Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2320

PCIe Treiber für ein Linux-System

Alexander Henning

Studiengang:	Elektrotechnik und Informationstechnik
Prüfer:	Prof. Dr. Sven Simon Prof. Dr. P. Levi
Betreuer:	Dipl.-Ing. Jürgen Hillebrand
begonnen am:	05. Oktober 2010
beendet am:	06. April 2011
CR-Klassifikation:	B.4.1, B.4.3, D.4.0, D.4.9

PCIe Treiber für ein Linux-System

Author: Alexander Henning
Supervisor: Dipl.-Ing. Jürgen Hillebrand

Abteilung Parallele Systeme
Institut für Parallele und Verteilte Systeme
Universität Stuttgart

Inhaltsverzeichnis

1 Einführung **1**

2 Aufbau von Linux **5**
2.1 Geschichte . . . 5
2.2 Kernel . . . 7
2.2.1 Mikrokernel . . . 8
2.2.2 Hybridkernel . . . 9
2.2.3 Monolithischer Kernel . . . 10
2.3 Linux-Kernel . . . 11
2.3.1 Interface für die System-Aufrufe . . . 12
2.3.2 Prozessverwaltung . . . 12
2.3.3 Speicherverwaltung . . . 15
2.3.4 I/O-Subsystem . . . 16
2.3.5 Geräte-Treiber . . . 17

3 PCIe - Schnittstelle **21**
3.1 Einführung . . . 21
3.2 PCIe . . . 23
3.3 PCIe Transaktionen . . . 25
3.3.1 Memory Transactions . . . 25
3.3.2 I/O Transactions . . . 25
3.3.3 Configuration Transactions . . . 26
3.3.4 Message Transactions . . . 26
3.4 PCIe Übertragungsschichten . . . 26
3.4.1 PCIe Transaktionsschicht . . . 28
3.4.2 PCIe Data Link Schicht . . . 29
3.4.3 PCIe Bitübertragungsschicht . . . 32

4 Treiber und Anwendung **37**
4.1 Registerbeschreibung . . . 38
4.2 Treiber . . . 39
4.2.1 Funktion `probe` . . . 42
4.2.2 Funktion `remove` . . . 46
4.2.3 Funktion `open` . . . 46
4.2.4 Funktion `release` . . . 47
4.2.5 Funktion `mmap` . . . 47
4.2.6 Funktion `read` . . . 48

4.2.7 Funktion `write` . . . 49
4.2.8 Funktion `ioctl` . . . 49
4.2.9 Funktion `ML_do_full_duplex_dma` . . . 53
4.3 Anwendung, grafische Benutzeroberfläche . . . 53
4.3.1 Aufbau des Programms . . . 54
4.3.2 Bestimmung des Datendurchsatzes . . . 57

5 Zusammenfassung und Ausblick **59**
5.1 Zusammenfassung . . . 59
5.2 Ausblick . . . 59

Literaturverzeichnis **61**

Abbildungsverzeichnis **62**

Tabellenverzeichnis **64**

Akronyme **67**

Kurzfassung

Am Institut für Parallele und Verteilte Systeme - Abteilung Parallele Systeme wird in mehreren Projekten ein Prototyp zur parallelen Berechnung elektrischer Feldgrößen mit Hilfe der Finite-Differenzen-Methode im Zeitbereich erstellt. Die zur Berechnung benötigten Algorithmen werden dazu in anderen Teilprojekten für die FPGA-Entwicklungsplattform in VHDL entwickelt. Um die zur Berechnung erforderliche große Menge an Daten und Ergebnisse möglichst schnell und effizient zwischen der FPGA-Entwicklungsplattform und dem PC austauschen zu können, soll dazu die PCIe-Schnittstelle verwendet werden.

Im Rahmen dieser Arbeit wurde ein Linux Treiber für eine Virtex 5 FPGA-Entwicklungspattform mit PCIe-Schnittstelle und eine Anwendung implementiert. Der Treiber soll die Steuerung, die Konfiguration und den Datenaustausch zwischen der FPGA-Entwicklungsplattform und dem PC über die PCIe-Schnittstelle ermöglichen. Da das System eine schnelle Übertragung großer Datenmengen zwischen PC und FPGA-Entwicklungsplattform erfordert, soll weiterhin der Datenaustausch mittels Speicherdirektzugriff (DMA) realisiert werden. Die zu implementierende Anwendung soll zudem die Datenübertragung und Verifikation des Treibers und der Vermessung der erzielten Übertragungsbandbreiten ermöglichen.

1 Einführung

In dieser Studienarbeit soll ein Linux Treiber für die FPGA-Entwicklungsplattform ML506 der Firma Xilinx mit einem Virtex-5 FPGA zwecks Datenaustausch über die PCIe (Peripheral Component Interconnect Express)-Schnittstelle [7] mit dem PC (Personal Computer) entwickelt werden. Das gegebene System, für das der Treiber entwickelt werden soll, zeigt die Abbildung 1.1. Aus der Abbildung kann entnommen werden, dass

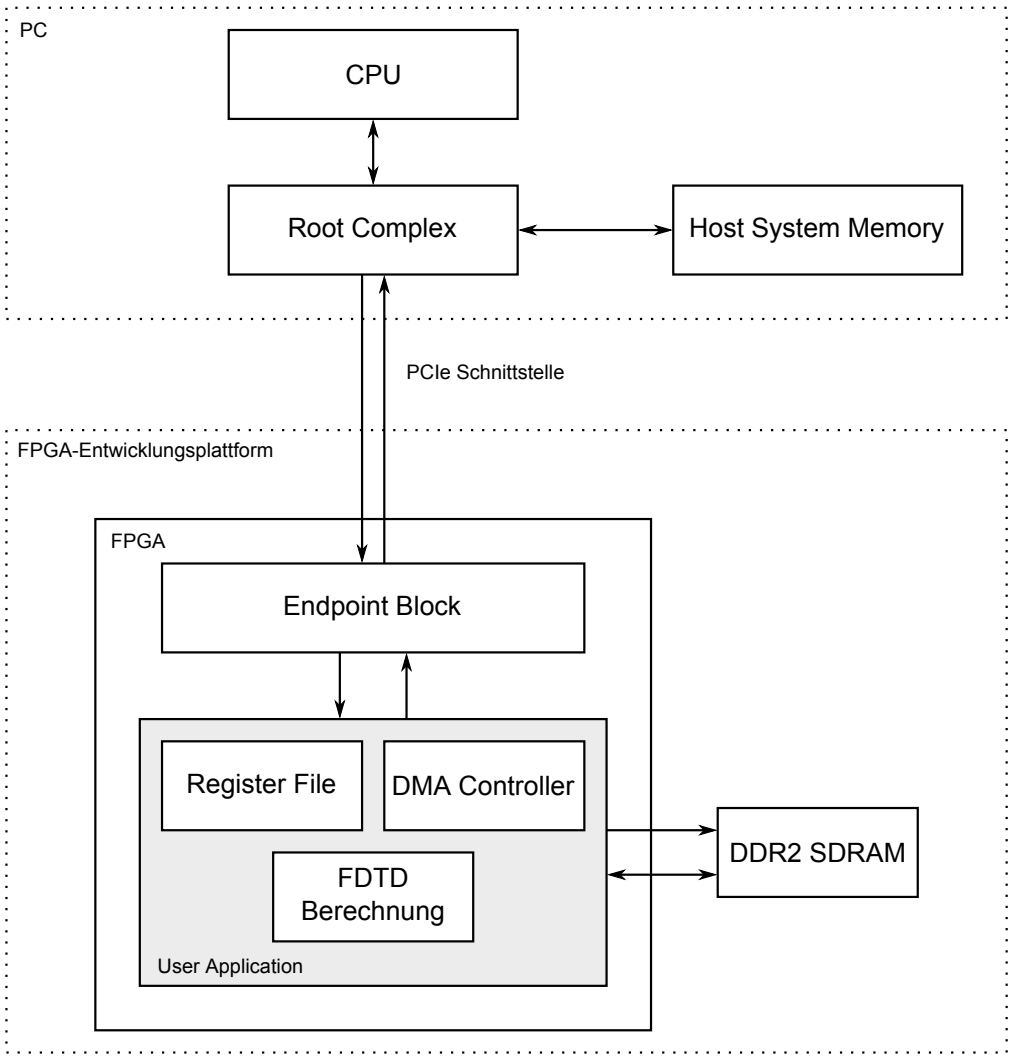

Abbildung 1.1: Blockdiagramm des Systems

die FPGA-Entwicklungsplattform mit dem PC über die PCIe-Schnittstelle verbunden ist. Der Datenaustausch erfolgt dabei unmittelbar zwischen dem Root Complex des PCs und dem Endpointblock [8] des FPGAs über einen x1 Link. Am Endpointblock sind zwei weitere logische Blöcke angeschlossen, der Register-File-Block und der DMA Controller Block, die gemäß der Anwendungsbeschreibung XAPP859 [9] implementiert sind. Der Register-File-Block dient der Einstellung des DMA (Speicherdirektzugriff, engl. Direct Memory Access) Controllers, der die Datenübertragung mittels Speicherdirektzugriffs [4, S. 440ff.] zwischen dem DDR2 Speicher und dem Host System Memory durchführt. Der in der Abbildung vorhandene FDTD [1] -Berechnungsblock wird im Rahmen eines anderen Projektes implementiert, weshalb in dieser Arbeit nicht näher auf diesen Block eingegangen wird. Die Durchführung der parallelen Berechnungen durch den FDTD-Berechnungsblock setzt einen breitbandigen Übertragungsweg zwischen dem PC und FPGA-Entwicklungsplattform voraus, da dabei große Mengen von Modell- und Ergebnisdaten ausgetauscht werden müssen.

Die große Menge an Daten erfordert eine besonders effektive Methode der Datenübertragung, weshalb in dieser Arbeit die Methode eines Speicherdirektzugriffs verwendet wird. Der Vorteil der Verwendung des Speicherdirektzugriffes liegt in der bestmöglichen Ausnutzung der Übertragungsbandbreite der PCIe-Schnittstelle, da die Daten nicht von der CPU (Central Processing Unit) übertragen werden müssen.

Der in dieser Studienarbeit implementierte Treiber soll weiterhin die

- Allokation und Initialisierung von Speicherbereichen im Kernel,
- Einstellung der Datenübertragung,
- Einbindung des Treibers in den Kernel des Betriebssystems zur Laufzeit,
- Kontrolle der Datenübertragung ermöglichen und
- eine definierte Schnittstelle für die Benutzeranwendungen auf dem PC bieten.

Zusätzlich zu den bereits aufgelisteten Anforderungen soll der Treiber den Anwendungen alle Funktionalitäten der Hardware zur Verfügung stellen und dabei sparsam mit den Systemressourcen umgehen.

Die erstellte Benutzeranwendung für den PC soll dem Benutzer die Steuerung und die Kontrolle der Datenübertragung ermöglichen. Dabei greift sie auf alle Funktionen des Treibers und damit auch auf alle bereits implementierten Funktionen der Hardware zurück, wodurch die Funktionalität der Hardware und des Treibers demonstriert werden kann. Eine weitere Anforderung an die Anwendung ist, dass sie die Übertragungsbandbreite zur Überprüfung der Effizienz des Treibers messen kann.

Die fünf Kapitel umfassende Arbeit gibt eine ausführliche Beschreibung über die erfolgte Implementierung des Treibers an. Im nachfolgenden Kapitel 2 werden zunächst wichtige Details des Linux Betriebssystems erläutert, die zur Implementierung des Treibers beachtet werden müssen. Zusätzlich erfolgt auch ein kurzer geschichtlicher Exkurs in das Linux Betriebssystem. Das Kapitel 3 beschreibt die PCIe-Schnittstelle. Im Kapitel 4 wird der entwickelte Treiber beschrieben und die Benutzeranwendung vorgestellt.

Abschließend werden im Kapitel 5 die Ergebnisse von Durchsatzmessungen vorgestellt, sowie eine Zusammenfassung und ein Ausblick gegeben.

2 Aufbau von Linux

2.1 Geschichte

Mit dem Begriff Linux sind meistens Mehrbenutzer-Betriebssysteme gemeint, die auf dem Linux-Kernel basieren. Dieser Begriff bezeichnet aber nur die Kernkomponente des Betriebssystems, den sog. „Kernel“. Zu dem Linux-Betriebssystem gehören neben dem Kernel selbst noch die System- und Anwendersoftware. Diese zusätzliche Software, vereint mit dem Kernel und der entsprechenden Installationsroutine, wird zu einer sogenannten Distribution zusammengefasst. Zumindest bei einigen kommerziellen Distributionen gehören die Handbücher und andere Dokumentation zum Lieferumfang. Zu den bekanntesten Distributionen zählen unter anderen Debian, mit seinen Abkömmlingen Knoppix und Ubuntu, Red Hat, Fedora, OpenSUSE, Mandriva und Gentoo.

Linux ist ein Unix-ähnliches Betriebssystem. Im Gegensatz zu den proprietären Unix-Systemen ist der vollständige Quellcode verfügbar und darf frei verändert und kopiert werden. Die Unix-Entwicklung startete 1965 und wurde von MIT, BELL und General Electric vorangetrieben. Das neue Betriebssystem war für ein Einsatz an damaligem Grossrechner MULTICS vorgesehen. Die ursprünglichen Unix-Programme wurden alle in Assembler geschrieben. Im Laufe der Zeit spalteten sich mehrere Gruppen ab und verfolgten eigene Ziele, meistens bedingt durch den Wechsel auf andere Zielplattformen. Die Programmierung in Assembler war sehr aufwendig und somit wurden eigens für die Entwicklung des Betriebssystems neue Programmiersprachen entwickelt. Die erste neue Programmiersprache war „B“, die stark durch die BCPL (Basic Combined Programing Language) beeinflusst wurde. Der Plattformwechsel im Jahre 1971 erforderte eine neue, diesmal eine byteorientierte Programmiersprache. Dennis Ritchie entwickelt die „C“-Programmiersprache. Diese Sprache zeichnet sich durch die Systemnähe aus und sie wird auch noch heute verwendet.

Das Jahr 1973 markiert die Geburt eines Multiuser-Multitasking-Betriebssystems, das vollständig in C geschrieben wurde. Im weiteren Verlauf wurde Unix weiterentwickelt und wegen der großen Anzahl von unterschiedlichen Entwicklerfirmen auf unterschiedliche Plattformen portiert. Durch die Vergabe von Quellcode-Lizenzen an Universitäten konnte zudem das Unix-Derivat BSD (Berkeley Software Distribution) entwickelt werden. BSD zeichnet sich durch die Implementierung und Integration des TCP/IP Stacks und der Berkeley Socket API (Application Programming Interface) aus. Dadurch wurden Standards geschaffen und umgesetzt, die bis heute ihre Gültigkeit besitzen und angewendet werden.

Die Entstehungsgeschichte des Linux Betriebssystemkerns ist eng mit dem Namen Linus Torvalds verbunden. Im Frühjahr 1991 begann er mit der Entwicklung seines eigenen

Betriebssystems. Er veröffentlichte seine erste Entwicklerversion am 17.09.1991 und lud interessierte Entwickler zur Mitarbeit ein. Zuerst wollte Linus Torvalds die kommerzielle Benutzung verbieten, aber erst die Lizenzierung des Linux-Kernels unter der GNU (GNU's Not Unix) GPL (General Public License) ermöglichte weite Verbreitung, auch im kommerziellen Bereich.

Die GNU General Public License wurde 1982 von Richard Stallman erarbeitet. Ebenfalls in diesem Jahr gründete er ein GNU-Projekt mit dem Ziel ein komplett freies Betriebssystem zu entwickeln. Dadurch wollte er der Weitergabe der Software in binärer Form entgegenwirken. Sein Projekt ergab eine große Fülle an Programmen, die für ein komplettes Betriebssystem notwendig sind. Die Entwicklung des dazugehörigen Betriebssystemkerns, des Kernel war aber noch nicht ausreichend voran geschritten. Das Betriebssystem basiert heute auf dem Linux-Kernel. Die Linux-Systeme wiederum benutzen für die wichtigsten Systemkomponenten und die Userspace-Software die GNU lizenzierte Programme.

Mit der 1998 erschienenen Kernel-Version 2.2 unterstützte das Betriebssystem mehrere Prozessoren. Mit den neuen Versionen 2.4 und 2.6 wurden immer mehr neue Features eingebaut.

- Unterstützung großer Arbeitsspeicher (bis zu 64 GByte)
- Unterstützung für Plug-and-Play-Geräte
- SATA-Unterstützung
- Bluetooth-Unterstützung
- Zahlreiche neue Dateisysteme (ReiserFS, Ext4, JFS)
- bessere Scheduler

Nachteilig ist, dass mit den größeren Versionsprüngen die Schnittstellen des Kernels sich zum Teil erheblich verändern. Viele Treiber müssen daraufhin angepasst werden und manchmal ist es sogar einfacher sie stattdessen neu zu entwickeln.

Auch noch heute leitet und überwacht Linus Torvalds die Entwicklung des Kernels. Die gemachten Veränderungen, meistens Verbesserungen, am Kernel werden an die verantwortlichen Kernel-Entwickler weitergereicht. Die hochqualitativen Anpassungen und von solchen nur die, die Linus Torvalds zusagen, werden dann von ihm in den offiziellen Kernel übernommen.

Für jeden Einsatzbereich gibt es entsprechende Linux-Varianten. Die Palette reicht von Desktop PC über die Mobiltelefone und Router bis hin zu den Supercomputern. Die Verbreitung in den einzelnen Einsatzbereichen ist jedoch unterschiedlich. Die Linux-Systeme im Servereisatz sind weit verbreitet, in Heimanwender-Bereich spielen sie nur eine geringe Rolle. Der gute Ruf, kostenfreie Verfügbarkeit und immer einfachere Handhabe tragen dazu bei, dass dieser Heimanwender-Bereich stetig wächst.

Auf dem freien Betriebssystem setzen weitere Großprojekte an, wie z.B. KDE, Gnome und X-Server von X.Org-Projekt. Diese Projekte haben erheblich dazu beigetragen, dass dieses Betriebssystem für die Benutzer einfacher und zugänglicher wurde.

2.2 Kernel

Der Kernel oder Systemkern ist ein zentraler Bestandteil des Betriebssystems und ist die unterste Softwareschicht. Der Kernel bildet die hardwareabstrahierende Schicht zwischen der Hardware und übrigen Softwarekomponenten. Da die Hardwarebasis sehr unterschiedlich sein kann, muss der Kernel alle Variationen abdecken. Da dies nicht möglich bzw. sinnvoll ist, muss der Kernel über Mechanismen verfügen, die eine Einbindung externer Module erlaubt. Der Systemkern bietet definierte Schnittstellen sowohl für die Treiber als auch für die Software des Benutzers, die unabhängig von der Rechnerarchitektur ist. Die grundlegenden Aufgaben des Kernels sind:

- Kontrolle über die vorhandenen Ressourcen wie CPU, Speicher, Geräte.
- Zuteilung der Ressourcen an die Anwenderprogrammen, z.B. die Rechenzeit.
- Bereitstellung einer Softwareschnittstelle für die Anwenderprogramme.
- Hierarchische Strukturierung der Ressourcen, z.B. Dateisysteme, Netzwerkprotokolle.
- Arbitrierung von Zugriffskonflikten und Bereitstellung von Warteschlangen bei knappen Ressourcen.
- Überwachung von Zugriffsrechten auf Dateien und Geräte bei Mehrbenutzersystemen.
- Speicher- und Prozessverwaltung.
- Virtualisierung: Verbergen der Komplexität der Maschine vor dem Anwender.

Der Kernel selber ist in Schichten aufgebaut. Das Schichtenmodell sieht vor, dass es eine unterste Schicht gibt, hier z.B. die Hardwareabstraktionsschicht. Die höher liegenden Schichten bauen auf den Funktionalitäten der jeweils unter ihnen liegenden Schichten auf. Die Unterteilung in Schichten erfolgt z.B. nach den Funktionalitäten oder Aufgaben.

Nach dem Umfang der im Kernel enthaltenen Schichten unterscheidet man zwischen drei verschiedenen Kerneltypen:

1. Monolithischer Kernel, alle Funktionen sind in dem Systemkern implementiert.
2. Mikrokernel, nur die grundlegenden Funktionen sind im Systemkern integriert. Die restlichen Funktionen werden in getrennten Prozessen ausgeführt.
3. Hybridkernel, ist eine Mischung von monolithischem Kernel und dem Mikrokernel.

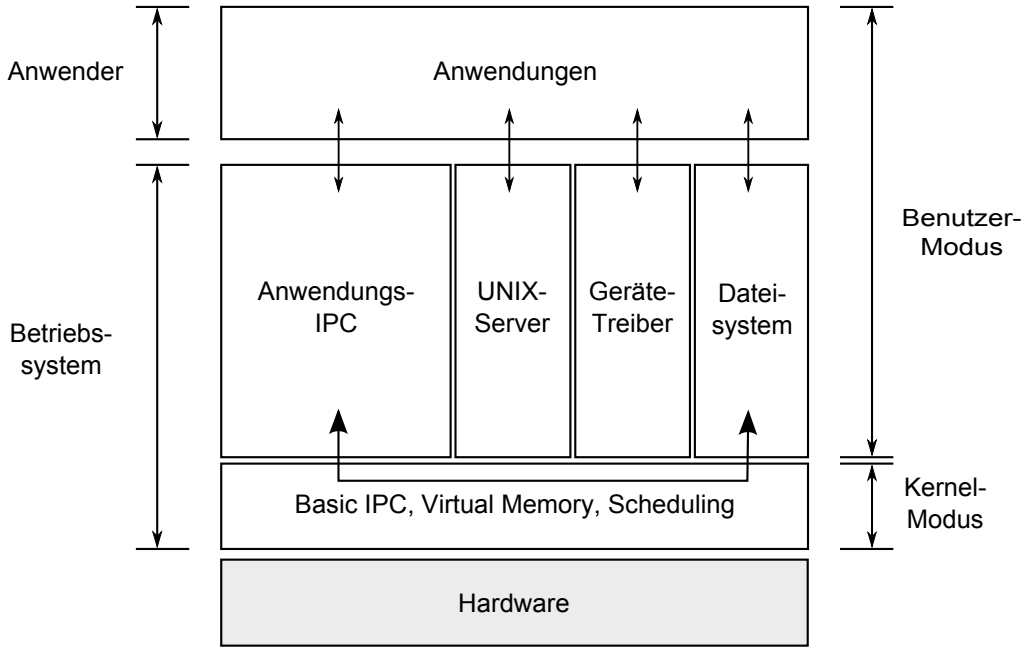

Abbildung 2.1: Mikrokernel Betriebssysteme

2.2.1 Mikrokernel

Bei einem Betriebssystem mit einem Mikrokernel werden nur die grundlegendsten Funktionen im Kernel implementiert. Alle anderen Teile des Betriebssystems laufen als getrennte Prozesse im Benutzer-Modus und sie stehen somit den Benutzerprogrammen zur Verfügung, oder sie werden als Programmbibliothek in die Benutzerprogramme mit eingebunden. Die Eigenschaft des Mikrokernels, die Teile des Betriebssystem auslagern zu können, begünstigt die Entwicklung von verteilten Betriebssystemen. Ein Nachteil der auf dem Mikrokernel basierten Betriebssystemen ist, dass ein Kontextwechsel öfter als bei anderen Betriebssystemen erfolgt, da die Teile des Betriebssystems als eigenständige Prozesse laufen. Die verschiedenen Teile des Betriebssystems können zur Laufzeit, z.B. wegen eines Absturzes, neu gestartet oder gänzlich ausgetauscht werden. Der Absturz einer einzelnen Komponente bedeutet nicht zwangsläufig den Absturz des ganzen Systems. Neben der schwer zu optimierenden Koordination der Kernel-Prozesse ist die Minimierung der mehrfachen Kopiervorgänge bei den Kontextwechseln eine der großen Herausforderungen beim Mikrokernel-Design.

Die Abbildung 2.1 stellt die Struktur der auf dem Mikrokernel basierten Betriebssystemen dar. Solche Systeme besitzen eine große Verbreitung in den Anwendungsbereichen mit hohen Anforderungen an die Robustheit, Sicherheit und Zuverlässigkeit, wie z.B. bei Militär, Luft- und Raumfahrt oder Automatisierungs- und Medizintechnik. Die prominentesten Vertreter der Mikrokernelsysteme sind: GNU/Hurd, L4Linux, Minix, QNX, Singularity.

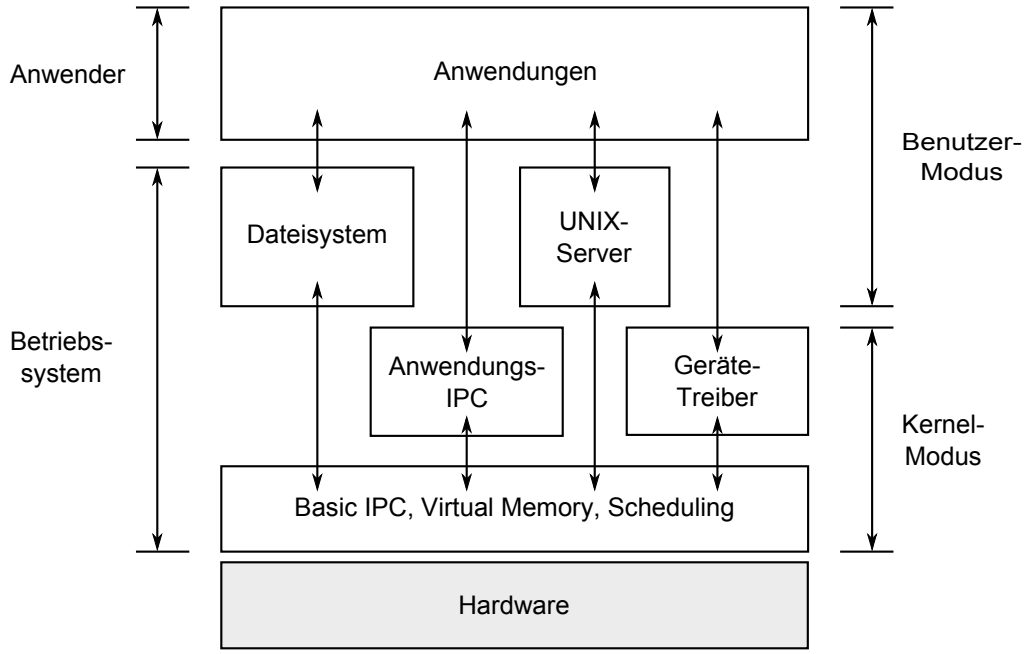

Abbildung 2.2: Hybridkernel Betriebssysteme

2.2.2 Hybridkernel

Ein Hybridkernel ist eine Mischung aus den Eigenschaften von Mikrokernel und einem monolithischen Kernel. Dabei werden einige zusätzliche Teile des monolithischen Kernels in den Kern mit aufgenommen, und dadurch ist es kein Mikrokernel mehr, jedoch auch noch kein vollwertiger monolithischer Kernel. Durch die Aufnahme der ausgewählten Funktionen in den Kernel, wie z.B. Grafiktreiber, erzielt man eine Steigerung der Leistungsfähigkeit bei der Darstellung von graphischen Elementen.

Jedes Betriebssystem, dass auf dem Hybridkernel aufgebaut ist, kann unterschiedliche Teilfunktionen im Kernel implementiert haben.

Die Aufnahme zusätzlicher Betriebssystemfunktionen in den Kernel bildet einen Vorteil gegenüber dem Mikrokernel, weil dadurch die Anzahl der Kontextwechsel reduziert wird und somit die Interprozesskommunikation vereinfacht wird. Diese Maßnahmen steigern die Geschwindigkeit des Kernels. Dieser Vorteil des Hybridkernels gegenüber dem Mikrokernel bringt gleichzeitig auch einen Nachteil mit sich. Dieser Nachteil äußert sich durch die Steigerung der Fehleranfälligkeit des gesamten Systems. Diese Fehleranfälligkeit ist geringer als bei einem monolithischen Kernel. Der Hybridkernel vereint nicht nur die Vorteile der beiden anderen Kernelarten, sondern auch deren Nachteile.

Die Abbildung 2.2 stellt eine mögliche Struktur des Hybridkernels dar. Die typischen Vertreter der Betriebssysteme mit dem Hybridkernel sind: alle auf Windows NT basierten Systeme sowie BeOS, MacOS X.

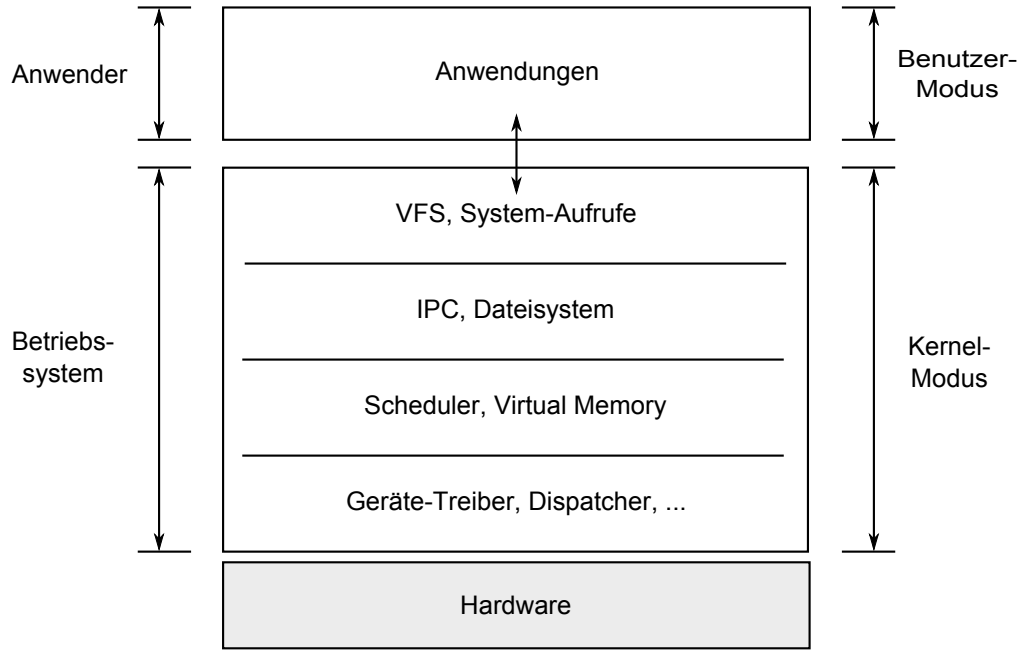

Abbildung 2.3: Betriebssysteme mit dem monolithischen Kernel

2.2.3 Monolithischer Kernel

Der monolithische Kernel ist die Kernelart, mit der die Entwicklung von Betriebssystemen began. Gegenüber anderen bereits erwähnten Kernelarten, besitzt der monolithische Kernel ein einfacheres Design. In einem monolithischen Kernel sind alle Funktionen und die Treiber für die Hardwarekomponenten direkt eingebaut. Monolithische Kernel stellen mit sich einen einzelnen Prozess dar, der in einem Adressraum abläuft. Der Kernel ist somit ein statisches Programm. Die ganzen Funktionen und die Kernel-Dienste laufen alle in einem großen Kernel-Adressbereich. Die Interprozesskommunikation innerhalb des Kernels lässt sich wegen des gemeinsamen Adressraums leicht implementieren, wobei die Funktionen im Kernel direkt aufgerufen werden können. Die erforderlichen Treiber für die Hardware müssen in dem Kernel enthalten sein. Im Vergleich zu den anderen Kernelarten erzielt man hierdurch einen Geschwindigkeitsvorteil, wenn die Treiber nicht als eigenständige Programme laufen.

Dadurch, dass alle Kernel-Dienste und die Treiber für die Hardware in einem Adressbereich laufen, sind monolithische Kernel fehleranfälliger. Es besteht eine große Gefahr, dass z.B. ein abgestürzter Kernel-Dienst das gesamte System abstürzen lässt.

Der Abbildung 2.3 kann man die Struktur des Betriebssystems entnehmen, das auf einem monolithischen Kernel aufbaut. Einige der Betriebssysteme, die auf monolithischen Kernel aufbauen, sind: MS-DOS (Microsoft Disk Operating System), Unix, BSD, OS/2.

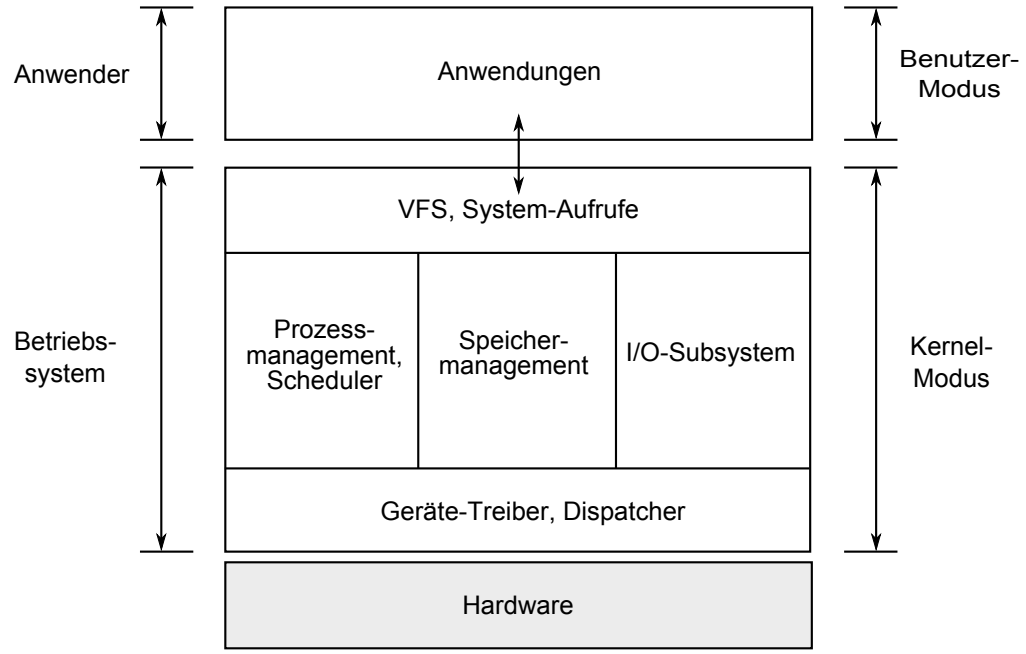

Abbildung 2.4: Linux Betriebssystem

2.3 Linux-Kernel

Der Linux-Kernel ist selbst ein monolithischer Betriebssystemkern. Er wird in einem Adressbereich im Kernel-Modus ausgeführt. Die Linux-Entwickler waren sich der mit dem monolithischen Kernel verbundenen Nachteilen bewusst und haben mehrere gute Ansätze des Mikrokernels umgesetzt. Der Linux-Kernel ist modular aufgebaut, unterstützt die Kernel-Threads, unterstützt präemptives Multitasking, selbst für Kernel Tasks, und bietet ein Interface mit dessen Hilfe man dynamisch zusätzliche Kernel-Module laden und entladen kann.

Die Abbildung 2.4 stellt die wesentlichen Komponenten des Betriebssystems Linux dar.

- Interface für die System-Aufrufe
- Prozessverwaltung
- Speicherverwaltung
- I/O-Subsystem
- Geräte-Treiber

Diese Komponenten werden nachfolgend näher vorgestellt.

2.3.1 Interface für die System-Aufrufe

Alle Anwendungen, die im Benutzer-Modus ausgeführt werden und die vom Betriebssystem zur Verfügung gestellten Dienste in Anspruch nehmen, müssen das Interface für die System-Aufrufe benutzen. Diese Schnittstelle wird über die Software-Interrupts realisiert. So kann die Anwendung auf die Dateien zugreifen oder z.B. die Rechte des Benutzers prüfen. Die Anwendungen, die Software-Interrupts auslösen, müssen über die übergebenen Parameter die zur Ausführung notwendige Information bereitstellen. Nach der Auslösung eines Interrupts führt dann der Kernel die entsprechende Interrupt-Service-Routine durch und gibt der Anwendung einen Rückgabewert zurück.

Die meisten Anwendungen werden von den Entwicklern mit Hilfe von Hochsprachen programmiert. In den Anwendungen werden z.B. die Funktionen aus den Bibliotheken benutzt. In solchen Funktionen werden dann die eigentlichen Systemcalls aufgerufen. Es ist aber auch möglich, dass die Entwickler die System-Aufrufe direkt aus der Anwendung einsetzen können.

Beim Ausführen einer Anwendung muss neben dem eigentlichen Code der Anwendung noch der Code der verwendeten Bibliotheken und der Kernelcode ausgeführt werden. Der Kernelcode wird z.B. ausgeführt, wenn die eingesetzte Bibliothek System-Aufrufe tätigt. Dabei lösen die System-Aufrufe Software-Interrupts aus, die vom Kernel abgearbeitet werden müssen.

Im Linuxkernel (2.6) sind rund 300 System-Aufrufe realisiert, die man alle in der Datei `<asm/unistd.h>` der Kernel-Sources nachschlagen kann.

Listing 2.1: Ausschnitt aus `linux/arch/x86/include/asm/unistd_32.h`

```
#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write              4
#define __NR_open               5
#define __NR_close              6
```

Der System-Aufruf mit der Nummer 1 beendet einen Rechenprozess. Mit dem System-Aufruf *fork*, dem Aufruf mit der Nummer 2, erzeugt man einen neuen Rechenprozess. System-Aufruf mit der Nummer 3 wird z.B. aufgerufen, wenn die Daten aus den Dateien oder von den Geräten gelesen werden sollen.

2.3.2 Prozessverwaltung

Ein weiterer Teil des Kernels ist die Prozessverwaltung. Die Prozessverwaltung trägt unter anderem dazu bei, dass mehrere Rechenrozesse quasi parallel auf einem Einprozessorsystem abgearbeitet werden können. Bei Systemen mit mehreren Prozessoren werden die auszuführenden Prozesse auf diese verteilt. Für den Kernel sind die Applikationen einfache Rechenprozesse. Jeder Rechenprozess besteht aus dem Codesegment und dem Datensegment. Beim Anlegen eines neuen Prozesses belegt das Betriebssystem mindes-

tens drei Speicherblöcke: für ausführbaren Code, für Daten und zuletzt für den Stack. Die Prozessverwaltung ist in der Lage den mehrfachen Verbrauch an Speicher zu vermeiden, wenn es z.B. mehrere Prozesse gibt, die den gleichen Code-Block verwenden.

Linux ist ein Multitasking-Betriebssystem. Seit der Kernel-Version 2.6 beherrscht Linux nicht nur präemptives Multitasking sondern auch präemptibles Multitasking. Beim präemptiven Multitasking stellt das Betriebssystem jedem lauffähigen Prozess nur einen Zeitschlitz zur Abarbeitung bereit. Nach Ablauf der zugewiesenen Zeit unterbricht der Kern den laufenden Prozess und startet den nächsten lauffähigen Rechenprozess. Weil die Zeitabschnitte sehr kurz sind, entsteht der Eindruck, dass die Programme parallel ablaufen. Die Wahl des nächsten rechenbereiten Prozesses erfolgt durch einen Scheduling-Algorithmus. Der Kernel verwaltet dazu zwei Listen:

- Liste der lauffähigen Prozesse: In dieser Liste stehen alle ablauffähigen Prozesse, die bei freien Prozessoren sofort abgearbeitet werden können. Dabei besteht kein Bedarf an weiteren Ressourcen. Wenn vor dem Ablauf der zugewiesenen Laufzeit der Prozess noch nicht fertig mit den Berechnungen ist, dann wird er wieder in die Liste der lauffähigen Prozesse eingetragen. Der Scheduler wählt den nächsten Prozess aus dieser Liste.

- Liste der „schlafenden“ Prozesse: In diese Liste werden alle Prozesse eingetragen, die z.B. auf die Daten aus dem Speicher warten, oder auf die langsame Peripherie angewiesen sind, und damit nicht lauffähig sind. Wenn die von dem Prozess angeforderte Ressource irgendwann zur Verfügung steht, dann wird der entsprechende Rechenprozess aus der Liste der „schlafenden“ Prozesse in die Liste der lauffähigen Prozesse verschoben. Der verschobene Prozess wird aber unter Umständen nicht sofort ausgeführt, sondern erst dann, wenn der Scheduler ihn aussucht.

Der Scheduler und die Speicherverwaltung sind die kritischsten Teile des Kernels. Deren Entwurf und die Implementierung beeinflussen die Entwicklung anderer Teile des Betriebssystems und damit die gesamte Leistungsfähigkeit des Systems.

Der Scheduler berechnet die Prioritäten aller Prozesse auf einmal. Die Berechnung der Prioritäten und damit die neue Positionierung der Prozesse wird erst dann ausgeführt, wenn alle Prozesse auf der Liste der lauffähigen Prozessen ihr Pensum an Rechenzeit verbraucht haben. Im Vergleich dazu haben frühere Versionen von Linux und ältere Unix-Versionen die Neupositionierung der Prozesse nach jeder abgelaufenen Zeitscheibe berechnet. Der Scheduler führt die Statistik über die Rechenzeit und kann den Prozessen mit der großen Laufzeit gegenüber den Prozessen mit sehr geringen Laufzeit die Prioritäten anpassen. Scheduler von Linux und Unix bevorzugen bei der Auswahl des nächsten abzuarbeitenden Prozesses interaktive[1] vor rechenintensiven Prozessen. Es wird versucht, dass die Reaktionszeit der Anwendungen auf die Benutzereingaben im Bereich zwischen 50 und 150 ms liegt. Der Scheduler ist in der Datei `<kernel/sched.c>` implementiert.

Die Prozesse in Linux-Kernel werden durch die sogenannten Prozess-Kontrollblöcke PCB (Process Control Block) repräsentiert. Der Prozess-Kontrollblock ist mit Hilfe der

[1] Prozesse, die sich in der Interaktion mit dem Benutzer befinden, wie z.B. Prozesse für die Tastatur- und die Mauseingaben.

Task-Struktur `struct task_struct` in der Datei `<linux/sched.h>` definiert. Die Kontrollblöcke beinhalten eine Reihe von Informationen, die der Scheduler speichern muss, wenn die Prozesse unterbrochen werden. Dabei wird der Prozesszustand, die Prozessidentifikationsnummer, der Inhalt aller Register der CPU zum Zeitpunkt der Unterbrechung im PCB gesichert. Der Scheduler speichert hier zusätzlich die Prozesspriorität und die verbrauchte Rechenzeit.

In Linux existiert eine Prozesshierarchie, d.h. es gibt einen aller ersten Prozess und alle weiteren Prozesse sind die Kinder von den jeweiligen Elternprozessen. Die Wurzel aller Prozesse in dem Hierarchiebaum ist der beim Systemstart erzeugte `init`-Prozess. Das ist der einzige Prozess, der keinen Elternprozess besitzt. Weil die Prozesse über die Eltern-Kind Beziehung miteinander verknüpft sind, ist es unter anderem möglich, dass das Beenden der Prozesse auf die möglichen Fehler hin überprüft werden kann.

Die Prozesse in Linux können sich in acht Zuständen befinden.

- aktiv: Der Prozess ist in diesem Zustand nur dann, wenn er gerade abgearbeitet wird.
- lauffähig: Die Prozesse sind in diesem Zustand, wenn sie nur auf die Ressource CPU warten, d.h. Prozess ist bereit und wartet auf die Prozessorzuteilung.
- ruhend/terminiert: Die Prozesse befinden sich in diesem Zustand, entweder bevor sie gestartet und damit lauffähig werden oder nachdem sie beendet wurden.
- zombie: Nach dem Beenden eines Rechenprozesses geht dieser nicht sofort in den Zustand `ruhend/terminiert`, sondern er kommt in den Zustand `zombi` und bleibt in diesem, solange der Elternprozess, der den beendeten Prozess erzeugt hat, den beim Betriebssystem gespeicherten Exitcode nicht abholt. Der Exitcode wird in dem entsprechenden PCB gespeichert. Erst nach dem der Elternprozess den Exitcode abgeholt hat, geht der beendete Prozess in den Zustand `ruhend/terminiert` über.
- unterbrechbar wartend: Die Rechenprozesse können ihren Verlauf über die Signale gegenseitig beeinflussen. Der wartende Prozess kann z.B. von einem anderen Prozess über den System-Aufruf in den Zustand `lauffähig` versetzt werden.
- nicht unterbrechbar wartend: Prozesse, die sich in diesem Zustand befinden, warten auf eine Ressource, die noch nicht frei ist. Im Unterschied zu dem Zustand `unterbrechbar wartend` kann die notwendige Ressource nicht von einem weiteren Prozess zur Verfügung gestellt werden.
- TASK_STOPPED: Dieser Zustand wird für das Debugging und Systemcal-Tracing benötigt.
- TASK_TRACED: Dieser Zustand wird für das Debugging und Systemcal-Tracing benötigt.

2.3.3 Speicherverwaltung

Die Speicherverwaltung ist eine weitere Komponente des Betriebssystems Linux. Das Betriebssystem weist jeder Anwendung ihren eigenen Adressraum zu. Dafür muss das Betriebssystem den Speicher virtualisieren. Somit ist es den Anwendungen nicht möglich bzw. nicht erlaubt direkt auf die physikalischen Adressen des Hauptspeichers zuzugreifen. Die Speicherverwaltung übernimmt die Übersetzung der virtuellen Adressen in die physikalischen Adressen. Diese Umrechnung funktioniert nur für die Task, die sich im Zustand `aktiv` befindet.

Die Speicherverwaltung wird durch das sogenannte Paging erleichtert. Dabei wird der Arbeitsspeicher in gleich große Seiten unterteilt. Bei der Übersetzung der virtuellen Adresse in die physikalische muss nicht jede Adresse einzeln übersetzt werden, sondern es muss lediglich festgestellt werden zu welcher Seite die Adresse gehört und zu welcher physikalischen Seite die virtuelle Seite verknüpft ist. Diese Seiten sind z.B. bei den x86 Systemen 4 kB groß. Eine Seite definiert also einen Adressbereich, der 4 kB Speicher adressiert. Die Zuordnung der Seiten zu den physikalischen Seiten erfolgt meistens so, dass keine Fragmentierung stattfindet. Eine interne Fragmentierung des Speichers ist aber nach wie vor möglich, wenn die Seite selbst z.B. nicht vollständig mit Daten gefüllt wird.

Die Aufteilung des Arbeitsspeichers auf die Seiten und der Verzicht auf die Behandlung einzelner Adressen erleichtert das sogenannte Swapping. Bei diesem Verfahren kann der Kernel bestimmte Seiten aus dem Arbeitsspeicher auf die Festplatte auslagern. Die Auslagerung der bestimmten Seiten erfolgt z.B. dann, wenn der freie Platz des Speichers erschöpft ist. Die Auslagerungs-Algorithmen müssen sich auf der Ebene der Seiten bewegen und sie müssen sich nicht um die einzelnen Adressen kümmern. Das Swapping war besonders wichtig, als der Speicher eine knappe Ressource darstellte.

Der Kernel verwaltet den virtuellen Speicher, d.h. jeder Prozess besitzt einen eigenen Speicherbereich und kann bei Bedarf benötigten Speicher anfordern. Die Zuordnung der virtuellen Seiten zu den physikalischen erfolgt durch die sogenannte Pagetable. Die Pagetable enthält neben der Zuordnung der Seiten noch weiter Informationen, z.B. wann der letzte Zugriff stattfand, ob die Seite ausgelagert ist oder ob sie verändert wurde.

Die beschriebene Speicherverwaltung mit Swapping und Paging ist auf die hardwareseitige Unterstützung angewiesen. Die Adressübersetzung erfolgt mit Hilfe der Memory Management Unit. Wenn der Prozess auf eine Speicheradresse zugreift, die keine Abbildung auf die entsprechende physikalische Adresse in der Pagetable besitzt, so löst die MMU einen Page Fault Interrupt aus und die Prozessausführung wird angehalten. Die Interrupt Service Routine des Betriebssystems lädt die betroffene Seite, falls sie vorher ausgelagert wurde, von der Festplatte in RAM und aktualisiert die Seitentabelle. Danach wird der Page Fault verursachende Befehl des Prozesses wiederholt und diesmal kann die MMU die geforderte Adresse auflösen. In den Fällen, in denen die Interrupt Behandlungsroutine merkt, dass die angeforderte virtuelle Adresse keiner entsprechenden physikalischen Seite zugeordnet werden kann, wird der Prozess mit einem Speicherzugriffsfehler beendet.

Die Adressräume aller Prozesse sehen gleich aus. Bei den Systemen mit der 32-Bit-

Architektur steht jedem Prozess einen 4 GB großen Adressraum zur Verfügung. In diesem Adressraum muss der Code und die Daten des Prozesses Platz befinden. Zusätzlich wird ein besonderer Bereich für das Betriebssystem reserviert. Zu jedem Prozess gehört ein sogenannter Stack. Der Stack wird bei den Funktionsaufrufen gebraucht um z.B. die Übergabeparameter an die Funktionen zu speichern. Die Threads, die in einem Prozess laufen, müssen jeweils ihre eigenen Stacks haben, die sich ebenfalls alle im Adressraum des Prozesses befinden müssen. Linux beansprucht das oberste Gigabyte Platz im Adressraum jedes Prozesses für sich. Der Grund für die Einrichtung eines speziellen Bereichs für das Betriebssystem im Adressraum eines Prozesses ist die Interrupt-Behandlung. Wenn der Interrupt auftritt, so geht der Kernel in den Ausführungsmodus Ring 0 und beginnt sofort mit der Abarbeitung der Interrupt Service Routine. Damit dies geschieht, muss sich die Startadresse der Routine bei jedem Prozess an der gleichen Stelle befinden.

Der Bereich für das Betriebssystem ist geschützt und ist aus dem Usermode weder lesbar noch schreibbar. Obwohl es in jedem virtuellen Adressraum den Kernelbereich gibt, verweisen alle diese virtuellen Seiten auf dieselben physikalischen Seiten.

2.3.4 I/O-Subsystem

Das I/O-Subsystem, auch I/O-Management genannt, ist eine weitere Komponente des Betriebssystems. Der gesamte Datenaustausch zwischen den Programmen und den Geräten wird über dieses Subsystem durchgeführt. Dieses System sollte einerseits einheitliche Schnittstellen zur Einbindung der Hardware an das Betriebssystem bieten und andererseits eine weitere einheitliche Programmierschnittstelle für die Anwendungen zum Zugriff auf die Peripherie zur Verfügung stellen.

In Linux (und Unix) unterscheidet man traditionell zwischen zwei Arten von Geräten. Die zeichenorientierten Geräte, auch „Character-Devices“ genannt, und die blockorientierten Geräte, die sogenannten „Block-Devices“, die ihre jeweiligen Schnittstellen zur systemkonformen Anbindung an das System erfordern. Die zeichenorientierten Geräte verarbeiten oder liefern ihre Daten zeichenweise. Den gesamten Datenaustausch kann man sich als eine Art Datenstrom vorstellen. Dabei kommen die Zeichen der Reihe nach einzeln hintereinander und ein Springen innerhalb dieses Streams ist normalerweise nicht möglich. Dadurch, dass der wahlfreie Zugriff auf die Daten nicht möglich ist, kann man die Reihenfolge der verfügbaren Daten nicht beeinflussen. Die typischen zeichenorientierten Geräte sind z.B. die Maus und die Tastatur.

Die blockorientierten Geräte sind in der Lage, im Vergleich zu den zeichenorientierten Geräten, ihre Daten blockweise zu verarbeiten, zu empfangen oder zu senden. Die Datenübertragung z.B. zwischen dem System und dem Gerät kann in einem kontinuierlichen Strom erfolgen. Die Reihenfolge der Daten kann jedoch zumindest blockweise beliebig sein. Die typischen blockorientierten Geräte sind z.B. Disketten-, CDROM-Laufwerke oder Festplatten.

Mittlerweile gibt es sehr viel mehr verschiedene Geräte, die sich nicht mehr eindeutig zu den zeichenorientierten bzw. zu den blockorientierten Geräten zuordnen lassen. Moderne Multimedia-Peripherie trug dazu bei, dass weitere Schnittstellen zu dem Kernel hinzugefügt wurden. Neben den Subsystemen, die speziell für die Integration von z.B.

Soundkarten, Grafikkarten oder Netzwerkkarten implementiert wurden, gibt es Subsysteme, die nicht für die bestimmten Gerätetypen implementiert sind, sondern für die Art der Anbindung dieser Geräte. Es gibt z.B. ein PCI (Peripheral Component Interconnect) -Subsystem, USB (Universal Serial Bus) -Subsystem, SCSI (Small Computer System Interface) -Subsystem usw.

Die oben beschriebenen Schnittstellen dienen innerhalb des I/O-Managements der systemkonformen Integration der Hardware. Die Aufgabe der einheitlichen Programmierschnittstelle der I/O-Verwaltung ist die Abbildung jeglicher Hardware auf die speziellen Dateien. Das Betriebssystem erzeugt die Gerätedateien, die die vorhandene Hardware repräsentieren. Die Anwendungen können über die normalen Dateizugriffsoperationen auf die Hardwarekomponenten zugreifen. Die tatsächlichen Zugriffe, sowohl auf die normalen Dateien als auch auf die Gerätedateien, sind innerhalb der I/O-Verwaltung realisiert. Mit der Einführung weiterer Geräteklassen wurde die Programmierschnittstelle um die eigenen Zugriffsfunktionen für die Multimediageräte erweitert.

2.3.5 Geräte-Treiber

Gerätetreiber sind die Software-Komponenten, die den Anwendungen die Funktionalitäten aller Geräte zur Verfügung stellen. Die Funktionalitäten der Geräte werden von den Treibern über die definierten Schnittstellen für die Anwendungen nutzbar gemacht. Fast alle Geräte, mit der Ausnahme von Prozessor, Speicher und wenigen anderen Komponenten, bedürfen Einstellungs- und Steuerungssoftware, die Treiber. Der Kernel muss die Treiber für die vorhandenen Hardwarekomponenten, die verwendet werden sollen, in sich eingebettet haben.

Dabei gibt es eine Fülle an unterschiedlichsten Geräten, die über die Treiber in das System integriert werden, wie z.B. die systemnahen Tastaturen, Bildschirme, Netzwerkkarten und im Allgemeinen die Drucker, Bandlaufwerke, Scanner, Erweiterungskarten.

Wie bereits im Kapitel 2.3.4 dargestellt wurde, können die zahlreichen Hardwarekomponenten über die speziellen Bussysteme mit dem Betriebssystem verbunden werden. Deswegen gibt es die unterschiedlichen Schnittstellen für die Treiber zu den entsprechenden Treiber-Subsystemen, die von dem Betriebssystem zur Verfügung gestellt werden. Nach der Art der Anbindung oder nach den anderen Eigenschaften unterscheidet man unter Anderen zwischen den folgenden Subsystemen.

- Character-Devices
- Block-Devices
- Netzwerk
- USB
- SCSI
- FireWire
- Bluetooth

- PCI/PCIe
- Cardbus und PCMCIA

Die breite Palette an den möglichen Hardwarekomponenten, die möglichst leicht und gleichzeitig auf die universelle Art und Weise in das System integriert werden sollen, brachte die Erweiterung der Standard-API (Standard-Programmierschnittstelle) mit sich. Die zusätzlichen Applikationsschnittstellen sind:

- Standard-API (mit *open*, *close*, *read*, *write* und *ioctl*)
- Multimedia-Schnittstellen (z.B. Video4Linux, die oft von Webcams verwendet wird)
- Kommunikationsschnittstellen
- Card-Services

Der Linux-Kernel ist ein monolithischer Kernel. Das bedeutet, dass alle notwendigen Treiber als Teil des Kernels vorhanden sein müssen. Bei den monolithischen Kerneln muss der gesamte Kernel neu generiert werden, wenn ein neuer Treiber hinzugefügt werden soll. Die Linux-Entwickler haben eine Möglichkeit geschaffen die Treiber als Module zum laufenden Kernel hinzu zu laden. Im Vergleich zu den fest eingebauten Treibern können die als Modul ausgeführten Treiber ohne der neuen Generierung des Kernels und ohne den Neustart des Systems zum Kernel hinzugefügt und wieder entfernt werden. Diese Fähigkeiten erleichtern das Entwickeln und das Testen der neuen Treiber erheblich. Ein weiterer Vorteil der Modularisierung der Treiber ist die erhöhte Robustheit des ganzen Systems. Zum Beispiel der Absturz eines als Modul geladenen Treibers wird meist nicht zum Absturz des gesamten Systems führen und ein entsprechendes Fehlverhalten kann durch das erneute Laden des Treibers behoben werden.

Die Treiber als Module erleichtern den Benutzern die Verwendung von gewünschten Geräten, weil sie den Treiber einfach laden können. Durch die Modul-Treiber entfällt die Notwendigkeit den gesamten Kernel erneut zu erstellen. Die Benutzer müssen keine Kenntnisse darüber besitzen wie sie einen Kernel neu generieren sollen. In der Realität spielt dieser Vorteil aber keine große Rolle. Es gibt zwei Effekte, die diesem Vorteil entgegenwirken.

1. Aus der Sicht des Benutzers ist es meistens eine Herausforderung den passenden Treiber zuerst zu finden. Viele Hardwarehersteller konzentrieren sich bei der Vermarktung ihrer Peripherie in erster Linie an die Betriebssysteme, die weite Verbreitung erfahren haben. Dadurch werden die Treiber in der Regel nicht von dem Hersteller zur Verfügung gestellt, sondern sie werden von den Programmierern auf Grund von zugänglicher Dokumentation erstellt. In solchen Fällen heißt das, dass die Treiber für Linux meistens nicht den vollen Funktionsumfang des Geräts abdecken und zweitens relativ spät, wenn überhaupt, nach dem Erscheinen des Geräts verfügbar sind. Mittlerweile gibt es immer mehr Hersteller, die Linux als Betriebssystem mit den Treibern unterstützen.

2. Die Linux-Entwicklung schreitet stetig voran. Das heißt aber auch, dass die spezifischen Funktionen, die die Treiber benutzen, und die Treiber-Schnittstellen selbst, sich innerhalb des Betriebssystems verändern. Dadurch kommt es oft vor, dass ein Kernelmodul, das für eine bestimmte Kernel-Version erstellt wurde, nicht mehr kompatibel zu der vom Benutzer verwendeten Kernel-Version ist. Für die Vermeidung der Instabilität beim Betrieb muss die Kernel-Version genau zu dem Modultreiber passen.

Die *Open Source* Treiber sind mit ihrem Quellcode verfügbar. Damit kann jeder den Treiber für die verwendete Kernel-Version selbst erstellen und damit Inkompatibilität vermeiden. Manche PC Komponentenhersteller, wie z.B. die Grafikchiphersteller Nvidia oder AMD, bieten ihre Treiber nicht als *Open Source* Treiber an, um ihr Know-How nicht preiszugeben.

Den logischen Aufbau von Treibern kann man in drei Schichten unterteilen: High-Level-Schicht, Kern-Schicht und die Low-Level-Schicht. Die High-Level-Schicht kümmert sich, wie z.B. bei den USB Geräten, um die Auswertung und um das Zusammensetzen von Kommandopaketen, da die Kommunikation zwischen dem Gerät und dem Treiber mit Hilfe von Paketen erfolgt. Die Kern-Schicht ist z.B. für die Verwaltung der angeschlossenen Geräte oder für die spezifische Hardwareerkennung zuständig. Dies ist notwendig, wenn z.B. mehrere Geräte im System vorhanden sind. Die Low-Level-Treiber führen die tatsächlichen Interaktionen mit der Hardware durch, wie z.B. das Auslesen und Beschreiben von Registern.

Jeder Linux-Treiber muss einen bestimmten Satz an Funktionen implementiert haben.

1. Funktionen, die die Integration des Treibers in den Kernel ermöglichen.

2. Funktionen, deren Ausführung von den Anwendungen ausgelöst werden.

3. Funktionen, die vom Kernel aufgerufen werden.

Die Integration des Treibers in den Kernel erfolgt mit Hilfe der folgenden Funktionen: *init_module*, *cleanup_module*, *probe*, *remove*. Beim Laden des Treibers oder bei dessen Aktivierung, falls dieser nicht als Modul sondern als fester Bestandteil des Kernels ist, müssen vom Treiber benötigte Ressourcen reserviert werden oder es muss die Hardwareerkennung durchgeführt werden. Entsprechende Funktionen werden z.B. beim Entladen des Treibers oder beim Herunterfahren des Systems benötigt, die die belegten Ressourcen wieder freigeben oder die Hardwarekomponenten in einen definierten Zustand bringen.

Die Anwendungen greifen auf die von Treibern bereitgestellten Funktionalität über die Funktionen wie *open*, *close*, *read*, *write* zu. Wenn der Kernel einen Systemaufruf von einer Anwendung bekommt, die die Dienste des Treibers fordert, dann wird die dem Systemaufruf entsprechende Funktion des Treibers aufgerufen.

Die Funktionen, die vom Kernel aufgerufen werden, sind z.B. die Interruptbehandlungsroutinen. Diese Funktionen werden bei der Treiberinitialisierung beim Kernel angemeldet. Dazu gehören unter Anderen die möglichen Kernel Threads oder die Tasklets. Die Interruptsbehandlungsroutine muss den Fall berücksichtigen, dass der evtl. notwendige

Datentransfer zwischen dem Treiber und der Anwendung während dieser Ausführungsphase nicht möglich ist, da der Treiber nicht in der Lage ist auf die Speicherbereiche der Anwendung zuzugreifen. In solchen Fällen setzt der Treiber intern ein Statusbit, dass ein Interrupt ausgelöst wurde. Als erstes, nachdem die Anwendung wieder aktiv wird, muss sie dieses Bit beim Treiber abfragen und somit einen eventuell notwendigen Datentransfer einleiten.

3 PCIe - Schnittstelle

In diesem Kapitel wird die PCIe-Architektur vorgestellt und den anderen Ein-/Ausgabebussen gegenübergestellt. Es werden die Vorteile und die Schlüsselqualifikationen der PCIe-Verbindungen dargestellt. Darüberhinaus werden detailliert die charakteristischen Merkmale des PCIe-Busses beschrieben. Es wird die Schichtarchitektur der über den PCIe-Bus angeschlossenen Teilnehmer mit der jeweils kurzen Funktionsbeschreibung einzelner Schichten vorgestellt.

3.1 Einführung

PCIe stellt mit sich ein Bussystem der dritten Generation dar. Ein Bussystem dient der Datenübertragung zwischen mehreren Teilnehmern über einen gemeinsamen Übertragungsweg. PCIe wird sowohl zum Anbinden von Peripherie als auch zur Kommunikation zwischen Endgerärten benutzt. Die Vertreter der ersten Generation sind: ISA (Industry Standard Architecture), VESA (Video Electronics Standards Association) Local Bus und Micro Channel. Die Vertreter der zweiten Generation sind: PCI und AGP (Accelerated Graphics Port).

Der ISA-Bus wurde als ein Teil des IBM-PC Projektes im Jahre 1981 entwickelt. Ursprünglich handelte es sich dabei um eine einfache Herausführung des 8 Bit breiten Systembusses. Die Erweiterung des Busses auf 16 Bit erfolgte mit der Einführung des neuen Intel 80286 Prozessors. Der Takt des Busses war synchron mit dem der CPU. Der Bus war für die Taktfrequenz von 6 bzw. 8 MHz ausgelegt. Mit der fortschreitenden Entwicklung der Prozessoren und stetig steigenden Taktfrequenzen wurden Chipsätze entwickelt, mit deren Hilfe man den ISA-Bus mit der CPU verbinden konnte. Die Entwicklung des PCI-Busses verdrängte den ISA-Bus nahezu vollständig. ISA-Busse werden auch noch heute in Industrie-PCs oder in eingebetteten Systemen eingesetzt. Die theoretische Bandbreite des ISA-Busses mit einem Bustakt von 8.3 MHz und der Busbreite von 16 Bit beträgt 15.9 MBytes pro Sekunde, da aber die Addressierung jedes Zugriffs einen Takt dauerte fiel die theoretische Bandbreite auf die ca 8 MBytes pro Sekunde. Die tatsächlich erzielte maximale Bandbreite lag zwischen 1 und 2 MBytes pro Sekunde.

VESA Local Bus wurde als Ergänzung zu dem ISA-Bus eingeführt. Die schnellen Grafikkarten erforderten einen höheren Datendurchsatz als dies der ISA-Bus ermöglichte. Der Bus bestand hauptsächlich aus den herausgeführten Signalen des i486 Prozessors, dadurch war die Portierung auf andere als x86 Systeme nahezu aussichtslos. Der Slot für die Erweiterungskarten musste 112 Pins aufnehmen. Es konnten maximal 3 Karten gleichzeitig eingesteckt werden. Die enge Anbindung an den speziellen Prozessortyp erforderte hohen Schaltungsaufwand für die Anpassung an die nächste Prozessorgeneration. Die große Länge der Schnittstelle in Verbindung mit der erhöhten Anzahl von Pins, in

Vergleich zur ISA-Schnittstelle, führte nicht selten beim Installieren oder Entfernen der Karten zum Bruch der Hauptplatinen. Der VESA Local Bus ist Rückwärtskompatibel zu dem ISA-Bus. Die maximale theoretische Bandbreite betrug 130 MBytes pro Sekunde wobei die im Einsatz erzielte Bandbreite durchschnittlich 32 MBytes pro Sekunde betrug.

Die Entwicklung des PCI-Busses wurde durch die Firma Intel im Jahre 1990 angestoßen. Intel wollte den VESA Local Bus als Nachfolger des ISA-Busses nicht unterstützen und eine neue, prozessorunabhängige Bus-Architektur etablieren. Der PCI-Bus wurde als eine Plattform angesehen, die die Ausnutzung aller Rechenkapazitäten der kommenden Pentium-Prozessoren erlaubte. Intel versuchte die PC-Industrie für den PCI-Bus zu gewinnen und gründete 1992 die sogenannte *Peripheral Component Interconnect Special Interest Group (PCI-SIG)*. Die Ziele dieser Organisation sind die Verwaltung, Weiterentwicklung und die Verbreitung des PCI-Standards. Mittlerweile zählen über 800 Mitglieder zu der Gruppe. Im Laufe der Zeit wurden mehrere Versionen des Standards beschlossen und umgesetzt. Der Bus zeichnet sich durch die Möglichkeit der Hierarchisierung aus. Der Bus wird je nach Version mit 33 MHz bzw. 66 MHz betrieben. Die Bandbreite beträgt für PCI 1.0 (1991) bis PCI 3.0 (2004) 133 MBytes pro Sekunde - 532 MBytes pro Sekunde. Die sogenannte Host-Bridge dient als Schnittstelle zwischen den CPU mit Arbeitsspeicher und dem Bus. Die masterfähigen Peripheriegeräte können über die Hostbridge als Target direkt in den Arbeitsspeicher schreiben und aus ihm lesen. Auf dem PCI-Bus kommuniziert immer ein Master mit einem Target. Die angeschlossenen Komponenten teilen sich die zur Verfügung stehende Bandbreite untereinander auf. Mit einer steigenden Anzahl an Busteilnehmern sinkt die Bandbreite entsprechend. Die gängigste Bus-Konfiguration in einem PC ist: eine Busfrequenz von 33 MHz und eine Busbreite von 32 Bit. Die dabei erzielbare maximale Bandbreite ist etwa 90 MBytes pro Sekunde. Der PCI-Bus hat sehr große Verbreitung in vielen Bereichen erfahren. Seit 2005 wird aber der PCI-Bus durch seinen Nachfolger den PCIe-Bus allmählich verdrängt.

Der PCI-Bus erfüllte die Anforderungen für Grafik-, Netzwerk- und andere Schnittstellenkarten über eine längere Zeit. Allerdings reichte nach einiger Zeit die verfügbare Bandbreite für die damals aufkommenden Grafikkarten mit 3D-Beschleunigung nicht mehr aus. Aus diesem Grund wurde das AGP Bus-System eingeführt. Der AGP-Bus stellte eine Punkt-zu-Punkt-Verbindung zur Anbindung der Grafikkarte an die Northbridge dar. Die Vorteile gegenüber dem PCIe-Bus haben sich z.B. dadurch ergeben, dass immer nur ein Teilnehmer an Datentransfers beteiligt war, oder dadurch, dass es kein „richtiger“ Bus war und man deswegen die Taktfrequenz höher wählen konnte. Die erste Version des AGP Systems wurde von Intel im Jahre 1997 veröffentlicht. Im Laufe der Zeit wurde der Standard erweitert und die mögliche maximale Bandbreite erhöht. Die AGP-Schnittstelle der Version 1.0 (1x) erlaubte den Datendurchsatz von 266 MBytes pro Sekunde. Die letzte Version der Schnittstelle erlaubt die Bandbreite von 2133 MBytes pro Sekunde. Es gab wenige Ausnahmen von Hauptplatinen, die über mehr als eine AGP-Schnittstelle verfügten. Die zusätzliche Schnittstelle konnte dann für einige RAID-Kontroller benutzt werden um den Datendurchsatz nicht mit den anderen Komponenten am PCI-Bus teilen zu müssen. Eine weitere Steigerung der Datentransferleistung von AGP ist ohne grundlegende Veränderungen an der Architektur nicht mehr

möglich. Die parallele Datenübertragung bei hohem Takt und die damit verbundenen, strengen Timing-Anforderungen machten das Platinendesign sehr Aufwendig. Die Anbindung der Grafikkarten an den Arbeitsspeicher des PCs ist nicht mehr so wichtig wie früher, weil die Grafikkarten im Zuge der gefallenen Preise für Speicherchips über genügend dedizierten Speichers verfügen. Der Hauptnachfolger für den AGP-Bus ist der PCIe-Bus.

Die Entwicklung der Prozessoren ist seit der Einführung des PCI-Busses schneller vorangegangen als die des Busses selber. Der PCI-Bus sollte seinerzeit verschiedene, bereits vorhandene Busse ersetzen und eine gemeinsame Plattform für unterschiedliche Aufgaben darstellen. Diese Funktion konnte er nicht lange aufrecht halten. Mit den neuen Möglichkeiten der neuen Prozessoren und immer größere, zur Verfügung stehende Menge an Speicher öffneten ganz neue Tätigkeitsfelder. Dabei entstanden neue Anwendungen, die weit größere Bandbreiten forderten als der PCI-Bus anbot. Es entstanden wieder zahlreiche Bus-Systeme, die nur für bestimmte Anwendungsfälle spezialisiert waren, wie z.B. AGP, ATA100 usw. Die Gemeinsamkeit aller erwähnten Busse war die parallele Datenübertragung. Dem immer weiter steigenden Bandbreitenbedarf entgegnete man mit der Erhöhung der Busfrequenz. Die hohe Anzahl an benötigten Pins verbrauchte viel Platz auf den Platinen. Steigende Frequenzen in Verbindung mit den vielen Leitungen brachten elektrische Problemen mit sich und somit stellt sich nun der Bus als Flaschenhals bei der Kommunikation zwischen der Peripherie und der CPU dar. Unter der Berücksichtigung solcher Aspekte hat die PCI-SIG den PCIe-Standard entworfen, der sowohl PCI als auch AGP ersetzen soll und eine größere Datenübertragungsrate als AGP bieten soll.

3.2 PCIe

PCIe ist eine separate, serielle Punkt-zu-Punkt Verbindung mit differentieller Signalübertragung. Dadurch sind viele Vorgehensweisen, wie sie bei dem PCI-Bus üblich waren, nicht mehr anwendbar. Im Vergleich zu dem PCI-Bus müssen die Kommunikationspartner nicht mehr um den Zugriff auf den Bus konkurrieren. Jeder Teilnehmer treibt exklusiv den eigenen Satz an Sendeleitungen und ist gleichzeitig der Empfänger über die Empfangsleitungen. Bei den Punkt-zu-Punkt Verbindungen gibt es immer nur zwei Kommunikationsteilnehmer, die entsprechende Leitungen treiben können.

Die Verbindung zwischen den zwei PCIe Geräten bezeichnet man als *Link*. Ein *Link* kann aus mehreren *Lanes* bestehen. Eine *Lane* wiederum besteht aus zwei Paaren der differentiellen Leitungen. Jedes Paar ist für die Kommunikation in eine Richtung verantwortlich. Außer der erwähnten Leitungen gibt es keine weiteren, wie z.B. für die Adressen, Daten oder für Kontrollsignale, wie beim PCI-Bus. Die bewusste Beschränkung an die wenigen Signale erleichtert die Skalierung der Verbindung für die steigenden Anforderungen in Zukunft und engt die Möglichkeiten der Implementierung der neuen Einsatzmodellen nicht ein. Die starke Veränderung des physischen Aufbaus gegenüber dem PCI erfordert eine gänzlich neue Infrastruktur der Systemkomponenten. Die Entwickler von der PCI-SIG haben Wert darauf gelegt, dass die Softwareschnittstelle für

den PCIe-Bus voll kompatibel zu dem PCI-Bus bleibt. Dadurch müssen weder Betriebssysteme, Treiber noch Anwendungsprogramme angepasst werden.

Die erste Version des PCIe Erweiterungsstandards arbeitet mit einer Datenrate je Lane von maximal 250 MByte/s pro Richtung beziehungsweise 500 MB/s in beide Richtungen zusammen. Für die Anwendungen mit hohen Anforderungen an Bandbreite kann man die *Lanes* koppeln und damit diese Anforderungen erfüllen. Die Tabelle 3.1 zeigt die theoretisch erreichbaren Bandbreiten in Abhängigkeit von der Anzahl der gekoppelten *Lanes* und der Version des Standards. Inzwischen existiert die Version 2.0 der PCIe-

Tabelle 3.1: Datenrate PCI-Express

	PCIe 1.0	PCIe 2.0	PCIe 3.0
x1	250 MB/s	500 MB/s	1000 MB/s
x2	500 MB/s	1000 MB/s	2000 MB/s
x4	1000 MB/s	2000 MB/s	4000 MB/s
x8	2000 MB/s	4000 MB/s	8000 MB/s
x16	4000 MB/s	8000 MB/s	16000 MB/s
x32	8000 MB/s	16000 MB/s	32000 MB/s

Spezifikation mit einer Datenrate von 500 MByte/s pro Lane. Die neueste Spezifikation in der Version 3.0, die bis 2011 festgelegt werden soll, soll eine Datenrate von 1000 MByte/s pro Lane ermöglichen.

Die PCIe-Spezifikation beschreibt einige Typen von PCIe Elementen: *root complex*, *PCIe-PCI bridge*, *endpoint* und *switch*.

- *Root complex*: Diese Komponente ist das Bindeglied zwischen dem Ein-/Ausgabesystem und der CPU mit dem Hauptspeicher. Der *root complex* verwaltet und konfiguriert die über den Bus angeschlossene Peripherie. Weiterhin übersetzt er die Zugriffe in beide Richtungen.

- *PCIe-PCI bridge*: Eine *PCIe-PCI bridge* ermöglicht die Koexistenz von älteren Bussystemen wie PCI/PCI-X neben PCIe.

- *Endpoint*: Ist ein konkretes Gerät, das die PCIe-Transaktionen empfangen oder selbst auslösen kann. Der *Endpoint* kann selbst z.B. eine Bridge zum USB sein. Man unterscheidet zwei Typen von *Endpoints*: *legacy* und *native*. Der Unterschied beruht auf der Fähigkeit bestimmte Transaktionen verarbeiten zu können.

- *Switch*: *Switch*e spannen die PCIe-Hierarchie auf. Mehrere *endpoint*-Geräte werden mit dem *Switch* verbunden. Der *Switch* ermöglicht die Verbindung entweder zwischen zwei Kommunikationspartnern oder zwischen dem *endpoint* und dem *root complex*.

3.3 PCIe Transaktionen

Die Transaktionen bilden die Basis der Informationsübertragung zwischen den PCIe-Geräten, wobei der Informationstransport paketbasiert ist. Die Transaktionen sind als eine Serie von Übertragungen eines oder mehrerer Pakete definiert, die für die komplette Informationsübertragung zwischen den Kommunikationspartnern notwendig sind. Eine Transaktion besteht aus zwei Abschnitten: einer Anfrage oder Aufforderung und Fertigstellung oder Erfüllung dieser Anfrage. Die Einheit, welche die Anfrage macht, sendet das entsprechende Paket zu der Einheit, welche diese Anfrage bearbeiten soll. Dieses Paket kann dabei über mehrere *Switche* hinweg zu dem Ziel geleitet werden. Die Reaktion auf das Aufforderungspaket kann aus keinem, einem oder auch mehreren Fertigstellungspaketen bestehen.

Die PCIe-Architektur beschreibt vier verschiedene Typen von Transaktionen:

1. Memory Transactions
2. I/O Transactions
3. Configuration Transactions
4. Message Transactions

3.3.1 Memory Transactions

Bei den sogenannten Speicher-Transaktionen werden über die Speichereinblendung verfügbare Daten von oder zu den PCIe-Geräten transportiert. In den meisten Fällen erfolgt der Datentransfer zwischen dem PCIe-Gerät und dem Arbeitsspeicher des Rechners. Man unterscheidet zwischen unterschiedlichen Arten der Speicher-Transaktionen, einige von ihnen sind: *Memory Read Request*, *Memory Read Completion* und *Memory Write Request*. Die Adressierung erlaubt die Verwendung von kurzen, d.h. 32 Bit langen Adressen als auch von langen, d.h. 64 Bit langen Adressen.

Beim lesenden Speicherzugriff eines PCIe-Geräts sendet dieses ein *Memory Read Request* mit der Angabe der Adresse und der gewünschten Datenmenge aus. Der *Root complex* führt den eigentlichen Speicherzugriff auf den Arbeitsspeicher aus und liefert die angeforderten Daten an das PCIe-Gerät mit den, evtl. mehreren, *Memory Read Completions*.

Wenn ein PCIe-Gerät in den Arbeitsspeicher schreibt, dann werden die *Memory Write Requests* abgeschickt. Auf die Bestätigung des erfolgreichen Schreibvorgangs kann dabei zu Gunsten der besseren Leistungsfähigkeit verzichtet werden.

3.3.2 I/O Transactions

Das sind Transaktionen, die den Speicherbereich der Ein-/Ausgabe betreffen. Dieser Speicherbereich wird aus Kompatibilitätsgründen zu den bereits vorhandenen Geräten unterstützt. Einige interessante Transaktionen sind: *I/O Read Request*, *I/O Read Completion*, *I/O Write Request* und *I/O Write Completion*. I/O Transactions verwenden immer nur 32 Bit breite Adressen.

3.3.3 Configuration Transactions

Die *Configuration Transactions* greifen auf den Konfigurationsspeicherbereich der PCIe-Geräte. Sie dienen der Konfiguration und der Einstellung aller am Bus hängenden Geräte. Der Konfigurationsspeicherbereich erstreckt sich nur über die Konfigurationsregister, die jedes PCI oder PCIe-Gerät hat. Im Unterschied zu dem PCI-Standard können die PCIe-Geräte einen wesentlich größeren Satz an Registern vorweisen. Die *Configuration Transactions* sind: *Configuration Read Request*, *Configuration Read Completion*, *Configuration Write Request* und *Configuration Write Completion*.

Die Konfigurations-Transaktionen gehen normalerweise nicht von den Endgeräten aus. Diese werden ausschließlich vom *root complex* aus getriggert.

3.3.4 Message Transactions

Eine weitere Transaktionsart, die es so bei PCI nicht gab, sind die *Message Transactions*. Unter den *Message Transactions* sind viele verschiedene Pakete zusammengefasst, die für die Kommunikation zwischen den Endgeräten eingesetzt werden. Mit diesen Transaktionen werden z.B. Interrupts oder Fehler signalisiert oder sie werden für die Energieverwaltung gebraucht.

3.4 PCIe Übertragungsschichten

Die PCIe-Spezifikation definiert eine Architektur der Übertragung, die aus drei Schichten besteht. Jedoch müssen sich die Geräteentwickler nicht an diese Architektur halten, solange die vom Standard geforderte Funktionalität geboten wird. Die Abbildung 3.1 stellt die definierten Schichten dar.

Jede Transaktion durchläuft diese drei Schichten. Die erste Schicht lässt sich zum *Transaction Layer* zusammenfassen. Die Hauptaufgabe dieser Schicht ist die Bildung eines Transaktionspaketes aus den Daten, die von Kern des Gerätes kommen. Die ankommenden Pakete werden ausgewertet und die enthaltenen Daten an den Kern weitergeleitet. Diese Schicht entspricht der Zusammenfassung von *Transport* und *Network* Schichten des OSI-Modells.

Die zweite Schicht ist der *Data Link Layer*. Die Hauptaufgabe dieser Schicht ist Sicherstellung des korrekten Senden und Empfangen aller Transaktionen. Die Funktionalität des *Data Link Layer* stimmt mit der des OSI-Modells überein.

Die dritte und letzte Schicht ist der *Physical Layer*. Diese Schicht führt tatsächlich das Senden und Empfangen der Transaktionspakete durch. Die Funktionalität der *Physical Layer* stimmt mit der des OSI-Modells überein.

Dabei lässt sich jede dieser Schichten in zwei Blöcke aufteilen: ein Block, der für das Senden verantwortlich ist, und einer, der für das Empfangen verantwortlich ist. Beim Senden wird der Paketinhalt aus den funktionsbezogenen Daten des Gerätekerns in der *Transaktionsschicht* gebildet. Dieses Paket wird in einem Puffer aufbewahrt und an den darunter liegenden *Data Link Layer* weitergegeben. Der *Data Link Layer* hängt an das Paket eigene zusätzliche Informationen zur Fehlererkennung auf der Empfangsseite an.

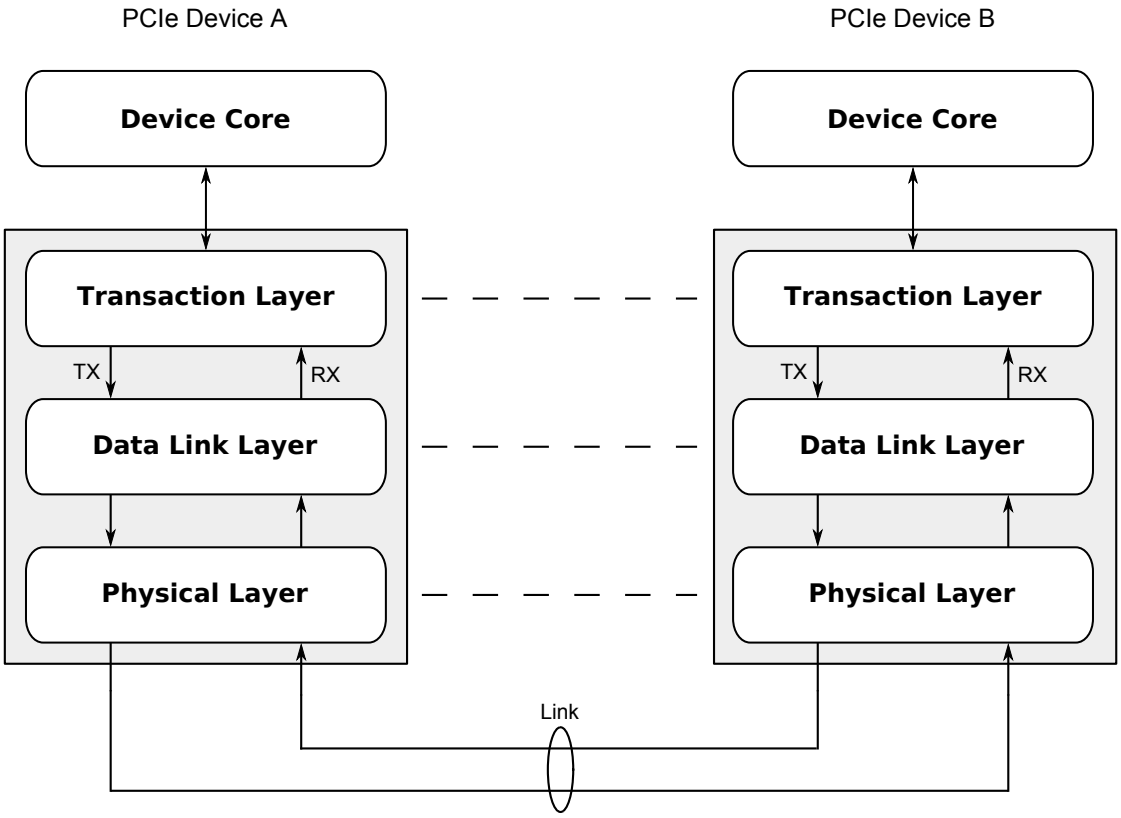

Abbildung 3.1: Übertragungsschichten der PCIe-Geräte

Dieses Paket wird schließlich an den *Physical Layer* weitergegeben. Das Paket wird dabei neu kodiert und mit zusätzlichem Header und Trailer über den Link analog, differentiell an den unmittelbaren Nachbarn übertragen.

Auf der Empfangsseite läuft das Paket in der umgekehrten Reihenfolge die drei Schichten durch. Der Empfänger dekodiert das empfangene Paket im *Physical Layer* und übergibt den Inhalt an die nächsthöhere Schicht. Der *Data Link Layer* überprüft das Paket auf Fehler, und falls es keine Fehler gab, wird das Paket an die *Transaktionsschicht* weitergereicht. Die *Transaktionsschicht* wandelt die Paketdaten in eine Form um, die der Gerätekern mit der entsprechenden Funktion verarbeiten kann.

3.4.1 PCIe Transaktionsschicht

Diese Schicht bildet das Rückgrat der Datenübertragung innerhalb des PCIe-Systems. Die Hauptaufgabe dieser Schicht ist das Generieren von entsprechenden Transaktionen, darunter die *Requests* und die *Completions*.

Die *Transaktionsschicht* empfängt auf der Sendeseite die Daten von dem Kern des Geräts, welche in PCIe-Transaktionen umgesetzt werden. Diese Daten können z.B. die Anforderung des Gerätes von bestimmten Daten sein oder aber die Antwort des Gerätes auf die vorangegangene Anfrage. Auf der Empfangsseite bekommt die *Transaktionsschicht* die PCIe-Transaktionen von der *Data Link* Schicht. Dabei geht die *Transaktionsschicht* davon aus, dass die ankommenden Pakete alle fehlerfrei und in der richtigen Reihenfolge sind. Die Sicherstellung der Fehlerfreiheit und der richtigen Reihenfolge ist die Aufgabe der *Data Link* Schicht.

Die *Transaktionsschicht* verwendet zur Kommunikation mit den anderen PCIe-Geräten die eigenen Pakete: TLP (Transaction Layer Packet). Die Vielfalt der verschiedenen Transaktionstypen und der zu erfüllenden Aufgaben lässt sich über die Daten im Header des Transaktionspaketes einstellen. Dabei werden die Pakete in der *Transaktionsschicht* eines „Sendegerätes“ erzeugt und in der *Transaktionsschicht* des Empfängers ausgewertet und verarbeitet. Neben der bereits erwähnten Funktionalität ist die *Transaktionsschicht* auch für die Flusskontrolle der TLP und für einige Funktionen der Energieverwaltung zuständig ist.

Ein Paket der *Transaktionsschicht* besteht aus dem Header, den optionalen Daten und einem optionalen Trailer. Die Abbildung 3.2 stellt die Struktur eines Pakets der *Transaktionsschicht* dar. Ein TLP hat immer einen Header, dessen Länge nicht konstant ist, aber immer ein vielfaches von vier Bytes beträgt. Die Länge des Header hängt von dem Typ der Transaktion ab. Vom Typ der Transaktion hängt auch das Vorhandensein von Nutzdaten ab. Die Menge an Nutzdaten muss ebenfalls ein Vielfaches von vier Bytes betragen. Wenn die Menge der zu übertragenden Daten nicht genau in einem Vielfachen von vier Bytes aufgeht, dann werden die fehlenden Bytes aufgefüllt. Das Auffüllen kann entweder bereits in den ersten vier Bytes oder aber in den letzten vier Bytes erfolgen. In dem Header des TLP werden die speziellen Bits gesetzt, die die Identifikation von „nutzlosen“ Bytes ermöglichen. Der Trailer des Paketes ist wie die Daten auch optional und wird nur selten benutzt. Der Trailer ist für die ECRC (end-to-end CRC)-Prüfsumme gedacht. Der *Data Link Layer* deckt mögliche Fehler ab, die bei der Übertragung über den

Link entstehen können. Die ECRC-Prüfsumme ermöglicht auch die Fehler zu erkennen, die während der Bearbeitung des Paketes auf der Ebene der *Transaktionsschicht* in den dazwischen liegenden Geräten, wie z.B. Switches, entstehen können.

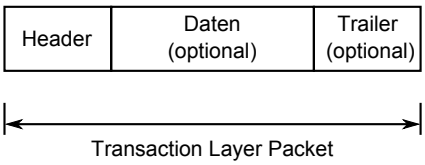

Abbildung 3.2: Struktur eines TLP-Paketes

Ein Aufforderungspaket kann mehrere Antwortpakete nach sich ziehen. Es ist aber nicht möglich mehrere anstehende Aufforderungen mit nur einem Antwortpaket zu erfüllen

3.4.2 PCIe Data Link Schicht

Die Hauptaufgabe dieser Schicht ist die Sicherstellung der korrekten Übertragung auf dem Link, d.h. Fehlererkennung und Fehlerkorrektur. Der *Data Link Layer* sorgt dafür, dass die Pakete fehlerfrei und in der richtigen Reihenfolge über die einzelnen Links transportiert werden.

Der *Data Link Layer* setzt die Bildung der PCIe-Transaktion fort, indem sie das von der *Transaktionsschicht* kommende Paket um eigene Informationen erweitert und das neu entstandene Paket an die *Bitübertragungsschicht* weiterleitet. Die *Data Link* Schicht fügt jedem zu sendenden TLP als Header eine sogenannte Sequenznummer hinzu. Als Trailer wird eine spezielle Prüfsumme hinzugefügt: LCRC (Link CRC). Die Abbildung 3.3 zeigt ein TLP, das um die zusätzlichen Informationen für die Fehlererkennung von dem *Data Link Layer* erweitert wurde.

Sowohl die Ein- als auch die Ausgangsseite des *Data Link Layer* haben jeweils ihre eigenen Aufgaben. Beim Eintreffen eines Paketes übernimmt die *Data Link* Schicht dieses von der *Bitübertragungschicht*. Es wird die Korrektheit des empfangenen Paketes anhand der Sequenznummer und der LCRC überprüft und dann, wenn sowohl die Sequenznummer als auch die LCRC zeigen, dass es keine Fehler gab, wird dieses Paket, bereinigt von der Sequenznummer und der LCRC, an die Empfangsseite der *Transaktionsschicht* weitergereicht. In diesem Fall, wenn das Paket als fehlerhaft erkannt wird,

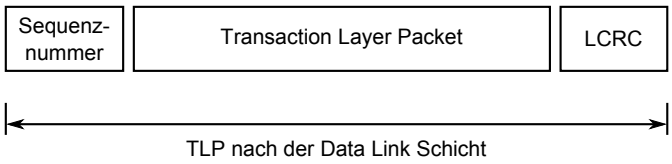

Abbildung 3.3: Erweiterung des TLPs um die zusätzlichen Informationen in der *Data Link* Schicht

wird es erst gar nicht an die *Transaktionsschicht* weitergeleitet. Die *Data Link* Schicht versucht mit der entsprechenden Protokollinstanz auf der Gegenseite den Fehler zu beseitigen. Die Fehlerbeseitigung erfolgt meistens durch die Aufforderung das fehlerhafte Paket noch einmal zu senden. Der *Data Link Layer* lässt also nur die korrekt empfangenen Pakete an die *Transaktionsschicht* durch. Die *Transaktionsschicht* geht dadurch immer davon aus, dass alle Pakete, die sie empfängt, auch korrekt sind.

Wenn ein Sender einen TLP an einen Empfänger über den Link sendet, dann erwartet er eine Quittierungsantwort. Es kann ein NACK (von engl. negative acknowledgement = negative Bestätigung) DLLP (Data Link Layer Packet), im Falle eines falschen LCRC, ein ACK (von engl. acknowledgment = Bestätigung) DLLP, wenn es keine Fehler gab, oder er bekommt gar keine Bestätigung, wenn das gesendete Paket z.B. beim Empfänger gar nicht ankommt. Die Quittierungsantworten folgen nicht unbedingt nach jedem empfangenen Paket. Zwei Kommunikationspartner auf einem Link tauschen die sogenannten „credits“ unter sich aus. Diese „credits“ erlauben dem Sender eine bestimmte Anzahl an Paketen abzusenden ohne eine Quittierungsnachricht abzuwarten. Der Empfänger kann mit einer Quittierungsantwort einen fehlerfreien Empfang eines bestimmten Paketes bestätigen. Und wenn es noch mehrere unbestätigte Pakete vor diesem Paket gegeben hat, dann gelten damit auch diese als bestätigt.

Auf der Sendeseite wird einem zu sendenden TLP eine Sequenznummer und LCRC hinzugefügt. Das gesendete Paket wird in einem Puffer der *Data Link* Schicht zwischengespeichert. Wenn der Empfänger das Paket auf der Linkebene fehlerfrei empfangen hat, dann bestätigt er es mit einem ACK DLLP, das die Sequenznummer des empfangenen Paketes enthält. Der Sender entfernt daraufhin das gespeicherte Paket aus dem Puffer. Falls andererseits der Empfänger einen LCRC-Fehler im Paket entdeckt, dann sendet er ein NACK DLLP mit der Sequenznummer. In diesem Fall veranlasst der Sender einen erneuten Versand des betroffenen Paketes. Die *Data Link* Schicht des Senders kann den Fehler zum Protokollieren an den Rootkomplex melden. Falls es für einen bestimmten TLP aus dem Puffer dreimal ein NACK DLLP kam, d.h. es wurde viermal erfolglos versucht das Paket zu übertragen, dann veranlasst die *Data Link* Schicht die Reinitialisierung des Links und meldet einen korrigierbaren Fehler an den Rootkomplex.

Auf der Empfangsseite überprüft die *Data Link* Schicht die LCRC der eintreffenden Pakete. Diese Seite veranlasst dann, je nach Ergebnis der Prüfung, das Versenden der ACK DLLP oder NACK DLLP. Die Empfangsseite erhält auch die Quittierungsantworten von der Sendeseite des Kommunikationspartners. Daraufhin wird innerhalb dieser Schicht auf der Sendeseite entweder das entsprechende TLP aus dem Puffer entfernt oder noch einmal versendet. Auf dieser Seite wird auch auf die Einhaltung der Reihenfolge geachtet. Dabei wird die Sequenznummer der eintreffenden Pakete überprüft. Durch diesen Mechanismus können fehlende oder sich wiederholende Pakete sicher erkannt werden.

Die Sequenznummer ist eine 12 Bit breite Zahl, die, zusammen mit den zusätzlichen ungenutzten 4 Bits zu einem 2 Byte großen Header zusammengefasst werden. Das LCRC-Feld ist 32 Bit breit und wird an das TLP angehängt. LCRC wird über alle Bits des zu sendenden Paketes berechnet, auch über die sogenannten „reserved“ und über die angehängte Sequenznummer.

Die *Data Link* Schicht unterscheidet nicht um welchen Typ von TLP es sich handelt,

wenn eine Sequenznummer dem Paket zugewiesen wird. Die Sequenznummern hängen auch nicht davon ab, wer die Anfrage startet und wer sie erfüllt. Die Instanz der *Data Link* Schicht des Senders ist die einzige Bestimmungsgröße bezüglich der Sequenznummer. Diese Nummer hat ihre Gültigkeit nur innerhalb eines Links, d.h. wenn ein Switch das Paket von einem Link auf das andere weiterleitet, dann hat das Paket eine unterschiedliche aber jeweils für den Link gültige Sequenznummer. Die nötigen Informationen zum Routing der Pakete innerhalb des PCIe-Netzwerks steckt im Header des TLP.

Die *Data Link* Schicht verwendet noch einen eigenen Pakettyp: DLLP. Diese Pakete dienen linkspezifischen Funktionen, wie z.B. Benachrichtigungen über die aufgetretenen Fehler und für die Energieverwaltung. Diese Pakete entstehen in der *Data Link* Schicht und werden immer in dieser verarbeitet. Sie werden auch anders als Pakete der *Transaktionsschicht* beim Senden behandelt. Die DLLP werden immer nur zwischen den direkten Nachbarn auf dem Link versendet im Unterschied zu den TLPs, die auch über mehrere Kommuniaktionsgeräte hinweg transportiert werden können. Neben den bereits beiden erwähnten Typen von DLLPs wie NACK und ACK gibt es noch zwei weitere:

- Flow control DLLP: Diese Pakete dienen der Datenflusssteuerung. Es gibt insgesamt drei Typen davon: zwei werden bei der ersten Initialisierung des Links verwendet und der dritte Typ wird im Betrieb gebraucht, um den direkten Kommunikationspartnern die noch zur Verfügung stehende Kapazität an freien Plätzen für die reinkommenden Pakete anzuzeigen. Die Erzeugung und den „Verbrauch“ von diesen Paketen übernimmt die *Data Link* Schicht, die Kontrolle über das Senden und über die Daten in diesen Paketen hat allein die *Transaktionsschicht*.

- Power management DLLP: Diese Pakete dienen der Steuerung und der Kontrolle vom Energiestatus des Links. Dabei entscheidet nicht die *Data Link* Schicht wann und ob sie die *power management* DLLP versendet, sondern es tut die Logik, die für Energieverwaltung zuständig ist, die dann das Versenden auslöst. Werden die *power management* DLLP empfangen, dann werden die Daten an die Energieverwaltung abgegeben.

Alle DLLPs haben eine konstante Länge von 6 Bytes: 4 Bytes für die Daten und 2 Bytes für die CRC (Cyclic Redundancy Check) Summe. Das CRC-Feld wird hierbei anders berechnet als das Feld für LCRC und ECRC. Die Verarbeitung dieser Pakete seitens der *Data Link* Schicht unterscheidet sich von den „normalen“ TLP-Paketen. Die *Bitübertragungsschicht* leitet die DLLP an die *Data Link* Schicht weiter und diese Schicht teilt auch mit ob es Fehler bei der Übertragung gab oder nicht. Erst wenn es keine Fehler gab, überprüft die *Data Link* Schicht das Paket mit Hilfe der CRC-Prüfsumme auf die Korrektheit. Falls die *Bitübertragungsschicht* einen Fehler meldet, wird das Paket von der *Data Link* Schicht verworfen. Das wiederholte Senden von DLLPs ist nicht vorgesehen und sie werden deswegen auch nicht in den Puffern zwischengespeichert. Die *Flow Control* DLLPs haben einen Einfluss auf die *Transaktionsschicht*, die *Power Management* DLLPs üben Einfluss auf die Energieverwaltung aus und die NACK und ACK DLLPs werden innerhalb der *Data Link* Schicht benutzt.

3.4.3 PCIe Bitübertragungsschicht

Die *Bitübertragungsschicht* ist die unterste Schicht. Eine ihrer Aufgaben ist es, die elektrische Verbindung zwischen zwei direkt miteinander verbundenen Geräten herzustellen. Diese Schicht übernimmt die Pakete von der *Data Link* Schicht, um sie über *„Link“*, eine logische Verbindung wegzuschicken und sie leitet die über den *„Link“* empfangenen Pakete an die höhere Schicht weiter.

Man kann die *Bitübertragungsschicht* in zwei weitere Schichten unterteilen: in eine logische und eine elektrische Schicht. Die Aufgaben des logischen Teils ist die Bearbeitung der zu sendenden und zu empfangenen Paketen. Der sogenannte elektrische Teil stellt eine elektrische Schnittstelle dar, die eine Verbindung zwischen zwei Geräten herstellt. Diese Schnittstelle beinhaltet unter anderem auch die differentiellen Treiber und Empfänger für jede *Lane*.

Auf der Sendeseite der logischen Schicht werden die Bytes des Pakets zuerst mittels eines *Scramblers* pseudozufällig umkodiert, diese Bytes werden in einem 8b/10b Verfahren kodiert und anschließend um die speziellen Paketmarker erweitert. Die Empfangsseite führt die gleichen Operationen in der umgekehrten Reihenfolge durch. Im Folgenden werden die einzelnen Operationen näher erläutert.

8b/10b - Kodierung

Eine weitere Aufgabe, die die *Bitübertragungsschicht* zu erfüllen hat, ist die 8b/10b - Kodierung der Daten. Der Hauptzweck dieser Kodierung ist es, das Taktsignal in den Datenstrom einzubinden. Bei dieser Kodierung werden 8 Bit Daten mit einem 10 Bit langen Symbol kodiert. Neben der erwähnten Taktrückgewinnung erlaubt die Kodierung den Gleichspannungsausgleich.

Bei den parallelen Bussen, wie bei PCI stellte sich heraus, dass bei steigenden Frequenzen, mit denen das System betrieben wird, die Anforderungen an die Signalführung auf der Platine, und genauer an die Länge der Signalleitungen, immer strenger werden. Das wird besonders ersichtlich, wenn man mehrere zusammengehörende Signale betrachtet, die von einem Takt abhängen. Wenn die Quelle zur steigenden Taktflanke die Signalleitungen mit den unterschiedlichen Längen treibt und die Senke die Signale zum selben Takt einliest, dann hat die Senke ein sehr schmales Zeitfenster, unter der Berücksichtigung der Setup und Hold-Zeiten, bei dem die Signalleitungen die korrekten Werte aufweisen. Bei den größeren Unterschieden in der Länge der Leitungen und sehr hohem Takt der Verarbeitung können sogar die falschen Werte von der Senke eingelesen werden.

Durch das Einbetten des Taktes in den Datenstrom ist es nicht mehr notwendig, dass die einzelnen *Lanes* innerhalb eines *Links* die gleichen Längen haben müssen. Um aus dem Empfangsignal den Sendetakt zurückgewinnen zu können, muss das Empfangssignal hinreichend viele Signalflanken aufweisen. Das bedeutet, dass lange Folgen von ’1’ oder ’0’ ohne Flanken in der Übertragung zu vermeiden sind. Die 8b/10b - Kodierung schreibt vor, dass die maximale Länge gleicher Bits auf fünf beschränkt ist. Da nicht alle 1024 Werte gebraucht werden, können die Daten entsprechend dieser Vorgabe zu

den neuen Symbolen kodiert werden. Ein weiterer Vorteil, dass die Anforderungen für die Längengleichheit nicht mehr so hoch sind, ist der verringerte Flächenbedarf für die Leiterbahnen auf der Platine, da diese nicht mehr im Zickzack verlegt werden müssen, um die Laufzeitunterschiede auszugleichen.

Eine weitere Funktionalität, die 8b/10b - Kodierung anbietet, ist die Möglichkeit einige Bitübertragungsfehler zu erkennen. Die zu übertragenden Daten werden so kodiert, dass die Anzahl der Einsen sich maximal um zwei von der Anzahl der Nullen in einem Symbol unterscheidet. Dies gilt auch für die Anzahl der Nullen gegenüber der Anzahl der Einsen. Der Empfänger kann anhand der Differenz der Einsen und Nullen entscheiden, ob das Symbol überhaupt gültig ist. Diese Methode ist allerdings nicht zuverlässig, da manche Fehler die Symbole so verändern, dass sie als gültig erkannt werden. Eine Fehlerkorrektur ist damit nicht möglich.

Ein weiterer Effekt, der durch die gleichmäßige Verteilung von Einsen und Nullen in den Symbolen entsteht, ist der Gleichspannungsausgleich. Dabei wird versucht den Pegel der durchschnittlichen Gleichspannung einer einzelnen Datenleitung in der Mitte zwischen den jeweiligen logischen Pegeln zu halten. Somit wird die Wahrscheinlichkeit der Intersymbolinterferenz verringert, in dem die Datenleitung sich nicht schnell genug von einem Zustand in den anderen umladen lässt.

Wie bereits am Anfang des Kapitels kurz erwähnt wurde, benutzt die *Bitübertragungsschicht* zwölf spezielle, 10-Bit lange Symbole, um z.B. den Anfang und das Ende eines Paketes zu kennzeichnen. Dabei werden je nach dem Ursprung des Paketes eigene, dafür vorgesehene „Marker“ benutzt.

Paketübertragung

Die Abbildung 3.4 zeigt beispielhaft, wie die einzelnen Bytes eines Paketes, auf dem Link verteilt und gesendet werden. Hierbei handelt es sich um einen x4 Link.

Auf der Empfangsseite verfügt jede *Lane* über einen eigenen Empfangspuffer. Dieser Puffer wird benötigt um aus den seriell ankommenden Bytes wieder ein komplettes Paket zusammenzustellen. Obwohl die Bytes vom Sender auf allen *Lanes* alle zum gleichen Zeitpunkt gesendet werden, kommen diese beim Empfänger nicht mehr synchron an. Laut PCIe-Spezifikation darf die Laufzeitdifferenz von *Lane* zu *Lane* bis zu 20 Nanosekunden betragen. Die Empfangspuffer dienen dazu, die Laufzeitdifferenz, die z.B. wegen der unterschiedlichen Längen der *Lanes* entsteht, zu kompensieren. Die tatsächliche Zeit, die kompensiert werden muss, wird während der Initialisierungsphase des *Links* von beiden Kommunikationspartnern ermittelt. Die Initialisierung erfolgt jedes mal, wenn das System eingeschaltet wird oder wenn ein neuer Kommunikationspartner auf der anderen Seite des *Links* erkannt wird. Während der Initialisierungsphase wird die Linkbreite und Datenrate festgelegt, zudem es werden die ersten „Credits“ für die Datenflusssteuerung ausgetauscht.

Der Sender koppelt den gewünschten Pegel auf die Sendeleitungen über die Kondensatoren ein. Der Gleichspannunganteil des Senders bleibt dem Empfänger verborgen. Dadurch können sowohl der Empfänger als auch der Sender ihre eigenen Gleichspannungspegel besitzen, die sie zum Betrieb brauchen.

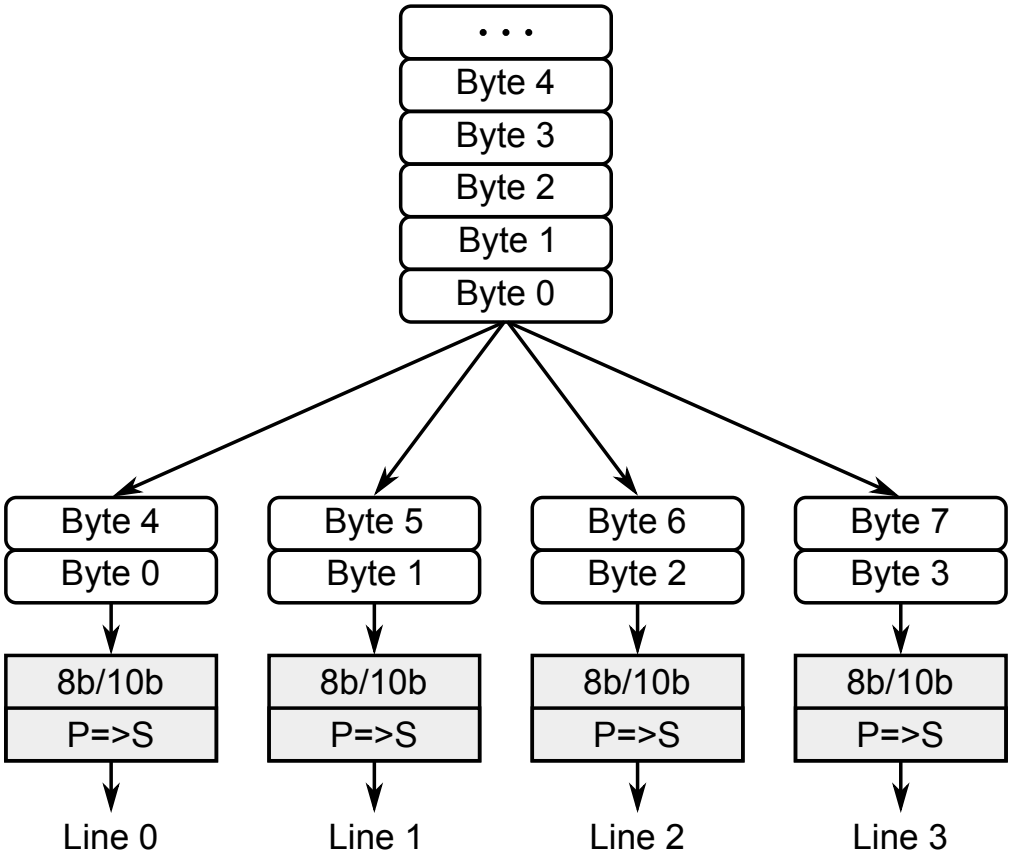

Abbildung 3.4: Aufteilung eines Pakets auf dem x4 Link.

Deemphase (Deakzentuierung)

Die PCIe-Spezifikation beschreibt ein spezielles Verfahren, das angewendet wird, um der Intersymbolinterferenz entgegenzuwirken.

Mit der steigenden Frequenz und der sinkenden Zeit für ein einzelnes Bit spielen kapazitiven Effekte eine Rolle. Die 8b/10b - Kodierung beschränkt die maximale Anzahl an Bits mit der gleichen Wertigkeit auf fünf. Bei der Übertragung von fünf gleichen Bits lädt sich das gesamte System, bestehend unter anderem aus den differentiellen Leitungen, auf einen bestimmten Wert auf. Wenn z.B. nach fünf logischen '1' eine logische '0' folgt und dann wieder weitere logischen Einser gesendet werden, dann kann es passieren, dass die ergebende Ladung nicht schnell genug abtransportiert werden kann, so dass die logische '0' vom Empfänger nicht registriert werden kann. Das ist der Effekt der Intersymbolinterferenz: die vorhergehenden '1' wirken noch beim Empfang von der folgenden '0' nach.

Um diesen Effekt nach Möglichkeit zu eliminieren, werden die nachfolgenden Bits der gleichen Polarität mit der kleineren Amplitude auf den *Lanes* getrieben. Im Falle von PCIe bedeutet dies die Abschwächung der Leistung von den nachfolgenden Bits

von der selben Polarität um 3,5 dB. Man kann hier auch von einer Präemphase reden, wenn man sich vorstellt, dass es immer jeweils das erste Bit einer Folge aus Bits mit der gleichen Polarität, im Vergleich zu den übrigen, verstärkt wird. Das erste Bit hat damit genug „Stärke“ die eingestellte Ladung in genügend kurzer Zeit abzuführen; der Empfänger ist in der Lage den Pegelwechsel zu erkennen. Die minimale Spannung des ersten Bits beträgt 800 mV und die minimale Spannung von nachfolgenden Bits der gleichen Polarität beträgt 505 mV.

Der Übertragungskanal hat Tiefpaßeigenschaften, d.h. die höheren Frequenzen werden stärker gedämpft als die niedrigeren. Das Signal vom Sender wird vom Übertragungskanal frequenzselektiv verzerrt beim Empfänger ankommen. Ein Pegelwechsel auf den Leitungen stellt eine hochfrequente Änderung dar.

Um das Signal beim Empfänger möglichst „glatt“ zu haben, das heißt ohne Auswirkungen von frequenzselektiven Eigenschaften des Übertragungskanals, müssen die Signale vom Sender vorverzerrt werden. Unter der Berücksichtigung von Tiefpaßeigenschaften des Kanals müssen somit entweder die hochfrequenten Anteile verstärkt oder die niederfrequenten Anteile abgeschwächt werden.

4 Treiber und Anwendung

In diesem Kapitel wird die im Rahmen dieser Arbeit entstandene Anwendung und der entwickelte Treiber vorgestellt. Dabei wird besonders auf die Auswahl der Werkzeuge und der eingesetzten Frameworks eingegangen.

Als Grundlage für die Entwicklung des Treibers und der PC-Anwendung, dient das Referenzdesign der Firma Xilinx, das für das Virtex-5 FPGA (Field Programmable Gate Array) ML555 PCIe-Entwicklungsboard bereitgestellt wird. Dieses Design umfasst den HDL-Code für den FPGA-Baustein, einen Treiber und eine Anwendung für das Windows XP Betriebssystem. Die Anwendung und der Treiber sind nur in binärer Form dem Referenzdesign beigefügt.

Das Referenzdesign ist eine Implementierung der von der Karte initialisierten Datenübertragung mittels direkten Speicherzugriffs (DMA) zwischen dem Host-PC und der Karte über die PCIe- Schnittstelle. Dabei werden die Daten zwischen dem Arbeitsspeicher des PCs und dem DDR2 Speicher auf dem Entwicklungsboard ausgetauscht. Die Aufgabe des Treibers und der Anwendung sind die Allozierung und Initialisierung der Pufferspeicher auf dem PC und Einstellung der Speicherdirektübertragungen.

Die bei dieser Arbeit verwendete Entwicklungsplattform ist die „XtremeDSP Development Platform — Virtex-5 FPGA ML506 Edition“ von der Firma Xilinx. Das Referenzdesign lässt sich auch auf dieser Karte implementieren. Im Vergleich zu der ML555, die über eine x8 PCIe-Schnittstelle verfügt, verfügt die ML506 Karte über eine x1 PCIe-Schnitstelle. Die Karte zeichnet sich u.a. durch folgenden Eigenschaften:

- XC5VSX50TFFG1136 Virtex-5 FPGA Baustein
- DDR2 SODIMM (256 MB)
- JTAG Programming Interface
- PCI Express Edge Connector (x1 Endpoint)
- GTP: PCIe

Die Entwicklung des Treibers und der Anwendung erfolgte unter dem Linux-Betriebssystem „Fedora Core 10“ mit dem Linuxkernel
2.6.27.24-170.2.68.fc10.i686.PAE. Die GUI-Anwendung wurde mit Hilfe der Entwicklunsumgebung KDevelop und des QT-Frameworks entwickelt.

4.1 Registerbeschreibung

Die Steuerung der Speicherdirektzugriffe und die Abfrage deren Zustände erfolgt über einen Satz von Registern. Diese Register werden mit Hilfe eines BAR0 [1] Registers in den PCI Adressraum des PCs eingeblendet. Diese Register dienen als eine Schnittstelle zwischen dem Referenzdesign und der CPU. Die CPU kann auf die Register mit Hilfe der lesenden und schreibenden Speicher-Transaktionen über den Bus zugreifen. Über diese Schnittstelle wird auch die Anforderungen zum lesenden und schreibenden Speicherdirektzugriff entgegengenommen.

Alle Speicherdirektzugriffe werden über die DMA-Register definiert. Die CPU leitet die Datenübertragung über die Initialisierung der Steuerregister ein. Das Beenden der Übertragung wird über die Statusregister angezeigt. Alle Register sind 32 Bit breit.

1. DMAWAS, `x00` : DMA Write: Quelladresse. Die Startadresse eines Bereiches im RAM auf der Karte, aus dem die Daten in den PC Speicher transferiert werden.

2. DMAWAD_L, `x04` : DMA Write: Zieladresse. Beinhaltet die unteren 32 Bit der Adresse des Pufferspeichers im PC, in den geschrieben wird. Dieses Register wird sowohl bei Systemen mit 32 Bit breiten Adressen als auch bei Systemen mit 64 Bit breiten Adressen verwendet.

3. DMAWAD_U, `x08` : DMA Write: Zieladresse. Beinhaltet die oberen 32 Bit der Adresse des Pufferspeichers in einem System mit 64 Bit breiten Adressen, in den die Daten geschrieben werden. Wird nur bei Systemen mit 64 Bit breiten Adressen verwendet.

4. DMARAS_L, `x0C` : DMA Read: Quelladresse. Beinhaltet die unteren 32 Bit der Adresse des Pufferspeichers im PC, von dem die Daten gelesen werden. Es wird sowohl bei Systemen mit 32 Bit breiten Adressen als auch bei Systemen mit 64 Bit breiten Adressen verwendet.

5. DMARAS_U, `x10` : DMA Read: Quelladresse. Beinhaltet die oberen 32 Bit der Adresse des Pufferspeichers im PC, von dem die Daten gelesen werden. Dieses Register wird nur bei Systemen mit 64 Bit breiten Adressen verwendet.

6. DMARAD, `x14` : DMA Read: Zieladresse. Die Startadresse eines Bereichs im RAM auf der Karte, in den die Daten aus dem PC Speicher transferiert werden.

7. DMAWXS, `x18` : DMA Write: Übertragungsgröße. Dient zur Festlegung der Anzahl der Bytes, die beim DMA Write übertragen werden sollen. Die Daten werden aus dem RAM-Speicher der Karte in den RAM-Speicher des PCs übertragen.

8. DMARXS, `x1C` : DMA Read: Übertragungsgröße. Dient zur Festlegung der Anzahl der Bytes, die beim DMA Read übertragen werden sollen. Die Daten werden aus dem RAM-Speicher des PCs in den RAM-Speicher der Karte übertragen.

[1] Dieses Register ist ein fester Bestandteil jedes PCI/PCIe Gerätes. Das BAR0 Register erhält die Adresse vom Betriebssystem, unter der das Gerät erreichbar wird.

9. Reserved, `x20` : Reserviert.
10. Reserved, `x24` : Reserviert.
11. DMACST, `x28` : DMA Steuerungs- und Statusregister. Die Bedeutung der darin vorhandenen Bits wird in der Tabelle 4.1 erläutert.
12. Reserved, `x2C` : Reserviert.
13. DMAWRP, `x30` : DMA Write: Übertragungszähler. Dieser 32 Bit breite Nurlese-Zähler kann zur Bestimmung des Datendurchsatzes der DMA Write Transaktionen verwendet werden. Dieser Zähler lässt sich nicht zurücksetzen, deswegen muss er vor dem Starten und direkt nach dem Beenden der Transaktion zur Bestimmung des Durchsatzes ausgelesen werden. Der Zähler läuft an, wenn das „Start Write DMA“ Bit gesetzt wird und stoppt, wenn das „End of Frame“ Bit in letzten Paket erkannt wird.
14. DMARDP `x34` : DMA Read: Übertragungszähler. Dieser 32 Bit breite Nurlese-Zähler kann zur Bestimmung des Datendurchsatzes der DMA Read Transaktionen verwendet werden. Dieser Zähler lässt sich nicht zurücksetzen, deswegen muss er vor dem Starten und gleich nach dem Beenden der Transaktion zur Bestimmung von Durchsatz ausgelesen werden. Der Zähler läuft an, wenn das „Start Read DMA“ Bit gesetzt wird und stoppt, wenn das „End of Frame“ Bit im letzten *Read Compleation* Paket erkannt wird.

4.2 Treiber

Wie bereits im Kapitel 2.3 erwähnt wurde, kann der Kernel um weitere Funktionalitäten zur Laufzeit erweitert werden. Diese Erweiterung kann z.B. durch das Kernelmodul erfolgen. Mit Hilfe dieser Technik können auch Treiber zum Kernel hinzugefügt werden. Die Kernelmodule bestehen aus dem Objektcode, der noch nicht zum lauffähigem Programm gelinkt wurde. Die Module können mit dem Programm `insmod` zum Kernel dynamisch hinzugelinkt und mit dem Programm `rmmod` wieder entfernt werden.

Das verwendete Referenzdesign lässt sich zu der Klasse der sogenannten Character-Devices zuordnen. Der Treiber für diese Klasse der Geräte muss mindestens die folgenden System-Aufrufe (2.3.1) implementieren: `open`, `close`, `read`, und `write`. Der System-Aufruf `mmap` wird dazu benutzt, einen bestimmten Speicherbereich, entweder für die Anwendung oder für den Treiber, einzublenden. In diesem Bereich ist es möglich, trotz der Zugehörigkeit des Gerätes zu der Character-Device Klasse, wahlfrei auf die Daten zuzugreifen. Framegrabber dienen hier als Beispiel für solche Geräte.

Der Quellcode für den Treiber besteht aus den zwei Quellcode-Dateien: `ML506_Modul.c` und `Ioctrl.h`, in denen alle benötigten Funktionen implementiert sind.

Die notwendigsten Teile eines „lauffähigen“ Moduls sind zwei Funktionen, die jeweils beim Laden und beim entladen des Moduls vom Kernel aufgerufen werden, in diesem Falle sind es die Funktionen:

Tabelle 4.1: Bits des Steuerungs- und Statusregisters

Name	Bitposition	Beschreibung
Write DMA Start	0	Startet den schreibenden Speicherdirektzugriff. Mit dem Schreiben einer '1' an diese Stelle wird der gewünschte schreibende DMA-Zugriff gestartet. Dieses Bit setzt sich automatisch zurück.
Write DMA Done	1	Dieses Bit ist gesetzt, wenn der schreibende DMA-Zugriff beendet wird. Um dieses Bit zu '0' zu setzen, muss an diese Stelle eine '1' geschrieben werden.
Read DMA Start	2	Startet den lesenden Speicherdirektzugriff. Mit dem Schreiben einer '1' an diese Stelle wird der gewünschte lesende DMA-Zugriff gestartet. Dieses Bit setzt sich automatisch zurück.
Read DMA Done	3	Dieses Bit wird gesetzt, wenn der lesende DMA-Zugriff beendet wird. Um dieses Bit zu '0' zu setzen, muss an diese Stelle eine '1' geschrieben werden.
DDR RAM Ready	4	Dieses Bit wird zu '1' gesetzt, wenn die Schnittstelle zum DDR2 Speicher erfolgreich initialisiert wurde.
Reserved	32..5	Nurlese-Bits, ohne Verwendung.

- `ML506_init`
- `ML506_exit`

Die Makros `module_init` and `module_exit` legen die Rollen der übergebenen Funktionen fest:

```
1 module_init(ML506_init);
2 module_exit(ML506_exit);
```

Eine weitere Besonderheit von Linux ist das `MODULE_LICENSE(„Dual BSD/GPL")` Makro. Es teilt dem Kernel mit, dass es sich um eine freie Software handelt und dass der Quellcode gemäß der Lizenzbedingungen zur Verfügung steht. Beim Fehlen dieser Zeile erzeugt der Kernel mehrere Warnmeldungen beim Laden eines solchen Moduls.

Die Initialisierungsfunktion `ML506_init` wird nur einmalig beim Laden des Moduls ausgeführt. Diese Funktion beendet sich sofort, wenn die Initialisierungsaufgaben durchgeführt wurden. Die Funktion „registriert“ den Treiber beim Kernel, und teilt ihm mit, für welches Gerät dieses Modul ein Treiber ist und welche Aufgaben es als Treiber erledigen kann. Die Deinitialisierungsfunktion muss alle Aktionen der Initialisierungsfunktion rückgängig machen, damit der Kernel weiß, dass die bereitgestellten Funktionalitäten nicht mehr zur Verfügung stehen.

Bei dem Referenzdesign handelt es sich um eine PCIe-Karte. Wie bereits im Kapitel 3.2 erwähnt wurde, ist die Softwareschnittstelle zur Anbindung von PCI und PCIe Geräten voll kompatibel. Für die System- und Treiberentwicklung bedeutet das, dass die bereits für PCI-Bus vorhandenen Funktionen des Kernels benutzt werden können.

Bei der Initialisierung wird die Kernelfunktion `pci_register_driver` aufgerufen. Als Übergabeparameter erwartet diese Funktion eine spezielle Treiber-Struktur und liefert ein ganzzahliges Ergebnis zurück. Diese `struct pci_driver` Struktur ist ein zentrales Element aller PCI/PCIe Treiber. Damit die Registrierung des Treibers beim Kernel fehlerfrei abläuft muss diese Struktur korrekt erzeugt werden. Das Listing 4.2 zeigt den Aufbau einer solchen Struktur auf.

```
1  /**
2  *Dient der korrekten Registrierung des Treibers
3  *beim Kernel.
4  */
5  static struct pci_driver ML506_driver = {
6      .name       = "ML506_driver",
7      .id_table   = ML506_ids,
8      .probe      = ML506_probe,
9      .remove     = __devexit_p(ML506_remove),
10 };
```

Diese Struktur beschreibt den Treiber für das PCI-Subsystem. Sie beinhaltet den Treibernamen, der nach dem Laden des Moduls im System unter */sys/bus/pci/drivers/* erscheint. Der Name muss natürlich eindeutig innerhalb des Kernels sein. Daneben enthält die Struktur die Zeiger auf die sogenannten Callback-Funktionen, die vom PCI-Subsystem des Kernels aufgerufen werden, und einen Zeiger auf eine weitere Struktur: `struct pci_device_id *id_table;` Diese Struktur bildet eine Liste der von diesem Treiber unterstützten Geräte. Die Liste beinhaltet die *Vendor_IDs* und die *Device_IDs*. Diese Nummern werden von der PCI-SIG den Geräteherstellern vergeben und bei den PCI Karten im Konfigurationsspeicher einprogrammiert. Das Listing 4.2 zeigt diese Struktur. Sie bildet eine Liste mit den Einträgen für die vom Treiber unterstützten Geräte nach.

```
1  /* Hersteller- und Geraetekennung */
2  #define XILINX_VENDOR_ID            0x10ee
3  #define XILINX_DEVICE_ID            0x0007
4
5  static const struct pci_device_id ML506_ids[] = {
```

```
6       { PCI_DEVICE(XILINX_VENDOR_ID, XILINX_DEVICE_ID) },
7       { 0 },
8   };
```

Das PCI-Subsystem erwartet, dass der PCI-Treiber folgende Funktionen zur Verfügung stellt:

- `probe` : Diese Funktion wird vom PCI-Kern aufgerufen, wenn der Kernel ein Gerät erkennt, dem noch kein Treiber zugeordnet ist und das in der Liste der unterstützten Geräte vorkommt. Der Treiber muss dann entscheiden, ob er die Kontrolle für dieses Gerät übernimmt oder nicht. Die Mitteilung dieser Entscheidung erfolgt durch den Rückgabewert. Der Rückgabewert ist NULL, wenn die Kontrolle übernommen wird, ansonsten eine negative Zahl.

- `remove` : Diese Funktion wird entweder aufgerufen wenn das Gerät vom System entfernt wird, oder wenn der Treiber für dieses Gerät entladen wird.

- `suspend` : Diese Funktion ist optional und ist nicht zwingend notwendig für den Betrieb des Geräts. Sie wird aufgerufen, wenn die Variable *state* der internen Kernelstruktur `pci_dev` den Wert *suspend* annimmt. Die Kernelstruktur `pci_dev` repräsentiert ein PCI-Gerät innerhalb des Kernels.

- `resume` : Diese Funktion ist ebenfalls nicht zwingend notwendig für den Betrieb des Geräts. Sie wird immer aufgerufen, wenn die interne Kernelstruktur `pci_dev` den *suspend*-Zustand verlässt.

Die zwei letzten Funktionen werden gebraucht, wenn die Karte bestimmte Aktionen durchführen muss, bevor sie z.B. vom Energie-Management des System abgeschaltet bzw. wieder eingeschaltet wird.

Bei dem Treiber für das Referenzdesign wurden nur die notwendigen Funktionen `ML506_probe` und die `ML506_remove` implementiert. Wie beim Laden des Treibermoduls die Funktion `pci_register_driver` aufgerufen wird, so wird beim Entladen des Moduls die Funktion `pci_unregister_driver` aufgerufen. Dabei werden die Zuweisungen der PCI-Geräte zu dem Treiber aufgehoben. Bevor die Struktur `struct pci_driver` des Treibers beim Kernel abgemeldet wird und damit die belegten Ressourcen wieder frei werden, wird noch die `remove` - Funktion (`ML506_remove`) aufgerufen, um ein von diesem Treiber kontrolliertes Gerät z.B. in einen sicheren Zustand zu bringen.

4.2.1 Funktion `probe`

Alle Treiber, die für PCI/PCIe-Geräte beim Kernel angemeldet werden, liefern auch die Liste der von diesen Treibern unterstützten Geräte. Der Kernel vergleicht die Hersteller- und die Gerätenummern der erkannten PCI-Geräte mit den Nummern, welche die registrierten Treiber mitteilen. Wenn es eine Übereinstimmung gibt, dann versucht der Kernel das erkannte PCI-Gerät an den entsprechenden Treiber zu übergeben. Diese Übergabe erfolgt mit Hilfe der bereits erwähnten `probe`-Funktion (`ML506_probe`). Der Kernel ruft

diese Funktion des entsprechenden Treibers auf und übergibt die Adresse auf die interne Struktur, die das erkannte PCI-Gerät repräsentiert. Diese Struktur fasst natürlich alle Informationen zusammen, die allen PCI/PCIe-Geräten gemeinsam sind und weiterhin auch noch solche, die zur Verwaltung der Geräte intern im Kernel notwendig sind.

Das Referenzdesign verfügt über die gerätespezifischen Register und weiteren gerätespezifischen Eigenschaften, die im Treiber eine Rolle spielen. Dafür wurde eine weitere Struktur eingeführt, die das eigentliche Gerät abbildet und alle notwendigen Variablen für den Treiber bereithält und zusammenfasst. Diese Struktur ist die `struct ML506_dev` und sie beinhaltet folgende Felder:

- Variablen für die virtuelle und physikalische Adresse des Registers BAR0.
- Variablen für physikalische und virtuelle Adressen des DMA-Puffers.
- Variablen für die Werte aus den Registern.
- Zeiger auf die kernel-interne Struktur `struct pci_dev *pci_dev`.
- Eine Struktur, die ein Character-Device beschreibt:
  `struct cdev ML506_cdev`.

Als erster Schritt in der `probe`-Funktion wird der notwendige Speicher für die `struct ML506_dev` Struktur angefordert. Beim Erfolg wird der Speicherbereich mit Nullen beschrieben und die Ausführung geht zum nächsten Schritt über. Die Übernahme des Gerätes durch den Treiber erfordert schrittweises Vorgehen. Beim Fehlschlagen eines Schrittes müssen alle bis dahin belegten Ressourcen wieder freigegeben werden. Die Freigabe der belegten Ressourcen erfolgt dann in der umgekehrten Reihenfolge. Erst nach der Freigabe der belegten Ressourcen wird die `probe`-Funktion mit einem negativen Rückgabewert beendet.

Nach dem die Struktur den Speicher im Kernelspeicherbereich zugewiesen bekommen hat, werden die ersten Variablen mit Werten belegt. Es wird der Name gesetzt und der übergebene Zeiger auf die `pci_dev` Struktur gespeichert. So kann der Treiber dann immer auf die kernel-interne Darstellung des Gerätes zugreifen. Die `pci_dev` Struktur bietet zusätzlich eine Möglichkeit über sich selbst auf die „privaten“ Daten des Treibers zuzugreifen. Die „privaten“ Daten des Treibers bestehen in diesem Fall aus der angelegten Struktur `ML506_dev`. Die Adresse der angelegten Struktur wird in die dafür vorgesehene Variable der `pci_dev` Struktur kopiert. Das heißt, wenn der Kernel eine Treiberfunktion aufruft und dabei nur den Zeiger auf die `pci_dev` Struktur übergibt, so hat man dennoch die Möglichkeit auf die spezifische, angelegte, gerätebeschreibende Struktur zuzugreifen.

Der nächste Schritt sieht eine Reservierung von 1 Megabyte großen, zusammenhängenden Speicherbereich für den Puffer im Kernelbereich vor. Dafür wird die Funktion `pci_alloc_consistent` benutzt. Im fehlerfreien Fall liefert sie zwei Ergebnisse: die virtuelle und die physikalische Startadresse des reservierten Bereiches. Diese werden in der `ML506_dev`-Struktur gespeichert. Die Funktion, welche die entsprechenden Ressourcen freigibt ist die `pci_free_consistent`

Als Nächstes wird die Kernelfunktion `pci_enable_device(struct pci_dev *dev)` aufgerufen. Damit weiß der Kernel welches Gerät aktiviert werden soll und wenn das Gerät vorhanden ist, werden die Interrupts zu den Leitungen und die Ein- und Ausgabebereiche zugewiesen. Erst nach dem Ausführen dieser Funktion ist der Treiber in der Lage auf die Ressourcen des Gerätes zuzugreifen. Der Kernel stellt mehrere Funktionen für den Zugriff auf die verschiedenen Speicherbereiche der PCI/PCIe-Geräte zur Verfügung. Die CPU hat selbst keine Möglichkeit auf die Hardware zuzugreifen, sie muss den Umweg über den Chipsatz nehmen, d.h. sie schreibt in und liest aus den bestimmten Registern des PCIe-Kontrollers. Diese Zugriffe sind Herstellerabhängig und werden durch die Treiber für diese Chipsätze abgedeckt. Die Abstraktion durch den Kernel stellt eine einheitliche Schnittstelle für die Zugriffe z.B. auf den Konfigurationsspeicher der Karten dar. Um das Gerät wieder zu deaktivieren wird die Funktion `pci_disable_device` gebraucht.

Bevor die von dem Gerät bereitgestellten Speicherbereiche benutzt werden dürfen, müssen diese noch als belegt markiert werden, damit es keine mehrfachen Zugriffe gibt. Die Reservierung erfolgt mit Hilfe der Funktion `pci_request_regions`. Dabei wird die Struktur des Gerätes `pci_dev` und der Treibername übergeben. Wenn die durch das Gerät angeforderten Ressourcen noch nicht als belegt markiert waren, dann werden diese als belegt markiert. Erst nach dem diese Funktion erfolgreich ausgeführt wurde, darf man auf die Ressourcen zugreifen. Die Funktion `pci_release_regions` markiert die belegten Ressourcen wieder als frei.

Im nächsten Schritt werden die Adressen für das BAR0 Register ermittelt und in der `ML506_dev` Struktur gespeichert. Dabei wurde der notwendige Quellcode in eine eigene Funktion ausgegliedert. Diese Funktion ist die `static int __devinit map_bars`. In dieser Funktion werden die Start- und die Endadresse des Speicherbereichs ermittelt. Dabei wird es noch überprüft, ob sich die Adressen tatsächlich von einander unterscheiden und somit einen Speicherbereich aufspannen. Daneben erfolgt noch die Überprüfung, ob es sich bei dieser Ein- / Ausgaberessource um einen Speicher handelt. Mit diesen Tests und der Kenntnis über den Aufbau der Hardware kann im Treiber sichergestellt werden, dass es sich um ein unterstütztes und korrekt funktionierendes Gerät handelt. Die virtuelle Adresse im Speicherbereich des Kernels erhält man mit Hilfe der Funktion `ioremap`. In Linux gibt es zum Teil mehrere Funktionen, die den gleichen Zweck erfüllen, wie auch in diesem Fall. Die Alternative zu `ioremap` ist z.B. `pci_iomap`. Die entsprechende Funktion, die den eingeblendeten Speicherbereich wieder freigibt, ist die `pci_iounmap`. Diese Funktion wird aus der `static void free_bars` heraus aufgerufen.

Alle bis jetzt verwendeten Funktionen werden vom PCI-Subsystem des Kernels zur Verfügung gestellt. Im weiteren Verlauf folgen noch die Funktionen, die das Gerät als ein Character-Device beim Kernel initialisieren und anmelden.

Als Erstes bei der Einrichtung eines Character-Devices erfolgt die Reservierung einer sogenannten *major*-Nummer. Die Zugriffe auf die Peripherie erfolgen meistens über die Gerätedateien, die sich im Verzeichnis */dev* befinden. Eine Gerätedatei ist die Schnittstelle für den Benutzer und für die Anwendungen um mit der Hardware zu kommunizieren. Die Zugriffe erfolgen dann über die Dateioperationen, wie `open`, `close`, `read`, `write` und weitere. Beim Auslösen einer Operation über eine Gerätedatei muss der

Kernel wissen an welchen Treiber er die Operation weiterleiten muss. Dafür ist die *major*-Nummer vorgesehen. Die Reservierung der *major*-Nummer erfolgt mit der `int register_chrdev_region` Funktion. Als einen der Parameter muss dabei die gewünschte zu reservierende Nummer übergeben werden. Es gibt eine Liste mit den bereits vergebenen Nummern, die für viele Linuxsysteme gilt. Der Nachteil dieser Vorgehensweise ist, dass die Nummer von einem anderen Treiber bereits belegt sein könnte. Als Lösung für das Problem dient die dynamische Zuweisung der *major*-Nummer mit Hilfe der `int alloc_chrdev_region` Funktion. Diese Funktion ist in der Lage gleich mehrere Nummern zu reservieren. Sie erwartet einen Zeiger auf die Variable, welche dann die Nummer beinhaltet, den Namen des Gerätes, unter dem es in */proc/devices* erscheint und die Anzahl der gewünschten Nummern. Neben einer *major* hat jedes Gerät auch eine *minor*-Nummer. Diese Nummer wird nicht vom Kernel sondern nur von Treibern selbst verwaltet und benutzt. Die *minor*-Nummer kann z.B. für die Verwaltung mehrerer Geräte desselben Typs durch einen einzigen Treiber eingesetzt werden. Die Funktion `unregister_chrdev_region` gibt die reservierte *major*-Nummer wieder frei.

Nach dem die *major*-Nummer im System reserviert ist, müssen noch die entsprechenden Treiberfunktionen mit dieser Nummer verknüpft werden. Die Verknüpfung erfolgt mit Hilfe einer weiteren Struktur die `struct file_operations`. Diese Struktur ist in *<linux/fs.h>* definiert und stellt eine Sammlung von Zeigern auf die zu implementierenden Funktionen dar. Nicht alle definierten Funktionen müssen auch tatsächlich implementiert werden, damit das Gerät korrekt funktioniert. Nur die vom Gerät unterstützten Funktionen müssen implementiert werden. Die Zeiger auf Funktionen, die nicht unbedigt notwendig sind und somit ggf. nicht vom Treiber bereitgestellt werden, müssen mit NULL belegt sein. Alle anderen Funktionen müssen im Treiber implementiert werden. Der Quellcode-Ausschnitt 4.2.1 zeigt die `file_operations` Struktur.

```
1 static const struct file_operations ML506_fops = {
2     .owner      = THIS_MODULE,
3     .open       = ML_open,
4     .release    = ML_release,
5     .mmap       = ML_mmap,
6     .ioctl      = ML_ioctl,
7     .read       = ML_read,
8     .write      = ML_write,
9 };
```

Wie man dem Quellcode entnehmen kann, entsprechen die Funktionen den Systemaufrufen, die wiederum von der Anwendung getriggert werden. Das Feld `.owner` stellt eine Ausnahme dar und ist keine Funktion. Dieses Feld ist notwendig um das Entladen des Treibers zu verhindern, wenn eine der Operationen der Gerätedatei aktiv ist. Der Funktion `ML_ioctl` kommt eine besondere Bedeutung zu, denn mit Hilfe dieser Methode können gerätespezifischen Funktionen realisiert werden, die nicht über die vorhandenen Systemaufrufe abgedeckt werden können.

Nach dem die Gerätedatei geöffnet wurde, kann die Anwendung auf die aufgelisteten Funktionen zugreifen. Alle Character-Devices werden im Kernel intern mit Hilfe

der Struktur vom Typ `struct cdev` repräsentiert. Bevor der Kernel die Funktionsaufrufe an den Treiber weiterleiten kann, muss diese `cdev` Struktur initialisiert und dem Kernel bekannt gemacht werden. Die Initialisierung dieser Struktur erfolgt mit `void cdev_init(struct cdev *cdev, struct file_operations *fops)`. Bei der Initialisierung der Character-Device beschreibenden Struktur wird die bereits erwähnte Struktur `file_operations` übergeben. Dadurch wird dem Kernel mitgeteilt welche Operationen dieses Character-Device ausführen kann.

Das angelegte, zeichenorientierte Gerät muss dem Kernel durch Hinzufügen bekannt gemacht werden. Dies erfolgt mit dem Aufruf der Funktion `cdev_add`. Der Aufruf dieser Funktion ist die letzte Aktion in der `probe`-Funktion. Beim erfolgreichen Ausführen erscheint das Gerät im System und kann sofort angesprochen werden, weshalb es wichtig ist, dass dieses Gerät zu diesem Zeitpunkt funktionsbereit ist.

4.2.2 Funktion `remove`

Bei der Registrierung des Treibers beim Kernel wurde neben dem Zeiger auf die `probe`-Funktion auch ein Zeiger auf die `remove`-Funktion übergeben. Diese Funktion wird vom PCI Subsystem des Kernels aufgerufen, wenn z.B. der Treiber entladen wird. Dabei werden die verwendeten Ressourcen abgegeben:

1. Beenden der Einblendung des Registers BAR0 in den Kernelspeicherbereich.
2. Freigabe des 1 MB großen Pufferspeichers.
3. Rückgabe der *major*-Nummer.
4. Löschen der `cdev`-Struktur.
5. Freigabe der PCI Ein- / Ausgabebereiche und der Speicherbereiche.
6. Löschen der gerätespezifischen Struktur.

Nach dem die `remove`-Funktion abgearbeitet wurde, wird zuletzt die `exit`-Funktion des Treibers vom Kernel aufgerufen. Die Hauptaufgabe dieser Funktion ist die Abmeldung der `struct pci_driver` Struktur vom Kernel. Die Abmeldung erfolgt mit `pci_unregister_driver`, damit wird der Treiber vom System abgemeldet.

4.2.3 Funktion `open`

Die Funktion `open` wird aufgerufen, wenn die Anwendung eine Gerätedatei öffnet, welche die vorhandene Hardware repräsentiert. In dieser Funktion soll überprüft werden, ob das Gerät bereits von z.B. einer anderen Anwendung „geöffnet“ ist d.h. benutzt wird. Dafür ist eine Variable in der gerätespezifischen Struktur vorgesehen: `int inUse`. Es wird überprüft, ob der Wert dieser Variable = 1 ist. Diese Variable hat den Wert 1, wenn das Gerät bereits „geöffnet“ wurde. Der Treiber 'öffnet“ das Gerät indem er eine 1 dieser Variable zuweist.

Beim Aufruf der Funktion werden vom Kernel als Parameter nur zwei Zeiger auf die Strukturen übergeben: `struct inode *inode, struct file *file`. Es muss also einen Mechanismus geben um auf die gerätespezifische Struktur des Treibers innerhalb dieser Methode zugreifen zu können.

Die Struktur `file` repräsentiert einen sogenannten *file descriptor*, ein Objekt, das eine geöffnete Datei repräsentiert. Die Struktur `inode` selbst hingegen beschreibt die Datei und beinhaltet sehr viele Informationen über diese. In einem System kann es mehrere Deskriptoren einer Datei geben, jedoch gibt es immer nur eine Struktur *inod* von dieser Datei.

Die Struktur `inode` hat zwei Felder, die für die Treiberentwiklung interessant sind: `dev_t i_rdev` und `struct cdev *i_cdev`. Falls der *inode*-Knoten eine Gerätedatei repräsentiert, dann beinhaltet das erste Feld die Gerätenummer und das zweite den Zeiger auf die entsprechende `cdev`-Struktur, wenn es sich dabei um ein Character-Device handelt. Zur Erinnerung: die `cdev`-Struktur wurde vorher in der `probe`-Funktion vorher angelegt und dem Kernel hinzugefügt. Über die *inode* können wir auf die Struktur `cdev` zugreifen, und mit Hilfe der Kernelfunktion `container_of` können wir die gewünschte gerätespezifische Struktur erhalten.

Die über diesen Umweg gewonnene Struktur `ml506_dev` kann auf die treiberinternen Variablen zugegriffen werden. Vorteilhaft ist, dass die `file`-Struktur ein Feld namens `private_data` besitzt. In diesem Feld wird die Referenz auf die gewonnene gerätespezifische Struktur abgelegt. Bei jedem weiteren Aufruf einer der Dateioperationen auf der offenen Gerätedatei kann man so leicht auf die gerätespezifische Struktur in den Treiberfunktionen zugreifen.

4.2.4 Funktion `release`

Diese Funktion wird vom Kernel aufgerufen, wenn die vorher geöffnete Gerätedatei von der Anwendung geschlossen wird. Die Übergabeparameter sind mit denen der `open`-Funktion identisch. Den Zugriff auf die gerätespezifische Struktur erhält man über das Feld `private_data` der `file`-Struktur. In dieser Funktion werden die internen Register der Karte mit den Standartwerten beschrieben, zurückgesetzt und die `inUse`-Variable in der gerätespezifischen Struktur wieder zu 0 gesetzt: eine erneute Öffnung des Geräts / der Gerätedatei wird wieder möglich.

4.2.5 Funktion `mmap`

Die Funktion `mmap` dient der Einblendung der Speicher- oder der Ein- und Ausgaberessourcen der Hardware in den Adressraum des aufrufenden Prozesses. Das Referenzdesign reserviert 128 Bytes an Speicherressourcen über das Register BAR0. In diesen 128 Bytes befindet sich der Registersatz der Karte, wobei der Registersatz sich nur über 56 Bytes erstreckt. Mit der `mmap`-Funktion kann die Anwendung die Register in dem ihr zugewiesenen Adressraum auslesen und damit die lesenden Hardwarezugriffe durchführen. Die `mmap`-Funktion kann die Einblendung nur Seitenweise durchführen, die typische Größe einer Seite ist dabei 4096 Bytes. Zudem kann man nicht davon ausgehen, dass die

gewünschten 56 Bytes des Registersatzes ganz am Anfang des eingeblendeten Adressbereiches liegen. Daher besitzt die gerätespezifische Struktur noch eine Variable, die den Offset beinhaltet, ab dem die Register innerhalb der Seite liegen. Bevor die Anwendung auf die Register zugreifen kann, muss sie noch den Offset beim Treiber abfragen. Beim Einblenden erlaubt der Treiber nur den lesenden Zugriff auf die Register, damit wird verhindert, dass die Anwendung über das direkte Beschreiben der Register, das heißt am Treiber vorbei, den DMA-Vorgang starten kann.

Die `mmap`-Funktion ermöglicht also direkte Zugriffe auf die Hardware direkt aus der Anwendung heraus. Sie erlaubt einen bequemeren Zugriff, diese Zugriffe aber bleiben vom Treiber unbemerkt. Das Beschreiben einzelner Register wird nur über die `ioctl`-Aufrufe erledigt.

4.2.6 Funktion `read`

Die Funktion `read` im Treiber wird aufgerufen, wenn die Anwendung den *read*-Systemaufruf über die Gerätedatei auslöst. Auch wenn die Anwendung die Daten liest, so werden diese eigentlich von der Karte aus dem eigenen DDR2 RAM Speicher in den PC RAM geschrieben. In dieser Funktion erfolgt also ein schreibender Speicherdirektzugriff.

Beim Aufruf der Funktion `read` wird als Erstes überprüft, ob die Werte für die Offsets und für die Anzahl der Wiederholungen und die Größe der Übertragung korrekt gesetzt sind. Falls irgendein Wert noch nicht gesetzt wurde aber die `read`-Funktion aufgerufen wird, so wird ein Standardwert angenommen. Als Nächstes überprüft der Treiber das Bit „0“ des Steuerungs- und Statusregisters, und wenn das Bit eine logische ’0’ darstellt, dann darf der Write-DMA Vorgang gestartet werden. Bevor die Datenübertragung erfolgt, müssen noch die entsprechenden Register auf der Einsteckkarte mit den übergebenen Werten aus der gerätespezifischen Struktur beschrieben werden. Die Übergabe der Werte erfolgt mit Hilfe der `ioctl`-Funktion und sie werden in der gerätespezifischen Struktur abgelegt. Das Schreiben in die Register erfolgt mit Hilfe der Kernelfunktion `iowrite32`. Folgende Quellcodezeile zeigt beispielsweise, wie der Register für die Größe der zu übertragenden Daten beschrieben werden kann:

```
iowrite32(ml506_dev->w_TransferSize, ml506_dev->bar0 +
                                    + DMA_W_TRANSFER_SIZE).
```

Mit dem ersten Parameter wird der Wert für die Datenmenge in Bytes übergeben und mit dem zweiten Parameter die Zieladresse, wohin der erste Parameter geschrieben werden soll. Im Feld `bar0` der gerätespezifischen Struktur `ml506_dev` befindet sich die virtuelle Kerneladresse des Registers BAR0. Das `DMA_W_TRANSFER_SIZE` ist der Offset in Bezug auf die BAR0 Adresse zu dem DMAWXS Register.

Bevor der DMA-Vorgang gestartet wird, wird noch der Wert des Schreibzählers ausgelesen. Erst jetzt startet der Treiber in einer `for`-Schleife durch das Schreiben einer ’1’ in Bit „0“ des DMACST Registers die geforderten Datenübertragungen.

Nach diesem Vorgang wird die Ausführung des Treibers für eine bestimmte Zeit angehalten um nach Ablauf dieser Zeit das Statusbit abzufragen, dass das Ende der Über-

tragung anzeigt. Der Treiber bleibt in dieser Abfrage bis das Bit gesetzt wird. Damit der Treiber im Falle einer Fehlfunktion der Hardware nicht für immer an dieser Stelle wartet, läuft ein Zähler, der mitzählt, wie oft der Treiber bereits das Bit ausgelesen hat. Dieser Zähler beschränkt das Pollen auf 100 mal.

Nach dem Beenden der `for`-Schleife liest der Treiber den neuen Wert aus dem Register DMAWRP und bestimmt die Differenz. Das Ergebnis wird in der gerätespezifischen Struktur abgelegt, um bei der Berechnung des Datendurchsatzes verwendet zu werden.

Der letzte Schritt in dieser Funktion ist das Kopieren der Daten, mit Berücksichigung aller Offsets, aus dem Kernelpufferspeicher in den Pufferspeicher des aufrufenden Prozesses. Dies erfolgt mit Hilfe der Funktion `copy_to_user`. Diese Funktion kann ein Block Daten aus dem Kernelspace in den Userspace kopieren.

4.2.7 Funktion `write`

Die Funktion `write` im Treiber wird aufgerufen, wenn die Anwendung den Systemaufruf *write* über die Gerätedatei auslöst. Auch wenn die Anwendung die Daten schreibt, so werden diese tatsächlich von der Karte, aus dem PC Arbeitsspeicher in den eigenen DDR2 RAM Speicher eingelesen. In dieser Funktion erfolgt also ein lesender Speicherdirektzugriff.

Die Vorgehensweise des Treibers in dieser Funktion ist ganz ähnlich, wie die in der Funktion `read`. Der Unterschied besteht darin, dass der Treiber zuerst die Daten aus dem Userspace, von der Anwendung, in den Kernelspace kopiert und erst danach den lesenden DMA-Vorgang startet. Das Kopieren erledigt eine weitere Kernelfunktion: `copy_from_user`.

Auch in dieser Funktion muss der Treiber die übergebenen Offsets, Adressen, Datengröße und Anzahl der Übertragungen berücksichtigen. Bei der Vorbereitung des Vorgangs werden die entsprechenden Register beschrieben und ausgewertet.

4.2.8 Funktion `ioctl`

Die Funktion `ioctl` spielt eine wichtige Rolle bei der Gerätesteuerung. Mit Hilfe dieser Funktion können beliebige Aufträge an den Treiber und die Hardware gestellt, wenn die Standardaufrufe wie „lesen“ und „schreiben“ nicht ausreichen.

Die Funktion `ioctl` wird aufgerufen, wenn der `ioctl`-Systemaufruf von der Anwendung erzeugt wird. Der Aufruf in der Anwendung hat folgendes Format: `int ioctl(int fd, unsigned long cmd, ...)`. Dabei bedeuten die Punkte als Parameter in den Klammern, dass es einen optionalen Parameter geben kann. Dabei kann dieser zusätzlicher Parameter eine Ganzzahl oder ein Zeiger auf eine beliebige Struktur sein. Das Vorhandensein und der Typ dieses Parameters hängt dann von dem verwendeten Befehl, dem zweiten Parameter, ab.

Die `ioctl`-Funktion ist eigentlich eine große `switch-case`-Anweisung, wobei jeder Befehl in einem eigenen Zweig abgearbeitet wird. Den einzelnen Befehlen sind eindeutige Nummern zugewiesen, die Zuweisung erfolgte in der `<Ioctrl.h>`-Datei. Damit die Anwendung die gleichen Befehle benutzen kann, muss diese Datei ebenfalls während der Entwicklung verwendet werden.

Der Quellcode-Ausschnitt 4.2.8 zeigt die Definitionen von zusätzlichen Befehlen und die Zuweisung der korrespondierenden Nummern.

```
1  #define ML506_IOC_MAGIC 0b11000101
2
3  #define ML_ioctl_Set_Read_Transfer_Size                    _IOW(
       ML506_IOC_MAGIC, 1, int)
4  #define ML_ioctl_Set_Read_NumberOfTransfers                _IOW(
       ML506_IOC_MAGIC, 2, int)
5  #define ML_ioctl_Set_Write_Transfer_Size                   _IOW(
       ML506_IOC_MAGIC, 3, int)
6  #define ML_ioctl_Set_Write_NumberOfTransfers               _IOW(
       ML506_IOC_MAGIC, 4, int)
7  #define ML_ioctl_Get_Bar0_Offset                           _IOR(
       ML506_IOC_MAGIC, 5, int)
8  #define ML_ioctl_Get_MaximumReadRequestSize                _IOR(
       ML506_IOC_MAGIC, 6, int)
9  #define ML_ioctl_Get_MaximumPayloadSize                    _IOR(
       ML506_IOC_MAGIC, 7, int)
10 #define ML_ioctl_Get_ReadCompletionBoundary                _IOR(
       ML506_IOC_MAGIC, 8, int)
11 #define ML_ioctl_Get_LinkWidth                             _IOR(
       ML506_IOC_MAGIC, 9, int)
12 #define ML_ioctl_Set_Write_ML506_Offset                    _IOW(
       ML506_IOC_MAGIC, 10, int)
13 #define ML_ioctl_Set_Write_HostPC_Offset                   _IOW(
       ML506_IOC_MAGIC, 11, int)
14 #define ML_ioctl_Set_Read_ML506_Offset                     _IOW(
       ML506_IOC_MAGIC, 12, int)
15 #define ML_ioctl_Set_Read_HostPC_Offset                    _IOW(
       ML506_IOC_MAGIC, 13, int)
16 #define ML_ioctl_Start_Full_Duplex_DMA                     _IOW(
       ML506_IOC_MAGIC, 14, unsigned long)
17 #define ML_ioctl_Get_Write_Performance                     _IOR(
       ML506_IOC_MAGIC, 15, int)
18 #define ML_ioctl_Get_Read_Performance                      _IOR(
       ML506_IOC_MAGIC, 16, int)
```

Wie man dem Quellcode entnehmen kann, wurden 16 zusätzliche Aufrufe definiert. Dabei gibt es eine Besonderheit bei der Vergabe der Nummern. Die Nummern dürfen nur einmal innerhalb des Systems vorkommen. Wenn dies gewährleistet ist, dann können die Fehler vermieden werden, die sich ergeben, wenn eine Anwendung an einer falschen Gerätedatei einen ioctl-Aufruf absetzt und es nicht merkt. Deswegen haben sich die Kernelentwickler auf ein bestimmtes Format geeinigt, bei dem es mehrere Bitfelder gibt, die z.B. anzeigen, ob es weitere Übergabeparameter gibt, und wenn ja, wie groß diese sind; in welche Richtung die Daten transferiert werden (vom Kernel zum User oder umgekehrt) usw. Es gibt eine spezielle 8 Bit große Zahl, die innerhalb des Treibers einmalig vorkommen darf. Mit Hilfe dieser Zahl und der Macros wie `_IO`(kein Datenaustausch), `_IOW` (Die Daten gehen von Userspace in den Kernelspace), `_IOR` und `_IOWR` werden die zusätzlichen Befehle für die `ioctl`-Aufrufe definiert. Damit der Treiber als auch die Anwendung diese Macros erkennen, muss die Headerdatei `<asm/ioctl.h>` während deren

Entwicklung eingebunden werden.

Mit der `ioctl`-Funktion lassen sich gerätespezifischen Funktionen und Aufrufe implementieren, die nicht von den vorhandenen Standardaufrufen abgedeckt werden.

In der `ioctl`-Funktion erfolgen die ersten Tests, die überprüfen, ob die aufgerufenen Befehle an den richtigen Treiber gerichtet sind. Es wird überprüft, ob die spezielle, sogenannte „magische“ Zahl des Befehls korrekt ist und ob die im Befehl angekündigte Datenflussrichtung mit der vom Treiber übereinstimmt. Wenn die Überprüfung ohne Fehler abläuft, wird der eigentliche Befehl ausgeführt.

Im Folgenden werden die zusätzlichen gerätespezifischen Befehle an den Treiber näher erläutert, die über den `ioctl`-Systemaufruf abgesetzt werden.

`ML_ioctl_Get_Read_Performance`

Liefert die Anzahl der durchgeführten DMA-Read Transaktionen vom letzten DMA-Read Vorgang (Anwendung hat „geschrieben“) zurück. Dieser Befehl wird zur Bestimmung des Datendurchsatzes genutzt.

`ML_ioctl_Get_Write_Performance`

Liefert die Anzahl der durchgeführten DMA-Write Transaktionen vom letzten DMA-Write Vorgang (Anwendung hat „gelesen“) zurück. Dieser Befehl wird zur Bestimmung des Durchsatzes benötigt.

`ML_ioctl_Start_Full_Duplex_DMA`

Startet die Funktion `ML_do_full_duplex_dma`. Diese Funktion leitet gleichzeitiges Lesen und Schreiben ein. Sie ist notwendig, da das Lesen und Schreiben gleichzeitig ablaufen soll und es keinen expliziten Systemaufruf wie `read` oder `write` gibt.

`ML_ioctl_Set_Write_ML506_Offset`

Die Anwendung setzt den Offset, von welcher Stelle die Daten aus dem DDR2 RAM der Karte in den eigenen Speicher übertragen werden sollen.

`ML_ioctl_Set_Write_HostPC_Offset`

Die PC-Anwendung setzt den Offset, an welche Stelle die Daten aus dem DDR2 RAM der Karte in den eigenen Speicher übertragen werden sollen.

`ML_ioctl_Set_Read_ML506_Offset`

Die Anwendung setzt den Offset, von welcher Stelle die Daten aus dem eigenen Speicher in den DDR2 RAM der Karte übertragen werden sollen.

`ML_ioctl_Set_Read_HostPC_Offset`

Die Anwendung setzt den Offset, an welche Stelle die Daten aus dem eigenen Speicher in den DDR2 RAM der Karte übertragen werden sollen.

`ML_ioctl_Set_Read_Transfer_Size`

Die Anwendung setzt die Größe der Daten in Bytes, die sie „schreibt“.

`ML_ioctl_Set_Read_NumberOfTransfers`

Die Anwendung setzt die Anzahl der Wiederholungen, wie oft sie „schreibt“.

`ML_ioctl_Set_Write_Transfer_Size`

Die Anwendung setzt die Größe der Daten in Bytes, die sie „liest“.

`ML_ioctl_Set_Write_NumberOfTransfers`

Die Anwendung setzt die Anzahl der Wiederholungen, wie oft sie „liest“.

`ML_ioctl_Get_Bar0_Offset`

Mit diesem Aufruf kann die Anwendung feststellen, ab welcher Stelle sich das Register BAR0 in dem eingeblendeten Bereich befindet. Das Einblenden des Registers BAR0 erfolgt in der Funktion `mmap`.

`ML_ioctl_Get_MaximumReadRequestSize`

Mit dieser Funktion kann die MaximumReadRequestSize-Eigenschaft der PCIe-Schnittstelle abgefragt werden. Der Wert dieser Eigenschaft wird während der Initialisierungsphase zwischen dem Hostsystem und der Karte ausgehandelt. Die GUI stellt diese Information in einem Infobereich dar.

`ML_ioctl_Get_MaximumPayloadSize`

Mit dieser Funktion kann die MaximumPayloadSize-Eigenschaft der PCIe-Schnittstelle abgefragt werden. Die GUI stellt diese Information in einem Infobereich dar.

`ML_ioctl_Get_ReadCompletionBoundary`

Mit dieser Funktion kann die ReadCompletionBoundary-Eigenschaft der PCIe-Schnittstelle abgefragt werden. Die GUI stellt diese Information in einem Infobereich dar.

`ML_ioctl_Get_LinkWidth`

Mit dieser Funktion kann die LinkWidth-Eigenschaft der PCIe-Schnittstelle abgefragt werden. Die GUI stellt diese Information in einem Infobereich dar.

4.2.9 Funktion `ML_do_full_duplex_dma`

Diese Funktion wird aus der `ioctl`-Funktion aufgerufen, wenn die Anwendung den vollduplexen Zugriff durchführen möchte. Im Unterschied zu den Read- und Write-DMA, die mit einem Zwischenpuffer im Kernel auskommen, braucht diese Funktion zwei davon, wenn der Datenaustausch fehlerfrei erfolgen soll. Da der DMA-Write Vorgang wesentlich schneller als der DMA-Read Vorgang abläuft, wird die Karte mit nur einem Pufferspeicher die gleichen Daten auslesen, welche im Laufe des DMA-Vorgangs bereits reingeschrieben wurden.

Der vollduplexe Datentransfer wird nur dann gestartet, wenn beide Bits: für DMA-Read und DMA-Write, im DMA Steuerungs- und Statusregister dies erlauben. Als Vorbereitung für den Datenaustausch werden alle übergebenen Parameter in die entsprechenden Register geschrieben, und die Übertragungszähler für Schreib- und Lesevorgänge ausgelesen. Im nächsten Schritt wird die eingestellte Anzahl an Wiederholungen miteinander verglichen. Wenn die Anzahl der Wiederholungen sowohl beim Lesen als auch beim Schreiben gleich ist, dann werden die beiden Startbits simultan gesetzt. Falls die Anzahl der Wiederholungen nicht gleich ist, erfolgt der Speicherdirektzugriff in beide Richtungen solange die maximale Anzahl entweder beim Lesen oder beim Schreiben nicht erreicht wurde. Mit dem Erreichen der maximalen Anzahl an Wiederholungen erfolgt der Speicherdirektzugriff nur noch in eine der beiden Richtungen. Die Vollduplex-Phase besitzt eine eigene Wartezeit beim Pollen der Fertigstellungs-Bits.

Nach dem Ende des Datentransfers kopiert der Treiber die Daten aus dem Zwischenpuffer im Kernel in den Puffer der Anwendung. Die Übertragungszähler werden noch einmal ausgelesen, deren Differenz gebildet und diese in der gerätespezifischen Struktur abgelegt.

4.3 Anwendung, grafische Benutzeroberfläche

Neben der Entwicklung des Treibers ist im Rahmen dieser Studienarbeit auch eine PC-Anwendung erstellt worden. Die PC-Anwendung greift auf die Funktionen des Treibers zu und gibt dem Benutzer über die grafische Oberfläche die Möglichkeit, die Datenübertragung bequem zu konfigurieren und zu steuern.

Die graphische Benutzeroberfläche wurde mit Hilfe von QT-Framework erstellt. QT-Framework ist eine Sammlung von C++-Klassen, die speziell für die plattformunabhängige Programmierung von Anwendungen entwickelt wurde. Diese Bibliothek wurde ursprünglich ausschließlich für die Entwicklung von grafischen Oberflächen konzipiert. Sie wurde in der Programmiersprache C++ geschrieben, nutzt strikt das Prinzip der Klassenvererbung und ist vollständig objektorientiert. Seit der Version 4.0 (Stand 2005) enthält Qt mehr als 500 Klassen mit insgesamt mehr als 9000 Funktionen. Bei Entwicklung von einer Qt-Anwendung muss nicht die gesamte Qt-Bibliothek eingebunden werden, sondern es können nach Bedarf mit Hilfe von Teilbibliotheken nur die benötigten Funktionen eingebunden werden. Qt-Framework wird für jede unterstützte Plattform in zwei Versionen angeboten: kommerziell und Open Source. Freie Verfügbarkeit, gute Dokumentation, hohe Portabilität und das Vorhandensein der integrierten Entwicklungs-

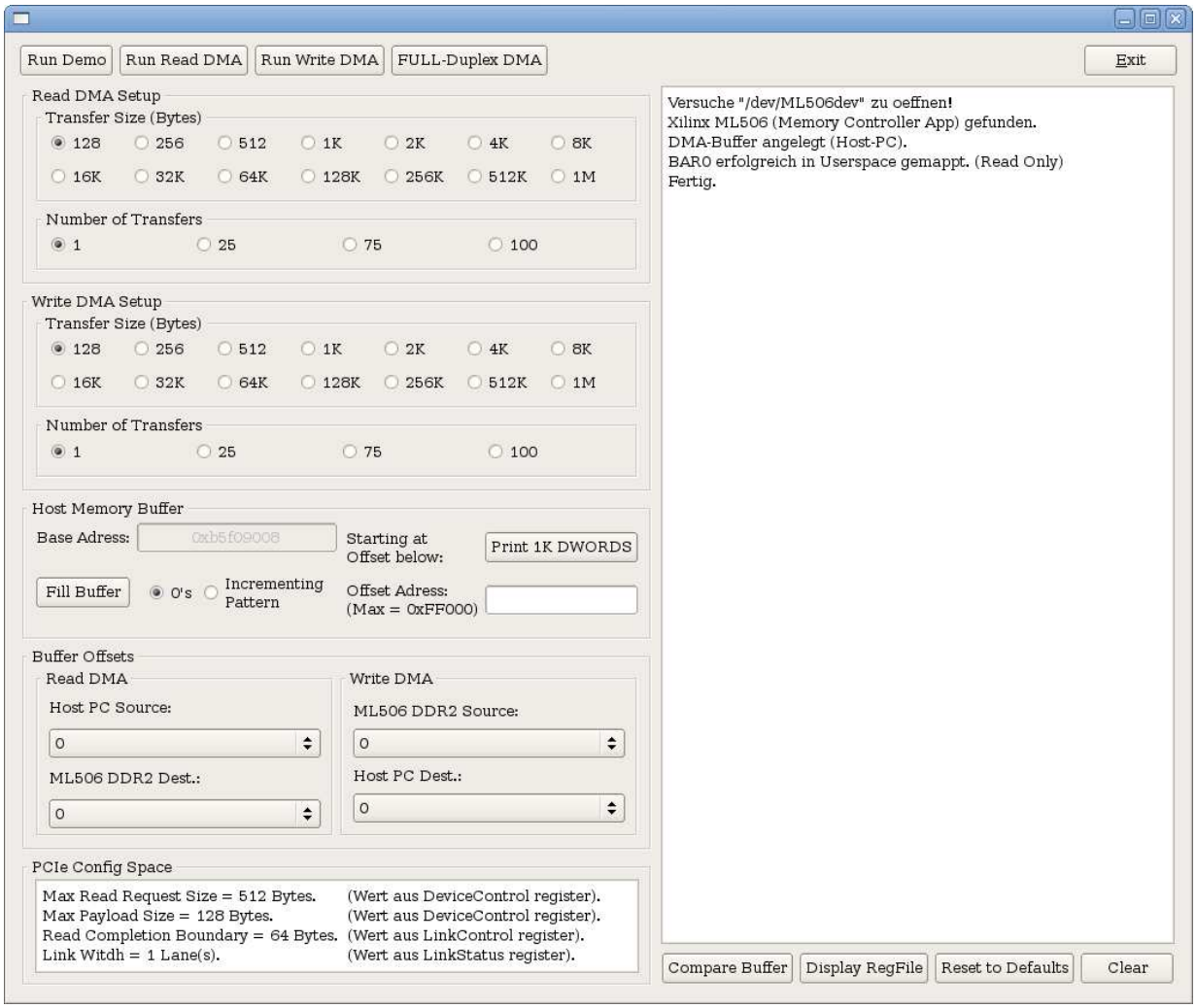

Abbildung 4.1: Grafische Benutzeroberfläche

umgebung waren ausschlaggebend für die Wahl dieser Bibliothek zur Implementierung der PC-Anwendung.

Die Benutzeroberfläche bietet direkten Zugriff auf alle Funktionen des Programms. Die Abbildung 4.1 zeigt das Hauptfenster der entwickelten PC-Anwendung.

4.3.1 Aufbau des Programms

Der Startpunkt eines C++ Programms, die Main-Funktion, befindet sich in der Datei `main.c`. Neben der Initialisierung und dem Starten der QT-Anwendung wird in der Main-Funktion versucht, die `"/dev/ML506dev"` Gerätedatei zu öffnen. Das Öffnen der Gerätedatei ist die erste Aktion, die den entwickelten Treiber betrifft. Den Erfolg der durchgeführten Aktion kann man am Wert des zurückgegebenen Dateideskriptors fest-

stellen. Bei einem negativen Wert kann es z.B. bedeuten, dass die Gerätedatei nicht vorhanden ist oder dass die Datei bereits von einer anderen Anwendung geöffnet wurde und der Treiber ein weiteres Öffnen abgelehnt hat.

Die PC-Anwendung reserviert den Speicher für den Puffer und falls das Öffnen der Gerätedatei erfolgreich war, wird versucht das Register BAR0 in den Benutzeradressbereich einzublenden. Wie bereits bei der Treiberbeschreibung erwähnt wurde, erfolgt die Einblendung immer Seitenweise. Da der Registerbereich kleiner als eine Seite ist, muss die genaue Lage dessen innerhalb der Seite noch bestimmt werden. Die Startadresse wird noch durch ein Offset, welches vom Treiber später abgefragt wird, genau bestimmt.

Die bereits gewonnenen Parameter, wie die Adresse des Pufferspeichers, die Adresse der eingeblendeten Speicherseite, die das Register BAR0 beinhaltet, und der Dateideskriptor werden an den Objekt der Qt-Anwendung durch die Funktion `init` übergeben.

Funktion `init`

Diese Funktion wird aus der `main`-Funktion aufgerufen. Anhand von Übergabeparametern werden die entsprechneden Aktionen ausgelöst. Es werden die `ioctl`-Zugriffe über den Dateideskrpitor beim Betriebssystem getriggert, die dann an den Treiber weitergegeben werden. Damit die Anwendung die gleichen `ioctl`-Aufrufe durchführen kann, die der Treiber auch anbietet, wird die `Ioctl.h` Datei eingebunden. In dieser Funktion wird der BAR0-Offset abgefragt und intern gespeichert, außerdem werden die PCIe - Link Parameter abgefragt und in der GUI in dem Infobereich dargestellt.

Bei den möglichen Fehlern, die in der `init`-Funktion erkannt werden, werden alle Steuerungselemente der Benutzeroberfäche ausgegraut/deaktiviert und die Fehlermeldungen mit Lösungsvorschlägen im rechten Teilbereich des Fensters ausgegeben.

Nach dem fehlerfreien Abarbeiten dieser Funktion ist die Anwendung bereit für die Benutzereingaben. Wie man der Abbildung 4.1 entnehmen kann, ist das Hauptfenster der Anwendung in mehrere Bereiche aufgeteilt. Im Folgenden werden diese Bereiche näher beschrieben.

`Run Demo`

`Run Demo`: Der Klick auf die Schaltfläche „Run Demo“ löst eine Reihe von Speicherdirektzugriffen aus. Dabei wird für jede unterstützte Übertragungsgröße fünf mal hintereinander ein lesender mit einem schreibenden Zugriff durchgeführt. Die Zugriffe erfolgen von der kleinsten Übertragungsgröße (128 KB) bis hin zu der größten (1 MB). Nach jeder Datenübertragung erfolgt eine Zusammenfassung der erzielten durchschnittlichen Bandbreite, die in dem rechten Teilbereich des Hauptfensters ausgegeben wird.

`Run Read DMA`

Betätigung dieser Schaltfläche löst den lesenden Speicherdirektzugriff aus. Dabei werden die eingestellten Werte aller Parameter, die für lesenden Speicherzugriff zuständig sind, berücksichtigt. Die Benutzeroberfläche beinhaltet einen separaten Bereich, in dem die Parameter für `Read DMA` eingestellt werden können. Am Ende der Übertragung(en)

wird der Datendurchsatz in dem rechten Teilbereich des Hauptfensters, dem Log-Fenster ausgegeben.

`Run Write DMA`

Mit Hilfe dieser Schaltfläche wird der schreibende Speicherdirektzugriff ausgelöst. Dabei werden die eingestellten Werte aller Parameter berücksichtigt, die für schreibenden Speicherzugriff zuständig sind. Die Benutzeroberfläche beinhaltet einen separaten Bereich, in dem die Parameter für `Write DMA` eingestellt werden können. Am Ende der Übertragung(en) wird der erzielte Datendurchsatz in dem rechten Log-Fenster ausgegeben.

`FULL-Duplex DMA`

`FULL-Duplex DMA`: Betätigung dieser Schaltfläche ermöglicht das gleichzeitige Starten von schreibenden und lesenden Speicherdirektzugriffen. Dabei werden die eingestellten Werte aller Parameter für die Speicherzugriffe beider Arten berücksichtigt. Die Parameterwerte für die Anzahl der Datenübertragungen können sich für lesende und schreibende Zugriffe unterscheiden. Die schreibende und lesende Speicherdirektzugriffe erfolgen solange gleichzeitig, bis der kleinere der beiden Werten erreicht wird. Die noch übrig bleibende Anzahl an Datenübertragungen, dann entweder nur lesen oder nur schreiben, erfolgt in einem halbduplexen Modus. Der im Schnitt erreichte Datendurchsatz aller Übertragungen wird in dem Log-Fenster zur Information ausgegeben.

`Read/Write DMA Setup`

Sowohl `Read DMA Setup` als auch `Write DMA Setup` sind die separaten Bereiche innerhalb der Benutzeroberfläche, in denen die Parameter für die lesenden und für die schreibenden Speichedirektzugriffe eingestellt werden können. Dabei kann der Benutzer die Menge an Daten auswählen (in Bytes), die übertragen werden soll. Neben der Datenmenge kann der Benutzer auch die Anzahl der Wiederholungen festlegen, wie oft die Datenübertragung stattfinden soll. Die eingestellten Werte dieser Parameter werden beim Auslösen der entsprechenden Datenübertragungen berücksichtigt.

`Host Memory Buffer`

In diesem Bereich der GUI kann man den von der PC-Anwendung angelegten Zwischenspeicher mit der Größe von 1 MB entweder mit ’0’ oder mit einem aufsteigenden Bitmuster ausfüllen. Mit der Schaltfläche „Print 1K DWORDS“ kann man 1024 Doppelworte, d.h. 1024 32 Bit Werte aus dem Zwischenpuffer ab der angegeben Offsetadresse ausgeben lassen. Die Eingabe für den Offset wird als hexadezimale Zahl interpretiert.

`Compare Buffer`

Mit dieser Schaltfläche lässt sich der Inhalt des von der Anwendung angelegten Zwischenspeichers daraufhin überprüfen, ob es dem gerade ausgewählten Füllmuster entspricht

oder nicht. Ergebnisse der Überprüfung werden im Infobereich ausgegeben. Bei der ersten Nichtübereinstimmung wird neben der Adresse auch der erwartete und der tatsächliche Wert mitausgegeben.

`Display RegFile`

Mit dem Klick auf diese Schaltfläche kann man den Inhalt aller Steuer- und Informationsregister der verwendeten PCIe Karte anzeigen lassen.

`Reset to Defaults`

Das Betätigen dieser Schaltfläche stellt die Voreinstellungen wieder her:

- Datengröße für die lesenden und schreibenden Speicherzugriffe: 128 Bytes.
- Anzahl der Wiederholungen von lesenden und schreibenden Datenübertragungen: 1.
- Adressoffset wird wieder zu 0x00000 gesetzt.

4.3.2 Bestimmung des Datendurchsatzes

Die PC-Anwendung errechnet den erzielten Durchsatz unter der Zuhilfenahme von beiden „Performanceregistern“: DMAWRP und DMARDP. Die Berechnung erfolgt entsprechend der Formel:

$$B = \frac{(TransferSize + Overhead) * NumberOfTransfers * 8 * 1000}{Latency + Performance * CycleDuration}$$

Wobei gilt:

- B: erzielter Durchsatz in Mbit/s.
- TransferSize: die eingestellte Datenmenge in Bytes, die übertragen werden soll.
- Overhead: header overhead bei den Transaction Layer Paketen: sind laut Datenblatt XAPP859 (Seite 48) 16 Bytes.
- NumberOfTransfers: die vom Benutzer eingestellte Anzahl an Wiederholungen, wie oft die Daten übertragen werden sollen.
- Latency: Latenzzeit ist die Zeit, die vergeht, bis das erste Paket nach dem Start der Übertragung gesendet wird. Dabei ist die Latenz je nach Übertragungsrichtung unterschiedlich. Für den lesenden Speicherdirektzugriff beträgt diese 2000 ns und für den schreibenden Speicherzugriff 400 ns. Diese Werte wurden dem Datenblatt entnommen und wurden durch die Messungen bestätigt.
- Performance: Anzahl der durchgeführten Transaktionen. Dieser Wert wird aus den Registern der PCIe Karte ausgelesen.

	Vers. 1	Vers. 2
DMA Read: 512	103 Mbit/s	102 Mbit/s
DMA Read: 1K	89 Mbit/s	87 Mbit/s
DMA Write: 2K	401 Mbit/s	401 Mbit/s
DMA Read: 2K	237 Mbit/s	239 Mbit/s
DMA FULL DUPLEX: 1K		
Read	125 Mbit/s	125 Mbit/s
Write	342 Mbit/s	344 Mbit/s

Tabelle 4.2: Erzielte Bandbreite

- CycleDuration: dieser Parameter stellt die Dauer zwischen den einzelnen Transaktionen dar. Sie beträgt laut Datenblatt 32 ns und wurde ebenfalls durch Messungen bestätigt.

Die Anzeige der erzielten Bandbreite ist für die Übertragungen mit kleineren Datenmengen nicht sehr genau. Die dargestellte Bandbreite entspricht eher dem Durchsatz innerhalb des FPGAs. Die Durchsatzmessungen beziehen sich auf den Punkt zwischen der FPGA Anwendung und dem PCIe-Kern. Es wurde keine Messungen auf dem PCIe Bus durchgeführt. Die Komponenten an dieser Stelle verwenden FIFOs an ihren Schnittstellen, deswegen können insbesondere kleine Datenmengen in FIFOs schneller reingeschrieben werden, als diese tatsächlich später über den Bus übertragen werden. Bei Übertragungen mit größeren Datenmengen nähert sich die errechnete Bandbreite der theoretisch möglichen an.

In der Tabelle 4.2 wurden durch die Versuche erzielten Bandbreiten zusammengefasst. Es wurden beispielhaft für die unterschiedlichen Datenmengen jeweils 2 Versuchen durchgeführt, bei denen lesende, schreibende und vollduplexe Speicherdirektzugriffe erfolgten.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Im Rahmen dieser Studienarbeit wurde ein Linux-Treiber für die FPGA-Entwicklungsplattform implementiert. Die zur seiner Implementierung besonders wichtigen Informationen über das Linux Betriebssystem und die PCIe-Schnittstelle wurden zudem näher erläutert. Neben dem entwickelten Treiber wurde zudem auch eine PC-Anwendung implementiert, die zu Demonstrations-, Verifikations- und Vermessungszwecken dient.

Eine besondere Eigenschaft des Treibers ist, dass dieser zur Laufzeit des Betriebssystems in den Kernel eingebunden werden kann. Hierdurch wird eine neue Übersetzung des Kernels vermieden, wodurch die Benutzung der FPGA-Entwicklungsplattform deutlich erleichtert wird. Die Möglichkeit des erneuten Ladens des Treibers ist besonders während der Entwicklungsphase des Treibers oder der logischen Schaltung im FPGA vorteilhaft, da auf einen Systemneustart verzichtet werden kann. Weil ein Neustart des Systems zeitaufwendig ist, lässt sich durch das Entladen und erneute Laden des Treibermoduls eine erhebliche Zeitersparnis erzielen.

Den Anwendungen stellt der Treiber weiterhin die gesamte Funktionalität der Hardware bereit, wobei er sparsam mit den Systemressourcen umgeht. Mit Hilfe der PC-Anwendung wurde demonstriert, dass die vom Treiber zur Verfügung gestellten Funktionen fehlerfrei eingesetzt werden können. Während der Evaluierung wurde auch die Übertragungsbandbreite der PCIe-Schnittstelle im Zusammenhang mit dem Treiber ermittelt. Die Ergebnisse zeigen, dass der Treiber den Datendurchsatz über die PCIe-Schnittstelle nicht verringert, wodurch eine effiziente Implementierung des Treibers bestätigt wurde.

5.2 Ausblick

Der in dieser Arbeit entwickelte Treiber unterstützt optimal die FPGA-Entwicklungsplattform. Für den Fall, dass mehrere identische Endgeräte zur Berechnung im PC eingesetzt werden sollen, muss der Treiber um die Unterstützung mehrerer Endgeräte erweitert werden. Diese Funktionalität wurde noch nicht berücksichtigt.

Zur Steigerung der Übertragungsbandbreite kann zudem die PCIe-Schnittstelle um zusätzliche Lanes erweitert werden, wodurch sich die zur Verfügung stehende Datenrate vervielfachen lässt. Da die eingesetzte FPGA-Entwicklungsplattform allerdings eine PCIe-Schnittstelle mit nur einem x1 Link besitzt, erfordert die Schnittstellenerweiterung einen Neubau der Hardware. Eine Erweiterung der Schnittstelle ist jedoch bereits im Treiber vorgesehen.

Literaturverzeichnis

[1] Allen, Taflove ; C. Hagness, Susan: *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House Inc, 2005

[2] Bovet, Daniel P. ; Cesati, Marco: *Understanding the Linux Kernel*. O'Reilly Media, 2000

[3] Budruk, Ravi ; Anderson, Don ; Shanley, Tom: *PCI Express System Architecture*. Boston : Addison-Wesley, 2008

[4] Corbet, Jonathan ; Rubini, Alessandro ; Kroah-Hartman, Greg: *Linux device drivers, 3 ed.* Beijing ; Köln [u.a.] : O'Reilly, 2005

[5] Love, Robert: *Linux-Kernel-Handbuch: Leitfaden zu Design und Implementierung von Kernel 2.6*. München : Addison-Wesley, 2005

[6] Quade, Jürgen ; Kunst, Eva-Katharina: *Linux-Treiber entwickeln*. Heidelberg : dpunkt Verlag, 2006

[7] Wilen, Adam H. ; Schade, Justin P. ; Thornburg, Ron: *Introduction to PCI Express*. Hillsboro : Intel Press, 2003

[8] XILINX (Hrsg.): *Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs: DDR2 SDRAM DMA Initiator Demonstration Platform*. 1. : XILINX, July 2008. http://www.xilinx.com/support/documentation/application_notes/xapp859.pdf. – XAPP859 (v1.1)

[9] XILINX (Hrsg.): *Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express® Solutions*. 1. : XILINX, September 2010. http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf. – XAPP1052 (v1.0)

Abbildungsverzeichnis

1.1 Blockdiagramm des Systems 1

2.1 Mikrokernel Betriebssysteme 8
2.2 Hybridkernel Betriebssysteme 9
2.3 Betriebssysteme mit dem monolithischen Kernel 10
2.4 Linux Betriebssystem 11

3.1 Übertragungsschichten der PCIe-Geräte 27
3.2 Struktur eines TLP-Paketes 29
3.3 Erweiterung des TLPs um die zusätzlichen Informationen in der *Data Link* Schicht 29
3.4 Aufteilung eines Pakets auf dem x4 Link. 34

4.1 Grafische Benutzeroberfläche 54

Tabellenverzeichnis

3.1 Datenrate PCI-Express 24

4.1 Bits des Steuerungs- und Statusregisters 40
4.2 Erzielte Bandbreite 58

Akronyme

ACK	von engl. acknowledgment = Bestätigung	30, 31
AGP	Accelerated Graphics Port	21–23
API	Application Programming Interface	5
BCPL	Basic Combined Programing Language	5
BSD	Berkeley Software Distribution	5, 10
CPU	Central Processing Unit	2, 7, 14, 21–24
CRC	Cyclic Redundancy Check	31
DLLP	Data Link Layer Packet	30, 31
DMA	Speicherdirektzugriff, engl. Direct Memory Access	2, 37
ECRC	end-to-end CRC	28, 29, 31
FPGA	Field Programmable Gate Array	37
GNU	GNU's Not Unix	6, 8
GPL	General Public License	6
ISA	Industry Standard Architecture	21, 22
LCRC	Link CRC	29–31
MS-DOS	Microsoft Disk Operating System	10
NACK	von engl. negative acknowledgement = negative Bestätigung	30, 31
PC	Personal Computer	1, 2, 6, 19
PCB	Process Control Block	13, 14
PCI	Peripheral Component Interconnect	17, 18, 21–24, 26, 32, 41
PCIe	Peripheral Component Interconnect Express	1, 2, 18, 21–26, 28, 29, 31, 33, 34, 37, 41
SCSI	Small Computer System Interface	17
TLP	Transaction Layer Packet	28–31
USB	Universal Serial Bus	17, 19
VESA	Video Electronics Standards Association	21, 22

Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Declaration

I assure that this work was completed by myself independently, and that I only used the stated resources.

18. Februar 2011, Stuttgart ——————