FHV
Vorarlberg University
of Applied Sciences

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 Betriebssystem 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

Abbildungsverzeichnis

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 Betriebsystem 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 ein 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	*Memory Management* 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.
Moving Head mit Digital Multiplex (Bus Protokoll) (DMX)	Auf dem Betriebssystem soll eine Anwendung zur Ansteuerung eines oder mehrerer *Moving Heads* 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 Projektmanagment 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 Projekt 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)**
DeviceManager	`InitDevice`, `OpenDevice`, `ReadDevice`
DriverManager	`GetDriver`
FileManager	`OpenFile`, `OpenExecutable`
MemManager	`GetFreePagesInProcessRegion`, `GetRegion`
ProcessManager	`StartProcess`, `KillProcess`
IpcManager	`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.

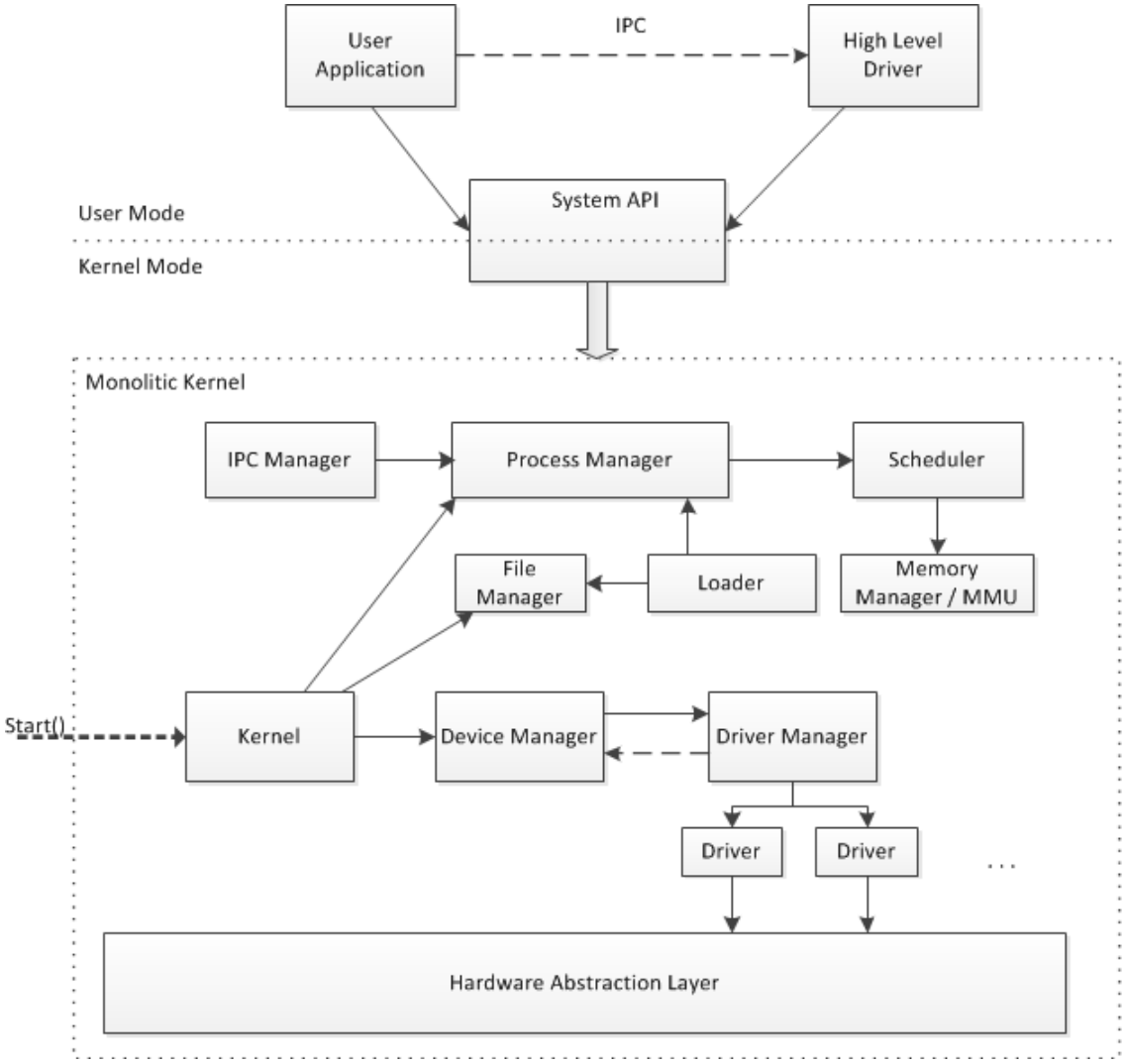

Abbildung 1: Allgemeiner Aufbau der Architektur

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 Geräteebene 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 6.3 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 Verwantwortlichkeit 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, sodass 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. Beispielswiese 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 Hardwareaddressen 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 höherpriore Interrupts die bei ihrem Auftreten niederpriorere 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 behandleten Interrupts aufzurufen.

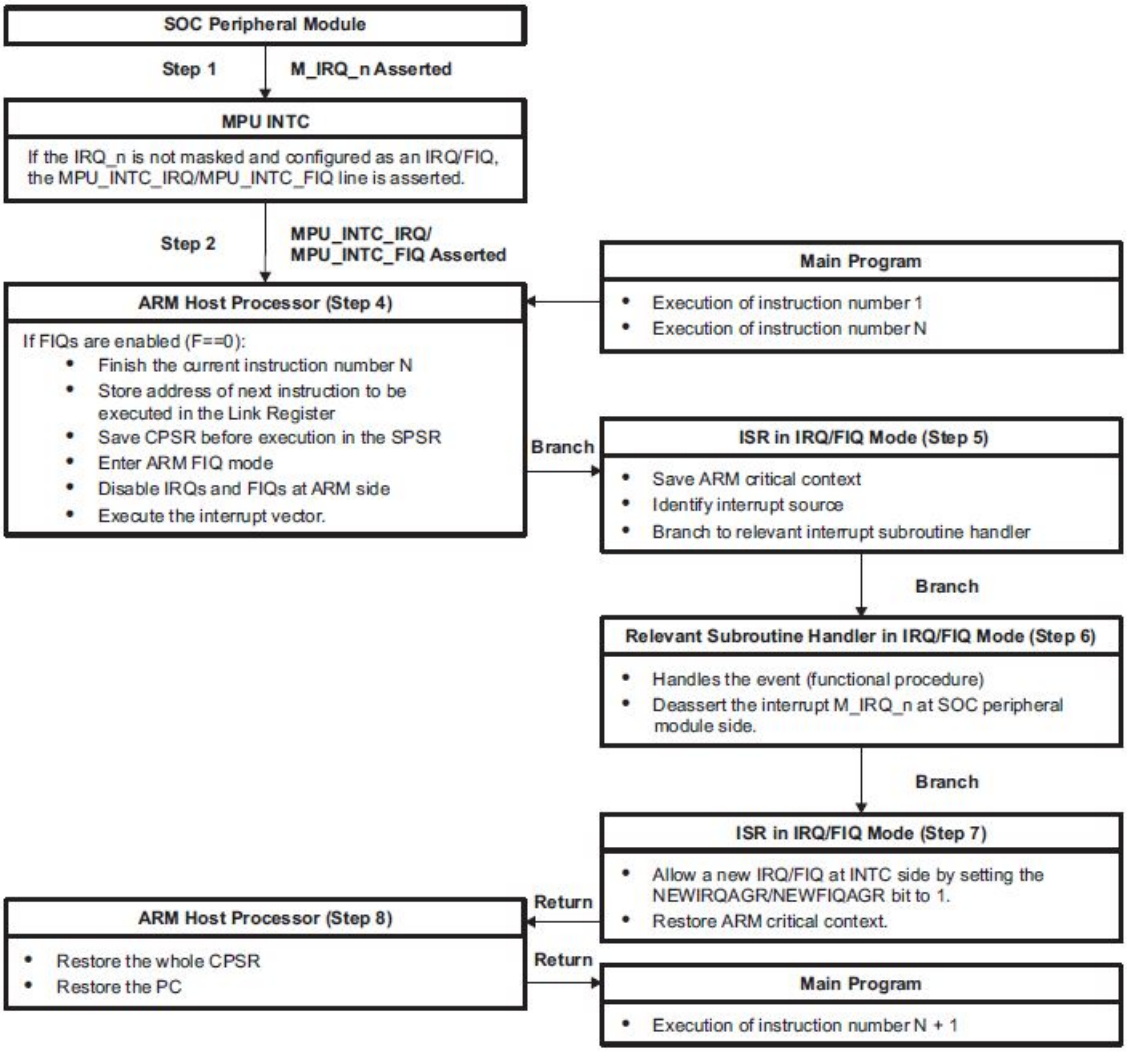

Abbildung 2: IRQ/FIQ Verarbeitung [2, S. 193]

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.

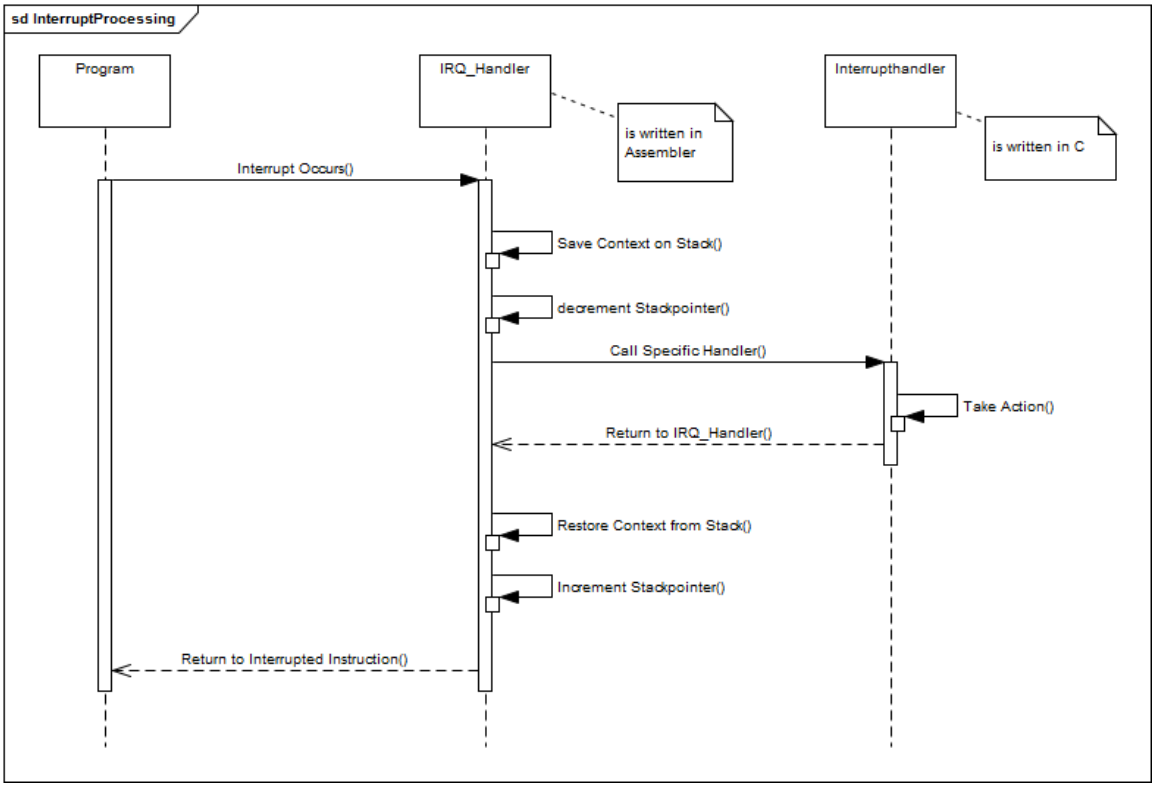

Abbildung 3: Interruptverarbeitung im Betriebssystem

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
          ↪ length);
8 } 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 payload, char* buf, uint16_t len)
3  {
4    if (payload > (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(payload), PIN_VALUE_HIGH);
12       break;
13     case '0':
```

```
14        GPIOSetPinValue(BOARD_LED(payload), 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 payload;
6    };
7  } device_t;
8
9  extern void DeviceManagerInit();
10 extern device_t DeviceManagerGetDevice(char* name, int len);
11 extern int DeviceManagerInitDevice(device_t device);
12 extern int DeviceManagerOpen(device_t device);
13 extern int DeviceManagerClose(device_t device);
14 extern int DeviceManagerRead(device_t device, char* buf, int len);
15 extern int DeviceManagerWrite(device_t device, char* buf, int len);
16 extern int DeviceManagerIoctl(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

Das Betriebssystem weist die Eigenschaften eines präemptiven Multi-Tasking-Systems, mit *Round-Robin*-Verfahren und fixer Zeitscheibe, auf. Für die Verwaltung der einzelnen Prozesse und den Wechsel des aktiven Prozesses, wurde eine Prozessverwaltung implementiert. Im Folgenden sind die beteiligten Komponenten und wichtigen Eigenschaften der Prozessverwaltung dokumentiert.

6.1 Prozesszustände

Jeder Prozess besitzt zu jedem beliebigen Zeitpunkt einen fix definierten Zustand, welche ebenfalls fix definierte Zustandsübergänge aufweisen, d.h. es können keine Inkonsistenzen diesbezüglich auftreten. Abbildung 4 zeigt die verschiedenen Zustände eines Prozesses sowie die jeweilig erlaubten Übergänge zu einem anderen Zustand auf.

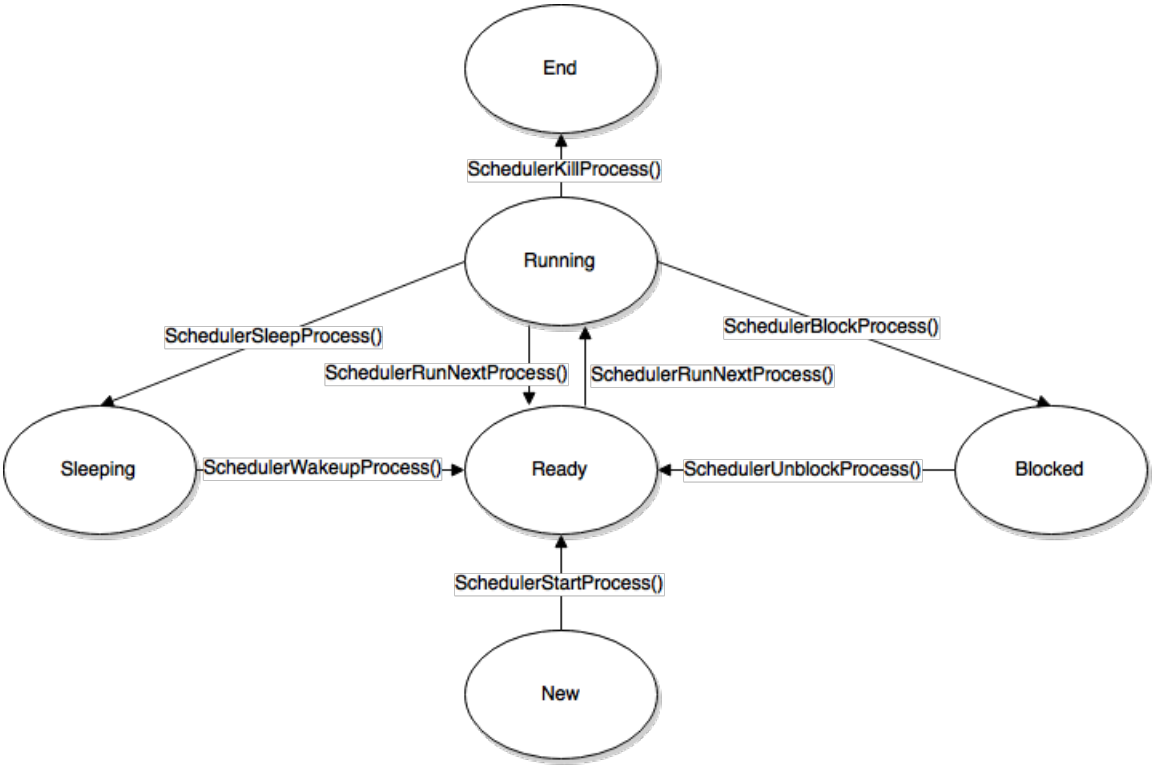

Abbildung 4: Prozesszustände und Zustandsübergänge

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

Ready

Der Zustand *Ready* tritt ein, wenn ein Prozess zur Ausführung bereit ist. Neu erstellte Prozesse befinden sich in jedem Fall im Zustand *Ready*.

Running
Ein Prozess weist diesen Zustand auf, wenn er sich in Ausführung befindet, respektive der Prozessselbst die aktuelle Zeitscheibe in Anspruch nimmt.

Blocked
Ein Prozess weist den Zustand *Blocked* auf, wenn dieser von der Ausführung durch ein bestimmtes Ereignis abgehalten wird, beispielsweise wartet der blockierte Prozess auf das Beenden eines anderen Prozesses.

Sleeping
Falls ein Prozess für eine gewisse Zeit keine Prozessorzeit benötigt, kann er sich selbst über die System-API in den Zustand *Sleeping* versetzen.

6.2 Scheduler

Der *Scheduler* ist eine Kernelkomponente welcher die Prozesse aus einer abstrakten Sicht, respektive lediglich die Prozesszustände und deren Kontext, verwaltet. Die Schnittstelle des *Scheduler* ist in Listing 9 angeführt.

```
1  #pragma pack(push,1)
2  typedef struct {
3    address_t* cpsr;
4    address_t* lr;
5    address_t* sp;
6    register_t registers[REGISTER_COUNT];
7    address_t* pc;
8  } context_t;
9  #pragma pack(pop)
10
11 struct process_t_struct {
12   processId_t id;
13   processFunc func;
14   processState_t state;
15
16   context_t* context;
17   address_t* pageTableL1;
18
19   long wakeupTime;
20
21   [...]
22 };
23
24 /*
25  * Scheduler functions
26  */
27 extern int SchedulerInit(device_t stdoutDevice);
28 extern int SchedulerStart(device_t timerDevice);
29 extern process_t* SchedulerStartProcess(processFunc func);
```

```
30 extern int SchedulerRunNextProcess(context_t* context);
31 extern int SchedulerKillProcess(processId_t id);
32 extern process_t* SchedulerGetRunningProcess(void);
33 extern void SchedulerBlockProcess(processId_t process);
34 extern void SchedulerUnblockProcess(processId_t process);
35 extern void SchedulerSleepProcess(processId_t process, unsigned int millis);
36 extern void SchedulerDisableScheduling(void);
37 extern void SchedulerEnableScheduling(void);
```

Listing 9: Implementierung der DriverManager Schnittstelle für den LED Treiber

Die *Scheduler*-Funktion `SchedulerRunNextProcess(..)` kann zu jedem beliebigen Zeitpunkt aufgerufen werden, an dem der aktuelle `context_t` vorhanden ist, vorwiegend wird diese Funktion aber vom Timer-*Interrupt* aufgerufen, welcher die Zeitscheibe von $10ms$ realisiert.
Die Zeitscheibe von $10ms$ wurde auf Basis verschiedener Experimente gewählt und hat sich, aufgrund dessen dass keine Priorisierung der Prozesse erfolgt, als annehmbar herausgestellt. Die durchgeführten Experimente in Kapitel 12 haben aber gezeigt, dass möglicherweise eine längere Zeitscheibe zu einer besseren Gesamtperformanz geführt hätten.

6.3 ProcessManager

Der *ProcessManager* ist eine dem *Scheduler* übergeordnete Komponente, welche eine weitere Verwaltung der Prozesse realisiert. Der *ProcessManager* fügt den abstrakten Eigenschaften der Prozesse des *Schedulers* zusätzlich noch Meta-Informationen, wie Prozessname, Startzeit usw. hinzu. Der *ProcessManager* stellt dabei die Schnittstelle für andere Kernelkomponenten zur Verfügung um Prozesse zu starten und zu beenden.

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 Meta-Informationen vom *ProcessManager* nicht weitergegeben werden. Der Scheduler speichert sich diesen neuen Prozess in seiner Prozesstabelle ab. Der *MemoryManager* allokiert nun den vom Prozess benötigtem Speicherplatz. Der erzeugte Prozess wird an den *ProcessManager* zurückgegeben. Dem Client wird nun der Erfolg der Prozesserstellung mitgeteilt, respektive wird diesem die erzeute Prozess-ID zurück geliefert.

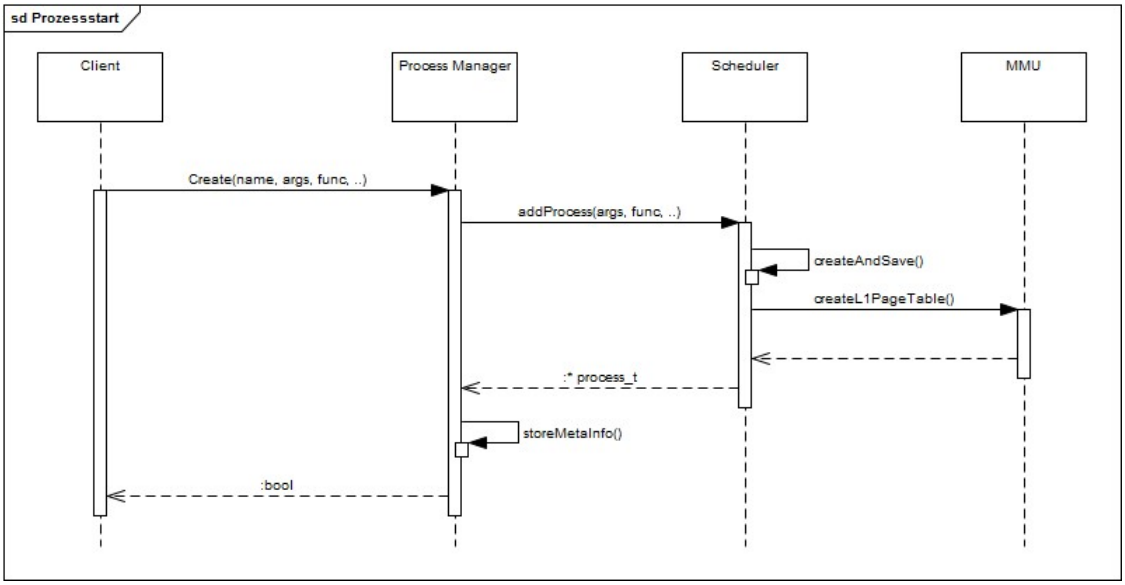

Abbildung 5: Sequenzdiagramm der Prozessverwaltung

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 Seitentabellensystem durchaus aus. Tabelle 4 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:	*small page* (4 kB), *large page* (64 kB)
Unterstützte Sectiongrößen:	*section* (1 MB), *supersection* (16 MB)

Tabelle 4: Eigenschaften der virtuellen Speicherverwaltung der ARMv7-Architektur

Generiert der *ARM*-Prozessor einen Speicherzugriff, wird von der MMU ein Suchlauf durchgeführt. Dieser Suchlauf wird *Translation Table Lookup* genannt. Dabei wird zuerst im Translation Lookaside Buffer (TLB) geprüft, 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 6. Aus einem der zwei Seitentabellenregister wird die Basisadresse der darin zuvor abgelegten Level-1-Seitentabelle geholt. Das Format des Page Table Entry (PTE) bestimmt dann, um welchen Typ von Verweis es sich handelt. Seitentabellen und ihre Einträge werden im nachfolgenden Abschnitt 7.3 genauer beschrieben.

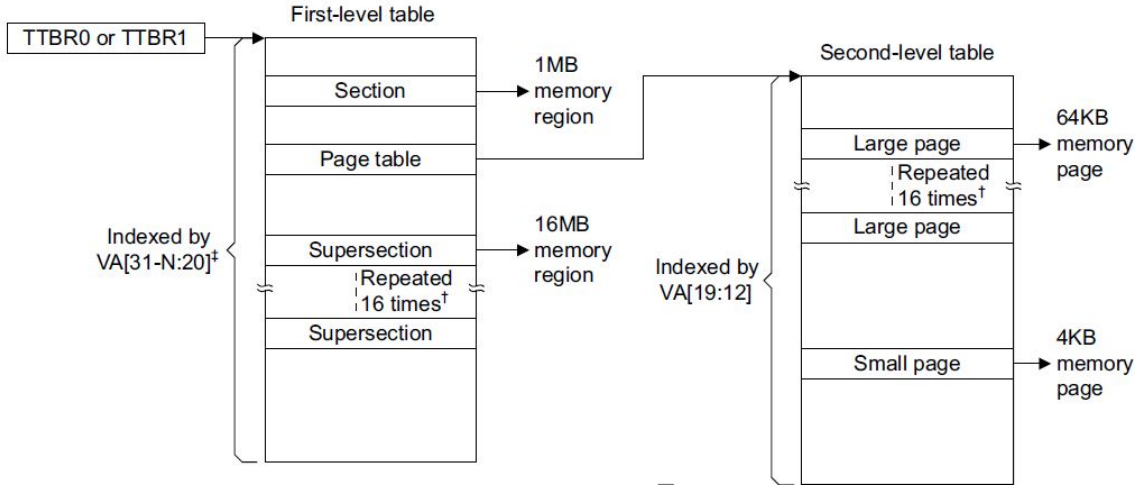

Abbildung 6: Zweistufiges Seitentabellensystem [1, S. B3-1325]

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 7 zeigt die Umwandlung einer virtuellen Adresse in die physikalische Adresse einer $1MB$ Section ohne Verwendung einer Level-2-Seitentabelle, Abbildung 8 diejenige einer virtuellen Adresse in ein $4kB$ *Page Frame* unter Verwendung einer Level-2-Seitentabelle. Die Umwandlung wird vollständig durch die Prozessor-Hardware durchgeführt.

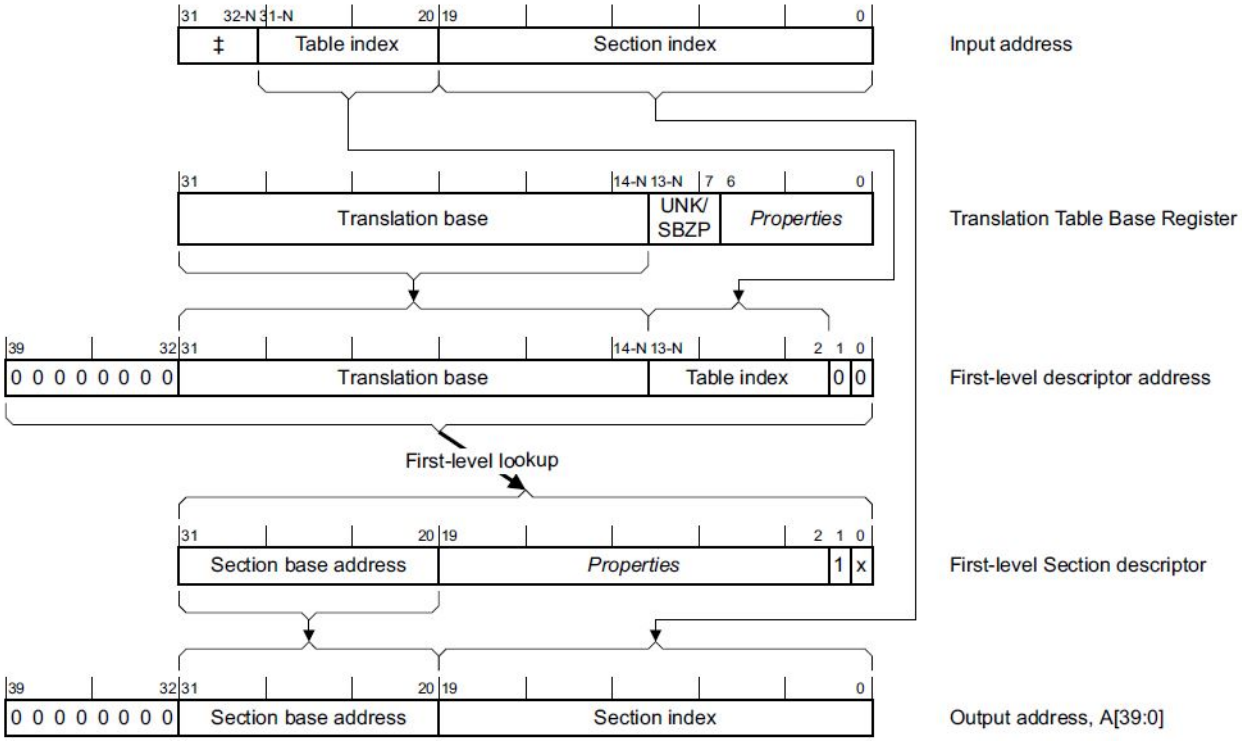

Abbildung 7: 1 MB Section Translation durch die ARM CPU [1, S. B3-1335]

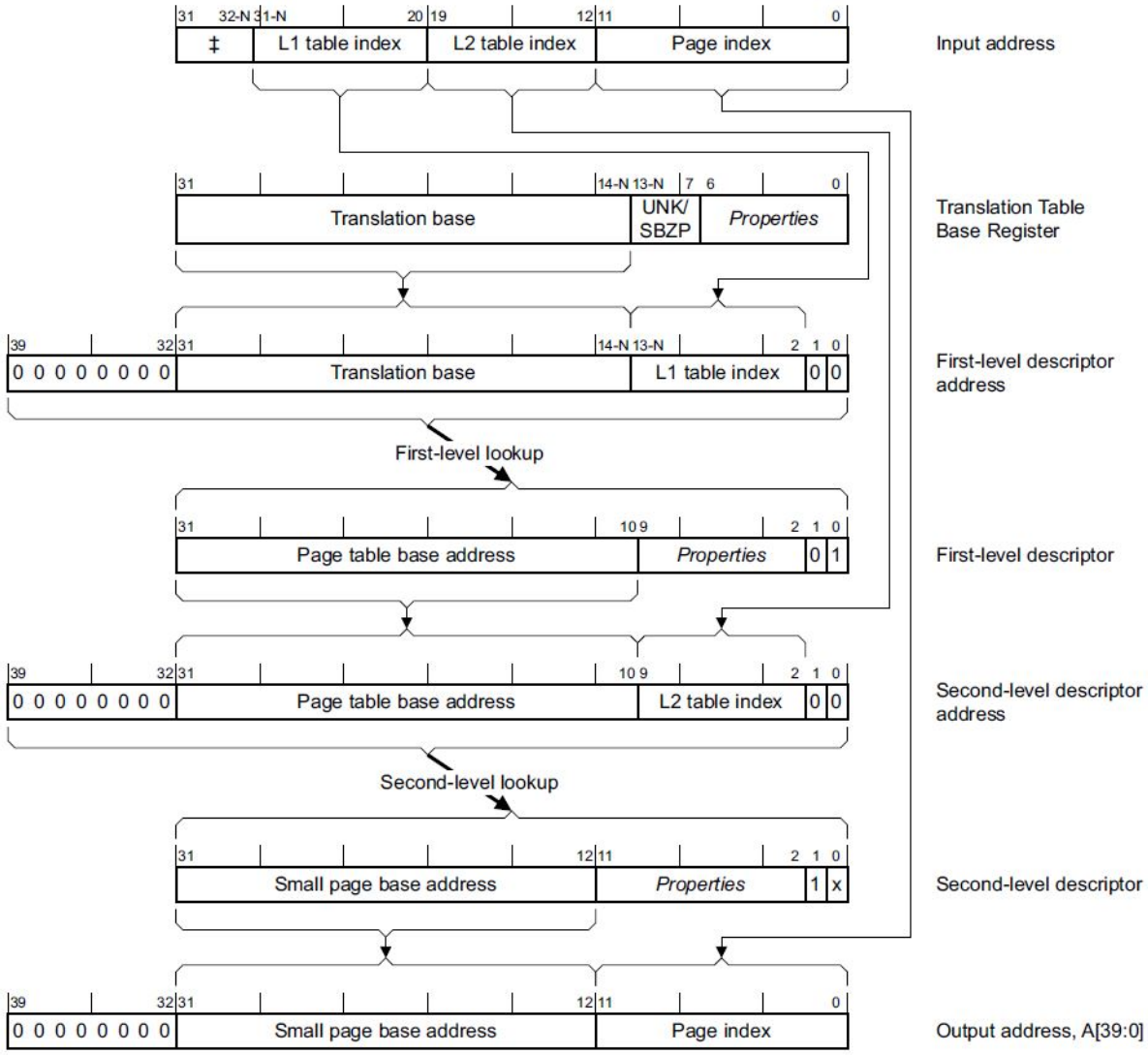

Abbildung 8: Small Page Translation durch die ARM CPU [1, S. B3-1337]

7.3 Seitentabellen und Seitentabelleneinträge

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 9 und 10 zu sehen. Diese Register übernehmen im Betriebssystem die folgende Funktion:

- TTBR0: Wird für prozessspezifische Adressen verwendet. Jeder Prozess enthält bei seiner Initialisierung eine eigene Level-1-Seitentabelle. Bei einem Kontextwechsel erhält das *TTBR0* eine Referenz auf Level-1-Seitentabelle des neuen Kontexts/Prozesses.

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

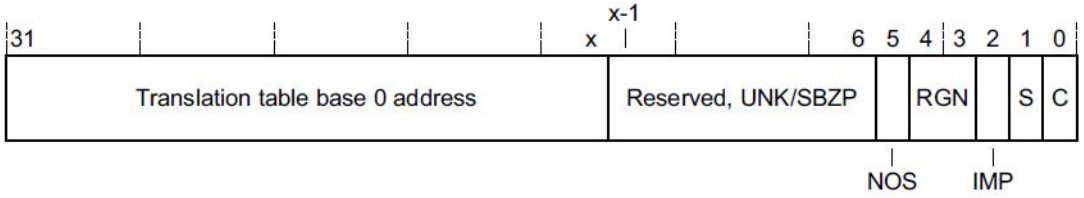

Abbildung 9: TTBR0 Format [1, S. B4-1726]

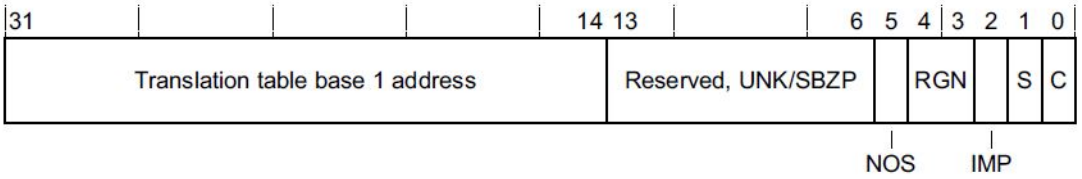

Abbildung 10: TTBR1 Format [1, S. B4-1730]

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 11 und 12 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 Level-1-Seitentabellen von Prozessen (siehe Abschnitt 7.4)

Für die Erstellung von *First-Level*-Deskriptoren wurde eine Struktur erstellt, welche in Listing 10 aufgeführt ist. Diese Struktur und jene des *Second-Level*-Deskriptors wird bei den nachfolgenden Erläuterungen zur MMU benötigt.

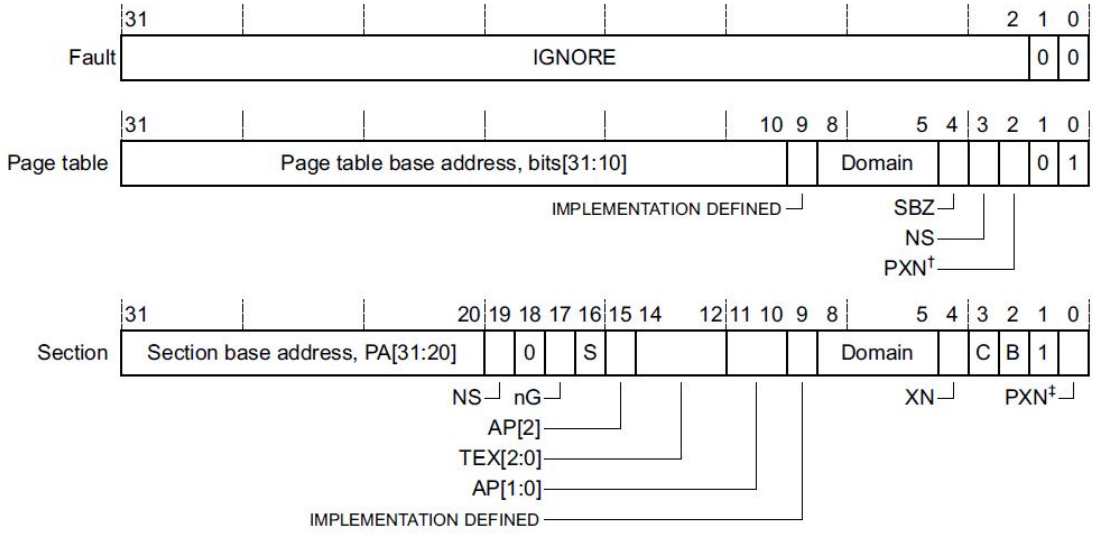

Abbildung 11: First-Level Deskriptorformate [1, S. B3-1326]

```
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 10: 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 Level-2-Seitentabelle eingetragen werden, *Large Pages* hingegen 16 mal
- Level-1- und Level-2-Seitentabellen, die $16kB$ bzw. $1kB$ 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 11 dargestellt.

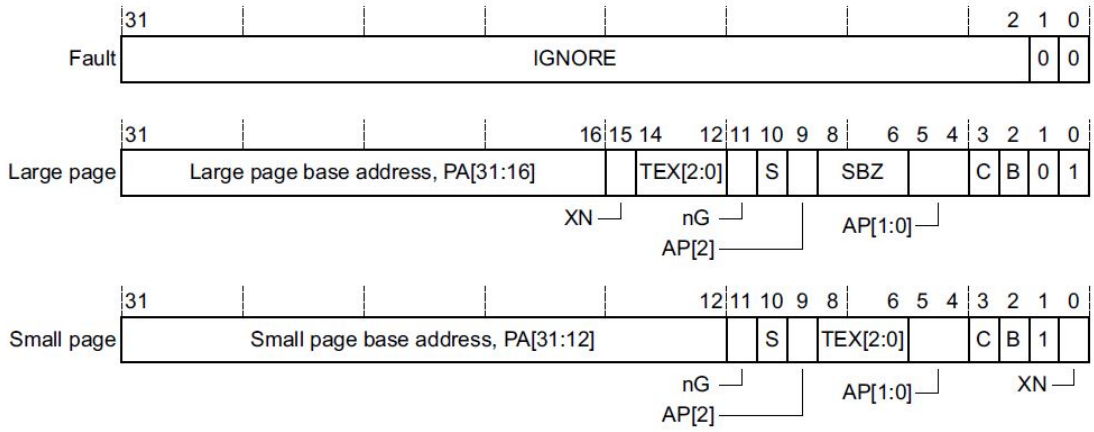

Abbildung 12: Second-Level Deskriptorformate [1, S. B3-1327]

```
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 11: Struktur für second-level Deskriptoren

7.4 Aufteilung des virtuellen Speichers und Mapping

Die Speicherverwaltung des Betriebssystems kann Abbildung 13 entnommen werden. Die rechte Seite stellt dabei 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 Prozess- 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 Translation Table Base Control Register (TTBCR) 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 Komponen-

ten der Kernelregion, die sich bei Prozesswechseln nicht ändern, beginnen bei Adressen ab `0x40000000`. Damit ergibt sich eine Aufteilung des virtuellen Speichers, wie sie in Abbildung 13 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.

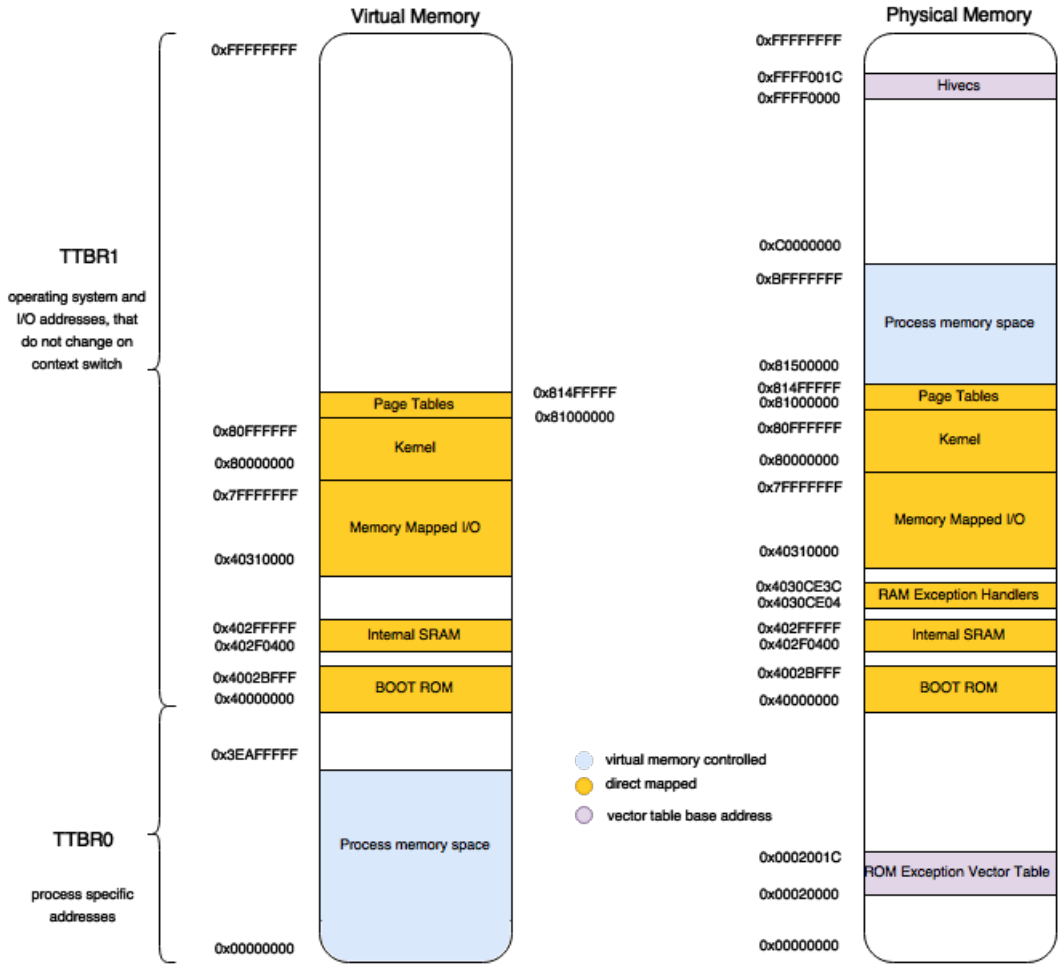

Abbildung 13: Memory Map des Betriebssystems

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

Tabelle 5: Eigenschaften der virtuellen Speicherverwaltung des OS

7.4.1 Speicherregionen

Das nachfolgende Listing 12 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 13 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 12: Struktur für die Verwaltung von Speicherregionen

Zusammengefasst dargestellt sind in Tabelle 6 alle Speicherregionen des Betriebssystems. Ein direktes Mapping bedeutet dabei, dass die virtuelle Adresse der physikalischen entspricht.

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

Tabelle 6: Angelegte Speicherregionen

7.4.2 Master Page Table

Um das Mapping der MPT verstehen zu können, wird nochmals auf den Adresstranslationsablauf in Abbildung 7 verwiesen. Alle direkt gemappten *Regions* aus Tabelle 6 werden in die L1 MPT als $1MB$ *Sections* gemapped.

Die Adresse eines Eintrags in der *Page Table* setzt sich zusammen aus der Basisadresse der entsprechenden *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 geshiftet werden, um das *Alignment* von 4 Byte einzuhalten. Schließlich wird der geshiftete 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 13 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 13: Funktion für direktes Mapping in die master page table

7.5 Allokierung der Page Frames

Für die Verwaltung der *Page Frames* wurde eine Bitsmap verwendet. Abbildung 14 zeigt das Prinzip. Die Bitsmap 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 Bitsmap definiert den Verwendungsstatus des $(n*8+i)$–ten *Page Frame*.

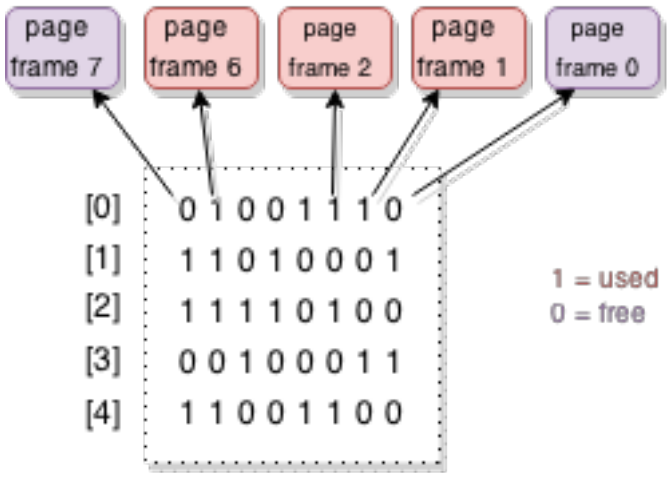

Abbildung 14: Beispiel einer Bitsmap zur Verwaltung der Page Frames

7.5.1 Allokation von Page Frames bei Data Abort Exception

Der Vorgang der Einlagerung von *Page Frames* durch den *Data Abort Handler* ist in Abbildung 15 dargestellt. Zuerst werden aus dem Data Fault Status 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 Zustände außer den Zuständen 5 und 7 der jeweilige Prozess beendet. Zustand betrifft die Erstellung einer *L2 Page Table* und die Einlagerung eines *Page Frame*. Zustand 7 betrifft die bloße Einlagerung eines *Page Frame*. Die Einlagerung der *Page Frames* erfolgt dabei auf sehr ähnliche Weise wie in Listing 13 vorgenommen.

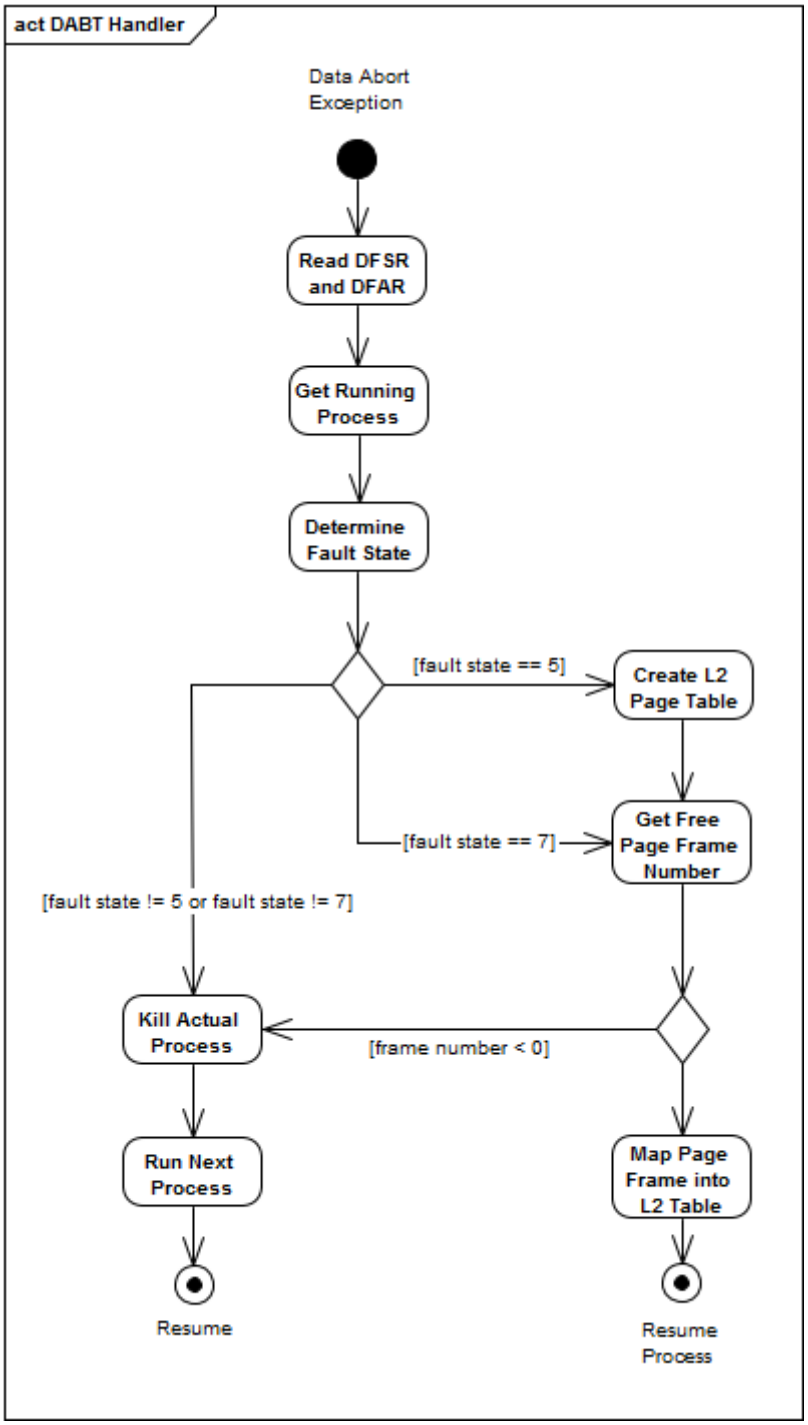

Abbildung 15: Einlagerung von page frames duch den DABT-Handler

7.6 Aktivieren der MMU

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

Listing 14 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 accidently reserving of pages can occur
8     MemoryManagerReserveAllDirectMappedRegions();
9
10    // master page table for kernel region must be created statically and before MMU is
          ↪ enabled
11    mmuCreateMasterPageTable(KERNEL_START_ADDRESS, KERNEL_END_ADDRESS);
12    mmuSetKernelMasterPageTable(kernelMasterPageTable);
13    mmuSetProcessPageTable(kernelMasterPageTable);
14
15    // MMU Settings
16    mmuSetTranslationTableSelectionBoundary(BOUNDARY_AT_QUARTER_OF_MEMORY);
17    mmuSetDomainToFullAccess();
18
19    MMUEnable();
20
21    return MMU_OK;
22 }
```

Listing 14: Aktivierung der MMU

7.7 Interaktion der MMU mit Prozessen

Die Schnittstelle des *MemoryManagers*, welche die MMU-Funktionalitäten implementiert, zeigt Listing 15.

```
1  // MMU Functions
2  extern int MMUInit(void);
3  extern int MMUSwitchToProcess(process_t* process);
4  extern int MMUInitProcess(process_t* process);
5  extern void MMUHandleDataAbortException(context_t* context);
6  extern int MMUFreeAllPageFramesOfProcess(process_t* process);
7
8  // MemoryManager Functions
9  extern int MemoryManagerInit();
10 extern pageAddressPointer_t MemoryManagerGetFreePagesInProcessRegion(unsigned int
       ↪ pagesToReserve);
11 extern memoryRegionPointer_t MemoryManagerGetRegion(unsigned int memoryRegionNumber);
12 extern int MemoryManagerReserveAllDirectMappedRegions(void);
```

Listing 15: Softwareschnittstelle der MMU und des MemoryManagers

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 Bitsmap wieder frei.

Die Funktionen des *MemoryManagers* selbst werden ausschließlich von oben angeführten Funktionen verwendet.

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 händelt die Interprozesskommunikation zwischen Prozessen. Listening 16 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 16: 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 17 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 17: Aufbau eines Systemcall Datenpakets

9.2 Vorgehensweise bei einem Systemcall

Das Vorgehen bei einem Systemcall ist in Abbildung 16 ersichtlich.

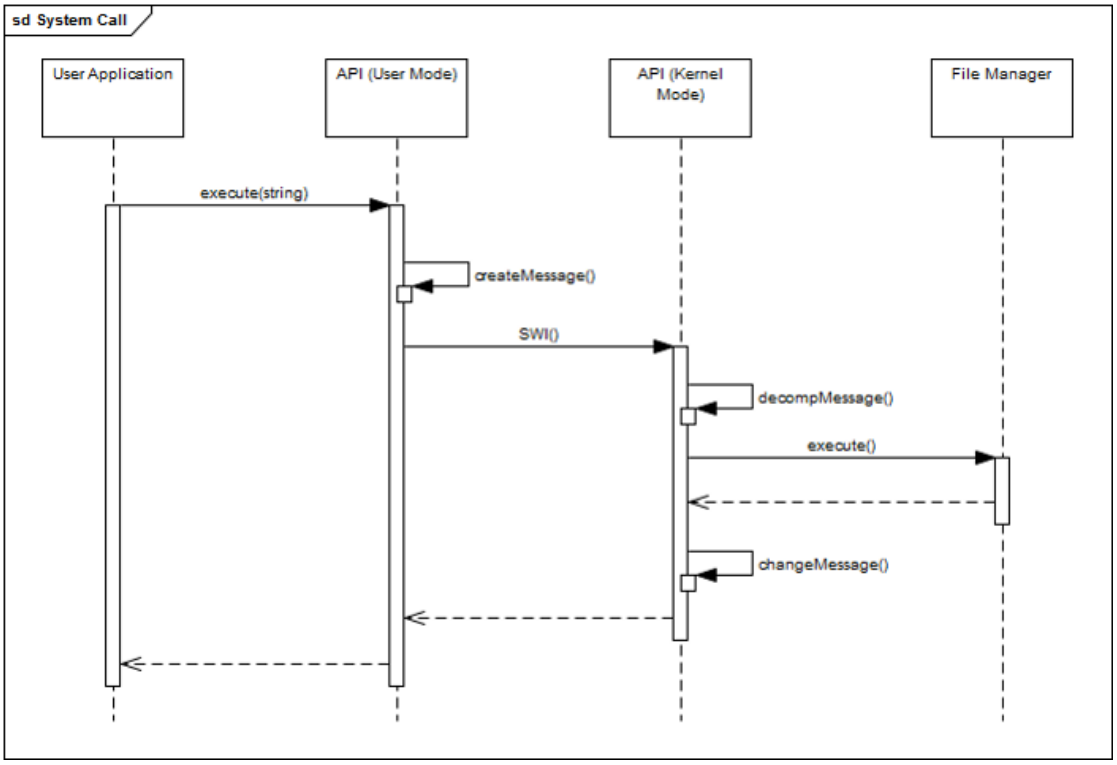

Abbildung 16: Sequenzdiagramm eines Systemcalls

10 Sicherheitsaspekte

Hinsichtlich der Sicherheit wurden an das Betriebssystems 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 Betrachung des Speichermodells in Abbildung 13. Wie die Abbildung zeigt, beginnt im virtuellen Speicher der Adressbereich für Prozesse ab der Adresse `0x00000000`. Gleichzeitig befindet sich die Startadresse für die *ROM Exception Vector Table* an der Adresse `0x00020000`. Durch diese Gegebenheiten bestehen zwei grundsätzliche Sicherheitsrisiken:

1. Nulladressenproblem: Adresse `0x00000000` 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 `0xFFFF0000`. Damit liegt die Basisadresse über der festgelegten Adressbereichsgrenze eindeutig im Kernelbereich, siehe dazu den physikalischen Bereich in Abbilung 13. 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 Startaddressen 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 deaktivert.

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 Spezifikation erläutert und anschließend zu Vergleichszwecken verwendet. Abbildung 17 dient zur Veranschaulichung des DMX-512 Protokolls.

TODO!!!

Abbildung 17: DMX Protokoll

Tabelle 7 beschreibt die einzeln nummerierten Markierungen aus Abbildung 17.

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

Tabelle 7: Eigenschaften des DMX-512-Protokolls

Die Übertragungsgeschwindigkeit ist bei allen Protokollarten identisch und beträgt 250 kBaud, d.h. jedes Bit hat eine Dauer von 4 μ s. Das DMX Protokoll besitzt 512 verschiedene Kanäle, wobei jeder Kanal mithilfe eines Datenbytes gesteuert wird. In Abbildung 17 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

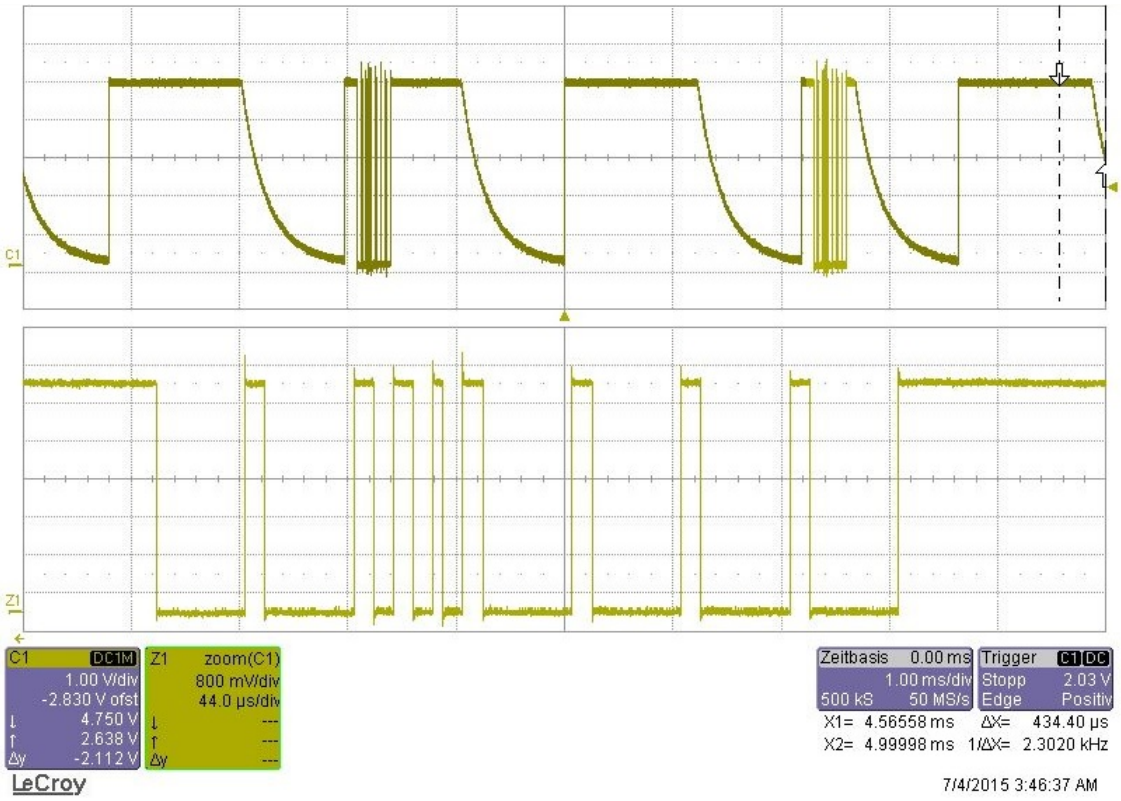

Abbildung 18: DMX Protokoll: Problem fallende Flanke

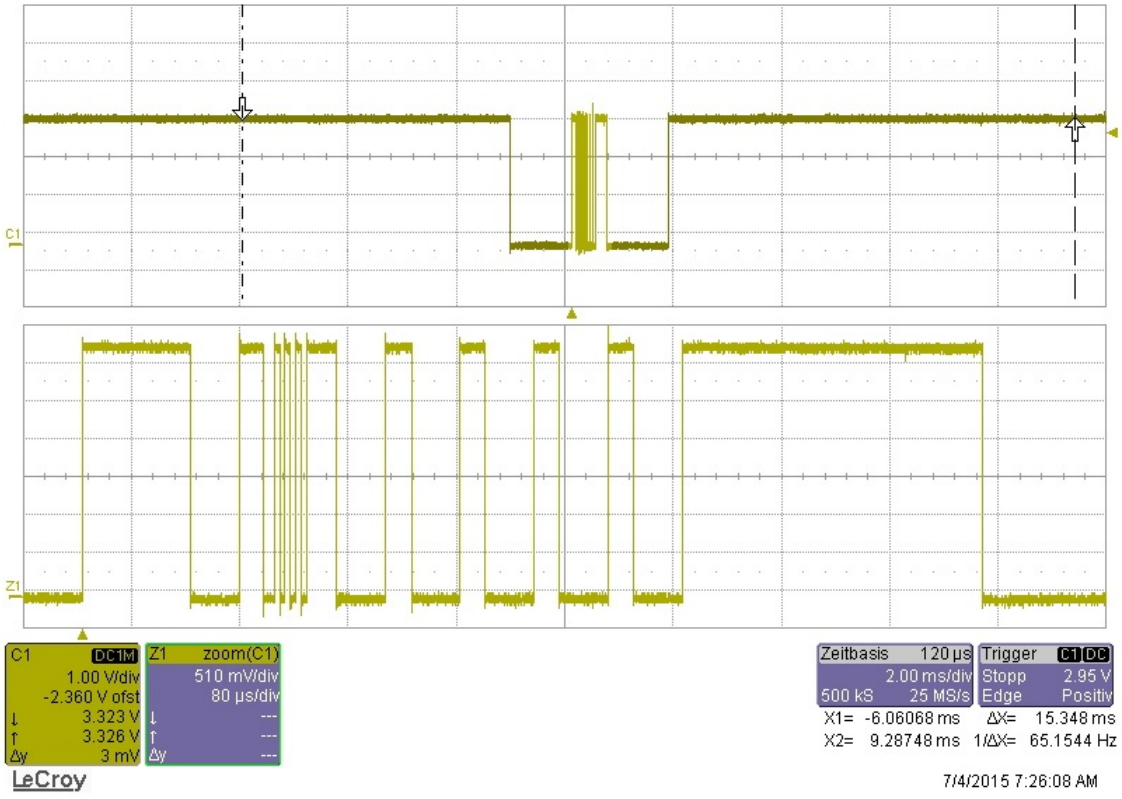

Abbildung 19: DMX Protokoll: Problem Offset Byte

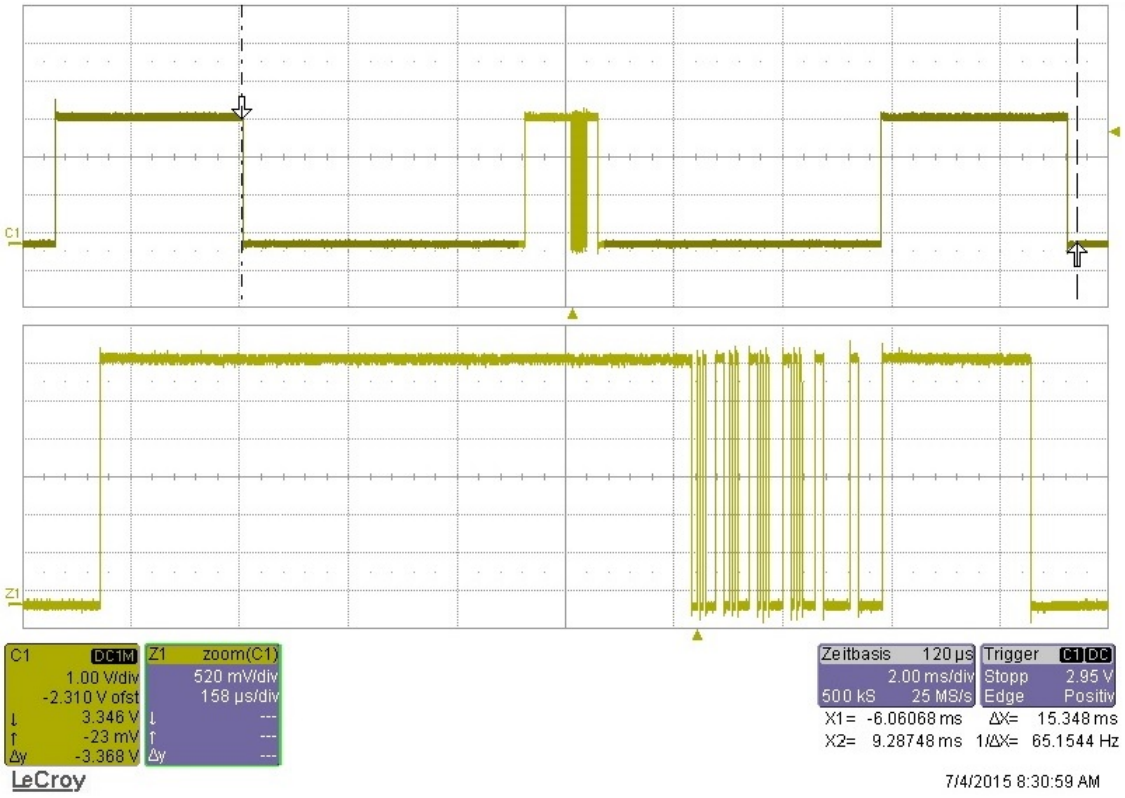

Abbildung 20: Funktionierendes DMX Protokoll

12 Performanzuntersuchungen

In diesem Kapitel werden Performanzeaspekte 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 8 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	$375\mu s$

Tabelle 8: 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 Gesamtperformanz 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 ausgereift.

13.2.1 Punkte mit Verbesserungspotential

Einige Punkte des Betriebssystem konnten nicht vollständig abgedeckt werden. Diese Punkte mit Verbesserungspotential werden in Tabelle 9 kurz beschrieben.

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

Tabelle 9: 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 10 zeigt diese Punkte auf.

Fehlender Punkt	Beschreibung
ResourceManager	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 *ResourceManager* würde hinsichtlich dieser Problematik die von einem Prozess verwendeten Ressourcen verwalten und ggf. den Zugriff durch andere Prozesse sperren.

Tabelle 10: 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.