

ENTWICKLUNG EINES MINIBETRIEBSSYSTEMS AUF BASIS EINES ARM926 PROZESSORS

BACHELORARBEIT ZUR ERLANGUNG DES BACHELOR OF SCIENCE FÜR ANGEWANDTE INFORMATIK

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

9. August 2013

Inhaltsverzeichnis

1. Einleitung	2
2. Anforderungskatalog	4
2.1. Einleitung	4
2.2. Startmechanismus	5
2.3. Interruptquellen und Interrupthandler	5
2.4. Vectored Interrupt Controllor	5
2.5. Geräte	6
2.5.1. Timer	6
2.5.2. Serielle Schnittstelle - UART	6
2.5.3. Speicher	6
2.6. Dateisystem	6
2.7. Threadmanagment	7
2.8. qemu	7
2.9. Programmiersprachen	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	12
3.6. Interrupt-Controller	13
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	27
5.6.1. Threadlayout	27
5.6.2. Scheduling	29
6. Implementation	31
6.1. Einleitung	31
6.2. Entwicklungsumgebung	31
6.3. Startmechanismus	32
6.4. Geräte	36
6.5. Interrupt-Controller	38
6.6. Interrupt Service Routinen	40
6.7. Syscalls	40
6.8. Threadmanagment	41
6.8.1. RAM-Disk	42
6.8.2. <i>MopS</i> Loader	43
6.8.3. Threadlayout	44
6.8.4. Thread Generierung	46
7. Fazit	48
8. Eigenständigkeitserklärung	49
Literatur	50
Anhang	50
A. Implementation	51
A.1. RAM-Disk Maker	51
B. Werkzeuge	54
B.1. Einleitung	54
B.1.1. arm-none-linux-gnueabi-as	54
B.1.2. arm-none-linux-gnueabi-gcc	54

B.1.3.	arm-none-linux-gnueabi-ld	55
B.1.4.	arm-none-linux-gnueabi-objcopy	55
B.1.5.	arm-none-linux-gnueabi-objdump	55
B.1.6.	make	55

Tabellenverzeichnis

3.1. Unterschiedliche ARM Versionen	8
3.2. Vergleich RISC vs. CISC	10

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 - relevante Register	24
5.4. Konfiguration am Beispiel eines Timer Interrupt	24
5.5. Interrupt Service-Routinen Timer & UART0	25
5.6. SWI-Handler	26
5.7. Übersicht Threadmanagement	27
5.8. Thread-Image	28
5.9. RAM-Disk	28
5.10. RAM-Disk	29

Quellcode-Ausschnitte

6.1. Laden der Kerneldatei in qemu	32
6.2. Linker-Datei	33
6.3. Startup-Datei	33
6.4. Sprungtabelle erstellen I	34
6.5. Sprungtabelle erstellen II	34
6.6. Stack erstellen I	35
6.7. Stack erstellen II	35
6.8. Interrupt-Handler	36
6.9. Timer Struktur	36
6.10. Linker Konfiguration des Timer	37
6.11. Konfigurationsmethoden des Timers	38
6.12. VIC	38
6.13. VIC Mapping	39
6.14. VIC Konfigurations Beispiel	39
6.15. UART0 ISR	40
6.16. Software Interrupt Handler	41
6.17. Beispiel Programm	42
6.18. Binär Kopie vom Beispielprogramm	42
6.19. RAM-Disk Headerdatei	43
6.20. RAM-Disk C-Datei	43
6.21. MopS Loader	43
6.22. Threadlayout	45
6.23. Threadlayout erstellen	45
6.24. Thread Generierung	46
A.1. RAM-Disk Maker	51
B.1. ARM-Assembler mit Optionen für ARM926	54

B.2. C/C++ Compiler	54
B.3. Linker mit Link-File 'link.ld'	55
B.4. Objektkopie in Binärformat	55
B.5. Objdump einer Objektdatei	55
B.6. make-File mit Hauptabhängigkeiten	56

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.«[Tan03, S. 911]

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 was 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 vorhandenen 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. Mobil Telefone, 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.« [Tan03, S. 915]

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.

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 Threadmanagment 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** vorrangig im Bereich der Handhelds eingesetzt werden soll ist es für die Arbeit nicht relevant eine MMU - Memory Managment 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. Startmechanismus

Der Startvorgang stellt in jedem Betriebssystem eine wichtige Rolle dar. Die Möglichkeiten bei **MopS** stellten sich hier, durch die Non-Existenz von Hardware, als begrenzt dar. Um diese Problematik zu lösen, wurde die Entwicklung in einen Emulator verlagert. Der Emulator wurde mit einer Datei befüllt die aus dem Entwicklungsprozess von **MopS** entstanden ist. Diese Datei beinhaltet alle Informationen die **MopS** ausmachen, so auch die Startroutinen.

2.3. 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[ARM05, S. 53].

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 Tabelle während der Initialisierung des Systems angelegt werden. Was **MopS** also benötigt ist ein System zur Behandlung und Konfiguration von Interrupts und Interruptquellen.

2.4. Vectored Interrupt Controller

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.

2.5. Geräte

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

2.5.1. 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[vgl. ARM05, S. 262]. Da es für **MopS** ausreicht nur auf einen Timer zurück zu greifen wird später nur noch von *Timer0* die Rede sein.

2.5.2. 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.

2.5.3. Speicher

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

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.

2.6. Dateisystem

In **MopS** gibt es **kein** Dateisystem. Da die Daten und der Code von Threads dennoch in das System kommen müssen, wurde hier eine Ersatzlösung gewählt, indem eine RAM-

Disk erstellt wird. Diese RAM-Disk dient dazu um den Code der Threads in das System zu laden.

2.7. 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. Ein Scheduler kann auf verschiedene Art und Weißen entwickelt werden. Es wurde sich hier aber auf eine 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.

2.8. 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 für *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.

2.9. Programmiersprachen

MopS wird ausschließlich in ARM-Assembler und C programmiert. Sämtliches Wissen wurde entweder aus vergangenen Lehrveranstaltungen der HTW-Berlin oder dem Buch *The C Programming Language*[Bri88] bezogen.

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, entstand. 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. Das bedeutet z.B. das eine RISC-Maschine nur Speicher-Register Operationen durchführt, sowie Berechnungen nur auf Registern und nicht im Speicher stattfindet. 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. Daraus lässt sich doch schließen das sich RISC eigentlich gegenüber CISC auf dem Markt hätte durchsetzen müssen. Doch das war nicht so. Warum? Tanenbaum hat dazu in seinem Buch über *Structured computer organisation* folgendes geschrieben:

»First of all, there is the issue of backward compatibility and the billions of dollars companies have invested in software for the Intel line. Second, surprisingly, Intel has been able to employ the same ideas even in a CISC architecture.«

[Tan05, S. 80]

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, dass mit weniger Befehlen umfangreichere Operationen durchgeführt werden können. Der Nachteil dabei ist jedoch, dass die Performance der Befehlsausführung, geringer als bei einem RISC ausfallen kann.

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.

	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

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 nicht auf diese Art von Prozessoren.

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 ARM926EJ-S Prozessor ist ein 32-Bit RISC Prozessor. Dieser Prozessor hat eine Gesamtzahl von 37 Registern[vgl. ARM05, S. 44 ff.], 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:[ARM05, S. 43]

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 per Programmcode oder durch einen Interrupt betreten werden:

➤ **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:[ARM05, S. 224]

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, ob ein Interrupt als IRQ oder FIQ ausgelöst werden 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 zurückgesetzt 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.

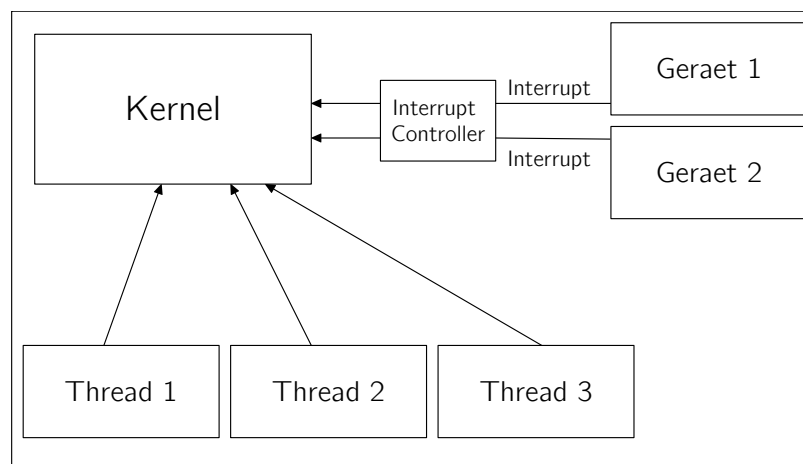


Abbildung 4.1.: **MopS** Überblick

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.

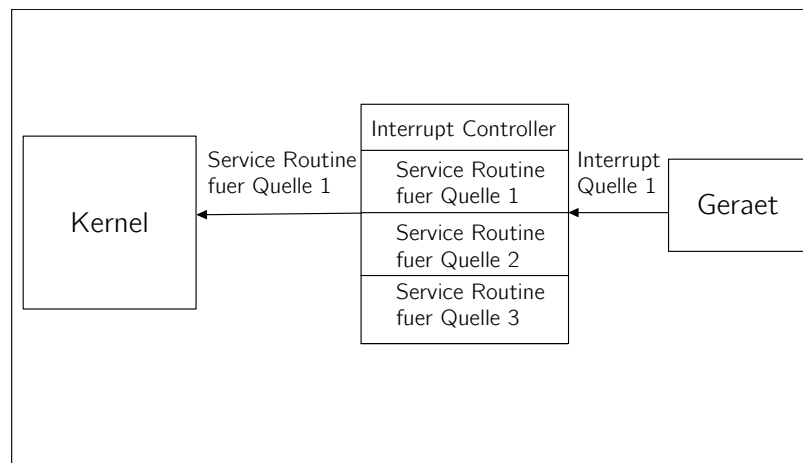


Abbildung 4.2.: Interrupt-Controller

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:

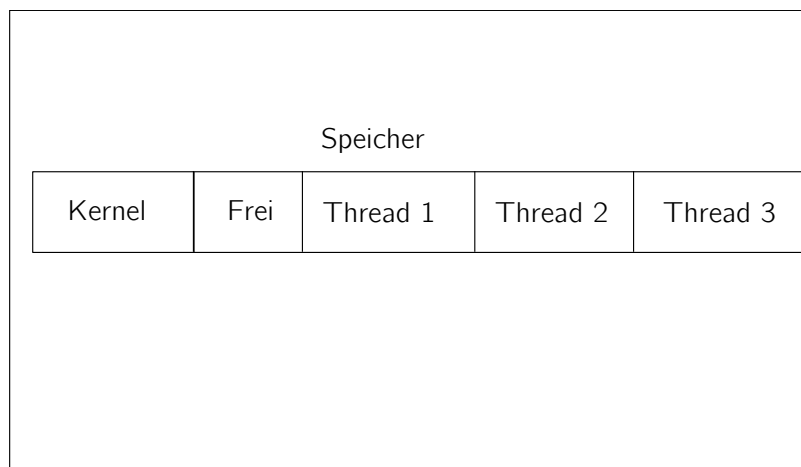


Abbildung 4.3.: Thread-Layout im RAM

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.**

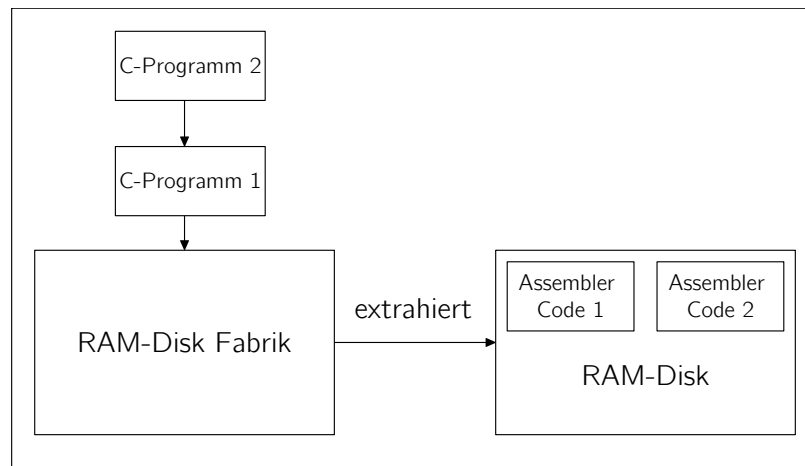


Abbildung 4.4.: Erstellung der RAM-Disk

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.

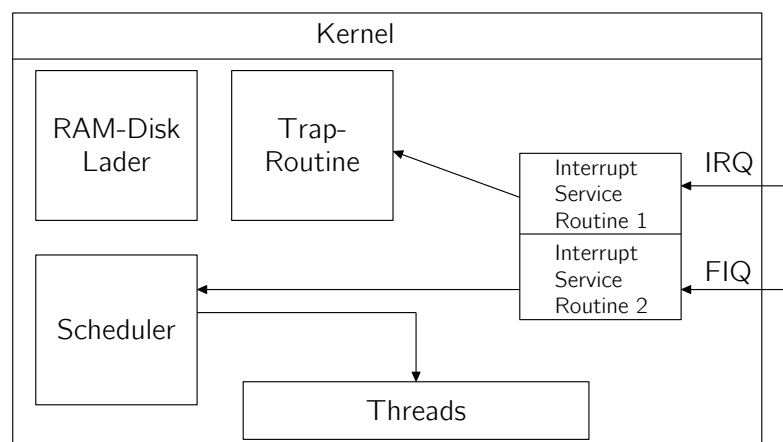


Abbildung 4.5.: Kernel - Überblick

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.

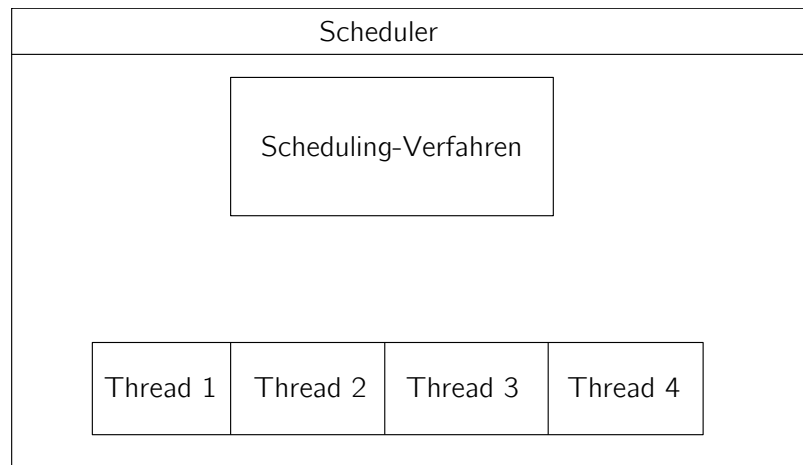


Abbildung 4.6.: Scheduler

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.

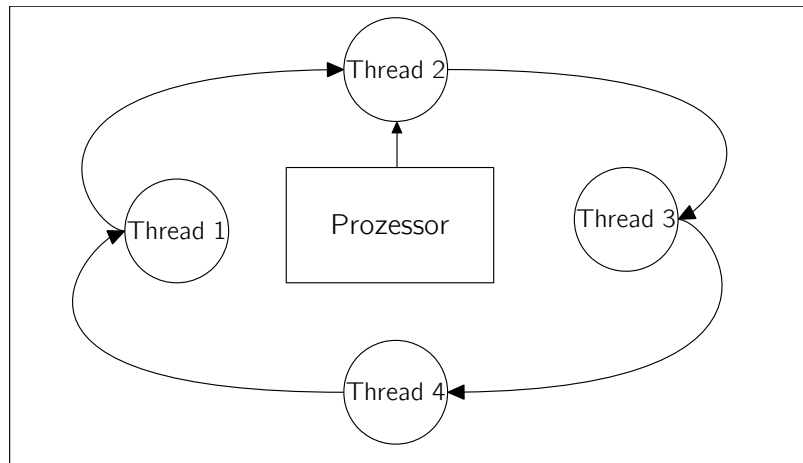


Abbildung 4.7.: Round-Robin Verfahren - Schematisch

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:

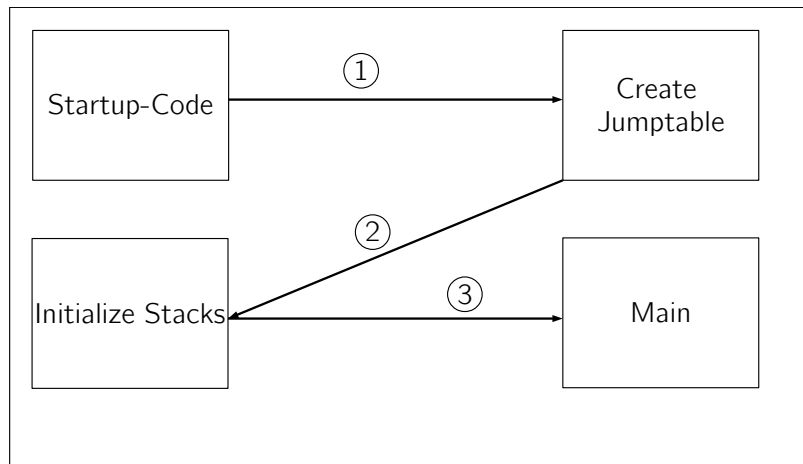


Abbildung 5.1.: Kernel-Image Version 1

1. Startup-Code

Der Emulator springt an die Adresse `0x000000`¹. An dieser Stelle steht die erste Assemblerroutine des Betriebssystems. Diese Routine dient dazu, 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 [vgl. ARM05, S. 54].

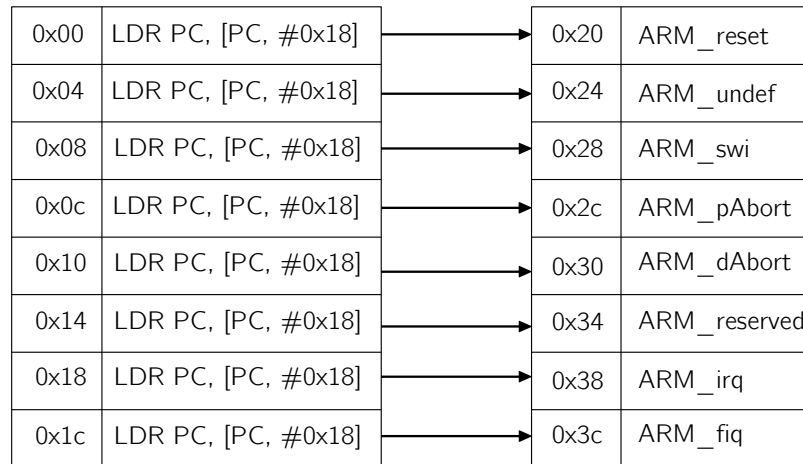


Abbildung 5.2.: Erzeugen der Sprungtabelle

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

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

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

3. Erstellung der Stacks mit anschließendem 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 eines Interrupts der Prozessor in den jeweiligen Modus wechselt und, wenn kein valider Stackpointer vorhanden ist, kann es zu undefiniertem 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 wurden, erfolgt der Sprung in die **main** Routine des Betriebssystems. Ab diesem Punkt finden nun weitere Schritte statt, um das System fertig zu initialisieren.

5.3. Interrupt-Controller

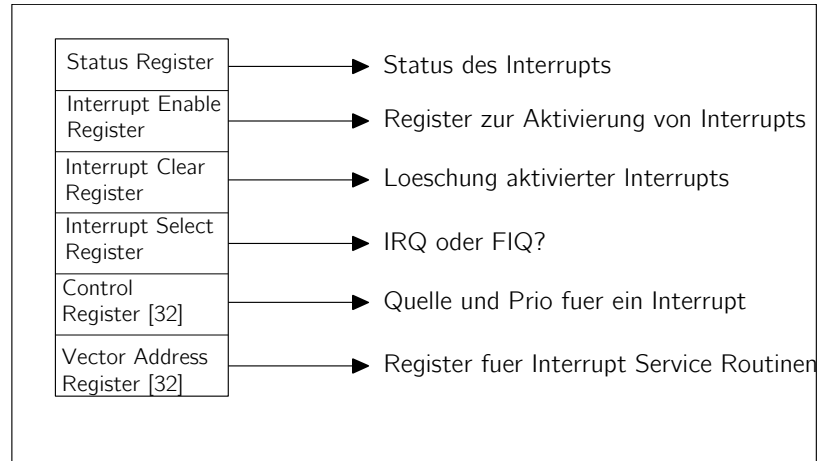
Ein Interrupt-Controller stellt in einem Betriebssystem die Schnittstelle zwischen den Interrupts und der Hardware dar. Er priorisiert die Interrupts, die von externen wie auch internen Quellen ausgelöst werden können, und leitet sie an das Betriebssystem weiter. Es gibt zwei Arten von Controllern:

- 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 direkt benutzen kann. 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 lostreten, 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. Die folgende Abbildung ist eine schematische Darstellung des Vectored Interrupt-Controller.

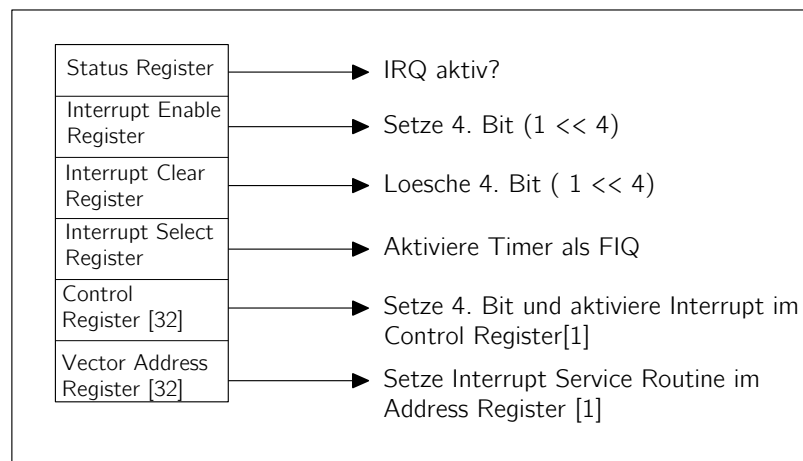
Abbildung 5.3.: Vectored Interrupt Controller - relevante Register



Sofern das Board einen Vectored Interrupt-Controller zur Verfügung stellt, ist dieser an einer bestimmten Adresse lokalisiert. Bei dem von **MopS** emulierten System ist dies die Adresse: 0x10140000[vgl. ARM05, S. 223]. 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 Register für die Interrupt Vektoren wie auch die Control Register für die jeweiligen Interrupts [vgl. ARM04b, S. 35] vorhanden.

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. Zudem kann konfiguriert werden, ob der 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.: Konfiguration am Beispiel eines Timer Interrupt



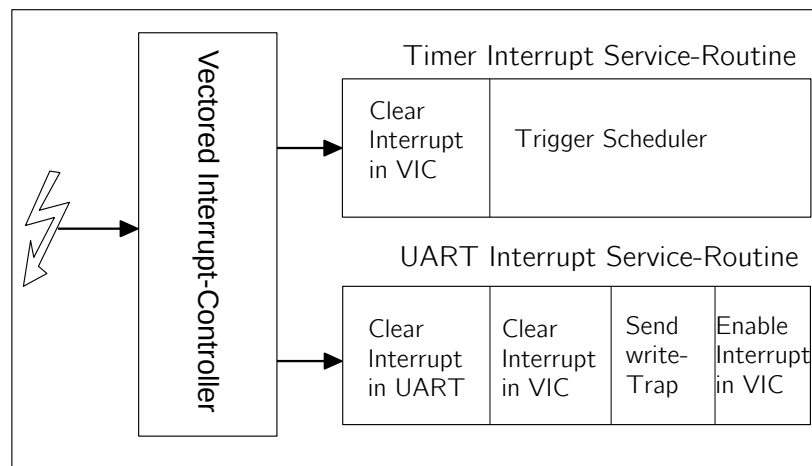
Dieses Beispiel zeigt eine Beispielkonfiguration des Timerinterrupts. Um diesen Interrupt zu konfigurieren, sind vier Schritte notwendig:

- Aktivierung des Interrupts im *Interrupt Enable Register*. Für den Timer-Interrupt muss hier das 4. Bit gesetzt werden [vgl. Tabelle 4-40 ARM05, S. 227]
- Das *Interrupt Select Register* muss auf 1 gesetzt werden, um den Timer Interrupt als FIQ zu konfigurieren [vgl. Tabelle 3-5 ARM04b, S. 39].
- In dem *Control Register* muss an der Array Stelle 1 das 4. Bit gesetzt werden, um die Quelle von dem Timer-Interrupt zu definieren.
- In dem *Vector Address Register* muss an der Array Stelle 1 die Adresse der Interrupt Service Routine, für den Timer-Interrupt, eingetragen werden.

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

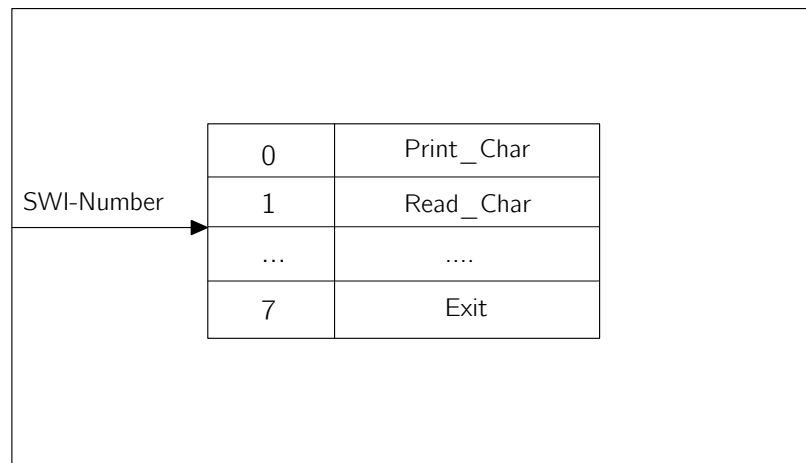
wird. Zum einen ist das der *Timer0*-Interrupt und zum anderen der Interrupt des *UART0*-Interface. Bei Bedarf kann man auch noch weitere Interrupts definieren, dazu muss jedoch die Konfiguration des VIC angepasst werden.

5.5. Syscalls

Neben den IRQ und FIQ spielen die Syscalls auch noch eine sehr wichtige Rolle. Um einen Syscall aufzurufen, ist es notwendig, eine sogenannte SWI - *Software Interrupt* Instruktion auszuführen. Diese Instruktion wird von einem Handler aufgefangen und dort wird entschieden, welcher Syscall ausgeführt wird.

Ein SWI ist eine Instruktion, 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 einer SWI-Instruktion die *ARM_swi* Routine angesprungen. Diese Routine leitet den Request an eine weitere Komponente weiter, die schematisch wie folgt aussieht.

Abbildung 5.6.: SWI-Handler



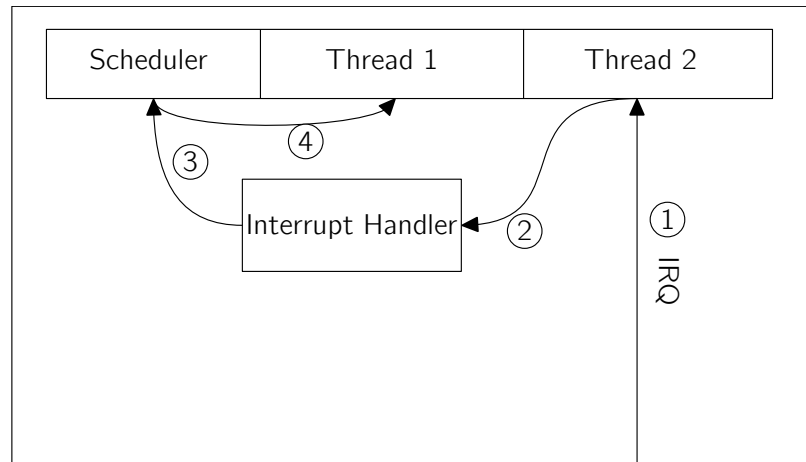
In der aktuellen Fassung von **MopS** ist nur der SWI-Handler für den *Print_Char*-SWI definiert. Es sollen jedoch Weitere folgen. Ein Syscall hat immer eine gewisse Anzahl von Parametern. Da für **MopS** nur *Print_Char* implementiert wurde, reicht es hier nur die Parameter von dieser Methode zu beschreiben.

Der Parameter für den beschriebenen Syscall besteht im Großen und Ganzen nur aus einem Parameter: Der Charakter, der auf der Console ausgegeben werden soll.

5.6. Threadmanagment

Das Threadmanagment stellt die Verwaltung der Threads in einem System dar. Das bedeutet: Wie wechseln sich die Threads ab und wodurch können sie unterbrochen werden. Wie diese Prozesse miteinander arbeiten, soll in der folgenden Grafik verdeutlicht werden.

Abbildung 5.7.: Übersicht Threadmanagement



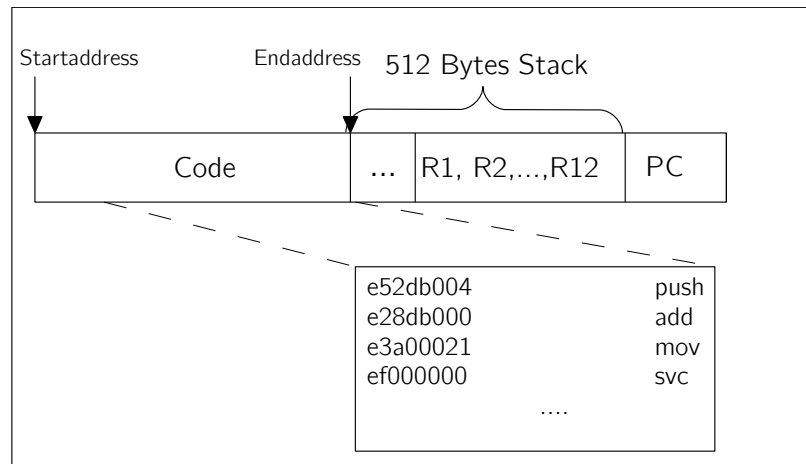
Die Ausgangssituation ist ein System, in dem drei Threads koexistieren können. Jeder dieser Threads hat, aufgrund des Scheduling-Verfahren, nur eine gewisse Zeit in der er arbeiten darf. Nun unterbricht ein Interrupt den Thread –hier mit 1 beschrieben. Der Thread wird unterbrochen und es wird der Interrupthandler für diesen Interrupt aufgerufen –z.B. der Timer-Interrupthandler. Bei Punkt zwei ist dieser Interrupt-Handler dargestellt. Er ruft im dritten Schritt den Scheduler auf. Der Scheduler ermittelt dann, basierend auf dem Scheduling-Verfahren, den nächsten Thread, der die Kontrolle bekommt, was im 4. Schritt stattfindet.

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 rudimentäres System gesetzt. Das bedeutet, dass keine Threads zur Laufzeit des Systems geladen werden können, sondern die Threads vor Beginn definiert werden mussten.

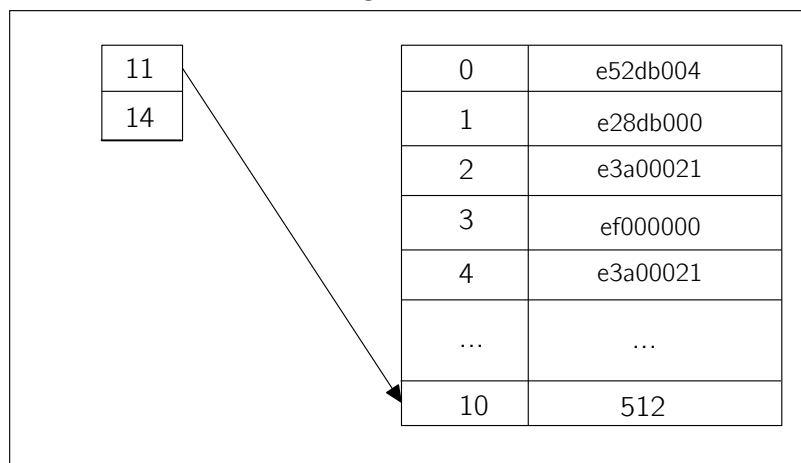
Eine abstrakte Darstellung dieses Thread-Image kann man sich wie folgt vorstellen:

Abbildung 5.8.: 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 5.9.: 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

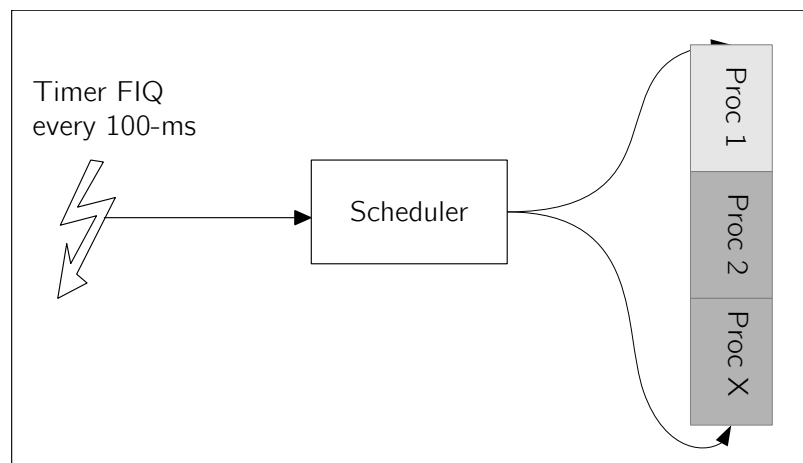
vorerst nur mit Nullen aufgefüllt.

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 eines jeden einzelnen Threads eingetragen sind. Für Abbildung 5.9 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 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 5.10.: RAM-Disk



In Grafik 5.10 wird veranschaulicht wie der Mechanismus, des Thread-Umschaltens, in **MopS** umgesetzt wurde. Hier sieht man, dass aller 100ms der Timer-Interrupt ausgelöst wird und dieser den Scheduler startet. Der Scheduler sucht dann den nächsten wartenden Prozess - hier dunkelgrau - heraus und schaltet diesen ein. Der andere - hier hellgrau - 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 Warteschlange an der Post vorstellen. Prozesse werden in eine Queue² gepackt und von dort bearbeitet. Dieses System hat

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

jedoch den Nachteil, dass lange Wartezeiten durch Prozesse, die die CPU sehr überdurchschnittlich lange in Anspruch nehmen, entstehen können.

➤ **Shortest-Job-First**

Bei diesem Verfahren wird darauf abgezielt, den Prozessen die CPU-Zeit zu überlassen welche am kürzesten sind. Wie in [Sil09, S. 189] geschrieben

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

versucht dieser Algorithmus anhand der CPU-Bursts[vgl. Sil09, S. 184] zu ermitteln, wieviel Zeit ein Prozess in etwa benötigen wird. Anhand dieser Informationen wird 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 ist natürlich, dass es wesentlich optimaler als die vorherig genannten ist, jedoch auch komplizierter zu implementieren ist.

➤ **High-Priority First**

Mit diesem Verfahren ist es möglich Threads mit Prioritäten zu versehen und anhand dieser Prioritäten kann der Scheduler dann den nächsten Thread ermitteln, der die CPU-Zeit bekommt. Aber auch hier gibt es einen Nachteil: Was passiert wenn alle Threads eine hohe Priorität aufweisen? Das bedeutet, dass dieses Verfahren nur in Kombination mit einem anderen Verfahren umsetzbar ist, denn es muss noch eine weitere Eigenschaft geben anhand der ein Thread ausgewählt werden kann.

➤ **Round-Robin**

Dieses Verfahren »*is designed especially for time-sharing systems.*«[vgl. Sil09, S. 194] Das bedeutet, dass 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, dass der Scheduler immer wieder jeden Prozess kurz startet. Der Vorteil dieses Mechanismus ist, dass jeder Prozess gleichermaßen bewertet wird und das es relativ einfach zu implementieren ist. Jedoch bringt genau dieser Vorteil auch einen Nachteil mit sich, nämlich, dass Prozesse, die eigentlich eine längere Zeitscheibe bräuchten, immer warten müssen bis sie wieder am Zug sind. Das kann natürlich zu sehr hohen Latenzen in der Ausführung führen.

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 Entscheidungen 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-basiertem Code kann die Entwicklungsumgebung Eclipse¹ genutzt werden. Dennoch wurde sich für den konservativen Weg entschieden und die Entwicklung läuft seither 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

¹<http://www.eclipse.org/>

IDE neben den eigentlich notwendigen noch durchführt. Da *vim* ein rein textbasierter Editor ist, kann man hier sicher sein, dass keine unklaren Prozesse im Hintergrund stattfinden.

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

Nachteile:

➤ **Komplex**

vim ist kein Werkzeug für Anfänger. Die Verwendung findet auf einem sehr rudimentären Level statt und es bedarf einer gewissen Zeit diese gut zu beherrschen. Herkömmliche IDE's, wie Eclipse, sind da wesentlich komfortabler.

➤ **Unintuitiv**

Die Benutzung eines rein textbasierten Editors, wie *vim*, ist insofern nachteilig, dass sämtliche Features einer IDE, wie Autovervollständigung, Intellisense, Fehlermeldungen während des Schreibens etc., verloren gehen. Weiterhin kommt dazu, dass 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 Herausforderung bei **MopS** war die Definition des Startprozesses. Dies umschließt:

➤ **Was bedeutet *Startprozess*?**

Die Frage zur Bedeutung des Startprozesses 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 *gemu*!

Code-Beispiel 6.1: Laden der Kerneldatei in *gemu*

```
1  gemu-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[GNU] 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 Interrupthandler mappt. Weiterhin ist in Zeile 26 der, in Abbildung 5.1 (2), erkennbare Sprung in die Methode, die die Stacks erzeugt. Nicht zuletzt erfolgt 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.

➤ Interrupthandler erstellen

Beim Erstellen der Sprungtabelle für die Interrupts kommt es darauf an, dass die Sprungadresse korrekt mit den passenden Methoden für die jeweiligen Interrupts befüllt werden. In der Abbildung 5.2 erkennt man, dass 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 Stellen, 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 Interrupthandler 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 gewünschte Ergebnis: `LDR PC, [PC, #0x18]`. Mit diesem Wissen kann man nun die Sprunganweisungen erstellen, die das notwendige Schema widerspiegeln.

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  . += 0x2000;
2  __sys_stack_top = .;
3
4  . += 0x2000;
5  __irq_stack_top = .;
6
7  . += 0x2000;
8  __fiq_stack_top = .;
9
10 . += 0x2000;
11 __svc_stack_top = .;
12
13
14 __k_heap_start = .;
15 . += 0x2000;
16 __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 sagt z.B. aus, dass 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 in das 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
14     // the initial mode)
15     mov pc, r1
16     .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, dass **MopS** nur die Modi **System**, **Supervisor**, **Interrupt** und **Fast Interrupt** als wichtig betrachtet, sind nur vier Stackpointer zu setzen. Damit das korrekt vonstatten 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. Geräte

Da die Konfiguration der Geräte fast immer gleich ist, wird im Folgenden nur die Konfiguration des Timers gezeigt.

Für den Timer wurde sich aus dem Manuel[vgl. Tabelle 3-1 ARM04a, S. 34] bedient und folgende Struktur entwickelt:

Code-Beispiel 6.9: Timer Struktur

```

1  #ifndef VERSATILE_TIMER_H
2  #define VERSATILE_TIMER_H
3
4  #include <stdint.h>
5  #include "uart.h"
6  #include "scheduler.h"
7
8  #define TIMER_INTENABLE (1<<4)
9  #define TIMER_CONTROL_MASK 0xEF
10 #define TIMER_CONTROL_ENABLED 0x80
11 #define TIMER_CONTROL_DISABLED 0x0
12 #define TIMER_CONTROL_MODE_PERIODIC 0x40
13 #define TIMER_CONTROL_MODE_FREE 0x0
14 #define TIMER_CONTROL_INT_ENABLED 0x20
15 #define TIMER_CONTROL_INT_DISABLED 0x0
16 #define TIMER_CONTROL_DIVIDE_1 0x0
17 #define TIMER_CONTROL_DIVIDE_16 0x4
18 #define TIMER_CONTROL_DIVIDE_256 0x8
19 #define TIMER_CONTROL_SIZE_16 0x0
20 #define TIMER_CONTROL_SIZE_32 0x2
21 #define TIMER_CONTROL_WRAPPING 0x0
22 #define TIMER_CONTROL_ONE_SHOT 0x1
23
24 typedef volatile struct
25 {
26     uint32_t TimerLoad;
27     const uint32_t TimerValue;
28     uint32_t TimerControl;
29     uint32_t TimerIntClr;
30     uint32_t TimerRIS;
31     uint32_t TimerMIS;
32     uint32_t TimerBGLoad;
33 } periph_timer;
34
35 typedef volatile struct
36 {
37     periph_timer Timer1;
38     periph_timer Timer2;
39 } periph_dualtimer;
40
41 extern periph_dualtimer dual_timer;
42
43 void init_timer(void);
44 void timer_clear_interrupt(void);
45 void isr_Timer(void);
46
47 #endif /* !defined VERSATILE_TIMER_H */

```

Diese Struktur wurde über das Linker-Script an eine vordefinierte Stelle gesetzt. Diese Adresse ist: 0x101E2000[vgl. Tabelle 4-72 ARM05, S. 262]. Im Linker-Script ist dann direkt erkennbar, dass diese Adresse auf eine globale Variable für den Timer gemappt wurde.

Code-Beispiel 6.10: Linker Konfiguration des Timer

```

1  . = 0x101E2000;
2  dual_timer = ABSOLUTE(.);

```

Weiterhin wurden Funktionen entwickelt mit denen man den Timer initial konfigurieren, Interrupts zurücksetzen und den Scheduler anspringen kann.

Code-Beispiel 6.11: Konfigurationsmethoden des Timers

```

1
2  #include "vic.h"
3  #include "timer.h"
4
5
6  void isr_Timer(void)
7  {
8      timer_clear_interrupt();
9      mops_resume(0);
10 }
11
12 /*****
13 timer_clear_interrupt - This function writes to the timer
14      interrupt clear register in the timer.
15 *****/
16
17 void timer_clear_interrupt(void)
18 {
19     dual_timer.Timer1.TimerIntClr = 0x0;
20     vic_clear_interrupt(TIMER_INTENABLE);
21     vic_enable_interrupt(TIMER_INTENABLE);
22 }
23
24 /*****
25 init_timer - This function initialises the timer whos base
26      address is defined externally, i.e. in a scatter or
27      header file.
28 *****/
29
30 void init_timer(void)
31 {
32     dual_timer.Timer1.TimerIntClr = 1;
33     dual_timer.Timer1.TimerLoad = 1000000;
34     dual_timer.Timer1.TimerControl |=
35         (TIMER_CONTROL_MODE_PERIODIC |
36          TIMER_CONTROL_INT_ENABLED |
37          TIMER_CONTROL_ENABLED);
38     uart_print("== Timer configured ==\n");
39 }

```

Die Funktion `init_timer` wird von der `main`-Funktion aufgerufen, um den Timer initial zu konfigurieren.

6.5. Interrupt-Controller

Nachdem alle Interrupthandler gemappt wurden, startet eine neue Methode, die den Interrupt-Controller konfiguriert. Doch ehe man den Controller konfigurieren kann, bedarf es einer Menge Vorarbeit. Auf Basis der Definition in [ARM04b, S. 35] kann man folgende Struktur definieren.

Code-Beispiel 6.12: 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 [vgl. Tabelle 4-37 ARM05, S. 223]) 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 6.13).

Code-Beispiel 6.13: VIC Mapping

```

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

```

Jetzt sind alle Vorbedingungen geschaffen, um den Controller zu konfigurieren. In der Abbildung 5.3 des Entwurfs kann man erkennen, welche Register man konfigurieren muss. Beispielhaft ist das in dem Code-Beispiel 6.14 zu erkennen.

Code-Beispiel 6.14: 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 Zeile sechs wird bestimmt, welche Quelle der Interrupt hat und in Zeile acht wird der Timer-Interrupt als FIQ geschaltet und abschließend wird der Interrupt noch aktiviert.

6.6. 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 6.15: 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. Daraufhin wird die Methode ausgeführt, die den gedruckten 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.

6.7. 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 6.16: 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         case 11:
16             mops_trap_schedule_handler(sp);
17         default:
18             return;
19     }
20 }
21
22
23 void mops_trap_writeC(uint32_t character)
24 {
25     asm("swi 0x0");
26 }
27
28
29 void mops_trap_writeC_handler(uint32_t character)
30 {
31     uart_send_char((char)character);
32 }
33
34 void mops_trap_schedule_handler(uint32_t* sp)
35 {
36     mops_resume(*sp);
37     // MOPS_resume(address);
38 }

```

Diese Routine wird von einem Handler aufgerufen, der in Assembler geschrieben ist, die sogenannte *ARM_swi* Routine. Diese Routine ermittelt die Interruptnummer, schreibt sie in das Register R0 und ruft dann den C-Handler auf. Angekommen im C-Handler kann nun aufgrund der SWI-Nummer entschieden werden, welcher Handler aufgerufen wird. Für den Fall, dass 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 der SWI-Nummer in dem SWI-Handler einzutragen.

6.8. Threadmanagment

Das Threadmanagment stellte sich als größte Herausforderung bei **MopS** heraus. Es musste ein Mechanismus entwickelt werden mit dem man aus ARM-Kompilierten C-Programmen den Assembler - für den Code-Abschnitt - sextrahiert kann. 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 im Binärformat in eine neue Datei kopiert werden. Für folgendes Programm soll das einmal gezeigt werden.

Code-Beispiel 6.17: Beispiel Programm

```

1
2 void klaus()
3 {
4 while(1){
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-objcopy -O binary -S klaus.c klaus.bin

```

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

Code-Beispiel 6.18: 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-Codes sind 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 herauszuschreiben, musste also ein Weg entwickelt werden, um dies automatisiert zu machen.

6.8.1. RAM-Disk

Die RAM-Disk ist der Ausgangspunkt der Prozesse in **MopS**. Sie wird nach Start des Systems in den Kernel-Heap geladen und von da an können die Threads im System hochgefahren werden. Jedoch stellte sich die Frage, wie man diese RAM-Disk erstellt. Zuvor wurde geklärt, wie man an den Assembler-Code eines jeden Prozesses herankommt. Nun muss dieser Code auch noch in einem für **MopS** passenden Format geschrieben werden. Dazu wurde ein Programm definiert, was die Binärdateien einliest und daraus

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

eine Header-Datei und passende C-Datei erzeugt. Ein Beispiel für so eine Header- und C-Datei kann wie folgt aussehen:

Code-Beispiel 6.19: RAM-Disk Headerdatei

```

1 #include <stdint.h>
2 extern const uint32_t imageDescriptor[3];
3 extern const uint32_t ramdisk[61];

```

Code-Beispiel 6.20: RAM-Disk C-Datei

```

1 #include "ramdisk.h"
2 const uint32_t imageDescriptor[3] = {48,5,5};
3 const uint32_t ramdisk[61] = {0xe52db004,
4 0xe28db000,0xe24dd00c,0xe3a03001,0xe50b3008,0xe51b3008,0xe3530003,0xda000001,0xe3a03001,
5 0xe50b3008,0xe59f0068,0xe28bd000,0xe8bd0800,0xe12fff1e,0xe92d4800,0xe28db004,0xe24dd008,
6 0xe50b0008,0xe59f1044,0xe51b2008,0xe3a03018,0xe3a0005c,0xe0020290,0xe0812002,0xe0823003,
7 0xe5933000,0xe3530001,0x1a000006,0xe51b3008,0xe3a0205c,0xe0020293,0xe59f3010,0xe0823003,
8 0xe1a00003,0xeb000003,0xe24bd004,0xe8bd8800,0xc0,0xc0,0xe92d5fff,0xe280101c,0xe1a00001,0xe92d0001,
9 0xe8b01fff,0xe8bd0001,0xe1a0e00f,0xe590f03c,0xe8bd9fff,512,0xe52db004,
0xe28db000,0xe3a00021,0xef000000,0xeafffffc,512,0xe52db004,
0xe28db000,0xe3a00041,0xef000000,0xeafffffc,512};

```

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

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

Um die unterschiedlichen Threads voneinander abgrenzen zu können, existiert noch ein Descriptoren-Array, was die Längen eines jeden Threads definiert. In dem Code-Beispiel bedeutet das, dass der erste Thread in den ersten 7 Stellen der RAM-Disk lokalisiert ist, dann kommt 1 Byte für den Stack und daraufhin beginnt der zweite Thread. Mit diesem Schema ist es möglich so viele Threads wie gewollt in **MopS** zu laden und diese separat zu identifizieren. Um genauere Informationen zu erhalten, wie der **ramdiskMaker** funktioniert, kann im Anhang A.1 nachgelesen werden.

6.8.2. MopS Loader

Neben des Mechanismuses, 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** in das 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 6.21: **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
38 }

```

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 herreingeschrieben 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).

6.8.3. Threadlayout

Nachdem der Thread erfolgreich in den RAM geladen wurde, war es an der Zeit das Threadlayout des Threads zu definieren. Als Vorlage dafür diente das Threadlayout von Dr. Prof. Burkhard Messer der HTW-Berlin. Er beschrieb in seinen Folien für die Vorle-

sung *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 6.22: Threadlayout

```

1  #ifndef THREAD_H
2  #define THREAD_H
3
4  #define MAX_THREADS    3
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 Threadlayout erzeugt. Folgende Methode erfüllt genau diese Aufgabe:

Code-Beispiel 6.23: Threadlayout 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;

```

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

```

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 Thread die passenden Start-, End-, Stackpointer- und Programmcounterwerte zuzuweisen, fügt die Methode zudem noch den Thread in eine globale Tabelle ein. In den Zeilen 26-29 kann man erkennen, wie die Adressen zugewiesen werden.

6.8.4. Thread Generierung

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

Code-Beispiel 6.24: Thread Generierung

```

1
2     .text
3     .code 32
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    ldmdfd sp!,{r0-r12}
25    str sp, [r0, #0x10]
26    ldmdfd sp!, {pc}
27    .endfunc
28    .end

```

Diese Methode erhält die Adresse des zu startenden Thread und führt dann eine Reihe wichtiger Sachen durch:

1. Stackpointer des aktuellen Modus retten (Z. 9)
2. Stackpointer des neuen Threads laden (Z. 11)

3. Alle Register auf den Stack des Prozesses retten (Z. 12-13)
4. Die Startadresse des Programms laden (Z. 16)
5. In den Prozess springen (Z. 18)
6. Alle Register wiederherstellen und an den Aufrufer zurückkehren (Z. 24-26)

Dieses Schema wird für jeden neuen Thread 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 Entwurfs 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 Speichermanagment, 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 Speichermanagment 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 besonderen Dank möchte ich an dieser Stelle Herr Prof. Dr. Burkhard Messer aussprechen. Er stand mir immer mit kompetenten Rat und Tat zu Seite. Ohne Ihn hätte das Projekt nicht diese besonderen Ausmaße angenommen.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin den 9. August 2013

Christopher Kruczek

Literatur

- [ARM04a] ARM-Limited. *ARM Dual-Timer Module (SP804)*. Januar 2004. ARM-Limited. 2004. URL: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0271d/index.html> (Letzter Zugriff 01.08.2013).
- [ARM04b] ARM-Limited. *PrimeCell Vectored Interrupt Controller (PL190)*. November 2004. ARM-Limited. 2004. URL: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0181e/index.html> (Letzter Zugriff 07.07.2013).
- [ARM05] ARM-Limited. *ARM Architecture Reference Manual I*. July 2005. ARM-Limited. 2005. URL: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html> (Letzter Zugriff 07.07.2013).
- [Bri88] Dennis M. Ritchie Brian W. Kerninghan. *THE C PROGRAMMING LANGUAGE*. 2. Aufl. 1988.
- [Com] Technical Commitee. *Rationale for American National Standard for Information Systems - Programming Language - C*. URL: <http://www.lysator.liu.se/c/rat/title.html> (Letzter Zugriff 31.07.2013).
- [GNU] GNU. *Using LD, the GNU linker*. URL: http://www.math.utah.edu/docs/info/ld_3.html (Letzter Zugriff 31.07.2013).
- [GNU10] GNU. *GNU 'make'*. 2010. URL: http://www.gnu.org/software/make/manual/html_node/index.html (Letzter Zugriff 31.07.2013).
- [Sil09] Gagne Silberschatz Galvin. *Operating Systems Concept*. 8. Aufl. John Wiley & Sons, Inc., 2009.
- [Tan03] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. 2. Aufl. Pearson Studium, 2003.
- [Tan05] Andrew S. Tanenbaum. *Structured computer organization*. 5. Aufl. 2005.

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 vorhandenen .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 *gemu* zu erstellen. Dazu ist es noch notwendig alle Informationen zusammen zu linkern. 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 Objektdaten 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 Objektdaten über folgenden Aufruf generiert werden:

Code-Beispiel B.5: Objdump einer Objektdaten

```
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 CFLAGS= -O0 -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
17 ARMPATH=/usr/arm-none-linux-gnueabi/libc
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 linkern. Neben diesen Kommandos gibt es noch Regeln um das Projekt komplett zu bauen und um alle unnötigen Objektdaten zu entfernen.

Sämtliche Informationen zur Benutzung von *make* wurden entweder aus alten Lehrveranstaltungen der HTW-Berlin oder der GNU Dokumentation [GNU10] entnommen.