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

30. 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 im 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 System API und die Funktionsweise der Systemcalls 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 disskutiert, welche die Steuerung von Scheinwerfern über das DMX512-Protokoll vornimmt.

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

DALI Digital Addressable Lighting Interface (Bus Protokoll)

DFAR Data Fault Address Register

DFSR Data Fault Status Register

DMX Digital Multiplex (Bus Protokoll)

HAL Hardware Abstraction Layer

KNX Konnex-Bus (Bus Protokoll)

MMU Memory Management Unit

MPT Master Page Table

OS Operating System

PTE Page Table Entry

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.

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 **??** werden.
Die Performanz des Betriebssystems wurde 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 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. Eine Abarbeitung dieser Tickets erfolgte 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`

3 Architektur

Die Architektur beschreibt den allgemeinen Aufbau des Betriebssystems. Eine genaue Beschreibung zu den einzelnen Teilen sind in weiteren Kapiteln in diesem Dokument enthalten.

3.1 Art des Kernels

Das Betriebssystem ist ein Monolithischer Kernel. Darunter versteht man einen Kernel, welcher neben Funktionen für Speicherverwaltung, Prozessverwaltung und Kommunikation zwischen Prozessen auch Treiber sowie weitere Komponenten (z.B. Dateisystem) enthält. Durch das Beinhalten dieser Komponenten hat der Monolithische Kernel einen Geschwindigkeitsvorteil gegenüber einem Mikrokernel (Vgl. XXX). Ein weiterer Grund für einen Monolithischen Kernel ist das entfallen der aufwändigen Kommunikationen zwischen den verschiedenen Komponenten des Betriebssystems.

3.2 Ansatz für die Abstraktionen im Betriebssystem

Um eine möglichst gute Abstraktionen im Betriebssystem zu gewährleisten werden Manager für die einzelnen Komponenten verwendet. Eine Übersicht der einzelnen Manager sowie eine bzw. mehrere zugehörige Funktionen, zum besseren Verständnis, ist in Tabelle 3 gegeben. Eine kurze Beschreibung zu den einzelnen Managern ist unter 3.3 gegeben.

Managername	**Beispiel Funktion(en)**
DeviceManager	InitDevice, OpenDevice, ReadDevice
DriverManager	GetDriver
FileManager	ListDirectoryContent, OpenFile, OpenExecutable
MemManager	GetFreePagesInProcessRegion, GetRegion
ProcessManager	StartProcess, KillProcess, ListProcesses
IpcManager	RegisterNamespace, SendMessage, HasMessage

Tabelle 3: Übersicht der Manager mit zugehöriger Funktion

3.3 Allgemeiner Aufbau der Architektur

In Abbildung 1 ist der allgemeine Aufbau mit den wesentlichen Teilen der Architektur ersichtlich.

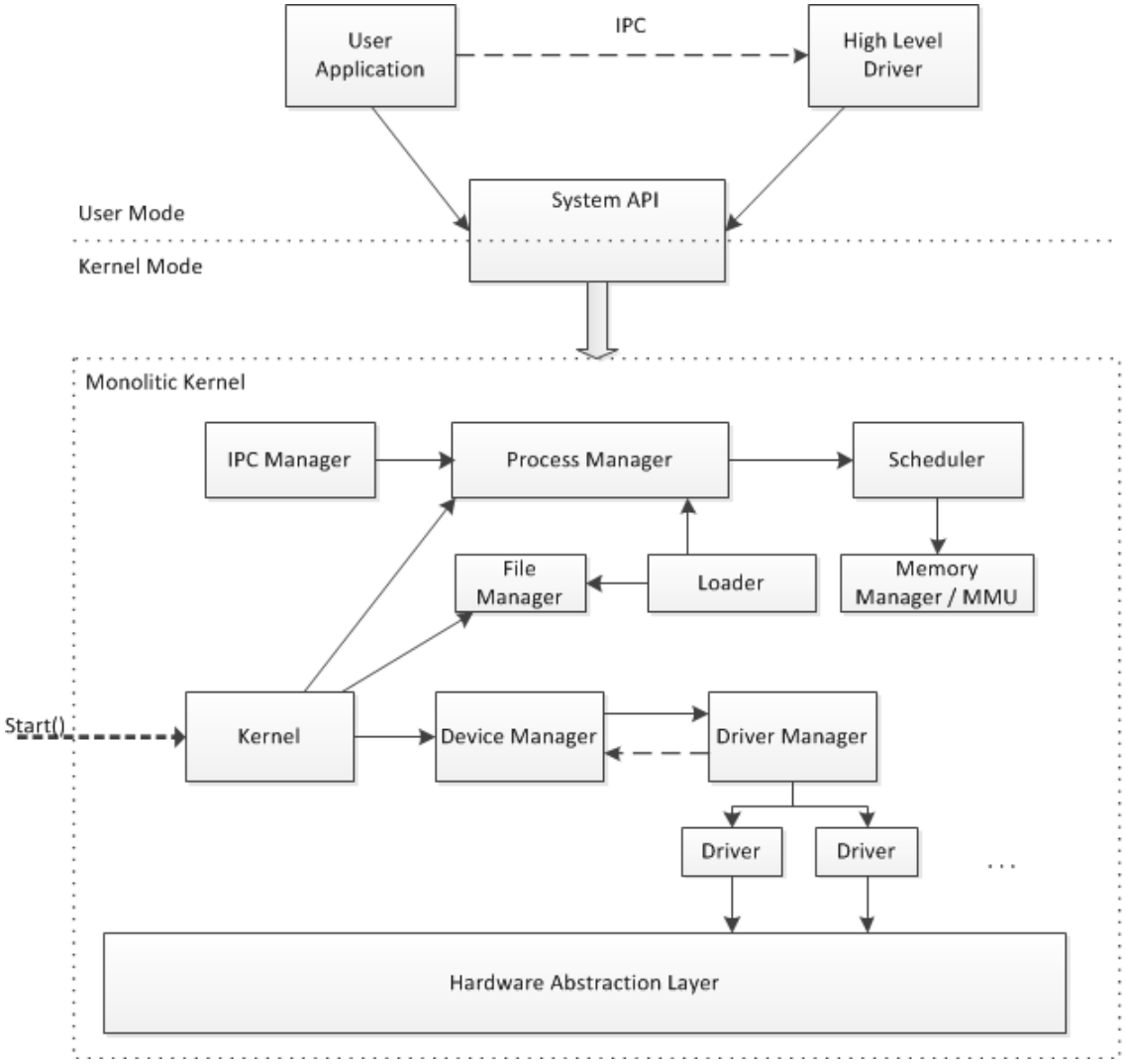

Abbildung 1: Allgemeiner Aufbau der Architektur

Im folgenden wird eine kurze Erläuterung zu den einzelnen Komponenten der Abbildung 1 gegeben. Für eine genauere Beschreibung wird auf die einzelnen Kapitel verwiesen.

Hardware Abstraction Layer (HAL) (Vgl. Kapitel 4)
Der Hardware Abstraction Layer wird, wie der Name bereits beschreibt, zur Abstraktion der Hardware vom eigentlichen Betriebssystem verwendet.

Driver (Vgl. Kapitel 5)
Ein Treiber ist eine abstrakte Schnittstelle zu der Hardware, sodass kein direkter Zugriff auf die HAL benötigt wird.

Driver Manager (Vgl. Kapitel 5.3)

Der Driver Manager dient zum ansprechen der Treiber, welche vom Betriebssystem zur Verfügung gestellt werden. Sollte ein Treiber benötigt werden, muss dieser nicht erzeugt werden sondern kann über Driver Manager geholt werden.

Device Manager

Der Device Manager dient wie bereits der Driver Manager zur Abstraktion der Treiber. D.h. eine Anwendung verwendet Geräte, welche vom Device Manager zur Verfügung gestellt werden. Ein Beispiel für Devices sind LEDs. Beim Ansprechen einer LED wird ein Treiber benötigt, ohne der Abstraktion auf Geräte müssten bei der Verwendung mehrerer LEDs auch mehrere LED Treiber geschrieben werden oder ein großer Treiber. Der Nachteil eines einzelnen Treibers ist, dass das Ansprechen einzelner LEDs viel Aufwand benötigt.

Kernel

Der Kernel ist der Kern des Betriebssystems und enthält das Starten aller Prozesse und Managern. Dazu zählen: Konsole, Device Manager, Driver Manager, Process Manager, File Manager, IPC Manager, etc.

Process Manager (Vgl. Kapitel 6)

Der Process Manager ist zuständig für das Starten und Stoppen (Killen) von Prozessen. Es besteht eine starke Kopplung zum Scheduler.

Scheduler (Vgl. Kapitel 6.2)

Der Scheduler wechselt die Prozesse in fix definierten Zeitscheiben (10ms). Auch ist das Händeln der verschiedenen Zustände eines Prozesses Aufgabe vom Scheduler. Gültige Zustände sind: Ready, Running, Blocked, Sleeping und Free.

Memory Manager/MMU (Vgl. Kapitel 7)

Über den Memory Manager können freie Pages in der Prozessregion allokiert werden sowie die eine bestehende Region zurückgeliefert werden.

File Manager

Der File Manager dient zum Verarbeiten von dateiabhängigen Operationen. Auflisten der einzelnen Inhalte in einem Verzeichnis, Öffnen einer Datei, Setzen des aktuellen Verzeichnis, etc. sind die Hauptaufgaben dieses Managers.

Loader

Der Loader ist dafür zuständig ein existierendes Programm in die Prozessregion zu laden. D.h. der Loader ladet ein auszuführendes Programm in den Speicher, sodass dieses Programm als Prozess ausgeführt werden kann.

IPC Manager (Vgl. Kapitel 8)

Der IPC Manager ist für die Kommunikation zwischen verschiedenen User-Anwendungen zuständig.

System API (Vgl. Kapitel 9)

Die System API stellt eine Schnittstelle für den Anwendungsentwickler/ die Anwendungsentwicklerin zur Verfügung. Dadurch sind die Betriebssystem Funktionen von der Anwendung entkoppelt. Es werden von der Anwendung nur System API Funktionen aufgerufen und keine System Funktionen. Dies führt zu einer höheren Sicherheit des Systems sowie zu einem einfacheren Implementieren von Endanwendungen.

User Application (Vgl. Kapitel 11)

Bei der User Application handelt es sich um das Ansprechen eines Moving Heads mittels DMX Protokoll. Vergleichbare Projekte wären das Ansprechen von Komponenten die zur Kommunikation KNX oder DALI verwenden.

High Level Driver

Der High Level Driver ist ein Treiber, welcher dazu dient der eigentlichen BenutzerInnen Anwendung eine verbesserte Schnittstelle zur Verfügung zu stellen. Ansonsten müsste ein Entwickler/ eine Entwicklerin wissen, dass das DMX Protokoll durchgehend sendet, somit wäre in der eigentlichen Anwendung Logik implementiert, welche gar nicht hinein gehört bzw. davon abstrahiert gehört.

4 Hardware Abstraction Layer (HAL)

Der Hardware Abstraction Layer (HAL) dient zur Abstraktion von der eigentlichen Hardware. Dies wird dann benötigt, wenn das Betriebssystem portierbar sein sollte. Ein weiterer Vorteil ist, dass nicht mehr auf die Hardware direkt zugegriffen werden muss, d.h. das Mapping auf Hardwareadressen wird von der HAL abgenommen und es kann mittels abstrakter Komponenten bzw. Ids oder Pins gearbeitet werden.

4.1 Aufbau der HAL Schnittstelle

Die HAL Schnittstelle ist für jede Hardwarekomponente unterschiedlich, dies ist in Listening 1 und Listening 2 dargestellt. Das zuvor erwähnte abstrakte Ansprechen der Komponenten über die Pins ist ebenfalls in den Listenings ersichtlich.

```
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: Schnittstelle für die UART

4.2 Interrupts

Die Interrupts stellen einen grundlegenden Teil der 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.

The Host is responsible for prioritizing all service requests from the system peripherals and generating either nIRQ or nFIQ to the host. The type of the interrupt (nIRQ or nFIQ) and the priority of the interrupt inputs are programmable.
Einstellungen betreffend der Interrupts werden im ARM Interrupt Controller (**AINTC**) getroffen. 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 Priorität der Interrupts und deren Art, ob *IRQ* (normaler Interrupt) oder *FIQ* (fast interrupt), einstellbar.

Die in der HAL implementierten Interruptfunktionalitäten dienen als Grundlage für das Handling spezifischer IRQs und FIQs, beispielsweise von Timer oder UARTs. Das nachfolgende Listing 3 zeigt die zur Verfügung gestellte Funktionalität. Die wichtigsten Funktionen betreffen das Resetten des AINTC, das globale Enablen bzw. Disablen von Interrupts sowie das einzelne Enablen bzw. Disablen 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. 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 ist immer am Ende der jeweiligen Interrupthandler aufzurufen.

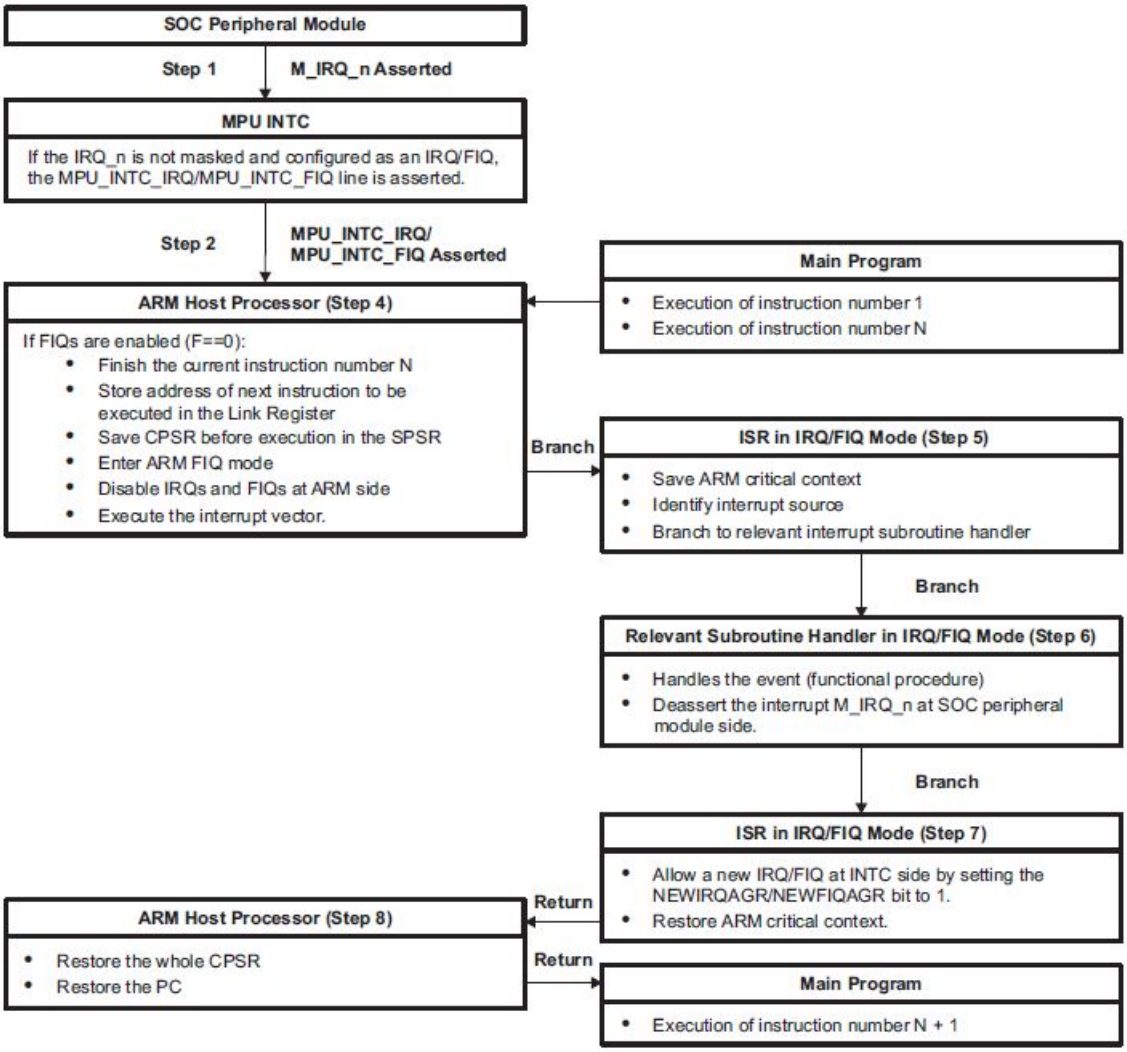

Abbildung 2: IRQ/FIQ Verarbeitung [2, S. 193]

Die Verarbeitung eines Interrupts im Betriebssystem ist in Abbildung 3 schematisch dargestellt. Sichern und Wiederherstellen des Kontextes, d.h. der Register R0-R13 und der Rücksprungadresse, wird durch den in Assembler geschriebenen *IRQ_Handler* vorgenommen. Dieser ruft nach dem Sichern des Kontextes den dem Interrupt zugewiesenen Handler auf. Danach wird der ursprüngliche Kontext wiederhergestellt und das Programm an der unterbrochenen Stelle fortgesetzt.

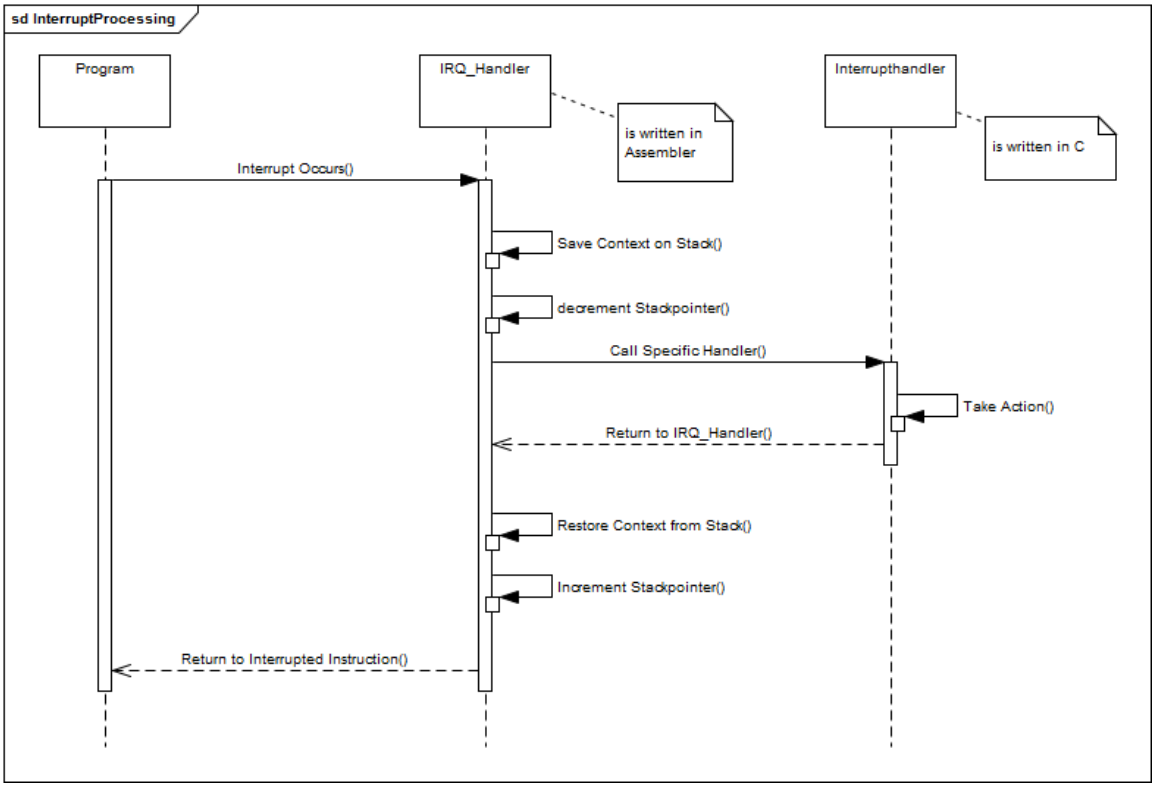

Abbildung 3: Interruptverarbeitung im Betriebssystem

5 Treiber

Die Treiber stellen eine abstrakte Schnittstelle auf den Hardware Abstraction Layer dar. Dadurch muss nicht mehr direkt auf die einzelnen Hardwarekomponenten zugegriffen werden. Ein wesentlicher Aspekt bei der Architektur der Treiber war Abstraktion. Dadurch wird gewährleistet, dass jeder Treiber über die selbe Schnittstelle angesprochen werden kann. Zudem ist die Verwaltung durch den Driver Manager erleichtert.

5.1 Allgemeiner Aufbau eines Treibers

In Listening 4 ist die allgemeine Schnittstelle für jeden Treiber ersichtlich.

```
typedef struct {
    int (*init)(uint16_t pin);
    int (*open)(uint16_t pin);
    int (*close)(uint16_t pin);
    int (*read)(uint16_t pin, char* buf, uint16_t length);
    int (*write)(uint16_t pin, char* buf, uint16_t length);
    int (*ioctl)(uint16_t pin, uint16_t cmd, uint8_t mode, char* buf, uint16_t length);
} driver_t;
```

Listing 4: Allgemeine Schnittstelle für einen Treiber

5.2 Beispiel Implementierung eines Treibers

Jeder implementierte Treiber muss diese vorweisen können, ein Beispiel (LED Treiber) dazu ist in Listening 5 zu sehen.

```
int LEDInit (uint16_t id)
{
  uint8_t ledCount = BOARD_LED_COUNT;
  if (id > ledCount - 1) return DRIVER_ERROR;
  // Set up the GPIO pin
  GPIOEnable(BOARD_LED(id));
  GPIOSetMux(BOARD_LED(id), MUX_MODE_LED);
  GPIOSetPinDirection(BOARD_LED(id), PIN_DIRECTION_OUT);
  return DRIVER_OK;
}

int LEDOpen (uint16_t id)
{
  uint8_t ledCount = BOARD_LED_COUNT;
  if (id > ledCount - 1) return DRIVER_ERROR;
  return DRIVER_OK;
}

int LEDClose (uint16_t id)
{
  // Turn off the led
  char buf[1] = { 0 };
  return LEDWrite(id, &buf[0], 1);
```

```
24 }
25
26 int LEDWrite (uint16_t id, char* buf, uint16_t len)
27 {
28   uint8_t ledCount = BOARD_LED_COUNT;
29   if (id > ledCount - 1) return DRIVER_ERROR;
30
31   if (len != 1) return DRIVER_ERROR;
32
33   switch(buf[0])
34   {
35     case '1':
36       GPIOSetPinValue(BOARD_LED(id), PIN_VALUE_HIGH);
37       break;
38     case '0':
39       GPIOSetPinValue(BOARD_LED(id), PIN_VALUE_LOW);
40       break;
41     default:
42       return DRIVER_ERROR;
43   }
44   return DRIVER_OK;
45 }
46
47 int LEDRead (uint16_t id, char* buf, uint16_t len)
48 {
49   return DRIVER_FUNCTION_NOT_SUPPORTED;
50 }
51
52 int LEDIoctl (uint16_t id, uint16_t cmd, uint8_t mode, char* buf, uint16_t len)
53 {
54   return DRIVER_FUNCTION_NOT_SUPPORTED;
55 }
```

Listing 5: Implementierung der allgemeinen Treiberschnittstelle (LED Beispiel)

5.3 DriverManager

Der DriverManager hat die Aufgabe die Treiber zu initialisieren sowie diese dann nach außen hin anzubieten. In Listening 6 ist die Schnittstelle des DriverManagers dargestellt. Eine Implementierung dieser Schnittstelle für das LED Beispiel ist in Listening 7 aufgezeigt. Für das Hinzufügen eines weiteren Treibers beim DriverManager, muss nur die *DriverManagerInit* Funktion angepasst werden. D.h. es muss ein zusätzlicher Treiber mit den seinen zugehörigen Funktionspointern in das *drivers* Array eingefügt werden.[1]

```
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);
```

[1] Die Verwendung von *malloc* ist hier nicht nötig und könnte durch eine nicht dynamische Allokierung ersetzt werden.

Listing 6: Allgemeine Schnittstelle des DriverManagers

```
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

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

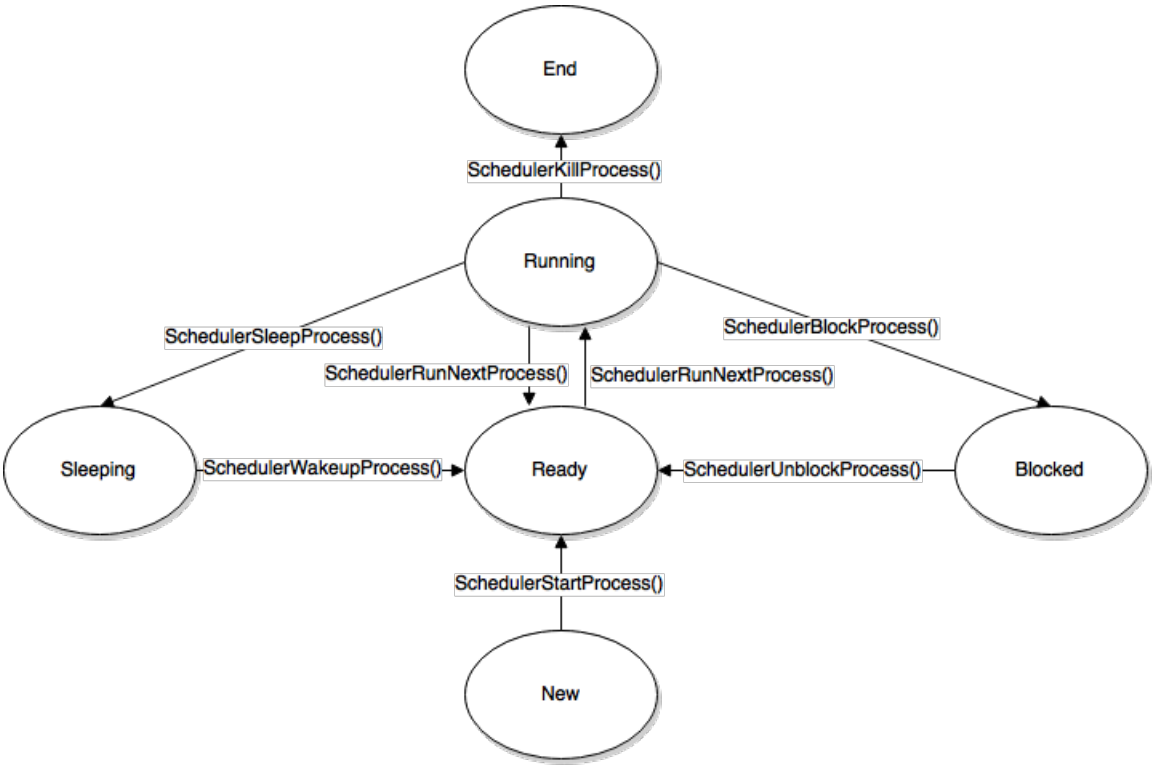

Abbildung 5: Prozesszustände und deren Transitionen

6.2 Scheduler Eigenschaften

Der Scheduler weißt 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 gleicher 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.

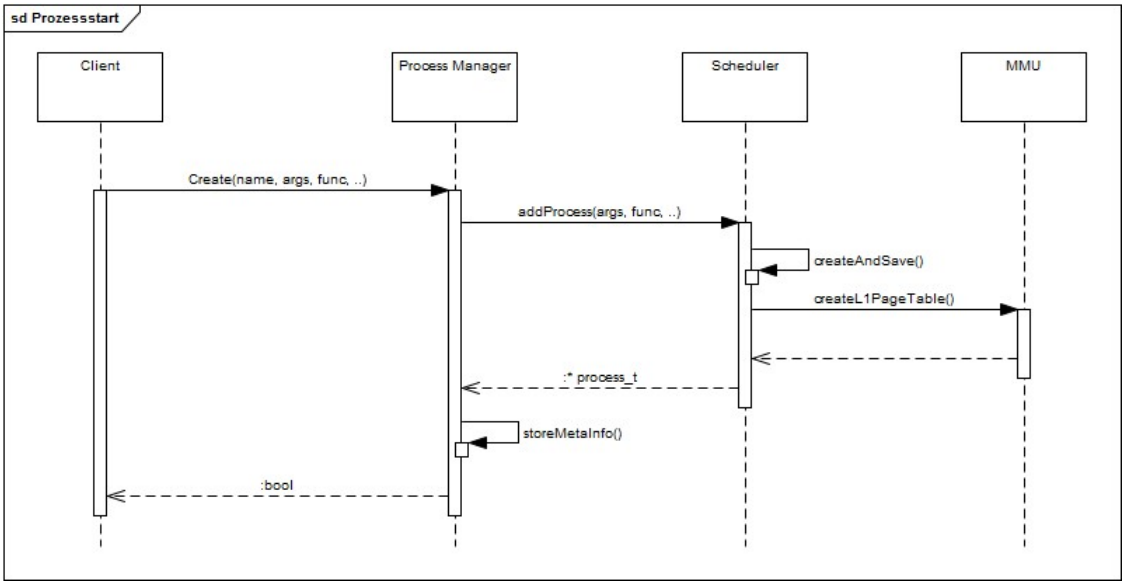

Abbildung 6: Sequenzdiagramm der Prozessverwaltung

7 Virtuelle Speicherverwaltung

Bei der virtuellen Speicherverwaltung erfolgt die Umwandlung von vom ARM Prozessor generierten, virtuellen Adressen in physikalische Adressen durch die Memory Management Unit (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 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:	*small page* (4 kB), *large page* (64 kB)
Unterstützte Sectiongrößen:	*section* (1 MB), *supersection* (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.

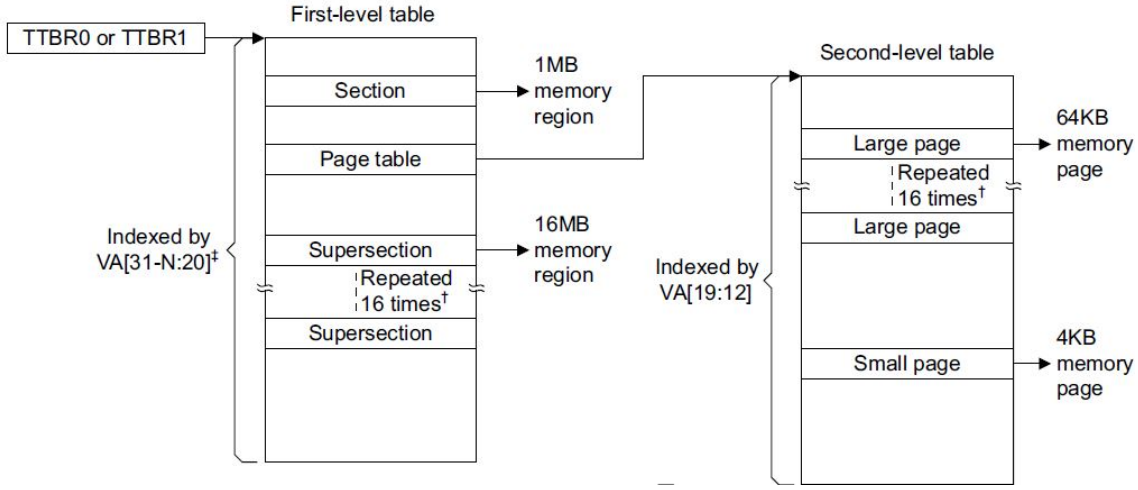

Abbildung 7: Zweistufiges Seitentabellensystem [1, S. B3-1325]

7.1.1 Einlagerungsvorgang und Data Abort Handler

DATA ABORT HANDLER BESCHREIBEN

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.

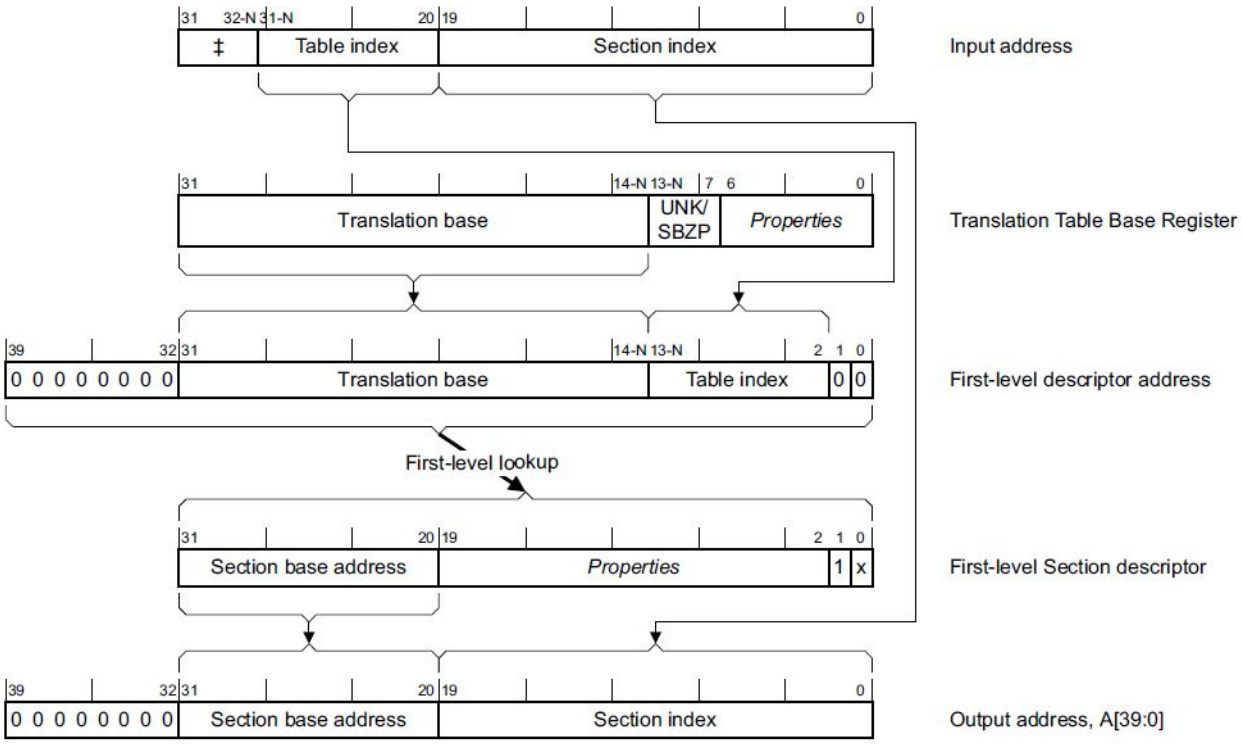

Abbildung 8: 1 MB Section Translation durch die ARM CPU [1, S. B3-1335]

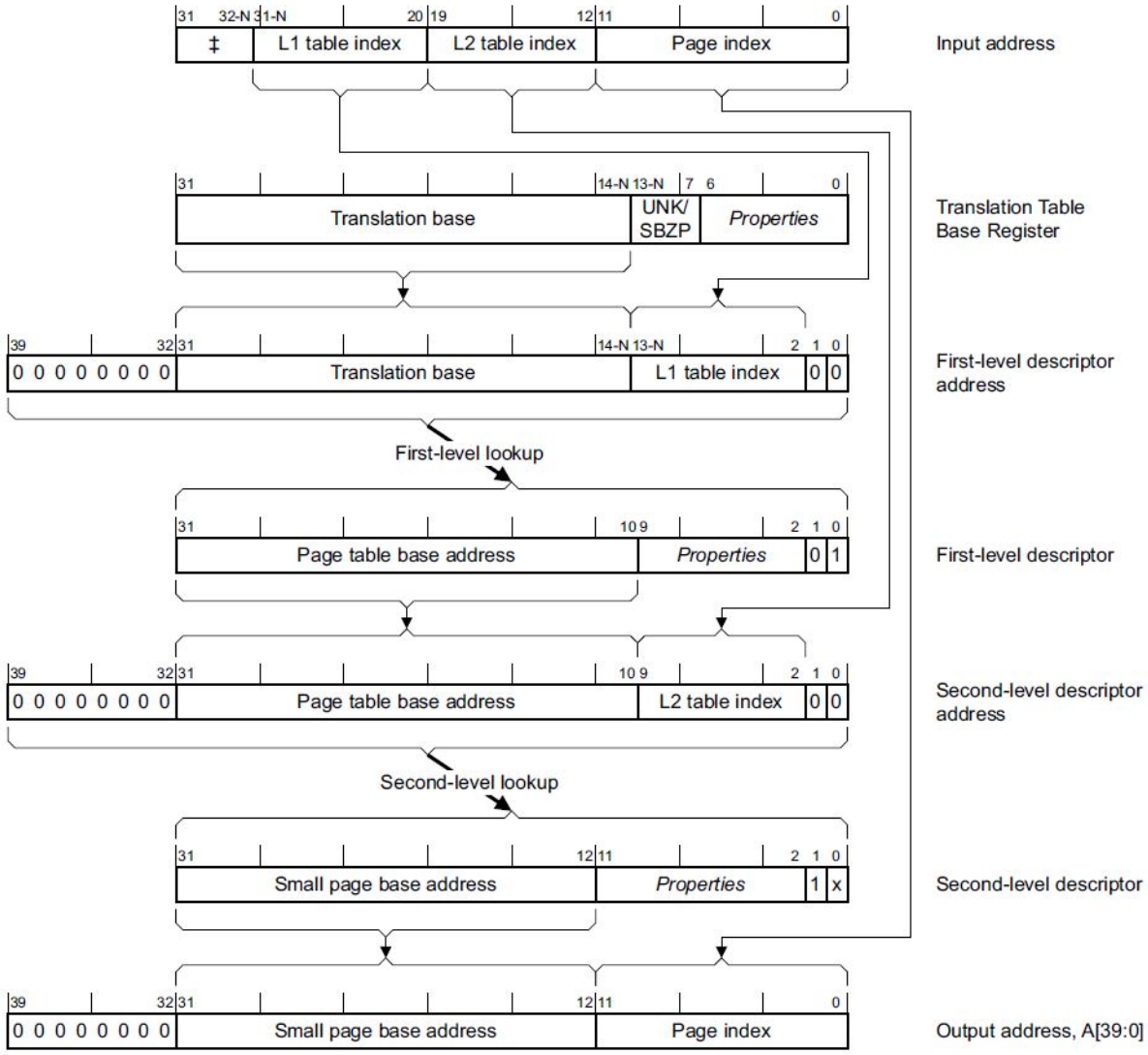

Abbildung 9: Small Page Translation durch die ARM CPU [1, S. B3-1337]

7.3 Seitentabellen und 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 10 und 11 zu sehen. Diese Register übernehmen im Betriebssystem die folgende Funktion:

- TTBR0: Wird für prozessspezifische Adressen verwendet. Jeder Prozess enthält 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.

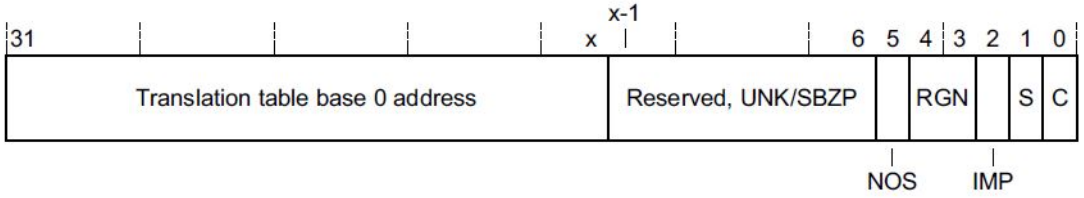

Abbildung 10: TTBR0 Format [1, S. B4-1726]

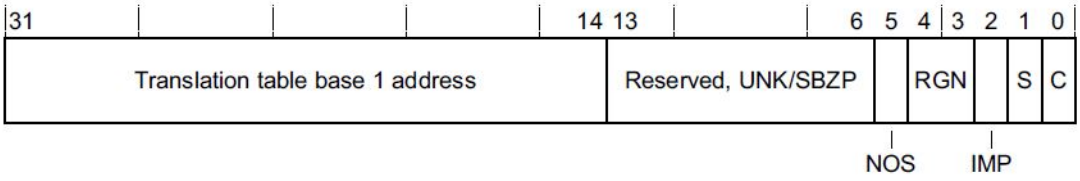

Abbildung 11: TTBR1 Format [1, S. B4-1730]

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 8 aufgeführt ist. Diese Struktur und jene des second-level Deskriptors wird bei den nachfolgenden Erläuterungen zur MMU benötigt.

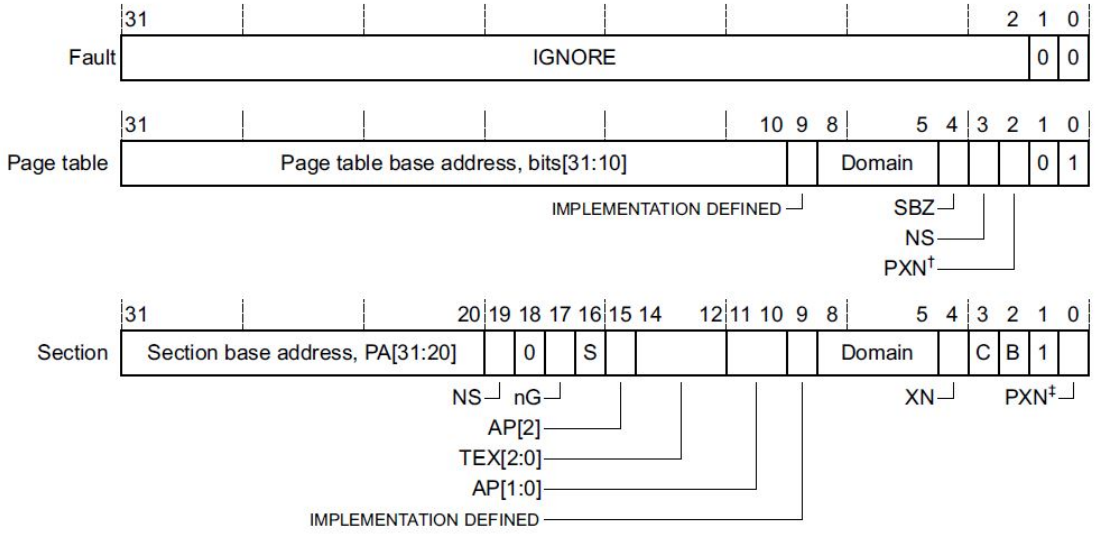

Abbildung 12: First-Level Deskriptorformate [1, S. B3-1326]

```
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 8: 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 9 dargestellt.

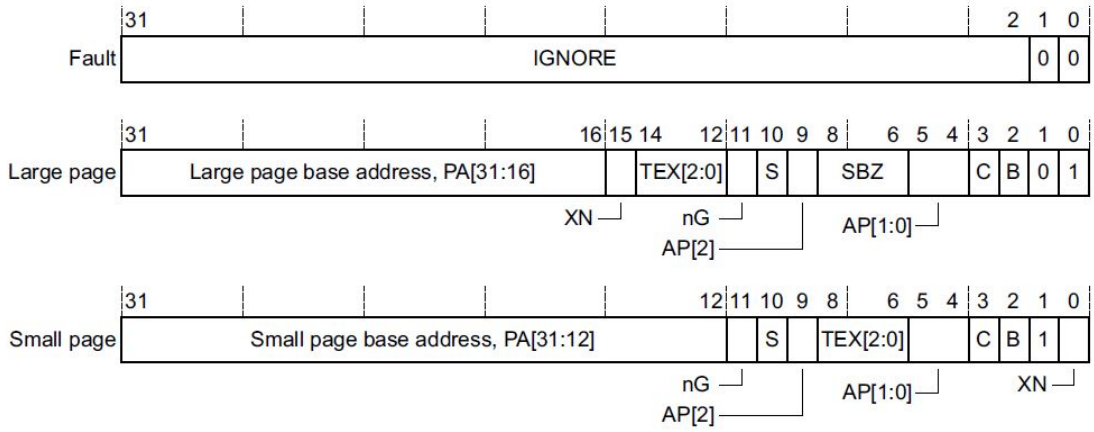

Abbildung 13: Second-Level Deskriptorformate [1, S. B3-1327]

```
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 9: 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.

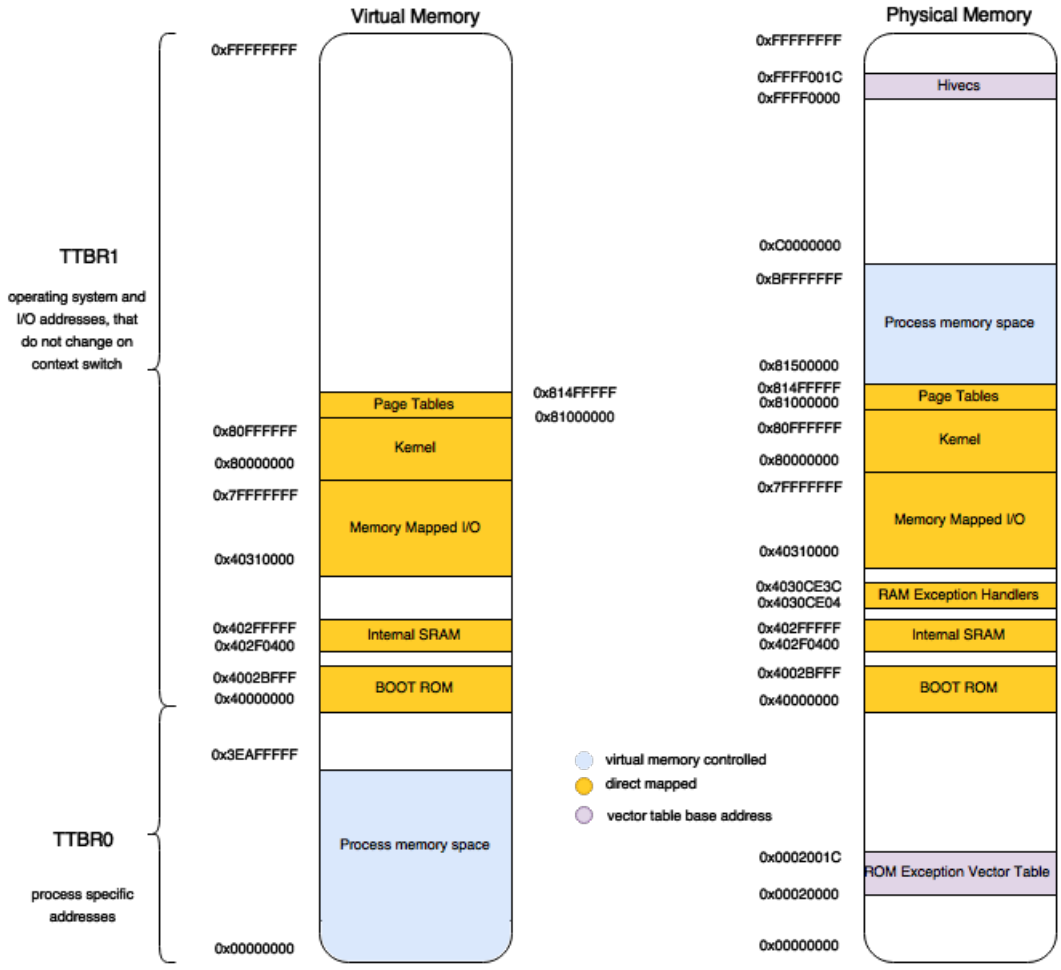

Abbildung 14: Memory Map des Betriebssystems

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 10 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 10: 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 Intex 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 11 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 11: 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 15 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.

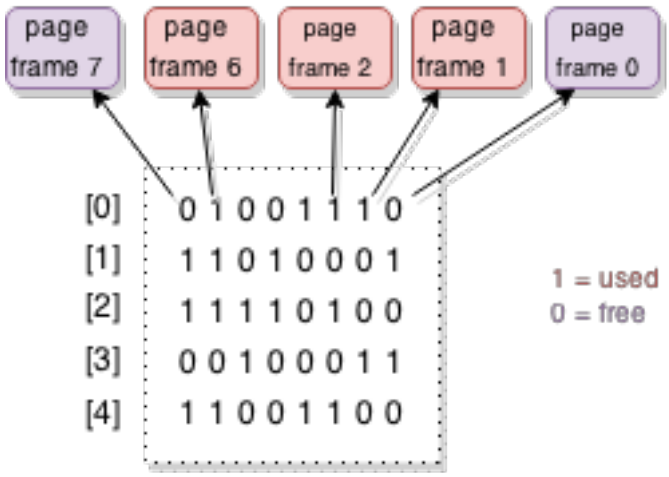

Abbildung 15: Beispiel einer Bitsmap zur Verwaltung der Page Frames

7.5.1 Allokation von Page Frames bei Data Abort Exception

7.6 Aktivieren der MMU

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

Listing 12 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 12: Aktivierung der MMU

7.7 Interaktion der MMU mit Prozessen

Die Schnittstelle der Softwareimplementierung der MMU zeigt Listing 13.

```
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 13: 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 Bitsmap 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 händelt die Interprozesskommunikation zwischen Prozessen. Listening 14 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 14: 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 15 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 15: Aufbau eines Systemcall Datenpakets

9.2 Vorgehensweise bei einem Systemcall

Das Vorgehen bei einem Systemcall ist in Abbildung 16 ersichtlich.

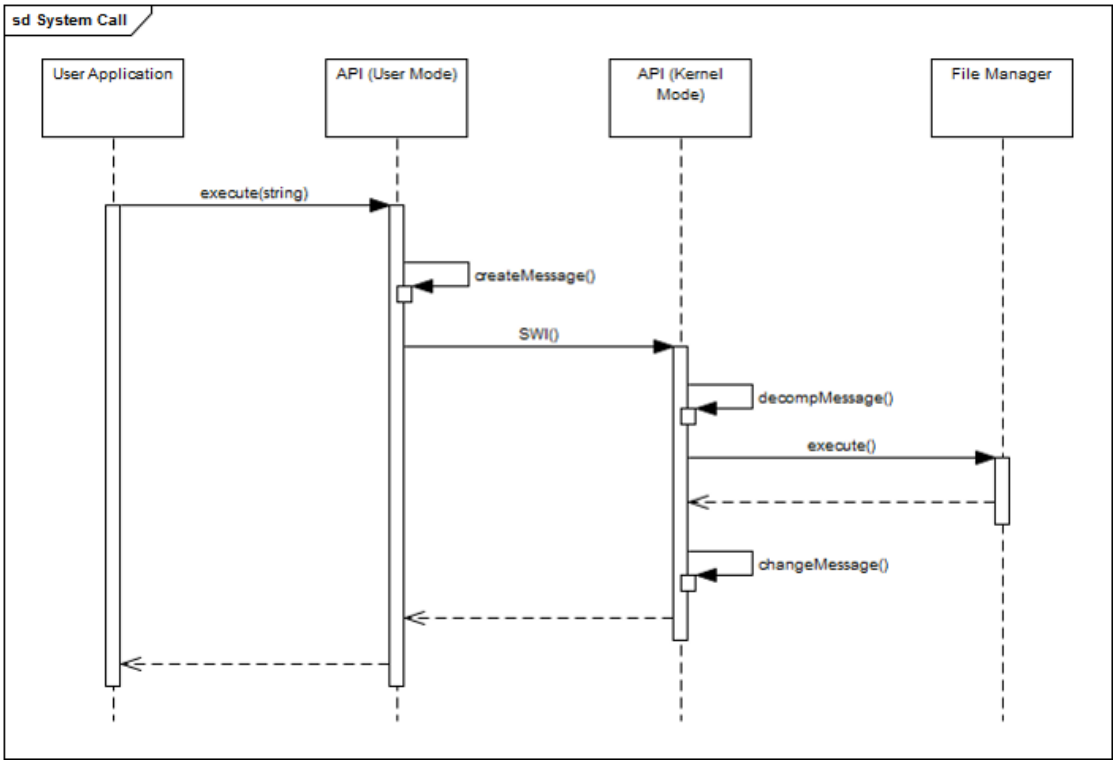

Abbildung 16: Sequenzdiagramm eines Systemcalls

10 Sicherheitsaspekte

Hinsichtlich der Sicherheit werden 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 14. 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: 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 gekillt und der nächste bereite Prozess zur Ausführung gebracht.

10.3 Implementierung der Hivecs

Das Problem der sauberen Trennung der Adressräume for 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 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 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

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

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 8 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 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 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

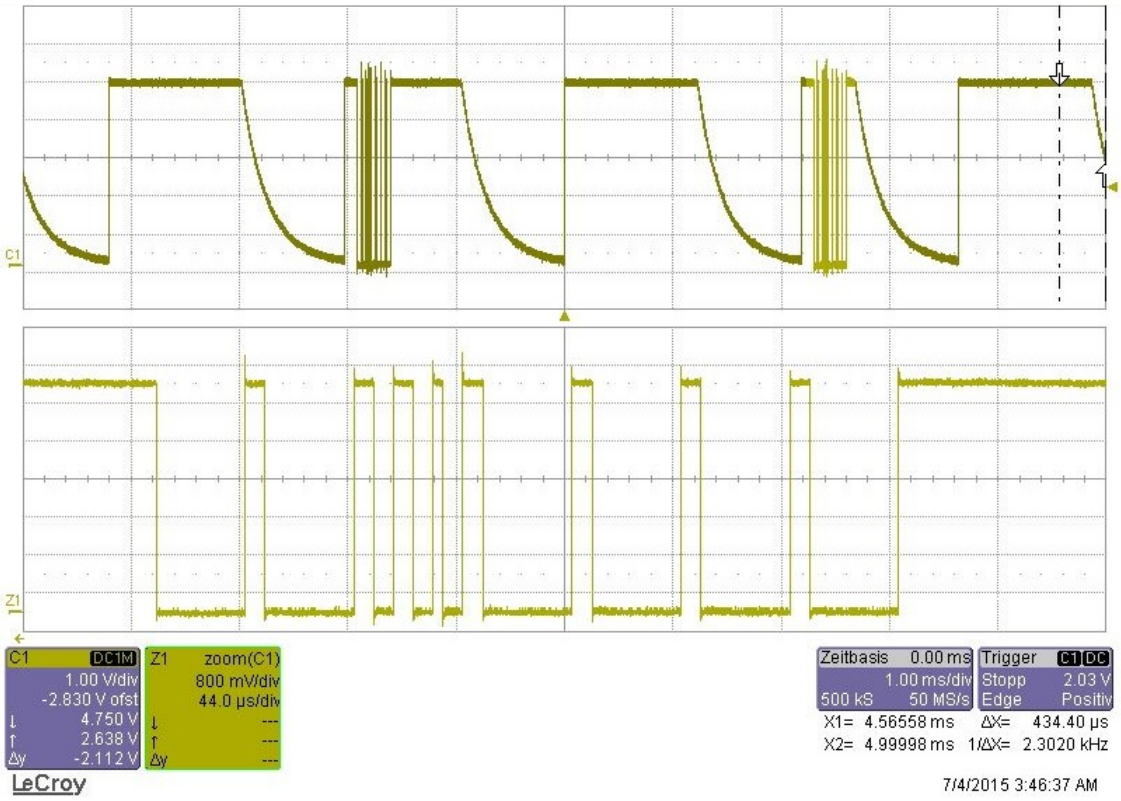

Abbildung 18: DMX Protokoll: Problem fallende Flanke

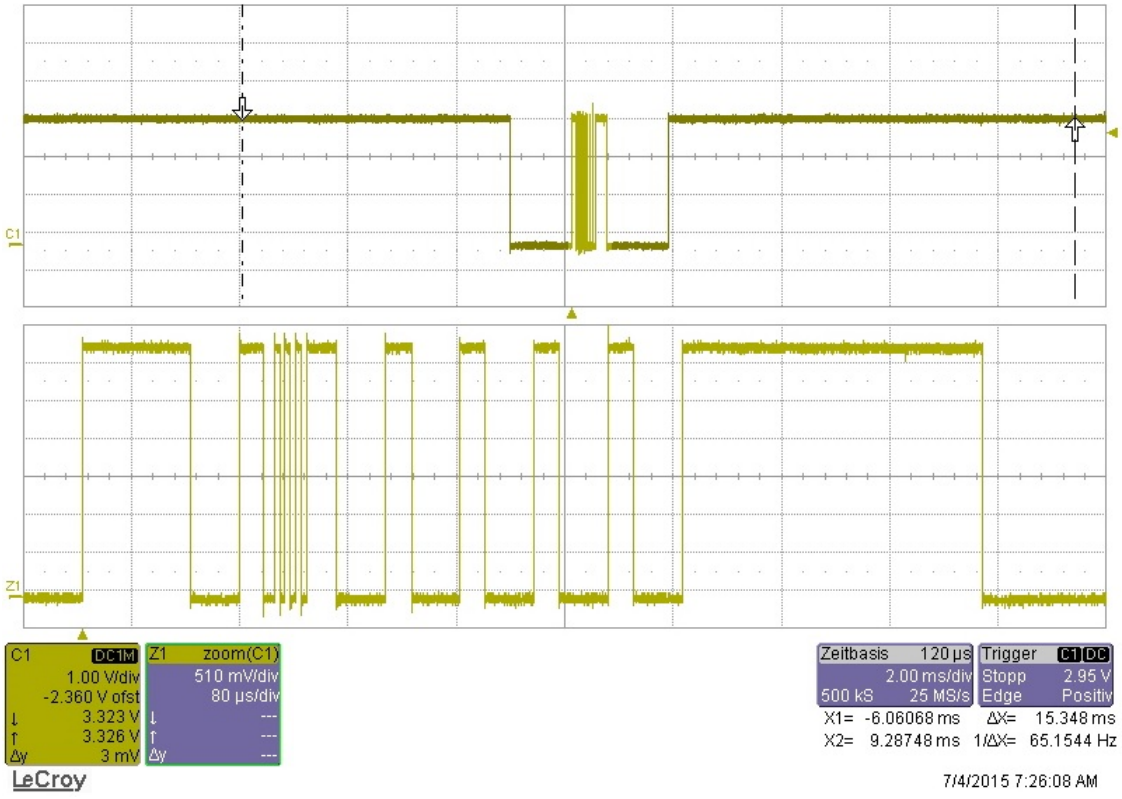

Abbildung 19: DMX Protokoll: Problem Offset Byte

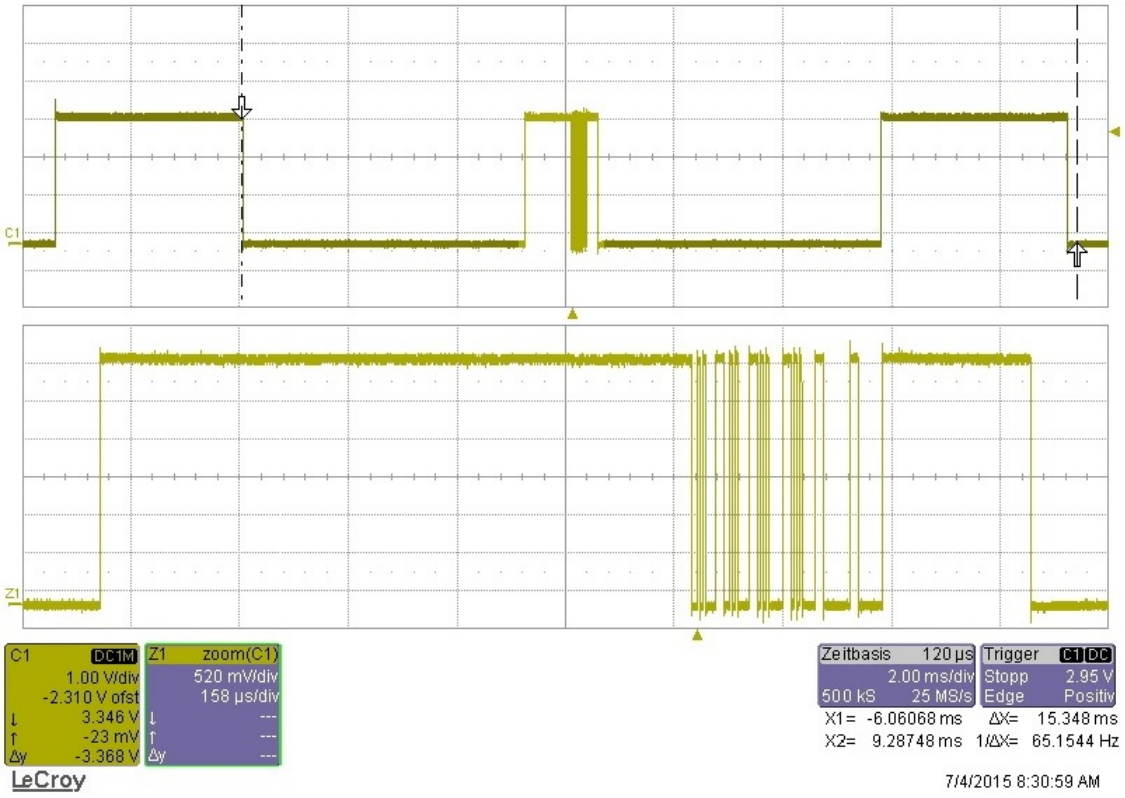

Abbildung 20: Funktionierendes DMX Protokoll

12 Performanz

12.1 Messergebnisse

Es wurden einen Messungen der Performanz mit dem Oszilloskop durchgeführt. Die erhaltenen Messergebnisse sind in Tabelle XXX aufgelistet.

Testfall	**Durchschnittliche Zeit**
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 9: Performanz-Messergebnisse

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 die HAL flexibel und ohne größere Aufwände protierbar. Zudem ist es möglich, von einer MicroSD-Karte Applikationen zu laden und auszuführen. Bei der Implementierung wurden sämtliche Grundaspekte moderner Betriebssysteme, wie beispielsweise die Interprozesskommunikation oder die virtuelle Speicherverwaltung, behandelt. 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 der HAL nur die für die vorgesehene DMX-Applikation benötigten Funktionen implementiert.

13.2 Ausblick

Das Betriebssystem ist in der vorliegenden Form voll einsatzbereit und erfüllt alle gesetzten Anforderungen. Daher wird an dieser Stelle seine Entwicklung seitens des Projektteams eingestellt. Unter dem in der Einleitung angegebenen Repository auf GitHub kann es frei zugänglich heruntergeladen werden. Nachfolgend werden Ansatzpunkte für Verbesserungen bzw. Weiterentwicklungen augelistet.

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 ist ein non-caching System. Eine Einführung des Cachings für Daten hätte eine Verbesserung der Performance zur Folge.

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
ResourceManager	Dieser Manager ist für die Verwaltung von Ressourcen zuständig. Dazu zählen XXX

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.