

ENTWICKLUNG EINES MINIBETRIEBSSYSTEM AUF BASIS EINES ARM926 PROZESSORS

BACHELORARBEIT ZUR ERLANGUNG DES BACHELOR OF SCIENCE FÜR ANGWANDTE INFORMATIK

an der Fachhochschule für Technik und Wirtschaft Berlin,
Fachbereich Wirtschaftswissenschaften II,
Studiengang Angewandte Informatik

1. Betreuer: Prof. Dr. Frank BAUERNÖPPEL
2. Betreuer: Prof. Dr. Burkhard MESSER

Eingereicht durch: Christopher KRUCZEK

28. Juli 2013

Inhaltsverzeichnis

1. Einleitung	2
2. Anforderungskatalog	4
2.1. Einleitung	4
2.2. Interruptquellen und Interrupthandler	5
2.3. Vectored Interrupt Controllor	5
2.4. Geräte	5
2.4.1. Timer	5
2.4.2. Serielle Schnittstelle - UART	6
2.4.3. Speicher	6
2.5. Threadmanagment	6
2.6. qemu	7
3. Vorstellung ARM926EJ-S Prozessor	8
3.1. Einleitung	8
3.2. RISC- vs CISC-Prozessoren	9
3.3. ARM926EJ-S	10
3.4. Register	10
3.5. Prozessor Modus	11
3.6. Interrupt-Controller	12
4. Entwurf	15
4.1. Einleitung	15
4.2. Geräte	16
4.3. Interrupt-Controller	16
4.4. Threads	17
4.5. Kernel	18

5. Konkretisierung des Entwurf	21
5.1. Einleitung	21
5.2. Startmechanismus	21
5.3. Interrupt-Controller	23
5.4. Interrupt-Service Routinen	25
5.5. Syscalls	26
5.6. Threadmanagment	26
5.6.1. Threadlayout	26
5.6.2. Scheduling	28
6. Implementation	30
6.1. Einleitung	30
6.2. Entwicklungsumgebung	30
6.3. Startmechanismus	31
6.4. Interrupt-Controller	35
6.5. Interrupt Service Routinen	37
6.6. Syscalls	37
6.7. Prozessmanagment	38
6.7.1. RAM-Disk	39
6.7.2. MopS Loader	40
6.7.3. Prozess-Layout	41
6.7.4. Prozess Generierung	44
7. Fazit	45
Anhang	45
A. Implementation	46
A.1. RAM-Disk Maker	46
B. Werkzeuge	49
B.1. Einleitung	49
B.1.1. arm-none-linux-gnueabi-as	49
B.1.2. arm-none-linux-gnueabi-gcc	49
B.1.3. arm-none-linux-gnueabi-ld	49
B.1.4. arm-none-linux-gnueabi-objcopy	50
B.1.5. arm-none-linux-gnueabi-objdump	50
B.1.6. make	50

Tabellenverzeichnis

3.1. Unterschiedliche ARM Versionen	8
3.2. Vergleich RISC vs. CISC	9

Abbildungsverzeichnis

3.1. ARM926EJ-S Register	11
3.2. VIC Register	13
4.1. MopS Überblick	15
4.2. Interrupt-Controller	16
4.3. Thread-Layout im RAM	17
4.4. Erstellung der RAM-Disk	18
4.5. Kernel - Überblick	18
4.6. Scheduler	19
4.7. Round-Robin Verfahren - Schematisch	20
5.1. Kernel-Image Version 1	21
5.2. Erzeugen der Sprungtabelle	22
5.3. Vectored Interrupt Controller	24
5.4. Konfigurationsbeispiel VIC	24
5.5. Interrupt Service-Routinen Timer & UART0	25
5.6. SWI-Handler	26
5.7. Thread-Image	27
5.8. RAM-Disk	27
5.9. RAM-Disk	28

Quellcode-Ausschnitte

6.1. Laden der Kerneldatei in qemu	31
6.2. Linker-Datei	32
6.3. Startup-Datei	32
6.4. Sprungtabelle erstellen I	33
6.5. Sprungtabelle erstellen II	33
6.6. Stack erstellen I	34
6.7. Stack erstellen II	34
6.8. Interrupt-Handler	35
6.9. VIC	35
6.10. VIC Mapping	36
6.11. VIC Konfigurations Beispiel	36
6.12. UART0 ISR	37
6.13. Software Interrupt Handler	37
6.14. Beispiel Programm	39
6.15. Binär Kopie vom Beispielprogramm	39
6.16. RAM-Disk Headerdatei	39
6.17. RAM-Disk C-Datei	40
6.18. MopS Loader	40
6.19. Prozess-Layout	41
6.20. Prozess-Layout erstellen	43
6.21. Prozess Generierung	44
A.1. RAM-Disk Maker	46
B.1. ARM-Assembler mit Optionen für ARM926	49
B.2. C/C++ Compiler	49
B.3. Linker mit Link-File 'link.ld'	50
B.4. Objektkopie in Binärformat	50

B.5. Objdump einer Objektdaten	50
B.6. make-File mit Hauptabhängigkeiten	51

Einleitung

»Der Entwurf eines Betriebssystems erfordert eher ein ingenieurmäßiges Vorgehen, als ein exaktes wissenschaftliches. Es ist schwieriger, klare Ziele zu definieren und diese zu erreichen.«[os]

Mit dieser Aussage leitet Tanenbaum das Thema der Entwicklung eines Betriebssystems ein. Und genau mit dieser Frage soll diese Bachelorarbeit eingeleitet werden. Was sollen also die Ziele dieser Bachelorarbeit sein?

Das Betriebssystem, das im folgenden vorgestellt wird, trägt den Akronymnamen **MopS** - Mini Operating System und soll seinen Haupteinsatzzweck im Lehrbereich der HTW-Berlin, für den Studiengang Angewandte Informatik und Wirtschaftsinformatik, finden.

Die Entwicklung eines Betriebssystems bringt viele Schwierigkeiten und Herausforderungen mit sich. Eine der Schwierigkeiten ist die Handhabung mit Embedded Systems. Hier zeigt **MopS** wie man mit einem verhältnismäßig kleinen Prozessor eine stabile Lösung erarbeiten kann. Eine weitere Schwierigkeit in diesem Bereich stellt die Benutzung der vorhandener Hardware dar. Das heißt z.B. wie wird der Timer konfiguriert, an welche Stelle im RAM muss er geladen oder wie können die Ticks behandelt werden. Weiterhin werden Komponenten wie die serielle Schnittstelle zwischen Tastatur und Monitor beleuchtet. Hier wird gezeigt, wie man eine Eingabe von der Tastatur abfangen kann, welches Hardwareteil dazu konfiguriert werden muss und wie die Daten auf dem Monitor angezeigt werden können. Aber auch der Punkt der Interrupts kommt in diesem Konzept nicht zu kurz. Was bedeuten Interrupts? Wie müssen sie konfiguriert werden? Welche Interrupthandler werden benötigt?

Neben all diesen Aspekten stellen Embedded Systems wie z.B. Mobiltelefone, Drucker, Kaffeemaschinen, Tablets den Entwickler vor große Herausforderungen, was die Thematik Prozessmanagement und Scheduling angeht. Deshalb werden diese Aspekte besonders beleuchtet. Das bedeutet, es gibt tiefe Einblicke in das Thema - Laden eines Prozesses -, - Starten eines Prozesses -, - Wechsel zwischen den Prozessen - und vieles mehr. Um der nächsten Generation von Informatikern einen leichten Einstieg in dieses Thema zu bieten, ist diese Bachelorarbeit entstanden. Sie

dient als Anschauungsmaterial und beschäftigt sich mit den Grundlagen der Betriebssystementwicklung in Embedded Systemen. Zudem soll die Arbeit den zukünftigen Projektteilnehmern die Angst vor der Betriebssystementwicklung nehmen.

Im Rahmen des Projektes **FOCOS - Family of configured operating systems**, für das Sommersemester 2013 begleitet durch Prof. Dr. Messer, soll diese Bachelorarbeit als Basis zur Weiterentwicklung von neuen Komponenten dienen.

Da dieses Projekt im Rahmen einer Lehrveranstaltung als Anschauungsmaterial dienen soll, wird besonderer Wert auf Quellcode und Grafische Untermalung im Entwurf gelegt. Zum Abschluss noch ein Zitat von Fernando Corbató, einem der Entwickler von CTSS¹ und MULTICS²

»... Meine Definition von Eleganz ist das Erreichen einer gegebenen Funktionalität mit einem Minimum an Mechanismen und einem Maximum an Klarheit.«[os]

Dieses Zitat führt uns zu einem weiteren wichtigen Punkt in dieser Arbeit. **MopS** ist auf Einfachheit und Überschaubarkeit ausgelegt. Das heißt es wurde Wert darauf gelegt keine unnötigen Sachen zu implementieren.

¹Compatible Time Sharing System

²Multiplexed Information and Computing Service

Anforderungskatalog

2.1. Einleitung

Moderne Betriebssysteme bestehen aus einer Vielzahl an Funktionen und bieten ein umfangreiches Portfolio an Möglichkeiten, jedoch muss bei der Entwicklung von Betriebssystemen im Embedded System Bereich darauf geachtet werden, nur die wichtigsten Komponenten zu implementieren und diese womöglich noch hochperformant zu gestalten.

In jedem Betriebssystem und auch in **MopS** gibt es verschiedene Stellen die eine Erklärung und darauffolgend eine Entwicklung bedarfen. Auch in **MopS** gibt es einen Startmechanismus, viel mehr unter *Booten* bekannt. Hier wurde jedoch begründet der Begriff Startmechanismus gewählt da kein wirkliches 'hochfahren' statt findet.

Des weiteren gibt es auch Interrupts und Quellen die Interrupts aussenden können. **MopS** stellt in der aktuellen Fassung nur zwei Interruptquellen zur Verfügung, zum einen den Timer und die Serielle-Schnittstelle(speziell die Tastatur). Diese Quellen können normale und Fast Interrupt Requests aussenden. Je nach Interrupt Request gibt es unterschiedliche Behandlungsmethoden, die sogenannten Interrupthandler. Natürlich gibt es auch unterschiedliche Herangehensweisen wie Interrupts behandelt werden können, **MopS** erledigt diese Aufgabe mit einem Vectored Interrupt Controller.

Neben diesen Faktoren gibt es aber auch weitere Mechanismen in einem Betriebssystem die auch **MopS** prägen. Eines davon sind die Trap-Handler, also Methoden die von einer Trap-Instruktion aufgerufen werden und somit Kernel-Methoden benutzbar machen.

Damit ist aber noch kein Betriebssystem funktionsfertig. Es fehlen noch zwei wichtige Mechanismen. Zum einen ist dass, das Threadmanagement also die Verwaltung von User-Programmen. Verwaltung ist hier der Oberbegriff für die Teilmechanismen des Ladens, Starten und der Wechsel zwischen den Threads.

Da **MopS** voranging im Bereich der Handhelds eingesetzt werden soll ist es für die Arbeit nicht relevant eine MMU - Memory Management Unit zu benutzen. Für eine einwandfreie Threadverwaltung ist natürlich auch ein funktionierendes Scheduling notwendig. Für **MopS** wurde sich entschieden das Round-Robin Verfahren zu benutzen.

2.2. Interruptquellen und Interrupthandler

Ein Prozessor muss im Falle eines Interrupts, z.B. ein Reset des Prozessors, IRQ, FIQ, Prefetch Abort, Software Interrupt oder eine undefinierte Aktion, in der Lage sein diese adäquat zu behandeln. Die meist verwandte Vorgehensweise ist es eine Tabelle im Speicher zu definieren, die genau für diese Zwecke Routinen bereitstellt[**archManI**].

Tritt nun eine Interrupt auf, gibt es eine vordefinierte Reihenfolge an Aktionen, die der Prozessor ausführt. Damit immer valide Routinen zur Behandlung der Interrupts in dieser Tabelle stehen, muss jene während der Initialisierung des Systems angelegt werden. Was **MopS** also benötigt ist ein System zur Behandlung und Konfiguration von Interrupts und Interruptquellen.

2.3. Vectored Interrupt Controllor

Dieses System stellt ein Hardware Interface zum Interruptsystem des Prozessors dar. In Systemen mit klassischen Interrupt Controllern muss die Software sowohl die Herkunft des Interrupt Request, als auch die Interrupt Service Routine ermitteln. Diese Aufgabe übernimmt der VIC nun komplett selbständig. Die korrekte Behandlung von sowohl Interrupt Requests(IRQ) als auch Fast Interrupt Requests(FIQ) stellen eine wichtige Rolle in Betriebssystemen dar.

Hier gibt es zwei unterschiedliche Mechanismen IRQ/FIQ's zu behandeln:

Non-Vectored Controller - Diese Systeme müssen wissen, woher der Request kommt und wo die Routine zur Behandlung des Requests liegt.

Vectored Interrupt Controller - Diese Controller vereinen beide oben genannten Fakten, denn sie werden initial mit den Prioritäten der Requests und den zugehörigen Routinen gefüllt und können dann zur Laufzeit die passende Routine direkt zurückgeben.

Bei **MopS** fiel die Entscheidung darauf, die 2. Variante, den Vectored Interrupt Controller, zu implementieren. Die Vorteile werden noch genauer erläutert.

2.4. Geräte

Im folgenden wird die Hardware definiert mit der **MopS** arbeitet. Zum einen sind das Timer und zum anderen die Serielle Schnittstelle.

2.4.1. Timer

Timer stellen wichtige Schlüsselfaktoren in Betriebssystemen dar. Ihr Hauptzweck ist die periodische Behandlung von Ereignissen und das ansteuern des Schedulers.

Der ARM926 stellt vier Timer zur Verfügung, welche sich auf unterschiedlichste Art und Weise konfigurieren lassen. Die verfügbaren Timer sind hier mit *Timer0* - *Timer3* zu benennen[**archManI**]

]. Da es für **MopS** ausreicht nur auf einen Timer zurück zu greifen wird später nur noch von *Timer0* die Rede sein.

2.4.2. Serielle Schnittstelle - UART

Das *Universal Asynchronous Receiver Transmitter* Interface bietet die Möglichkeit der seriellen Datenübertragung auf Mikrocontroller. Mittels dieser Schnittstelle ist es unter anderem möglich, die Tastatureingaben abzufragen und Daten auf dem Monitor anzuzeigen.

Sicherlich gibt es hier noch speziell dafür ausgelegte Geräte, wie z.B. *KMI*, für Eingaben von der Tastatur oder das *Character LCD Display*, um Daten auf dem Monitor darzustellen, jedoch ist das UART Interface für diese Zwecke ausreichend. Im Folgenden findet ausschließlich die technische Bezeichnung *UART0* Verwendung.

2.4.3. Speicher

Der Speicher von **MopS** soll als ein Ganzes betrachtet werden. Es gibt keine MMU und kein Swapping da keine Festplatte zur Verfügung gibt.

Die MMU, Memory Management Unit, spielt in vielen Betriebssystemen eine große Rolle, da sie für die Virtuallisierung von Speicher zuständig ist und somit die Möglichkeit bietet Threads in getrennten Speicherbereichen zu kontrollieren. Aufgrund der Tatsache das in Embedded Systemen wie Handhelds fast nie MMU's existieren wird auch **MopS** darauf verzichten.

Da keine Festplatte vorhanden ist und es nicht üblich ist in Handheld-Geräten zu swappen, wird dieser Punkt auch ignoriert.

2.5. Threadmanagment

Eine weitere wichtige Anforderung an **MopS** ist die Verwaltung von Threads. Hierbei spielt es eine wichtige Rolle wie die Threads vorbereitet werden, wie sie geladen werden und wie der allgemeine Mechanismus des Umschalten zwischen den Threads stattfindet.

Da eine Umschaltung nach gewissen Kriterien passieren muss ist es relevant einen Scheduler zu entwickeln. Da ein Scheduler auf verschiedene Art und Weisen entwickelt werden kann wurde sich auf konservative Methode beschränkt. Diese Methode nennt sich Round-Robin. Die Entscheidung diesen Mechanismus zu wählen fiel deshalb weil in Handheld-Geräten keine hochrangigen Prioritäten wie in einem Echt-Zeit-System beachtet werden müssen.

Für **MopS** wurde definiert das es nur drei Threads maximal geben darf. Das sind Scheduler, und zwei User-Programme.

2.6. qemu

Neben einer Umgebung für die Entwicklung, war es auch notwendig den Code auszuführen. Da keine echte Hardware zur Verfügung stand, auf welcher **MopS** getestet werden konnte, musste eine Emulationsumgebung benutzt werden, hier fiel die Wahl auf *qemu*¹. *qemu* ist eine Open Source Software zur Emulation und Virtualisierung von Hardware und Geräten. Die Entscheidung fuer *qemu* viel aufgrund der weit verbreiteten Benutzung und durch die Unterstützung der ARM-Prozessoren.

Ein weiterer Grund für die Wahl einen Simulator zu benutzen ist der Fakt das er einen ideale Umgebung bereitstellt. Das heißt das keine Seiteneffekte wie z.B. Spurious Interrupts, also Interrupts ohne erkennbarer Quelle, auftreten.

¹http://wiki.qemu.org/Main_Page[Letzter Zugriff 25.06.2013]

Vorstellung ARM926EJ-S Prozessor

3.1. Einleitung

Die Firma ARM bietet eine breite Palette von Prozessoren, hierbei ist zu sagen das im Laufe der Zeit verschiedene Versionen, wie ARMv1-ARMv8, in Betrieb waren. Diese Versionen beziehen sich nicht auf einen speziellen Prozessor sondern definieren eine Spezifikation auf deren Basis ein Prozessor, wie für diese Bachelorarbeit der ARM926EJ-S in der Version ARMv5. Für die Bachelorarbeit wurde die ARMv5 gewählt weil diese Prozessoren in vielen Handheld verbaut werden und im gegenteil dazu die Rechenkraft eines ARMv7 nicht benötigt wurde.

Version	Beispielprozessor	Beispielverwendung
ARMv1 (1985)	ARM1	BBC Master
ARMv2 (1986)	ARM2, ARM3	Acorn-Archimedes
ARMv3 (1991)	ARM6, ARM7	Apple Newton, RISC PC
ARMv4 (1995)	ARM7TDMI, ARM8	Gameboy Advanced, Nintendo DS
ARMv5 (1997)	ARM7EJ, ARM926EJ-S	Palm Tungsten
ARMv6 (2002)	ARM11, ARM-Cortex-M0	nvidia, Texas Instruments
ARMv7 (2004)	ARM-Cortex-M1	nvidia, Texas Instruments
ARMv8 (2014)	ARM Cortex-A50	Mobilefunkgeräte, Tablets

Quelle:<https://de.wikipedia.org/wiki/ARM-Architektur#Modelle>, Letzter Aufruf: 24.07.2013

Tabelle 3.1.: Unterschiedliche ARM Versionen

3.2. RISC- vs CISC-Prozessoren

Der Begriff ARM bedeutet *Advanced RISC Machine*. Hier muss jedoch ein weiterer Begriff herausgezogen werden, RISC. RISC bedeutet *Reduced Instruction Set Computer*, der Begriff *Reduced* bezieht sich jedoch nicht auf einen kleineren Instruktionssatz sondern mehr auf die Komplexität der Instruktionen selbst. Die Instruktionen bei einer RISC Maschine sind wesentlich einfacher als die einer CISC. So ist es deshalb möglich das Chipdesign zu vereinfachen.. Durch das einfachere Chipdesign können mehr Register auf den Chip gebracht werden und die Performance von Operationen ist höher. Die Daten können in Register geladen und die Performanceintensiven Speicherzugriffe reduziert werden.

Im Gegenteil dazu stehen die CISC Maschinen - *Complex Instruction Set Computer*. Diese Familie der Computer ist die wohl am weitesten verbreitete Technologie am Markt. Chipdesigner wie Intel und AMD bauen zum großteil diese Architekturen. Ein CISC hat im Gegenteil zu einem RISC ein weitaus kleineren Instruktionssatz, aber dafür ist die Komplexität höher. Dadurch wird erreicht das mit weniger Befehlen umfangreiche Operationen durchgeführt werden können. Der Nachteil dabei ist jedoch das die Performance, der Befehlsausführung, geringer als bei einem RISC ausfallen kann.

	RISC	CISC
CPU Zyklen pro Instruktion	wenig Zyklen pro Instruktion	mehrere Zyklen
Komplexität der Instruktionen	wesentlich geringer	hoch bis sehr hoch
Umfang Instruktionssatz	hoch	gering
Instruktionsgeschwindigkeit	hoch bis sehr hoch	gering
Verwendung	Smartphones, Tablets und andere Geräte die wenig Energie verbrauchen sollen.	Computer

Tabelle 3.2.: Vergleich RISC vs. CISC

Fazit dieses Vergleiches ist dass, beide Architekturen ihre Da-Seins Berechtigung in der aktuellen Technologischen Welt haben. Beide Versionen bringen Vor- und Nachteile mit sich, aber einen echten Gewinner gibt es in dem Spiel nicht. CISC machen sich viele Mechanismen der RISC zu nutze und nähern sich ihnen so immer mehr an. Jedoch wird sich RISC niemals in der Welt der Heim-Computer durchsetzen aber immer Vorreiter im Bereich Embedded Systemen bleiben.

3.3. ARM926EJ-S

Betriebssysteme lassen sich auf jeder Prozessorarchitektur entwickeln die man sich vorstellen kann. Die Wahl auf den ARM926EJ-S fiel aufgrund diverser Recherchen. Aufgrund der Tatsache das es für die gängigen Intel und AMD Prozessoren bereits weitverbreitete Betriebssysteme gibt fiel die Wahl im vorhinein nicht auf diese Art.

Nach dem klar war wo ARM-Prozessoren eingesetzt werden, wie z.B. in Druckern, Handys, Tablets und vielen mehr, fiel die Entscheidung auf diesen Prozessor. Zudem kommt hinzu es gibt momentan noch nicht so viele Betriebssysteme wie bei den anderen Systemen. Vorteile sind z.B.


- **Energieeffizienz**
- **Schnelligkeit**
- **geringe Produktionskosten**
- **minimale Bauweise.**

3.4. Register

Der ARM926 Prozessor ist ein 32-Bit RISC Prozessor. Dieser Prozessor hat eine Gesamtzahl von 37 Registern[archManI], wobei 30 dieser Register den allgemeinen Zwecken und 6 als Statusregister dienen. Von diesen hier explizit die Register **R0-R7** und **R13-R15** zu erwähnen sind. **R0-R7** sind tatsächlich allgemein verwendbare Register, die unabhängig von dem aktuellen Prozessormodus sind, **R13** ist der *Stackpointer*, **R14** stellt das *Linkregister* dar und **R15** bezeichnet den *Program Counter*.

- **Stackpointer**
Zeigt auf die 'Top-Of-Stack' Adresse
- **Linkregister**
Zeigt auf den aktuell geretteten Programmcounter bevor eine Routine betreten wird
- **Programmcounter**
Zeigt auf die nächste Instruktion

Modes						
Privileged modes						
Exception modes						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Quelle:[archManI]

Abbildung 3.1.: ARM926EJ-S Register

Teilweise sind die Register mehrfach vergeben, denn im ARM Prozessor gibt es unterschiedliche Prozessor-Modi und für ein Großteil dieser Modi stellt der Prozessor für R13 und R14 neue Register zur Verfügung. Das spielt dann eine wichtige Rolle, wenn man unterschiedliche Stacks für die Modi aufbauen muss.

3.5. Prozessor Modus

Der ARM926EJ-S stellt sieben unterschiedliche Modi bereit, in der sich der Prozessor befinden kann. Jeden dieser Modi kann man per Programmcode oder durch einen Interrupt betreten.

➤ User

Das ist der Modus, in dem alle Benutzerprogramme laufen. Sie haben keinen direkten Zugriff auf die Kernel-Routinen, sondern müssen dafür Trap-Instruktionen benutzen.

➤ FIQ(extra R8-R14)

Dieser Modus wird nur von sehr wenigen Interrupts betreten. Das sind die Interrupts, die eine sehr hohe Priorität haben und umgehend von dem Prozessor behandelt werden müssen.

➤ **IRQ(extra R13-R14)**

Alle Standard Interrupts, wie Tastatureingabe und andere, die darauf konfiguriert werden, landen in diesem Modus. Routinen in diesem Modus können in den User-Modus wechseln, um User-Programme auszuführen.

➤ **Supervisor(extra R13-R14)**

Dieser Modus ist ausschließlich für Trap-Routinen reserviert. Das bedeutet, alle Instruktionen, die von einem Userprogramm aufgerufen wurden, um Kernel-Methoden auszuführen.

➤ **Abort(extra R13-R14)**

Abort ist der Modus, in den der Prozessor fällt, wenn entweder eine Instruktion aufgrund eines Fehlers abgebrochen werden muss oder ein Fehler beim Abruf einer Speicherstelle auftritt.

➤ **Undefined(extra R13-R14)**

Dieser Modus wird nur dann betreten, sofern der ARM-Prozessor eine Instruktion von einem Co-Prozessor anfordert, dieser aber nicht reagiert.

➤ **System**

Dieser Modus ist nur für den Kernel. Kein User-Programm darf diesen betreten.

Der System-Mode ist ein spezieller Modus. Dieser wird über keinen Interrupt ausgelöst. Er ist deshalb vorhanden, weil das Betriebssystem ihn benutzt, um Betriebssystem-relevante Ressourcen zu benutzen. Weiterhin verfügt dieser Modus die gleichen Register wie der User-Modus.

3.6. Interrupt-Controller

Der ARM926EJ-S besitzt einen speziellen Interrupt-Controller, den sogenannten *Vectored Interrupt Controller* - *VIC*. Das ist eine Hardwarekomponente, die ein System zwischen Interrupts und dem Betriebssystem darstellt. Jede Hardware muss auf einem Chip an eine bestimmte Adresse gelegt werden. Für den VIC gibt es folgende Spezifikation:

Address	Name	Access	Description
0x10140000	PICIRQStatus	Read	IRQ status register
0x10140004	PICFIQStatus	Read	FIQ status register
0x10140008	PICRawIntr	Read	Raw interrupt status register
0x1014000C	PICIntSelect	Read/write	Interrupt select register
0x10140010	PICIntEnable	Read/write	Interrupt enable register
0x10140014	PICIntEnClear	Write	Interrupt enable clear register
0x10140018	PICSoftInt	Read/write	Software interrupt register
0x1014001C	PICSoftIntClear	Write	Software interrupt clear register
0x10140020	PICProtection	Read/write	Protection enable register
0x10140030	PICVectAddr	Read/write	Vector address register
0x10140034	PICDefVectAddr	Read/write	Default vector address register
0x10140100– 0x1014013C	PICVectAddr0– PICVectAddr15	Read/write	Vector address 0 register to Vector address 15 register
0x10140200– 0x1014023C	PICVectCntl0– PICVectCntl15	Read/write	Vector control 0 register to Vector control 15 register

Quelle:[archManI]

Abbildung 3.2.: VIC Register

Der Entwickler hat nun die Möglichkeit, diese Adresse programmatisch anzusteuern, um Informationen aus dem VIC zu erhalten. Die für dieses Projekt wichtigsten Register sind Folgende:

➤ **IRQ/FIQ Status Register - 0x10140000/0x10140003**

Mit diesem Register kann ermittelt werden, wie der aktuelle Status eines IRQ/FIQ ist.

➤ **Select Register - 0x01014000C**

Mit diesem Register kann bestimmt werden , welcher Interrupt ein IRQ oder FIQ sein soll.

➤ **Interrupt enable register - 0x10140010**

Gibt an, ob ein bestimmter IRQ/FIQ von dem VIC beachtet werden soll.

➤ **Interrupt enable clear register - 0x10140014**

Mit diesem Register kann ein Interrupt nach Auslösung geleert werden.

➤ **Vector address register - 0x10140030**

In dieses Register schreibt der VIC die Adresse der Interrupt Service Routine des momentan ausgelösten Interrupts.

➤ **Vector address register 0 - 15 - 0x10140100-0x1014013C**

In diese Register müssen die Adressen von den Interrupt Service Routinen geschrieben werden, die für diesen Interrupt zuständig sind.

➤ **Vector control register 0 - 15 - 0x10140200-0x1014023C**

In diese Register muss man äquivalent zu den 'Vector address registern' die Quelle des Interrupts und, ggf. ob der Interrupt aktiviert werden soll, schreiben.

Entwurf

4.1. Einleitung

In den vorherigen Kapiteln wurde geklärt, welche Anforderungen an **MopS** gestellt werden. In dem nun folgenden Kapitel wird eine Idee präsentiert, wie diese Komponenten miteinander interagieren sollen. Um einen groben Überblick über die Idee zu verschaffen, soll dieses Kapitel mit einer Grafik eingeleitet werden.

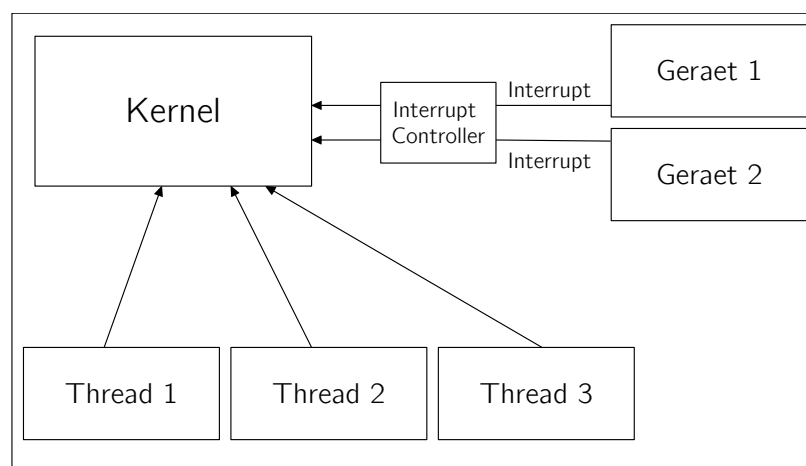


Abbildung 4.1.: **MopS** Überblick

MopS besteht im Großen und Ganzen aus vier übergeordneten Komponenten:

➤ **Geräte**

Wie in den Anforderungen beschrieben, besteht **MopS** aus zwei Hardwarekomponenten. Diese Komponenten können Interrupts auslösen, um mit dem Betriebssystem zu kommunizieren.

➤ **Interrupt Controller**

Damit kommen wir zu einem weiteren Fakt, der **MopS** prägt. Der Interrupt-Controller ist, wie in den Anforderungen beschrieben, eine Hardwarekomponente, die auf dem Chip

integriert ist und die Priorisierung und Weiterleitung von Interrupts an das Betriebssystem steuert.

➤ **Threads**

Die Threads stellen unter anderem die User-Programme in dem Betriebssystem dar. Sie können mit dem Kernel interagieren und werden von dem Kernel verwaltet.

➤ **Kernel**

Der Kernel ist der Hauptbestandteil von **MopS**. Er stellt die Schnittstelle zu sämtlicher Hardware und den Threads dar.

4.2. Geräte

Die Geräte stellen neben dem Kernel und den Threads eine sehr wichtige Rolle in **MopS** dar. Die Geräte können über Interrupts mit dem Kernel kommunizieren. Hierbei handelt es sich um eine hardware-basierte Interprozesskommunikation.

4.3. Interrupt-Controller

Wie aus der Abbildung 4.1 ersichtlich, liegt zwischen den Geräten und Kernel der sogenannte Interrupt-Controller, eine Hardwarekomponente, die die Priorisierung und Verwaltung von Interrupts übernimmt. In der Idee von **MopS** hat der Interrupt-Controller die Funktion die Quellen der Interrupts zu lokalisieren und die passenden Interrupt-Service-Routinen bereitzustellen, sodass diese von dem Kernel ausgeführt werden können. Abbildung 4.2 soll das verdeutlichen.

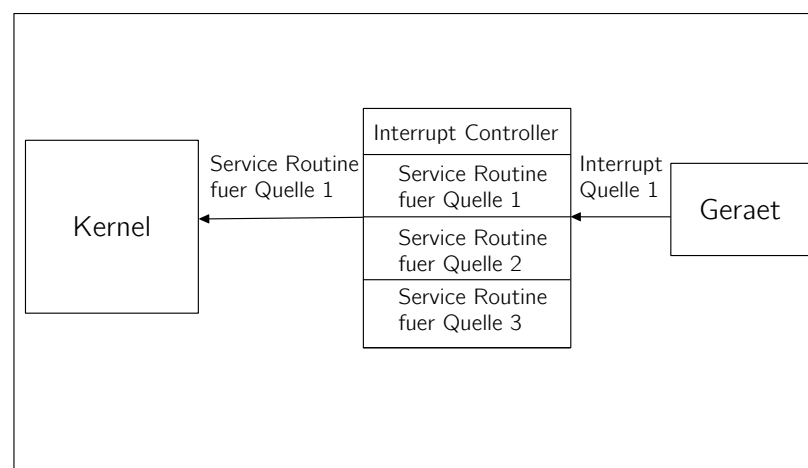


Abbildung 4.2.: Interrupt-Controller

4.4. Threads

In den Anforderungen wurde beschrieben, dass **MopS** aus drei Threads besteht. Diese Threads stellen User-Programme dar, die über bestimmte Routinen mit dem Kernel kommunizieren, jedoch aber auch von Interrupts unterbrochen werden können. Ein Thread ist in **MopS**, wie auch in vielen anderen Betriebssystemen, ein Abbild von einem Programm, der definierte Aktionen ausführt, z.B. eine Berechnung oder eine Ausgabe. Damit diese Aufgabe nicht der Kernel übernehmen muss, existieren eben genannte Threads. Aufgrund der Tatsache das **MopS** keine Festplatte oder ein anderes dauerhaft beschreibbares Medium besitzt, müssen die Threads alle in den RAM geladen werden. Die Idee von **MopS** war es also, die Threads in einen definierten Bereich im RAM zu laden. Die folgende Abbildung zeigt diesen Sachverhalt:

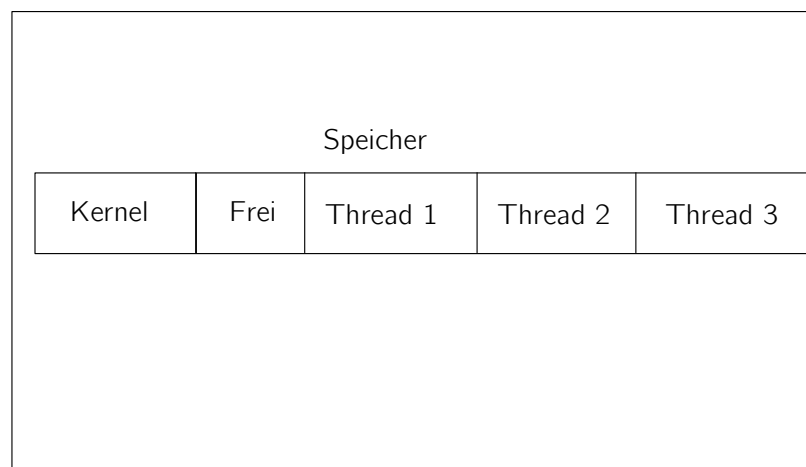


Abbildung 4.3.: Thread-Layout im RAM

An erste Stelle liegt der Kernel, daneben kann noch etwas freier Platz für andere Zwecke koexistieren und dahinter liegen alle Threads im RAM.

Wie schon in den Anforderungen beschrieben, ist es **MopS** nicht möglich, Threads zur Laufzeit zu erzeugen. Es kann nur die Vordefinierten starten und verwalten. Es musste also eine Vorgehensweise entwickelt werden, wie diese Threads in den RAM kommen, und vor allem, wie sie in **MopS** integriert werden können.

Hierzu entstand die Idee der RAM-Disk. Das ist eine Datei in der die Informationen zu den Threads gespeichert werden. Nun stellt sich natürlich die Frage, welche Informationen das sind. Die Frage lässt sich ganz einfach beantworten: **Die Information, die einen Thread klassifiziert, muss ohne Zweifel der Assembler-Code des Threads sein.**

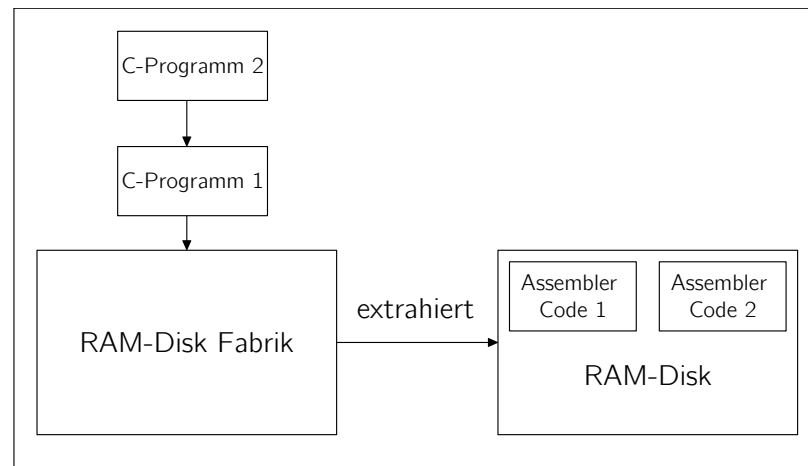


Abbildung 4.4.: Erstellung der RAM-Disk

Die Idee, die in **MopS** verfolgt wurde, bestand darin, einfache C-Programme zu entwerfen und diese durch ein Programm zu schicken, das den Assembler-Code dieser Programme extrahiert. In Abbildung 4.4 ist zu erkennen, dass zwei Programme durch die sogenannte RAM-Disk-Fabrik wandern. Diese extrahiert den Assembler-Code und verpackt ihn in die dafür vorgesehene RAM-Disk. Der entstandene Assembler-Code kann dann von einem **MopS**-definierten Lader in den Speicher befördert werden.

Neben dem Vorbereiten und Laden der Threads in den RAM, ist es aber auch notwendig, die Threads zu verwalten. Diese Aufgabe übernimmt der Kernel.

4.5. Kernel

Der Kernel ist der Teil von **MopS**, der sämtliche Verwaltungsaufgaben übernimmt, die man sich vorstellen kann. Das geht von den Interrupts bis hin zur Verwaltung der Threads.

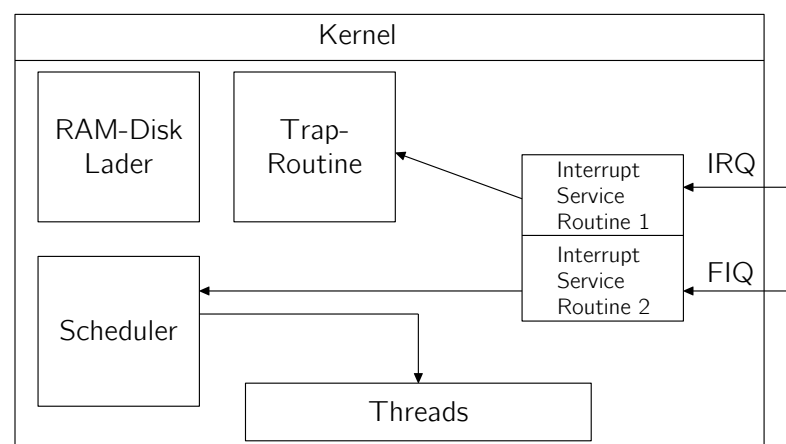


Abbildung 4.5.: Kernel - Überblick

Die Abbildung 4.5 gibt noch einmal einen detaillierten Einblick in das Innenleben des Kernels. Zu sehen sind hier alle Komponenten, die auf die Idee zusätzlich Einfluss hatten:

Der RAM-Disk-Lader, Interrupt-Service Routinen, der Scheduler und die Trap-Routinen.

Diese Abbildung soll darstellen, wie ein normaler Ablauf in dem Kernel aussehen kann.

Im ersten Schritt kommt ein Interrupt oder Fast-Interrupt-Request in das System, der Interrupt-Controller priorisiert diesen dann und stellt dem Kernel die passende Interrupt-Service Routine zur Verfügung. Die jeweilige Routine kann daraufhin z.B. entweder eine Trap-Routine oder aber, was wesentlich interessanter ist, den Scheduler aufrufen. Dieser Scheduler geht nun an die Thread-Tabelle und greift sich einen neuen Thread, der jetzt in den Prozessor geladen wird. Womit wir zum nächsten Schritt kommen.

Der Scheduler ist ein Manager für die Verwaltung der Prozessorzeiten und Priorisierung der Threads.

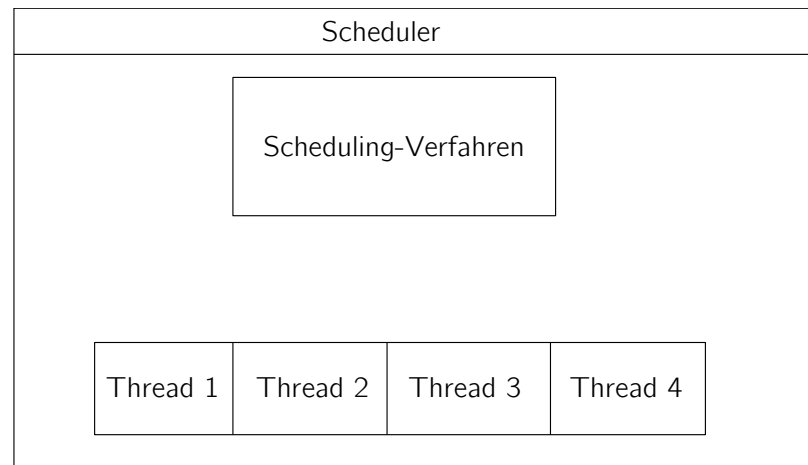


Abbildung 4.6.: Scheduler

Er setzt sich aus zwei wichtigen Komponenten zusammen:

Dem Scheduling-Verfahren, welches frei gewählt werden kann, und einer Tabelle von Threads. Aus dieser Tabelle wählt der Scheduler, je nach Scheduling-Verfahren, einen Thread aus und übergibt ihm die Kontrolle. Es gibt viele Scheduling-Verfahren, aber hier wurde sich jedoch einer Idee bedient, die in der frühzeitigen Entwicklung von Schedulingern weit verbreitet war.

Das Verfahren nennt sich **Round-Robin-Verfahren**. Die folgende Grafik soll das Verfahren beschreiben.

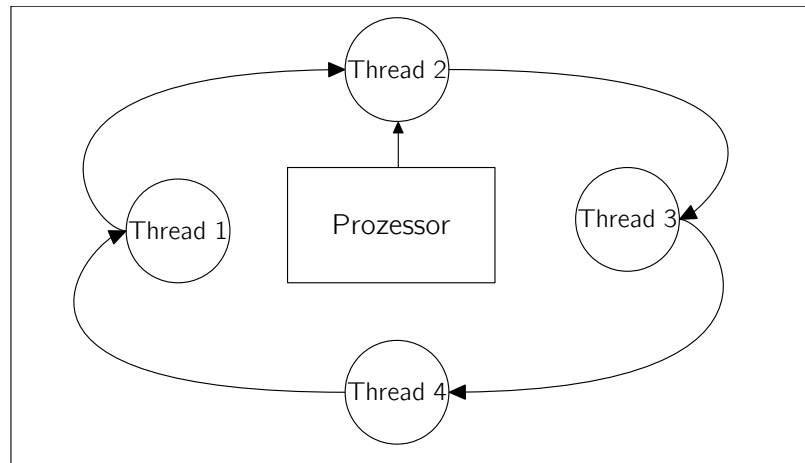


Abbildung 4.7.: Round-Robin Verfahren - Schematisch

Beim Round-Robin Verfahren wird der Scheduler so entworfen, dass er jedem Thread eine fixe Zeitspanne an Prozessorzeit zusichert und die Kontrolle dann an den jeweiligen Thread übergibt. Nach Ablauf der Zeit wird dann der nächste Thread in der Tabelle aufgerufen. Dieses Verfahren wurde für **MopS** deshalb gewählt, weil in Handheld Geräten keine Sonderpriorisierungen stattfinden müssen und die Umsetzung in den Zeitrahmen passte.

Konkretisierung des Entwurf

5.1. Einleitung

Die Entwicklung von **MopS** erfolgte in mehreren Schritten. Wichtige Zwischenstopps waren hier der Startprozess, die Interrupthandler, der Interrupt-Controller, die Interrupt-Service-Routinen und das Threadmanagment. Jeder dieser Punkte bedurfte einer einzelnen Entwurfsphase, auf die jetzt genauer eingegangen wird.

5.2. Startmechanismus

Der Startprozess unterteilt sich in drei Schritte:

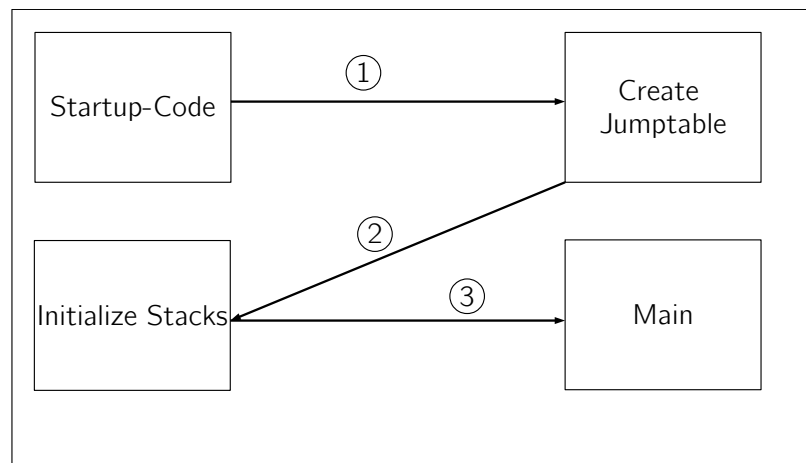


Abbildung 5.1.: Kernel-Image Version 1

1. Startup-Code

Der Emulator springt an die Adresse `0x000000`¹, an dieser Stelle steht die erste Assembleroutine des Betriebssystems. Diese Routine dient dazu um gewisse Vorbedingungen zu erstellen. Eine davon ist die nachfolgende.

¹Diese Adresse wird über das Linker-Script bestimmt

2. Interrupt Handler

Interrupts können von externen oder internen Ressourcen ausgelöst werden. Damit der Prozessor weiß wo er im Falle eines Interrupts hinspringen muss, schreibt ARM eine Struktur vor die eingehalten werden muss [**archManI**].

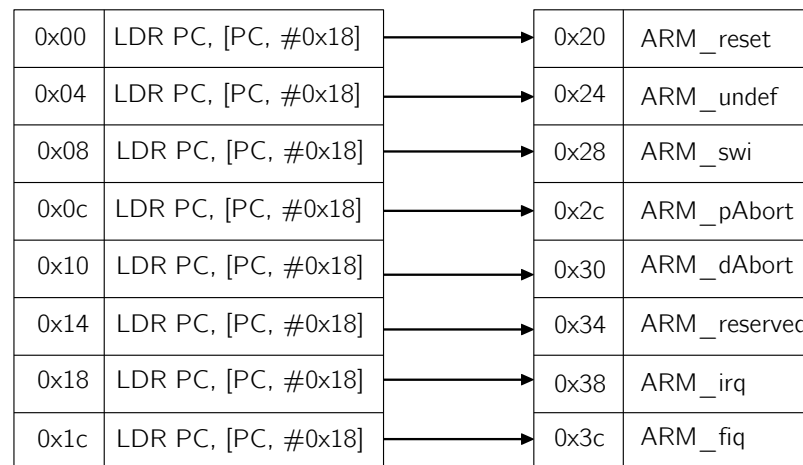


Abbildung 5.2.: Erzeugen der Sprungtabelle

In diesem Schritt ist erkennbar, dass es zwei Tabelle gibt die von dem Programmierer erstellt werden müssen. Die erste Tabelle enthält Programm-Relative Adressen auf eine zweite Tabelle. In der zweiten Tabelle befinden sich dann die konkreten Adressen der Exceptionhandler. Neben der Exception für einen Reset des Systems gibt es noch weitere Exceptions wie die Undefined Operation (ARM_undef), Softwareinterrupt (ARM_swi), Prefetch-Abort (ARM_pAbort), Data-Abort (ARM_dAbort), die Reserved Exception (ARM_reserved) und ganz wichtig zu erwähnen der IRQ (ARM_irq und FIQ (ARM_fiq).

Der Vorteil des Mechanismus, Programmrelative Adresse anstatt die direkten Handler zu laden, besteht darin das man so leicht die Handler austauschen oder zusätzliche hinzufügen kann ohne dabei den Assemblercode zu ändern.

Neben der Erstellung der Sprungtabelle für die Exceptionhandler, ist es weiterhin notwendig den Stack für die jeweiligen Prozessormodi zu definieren, dies geschieht im nächsten Schritt.

3. Erstellung der Stacks mit anschließenden Sprung in main

Die für **MopS** relevanten Modi des Prozessors sind

- IRQ-Modus
- FIQ-Modus
- System-Modus
- Supervisor-Modus

Jeder dieser vier Modi, bis auf den System-Modus, hat seinen eigenen Stackpointer und für jeden muss dementsprechend der passende Stackpointer gesetzt werden. Das ist deshalb

notwendig da im Falle einer Exception der Prozessor in den jeweiligen Modus wechselt und wenn kein valider Stackpointer vorhanden ist kann es zu undefinierten Verhalten kommen. Die Größe der Stackpointer lässt sich über das Link-File bestimmen, für **MopS** wurde eine Größe von 8KB je Modus gewählt.

Sobald die Stacks alle initialisiert sind erfolgt der Sprung in die **main** Routine des Betriebssystems. Ab hier finden nun weitere Schritte statt um das System fertig zu initialisieren.

5.3. Interrupt-Controller

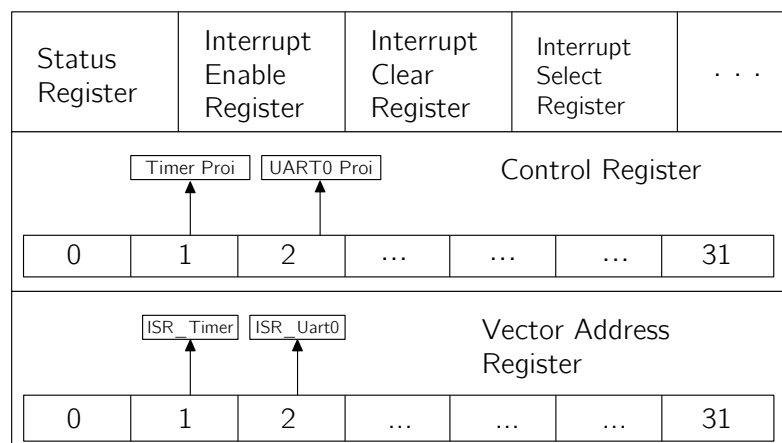
Ein Interrupt-Controller stellt in einem Betriebssystem die Schnittstelle zwischen den Interrupts und der Hardware dar. Er priorisiert die Interrupts, die von externen wie auch internen Quellen ausgelöst werden können, und leitet sie an das Betriebssystem weiter. Es gibt zwei Arten von Controllern:

- Non-Vectored Interrupt-Controller
- Vectored Interrupt-Controller

Die erste Version, **Non-Vectored Interrupt-Controller**, stellt nur die Möglichkeit bereit einen Interrupt abzufangen, jedoch muss sich der Programmierer darum kümmern welche Quelle den Interrupt ausgelöst hat, die Priorität ermitteln und die passende Interrupt-Service Routine herausfinden. Das klingt zwar im ersten Moment ganz logisch und sinnvoll, ist aber mit einer Menge Code verbunden und stellt deshalb eine sehr große Fehlerquelle dar.

Der Vectored Interrupt-Controller ist eine in Hardware gegossene Komponente auf dem Board welches man benutzt. Er bietet die Konfigurationsmöglichkeit zu definieren, welche Interrupts von welchen Quellen ausgelöst werden können, welche Prioritäten sie haben und welche Interrupt-Service Routinen für diese Interrupts zur Verfügung gestellt werden. Nun braucht man im Falle eines Interrupts keine umfangreichen Mechanismen losstreifen um die Quellen zu ermitteln, sondern der Interrupt-Controller stellt jetzt alle diese Informationen bereit. Die Installation dieses Controllers ist zwar komplexer als die der ersten Version, aber die Möglichkeiten sind breiter und die Benutzung ist komfortabler. Aus diesen Gründen wurde sich bei **MopS** für den **Vectored Interrupt Controller** entschieden. Das folgende Bild ist eine schematische Darstellung des Vectored Interrupt-Controller.

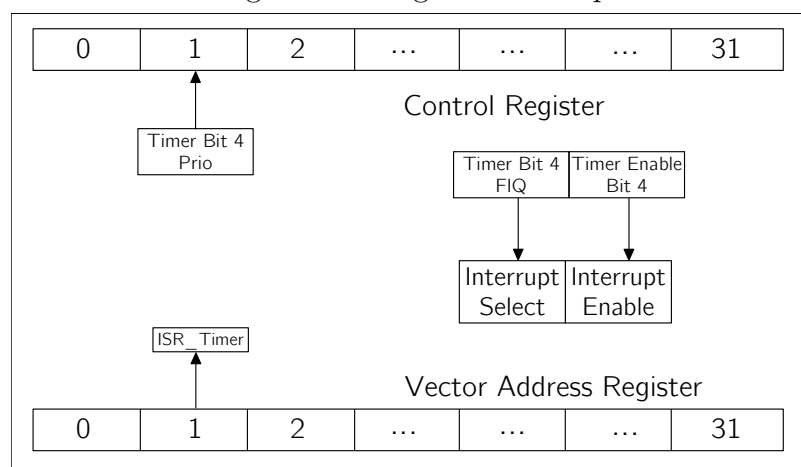
Abbildung 5.3.: Vectored Interrupt Controller



Sofern das Board einen Vectored Interrupt-Controller zur Verfügung stellt ist dieser an einer bestimmten Adresse lokalisiert. Bei dem von **MopS** emulierten System ist die Adresse `0x10140000` [archM]. Ab dieser Adresse beginnt der Adressbereich des Controllers, hier befinden sich oben genannte Register wie Statusregister, Interrupt Enable Register, Interrupt Clear Register, Interrupt Select Register etc., desweiteren sind hier auch die für die Interrupt Vektoren wie auch die Control Register für die jeweiligen Interrupts [vic].

Die relevanten Register für **MopS** sind die Vector Address Register, Control Register, Interrupt Enable, Interrupt Clear und Interrupt Select Register. Über diese ist es möglich die Service Routinen für die Interrupts zu definieren, wie auch die Prioritäten und ob der konfigurierte Interrupt als ein IRQ oder FIQ behandelt werden soll. Mit der folgenden Grafik wird schematisch dargestellt wie eine Konfiguration des Controllers aussehen kann.

Abbildung 5.4.: Konfigurationsbeispiel VIC



Dieses Beispiel zeigt eine Beispielkonfiguration des Timerinterrupts. Um diesen Interrupt zu konfigurieren sind vier Schritte notwendig:

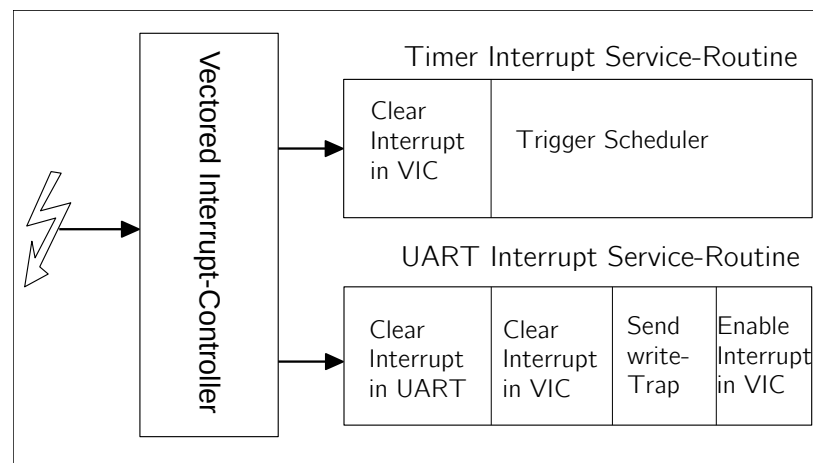
- Installation der Adresse der Interrupt Service Routine
- Konfiguration des Control Register mit Priorität des Timers
- Entscheidung ob der Interrupt als IRQ oder FIQ ausgelöst werden soll
- Aktivierung der Interrupt-Auslösung

Für den *Timer0-Interrupt* muss an dieser Stelle das 4. Bit [**archManI**] in dem Interrupt-Controller setzen. Das gilt sowohl für die Aktivierung, Auswahl des Interrupts als auch die Priorität.

5.4. Interrupt-Service Routinen

Nachdem die Interrupts konfiguriert wurden ist es notwendig die Interrupt-Service Routinen der Interrupts zu definieren. Beispielhaft werden hier die Routinen des Timers und des UART0-Interrupts präsentiert.

Abbildung 5.5.: Interrupt Service-Routinen Timer & UART0



Sobald der Interrupt ausgelöst wurde behandelt der Controller diesen und leitet es an die zugehörige Routine zur Behandlung weiter. Im Beispiel des Timers, wird hier der Interrupt erst im Controller auf 'Behandelt' gesetzt und dann wird der Scheduler aufgerufen um dem nächsten Prozess zu starten. Das Löschen des Interrupts im Controller ist deshalb erforderlich, damit ein neuer Interrupt ausgelöst werden kann, denn nur nachdem der Interrupt als behandelt markiert wurde, kann ein neuer erzeugt werden.

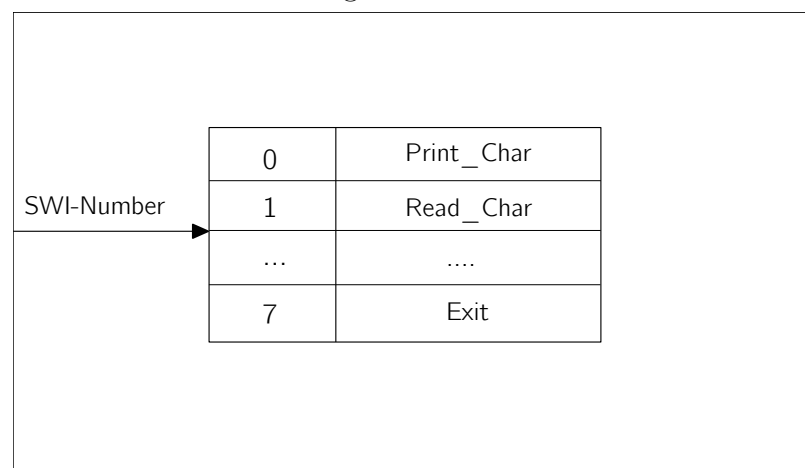
In der aktuellen Implementation von **MopS** gibt es zwei Interrupts auf die reagiert wird. Zum einen ist das der *Timer0*-Interrupt und zum anderen der Interrupt des *UART0*-Interface. Bei Bedarf kann man auch noch weitere Interrupts definieren, dazu muss jedoch die Konfiguration des VIC abgeändert werden.

5.5. Syscalls

Neben den IRQ und FIQ spielen die Syscall auch noch eine sehr wichtige Rolle. Im Rahmen der ARM-Architektur wurden die Syscalls mit dem Namen *Software Interrupts* bezeichnet. Sicherlich ist der Name Syscalls oder Traps geläufiger. Um aber konsistent zu bleiben, wird im laufenden der Name Software Interrupt(SWI) verwendet.

Ein SWI ist eine Instruktuin die nicht im Kernel-Modus läuft aber Kernel-Routinen aufrufen darf. Das ist dann sinnvoll, wenn ein User-Programm Zugriff auf eine Kernel-Routine (wie das Schreiben auf die Konsole) benötigt. Wie in Abbildung 5.2 zu erkennen ist, wird bei einem SWI-Interrupt die ARM_swi Routine angesprungen. Diese Routine leitet den Interrupt-Request an eine weitere Komponente weiter die schematisch wie folgt aussieht.

Abbildung 5.6.: SWI-Handler



In der aktuellen Fassung von **MopS** ist nur der SWI-Handler für den `Print_Char`-SWI definiert, es sollen jedoch weitere folgen.

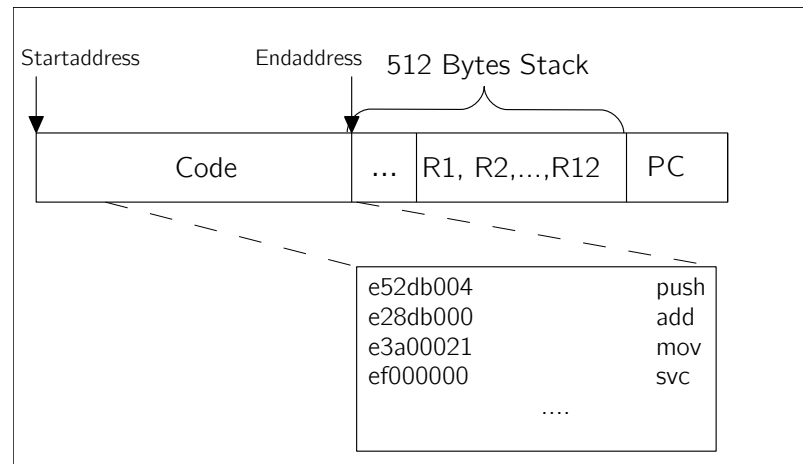
5.6. Threadmanagment

5.6.1. Threadlayout

Das Threadmanagment spielt eine wichtige Rolle in jedem Betriebssystem. Aufgrund der Komplexität und des zeitlichen Faktors wurde bei **MopS** auf ein rudimentäreres System gesetzt. Das bedeutet das keine Threads zur Laufzeit des Systems geladen werden können sondern die Threads vor Beginn definiert werden mussten.

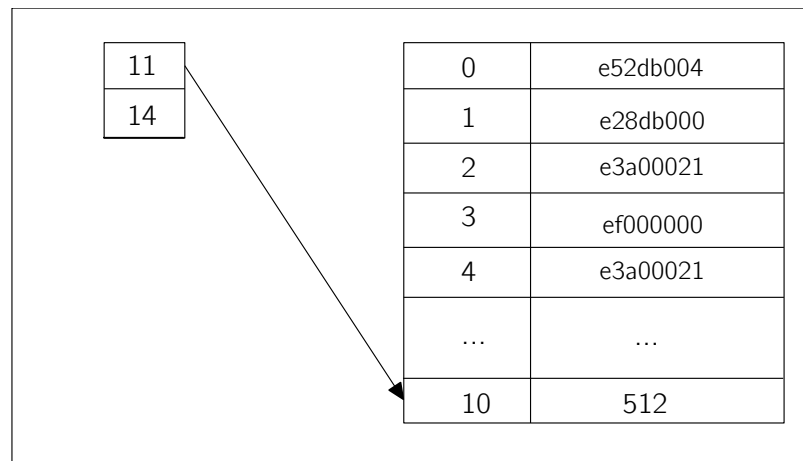
Eine abstrakte Darstellung dieses Thread-Image kann man sich wie folgt vorstellen:

Abbildung 5.7.: Thread-Image



Diese Struktur wird zu Beginn des Betriebssystems im RAM hergestellt. Zuvor muss jedoch der Assemblercode aus dem zu ladenden Thread extrahiert werden. Die Threads werden als einfache C-Programme dargestellt. Diese C-Programme werden so rudimentär wie möglich kompiliert und gelinkt, das bedeutet, dass sämtliche Standardbibliotheken nicht mitgelinkt werden und keine main-Funktion bereitgestellt wird. Die entstandene .ELF-Datei wird dann in das Binärformat umkopiert und danach extrahiert ein Programm den Assemblercode aus der Binärdatei und schreibt diesen in eine RAM-Disk. Für **MopS** wurde eine sehr proprietäre RAM-Disk gewählt. Folgende Grafik zeigt eine schematische Darstellung dieser RAM-Disk.

Abbildung 5.8.: RAM-Disk



Nachdem diese RAM-Disk erstellt wurde, kann **MopS** diese im Betrieb laden und den Assemblercode an die passende Position im RAM laden. Neben den Informationen über den Assembler-Code enthält die RAM-Disk unter anderem die Information, wieviel Bytes an Stack für den Prozess reserviert werden. Dieser Bereich wird dann beim Kopieren vorerst nur mit Nullen aufgefüllt.

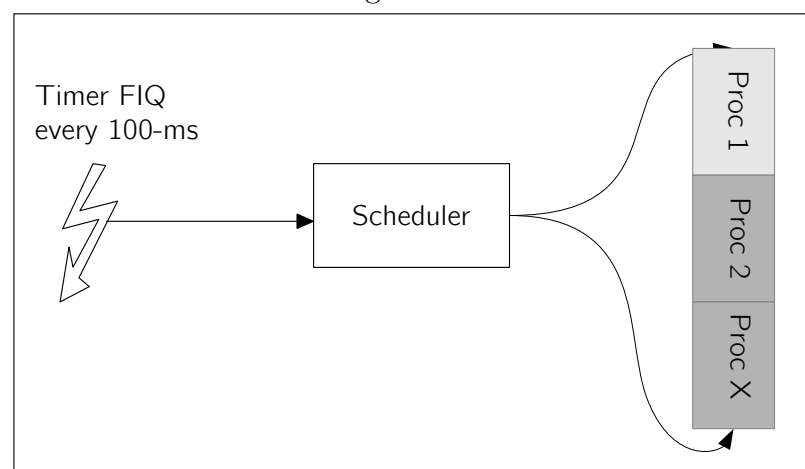
Um die Möglichkeit offen zu halten, mehr als einen Prozess in den RAM zu laden, stellt die RAM-Disk eine weitere Tabelle zur Verfügung, in welcher die Informationen zur Länge jeder einzelnen

Threads eingetragen sind. Für Abbildung 5.8 bedeutet das, dass der erste Thread eine Länge von 11 aufweist und ab dem 12. Eintrag der nächste Thread beginnt.

5.6.2. Scheduling

Sobald der Thread in den RAM geladen wurde, kann das Betriebssystem jetzt den ersten dieser Prozesse starten. Dieser Start manifestiert sich dadurch, dass der Stackpointer, vom Kernel auf den Start-Bereich des neuen Stacks von dem Thread, umgemappt werden muss. Danach werden alle Register auf dem neuen Stack gesichert und es erfolgt ein Sprung in die Routine, die zuletzt aus der RAM-Disk geladen wurde.

Abbildung 5.9.: RAM-Disk



In Grafik 5.9 wird veranschaulicht, wie der Mechanismus des Thread-Umschaltens in **MopS** umgesetzt wurde. Hier sieht man, dass alle 100ms der Timer-Interrupt ausgelöst wird und dieser startet den Scheduler. Der Scheduler sucht dann den nächsten wartenden Prozess, hier Gelb, raus und schaltet diesen ein. Der andere, momentan grüne, wird dann in den wartenden Status geschaltet. Es gibt einige bekannte Scheduling-Verfahren in modernen Betriebssystemen, viele dieser Verfahren kooperieren auch unter bestimmten Umständen, um die beste Performance herauszuholen. Hier sind ein paar der wichtigsten Verfahren etwas näher beschrieben:

➤ **First-Come First-Serve**

Dieses System kann man sich wie eine Schlange an der Post vorstellen. Prozesse werden in eine Queue² gepackt und von dort bearbeitet. Dieses System hat jedoch den Nachteil, dass lange Wartezeiten, durch Prozesse, die die CPU sehr überdurchschnittlich lange in Anspruch nehmen, entstehen können.

➤ **Shortest-Job-First**

²Eine weitverbreitete Datenstruktur in modernen Programmiersprachen, die nach dem *First-In First-Out* Prinzip funktioniert.

Während bei diesem Verfahren darauf abgezielt wird den Prozessen die CPU-Zeit zu überlassen welche am kürzesten sind. Wie in [scheduling] geschrieben

» This algorithm associates with each process the length of the process's next CPU burst.«

versucht dieser Algorithmus anhand der CPU-Bursts[scheduling] zu ermitteln wie lange ein Prozess in etwa benötigen wird. Anhand dieser Informationen wird also dann der nächste Prozess ermittelt der an der Reihe ist. Sollten jedoch zwei Prozesse die gleichen CPU-Bursts haben, so wird das *First-Come First-Serve* Verfahren angewendet. Der Vorteil hier ist natürlich das es wesentlich optimaler als die vorherig genannten, jedoch ist es auch komplizierter zu implementieren.

➤ Round-Robin

Dieses Verfahren *»is designed especially for time-sharing systems.«*[scheduling] Das bedeutet das der Algorithmus eine kleine Zeitscheibe definiert in der der Prozess die CPU bekommt. Danach werden die Prozesse die sich in der "Ready-Queue" befinden abgearbeitet und jeder bekommt für die vorher bestimmte Zeit die CPU. Diese Queue wird als eine Ring-Liste behandelt, das hat den Effekt das der Scheduler immer wieder jeden Prozess kurz rannimmt. Der Vorteil dieses Mechanismus ist das jeder Prozess gleich bewertet wird und das es relativ einfach zu implementieren ist. Jedoch bringt genau dieser Vorteil auch einen Nachteil mit sich, nämlich das Prozesse die eigentlich eine längere Zeitschreibe bräuchten immer warten müssen bis sie wieder am Zug sind und das kann natürlich zu sehr hohen Latenzen in der Ausführung kommen.

Es wurden jetzt eine Reihe von Scheduling-Mechanismen vorgestellt und es musste eine Entscheidung für **MopS** getroffen werden. Aufgrund der einfachen Implementation, fiel die Wahl auf das **Round-Robin** Verfahren. Die Nachteile konnten für die erste Version von **MopS** vernachlässigt werden.

Implementation

6.1. Einleitung

Bei der Entwicklung von *MopS* mussten einige wichtige Entscheidung bezüglich der Entwicklungs- wie auch Emulationsumgebung getroffen werden. Im folgenden werden diese Entscheidungen von allen Gesichtspunkten beleuchtet. Neben diesen Aspekten gibt es in diesem Kapitel einen tiefen Einblick in die Implementation von *MopS*.

6.2. Entwicklungsumgebung

Mit der Entscheidung ein Mini-Betriebssystem zu programmieren stellt sich natürlich auch die Frage mit welchen Werkzeugen man den Code entwickelt. Zur Entwicklung von ARM-basierten Code kann die Entwicklungsumgebung Eclipse¹ genutzt werden. Dennoch wurde sich für den konservativen Weg entschieden und die Entwicklung läuft seit her mit dem Linux-integrierten Editor *vim*. Die Vorteile gegenüber einer Integrierten Entwicklungsumgebung sind die folgenden:

Vorteile:

➤ **Schnelligkeit**

Es ist keine separate Installation einer IDE notwendig, denn *vim* ist auf jedem Linux System vorinstalliert. Weiterhin startet *vim* in einer sehr kurzen Zeit.

➤ **Unabhängigkeit**

Sollte die Entwicklung auf einem anderen System weitergehen, so ist es nicht notwendig IDE abhängige Einstellungen vorzunehmen.

➤ **Kontrolle**

Viele IDEs bringen ein umfangreiches Portfolio an Funktionen mit sich, die jedoch auch Problematisch werden können wenn nicht mehr klar ist was für Schritte die IDE neben den

¹<http://www.eclipse.org/>

eigentlich notwendigen noch durchführt. Da *vim* ein rein textbasierter Editor ist, kann man hier sicher sein das keine unklaren Sachen im Hintergrund passieren.

Jedoch bringt die Entwicklung ohne IDE auch Nachteile mit sich, die hier nicht außen vorgelassen werden dürfen.

Nachteile:

➤ **Komplex**

vim ist kein Werkzeug für Anfänger, da es sehr nativ und durchaus eine gewisse Zeit bedarf die Verwendung gut zu beherrschen, hier sind die IDEs teilweise klar im Vorteil.

➤ **Unintuitiv**

Die Benutzung eines rein textbasierten Editors, wie *vim*, ist insofern Nachteilig das sämtliche Features einer IDE, wie Autovervollständigung, Intellisense, Fehlermeldungen während des Schreibens etc., verloren gehen. Weiterhin kommt dazu das bei der Benutzung von *vim* die Navigation und Steuerung, im wie auch von dem Dokument, relativ komplex ist, sofern man es nicht gewohnt ist.

6.3. Startmechanismus

Der Startmechanismus ist einer der wichtigsten Prozesse eines jeden Betriebssystems. Eine große Herausforderungen bei **MopS** war die Definition des Startprozess. Dies umschließt:

➤ **Was bedeutet *Startprozess*?**

Die Frage zur Bedeutung des Startprozess konnte sehr schnell beantwortet werden. Da keine Hardware vorlag auf der ein Knopf hätte gedrückt werden können verlief der Startprozess sehr unspektakulär: Laden eines fertig assemblierten, kompilierten und gelinkten Kernel-Image[siehe Abbildung 5.1] in den qemu!

Code-Beispiel 6.1: Laden der Kerneldatei in qemu

```
1 qemu-system-arm -M versatilepb -m 128M -nographic -s -S -kernel mops.bin
```

➤ **Welche Komponenten sind daran beteiligt?**

Nachdem geklärt wurde was der Startprozess für **MopS** bedeutet stand dann auf dem Plan, herauszufinden welche Komponenten am Startprozess beteiligt sind und ob diese in einer definierten Reihenfolge ausgeführt werden müssen. Die erste und wichtigste Komponente ist die Definition der Startadresse an welche die erste Assembler Datei geladen werden musste. In dem Linkerscript wurde als Startadresse die Adresse 0x000000 gewählt, an diese Stelle wird nun der Code geladen.

Code-Beispiel 6.2: Linker-Datei

```

1 ENTRY(_start)
2 SECTIONS
3 {
4     . = 0x000000;
5     .ramvect :
6     {
7         __ram_start = .;
8         . += 0x1000;
9     }
10    . = ALIGN(4);
11    . = 0x10000;
12
13    .startup :
14    {
15
16        startup/startup.o(.text)
17        startup/initstacks.o(.text)
18    }

```

Wie man in Zeile 16 und 17 erkennen kann werden im nächsten Schritt die *startup.o* und *initstacks.o* geladen. Diese Dateien stellen die Grundeinstellungen des Betriebssystems her. Um zu verdeutlichen was diese Dateien machen, folgt die *startup.s* Datei.

Code-Beispiel 6.3: Startup-Datei

```

1 .text
2 .code 32
3 .global vectors_start
4 .global vectors_end
5 .global _start
6
7 .equ I_BIT, 0x80
8 .equ Mode_USR, 0x10
9
10 _start:
11     b reset_handler
12     b .
13     b .
14     b .
15     b .
16     b .
17     b .
18     b .
19
20 reset_handler:
21
22     ldr r0, =reset_handler
23     ldr sp, =stack_top
24     bl map_vectors
25     // init all stacks
26     bl initstacks
27     // save the current programm status register
28     mrs r0, cpsr
29     // enable irq mode
30     bic r0, r0, #I_BIT
31     // and save back the newly enabled mode
32     msr cpsr, r0
33     msr cpsr_c, #(Mode_USR)

```

```

34  BL main
35  B .
36
37  .end

```

In Zeile 24 sieht man den in Abbildung 5.1 (1) beschriebenen Sprung in die Methode die die Exceptionhandler mappt. Weiterhin ist in Zeile 26 der, in Abbildung 5.1 (2), erkennbare Sprung in die Methode die die Stacks erzeugt. Nicht zuletzt dann der Sprung in die Methode `main`, in Zeile 34. Somit ist der Kreislauf aus Abbildung 5.1 geschlossen.

Auch hier wird auf die jeweiligen Methoden eingegangen die in 5.2 schematisch dargestellt wurden.

➤ Exceptionhandler erstellen

Beim Erstellen der Sprungtabelle für die Exceptions kommt es drauf an das die Sprungadresse korrekt mit den passenden Methoden für die jeweilige Exceptions befüllt werden. In der Abbildung 5.2 erkennt man das die Adresse `0x00 - 0x1c` mit den passenden Assemblerbefehlen gefüllt werden die einen Sprung an die passende Adresse der Methode erlauben. In C wird das ganze auf folgende Art und Weise getan.

Code-Beispiel 6.4: Sprungtabelle erstellen I

```

1  extern uint8_t __ram_start;
2  uint32_t LDR_PC_PC = 0xe59ff000U;

```

Wichtige Stellen in diesem Quellcode sind die Teile wo man auf die Adresse der Variable `__ram_start` zugreift. Denn das ist die Adresse `0x00000000` die im Linker-Script (Code-Ausschnitt 6.3) definiert wurde. Sie ist deshalb so wichtig, weil die Exceptionhandler nach dem von ARM definierten System bereitgestellt werden müssen. Eine weitere wichtige Variable ist die `LDR_PC_PC`, diese beinhaltet die in Hex formatierte Assemblerroutine `LDR PC, [PC, #0x18]`. Sobald man diese Adresse mit `0x18` in eine ODER-Verknüpfung bringt entsteht das gewollte Ergebnis `LDR PC, [PC, #0x18]`. Mit diesem Wissen kann man nun die Sprunganweisungen erstellen die das notwendige Schema widerspiegelt.

Code-Beispiel 6.5: Sprungtabelle erstellen II

```

1  *(uint32_t volatile *)(&__ram_start + 0x00) = LDR_PC_PC | 0x18;
2  *(uint32_t volatile *)(&__ram_start + 0x04) = LDR_PC_PC | 0x18;
3  *(uint32_t volatile *)(&__ram_start + 0x08) = LDR_PC_PC | 0x18;
4  *(uint32_t volatile *)(&__ram_start + 0x0c) = LDR_PC_PC | 0x18;
5  *(uint32_t volatile *)(&__ram_start + 0x10) = LDR_PC_PC | 0x18;
6  *(uint32_t volatile *)(&__ram_start + 0x14) = LDR_PC_PC | 0x18;
7  *(uint32_t volatile *)(&__ram_start + 0x18) = LDR_PC_PC | 0x18;
8  *(uint32_t volatile *)(&__ram_start + 0x1c) = LDR_PC_PC | 0x18;
9
10 *(uint32_t volatile *)(&__ram_start + 0x20) = (uint32_t)reset_addr;
11 *(uint32_t volatile *)(&__ram_start + 0x24) = 0x04U;
12 *(uint32_t volatile *)(&__ram_start + 0x28) = 0x08U;
13 *(uint32_t volatile *)(&__ram_start + 0x2c) = 0x0cU;

```



```

14  *(uint32_t volatile *)(&__ram_start + 0x30) = 0x10U;
15  *(uint32_t volatile *)(&__ram_start + 0x34) = 0x14U;
16  *(uint32_t volatile *)(&__ram_start + 0x38) = 0x18U;
17  *(uint32_t volatile *)(&__ram_start + 0x3C) = 0x1CU;

```

➤ Erstellung des Stacks

Bei der Erstellung der Stacks für die verschiedenen Modi sind zwei Komponenten notwendig. Zum einen das Linkerscript und hier die folgenden Zeilen:

Code-Beispiel 6.6: Stack erstellen I

```

1  stack_top = .;
2  . += 0x2000;
3  __sys_stack_top = .;
4
5  . += 0x2000;
6  __irq_stack_top = .;
7
8  . += 0x2000;
9  __fiq_stack_top = .;
10
11 . += 0x2000;
12 __svc_stack_top = .;
13
14
15 __k_heap_start = .;
16 . += 0x2000;
17 __k_heap_end = .;

```

Hier wird definiert an welcher Stelle die Stacks lokalisiert sind, das gibt immer der “.” an. Dieser Punkt gibt die aktuelle Stelle im RAM an, an der sich gerade der Linker beim Linken befindet. Die Zeile 2 z.B. sagt aus das der `__sys_stack_top` eine Länge von 8.192 Bytes hat. Diese Zahl wurde nach Gefühl gewählt. Mittels dieser Variablen haben wir wieder Zugriff auf die jeweiligen Adressen nach dem Linken, nun kommt die zweite Komponente ins Spiel, mit der wir die tatsächlichen Stackpointer setzen.

Code-Beispiel 6.7: Stack erstellen II

```

1  initstacks:
2  // stack_base could be defined above, or located in a scatter file. CPSR currently in
   SVC mode.
3  mov r1, lr
4  // Enter each mode in turn and set up the stack pointer
5  MSR CPSR_c, #(Mode_FIQ|I_Bit|F_Bit)
6  ldr sp, __fiq_stack_top
7  MSR CPSR_c, #(Mode_IRQ|I_Bit|F_Bit)
8  ldr sp, __irq_stack_top
9  MSR CPSR_c, #(Mode_SYS|I_Bit|F_Bit)
10 ldr sp, __sys_stack_top
11 MSR CPSR_c, #(Mode_SVC|I_Bit|F_Bit)
12 ldr sp, __svc_stack_top
13 // Branch back to current mode return address (SVC in this case, as it was the
   initial mode)
14 mov pc, r1
15 .end

```

Da für die Modi **Supervisor**, **Abort**, **Undefined**, **Interrupt** und **Fast Interrupt** unterschiedliche Register für den Stackpointer belegt werden, ist es notwendig für diese Register die richtigen Adressen zu setzen. Aufgrund der Tatsache das **MopS** nur die Modi **System**, **Supervisor**, **Interrupt** und **Fast Interrupt** als wichtig ansieht sind nur vier Stackpointer zu setzen. Damit das korrekt von statten läuft wechselt man in den jeweiligen Modus und lädt die Adresse aus der Stack-Variable die im Linkerscript definiert ist.

Sobald diese Schritte abgearbeitet sind, erfolgt der Sprung(`bl main`) in die `main` Routine. In der `main` Routine gibt es noch eine wichtige Methode die dazu dient die Adressen der Interrupt-Handler in die vorherige erstellte Sprungtabelle zu mappen.

Code-Beispiel 6.8: Interrupt-Handler

```

1  #include "arm_init.h"
2
3  void arm_init()
4  {
5      uart_print("== Start map exception handler ==\n");
6      *(uint32_t volatile *)0x24 = (uint32_t)&ARM_undef;
7      *(uint32_t volatile *)0x28 = (uint32_t)&ARM_swi;
8      *(uint32_t volatile *)0x2C = (uint32_t)&ARM_pAbort;
9      *(uint32_t volatile *)0x30 = (uint32_t)&ARM_dAbort;
10     *(uint32_t volatile *)0x34 = (uint32_t)&ARM_reserved;
11     *(uint32_t volatile *)0x38 = (uint32_t)&ARM_irq;
12     *(uint32_t volatile *)0x3C = (uint32_t)&ARM_fiq;
13     uart_print("== Finished mapping exception handler\n");
14 }

```

Hier sieht man eindeutig wie an die Stellen `0x24 - 0x3c` die passenden Handler der Interrupts geschrieben werden.

6.4. Interrupt-Controller

Nachdem alle Exceptionhandler gemappt wurden startet eine neue Methode, die den Interrupt-Controller konfiguriert. Doch ehe man den Controller konfigurieren kann bedarf es eine Menge vorarbeit. Angefangen davon eine passende Struktur für den Controller zu erstellen. Auf Basis der Definition in `[vic]` kann man folgende Struktur definieren.

Code-Beispiel 6.9: VIC

```

1  typedef volatile struct
2  {
3      const uint32_t IRQStatus;
4      const uint32_t FIQStatus;
5      const uint32_t RawIntr;
6      uint32_t IntSelect;
7      uint32_t IntEnable;
8      uint32_t IntEnClear;
9      uint32_t SoftInt;
10     uint32_t SoftIntClear;

```

```

11  uint32_t Protection;
12  const uint32_t Reserved[3];
13  uint32_t VectAddr;
14  uint32_t DefVectAddr;
15  const uint32_t Reserved2[50];
16  uint32_t VectAddrs[32];
17  const uint32_t Reserved3[32];
18  uint32_t VectCntl[32];
19  const uint32_t Reserved4[800];
20  uint32_t VicAddress;
21  const uint32_t Reserved5[896];
22  uint32_t VICPeripheral[4];
23  uint32_t VICPrimecell[4];
24  } periph_primary_vic;

```

Weiter geht es mit der Definition einer globalen Variable, der eine fixe Adresse(0x10140000 [**archManI**]) im RAM zugewiesen wird. Somit ist es möglich dass die Struktur exakt auf die Stelle des VIC im ARM926EJ-S gemappt werden kann (Code-Beispiel 6.10).

Code-Beispiel 6.10: VIC Mapping

```

1  . = 0x10140000;
2  primary_vic =ABSOLUTE(.);

```

Jetzt sind alle Vorbedingungen geschaffen um den Controller zu konfigurieren. In der Abbildung 5.3 in E2 des Entwurf kann man erkennen welche Register man konfigurieren muss. Beispielhaft ist das in dem Code-Beispiel 6.11 zu erkennen.

Code-Beispiel 6.11: VIC Konfigurations Beispiel

```

1  void init_vic()
2  {
3
4  /** INIT TIMER AND ENABLE INTERRUPT IN VIC ** */
5  primary_vic.VectAddrs[1] = (uint32_t) &isr_Timer;
6  primary_vic.VectCntl[1] |= (uint32_t)(PVICSOURCE_TIMER | VIC_VECTOR_ENABLE);
7
8  vic_enable_as_fiq(TIMER_INTENABLE);
9  vic_enable_interrupt(TIMER_INTENABLE);
10
11 /** INIT UART0 AND ENABLE INTERRUPTS IN VIC ** */
12 primary_vic.VectAddrs[2] = (uint32_t)&isr_uart;
13 primary_vic.VectCntl[2] |= (uint32_t)(PVICSOURCE_UART0 | VIC_VECTOR_ENABLE);
14
15 vic_enable_as_irq(UART0_INTENABLE);
16 vic_enable_interrupt(UART0_INTENABLE);
17
18 }

```

Mit der fünften Zeile definiert man die Interrupt-Service Routine für den darauf folgenden Interrupt. In dem Controll-Register in Zeil sechs wird bestimmt welche Quelle der Interrupt hat und in Zeile acht wird der Timer-Interrupt als FIQ geschalten und abschließenden wird der Interrupt aktiviert.

6.5. Interrupt Service Routinen

Neben dem Interrupt Controller ist es auch wichtig die passenden Interrupt Service Routinen zu definieren. Beispielhaft soll hier die Service Routine für den *UART0* Interrupt analysiert werden.

Code-Beispiel 6.12: UART0 ISR

```

1  #include "uart.h"
2  #include "syscalls.h"
3  /*****
4  init_uart - This function handles initialisation in the uart, by
5              setting the constants used to defined the baud rate,
6              and also the control bits.
7  *****/
8
9  void isr_uart()
10 {
11
12     board_uart.InterruptClear |= UART_MASK_RXIM;
13     vic_clear_interrupt(UART0_INTENABLE);
14     mops_trap_writeC((char)board_uart.DataRegister);
15     vic_enable_interrupt(UART0_INTENABLE);
16 }

```

Sobald der Interrupt ausgelöst wurde wird die Methode ausgeführt. Nun sind ein paar wichtige Schritte notwendig um den Interrupt zu behandeln. Der erste Schritt ist, den Interrupt aus dem UART0 zu löschen, danach muss der Interrupt im VIC gelöscht werden. Danach wird die Methode ausgeführt die den gedrückten Buchstaben auf dem Monitor ausgibt. Nachdem das alles geschehen ist kann der Interrupt wieder aktiviert werden.

Die Vorgehensweise für neue Interrupt Service Routinen ist grundsätzlich die gleiche wie hier beschrieben wurde. Als erstes sollte der Interrupt im Gerät und dann im VIC gelöscht werden. Danach kann man benutzerdefinierte Funktionen aufrufen.

6.6. Syscalls

Wie im Entwurf bereits angesprochen ist es notwendig User-Prozessen die Möglichkeit zu gewähren auf Kernel-Methoden zuzugreifen. Um zu untermalen wie SWI's behandelt werden, folgt ein Auszug aus dem Quellcode von *MopS*.

Code-Beispiel 6.13: Software Interrupt Handler

```

1  #include "syscalls.h"
2
3
4  void mops_trap_handler(uint32_t trapNumber, uint32_t *sp)
5  {
6
7      switch(trapNumber)
8      {
9          // write to uart
10         case 0:
11             mops_trap_writeC_handler(*sp);
12             break;

```

```
13     case 1:
14         break;
15
16     default:
17         return;
18 }
19
20 }
21
22 void mops_trap_writeC(uint32_t character)
23 {
24     asm("swi 0x0");
25 }
26
27
28 void mops_trap_writeC_handler(uint32_t character)
29 {
30     uart_send_char((char)character);
31 }
```

Diese Routine wird von einem Handler aufgerufen der in Assembler geschrieben ist, die sogenannte ARM_swi Routine. Diese Routine ermittelt die Interruptnummer und schreibt sie in das Register R0 und ruft dann die den C-Handler auf. Angekommen im C-Handler, kann nun aufgrund der SWI-Nummer entschieden werden welcher Handler aufgerufen wird. In dem Fall das eine 0 als SWI-Nummer durchgeroutet wird, wird der Handler für die Ausgabe auf der Konsole aufgerufen. Um einen weiteren Handler hinzuzufügen bedarf es außerdem die SWI-Nummer in dem SWI-Handler einzutragen.

6.7. Prozessmanagment

Das Prozessmanagment stellte sich als größte Herausforderung bei **MopS** heraus. Es musste ein Mechanismus entwickelt werden mit dem man aus ARM-Compilierten C-Programmen den Assembler für den Code-Abschnitt extrahiert werden konnte. Nach dem kompilieren entstehen sogenannte .elf-Dateien, diese Dateien können mit einer Bibliothek Namens libelf² von **mr511** geparkt und bearbeitet werden. Leider unterstützt diese Bibliothek keine ARM-Formate, und somit konnte die Bibliothek nicht benutzt werden.

Es musste eine neue Möglichkeit entwickelt werden den Assemblercode aus der Output-Datei zu extrahieren.

Um das zu bewerkstelligen wurde das Tool **arm-none-linux-gnueabi-objcopy**B.1.4 benutzt. Mit diesem Tool kann man bestimmte Abschnitte aus einer Output-Datei kopieren. Mit diesem Wissen konnte nun der Code-Abschnitt in binär in eine neue Datei kopiert werden. Für folgendes Programm soll das einmal gezeigt werden.

²<http://www.mr511.de/software/index.html> Letzter Zugriff 13.07.2013

Code-Beispiel 6.14: Beispiel Programm

```

1
2 void klaus()
3 {
4     {
5         asm("mov r0, #33");
6         asm("swi 0x0");
7     }
8 }

```

Dieses Programm ist relativ einfach gehalten, es bewegt den Wert 33, was in ASCII für das Ausrufezeichen '!' steht, in das Register 0 und ruft dann den Syscall 0 auf. Das oben genannte Tool kann nun wie folgt benutzt werden um eine binäre Kopie von dem Code-Abschnitt des Programms zu erstellen.

```

1 arm-none-linux-gnueabi-objdump -O binary -S klaus.c klaus.bin

```

Schaut man sich nun die Datei im Hex-Editor an, so erhält man folgende Ausgabe:

Code-Beispiel 6.15: Binär Kopie vom Beispielprogramm

```

1 00000000: 04b0 2de5 00b0 8de2 2100 a0e3 0000 00ef  ..-.....!.....
2 00000010: 00d0 8be2 0008 bde8 1eff 2fe1 0a      ...../..

```

Diese Hex-Code ist für **MopS** relevant, denn diese Codes sind exakt die Assembler-Codes die der Assembler generiert. Da es auf dauer sehr umfangreich geworden wäre diese Codes per Hand rauszuschreiben musste also ein Weg entwickelt werden um dies automatisiert zu machen.

6.7.1. RAM-Disk

Die RAM-Disk ist der Ausgangspunkt der Prozesse in **MopS**. Sie wird nach Start des Systems in den Kernel-Heap geladen und ab dann können die Prozesse im System hochgefahren werden. Jedoch stellte sich die Frage wie man diese RAM-Disk erstellt. Zuvor wurde geklärt wie man an den Assembler-Code jedes Prozesse ran kommt. Nun muss dieser Code auch noch im für **MopS** passenden Format geschrieben werden. Dazu wurde ein Programm definiert was die binär Dateien einliest und daraus eine Header-Datei und passende C-Datei erzeugt. Ein Beispiel für so eine Header und C-Datei sieht kann wie folgt aussehen:

Code-Beispiel 6.16: RAM-Disk Headerdatei

```

1 #include<stdint.h>
2 extern const uint32_t imageDescriptor[3];
3 extern const uint32_t ramdisk[105];

```

Code-Beispiel 6.17: RAM-Disk C-Datei

```

1  #include "ramdisk.h"
2  const uint32_t imageDescriptor[3] = {88,7,7};
3  const uint32_t ramdisk[105] = {0xe92d0810,
4  0xe28db004,0xe24dd038,0xe59f3104,0xe5933000,0xe3530003,0xda000002,0xe59f30f4,0xe3a02000,0
5  xe5832000,0xe3a03000,0xe50b3008,0xea00002c,0xe59f10e0,0xe51b2008,0xe1a03002,0xe1a03083,
6  0xe0833002,0xe1a03183,0xe0813003,0xe24bc038,0xe1a04003,0xe8b4000f,0xe8ac000f,0xe8940003,0
7  xe88c0003,0xe51b3034,0xe3530000,0xa0000019,0xe51b3034,0xe3530002,0xa0000006,0xe3530003,
8  0xa0000001,0xe3530001,0x1a000005,0xe3a03002,0xe50b3034,0xea000002,0xe3a03003,0xe50b3034,0
9  xe1a00000,0xe59f106c,0xe51b2008,0xe1a03002,0xe1a03083,0xe0833002,0xe1a03183,0xe0813003,
10 0xe1a0c003,0xe24b4038,0xe8b4000f,0xe8ac000f,0xe8940003,0xe88c0003,0xe51b3008,0xe2833001,0
11 xe50b3008,0xe51b3008,0xe3530002,0xdaffffcf,0xe59f301c,0xe5933000,0xe2832001,0xe59f3010,
0xe5832000,0xe3a0002a,0xe24bd004,0xe8bd0810,0xe12fff1e,0,0,0xe92d4800,0xe28db004,0xe24dd008,0
xe50b0008,0xe51b2008,0xe1a03002,0xe1a03083,0xe0833002,0xe1a03183,
0xe59f2010,0xe0833002,0xe1a00003,0xebfffffe,0xe24bd004,0xe8bd8800,0,512,0xe52db004,
0xe28db000,0xe3a00021,0xef000000,0xe28bd000,0xe8bd0800,0xe12fff1e,512,0xe52db004,
0xe28db000,0xe3a00041,0xef000000,0xe28bd000,0xe8bd0800,0xe12fff1e,512};

```

Die Struktur für jeden Prozess in der RAM-Disk ist der folgende:

$$\text{Prozess} = x - \text{Bytes Code} + 1 \text{ Byte Stack}$$

Um die unterschiedlichen Prozesse voneinander abgrenzen zu können existiert noch ein imageDescriptor-Array was die Längen jedes Prozesses definiert. In dem Code-Beispiel bedeutet dass, der erste Prozess ist in den ersten 7 Stellen der RAM-Disk lokalisiert, dann kommt 1 Byte für den Stack und dann fängt der zweite Prozess an. Mit diesem Schema ist es möglich so viele Prozess wie gewollt in **MopS** zu laden und diese separat zu identifizieren. Für genauere Informationen wie der ramdiskMaker funktioniert, bitte im Anhang A.1 nachlesen.

6.7.2. MopS Loader

Neben des Mechanismus das passende Format für die RAM-Disk zu erstellen ist es weiterhin notwendig das geschriebene Format auch korrekt einzulesen und zu verarbeiten. Hier kommt der *Loader* von **MopS** ins Spiel. Der Name scheint im ersten Moment etwas verwirrend da er nicht wirklich das widerspiegelt was ein echter Loader macht, aber die Begrifflichkeit ist für das was er tut dennoch passend.

Code-Beispiel 6.18: **MopS** Loader

```

1  #include "mops_loader.h"
2  #include "thread.h"
3
4  void mops_load_ramdisk()
5  {
6      extern uint32_t __k_heap_start;
7      int length = sizeof(ramdisk) / sizeof(ramdisk[0]);
8
9      uint32_t* start = &__k_heap_start;
10     uint32_t* dst = &__k_heap_start;
11     int i = 0;
12     int j = 0;
13     int imageLength = sizeof(imageDescriptor) / sizeof(imageDescriptor[0]);

```

```

14
15     for(; i < imageLength; i++)
16     {
17         int bufferLength = imageDescriptor[i];
18         bufferLength += j;
19         // copy the essential assembler codes
20         for(; j < bufferLength; j++)
21         {
22             *dst++ = ramdisk[j];
23         }
24         int stackSize = ramdisk[j];
25         j++;
26         int k = 0;
27         // copy the essential stack size, just zeros
28         for(; k < stackSize; k++)
29         {
30             *dst++ = 0x0;
31         }
32         mops_create_thread_layout(start, dst);
33         start = (dst + 0x04);
34         dst = start;
35     }
36
37     threadTable[0].canBeScheduled = 0;
38 }
39

```

Hier wird dieselbe Technik angewendet wie beim erstellen der Sprungtabelle. Es wird sich auf eine externe Variable `__k_heap_start` bezogen um den Einstiegspunkt in den Kernel-Heap zu finden. Danach wird über das `imageDescriptor` Array herausgefunden wieviele Bytes kopiert werden müssen. Das kopieren ist dann ein sehr einfacher Mechanismus: Es wird ausschließlich der Zeiger auf den Kernel-Heap dereferenziert und der Wert aus der `ramdisk` reingeschrieben (siehe Zeile 22). Der Stack wird sehr einfach initialisiert, indem einfach nur der Wert `0x0` so oft reingeschrieben wird wie es in der RAM-Disk definiert wurde. Ist das getan muss noch der neue Start-Wert des Kernel-Heaps umgesetzt werden damit weitere Prozesse in den Heap geschrieben werden können (siehe Zeile 33-34).

6.7.3. Prozess-Layout

Nachdem der Prozess erfolgreich in den RAM geladen wurde war es an der Zeit das Prozess-Layout des Prozesses zu definieren. Als Vorlage dafür diente das Prozess-Layout von Dr. Prof. Burkhard Messer der HTW-Berlin. Er beschrieb in seinen Folien für die Vorlesung *Betriebssysteme* und dem Thema *Threads-1* ein Layout³ das bei **MopS** übernommen wurde. Der Entwurf für die **erste** Version sieht wie folgt aus:

Code-Beispiel 6.19: Prozess-Layout

```

1  #ifndef THREAD_H
2  #define THREAD_H
3
4  #define MAX_THREADS    3

```

³<http://wi.f4.htw-berlin.de/users/messer/LV/AI-BS-SS13/index.html> Letzter Besuch 13.07.2013


```
5
6 #include <stdint.h>
7
8 typedef struct ThreadLayout
9 {
10     uint32_t *start;
11     uint32_t *end;
12     uint32_t *sp;
13     uint32_t *pc;
14 } ThreadLayout;
```

Mit diesem Entwurf konnte nun ein rudimentärer Thread erzeugt werden. Hierzu musste die Start-Adresse, End-Adresse, Stackpointer und der Programmcounter gesetzt werden. Das war die Aufgabe der Methode die das Prozess-Layout erzeugt. Folgende Methode erfüllt genau diese Aufgabe:

Code-Beispiel 6.20: Prozess-Layout erstellen

```
1  #include "thread.h"
2  int mops_create_thread_layout(uint32_t* startAddr, uint32_t* endAddr)
3  {
4      int i = 0;
5      int maxId = 0;
6      int threadsFullCount = 0;
7      int emptyThreadTableIndex = 0;
8      for(; i < MAX_THREADS; i++)
9      {
10         // get the max id to increment for next thread
11         if(maxId <= threadTable[i].id)
12             maxId = threadTable[i].id;
13
14         if(threadTable[i].state != UNDEFINED)
15             threadsFullCount++;
16         else
17         {
18             emptyThreadTableIndex = i;
19             break;
20         }
21     }
22
23     if(threadsFullCount == MAX_THREADS)
24         return -1;
25     Thread t = threadTable[emptyThreadTableIndex];
26     t.data.start = startAddr;
27     t.data.end = endAddr;
28     t.data.sp = endAddr - 12;
29     t.data.pc = startAddr;
30     t.id = ++maxId;
31     t.state = NEW;
32     t.canBeScheduled = 1;
33     threadTable[emptyThreadTableIndex] = t;
34     return t.id;
35
36 }
```

Neben der Aufgabe dem Prozess die passenden Start-, End-, Stackpointer- und Programmcounterwerte zuzuweisen, fügt die Methode zudem noch den Prozess in eine globale Tabelle ein. In den Zeilen 23-29 kann man erkennen wie die Adressen zugewiesen werden.

6.7.4. Prozess Generierung

All diese Schritte sind notwendig um einen Prozess zu generieren. Der nächste logische Schritt ist jetzt den Prozess ins Leben zu rufen. Das geschieht in der **ersten** Version über folgende Assembler-Routine:

Code-Beispiel 6.21: Prozess Generierung

```

1  .text
2  .code 32
3
4
5  .global MOPS_resume
6  .func MOPS_resume
7  MOPS_resume:
8
9      mov r12,sp
10     // this loads the new stackpointer
11     ldr sp, [r0,#0x10]
12     stmfd sp!, {lr}
13     stmfd sp!, {r0-r12}
14
15     // this operation loads the startaddress of the new thread to r1
16     ldr r1, [r0,#0x14]
17     mov lr,pc
18     bx r1
19     // r0 contains the address to the thread,
20     // r0 + 0x08 -> thread start address
21     // r0 + 0x0c -> thread end address
22     // r0 + 0x10 -> thread sp address
23     // r0 + 0x14 -> thread pc
24     ldmfd sp!,{r0-r12}
25     str sp, [r0, #0x10]
26     ldmfd sp!, {pc}
27     .endfunc
28     .end

```

Diese Methode bekommt die Adresse des zu startenden Prozess übergeben und macht dann eine Reihe wichtiger Sachen:

1. Stackpointer des aktuellen Modus retten (Z. 9)
2. Stackpointer des neuen Prozess laden (Z. 11)
3. Alle Register auf den Stack des Prozess retten (Z. 12-13)
4. Die Startadresse es Programms laden (Z. 16)
5. In den Prozess springen (Z. 18)
6. Alle Register wieder herstellen und an den Aufrufer zurückkehren (Z. 24-26)

Dieses Schema wird für jeden neuen Prozess durchgeführt.

Fazit

Ziel der vorliegenden Arbeit war es, wie in der Einleitung beschrieben, einen Entwurf eines Betriebssystems mit maximal notwendigem Funktionsumfang, aber mit minimalem Aufwand, zu erschaffen.

Es wurden die wichtigsten, für ein Lehrmaterial notwendigen, Mechanismen umgesetzt und anhand des Entwurfes ist ein klares Bild von **MopS** entstanden. Was den Entwurf betrifft, konnte gezeigt werden, dass der Umfang einer minimalistischen Definition keine große Hürde darstellt. Andererseits musste festgestellt werden, dass die Umsetzung dieses Entwurf durchaus komplizierter war als Anfangs angenommen wurde. Dieses Ergebnis konnte deshalb evaluiert werden, weil die Implementationsphase sich fast bis zum Ende der Bachelorarbeit erstreckte. Dennoch kann im Rahmen des Projektes festgehalten werden, dass es faktisch möglich ist, solch ein Projekt zu realisieren. Dieses System kann also als Beispiel für die schnelle Implementation eines Betriebssystems angesehen werden.

Natürlich musste man sich auch klar von einigen Features distanzieren. Dies betreffend sollen hier nur exemplarisch die Stichpunkte Speichermanagement, Datei-System und Multiprozessorunterstützung genannt werden. Jedoch stellt sich auch die Frage, wie es in der Zukunft mit **MopS** weitergehen soll und welche Features noch umgesetzt werden sollen. Hierzu soll gesagt sein, dass dieses Projekt definitiv weiterverfolgt wird und weiterhin von Prof. Dr. Messer, im Rahmen seines Projektes **FOCOS - Family of Configured Operating Systems**, unterstützt wird. Mit einer weiteren Version soll zunächst die Code-Basis aufgeräumt und weiter optimiert werden, eine Unterstützung zum Starten von Prozessen zur Laufzeit und ein Speichermanagement angeboten werden.

Abschließend kann gesagt werden, dass die Entwicklung eines Betriebssystems im Rahmen einer Bachelorarbeit ein sehr komplexe Aufgabenstellung darstellt, andererseits aber einen umfangreichen Einblick und sehr viele neue Erfahrungswerte mit sich bringt. Einen positiven, erwähnenswerten Einfluss auf den Prozess und Erfolg der Arbeit hat Prof. Dr. Burkhard Messer, dank seines unverwechselbaren Interesses und seines immer kompetenten und dauerhaften Einsatzes.

Implementation

A.1. RAM-Disk Maker

Code-Beispiel A.1: RAM-Disk Maker

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <regex.h>
5
6  typedef uint16_t BUFFER_TYPE;
7  typedef uint32_t FINAL_BUFFER_TYPE;
8
9  const char* BUFFER_NAME = "uint32_t";
10
11 void writeRamDisk(FILE *ramdisk, FINAL_BUFFER_TYPE *buffer, unsigned long bufferSize, int
    stackSize)
12 {
13     for(int i =0; i < bufferSize+1 ; i++)
14     {
15         if(i != bufferSize)
16         {
17             fprintf(ramdisk, "%#x", buffer[i]);
18             fprintf(ramdisk, "%s", ",");
19         }
20         else
21             fprintf(ramdisk, "%i", stackSize);
22         if(i % 16 == 0)
23             fprintf(ramdisk, "%s", "\n");
24     }
25 }
26
27 void convertToUint32(BUFFER_TYPE *buffer, size_t length, FINAL_BUFFER_TYPE *newBuffer)
28 {
29     for(int i = 0, j = 0; i < length; i+=2,j++)
30     {
31         newBuffer[j] = ((FINAL_BUFFER_TYPE)buffer[i+1] << 16 | buffer[i]);
32     }
33 }
34
35 void readFile(FILE *ramdisk, const char* name)
36 {
37     FILE *file;
```

```

38  BUFFER_TYPE *buffer;
39  unsigned long fileLen;
40
41  file = fopen(name, "rb");
42
43  size_t bufferSize = sizeof(BUFFER_TYPE);
44  // file lenght
45  fseek(file,0, SEEK_END);
46  fileLen = ftell(file);
47  fileLen /= bufferSize;
48  rewind(file);
49
50  printf("Read %s with %ld bytes\n",name,fileLen);
51  buffer = (BUFFER_TYPE*)malloc(fileLen * bufferSize);
52  if(buffer)
53  {
54      size_t result = fread(buffer,sizeof(BUFFER_TYPE),fileLen ,file);
55      size_t bufferLength = (fileLen * sizeof(FINAL_BUFFER_TYPE)) / 2;
56      FINAL_BUFFER_TYPE *finalBuffer = (FINAL_BUFFER_TYPE*)malloc(bufferLength );
57      convertToUint32(buffer,fileLen, finalBuffer);
58
59      if(result != fileLen)
60      {
61          printf("Error reading file\n");
62          free(buffer);
63          free(finalBuffer);
64          return;
65      }
66      writeRamDisk(ramdisk,finalBuffer, bufferLength / sizeof(FINAL_BUFFER_TYPE), 512);
67      free(finalBuffer);
68      printf("Finished %s\n\n",name);
69      free(buffer);
70
71  }
72
73  fclose(file);
74
75 }
76
77 void writeImageDescriptor(char* fileNames[], int length)
78 {
79     FILE *ramdisk = fopen("ramdisk.c", "w+");
80     fprintf(ramdisk, "#include \"ramdisk.h\"\n");
81     fprintf(ramdisk, "const %s imageDescriptor[%i] = {",BUFFER_NAME,length-1);
82     int sum = 0;
83     for(int i = 1; i < length; i++)
84     {
85         FILE *binFile = fopen(fileNames[i], "rb");
86         fseek(binFile,0, SEEK_END);
87         size_t filelength = ftell(binFile);
88         filelength /= sizeof(FINAL_BUFFER_TYPE);
89         fprintf(ramdisk,"%i", filelength);
90         if(i != length - 1)
91             fprintf(ramdisk,",");
92
93         fclose(binFile);
94         sum += filelength+1;
95     }
96     fprintf(ramdisk,"};\n");
97     fclose(ramdisk);

```

```
98
99     ramdisk = fopen("ramdisk.h", "a+");
100     fprintf(ramdisk, "extern const %s ramdisk[%i];\n", BUFFER_NAME, sum);
101     fclose(ramdisk);
102     ramdisk = fopen("ramdisk.c", "a+");
103     fprintf(ramdisk, "const %s ramdisk[%i] = {", BUFFER_NAME, sum);
104     fclose(ramdisk);
105 }
106
107 int main(int argc, char* argv[])
108 {
109
110     if(argc > 1)
111     {
112         FILE* ramdisk = fopen("ramdisk.h", "w+");
113         fprintf(ramdisk, "#include<stdint.h>\n");
114         fprintf(ramdisk, "extern const %s imageDescriptor[%i];\n", BUFFER_NAME, argc-1);
115         fclose(ramdisk);
116
117         writeImageDescriptor(argv, argc);
118
119         ramdisk = fopen("ramdisk.c", "a+");
120
121         for(int i = 1; i < argc; i++)
122         {
123             readFile(ramdisk, argv[i]);
124             if(i < argc - 1)
125                 fprintf(ramdisk, ",");
126         }
127         fprintf(ramdisk, "};\n");
128         fclose(ramdisk);
129
130     }
131     else
132     {
133         printf("Not enough arguments\n");
134     }
135
136 }
```

Werkzeuge

B.1. Einleitung

Ohne vernünftige Werkzeuge ist es nicht möglich ein Betriebssystem oder eine andere Software zu entwickeln. Im folgenden wird beschrieben welche Werkzeuge bei der Entwicklung von **MopS** mit beteiligt waren und es wird ein kurzes Beispiel der Benutzung präsentiert. Für die Entwicklung von ARM-basierten Programmen wurde das Linux-Paket der GCC-Utills für ARM verwendet.

B.1.1. arm-none-linux-gnueabi-as

Das Fundament eines Betriebssystems besteht zu einem großen Teil aus Assembler Code, so ist es auch bei **MopS**. Damit dieser Code auch übersetzt werden kann bedarf es einen Assembler. Dieser Assembler nennt sich *arm-none-linux-gnueabi-as*. Durch folgendes Kommando kann eine Assembler Datei gegen die ARM926 Architektur assembliert werden.

Code-Beispiel B.1: ARM-Assembler mit Optionen für ARM926

```
1 arm-none-linux-gnueabi-as -mcpu=arm926ej-s -g startup.s -o startup.o
```

B.1.2. arm-none-linux-gnueabi-gcc

Neben Assembler spielt natürlich auch C eine wichtige Rolle in Betriebssystemen. So muss man mit dem GCC vorhanden .c Dateien auf folgende Weise kompilieren.

Code-Beispiel B.2: C/C++ Compiler

```
1 arm-none-linux-gnueabi-gcc std=c99 -mcpu=arm92ej-s -c -g file.c -o file.o
```

B.1.3. arm-none-linux-gnueabi-ld

Die Schritte des Assembler und Kompilieren reichen jedoch nicht um eine zusammenhängende Datei für den *qemu* zu erstellen. Dazu ist es noch notwendig alle Informationen zusammen zu linken. Das geschieht mit folgenden Kommando:

Code-Beispiel B.3: Linker mit Link-File 'link.ld'

```
1 arm-none-linux-gnueabi-ld -T link.ld first.o second.o -o mops.elf
```

Nun stellt sich die Frage nach welchen Regeln die Dateien verlinkt werden, an welchen Stellen im RAM welche Informationen stehen und wo z.B. die Stackpointer oder andere wichtige Informationen gesetzt werden. Diese Informationen erhält der Linker aus einem **Linkfile**.

B.1.4. arm-none-linux-gnueabi-objcopy

Dieses Werkzeug stammt ebenfalls aus den GCC-Utills und dient hat den Zweck, Informationen als Binar-Dump aus einer .o Datei zu extrahieren. Hierbei ist es möglich spezielle Sektionen zu kopieren, wie die Sektionen wo der Code lokalisiert ist. Dieses Tool ist weiterhin so konfigurierbar das sämtliche Informationen wie Relocationtabelle und/oder Symboltabelle entfernt oder mit übernommen werden können. Mehr Möglichkeiten sind hier nicht zu erwähnen da ausschliesslich die vorherigen genannten für die Bachelorarbeit relevant waren. Die Benutzung ist ebenfalls sehr simpel:

Code-Beispiel B.4: Objektkopie in Binärformat

```
1 arm-none-linux-gnueabi-objcopy -O binary -S mops.elf mops.bin
```

B.1.5. arm-none-linux-gnueabi-objdump

Dieses Tool stellt ein Interface bereit um spezifische Informationen einer Objektdatei auszugeben. Diese Informationen sind z.B. Sektionen, Symboltabellen, Relocationeinträge, Debuggingeinträge (sofern man jene mit einkompiliert hat) sowie die Disassemblierung von ganzen Sektionen. Diese Informationen können sehr hilfreich sein wenn es darum geht den generierten Assembler Code oder die Sektionen zu analysieren. Beispielsweise kann die Information über die Code-Sektion einer Objektdatei über folgenden Aufruf generiert werden:

Code-Beispiel B.5: Objdump einer Objektdatei

```
1 arm-none-linux-gnueabi-objdump -d object.o
```

B.1.6. make

make ist ein weitverbreitetes und bekanntes Programm zur Erstellung und zum Management von Build-Prozessen. Neben den ganzen obigen genannten Werkzeugen ist es jedoch notwendig das diese Prozesse zusammengefasst werden. Da bei der Entwicklung nicht immer nur eine Datei an dem Prozess beteiligt ist sondern mehrere ist es wichtig ein Mechanismus zu finden der das Assemblieren, Kompilieren, Linken und die Objektkopie zusammenfasst und sich mit einem Kommando ausführen lässt. Hier kommt *make* ins Spiel. Als ein Regelbasiertes System können Regeln und Abhängigkeiten definiert werden unter denen die auszuführenden Kommandos bestimmt werden. Einen kleinen Ausschnitt aus dem make-File des Projekts möchte ich hier anbringen:

Code-Beispiel B.6: make-File mit Hauptabhängigkeiten

```
1 TARGET=mops.elf
2 BIN=mops.bin
3 LINKFILE=link.ld
4
5 KLAUSNAME=klaus
6 CC=arm-none-linux-gnueabi-gcc
7 AS=arm-none-linux-gnueabi-as
8 LD=arm-none-linux-gnueabi-ld
9 OBJCOPY=arm-none-linux-gnueabi-objcopy
10 RAMDISK=./ramdiskMaker.o $(1)
11
12 CCFLAGS=-c -mcpu=arm926ej-s -g -Iinclude/devices -Iinclude/system -I.
13 CCLINKFLAGS= -nostdlib -nodefaultlibs -nostartfiles
14 ASFLAGS=-mcpu=arm926ej-s -g
15 LDFLAGS=-T
16 OBJCOPYFLAGS=-O binary -S
17
18 RM=rm -f $(1)
19
20 all: $(TARGET)
21     $(OBJCOPY) $(OBJCOPYFLAGS) $(TARGET) $(BIN)
22
23
24 rebuild: clean all proc
25
26 ##### Klaus stuff #####
```

Klar zu erkennen sind die wiederverwendbaren Kommandos wie *CC*, *AS*, *LD* usw. die dazu genutzt werden um zu assemblieren, kompilieren und zu linken. Neben diesen Kommandos gibt es noch Regeln um das Projekt komplett zu bauen und um alle unnötigen Objektdatei zu entfernen.