entweder durch *Cooperative* oder *Preemptive Multitasking* die Ausführung beendet.

○ ○ ○

src/kernel.rs

```
1 pub fn kernel_loop() {
2     loop {
3         let scheduler = scheduler();
4         scheduler.switch();
5     }
6 }
```

Paging

Durch das Paging des Kernels kann gewährleistet werden, dass kein User Prozess auf das Memory des Kernels zugreifen kann. Möglich wird dies dadurch, dass in der Memory Region des Kernels sowie jenen der Peripherie nur privilegierte Modi darauf zugreifen können. Da die verschiedenen Tasks im User-Modus ausgeführt werden und dieser nicht-privilegiert ist, können diese nicht darauf zugreifen. Unter kann durch das Paging jeder Prozess selbstständig kompiliert werden, ohne eine Annahme über seine Umgebung treffen zu müssen. Dies wird dadurch möglich, dass der Code einer jeden Task zwar in verschiedenen Orten gespeichert ist, bei der Ausführung einer Task dieser Block von Memory auf *0x0* gemappt wird, ebenfalls wie dessen Stack.

In der folgenden Implementation wurde der L1 Page Table für das Setzen der Zugangsberechtigungen von Peripherie und Kernel Memory verwendet. Der L2 Page Table hingegen, welcher nur einmal implementiert wurde, wurde nur für die verschiedenen Memory Regionen der einzelnen Prozesse verwendet.

Initialisierung

Zu Beginn werden die *L1*- und *L2-Translation Table* initialisiert. Anschließend werden das *TTBCR* und das *TTBR0* initialisiert, wobei das *TTBCR* das Verhältnis zwischen dem *TTBR0* und *TTBR1* festlegt. Das *TTBR0* hingegen speichert die Adresse des L1 Page Tables. Dann wird der TLB invalidiert und die Domains initialisiert, auf welche ich jedoch nicht eingehen werde, da diese nicht verwendet wurden und zudem seit *ARMV7* deprecated sind. Abschließend wird die *MMU* selbst aktiviert, woraufhin die Adressübersetzungen in Kraft treten.

○ ○ ○

src/internals/mmu/setup.rs

```
1 pub fn initialize() {
2     unsafe {
3         l1::initialize();
4         l2::initialize();
5         initialize_ttbcr();
6         initialize_ttbr0();
7         invalidate_tlb();
8         setup_domains();
9         enable_mmu();
10    }
11 }
```

L1 Page Table

Der L1 Page Table ist global im Memory gespeichert. Es muss hierfür ein *struct* verwendet werden, um den Page Table auf 16 KB auszurichten. Dabei werden alle Sektionen der RAM standardmäßig mit Faul Page Table Entries gefüllt.



src/internals/mmu/l1.rs

```
1 pub static mut LEVEL1_PAGE_TABLE: L1PageTable = L1PageTable::new();
2
3 #[repr(align(16384))]
4 pub struct L1PageTable(pub [u32; PAGE_TABLE_SIZE]);
5
6 impl L1PageTable {
7     const fn new() -> Self {
8         L1PageTable([FAULT_PAGE_TABLE_ENTRY; PAGE_TABLE_SIZE])
9     }
10 }
```

Bei der Initialisierung werden anschließend die Regionen, welche das Kernel und die Peripherie beinhalten, mit Section Page Table Entries gefüllt, wobei diese privilegierte Zugangsberechtigungen besitzen, sodass kein User Task auf diese Regionen zugreifen kann.



src/internals/mmu/l1.rs

```
1 const PERIPHERAL_MEMORY: Range<u32> = 0x4400_0000 .. 0x8000_0000;
2 const KERNEL_MEMORY: Range<u32> = 0x4020_0000 .. 0x4040_0000;
3
4 pub fn initialize() {
5     enable_memory_range(KERNEL_MEMORY, AccessPermissions::Privileged);
6     enable_memory_range(PERIPHERAL_MEMORY, AccessPermissions::Privileged);
7 }
8
9 fn enable_memory_range(range: Range<u32>, permissions: AccessPermissions) {
10    for page in range.step_by(PAGE_SIZE as usize) {
11        let section = L1SectionPageTableEntry::new(page, permissions);
12        unsafe {
13            LEVEL1_PAGE_TABLE.0[page as usize >> PAGE_SIZE_BITS] = section.into();
14        }
15    }
16 }
```

L2 Page Table

Der L2 Page Table muss wie der L1 Page Table ebenfalls global gespeichert werden, muss jedoch im Unterschied zu diesem nur um 1 KB ausgerichtetet werden.

○ ○ ○

src/internals/mmu/l2.rs

```
1 #[no_mangle]
2 static mut LEVEL2_PAGE_TABLE: L2PageTable = L2PageTable::new();
3
4 #[repr(align(1024))]
5 pub struct L2PageTable([u32; PAGE_TABLE_SIZE]);
6
7 impl L2PageTable {
8     const fn new() → Self {
9         L2PageTable([L2FAULT_PAGE_TABLE_ENTRY; PAGE_TABLE_SIZE])
10    }
11 }
```

Um ihn zu initialisieren, wird das erste Element des L1 Page Tables mit einem Pointer zum L2 Page Table ersetzt, sodass nun alle Adressanfragen von *0x0-0x100000* auf die entsprechenden L2 Page Table Entries fallen.

○ ○ ○

src/internals/mmu/l2.rs

```
1 pub fn initialize() {
2     let l1_pointer = L1PointerTableEntry::new(&raw mut LEVEL2_PAGE_TABLE);
3
4     unsafe {
5         LEVEL1_PAGE_TABLE.0[0] = l1_pointer.into();
6     }
7 }
```

Erstellen eines L2 Page Tables

Um eine L2 Page Table Entry zu erstellen, muss der Programmierer die virtuelle Adresse sowie die ASID hinzufügen. Da die virtuelle Adresse bei der folgenden Implementation des Kernels immer 0 ist, wird ein weiteres globales Array verwendet, welches die Aufgabe hat, die bereits verwendeten Pages zu tracken, sodass diese nicht für einen anderen Prozess verwendet werden können.

```
○ ○ ○          src/internals/mmu/l2.rs

1 pub struct L2SmallPageTableEntry {
2     asid: Option<u32>,
3     virtual_address: u32,
4     physical_address: u32,
5     permissions: AccessPermissions,
6 }
7
8 impl L2SmallPageTableEntry {
9     pub fn try_new(virtual_address: u32, asid: Option<u32>) → Option<Self> {
10         let current_index =
11             (0 .. PAGE_TABLE_SIZE as u32).find(|&i| unsafe { !USED_PAGES[i as usize] })?;
12         unsafe {
13             USED_PAGES[current_index as usize] = true;
14         }
15         let offset = current_index << PAGE_SIZE_BITS;
16
17         Some(L2SmallPageTableEntry {
18             asid,
19             virtual_address: virtual_address & !0xFFFF,
20             physical_address: BASE_ADDRESS + offset,
21             permissions: AccessPermissions::Full,
22         })
23     }
24
25     ...
}
```

Damit eine L2 Page Table Entry auch verwendet werden kann, muss sie “registriert” werden. Dabei überschreibt die ASID der Page Table Entry jene des Prozessors und setzt die L2 Page Table Entry in dem L2 Page Table welche der virtuellen Adresse entspricht. Abschließend werden noch eine *Data Synchronous Barrier* und eine *Instruction Synchronous Barrier* hinzugefügt, um zu garantieren, dass alle Operationen abgeschlossen werden.

Wenn die Page Table Entry nicht mehr verwendet wird, wird eine Fault Entry an deren Virtueller Adresse gesetzt und die entsprechende Region als nicht mehr verwendet markiert. Abschließend wird noch der Eintrag dieser Page Table Entry im TLB invalidiert, wobei die übrigen Einträge des TLB nicht verändert werden. Abschließend werden auch hier eine *isb* und eine *dsb* hinzugefügt.

```

○ ○ ○
src/internals/mmu/l2.rs

1 impl L2SmallPageTableEntry {
2 ...
3
4     pub fn register(&self) {
5         self.set_asid();
6
7         unsafe {
8             LEVEL2_PAGE_TABLE[<self.virtual_address as usize >> PAGE_SIZE_BITS] = self.into();
9
10            asm!("dsb", "isb");
11        }
12    }
13
14    pub fn unregister(&self) {
15        unsafe {
16            LEVEL2_PAGE_TABLE[<self.virtual_address as usize >> PAGE_SIZE_BITS] =
17                L2_FAULT_PAGE_TABLE_ENTRY;
18            USED_PAGES[(self.physical_address - BASE_ADDRESS) as usize >> PAGE_SIZE_BITS] = false;
19        }
20
21        self.invalidate_tlb();
22
23        unsafe {
24            asm!("dsb", "isb");
25        }
26    }
27
28    fn invalidate_tlb(&self) {
29        unsafe {
30            asm!("mcr p15, 0, {mva}, c8, c7, 1", mva = in(reg) (<self.virtual_address & !0xFFFF) |
31                self.asid.unwrap_or(0));
32        }
33    }
34    ...

```

Anwendung

Das Paging wird beim Zuweisen der Regionen der RAM verwendet. Dabei wird jede ausführbare Task in einer Region der RAM gespeichert, während der Stack sowie der Heap dieser Task in einer weiteren Region gespeichert wird.

Wenn eine neue Task erstellt wird, werden eine Code und eine Daten L2 Page Table Entry erstellt, woraufhin der Code in die Code Page kopiert wird. Anschließend werden beide Page Table Entries der Task hinzugefügt, der Status wird gesetzt sowie der Stack Pointer und der Program Counter. Abschließend wird noch der Heap initialisiert, mehr dazu jedoch später.

```
○ ○ ○                                     src/internals/tasksrs

1 pub fn create_task(&mut self, code: &[u8]) -> Option<usize> {
2     let task_id = self.task_with_state(TaskState::Terminated)?.id;
3
4     let code_page = L2SmallPageTableEntry::try_new(CODE_PAGE_LOCATION, Some(task_id as u32))?;
5     let data_page = L2SmallPageTableEntry::try_new(DATA_PAGE_LOCATION, Some(task_id as u32))?;
6
7     let dest = code_page.start() as *mut u8;
8     unsafe {
9         ptr::copy_nonoverlapping(code.as_ptr(), dest, code.len());
10    }
11
12    let task = self.task_mut(task_id);
13    task.code_page = code_page;
14    task.data_page = data_page;
15    task.state = TaskState::Ready;
16    task.context.sp = task.data_page.end();
17    task.context.pc = task.code_page.start();
18    task.allocator
19        .init(task.data_page.start() as usize, task.data_page.end() as usize - STACK_GUARD);
20    Some(task.id)
21 }
```

Hinzufügen von Tasks

Um auch verschiedene Tasks hinzuzufügen, wurde zu Beginn der Entwicklung des Kernels einfach eine Funktion verwendet, welche zum Scheduler hinzugefügt wurde und ausgeführt wurde. Nachdem das Kernel jedoch Features wie Paging integriert hatte, beschloss ich, dies außerhalb des Kernel Codes zu machen. Da ich bei meinem Betriebssystem leider nicht wie bei vielen anderen Betriebssystemen verschiedene Programme bei Runtime hinzufügen kann, war ich hier auf das Hinzufügen von Tasks bei Completetime limitiert.

Dabei wird jedes Programm einzeln kompiliert und das Binary anschließend in den *programs* Ordner eingefügt. Dieses wird durch ein *Procedural Macro* ausgelesen und die Binaries in den Code eingefügt. Durch *Procedural Macros* kann Code vor dem Kompilieren des Codes ausgeführt werden. Dieser kann dabei auf die Standard Library von Rust zugreifen und somit die Inhalte des *programs* Ordners auslesen und die kompilierten Programme direkt in das Programm einbetten.

Dabei wird zu Beginn der *programs* Ordner ausgelesen und anschließend in ein Array konvertiert, das die jeweiligen Inhalte der kompilierten Programme enthält. Anschließend werden diese durch die *quote* crate in das Programm als Array eingebettet.

```
○ ○ ○ include_programs/src/lib.rs

1 #[proc_macro]
2 pub fn include_programs(_input: TokenStream) → TokenStream {
3     let current_dir = env::current_dir().unwrap();
4     let program_dir = current_dir.join(PathBuf::from("kernel/programs"));
5
6     let program_files = fs::read_dir(program_dir)
7         .expect("No programs directory is present")
8         .flatten()
9         .filter_map(|file| {
10             let path = file.path();
11
12             if path.is_file() {
13                 let contents = fs::read(path).unwrap();
14                 return Some(contents);
15             }
16
17             None
18         })
19         .collect::<Vec<_>>();
20
21     let tokens: Vec<_> = program_files
22         .iter()
23         .map(|inner_vec| {
24             quote! {
25                 &[#(inner_vec), *]
26             }
27         })
28         .collect();
29
30     let expanded = quote! {
31         &[
32             #(
33                 #tokens
34             ),*
35         ]
36     };
37
38     expanded.into()
39 }
```

Das Array, welches vom Makro zurückgegeben wird, wird global gespeichert. Anschließend wird für jedes dieser Programme eine eigene Task erstellt.

○ ○ ○

src/main.rs

```
1 static PROGRAMS: &[&[u8]] = include_programs!();
2
3 #[no_mangle]
4 pub fn _start() {
5
6     ...
7
8     for program in PROGRAMS {
9         create_task(program);
10    }
11
12    ...
13 }
```

User Binary

Das User Binary verwendet hier ein etwas verändertes Linker Script wie jenes des Kernels. Dabei liegt hier die Position des Binaries bei `0x0` und hat eine maximale Länge von 4 KB. Dies kann je nach Programm eine Limitation sein, es wurde jedoch aus Mangel an Priorität keine Lösung hierfür umgesetzt.



user/boot/linker.ld

```
1 MEMORY {
2     ram (rwx) : ORIGIN = 0x0, LENGTH = 0x1000
3 }
4
5 ENTRY(_start)
6 SECTIONS {
7     .text : {
8         *(.text._start)
9         *(.text*)
10    } > ram
11
12    .bss : {
13        *(.bss*)
14    } > ram
15
16    .data : {
17        *(.data*)
18    } > ram
19
20    _end = .;
21 }
```

Strukturierung

Um verschiedene Funktionen im User Code verwenden zu können, müssen diese von einer Crate bzw. Library bereitgestellt werden. Dabei wird hier eine Konvention für die Benennung von Libraries eingehalten und die Crate somit “libfenix” genannt. Da es jedoch Code gibt, der vom Kernel wie von der “libfenix” Crate verwendet wird, wird deshalb noch eine “shared” Crate implementiert, die vor allem Structs zur Kommunikation der beiden bereitstellt.

Um diese Module in einem einzigen Repository zu enthalten, wird ein Cargo Workspace verwendet. Durch diesen können mehrere Crates zusammengefasst werden, wobei meist ein weiterer Vorteil darin besteht, dass die Dependencies somit dieselbe Version verwenden, was jedoch bei der folgenden Implementation nicht von Belang ist.

So können im Root *Cargo.toml* die verschiedenen Einstellungen des Workspace eingestellt werden, so können hier die verschiedenen “Member” des Workspaces konfiguriert werden, wobei der “resolver” angibt, wie verschiedene Features von Crates kompiliert werden, was jedoch hier keinen Einfluss hat. Unter anderem sind in der folgenden Workspace Konfiguration auch die “include_programs” Crate, welche das Makro für das Einbetten der Programme beinhaltet und die “user” Crate, welche für das Erstellen der verschiedenen Programme zuständig ist.

```
○ ○ ○                                     Cargo.toml

1 [workspace]
2 members = [
3     "libfenix",
4     "include_programs",
5     "user",
6     "kernel",
7     "shared",
8 ]
9 resolver = "2"
```

Heap

Es wurde in diesem Kernel nur ein minimaler Heap verwendet. Der Einfachheit halber wurde hier ein sog. BufferAllocator verwendet, welcher im Unterschied zu einem normalen Allocator kein Memory mehr freigeben kann. Dabei hat der Bufferallocator einen internen Pointer, welcher bei jeder Allocation den Pointer weiter schiebt, demnach aber nicht mehr bereits verwendetes Memory verwenden kann.

○ ○ ○

shared/src/allloc/heap.rs

```
1 pub struct BumpAllocator {
2     heap_start: usize,
3     heap_end: usize,
4     next: CriticalSection<usize>,
5 }
6
7 unsafe impl GlobalAlloc for BumpAllocator {
8     unsafe fn alloc(&self, layout: core::alloc::Layout) → *mut u8 {
9         let mut current = self.next.lock();
10        let alloc_start = align_up(*current, layout.align());
11        let alloc_end = alloc_start.saturating_add(layout.size());
12
13        if alloc_end > self.heap_end {
14            ptr::null_mut()
15        } else {
16            *current = alloc_end;
17            alloc_start as *mut u8
18        }
19    }
20
21    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: core::alloc::Layout) {}
22 }
```

Kernel

Dieser Bufferallocator wird in global gespeichert und das *global_allocator* Attribut verwendet. Durch dieses Attribut können Objekte der Standard Library wie das *Vec*, die ein Heap benötigen, verwendet werden. Dies gilt jedoch nur für den Kernel Code.

```
○ ○ ○                                     src/heap.rs

1 pub fn initialize() {
2     let allocator = &raw mut ALLOCATOR;
3
4     unsafe {
5         (*allocator).init(
6             &heap_start as *const usize as usize,
7             &heap_end as *const usize as usize,
8         );
9     }
10 }
11
12 #[global_allocator]
13 static mut ALLOCATOR: BumpAllocator = BumpAllocator::new();
14
15 extern "C" {
16     static heap_start: usize;
17     static heap_end: usize;
18 }
19
```

Die *heap_start* und *heap_end* Variablen werden hierbei von dem Linker Script gesetzt, wobei das Heap eine Größe von 4 KB hat.

```
○ ○ ○                                     boot/linker.ld

1 SECTIONS {
2     ...
3
4     . = ALIGN(4);
5     heap_start = .;
6     . += 4K;
7     heap_end = .;
8
9     ...
10 }
```

Libfenix

In der *libfenix* Library wird ebenfalls der Allocator definiert. Hierbei wird jedoch die Allocation sowie die Deallocation dem Kernel überlassen. Dabei hat jede Task einen eigenen Heap welcher ebenfalls einen BufferAllocator verwendet.

```
○ ○ ○ libfenix/src/alloc.rs

1 #[global_allocator]
2 static ALLOCATOR: Allocator = Allocator {};
3
4 struct Allocator {}
5
6 unsafe impl GlobalAlloc for Allocator {
7     unsafe fn alloc(&self, layout: core::alloc::Layout) -> *mut u8 {
8         let syscall = Syscall::Alloc { layout };
9         unsafe { syscall.call().unwrap().alloc }
10    }
11
12    unsafe fn dealloc(&self, ptr: *mut u8, layout: core::alloc::Layout) {
13        let syscall = Syscall::Dealloc { ptr, layout };
14        syscall.call();
15    }
16 }
17
```

Dieser Heap wird beim Erstellen einer Task initialisiert, wobei der Start hier dem Start der Daten Page entspricht, und das Ende am Ende der Daten Page liegt, wobei noch ein Stack Guard davon abgezogen wird, welcher dafür sorgt, dass die Daten des Stacks nicht überschrieben werden.

```
○ ○ ○ src/tasks.rs

1 pub fn create_task(&mut self, code: &[u8]) -> Option<usize> {
2
3     ...
4
5     task.allocator
6         .init(task.data_page.start() as usize, task.data_page.end() as usize - STACK_GUARD);
7
8     ...
9
10 }
```

Im Syscall wird die Allocation und Deallocation der verschiedenen Tasks gehandhabt. Dabei wird der Allocator der aktuellen Task genommen und durch diesen anschließend die Allocation vorgenommen, woraufhin der Pointer, welcher vom Allocator zurückgegeben wird, vom Syscall zurückgegeben wird.

```
1 Syscall::Alloc { layout } => {
2     let scheduler = scheduler();
3     if let Some(task) = scheduler.current() {
4         let ptr = unsafe { task.allocator.alloc(layout) };
5         return SyscallReturn::value(SyscallReturnValue { alloc: ptr });
6     }
7
8     SyscallReturn::none()
9 }
10 Syscall::Dealloc { ptr, layout } => {
11     let scheduler = scheduler();
12     if let Some(task) = scheduler.current() {
13         unsafe { task.allocator.dealloc(ptr, layout) };
14     }
15
16     SyscallReturn::none()
17 }
```


Projektmanagement

Im Projektmanagement sind all jene Unterlagen enthalten, die bei unserem Projekt gefordert waren. Dies beinhaltete unter anderem den Projektantrag, dieser wurde zu Beginn geschrieben und anschließend den zuständigen Professoren überreicht, die das Projekt dann genehmigten. Im Pflichtenheft sind all jene Funktionen des Projektes enthalten, die bei Projektabschluss funktionieren sollten. Das Gantt-Diagramm hingegen hilft beim Zeitmanagement und gibt Aufschluss, ob man im Zeitplan liegt.

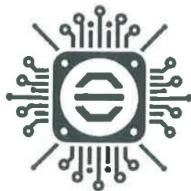
Projektantrag

Seite 1/2

Ansuchen für ein Maturaprojekt, Fassung September 2024

Projekttitel:	MicroOS	
Projektleiter:	Felix Salcher	
Klasse und Schuljahr:	5. BEL	Schuljahr 2024 / 2025

Logo des Projektes:



Kurze Beschreibung des Projektes:

Das Ziel dieses Projektes ist es, ein minimalistisches Betriebssystem bzw. ein Kernel zu schreiben, welches die Grundfunktionen wie Paging, Multitasking, CPU Exceptions implementiert. Hierbei wird die Platine selbst erstellt, welche unter anderem eine CPU beinhaltet, aber auch ein externes RAM-Modul, da der Prozessor auch mehr Prozesse ausführen sollte. Das Kernel wird in der Programmiersprache Rust geschrieben. Zudem sollten auf der Platine auch einige LEDs und Taster sein, um den Controller besser testen zu können.

ARM CORTEX A53

Benotungsrelevante Meilensteine des Projektes:

- Meilenstein 1: Platinenabgabe in PW... (1. Semester)
- Meilenstein 2: Booten einer minimalen Version (2. Semester)
- Meilenstein 3: Externe Kommunikation über GPIO Pins (2. Semester)
- Meilenstein 4: Paging, Multitasking, CPU Exceptions (2. Semester)
- Meilenstein 5: Abgabe des Projektes in PW... (2. Semester)

Abbildung 28. Projektantrag Seite 1

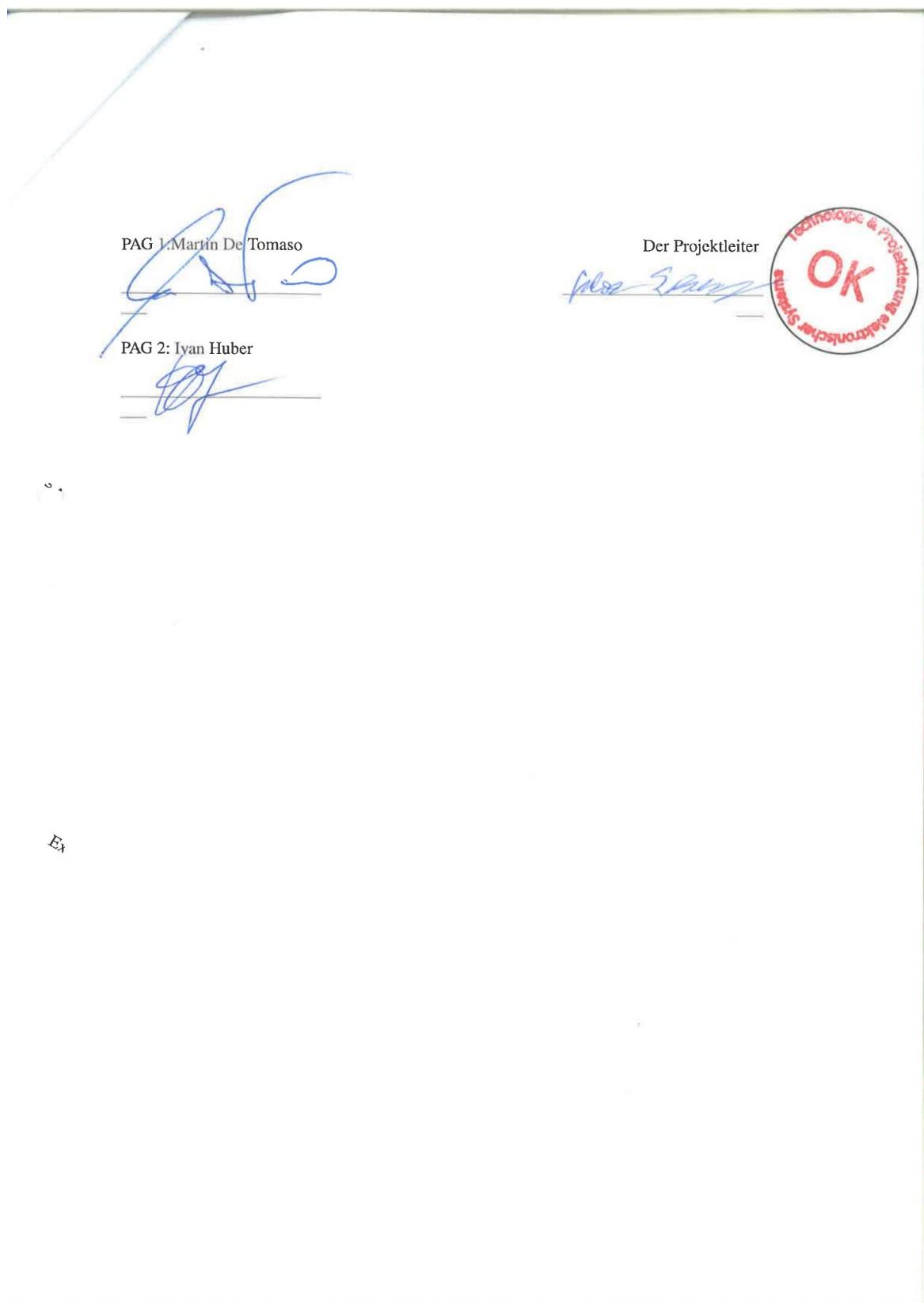


Abbildung 29. Projektantrag Seite 2

Pflichtenheft

Projektbeschreibung:

Das Ziel dieses Projektes ist es, ein minimalistisches Betriebssystem bzw. ein Kernel zu schreiben, welches die Grundfunktionen wie Paging, Multitasking, CPU Exceptions implementiert. Hierbei wird die Platine selbst erstellt, welche unter anderem eine CPU beinhaltet, aber auch ein externes RAM-Modul, da der Prozessor auch mehr Prozesse ausführen sollte. Das Kernel wird in der Programmiersprache Rust geschrieben. Zudem sollten auf der Platine auch einige LEDs und Taster sein, um den Controller besser testen zu können

Platine:

Auf der Platine sollte der Arm Cortex-A53 Controller vorhanden sein. Dieser verwendet ein externes RAM Modul, damit er mehr Speicher zur Verfügung hat, nicht nur einige Kb, welche im Controller selbst integriert sind.

Auf der Platine sollten zudem noch einige Pinheads vorhanden sein, um externe Sensoren oder andere Geräte anzuschließen, welche anschließend durch den Controller gesteuert werden können. Außerdem werden noch einige Taster und LEDs auf der Platine integriert, um die Funktionalität der Platine direkt testen zu können, ohne externe Geräte anzuschließen. Hierbei werden jeweils 3 Taster und LEDs verwendet. Zudem sollte die Platine noch eine LED enthalten, welche vom Controller nach dem Boot automatisch eingeschaltet wird, um zu zeigen, dass der Controller richtig gebootet wurde. Das Booten sollte hierbei über USB vonstatten gehen, genauer gesagt über USB-C.

Software:

Die Software wird ein Kernel beinhalten, welches die Aufgabe hat, alle Tasks zu managen, das Speicher-layout festzulegen sowie CPU Exceptions zu handeln. Die CPU Exceptions sollten so gehandelt werden, dass das Betriebssystem sich immer davon erholen kann und niemals dadurch abstürzen sollte, hierbei sollte nur das jeweilige Programm, welches die Exception hervorgerufen hatte crashen. Zudem sollte das Betriebssystem Paging implementieren, d.h. dass jeder Prozess nur auf einen bestimmten Teil im Speicher zugreifen kann und somit nicht den Speicher anderer Programme zu korrumptieren. Abschließend sollte das Betriebssystem noch in der Lage sein, mehrere Tasks gleichzeitig laufen zu lassen, entweder durch physische Threads oder durch das Freischalten einer bestimmten Zeit für jeden Prozess, in dem dieser ausgeführt werden kann. Außerdem sollte man über die Software noch die verschiedenen GPIO Pins steuern können, um mit der Außenwelt auch kommunizieren zu können.

Abbildung 30. Pflichtenheft Seite 1

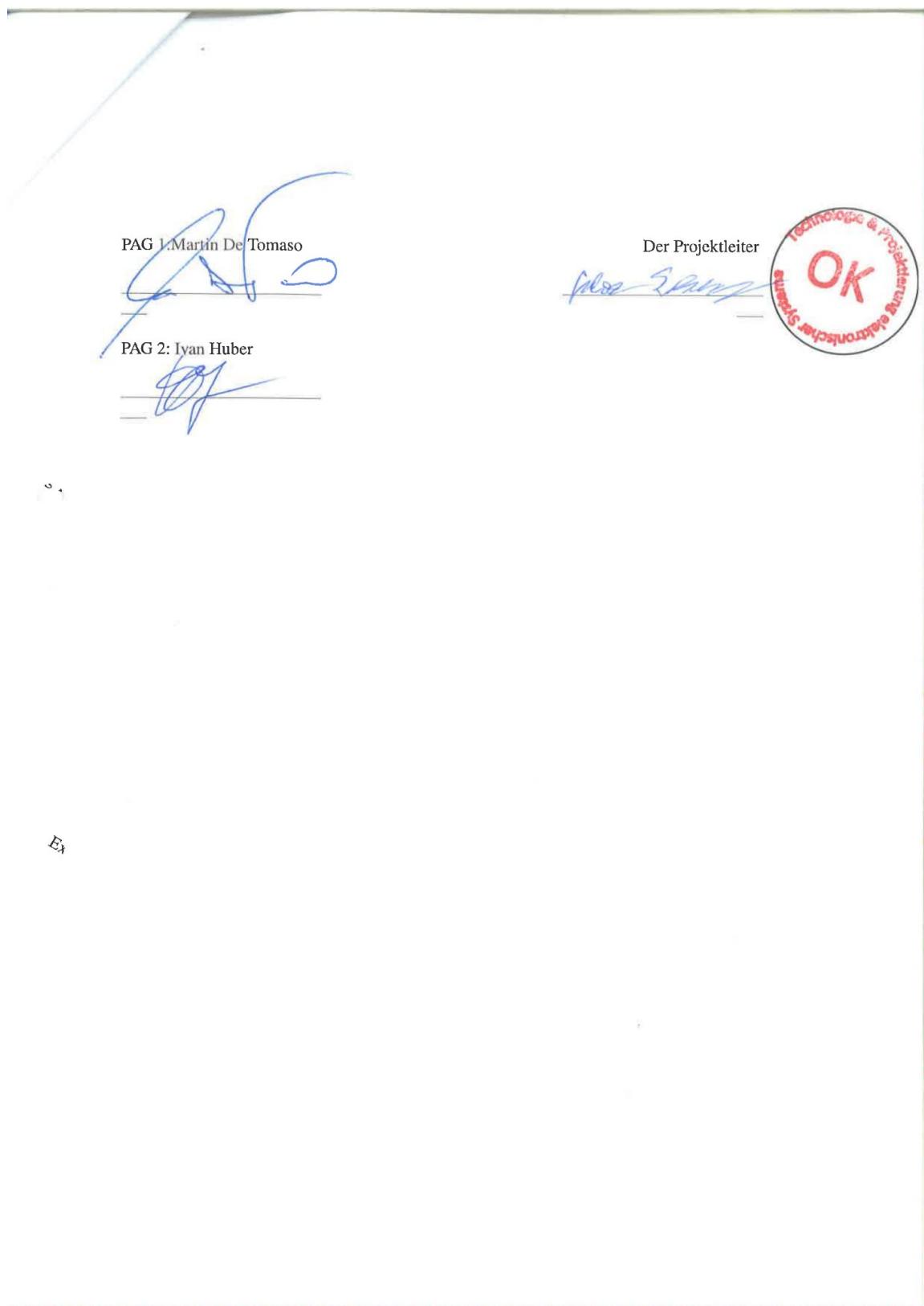


Abbildung 31. Pflichtenheft Seite 2

Gantt-Diagramm

Das Gantt-Diagramm wurde verwendet, um das Projekt in verschiedene Working Packages zu unterteilen. Jedem dieser Working Packages wird anschließend ein gewisser Zeitraum zugewiesen, in welchem es abgeschlossen werden sollte. Während der Entwicklung des Projektes trägt man dann bei jedem Working Package ein, ob es im vorgegebenen Zeitraum abgeschlossen wurde. Meist wird dies durch Farben wie Rot und Grün gekennzeichnet.

Dadurch bekommt man einen sehr guten Überblick, ob das Projekt im Zeitplan liegt.

Auf den folgenden Seiten ist das Gantt-Diagramm dargestellt.

Man kann dabei erkennen, dass ich besonders bei der Entwicklung der Hardware im Zeitplan zurücklag. Das liegt daran, dass ich erst im November mich entschieden hatte, vom AM62 auf den AM335 umzusteigen. Dies bedeutete, dass ich praktisch von Null anfangen musste, wodurch ich sehr im Zeitplan zurücklag.

Was man zudem erkennen kann ist, dass ich bei der Implementation des Paging ebenfalls sehr zurücklag. Dies liegt daran, dass ich zu Beginn die falsche Annahme traf, dass es leichter sei, zuerst mit dem Paging zu beginnen. Der Grund dafür ist, dass Paging meist erst bei Multitasking sinnvoll ist. Deshalb begann ich zuerst mit der Implementierung von Multitasking und programmierte später das Paging aus.

Projekt "MicroOS" - 5BEL – Salcher Felix

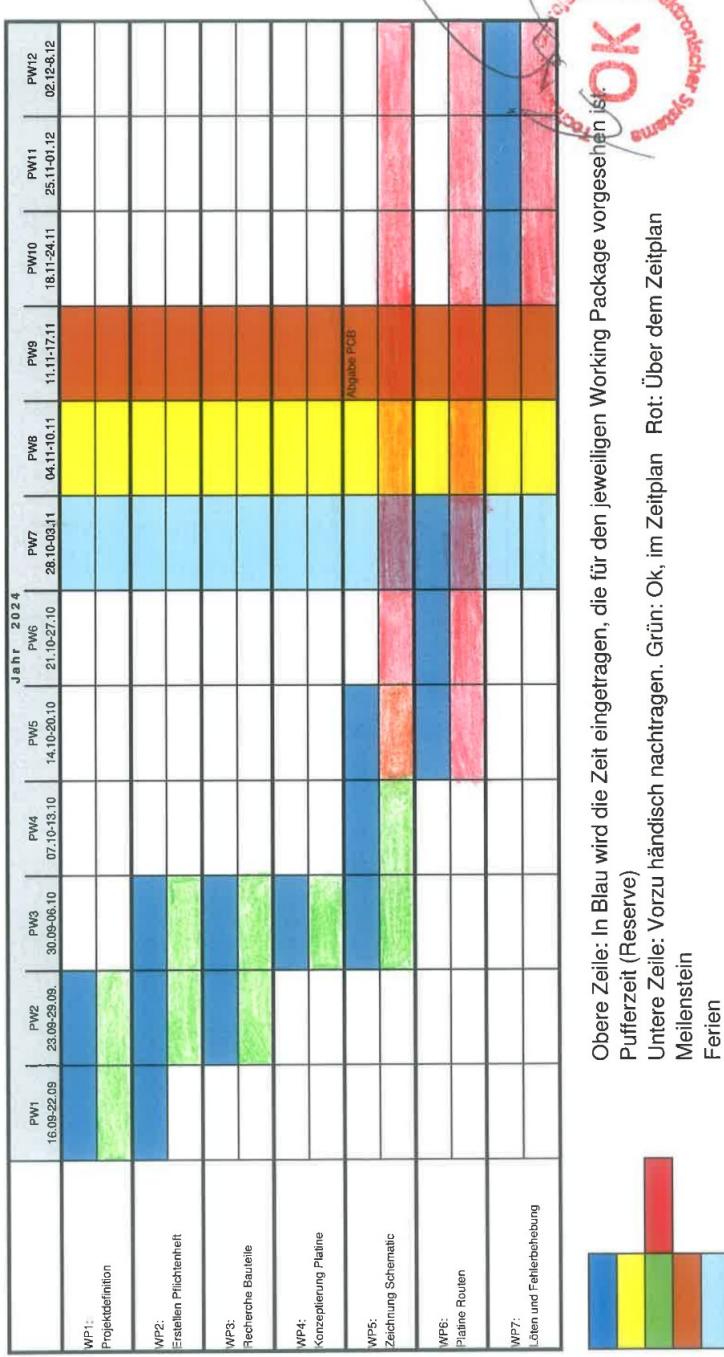


Abbildung 32. Gantt-Diagramm Seite 1

Projekt "MicroOS" - 5BEL – Salcher Felix

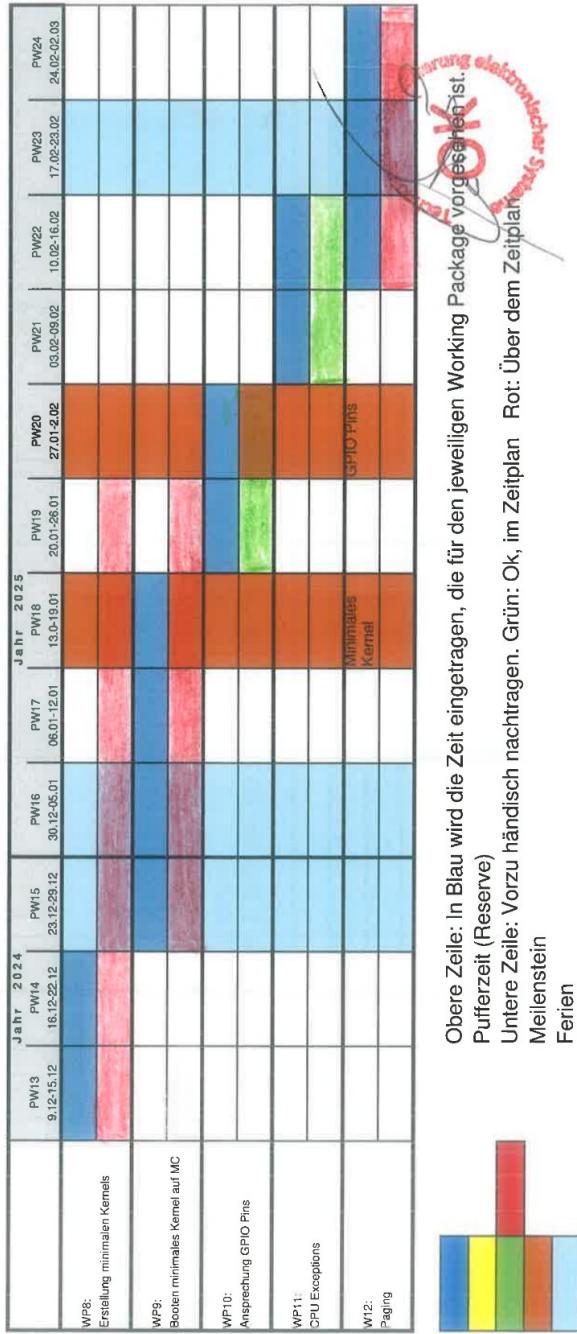
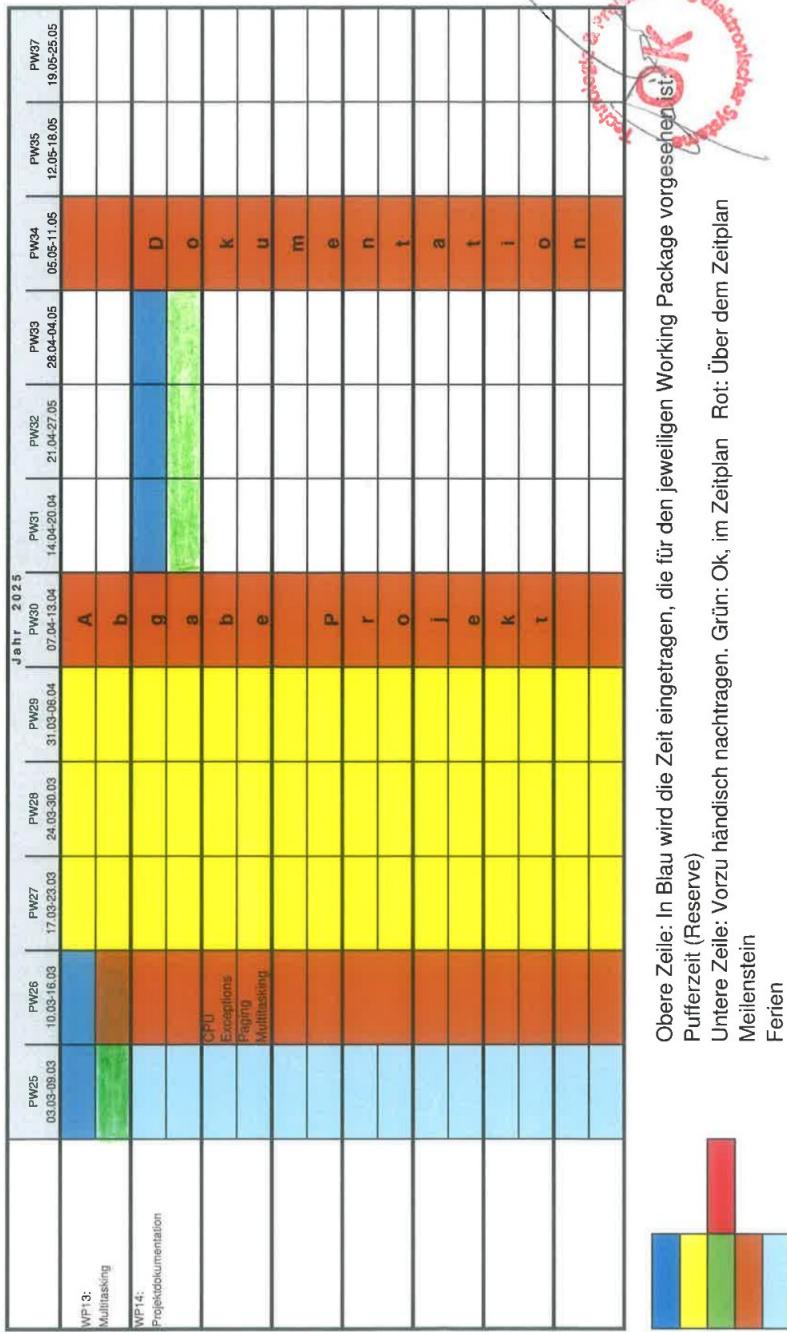


Abbildung 33. Gantt-Diagramm Seite 2

Projekt "MicroOS" - 5BEL – Salcher Felix



Obere Zeile: In Blau wird die Zeit eingetragen, die für den jeweiligen Working Package vorgesehen ist.
 Pufferzeit (Reserve)
 Untere Zeile: Vorzu händisch nachtragen. Grün: Ok, im Zeitplan Rot: Über dem Zeitplan
 Meilenstein
 Ferien

Abbildung 34. Gantt-Diagramm Seite 3

Kostenberechnung

In diesem Kapitel werden die verschiedenen Kosten des Projektes aufgelistet und berechnet, um einen Überblick darüber zu haben, wie viel das Projekt einer Firma gekostet hätte, mit allen Ausgaben eingerechnet.

Beraterstunden

Wenn Hilfe von externen Personen benötigt wird, müssen diese Berater ebenfalls in die Kosten mit einberechnet werden. Dabei wird die Stunde eines Beraters bei 40€ pro Stunde gerechnet.

Dabei hat mir nur mein Professor Huber Ivan bei der Entwerfung der Platine geholfen, andere Personen haben bei diesem Projekt nicht geholfen.

Datum	Stundenanzahl	Name Berater
29.11.2024	0.5	Huber Ivan
10.12.2024	0.25	Huber Ivan
09.01.2025	0.75	Huber Ivan

Gesamtstunden Berater: 1.5

Kosten externe Berater: 60€

Arbeitsstunden

Jede Woche wurden 5 Stunden im Labor an dem Projekt gearbeitet. Besonders im zweiten Semester wurde anschließend sehr intensiv auch zu Hause an dem Projekt weitergearbeitet. Dabei wurde in einigen Wochen bis zu 20-30 Stunden zu Hause gearbeitet. Insgesamt arbeitete ich somit 500 Stunden. Bei einem Stundenlohn von 40€, hätte die Entwicklung des Projektes meinen Arbeitgeber 10.000€ gekostet.

Platine

Die Platine wurde bei JLCPCB bestellt und dort auch direkt bestückt. Die Platine sowie deren Bestückung kostete am Ende mit Versand insgesamt 140€. Es wurde hierbei entschieden, aus Kostengründen nur zwei Platinen bestücken zu lassen, da dies pro Platine über 50€ kostet.

Es gilt hierbei jedoch zu beachten, dass jene Bauteile, welche auf der Platine verwendet werden, bereits in diesem Preis inbegriffen sind und auf einen Preis von 17.08€ kommen. Dies muss anschließend vom Preis der BOM abgezogen werden.

Product	File Name	Order Number	QTY	Unit Price	Ext.Price
1	6-layer Bare Rigid Printed circuit board	maturaprojekt_Y61	Y61	3	EUR €6.76
2	Rigid Populated printed circuit board	maturaprojekt_Y61	SMT025011561535-Y61	2	EUR €55.69
					Merchandise Total: EUR €131.68
					Shipping: EUR €12.57
					Merchandise Discount: -EUR €29.07
					Subtotal: EUR €115.18
					Import Taxes(22%): EUR €25.34
					Grand Total: EUR €140.52

Abbildung 35. Rechnung der Platine von JLCPC

BOM

Designator	Package	Quantity	Value	Unit Price(\$)	Price(\$)
R25,R26	R_0805	4	4k7	0.013	0.13
R21,R18,R16,R23,R11,R15,R22,R17,R24, ...	R_0805	17	100k	0.0009	0.05
U4	IC_TPS65217CRSLR	1	TPS65217CRSLR	2.23	4.46
L3,L2,L1	L_1008	3	2.2μH	0.167	0.167
D1	SMASeries_LTF	1	SMAJ5.0A	0.3	0.9
U1	NFBGA-324_L15.0-W15.0-R18-C18-P0.80-BL	1	AM3352BZCZ80	5.47	10.94
C77,C76	C_0805	2	18pF	0.048	0.48
C95,C4,C36,C31,C33,C34,C7,C10,C6,C32, ...	C_0805	18	10μF	0.003	0.17
C14,C15	C_0805	2	2.2μF	0.058	0.058
R27,R30,R29,R28	R_0805	4		0	0.01
C3,C86,C87,C72,C52,C94,C91,C85,C51,C49, ...	C_0805	20	100nF	0.29	0.58
U3	CRYSTAL-SMD_4P-L3.2-W2.5-BL	1	CS325S24000000ABJT	0.95	0.95
C56,C61,C58,C59	C_0805	4	1μF	0.039	0.59
R36	R_0805	1	10k, 1%	0.2	0.2
U8	TF-SMD_TF-PUSH	1	TF PUSH	0.05	0.1
D3	LED_0805	1	LED	0.17	0.85
SW1,SW2	SW_1825910-6-4	2	1825910-6	0.132	1.32
U2	UX60SC-MB-5ST	1	UX60SC-MB-5ST(80)	0.595	1.19
C50,C78,C66,C63,C69,C65,C60,C42,C64, ...	C_0402	32	100nF	0.0012	0.08
R43,R47,R49,R44,R45,R50	R_0805	6		33	0.009
R40,R39,R46,R41,R37,R20,R38,R42	R_0805	8	10k		0.009
U6	SC-70-5_L2.1-W1.3-P0.65-LS2.1-BR	1	74LVC1G07	0.1	0.5
FB1	L_0805	1	150R800mA	0.19	0.95
R14	R_0805	1	1M	0.013	0.13
R31	R_0805	1		470	0.013
C1	C_0805	1	4.7μF	0.048	0.48
J1	PinHeader_2x09_P2.54mm_Vertical	1	Conn_01x18_Pin	0.23	1.15
				Gesamt(\$)	27.015
				Gesamt(€)	24.06

Abbildung 36. Bom

Gesamtausgaben

Zu den Gesamtausgaben muss zudem noch das Beaglebone Black hinzugerechnet werden, welches für die Entwicklung des Betriebssystems verwendet wurde, da meine Platine ja nicht funktioniert hatte. Der Preis des Boards beträgt hierbei 53.77€.

Beaglebone Black	53.77€
Bauteile	7€
Platine	140.52€
Arbeitsstunden	10.000€
Gesamt	10.201,29

Wie man erkennen kann, liegen die meisten der Ausgaben bei den Arbeitsstunden, die Kosten der Hardware können hierbei vernachlässigt werden.

Abbildungsverzeichnis

Im folgenden Abschnitt sind alle Bilder und deren Quellen angeführt. Wenn ein Bild selbst erstellt wurde, wurde die Quelle nicht angeführt.

■ Abbildung 1. Blockschaltbild der Platine	Seite 15
■ Abbildung 2. AM335 Prozessor https://www.ti.com/product/AM3358	Seite 16
■ Abbildung 3. Prozessor Schematic	Seite 18
■ Abbildung 4. PSU Schematic	Seite 20
■ Abbildung 5. PSU AM335 Schematic	Seite 21
■ Abbildung 6. SD-Karte Schematic	Seite 23
■ Abbildung 7. Verwendete Boot Konfiguration TRM Seite 5035	Seite 24
■ Abbildung 8. Boot Pins Schematic	Seite 25
■ Abbildung 9. Gesamte Platine	Seite 27
■ Abbildung 10. Platine Top Layer	Seite 28
■ Abbildung 11. Platine GND Layer	Seite 29
■ Abbildung 12. Platine Layer 3	Seite 30
■ Abbildung 13. Platine Layer 4	Seite 31
■ Abbildung 14. Platine POWER Layer	Seite 32
■ Abbildung 15. Platine Bottom Layer	Seite 33
■ Abbildung 16. QR Code Github Repository	Seite 36
■ Abbildung 17. ARMV7a CPU Modi https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Register	
■ Abbildung 18. ARMV7a Register https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Registers/Registers	Seite 40
■ Abbildung 19. ARMV7a Exception Vektor https://developer.arm.com/documentation/den0013/d/Exception-Handling/Exception-priorities	Seite 42
■ Abbildung 20. ARMV7a L1 Translation Table https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/First-level-address-translation	Seite 44
■ Abbildung 21. ARMV7a L1 Page Table Entry https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/First-level-address-translation	Seite 45
■ Abbildung 22. ARMV7a L2 Page Table Entry https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/Level-2-translation-tables	Seite 45
■ Abbildung 23. ARMV7a Translation Lookaside Buffer https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/The-Translation-Lookaside-Buffer	
■ Abbildung 24. AM335 I2C TXTRSH TRM Seite 4595	Seite 46
■ Abbildung 25. AM335 Timer TCRR Register TRM Seite 4442	Seite 65
■ Abbildung 26. AM335 Timer Intervall Reichweite TRM Seite 4437	Seite 72
■ Abbildung 27. Round Robin Scheduling	Seite 72
■ Abbildung 28. Projektantrag Seite 1	Seite 92
■ Abbildung 29. Projektantrag Seite 2	Seite 115
	Seite 116

■ Abbildung 30. Pflichtenheft Seite 1	Seite 117
■ Abbildung 31. Pflichtenheft Seite 2	Seite 118
■ Abbildung 32. Gantt-Diagramm Seite 1	Seite 120
■ Abbildung 33. Gantt-Diagramm Seite 2	Seite 121
■ Abbildung 34. Gantt-Diagramm Seite 3	Seite 122
■ Abbildung 35. Rechnung der Platine von JLCPC	Seite 125

Technical Reference Manual AM335: <https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf>

Fazit

Mit der Entwicklung des Betriebssystems bin ich sehr zufrieden. Ich konnte die meisten Features implementieren und zudem einige nicht vorgesehene Features wie zum Beispiel I2C oder das Hinzufügen von mehreren Programmen.

Rückblickend kann ich jedoch sagen, dass ich viele Dinge anders gemacht hätte. Dabei wäre zum Beispiel die Informierung und Strukturierung des Projektes. Ich wünschte, ich hätte zu Beginn eingehendere Recherchen zum Prozessor gemacht und mich nicht erst zwei Monate nach Beginn des Projektes für einen anderen entschieden.

Ein weiteres Problem war, dass ich bei der Programmierung des Kernels jeden Teil einzeln durchdacht und dann implementiert habe, und mich nicht zu Beginn intensiv mit der Struktur des Kernels auseinandergesetzt habe. So wären zum Beispiel Synchronisationsmethoden zwischen verschiedenen Prozessen sehr wichtig zu op gewesen, da diese ein jedes Betriebssystem benötigt. Zudem hätte ich mich auch zu Beginn über die ARM Architektur besser informieren sollen, was besonders bei der Programmierung des Timers ein Problem war, wo ich zwei Wochen versucht habe, ihn hinzubekommen. Am Ende war das Problem dabei, dass der Stack Pointer des IRQ Modus nicht gesetzt wurde. Durch ein bisschen Recherche konnte ich dieses Problem glücklicherweise finden.

Ich bin sehr froh, dass ich dieses Projekt für mein Maturaprojekt gewählt habe. Durch das Projekt wurde mein Interesse für Betriebssysteme geweckt und ich konnte eine Menge über deren Funktionsweise lernen. Unter anderem konnte ich so Heap und Stack vollständig verstehen, die Zusammenhänge zwischen Kompilieren und Linken eines Programmers erkennen und meine Fähigkeiten in Rust weiterentwickeln.