

Embedded Betriebssystem

für ARM Cortex-A8

eine Arbeit von

Nicolaj Höss, Marko Petrović, Kevin Wallis

Master Informatik (ITM2)

für die Lehrveranstaltung

**S1: Softwarelösungen für ressourcenbeschränkte
Systeme**

Fachhochschule Vorarlberg

31. Juli 2015, Dornbirn

Kurzfassung

Diese Arbeit befasst sich mit der Entwicklung eines Embedded-Betriebssystems basierend auf der Hardware des Einplatinencomputers *BeagleBone* von *Texas Instruments*. Zielstellung der Arbeit ist es, ein voll funktionsfähiges Betriebssystem zu erstellen, mit welchem über eine Applikation Scheinwerfer über das DMX512-Kommunikationsprotokoll gesteuert werden können.

Zu Beginn werden die zu erfüllenden Anforderungen an das Betriebssystem vorgestellt. Es werden neben von den Betreuern vordefinierten Zielen auch eigene Ziele des Projektteams genannt. Danach wird der Leser mit der Hardware des Systems und mit den spezifischen Eigenschaften einiger Komponenten bekannt gemacht.

Nach dem Projektmanagement wird im dritten Kapitel die Architektur des Betriebssystems als Schichtenmodell erläutert. Dies beinhaltet eine kurze Beschreibung des Kernels, sowie der angedachten Abstraktion des Kernels. Dem Architekturentwurf folgt die Erstellung eines *Hardware Abstraction Layer* (HAL), welcher die Protabilität des Betriebssystems ermöglicht.

Aufbauend auf die HAL wird das Treiberkonzept des Betriebssystems und die Verwendung des Device Managers präsentiert. Unter Prozessverwaltung werden die Prozesszustände, deren Transitionen und die Eigenschaften des Schedulers aufgezeigt.

Die Implementierung der virtuellen Speicherverwaltung stellt einen der Kernpunkte der Arbeit an diesem Betriebssystem dar. Zuerst wird die allgemeine Funktionsweise eines Tabellensystems und die Umwandlung von virtuellen in physikalische Adressen des *ARM Cortex-A8* erklärt. Es folgen die Spezifikation des Speichermodells des Betriebssystems sowie die Vorstellung der Funktionsschnittstelle der *Memory Management Unit* (MMU).

Die Interprozesskommunikation sowie die Systemschnittstelle und die Funktionsweise der *System Calls* werden kurz umrissen. Sicherheitskritische Aspekte bezüglich des Nulladressenproblems und der sauberen Trennung von Prozess- und Kerneladressbereichen sowie der Benutzermodi werden ebenfalls behandelt. Im Anschluss wird die Benutzerapplikation diskutiert, welche die Steuerung von Scheinwerfern über das DMX512-Protokoll vornimmt.

Abschließend werden die Resultate der Performanzauswertung des Betriebssystems sowie eine Zusammenfassung der Ergebnisse dieses Projektes vorgestellt.

Inhaltsverzeichnis

1	Allgemein	7
1.1	Vorgegebene Anforderungen an das Betriebssystem	7
1.2	Eigene Anforderungen an das Betriebssystem	8
1.3	Erfüllung der Anforderungen	8
2	Projektmanagement	9
2.1	Prozessmodell	9
2.2	Versionsverwaltung	9
2.3	Repository	9
2.4	Zeitplan	9
3	Architektur	10
3.1	Art des Kernels	10
3.2	Ansatz für die Abstraktionen im Betriebssystem	10
3.3	Allgemeiner Aufbau der Architektur	10
4	Hardware Abstraction Layer (HAL)	14
4.1	Aufbau der HAL Schnittstelle	14
4.2	Interrupts	14
5	Treiber	18
5.1	Allgemeiner Aufbau eines Treibers	18
5.2	Beispiel-Implementierung eines Treibers	18
5.3	DriverManager	19
5.4	DeviceManager	20
6	Prozessverwaltung	21
6.1	Prozesszustände	21
6.2	Scheduler Eigenschaften	23
6.3	Vorgehen bei der Prozessverwaltung	23
7	Virtuelle Speicherverwaltung	25
7.1	Grundlegende Funktionsweise	25
7.2	Umwandlung virtueller Adressen zu physikalische Adressen	26
7.3	Seitentabellen und Seitentabelleneinträge	28
7.4	Aufteilung des virtuellen Speichers und Mapping	31
7.4.1	Speicherregionen	33
7.4.2	Master Page Table	34
7.5	Allokierung der Page Frames	35
7.5.1	Allokation von Page Frames bei Data Abort Exception	35
7.6	Aktivieren der MMU	37
7.7	Interaktion der MMU mit Prozessen	37

8 Interprozesskommunikation	39
8.1 Aufbau	39
8.2 IpcManager	39
9 System API	40
9.1 Aufbau eines Systemcall Datenpakets	40
9.2 Vorgehensweise bei einem Systemcall	40
10 Sicherheitsaspekte	42
10.1 Sicherheitsrisiken	42
10.2 Vermeidung des Nulladressenproblems	42
10.3 Implementierung der <i>High Vectors</i>	42
10.4 Umgesetzte Sicherheitsaspekte	43
11 BenutzerInnen-Anwendung	44
11.1 Grundlegender Aufbau des DMX Protokolls	44
11.2 Messergebnisse des Implementierten DMX Protokolls	46
12 Performanzuntersuchungen	49
12.1 Messung und Ergebnisse	49
13 Zusammenfassung und Ausblick	50
13.1 Zusammenfassung	50
13.2 Ausblick	50
13.2.1 Punkte mit Verbesserungspotential	50
13.2.2 Fehlende Punkte für eine praktische Verwendung des Betriebssystems	51
Literaturverzeichnis	52

Abbildungsverzeichnis

1	Allgemeiner Aufbau der Architektur	11
2	IRQ/FIQ Verarbeitung [2, S. 193]	16
3	Interruptverarbeitung im Betriebssystem	17
4	Erlaubte Prozesszustände und Prozessübergänge	21
5	Prozesszustände und deren Transitionen	22
6	Sequenzdiagramm der Prozessverwaltung	24
7	Zweistufiges Seitentabellensystem [1, S. B3-1325]	26
8	1 MB Section Translation durch die ARM CPU [1, S. B3-1335]	27
9	Small Page Translation durch die ARM CPU [1, S. B3-1337]	28
10	TTBR0 Format [1, S. B4-1726]	29
11	TTBR1 Format [1, S. B4-1730]	29
12	First-Level Deskriptorformate [1, S. B3-1326]	30
13	Second-Level Deskriptorformate [1, S. B3-1327]	31
14	Memory Map des Betriebssystems	32
15	Beispiel einer Bitmap zur Verwaltung der Page Frames	35
16	Einlagerung von page frames durch den DABT-Handler	36
17	Sequenzdiagramm eines Systemcalls	41
18	DMX Protokoll	44
19	DMX Protokoll: Problem fallende Flanke	46
20	DMX Protokoll: Problem Offset Byte	47
21	Funktionierendes DMX Protokoll	48

Abkürzungsverzeichnis

AINTC	ARM Interrupt Controller
CWD	Current Working Directory
DALI	Digital Addressable Lighting Interface (Bus Protokoll)
DFAR	Data Fault Address Register
DFSR	Data Fault Status Register
DMX	Digital Multiplex (Bus Protokoll)
GPIO	General Purpose Input Output
UART	Universal Asynchronous Receiver Transmitter
HAL	Hardware Abstraction Layer
KNX	Konnex-Bus (Bus Protokoll)
MMU	Memory Management Unit
MPT	Master Page Table
OS	Operating System
PTE	Page Table Entry
SCTLR	System Control Register
TLB	Translation Lookaside Buffer
TTBCR	Translation Table Base Control Register
TTBR	Translation Table Base Register
VMSAv7	Virtual Memory System Architecture for ARMv7

1 Allgemein

In diesem Kapitel werden allgemeine Aspekte zum Betriebssystem erläutert. Dazu zählen insbesondere die durch das Studienprojekt definierten Anforderungen, zusätzlich durch die Studierenden gesetzte Anforderungen und die erreichten Ergebnisse hinsichtlich dieser Anforderungen.

1.1 Vorgegebene Anforderungen an das Betriebssystem

Eine Auflistung aller vorgegebenen Anforderungen, insbesondere funktionale Anforderungen, an das Betriebssystem sind in Tabelle 1 angegeben.

Anforderung	Erklärung
Single-User	Das Betriebssystem muss zu jedem Zeitpunkt nur eine Benutzerin bzw. einen Benutzer verwalten.
Lauffähige Anwendung	Auf dem Betriebssystem muss zumindest eine lauffähige Anwendung ausführbar sein.
Präemptives Multitasking	Das Betriebssystem muss gleichzeitig mehrere Prozesse ausführen können, wobei für jeden Prozess eine bestimmte Zeitscheibe vorgesehen ist.
Konsole	Es muss eine Konsole zum Absetzen von Befehlen vorhanden sein.
Interprozess-Kommunikation	Das Betriebssystem muss eine Möglichkeit zur Kommunikation zwischen Prozessen zur Verfügung stellen.
Sicherheit	Es muss eine strikte Trennung zwischen User- und Systemmodus vorhanden sein.
Robustheit	Das Betriebssystem, respektive dessen Stabilität, darf von Programmabstürzen nicht beeinflusst werden.
Virtueller Speicher	<i>Memory Management</i> muss für größere Anwendungen vorhanden sein.
SD-Karte	Externe Anwendungen sollen von der SD-Karte nachgeladen werden können.
Dateisystem	Das Betriebssystem muss ein beliebiges Dateisystem verwalten können.
Portierbarkeit	Für eine Portierbarkeit des Systems muss ein Hardware Abstraction Layer (HAL) umgesetzt werden.
Integration von Geräten	Das Betriebssystem muss eine einfache Integration von verschiedenen Geräten gewährleisten.
Performanztests	Es müssen Performanztests zur Leistungsfeststellung des Systems durchgeführt werden.

Tabelle 1: Vorgegebene Anforderungen

1.2 Eigene Anforderungen an das Betriebssystem

Zusätzlich zu den oben angeführten Anforderungen wurden weitere, nicht-funktionale Anforderungen an das Betriebssystem, durch die an der Entwicklung beteiligten Studierenden definiert. Eine Auflistung aller eigenen Anforderungen an das Betriebssystem sind in Tabelle 2 angegeben.

Anforderung	Erklärung
Hoher Abstraktionsgrad	Alle Komponenten des Betriebssystems sollen einen hohen Abstraktionsgrad aufweisen.
Intuitiver Aufbau	Die Komponenten des Betriebssystem sollen eine intuitive Programmierschnittstelle aufweisen.
Leichte Erweiterbarkeit	Mögliche Erweiterungen sollen ohne große Veränderungen an der Architektur umgesetzt werden können.
Einfache Wartung	Das Betriebssystem soll eine einfache Wartbarkeit hinsichtlich Fehlern aufweisen.
<i>Moving Head</i> mit Digital Multiplex (Bus Protokoll) (DMX)	Auf dem Betriebssystem soll eine Anwendung zur Ansteuerung eines oder mehrerer <i>Moving Heads</i> mittels DMX-Protokoll ausführbar sein.

Tabelle 2: Vorgegebene Anforderungen

1.3 Erfüllung der Anforderungen

Im Allgemeinen wurden alle zuvor erwähnten funktionalen Anforderungen an das Betriebssystem erfüllt. Einzelne Verbesserungs- bzw. Erweiterungsmöglichkeiten können aus Kapitel 13 werden.

Die Performanz des Betriebssystems wird in Kapitel 12 dokumentiert. Hinsichtlich der Stabilität wurden keine konkreten Experimente durchgeführt, allerdings haben verschiedene Benutzungstests gezeigt, dass das Betriebssystem über mehrere Stunden ohne Abstürze lauffähig ist.

2 Projektmanagement

Im folgenden Abschnitt wird das Projektmanagement und das verwendete Prozessmodell beschrieben. Weiters sind hier die Zugänge zum *Repository* und dem Ticketsystem dokumentiert.

2.1 Prozessmodell

Als Prozessmodell wurde *SCRUM*, allerdings mit einigen Adaptionen umgesetzt, wobei das zentrale Vorgehen in Bezug auf Agilität bestmöglich übernommen wurde. Gründe für die Verwendung von *SCRUM* waren vor allem die Möglichkeit zur agilen Umsetzung der vorhandenen und neuer Anforderungen.

Es wurde auf das Konzept eines *SCRUM*-Boards verzichtet, stattdessen wurden sämtliche *Stories* als eigenes Ticket in einem passenden Ticketsystem angelegt. Es erfolgte eine Priorisierung der jeweiligen Tickets. Die Abarbeitungsreihenfolge dieser Tickets ergab sich schließlich nach der jeweiligen Priorität selbst.

Die angelegten Tickets finden sich unter folgendem Link:

<https://github.com/Blackjack92/fhvOS/issues>

Durch Einsicht der offenen und geschlossenen Tickets lässt sich sowohl der Entwicklungsfortschritt, als auch die jeweiligen Designentscheidungen sehr gut nachvollziehen.

2.2 Versionsverwaltung

Als Versionsverwaltung wurde *Git* verwendet. Die Entscheidung für die Verwendung von *Git* sind insbesondere die leichte Einbindung im Zusammenhang mit dem angelegten *Repository* und die Möglichkeit der Nicht-linearen Entwicklung.

2.3 Repository

Das *Repository* für den Source-Code und weitere relevante Dokumente für die Entwicklung des Betriebssystems, wurde auf *Github* angelegt und veröffentlicht. Der Source-Code des Betriebssystems war während der gesamten Entwicklungszeit öffentlich zugänglich. Unter folgendem Link ist das *Repository* einsehbar

<https://github.com/Blackjack92/fhvOS>

2.4 Zeitplan

Der Zeitplan des Projekts wurde mittels *Microsoft Project* erstellt und verwaltet. Der Zeitplan selbst wurde während des gesamten Projekts im Allgemeinen sehr gut eingehalten. Das angelegte *Microsoft Project*-Projekt ist ebenfalls im oben beschriebenen *Repository* im Ordner *projectmanagement* einsehbar.

3 Architektur

Dieses Kapitel beschreibt die entwickelte Architektur des Betriebssystems und gibt einen groben Überblick über die einzelnen Komponenten. Detailliertere Informationen zu den einzelnen Komponenten werden in den folgenden Kapiteln beschrieben.

3.1 Art des Kernels

Das Betriebssystem wurde als Monolithischer Kernel umgesetzt, respektive wurden die Speicherverwaltung, Prozessverwaltung, Treiber und andere Kernelkomponenten in einem einzelnen Kernelprozess implementiert. Durch die Zusammenführung dieser Komponenten verliert das Betriebssystem zwar die Eigenschaft, nach einem Absturz einer einzelnen Komponente weiter lauffähig zu sein, allerdings darf mit diesem Konzept durchaus von einer höheren Performanz ausgegangen werden. Ein weiterer Grund für einen Monolithischen Kernel ist das entfallen der aufwändigen Kommunikationen zwischen den verschiedenen Komponenten des Betriebssystems, welche besonders in der Anfangsphase der Entwicklungsarbeiten zu Problemen führen kann.

3.2 Ansatz für die Abstraktionen im Betriebssystem

Um eine möglichst gute Abstraktionen der Betriebssystem-Implementierung zu gewährleisten, wurden für die Verwaltung einzelner Kernelkomponenten *Manager* verwendet. Im Allgemeinen gilt, dass die einzelnen *Manager* jeweils die Schnittstelle für eine Komponente darstellen. Die Kommunikation zwischen zwei Komponenten erfolgt ausschließlich über die bereitgestellte Schnittstelle des *Managers*. Eine Übersicht der einzelnen *Manager* sowie eine bzw. mehrere zugehörige Funktionen, ist in Tabelle 3 angegeben.

Managername	Beispiel Funktion(en)
<i>DeviceManager</i>	InitDevice, OpenDevice, ReadDevice
<i>DriverManager</i>	GetDriver
<i>FileManager</i>	OpenFile, OpenExecutable
<i>MemManager</i>	GetFreePagesInProcessRegion, GetRegion
<i>ProcessManager</i>	StartProcess, KillProcess
<i>IpcManager</i>	RegisterNamespace, SendMessage

Tabelle 3: Übersicht der Manager mit beispielhaften Funktionen

3.3 Allgemeiner Aufbau der Architektur

In Abbildung 1 ist der allgemeine Aufbau mit den wesentlichen Kernelkomponenten ersichtlich. Zusätzlich zu den einzelnen Komponenten ist ebenfalls der Informationsfluss dargestellt.

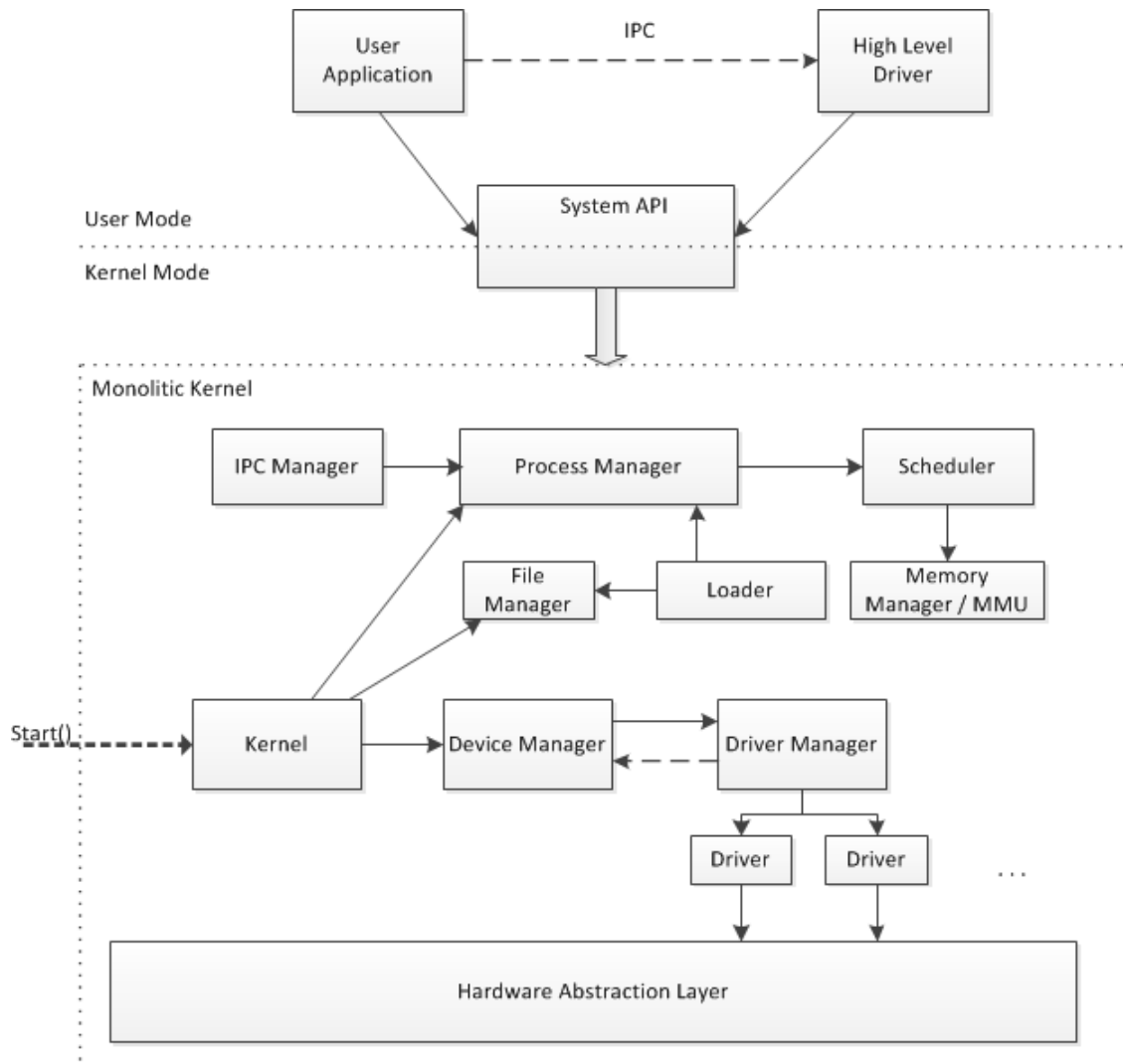


Abbildung 1: Allgemeiner Aufbau der Architektur

Die oben angeführten Komponenten und deren Verantwortlichkeiten sind im Folgenden grob beschrieben:

Hardware Abstraction Layer (HAL)

Der HAL abstrahiert sämtlichen Hardwarezugriff des Betriebssystems und erlaubt so eine einfache Portierbarkeit auf andere Plattformen. Der Aufbau des HAL wird in Kapitel 4 beschrieben.

Driver

Ein Treiber bietet eine abstrakte Schnittstelle zur jeweiligen Hardware, respektive im-

plementiert dieser die konkrete Ansteuerung. Der Aufbau der Treiber ist in Kapitel 5 dokumentiert.

DriverManager

Der *DriverManager* dient zur Verwaltung der Treiber, welche vom Betriebssystem zur Verfügung gestellt werden. Sollte ein Treiber benötigt werden, kann über die Schnittstelle des *DriverManagers* der Zugriff auf den Treiber erfolgen. Der detaillierte Aufbau des *DriverManager* ist in Kapitel 5.3 erläutert.

DeviceManager

Der *DeviceManager* ist eine weitere Abstraktion zur Verwaltung von Treibern, respektive einzelnen Geräten. So sind beispielsweise auf Treiberebene alle vier Board-LEDs als identisch anzusehen, allerdings stellen sie auf Geräteebeke jeweils vier unterschiedliche Geräte dar, welche aber vom selben Treiber angesprochen werden können. In Kapitel 5.4 wird die Verantwortlichkeit des *DeviceManagers* genauer diskutiert.

Kernel

Der Kernel weist die Verantwortlichkeit für das Starten der einzelnen Komponenten auf und stellt grundlegende Funktionsschnittstellen für die einzelnen Komponenten zur Verfügung. Beispielsweise werden sämtliche Fehler oder Ausgaben von Kernelkomponenten an den Kernel selbst propagiert.

ProcessManager

Der *ProcessManager* stellt eine Schnittstelle für den Zugriff auf Prozesse zur Verfügung. Weiters verwaltet der *ProcessManager* Meta-Daten zu den einzelnen Prozessen, wie Prozessname, Startzeit usw. Implementierungsdetails zum *ProcessManager* werden in Kapitel 6 beschrieben.

Scheduler

Der *Scheduler* verwaltet die Prozesse auf einer abstrakten Ebene, respektive hinsichtlich ihrer Zustände, ihres Kontexts usw. Weiters ist der *Scheduler* für die Umsetzung des präemptiven Multi-Taskings verantwortlich. Die konkrete Implementierung ist in Kapitel 6.2 beschrieben.

***MemoryManager* / Memory Management Unit (MMU)**

Der *MemoryManager* stellt die Schnittstelle zum virtuellen Speichermanagement zur Verfügung. Diese Komponente übernimmt ebenfalls die Verantwortlichkeit für die Verwaltung des virtuellen und physischen Speichers. Der *MemoryManager* bzw. das Speichermanagement im Allgemeinen ist im Kapitel 7 beschrieben.

FileManager

Der *FileManager* stellt die Schnittstelle zum Zugriff auf das Dateisystem, respektive Dateien und Ordner zur Verfügung. Der *FileManager* verwaltet ebenfalls die Current Working Directory (CWD) des Benutzers bzw. der Benutzerin.

Loader

Der *Loader* ist für das Laden von Anwendungen verantwortlich, respektive lädt der *Loader* eine Anwendung von einem externen Speichermedium in den Arbeitsspeicher, so dass dieser als Prozess ausgeführt werden kann.

IPCManager

Der *IPCManager* ist für die Kommunikation zwischen verschiedenen User-Anwendungen zuständig. Die Dokumentation der Interprozesskommunikation ist in Kapitel 8 festgehalten.

System-API

Die System-API stellt eine Schnittstelle zum Betriebssystem für Anwendungen zur Verfügung. Hierdurch sind die Betriebssystemfunktionen von der Anwendung selbst entkoppelt. Dies erlaubt einen sicheren Zugriff auf Geräte und Ressourcen. Die System-API ist in Kapitel 9 beschrieben.

User Application

User Applications (dt. Anwendung) sind Prozesse welche von einem externen Speichermedium geladen werden. Diese können mittels Interprozesskommunikation mit anderen Anwendungen oder mittels der System-API mit anderen Komponenten kommunizieren. Im Kapitel 11 ist eine konkrete Implementierung einer Anwendung dokumentiert.

High Level Driver

High Level Driver sind Treiber, welche einer Anwendung eine breitere Schnittstelle als Kernel-interne Treiber anbieten und unter anderem auch erweiterte Funktionen implementieren. Beispielsweise wird durch den *High Level Driver* für das Ansteuern eines *Moving Heads* über das DMX-Protokoll das zyklische Senden des aktuellen Zustands realisiert.

4 Hardware Abstraction Layer (HAL)

Der HAL dient zur Abstraktion der eigentlichen Hardware und erlaubt somit eine einfache Portierbarkeit des Betriebssystems auf andere Hardwarearchitekturen. Durch den implementierten HAL wurde ebenfalls der direkte Zugriff auf Hardwareadressen und Register entkoppelt.

4.1 Aufbau der HAL Schnittstelle

Die HAL-Schnittstelle wurde für alle Hardwarekomponenten unterschiedlich implementiert und bietet für die einzelnen Funktionen und Eigenschaften einer Hardwarekomponente jeweils eigene Funktionen. In Listing 1 und Listing 2 sind beispielhaft die HAL-Schnittstellen für das Ansprechen der General Purpose Input Output (GPIO) und des Universal Asynchronous Receiver Transmitter (UART) angeführt.

```
1 extern void GPIOEnable(uint16_t pin);
2 extern void GPIODisable(uint16_t pin);
3 extern void GPIOReset(uint16_t pin);
4 extern void GPIOSetMux(uint16_t pin, mux_mode_t mux);
5 extern void GPIOSetPinDirection(uint16_t pin, pin_direction_t dir);
6 extern void GPIOSetPinValue(uint16_t pin, pin_value_t value);
7 extern pin_value_t GPIOGetPinValue(uint16_t pin);
```

Listing 1: HAL-Schnittstelle für die GPIOs

```
1 extern int UARThalEnable(uartPins_t uartPins);
2 extern int UARThalDisable(uartPins_t uartPins);
3 extern int UARThalSoftwareReset(uartPins_t uartPins);
4 extern int UARThalFifoSettings(uartPins_t uartPins);
5 extern int UARThalSettings(uartPins_t uartPins, configuration_t* config);
6 extern int UARThalFifoWrite(uartPins_t uartPins, uint8_t* msg);
7 extern int UARThalFifoRead(uartPins_t uartPins, uint8_t* msg);
8 extern boolean_t UARThalIsFifoFull(uartPins_t uartPins);
9 extern boolean_t UARThalIsCharAvailable(uartPins_t uartPins);
```

Listing 2: HAL-Schnittstelle für die UART

4.2 Interrupts

Die Interrupts stellen einen grundlegenden Teil des HAL dar. Um die Komplexität der Software nicht zu erhöhen, wurden keinerlei sogenannte *nested interrupts*, dies sind höherpriorige Interrupts die bei ihrem Auftreten niederpriorige Interrupts unterbrechen können, verwendet.

Einstellungen betreffend der Interrupts werden im ARM Interrupt Controller (AINTC) vorgenommen. Dieser ist zuständig für die Priorisierung der Interrupts und Verarbeitung von Interruptrequests durch die Systemperipherie. Der AINTC kann bis zu 128 Interrupts verarbeiten, eine Liste aller vom AM335x unterstützten Interrupts findet sich unter [2, S. 199].

Grundsätzlich sind die Prioritäten der Interrupts und deren Art, ob *IRQ* (normaler Interrupt) oder *FIQ* (fast interrupt), einstellbar.

Die im HAL implementierten Interruptfunktionalitäten dienen als Grundlage für das Handling spezifischer IRQs und FIQs, beispielsweise von Timern oder UARTs. Das nachfolgende Listing 3 zeigt die zur Verfügung gestellte Funktionalität. Die wichtigsten Funktionen betreffen das Resetten des AINTC, das globale Ein- bzw. Ausschalten von Interrupts sowie das einzelne Ein- bzw. Ausschalten der Peripherieinterrupts.

```

1 extern void InterruptResetAINTC(void);
2 extern void InterruptPrioritySet(unsigned int intrNum, unsigned int priority);
3 extern void InterruptHandlerEnable(unsigned int intrNum);
4 extern void InterruptHandlerDisable(unsigned int intrNum);
5 extern void InterruptAllowNewIrqGeneration();
6 extern void InterruptHandlerRegister(unsigned int interruptNumber, intHandler_t
    ↪ fnHandler);
7 extern void InterruptUnRegister(unsigned int interruptNumber);
8 extern void InterruptSetGlobalMaskRegister(unsigned int interruptMaskRegister, unsigned
    ↪ int mask);
9 extern void InterruptClearGlobalMaskRegister(unsigned int interruptMaskRegister,
    ↪ unsigned int mask);
10 extern unsigned int InterruptActiveIrqNumberGet(void);
11 extern intHandler_t InterruptGetHandler(unsigned int interruptNumber);
12 extern void InterruptSaveUserContext(void);
13 extern void InterruptRestoreUserContext(void);
14 extern void InterruptMasterIRQEnable(void);
15 extern void InterruptMasterIRQDisable(void);
16 extern void InterruptMasterFIQEnable(void);
17 extern void InterruptMasterFIQDisable(void);

```

Listing 3: Schnittstelle für die Interrupts

Die Verarbeitungsprozedur sowohl von IRQs als auch von FIQs zeigt Abbildung 2. Im Betriebssystem codiert sind die Behandlungsschritte 5 (Ablegen des aktuellen Kontextes auf den Stack), 6 (Aufruf des zugewiesenen Interrupthandlers), 7 (Erlauben neuer Interruptauftritte) und 8 (Wiederherstellung des ursprünglichen Kontextes vom Stack).

Eine häufige Fehlerquelle stellt das versehentliche Weglassen von Schritt 7 dar. Das *NEWIRQAGR* (new IRQ agreement) ist dafür zuständig dem System anzuzeigen, dass ein unbehandelter Interruptrequest vorliegt. Wird Schritt 7 ausgelassen, wird beim erstmaligen Auftritt eines Interrupts der Interrupthandler wie erwartet aufgerufen. Nach dem Abarbeiten des Interrupthandlers wird die Programmausführung aber sofort wieder in den Interrupthandler springen, da das *NEWIRQAGR* nicht zurückgesetzt wurde und einen nicht behandelten Request anzeigt. Um dieses Problem zu umgehen wurde die Funktion *InterruptAllowNewIrqGeneration* erstellt. Diese Funktion ist immer am Ende der eines behandelten Interrupts aufzurufen.

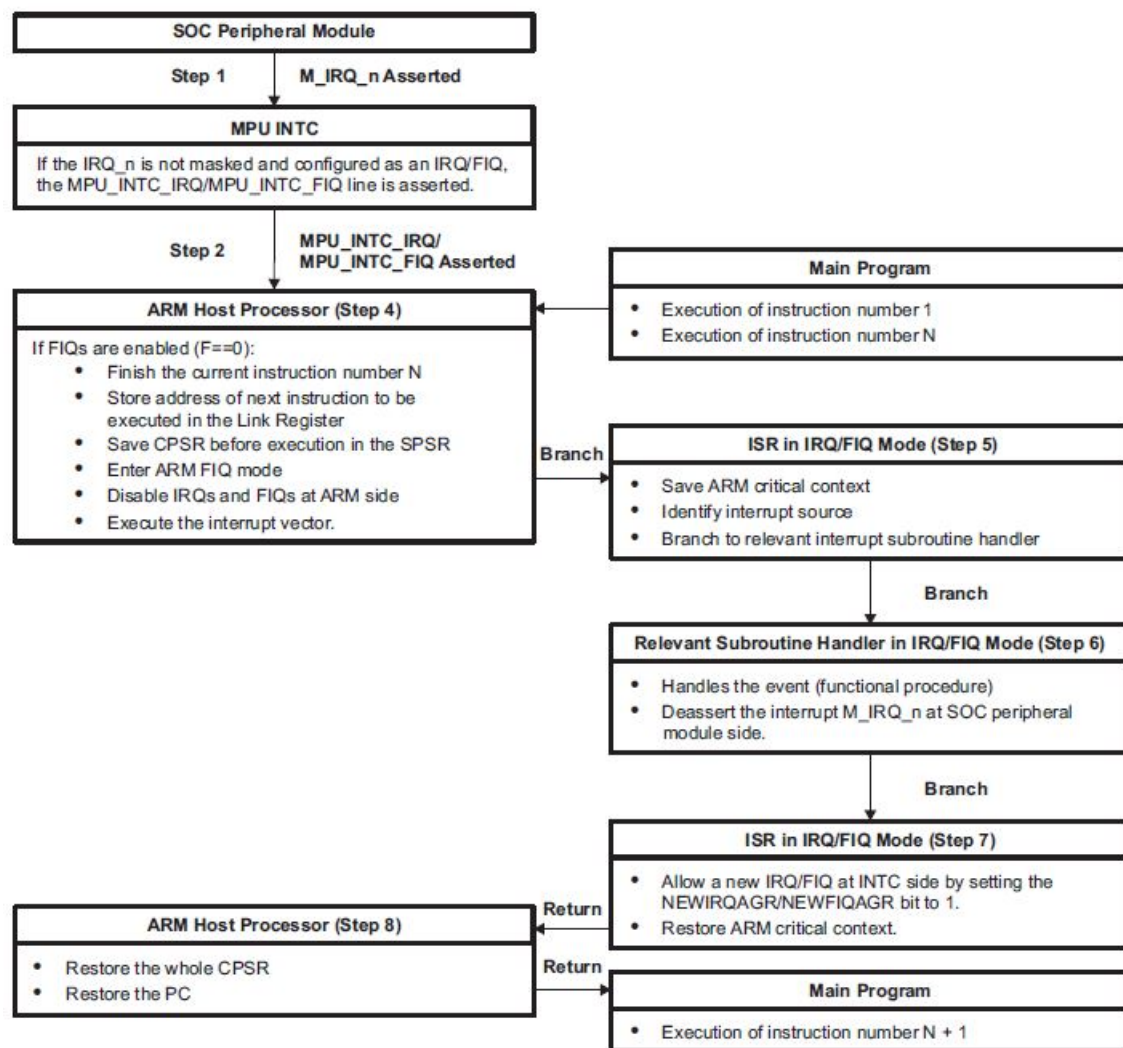


Abbildung 2: IRQ/FIQ Verarbeitung [2, S. 193]

Die Verarbeitung eines Interrupts im Betriebssystem ist in Abbildung 3 schematisch dargestellt. Sichern und Wiederherstellen des aktuellen Kontexts wird durch den in Assembler geschriebenen *IRQ-Handler* vorgenommen. Dieser ruft nach dem Sichern des Kontexts die dem Interrupt zugewiesene Handler-Funktion auf. Nach Behandlung des Interrupts findet eine Wiederherstellung des Kontexts statt, wobei der ursprüngliche Kontext durch die aufgerufene Handler-Funktion durchaus ersetzt oder geändert werden kann.

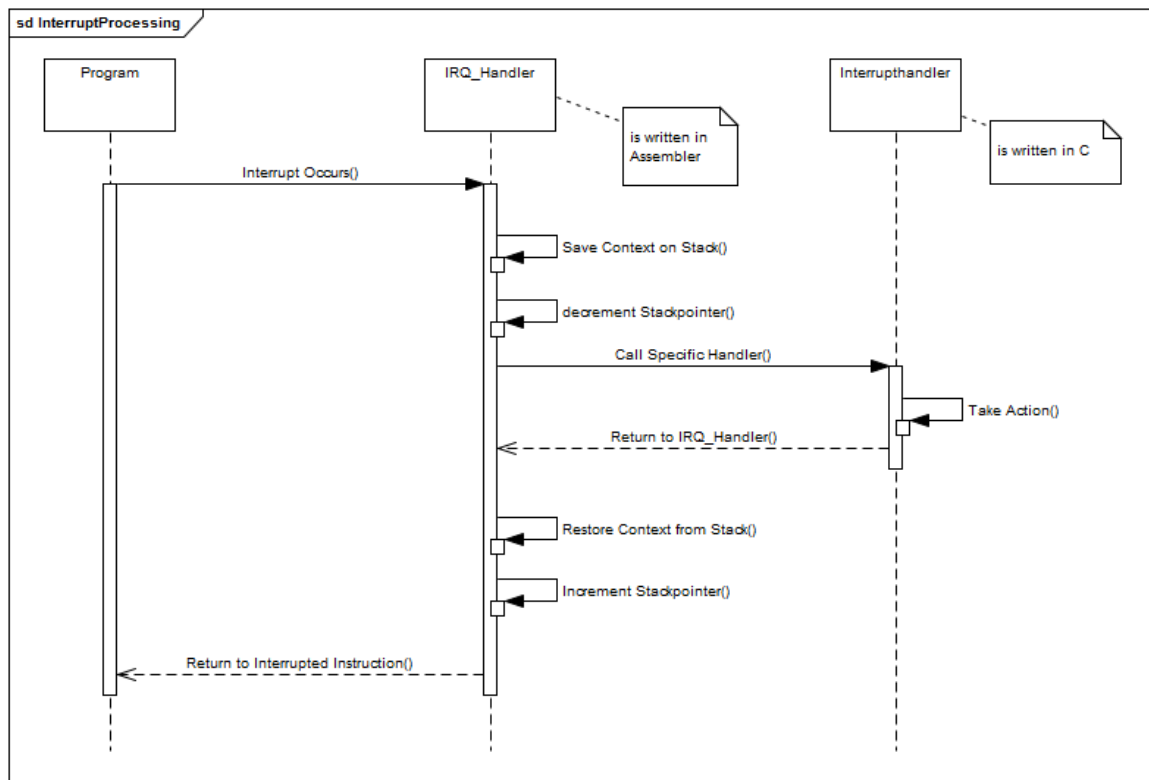


Abbildung 3: Interruptverarbeitung im Betriebssystem

5 Treiber

Die Treiber stellen eine abstrakte Schnittstelle auf den HAL dar, respektive implementieren diese eine Schnittstelle für den Zugriff auf Peripheriegeräte. Ein wesentlicher Aspekt bei der Architektur der Treiber war Abstraktion. Dadurch wird gewährleistet, dass jeder Treiber über die selbe Schnittstelle angesprochen werden kann. Die Verwaltung und der konkrete Zugriff auf Treiber erfolgt schließlich mittels des implementierten *DriverManager*

5.1 Allgemeiner Aufbau eines Treibers

In Listing 4 ist die allgemeine Schnittstelle eines Treibers ersichtlich. Es ist hier anzumerken, dass sämtliche Treiber dieselbe Schnittstelle implementieren.

```
1 typedef struct {
2     int (*init)(uint16_t payload);
3     int (*open)(uint16_t payload);
4     int (*close)(uint16_t payload);
5     int (*read)(uint16_t payload, char* buf, uint16_t length);
6     int (*write)(uint16_t payload, char* buf, uint16_t length);
7     int (*ioctl)(uint16_t payload, uint16_t cmd, uint8_t mode, char* buf, uint16_t
8         ↪ length);
9 } driver_t;
```

Listing 4: Allgemeine Schnittstelle für einen Treiber

In oben angeführtem Listing ist ersichtlich, dass sämtliche Treiberfunktionen einen Parameter `payload` aufweisen. Dieser Parameter erlaubt der Treiberimplementierung die Unterscheidung verschiedener Geräte, welche durch denselben Treiber implementiert werden. Die Propagierung dieses Parameters erfolgt durch den *DeviceManager* und *DriverManager*.

5.2 Beispiel-Implementierung eines Treibers

Listing 5 zeigt beispielhaft einen Auszug der konkreten Treiberimplementierung für die *Board LEDs*, respektive die Funktion `LEDWrite(...)`. Hier ist sehr gut ersichtlich, wie die Treiberimplementierung unterschiedliche Geräte auf Basis des `payload`-Parameters ansteuert.

```
1 [...]
2 int LEDWrite (uint16_t id, char* buf, uint16_t len)
3 {
4     if (id > (BOARD_LED_COUNT - 1)) return DRIVER_ERROR;
5
6     if (len != 1) return DRIVER_ERROR;
7
8     switch (buf[0])
9     {
10        case '1':
11            GPIOSetPinValue(BOARD_LED(id), PIN_VALUE_HIGH);
12            break;
13        case '0':
```

```

14     GPIOSetPinValue(BOARD_LED(id), PIN_VALUE_LOW);
15     break;
16     default:
17         return DRIVER_ERROR;
18     }
19     return DRIVER_OK;
20 }
21 [...]

```

Listing 5: Implementierung der allgemeinen Treiberschnittstelle (LED Beispiel)

5.3 DriverManager

Der *DriverManager* ist verantwortlich für die Initialisierung und Verwaltung der Treiber. In Listing 6 ist die Schnittstelle des *DriverManager* ersichtlich.

```

1 #define DRIVER_ID_LED    123
2
3 extern void DriverManagerInit(void);
4 extern driver_t* DriverManagerGetDriver(driver_id_t driver_id);
5 extern void DriverManagerDestruct(void);

```

Listing 6: Allgemeine Schnittstelle des DriverManagers

Eine Implementierung dieser Schnittstelle für den Treiber der *Board LEDs* ist in Listing 7 aufgezeigt. Für das Hinzufügen eines weiteren Treibers zum *DriverManager* muss so nur die *DriverManagerInit*-Funktion angepasst werden, respektive muss ein zusätzlicher Treiber mit seinen zugehörigen Funktionspointern in das *drivers*-Array eingefügt werden. Es ist zum angeführten Listing anzumerken, dass die Verwendung von *malloc* hier nicht zwingend notwendig wäre und diese auch durch eine statische Allokierung implementiert werden kann.

```

1 static driver_t* drivers[MAX_DRIVER];
2
3 void DriverManagerInit(void)
4 {
5     // LED Driver
6     driver_t* led = malloc(sizeof(driver_t));
7     led->init = &LEDInit;
8     led->open = &LEDOpen;
9     led->close = &LEDClose;
10    led->read = &LEDRead;
11    led->write = &LEDWrite;
12    led->ioctl = &LEDIoctl;
13    drivers[DRIVER_ID_LED] = led;
14 }
15
16 driver_t* DriverManagerGetDriver(driver_id_t driver_id)
17 {
18     return drivers[driver_id];

```

```
19 }
20
21 void DriverManagerDestruct(void)
22 {
23     int i;
24     for (i = 0; i < MAX_DRIVER; i++) {
25         if (drivers[i] != NULL) {
26             free(drivers[i]);
27             drivers[i] = NULL;
28         }
29     }
30 }
```

Listing 7: Implementierung der DriverManager Schnittstelle für den LED Treiber

5.4 DeviceManager

Der *DeviceManager* ist verantwortlich für die Initialisierung und Verwaltung der einzelnen Geräte, welche ebenfalls die Zuordnung des Treibers für das Gerät beinhaltet. In Listing 8 ist die Schnittstelle des *DeviceManagers* ersichtlich.

```
1 typedef union device_t {
2     int device;
3     struct {
4         short driverId;
5         short driverMsg;
6     };
7 } device_t;
8
9 extern void DriverManagerInit();
10 extern device_t DriverManagerGetDevice(char* name, int len);
11 extern int DriverManagerInitDevice(device_t device);
12 extern int DriverManagerOpen(device_t device);
13 extern int DriverManagerClose(device_t device);
14 extern int DriverManagerRead(device_t device, char* buf, int len);
15 extern int DriverManagerWrite(device_t device, char* buf, int len);
16 extern int DriverManagerIoctl(device_t device, int msg, int mode, char* buf, int len);
```

Listing 8: Allgemeine Schnittstelle des DeviceManagers

Wie aus obigem Listing ersichtlich ist, sind die konkreten Treiberzugriffe durch den *DeviceManager* entkoppelt. In den einzelnen Funktionsaufrufen des *DeviceManagers* wird jeweils der Funktionsaufruf an den Treiber, durch das Interpretieren des übergebenen *device_t*, an den richtigen Treiber (*driverId*) mit dem konkreten *payload*-Parameter, ausgeführt.

6 Prozessverwaltung

Als Prozessverwaltung wird hauptsächlich das Verwalten von Prozessen durch das Betriebssystem verstanden (Vgl. <http://www.lowlevel.eu/wiki/Prozessverwaltung>). Jeder Prozess besitzt eine eindeutige Identifikation (PID), durch welche dieser vom System angesprochen werden kann.

6.1 Prozesszustände

Jeder Prozess besitzt zu einem bestimmten Zeitpunkt einen fix definierten Zustand, d.h. es können keine Inkonsistenzen auftreten. Abbildung 4 zeigt die verschiedenen Zustände eines Prozesses sowie die jeweilig erlaubten Übergänge zu einem anderen Zustand auf.

TODO!!!

Abbildung 4: Erlaubte Prozesszustände und Prozessübergänge

Im folgenden wird eine detaillierte Erklärung zu den einzelnen Zuständen aus Abbildung 4 gegeben.

Ready

Der Zustand ready tritt ein, wenn ein Prozess bereit wäre um ausgeführt zu werden.

Running

Ein Prozess weist diesen Zustand auf, wenn er gerade ausgeführt wird. Es gibt immer nur einen running Prozess zu einem bestimmten Zeitpunkt.

Blocked

XXX

Sleeping

XXX

Free

XXX

Es gibt unterschiedliche Zustandsübergänge, welche im Betriebssystem erlaubt sind. Tabelle 4 stellt die verschiedenen Übergänge mit einem dazu passenden Beispiel dar.

Ausgangszustand	Nächster Zustand	Beispiel
Ready	Running	XXX
Running	Blocked	XXX
Running	XXX	XXX
XXX	XXX	XXX

Tabelle 4: Erlaubte Zustandsübergänge mit Beispiel

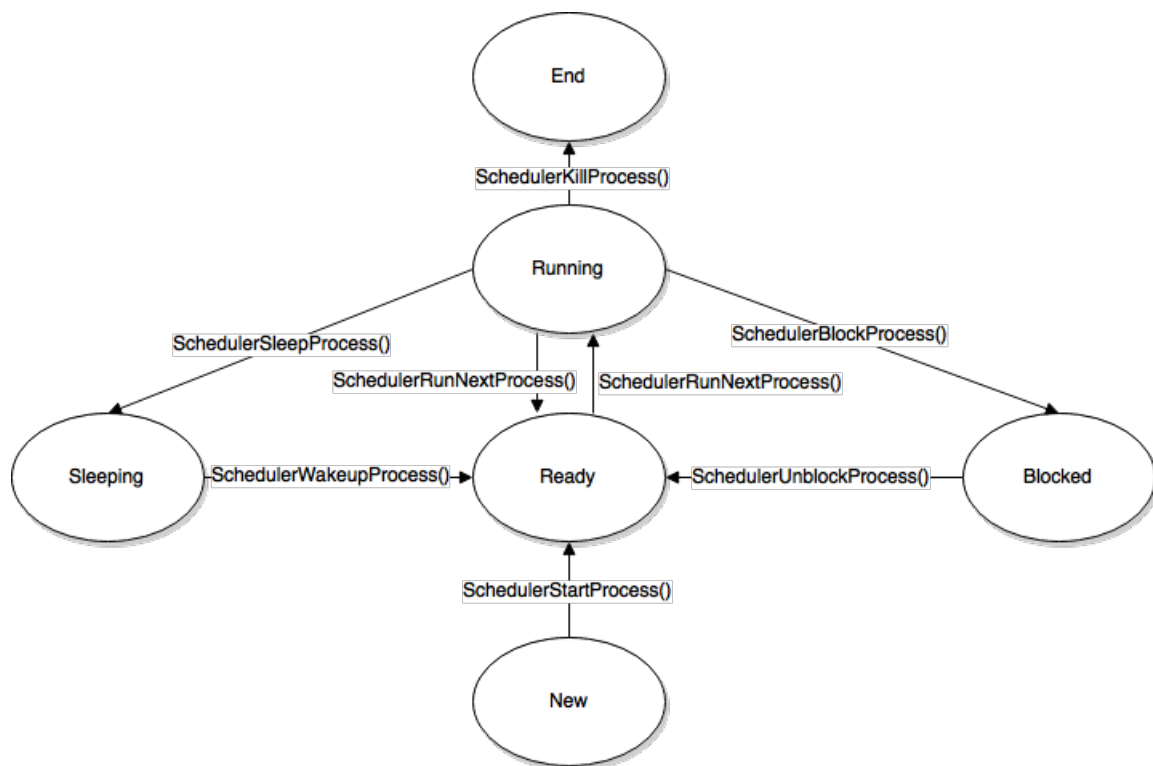


Abbildung 5: Prozesszustände und deren Transitionen

6.2 Scheduler Eigenschaften

Der Scheduler weist eine Zeitscheibe von $10ms$ auf, d.h. jeder Prozess hat $10ms$ bevor er gewechselt wird. Sollte ein anderer Prozess zur Verfügung stehen wird dieser genommen ansonsten bekommt der gleiche Prozesse erneut eine Zeitscheibe von $10ms$. Diese Zeitscheibendauer wurde aufgrund mehrerer Aspekte gewählt: eine zu große Zeitscheibe $> 100ms$ würde merkbaren Verzögerungen im Betriebssystem führen, bei einer zu kleinen Zeitscheibe $< 1ms$ würde die benötigte Zeit für einen Wechsel im Vergleich zu lange dauern. Durchgeführte Performanztests sind im Kapitel XXX aufgeführt.

Das verwendete Schedulingverfahren ist Round Robin. Bei der Verwendung eines Verfahrens mit Prioritäten hätte der Fall mit einem hoch priorisierten Prozesse beachtet werden müssen. Ein Beispiel dazu: Es gibt drei Prioritäten (hoch, mittel und niedrig). Die Konsole wird als hoch eingestuft alle anderen Prozesse sind niedriger priorisiert. Wie bekommt nun ein mittel/niedrig priorisierter Prozess eine Zeitscheibe?

6.3 Vorgehen bei der Prozessverwaltung

Das Vorgehen bei der Prozessverwaltung wird im Sequenzdiagramm von Figure ?? dargestellt. Ein Client übergibt dem ProcessManager die Aufgabe einen Prozess zu erzeugen. Dieser delegiert das Erzeugen des Prozesses an den Scheduler weiter. Hierbei ist zu beachten, dass die Metadaten vom Prozess nicht weitergegeben werden. Der Scheduler speichert sich diesen neuen Prozess in seiner Prozesstabelle ab. Das MemoryManagement dient zum Allokieren von benötigtem Speicherplatz für den neuen Prozess. Der erzeugte Prozess wird an den ProcessManager zurückgegeben, wobei dieser noch MetaInformationen beifügt (z.B. den Namen des Prozesses). Dem Client wird am Ende mitgeteilt, ob das Erzeugen erfolgreich war. Ein alternativer Rückgabeparameter wäre die PID, wobei hier darauf zu achten ist, dass im Fehlerfall eine ungültige PID zurückgeliefert wird.

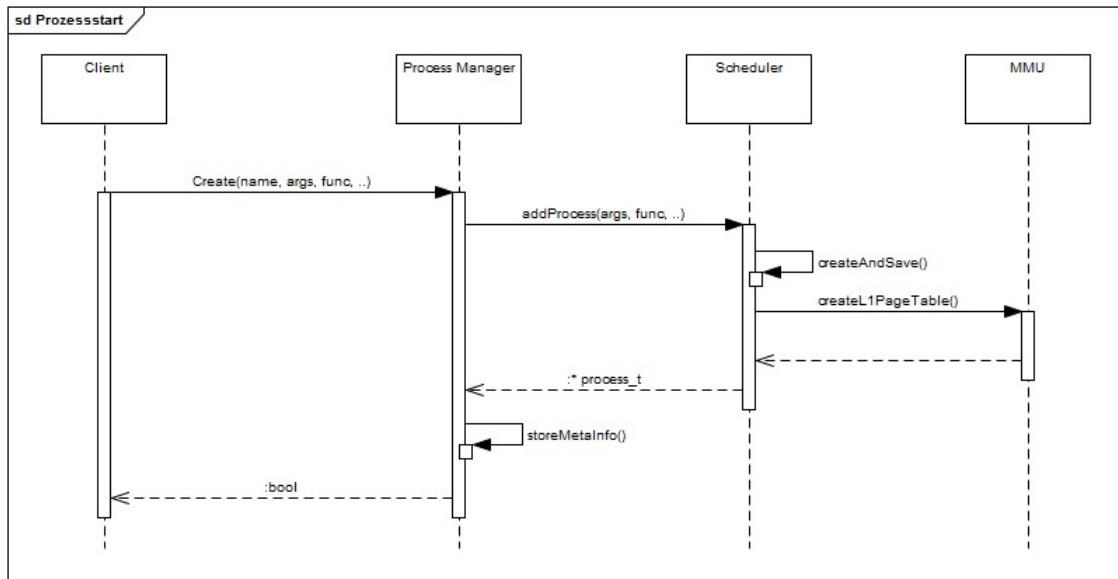


Abbildung 6: Sequenzdiagramm der Prozessverwaltung

7 Virtuelle Speicherverwaltung

Bei der virtuellen Speicherverwaltung erfolgt die Umwandlung von vom ARM Prozessor generierten, virtuellen Adressen in physikalische Adressen durch die MMU. Dieses Kapitel enthält die Beschreibung des Designs und der Implementierung der virtuellen Speicherverwaltung des Betriebssystems sowie der Einstellungen der MMU.

7.1 Grundlegende Funktionsweise

Die Virtual Memory System Architecture for ARMv7 (VMSAv7) definiert zwei unabhängige Formate für translation tables [1, S. B3-1318]:

- *Short-descriptor format:*
 - zweistufige Seitentabelle
 - 32-bit Deskriptoren (PTE)
 - 32-bit virtuelle Eingangsadresse
 - bis zu 40-bit große physikalische Ausgangsadresse
- *Long-descriptor format:*
 - dreistufige Seitentabelle
 - 64-bit Deskriptoren (PTE)
 - verwendet *Large Physical Address Extension* (LPAE)
 - bis zu 40-bit große virtuelle Eingangsadresse
 - bis zu 40-bit große physikalische Ausgangsadresse

Um die Anforderungen an das Betriebssystem zu erfüllen, reicht das zweistufige Seitentablensystem vollkommen aus. Tabelle 5 fasst die wichtigsten gegebenen Eigenschaften unter Verwendung des Short-descriptor format zusammen.

Eigenschaft	Speicherbedarf
Virtueller Speicher	4 GB
Größe eines Page Table Entry (PTE)	4 Byte
Einträge L1 Page Table	4096
Einträge L2 Page Table	256
Speicherbedarf L1 Page Table	4 Byte * 4096 = 16kB
Speicherbedarf L2 Page Table	4 Byte * 256 = 1kB
Unterstützte Pagegrößen:	<i>small page</i> (4 kB), <i>large page</i> (64 kB)
Unterstützte Sectiongrößen:	<i>section</i> (1 MB), <i>supersection</i> (16 MB)

Tabelle 5: Eigenschaften der virtuellen Speicherverwaltung der ARMv7-Architektur

Generiert die ARM CPU einen Speicherzugriff, wird von der MMU ein Suchlauf durchgeführt. Dieser Suchlauf wird *translation table lookup* genannt. Dabei wird zuerst im Translation Lookaside Buffer (TLB) nachgesehen, ob einer der 64 Einträge des TLB die zur virtuellen Adresse korrespondierende physikalische Adresse enthält. Ist dies der Fall (so genannter *TLB hit*), wird der Suchlauf an dieser Stelle erfolgreich beendet.

Ist die angeforderte virtuelle Adresse nicht im TLB enthalten (TLB miss), wird ein page table walk durchgeführt. Das Funktionsprinzip des zweistufigen Seitentabellensystems zeigt Abbildung 7. Aus einem der zwei Seitentabellenregister wird die Basisadresse der darin zuvor abgelegten L1-Seitentabelle geholt. Das Format der PTE bestimmt dann, um welchen Typ von Verweis es sich handelt. Seitentabellen und ihre Einträge werden im nachfolgenden Abschnitt 7.3 genauer beschrieben.

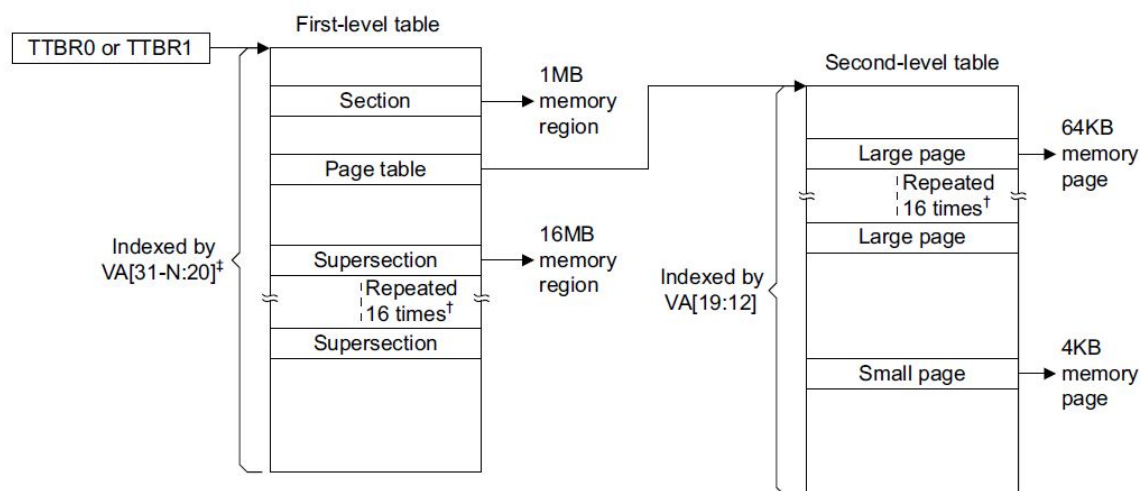


Abbildung 7: Zweistufiges Seitentabellensystem [1, S. B3-1325]

7.2 Umwandlung virtueller Adressen zu physikalische Adressen

Der genaue Vorgang der Umwandlung einer vom ARM Prozessor erzeugten virtuellen Adresse in eine physikalische Speicheradresse zeigen die nachfolgenden beiden Abbildungen. Abbildung 8 zeigt die Umwandlung einer virtuellen Adresse in die physikalische Adresse einer 1 MB Section ohne Verwendung einer L2-Seitentabelle, Abbildung 9 diejenige einer virtuellen Adresse in ein 4 kB page frame unter Verwendung einer L2-Seitentabelle. Die Umwandlung wird vollständig durch die Prozessor-Hardware durchgeführt.

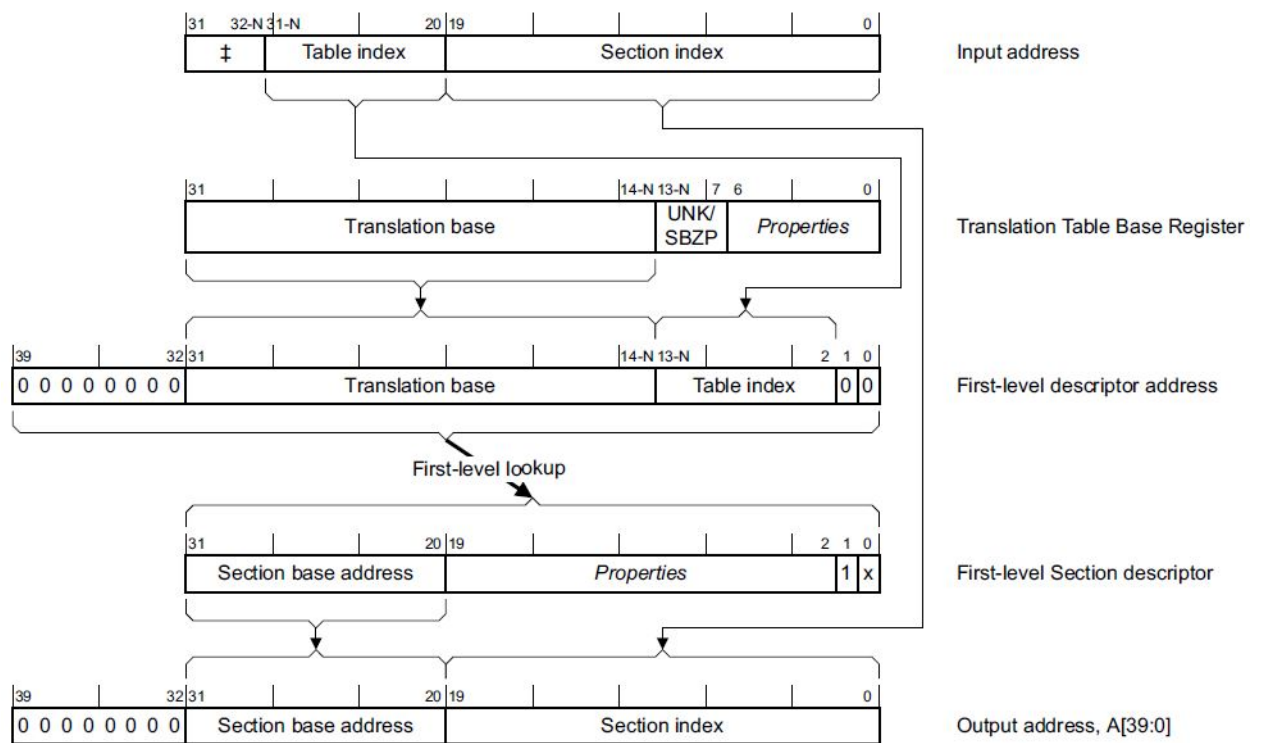


Abbildung 8: 1 MB Section Translation durch die ARM CPU [1, S. B3-1335]

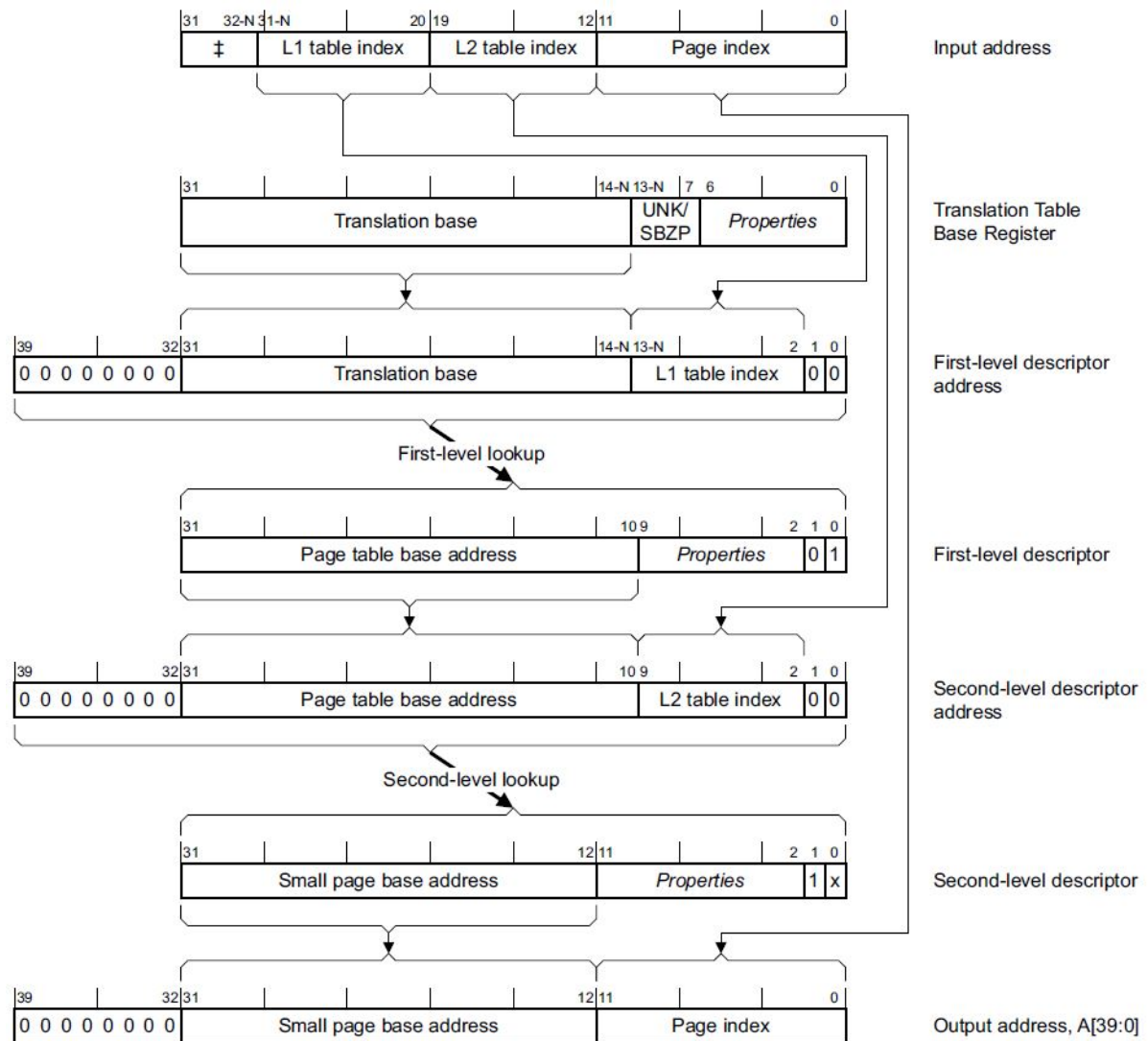


Abbildung 9: Small Page Translation durch die ARM CPU [1, S. B3-1337]

7.3 Seitentabellen und Seitentableneinträge

Der verwendete ARM Prozessor verfügt über zwei Register (Translation Table Base Register (TTBR), *TTBR0* und *TTBR1*), welche Startadressen von Seitentabellen enthalten [1, S. B3-1320]. Ihre Formate sind nahezu identisch und in den Abbildungen 10 und 11 zu sehen. Diese Register übernehmen im Betriebssystem die folgende Funktion:

- **TTBR0:** Wird für prozessspezifische Adressen verwendet. Jeder Prozess erhält bei seiner Initialisierung eine eigene L1-Seitentabelle. Bei einem Kontextwechsel erhält das TTBR0 eine Referenz auf L1-Seitentabelle des neuen Kontextes/Prozesses.

- TTBR1: Wird für das Betriebssystem selbst und für memory-mapped I/O verwendet. Diese ändern sich bei einem Kontextwechsel nicht.

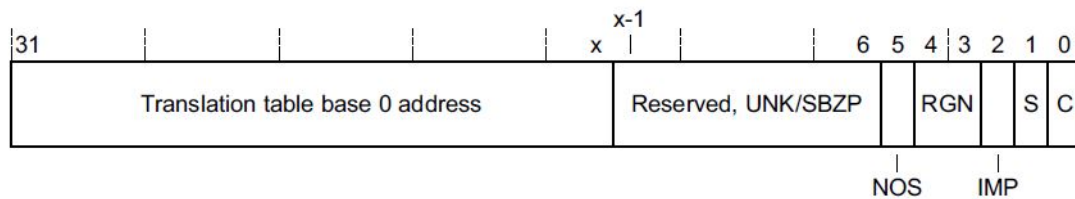


Abbildung 10: TTBR0 Format [1, S. B4-1726]

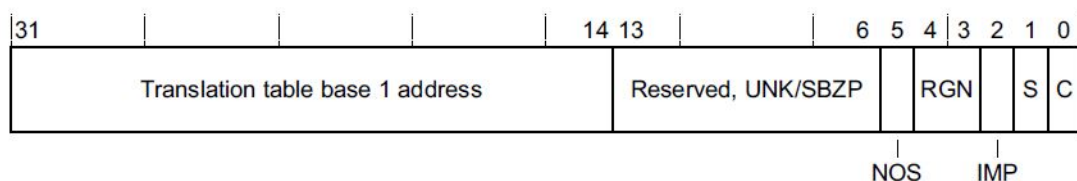


Abbildung 11: TTBR1 Format [1, S. B4-1730]

Das Beschreiben der Seitentabellenregister erfolgt, wie bei nahezu jeder MMU-Funktionalität, mittels Assemblerbefehlen, die auf die CP15 Coprozessor Register zugreifen.

Beim Füllen der Seitentabellen sind vorgegebene Formate für die beiden Typen von Deskriptoren unbedingt zu beachten. Die Abbildungen 12 und 13 fassen die Formate für first-level und second-level Deskriptoren zusammen. Beiden Deskriptortypen gleich ist die vorgeschriebene Länge von 32 Bit.

First-level Deskriptoren

Die First-Level Deskriptortypen werden auf folgende Weise verwendet:

- sections für die Master Page Table (MPT) (siehe Abschnitt 7.4)
- page table für L1-Seitentabellen von Prozessen (siehe Abschnitt 7.4)

Für die Erstellung von first-level Deskriptoren wurde eine Struktur erstellt, welche in Listing 9 aufgeführt ist. Diese Struktur und jene des second-level Deskriptors wird bei den nachfolgenden Erläuterungen zur MMU benötigt.

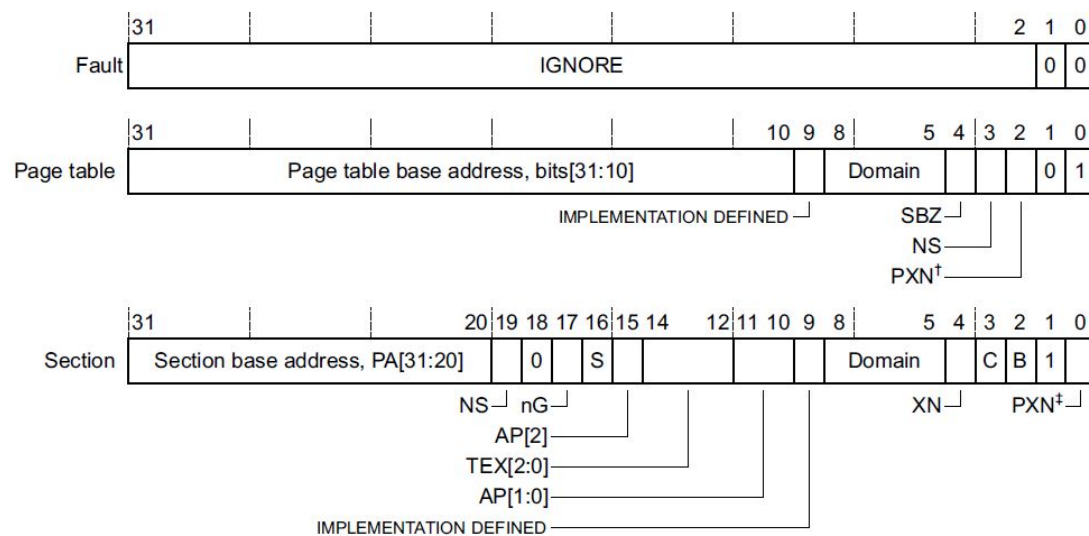


Abbildung 12: First-Level Deskriptorformate [1, S. B3-1326]

```

1 typedef struct
2 {
3     unsigned int sectionBaseAddress;
4     unsigned int accessPermission : 2;
5     unsigned int domain : 4;
6     unsigned int cachedBuffered : 2;
7     unsigned int descriptorType : 2;
8 }
9 firstLevelDescriptor_t;

```

Listing 9: Struktur für first-level Deskriptoren

Second-level Deskriptoren

In der Speicherverwaltung des Betriebssystems werden ausschließlich small pages verwendet. Ausschlaggebende Gründe, warum small pages den Vorzug gegenüber large pages erhielten, sind die folgenden:

- small pages müssen nur einmal in die L2-Seitentabelle eingetragen werden, large pages hingegen 16 mal
- L1- und L2-Seitentabellen, die 16 kB bzw. 1 kB Speicher benötigen, belegen bei ihrer Erzeugung nur vier volle page frames bzw. ein page frame physikalischen Speichers zu einem Viertel. Dadurch wird die Speicherfragmentierung verglichen mit large pages stark verringert

Die Zusammensetzung der Struktur für second-level Deskriptoren ist in Listing 10 dargestellt.

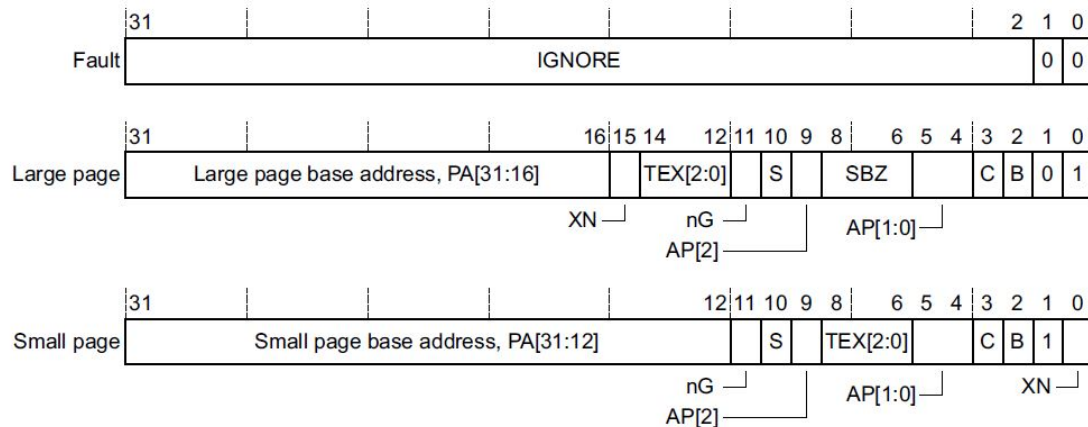


Abbildung 13: Second-Level Deskriptorformate [1, S. B3-1327]

```

1 typedef struct
2 {
3     unsigned int pageBaseAddress;
4     unsigned int accessPermission : 2;
5     unsigned int cachedBuffered : 2;
6     unsigned int descriptorType : 2;
7 } secondLevelDescriptor_t;

```

Listing 10: Struktur für second-level Deskriptoren

7.4 Aufteilung des virtuellen Speichers und Mapping

Die Speicherverwaltung des Betriebssystems kann Abbildung 14 entnommen werden. Die rechte Seite stellt das physikalische Speichermapping dar und wurde dem Datenblatt des ARM [2, S. 155] entnommen. Die linke Seite zeigt die Aufteilung des virtuellen Speichers.

Organisiert ist der virtuelle Speicher in Speicherregionen. Eine zusätzliche Aufteilung betrifft die Zuständigkeitsbereiche für die Seitentabellenregister TTBR0 und TTBR1. Der ARM Cortex-A8 bietet die Möglichkeit, den virtuellen Speicher in einen *Prozessbereich* und einen *Kernelbereich* aufzuteilen. Der Prozessbereich enthält dabei alle virtuellen Adressen, die für Prozesse zugänglich sind. Der Kernelbereich enthält Komponenten, die sich bei Prozesswechseln nicht ändern. Dazu zählen das Betriebssystem selbst sowie die memory-mapped I/O.

Die Einstellungen zur Aufteilung des virtuellen Speichers werden im TTBCR (Translation Table Base Control Register) vorgenommen. Die möglichen Aufteilungsbereiche finden sich in Tabelle B3-1, [1, S. B3-1330].

Physikalisch steht 1 GB Speicher für die page frames zur Verfügung. Dieser wird im virtuellen Speicher an die Adressen 0x00000000 bis 0x3FFFFFFF gemapped. Die Komponenten der Ker-

nelregion, die sich bei Prozesswechseln nicht ändern, beginnen bei Adressen ab 0x40000000. Damit ergibt sich eine Aufteilung des virtuellen Speichers, wie sie in Abbildung 14 dargestellt ist, mit der Bereichsgrenze 0x40000000.

Die Adressen ab der Bereichsgrenze bis zu den vollen 4 GB virtuellem Speicher bei der Adresse 0xFFFFFFFF werden in eine so genannte L1 MPT gemapped. Bei der Aktivierung der MMU wird die Adresse dieser master page table in das Register TTBR1 geschrieben. Danach wird TTBR1 während der Laufzeit des Betriebssystems nicht mehr verändert.

Bei der Initialisierung eines Prozesses wird für den Prozess eine L1 page, die den Prozessbereich abdeckt, angelegt. Soll ein Prozess zur Ausführung gebracht werden, muss seine L1 page table in das TTBR0 geschrieben werden. Das TTBR0 muss zur Laufzeit des Betriebssystems bei Kontextwechseln von Prozessen aktualisiert werden.

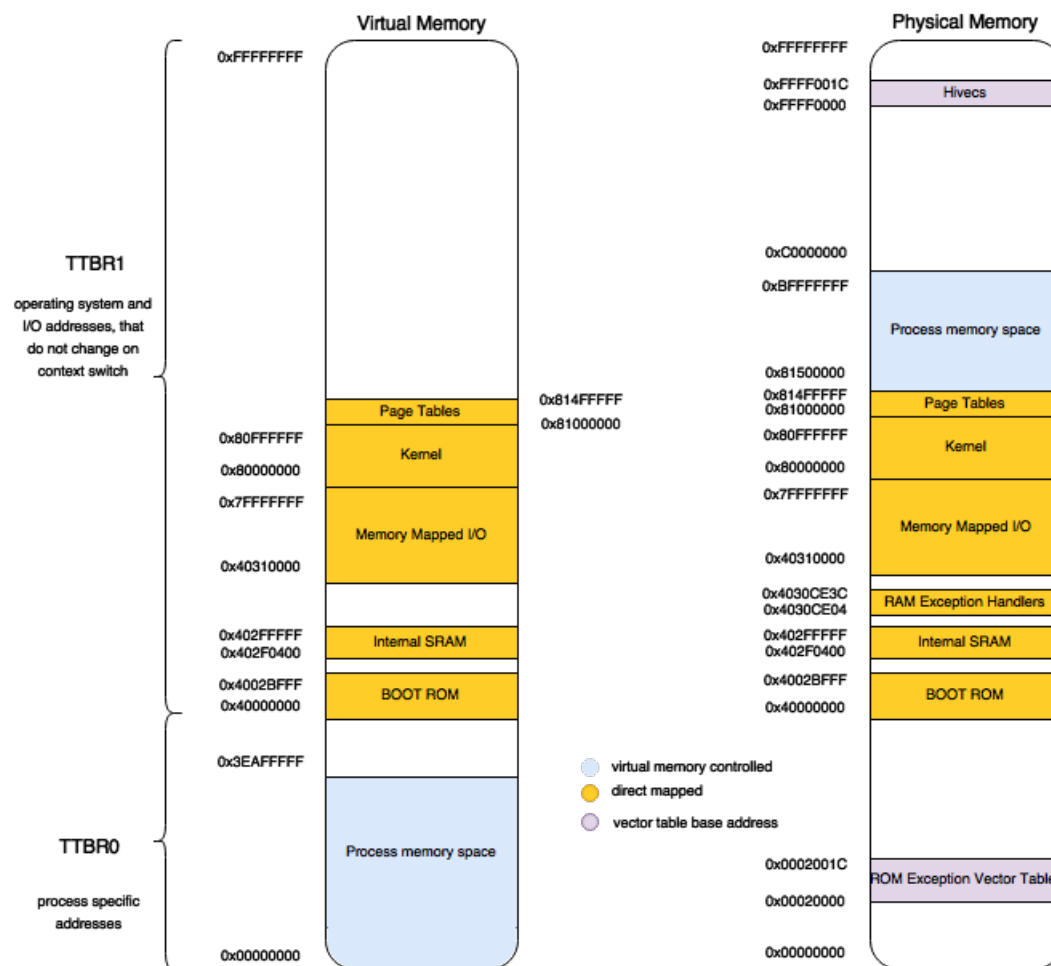


Abbildung 14: Memory Map des Betriebssystems

Eigenschaft	Beschreibung
Größe der Pages	4 kB
Virtueller Speicher für Prozesse	1003 MB
Max. Anzahl von L1 und L2 Page Tables	320 L1 Page Tables oder 1 L1 Page Table + 1276 L2 Page Table
Theoretisch Max. Anzahl von Prozessen	320

Tabelle 6: Eigenschaften der virtuellen Speicherverwaltung des OS

7.4.1 Speicherregionen

Das nachfolgende Listing 11 zeigt die Struktur, mit welcher Regionen im virtuellen Speicher erstellt und verwaltet werden. Sie bieten die Möglichkeit, unterschiedlich große Bereiche des virtuellen Speichers mit denselben Eigenschaften und Zugriffsrechten zu versehen.

Erstellt werden solche Speicherregionen sämtliche in Abbildung 14 gezeigten Bereiche. Sie enthalten die virtuelle Anfangs- und Endadresse der Region sowie Pagegröße und Zugriffsrechte auf die Region. Weiters enthalten sie eine verkettete Liste von Strukturen, die den Status(reserviert oder nicht reserviert) der einzelnen Pages verwaltet.

```

1 typedef struct region
2 {
3     unsigned int startAddress;
4     unsigned int endAddress;
5     unsigned int pageSize;
6     unsigned int accessPermission;
7     unsigned int cacheBufferAttributes;
8     unsigned int reservedPages;
9     pageStatusPointer_t pageStatus;
10 } memoryRegion_t;

```

Listing 11: Struktur für die Verwaltung von Speicherregionen

Zusammengefasst dargestellt sind in Tabelle 7 alle Speicherregionen des Betriebssystems. Ein direktes Mapping bedeutet dabei, dass die virtuelle Adresse der physikalischen entspricht.

Region	Mapping	Größe	Beschreibung
Page Tables	direkt	5 MB	Speicherort für L1 und L2 page tables
Kernel	direkt	16 MB	Speicherort für das Betriebssystem
Memory-Mapped I/O	direkt	1 GB	Peripheriemodule
Exception Handlers	direkt	4 kB	Enthält die Exception vector table
Internal SRAM	direkt	64 kB	Enthält die Exception handler
BOOT ROM	direkt	192 kB	für zukünftige Erweiterungen
Process memory space	virtuell	1 GB	Speicherbereich für Prozesse

Tabelle 7: Angelegte Speicherregionen

7.4.2 Master Page Table

Um das Mapping der MPT verstehen zu können, wird nochmals auf den Adresstranslationsablauf in Abbildung 8 verwiesen. Alle direkt gemappten Regions aus Tabelle 7 werden in die L1 MPT als 1 MB Sections gemapped.

Die Adresse eines Eintrags in der page table setzt sich zusammen aus der Basisadresse der entspreche page table und einem Index. Nach dem setzen der Attribute des page table Eintrags wird durch die Funktion *mmuGetTableIndex* aus den obersten Bits der physikalischen Adresse der Index in der page table berechnet. Der Index muss um 2 bit nach links geschiftet werden, um das Alignment von 4 Byte einzuhalten. Schließlich wird der geschiftete Index noch durch die Datentypgröße von 4 Byte geteilt. Damit wird die korrekte Adresse des zu schreibenden Tabelleneintrags durch Pointerarithmetik ermittelt. An diese Adresse wird nun der Eintrag geschrieben, der zuvor durch die Funktion *mmuCreateL1PageTableEntry* aus der übergebenen first-level Deskriptorstruktur erstellt wurde. Listing 12 zeigt die praktische Ausführung des direkten Mappings in die MPT.

```

1 static void mmuMapDirectRegionToKernelMasterPageTable(memoryRegionPointer_t memoryRegion
    ↳ , pageTablePointer_t table)
2 {
3     unsigned int physicalAddress;
4     firstLevelDescriptor_t pageTableEntry;
5
6     for(physicalAddress = memoryRegion->startAddress; physicalAddress < memoryRegion->
    ↳ endAddress; physicalAddress += 0x100000)
7     {
8         pageTableEntry.sectionBaseAddress = physicalAddress & UPPER_12_BITS_MASK;
9         pageTableEntry.descriptorType     = DESCRIPTOR_TYPE_SECTION;
10        pageTableEntry.cachedBuffered     = WRITE_BACK;
11        pageTableEntry.accessPermission   = AP_FULL_ACCESS;
12        pageTableEntry.domain             = DOMAIN_MANAGER_ACCESS;
13
14        uint32_t tableOffset = mmuGetTableIndex(physicalAddress, INDEX_OF_L1_PAGE_TABLE,
    ↳ TTBR1);
15

```

```

16 // see Format of first-level Descriptor on p. B3-1335 in ARM Architecture Reference
    ↪ Manual ARMv7 edition
17 uint32_t *firstLevelDescriptorAddress = table + (tableOffset << 2)/sizeof(uint32_t);
18 *firstLevelDescriptorAddress = mmuCreateL1PageTableEntry(pageTableEntry);
19 }
20 }

```

Listing 12: Funktion für direktes Mapping in die master page table

7.5 Allokierung der Page Frames

Für die Verwaltung der page frames wurde eine Bitmap verwendet. Abbildung 15 zeigt das Prinzip. Die Bitmap wird durch ein Array der Länge $N/8$ Bytes realisiert. N steht hier für die Anzahl der page frames. Das i -te Bit im n -ten Byte der Bitmap definiert den Verwendungszustand des $(n*8 + i)$ -ten page frame.

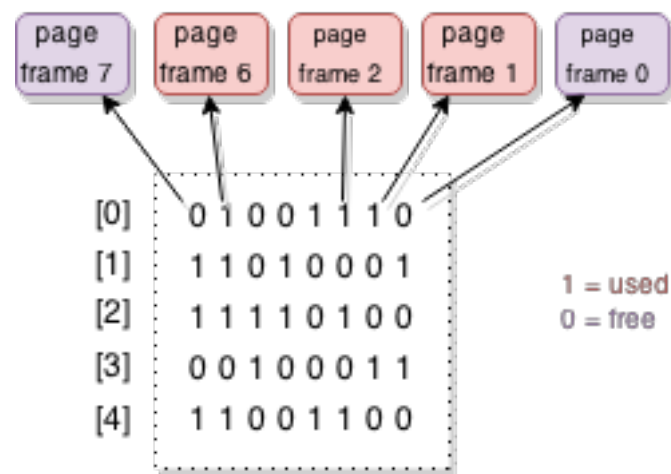


Abbildung 15: Beispiel einer Bitmap zur Verwaltung der Page Frames

7.5.1 Allokation von Page Frames bei Data Abort Exception

Der Vorgang der Einlagerung von page frames durch den Data Abort (DABT) Handler ist in Abbildung 16 dargestellt. Zuerst werden aus dem *Data Fault State Register* (DFSR) der Fehlerzustand und aus dem *Data Fault Address Register* (DFAR) die zugegriffene virtuelle Adresse geladen. Eine Liste aller möglichen Fehlerzustände kann unter [1, B3-1415] eingesehen werden.

Ohne Ausnahme wird bei allen Zuständen außer den Zuständen 5 und 7 der jeweilige Prozess getötet. Zustand 5 betrifft die Erstellung einer L2 page table und die Einlagerung eines page frame. Zustand 7 betrifft die bloße Einlagerung einer page frame. Die Einlagerung der page frame erfolgt dabei auf sehr ähnliche Weise wie in Listing 12 vorgenommen.

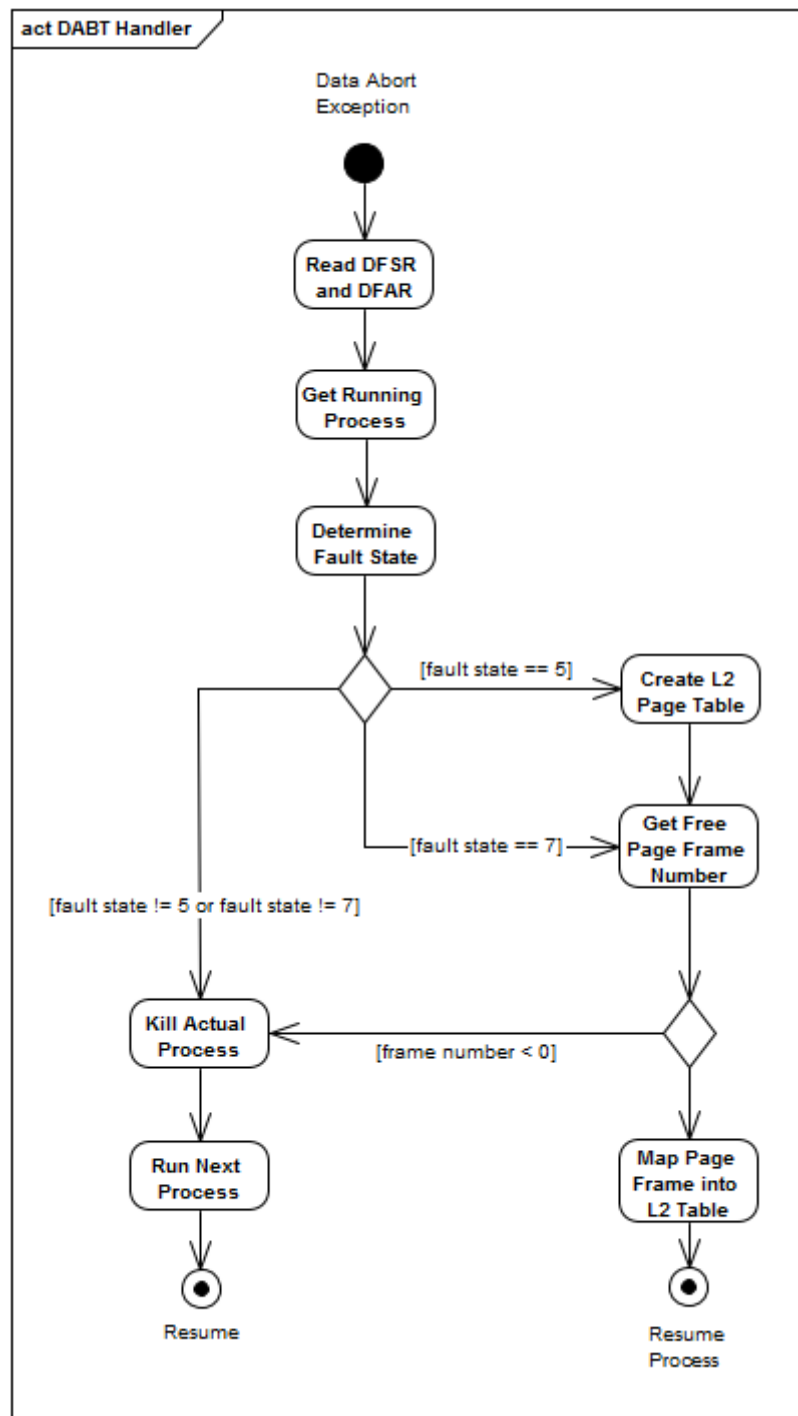


Abbildung 16: Einlagerung von page frames durch den DABT-Handler

7.6 Aktivieren der MMU

Bevor die MMU erfolgreich aktiviert werden kann, muss vorher eine Reihe von Einstellungen gesetzt werden.

Listing 13 zeigt den kompletten Ablauf zur Aktivierung der MMU.

```
1 int MMUInit()  
2 {  
3     MemoryManagerInit();  
4  
5     MMUDisable();  
6  
7     // reserve direct mapped regions so no accidentally reserving of pages can occur  
8     MemoryManagerReserveAllDirectMappedRegions();  
9  
10    // master page table for kernel region must be created statically and before MMU is  
11    // → enabled  
12    mmuCreateMasterPageTable(KERNEL_START_ADDRESS, KERNEL_END_ADDRESS);  
13    mmuSetKernelMasterPageTable(kernelMasterPageTable);  
14    mmuSetProcessPageTable(kernelMasterPageTable);  
15  
16    // MMU Settings  
17    mmuSetTranslationTableSelectionBoundary(BOUNDARY_AT_QUARTER_OF_MEMORY);  
18    mmuSetDomainToFullAccess();  
19  
20    MMUEnable();  
21  
22    return MMU_OK;  
23 }
```

Listing 13: Aktivierung der MMU

7.7 Interaktion der MMU mit Prozessen

Die Schnittstelle der Softwareimplementierung der MMU zeigt Listing 14.

```
1 extern int MMUInit(void);  
2 extern int MMUSwitchToProcess(process_t* process);  
3 extern int MMUInitProcess(process_t* process);  
4 extern void MMUHandleDataAbortException(void);  
5 extern int MMUFreeAllPageFramesOfProcess(process_t* process);
```

Listing 14: Softwareschnittstelle der MMU

Die Schnittstellenfunktionen werden auf die folgende Weise verwendet:

MMUInit

Initialisiert die Regionen des virtuellen Speichers und die MMU für die Verwendung.

Nach dem Ausführen dieser Funktion ist die MMU eingeschaltet. Bei nach erfolgreichem Ausführen wird als Rückgabewert 1 zurückgeliefert. Diese Funktion wird bei der Initialisierung des Prozess Managers aufgerufen.

MMUSwitchToProcess

Bringt den als Parameter übergebenen Prozess zur Ausführung. Dabei wird der TLB geflusht und die Adresse der L1 page table des Prozesses in das TTBR0 geschrieben.

MMUInitProcess

Erstellt beim Erzeugen eines neuen Prozesses eine L1 page table für diesen Prozess. Die page table wird mit fault entries initialisiert.

MMUHandleDataAbortException

Diese Funktion wird bei jeder Data Abort Exception ausgeführt. Sie wird durch einen in Assembler implementierten Dabt Handler aufgerufen. Die Funktion lädt die virtuelle Adresse, bei deren Zugriff die Data Abort Exception ausgelöst wurde aus dem Data Fault Address Register (DFAR) sowie den Fehlerstatus aus dem Data Fault Status Register (DFSR). Die weitere Vorgehensweise wird in Abhängigkeit vom Fehlerstatus durchgeführt.

MMUFreeAllPageFramesOfProcess

Beim Killen eines Prozesses gibt diese Funktion sämtliche von diesem Prozess belegten page frames in der zur Verwaltung der page frames eingesetzten Bitmap wieder frei.

8 Interprozesskommunikation

Interprozesskommunikation dient zur Kommunikation zwischen verschiedenen Prozessen. Dabei ist entscheidend, dass beiden zu kommunizierenden Prozesse in unterschiedlichen Speicherbereichen bzw. in strikt voneinander getrennten Speicherbereichen sind.

8.1 Aufbau

bla

8.2 IpcManager

Der IpcManager hndelt die Interprozesskommunikation zwischen Prozessen. Listing 15 zeigt die Schnittstelle des Managers.

```
1 extern int IpcManagerRegisterNamespace(char* namespace_name);
2 extern int IpcManagerSendMessage(char* sender_namespace, char* namespace_name, char*
   ↳ message, int len);
3 extern int IpcManagerHasMessage(char* namespace_name);
4 extern int IpcManagerGetNextMessage(char* namespace_name, char* message_buffer, int
   ↳ msg_buf_len, char* sender_namespace, int ns_buf_len);
5 extern int IpcManagerCloseNamespace(char* namespace_name);
6 extern int IpcManagerChannelCount();
7 extern int IpcManagerGetChannel(int index, char* buf, int len);
```

Listing 15: Schnittstelle des IpcManagers

9 System API

Die System API dient als Schnittstelle zum Benutzer bzw. zur Benutzerin. Dabei ist eine saubere Trennung zwischen BenutzerInnen Schnittstelle und Betriebssystem zwingend notwendig.

9.1 Aufbau eines Systemcall Datenpakets

Jedem Systemcall werden Daten mitgegeben, dieses müssen zuvor in eine geeignete Datenstruktur transformiert werden. In Listing 16 ist die gewählte Datenstruktur abgebildet.

```
1 typedef struct {  
2     int callArg;  
3     int callArg2;  
4     int callArg3;  
5     int callArg4;  
6     char* callBuf;  
7     int* returnArg;  
8     char* returnBuf;  
9 } messageArgs_t;
```

Listing 16: Aufbau eines Systemcall Datenpakets

9.2 Vorgehensweise bei einem Systemcall

Das Vorgehen bei einem Systemcall ist in Abbildung 17 ersichtlich.

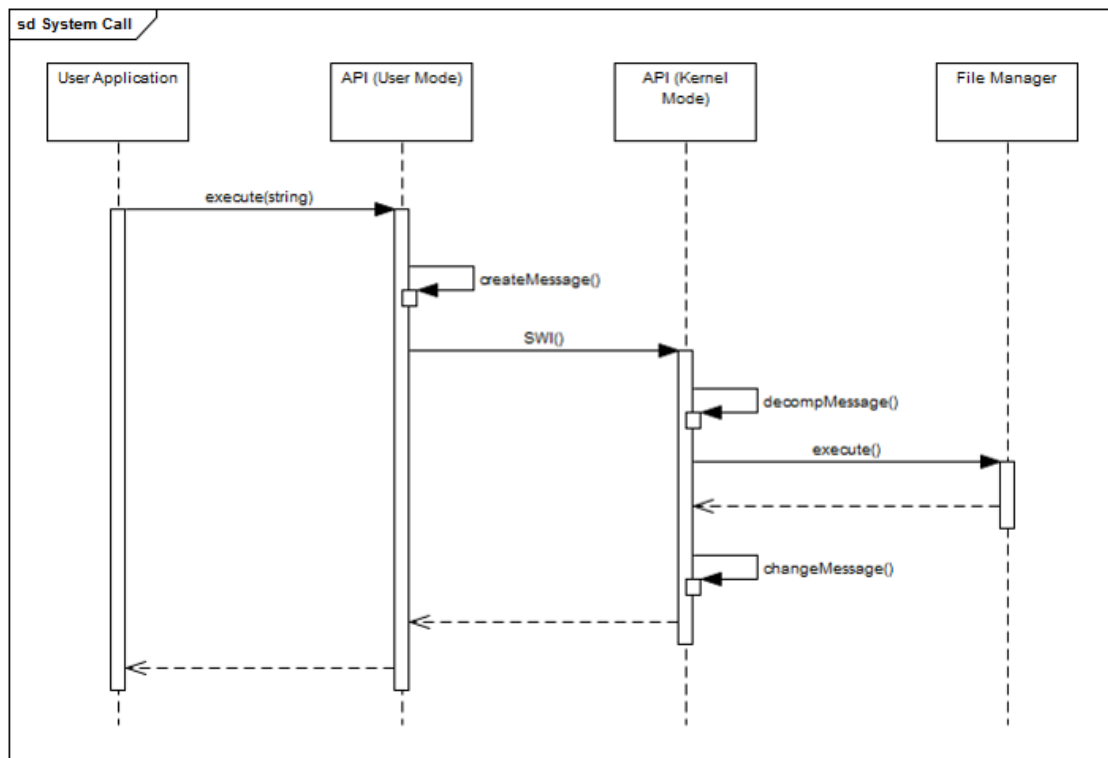


Abbildung 17: Sequenzdiagramm eines Systemcalls

10 Sicherheitsaspekte

Hinsichtlich der Sicherheit wurden an das Betriebssystem die Anforderungen der strikten Trennung von Prozessadressräumen und der Trennung von privilegierten und nichtprivilegierten Modi gestellt. Mögliche Sicherheitsrisiken und deren Vermeidung werden nachfolgend beschrieben.

10.1 Sicherheitsrisiken

Aufschluss über mögliche Sicherheitsrisiken ergibt eine nähere Betrachtung des Speichermodells in Abbildung 14. Wie die Abbildung zeigt, beginnt im virtuellen Speicher der Adressbereich für Prozesse ab der Adresse 0×00000000 . Gleichzeitig befindet sich die Startadresse für die *ROM Exception Vector Table* an der Adresse 0×00020000 . Durch diese Gegebenheiten bestehen zwei grundsätzliche Sicherheitsrisiken:

1. Nulladressenproblem: Adresse 0×00000000 ist im Regelfall reserviert für *Nullpointer*.
2. Anfälligkeit für Hacking durch unsaubere Adressraumtrennung: Die *ROM Exception Vector Table* muss bei dieser Konstellation in den *Page Tables* für Prozesse direkt, d.h. eins-zu-eins, gemappt sein.

Letzteres Sicherheitsrisiko bietet Hackern die Möglichkeit, durch sukzessives Erhöhen der angesprochenen Adresse vom *User Mode* in den *System Mode* zu gelangen. Damit wäre eine Hackeranwendung in der Lage, mit voller Befugnis auf die Hardware zuzugreifen und Programmteile des Kernels auszuführen.

Die Lösung für diese beiden Sicherheitsrisiken wird im Folgenden vorgestellt.

10.2 Vermeidung des Nulladressenproblems

Die Lösung des Nulladressenproblems kann mit relativ wenig Aufwand erreicht werden. In der *Memory Region* für den Prozessadressbereich wird die erste *Page* für alle Prozesse bereits beim Erstellen reserviert. Dadurch wird vermieden, dass bei einer Speicherallokation die Nulladresse oder eine *non-aligned* Adresse ausgegeben wird. Zusätzlich werden im *DABT-Handler*, der für die Einlagerung von Adressen von *Page Frames* in die *Page Tables* von Prozessen zuständig ist, diese nun nicht erlaubten Adressen abgefangen. Tritt aus welchem Grund auch immer eine Adresse aus dem Adressbereich der ersten *Page* im *DABT-Handler* auf, wird der entsprechende Prozess beendet und der nächste bereite Prozess zur Ausführung gebracht.

10.3 Implementierung der *High Vectors*

Das Problem der sauberen Trennung der Adressräume für Prozesse und für den Kernel sowie der sauberen Trennung der Benutzermodi wird durch die Implementierung der *High Vectors* oder auch *Hivecs* erreicht.

Die Implementierung der *Hivecs* versetzt die Basisadresse der Exceptions auf die Adresse $0 \times \text{FFFF}0000$. Damit liegt die Basisadresse über der festgelegten Adressbereichsgrenze eindeutig im Kernelbereich, siehe dazu den physikalischen Bereich in Abbildung 14. Bei den *low*

vecs mussten bei der Erstellung eines jeden Prozesses die Adressen der *Exception Vector Table* ab 0x00020000 bis 0x0002001C in die *Page Table* der Prozesse direkt gemappt werden. Bei den *Hivecs* werden die Adressen 0xFFFF0000 bis 0xFFFF001C in die *Kernel Master Page Table* direkt gemappt. Damit ist ein Hackangriff durch eine Anwendung wie oben beschrieben nicht mehr möglich. Die Adressen 0x00000000 bis exklusive 0x40000000 stellen nun ausschließlich den Prozessbereich und die Adressen 0x40000000 bis 0xFFFFFFFF ausschließlich den Kernelbereich dar. Insgesamt müssen für die Implementierung der *Hivecs* folgende Schritte unternommen werden:

Laden der *Exception Vecotrs* : Im *Linker Script* müssen die Startadressen der *RAM Exceptions* (siehe [2, S. 4100]) an die Basisadresse 0xFFFF0000 gelegt werden

Mappen der *Hivecs* : Die Basisadresse der *Hivecs* muss in die *Kernel Master Page Table* direkt gemapped werden

Einschalten der *Hivecs* : Im System Control Register (SCTLR) muss das 13. Bit (V-bit) gesetzt werden [1, S. 1164]

10.4 Umgesetzte Sicherheitsaspekte

In der aktuellen Implementierung wurden die oben angeführten Sicherheitsaspekte vollumfänglich umgesetzt, allerdings wurde das Mapping der *High Vectors*, aufgrund fehlender Testzeit, vorläufig deaktiviert.

11 BenutzerInnen-Anwendung

Bei der BenutzerInnen-Anwendung handelt es sich um die Ansteuerung eines Moving Heads mittels Digital Multiplex (DMX) Protokoll.

11.1 Grundlegender Aufbau des DMX Protokolls

Es gibt mehrere verschiedene Spezifikationen für das DMX Protokoll. Im folgenden wird eine dieser unterschiedlichen Spezifikationen erläutert und anschließend zu Vergleichszwecken verwendet. Abbildung 18 dient zur Veranschaulichung des DMX-512 Protokolls.

TODO!!!

Abbildung 18: DMX Protokoll

Tabelle 8 beschreibt die einzeln nummerierten Markierungen aus Abbildung 18.

Nummer	Signalname	Min.	Typ.	Max.	Einheit
1	Reset	88.0	88.0	-	μ s
2	Mark zwischen Reset- und Startbyte	8.0	-	1 s	μ s
3	Frame-Zeit	43.12	44.0	44.48	μ s
4	Startbit	3.92	4.0	4.08	μ s
5	LSB (niederwertigstes Datenbit)	3.92	4.0	4.08	μ s
6	MSB (höchstwertigstes Datenbit)	3.92	4.0	4.08	μ s
7	Stopbit	3.92	4.0	4.08	μ s
8	Mark zwischen Frames (Interdigit)	0	0	1.0	s
9	Mark zwischen Paketen	0	0	1.0	s
-	Reset-Reset (Paketabstand)	1094	-	-	μ s

Tabelle 8: Eigenschaften des DMX-512-Protokolls

Die Übertragungsgeschwindigkeit ist bei allen Protokollarten identisch und beträgt 250 kBaud, d.h. jedes Bit hat eine Dauer von 4μ s. Das DMX Protokoll besitzt 512 verschiedene Kanäle, wobei jeder Kanal mithilfe eines Datenbytes gesteuert wird. In Abbildung 18 ist ersichtlich, dass jedes übertragene Datenbyte zusätzlich ein Startbit sowie zwei Stopbits besitzt. Somit ergeben sich für jeden Kanal genau elf Bits.

11.2 Messergebnisse des Implementierten DMX Protokolls

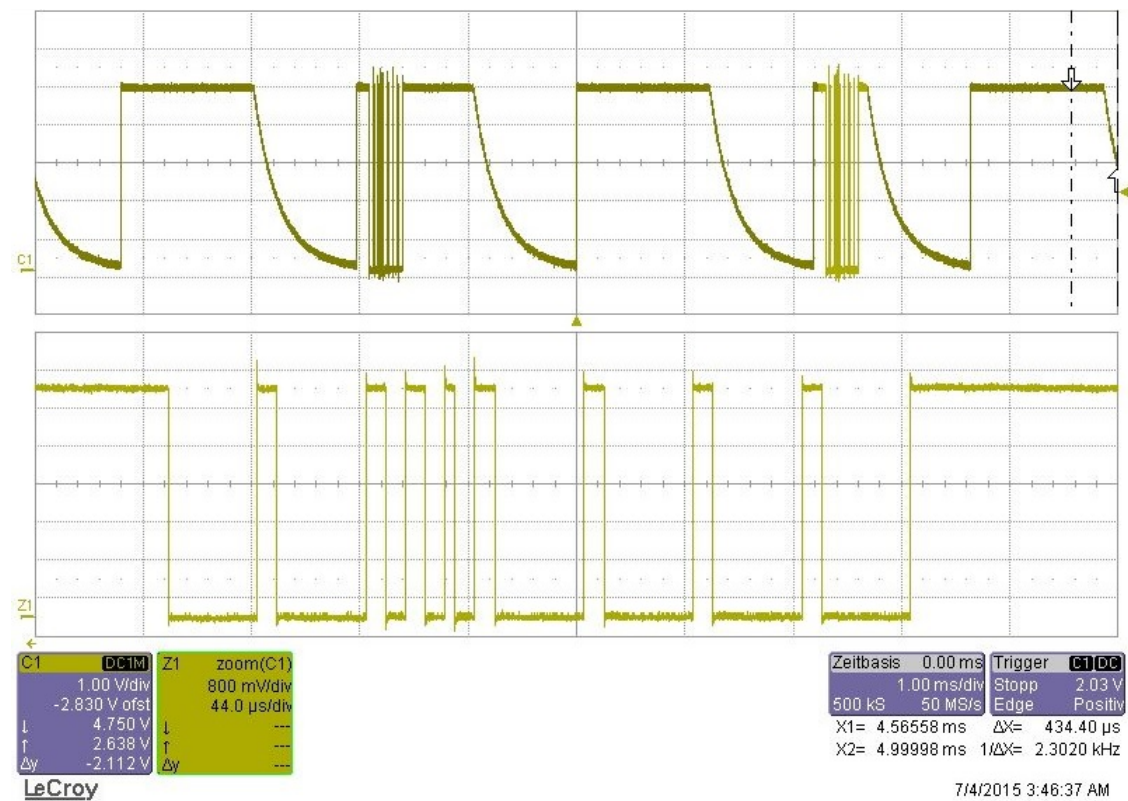


Abbildung 19: DMX Protokoll: Problem fallende Flanke

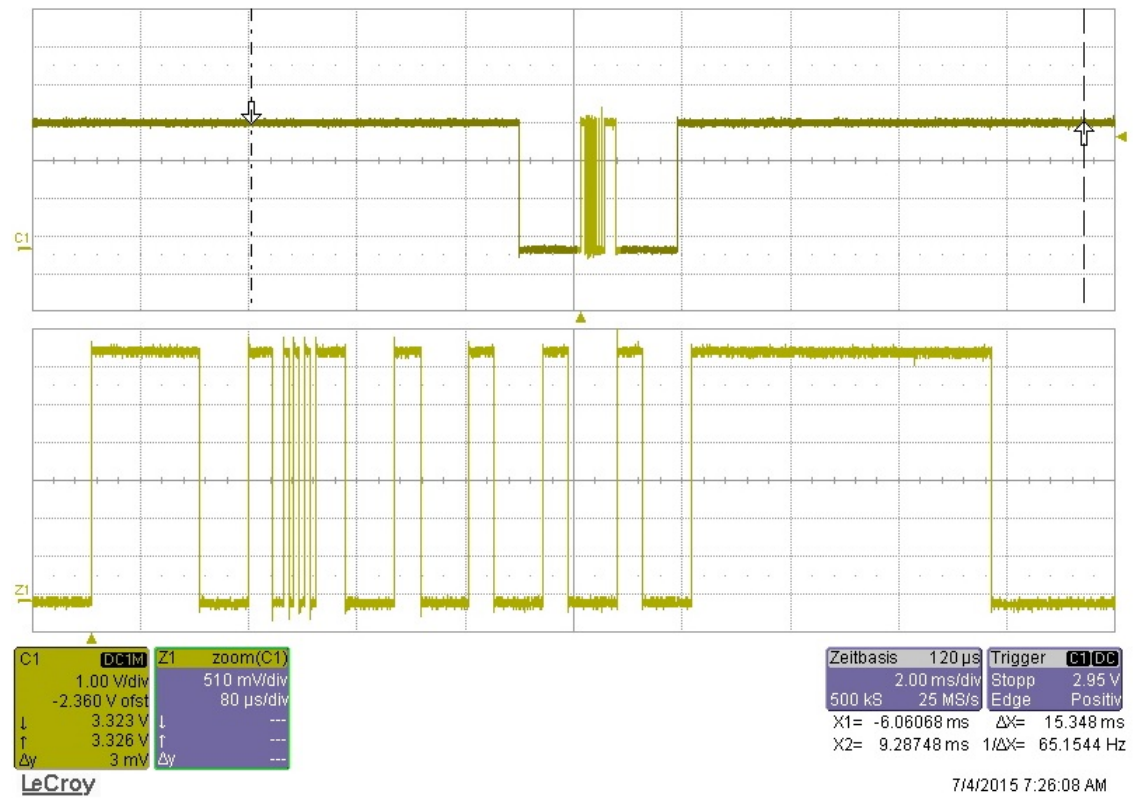


Abbildung 20: DMX Protokoll: Problem Offset Byte

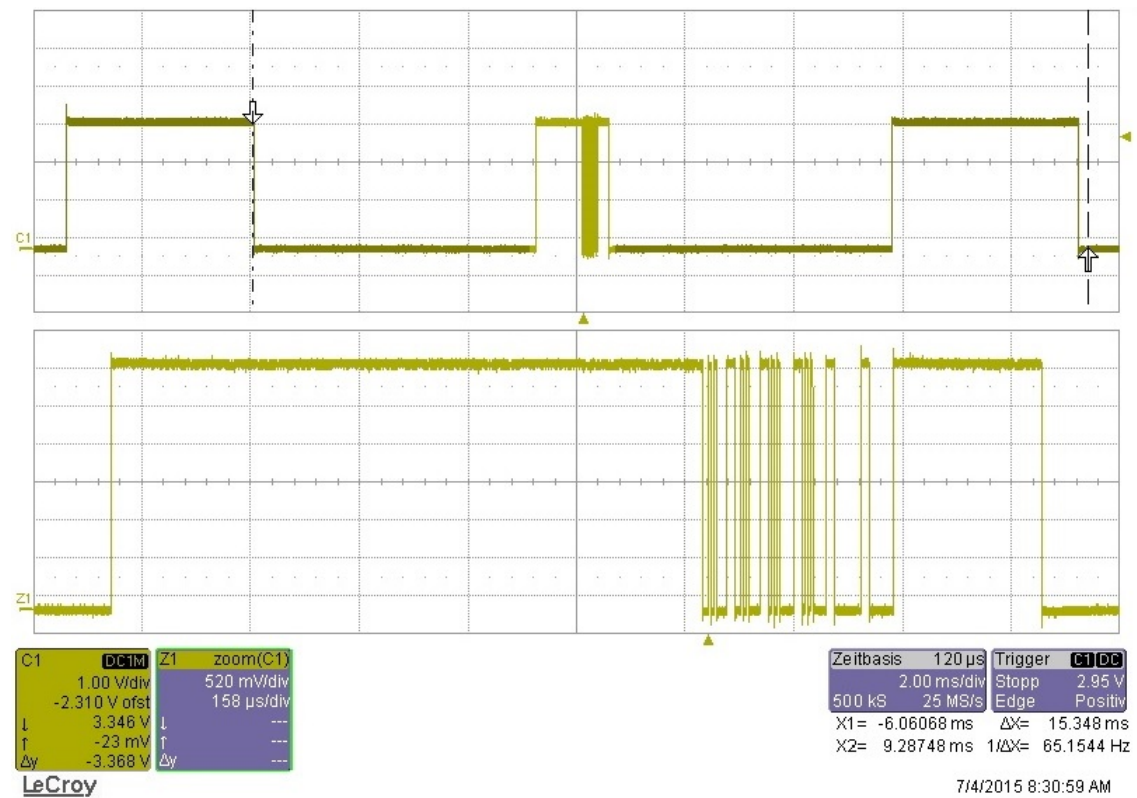


Abbildung 21: Funktionierendes DMX Protokoll

12 Performanzuntersuchungen

In diesem Kapitel werden Performanzaspekte des Betriebssystems dokumentiert und diskutiert. Die Performanz des Betriebssystems wurde mittels verschiedener Experimente und Messungen untersucht, deren Resultate im Folgenden beschrieben sind.

12.1 Messung und Ergebnisse

Um die Performanz des Betriebssystems beurteilen zu können, wurden zeitliche Messungen der relevanten Unterbrechungen vorgenommen. Die Messungen der einzelnen Unterbrechungen wurden mit Hilfe eines Oszilloskops durchgeführt, wobei alle Messungen jeweils zehn mal vorgenommen wurden und schließlich der Durchschnitt über alle Messungen als Resultat herangezogen wurden. In Tabelle 9 sind die einzelnen Messungen und Resultate aufgelistet.

Testfall	Durchschnittszeit ($n = 10$)
MMU Fault State 5	18.90ms
MMU Fault State 7	11.80ms
MMU Freeing Page Frame	19.80ms
Zeitscheibe (effektiv)	10.04ms
Context Switch	375μs

Tabelle 9: Performanz-Messergebnisse

Interessant ist vor allem die vergleichsweise lange Unterbrechung eines *Context Switch*, welcher knapp 3% der gesamten Zeitscheibe eines Prozesses in Anspruch nimmt. In Anbetracht dieser Messergebnisse wäre für die Gesamtp Performanz des Betriebssystems durchaus zu überlegen, entweder die Zeitscheibe eines Prozesses zu verlängern oder aber die Logik für den *Context Switch* zu optimieren.

Von Interesse sind auch die beiden Fälle der Einlagerung eines page frame. MMU *Fault State 5* stellt dabei den Fall der Erstellung einer *Level 2 page table* samt Einlagerung eines *page frame* dar, MMU *Fault State 7* stellt dagegen lediglich die bloße Einlagerung eines *page frame* in eine *Level 2 page table* dar. Nachdem diese Unterbrechungen vergleichsweise selten auftreten, sind hier Optimierungen nicht unbedingt hochprior anzunehmen.

13 Zusammenfassung und Ausblick

In diesem Kapitel werden die erreichten Ergebnisse zusammengefasst. Weiters wird ein Ausblick auf Möglichkeiten der Weiterentwicklung geboten.

13.1 Zusammenfassung

Das primäre Ziel dieses Projektes war die Erlangung tiefergehende Kenntnisse in Bezug auf die Systemprogrammierung von Systemen mit beschränkten Ressourcen. Dabei sollten vor allem die theoretische Grundlagen von Betriebssystemen praktisch umgesetzt werden.

Implementiert und getestet wurde ein Betriebssystem welches sich auch in Langzeittests als stabil erwiesen hat. Das Betriebssystem ist durch den HAL flexibel und ohne größere Aufwände protierbar. Zudem ist es möglich, von einem externen Speichermedium, respektive einer SD-Karte, Applikationen zu laden und auszuführen. Bei der Implementierung wurden sämtliche Grundaspekte moderner Betriebssysteme, wie beispielsweise die Interprozesskommunikation oder die virtuelle Speicherverwaltung, behandelt und umgesetzt. Zudem wurden die Sicherheitsrisiken durch das saubere Trennen der Adressräume und Benutzermodi stark verringert.

Es ist zu erwähnen, dass während des Entwicklungsprozesses erwartete wie auch nichterwartete Probleme aufgetreten sind. Diese betreffen in erster Linie Komplikationen, die durch falsches Setzen der Hardwareregister entstanden sind.

Für das entworfene Betriebssystem wird kein Anspruch auf Vollständigkeit erhoben, da seine Entwicklung agil vorgenommen wurde. So wurden aus Zeitgründen bei der Erstellung des HAL nur die für das Betriebssystem selbst und die vorgesehene DMX-Applikation benötigten Funktionen implementiert.

13.2 Ausblick

Das Betriebssystem ist in der vorliegenden Form voll einsatzfähig und erfüllt alle gesetzten Anforderungen. Durchaus sind aber noch einige essentielle Anforderungen an ein Betriebssystem nicht erfüllt, respektive sind einige Implementierungsdetails noch nicht ganz ausge-reift.

13.2.1 Punkte mit Verbesserungspotential

Einige Punkte des Betriebssystem konnten nicht vollständig abgedeckt werden. Diese Punkte mit Verbesserungspotential werden in Tabelle 10 kurz beschrieben.

Sachverhalt	Beschreibung
Caching	Die aktuelle Speicherverwaltung verwendet über den TLB hinausgehend kein weiteres Caching. Eine Einführung des Cachings hätte eine Verbesserung der Performance zur Folge.
Watchdog	Derzeit lässt durch das forcierte Beenden aller laufenden Prozesse (einschließlich des Leerlaufprozesses) das Betriebssystem in einen Absturz führen. Ein <i>Watchdog</i> könnte für das Wiederherstellen des Leerlaufprozesses bei unerwartetem Beenden verantwortlich sein.
Erweiterung der Konsole	Derzeit ist es nicht möglich einen im Vordergrund gestarteten Prozess durch eine Tastenkombination zu Beenden.

Tabelle 10: Übersicht der Punkte mit Verbesserungspotential

13.2.2 Fehlende Punkte für eine praktische Verwendung des Betriebssystems

Das Betriebssystem weist einige wenige Punkte auf, welche noch nicht implementiert wurden, aber für eine praktische Verwendung fehlen. Tabelle 11 zeigt diese Punkte auf.

Fehlender Punkt	Beschreibung
<i>ResourceManager</i>	Derzeit findet im Betriebssystem keine erweiterte Verwaltung von Ressourcen, respektive Dateien, Geräten usw. statt. Damit ist es grundsätzlich möglich, dass mehrere Prozesse dieselbe Ressource gleichzeitig verwenden und damit Konflikte und unerwartete Ergebnisse auftreten. Ein <i>ResourceManager</i> würde hinsichtlich dieser Problematik die von einem Prozess verwendeten Ressourcen verwalten und ggf. den Zugriff durch andere Prozesse sperren.

Tabelle 11: Übersicht der fehlenden Punkte

Literatur

- [1] ARM Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, 2012. ARM DDI 0406C.b.
- [2] Texas Instruments. *AM335x ARM Cortex-A8 Microprocessors (MPUs) Technical Reference Manual*, 2011. Revised April 2013.