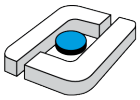

Hochschule Osnabrück
University of Applied Sciences

Entwicklung eines Treibers und einer Toolchain zur Administration eines Embedded Systems

Am Beispiel einer ZPU unter Linux

Martin Helmich (martin.helmich@hs-osnabrueck.de)
Oliver Erxleben (oliver.erxleben@hs-osnabrueck.de)

Hochschule Osnabrück
Ingenieurswissenschaften und Informatik
Informatik - Mobile und Verteilte Anwendungen

27. Februar 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Zusammenfassung:
Die vorliegende Arbeit wurde mit LaTeX verfasst und ist eine gemeinsame Arbeit von Oliver Erxleben und Martin Helmich für das Modul *Hardwarenahe System- und Treiberprogrammierung* aus dem Master-Studiengang *Informatik - Verteilte und Mobile Anwendungen* im Wintersemester 2012/13 an der Hochschule Osnabrück / University of Applied Sciences.

Das Thema der Arbeit lautet *Entwicklung eines Treibers und einer Toolchain zur Administration eines Embedded Systems*. Im Kern beschreibt die Arbeit die Entwicklung eines Treibers für ein eingebettetes System unter Linux. Treiberprogrammierung unter anderen Systemen, wie zum Beispiel Microsoft Windows oder Apple Mac OS werden nicht betrachtet. Genauer wird zum einen ein Linux-Treiber für eine ZPU vorgestellt. Zum anderen werden Bibliotheken und Werkzeuge zur Verwendung in Anwenderprogrammen und zur Steuerung der ZPU beschrieben.

Im ersten Teil der Arbeit wird das Konzept der Hardware und das Konzept des Treibers und der Programmierbibliotheken vorgestellt. In den darauffolgenden Abschnitten werden detailliert die einzelnen Komponenten des Treibers beschrieben.

Die Anwenderprogramme zum Testen werden in einem eigenen Abschnitt vorgestellt, jedoch an geeigneten Stellen wird auf die Testprogramme verwiesen. Sofern nicht anders angegeben, wird zum Kompilieren des Quellcodes der GNU-Compiler GCC unter Linux (Kernel-Version 2.6.XX) verwendet.

1 Einleitung

1.1 Motivation

Täglich verwendet und verlässt sich unsere Gesellschaft auf computergestützte Anwendungen. Wir suchen, schreiben, drucken, kopieren, erstellen oder entfernen Daten. Nur allzu oft werden dabei Geräte verwendet, die Funktionen für den Anwender bereitstellen, oder aber das Funktionsspektrum des Comutersystems erweitern. Funktionen der Computerkomponenten werden durch Anwenderprogramme durchgeführt, die von einem Betriebssystem verwaltet werden. Das Betriebssystem verwaltet auch alle Komponenten des Computersystems. Diese Komponenten sind Software oder auch Hardware. Erst die Teamarbeit zwischen Hardware und Software ermöglicht für den Anwender die Ausführung komplexer Programme.

Das Betriebssystem steuert den Zugriff auf Hardware und benötigt Kenntnisse über die installierten Hardwarekomponenten. Das *Wissen* über angeschlossene Hardwarekomponenten ist in sog. Gerätetreibern[1] hinterlegt. Es stellt einen Teil des Betriebssystemkerns dar, der für den Zugriff auf Hardware verantwortlich ist. Für jedes Gerät wird ein eigener Treiber benötigt.[2]

Die Entwicklung eines Gerätetreibers und zugehörige Softwarekomponenten stellen einen interessanten, wenn nicht sogar erstrebenswerten, Teil der Programmierung von Computersystemen dar.

1.2 Aufgabenstellung

Diese Arbeit behandelt die Entwicklung eines Linux-Treibers für ein PCI-Board mit eingebettetem ZPU-Prozessor. An den im Rahmen dieser Arbeit zu entwickelnden Treiber werden die folgenden Anforderungen gestellt:

- IOCTL/MMAP: Es soll möglich sein das eingebettete Prozessorsystem durch ein Applikationsprogramm zurückzusetzen und mit neuer Software zu laden.
- Read-/Write-Funktionen: Im Betrieb des eingebetteten Prozessorsystems soll es möglich sein über Anwenderprogramme Eingaben von der Standardeingabe zu senden und Ausgaben von der Standardausgabe zu lesen.
- Entwicklung geeigneter Anwenderprogramme zum Test des Treibers.

[1] Gerätetreiber: Softwaremodul, welches Zugriffe und Interaktionen mit installierter Hardware steuert.
[2] Vgl. Jürgen Quade (2004): Linux Treiber Entwicklen, Kapitel 1.

2 Architektur

2.1 ZPU

Der ZPU ist nach Angaben des Herstellers, Zylin AS, der kleinste 32-Bit-Mikroprozessor der Welt und zählt zu der Kategorie der *Soft CPU*[3]s. Die grundlegende Idee, bzw. Aufgabe einer Soft-CPU, bzw. der ZPU ist so wenig FPGA[4]-Speicher wie möglich aufzunehmen und Berechnungen dem HDL[5]-Programm zu überlassen.

Zur Programmierung der ZPU kann die GCC-Toolchain verwendet werden. Neben *GDB* werden *newlib* und *libstdc+* unterstützt.

Weiterhin ist es möglich den ZPU mit einem eingebetteten Betriebssystem zu betreiben. Das *eCos*[6]-Embbedded Operating System wird vom ZPU unterstützt.

Neben dem ZPU sind weitere Komponenten auf dem PCI-Board enthalten. Zum einen zwei Ein- und Ausgabe-Fifos, welche zum Laden des Programms für die ZPU genutzt werden und zum anderen eine 7-Segment-Anzeige.

Die Hardware-Architektur der ZPU lässt sich in Abbildung 1 veranschaulichen. Das Prozessorsystem besteht aus dem ZPU-Prozessor, RAM- und ROM-Speicher und zwei FIFOs aus 8-Bit-Zeichen. Die Ein- und Ausgabe von Daten lassen sich über das Sende- und Empfangs-Fifo steuern. Zu beachten ist an dieser Stelle das die Fifos nach dem Ausgangspunkt (ZPU oder Linux-Kernel) zum Empfangen oder Senden genutzt werden. Aus Sicht der ZPU werden Daten gesendet wenn diese im Empfangs-Fifo des Kernels geschrieben werden. Empfangen werden Daten vom Kernel wenn diese in das Sende-Fifo geschrieben werden.

Das Prozessorsystem kann über den PCI-Bus (und den darüber gesteuerten Wishbone-Bus) unterschiedlich angesprochen werden.

- Über ein Kontrollregister kann das Prozessorsystem zurückgesetzt werden.
- Der ROM- und der RAM-Speicher sind als Dual-Port-Speicher ausgeführt. Ein Port ist an die ZPU angeschlossen, über den zweiten Port kann durch den PCI-Bus auf die Speicher zugegriffen werden. Der ROM-Speicher ist dabei nur aus Sicht der ZPU ein reiner Lesespeicher. Aus Sicht des PCI-Busses kann der Speicher geschrieben und gelesen werden.
- Das Sende-Fifo des Prozessorsystems kann mittels Interrupt gelesen und das Empfangs-Fifo mittels Interrupt geschrieben werden. Ein Interrupt-Controller schaltet die beiden Interrupts auf das gemeinsame PCI-Interrupt-Signal.

[3]Soft-CPU: Nicht fest platziert, befindet sich in einem FPGA (Vgl. siehe FPGA Soft Core)
[4]FPGA: Field Programmable Gate Array
[5]HDL: Hardware Description Language
[6]eCos: Real-time OS for embedded Applications (weitere Informationen, siehe eCos - open source real time operating system

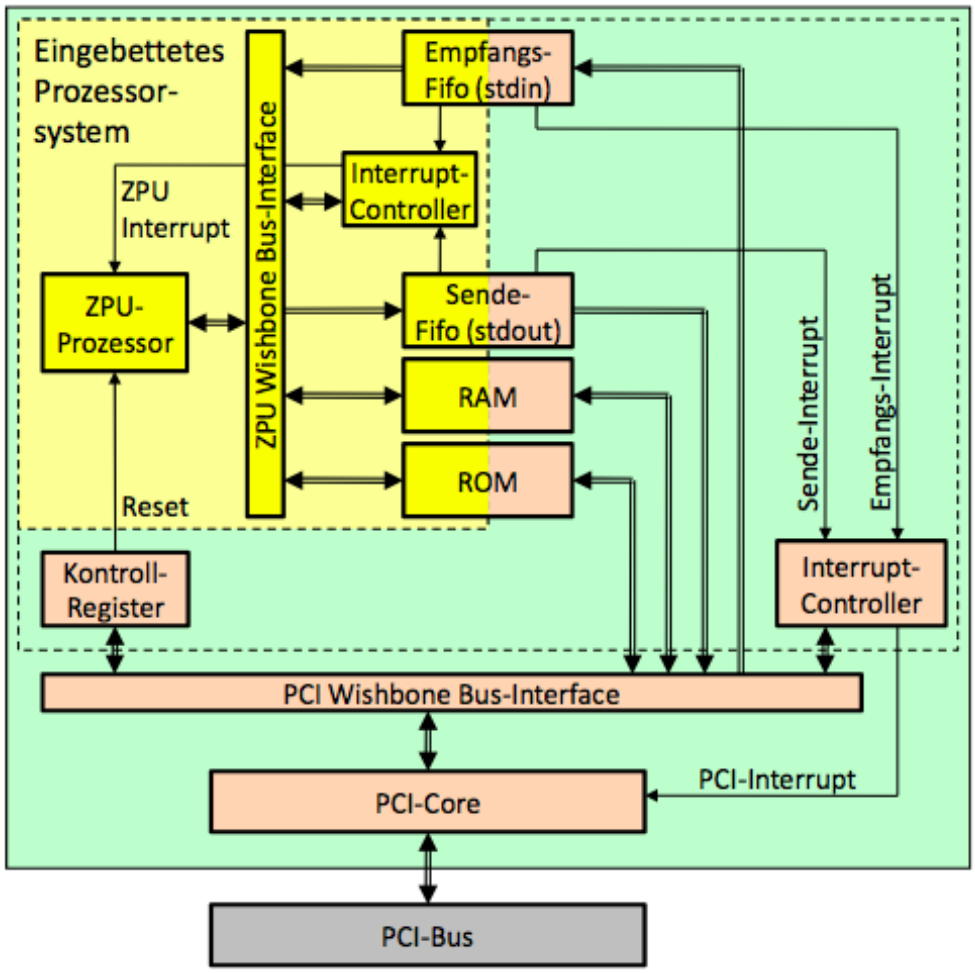

Abbildung 1: Architektur der ZPU

Auf der Hardware befinden sich verschiedene Register, zum Beispiel um die ZPU zu kontrollieren. Die Abbildung 2 visualisiert die verschiedenen Register.

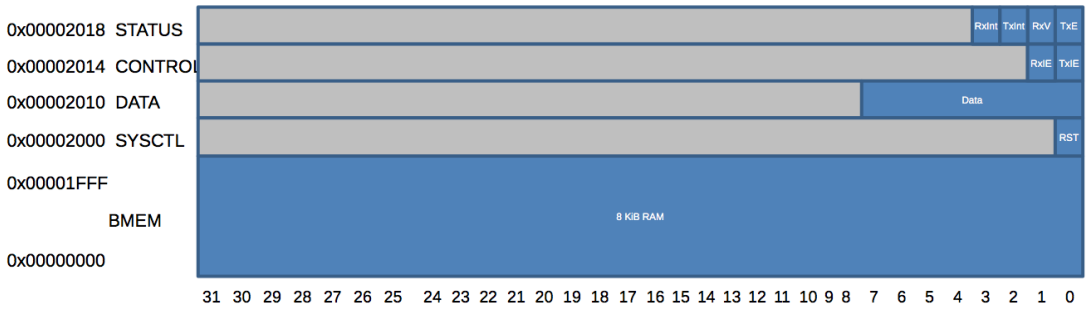

Abbildung 2: Register der ZPU

0x00002000 SYSCTL Solange RST=1 ist, wird das eingebettete System im *reset*-Modus gehalten. Wenn RST=0, wird der normale Betriebszustand ausgeführt, d. h. das Programm wird ausgeführt.

0x00002018 DATA Das Datenregister: Daten für die ZPU-Standardeingabe werden hier geschrieben. Daten der ZPU-Standardausgabe können hier ebenfalls gelesen werden. Ein lesender Zugriff blockiert das Empfangs-FIFO, sollte es leer sein und ein schreibender Zugriff blockiert das Sende-FIFO, sollte es voll sein.

0x00002014 CONTROL Das Kontrollregister `RxIE` kann aktiviert werden um einen Interrupt beim Empfangs-FIFO auszulösen, sollte dieses nicht leer sein. Bei Aktivierung des Kontrollregisters `TxIE` wird ein Interrupt beim Sende-FIFO ausgelöst, sollte es nicht voll sein.

0x00002018 STATUS Das Register `RxInt` zeigt an, dass ein Empfangs-Interrupt angefordert wird. Status wird durch eine AND-Verknüpfung von `RxV` und `RxIE` gesetzt. `TxInt` zeigt an das ein Sende-Interrupt angefordert wird. Es wird gesetzt durch AND-Verknüpfung von `TxE` und `TxIE`. Bei `RxV` handelt es sich um die Anzeige, dass das Empfangs-FIFO nicht leer ist. Lesen aus dem Datenregister ist möglich. `TxE` zeigt an, dass das Sende-FIFO nicht voll ist und ein Schreiben in das Datenregister möglich ist.

2.2 Überblick über die Treiber-Toolchain

Wie in der Einleitung bereits beschrieben, sollen neben dem Treiber-Modul auch Bibliotheken für Benutzerprogramme, sowie Benutzerprogramme selbst entwickelt werden. Die Abbildung 3 zeigt die drei Ebenen die entwickelt werden.

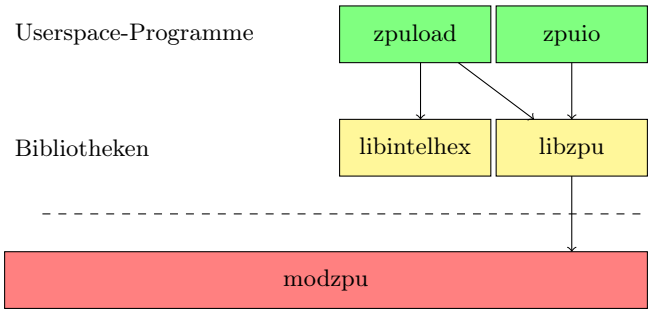

Abbildung 3: Überblick ZPU-Toolchain

3 Linux-Treibermodul: modzpu

Der Treiber für das Embedded System stellt die zentrale Steuerungssoftware für das Hostsystem und den Treiber dar. Dieser Abschnitt gibt einen Überblick über das Treibermodul und stellt die wichtigsten Funktionen im Modul vor. Zur Übersichtlichkeit wurden Funktionen für den PCI-Treiber in mehrere Dateien aufgesplittet. Das Treibermodul ist wie folgt aufgebaut:

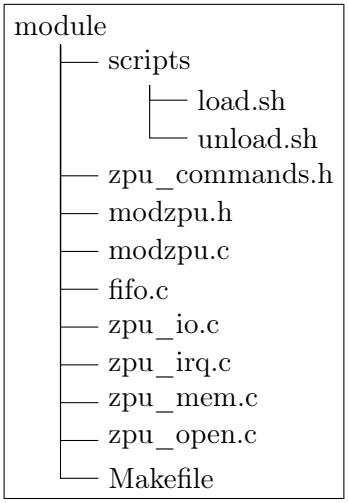

Abbildung 4: Dateien des ZPU-Kernelmoduls.

3.1 Installation

Das Treibermodul lässt sich einfach über ein Makefile installieren, bzw. deinstallieren. Es befindet sich im Wurzelverzeichnis des Treibermoduls und benötigt die sich im Unterverzeichnis *scripts* befindenen Bash-Skripte *load.sh* und *unload.sh*. Die Bash-Skripte sind für das Laden und Entladen des Treibermoduls aus dem Kernel verantwortlich. Das Listing 1 zeigt die bequeme Installation des Treibers. Anzumerken ist, dass der Eingriff in den Kernel und das Laden/Entladen des Moduls in den Kernel, bzw. aus dem Kernel, root-Rechte am Sytem benötigt. Äquivalent zum Listing 1 kann das Modul mittels `make uninstall` wieder entladen werden.

Listing 1: Installation des Treibermoduls

```
1 > cd to/the/module/folder
2 > su # optional, if not root
3 > make
4 > make install
```

3.2 Schnittstellen

3.2.1 Funktionen

Das Treibermodul stellt alle wichtigen Funktionen für die Hardware zur Verfügung. Dies sind neben den Basisfunktionen eines Treibers, wie z.B. `init` oder `mypci_probe`, auch Funktionen zur Speicherverwaltung. Das Listing 2 zeigt die Deklaration der Funktionen aus der `modzpu.h`.

3.2.2 Konstanten

Der Treiber deklariert neben den Funktionen auch benötigte und hilfreiche Konstanten. Zum Beispiel werden die

3.2.3 Strukturen

Weiterhin werden in der `modzpu.h` Strukturen für Dateioperationen und zur Speicherverwaltung deklariert. Das Listing 4 zeigt dessen Deklaration. Der Struktur für Dateioperationen werden die in Listing 2 deklarierten Gerätefunktionen übergeben. Die Sturktur wird benötigt um Methoden des Treibers beim Registrieren des Geräts mit dem System bekannt zu machen.

Listing 2: Funktionsüberblick des Treibermoduls

```
47 // driver specific init and exit functions
48 static int  __init init    (void);
49 static void __exit cleanup (void);
50
51 // PCI functions
52 static int  mypci_probe  (struct pci_dev *dev, const struct
       pci_device_id *id);
53 static void mypci_remove (struct pci_dev *dev);
54
55 // Character device  functions
56 static int      mychr_open  (struct inode* inodep, struct file* filep);
57 static int      mychr_close (struct inode* inodep, struct file* filep);
58 static int      mychr_mmap  (struct file *filep, struct vm_area_struct
       *vma);
59 static int      mychr_ioctl (struct inode *inodep, struct file* filep,
       unsigned int cmd, unsigned long param);
60 static ssize_t mychr_write (struct file *filep, const char __user *
       data, size_t count, loff_t *offset);
61 static ssize_t mychr_read  (struct file *filep, char __user *data,
       size_t count, loff_t *offset);
62
63 // Memory management functions
64 void myvma_open  (struct vm_area_struct *vma);
65 void myvma_close (struct vm_area_struct *vma);
66
67 // Interrupt function
68 irqreturn_t myirq_handler(int irq, void *dev_id);
```

Listing 3: Verwendete Konstanten im Treibermodul

```
13 #define MINOR_COUNT 1
14 #define MINOR_FIRST 0
15
16 #define VENDOR_ID 0x10EE
17 #define DEVICE_ID 0x0100
18
19 #define ADDR_RAM     0x0000000
20 #define ADDR_SYSCTL  0x0002000
21 #define ADDR_DATA    0x0002010
22 #define ADDR_CONTROL 0x0002014
23 #define ADDR_STATUS  0x0002018
24
25 #define SYSCTL_STOP  0x01
26 #define SYSCTL_START 0x00
27
28 #define CTRL_RXIE    (1 << 1)
29 #define CTRL_TXIE    (1     )
30 #define CTRL_STDIN_READY  CTRL_TXIE
31 #define CTRL_STDOUT_READY CTRL_RXIE
32
33 #define STAT_RXINT   (1 << 3)
34 #define STAT_TXINT   (1 << 2)
35 #define STAT_RXV     (1 << 1)
36 #define STAT_TXE     (1     )
37 #define STAT_STDIN_READY  STAT_TXINT
38 #define STAT_STDOUT_READY STAT_RXINT
39
40 #define ZPU_IO_BUFFER_SIZE 4096
41 #define ZPU_RAM_SIZE 8192
```

Listing 4: modzpu Strukturen

```
69 struct file_operations mychr_fops = {
70   .owner    = THIS_MODULE,
71   .open     = mychr_open,
72   .release  = mychr_close,
73   .read     = mychr_read,
74   .write    = mychr_write,
75   .mmap     = mychr_mmap,
76   .ioctl    = mychr_ioctl
77 };
78
79 static struct vm_operations_struct myvma_ops = {
80   .open = myvma_open,
81   .close = myvma_close
82 };
```

3.3 Initialisieren des Moduls

Wie im Listing 2 zu sehen ist, wird die Initialisierungsfunktion in der *modzpu.h* deklariert und in der *modzpu.c* implementiert. Das Listing 5 zeigt die Implementierung der *init*-Funktion.

Listing 5: modzpu Init-Funktion

```
67 static int __init init(void)
68 {
69   int r;
70   struct pci_driver *d = &(mypci_driver);
71
72   d->name     = "zpu";
73   d->id_table = (const struct pci_device_id*) &(id_table);
74   d->probe    = mypci_probe;
75   d->remove   = mypci_remove;
76
77   if ((r = pci_register_driver(d)) != 0)
78   {
79     OUT_WARN("Could not register ZPU device driver.\n");
80     return r;
81   }
82
83   OUT_DBG("Initialized module.\n");
84
85   return 0;
86 }
```

Benötige Treiberinformationen werden in der Struktur *mypci_ driver* gehalten. Die Struktur ist in *linux/pci.h* deklariert und wird zur Registrierung der PCI-Hardware im Kernel benötigt.[7]

Zur Fehlersicherung wird der Rückgabewert der Funktion `pci_register_driver(d)` überprüft und ggf. wird der Wert, sollte er nicht 0 sein, zurückgeliefert.

3.4 Initialisierung des PCI-Geräts

Die eigentliche Initialisierung des PCI-Geräts erfolgt in der Methode `mypci_probe`. Diese wurde zuvor in der Initialisierungsmethode registriert und wird vom Kernel aufgerufen, sobald ein entsprechendes Gerät erkannt wurde.

Die `mypci_probe`-Methode übernimmt die folgenden Aufgaben:

1. Aktivieren des Geräts per `pci_enable_device`.
2. Abbilden des Gerätespeichers in den Adressraum des Kernels. Hierzu werden zunächst mit `pci_resource_start` und `pci_resource_len` physikalische Adresse und

[7] Vgl. Lang (2012): Hardwarenahe System- und Treiberprogrammierung, Seite 253

Größe des Gerätespeichers ermittelt. Anschließend wird per `pci_request_regions` Zugriff auf den Bereich angefordert und dieser anschließend per `ioremap` in den Adressraum des Kernels abgebildet.

3. Erstellen eines *Character Device* mit dem Namen *zpu* per `alloc_chrdev_region`, `cdev_init` und `cdev_add`.
4. Initialisierung zweier FIFO-Speicher, die zum Puffern der Ein- und Ausgabeströme der ZPU dienen. Die Größe der Ein- und Ausgabepuffer wird von der Konstante `ZPU_IO_BUFFER_SIZE` bestimmt.
5. Ermittlung der zugehörigen Interrupt-Nummer (diese wird bei PCI-Geräten vom Kernel automatisch erkannt)[8] und Registrierung eines Interrupt-Handlers.
6. Freigabe des *stdout*-Interrupts durch Setzen der entsprechenden Register im Gerätespeicher.

Sollte es beim Ablauf dieser Funktion zu Fehlern kommen, werden bisher durchgeführte Aktionen wieder rückgängig gemacht, damit das System auch beim Fehlerfall in einem konsistenten Zustand hinterlassen wird.

3.5 IOCTRL

Um die ZPU in den Reset-Modus versetzen zu können, werden zwei *ioctrl*-Kommandos angeboten: `RAGGED_ZPU_STOP` versetzt die ZPU in den Reset-Modus und `RAGGED_ZPU_START` lässt die ZPU weiterlaufen. Beide Kommandos setzen das RST-Register an Adresse `0x2000` auf jeweils 1 oder 0. Sie werden später benötigt, um neue Software in den ZPU-Speicher laden zu können.

3.6 Lese- und Schreibzugriffe

Die Lese- und Schreibzugriffe auf, bzw. von der ZPU, sind in Abbildung 5 veranschaulicht. Das Treibermodul kann aus dem Kernel Daten über das Eingabe-FIFO schreiben und über das Ausgabe-FIFO Daten von der PCI-Hardware entgegen nehmen.

3.6.1 FIFO

Daten eines FIFOs werden über eine FIFO-Struktur gehalten, welche in der Datei `fifo.c` implementiert ist. Nachfolgend werden die Funktionen und Makros der `fifo.c` kurz vorgestellt:

void fifo_init (fifo_t *f, size_t s); In der Initiierungsfunktion werden die Lese- und Schriebzeiger, sowie Queues initialisiert. Ausserdem wird Speicher für Daten per `vmalloc` alloziiert.

[8] Vgl. A. a. O., Sete 235 ff.

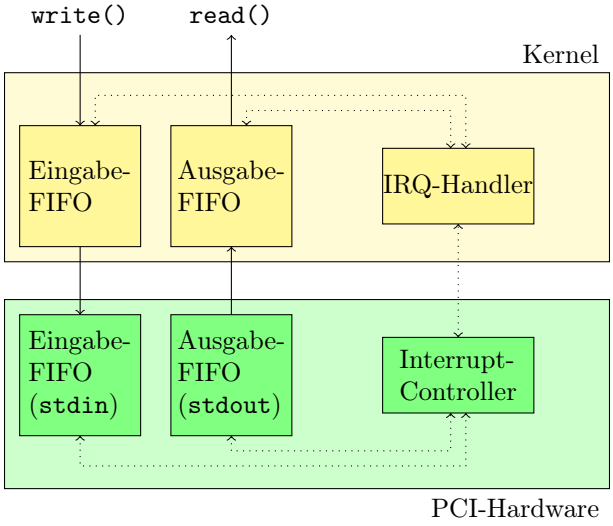

Abbildung 5: ZPU-Ein und Ausgabe

void fifo_delete (fifo_t *f); Gibt reservierten Speicher wieder frei.

int fifo_transfer (fifo_t *f1, fifo_t *f2, int n); Überträgt n Bytes von dem übergebenen FIFO f1 zu FIFO f2.

u8 fifo_read_byte (fifo_t *f); Wenn der übergebene FIFO nicht leer ist, wird ein Byte aus dem FIFO gelesen und der Füllstand verringert.

void fifo_write_byte (fifo_t *f, u8 b); Wenn der übergebene FIFO nicht voll ist, wird ein Byte in das FIFO geschrieben und der Füllstand erhöht.

bool FIFO_(NOT_)FULL(fifo_t *f) Wahr, wenn das FIFO (nicht) voll ist

bool FIFO_(NOT_)EMPTY(fifo_t *f) Wahr, wenn das FIFO (nicht) leer list.

int FIFO_FREE(fifo_t *f) Liefert die Anzahl der noch verfügbaren Bytes zurück.

3.7 Modul-Cleanup

Das Entfernen des PCI-Geräts wird in der Methode `mypci_remove` vorgenommen. Hier werden die bei der Initialisierung durchgeführten Schritte (siehe Abschnitt 3.4) in umgekehrter Reihenfolge wieder rückgängig gemacht.

Beim Entladen des Kernel-Moduls wird schließlich noch die Methode `cleanup` aufgerufen. Diese entlädt noch den PCI-Treiber selbst aus dem Kernel.

Listing 6: Entfernen des PCI-Geräts

```
179 static void mypci_remove(struct pci_dev *dev)
180 {
181   // Free interrupt handler
182   free_irq(dev->irq, pcidev_config);
183
184   // Unmap PCI address mapping
185   iounmap(pcidev_config);
186   pci_release_regions(dev);
187
188   // Disable PCI device
189   pci_disable_device(dev);
190
191   // Disable and deallocate character devices.
192   cdev_del(&c_dev);
193   unregister_chrdev_region(dev_number, MINOR_COUNT);
194 }
```

3.8 Speicher-Mapping

Um aus dem Userspace heraus neue Programme in den Speicher der ZPU laden zu können, bietet das Kernelmodul den `mmap`-Systemaufruf an. Dieser blendet den 8 KiB großen RAM des ZPU-Boards in den Adressraum eines Benutzer-Prozesses ein. Hierzu wird die Methode `io_remap_pfn_range` genutzt (siehe Listing 7).

3.9 Interrupt-Bearbeitung

Die Bearbeitung von Interrupts erfolgt in der Methode `myirq_handler`. Da über den Interrupt-Controller mehrere Interrupts ausgelöst werden können, muss hier anhand der Statusregister TXINT und RXINT zunächst überprüft werden, welcher Interrupt-Typ vorliegt.

Der TXINT-Interrupt wird ausgelöst, sobald Daten in den Eingabe-FIFO der ZPU geschrieben werden können. Der RXINT-Interrupt hingegen wird ausgelöst, sobald Daten aus dem Ausgabe-FIFO der ZPU gelesen werden können.

Tritt der TXINT-Interrupt auf, versucht der Interrupt-Handler so viele Bytes wie möglich aus dem Kernel-Puffer in den Eingabe-FIFO der ZPU zu kopieren. Das Kopieren erfolgt byteweise; nach jedem Byte wird anhand des TXE-Registers überprüft, ob noch ein weiteres Byte geschrieben werden kann.

Beim RXINT-Interrupt wird analog dazu verfahren. Hier versucht der Interrupt-Handler, möglichst viele Bytes aus dem Ausgabe-FIFO der ZPU in den Puffer im Kernelspace zu kopieren. Hier wird anhand des RXE-Registers überprüft, ob noch weitere Daten gelesen werden können.

Listing 7: Implementierung der mmap-Funktion

```
 9 static int mychr_mmap(struct file *filep, struct vm_area_struct *vma)
10 {
11   unsigned long vm_size, map_size;
12
13   vm_size  = vma->vm_end - vma->vm_start;
14   map_size = vm_size < ZPU_RAM_SIZE ? vm_size : ZPU_RAM_SIZE;
15
16   if (io_remap_pfn_range(vma, vma->vm_start,  (paddr + ADDR_RAM) >>
       PAGE_SHIFT,  map_size,  vma->vm_page_prot) != 0) return -EAGAIN;
17
18   vma->vm_ops = &myvma_ops;
19   myvma_open(vma);
20
21   return 0;
22 }
```

Nachdem so viele Daten wie möglich kopiert wurden, werden am Ende des Interrupt-Handlers eventuell wartende Lese- oder Schreibprozesse (je nachdem, welcher Interrupt aufgetreten ist) wieder aufgeweckt.

Listing 8: Implementierung des Interrupt-Handlers

```
12 irqreturn_t myirq_handler(int irq, void *dev_id)
13 {
14   unsigned int status = ZPU_IO_READ(ADDR_STATUS);
15   if ((status & STAT_STDIN_READY) > 0)
16   {
17     fifo_t *f = &(zpu_io_stdin);
18     while ((status & STAT_STDIN_READY) > 0 && FIFO_NOT_EMPTY(f))
19     {
20       ZPU_IO_WRITE(ADDR_DATA, fifo_read_byte(f));
21       udelay(75);
22       status = ZPU_IO_READ(ADDR_STATUS);
23     }
24
25     if (FIFO_EMPTY(f)) ZPU_DISABLE_STDIN_IR();
26
27     wake_up_all(&(f->queue));
28     return IRQ_HANDLED;
29   }
30   else if ((status & STAT_STDOUT_READY) > 0)
31   {
32     fifo_t *f = &(zpu_io_stdout);
33     while ((status & STAT_STDOUT_READY) > 0 && FIFO_NOT_FULL(f))
34     {
35       fifo_write_byte(f, ZPU_IO_READ(ADDR_DATA));
36       udelay(75);
37       status = ZPU_IO_READ(ADDR_STATUS);
38     }
39
40     if (FIFO_FULL(f)) ZPU_DISABLE_STDOUT_IR();
41
42     wake_up_all(&(f->queue));
43     return IRQ_HANDLED;
44   }
45   return IRQ_NONE;
46 }
```

4 Parsing der Intel HEX-Dateien: Die libcintelhex

Die Logik zum Parsen der Intel HEX-formatierten Eingabedateien wurde in eine eigene Programmbibliothek ausgegliedert. Der folgende Abschnitt beschreibt zunächst das Intel HEX-Datenformat, erläutert anschließend Installation und Schnittstellen der (im Rahmen dieser Arbeit entstandenen) cintelhex-Bibliothek und erklärt die dort verwendeten Algorithmen.

4.1 Das Intel HEX-Format

Bei dem *INTEL Hexadecimal Object File Format* (kurz *INTEL HEX*) handelt es sich um ein Dateiformat zur Codierung binärer Daten in ASCII-Dateien. Ursprünglich ausgelegt für 8-Bit-Intelprozessoren mit Adressräumen von 16 Bit, wurde das Format später erweitert für 16-Bit-Prozessoren mit 20-Bit-Adressräumen und für 32-Bit-Prozessoren. [9]

Der Aufbau eines Intel HEX-Datensatzes ist in Abbildung 6 dargestellt. Listing 9 zeigt einen Auszug aus einer Beispiel-Datei. Jeder Datensatz beginnt mit einem *Record Mark* (ein `:`-Zeichen, oder `0x3A` in ASCII). Es folgen die Länge des Datensatzes (1 Byte, codiert in 2 ASCII-Zeichen), der Adress-Offset (2 Byte), der Typ des Datensatzes (1 Byte), der eigentliche Datensatz (variabel, bis zu 255 Bytes) und eine Prüfsumme (1 Byte), anhand derer die Korrektheit des Datensatzes überprüft werden kann.

:	10	0400	00	B0B0B98B02D0B0B0B88800400000000	29
	Länge	Adresse	Typ	Daten	Prüfsumme

Abbildung 6: Aufbau eines Intel HEX-Datensatzes

Neben regulären Datensätzen (Typ `0x00`) und einem EOF-Datensatz (Typ `0x01`), der das Dateiende markiert, kennt die Spezifikation noch weitere Datensatz-Typen, die beispielsweise die Nutzung eines 20- oder 32-Bit-Adressraums ermöglichen. So kann beispielsweise ein spezieller *Extended Linear Address Record* (Typ `0x04`) verwendet werden, um durch Angabe einer *Upper Linear Base Address* den Adressraum auf 32 Bit zu erweitern.

Da der Speicher des verwendeten ZPU-Boards nur 8 KiB umfasst, ist eine Verwendung der 32-Bit-Funktionen nicht notwendig (16 Bit adressieren maximal $2^{16} = 64$ KiB). Aus diesem Grund bietet die cintelhex-Bibliothek nur experimentelle Unterstützung für 32-Bit-Adressen.

4.2 Installation

Die Installation der cintelhex-Bibliothek erfolgt mit den Unix-üblichen `configure`- und `make`-Befehlen. Außer dem Build-Tool `make`, der C-Standardbibliothek und dem GCC

[9] Vgl. Intel Corporation (1988): Hexadecimal Object File Format Specification, S. 4

Listing 9: Beispiel einer Intel HEX-Eingabedatei

```
:100000000000000000000000000000000000000F0
:0400100000000000EC
:100400000B0B0B98B02D0B0B0B8880040000000029
:10041000000000000000000000000000000000DC
// ...
:1010E200000008D9000008D9000008D9000008D97A
:1010F200000008D9000008D9000008D9000008D96A
:0C110200000008D9000008D9000008D93E
:0400000300000400F5
:00000001FF
```

werden keine Abhängigkeiten benötigt. Bei der Installation wird die Datei `libcintelhex.so` nach `<prefix>/lib`, und die Datei `cintelhex.h` nach `<prefix>/include` installiert. Das Standardpräfix ist `/usr/local`.

Listing 10: Installation der cintelhex-Bibliothek

```
> ./configure # optional: --prefix=/usr/local
> make
> make install
> ldconfig
```

Bei Verwendung des Standardpräfix kann die cintelhex-Bibliothek beim Linken durch den GCC-Parameter `-l cintelhex` aktiviert werden.

4.3 Schnittstellenbeschreibung

Die Methoden der cintelhex-Bibliothek können durch inkludieren der Header-Datei `cintelhex.h` verwendet werden. In der Datei werden folgende wichtigen Methoden deklariert:

ihex_recordset_t* ihex_rs_from_file(char* filename) Diese Methode liest den Inhalt einer Intel HEX-Datei (der Dateiname wird durch `filename` beschrieben) in eine `ihex_recordset_t`-Struktur (siehe auch Abschnitt 4.4). Sie liefert einen Zeiger auf die ausgelesene Struktur zurück, oder `NULL` im Fehlerfall.

ihex_recordset_t* ihex_rs_from_string(char* data) Diese Methode verhält sich genau wie `ihex_rs_from_file`, erwartet als Eingabeparameter jedoch direkt einen Zeiger auf einen Intel HEX-String.

int ihex_mem_copy(ihex_recordset_t *rs, void* dst, ulong_t n, ihex_width_t w, ihex_byteorder_t o) Diese Methode kopiert das durch eine Intel HEX-Datei beschriebene Programm an eine beliebige Stelle im Speicher. Das zu ladende Programm wird durch die Struktur `rs` beschrieben. Der Zeiger `dst` beschreibt die Zielposition im Speicher (dies kann auch ein durch `mmap` abgebildeter Gerätespeicher sein). `n` beschreibt die Größe des Zielbereichs. Die Methode liefert im Erfolgsfall 0, ansonsten einen Fehlercode zurück.

char* ihex_error() Diese Methode liefert eine Beschreibung des zuletzt aufgetretenen Fehlers zurück. Falls kein Fehler aufgetreten ist, wird `NULL` zurückgeliefert.

ihex_error_t ihex_errno() Diese Methode liefert den Fehlercode des zuletzt aufgetretenen Fehlers zurück. Falls kein Fehler aufgetreten ist, wird `0` zurückgeliefert.

4.4 Datenstrukturen

Die cintelhex-Bibliothek definiert die beiden Datenstrukturen `ihex_recordset_t` und `ihex_record_t`. Sie sind wie folgt definiert:

Listing 11: Definition der `ihex_record_t`- und `ihex_recordset_t`-Strukturen

```
1  /// Models a single Intel HEX record.
2  typedef struct ihex_record {
3    uint_t       ihr_length;   //!< Length of the record in bytes.
4    ihex_rtype_t ihr_type;     //!< Record type (see ihex_rtype_t).
5    ihex_addr_t  ihr_address;  //!< Base address offset.
6    ihex_rdata_t ihr_data;     //!< Record data.
7    ihex_rchks_t ihr_checksum; //!< The record's checksum.
8  } ihex_record_t;
9
10 /// Models a set of Intel HEX records.
11 typedef struct ihex_recordset {
12   uint_t         ihrs_count;   //!< Amount of records.
13   ihex_record_t *ihrs_records; //!< A list of record (with ihrs_count
         elements).
14 } ihex_recordset_t;
```

Abbildung 7 zeigt zudem die Beziehungen der Datenstrukturen untereinander. Das Attribut `ihex_data` der Struktur `ihex_record_t` ist ein Zeiger auf ein `uint8_t`-Feld der Länge `ihr_length`.

4.5 Fehlercodes

Die `cintelhex.h` definiert die folgenden Fehlerkonstanten:

IHEX_ERR_INCORRECT_CHECKSUM Tritt auf, wenn die Prüfsumme eines Eintrages ungültig ist. Zur Verifizierung der Prüfsumme werden alle Bytes eines Eintrages (einschließlich der Prüfsumme selbst) aufsummiert. Ist das unterste Byte der Summe = 0, ist die Prüfsumme korrekt.

IHEX_ERR_NO_EOF Tritt auf, wenn die Eingabedatei keinen EOF-Eintrag (Typ `0x01`) enthält.

IHEX_ERR_PARSE_ERROR Tritt auf, wenn die Eingabedatei ungültig formatiert ist (etwa bei Fehlen des Record Marks (`:`)).

IHEX_ERR_WRONG_RECORD_LENGTH Tritt auf, wenn die tatsächliche Länge eines Datensatzes nicht der im ersten Byte angegebenen entspricht.

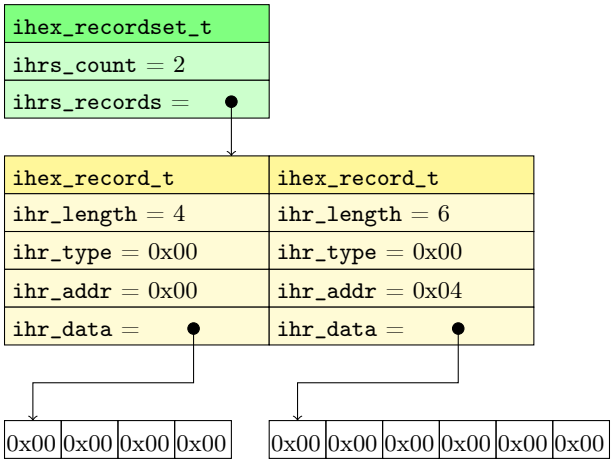

Abbildung 7: Datenstrukturen der libcintelhex

IHEX_ERR_NO_INPUT Tritt auf, wenn die Eingabedatei nicht existiert, oder nicht geöffnet werden konnte.

IHEX_ERR_UNKNOWN_RECORD_TYPE Tritt auf, wenn ein Datensatz eine ungültige Typ-Bezeichnung (Byte 4) hat).

IHEX_ERR_PREMATURE_EOF Tritt auf, wenn der EOF-Datensatz (Typ `0x01`) in einer anderen Zeile als der letzten auftritt.

IHEX_ERR_ADDRESS_OUT_OF_RANGE Tritt auf, wenn an eine ungültige Adresse geschrieben werden soll.

IHEX_ERR_MMAP_FAILED Tritt auf, wenn eine Eingabedatei nicht in den Speicher gemappt werden konnte.

4.6 Anwendungsbeispiel

Listing 12 zeigt ein einfaches Anwendungsbeispiel der cintelhex-Bibliothek. Der Einfachheit halber zeigt die Variable `d` hier nur auf einen dynamisch alloziierten Speicherbereich, könnte jedoch auch ebenso gut auf den per `mmap` abgebildeten Speicher des ZPU-Boards zeigen.

4.7 Besonderheiten der Umsetzung

Erste Versionen der cintelhex-Bibliothek versuchten, den Zielspeicherbereits byteweise zu befüllen. Während dies bei Tests an dynamisch alloziiertem RAM kein Problem darstellte,

Listing 12: Anwendungsbeispiel der cintelhex-Bibliothek

```
1 #include <stdlib.h>
2 #include <cintelhex.h>
3
4 int main()
5 {
6   ihex_recordset_t* r = ihex_rs_from_file("input.hex");
7   void*             d = malloc(8192);
8
9   if (r != NULL)
10  {
11    return ihex_mem_copy(r, d, 8192, IHEX_WIDTH_32BIT,
          IHEX_ORDER_BIGENDIAN);
12  }
13  return ihex_errno();
14 }
```

Daten	0B	0B	0B	98	B0	2D	0B	0B	...
Adresse	403	402	401	400	407	406	405	404	...

Abbildung 8: Übersetzung von Binärdaten in lauffähige ZPU-Programme.

wurde schnell deutlich, dass diese Art des Zugriffs bei dem Speicher des ZPU-Boards nicht funktioniert. Listings 13 und 14 verdeutlichen das Problem.

Weiterhin fiel auf, dass das geladene Programm von der ZPU nur dann korrekt ausgeführt werden konnte, wenn die Intel HEX-Daten als 32-Bit-Blöcke in *Big Endian*-Reihenfolge interpretiert wurden. Abbildung 8 verdeutlicht diesen Zusammenhang.

Die cintelhex-Bibliothek berücksichtigt diesen Zusammenhang. So akzeptiert die Methode `ihex_mem_copy` einen Parameter `w`, über welchen die Wortbreite angegeben werden kann, sowie einen Parameter `o`, welcher die Byte-Reihenfolge beschreibt. Listing 15 beschreibt den Algorithmus zum Laden eines Intel HEX-Datensatzes in den ZPU-Speicher.

Listing 13: Unerwartetes Verhalten des ZPU-Speichers bei 8-Bit-Zugriff

```
1 uint8_t *d; // d zeigt auf den gemappten Geraetespeicher
2
3 d[0x00] = 0x0b;
4 printf("%02x \n", d[0x00]); // Gibt "0b" aus (korrekt).
5
6 d[0x01] = 0x98;
7 printf("%02x \n", d[0x00]); // Gibt "00" aus (sollte immer noch "0b"
      sein)!
8 printf("%02x \n", d[0x01]); // Gibt "98" aus (korrekt).
```

Listing 14: Korrektes Verhalten des ZPU-Speichers bei 32-Bit-Zugriff

```
1 uint8_t *d; // d zeigt auf den gemappten Geraetespeicher
2
3 *((uint32_t*) d) = 0x0000980b;
4
5 printf("%02x \n", d[0x00]); // Gibt "0b" aus (korrekt).
6 printf("%02x \n", d[0x01]); // Gibt "98" aus (korrekt).
```

Listing 15: Algorithmus zum Laden eines Intel HEX-Datensatzes in den ZPU-Speicher

```
1 ihex_width_t     w = IHEX_WIDTH_32BIT;     //<! Wortbreite in Byte
2 ihex_byteorder_t o = IHEX_ORDER_BIGENDIAN; //<! Byte-Reihenfolge
3 uint8_t          j, l;                     //<! Laufvariablen
4 ihex_record_t   *x;                        //<! Ein IHEX-Datensatz
5
6 for (j = 0; j < x->ihr_length; j += w)
7 {
8   uint32_t  v      = 0;
9   uint32_t *target = (uint32_t*) (d + address + j);
10
11  for (l = 0; (l < w) && (j + l < x->ihr_length); l ++)
12  {
13    v += x->ihr_data[j+l] << (8 * ((o == IHEX_ORDER_BIGENDIAN) ? ((w -
          1) - l) : l));
14  }
15  *(target) = v;
16 }
```

5 Programmierung der ZPU: Die libzpu

Die *libzpu* stellt die für Anwenderprogramme benötigten Funktionen des Treibers zur Verfügung. Die Schnittstelle wurde zur Abstrahierung der Logik für Anwenderprogramme erstellt. Dieser Abschnitt beschreibt die Bibliotheksfunktionen der zpulib. Anwendungsbeispiele finden sich im Abschnitt 6.

Die Quelltexte für die Bibliothek finden sich auf dem beigelegten Datenträger oder aber auf Github: `https://github.com/olivererxleben/zpulib`.

5.1 Installation

Um die Bibliothek auf einem Linux-System zu installieren muss in das Wurzelverzeichnis der Quellen gewechselt werden und dort über ein Unix-übliches Makefile ausgeführt werden.

Listing 16: Installation der zpulib-Bibliothek

```
1 # in directory
2 > ./configure
3 > make
4 > make install
```

5.2 Typdefinitionen und Variablen

Die zpulib verwendet zur Fehlerbehandlung und für Funktionsrückgaben zwei eigene Typen und zwei globale Variablen. Das Listing 17 zeigt dessen Deklaration in der zpu.h.

Listing 17: zpulib Typen und Variblen

```
28 // TYPE DEFINITIONS
29
30 typedef unsigned int uint_t;
31 typedef uint_t       zpu_error_t;
32
33 // GLOBAL VARIABLES
34
35 #ifdef ZPU_C
36 static zpu_error_t zpu_last_errno = 0;
37 static char*       zpu_last_error = NULL;
38 #else
39 extern zpu_error_t zpu_last_errno; //!< Error code of last error.
40 extern char*       zpu_last_error; //!< Description of last error.
41 #endif
```

`zpu_error_t` wird beispielsweise als Rückgabetyp von der Funktion `zpu_load_from_file` verwendet.

5.3 Schnittstellenbeschreibung

Die Methoden der libzpu können durch das Inkludieren der `zpu.h` in ein Anwenderprogramm benutzt werden. Die *zpulib* benutzt zudem Funktionen der *libcintelhex* um Hex-Dateien zu laden. Nachfolgend werden die Funktionen der zpulib beschrieben.

zpu_error_t zpu_load_from_file(char* filename); Die Methode wird verwendet um eine neues Programm in die ZPU zu laden. Vor dem Laden des Programms wird die ZPU in in den sog. *reset*-Modus gesetzt. Erst danach wird das zu ladene Programm von der Intel Hex Bibliothek geparsed und in den ROM der ZPU kopiert. Die Programm-Datei muss vom Typ INTEL Hex sein.

int zpu_stop(); Versetzt die ZPU in den *reset*-Modus. Liefert `ZPU_ERR_OK` bei Erfolg zurück. Andernfalls einen Error Code.

int zpu_start(); Versetzt die ZPU aus denm *reset*-Modus in den regulären Betriebszustand zurück. Die Methode liefert `ZPU_ERR_OK` bei Erfolg zurück. Andernfalls einen Error Code.

zpu_error_t zpu_errno(); Eine Hilfsmethode zum Debuggen von Anwenderprogrammen. Sie liefert die Nummer des zuletzt aufgetretenen Fehlers zurück.

char* zpu_error(); Eine Hilfesmethode zum Debuggen von Anwenderprogrammen. Sie liefert den zuletzt aufgetretenen Fehler zurück.

5.4 Fehlercodes

Neben Typen, Variablen und Funktionen definiert die zpulib auch Fehlercodes, welche beim Debugging von Anwenderprogrammen genutzt werden können. Folgende Fehlerkonstanten werden in der `zpu.h` deklariert:

#define ZPU_ERR_OK 0x00 OK, bzw. 0, wird zurückgegeben wenn kein Fehler aufgetreten ist.

#define ZPU_ERR_PARSEERROR 0x01 Dieser Fehlercode wird von der zpulib zurückgegeben, wenn ein ZPU-Programm von der libchintelhex nicht analysiert werden konnte.

#define ZPU_ERR_DEVFILEOPEN 0x02 Der Fehlercode tritt auf wenn ein ZPU-Programm nicht geöffnet werden konnte. Zum Beispiel wenn die Datei nicht gefunden wurde.

#define ZPU_ERR_MMAP 0x03 Der MMAP-Fehlercode wird zurückgeliefert, wenn das Mapping der ZPU fehlgeschlagen ist.

#define ZPU_ERR_MEMCOPY 0x04 Bei dieser Fehlerkonstanten handelt es sich um den Fehler der beim Kopieren in den RAM der ZPU geschehen kann.

#define ZPU_ERR_COULDNOTSTOP 0x05 Der Fehler wird zurückgegeben wenn die ZPU nicht in den *reset*-Modus versetzt werden konnte.

#define ZPU_ERR_COULDNOTSTART 0x06 Wird zurückgegeben, wenn die ZPU nicht aus dem *reset*-Modus zurück in den normalen Betriebszustand versetzt werden konnte.

Anzumerken ist, dass alle Fehlercodes in der Funktionsimplementierung Verwendung finden (siehe weitere Abschnitte).

6 Anwenderprogramme

Zur Steuerung des ZPU-Boards wurden schließlich die Werkzeuge *zpuload* und *zpuio* entwickelt. Das Paket liegt dieser Arbeit auf Datenträger bei, und befindet sich im Verzeichnis `apps/zputools`.

6.1 Installation

zpuload und *zpuio* gehören beide zum *zputools*-Paket, können also gemeinsam installiert werden. Wie bei den anderen im Rahmen dieser Arbeit entstandenen Pakete, reicht auch hier der übliche Dreizeiler zur Installation aus:

Listing 18: Installation der zputools-Werkzeuge

```
1 > ./configure # optional mit --prefix=...
2 > make
3 > make install
```

Bei der Installation werden die ausführbaren Programme *zpuload* und *zpuio* (abhängig vom prefix) nach `/usr/local/bin` kopiert.

Die *zputools* benötigen einen GCC, die C-Standardbibliothek, sowie eine installierte cintelhex-Bibliothek (siehe Abschnitt 4) und eine installierte zpu-Bibliothek (siehe Abschnitt 5). Das *configure*-Skript überprüft diese Abhängigkeiten automatisch.

6.2 Programme laden mit zpuload

Das *zpuload*-Programm verwendet die Funktionen der *cintelhex*- und *zpu*-Bibliotheken, um ein neues Programm in den ZPU-Prozessor zu laden.

Hierzu wird per `zpu_load_from_file(file)`-Methode ein neues Programm in den RAM der ZPU geladen (diese Methode versetzt die ZPU zuvor implizit in des Reset-Modus und startet sie anschließend wieder).

Listing 19: Verwendung des zpuload-Werkzeugs

```
1 > /usr/local/bin/zpuload input.hex
2 Loading program from "input.hex"...
3 Loaded program.
```

6.3 Ein- und Ausgabe mit zpuio

Das *zpuio*-Werkzeug dient dazu, Ein- und Ausgaben an den ZPU-Prozessor zu übermitteln. Dieses Programm ist vor allem als interaktive Schnittstelle gedacht.

Listing 20: Verwendung des zpuio-Werkzeugs

```
1 > /usr/local/bin/zpuio
2 output (11) > Hello World
3 input > abcd
4 output (4) > Abcd
5 input >
```

Neben der Verwendung des *zpuio*-Werkzeugs kann auch direkt in/aus der Gerätedatei */dev/zpu* geschrieben und gelesen werden.

7 Fazit

Mit den im Rahmen dieser Arbeit entwickelten Kernel-Module, Bibliotheken und Anwenderprogramme steht eine einfach benutzbare und gut erweiterbare Toolchain zur Arbeit mit einem ZPU-Board zur Verfügung.

Um die Arbeit mit diesen Werkzeugen weiter zu vereinfachen, wäre in einer fortführenden Arbeit beispielsweise möglich, über entsprechende udev-Regeln das Kernelmodul automatisch zu laden, die entsprechenden Gerätedateien zu erstellen. Zudem könnte über eine udev-Regel bereits beim Booten durch einen Aufruf des *zpuload*-Programms eine initiale Firmware in die ZPU geladen werden.

Um die Upgrade-Fähigkeit des Kernel-Moduls über mehrere Kernel-Versionen hinweg sicherzustellen, böte sich zudem die Verwendung des von Dell entwickelten DKMS-Frameworks an.[10]

[10] Vgl. Lerhaupt (2003): Linux Journal.

Literatur

eCos - open source real time operating system. (URL: http://ecos.sourceware.org).

FPGA Soft Core. (URL: http://www.mikrocontroller.net/articles/FPGA_Soft_Core).

Intel Corporation: Hexadecimal Object File Format Specification. 1988 (URL: http://microsym.com/editor/assets/intelhex.pdf) – Zugriff am 24. 2. 2013.

Jürgen Quade, Eva-Katharina Kunst: Linux Treiber Entwicklen. 1. Auflage. dpunkt., 2004 (URL: ftp://ftp.tm.informatik.uni-frankfurt.de/pub/papers/ir/kernelbook.pdf).

Lang, Bernhard: Hardwarenahe System- und Treiberprogrammierung. HS Osnabrück, 2012 – Script.

Lerhaupt, Gary: Kernel Korner - Exploring Dynamic Kernel Module Support (DKMS). In: Linux Journal September 2003 (URL: http://www.linuxjournal.com/article/6896) – Zugriff am 24. 2. 2013.