Humboldt University Berlin
Computer Science Department
Systems Architecture Group

Rudower Chaussee 25
D-12489 Berlin-Adlershof
Germany

Phone: +49 30 2093-3400
Fax: +40 30 2093-3112
http://sar.informatik.hu-berlin.de

Linux-Hardwaretreiber für die HHI CinCard-Familie

HU Berlin Public Report
SAR-PR-2005-01

May 2005

Author:
Robert Sperling

Linux-Hardwaretreiber für die HHI CinCard-Familie

Robert Sperling

Zusammenfassung. Die Kinotechnik ist heute mit Ausnahme von Computer-Animationen und einigen Spezialeffekten in der Postproduktion noch immer eine Welt des 35mm Films, während bei den Audio-, Video und Internetmedien die vollständige Digitalisierung bereits Realität ist. Im Rahmen des vom BMWA geförderten Projekts ”Technologien & Systeme für das digitale Kino“ (TSDK) haben sich die Projektpartner verpflichtet, als Fernziel eine vollständige, digitale Verarbeitungskette für das Kino, beginnend bei der Kamera über Produktion, Postproduktion, Vertrieb bis zur Wiedergabe, zu realisieren. Schlüsselkomponenten des Electronic Cinemas sind dabei Kameras und Projektoren mit einer Auflösung, die über die von heutigen HDTV-Systemen hinausgeht. In den entsprechenden Standardisierungsgremien werden Auflösungen von bis zu 4096 Bildpunkten pro Zeile mit mindestens 36 Bit pro Bildpunkt (je 12 Bit für RGB) bei einem Bildseitenverhältnis von 2:1 diskutiert. Somit entstehen bei der Aufnahme und Wiedergabe extrem hohe Datenraten von bis zu 20 Gbit/s. Das Heinrich-Hertz-Institut (HHI) befasst sich im Rahmen dieses Konsortiums vornehmlich mit Projektionstechniken. Um auch in mittelbarer Zukunft diese Kinoformate projizieren zu können, wird hier der Ansatz der Multi-Projektion verfolgt, das heißt, viele kleine kostengünstige Projektoren sollen virtuell einen kinogerechten Hochleistungsprojektor ersetzen. Für dieses Projekt wurde die HHI HyperCard entwickelt. Mit der CineCard wird diese Entwicklung konsequent weitergeführt. Ziel dieser Arbeit ist die Implementierung einer geeigneten Treiberbasis, die den (auch gleichzeitigen) Einsatz der HHI HyperCard II und der neuen HHI CineCard ermöglicht und eine einfache Erweiterungsmöglichkeit auf nachfolgende Kartengeneration bietet.

Inhaltsverzeichnis

Zusammenfassung

Im Rahmen des vom BMWA geförderten Projekts ”Technologien & Systeme für das digitale Kino“ (TSDK) haben sich die Projektpartner verpflichtet, als Fernziel eine vollständige, digitale Verarbeitungskette für das Kino zu realisieren. Schlüsselkomponenten des Electronic Cinemas sind dabei Kameras und Projektoren mit einer Auflösung, die über die von heutigen HDTV-Systemen hinausgeht. Es entstehen bei der Aufnahme und Wiedergabe extrem hohe Datenraten von bis zu 20 Gbit/s. Um auch in mittelbarer Zukunft diese Kinoformate projizieren zu können, wird hier der Ansatz der Multi-Projektion verfolgt, das heißt, viele kleine kostengünstige Projektoren sollen virtuell einen kinogerechten Hochleistungsprojektor ersetzen. Für dieses Projekt wurde die HHI HyperCard entwickelt. Ziel dieser Arbeit ist die Implementierung einer geeigneten Treiberbasis, die den (auch gleichzeitigen) Einsatz der HHI HyperCard II und der neuen HHI CineCard ermöglicht und eine einfache Erweiterungsmöglichkeit auf nachfolgende Kartengeneration bietet.

Kapitel 1

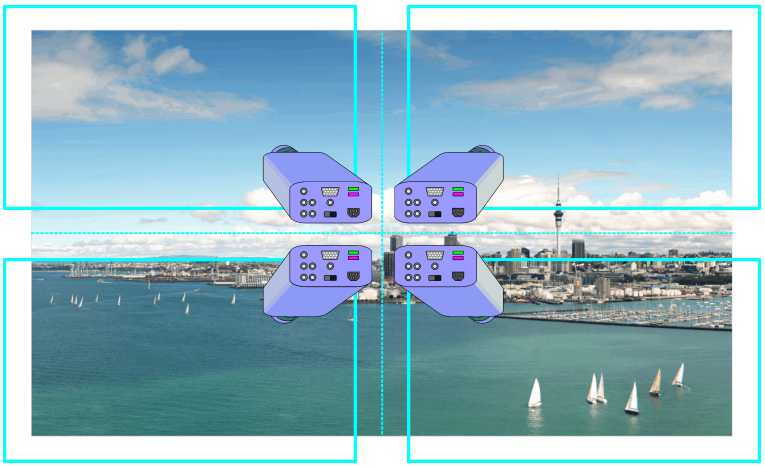
Einführung

1.1 Projektübersicht

1.1.1 Was ist Multiprojektion?

Das Ziel des Projektes CineCard ist die Schaffung einer geeigneten Infrastruktur zur flexiblen Projektion digitaler Daten auf sehr große Bildflächen. Um mindestens die in der bisherigen Kinotechnik gängige Qualität auf großen Leinwänden zu erreichen, sind projizierte Auflösungen nötig, die das Darstellungsvermögen marktüblicher Projektoren bei weitem überschreiten[1]. Die nötigen hohen Auflösungen erfordern einen höheren technischen Aufwand, der sich in erster Linie im Preis der verwendeten Komponenten niederschlägt. Speziallösungen einiger Projektoren-Hersteller disqualifizieren sich zudem oft durch mangelnde Flexibilität.

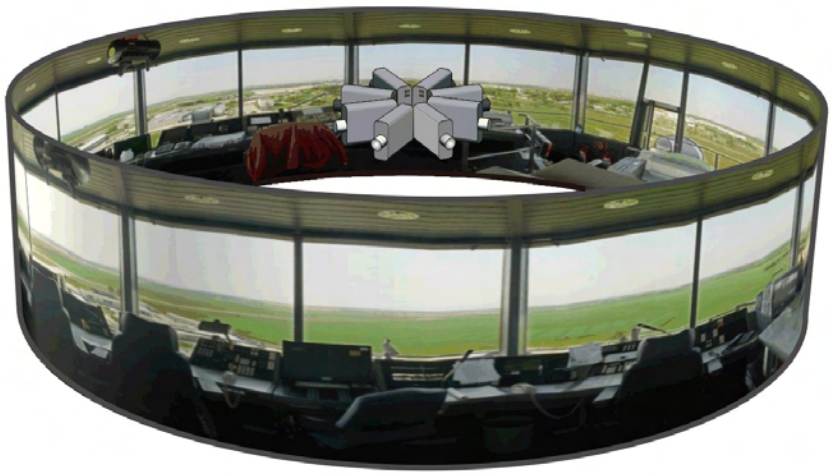
Abbildung 1.1: Anwendungsbeispiel Multiprojektion



[1] Im aktuellen Projekt der Abteilung Bildsignalverarbeitung im Heinrich-Hertz Institut wird eine Rotundenprojektion mit 8 Projektoren und einer Gesamtauflösung von 8000×1400 Bildpunkten angestrebt (Siehe Abbildung 1.2). In einer späteren Ausbaustufe soll mit 20 Projektoren eine Auflösung von 11776×2048 Bildpunkten erreicht werden. Diese Auflösungen sind mittelfristig mit keinem einzelnen handelsüblichen Projektor realisierbar.

Die Multiprojektion dagegen, ermöglicht den Einsatz handelsüblicher, preiswerter LCD- oder DLP-Projektoren aus dem PC-Bereich, die durch beliebige Anordnung eine größere Flexibilität bei der Wahl des projizierten Bildseitenformates erlauben. Auch die nichtplanare Projektion (zum Beispiel in einer Kuppel oder in einer Rotunde) wird durch diesen Ansatz gegenüber der Projektion mit nur einem Projektor entscheidend vereinfacht. Andere Anwendungen, wie zum Beispiel die 3D-Projektion mit Polarisationslinsen oder Shutterbrillen werden erst durch die Multiprojektion möglich. Neben der reinen Auflösungsverbesserung bietet die Multiprojektion auch die Möglichkeit, mehr als eine Datenquelle zu verwenden. Bei den in der Kinotechnik üblichen hohen Datenraten, sind herkömmliche PC-Systeme schon bei wenigen Projektoren der limitierende Faktor. Durch den Einsatz von mehreren PC-Systemen kann die, über die Projektoren verteilte, Gesamtdatenrate der Videodaten erheblich gesteigert werden.

Abbildung 1.2: Projektion in einer Rotunde



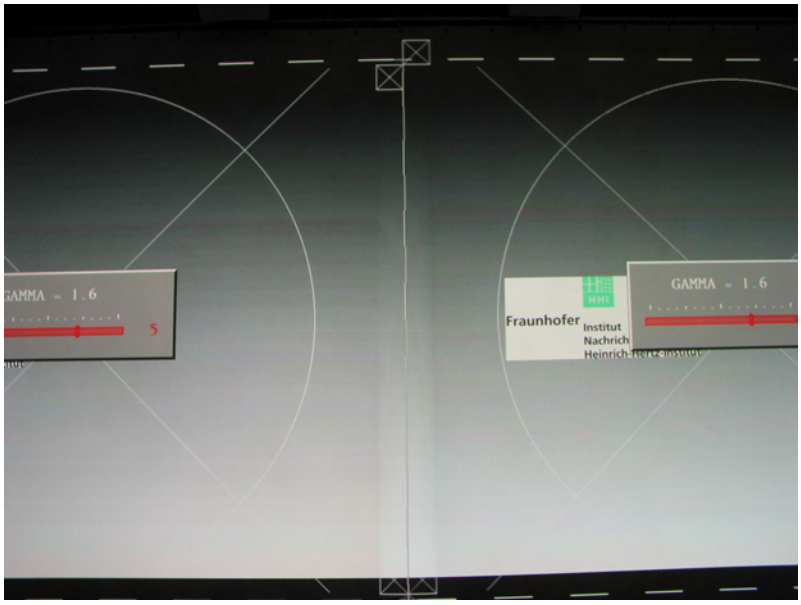
1.1.2 Probleme der Multiprojektion

Die Verwendung von mehreren Projektoren und Quellen bringt jedoch auch zwei wichtige technische Herausforderungen mit sich:

1. Die nahtlose, unverzerrte und farbstabile Projektion und
2. die mindestens bildgenau synchronisierte Ausgabe verschiedener (MPEG2 komprimierter) Bildquellen.

Preiswerte Projektoren besitzen oft nur unzureichende Kalibrierungsmöglichkeiten hinsichtlich der geometrischen Verzerrung. Ein nahtloses, nicht überlappendes Aneinanderkacheln ist deshalb unmöglich. LCD-Projektoren projizieren relativ viel Restlicht, womit überlappende (schwarze) Randbereiche als graue Zonen wahrnehmbar sind. Fertigungstoleranzen bei den Leuchtmitteln erzeugen deutlich sichtbare Farbunterschiede der Projektoren. Dieser Effekt wird durch den Einsatz unterschiedlich alter Leuchtmittel noch verstärkt. Die von einigen (hochpreisigen) Projektoren angebotenen Überblendungsmöglichkeiten in überlappenden Bereichen erfüllen nicht die hohen Qualitätsansprüche der digitalen Kinoprojektion.

Abbildung 1.3: Probleme bei der Überblendung



Eine softwareseitige Anpassung der Helligkeit und die Erzeugung der notwendigen pixelgenauen, weichen Überblendung gestaltet sich bei dem verwendeten MPEG2-kodierten Bildmaterial als sehr schwierig, weil eine solche Bildmanipulation vor der Dekodierung ein erneutes Enkodieren der Videodaten nötig macht. Die durch den hohen Qualitätsanspruch bedingten hohen Datenraten machen darüber hinaus den Einsatz vieler autarker Wiedergabesysteme sinnvoll. Damit aus den verschiedenen Einzelbildern ein großes Gesamtbild entstehen kann, muss die Ausgabe der Einzelbilder auch über verschiedene Rechner hinweg synchronisiert werden. Die dafür notwendige, mindestens bildgenaue Synchronisation beliebig vieler MPEG2-Datenströme ist, ohne einen für alle Projektoren gemeinsam genutzten Bildspeicher, keine triviale Aufgabe.

1.1.3 Die HHI CineCard Familie

Die HHI HyperCard II ist das älteste Mitglied der CineCard-Familie. Sie ist in der Lage Videodatenströme aufzuzeichnen und kann mit einem Tuner für DVB-S (bzw. DVB-T oder DVB-C) bestückt werden. Die Videodaten können über einen analogen Ausgang auf dem HyperDAC auf einem Projektor mit einer Farbtiefe von 36 Bit wiedergegeben werden. Die zum Patent angemeldete Synchronisationseinheit der HyperCard II ermöglicht es, durch ständige Überwachung des Phasenversatzes der Datenströme, auf beliebig vielen HyperCards synchron MPEG-2 Live-Übertragungen und aufgezeichnete Videodaten wiederzugeben. Mit der im HyperDAC in Echtzeit berechneten pixelgenauen Überblendung und einer fein justierbaren Farbanpassung ermöglicht es die HyperCard II, sehr flexibel beliebige Bildseitenformate zu projizieren.
Die HHI CineCard ist eine Weiterentwicklung der HyperCard II, die bis zu fünf voll synchronisierte DVI-D Projektorausgänge besitzt. Mit ihr können Video-Datenströme ausschließlich (synchronisiert) wiedergeben, jedoch nicht aufgezeichnen werden. Die CineCard Professional beherrscht

neben der Berechnung der vertikalen und horizontalen Überblendungen, zusätzlich auch eine Objektiv- und Geometrieentzerrung in Echtzeit und das (digitale) Drehen des Bildes um 90° (separat für jeden Ausgang), um noch mehr Flexibilität bei der Anordnung der Projektoren zu bieten. Mit Hilfe einer externen Digitalkamera zur Aufnahme und Auswertung der Gesamtprojektion wird durch den neuen (semi-)automatischen Helligkeits- und Farbabgleich eine nahtlose, unverzerrte und farbstabile Projektion garantiert.

1.2 Die Studienarbeit

1.2.1 Motivation für die Studienarbeit

Der praktische Einsatz der HyperCard II in verschiedenen Hardwareversionen zeigte einige Schwachstellen im Design des alten, bisher verwendeten Treibers für den Linux-Kernel 2.4 auf. Unterschiedliche Hardwareversionen der HyperCard II erforderten an vielen Stellen im Quelltext neue `if/else`-Abfragen. Strukturelle Änderungen, wie sie für den Betrieb der CineCard notwendig wären, sind mit der vorhandenen Treiberarchitektur nur mit erheblichem Aufwand zu realisieren. Der angestrebte gleichzeitige Betrieb unterschiedlicher Versionen der HyperCard II und der CineCard in einem System verlangte daher einen Neuentwurf des Treibers. Zusätzlich war das Erscheinen der neuen Linux-Kernelgeneration 2.6 und die damit notwendige Portierung des Treibers ausschlaggebend für die Entscheidung, diese Studienarbeit zu schreiben. Neben einer besseren Unterstützung moderner Hardware wie Serial-ATA und Mehrprozessorsysteme, versprach die mit der 2.6er Linux-Generation eingeführte Kernel-Preemption[2] eine erhebliche Verbesserung der Geschwindigkeit und des Antwortverhaltens des Treibers unter hoher Systemlast.

1.2.2 Anmerkungen zu Fachbegriffen und Notation

Einige englische Fachbegriffe haben sich im Laufe der Entwicklung des Linux-Kernels gegen alle Übersetzungsversuche in andere Sprachen erfolgreich durchgesetzt. Sowohl in der Fachliteratur als auch in den Internetforen der Kernelentwickler und anderen Dokumentationen werden bis auf wenige Ausnahmen die englischen Bezeichnungen und Akronyme verwendet. Diese Studienarbeit wird diesem Beispiel weitestgehend folgen. Wo es nötig ist, werden die verwendeten Fachbegriffe kurz erklärt. Treiber und Treiberkomponenten in Form von ladbaren Modulen, werden hier als Modultreiber und fest in den Kernel eingebundene Treiber analog als Kerneltreiber bezeichnet. In den verwendeten Quelltext-Ausschnitten werde ich zugunsten besserer Lesbarkeit und Verständlichkeit auf effizienzsteigernde Konstrukte wie `goto`-Recovery und `likely` / `unlikely` verzichten. Im gesamten Kernelumfeld sowie in den Quelltexten der Modul- und Kerneltreiber ist die Kommentarsprache Englisch. Aus diesem Grunde werden die Beispiele dieser Studienarbeit ebenfalls in englischer Sprache kommentiert.

1.2.3 Struktur der Studienarbeit

Der erste Teil der Studienarbeit beschäftigt sich dem Entwurf und der Struktur des Treibers. Ausgehend von den Voraussetzungen und den gegebenen Entwurftsparametern werden hier der Aufbau des Treibersystems und die Aufgaben der einzelnen Treiberkomponenten besprochen.
„Der Kernel und seine Hardwaretreiber“ widmet sich der allgemeinen Kernelprogrammierung. Hier werden die Neuerungen, die die 2.6er Kernelgeneration mit sich bringt, näher erläutert und einige der Schnittstellen von Linux-Treibern mit der geräteunabhängigen Systemsoftware[3] und dem Userspace erklärt.
Im Abschnitt „Implementation - vom Userspace bis zur Hardware“ werden die (programmiertechnischen) Besonderheiten der HyperCard II näher beleuchten. Dabei wird auf die Schnittstelle der Treiber zur Hardware und auf die Aspekte der hardwarenahen Programmierung eingegangen.

[2] Kernel-Preemption bezeichnet die Technik, auf Kernelebene Prozesse zu Gunsten höher priorisierter zu unterbrechen. Langsame E/A-Operationen oder Gerätefehler blockieren damit nicht länger das gesamte System.

[3] Als geräteunabhängige Systemsoftware wird der von den Gerätetreibern unabhängige Teil des Betriebssystemkernels bezeichnet. Sie enthält zum Beispiel die Prozessverwaltung, die Schnittstellen zum Userspace und das Speichermanagement.

Kapitel 2

Der Treiber - Entwurf & Struktur

2.1 Basis und Voraussetzungen

2.1.1 Linux als Betriebssystembasis

Vorgängerprojekte der Abteilung Bildsignalverarbeitung des HHI verwendeten MS-DOS als Basis, das einen sehr einfachen Zugriff auf die Hardware des Computers ermöglicht. Der Wechsel zu Linux als Betriebssystembasis hat, neben lizenzrechtlichen, überwiegend technische Gründe:

- gute Multitaskingunterstützung,
- einfache und sichere Fernadministration,
- gute Unterstützung moderner Hardware,
- offenes Programminterface,
- hohe Performance unter Last,
- parametrisierbare IO-Scheduler,
- leichte Portierbarkeit auf andere Hardware-Plattformen (PowerPC, Ultrasparc) und
- sehr gute SMP/64-Bit Unterstützung (z.B. auf AMD-64 oder Sparc64 Basis).

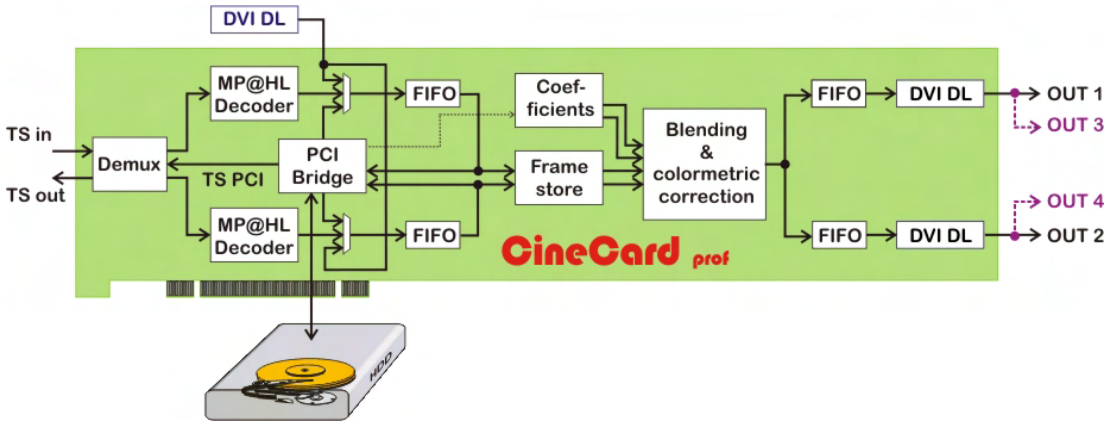
2.1.2 Die Hardware aus Sicht des Treibers

Allen HHI CineCard-Karten gemein ist ein lokaler Bus, dessen Adressbereich über eine PCI-Brücke in den Hauptspeicher des Computers einblendbar ist. Über diesen Adressbereich können die verschiedenen Register der Subsysteme der PCI-Karte angesprochen werden. Für die Aufnahme und Wiedergabe von Videodaten steht ein bis zu zwei Megabit großes Daten-FIFO zur Verfügung. Neben den großen Gemeinsamkeiten der PCI-Karten unterscheiden sich diese vor allem durch:

- Unterschiede in den Basisadressen der Subsysteme,
- Art und Management des Video-FIFOs, speziell in der Interrupt-Behandlung und
- die Anzahl von MPEG2 Dekodereinheiten.

Zur Kommunikation mit dem PC besitzen die HHI HyperCard II und die HHI CineCard eine PLX-9054 PCI-Brücke, die jedoch möglicherweise in einer späteren Version durch eine PCI-Express-Lösung ersetzt werden könnte. Die PCI-Brücke stellt zur Konfiguration, zur Interrupt-Behandlung und zum Auslösen von DMA-Transfers einen 256 Byte großen Speicherbereich mit Steuerregistern zur Verfügung, der ebenfalls in den Hauptspeicher des Computers eingeblendet werden kann.

Abbildung 2.1: Hardwareaufbau der CineCard Professional



2.1.3 Erwartete Schnittstelle zum Userspace

Um eine möglichst hohe Kompatibilität zur bestehenden Test- und Applikationsumgebung zu gewährleisten, lehnt sich die Schnittstelle zum Userspace an die des alten HyperCard-Treibers an. Es werden pro PCI-Karte zwei Gerätedateien zur Verfügung gestellt:

- `/dev/hhi-cinecard/control[0..8]`
  zur Steuerung und Konfiguration der Karte per I/O-Controls und
- `/dev/hhi-cinecard/dvr[0..8]`
  zur Aufnahme und Wiedergabe von Videodatenströmen.

Die `control`-Gerätedateien müssen von mehreren Applikation gleichzeitig zu öffnen sein, um z.B. die Fehlersuche bei der Applikationsentwicklung zu erleichtern.

2.2 Struktur des Treibers

Ein wichtiger Parameter für den Entwurf des Treibers war eine hohe Modularität. Das Austauschen zentraler Komponenten der Hardware (MPEG2 Dekoder bzw. PCI-Brücke) und der Software (Applikation bzw. Treibersubsysteme) soll auch in Zukunft jederzeit möglich sein. Allen (auch zukünftigen) Hardwaretypen gemein ist der lokale Bus, dessen Adressbereich in den Hauptspeicher des Computer eingeblendet werden kann. Größere Datenmengen können durch die DMA-Einheit in und aus dem FIFO des (De)-Multiplexers und in die Blend-Koeffizienteneinheit der Karten übertragen werden. Der Einsatz nicht DMA-fähiger PCI/PCI-Express Brücken ist

nicht geplant (kann aber auch nicht ausgeschlossen werden). Für die Struktur des Treibersystems ergeben sich damit drei mögliche Vorgehensweisen:

1. Ein unifizierter Treiber steuert die unterschiedlichen Kartentypen an und bestehende Unterschiede werden an den betreffenden Stellen durch `if`- oder `case`-Anweisungen behandelt. Dieser Ansatz erfordert viele zusätzliche Abfragen und macht damit den Quelltext des Treibers unübersichtlich und behindert die Fehlersuche. Zusätzlich wird das Laufzeitverhalten durch die vielen `if`- und `case`-Anweisungen beeinträchtigt. Größere strukturelle Änderungen, wie der Austausch der PCI-Brücke wären ausgeschlossen.

2. Mehrere getrennte Treiber mit einer speziellen Anpassung auf den jeweiligen Kartentyp arbeiten unabhängig voneinander. Für diese Lösung besitzen die Treiber jedoch eine zu große gemeinsame Codebasis - werden gleiche Teile mehrfach implementiert, führt dies schnell zu Fehlern und Inkonsistenzen. Auch die Aufteilung in mehrere Untermodule, die für die verschiedenen Treiber angepasst zusammengelinkt werden, ist unzweckmäßig. Bei der gleichzeitigen Verwendung unterschiedlicher Kartentypen in einem System müssten gleiche Programmteile mehrmals im Speicher gehalten werden. Außerdem wird durch diesen Ansatz eine Kommunikation zwischen den Treibern (unterschiedlichen Typs) und die konsistente Verwaltung der Gerätedateien unnötig erschwert.

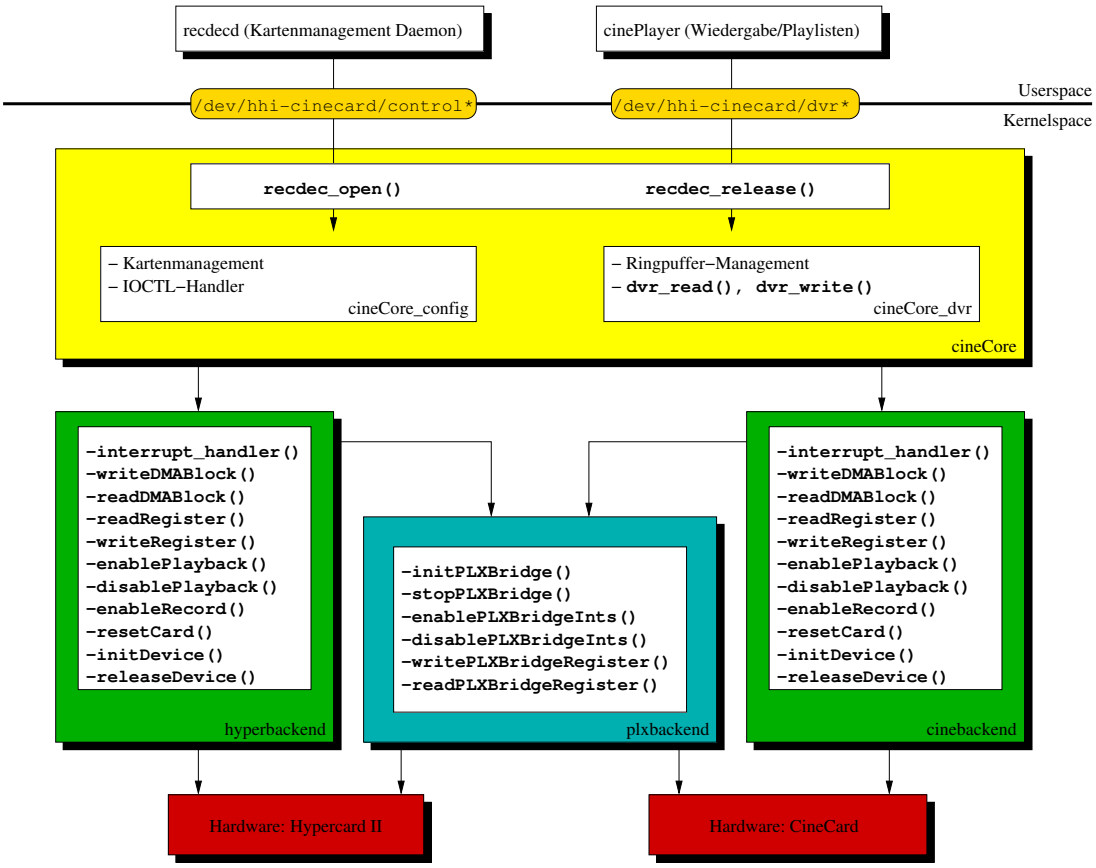
3. Ein modularer Treiber fasst in einem zentralen Kern die Gemeinsamkeiten der PCI-Karten und Verwaltung der Geräte zusammen und stellt damit eine einheitliche Schnittstelle zum Userspace zur Verfügung. Unterschiede in der Hardware werden, wo es sinnvoll und möglich ist, in Backends gekapselt. In Hinblick auf die ungewisse Zukunft der Entwicklung der CineCard-Familie ist dieser Ansatz der zweckmäßigste.

Ein modularer Aufbau des Treibers kann die Trennung von kartenspezifischen Programmteilen mit der gleichzeitigen, gemeinsamen Nutzung gleichen Codes kombinieren. Mit dieser Lösung bleiben sowohl der Treiberkern als auch seine Backends im Quelltext übersichtlich, gut zu warten und eine Mehrfachimplementierung von gleichen Programmteilen wird ausgeschlossen. Damit ist die Portierung des Treibers auf spätere Hardwareversionen mit wenig Aufwand möglich. Zusätzlich sinkt der Speicherplatzbedarf zur Unterstützung mehrerer Kartentypen in einem System auf ein Minimum und eine kartenübergreifende Kommunikation ist möglich.
Unterschiedliche Ausstattungen innerhalb eines Kartentyps wie zum Beispiel bei der HyperCard II, die mit und ohne integriertes Video-ISP existiert, werden in den jeweiligen Backends modelliert.

2.2.1 CineCore - Zentrale Verwaltungsinstanz

Für alle Kartentypen notwendigen Funktionen werden im zentralen Kern, dem so genannten CineCore, zusammengefasst. Der CineCore bildet gegenüber den Applikationen im Userspace ein einheitliches Frontend zu den eigentlichen Hardwaretreibern.
Die Backends für die verschiedenen Kartentypen registrieren die erkannten PCI-Karten im CineCore. Je nach Ausstattung der Karten werden hier die notwendigen Softwarestrukturen initialisiert. Im CineCore wird zusätzlich der zentrale Ringpuffer für die Aufnahme und Wiedergabe von Datenströmen verwaltet.
Nur die Aufteilung in eine solche Frontend/Backend-Struktur ermöglicht eine konsistente und leicht zu wartende Schnittstelle zum Userspace.

Abbildung 2.2: Treiberstruktur



2.2.2 Backends für die Kartentypen

Die so genannten Backends fassen alle zur Kommunikation mit der Hardware notwendigen Funktionen zusammen. In einem Backend befinden sich die Funktionen für Registerzugriffe und DMA-Transfers sowie der lokale Interrupt-Handler und die Hardware-Initialisierungsroutine.

2.2.3 Backend für die PLX-Brücke

Das PLX-Backend beinhaltet grundlegende Funktionen zur Ansteuerung der *PCI 9054*, einer PCI-Brücke von PLX-Technology. Die überschaubare Menge von (kurzen) Funktionen, die zur direkten Kommunikation mit der PCI-Brücke notwendig sind, rechtfertigen kein eigenständiges Modul. Darum werden die in diesem Backend zusammengefassten Funktionen in das übergeordnete Backend für den jeweiligen Kartentyp eingebunden.

Kapitel 3

Der Kernel und seine Hardwaretreiber

3.1 Das Kernelumfeld von Linux

3.1.1 die Build-Umgebung - Makefiles & Co.

Mit der Einführung des Linux Kernels 2.6 wurde das neue Kernel-Build-System vorgestellt. Es beruht auf dem klassischen `make`, erleichtert jedoch zusätzlich das Auflösen von Abhängigkeiten und kann beim Übersetzen der Kernel-Quellen viel Zeit sparen.
Durch intelligenteres Kompilieren müssen nun unveränderte Programmteile nicht mehr neu übersetzt werden, wenn sich nur ein Teil der Kernelkonfiguration geändert hat. Wegen der durch das Build-System separat in die Module eingebundenen Versioninformationen kann auf ein „`make dep clean`“ vor dem Übersetzen der Kernelquellen verzichtet werden[1]. Menüs für Konfigurationseinstellungen und dazu gehörige Hilfetexte werden durch „`Kconfig`“-Dateien direkt vom Programmierer des Modules bereitgestellt. Damit wird der möglichen Inkonsistenz von Konfigurationsoption und Beschreibungstext vorgebeugt, unter der Linux bis zur Version 2.4 litt.
Ein Nachteil des neuen Kernel-Build Systems ist allerdings, dass auch das Übersetzen eines Moduls außerhalb des Kernelquellenverzeichnisses nun Root-Rechte erfordert, weil bei jedem Übersetzungslauf auch Dateien im Verzeichnis der Kernelquellen verändert werden. [2]

Listing 3.1: Beispiel-Makefile für einen Modultreiber

```
1 # ************************
2 # kernel-build-system part
3 # ************************
4
5 # are we called via kernel-build-system?
6 ifneq ($(KERNELRELEASE),)
7
8 # always include "modulName" and "directory1"
9 obj-m   := modulName.o directory1/
```

[1] Durch `make dep clean` wurden bis zum 2.4er Kernel die Abhängigkeiten der Module untereinander neu berechnet und anschließend alle schon übersetzten Objektdateien (mit den alten Abhängigkeiten) gelöscht.

[2] Die Informationen über die Abhängigkeiten der Module und Kernelkomponenten bezüglich der Übersetzung werden vom Kernel-Build-System zentral in den Kernelquellen gespeichert.

```
10
11 # include "directory2" if CONFIG_DEPENDENCY is set
12 obj-$(CONFIG_DEPENCENCY)    += directory2/
13
14 else
15 # ****************************
16 # local part for kernel modules
17 # ****************************
18
19 # get directory of currently running kernel
20 KDIR    := /lib/modules/$(shell uname -r)/build
21
22 # current directory, needed for kernel modules
23 PWD     := $(shell pwd)
24
25 # execute the kernel-build-system
26 default:
27         $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
28
29 endif
```

Das Makefile muss „`Makefile`“ (erster Buchstabe groß) geschrieben werden, weil es durch das Kernel-Build System eingelesen und interpretiert wird. Seit der Kernel-Version 2.6.10 wird empfohlen, die Makefiles „`Kbuild`“ zu nennen, um den Unterschied zu normalen Makefiles für Userspace-Applikationen hervorzuheben. (Jedoch findet sich selbst in den Quellen von Kernel 2.6.10 und 2.6.11 keine einzige `Kbuild`-Datei.)
Der Übersetzungslauf beginnt in Zeile 27. Hier wird in das Verzeichnis der Kernel-Quellen gewechselt und `make` mit der Regel `modules` und dem aktuellen Pfad als Parameter `SUBDIRS` gestartet. Eigene Moduldateien und ganze Unterverzeichnisse werden, wie in Zeile 9 und 12 zu sehen ist, hinzugefügt. Hierbei ist darauf zu achten, dass an dieser Stelle noch nicht die `.ko`-Dateiendung der finalen Kernelmodule verwendet werden darf. Das Kernel-Build System erzeugt die `.ko` Datei erst nach dem Kompilieren durch das Einbinden der Datei `init/vermagic.o`, welche die Kernel-Versionsinformationen enthält.
Die Regel `obj-m` wird ausgeführt, wenn das Kernel-Build-System Module erzeugt. Ihr Pendant zu fest einkompilierten Kerneltreibern lautet `obj-y`. Um eine Treiberoption von der Konfiguration des Kernels abhängig zu machen, wird `obj-$(CONFIG_DEPENDENCY)` verwendet, wobei `DEPENDENCY` für die Art der Abhängigkeit steht. `$(CONFIG_PCI)` kann verwendet werden, um den eigenen Treiber vom PCI-Subsystem[3] des Kernels abhängig zu machen. Neben den Konfigurationsoptionen des Kernels und seiner Subsysteme können auch selbst definierte Schalter für die Übersetzung verwendet werden.
Eigene Menüs für die Kernelkonfiguration können im Verzeichnis des Treibers mit der Datei `Kconfig` erstellt werden. Um diese z.B. in `make menuconfig` verwenden zu können, muss die eigene `Kconfig`-Datei des Treibers in einer übergeordneten `Kconfig`-Datei des Kernels mittels:

```
source "relativer/Pfad/zum/eigenen/Modul/Kconfig"
```

eingebunden werden. Dafür muss sich das Verzeichnis des Treibers jedoch unterhalb des Wurzelverzeichnisses der Kernelquellen befinden. Beim Erstellen von `Kconfig`- und anderen Dateien im Kernelumfeld sollte auf Umlaute und andere nicht im 7-Bit ASCII-Standard vorkommende

[3] Im Kernelkontext bezeichnet Subsystem eine bestimmte Sicht auf die Struktur des Systems. Subsysteme können sich auf z.B. (Hardware-)Geräte, Busse und (Software-)Geräteklassen beziehen.

Zeichen verzichtet werden. Neben eventuellen Schwierigkeiten mit `make` und einigen Compilern ist auch die Verwendung in verschiedenen Editoren mit unterschiedlichen Zeichenkodierungen problematisch, was den Austausch des Quelltextes mit anderen (oft nicht deutschsprachigen) Kernelentwicklern unnötig erschwert.

Listing 3.2: Kconfig-Datei für einen einfachen Linux-Treiber

```
1  # create a new sub-menu
2  menu "Sound"
3
4  # create a new menu-entry with our kernel option
5  config MYDRIVER
6
7  # with tristate [Y]es, [M]odule and [N]o are possible
8  # for [Y]es/[N]o options use the keyword: bool
9  # for integer numbers use: int  and in the next line: default "number"
10         tristate "support for my own driver"
11
12 # sample: driver with PCI-subsystem dependency
13         depends on PCI
14
15 # some description for this configuration-option
16         ---help---
17           This is a small help text for my own module. It is written
18           in english, since the official Linux-kernel language is
19           english, too.
20
21 # inclusion of another Kconfig file in a subdirectory of this driver
22 # do NOT include other files than Kconfig - or your build will fail
23 source "relative/path/to/the/driver/subdirectory/and/its/Kconfig"
24
25 # end of the submenu
26 endmenu
```

3.1.2 Konfiguration des Kernels

Neben den nötigen Kerneleinstellungen, um überhaupt Treiber für PCI-Geräte übersetzen zu können, sind folgende Kerneloptionen für die Fehlersuche von besonderem Interesse:

```
General setup  --->
    [*] Configure standard kernel features (for small systems)  --->
        [*]   Load all symbols for debugging/kksymoops (NEW)
Kernel hacking  --->
    [*] Kernel debugging
        [*]   Debug memory allocations
        [*]   Spinlock debugging
        [*]   kobject debugging
        [*]   Verbose BUG() reporting (adds 70K)
```

Die genannten Punkte werden im Menü der Kernel-Konfiguration ausführlich beschrieben. Ab dem Kernel 2.6.11 steht zusätzlich das `debugFS` zur Verfügung, das es ermöglicht, treiberinterne Informationen zur Fehlersuche in einem virtuellen Dateisystem zur Verfügung zu stellen.

3.2 Wichtige Neuerungen in der 2.6er Kernelserie

Die Entwicklung am Linux-Kernel 2.6 ist trotz seines Erscheinens vor über einem Jahr, noch nicht abgeschlossen. Es werden immer noch zum Teil zentrale Komponenten des Kernels verändert und ausgetauscht. Viele Bücher über die neue Kernelgeneration enthalten deshalb Beispiele, die mit den aktuellen Kernelquellen nicht mehr funktionieren. Die Internetseite Linux Weekly News [6] hat die wichtigsten Änderungen der Kernel-API zusammengefasst und bildet neben der Kernel-Mailingliste [9] die wichtigste Referenz, wenn bestehende Module sich nicht mehr mit der aktuellen Version des Kernels übersetzen lassen.

3.2.1 Kernel-Preemption

In allen vorhergehenden Generationen des Linux-Kernels konnten auf Kernelebene ablaufende Prozesse nicht unterbrochen werden. Dies führte dazu, dass nur wenige Linux-Treiber auf SMP[4]-Festigkeit überprüft und kritische Sektionen gegen gleichzeitigen Zugriff geschützt wurden. Es konnte auch auf SMP-Systemen davon ausgegangen werden, dass an einen Prozessor gebundene Programmteile nicht auf dem selben unterbrochen wurden. Somit wurden nur in sehr wenigen Treibern Semaphoren, Mutexe oder Spinlocks zum Schutz dieser Sektionen verwendet.
Mit der in der aktuellen Linux-Version eingeführten Kernel-Preemption ist es möglich, wartende Prozesse[5] feiner zu priorisieren, damit Anwendungen in zeitkritischen Bereichen schneller auf Benutzerinteraktionen reagieren können. Voraussetzung dafür ist jedoch, dass alle Bestandteile des Kernels unterbrechbar und eventuell reentrant programmiert sein müssen. Nicht unterbrechbare Regionen müssen dabei gesondert geschützt werden. Diesem Konzept folgend, sollten neue Treiber statt des alten `IO-Controls`-Mechanismus, das preemptive Pendant `ioctl_unlocked` verwenden und sich selbst um den Schutz gegen gleichzeitigen Zugriff kümmern. Eine weitere preemptive Methode, `compat_ioctl()` kann verwendet werden, um auf 64-Bit Systemen die Kompatibilität zu 32-Bit Applikationen zu gewährleisten[6].

3.2.2 SysFS, kObject und das neue Geräte- und Treibermodell

Neben der Kernel-Preemption und anderen Verbesserungen, die hauptsächlich das Laufzeitverhalten und die Interaktivität des Kernels unter hoher Systemlast verbessern, wurden im Kernel 2.6 drei wesentliche Komponenten eingeführt:

1. Das *Gerätemodell* beschreibt die Beziehungen der Hardwaretreiber zueinander.
2. Das *vereinheitlichte Treibermodell* definiert eine einheitliche Menge von Informationen, die die Treiber zur Verfügung stellen. Es standardisiert, wie Treiber unterstützte Hardware erkennen und deren Unterstützung bekanntgeben.
3. Die `kObject`-*Stuktur* bildet über das `SysFS` eine Vermittlungsschicht zwischen den Modellen.

[4] Symmetric(al) Multi Processor/Processing

[5] Mit wartenden Prozessen sind sowohl schlafende als auch aktiv wartende (Polling) Prozesse gemeint. Anstehende E/A-Operationen können nicht nur neu geordnet, sondern auch bei längerem Zugriff unterbrochen und später fortgesetzt werden.

Das Gerätemodell wurde in einem sehr frühen Entwicklungsstadium in den Kernel aufgenommen. Bisher existiert noch eine große Anzahl von Treibern, die das Gerätemodell noch nicht (aktiv) unterstützen. Die Kernelentwickler setzen dennoch hohe Erwartungen in diese neue Komponente. Sie ermöglicht eine klare Strukturierung der Gerätetreiber in Bezug auf Bussysteme, Controller und Perepheriegeräte und unterstützt damit die Arbeit des `udev`-Daemons, der für die stabile Zuordnung von Geräten zu Gerätedateien zuständig ist (Siehe Abschnitt 3.2.4). Das Gerätemodell besitzt in großen Teilen eine an den Aufbau der Hardware des Computer angelehnte Struktur und ist damit besser in der Lage sowohl Abhängigkeiten beim Laden und Entladen von Modulen aufzulösen als auch das Energiespar-Management des Computers aktiv zu unterstützen.
Das so genannte System-Dateisystem (`SysFS`) bildet die Schnittstelle des Gerätemodells zum Userspace. Dieses neu eingeführte Dateisystem fasst alle wichtigen Informationen über die Hardware und deren Zustand zusammen und soll langfristig auch das `dev`-Dateisystem[6] und das `/dev/pts`-Verzeichnis (für Pseudo-Terminals nach dem UNIX98-Standard) vollständig ersetzen. Zusätzlich sollen alle nicht zu Prozessen gehörende Informationen, wie zum Beispiel Statusinformationen von Hardwaretreibern und das Verzeichnis `/proc/sys` aus dem `proc`-Dateisystem in das `SysFS` verschoben werden.
Das `SysFS` enthält neben den Informationen über die von geladenen Gerätetreibern erkannten Geräte (Gerätename, Treibername, Bus usw.) auch Geräte-Attribute wie zum Beispiel IRQ, DMA-Ressourcen und Energiestatus.
Linux 2.2 und frühere Versionen besaßen keine Unterstützung für ein vereinheitlichtes Treibermodell. Alles Wissen um unterstützte Hardware, deren Konfigurationsmöglichkeiten, verwendete Ressourcen und deren Anbindung an den Computer lag ausschließlich in den Treibermodulen selbst vor. Um die in modernen Computern verwendeten Bussysteme wie z.B. USB und auch ACPI zu unterstützen, werden mehr Informationen über Treibermodule benötigt, als die Hardware-Ressourcen die die Treiber nutzen. Es ist ebenfalls wichtig zu wissen:

- an welchem Bus das unterstützte Gerät hängt,
- welche (potentiellen) Untergeräte existieren[7],
- wie der aktuelle Stromsparmodus ist,
- ob das Gerät rekonfiguriert werden kann oder muss, um andere Ressourcen zu nutzen,
- und ob sich dessen Systemanbindung und Status dabei[8] ändert.

Zwischen dem `SysFS`, dem Gerätemodell und dem Treiber vermittelt als zentrale Schnittstelle die `kObject`-Struktur. Sie enthält Informationen über die vom Treiber unterstützten Geräte, die unterstützte Schnittstelle zum Userspace und wichtige Details wie Referenzzähler und den Energiemanagementstatus. Durch die `kObject`-Struktur bietet das Gerätemodell eine direkte Unterstützung für hotplug-fähige Geräte wie z.B. PC Cards, USB und Firewire und PCI-Hotplugging. Damit verbunden ist auch eine bessere Unterstützung von modernen Energiesparteчniken wie ACPI.

[6] Alle dafür notwendigen Schnittstellen stellt der Kernel, teils noch in sehr experimenteller Form, schon in der aktuellen Version bereit. Wie genau der Ersatz für das `/dev`-Verzeichnis aussehen wird, und ob ein Ersatz überhaupt zweckmäßig ist, wurde noch nicht abschließend geklärt.

[7] Dabei sind auch Informationen über verwendete Ressourcen von Geräten wichtig, deren Treibermodul noch nicht geladen wurde.

[8] Ein Beispiel dabei könnten Geräte sein, die sowohl USB2 als auch Firewire unterstützen.

3.2.3 Verwaltung von Gerätedateien

Device-Filesystem

Das Device-Filesystem, kurz `DevFS`, wurde mit dem Erscheinen des 2.6er Kernels offiziell als veraltet eingestuft. Der häufigste Kritikpunkt seitens der Kernelentwickler ist die notwendige aktive Unterstützung des `DevFS` durch die Hardwaretreiber. Verzeichnisse und Gerätedateien müssen von den Gerätetreibern via `devfs_mk_dir` und `devfs_mk_bdev` bzw. `devfs_mk_cdev` erzeugt, registriert und beim Entladen der Module wieder entfernt werden.
Damit ergibt sich eine fest im Kernel bzw. den Treibern vorgegebene Anordnung und Benennung der Gerätedateien. Die einzige Möglichkeit des Benutzers, Einfluss auf die Benennung der Gerätedateien zu nehmen, ist das Anlegen von symbolischen Links und das manuelle Erstellen von Gerätedateien. Die persistente Speicherung der durch den Benutzer durchgeführten Änderungen stellt darüber hinaus ein weiteres Problem für die Hersteller von Linux-Distributionen dar. Unterstützt ein Treiber das Device-Filesystem nicht, müssen die Gerätedateien z.B. durch ein Startskript erzeugt werden - ist die Unterstützung bei Verwendung einer neueren Treiberversion vorhanden, kann es zu Namenskollisionen kommen.
Die Zuordnung von Major- und Minornummern zu den Namen der Gerätedateien ist eine relativ komplexe und fehleranfällige Aufgabe und gehört nach der Philosophie der Kernelentwickler in den Userspace. Es hat sich gezeigt, dass bei einer ungünstigen Event-Reihenfolge auch so genannte Race-Conditions[9] auftreten können. Ein weiterer Nachteil des Device-Filesystems ist, dass der Inhalt des `/dev`-Verzeichnisses nur die Gerätedateien der momentan verbundenen Geräte enthält, deren Modulen auch geladen sind.

3.2.4 udev und das Hotplug-System

Die dynamische Verwaltung von Gerätedateien über den `udev`-Daemon versucht viele der Schwachstellen des Device-Filesystems zu beseitigen, indem die zentrale Verwaltung der Gerätedateien wieder zurück in den Userspace verlegt wird. `Udev` verwendet den schon längere Zeit im Kernel vorhandenen Hotplug-Mechanismus, der bei der Registrierung eines Gerätes bzw. Gerätetreibers ein zuvor spezifiziertes Programm im Userspace aufrufen kann. Ein Hotplug-Event, der die Erstellung einer Gerätedatei veranlassen kann, wird ausgelöst, wenn dem Gerätemodell eine `kObject`-Struktur übergeben und dieses mit einer Gerätenummer verknüpft wurde. (Siehe Abschnitt 3.3.2)
Bei der Zuordnung einer zu erzeugenden Gerätedatei zu der im Kernel verwendeten Gerätenummer, wird der `udev`-Daemon von dem durch das `SysFS` bereitgestellten globalen Namensraum unterstützt. Beispielsweise kann die Gerätenummer[10] `3:0` der ersten (IDE-)Festplatte im System (`/dev/hda`), aus der Datei: `/sys/block/hda/dev` ausgelesen werden. Wie der `udev`-Daemon auf Hotplug-Events reagieren soll, wird durch Regeln im Verzeichnis `/etc/udev/rules.d/` festgelegt. Die `udev`-Regeln definieren wie die Gerätedateien benannt werden, ob symbolische Links angelegt und ob Geräte einer Gruppe in einem gemeinsamen Verzeichnis zusammengefasst werden sollen. Eine Regel für die PCI-Karten der CineCard-Familie könnte beispielsweise so aussehen:

```
BUS= "pci", KERNEL="cinecontrol[0-9]*", NAME="hhi-cinecard/control%e"
```

Dieser Mechanismus ist vollständig unabhängig von der Gerätenummer bzw. der Major- und Minornummer, da in zukünftigen Kernelgenerationen, Gerätenummern ausschließlich dynamisch

[9] Race-Condition nennt man eine Situation, in der mehrere Treiberabschnitte konkurrierend auf eine gemeinsam genutzte Ressourcen zugreifen wollen. Damit ist das Ergebnis nach dem Zugriff undefiniert. Dies kann zum Beispiel beim gleichzeitigen Verbinden und Entfernen von zwei Geräten gleichen Typs geschehen.
[10] Im Format: (Erste Nummer der Gerätegruppe:Offset), entsprechend dem alten (Major:Minor)-Schema.

(und eventuell sogar zufällig) zugewiesen werden sollen. Damit lassen sich viele durch die in der Vergangenheit fest vergebenen Major-Nummern entstandenen Probleme vermeiden. Durch den `udev`-Daemon lassen sich, auch über einen Neustart des Systems hinweg, stabil zugeordnete Gerätenamen erzeugen. Die Erzeugung kann sogar an die Seriennummer eines Gerätes geknüpft werden. Die Reihenfolge, in der zum Beispiel USB-Geräte mit dem System verbunden werden, hat damit keinen Einfluss auf die Namenszuordnung der Gerätedateien:

```
BUS= "usb", SYSFS_serial="HP23SAMPLE2349FD", NAME="myInkjetPrinter%e"
BUS= "usb", SYSFS_serial="KY3OTHEROSAMPLE2349", NAME="myLaserPrinter%e"
```

3.3 Implementation eines (sehr einfachen) zeichenorientierten Treibers

Hardwaretreiber im Linux-Kernel bilden eine Vermittlungsinstanz zwischen den Applikationen im Userspace und der Hardware und müssen damit zwei verschiedene Programmierschnittstellen benutzen. Neben der obligatorischen Schnittstelle zum Userspace wird in dieser Studienarbeit exemplarisch die Hardwareschnittstelle des PCI-Subsystems erläutert.
Um das Implementieren von Treibern für andere Geräteklassen zu erleichtern, existieren schon vorgefertigte Treibergerüste in den Kernelquellen. Die Vorlage für USB-Gerätetreiber befindet sich unter:

- `drivers/usb/usb-skeleton.c`.

Netzwerkkartentreiber verwenden nicht die „übliche“ auf Gerätedateien basierende Schnittstelle zum Userspace. Für diesen speziellen Treibertyp bieten die Dateien:

- `drivers/net/pci-skeleton.c` und
- `drivers/net/isa-skeleton.c`

ein kommentiertes Grundgerüst, mit dem sich eigene Treiber relativ einfach implementieren lassen. Dieses Kapitel widmet sich speziell den zeichenorientierten Gerätetreibern, da sie beim Implementieren eigener Projekte am häufigsten Verwendung finden.
Mit der Programmierung von blockorientierten Treibern beschäftigte sich sehr ausführlich ein Artikel der Serie *Kernel-Technik*[4] im Linux-Magazin. Weitere wertvolle Informationen zu blockorientierten Gerätetreibern finden sich in der Kerneldokumentation in der Datei `biodoc.txt`[11].

3.3.1 Modulinitialisierung

Das folgende Beispiel zeigt anhand des unvermeidlichen „Hello World!“, wie sich Modultreiber nach dem Laden via `modprobe` oder `insmod` beim Linux-Kernel anmelden. Das folgende Beispiel ist sowohl als Modultreiber als auch als Kerneltreiber einbindbar:

Listing 3.3: „Hello World“ als Linux-Treiber

```
1  #include <linux/version.h>  /* kernel version this module */
2                              /* was compiled with          */
3  #include <linux/module.h>   /* macros and definitions     */
4                              /* (i.e. MODULE_LICENSE)      */
5
6  /* called on module-initialization */
7  static int __init init_myModule(void)
8  { /* initialization... */
9    printk(KERN_INFO "Hello World!\n");
10
11   return 0; /* modul-loading successful */
12 }/*init_myModule*/
13
14 /* called on module-deinitialization */
15 static void __exit cleanup_myModule(void)
16 { /* cleanup of used ressources */
17   printk(KERN_INFO "Goodbye World!\n");
18 }/*cleanup_MyModule*/
19
20 /* ******** module-description ******** */
21 MODULE_LICENSE("GPL");
22 MODULE_AUTHOR("Robert Sperling");
23 MODULE_DESCRIPTION("Dummy");
24 module_init (init_myModule);     /* register init functions   */
25 module_exit (cleanup_myModule); /* register cleanup function */
```

Jedes Modul für den Linux-Kernel muss mindestens Informationen über die verwendete Lizenz und den Autor bereitstellen. Die Schlüsselwörter `__init` und `__exit` kennzeichnen die Zugehörigkeit der Funktionen zur Initialisierung und Deinitialisierung. Die Kennzeichnung ist nicht zwingend notwendig, hilft jedoch wertvollen Kernelspeicher zu sparen, wenn der Treiber fest in den Kern eingebunden wird.
Die ausschließlich für das Entladen von Modulen und zum Zurücksetzen der Hardware nötige `__exit`- und `__dev_exit`-Funktionen werden bei (fest eingebundenen) Kerneltreibern gar nicht erst übersetzt[11]. Der durch die Initialisierungsroutine (`__dev_init` bzw. `__init`) und deren Daten (`__initdata` bzw. `__devinitdata`)[12] belegte Speicher wird nach dem Booten des Kernels wieder freigegeben. Daher sollte `EXPORT_SYMBOL(Funktionsname)`, das die angegebene Funktion anderen Treibern zur Verfügung stellt, nicht für die mit `__init` oder `__devinit` markierte Funktionen verwendet werden.
Die mit `__dev` beginnenden Schlüsselwörter beziehen sich auf das Hotplug-System des Kernels – damit gekennzeichnete Ressourcen werden nicht aus dem Speicher entfernt, wenn der Kernel mit Hotplug-Funktionalität übersetzt wurde. Die Position der Schlüsselwörter hängt vom syntaktischen Kontext und deren Verwendung ab. Nachfolgend sind einige Beispiele für die Verwendung von Schlüsselwörtern der impliziten Ressourcenverwaltung aufgelistet:

[11] Im Kernel 2.6 ist es möglich, das Entladen von Modulen zu verbieten. In diesem Falle werden die Deinitialisierungsfuktionen und deren Daten auch nicht übersetzt.

[12] Um die Initialisierung von statischen Variablen muss sich der Programmierer selbst kümmern, sie werden nicht mit Nullen vorinitialisiert.

Listing 3.4: Beispiele für die impliziten Ressourcenverwaltung

```
1  static int initVar __initdata;
2  static char someCharacterArray[] __initdata = {0x11, 0x12, ...};
3
4  extern int initFunction(int value) __init;
5  static int __init otherInitFunction(int value)
6  { ... }
7
8  static int __devexit hotplug_unplugDevice(int minorNumber)
9  { ... }
```

3.3.2 SysFS-Anbindung über das Gerätemodell

Notwendige Datenstrukturen

In den folgenden beiden Abschnitten wird der *HelloWorld*-Treiber schrittweise um eine zeichenorientierte Ein- und Ausgabe mit Unterstützung des Gerätemodells erweitert. Im aktuellen Kernel (2.6.11-7) ist es zwar immer noch möglich, zeichenorientierte Treiber nach dem (einfacheren) aus dem Kernel 2.4 bekannten Schema zu schreiben, aber um Hotplugging mit `udev`, das `SysFS` und die ACPI-Unterstützung vollständig nutzen zu können, ist es notwendig, Treiber mit expliziter Unterstützung für das neue Gerätemodell zu schreiben:

Listing 3.5: Erweiterung von helloWorld um die SysFS-Anbindung

```
1  #include <linux/fs.h>        /* i-nodes and file_operations   */
2  #include <asm/uaccess.h>     /* copy_to/from_user             */
3  #include <linux/version.h>   /* kernel version the module was */
4                               /* compiled with                 */
5  #include <linux/module.h>    /* macros and definitions        */
6                               /* (ie. MODULE_LICENSE)          */
7  #include <linux/cdev.h>      /* cdev_init...                  */
8  #include <linux/device.h>    /* driver model related things   */
9
10 #define BUFFER_LENGTH 255
11 #define NUMBER_OF_MINORS 1
12
13 /* zero: let the kernel choose the device number to use  */
14 #define DESIRED_MAJOR 0
15
16 static char buffer[BUFFER_LENGTH] = "this is the default text\n";
17
18 static dev_t myDevice;       /* remember device number for exit function */
19
20 /* forward declarations for file operations */
21 static int myOpen     (struct inode *openInode,struct file  *openFile);
22 static int myRelease  (struct inode *releaseInode,
23                        struct file *releaseFile);
24 static ssize_t myRead (struct file *readFile, char __user *userBuffer,
25                        size_t count, loff_t *offset );
26 static ssize_t myWrite(struct file *writeFile, const char *userBuffer,
27                        size_t count, loff_t *offset );
28
```

```
29 /* entry-points for file operation functions */
30 static struct file_operations helloWorld_fops =
31 { .owner          = THIS_MODULE,
32   .open           = myOpen,
33   .release        = myRelease,
34   .read           = myRead,
35   .write          = myWrite,
36 };
37
38 /* kernel object container for character devices */
39 static struct cdev helloWorld_cdev = {
40         .kobj   = { .name = "hello", },
41         .owner = THIS_MODULE,
42 };
43 /* container for new device class and SysFS entry */
44 static struct class_simple * mySysfs_class;
```

Die Header-Datei `linux/fs.h` stellt alle Funktionen und Strukturen zur Anbindung von Gerätedateien bereit. Zeichenorientierte Geräte für das neue Gerätemodell erfordern das Einbinden der Dateien `linux/device.h` und `linux/cdev.h`.
Den Kern der Gerätemodellunterstützung bildet die `cdev`-Struktur. Sie beinhaltet neben der *kObject*-Struktur und einem Zeiger auf das Modul selbst, zwei sehr wichtige Komponenten: Den Referenzzähler für die Zugriffe auf die Gerätedatei und die momentan verwendete `file_operations`-Struktur. Die `file_operations`-Struktur enthält die Treiber-Einsprungspunkte für die anzumeldende Gerätedatei. Je nach internem Zustand des Treibers könnten sich diese auch ändern. Das `__user`-Schlüsselwort vor der Userspace-Adresse in den Funktionen `myRead` und `myWrite` ist sehr wichtig. Es steht für `__attribute__((noderef, address_space(1)))` und veranlasst den Compiler beim Zugriff auf die nicht im Kerneladressraum befindliche Adresse einen Kontextwechsel durchzuführen.

Listing 3.6: SysFS-erweiterte Initialisierungsroutine von helloWorld

```
45 static int __init init_myModule(void)
46 { int returnCode;
47
48   // allocate a random major number with one minor number
49   returnCode = alloc_chrdev_region(&myDevice, DESIRED_MAJOR,
50                                    NUMBER_OF_MINORS,
51                                    "helloWorld_IO");
52   if (returnCode) return returnCode;
53
54   // driver model: init character device structure
55   cdev_init(&helloWorld_cdev, &helloWorld_fops);
56
57   // connect assigned device numbers to it
58   returnCode = cdev_add(&helloWorld_cdev, myDevice, NUMBER_OF_MINORS);
59
60   if(returnCode) {
61         //something went wrong
62         printk(KERN_ERR "unable to get device numbers: major# %d\n",
63                MAJOR(myDevice));
64         //decrement the reference count since cdev_add failed
65         kobject_put(&helloWorld_cdev.kobj);
```

```
66          //free used device numbers
67          unregister_chrdev_region(myDevice, NUMBER_OF_MINORS);
68          return returnCode;
69      }
70
71      // create a new SysFS class and directory
72      mySysfs_class = class_simple_create(THIS_MODULE, "helloWorld");
73
74      // add devicefile to it
75      class_simple_device_add(mySysfs_class, myDevice, NULL, "helloWorld_dev");
76
77      return 0; // initialization successful
78  }//init_myModule
79
80  module_init (init_myModule);
```

Um einen Treiber mit Unterstützung für das Gerätemodell beim Kernel anzumelden, sind drei Schritte nötig:

1. Gerätenummer(n) reservieren,
2. ein kObject für die Gerätedateien erzeugen, initialisieren und
3. das kObject dem Kernel übergeben.

Zusätzlich zum (einmaligen) Anmelden des Gerätetreibers müssen alle unterstützten Hardwaregeräte mit dem implementierten Treiber und seinem `kObject` verknüpft werden. Die dafür notwendigen Schritte werden im Abschnitt 4.2.1 besprochen.
Laut Linus Torwalds, dem Schöpfer von Linux, sollte die Verwendung fester Major- und Minor-Nummern (bzw. Gerätenummern), wenn möglich, vermieden werden:

> "We should resist any efford that makes the numbers 'mean' something. They are random cookies. Not 'unique identifies' and not 'addresses'."[9]

Der Artikel „The future of device numbers"[7] beschreibt ausführlich die Hintergründe der neuen Vorgehensweise. Die Verwaltung von Gerätedateien wird mit jeder Kernelversion schrittweise von dem alten Major/Minor-Nummern-System auf Gerätenummern umgestellt.

Reservieren von Gerätenummern

Die feste Einteilung in 256 Major- und 256 Minor-Nummer existiert im aktuellen Kernel nicht mehr, bisher unterstützen jedoch viele Applikationen und Dateisysteme noch keine Gerätenummern. Aus diesem Grunde verwendet der 2.6er Kernel ein (evtl. vorübergehendes) Zuordnungsschema von Gerätenummern zu Major- und Minor-Nummern. Die oberen 12 Bit der Gerätenummer entsprechen der Major-Nummer und die unteren 20 Bit der Minor-Nummer. Um nach diesem Schema aus einer Major- und Minor-Nummer, eine Gerätenummer zu konstruieren, kann das Makro `MKDEV(major,baseminor)` verwendet werden. Zum Reservieren einer vom Kernel dynamisch zugewiesenen Gerätenummer bzw. Folge von Gerätenummern, wird die Funktion:

`int alloc_chrdev_region(dev_t *dev, unsigned base, unsigned count, char* name);`

verwendet. Die erste der Folge von Gerätenummern wird als Rückgabe im Parameter `dev` abgelegt. Obwohl `dev_t` momentan vom Typ `unsigned int` ist, sollte auf die Gerätenummer ausschließlich über die Makros `MAJOR(dev)` und `MINOR(dev)` zugegriffen werden, um die Kompatibilität mit späteren Linux-Versionen zu wahren. Der Parameter `count` gibt die Anzahl der zu reservierenden Gerätenummern (bzw. Minor-Nummern) an. Der Parameter `base` ermöglicht es, die

erste zu verwendende Minor-Nummer zu spezifizieren. So es der Anwendungsfall nicht unbedingt erfordert (z.B. aus Kompatibilität zu altem Code), sollte hier `0` übergeben werden. Spätestens im Kernel 2.7 wird die Major-Nummer ausschließlich die erste verwendete Gerätenummer und die Minor-Nummer nur noch ein Offset dazu repräsentieren. Innerhalb des Kernels werden die reservierten Gerätenummern unter dem im Parameter `name` angegebenen Gerätetreibernamen verwaltet.

Erzeugung und Initialisierung eines kObjects

Die reservierten Gerätenummern der zeichenorientierten Geräte sind ohne Anmeldung des Treibers beim Gerätemodell nicht nutzbar. Die Anmeldung geschieht mit Hilfe der Kernel-Objekt-Struktur (`kObject`)[13]. Um einen eigenen Treiber beim Gerätemodell zu registrieren, ist zunächst eine `cdev`-Struktur für den Treiber zu erzeugen, in die die `kObject`-Struktur eingebettet ist. Es ist sinnvoll, wenn auch nicht zwingend erforderlich, diese wie in Zeile 40 mit einem Treibernamen zu initialisieren. Die Funktion:

`void` *`cdev_init`*`(struct cdev *cdev, struct file_operations *fops);`

initialisiert die nicht vom Programmierer angebenenen Felder der Struktur (Referenzzähler, das kObject und die Listenverwaltung) und verknüpft sie mit der Funktionstabelle `fops`, die die Einsprungspunkte des Treibers enthält.

Verbindung mit dem Gerätemodell

Die Verknüpfung des Treibers mit den zuvor reservierten Gerätenummern erledigt:

`int` *`cdev_add`*`(struct cdev *cdev, dev_t *dev, unsigned count);` .

Damit ist der Gerätetreiber im System angemeldet und wird in der Datei `/proc/devices` mit der zugewiesenen Major-Nummer aufgelistet. Der `SysFS`-Eintrag des Modules befindet sich unter: `/sys/modules/helloWorld_IO`. Schlägt die Funktion `cdev_add` fehl, muss der Referenzzähler für die initialisierte Struktur wieder dekrementiert werden:

`void` *`kobject_put`*`(struct kobject *objectToDestroy);` .

Wurde das erstellte `kObject` dem Gerätemodell übergeben, ist der Treiber in der Lage, über zu definierende Funktionen mit dem Userspace zu kommunizieren. In einigen Fällen ist es jedoch zusätzlich sinnvoll, eine eigene Geräteklasse für das Gerät anzulegen, zum Beispiel wenn der implementierte Treiber zu keinem bereits existierenden Subsystem passt oder eine eigene (spezielle) Hotplug-Funktion zur Verfügung stellt[8].

Erstellung von Geräteklassen

Im Gerätemodell werden Treiber und Geräte in verschiedene Geräteklassen unterteilt. Vom Kernel vordefinierte Klassen sind zum Beispiel: `input`, `usb` und `printer`. Im Kernel existiert (noch) keine Geräteklasse für HDTV Wiedergabesysteme. Aus diesem Grunde verwenden die Vertreter der HHI CineCard-Familie die selbst erzeugte Geräteklasse `hhi-cinecard`. Um eine eigene (einfache) Hardwareklasse im Gerätemodell zu definieren und diese zum `class`-Verzeichnis des `SysFS` hinzuzufügen, kann die Funktion:

`struct class_simple *`*`class_simple_create`*`(struct module *owner, char *name);`

[13] Blockorientierte Geräte nutzen bisher noch die alte Schnittstelle des 2.4er Kernels und werden über die Funktion `register_blkdev` angemeldet.

verwendet werden. Als Parameter `owner` ist im Normalfall `THIS_MODULE` zu übergeben. Mehr Kontrolle über die zu erstellende Klasse bietet:

`int class_register(struct class *toRegister);` .

Die erstellte Klasse wird in diesem Falle unter dem in `toRegister->name` übergebenen Namen registriert. Das zuvor erstellte und dem Gerätemodell übergebene `kObject` des Gerätetreibers kann mit der Funktion:

```
struct class_device class_simple_device_add(struct class_simple *class,
                                            dev_t deviceNumber,
                                            struct device *device,
                                            const char *fmt, ...);
```

mit der zuvor definierten und in `class` übergebenen Klasse verknüpft werden.

Deinitialisierung

Beim Deinitialisieren ist die umgekehrte Reihenfolge der Initialisierung einzuhalten, damit noch benötigte Ressourcen nicht vorzeitig freigegeben werden. Eine dem Gerätemodell erfolgreich übergebenen `kObject`-Struktur wird über die Funktion:

`void cdev_del(struct cdev *cdev);`

abgemeldet. Wird die damit verbundene Gerätedatei noch verwendet, wird zunächst nur der Referenzzähler dekrementiert und die Gerätedatei gegen erneutes Öffnen gesperrt. Die Struktur wird erst aus dem Gerätemodell entfernt, wenn keine Referenzen auf das `kObject` mehr bestehen.

Listing 3.7: SysFS-erweiterte Deinitialisierungsroutine von helloWorld

```
81 static void __exit cleanup_myModule(void)
82 { /* remove device entry in "helloWorld" subdirectory */
83   class_simple_device_remove(myDevice);
84
85   class_simple_destroy(mySysfs_class); /* delete subdirectory */
86   cdev_del(&helloWorld_cdev); /* delete kobject and container */
87
88   /* unregister the character driver itself */
89   unregister_chrdev_region(myDevice,NUMBER_OF_MINORS);
90 }/* cleanup_MyModule */
91 module_exit (cleanup_myModule);
```

3.3.3 Datenaustausch mit dem Userspace

Die Kommunikation des Treibers mit dem Userspace findet bei zeichen- und blockorientierten Geräten über die in der `file_operations`-Struktur (fops) definierten Einsprungspunkte in den Treiber statt. Die mit `.open` und `.release` verknüpften Funktionen müssen nicht unbedingt implementiert werden. Hauptaufgabe der `.open`-Funktion ist die Initialisierung von evtl. verwendeten Datenstrukturen. Hier muss auch geprüft werden, ob der anfordernde Prozess die Gerätedatei öffnen darf (z.B. wenn die Gerätedatei nur schreibar ist, oder nur von einer bestimmten Anzahl von Prozessen gleichzeitig geöffnet werden soll). In vielen Fällen reicht die Standardfunktion des

Kernels für `.open` aus. Sind keine Ressourcen nach dem Schließen der Gerätedatei freizugeben, kann ebenfalls die vom Kernel angebotene `.release`-Funktion verwendet werden.

Listing 3.8: helloWorld mit einfachen E/A-Operationen

```
92  static int myOpen (struct inode *myInode, struct file  *myFile)
93  { printk(KERN_INFO "device opened!\n");
94    return 0;
95  }/* myOpen */
96
97  static int myRelease (struct inode *myInode, struct file  *myFile)
98  { printk(KERN_INFO "device closed!\n");
99    return 0;
100 }/* myRelease */
101
102 static ssize_t myRead (struct file *myFile, char __user *userBuffer,
103                        size_t count, loff_t *offset )
104 { ssize_t not_copied;
105   ssize_t to_copy = strlen(buffer)+1;
106
107   if (count<to_copy) to_copy = count;
108
109   not_copied  = copy_to_user(userBuffer,buffer,to_copy);
110   return to_copy - not_copied;
111 }/* myRead */
112
113 static ssize_t myWrite (struct file *myFile, const char *userBuffer,
114                         size_t count, loff_t *offset )
115 { ssize_t not_copied;
116   ssize_t to_copy = BUFFER_LENGTH;
117
118   if (count<to_copy) to_copy = count;
119
120   not_copied  = copy_from_user(buffer,userBuffer,to_copy);
121   buffer[to_copy - not_copied]=0;
122
123   return to_copy - not_copied;
124 }/* myWrite */
```

Das Lesen und Beschreiben von Gerätedateien findet über den von der Applikation im Userspace bereitgestellten Pufferspeicher statt. Auf diesen Puffer kann mit den folgenden Funktionen bzw. Makros zugegriffen werden:

- `ssize_t` *`copy_from_user`*`(void *to, const void *from, ssize_t count);`
  Zum Lesen eines (größeren) Datenblockes aus dem Userspace. Die Funktion liefert die Anzahl der Bytes zurück, die *nicht* kopiert werden konnten, also `0` im fehlerfreien Falle.
- `ssize_t` *`copy_to_user`*`(void *to, const void *from, ssize_t count);`
  Zum Schreiben eines (größeren) Datenblockes in den Userspace. Die Funktion liefert die Anzahl der Bytes zurück, die *nicht* geschrieben werden konnten.
- `int` *`get_user`*`(kernel-variable, userspace-variable);`
  Zum Lesen einer einzelnen Variablen. Die Anzahl der transferierten Bytes hängt vom Typ der Variablen ab.

- `int` *`put_user`*`(kernel-variable, userspace-variable);`
  Zum Schreiben einer einzelnen Variablen. Die Anzahl der transferierten Bytes hängt vom Typ der Variablen ab.

Alle Funktionen zur Kommunikation können die zugehörige Treiberinstanz schlafen legen und sollten damit nicht im Interrupt-Kontext verwendet werden. Ein direkter Zugriff auf den Adressbereich des Prozesses im Userspace, wie er z.B. auf MIPS-Systemen möglich wäre, ist auf vielen Hardwarearchitekturen nicht möglich - ratsam ist er jedoch auf keiner. Auf Systemen mit einer Memory-Management-Unit, kurz MMU, ist dafür eine MMU-Adressumrechnung oder ein Kontextwechsel in der virtuellen Speicherverwaltung notwendig. Es sollte also schon aus Kompatibilitätsgründen nicht versucht werden, direkt auf den angegebenen Speicherbereich zuzugreifen. Darüber hinaus ist die physikalische Adresse des Speicherbereiches im Segment der Applikation, dem Treiber normalerweise nicht bekannt und sie kann sich auch im Falle der Auslagerung von Speicherseiten jederzeit ändern. Die übergebenen Zeiger müssen deshalb überprüft werden, ob sie gültig sind (falls eine falsche Adresse übergeben wurde) und es muss sicher gestellt werden, dass ausschließlich auf den Speicherbereich des aktuellen Prozesses zugegriffen wird. Die dafür notwendige Funktionalität stellen die oben genannten Funktionen zu Verfügung.
Sollen pro Zugriff mehrere (kleinere) Datenblöcke oder viele einzelne Variablen kopiert werden, kann auf die wiederholte Überprüfung der unveränderten Adresse verzichtet werden, um das Laufzeitverhalten des Treibers zu optimieren:

1. Zugriffsadresse prüfen und evtl. das Wiedereinlagern von Speicherseiten veranlassen:
   `int` *`access_ok`*`(TYPE, address, size);`
   `TYPE` kann entweder `VERIFY_READ` oder `VERIFY_WRITE` sein. Ist der Zugriff nicht möglich, liefert die Funktion `-EFAULT` zurück, sonst `0`.

2. Zugriff auf den Speicherbereich mit:
   `ssize_t` *`__copy_from_user`*`(void *to, const void *from, ssize_t count);`
   `ssize_t` *`__copy_to_user`*`(void *to, const void *from, ssize_t count);`
   `int` *`__get_user`*`(kernel-variable, userspace-variable);`
   `int` *`__put_user`*`(kernel-variable, userspace-variable);`
   Diese Funktionen bzw. Makros prüfen nicht die Gültigkeit des Speicherbereiches und sollten nur innerhalb *eines* `read-`/`write-` oder `ioctl`-Zyklusses[14] verwendet werden.

[14] Der Abschnitt 4.1.2 beschäftigt sich näher mit IO-Control-Handler und den Vor- und Nachteilen dieser Kommunikationsmöglichkeit.

Kapitel 4

Implementation - vom Userspace bis zur Hardware

4.1 HyperCore - die zentrale Verwaltungsinstanz

4.1.1 Kartenmanagement

Die zentrale Komponente des Kartenmanagements innerhalb des CineCard-Treibers ist die Funktion `registerCard`. Mit ihrer Hilfe werden die durch die Hardware-Backends erkannten PCI-Karten im CineCore registriert. Die Verwaltung aller in einem System verwendeten PCI-Karten der HHI CineCard Familie geschieht mittels eines Feldes von `recdecCard_t`[1]-Strukturen.

Listing 4.1: die recdecCard-Struktur

```
49 struct recdecCard_s {
50         /* ************** hardware type information ****************** */
51         int cardType;    //! 0 unused, 1 dummy, 2 hypercard2, 3 cinecard
52         u32 revision;    //! hardware revision
53         u8 subrevision;
54         u8 videoISP_enable;
55         int playing;    // 0=stop, 1=playing
56         int recording; // 0=stop, 1=recording
57
58         /* ************** backend funktions ************************** */
59         int  (* readRegister)        ( recdecCard_t *, void * );
60         int  (* writeRegister)       ( recdecCard_t *, void * );
61         int  (* readBridgeRegister)  ( recdecCard_t *, void * );
62         int  (* writeBridgeRegister) ( recdecCard_t *, void * );
63         int  (* resetCard)           ( recdecCard_t *, void * );
64         int  (* enablePlayback)      ( recdecCard_t *);
65         int  (* writeDMABlock)       ( recdecCard_t *);
66
67         /* ************* kernel API ********************************** */
68         struct          pci_dev * dev;
69         struct          class_device* classDevice;
70         u32             minor; // card ID
```

[1] recdec steht im CineCard-Treiber für Recorder/Decoder.

```
71
72          void *backendData;      /* hardware specific data */
73          void *bridgeBackendData;
74
75          ringBuffer_t playBuffer;
76          ringBuffer_t recordBuffer;
77
78          wait_queue_head_t playerWaitQueue; /* to suspend player-process */
79          signed long playerTimeout;
80          wait_queue_head_t recorderWaitQueue;
81          signed long recordTimeout;
82  };/* recdecCard_s */
```

Die `recdecCard_t`-Struktur selbst, enthält nur hardwareunabhängigen Informationen über die registrierte PCI-Karte der CineCard-Familie. Hardwareabhängige Informationen werden mit lediglich ihr verknüpft. In der `registerCard`-Funktion werden je nach Ausstattung und Funktionsumfang der PCI-Karte, die dafür nötigen Datenstrukturen initialisiert. So wird zum Beispiel bei einer reinen Wiedergabekarte kein Ringpuffer zur Aufnahme angelegt. Nach der Registrierung initialisiert das jeweilige Backend die Zeiger auf die von ihm unterstützten Funktionen (Zeilen 59 bis 65) und bettet alle weiteren zum Betrieb notwendigen Daten in eine eigene, mit `backendData` verknüpfte Struktur ein. Damit werden alle hardwareabhängigen Daten vollständig gegenüber dem CineCore gekapselt, was die Backends jederzeit (auch zur Laufzeit) einfach austauschbar macht.

4.1.2 Interface zum Userspace

Gerätedateien

Die Gerätedateien des CineCore gliedern sich in zwei Gruppen auf: `control`- und `dvr`-Dateien. Über die `control`-Dateien werden die PCI-Karten konfiguriert und gesteuert. Die `dvr`-Dateien (digital video recorder) sind ausschließlich für die Wiedergabe und Aufnahme von Datenströmen bestimmt. Die Minor-Nummer der `control`-Gerätedatei entspricht der Position der CineCard im Feld der `recdecCard_t`-Strukturen. Die Zuordung der Minor-Nummer zur jeweiligen `dvr`-Gerätedatei ist abhängig von der im Treiber definierten Konstante `MAX_RECDEC_CARDS` und entspricht `control`-Minor + `MAX_RECDEC_CARDS`. Damit ergibt für sich für `MAX_RECDEC_CARDS=8` folgendes Zuordnungsschema:

Gerätedatei im `/dev/` Verzeichnis	Minor-Nummer	Verwendung
`hhi-cinecard/control0`	0	zur Konfiguration der ersten CineCard im System
`hhi-cinecard/control1`	1	zur Konfiguration der zweiten CineCard im System
`hhi-cinecard/control2`	2	zur Konfiguration der dritten CineCard im System
⋮	⋮	⋮
`hhi-cinecard/dvr0`	8	für die Videodaten der ersten CineCard im System
`hhi-cinecard/dvr1`	9	für die Videodaten der zweiten CineCard im System
`hhi-cinecard/dvr2`	10	für die Videodaten der dritten CineCard im System
⋮	⋮	⋮

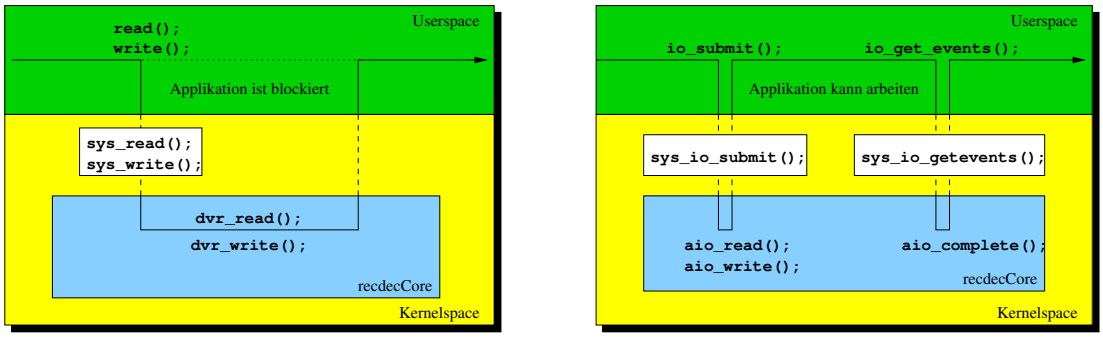
Eine Gerätedatei kann erst von einer Applikation im Userspace geöffnet werden, wenn der Gerätetreiber beim Gerätemodell korrekt angemeldet und der zugehörigen Gerätenummer, wie im Ab-

schnitt 3.2.4 beschrieben, eine Gerätedatei zugeordnet wurde. Dabei werden von der geräteunabhängigen Systemsoftware die in der `file_operations`-Struktur definierten Einsprungspunkte des Treibers verwendet. Allen in dieser Struktur registrierten Funktionen werden vom Kernel jeweils zwei Datenstrukturen übergeben: `struct inode` und `struct file`.
Die `inode`-Struktur enthält alle Daten über die Gerätedatei, auf die zugegriffen wurde. Dazu gehören die Besitzerinformationen, Zugriffsrechte und auch die Major- und Minor-Nummer (bzw. die Gerätenummer und der Geräte-Offset).
Wird eine Gerätedatei mehrfach geöffnet, werden die Zugriffe durch mehrere Treiberinstanzen verarbeitet. Die `file`-Datenstruktur repräsentiert den zugreifenden Prozess im Userspace und enthält sämtliche Informationen der bearbeitenden Treiberinstanz. Zusätzlich besteht die Möglichkeit treibereigene Daten über den `void *private_data`-Zeiger in die `file`-Struktur einzubetten. Der CineCore nutzt diese Möglichkeit, um die zur zuvor geöffneten Gerätedatei gehörende `recdecCard_t`-Struktur mit der verarbeitenden Treiberinstanz zu verknüpfen. Damit ist es möglich die Verwaltung der Kartenstrukturen vollständig von deren Verwendung zu trennen. Die zur Gerätedatei und PCI-Karte gehörende `recdecCard_t`-Struktur wird in der Funktion `recdecOpen` ermittelt. Dazu wird die in der `inode`-Struktur gespeicherte Minor-Nummer verwendet.

AsyncIO vs. SyncIO

Der asynchrone Zugriffsmodus des Kernel 2.6 verspricht Geschwindigkeitsvorteile bei der Ein- und Ausgabe. Eine auf die Festplatte schreibende Applikation wird nicht mehr durch blockierende Schreiboperationen angehalten und kann sich in der Zeit bis zum tatsächlichen Schreiben[2] der Daten auf die Festplatte anderen Aufgaben widmen oder schon die nächste Schreiboperation vorbereiten. Bei einer synchronen I/O-Operation hingegen, muss ein Programm einen zweiten Thread starten, um während der blockierenden I/O-Operation weiter arbeiten zu können. Zusätzlich muss mit jeder I/O-Operation der Scheduler aktiv werden, was unnötige Geschwindigkeitseinbußen mit sich bringt. Die gegenwärtige Implementierung von Async-I/O im Kernel 2.6

Abbildung 4.1: Vergleich synchroner und asynchroner Ein- und Ausgabe



Synchrone I/O–Operation

Asynchrone I/O–Verarbeitung

besitzt jedoch noch viele Unzulänglichkeiten. Neben der mangelhaften Dokumentation, wie sich diese Zugriffsart sinnvoll nutzen lässt, bringt die Implementierung bisher keine nennenswerten

[2]Die zu schreibenden Daten werden normalerweise zunächst in den Festplattencache übertragen, womit der theoretische Geschwindigkeitsvorteil des asynchronen Zugriffs nicht zum Tragen kommt.

Geschwindigkeitsvorteile. Daher überrascht es nicht, dass neben Oracle nur wenige den asynchronen Zugriff benutzen. Weitere Informationen zu den Vor- und Nachteilen der asynchronen Zugriffsmodi sind im Artikel Kern-Technik des Linux-Magazins 02/2005 [5] nachzulesen.
Die bisher für die HyperCard II verwendeten Applikationen benutzen ausschließlich synchrone Ein- und Ausgabeoperationen. Es ist nicht zu erwarten, dass in naher Zukunft vom Treiber asynchrone Zugriffsmodi unterstützt werden müssen. Aus diesem Grunde fiel die Entscheidung gegen die Implementation asynchroner Zugriffsmodi.

Der IO-Controls-Handler

Der im CineCore implementierte IO-Control-Handler dient in erster Linie der Kompatibilität mit dem schon bestehenden Applikationsumfeld (die rcontrol2 Software der HyperCard I). IO-Controls bieten eine sehr flexible Möglichkeit, mit dem Treiber zu kommunizieren und sind nicht standardisiert. Userspace-Applikationen benötigen dafür eine spezielle Unterstützung, die z.B. in vielen Skriptsprachen nicht vorhanden ist. Aus diesem Grunde sollte wo es möglich ist, auf IO-Controls verzichtet werden.
Der IO-Control-Handler des CineCore bietet die Möglichkeit, Steuerbefehle an das Treibersystem zu senden und Register der HyperCard II und CineCard gezielt zu beschreiben und auszulesen. Den Kern des Handlers bildet eine große `case`-Anweisung. Je nach Identifikationsnummer des IO-Controls werden hier die eventuell notwendigen Parameter aus dem Userspace kopiert, in die behandelnde Unterfunktion verzweigt und anschließend das Ergebnis der Operation an den aufrufenden Prozess im Userspace übermittelt. Die zur PCI-Karte zugehörige `recdecCard_t`-Struktur wurde, wie zuvor beschrieben in die `File`-Struktur des Kernels eingebettet und steht dem IO-Control-Handler unmittelbar zu Verfügung. Damit entfällt auch hier die Suche nach der Struktur im Feld aller CineCard-Karten.

Listing 4.2: Ausschnitt aus dem IO-Control-Handler des CineCore

```
223 static int controlIOctlHandler (struct inode *ioctlInode,
224                                 struct file  *ioctlFile,
225                                 unsigned int  ioctlCommand,
226                                 unsigned long ioctlArgs)
227
228 { cineCoreCall_t call;  /* the call to construct */
229   int            returnValue = 0;
230   recdecCard_t  *card        = ioctlFile->private_data;
231
232   if (!card) {
233         eprintk ("ERROR: card information lost\n");
234         return -EPERM;
235   }
236
237   tprintk("ioctlHandler (device @minor %i)\n",card->minor);
238
239   if (((ioctlFile->f_flags&O_ACCMODE)==O_RDONLY))
240         return -EPERM;
241
242   switch (ioctlCommand) {
243         /* alters a register of the pci-bridge */
244         case CONT_WRITE_BRIDGE: {
245                 dprintk(2, "got CONT_WRITE_BRIDGE\n");
246                 call.function = card->writeBridgeRegister;
247                 call.arg = kmalloc(sizeof(recdecAddress_t),GFP_KERNEL);
```

```
248                 if (call.arg == NULL) return -ENOMEM;
249                 if (copy_from_user(call.arg,
250                                    (recdecAddress_t*)ioctlArgs,
251                                     sizeof(recdecAddress_t))) {
252                         kfree(call.arg);
253                         return -EFAULT;
254                 }
255                 returnValue = executeFunction(card,&call);
256                 if (returnValue)
257                         eprintk ("ERROR: write_bridge failed\n");
258                 kfree(call.arg);
259         }; break;
260   .
261   .
262   .
```

In einer späteren Ausbaustufe soll der Treiber um ein sehr flexibles System zur Behandlung von Interrupts des MPEG2-Dekoders erweitert werden, welches mit der Hilfe von aus dem Userspace ladbaren Interrupt-Skripten auf Datenstromfehler und andere schwer vorherzusehende Fehler reagieren soll. Um nicht zwei verschiedene Schnittstellen für die allgemeinen Konfiguration der PCI-Karten (über Konfigurations-Skripte der Applikation) und die Interrupt-Skripte verwenden zu müssen, werden beide Arten von Skripten über die gleiche Schnittstelle des IO-Control-Handlers geladen. Nach der Konstruktion eines Konfigurationsaufrufes in den Zeilen 246 bis 254, wird dieser der Funktion `executeFunktion` übergeben. Statt den Konfigurationsaufruf direkt auszuführen, wäre es möglich, den Aufruf auch zu einem Interrupt-Skript hinzuzufügen und damit die Unterstützung für Interrupt-Skripte nachzurüsten. Das Ergebnis des Aufrufs wird in der `call`-Struktur zurückgeliefert und kann mit der Funktion `copy_to_user` in den Userspace kopiert werden (Siehe auch Abschnitt 3.3.3).

4.1.3 DMA-Transfer für Videodaten

Bisherige Strategie - separate DMA-Puffer

Die Zeitspanne zwischen der Anforderung neuer Daten, die sich im Userspace befinden, bis zu deren Bereitstellung für die Hardware, macht in vielen Fällen einen großen Puffer auf Kernelebene notwendig[3].
Mit Hilfe der Funktion `kmalloc(Speichermenge,GFP_DMA);` lassen sich jedoch nur Speicherbereiche von maximal 128 Kilobyte anfordern. Mit der Funktion `vmalloc` können zwar große Speicherblöcke alloziiert werden, allerdings ist der Speicher nur virtuell zusammenhängend, physikalisch kann er auf viele verschiedene Blöcke verteilt sein. Dieser Speicher eignet sich nicht für direkte (SAC-)DMA-Zugriffe[4], weil die DMA-Einheit nur mit physikalisch zusammenhängenden Adressen arbeiten kann.
Eine mögliche Lösung wäre, physikalisch zusammenhängende Blöcke zu suchen und diese jeweils einzeln zu verschicken (Scatter-Gather-Strategie) - diese Lösung disqualifiziert sich jedoch durch den entstehenden hohen Verwaltungaufwand und die zu erwartende mangelhafte Geschwindigkeit.
In vielen bisherigen Treibern werden per DMA zu versendende Daten zunächst in einen mehrere

[3] Im ungünstigsten Falle müssen die Daten erst von der Festplatte gelesen werden, was neben dem zeitaufwändigen Lesen vom Datenträger zwei weitere Kontextwechsel nötig macht. Jede E/A-Operation birgt bei einem preemptiven Kernel zusätzlich das Risiko, zugunsten eines höher priorisierten Prozesses unterbrochen zu werden.
[4] Single-Address-Cycle - dies ist der normale, performanteste DMA-Zugriffsmodus.

Megabyte fassenden Ringpuffer im Kernelbereich geschrieben. Aus dem Ringpuffer heraus werden die Daten in einen zusätzlichen kleinen DMA-fähigen Puffer kopiert, aus dem anschließend der DMA-Transfer initiiert wird. Diese Strategie hat jedoch einige entscheidende Nachteile, die sich besonders bei hohen Datenraten bemerkbar machen:

1. das Umkopieren der Daten vor dem DMA-Transfer erzeugt zusätzlichen (wenn auch geringen) Aufwand in der Verwaltung der Puffer und benötigt (eventuell sehr viel)[5] zusätzliche Rechenzeit,
2. die entstehende Latenzzeit durch den Kopiervorgang erhöht das Risiko von Pufferunterläufen (auf Hardwareebene) und
3. per `kmalloc` erzeugte Puffer können maximal 128 Kilobyte groß sein.

DMA-fähiger Ringpuffer

Die zahlreichen Nachteile der Ringpuffer/DMA-Puffer-Kombination lassen sich durch den direkten DMA-Zugriff auf den (großen) Ringpuffer vermeiden. Der Linux-Kernel bietet dafür seit einiger Zeit die Möglichkeit, große zusammenhängende Speicherblöcke zu alloziieren, falls die verwendete Hardware dies auch unterstützt. Moderne Computersysteme mit PCI-Bus, die leistungsfähig genug sind, um für den Einsatz mit (mindestens) einer der HHI CineCards in Betracht zu kommen, können auf mindestens ein Gigabyte des adressierbaren Speicherbereiches per DMA zugreifen. Ein DMA-Transfer direkt aus dem Ringpuffer heraus kann die Gefahr von (hardwareseitigen) Pufferunterläufen entscheidend verringern, weil sofort Daten zum Versenden bereitstehen, ohne zuvor in den DMA-Puffer umkopiert zu werden. Der dadurch eingesparte Verwaltungsaufwand bietet zudem einen leichten Geschwindigkeitsgewinn.
Über 128 Kilobyte große, DMA-fähige Speicherbereiche können über das PCI-Subsystem angefordert werden[6]:

```
void *pci_alloc_consistent(struct pci_dev * device, ssize_t count,
                           dma_addr_t *dma_handle);
```

Die Funktion alloziiert einen konsistenten, permanenten Speicherbereich. Die Funktion liefert einen Zeiger auf die vom Prozessor aus erreichbare (virtuelle) Startadresse des Speichers und als dritten Parameter `dma_handle` die von der DMA-Einheit verwendbare Adresse zurück. Der angeforderte Speicherbereich muss vor dem Entladen des Modultreibers wieder per:

```
void pci_free_consistent(struct pci_dev * device, ssize_t count, void *cpu_addr,
                         dma_addr_t *dma_handle);
```

freigegeben werden. Konsistente (auch als koherent oder synchron bezeichnete) Speicherbereiche eignen sich allerdings nicht für alle Anwendungen. Sie sind hauptsächlich für Pufferspeicher bestimmt, die beim Laden des Treibers alloziiert und erst beim Entladen des Modules wieder freigegeben werden.

[5] Je nachdem, wo sich der Ringpuffer des Treibers im Speicher befindet, können die Daten eventuell nur durch sehr zeitaufwändige High-Memory Zugriffe erreicht werden. Die DMA-Einheit kann jedoch in vielen Fällen direkt auf HMA-Speicher zugreifen. Ist der Systemspeicher sehr begrenzt, kann die Speicherseite sogar ausgelagert sein.

[6] Dies ist sogar im Interrupt-Kontext möglich, allerdings kann die Funktion auch fehlschlagen, wenn kein ausreichend großer Speicherbereich verfügbar ist.

Eine andere Art, DMA-fähigen Speicher anzufordern, ist das so genannte *streaming DMA-Mapping* (auch als asynchron oder „outside the coherency domain“ bezeichnet). Dabei werden viele kleinere Bereiche verwendet, die nach dem Zugriff sofort wieder freigegeben werden (Scatter-Gather-Betrieb). Einige Hardwarearchitekturen besitzen spezielle Optimierungsmöglichkeiten für diesen Zugriffsmodus. Das *streaming DMA-Mapping* findet oft Anwendung in Gerätetreibern, die kurzzeitig große Speicherbereiche benötigen, wie zum Beispiel Treiber für Dateisysteme oder Netzwerkgeräte. Eine ausführliche Übersicht über die Vor- und Nachteile der beiden Techniken und deren Anwendung bietet das Dokument `DMA-mapping.txt` in den Kernelquellen[10].

4.2 Hyperbackend - Treiber für die HyperCard II

4.2.1 Hardwaredetektion

In der Initialisierungsfunktion des Moduls `init_hyperbackend` wird durch den Aufruf von:

```
int pci_module_init(struct pci_driver *driver);
```

dem PCI-Subsystem mitgeteilt, welche Art von Hardware vom Hyperbackend unterstützt wird. Hier wird die erkannte PCI-Karte auch beim Gerätemodell registriert. Dabei wird die Struktur `hyperbackend_drv` vom Typ `pci_driver` übergeben. Sie enthält neben einer Tabelle zur Identifikation des unterstützten PCI-Gerätes auch die Zeiger auf die `.probe`-Funktion zum Suchen nach erkannter Hardware und die `.remove`-Funktion zur Deinitialisierung der Hardware beim Entladen des Treibers.

Listing 4.3: Die PCI-Driver Struktur für das PCI-Subsystem

```
712 static struct pci_device_id hyperbackend_tbl[] __devinitdata = {
713   { PCI_VENDOR_ID_PLX, PCI_DEVICE_ID_PLX_9054,
714     RECDEC_PCI_SUBSYSTEM_VENDOR_ID, RECDEC_PCI_SUBSYSTEM_ID,
715     PCI_CLASS_MULTIMEDIA_OTHER, 0, 0 }, {0,}
716 };
717
718 static struct pci_driver hyperbackend_drv = {
719   .name="hyperbackend",
720   .id_table=hyperbackend_tbl,
721   .probe=initDevice,
722   .remove=releaseDevice,
723 };
```

Die Erkennung und Initialisierung des Gerätes findet in der Funktion `initDevice` statt. Das PCI-Subsystem erkennt die PLX-Brücke der HyperCard II aufgrund der in der `hyperbackend_tbl`-Tabelle übergebenen Informationen und ruft die `initDevice`-Funktion auf, der die `pci_dev` Struktur des erkannten PCI-Gerätes übergeben wird. Mit Hilfe der Funktion:

```
int pci_read_config_byte(pci_dev *deviceToInit, offset, u8 *value);
```

wird die Versionsnummer der HyperCard II aus dem Konfigurationsspeicher der PLX-Brücke ausgelesen, um zu verhindern, dass der Treiber auf eine nicht mehr unterstützte aber eventuell noch im System befindliche HyperCard I zugreift. Nach Aktivierung des PCI-Gerätes durch:

```
int pci_enable_device(pci_dev *deviceToInit);
```

wird die erkannte HyperCard II im CineCore via `registerCard` registriert.

4.2.2 Registerzugriff

Durch das Einblenden des Adressbereiches des lokalen Busses der PCI-Brücke ist der Zugriff auf die Konfigurationsregister der CineCards sehr einfach. Dieser Zugriffsmodus wird „Memory Mapped I/O“ genannt. Aus Sicht der CPU besteht auf den meisten PC-Architekturen kein Unterschied zwischen den Konfigurations-Registern der CineCards und normalen Speicherzellen. Um die Kompatibilität zu anderen Architekturen zu wahren, sollte trotzdem auf die eingeblendeten Speicherbereiche nie direkt zugegriffen werden, sondern auf die in der Headerdatei `asm/io.h` definierten Makros `readb`, `readw`, `readl` bzw. `readb`, `readw` und `readl` zurückgegriffen werden.

4.2.3 ISR - Interrupt Service Routine

Der größte Fortschritt in der Entwicklung von Linux bezüglich der Interrupt-Behandlung ist die Einführung eines Rückgabewertes. Die Interrupt-Service-Routinen (kurz ISR) müssen jetzt vom Typ `irqreturn_t` sein. Wurde ein Interrupt erfolgreich behandelt, ist `IRQ_HANDLED` zurückzugeben. Im Fehlerfalle, oder wenn der aufgetretene Interrupt nicht von dem durch den Treiber angesteuerten Gerät ausgelöst wurde[7], sollte `IRQ_NONE` zurückgegeben werden. Wird der Interrupt-Handler mit diesem Wert verlassen, ruft der Kernel die nächste zuständige ISR auf. Wurde der Interrupt nicht behandelt und droht damit die Systemstabilität zu gefährden, wird er so lange deaktiviert, bis sich eine neue ISR beim Kernel anmeldet. Seit der Version 2.6.10 des Kernels ist es zusätzlich zwingend erforderlich, vor dem Anmelden einer ISR, das betreffende PCI-Gerät mit der Funktion `pci_enable_device` zu aktivieren[6].
Die Interrupt-Behandlung ist eine sehr zeitkritische Aufgabe. Es ist wichtig, dass während der Interrupt-Behandlung keine Funktionen aufgerufen werden, die eine längere Zeit blockieren, und dass die Ausführung des Interrupt-Handlers auch nicht zeitweilig unterbrochen wird. Einer der häufigsten Fehler bei der Interrupt-Behandlung ist die falsche Anforderung von Speicher per `kmalloc` z.B. für die Bereitstellung von DMA-Puffern. Ist kein ausreichend großer zusammenhängender Speicherbereich verfügbar, legt der Kernel den anfordernden Kernelthread schlafen, bis der angeforderte Speicher verfügbar ist. Damit die Interrupt-Behandlung nicht unterbrochen wird, sollte auf Speicheranforderung im Interrupt-Kontext ganz verzichtet werden. Ist es dennoch notwendig, in einem Interrupt-Handler Speicher anzufordern, verhindert das Flag `GFP_ATOMIC` bei `kmalloc`, dass der Kernelthread schlafen gelegt wird[2, S. 104]. Es besteht allerdings eine relativ hohe Wahrscheinlichkeit, dass die Speicheranforderung fehl schlägt. Die Verwendung des Flags `__GFP_ZERO`[8], verfügbar seit Kernel 2.6.11, sollte in diesem Zusammenhang vermieden werden. Generell sollten Interrupt-Handler so kurz wie möglich gehalten werden und rechenaufwändige Teile nach dem Löschen der Interrupt-Anforderung außerhalb des Interrupt-Kontextes ausgeführt werden. Der Linux-Kernel stellt für diese Aufgaben Soft-IRQs, Tasklets, Kernelthreads, Workqueues und Event-Workqueues zur Verfügung. Nach dem Auftreten eines Interrupts kann im Normalfall sofort wieder ein Interrupt ausgelöst werden. Damit ist es möglich, dass die Interrupt-Service-Routine von einem neu aufgetretenden Interrupt unterbrochen wird. Ist der Interrupt-Handler für das Gerät nicht reentrant, muss beim Anmelden der ISR das Flag `SA_INTERRUPT` gesetzt werden. Damit werden auf dem lokalen Prozessor (bei SMP-Maschinen der Prozessor, der den Interrupt gerade bedient) die Interrupts gesperrt, bis der Interrupt-Handler beendet ist.
Die Interrupt-Behandlung der CineCards ist ein dreistufiger Prozess, dessen Stufen nach der Priorität geordnet sind:

[7] Dieser Fall kann beispielsweise auftreten, wenn sich mehrere Geräte einen Interrupt teilen und alle verwendeten Treiber bei der Anmeldung der jeweiligen ISR das `SA_SHIRQ`-Bit gesetzt haben (Shared-Interrupt Fähigkeit).

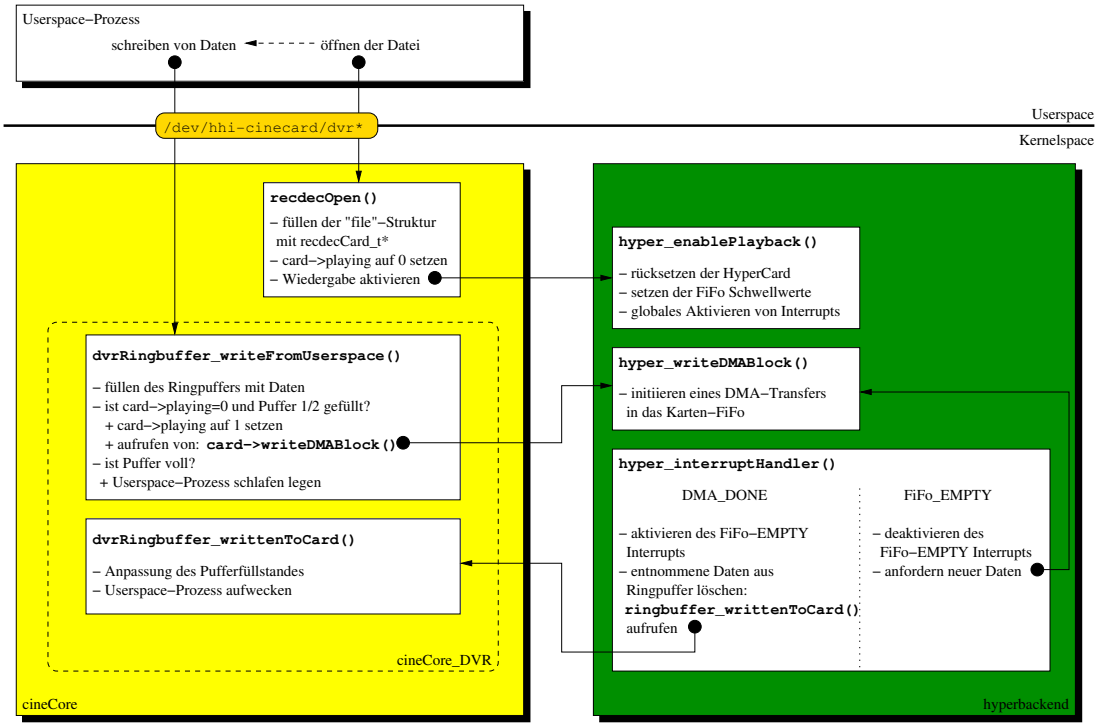
[8] `__GFP_ZERO` veranlasst den Kernel, den alloziierten Speicherbereich zuvor zu löschen, was bei vielen Speicheranforderungen oder im Interrupt-Kontext zu größeren Verzögerungen führen kann.

1. Auslesen und Behandeln der Interrupts der PCI-Brücke,
2. Behandlung der Interrupts, die durch das Daten-FIFO ausgelöst wurden,
3. (geplante) Auswertung von Interrupts anderer Subsysteme auf der PCI-Karte (z.B. bei Dekoderfehlern) durch vom Userspace ladbare Interrupt-Skripte.

4.2.4 Datenflüsse - die „was passiert dann Maschine“

Ein entscheidender Schwachpunkt des alten Treibers, war das Auftreten von Pufferunterläufen zu Beginn der Wiedergabe von Datenströmen. Der vor der Wiedergabe leere Ringpuffer des Treibers konnte sich bei hohen Datenraten und hoher Systemlast nicht ausreichend füllen, bevor neue Daten durch einen Interrupt des Videodaten-FiFos angefordert wurden. Nach dem Auftreten eines Pufferunterlaufes wurde MPEG2-Dekoder zurückgesetzt und die Wiedergabe erneut begonnen - womit der Fehler erneut auftrat. Dieses Problem tritt bei der Aufnahme von Datenströmen nicht auf, hier ist ein möglichst leerer Ringpuffer im Kernel wünschenswert.
Durch die Anlaufschwierigkeiten bei der Wiedergabe, konnte der Treiber Datenströme nur mit einer geringeren Datenrate wiedergeben, als er sie aufzeichnen konnte und blieb speziell bei der Wiedergabe mehrerer Datenströme auf einem Computersystem, weit unter seiner technischen Leistungsfähigkeit.
Im neuen Treiber wird der Ringpuffer vor der Wiedergabe zunächst mindestens bis zur Hälfte gefüllt. Die folgende Abbildung erklärt die zur Wiedergabe von Videodaten notwendigen Schritte:

Abbildung 4.2: Wiedergabe von Datenströmen



4.3 Cinebackend - Eine Vorlage für den Treiber der CineCard Professional

Die CineCard gleicht in ihrem für die Software relevanten Aufbau sehr den HyperCard II. Zur Wiedergabe der Datenströme auf den bis zu fünf MPEG2-Dekodern, wird wie bei der HyperCard II, nur ein einziges Hardware-FiFo verwendet. Die Aufteilung der Videodaten geschieht, für die Software transparent, in der Hardware auf der CineCard. Das Generieren von DMA- und FiFo-Interrupts zur Wiedergabe, folgt ebenfalls dem schon von der HyperCard II bekannten Konzept. Aus diesem Grunde wurde die Vorlage für das Cinebackend nicht auf der Basis des (annähernd leeren) Dummybackends[9] erstellt, sondern mit Hilfe des Backends für die HyperCard II. Häufig benutzte Offset-Adressen von wichtigen Konfigurationsregistern wurden schon im Hyperbackend in einer Headerdatei zusammengefasst, womit das Erstellen einer Vorlage für das CineBackend zusätzlich erleichtert wurde.

Listing 4.4: ein Beispiel für die Konfigurations-Headerdatei für das CineBackend

```
1  #ifndef _CINEBACKEND_CONF_H_
2  #define _CINEBACKEND_CONF_H_
3
4  /* hardware-detection information */
5  #define CINECARD_PCI_SUBSYSTEM_VENDOR_ID 0x15EB
6  #define CINECARD_PCI_SUBSYSTEM_ID        0x9001
7
8  /* ***** base addresses ***** */
9  #define LOCALBUS_START_OFFSET            0x02000000
10 #define VIDEO_ISP_BASE                   0x10000000
11 #define HIPEG1_BASE_ADDR                 0x02000000
12 #define HIPEG2_BASE_ADDR                 0x02200000
13 #define HIPEG3_BASE_ADDR                 0x02400000
14 #define HIPEG4_BASE_ADDR                 0x02600000
15 #define HIPEG5_BASE_ADDR                 0x02800000
16 #define   IRQ_REG                        0x00
17
18 /*        Bit 15=0: all ints suppressed    */
19 #define   IRQ_PIN_REG                    0x08
20 #define CONTROL_BASE                     0x04000000
21 #define   IRQ_MSK_REG                    0x04
22 #define   CTRL_REV_REG                   0x10
23
24 #define FIFO_BASE                        0x06000000
25 #define FIFO_CTRL_REG                    0x0400001c
26 #define DEMUX_BASE                       0x08000000
27
28 #define FIFO_SIZE                        100 * 1024 /* bytes */
29
30
31 // ***** address check masks *****
32 #define CHECK_MASK                       0x0F000000
33 #define MAX_ADDR_VIDEO_ISP               0x0FFFFFFF+32
34 #define MAX_ADDR_NOVIDEO_ISP             0x0FFFFFFF
35
```

[9] Das Dummybackend wurde zu Testzwecken geschrieben und emuliert eine beliebige Anzahl von CineCards.

```
36 // ***** FiFo flags ******
37 #define FIFO_EFA 0x01 //!port A, empty
38 #define FIFO_AEA 0x02 //!port A, almost-empty
39 #define FIFO_AFA 0x04 //!port A, almost-full-Flag, (DMA trigger)
40 #define FIFO_FFA 0x08 //!port A, full-flag (error)
41 #define FIFO_EFB 0x10 //!port B, empty-flag (never reached)
42 #define FIFO_AEB 0x20 //!port B, almost empty (used as empty)
43 #define FIFO_AFC 0x40 //!port C, almost full
44 #define FIFO_FFC 0x80 //!port C, full
45
46 #endif//_CINEBACKEND_CONF_H_
```

Die CineCard Professional benötigt keine spezielle Unterstützung des Treibers zum Laden der im Abschnitt 1.1.3 beschriebenen Blend-Koeffizienten für die Überlappungsbereiche der Projektore und Warping-Parameter (für die geometrische Verzerrung). Die Daten werden von der Userspace-Applikation über die vom IO-Control-Handler erreichbaren Funktionen `cine_readRegister` und `cine_writeRegister` (zum Lesen und Beschreiben von Kartenregistern) transferiert.
Einer umfangreicheren Anpassung bedürfen besonders der Interrupt-Handler (zur Unterstützung von bis zu fünf MPEG-2 Dekodern)und die Hardwareerkennungsroutine.

Kapitel 5

Abschließende Betrachtung

Der im Rahmen dieser Studienarbeit entstandene Linux-Hardwaretreiber für die HHI CineCard-Familien hat sich im praktischen Einsatz bewährt. Die Optimierungen gegenüber dem alten, nur für den Kernel 2.4 verfügbaren, Treiber erlauben bei der gleichzeitigen Wiedergabe mehrerer Datenströme eine mindestens 20 Prozent höhere Datenrate. Unter hoher Systemlast treten die treiberbedingten Unterläufe des Videodaten-FIFOs nicht mehr auf. Die entscheidenden Verbesserungen in Bezug auf die Latenzzeit des Interrupt-Handlers, lassen eine Verkleinerung des (hardwareseitigen) Videodaten-FIFOs zu. Die Größe des im Treiber verwendete Ringpuffers konnte ebenfalls mehr als halbiert werden.
In einem 14-tägigen Dauertest mit ständig wechselnder Systemlast traten keine Systemabstürze, (interne) Treiberfehler und Pufferunterläufe auf. Alle zu implementierenden Teile des Treibers entsprechen bezüglich des Funktionsumfanges voll den Erwartungen und übertreffen diese im Hinblick auf die erzielte Geschwindigkeit.
Die modulare Treiberstruktur reduziert die zur Anpassung des Backends für die CineCard notwendige Zeit auf ein Minimum. Testfunktionen und eine fein enstellbare Menge von Fehler- und Statusmeldungen werden die Inbetriebnahme und den Test des neuen Prototypen der CineCard erleichtern.
Das implementierte Dummy-Backend bildet eine gute Basis für die folgende intensive Tests unter voller Systemlast und zur Bestimmung der maximal erreichbaren Gesamtdatenrate pro PC-System.
Funktionale Änderungen am Treiber und an den Schnittstellen zu Userspace können ebenfalls mit wenig Aufwand vorgenommen werden. Durch die Vorbereitung des IO-Control-Handlers und des Systems zur Ausführung von Konfigurationsbefehlen, wird die Entwicklung einer flexiblen Behandlung von Dekoderfehler durch Interrupt-Skripte sehr vereinfacht.
Damit stellt das enwickelte Treibersystem eine gute Ausgangsbasis für zukünfige Entwicklungen bereit, womit sich die Entscheidung zum Neuentwurf des Treibers als richtig herausgestellt hat.

Literaturverzeichnis

[1] Wolfgang Mauerer: Linux Kernelarchitektur;
Carl Hanser Verlag M ünchen Wien, 2004, ISBN 3-446-22566-8

[2] Eva-Katharina Kunst, J ürgen Quade: Linux-Treiber entwickeln;
dpunkt.verlag GmbH, 2004, ISBN 3-89864-238-0

[3] Kern-Technik, Folge 1-20;
LINUX Magazin 2003-2005

[4] Eva-Katharina Kunst, J ürgen Quade: Kern-Technik, Folge 8;
Linux Magazin 03/04, Seite 87

[5] Eva-Katharina Kunst, J ürgen Quade: Kern-Technik, Folge 18;
Linux Magazin 02/05, Seite 86

[6] Linux Weekly News: API changes in the 2.6 kernel series;
http://www.lwn.net/Articles/2.6-kernel-api/

[7] Linux Weekly News: The future of device numbers;
http://www.lwn.net/Articles/65195/

[8] Linux Weekly News: Driver porting: Device classes;
http://www.lwn.net/Articles/31370/

[9] Mailing-Liste der Kernelentwickler;
http://www.kernel.org

[10] Kernel-API Dokumentation des DMA-Subsystems;
Dateien DMA-API.txt und DMA-mapping.txt im Verzeichnis Documentation in den Quellen des Kernels

[11] Hinweise zur neuen generischen Blockgeräte-Schicht;
Datei biodoc.txt im Verzeichnis Documentation/block in den Quellen des Kernels

Reports published by Humboldt University Berlin, Computer Science Department, Systems Architecture Group.

---

1. SAR-PR-2005-01: Linux-Hardwaretreiber für die HHI CineCard-Familie. Robert Sperling, 37 pages.