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

19. Juli 2015, Dornbirn

Abstract

Inhaltsverzeichnis

1 Allgemein **6**
- 1.1 Vorgegebene Anforderungen an das Betriebssystem . . . 6
- 1.2 Eigene Anforderungen an das Betriebssystem . . . 7
- 1.3 Resultat des Betriebssystems . . . 7

2 Projektmanagement **8**
- 2.1 Prozessmodell . . . 8
- 2.2 Versionsverwaltung . . . 8
- 2.3 Repository . . . 8
- 2.4 Zeitplan . . . 8

3 Architektur **9**
- 3.1 Art des Kernels . . . 9
- 3.2 Ansatz für die Abstraktionen im Betriebssystem . . . 9
- 3.3 Allgemeiner Aufbau der Architektur . . . 9

4 Hardware Abstraction Layer (HAL) **13**
- 4.1 Aufbau der HAL Schnittstelle . . . 13
- 4.2 Interrupts . . . 13

5 Treiber **14**
- 5.1 Allgemeiner Aufbau eines Treibers . . . 14
- 5.2 Beispiel Implementierung eines Treibers . . . 14
- 5.3 DriverManager . . . 15

6 Prozessverwaltung **17**
- 6.1 Prozesszustände . . . 17

7 Virtuelle Speicherverwaltung **19**
- 7.1 Grundlegende Funktionsweise . . . 19
  - 7.1.1 Data Abort Handler . . . 20
- 7.2 Umwandlung virtueller Adressen zu physikalische Adressen . . . 21
- 7.3 Seitentabellen und Seitentabelleneinträge . . . 22
- 7.4 Aufteilung des virtuellen Speichers und Mapping . . . 25
  - 7.4.1 Speicherregionen . . . 27
  - 7.4.2 Master Page Table . . . 28
- 7.5 Allokierung der Page Frames . . . 29
  - 7.5.1 Allokation von Page Frames bei Data Abort Exception . . . 29
- 7.6 Aktivieren der MMU . . . 29
- 7.7 Interaktion der MMU mit Prozessen . . . 30

8 Interprozesskommunikation **32**
- 8.1 Aufbau . . . 32

9 System API **33**
9.1 Aufbau 33

10 BenutzerInnen-Anwendung **34**
10.1 Grundlegender Aufbau des DMX Protokolls 34
10.2 Messergebnisse des Implementierten DMX Protokolls 36

11 Performanz **39**
11.1 Aufbau 39

12 Zusammenfassung **40**
12.1 xxx 40

13 Ausblick **41**
13.1 Aufbau 41

Abbildungsverzeichnis

1 Allgemeiner Aufbau der Architektur 10
2 Erlaubte Prozesszustände und Prozessübergänge 17
3 Zweistufiges Seitentabellensystem [**?**, S. B3-1325] 20
4 1 MB Section Translation durch die ARM CPU [**?**, S. B3-1335] 21
5 Small Page Translation durch die ARM CPU [**?**, S. B3-1337] 22
6 TTBR0 Format [**?**, S. B4-1726] 23
7 TTBR1 Format [**?**, S. B4-1730] 23
8 First-Level Deskriptorformate [**?**, S. B3-1326] 24
9 Second-Level Deskriptorformate [**?**, S. B3-1327] 25
10 Memory Map des Betriebssystems 26
11 Beispiel einer Bitsmap zur Verwaltung der Page Frames 29
12 DMX Protokoll 34
13 DMX Protokoll: Problem fallende Flanke 36
14 DMX Protokoll: Problem Offset Byte 37
15 Funktionierendes DMX Protokoll 38

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

Allgemeine Aspekte zum Betriebssystem werden in diesem Kapitel erläutert. Dazu zählen vorgegebene Anforderungen, eigene Anforderungen und das Ergebnis in Zusammenhang mit den zuvor definierten Anforderungen.

1.1 Vorgegebene Anforderungen an das Betriebssystem

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

Anforderung	**Erklärung**
Single-User	Das Betriebssystem muss zu jedem Zeitpunkt nur einen Benutzer bzw. eine Benutzerin händeln.
Lauffähige Anwendung	Auf dem Betriebssystem muss sich eine lauffähige Anwendung befinden.
Präemptives Multitasking	Das Betriebssystem kann gleichzeitig mehrere Prozesse ausführen.
Konsole	Es muss ein Konsole zur Kommunikation mit dem Betriebssystem geben.
Interprozess-Kommunikation	Das Betriebssystem muss Möglichkeit zu Kommunikation zwischen Prozessen zur Verfügung stellen.
Sicherheit	Es muss eine strikte Trennung zwischen User- und Systemmode vorhanden sein.
Robustheit	Das Betriebssystem darf von Programmabstürzen nicht beeinflusst werden.
Virtueller Speicher	Memory Management muss für größere Anwendungen vorhanden sein.
SD-Karte	Externe Anwendungen sollten von der SD-Karte geladen werden können.
Dateisystem	Das Betriebssystem muss ein Dateisystem (FAT oer FAT32) besitzen.
Portierbarkeit	Für eine Portierbarkeit des Systems muss ein 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

Eine Auflistung aller eigenen Anforderungen an das Betriebssystem sind in Tabelle 2 angegeben.

Anforderung	**Erklärung**
Hoher Abstraktionsgrad	Alle Komponenten sollten voneinander abstrahiert sein.
Intuitiver Aufbau	Die Komponenten des Betriebssystem sollten möglichst selbsterklärend sein.
Leichte Erweiterbarkeit	Laufende Erweiterungen sollten ohne große Veränderungen umgesetzt werden können.
Einfache Wartung	Das Betriebssystem sollte so Aufgebaut sein, dass eventuelle Ausbesserungen am Betriebssystem nur an einer dafür zuständigen Komponente vorgenommen werden müssen.

Tabelle 2: Vorgegebene Anforderungen

1.3 Resultat des Betriebssystems

Im Allgemeinen wurden alle zuvor erwähnten Anforderungen an das Betriebssystem erfüllt. Einzelne Verbesserungs- bzw. Erweiterungsvorschläge können aus Kapitel XXX - Ausblick entnommen werden.
Stabilitätstests, welche während dem Projekt durchgeführt wurden, haben aufgezeigt, dass das System über zwölf Stunden fehlerlos durchläuft.

2 Projektmanagement

Im folgenden Abschnitt genauer auf das Projektmanagement eingegangen. Die zentrale Punkte in diesem Kapitel sind: das Prozessmodell, die Versionsverwaltung, das Repository sowie der Zeitplan.

2.1 Prozessmodell

Als Prozessmodell wurde SCRUM mit einigen Abänderungen umgesetzt. Wobei das zentrale Vorgehen in Bezug auf Agilität bestmöglich übernommen wurde. Gründe für die Verwendung von SCRUM sind: Agilität, nach jedem Sprint ein lauffähiges System, klares Ziel für jeden Sprint, einfaches Hinzufügen fehlender Aufgaben (neue Stories), einfaches neu priorisieren von Aufgaben, Ereignisse die den Entwicklungszyklus beeinflussen (z.B. Klausuren) beeinträchtigen das weitere Vorgehen nicht, klare Übersicht der fehlenden Stories.

Alle Aufgaben wurden im Repository als Issues aufgenommen und können unter folgendem Link eingesehen werden:

```
https://github.com/Blackjack92/fhvOS/issues
```

Insgesamt wurden mehr als einhundert Issues aufgenommen und davon über neunzig Prozent gelöst. Alle offenen Punkte sind mit einer niederen Priorität eingestuft und beeinflussen die korrekte Funktionsfähigkeit des Betriebssystem nicht.

2.2 Versionsverwaltung

Als Versionsverwaltung wurde Git verwendet. Zwei Gründe für die Verwendung von Git sind leichte Einbindung im Zusammenhang mit dem Repository (siehe **??**) und die Möglichkeit der Nicht-linearen Entwicklung.

2.3 Repository

Das Repository wurde auf Github angelegt und veröffentlicht. Das Veröffentlichen war nötig, da dies eine Voraussetzung für die kostenlose Nutzung von Github ist. Unter dem folgenden Link kann das Projekt eingesehen werden:

```
https://github.com/Blackjack92/fhvOS
```

2.4 Zeitplan

Der Zeitplan des Projekts wurde mittels Microsoft Project erstellt und wurde während des gesamten Projekts, bis auf wenige Ausnahmen, eingehalten. In den beiliegenden Unterlagen ist der Zeitplan unter der Bezeichnung XXX zu finden.

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.

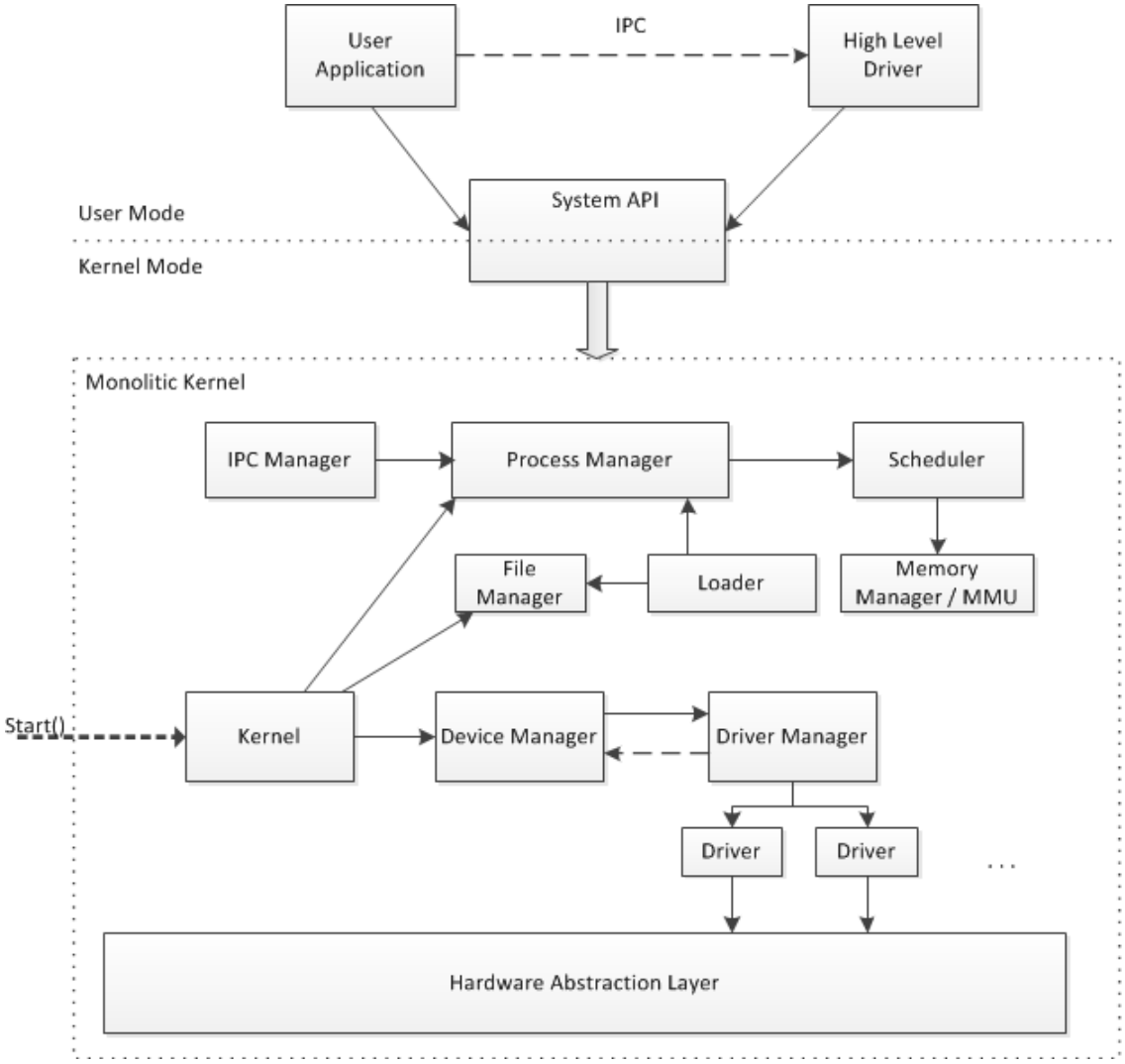

Abbildung 1: Allgemeiner Aufbau der Architektur

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 XXX)
Der Hardware Abstraction Layer wird, wie der Name bereits beschreibt, zur Abstraktion der Hardware vom eigentlichen Betriebssystem verwendet.

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

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

Scheduler (Vgl. Kapitel XXX)
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 XXX)
Über den Memory Manager können freie Pages in der Prozessregion allokiert werden sowie die eine bestehende Region zurückgeliefert werden.

File Manager (Vgl. Kapitel XXX)
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 (Vgl. Kapitel XXX)
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 XXX)
Der IPC Manager ist für die Kommunikation zwischen verschiedenen User-Anwendungen zuständig.

System API (Vgl. Kapitel XXX)

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 XXX)

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 (Vgl. Kapitel XXX)

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.

Listing 1: HAL Schnittstelle für die GPIOs

```
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 2: Schnittstelle für die UART

```
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);
```

4.2 Interrupts

bla

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 3 ist die allgemeine Schnittstelle für jeden Treiber ersichtlich.

Listing 3: Allgemeine Schnittstelle für einen Treiber

```
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;
```

5.2 Beispiel Implementierung eines Treibers

Jeder implementierte Treiber muss diese vorweisen können, ein Beispiel (LED Treiber) dazu ist in Listening 4 zu sehen.

Listing 4: Implementierung der allgemeinen Treiberschnittstelle (LED Beispiel)

```
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 };
```

```
23    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  }
```

5.3 DriverManager

Der DriverManager hat die Aufgabe die Treiber zu initialisieren sowie diese dann nach außen hin anzubieten. In Listening 5 ist die Schnittstelle des DriverManagers dargestellt. Eine Implementierung dieser Schnittstelle für das LED Beispiel ist in Listening 6 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]

Listing 5: Allgemeine Schnittstelle des DriverManagers

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

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

```
5 extern void DriverManagerDestruct();
```

Listing 6: Implementierung der DriverManager Schnittstelle für den LED Treiber

```
1  static driver_t* drivers[MAX_DRIVER];
2
3  void DriverManagerInit()
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()
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 }
```

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 2 zeigt die verschiedenen Zustände eines Prozesses sowie die jeweilig erlaubten Übergänge zu einem anderen Zustand auf.

TODO!!!

Abbildung 2: Erlaubte Prozesszustände und Prozessübergänge

Im folgenden wird eine detaillierte Erklärung zu den einzelnen Zuständen aus Abbildung 2 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

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 [**?**, 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	**Beschreibung**
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 3. 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.

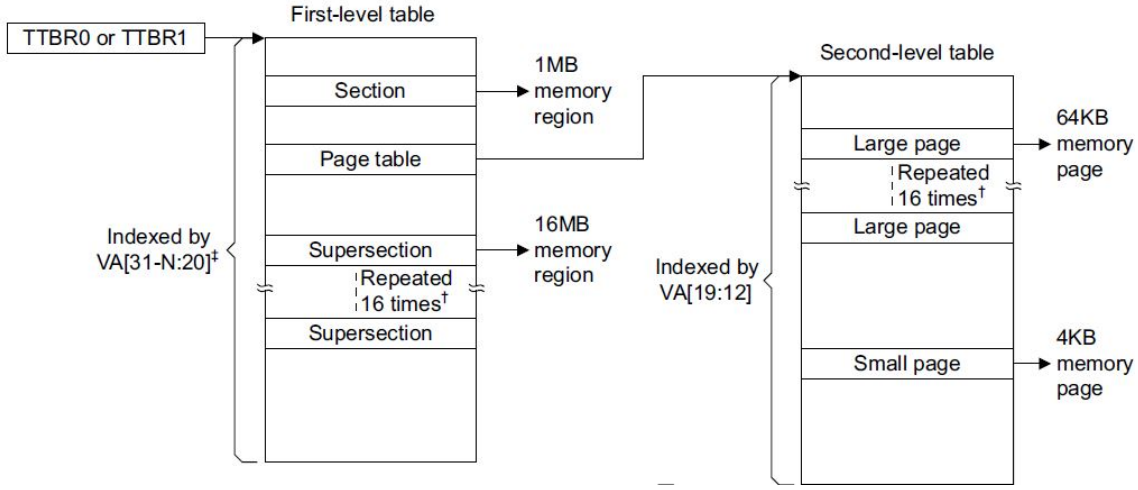

Abbildung 3: Zweistufiges Seitentabellensystem [**?**, S. B3-1325]

7.1.1 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 4 zeigt die Umwandlung einer virtuellen Adresse in die physikalische Adresse einer 1 MB Section ohne Verwendung einer L2-Seitentabelle, Abbildung 5 diejenige einer virtuellen Adresse in ein 4 kB page frame unter Verwendung einer L2-Seitentabelle. Die Umwandlung wird vollständig durch die Prozessor-Hardware durchgeführt.

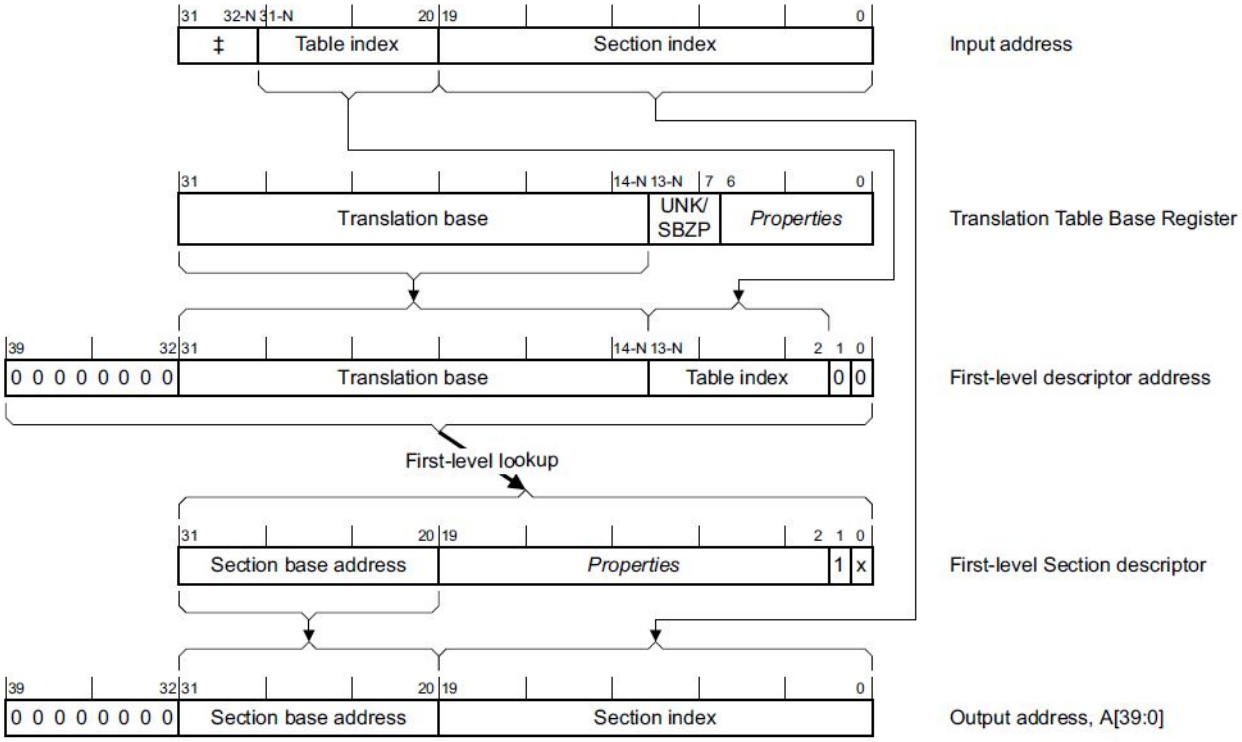

Abbildung 4: 1 MB Section Translation durch die ARM CPU [**?**, S. B3-1335]

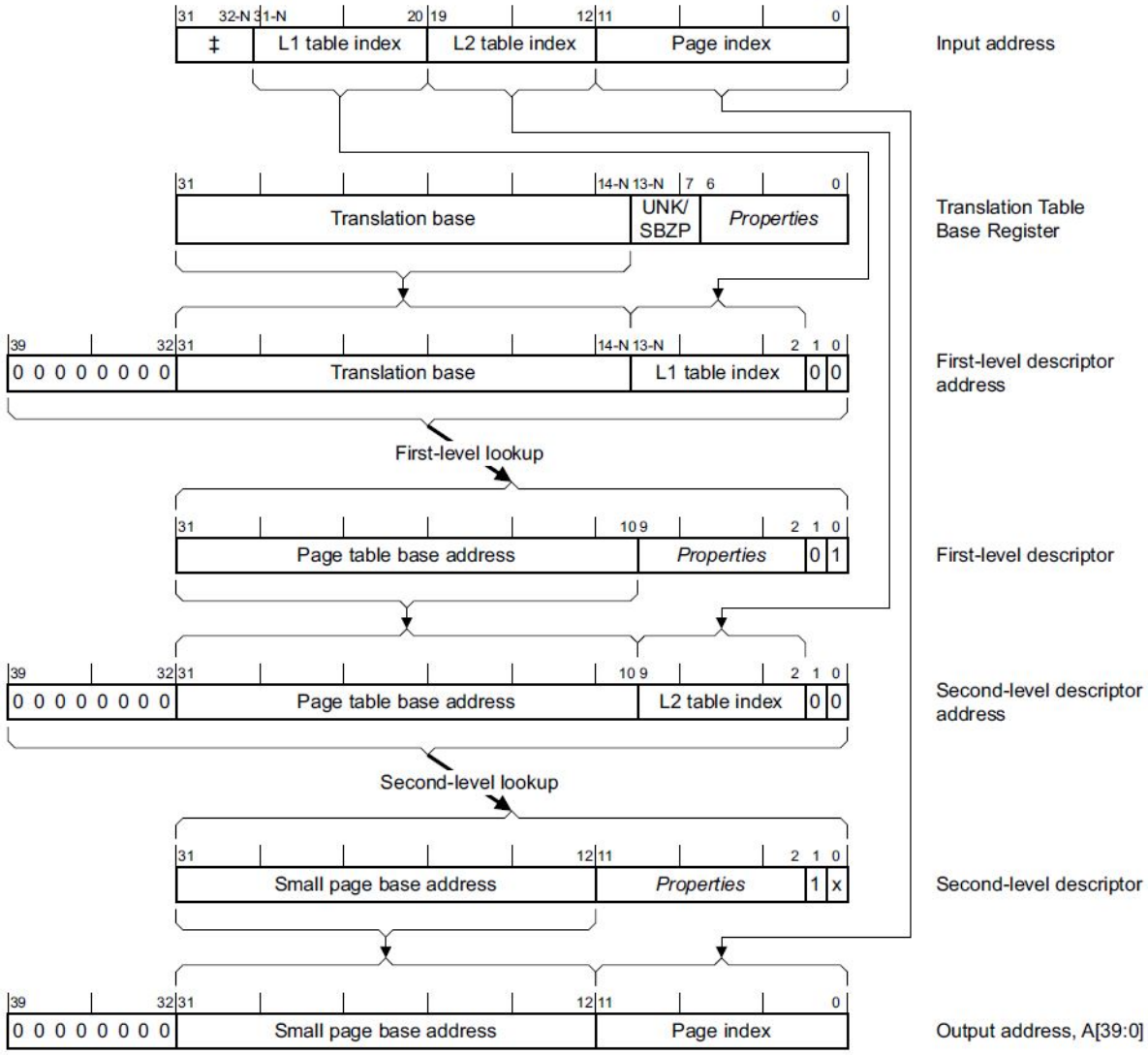

Abbildung 5: Small Page Translation durch die ARM CPU [**?**, S. B3-1337]

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 [**?**, S. B3-1320]. Ihre Formate sind nahezu identisch und in den Abbildungen 6 und 7 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.

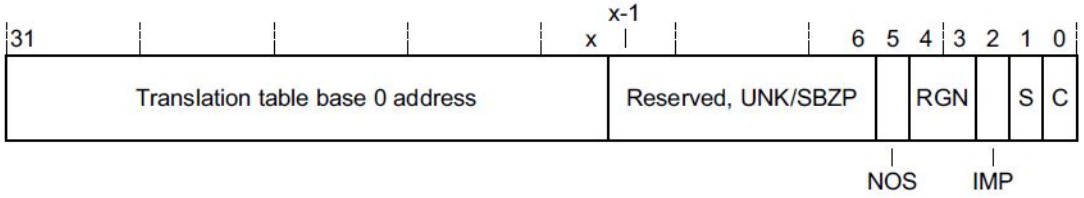

Abbildung 6: TTBR0 Format [**?**, S. B4-1726]

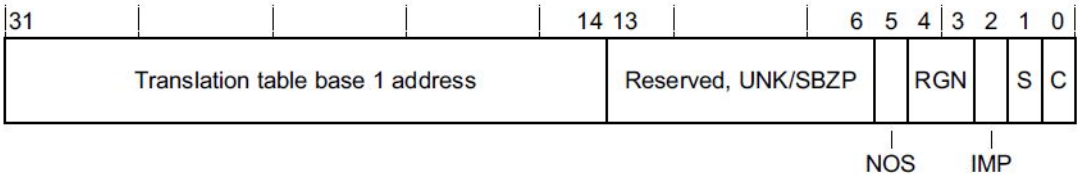

Abbildung 7: TTBR1 Format [**?**, S. B4-1730]

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 8 und 9 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 7 aufgeführt ist. Diese Struktur und jene des second-level Deskriptors wird bei den nachfolgenden Erläuterungen zur MMU benötigt.

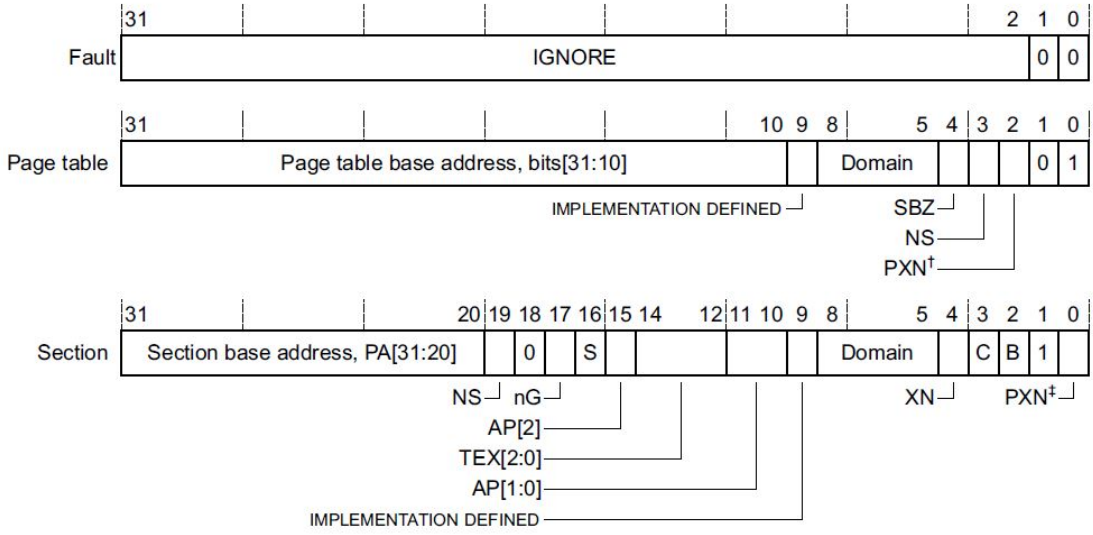

Abbildung 8: First-Level Deskriptorformate [**?**, S. B3-1326]

Listing 7: Struktur für first-level Deskriptoren

```
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;
```

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 8 dargestellt.

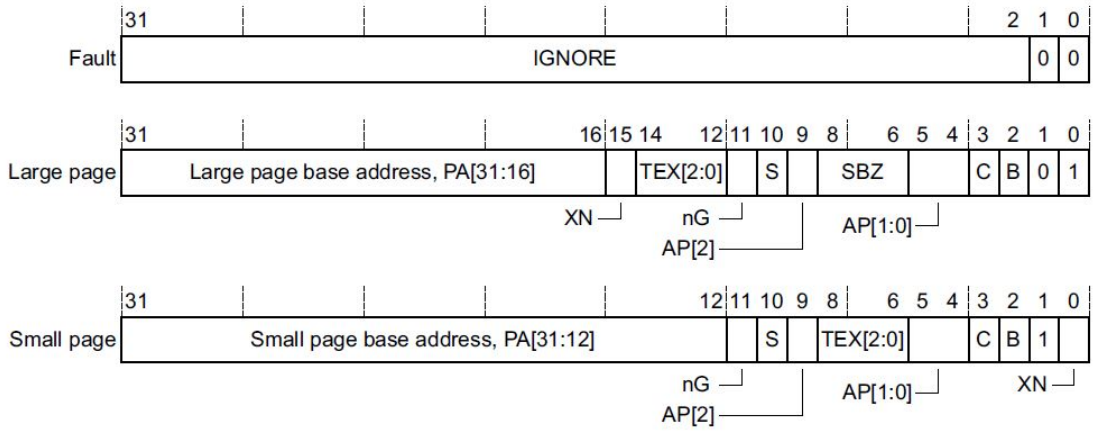

Abbildung 9: Second-Level Deskriptorformate [**?**, S. B3-1327]

Listing 8: Struktur für second-level Deskriptoren

```
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;
```

7.4 Aufteilung des virtuellen Speichers und Mapping

Die Speicherverwaltung des Betriebssystems kann Abbildung 10 entnommen werden. Die rechte Seite stellt das physikalische Speichermapping dar und wurde dem Datenblatt des ARM [**?**, 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, [**?**, 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 10 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.

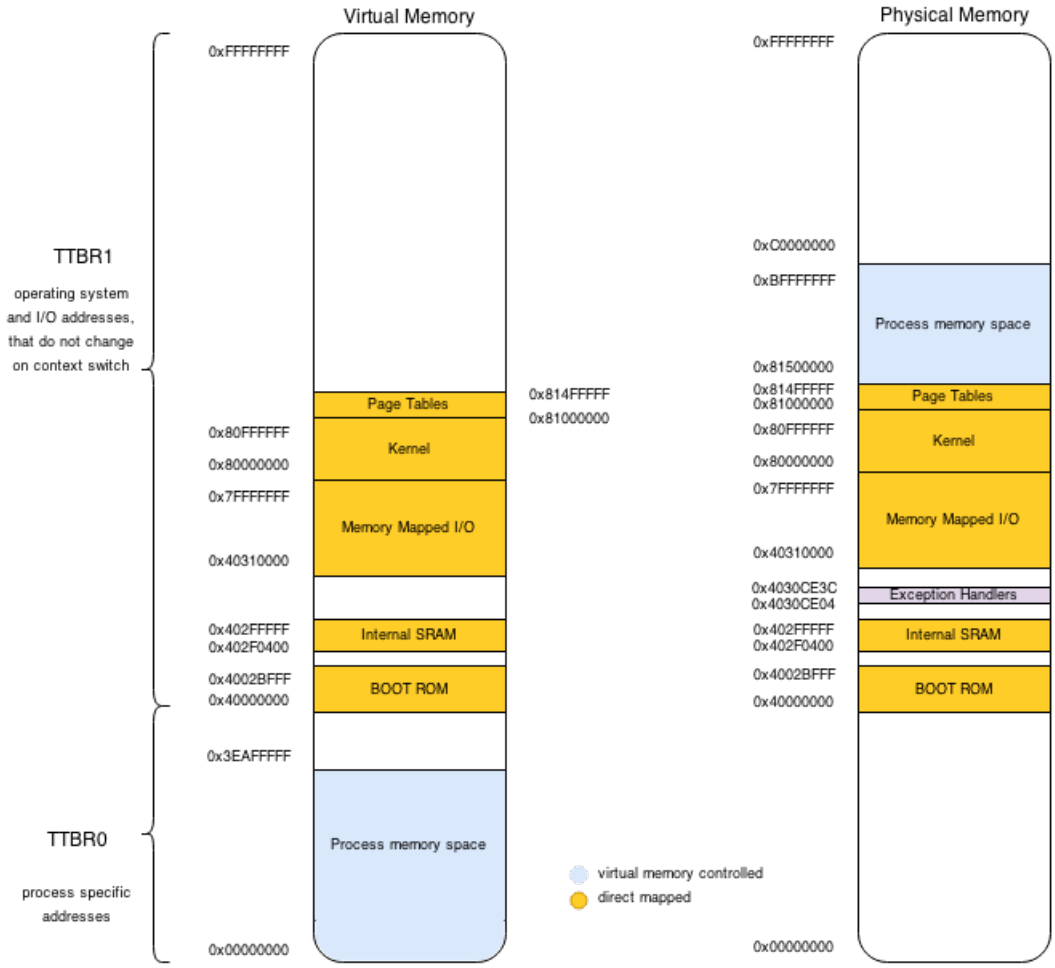

Abbildung 10: Memory Map des Betriebssystems

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 9 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 10 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.

Listing 9: Struktur für die Verwaltung von Speicherregionen

```
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;
```

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 4 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 10 zeigt die praktische Ausführung des direkten Mappings in die MPT.

Listing 10: Funktion für direktes Mapping in die master page table

```
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  }
```

7.5 Allokierung der Page Frames

Für die Verwaltung der page frames wurde eine Bitsmap verwendet. Abbildung 11 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.

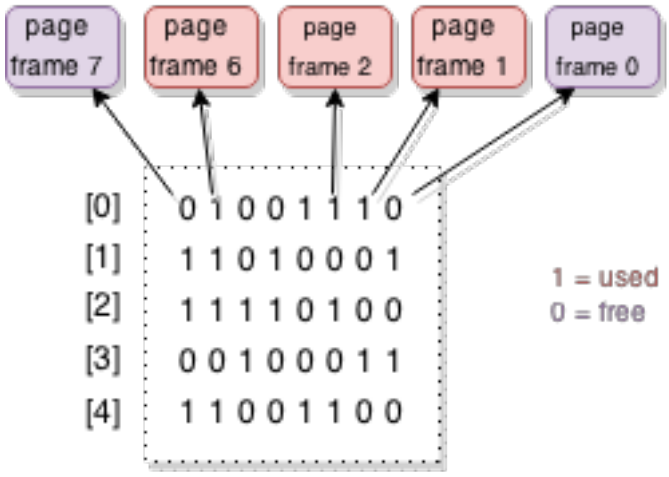

Abbildung 11: Beispiel einer Bitsmap zur Verwaltung der Page Frames

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 11 zeigt den kompletten Ablauf zur Aktivierung der MMU.

Listing 11: 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 }
```

7.7 Interaktion der MMU mit Prozessen

Die Schnittstelle der Softwareimplementierung der MMU zeigt Listing 12.

Listing 12: Softwareschnittstelle der MMU

```
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);
```

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

bla

8.1 Aufbau

bla

9 System API

bla

9.1 Aufbau

bla

10 BenutzerInnen-Anwendung

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

10.1 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 12 dient zur Veranschaulichung des DMX-512 Protokolls.

TODO!!!

Abbildung 12: DMX Protokoll

Tabelle 8 beschreibt die einzeln nummerierten Markierungen aus Abbildung 12.

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 12 ist ersichtlich, dass jedes übertragene Datenbyte zusätzlich ein Startbit sowie zwei Stopbits besitzt. Somit ergeben sich für jeden Kanal genau elf Bits.

10.2 Messergebnisse des Implementierten DMX Protokolls

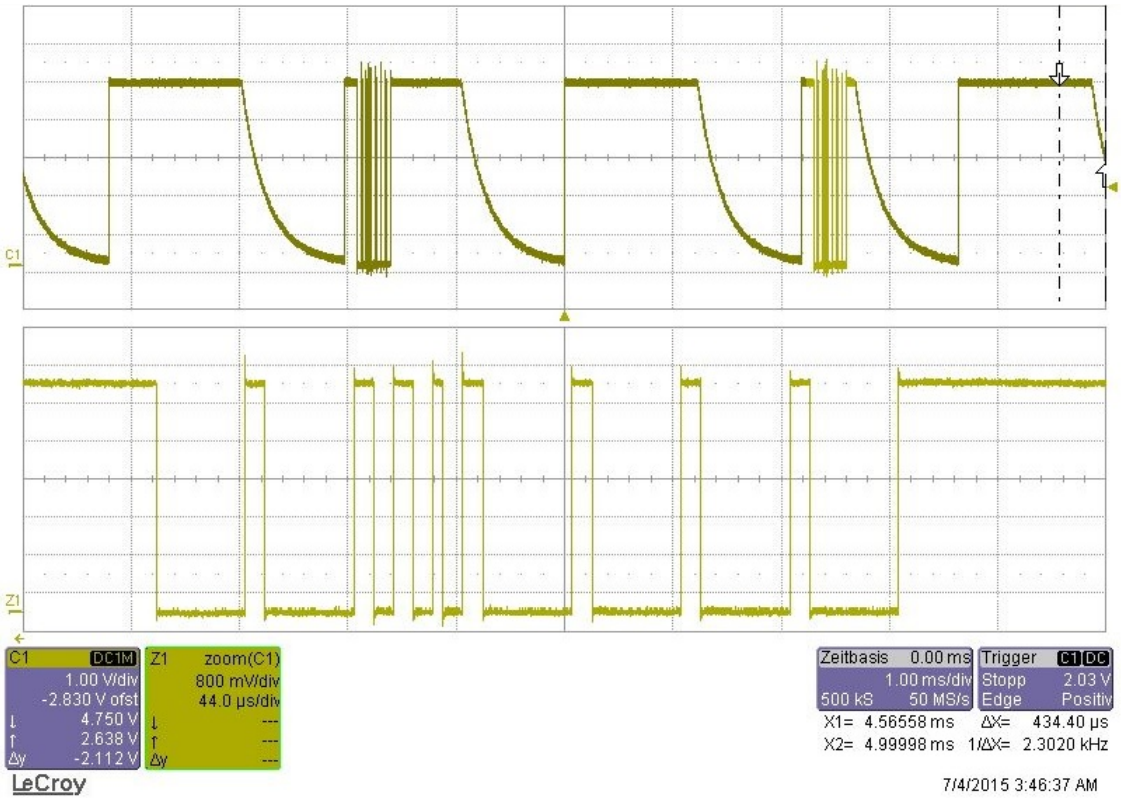

Abbildung 13: DMX Protokoll: Problem fallende Flanke

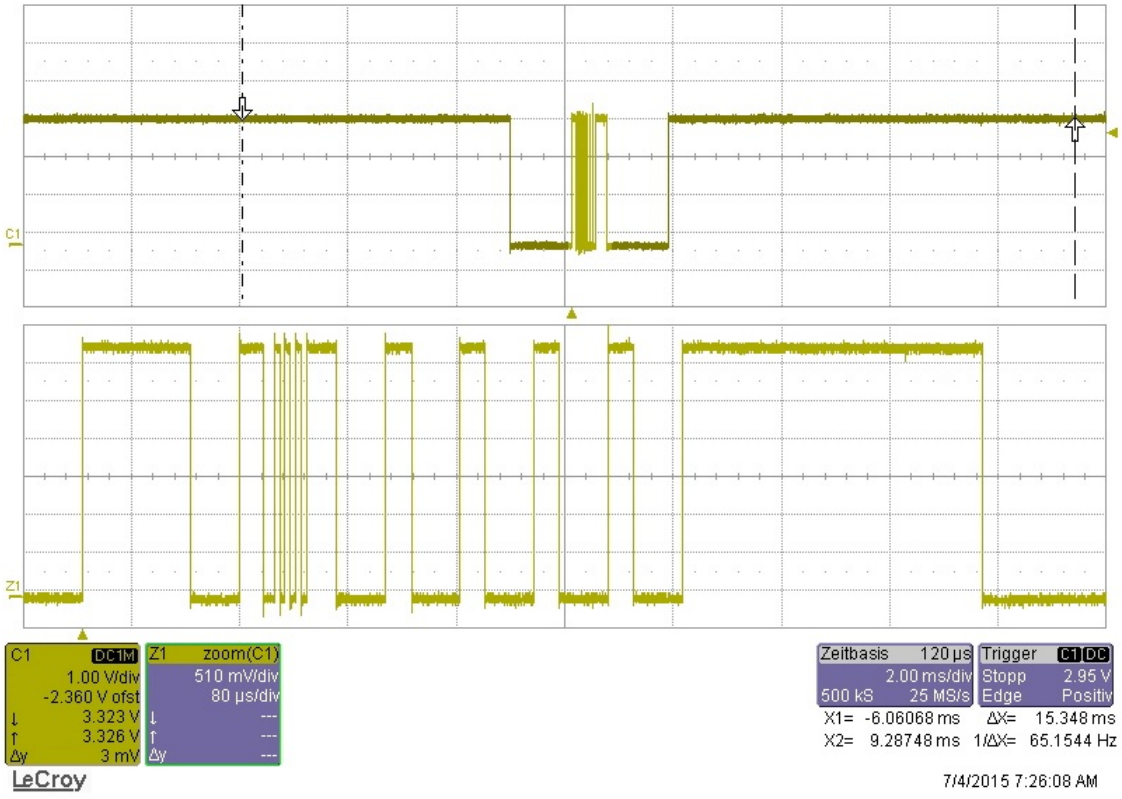

Abbildung 14: DMX Protokoll: Problem Offset Byte

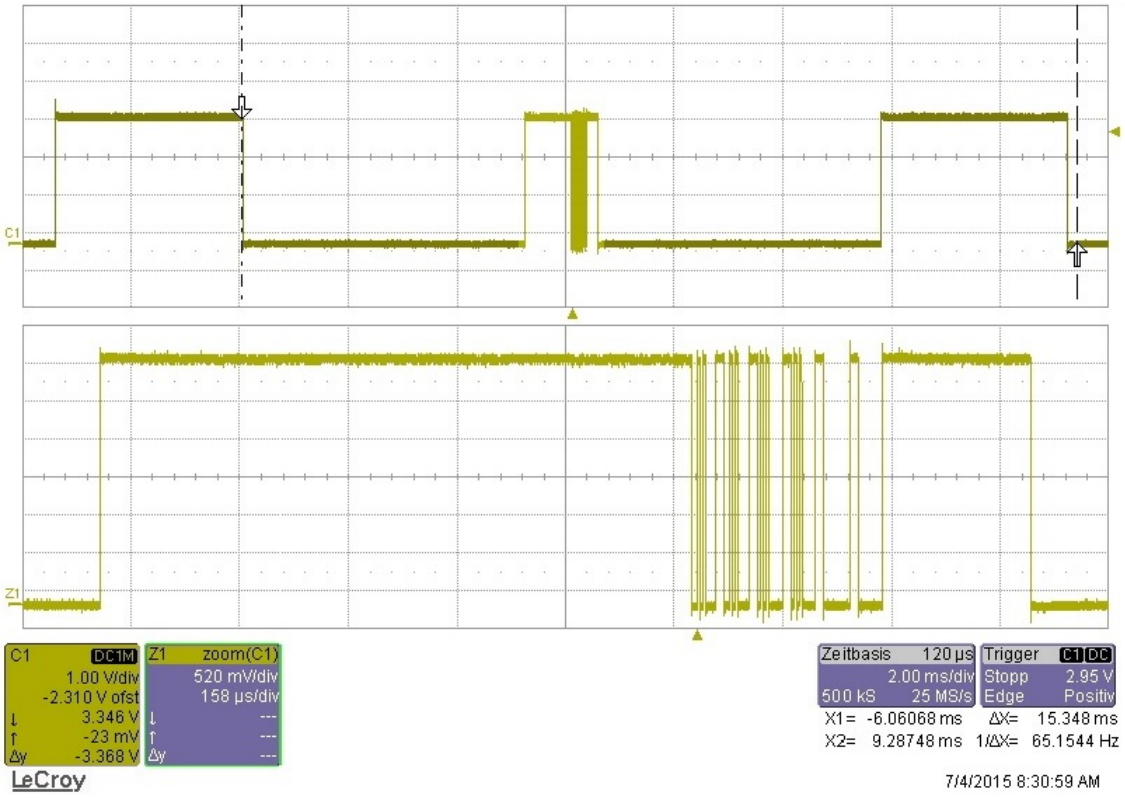

Abbildung 15: Funktionierendes DMX Protokoll

11 Performanz

bla

11.1 Aufbau

bla

12 Zusammenfassung

bla

[**?**]

[**?**]

12.1 xxx

bla

13 Ausblick

bla

13.1 Aufbau

bla