Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Martin Hepke

Linux-Implementierung für eine Zynq-Plattform zur Realisierung von SMP- und AMP-Konfigurationen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Martin Hepke

Linux-Implementierung für eine Zynq-Plattform zur Realisierung von SMP- und AMP-Konfigurationen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bernd Schwarz
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 17. November 2014

Martin Hepke

Thema der Arbeit
Linux-Implementierung für eine Zynq-Plattform zur Realisierung von SMP- und AMP-Konfigurationen

Stichworte
Linux, Embedded, SMP, AMP, Zynq, FPGA, Bildverarbeitung, MPSoC, Baremetal, Zedboard

Kurzzusammenfassung
Dieses Dokument beschreibt die Erstellung eines Linux Systems für die Zynq Plattform. Die Komponenten des und die Erstellung des Systems wird Schrittweise erklärt. Die Softwaresteuerung einer BV Kette wurde aus Baremetalcode in ein Linux Programm konvertiert. Die Änderungen und die Kommunikation mit der PL werden erläutert.
Martin Hepke

Title of the paper
Linux Implementation for a Zynq-Platform to realize SMP- and AMP-Configurations

Keywords
Linux, embedded, SMP, AMP, Zynq, FPGA, image-processing, MPSoC, Baremetal, Zedboard

Abstract
This document describes the creation of a linux system for the Zynq platform. The components and the build process of the system are explained step by step. The control application for an image processing pipeline has been converted from baremetal code to a linux application. The changes and the communication with the programmable logic are being clarified.

Inhaltsverzeichnis

1 Einleitung

Im Rahmen dieser Bachelorarbeit wird die Erstellung eines HW/SW Co-Designs für ein Fingerabdruck-Identifikationssystem auf Linux Basis beschrieben[vgl. Abbildung 2.1]. Das System soll ein Kamerainterface mit Beleuchtung parametrieren und die Bilddaten mit Verfahren für Weißausgleich, Stauchung und Geometriekorrektur korrigieren. Diese Bildverarbeitungsschritte sind in IP-Cores vorhanden, die steuernde Software ist eine Baremetal-Applikation und wird im Zuge dieser Arbeit in eine Linux-kompatible Anwendung konvertiert. Die Schwerpunkte der Aufgabenstellung sind:

- Das Erstellen eines Linux Systems für die Zynq-Plattform als Basis für Projekte für die fingerabdruckbasierte Identifikationssysteme.
- Erweiterung des Systems mit einem IP-Core zur Steuerung der GPIO Ports. Die Peripherie soll mit einer Linux Software gesteuert werden, die per Memory Mapped Zugriff auf die Register lesend/schreibend zugreift.
- Integration der Bildvorverarbeitungskette aus Vorarbeiten [Web14] auf die Linux Plattform. Die Baremetal Implementierung der Software soll in ein Linux Programm konvertiert werden, die den Ablauf der Anwendung beibehält. Die Ergebnisse der beiden Implementationen sollen verglichen werden.
- Es ist ein Konzept zur Erweiterung der Steuerungssoftware für SMP- und AMP-Betrieb zu erstellen.

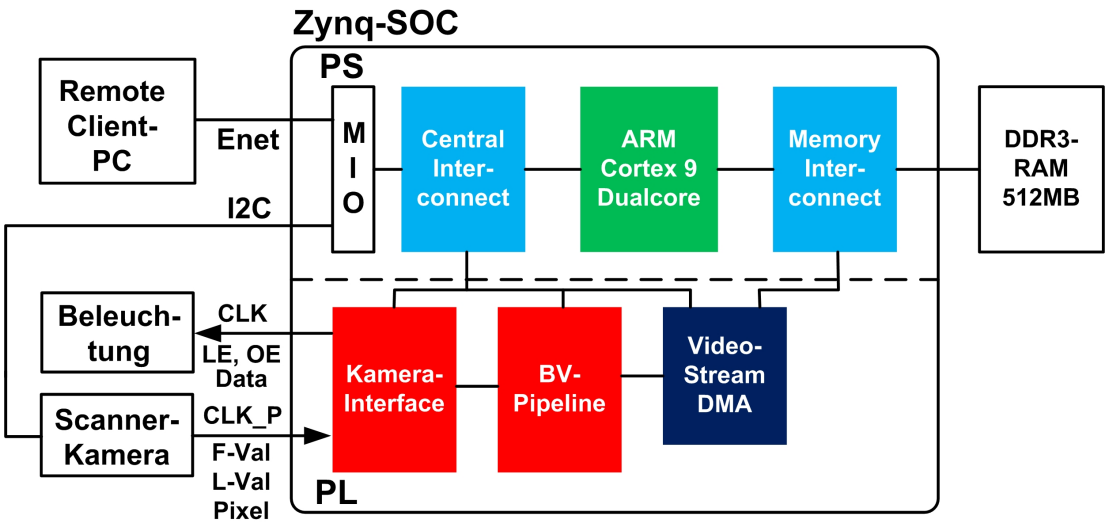

Abbildung 1.1: Bildverarbeitungspipeline eines auf Fingerabdruck basierendem Indentifikationssystems auf Basis eines MPSoC

Im Department Informatik ist im Bereich des Hard - & Software Co-Design das Thema der Bildverarbeitung Bestandteil zahlreicher Projekt- und Abschlussarbeiten, ebenso wie der Einsatz von Embedded Linux Systemen im Zusammenspiel mit Mikroprozessoren und FPGA-Ressourcen.

Embedded Linux Systeme haben, auf Plattformen bei denen Rechen- und Speicherressourcen selten knapp werden, einige Vorteile zur Implementierung von Algorithmen. Die Zynq-7000 Plattform von Xilinx bietet genügend Hardwareressourcen zum Betrieb eines Linux-Systems mit User-Applikationen. Das erlaubt die Nutzung des Linux System Call Interface, zur Verwaltung von Prozessen und Threads, sowie bestehender Programme und Libraries, die den Programmieraufwand verringern und die Struktur des Codes vereinfacht.[Hal05] Weiterhin hat man durch Programmierbare Logikbausteine die Möglichkeit rechenintensive sowie zeitkritische Anwendungen in IP-Cores auszulagern.

1.1 Kapitelübersicht

In **Kapitel 2** wird die Funktionsweise der Bildvorverarbeitungskette aus Vorarbeiten erklärt, sowie die Beweggründe zur Wahl von Betriebssystem-Entwicklung anstelle von Baremetal Implementierung der Softwarekomponente des Hardware/Software Co-Designs. Das für die Aufgabenstellung gewählte Betriebssystem und dessen Vorteile werden beleuchtet.

Kapitel 3 bietet eine Übersicht der eingesetzten Technologien. Dazu gehören die Zynq-7000 Plattform mit Peripherie wie USB, I2C und Ethernet. Grundlagen der Anbindung des FPGAs an die Prozessoren via AXI wird erklärt.

In **Kapitel 4** werden die Komponenten zur Erstellung des Linux Systems anhand des Bootvorgangs beschrieben. Die Schritte zur Installation und Konfiguration des Linux Systems für den Zynq-7000 MPSoC werden erläutert.

In **Kapitel 5** wird auf die Baremetal Software zur Steuerung der BV-Kette und Auswertung der Test-Ergebnisse eingegangen. Benötigte Änderungen und der modifizierte Quellcode zur Operation unter Linux werden beschrieben. Zudem wird die Konfiguration des laufenden Systems zur Kommunikation mit dem Remote PC beschrieben.

Kapitel 6 zeigt die Möglichkeiten der Erweiterung des Systems auf. Dabei wird auf Interprozesskommunikation sowie auf die Erfüllung von Echtzeit-Anforderungen eingegangen. Dazu werden sowohl SMP- als auch AMP-Betriebsmöglichkeiten aufgezeigt. Eine Zusammenfassung dieser Arbeit ist in **Kapitel 7** zu finden.

2 Übersicht und Konzepte

In diesem Kapitel werden die Arbeiten auf denen aufgebaut wird in einem Überblick erläutert und die Konzepte die in dem Projekt genutzt werden vorgestellt. Dazu wird auf den Nutzen eines Betriebssystems für eingebettete Systeme eingegangen und die Vorteile von Linux angesprochen.

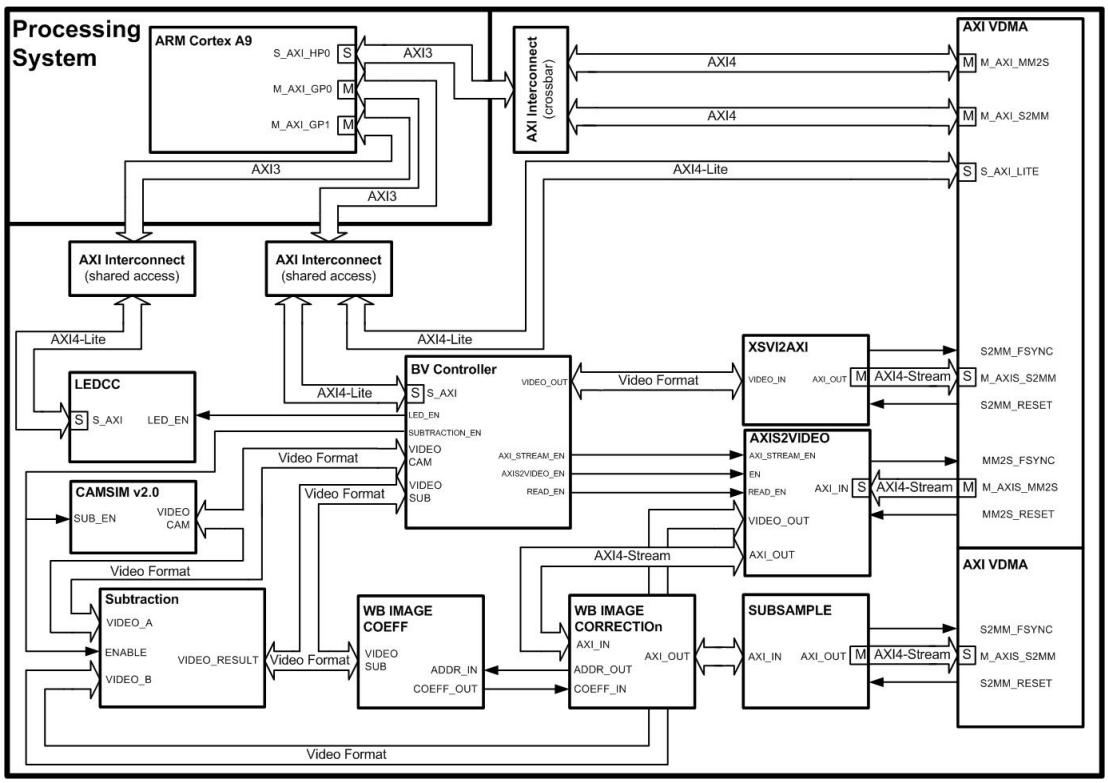

Abbildung 2.1: BV-System

2.1 Systemüberblick und HW/SW Co-Design

Das System besteht aus einer, in Programmierbarer Logik synthetisierten, BV-Kette [vgl. Abbildung 2.1] sowie einer Software zum Steuern der Control Register der PL und dem Abholen der verarbeiteten Daten. Die Steuersoftware, die als Baremetal-Anwendung vorliegt, soll zu einer Anwendung konvertiert werden, die auf einem Linuxsystem auf dem Zynq-MPSoC ausgeführt wird. Die synthetisierte User-Logic schreibt die Bilder einer Kamera oder eines Simulationsmodells per VDMA in Framebuffer die sich im DDR3-RAM befinden, bis diese zur Weiterverarbeitung abgerufen werden. Das Ergebnis des letzten Verarbeitungs-Schritts wird in einen zweiten Framebuffer geschrieben. Die Software kann die Ergebnisse nun aus dem Speicher lesen und weiter verarbeiten. Die Bildvorverarbeitungskette besteht aus einer Sequenz von parallelen Verarbeitungsschritten[vgl. Abbildung 2.2]:

Bildsubtraktion zur Neutralisierung des Umgebungs-Lichtes. Hierbei wird ein Beleuchtetes Bild aufgenommen und vom Video Stream in ein AXI konformes Stream Format konvertiert und in den Framebuffer geschrieben. Von diesem Bild wird ein NIR-Bild subtrahiert und das Ergebnis wird wieder in den Framebuffer geschrieben.

Weißabgleich(Bayer White Balance) zur Reduzierung des Schachbretteffektes. Auf das Bild wird der Bayer Filter angewendet.

Horizontale Stauchung zum verringern des Speicherbedarfs des Bildes.

Geometrische Korrektur zum korrigieren von Verzerrungen, die durch den Linsengang des Bildes über Spiegel und Prismen auftreten, werden vorgenommen um staatlichen Kriterien zur Erfassung von Fingerabdrücken gerecht zu werden. Hierbei wird ein Target-to-Source Mapping angewendet, welches durch eine Transformationsvorschrift [Web14, S.15 Gleichung 1-2] beschrieben ist.

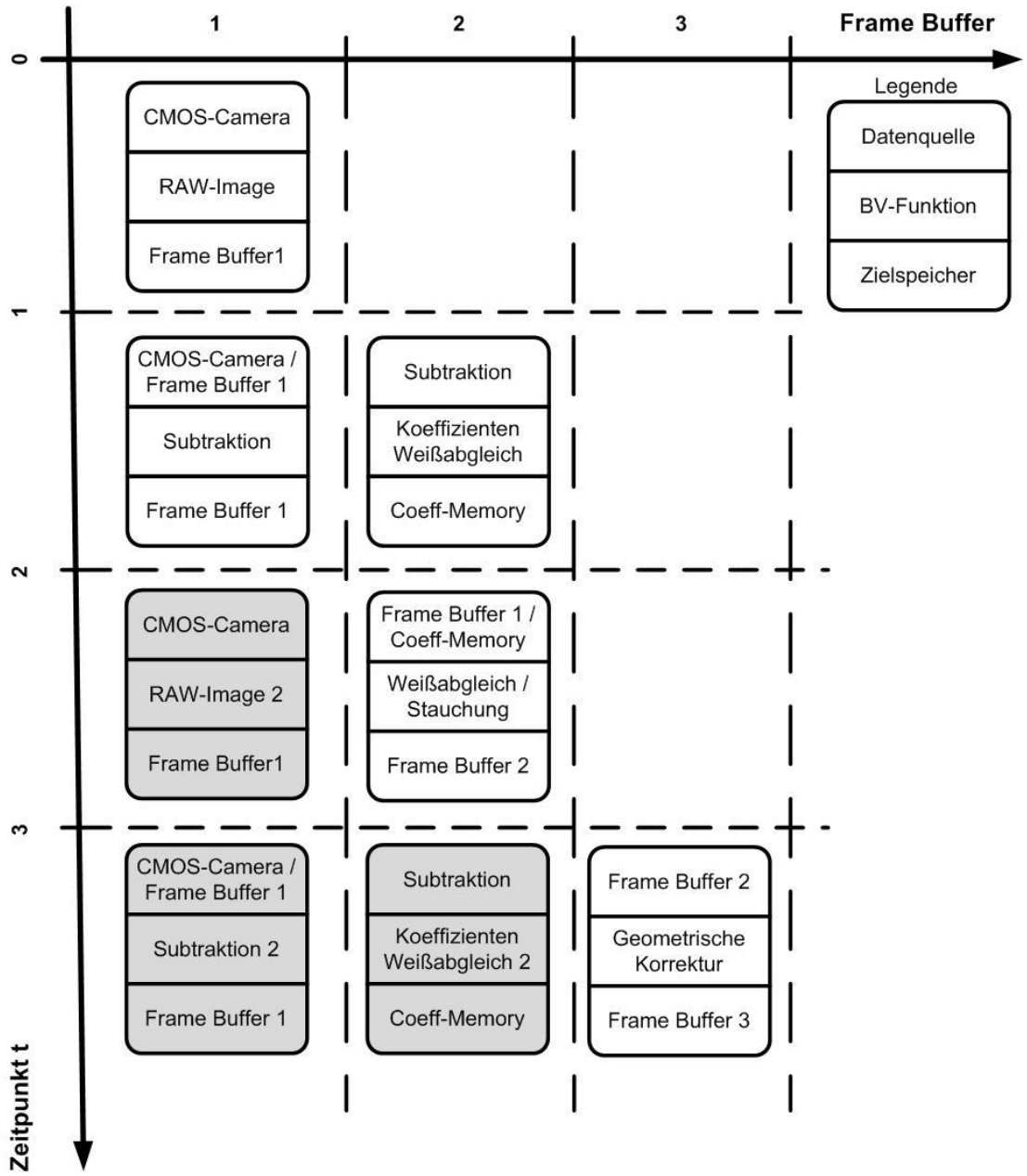

Abbildung 2.2: Phasendiagramm der BV-Kette

2.2 Embedded Linux

Bei der Embedded Softwareentwicklung entscheidet man sich ob man Baremetal oder auf Betriebssystembasis programmiert. Ein Betriebssystem bietet gegenüber der direkten Systemprogrammierung Vorteile die man in Betracht zieht. Im Falle von Linux liegen folgende Vorteile vor:

- Treiber Support für viele Devices
- Prozess und Speicherverwaltung
- Menge an vorhandenen Applikationen und Netzwerkprotokollen
- Skalierbarkeit
- Große Entwickler Gemeinde

Man spart nicht nur Zeit, sondern trägt auch zur Wartbarkeit der Software bei, wenn man bereits vorhandene Software nutzt. Beim neu Entwickeln solcher Komponenten hat man eine Quelle für mögliche Bugs, die OS zugehörige Software durch die weite Verbreitung und Support von Community und Entwicklern meist minimiert. Ein weiterer Vorteil von Betriebssystemen ist, dass die Software einfacher auf Nachfolge-Plattformen portiert werden kann, und durch Standards wie POSIX auf andere Betriebssysteme.

Wenn man sich entscheidet ein Betriebssystem zu nutzen, gibt es einige die dafür in Frage kommen[Hal05, S 380 ff.] Für dieses Projekt wurde Linux genutzt, da es sehr gut modular erweiterbar ist und man dadurch ein schlankes angepasstes System oder ein System mit vielen Features erstellen kann. Die Entwickler-Community sorgt außerdem für ein breites Spektrum an freier Software und Hilfe bei Problemstellungen.

In dieser Arbeit kommt ein Linux Kernel zum Einsatz, der auf der Version 3.0 von Kernel.org basiert und mit einigen Zynq spezifischen Features in der Form von Treibern erweitert wurde. Eine Liste der von Xilinx bereitgestellten Treiber gibt es in der Offiziellen Xilinx Wiki. [ARM12a]

3 Zynq Plattform und Prozessor Interfaces

In diesem Kapitel wird die Zynq-Plattform und die Peripherie zur Steuerung der BV-Kette beschrieben. Dabei werden die Eigenschaften vorgestellt die für das Projekt relevant sind. Für mehr Details sind für die Zynq-Plattform das Technical Reference Manual[Xil14b] und für die BV-Kette die vorangegangene Masterarbeit[And13] als Informationsquellen vorgesehen. Das Betriebssystem soll auf mindestens einem der zwei physischen Cortex A9 Prozessoren des MPSoC betrieben werden, mit der Option eine Baremetal-Anwendung auf dem zweiten Prozessor rechnen zu lassen. Der Zynq-7000 besteht aus der Physischen Peripherie (Processing System, PS) sowie dem FPGA (Programmable Logic, PL).

3.1 ARM-Cortex A9

Als Prozessor-System kommt auf dem Zynq-7000 MPSoC ein ARM-Cortex A9[ARM12a] Dual Core Prozessor mit FPU[ARM12b] und NEON[Xil14a] Extensions zum Einsatz. Die Prozessor-Kerne können mit bis zu 1GHz betrieben werden und haben je 32kb Level 1 Cache für Daten- sowie Instruktionen, ebenso wie einen gemeinsamen 512kb großen Level 2 Cache und 256kb On-Chip-Memory[Xil14b][S.722 ff.] (OCM). Die NEON Extension[ARM12c] bietet Beschleunigung für die Verarbeitung von Medien.

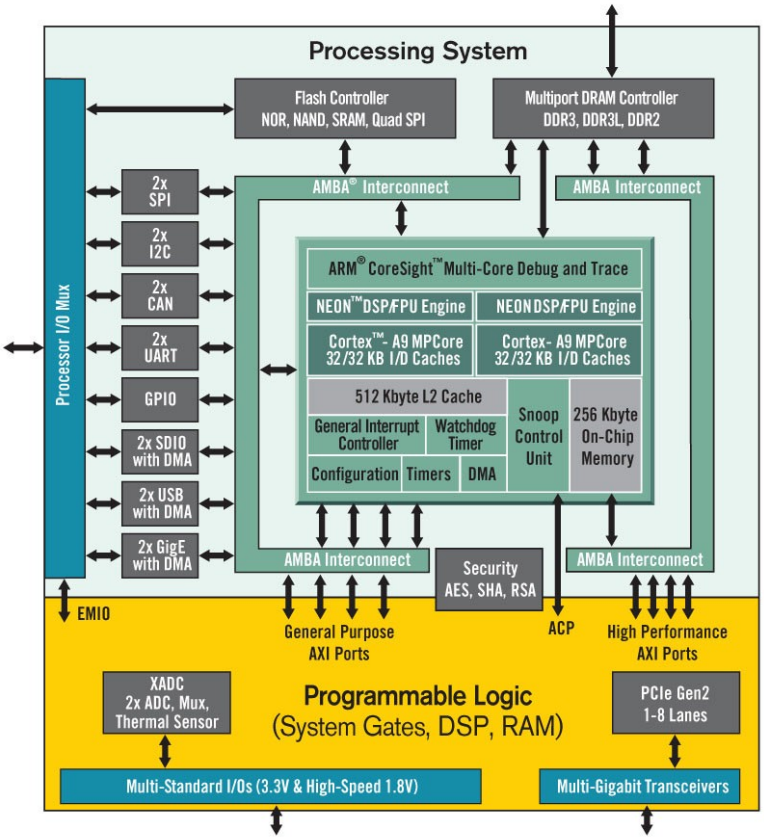

Abbildung 3.1: Zynq-7000 Processing System und Interface

3.2 7-Series FPGA und AXI-Interfaces

Die Programmable Logic der Zynq-Plattform ist abhängig vom genutzten Board ein Kintex-7 oder Artix-7 FPGA von Xilinx. Wie andere FPGAs der 7er Serie besteht auch die PL der Zynq Plattform aus Slices (LUTs FFs) bieten jedoch zusätzlich noch Block-RAM (BRAM) aus dem RAM, ROM oder FIFO Buffer erstellt werden können. Außerdem enthalten alle Series-7 FPGAs DSP48E1 Slices[Xil14a].

Das PS kommuniziert mit der PL über das Advanced eXtensible Interface (AXI) welches zur Advanced Mikrocontroller Bus Architecture (AMBA) von ARM gehört[ARM13].

AXI 3/4 ist eine adressbasierte AXI Variante, die Memory Mapped Zugriff auf die Peripherie erlaubt. Erlaubt Burst Mode bei der pro Transaktion bis zu 16(AXI3) bzw. 256(AXI4) Pakete transportiert werden.

AXI 4 Lite ist ebenso eine adressbasierte AXI Variante, die jedoch keinen Burst Mode hat und damit für den Zugriff auf Control und Status Register geeignet ist.

AXI 4 Stream ist eine adresslose AXI Variante für Datenströme wie Videos.

3.3 I^{2}C Schnittstelle zur Kamera-Parametrierung

Der bidirektionale Inter-Integrated Circuit[NXP14] (I^{2}C, I2C) Datenbus dient der Parametrierung des LF10 Vier-Finger-Scanners von Dermalog. Die Kamera enthält einen Aptina MT9J003 Kamerachip mit 10MP und parallelem Dateninterface. I^{2}C zeichnet sich durch die einfache Implementierung aus, da mit nur 2 Pins große Netzwerke von Integrierten Schaltungen angesteuert werden können. Auf dem Zynq-7000 MPSoC sind zwei physische I^{2}C-Controller vorhanden, die bei einer maximalen Datenübertragungsrate von 400Kb/s (FastMode) im Master- und Slavebetrieb arbeiten. Bei Bedarf können weitere I^{2}C-Controller in der PL erstellt werden.

3.4 Ethernet Schnittstelle zur Kommunikation mit Client-PC

Der Zynq-7000 MPSoC verfügt über zwei IEEE 802.3-2008-kompatible Gigabit Ethernet Controller mit 10/100/1000 MBit/s Übertragungsgeschwindigkeit in Full- oder Half-Duplex Betrieb. Mit diesem Interface soll der Datenaustausch über das Netzwerk per TCP/IP realisiert werden. Dadurch wird Remote-Konfiguration und Datenübertragung zwischen Client und Entwicklungs-Plattform ermöglicht. Das Ziel ist die von der BV-Kette verarbeiteten Bilder an einen Remotecomputer zur Auswertung zu übertragen.

3.5 Die Entwicklungsplattform: Zedboard

Das Zedboard ist ein Kooperationsprodukt von Xilinx, Avnet und Digilent Inc. Das Board implementiert die 7020-Variante der Zynq-7000 Plattform.

- SD-Karten Slot
- Micro USB Ports: UART, OTG, JTAG
- 10/100/1000MBit/s Ethernet Port (RJ45)

- 5 PMODs
- GPIO (LEDs, Switches, Buttons)
- 4 3,5mm Audio Jacks
- Video Ports (HDMI, VGA)

Diese Version des Zynq-MPSoC stellt eine Artix-7 basierte PL mit 13.300 Slices, 220 DSP48E1s und 140 BlockRAMs zur Verfügung[Xil14a]. Der Prozessor ist mit bis zu 667MHz taktbar und besitzt die FPU und NEON Processing Extensions. Zwei Oszillatoren für PS (33.333 MHz) und PL (100MHz) ermöglichen es die Komponenten mit voneinander unabhängigen Takt zu betreiben. Als Speicher stehen 512MB DDR3-RAM[ASS10] zur Verfügung. Neben der Möglichkeit für industrielle Entwicklung dient das Zedboard durch den Internetauftritt[www.zedboard.org] auch als Community für interessierte Programmierer und als Einstiegsplattform für die Entwicklung mit dem Zynq-MPSoC.

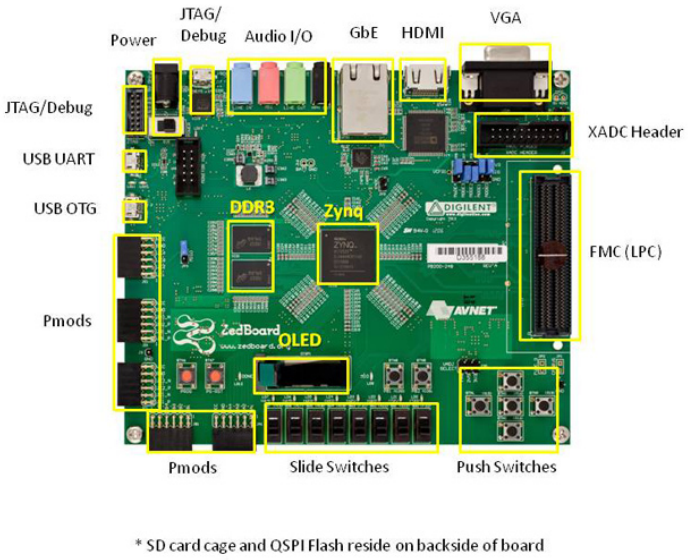

Abbildung 3.2: Zedboard Entwicklungsplattform

4 Linux Implementierung für SMP-Betrieb

In diesem Kapitel werden die Komponenten eines Embedded Linux System in einer Übersicht vorgestellt und der Prozess der Erstellung für die Zynq-Plattform des Zedboards erläutert. Als Werkzeug für den Design Workflow von PL- und Softwaredesign standen Xilinx ISE sowie Vivado SDK zur Verfügung. Das Projekt auf dem diese Arbeit aufbaut wurde mit Xilinx ISE erstellt, ebenso wie weitere Projekte in der HW/SW-Codesign Arbeitsgruppe, daher fiel die Wahl auf Xilinx ISE 14.6. Die Erstellung des Linux Systems basiert auf der internen Dokumentation [Küh13] sowie der Xilinx ISE basierten Anleitung[Dig13].

4.1 Komponenten des Embedded Linux

Das Embedded Linux System besteht aus mehreren Komponenten, die heruntergeladen, konfiguriert und kompiliert werden, um das System einzurichten:

- First Stage Bootloader
- Bitstream
- Devicetree[GH06]
- Second Stage Bootloader: "U-Boot"[Hal05, S.166 ff.]
- Das Linux System: Kernel[Jon] und Root-Filesystem[YMBYG08]

Der Boot Vorgang besteht aus drei Phasen(Stages 0-2) [CEES14, S. 409 ff.]. Auf dem Zynq Device startet zuerst der Startup-Code von einem fest verdrahteten Boot-ROM. Dieser Code bestimmt mit Jumper-Stellungen von welcher Quelle(JTAG, Quad-SPI, SD-Karte) gebootet wird.

JTAG Bootmodus Das Board wird entweder über JTAG oder USB-JTAG mit den Xilinx ISE Tools programmiert.

QSPI Bootmodus Der Startcode wird aus dem seriellen 4-Bit SPI NOR-Speicher(256Mb)[Avn12, S.9 ff.] gelesen.

SD-Card Bootmodus Laden des Startcodes von einer Class 4 oder höheren SD-Karte die am SDIO Device angeschlossen ist.

Im vorliegenden Zedboard-System wird der FSBL von der SD-Karte geladen, und der Zynq-IC geht durch die verschiedenen Phasen des Bootprozesses.

Stage-0 Beim Einschalten lädt Prozessor 0 den Code vom Boot-ROM und initialisiert die CPU. Der Startup-Code des ROMs lädt anschließend den First Stage Bootloader(FSBL) in den On Chip Memory(OCM) der ARM-CPU.

Stage-1 Der FSBL konfiguriert das Prozessorsystem (DDR, MIO und SLCR)[Xil14c, S.48] und programmiert die PL falls ein Bitstream vorliegt. Er lädt den Second Stage Bootloader(SSBL) oder den Baremetal Anwendungscode in den DDR-RAM.

Stage-2 Der Second Stage Bootloader initialisiert die Hardware weiter. Er dekomprimiert das Linux-Kernel-Image und lädt den Kernel, den Device Tree und das Root-Filesystem in den DDR-RAM und übergibt die Kontrolle über die CPU an den Linux Kernel.

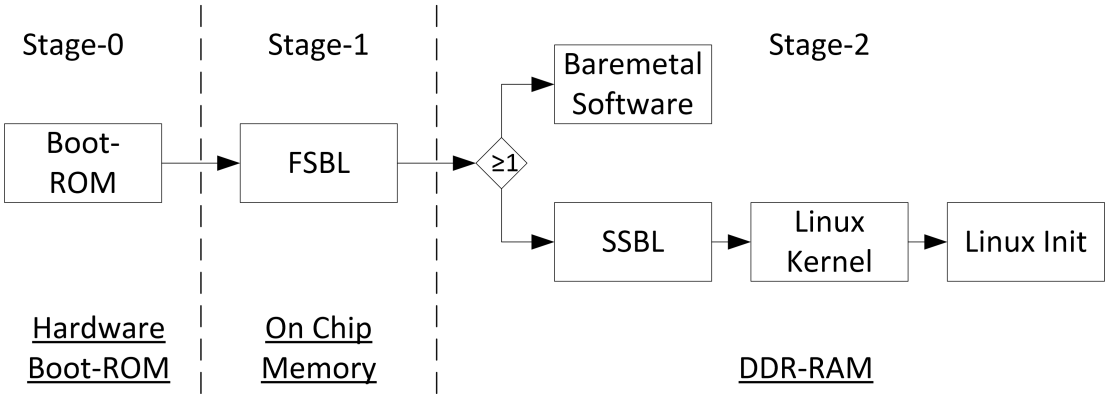

Abbildung 4.1: Ablauf des Bootvorgangs anhand der Stages 0-2

Dieser enthält die Treiber und Library-Funktionen zum Betrieb des Systems. Er hat im wesentlichen 2 Aufgaben: Hardware verwalten und das System Call Interface zur Verfügung stellen. [vgl. Tabelle 4.1]. Die System Calls erlauben dem Benutzer Prozesse, Dateien und andere Ressourcen zu erstellen und zu verwalten.

Component	Description
arch/arm/kernel/head.o	Kernel architecture-specific startup code.
init_task.o	Initial thread and task structs required by kernel.
init/built-in.o	Main kernel-initialization code. See [Hal05, Chapter 5].
usr/built-in.o	Built-in initramfs image. See [Hal05, Chapter 5].
arch/arm/kernel/built-in.o	Architecture-specific kernel code.
arch/arm/mm/built-in.o	Architecture-specific memory-management code.
arch/arm/common/built-in.o	Architecture-specific generic code. Varies by architecture.
arch/arm/mach-ixp4xx/built-in.o	Machine-specific code, usually initialization.
arch/arm/nwfpe/built-in.o	Architecture-specific floating point-emulation code.
kernel/built-in.o	Common components of the kernel itself.
mm/built-in.o	Common components of memory-management code.
ipc/built-in.o	Interprocess communications, such as SysV IPC.
security/built-in.o	Linux security components.
lib/lib.a	Archive of miscellaneous helper functions.
arch/arm/lib/lib.a	Architecture-specific common facilities. Varies by architecture.
lib/built-in.o	Common kernel helper functions.
drivers/built-in.o	All the built-in driversnot loadable modules.
sound/built-in.o	Sound drivers.
net/built-in.o	Linux networking.
.tmp_kallsyms2.o	Symbol table.

Tabelle 4.1: Übersicht des Linux Kernel Images. [Hal05, S.75-76]

Im Einstiegspunkt des Kernels wird:

- überprüft ob der Prozessor und die Architektur zugelassen sind.
- die initiale Page Table erstellt.
- die Memory Management Unit(MMU) des Prozessors aktiviert.
- begrenzte Fehlererkennung und -meldung eingerichtet.
- der eigentliche Kernel gestartet (es wird zu `start_kernel()` in der `main.c` des kernel proper gesprungen)

Die `start_kernel()` Routine ist die größte Funktion in der `main.c` des Linux Kernel Propers und übernimmt den größten Teil der Kernel-Initialisierung:

- Architektur- und Prozessorspezifische Initialisierungsroutinen
- Frühe Kernel Initialisierung und Anzeigen der Kommando Zeile

- Initialisierung der Subsysteme
- Erzeugen des ersten Kernel Threads: `kernel_init()`

Der Open Firmware Device Tree liegt dem Linux als Device Tree Blob (DTB, Flattened Device Tree) vor [dev]. Der Device Tree ist eine Datenstruktur, die dem Linux Kernel die Hardware beschreibt. Diese Baumstruktur enthält neben den Geräte-Informationen, die in Knoten untergliedert sind auch die Boot-Argumente für Linux. Den Devicetree zum genutzten System gibt es vom Board-Lieferanten. User-Logik die über den Kernel angesprochen werden soll, wird manuell in den Device Tree eingefügt.[vgl. Listing 4.12] Bei der Architektur und Prozessorinitialisierung wird er geparst damit der Kernel die Treiber der Peripherie initialisiert.

Der `kernel_init()` Thread wird zum ersten Userspace Prozess und Parent aller anderen Prozesse: `init`[vgl. Listing 4.1]. Dieser befindet sich i.d.R. unter `/sbin/init`. Damit dieser geladen wird, mountet Linux das Root-Filesystem (Root-FS), die Basis des Dateisystems. Da Dateisysteme hierarchisch organisiert sind, werden unter dem Root-FS `/` mehrere echte oder virtuelle Dateisysteme gemountet. Der Aufbau eines Dateisystems ist, für Kompatibilität mit Linux, durch den File System Hierarchy Standard (FHS) standardisiert. Das Root-FS bezieht man von der Xilinx Website[Xila], es ist aber auch möglich das Root-FS selber zu erstellen oder zu erweitern. Werden in der User-Software Shared Libraries genutzt, sollte die glibc Library des Root-FS mit einer aktuellen Version ersetzt werden[Hal05, S.177 ff.].

```
1 run_init_process( /sbin/init);
2 run_init_process( /etc/init);
3 run_init_process( /bin/init);
4 run_init_process( /bin/sh);
5
6 panic(No init found. Try passing init= option to kernel. );
```

Listing 4.1: Auszug dem Code des kernel init Threads

4.2 Erstellung eines Embedded Linux Systems für das Zedboard

Es wurde ein Linux Mint 17 System mit Mate Desktop[Proa] auf dem Entwicklungsrechner verwendet. Die auf dem System installierten Entwicklungs-Tools sind Xilinx ISE 14.6, das build-essential Paket sowie das Versionsverwaltungstool git. Das build-essential Paket enthält Compiler für C & C++ (gcc/g++) sowie Entwicklungsbibliotheken und Header Dateien und

wird mit dem Befehl `apt-get install build-essential` bezogen. Mit git werden die Quelldateien für U-Boot und das Zynq-Linux geklont, es kann mit dem Befehl `apt-get install git` installiert werden. Xilinx ISE 14.6 stellt mehrere Tools zur Verfügung[vgl. Abbildung 4.2] und steht auf der Xilinx Website als Download Bereit[Xile].

- Xilinx Platform Studio(XPS) dient als Werkzeug zur Erstellung der Intellectual Properties in der PL.
- Xilinx Software Development Kit ist eine für HW/SW-Codesign modifiziertes Eclipse IDE. Ein zentrales Tool für die Erstellung des Linux Systems ist `bootgen` welches sowohl aus XSDK, als auch dem Terminal gestartet werden kann.[CEES14, S.423]

Es wurde eine Ordnerstruktur erstellt um alle Quellen und Dateien die weiter verarbeitet werden ohne Umwege zu erreichen[Küh13, Kapitel 2.6]. Dazu empfiehlt es sich im Homeverzeichnis des angemeldeten Benutzers (`/home/<username>`) einen Ordner `XilinxLinux` sowie 3 Unterordner zu erstellen:

files bietet sich als Ordner für die erstellten Daten an, die für das Endergebnis geammelt werden.

edk ist der Ordner in dem das XPS Projekt für die User PL erstellt oder kopiert wird.

bootgen dient dem Zwischenspeichern der Dateien die zur Erstellung der BOOT.BIN gebraucht werden..

Mit git klont man die Xilinx Linux[vgl. Listing 4.2] und U-Boot Quellen[vgl. Listing 4.3] aus dem Xilinx Git Repository. Dazu wird der Befehl `git clone` in `XilinxLinux` ausgeführt. Wenn Git die Daten heruntergeladen hat sollte man im XilinxLinux Ordner 5 Unterordner haben [vgl. Listing 4.4].

```
1 $ git clone https://github.com/Xilinx/linux-xlnx.git
```

Listing 4.2: Klonen der Xilinx Linux Sourcen

```
1 $ git clone https://github.com/Xilinx/u-boot-xlnx.git
```

Listing 4.3: Klonen der U-Boot Sourcen

```
1 XilinxLinux
2 |__ bootgen
3 |__ edk
```

```
|    |__ XPS_Projekt
|__ files
|__ linux-xlnx
|__  |__ ...
|__ u-boot-xlnx
     |__ ...
```

Listing 4.4: Vorgeschlagene Ordnerstruktur für das weitere Vorgehen

Man bereitet die Entwicklung des Linux System damit vor, dass in der Konsole die Variablen für die benötigten Tools bekannt gemacht werden. Das erreicht man mit dem `export` Befehl. Dieser macht Variablen nicht nur in der aktuellen Kommandozeile bekannt, sondern auch in aus ihr aufgerufenen Programmen. Es müssen mindestens 2 Variablen bekannt gemacht werden:

```
$ export CROSS_COMPILE=arm_xilinx_linux_gnueabi-
```

Listing 4.5: Bekanntmachung des Pfades der Xilinx Toolchain

```
$ export PATH=/home/<username>/XilinxLinux/u-boot-xlnx/tools:$PATH
```

Listing 4.6: Bekanntmachung des Pfades für mkimage

Die dritte Variable wird nur im Falle einer Floating License, also einer Umgebung mit einem Lizenz-Server nötig[vgl. Listing 4.7].

```
$ export XILINXD_LICENSE_FILE=<PORT>@<IP-ADRESSE>
```

Listing 4.7: Bei floating License, IP-Adresse und Port des Lizenz Servers angeben

Diese müssen in jeder Shell aus der man die Entwicklungstools startet, ausgeführt werden. Alternativ kann man diese in die `/etc/profile` schreiben, somit geschieht das beim Start eines neuen Terminalfensters automatisch.

```
$ xlcm
```

Listing 4.8: Bei Einzelplatz Lizenz xlcm starten

Nutzer einer lokalen Lizenz können diese über den Xilinx License Configuration Manager eintragen[vgl. Listing 4.8]. Dieser kann aber erst gestartet werden wenn das Xilinx Settings Skript ausgeführt wurde, um die Xilinx Tools (`xps`, `xsdk`, `bootgen&` `xlcm`) bekannt zu machen. Je nach Systemarchitektur des PCs ist `settings32.sh` oder `settings64.sh`, aus dem Xilinx Installationsordner `.../Xilinx/14.6/ISE/`auszuführen.

```
$ source <Xilinx ISE Installationspfad>/ISE/14.6/settings64.sh
```

Beim synthetisieren der Hardware kann es aufgrund eines Bugs zu Fehlermeldungen führen. Grund hierfür scheint die Nutzung von Komma statt Punkt als Trennzeichen für reele Zahlen zu sein. Sollte dieses Verhalten beobachtet werden, sollten die Sprach-Variablen des Systems mit `unset` aufgehoben werden, bevor `xps` gestartet wird.

```
$ unset LANGUAGE
$ unset LANG
$ unset LC_NUMERIC
$ unset LC_MEASUREMENT
```

Listing 4.9: Entfernen der Sprach-Variablen zum vermeiden von Rundungsfehlern bei der Clock Berechnung

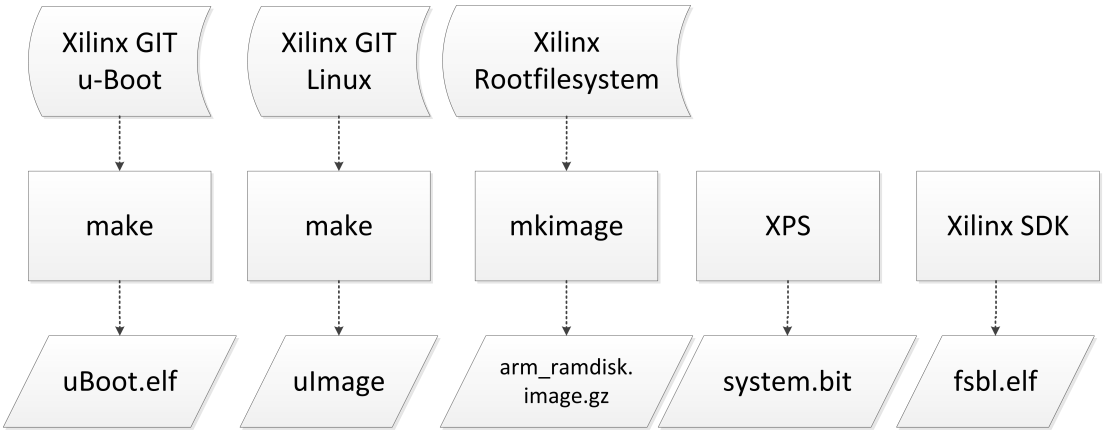

Abbildung 4.2: Übersicht Quellen und Werkzeuge zur Erstellung der Linux Komponenten

Der Second Stage Bootloader ”Das U-Boot” ist ein universeller Boot Loader für den Embedded Bereich, der für viele Plattformen verfügbar ist. Die Quelldateien gibt es im offiziellen Xilinx GIT-Repository [Xild]. Aus den heruntergeladenen Dateien wird die Konfiguration für das Board geladen und U-Boot wird mit den gewählten Einstellungen kompiliert[vgl. Listing 4.10]. Im U-Boot Verzeichnis führt man die Kommandos aus. Die erzeugte Datei `u-Boot` wird nach `bootgen` kopiert und in `u-Boot.elf` umbenannt.

```
$ cd /home/<username>/XilinxLinux/xlnx-uboot
$ make zynq_zed_config
$ make
```

Listing 4.10: Laden der Zedboard Konfiguration und Kompilieren des U-Boot Images

Das Linux Kernel Image wird aus einer von Xilinx modifizierten Version des 3.0 Kernels erstellt, der Treiber zur Verfügung stellt die noch nicht im Linux Mainline Kernel enthalten sind[Xilb]. Die Sourcen des Xilinx-Linux erhält man aus dem Xilinx GIT Repository [Xilc]. Wie bei U-Boot wird auch hier erst die Konfiguration und dann der eigentlich Kernel kompiliert. Die Architektur des Prozessorsystems wird mit `ARCH=arm` angegeben[vgl. Listing 4.11]. Die Linux Startadresse im Arbeitsspeicher ist 0x8000[vgl. Listing 4.11] und wird als Parameter übergeben. Damit wurde das Linux Kernel Image `uImage` generiert, welches nach `files` kopiert wird.

```
1 $ cd /home/<username>/XilinxLinux/xlnx-linux
2 $ make ARCH=arm xilinx_zynq_defconfig
3 $ make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

Listing 4.11: Laden der Zynq Konfiguration und Kompilieren Linux Kernels

Das Root-Filesystem wird in das Verzeichnis `files` heruntergeladen[Xila] und mit dem U-Boot Header versehen damit U-Boot damit booten kann:

```
1 $ cd /home/<username>/XilinxLinux/files
2 $ mkimage -A arm -T ramdisk -C gzip -d ramdisk.image.gz
    uramdisk.image.gz
```

Zynq | Bus Interfaces | Ports | Addresses

Instance	Base Name	Base Address	High Address
processing_system7_0's Address Map			
processing_system7_0	C_DDR_RAM_BASEADDR	0x00000000	0x1FFFFFFF
axi_vdma_1	C_BASEADDR	0x43000000	0x4300FFFF
axi_vdma_0	C_BASEADDR	0x43020000	0x4302FFFF
axi_bv_controller_0	C_BASEADDR	0x6DA00000	0x6DA0FFFF
ledccip_0	C_BASEADDR	0xB6600000	0xB660FFFF
processing_system7_0	C_UART1_BASEADDR	0xE0001000	0xE0001FFF
processing_system7_0	C_USB0_BASEADDR	0xE0002000	0xE0002FFF
processing_system7_0	C_GPIO_BASEADDR	0xE000A000	0xE000AFFF
processing_system7_0	C_ENET0_BASEADDR	0xE000B000	0xE000BFFF
processing_system7_0	C_SDIO0_BASEADDR	0xE0100000	0xE0100FFF
processing_system7_0	C_TTC0_BASEADDR	0xF8001000	0xF8001FFF

Abbildung 4.3: Anzeige der Adressen in XPS

In XPS wird das PL-Design erstellt oder ein vorhandenes Design geladen. Sobald das PL-Projekt in XPS komplettiert wurde, werden die Adressen im System Assembly View vom Reiter Addresses[vgl. Abbildung 4.3] notiert und die Devicetree Datei angepasst um die Geräte via dem UIO-Framework oder mit einem Kernel Modul anzusprechen.

Der Device Tree Blob wird mit dem Device Tree Compiler (DTC) aus einer in Text-Format

geschriebenen Device Tree Source (DTS) kompiliert. Die DTS und des Zynq Boards liegt unter `/home/<username>/XilinxLinux/linux-xlnx/arch/arm/boot/dts` und wird um die PL ergänzt[vgl. Listing 4.12]. Es wurden nur Memory Mapped Geräte verwendet, Einträge für andere Hardware sind dem Device Tree Usage Guide[dev] zu entnehmen.

```
ps7_axi_interconnect_0: amba@0 {
  #address-cells = <1>;
  #size-cells = <1>;
  compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
  ranges ;
    ...
    axi_bv_controller {
      compatible = "xlnx,axi-bv-controller-1.00.a", "generic-uio";
      reg = <0x6da00000 0x10000>;
    };

    axi_vmda0 {
      compatible = "xlnx,axi-vdma-5.04.a", "generic-uio";
      reg = <0x43020000 0x10000>;
    };

    axi_vmda1 {
      compatible = "xlnx,axi-vdma-5.04.a", "generic-uio";
      reg = <0x43000000 0x10000>;
    };

    ledccip {
      compatible = "xlnx,ledccip-1.00.a", "generic-uio";
      reg = <0xb6600000 0x10000>;
    };
} ;
```

Listing 4.12: Device Tree Benutzer-Peripherie

Dieser Auszug der DTS zeigt den AXI Interconnect Knoten mit dem Untergeordneten User-Device aus der Adress-Übersicht aus XPS. Die Eigenschaft `compatible` wird vom Betriebssystem genutzt, um die PL-Interface-Treiber an das Gerät zu binden. `generic-uio` markiert das Gerät als UIO-Device und wird vom System an eins der `uioX` Devices gemountet. Gleichzeitig sind die Geräte mit dem eigenen Namen aus dem PL Design angegeben, um einen spezifischen Treiber für das Gerät einzubinden. `reg` gibt bei Memory Mapped Geräten die Startadresse

sowie die Größe des Speicherbereichs in Hex an. [ker13] [dev] Sollen Interrupts von Geräten genutzt werden, müssen diese ebenso eingetragen werden [Bil]. Die zynq-zed.dts wird mit `dtc` zur `devicetree.dtb` kompiliert und danach in das Verzeichnis `files` kopiert. Der Device Tree Compiler liegt im Verzeichnis `xlnx-linux/scripts/dtc/`.

```
1 $ cd /home/<username>/XilinxLinux/xlnx-linux
2 $ /scripts/dtc/dtc -I dts -O dtb  o  ../files/devicetree.dtb
    /arch/arm/boot/dts/zynq-zed.dts
```

Das PL-Design wird aus XPS an das SDK exportiert. Hierbei wird das Desgin synthetisiert und der Bitstream zum Programmieren der PL erstellt. Der Bitstream befindet sich im Exportierten Verzeichnis unter `Projektname.sdk/SDK/SDK_Export/system_hw_platform/system.bit`.

Den FSBL erstellt man durch das Erzeugen eines neuen Projekts im SDK mit dem Zynq FSBL Template. Nach dem kompilieren ist der die `FSBL.elf` im `edk` Verzeichnis unter `<Projektname>.sdk/SDK/SDK_Export/fsbl/Debug`. Die Dateien `system.bit`, `fsbl.elf` und `u-boot.elf` werden in das Verzeichnis `bootgen` kopiert.

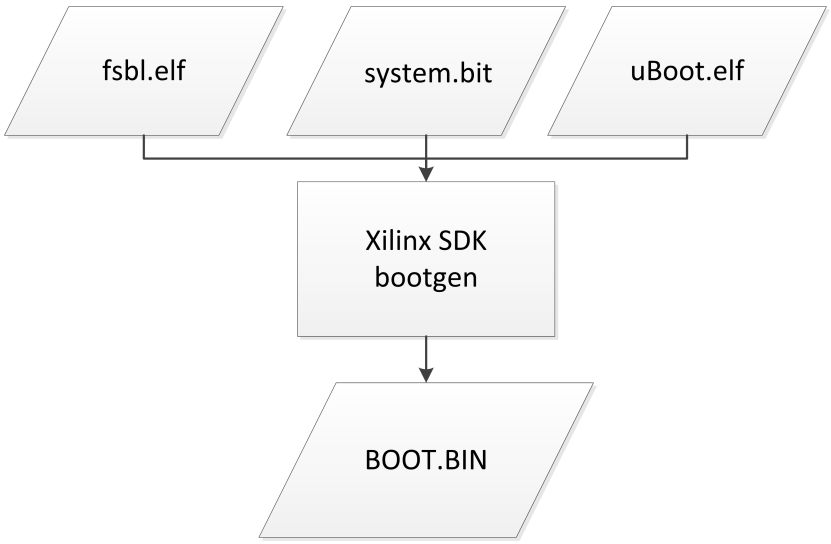

Abbildung 4.4: Erstellung der BOOT.BIN aus den Einzelkomponenten

Das SDK bietet die Möglichkeit, die `BOOT.BIN` aus der grafischen Oberfläche zu erstellen. Es ist auch möglich diesen Prozess aus der Kommandozeile zu initiieren, in dem Fall wird im

Ordner eine `.bif`-Datei erstellt und die FSBL, der Bitstream und der SSBL eintragen. Die Reihenfolge muss dabei eingehalten werden [vgl. Abbildung 4.4]:

```
the_ROM_image:
{
  [bootloader]<Dateipfad>/fsbl.elf
  <Dateipfad>/system.bit
  <Dateipfad>/u-boot.elf
}
```

Listing 4.13: bootimage.bif

Wird das grafische Tool aus dem SDK genutzt, wird die `bootimage.bif` von diesem erstellt, jedoch muss auch dort die Reihenfolge eingehalten werden. Das grafische Tool startet mit einem Klick auf "Create Image" das Tool `bootgen` mit der `bootimage.bif`. Wenn man die Konsole nutzt, gibt man folgenden Befehl ein:

```
$ cd /home/<username>/XilinxLinux/bootgen
$ bootgen -image bootimage.bif -o i BOOT.BIN
```

Listing 4.14: bootgen

Die `BOOT.BIN`, der Devicetree `devicetree.dtb`, das Linux Image `uImage` sowie das Root-Filesystem `uramdisk.image.gz` werden auf das Bootmedium kopiert [vgl. Abbildung 4.5]. Die SD-Karte muss mit dem FAT32 Dateisystem mit gesetztem BOOT-Flag und dem Label "BOOT" formatiert sein. Unter Linux kann man das mit dem Tool `gparted` erreichen, unter Windows ist die Option in den Datenträgereigenschaften vorhanden. Sollte gparted auf dem System nicht vorhanden sein kann es mit `apt-get install gparted` installiert werden. Damit die Entwicklungsplattform die Software von der SD-Karte starten kann, werden Jumper auf dem Board gesetzt [vgl. Abbildung 4.6], das Board lädt den FSBL beim Start von der SD-Karte.

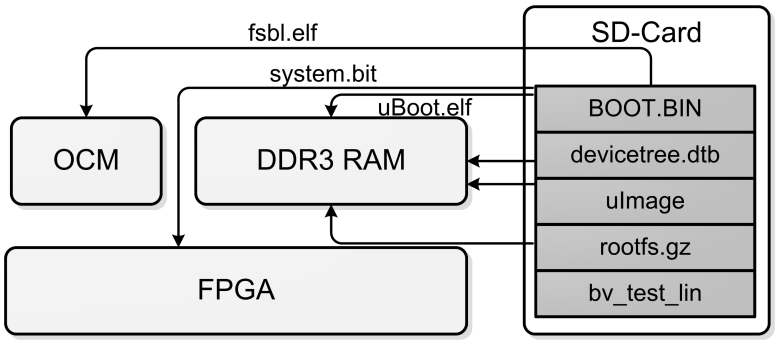

Abbildung 4.5: Inhalt der SD-Karte mit Zielspeicherbereichen im Zynq-7000

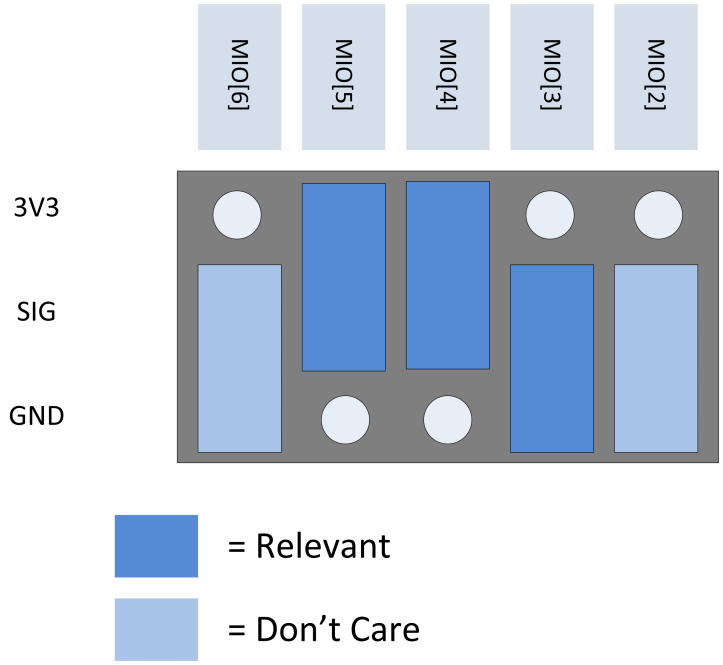

Abbildung 4.6: Stellung der Jumper für SD-Boot

4.3 LED Anwendungsbeispiel

Für einen Test wurde eine LED-Steuerung auf dem Zedboard in C implementiert. Hierfür wurde ein Tutorial[Dig13] nachempfunden, der Zugriff auf die PL wurde via `mmap()` realisiert anstatt ein Kernelmodul zu schreiben.

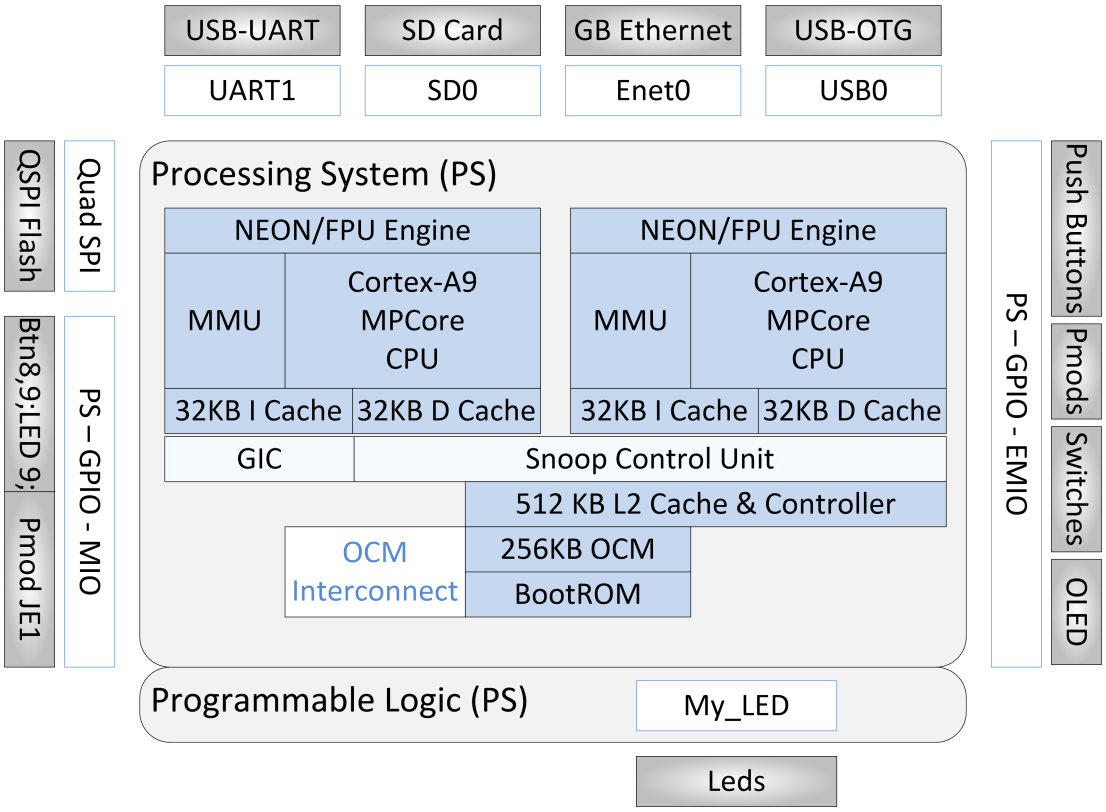

Abbildung 4.7: Architektur mit IP-Core für LED Steuerung

Es wird der Speicherbereich des LED-Steuerungs IP-Cores gemapped und daraufhin wird das Steuer-Register gelesen, sowie optional mit dem übergebenen Wert beschrieben[vgl. Listing 4.15]. Jedes Bit des Wertes repräsentiert eine LED auf dem Zedboard[vgl. Abbildung 4.8]. Bei 0 werden die LEDs alle ausgeschaltet, bei 255 werden alle LEDs aktiviert.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/mman.h>
5  #include <fcntl.h>
6
7  #define LED_BASEADDRESS 0x7e400000
8
9  int main(int argc, char *argv[])
10 {
```

```
11  int fd;
12  unsigned char* LEDVirtAddr;
13  fd = open ("/dev/mem", O_RDWR);
14  LEDVirtAddr = (unsigned char*)mmap(0, sysconf(_SC_PAGESIZE),
        PROT_READ | PROT_WRITE, MAP_SHARED, fd,
        (off_t)LED_BASEADDRESS);
15  printf("Old Value: %d\n", LEDVirtAddr[0]);
16
17  if(argc>1) {
18    int value = atoi(argv[1]);
19    LEDVirtAddr[0] = (unsigned char*)value;
20    printf("New value = %d Pagesize = %d\n", LEDVirtAddr[0],
          page_size);
21  }else{
22    printf("No Change in Value: %d\n", LEDVirtAddr[0]);
23  }
24  munmap(LEDVirtAddr, sysconf(_SC_PAGESIZE));
25  close(fd);
26  return 0;
27 }
```

Listing 4.15: myled.c Quellcode

Die Steuerungssoftware liegt auf der SD-Karte und kann nach einem Wechsel in das Verzeichnis `/mnt` via `./myled <Wert>` gestartet werden. Wird kein Wert angegeben, wird der aktuelle Wert des Registers ausgegeben[vgl. Listing 4.16].

```
1  $ cd /mnt
2  $ ./myled 45
3  Old Value: 0
4  New value = 45 Pagesize = 4096
```

Listing 4.16: Aufruf der LED Steuerung mit Konsolenausgabe

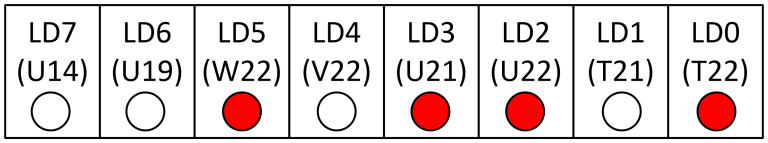

Abbildung 4.8: Status der LEDs nach der Ausführung des Codes[vgl. Listing 4.16].

5 Softwaresteuerung der Bild-Vorverarbeitungs-Kette

In diesem Kapitel wird die Softwaresteuerung der BV-Kette beschrieben, die genutzt wird um die User Logic zu steuern, sowie die Schritte die unternommen wurden, um die Software auf Linux zu portieren. Weiterhin wird die Einrichtung der Kommunikation zwischen dem Entwicklungsboards und dem Arbeitsplatz erläutert. Am Ende des Kapitels werden die Ergebnisse aus der Baremetal-Implementierung und der Linux Version verglichen.

5.1 Funktionsübersicht der Softwaresteuerung BV-Kette mit Kamera Testbench

Als Grundlage für die Softwaresteuerung dient der Quelltext der Baremetal-Implementierung aus der Masterarbeit "MPSoC-basierte Bildvorverarbeitung für ein biometrisches Identifikationssystem" [Web14]. Die Merkmalsextration ist in der PL realisiert, das C-Programm dient der Steuerung der Controlregister über Buttons und Switches auf dem Zedboard und als Schnittstelle zu den Daten für einen Client PC [vgl. Abbildung 5.1]. Dazu werden die, in die Framebuffer geschriebenen Daten, über ein Serielles Interface an einen Remote Terminal übertragen. Die über den Terminal erhaltenen Daten werden in ein .log file geschrieben und mit einem Java Programm ausgewertet. Die Auswertung erstellt eine Bitmap aus den kopierten Bildinformationen.

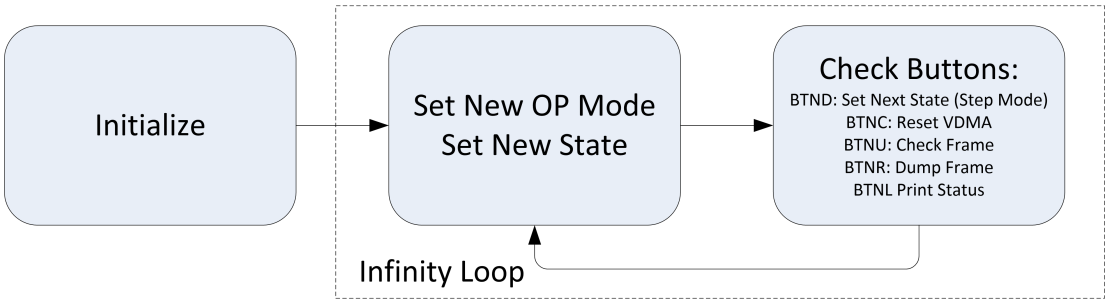

Abbildung 5.1: Vereinfachter Ablauf der Software.

Die Software initialisiert zuerst die PL-Komponenten, dazu gehört das Zurücksetzten der VDMA Controller und das Initialisieren der LED Steuerung. Danach startet eine Endlosschleife, welche die Statusregister der User-IP-Cores mit den Informationen aus den Buttons und Switches beschreibt. Die Switches SW0 und SW1 geben den Betriebsmodus an[vgl. Tabelle 5.1].

SW1	**SW0**	**Modus**
0	0	Cycle
0	1	One-Shot
1	0	Step
1	1	Passthrough

Tabelle 5.1: Übersicht der unterstützen Betriebsmodi. [Web14][S.32]

Cycle Die Bildverarbeitungskette wird zyklisch durchgegangen.

One-Shot Die BV-Kette wird einmal durchgearbeitet.

Step Die BV-Kette ändert den Zustand der BV-Controller-FSM nur bei User-Freigabe.

Passtrough Es werden keine Bildverarbeitungsschritte vorgenommen.

Neben den Switches, wurden auch die GPIO Buttons auf den Bits 15-19 mit Funktionen belegt. Mit diesen wird das Programm gesteuert:

BTNU startet die Funktion checkFrame.

BTND gibt das Freigabe Signal für den Zustandsübergang im Step Mode.

BTNL gibt den Aktuellen Status (Zustand und Betriebsmodus) in der Kommandozeile aus.

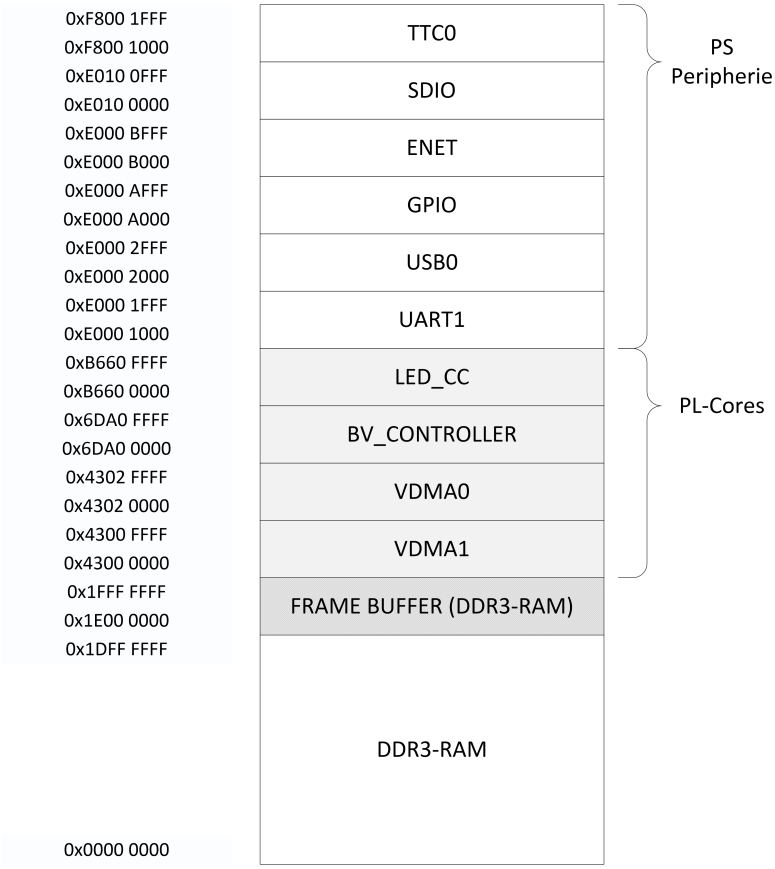

Abbildung 5.2: Übersicht der Peripheriespeicherbereiche

BTNR gibt die Daten der Framebuffer als Dump in der Konsole aus.

BTNC setzt die VDMA Controller zurück.

Die Auswertung des Dumps erfolgt durch ein Java Tool, welches die Byte Daten in Bitmap Files konvertiert.

5.2 Softwareportierung auf Linux

Um die Software in ein Linux Programm umzusetzen, wurden alle Xilinx Baremetal Funktionen und Dateitypen durch Linux-Äquivalente ersetzt. Alle Adressen, auf die von der Software zugegriffen wird, wurden in neue Headerdateien eingefügt. Die Defines wurden, mit Hilfe der Basisadressen der IP-Cores, aus XPS gesucht und die Definition von Basisadresse + Offset zu Position nach Basisadresse geändert. Um die Aussagekräftigen Defines der Baremetal

Anwendung weiter zu nutzen wird der Offset der Quelldatei durch 4 geteilt damit man die Position in der Memory Map erhält. Der Zugriff auf die Register erfolgt als Arrayobjekte.

```
#define VDMA0_BASE_ADDRESS XPAR_AXI_VDMA_0_BASEADDR
...
#define VDMA_REG32   (volatile unsigned int*)
...
#define VDMA0_MM2S_DMACR   (*(VDMA_REG32(VDMA0_BASE_ADDRESS + 0x00)))
#define VDMA0_S2MM_DMACR   (*(VDMA_REG32(VDMA0_BASE_ADDRESS + 0x30)))
```

Listing 5.1: Auszug von Defines der Adressen in Baremetal Anwendung

```
#define VDMA0_BASE_ADDRESS 0x43020000
#define VDMA_MM2S_DMACR     0
#define VDMA_MM2S_DMASR     1
```

Listing 5.2: Auszug von Defines der Adressen in Linux Anwendung

```
VDMA0_S2MM_DMACR |= (1 << 2); //reset
VDMA0_MM2S_DMACR |= (1 << 2); //reset
```

Listing 5.3: Zugriff auf VDMA Controller in Baremetal Anwendung

```
VDMA0VirtAddr[VDMA_S2MM_DMACR] |= (1 << 2); //reset
VDMA0VirtAddr[VDMA_MM2S_DMACR] |= (1 << 2); //reset
```

Listing 5.4: Zugriff auf VDMA Controller in Linux Anwendung als ArrayObjekt

Anhand der Physischen Adresse der User-Peripherie wurde mit Hilfe des `mmap()` Befehls von Linux ein Mapping auf eine Virtuelle Adresse erstellt, mit der unser Linux Programm arbeiten kann [MS99, S.123-124]. Dazu wird ein File Descriptor `fd` genutzt, der auf `/dev/mem` zeigt - den Physischen Speicher des Systems [vgl. Abbildung 5.3]. Mit den angegebenen Parametern [vgl. Tabelle 5.2 & Abbildung 5.3] von `mmap()` wird sichergestellt, dass auf die Adressen der User-Logik zugegriffen wird und auf keine von Linux genutzten Speicherbereiche.

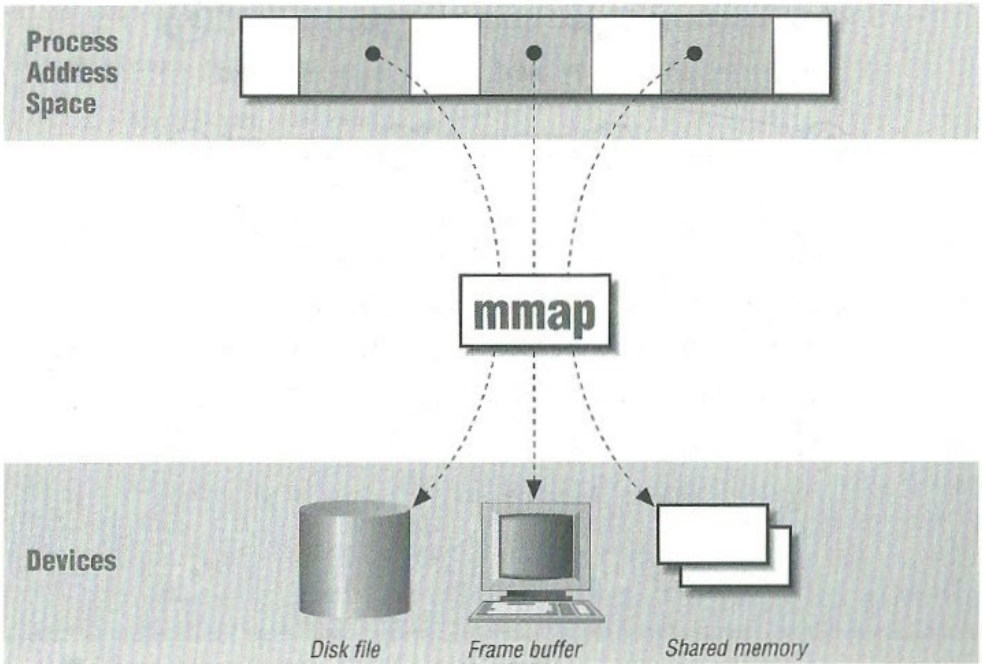

Abbildung 5.3: Zugriff auf Memory Mapped Geräte via mmap

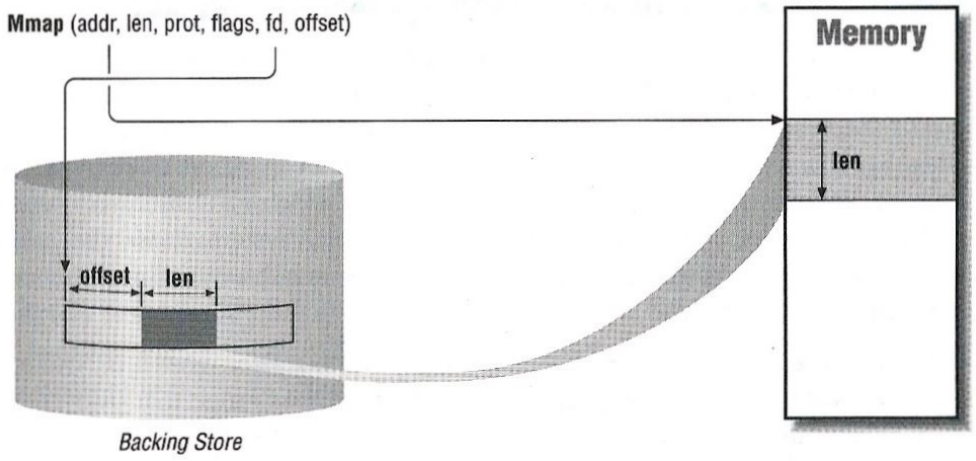

Abbildung 5.4: mmap Speicherzugriff

```
1 unsigned* VDMA0VirtAddr;
2 ...
```

```
3 VDMA0VirtAddr = (unsigned*)mmap(0, VDMA_MAP_LENGTH, PROT_READ |
    PROT_WRITE, MAP_SHARED, fd, (off_t)VDMA0_BASE_ADDRESS);
```

Listing 5.5: Mappen

Name	Type	Value	Result
addr	void*	0	Kernel wählt Zieladresse
len	size_t	VDMA_MAP_LENGTH	Größe des Mapping
prot	int	PROT_READ | PROT_WRITE	Lese- & Schreib Zugriff
flags	int	MAP_SHARED	Mapping wird mit allen anderen Prozessen geteilt die auf das Objekt zugreifen
fildes	int	fd	file Descriptor für /dev/mem
off	off_t	(off_t)VDMA0_BASE_ADDRESS	Basis Adresse von VDMA0

Tabelle 5.2: Übersicht der mmap() Parameter.

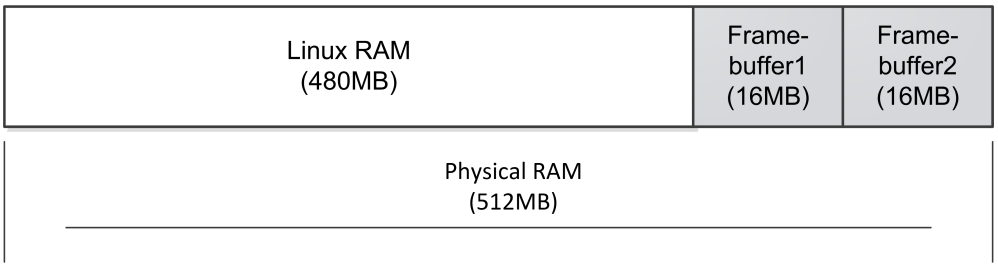

Abbildung 5.5: Anpassung des RAMs für Linux

Die Programmable Logic schreibt per VDMA direkt in die Framebuffer, die im DDR3-Speicher liegen [vgl. Abbildung 5.2]. Als Konsequenz wurde der Arbeitsspeicher des Linux Systems auf 480MB[vgl. Abbildung 5.5] begrenzt, um die Daten im RAM konsistent zu halten. Die RAW-Daten der Kamera sind ca 10MB groß, 16MB Speicher wurden pro Framebuffer reserviert, um bei Nutzung des Kamerainterfaces kompatibel zu sein. Änderungen im Devicetree [vgl. Listing 5.6] sowie der `zynq_zed.h`[vgl. Listing 5.7] sorgen für die Abgrenzung der Framebuffer vom Arbeitsspeicher, den das Betriebssystem nutzt. In der DTS werden die Bootargumente für Linux um den Ausdruck mem=480M erweitert. Die `zynq-zed.dts` befindet sich im Xilinx Linux Verzeichnis (`xlnx-linux`) unter `/arch/arm/boot/dts`.

```
1 bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk
```

```
2 mem=480M";
```

Listing 5.6: Boot Args für Linux in der DTS

Der Parameter `console=ttyPS0,115200` gibt an, dass der erste Serielle Port mit einer Baudrate von 115200 als Standardkonsole genutzt wird. Das `root=/dev/ram` Argument gibt an, das der RAM als Root-Filesystem genutzt werden soll. `rw` markiert das Root-FS zudem als Read/Write. Kernel Log-Nachrichten die auftreten bevor die traditionelle Konsole initialisiert wurde werden durch `earlyprintk` bereits beim Bootvorgang ausgegeben. Nach den Boot Parametern setzt man die Größe des SD-RAMs in der `zynq_zed.h` fest. Diese befindet sich im Ordner `u-boot-xlnx/include/configs`.

```
1 #define CONFIG_SYS_SDRAM_SIZE   (480 * 1024 * 1024)
```

Listing 5.7: Zedboard Konfigurationsdatei

5.3 Kopplung der Zynq-Plattform über Linux mit Remote-PC

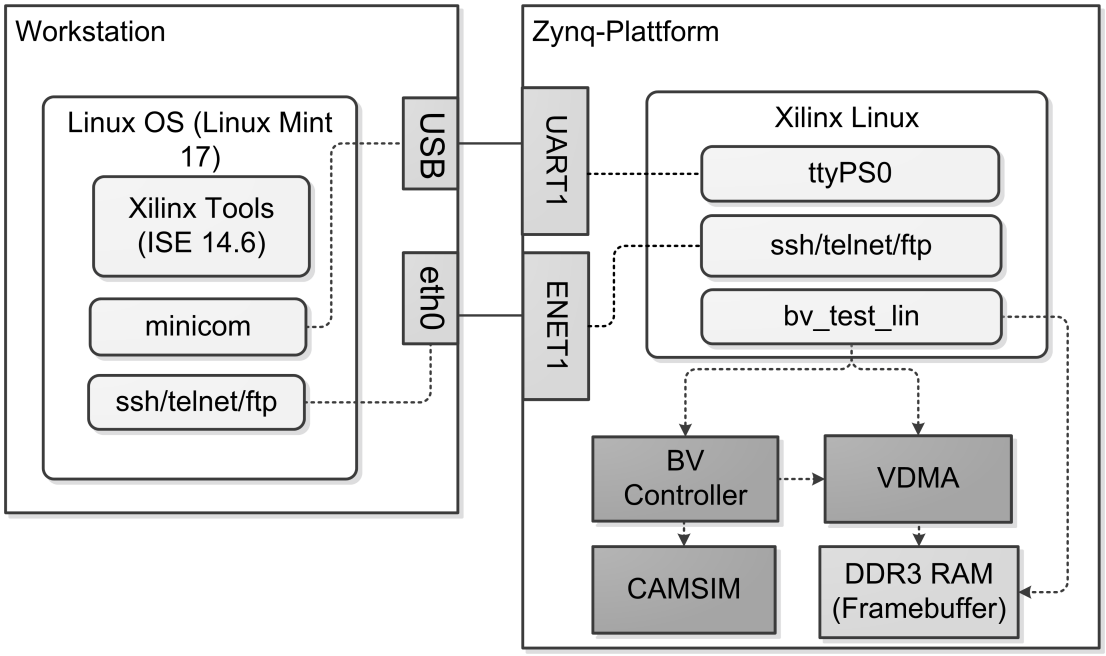

Abbildung 5.6: Verbindung zwischen Arbeitsumgebung und Zynq Entwicklungsboard

Die Kommunikation des Benutzer mit der Zynq-Plattform wird initial mit einer Terminalverbindung über den USB-UART erzeugt. Unter Linux wird dazu minicom[Joh] verwendet. Unter Windows wird ein Terminal Emulator wie TerraTerm[Prob] oder HTerm[Ham] gebraucht.

```
$ sudo minicom -s
```

Listing 5.8: minicom in Settings Menü starten

Linux In minicom5.8 wird unter dem Menüpunkt Serial Device das UART Device eintragen (ttyACM<x>). Unter bps wird die Baudrate 115200 eingestellt.

Windows Der installierte wird Terminal gestartet. Nach dem Anschließen der Hardware wird der neue COM-Port in der Auswahl des Terminal Emulators angezeigt[vgl. Abbildung 5.7]. Auch hier ist vor dem verbinden die Baudrate auf 115200 einzustellen. Das erreicht man indem man mit cancel in das Hauptmenü wechselt und unter Options->Serial Port den COM-Port konfiguriert.

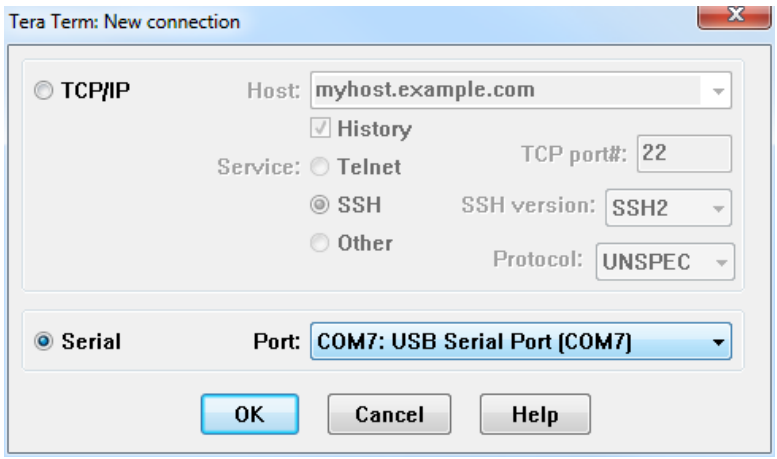

Abbildung 5.7: Terra Term Terminalemulator unter Windows

Nun kann man auf die Konsole des Zynq-Linux zugreifen, Programme starten und Konfigurationen am System vornehmen. Um Dateien zu übertragen wird das Ethernet Interface konfiguriert. Mit `ifconfig` lässt sich das Interface aktivieren und an eine feste IP binden 5.9 oder eine dynamische IP vom DHCP Server anfordern. [vgl. Listing 5.10] udhcpc(µDHCPc) ist ein für Embedded Systems angepasster DHCP Client der zum BusyBox Paket gehört

```
$ ifconfig eth0 <IP-Adresse> netmask <Subnetzmaske>
```

Listing 5.9: Ethernet Konfiguration mit fester IP

```
$ ifconfig eth0 dynamic
$ udhcpc
```

Listing 5.10: Ethernet Konfiguration für DHCP

Über den mitgelieferten FTP Dienst überträgt man die Daten und die entwickelte Software auf das Board[vgl. Listing 5.11 & Tabelle 5.3]. Der FTP-Serverdienst ist nicht durch ein Passwort geschützt, bei einer entsprechenden Abfrage wird also das leere Passwort mit Enter bestätigt. Die SD-Karte wird beim Start von Linux auf `/mnt/` gemountet. Man ändert das Remote Verzeichnis also auf `/mnt/` und das lokale Verzeichnis auf das Entwicklungsverzeichnis der Software[vgl. Listing 5.12]. Mit `put` überträgt man die zum Programmstart nötigen Komponenten. Über die telnet Konsole hat man nun Zugriff auf das Programm und kann es mit `./<Programmname>` auf der Zynq-Plattform ausführen. Ausgaben von `printf` können in der Konsole verfolgt oder mit dem Zusatz `>> <Dateiname>` in eine Log-Datei geschrieben werden, die mit dem FTP Befehl `get` zum Client übertragen werden kann.

```
$ ftp <IP-Address>
```

Listing 5.11: Mit FTP Server verbinden.

```
$ cd /mnt
$ lcd <Entwicklungsverzeichnis>
$ put <meinProgramm>
```

Listing 5.12: Übertragen einer Datei auf die SD-Karte des Zedboards

Kommando	**Ergebnis**
open <IP-Adresse>	Zu IP Adresse Verbinden
ls	Inhalt des Remote Verzeichnis anzeigen
!ls	Inhalt des lokales Verzeichnis anzeigen
cd <Verzeichnis Pfad>	Ändern des Remote Verzeichnis
lcd <Verzeichnis Pfad>	Ändern des lokalen Verzeichnis
get <Dateiname>	Datei von Server zu Client übertragen
put <Dateiname>	Datei von Client zu Server übertragen

Tabelle 5.3: Übersicht der ftp Kommandos

5.4 Auswertung

Da der Ablauf des Programms beibehalten wurde, lassen sich die Ergebnisse der zwei Softwarevarianten miteinander vergleichen. Es wurden Logfiles von Baremetal und Linux mit dem Java Konverter zu Bitmaps verarbeitet und überprüft.

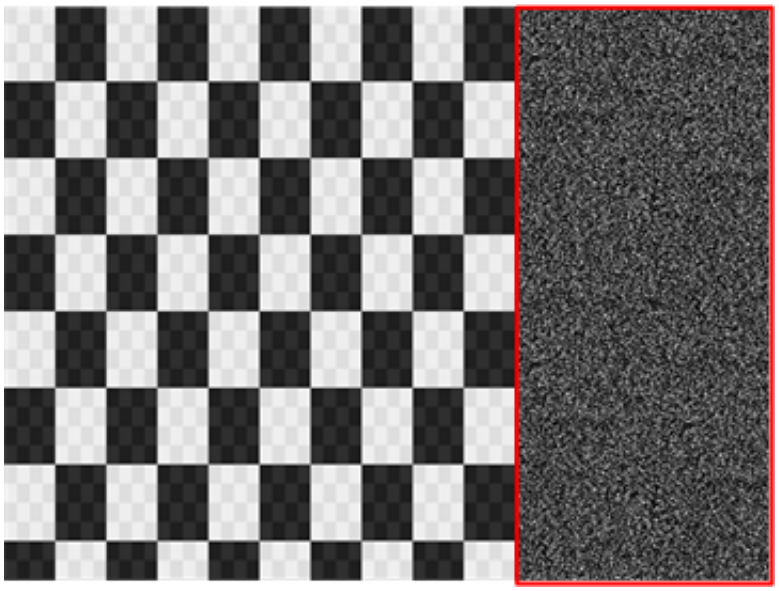

Abbildung 5.8: Ergebnis Bild unter Linux

Bis auf die Füllung der leeren Daten des gestauchten Bilds[vgl. Abbildung 5.8 & 5.9], sind die Ergebnisse identisch. Das Rauschen ausserhalb des gestauchten Bildes wurde nicht untersucht, da die Daten für die Anwendung nicht relevant sind.

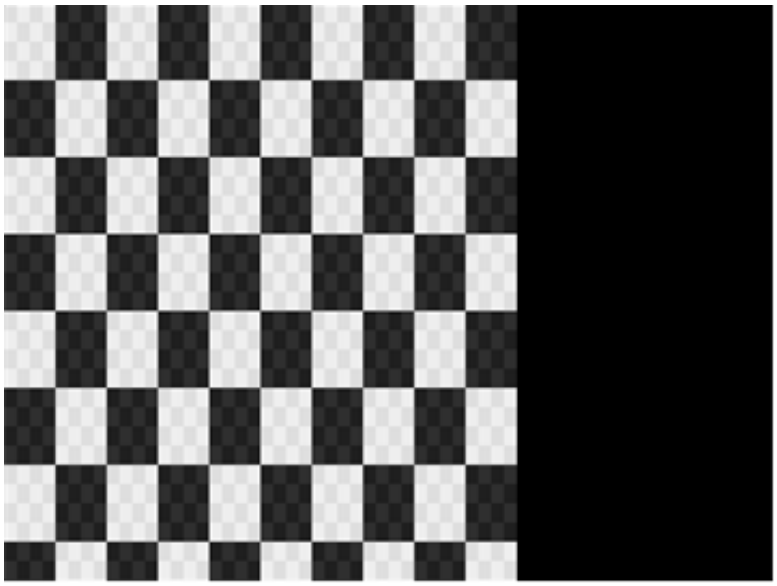

Abbildung 5.9: Referenz Bild aus Baremetal Software

6 Perspektive für Softwareweiterentwicklung

Dieses Kapitel gibt eine Feature-Übersicht von der Linux Portierung der BV-Ketten-Steurungssoftware, sowie eine Übersicht wie diese erweitert werden kann. Dabei werden die Möglichkeiten bei wachsenden Echtzeitanforderungen erläutert.

6.1 SMP Betrieb der bestehenden Software

Die migrierte Software benutzt keine Interprozesskommunikation, der Prozess arbeitet ohne Koordination mit anderen Softwarekomponenten. Es können jedoch weitere Anwendungen zur gleichen Zeit gestartet werden die unabhängig voneinander arbeiten. Das Linux System führt bereits verschiedene Dienste im Hintergrund aus (SSH, TELNET, FTP). Weiterhin können Aufgaben die von externen Programmen ausgeführt werden, wie die Codierung der Daten in Bitmaps, in Software oder PL-Logik ausgelagert werden. Librarys sowie PLs für JPEG-Codierung sind erhältlich und können in Software eingebunden werden. Wenn keine Komprimierung für die Bilddaten benötigt wird kann anhand der Bitmap-Spezifikation zudem mit Strukturen eine direkte Bitmapspeicherung erfolgen. Memory Mapped IP-Cores ohne Interrupt Verarbeitung können im laufenden Betrieb hinzugefügt werden.

Im Konzept der BV-Kette ist vorgesehen, dass die Daten an einen Client-PC gesendet werden. Ein Softwaremodul für den Client/Server Betrieb kann den Umweg über Tools wie ftp vermeiden. Auch können die Byte-Daten erst auf dem Zielgerät zu einer codierten Datei verarbeitet werden.

Die Nutzung des UIO Frameworks wurde in Kapitel 4 bereits erwähnt. Sollten künftige Funktionen Interrupts der PL verarbeiten, können diese durch das Framework in der Software behandelt werden. Dabei wird der Zugriff auf die Memory Mapped Geräte weiterhin über mmap() zu Verfügung gestellt und Funktionen für die Interrupt Verarbeitung zur Verfügung gestellt. IP-Cores die mit Interrupts arbeiten müssen in der .dts mit dem `compatible` Pa-

rameter `generic-uio` eingetragen werden[vgl. Listing 4.12]. Nach dem start von Linux erscheinen die Geräte unter `/dev/uioX` (X ist fortlaufend nummeriert) im Filesystem.

Um noch mehr Kontrolle über die User-Peripherie zu erreichen kann ein Kernel Modul entwickelt werden. Für ein produktives System sollte ein solcher vorhanden sein, zu Prototypisierung wird üblicherweise mit `mmap()` und UIO gearbeitet.

6.2 Interprozesskommunikation im SMP Betrieb

Um die Software optimal zu nutzen, empfiehlt es sich das Programm in mehrere Prozessen oder Threads zu gliedern, die das Programm um neue Funktionen erweitern. Die erwähnte Ethernet und Bildkodierungsmodule können das System mit Interprozesskommunikation bereichern. Das fertige Bild kann aus dem Framebuffer geladen werden und auf die SD-Karte geschrieben werden, damit das nächste Bild die Kette durchlaufen kann. Das fertige Bild kann wiederum mit einem Netzwerkmodul an einen Client Computer gesendet werden, der dann die Extrahierten Merkmale mit einer Datenbank vergleicht oder per Software weiter bearbeitet werden. Die Kommunikation zwischen den Prozessoren sowie die Zugehörigkeit eines Prozesses zu einem CPU-Kern verwaltet Linux im SMP Betrieb selbst, es sei denn die Prozessoraffinität wird explizit angegeben[vgl. Listing 6.1]. Damit wird der Prozess zwar an einen festen CPU Kern gebunden, jedoch ist damit das Scheduling nur indirekt beeinflusst, da nur Kontextwechsel vermieden werden. Der Zugriff auf geteilte Ressourcen und die Koordination zwischen den Threads wird vom Programmierer verwaltet und je nach Anforderungen auf unterschiedliche Wege der Interprozesskommunikation(IPC) gehandhabt. Für Kommunikation zwischen zwei entfernten Systemen kommt Message Passing zum Einsatz. Damit kann ein Client in die BV Kette eingebunden werden. Die Kommunikation zwischen Prozessen auf Linux kann mit einer Kombination aus Mutexen,Shared Memory und Condition-Variablen synchronisiert werden. Mit dem Kommando `taskset` wird ein neuer Prozess von `<applikation>` gestartet der an CPU1 gebunden ist. So können Kontextwechsel vermieden werden, um mögliche Latenz oder Echtzeitanforderungen zu erreichen. Im Falle einer softwareseitigen Erweiterung der Bildverarbeitung, kann so gesteuert werden welche Prozesse sich eine CPU teilen:

```
1 $ taskset -c 1 <application>
```

Listing 6.1: Setzen der Prozessoraffinität

Um härtere Echtzeit- und Latenzanforderungen zu erfüllen kann unter Linux der RT-Patch eingespielt werden. Dieser Patch macht einige Änderungen am Linux Kernel, um in einer

Echtzeit-Umgebung angemessen zu reagieren. Interrupt Service Routinen dürfen nicht mehr bedingungslos auf der CPU laufende Prozesse ausstechen(preempt) können, kritische Bereiche müssen nurnoch für die geblockt werden, die auf sie zugreifen könnten und die Unbounded Priority Inversion muss vermieden werden. Zum Beispiel startet ein Interrupt der Hardware einen Thread, der entweder auf einer anderen CPU gestartet wird oder wartet bis ein High Priority Thread die CPU abgibt. Wenn Linux mit dem RT-Patch nicht in der Lage ist die geforderten Deadline oder Latenzen einzuhalten, man jedoch weiter von den Linux Funktionen Gebrauch machen möchte, so ist eine AMP-Konfiguration mit Linux zu erwägen.

6.3 AMP-Betrieb mit Linux und Baremetal-Software

Ein Xilinx User Guide[Xil13a] für den Linux-Baremetal AMP Betrieb beschreibt wie man eine Linux/Baremetal Konfiguration auf dem Zynq 7000 mit Xilinx ISE umsetzt. In der AMP Konfiguration mit Linux und einer Baremetal-Anwendung startet Linux auf dem ersten Prozessorkern und startet auf dem zweiten Prozessorkern den Code der im FSBL liegt.

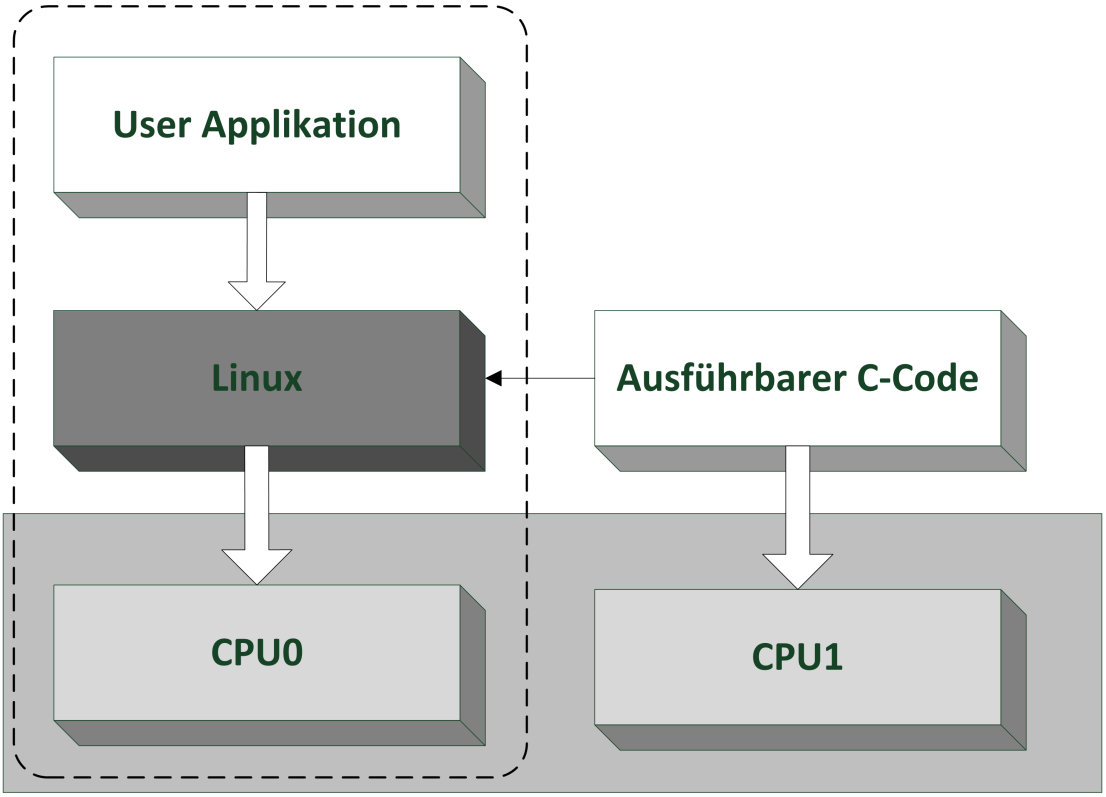

Abbildung 6.1: Schematische Darstellung der Prozessorkommunikation im AMP Betrieb mit Baremetal Code.

Der Level 2 Cache wird für die Baremetal CPU deaktiviert. MMU0 wird von Linux normal genutzt und MMU1 wird genutzt um den Speicherkontext der Baremetal-Anwendung, die nicht im Kontext des Linux Speichers läuft, festzulegen. Der Speicher der von CPU1 genutzt wird ist für Linux unbekannt. Zugriff auf den Linux Speicherbereich (Speicher und geteilte Devices wie die SCU) von CPU1 ist nicht gesperrt, aber Linux kann diese Zugriffe erkennen und entsprechend handeln. Linux und der Baremetal-Code kommunizieren über den On Chip Memory.

6.4 AMP Betrieb mit Linux und RTOS

Wenn eine höhere Abstraktionsebene sowie ein RT-Scheduling erforderlich kann man auf dem 2ten Prozessorkern ein Real Time Operating System (RTOS) laufen lassen. Es gibt eine Menge

verschiedener Echtzeitbetriebssysteme. Für die Zynq Platform gibt es eine Anleitung[Xil13b] auf Basis von Petalinux und FreeRTOS bei der Vivado als Entwicklungsumgebung eingesetzt wird. Das Desgin enthält eine Linux Demo-Applikation die Latenzdaten vom RTOS anfordert und eine RTOS Demonstrationsanwendung welche die Latenzdaten mit Hilfe des Triple Timer Counter ermittelt.

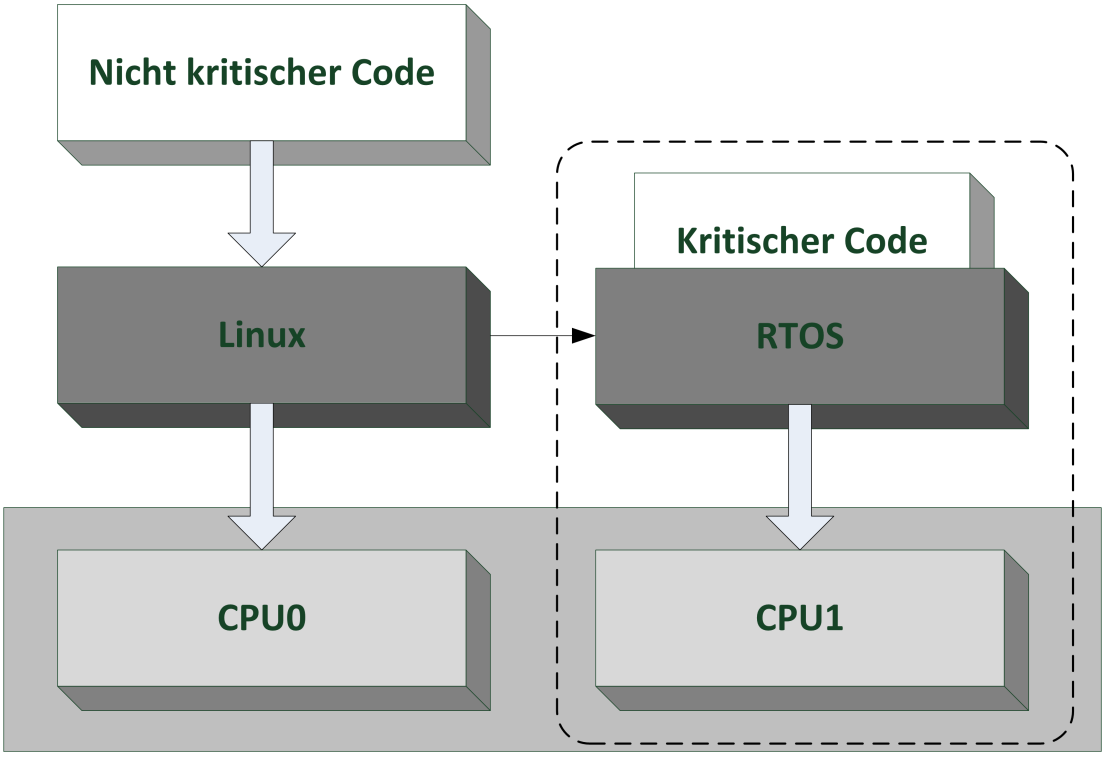

Abbildung 6.2: Schematische Darstellung der Prozessorkommunikation im AMP Betrieb mit RTOS.

In der AMP Konfiguration mit Linux und FreeRTOS läuft die FreeRTOS Firmware in einem dedizierten Speicherbereich an der Adresse 0x0000 0000. Der RTOS Kernel wird außerhalb des Linux Speicher-Kontext betrieben, Linux kann jedoch in diesen Speicherbereich, wie in jeden anderem Gerätespeicher, lesen und schreiben.

Die MMU limitiert die Effekte die das RTOS auf Linux haben kann. Es wird mit mit einer Granularität von 1MB definiert auf welche Adressen das RTOS normal schreiben und lesen kann. Den Global Interrupt Controller(GIC) teilen sich beide Systeme, die Interrupts werden

explizit zu Linux oder RTOS geroutet.

Der Linux Kernel kann die RTOS-Firmware im Betrieb beschädigen, die meisten Verletzungen werden aber durch explizite Aufrufe oder spezifische Abstürze hervorgerufen. Dies kann mit einem gutem Systemdesign und umfassenden Testing jedoch vermieden werden.

Die Anleitung von Xilinx bezieht sich auf PetaLinux und FreeRTOS, kann jedoch prinzipiell auf andere Linux- und RTOS-Distributionen angewendet werden.

7 Zusammefassung

Ziel dieser Bachelorarbeit war es, ein Linux System für den Zynq-7000 MPSoC zu erstellen. Als Betriebssystem kommt ein Xilinx Linux mit der Versionsnummer 3.13.0 zum Einsatz, welches auf dem offiziellen Kernel basiert. Es wurde von Xilinx mit Treibern die noch nicht im Mainline Kernel vorhanden sind erweitert. Durch die Nutzung von Linux soll der Programmieraufwand bei künftigen Projekten, durch die Nutzung der vorhandenen Linux Programme und Libraries verringert werden. Weiterhin war der Zugriff auf IP-Cores der Programmierbaren Logik zu errichten.

Die Erstellung des Linux Systems sollte nachvollziehbar dokumentiert werden, um bei künftigen Projekten ein angepasstes System für eine Aufgabenstellung erstellen zu können. Daher wurden die erstellten und modifizierten Komponenten anhand des Linux Bootvorgangs mit ihren Funktionen erläutert und die Techniken zur Erstellung der Einzelkomponenten beschrieben. Die auf dem Bootvorgang basierende Beschreibung erlaubt es, Konfigurationsänderungen von Komponenten Systemübergreifend zu betrachten.

Ein Beispielcode zum Zugriff auf GPIO Ports erlaubt den Zugriff auf einen IP-Core der eine LED-Steuerung implementiert. Die funktionsfähige PL-basierte Bildvorverarbeitungskette, die mit Baremetal-Software gesteuert wurde, wurde in eine Linux Basierte Softwaresteuerung eingebettet. Die Sequenz der Programmschritte wurde beibehalten und es werden weiterhin die Switches und die Buttons der Zynq-Plattform zum Steuern der PL benutzt. Die Codestruktur hingegen wurde vereinfacht, indem verschachtelte Headerdateien mit nicht genutzten Low Level Hardware Steuerfunktionen entfernt wurden. Die relevanten Adressinformationen zur Memory-Map des MPSoCs sind erhalten worden. Die BV-Kette wird mit Memory-Mapped Zugriffen auf Steuerregister geregelt, in dem der von XPS bereitgestellte Speicherbereich gemapped wurde. Anhand einer Umrechnung von Offsets zu einer positionsbasierten Adressierung werden spezifische Register innerhalb der IP-Cores beschrieben und gelesen. Kommende Linux-basierte Projekte können nach diesem Muster implementiert und konvertiert werden. Die BV-Kette ist in Linuxsoftware eingebettet und kann mit eine Prozessstruktur und Threads

erweitert werden.

Neben der SMP-Konfiguration, die für die BV-Kette benutzt wird, sollte eine AMP-Konfiguration in Betracht gezogen werden. Da die Verarbeitungskette jedoch bisher ausschließlich in einer PL-Architektur implementiert ist, wurde keine AMP-Konfiguration erstellt. Allerdings wurden Modifikationen der Systemkonfiguration vorgestellt, die auf Basis dieser Arbeit eine Richtung für angepasste Anforderungen gibt. Dazu gehören sowohl AMP-Konfigurationen als auch SMP-Techniken, die in diesem Anwendungsfall nicht genutzt werden, jedoch in vielen Fällen einen Performance-Gewinn oder erweiterte Funktionalität vorbringen. Diese Konfigurationen geben einen Überblick der Möglichkeiten der Zynq-Plattform Linux Konfiguration, die über Ressourcen schonende Single-Core Applikationen hinaus geht und zeigen die Quellen für detaillierte Informationen der selben.

Literaturverzeichnis

[And13] Andresen, Erik: *ARM-basierter MPSoC unter Linux für eine Fahrspurführung mit Bildverarbeitung.* Hamburg, Hochschule für Angewandte Wissenschaften, Masterarbeit, 2013. `http://edoc.sub.uni-hamburg.de/haw/volltexte/2013/2124/pdf/MA_Andresen.pdf`, Abruf: 2013-01-07

[CEES14] Crockett, Louise H. ; Elliot, Ross A. ; Enderwitz, Martin A. ; Stewart, Robert W.: *The Zynq Book.* 1. Aufl. Glasgow, Scotland, UK : Strathclyde Academic Media, 2014. – ISBN 978–0–9929787–0–9

[Hal05] Hallinan, Christopher: *Embedded Linux Primer - A Real-World Approach.* Boston : Pearson Education Inc., 2005. – ISBN 0–13–701783–9

[Küh13] Kühle, Michael: *Linuxsystem für SMP-Betrieb des ARM Cortex A9 Dualcore auf dem Zynq-Soc.* Hamburg, Dezember 2013

[MS99] Matthew, Neil ; Stones, Richard: *Beginning Linux Porgramming.* 2. Aufl. Birmingham, UK : Wrox Press, 1999. – ISBN 1–861002–97–1

[Web14] Weber, Rolf: *MPSoC-basierte Bildvorverarbeitung für ein biometrisches Identifikationssystem.* Hamburg, Hochschule für Angewandte Wissenschaften, Masterarbeit, 2014

[YMBYG08] Yaghmour, Karim ; Masters, Jon ; Ben-Yossef, Gilad ; Gerum, Philippe: *Building Embedded Linux Systems.* Sebastopol, CA : O'Reilly Associates, 2008. – ISBN 978–0–596–52968–0

Onlinequellen

[ARM12a] ARM: *ARM Cortex A9 Technical Reference Manual*, 2012. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf, Abruf: 2014-11-11

[ARM12b] ARM: *Cortex-A9 Floating-Point Unit*, 2012. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0408g/DDI0408G_cortex_a9_fpu_r3p0_trm.pdf, Abruf: 2014-10-21

[ARM12c] ARM: *Cortex-A9 NEON Media Processing Engine*, 2012. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0408g/DDI0408G_cortex_a9_fpu_r3p0_trm.pdf, Abruf: 2014-10-21

[ARM13] ARM: *AMBA AXI and ACE Protocol Specification*, Mai 2013. https://silver.arm.com/download/download.tm?pv=1198016, Abruf: 2013-10-27

[ASS10] ASSOCIATION, JEDEC SOLID STATE T.: *DDR3 SDRAM Standard*, Juli 2010. http://www.jedec.org/sites/default/files/docs/JESD79-3F.pdf, Abruf: 2014-10-13

[Avn12] Avnet: *Zynq Evaluation and Development Hardware User's Guide*, August 2012. http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf, Abruf: 2014-10-28

[Bil] Billauer, Eli: *Interrupt definitions in DTS (device tree) files for Xilinx Zynq-7000 / ARM*. http://billauer.co.il/blog/2012/08/irq-zynq-dts-cortex-a9/, Abruf: 2014-10-29

[dev] devicetree.org: *Device Tree Usage*. http://www.devicetree.org/Device_Tree_Usage, Abruf: 2014-10-23

[Dig13] Digilent: *Embedded Linux Hands-on Tutorial – ZedBoard*. 1, März 2013. http://www.digilentinc.com/Data/Documents/ProductDocumentation/ZedBoard_ELHoT.zip, Abruf: 2014-09-22

[GH06] Gibson, David ; Herrenschmidt, Benjamin: Device Trees Everywhere. Version: Februar 2006. http://ozlabs.org/~dgibson/papers/dtc-paper.pdf, Abruf: 2014-10-22. Oz Labs, IBM Linux Technology Center, Februar 2006. – Forschungsbericht

[Ham] Hammer, Tobias: *HTerm Project Page*. http://www.der-hammer.info/terminal/, Abruf: 2014-11-14

[Joh] Johnson, Michael K.: *minicom(1) - Linux man page*. http://linux.die.net/man/1/minicom, Abruf: 2014-11-14

[Jon] Jones, M. T.: Anatomy of the Linux kernel. https://www.ibm.com/developerworks/linux/library/l-linux-kernel/l-linux-kernel-pdf.pdf, Abruf: 2014-10-27. IBM Corporation. – Forschungsbericht

[ker13] kernel.org: *Linux and the device tree*, August 2013. https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt, Abruf: 2014-10-22

[NXP14] NXP: *I2C-bus specification and user manual*. 6, April 2014. http://www.nxp.com/documents/user_manual/UM10204.pdf, Abruf: 2014-10-22

[Proa] Project, LinuxMint: *Linux Mint Download Page*. http://www.linuxmint.com/download.php, Abruf: 2014-11-14

[Prob] Project, TeraTerm: *Terra Term Project Page*. http://en.sourceforge.jp/projects/ttssh2/, Abruf: 2014-11-14

[Xila] Xilinx: *ARM RAM Disk Linux*. http://www.wiki.xilinx.com/file/view/arm_ramdisk.image.gz/419243558/arm_ramdisk.image.gz, Abruf: 2014-10-23

[Xilb] Xilinx: *Linux Drivers*. http://www.wiki.xilinx.com/Linux+Drivers, Abruf: 2014-10-29

[Xilc] Xilinx: *The official Linux kernel from Xilinx*. https://github.com/Xilinx/linux-xlnx.git, Abruf: 2014-10-23

[Xild] Xilinx: *The official Xilinx u-boot repository*. https://github.com/Xilinx/u-boot-xlnx.git, Abruf: 2014-10-23

[Xile] Xilinx: *Xilinx ISE 14.6 Download Page*. http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_4---14_6.html, Abruf: 2014-11-14

[Xil13a] Xilinx: *Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors*, Februar 2013. http://www.xilinx.com/support/documentation/application_notes/xapp1078-amp-linux-bare-metal.pdf, Abruf: 2014-11-14

[Xil13b] Xilinx: *Zynq All Programmable SoC Linux-FreeRTOS AMP Guide*, November 2013. http://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_2/ug978-petalinux-zynq-amp.pdf, Abruf: 2014-11-14

[Xil14a] Xilinx: *7 Series FPGAs Overview*. 1.16, Oktober 2014. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, Abruf: 2014-10-22

[Xil14b] Xilinx: *Zynq-7000 All Programmable SoC - Technical Reference Manual*. 1.8.1, 2014. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, Abruf: 2014-10-21

[Xil14c] Xilinx: *Zynq-7000 All Programmable SoC Software Developers Guide*. 9.0, Juli 2014. http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf, Abruf: 2014-10-28

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang

Im Wurzelverzeichnis befindet sich die Arbeit

1 Zynq Config Files

Die Sourcedatei für den Devicetree des Zedboard ist die zynq-zed.dts. Die Knoten `axi_bv_controller`, `axi_ vmda0`, `axi_vmda1` und `ledccip` sind die vom User eingetragenen IP-Cores. Zudem wurden die Bootargs um `mem=480M` erweitert. Wenn man diese Optionen entfernt erhält man die Original Version des Files die mit XilinxLinux ausgelifert wird.

```
/*
 * Device Tree Generator version: 1.1
 *
 * (C) Copyright 2007-2013 Xilinx, Inc.
 * (C) Copyright 2007-2013 Michal Simek
 * (C) Copyright 2007-2012 PetaLogix Qld Pty Ltd
 *
 * Michal SIMEK <monstr@monstr.eu>
 *
 * CAUTION: This file is automatically generated by libgen.
 * Version: Xilinx EDK 14.5 EDK_P.58f
 *
 */

/dts-v1/;
/ {
  #address-cells = <1>;
  #size-cells = <1>;
  compatible = "xlnx,zynq-7000";
  model = "Xilinx_Zynq";
  aliases {
    ethernet0 = &ps7_ethernet_0;
    serial0 = &ps7_uart_1;
```

```
24      spi0 = &ps7_qspi_0;
25    } ;
26    chosen {
27      bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk
            mem=480M";
28      linux,stdout-path = "/amba@0/serial@e0001000";
29    } ;
30    cpus {
31      #address-cells = <1>;
32      #size-cells = <0>;
33      ps7_cortexa9_0: cpu@0 {
34        bus-handle = <&ps7_axi_interconnect_0>;
35        clock-latency = <1000>;
36        clocks = <&clkc 3>;
37        compatible = "arm,cortex-a9";
38        device_type = "cpu";
39        interrupt-handle = <&ps7_scugic_0>;
40        operating-points = <666667 1000000 333334 1000000 222223
              1000000>;
41        reg = <0x0>;
42      } ;
43      ps7_cortexa9_1: cpu@1 {
44        bus-handle = <&ps7_axi_interconnect_0>;
45        clocks = <&clkc 3>;
46        compatible = "arm,cortex-a9";
47        device_type = "cpu";
48        interrupt-handle = <&ps7_scugic_0>;
49        reg = <0x1>;
50      } ;
51    } ;
52    pmu {
53      compatible = "arm,cortex-a9-pmu";
54      interrupt-parent = <&ps7_scugic_0>;
55      interrupts = <0 5 4>, <0 6 4>;
56      reg = <0xf8891000 0x1000>, <0xf8893000 0x1000>;
57      reg-names = "cpu0", "cpu1";
58    } ;
59    ps7_ddr_0: memory@0 {
60      device_type = "memory";
61      reg = <0x0 0x20000000>;
```

```
62  } ;
63  ps7_axi_interconnect_0: amba@0 {
64    #address-cells = <1>;
65    #size-cells = <1>;
66    compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
67    ranges ;
68    ps7_afi_0: ps7-afi@f8008000 {
69      compatible = "xlnx,ps7-afi-1.00.a";
70      reg = <0xf8008000 0x1000>;
71    } ;
72    ps7_afi_1: ps7-afi@f8009000 {
73      compatible = "xlnx,ps7-afi-1.00.a";
74      reg = <0xf8009000 0x1000>;
75    } ;
76    ps7_afi_2: ps7-afi@f800a000 {
77      compatible = "xlnx,ps7-afi-1.00.a";
78      reg = <0xf800a000 0x1000>;
79    } ;
80    ps7_afi_3: ps7-afi@f800b000 {
81      compatible = "xlnx,ps7-afi-1.00.a";
82      reg = <0xf800b000 0x1000>;
83    } ;
84    ps7_ddrc_0: ps7-ddrc@f8006000 {
85      compatible = "xlnx,zynq-ddrc-1.0";
86      reg = <0xf8006000 0x1000>;
87      xlnx,has-ecc = <0x0>;
88    } ;
89    ps7_dev_cfg_0: ps7-dev-cfg@f8007000 {
90      clock-names = "ref_clk", "fclk0", "fclk1", "fclk2", "fclk3";
91      clocks = <&clkc 12>, <&clkc 15>, <&clkc 16>, <&clkc 17>,
            <&clkc 18>;
92      compatible = "xlnx,zynq-devcfg-1.0";
93      interrupt-parent = <&ps7_scugic_0>;
94      interrupts = <0 8 4>;
95      reg = <0xf8007000 0x100>;
96    } ;
97    ps7_dma_s: ps7-dma@f8003000 {
98      #dma-cells = <1>;
99      #dma-channels = <8>;
100     #dma-requests = <4>;
```

```
101       clock-names = "apb_pclk";
102       clocks = <&clkc 27>;
103       compatible = "arm,primecell", "arm,pl330";
104       interrupt-names = "abort", "dma0", "dma1", "dma2", "dma3",
105         "dma4", "dma5", "dma6", "dma7";
106       interrupt-parent = <&ps7_scugic_0>;
107       interrupts = <0 13 4>, <0 14 4>, <0 15 4>, <0 16 4>, <0 17
              4>, <0 40 4>, <0 41 4>, <0 42 4>, <0 43 4>;
108       reg = <0xf8003000 0x1000>;
109     } ;
110     ps7_ethernet_0: ps7-ethernet@e000b000 {
111       #address-cells = <1>;
112       #size-cells = <0>;
113       clock-names = "ref_clk", "aper_clk";
114       clocks = <&clkc 13>, <&clkc 30>;
115       compatible = "xlnx,ps7-ethernet-1.00.a";
116       interrupt-parent = <&ps7_scugic_0>;
117       interrupts = <0 22 4>;
118       local-mac-address = [00 0a 35 00 00 00];
119       phy-handle = <&phy0>;
120       phy-mode = "rgmii-id";
121       reg = <0xe000b000 0x1000>;
122       xlnx,eth-mode = <0x1>;
123       xlnx,has-mdio = <0x1>;
124       xlnx,ptp-enet-clock = <111111115>;
125       mdio {
126         #address-cells = <1>;
127         #size-cells = <0>;
128         phy0: phy@0 {
129           compatible = "marvell,88e1510";
130           device_type = "ethernet-phy";
131           reg = <0>;
132         } ;
133       } ;
134     } ;
135     ps7_globaltimer_0: ps7-globaltimer@f8f00200 {
136       clocks = <&clkc 4>;
137       compatible = "arm,cortex-a9-global-timer";
138       interrupt-parent = <&ps7_scugic_0>;
139       interrupts = <1 11 0x301>;
```

```
140       reg = <0xf8f00200 0x100>;
141     } ;
142     ps7_gpio_0: ps7-gpio@e000a000 {
143       #gpio-cells = <2>;
144       clocks = <&clkc 42>;
145       compatible = "xlnx,zynq-gpio-1.0";
146       emio-gpio-width = <64>;
147       gpio-controller ;
148       gpio-mask-high = <0xc0000>;
149       gpio-mask-low = <0xfe81>;
150       interrupt-parent = <&ps7_scugic_0>;
151       interrupts = <0 20 4>;
152       reg = <0xe000a000 0x1000>;
153     } ;
154     ps7_iop_bus_config_0: ps7-iop-bus-config@e0200000 {
155       compatible = "xlnx,ps7-iop-bus-config-1.00.a";
156       reg = <0xe0200000 0x1000>;
157     } ;
158     ps7_ocmc_0: ps7-ocmc@f800c000 {
159       compatible = "xlnx,zynq-ocmc-1.0";
160       interrupt-parent = <&ps7_scugic_0>;
161       interrupts = <0 3 4>;
162       reg = <0xf800c000 0x1000>;
163     } ;
164     ps7_pl310_0: ps7-pl310@f8f02000 {
165       arm,data-latency = <3 2 2>;
166       arm,tag-latency = <2 2 2>;
167       cache-level = <2>;
168       cache-unified ;
169       compatible = "arm,pl310-cache";
170       interrupt-parent = <&ps7_scugic_0>;
171       interrupts = <0 2 4>;
172       reg = <0xf8f02000 0x1000>;
173     } ;
174     ps7_qspi_0: ps7-qspi@e000d000 {
175       clock-names = "ref_clk", "aper_clk";
176       clocks = <&clkc 10>, <&clkc 43>;
177       compatible = "xlnx,zynq-qspi-1.0";
178       interrupt-parent = <&ps7_scugic_0>;
179       interrupts = <0 19 4>;
```

```
180       is-dual = <0>;
181       num-chip-select = <1>;
182       reg = <0xe000d000 0x1000>;
183       xlnx,fb-clk = <0x1>;
184       xlnx,qspi-mode = <0x0>;
185       #address-cells = <1>;
186       #size-cells = <0>;
187       flash@0 {
188         compatible = "n25q128";
189         reg = <0x0>;
190         spi-max-frequency = <50000000>;
191         #address-cells = <1>;
192         #size-cells = <1>;
193         partition@qspi-fsbl-uboot {
194           label = "qspi-fsbl-uboot";
195           reg = <0x0 0x100000>;
196         };
197         partition@qspi-linux {
198           label = "qspi-linux";
199           reg = <0x100000 0x500000>;
200         };
201         partition@qspi-device-tree {
202           label = "qspi-device-tree";
203           reg = <0x600000 0x20000>;
204         };
205         partition@qspi-rootfs {
206           label = "qspi-rootfs";
207           reg = <0x620000 0x5E0000>;
208         };
209         partition@qspi-bitstream {
210           label = "qspi-bitstream";
211           reg = <0xC00000 0x400000>;
212         };
213       };
214
215     } ;
216     ps7_qspi_linear_0: ps7-qspi-linear@fc000000 {
217       clock-names = "ref_clk", "aper_clk";
218       clocks = <&clkc 10>, <&clkc 43>;
219       compatible = "xlnx,ps7-qspi-linear-1.00.a";
```

```
220       reg = <0xfc000000 0x1000000>;
221     } ;
222     ps7_scugic_0: ps7-scugic@f8f01000 {
223       #address-cells = <2>;
224       #interrupt-cells = <3>;
225       #size-cells = <1>;
226       compatible = "arm,cortex-a9-gic", "arm,gic";
227       interrupt-controller ;
228       num_cpus = <2>;
229       num_interrupts = <96>;
230       reg = <0xf8f01000 0x1000>, <0xf8f00100 0x100>;
231     } ;
232     ps7_scutimer_0: ps7-scutimer@f8f00600 {
233       clocks = <&clkc 4>;
234       compatible = "arm,cortex-a9-twd-timer";
235       interrupt-parent = <&ps7_scugic_0>;
236       interrupts = <1 13 0x301>;
237       reg = <0xf8f00600 0x20>;
238     } ;
239     ps7_scuwdt_0: ps7-scuwdt@f8f00620 {
240       clocks = <&clkc 4>;
241       compatible = "xlnx,ps7-scuwdt-1.00.a";
242       device_type = "watchdog";
243       interrupt-parent = <&ps7_scugic_0>;
244       interrupts = <1 14 0x301>;
245       reg = <0xf8f00620 0xe0>;
246     } ;
247     ps7_sd_0: ps7-sdio@e0100000 {
248       clock-frequency = <50000000>;
249       clock-names = "clk_xin", "clk_ahb";
250       clocks = <&clkc 21>, <&clkc 32>;
251       compatible = "arasan,sdhci-8.9a";
252       interrupt-parent = <&ps7_scugic_0>;
253       interrupts = <0 24 4>;
254       reg = <0xe0100000 0x1000>;
255       xlnx,has-cd = <0x1>;
256       xlnx,has-power = <0x0>;
257       xlnx,has-wp = <0x1>;
258     } ;
259     ps7_slcr_0: ps7-slcr@f8000000 {
```

```
260       #address-cells = <1>;
261       #size-cells = <1>;
262       compatible = "xlnx,zynq-slcr", "syscon";
263       ranges ;
264       reg = <0xf8000000 0x1000>;
265       clkc: clkc@100 {
266         #clock-cells = <1>;
267         clock-output-names = "armpll", "ddrpll", "iopll",
              "cpu_6or4x", "cpu_3or2x",
268           "cpu_2x", "cpu_1x", "ddr2x", "ddr3x", "dci",
269           "lqspi", "smc", "pcap", "gem0", "gem1",
270           "fclk0", "fclk1", "fclk2", "fclk3", "can0",
271           "can1", "sdio0", "sdio1", "uart0", "uart1",
272           "spi0", "spi1", "dma", "usb0_aper", "usb1_aper",
273           "gem0_aper", "gem1_aper", "sdio0_aper", "sdio1_aper",
                 "spi0_aper",
274           "spi1_aper", "can0_aper", "can1_aper", "i2c0_aper",
                 "i2c1_aper",
275           "uart0_aper", "uart1_aper", "gpio_aper", "lqspi_aper",
                 "smc_aper",
276           "swdt", "dbg_trc", "dbg_apb";
277         compatible = "xlnx,ps7-clkc";
278         fclk-enable = <0xf>;
279         ps-clk-frequency = <33333333>;
280         reg = <0x100 0x100>;
281       } ;
282     } ;
283     ps7_ttc_0: ps7-ttc@f8001000 {
284       clocks = <&clkc 6>;
285       compatible = "cdns,ttc";
286       interrupt-names = "ttc0", "ttc1", "ttc2";
287       interrupt-parent = <&ps7_scugic_0>;
288       interrupts = <0 10 4>, <0 11 4>, <0 12 4>;
289       reg = <0xf8001000 0x1000>;
290     } ;
291     ps7_uart_1: serial@e0001000 {
292       clock-names = "ref_clk", "aper_clk";
293       clocks = <&clkc 24>, <&clkc 41>;
294       compatible = "xlnx,xuartps";
295       current-speed = <115200>;
```

```
296       device_type = "serial";
297       interrupt-parent = <&ps7_scugic_0>;
298       interrupts = <0 50 4>;
299       port-number = <0>;
300       reg = <0xe0001000 0x1000>;
301       xlnx,has-modem = <0x0>;
302     } ;
303     ps7_usb_0: ps7-usb@e0002000 {
304       clocks = <&clkc 28>;
305       compatible = "xlnx,ps7-usb-1.00.a", "xlnx,zynq-usb-1.00.a";
306       dr_mode = "host";
307       interrupt-parent = <&ps7_scugic_0>;
308       interrupts = <0 21 4>;
309       phy_type = "ulpi";
310       reg = <0xe0002000 0x1000>;
311     } ;
312     ps7_xadc: ps7-xadc@f8007100 {
313       clocks = <&clkc 12>;
314       compatible = "xlnx,zynq-xadc-1.00.a";
315       interrupt-parent = <&ps7_scugic_0>;
316       interrupts = <0 7 4>;
317       reg = <0xf8007100 0x20>;
318     } ;
319
320     axi_bv_controller {
321       compatible = "xlnx,axi-bv-controller-1.00.a", "generic-uio";
322       reg = <0x6da00000 0x10000>;
323     };
324
325     axi_vmda0 {
326       compatible = "xlnx,axi-vdma-5.04.a", "generic-uio";
327       reg = <0x43020000 0x10000>;
328     };
329
330     axi_vmda1 {
331       compatible = "xlnx,axi-vdma-5.04.a", "generic-uio";
332       reg = <0x43000000 0x10000>;
333     };
334
335     ledccip {
```

```
336         compatible = "xlnx,ledccip-1.00.a", "generic-uio";
337         reg = <0xb6600000 0x10000>;
338       };
339
340     } ;
341 } ;
```

Listing 1: Zynq Zed Device Tree Source

Die Datei `zynq_zed.h` enthält Parameter für das Zedboard. In dieser Datei wurde eine Anpassung für den RAM des Linux Systems angepasst. Generelle Zynq Einstellung können in der `zynq-common.h` gesetzt werden. Beide Dateien findet man im Ordner `u-boot-xlnx/include/configs`.

```
1  /*
2   * (C) Copyright 2013 Xilinx, Inc.
3   *
4   * Configuration for Zynq Evaluation and Development Board -
        ZedBoard
5   * See zynq-common.h for Zynq common configs
6   *
7   * SPDX-License-Identifier: GPL-2.0+
8   */
9
10 #ifndef __CONFIG_ZYNQ_ZED_H
11 #define __CONFIG_ZYNQ_ZED_H
12
13 #define CONFIG_SYS_SDRAM_SIZE   (480 * 1024 * 1024)
14
15 #define CONFIG_ZYNQ_SERIAL_UART1
16 #define CONFIG_ZYNQ_GEM0
17 #define CONFIG_ZYNQ_GEM_PHY_ADDR0 0
18
19 #define CONFIG_SYS_NO_FLASH
20
21 #define CONFIG_ZYNQ_USB
22 #define CONFIG_ZYNQ_SDHCI0
23 #define CONFIG_ZYNQ_QSPI
24
25 #define CONFIG_ZYNQ_BOOT_FREEBSD
26 #define CONFIG_DEFAULT_DEVICE_TREE  zynq-zed
```

```
27
28 #include <configs/zynq-common.h>
29
30 #endif /* __CONFIG_ZYNQ_ZED_H */
```

Listing 2: Zynq Zed Headerdatei für U-Boot

2 CD Anhang

Die CD enthält mehrere Ordner für verschriedene Teil-Aufgabenstellungen. Die Bachelorarbeit befindet sich als PDF im Wurzelverzeichnis der CD mit dem Namen `Bachelorarbeit.pdf`. In `myled` und `bv_lin` sind im Unterordner `bin` die Binärdateien für das Erstellen einer bootfähigen SD-Karte. Die Dateien sind kompiliert und müssen nur auf eine FAT32 formatierte SD-Karte kopiert werden[vgl. Kapitel 4]. Die enthaltenen Binärdateien sind:

BOOT.BIN - Bootloader und Bitstream

uImage - Linux Kernel Image

devicetree.dtb - Device Tree Blob

uramdisk.image.gz - Root Filesystem in Form einer RAM Disk.

2.1 MyLED

Der Ordner myled enthält das LED Steurungsbeispiel. Im Ordner `bin` sind die Linux Komponenten sowie das kompilierte Steuer-Programm. Zusammen mit den Linux Komponenten kann es auf die SD-Karte kopiert werden um ein bootfähiges System mit Beispielanwendung zu starten. Wenn das System bootet und man in das Verzeichnis /mnt wechselt kann mit `./myled <Wert>` einen Wert für die LEDs gesetzt werden.

Im Unterordner sources liegen die Device Tree Source und die Quelldatei von `myled`: `myled.c` sowie das genutzte MAKEFILE.

2.2 BV-Kette Linux

Die kompilierten Dateien für das Linux System das die BV-Kette integriert ist im Ordner `bv_lin` zu finden. Der komplette Inhalt von `bin` wird dafür auf eine FAT32 formatierte SD-Karte kopiert. Das Programm wird aus dem Ordner `/mnt` mit dem Befehl `./bv_test_lin` gestartet. Mit >> kann die Standardausgabe auf ein beliebiges file umgeleitet werden.

Im Ordner `Diverses` im Wurzelverzeichnis der CD befinden sich 2 Unterordner. `com_test_dump` enthält Referenzbilder und logfiles aus der vorangegangenen Masterarbeit. Der Ordner `ComPortReader` enthält das Java Programm zum Auswerten der .log Dateien. Es kann aus dem Ordner `bin` mit dem Befehl `java rolf.ma.comportreader.COMTOIMAGE` gestartet werden. Es öffnet sich ein Fenster zum Auswählen einer Logdatei, aus der Bitmaps generiert werden.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 17. November 2014	
Ort, Datum	Martin Hepke