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

14. Juli 2015, Dornbirn

## Abstract

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

**DMX** Digital Multiplex

**VMSAv7** Virtual Memory System Architecture for ARMv7

**MMU** Memory Management Unit

**OS** Operating System

**HAL** Hardware Abstraction Layer

**PTE** Page Table Entry

**TLB** Translation Lookaside Buffer

**TTBR** Translation Table Base Register

**TTBCR** Translation Table Base Control Register

**DFSR** Data Fault Status Register

**DFAR** Data Fault Address Register

**MPT** Master Page Table

# 1 Allgemein

bla

## 1.1 Aufbau

bla

# 2 Projektmanagement

Im folgenden Abschnitt werden einige Punkte zum Projektmanagement erklärt.

## 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:

- 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. Das Repository wurde veröffentlicht, 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

bla

## 3.1 Aufbau

bla

# 4 HAL

bla

## 4.1 Aufbau

bla

## 4.2 Interrupts

bla

# 5 Treiber

bla

## 5.1 Aufbau

bla

# 6 Prozessverwaltung

bla

## 6.1 Aufbau

bla

# 7 Virtelle 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 Ausgangsaddresse
- *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 Ausgangsaddresse

Um die Anforderungen an das Betriebssystem zu erfüllen, reicht das zweistufige Seitentabellensystem vollkommen aus. Tabelle 1 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 1: 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 1. 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.

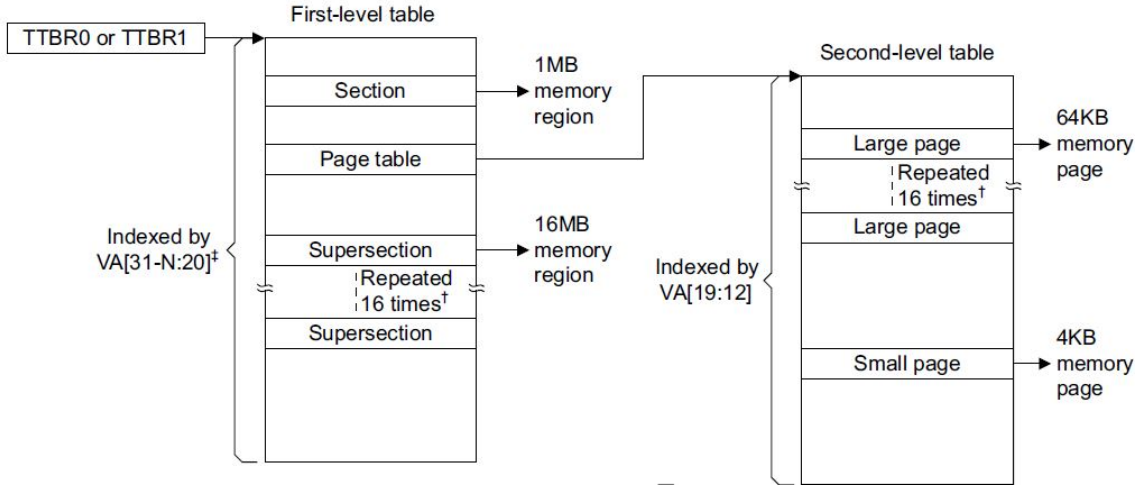


Abbildung 1: Zweistufiges Seitentabellensystem [**?**, S. B3-1325]

### 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 2 zeigt die Umwandlung einer virtuellen Adresse in die physikalische Adresse einer 1 MB Section ohne Verwendung einer L2-Seitentabelle, Abbildung 3 diejenige einer virtuellen Adresse in ein 4 kB page frame unter Verwendung einer L2-Seitentabelle. Die Umwandlung wird vollständig durch die Prozessor-Hardware durchgeführt.

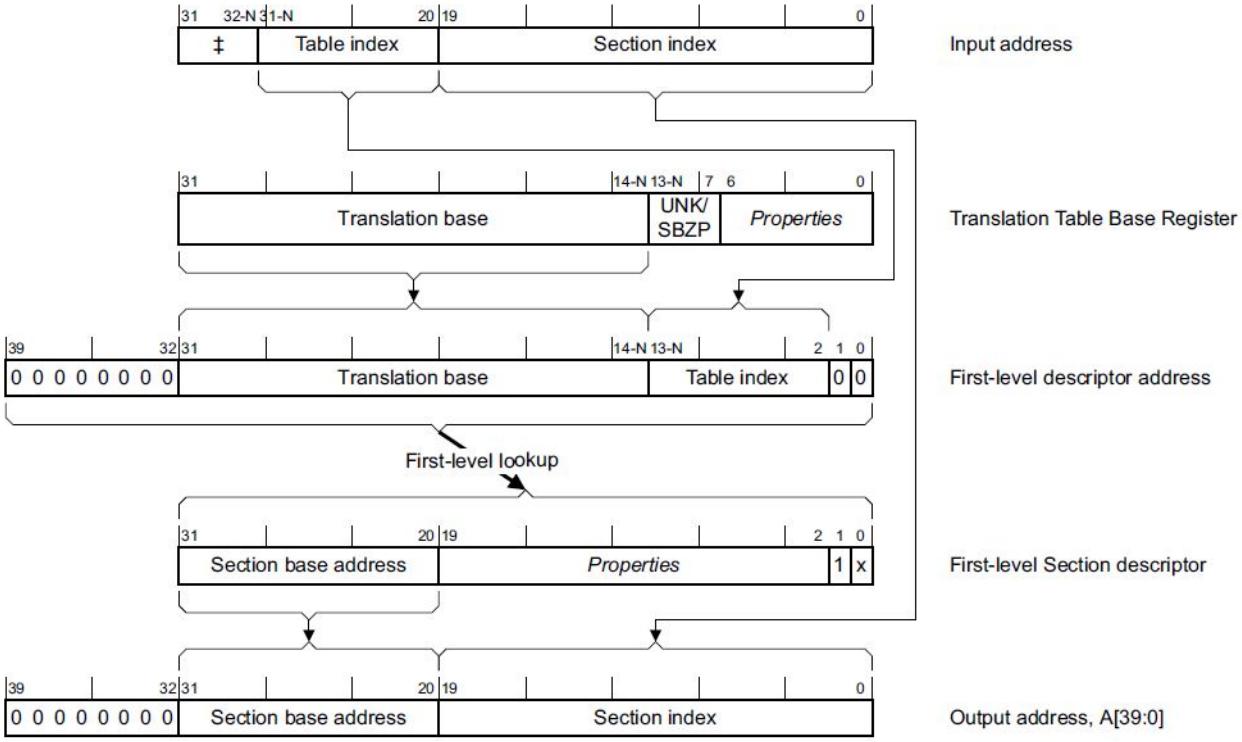


Abbildung 2: 1 MB Section Translation durch die ARM CPU [**?**, S. B3-1335]

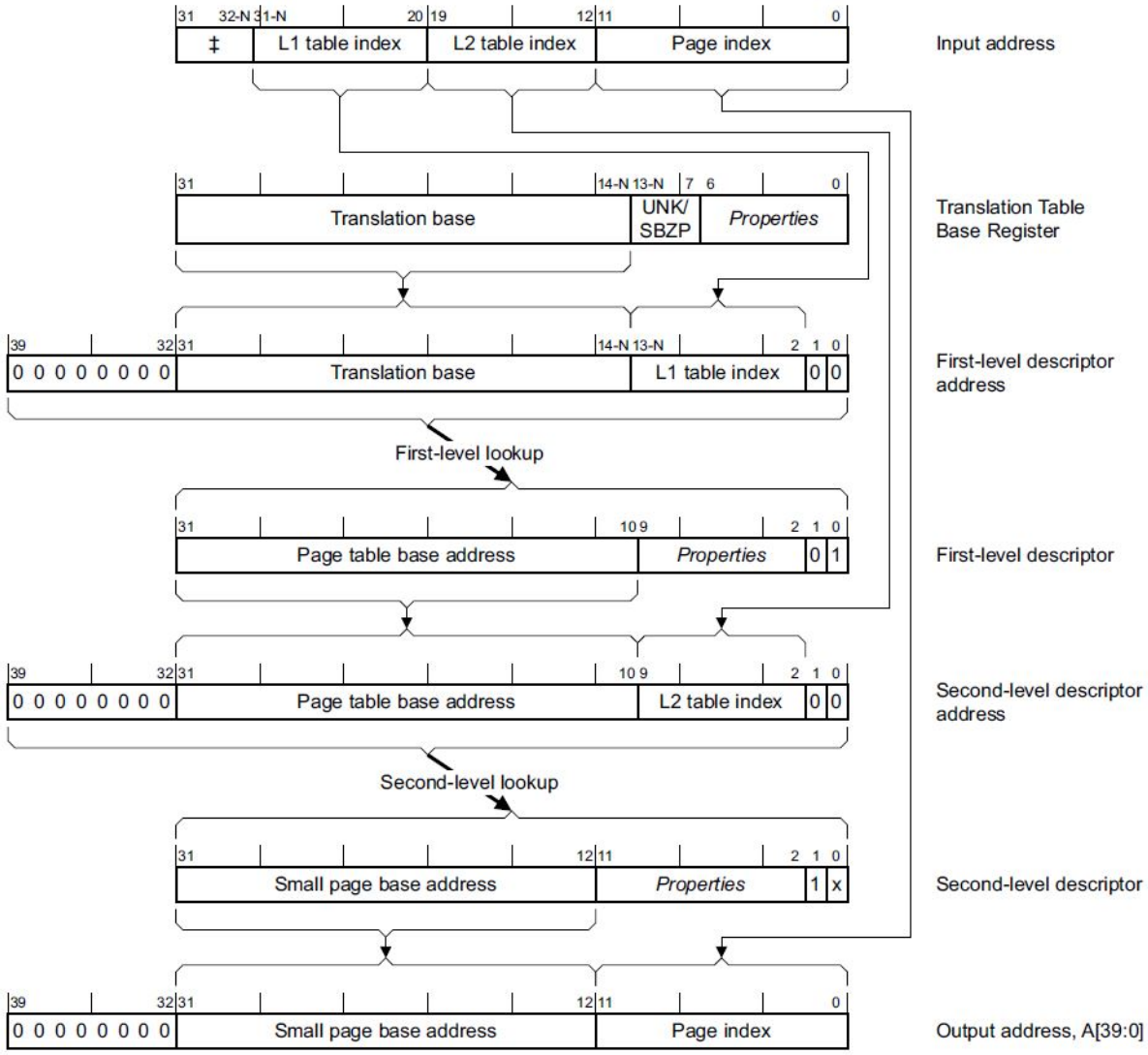


Abbildung 3: Small Page Translation durch die ARM CPU [**?**, S. B3-1337]

## 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 4 und 5 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.

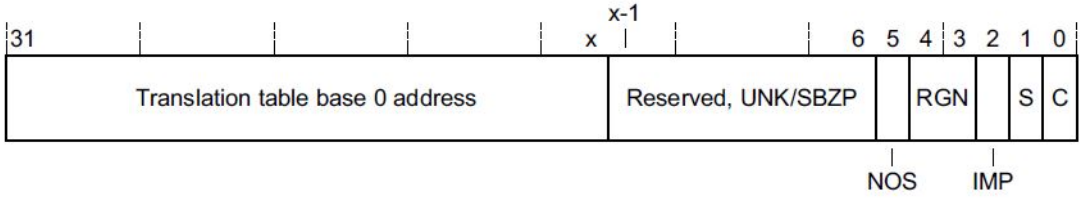


Abbildung 4: TTBR0 Format [**?**, S. B4-1726]

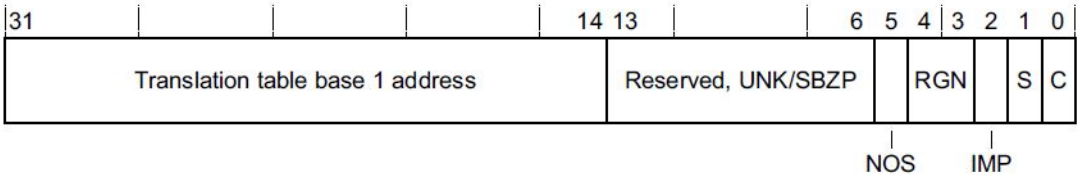


Abbildung 5: TTBR1 Format [**?**, S. B4-1730]

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

Beim Füllen der Seitentabllen sind vorgegebene Formate für die beiden Typen von Deskriptoren unbedingt zu beachten. Die Abbildungen 6 und 7 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 1 aufgeführt ist. Diese Struktur und jene des second-level Deskriptors wird bei den nachfolgenden Erläuterungen zur MMU benötigt.

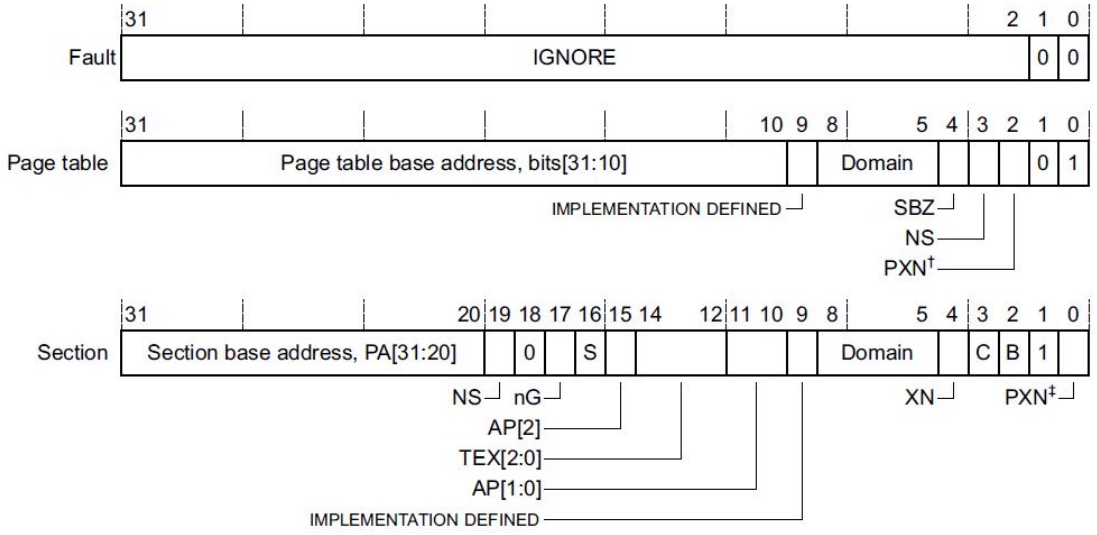


Abbildung 6: First-Level Deskriptorformate [**?**, S. B3-1326]

Listing 1: 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 vergliechen mit large pages stark verringert

Die Zusammensetzung der Struktur für second-level Deskriptoren ist in Listing 2 dargestellt.

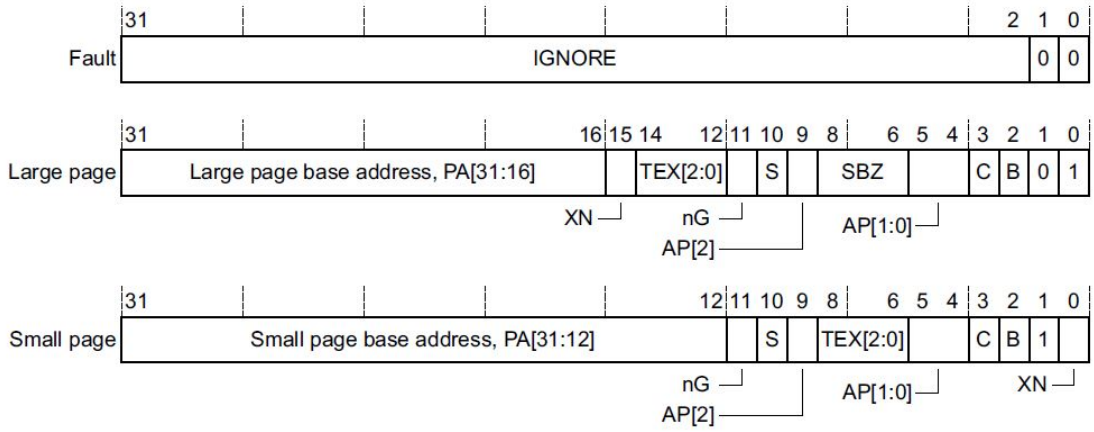


Abbildung 7: Secondt-Level Deskriptorformate [**?**, S. B3-1327]

Listing 2: 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 8 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 8 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.

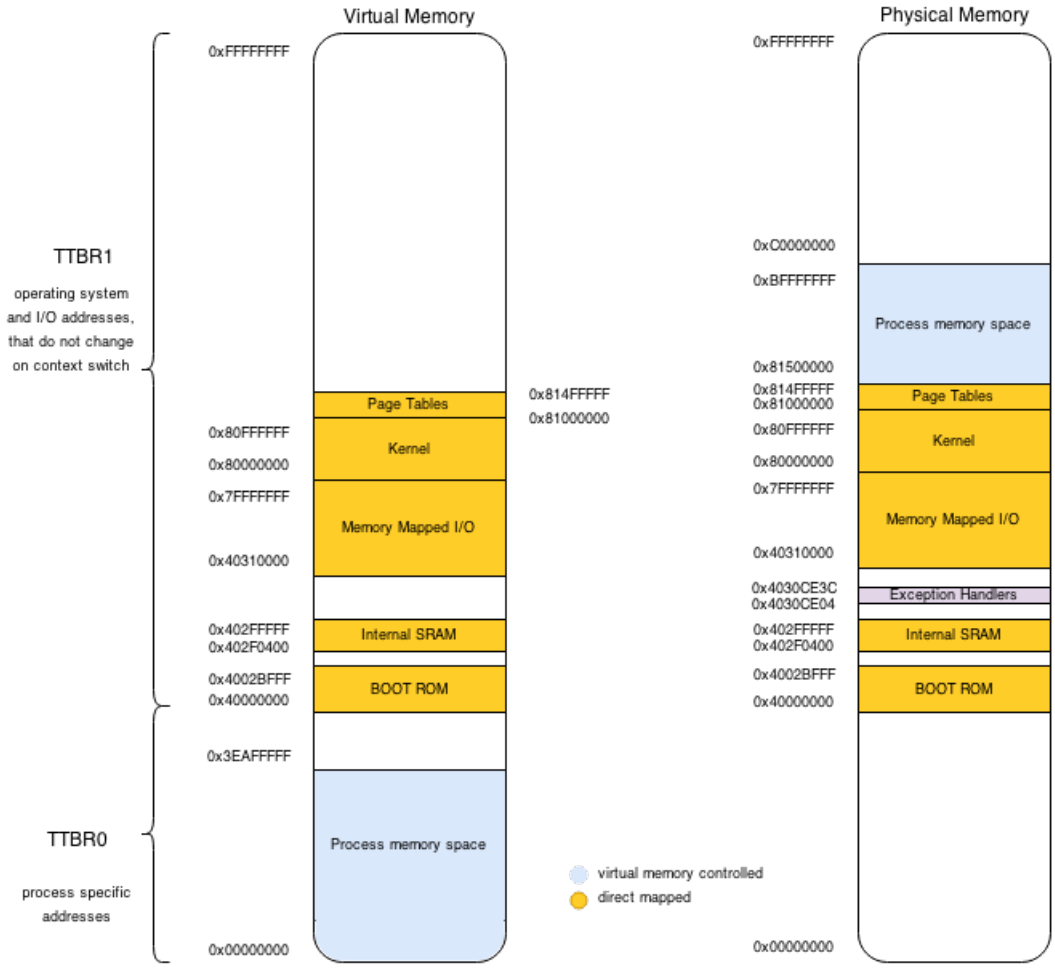


Abbildung 8: Memory Map des Betriebssystems

| 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 2: Eigenschaften der virtuellen Speicherverwaltung des OS

### 7.4.1 Speicherregionen

Das nachfolgende Listing 3 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 8 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 3: 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 3 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 3: Angelegte Speicherregionen

### 7.4.2 Master Page Table

Um das Mapping der MPT verstehen zu können, wird nochmals auf den Adresstranslationsablauf in Abbildung 2 verwiesen. Alle direkt gemappten Regions aus Tabelle 3 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 4 zeigt die praktische Ausführung des direkten Mappings in die MPT.

Listing 4: 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 9 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.

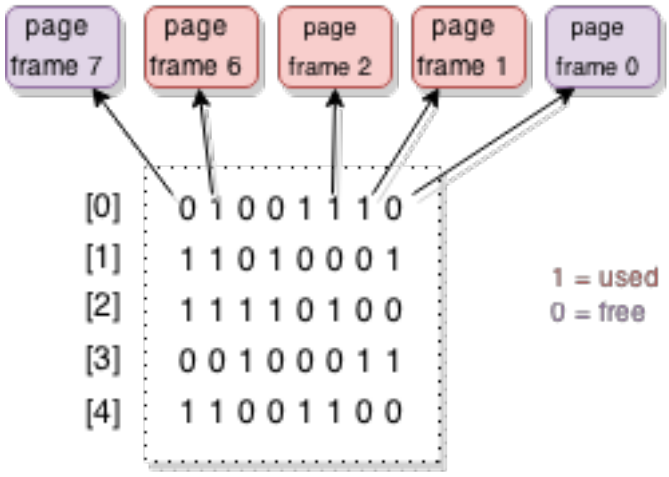


Abbildung 9: Beispiel einer Bitsmap zur Verwaltung der Page Frames

### 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 5 zeigt den kompletten Ablauf zur Aktivierung der MMU.

Listing 5: 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 6.

Listing 6: 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 Treiber

bla

## 9.1 Aufbau

bla

# 10 System API

bla

## 10.1 Aufbau

bla

# 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 10 dient zur Veranschaulichung des DMX-512 Protokolls.

TODO!!!

Abbildung 10: DMX Protokoll

Tabelle 4 beschreibt die einzeln nummerierten Markierungen aus Abbildung 10.

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

Tabelle 4: 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 $\mu$ s. Das DMX Protokoll besitzt 512 verschiedene Kanäle, wobei jeder Kanal mithilfe eines Datenbytes gesteuert wird. In Abbildung 10 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

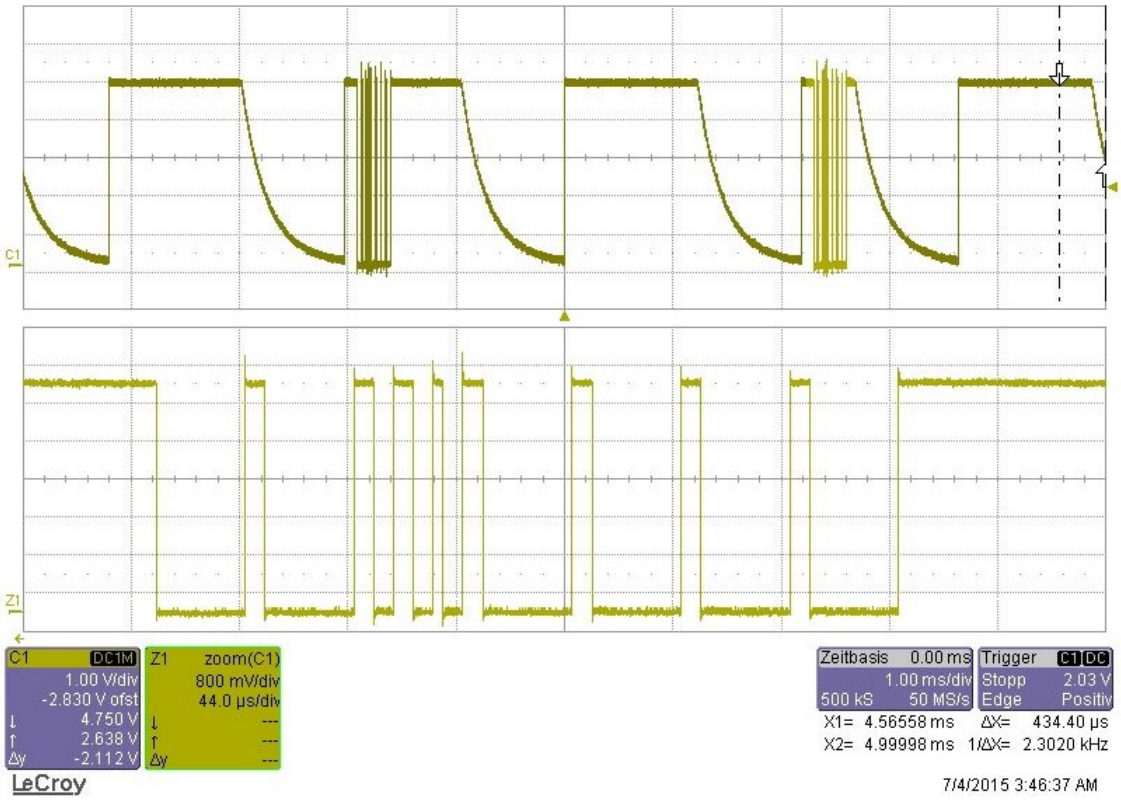


Abbildung 11: DMX Protokoll: Problem fallende Flanke

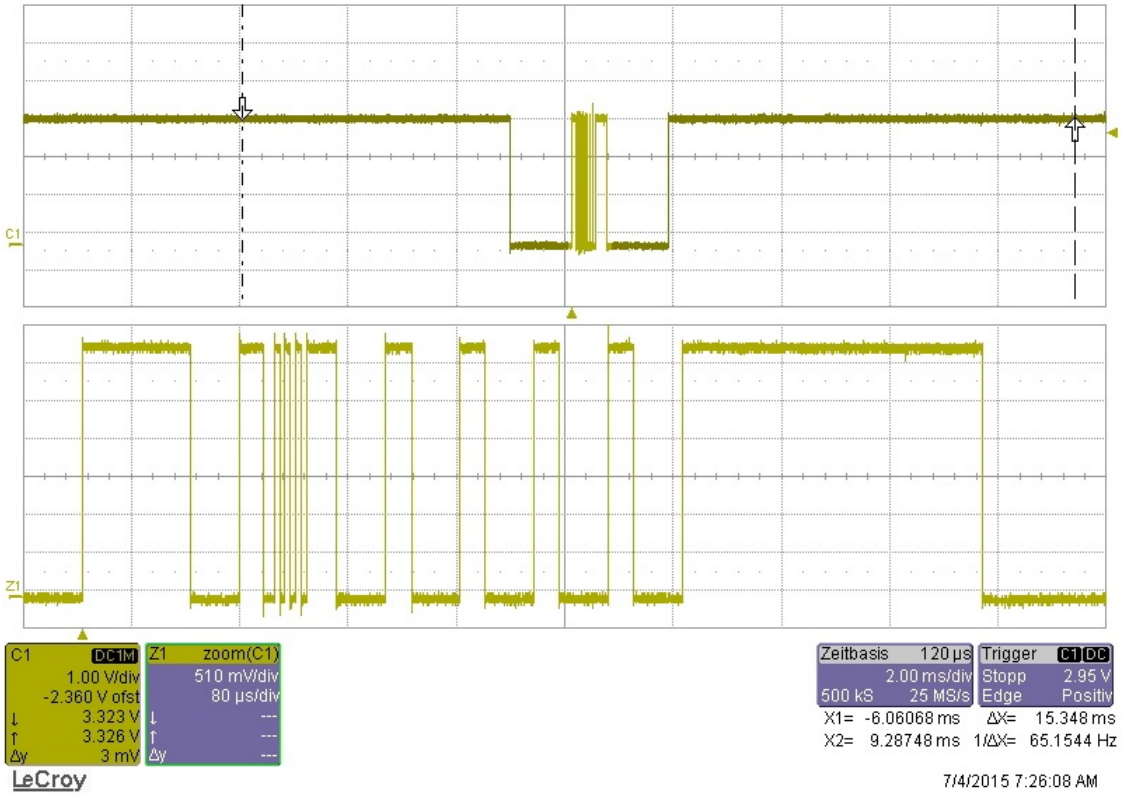


Abbildung 12: DMX Protokoll: Problem Offset Byte

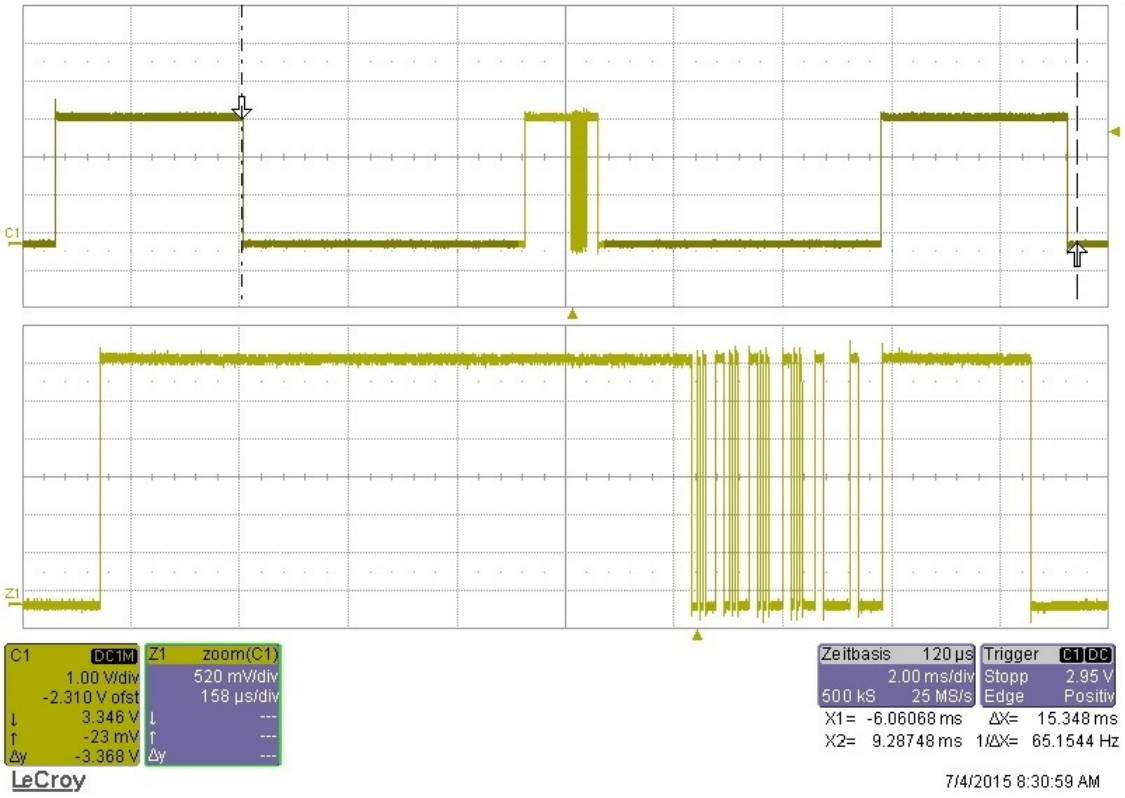


Abbildung 13: Funktionierendes DMX Protokoll

# 12 Performanz

bla

## 12.1 Aufbau

bla

# 13 Zusammenfassung

bla

[**?**]

[**?**]

## 13.1 xxx

bla

# 14 Ausblick

bla

## 14.1 Aufbau

bla