RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Christoph Georg Baumann

Hard- und Software zur Videodigitalisierung und Bildverarbeitung unter Linux

Diplomarbeit

HD-KIP-01-09

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Diplomarbeit

im Studiengang Physik

vorgelegt von

Christoph Georg Baumann

aus Speyer

im Juli 2001

Hard- und Software zur Videodigitalisierung und Bildverarbeitung unter Linux

Die Diplomarbeit wurde von Christoph Georg Baumann ausgeführt

am Kirchhoff-Institut für Physik

unter der Betreuung von

Herrn Prof. Dr. Karlheinz Meier

Zusammenfassung – In der vorliegenden Arbeit wird die Inbetriebnahme eines bereits vorhandenen Spezialrechners zur Bildverarbeitung unter dem Betriebssystem GNU/Linux erläutert. Dabei ist das Ziel den Resourcen- und Strombedarf des Systems möglichts gering zu halten, da der Rechner Hauptbestandteil in einem tragbaren Blindenhilfssystem sein soll. Bei dem Rechner handelt es sich um ein PC/104plus-System, das ein speziell entwickeltes Modul mit einem ASIC zur Bildverarbeitung enthält. Neben den nötigen Linux-spezifischen Softwareentwicklungen befaßt sich diese Arbeit auch mit Untersuchungen zum Rauschverhalten des Bildverarbeitung-ASICs und der Entwicklung eines Videodigitalisierers, um analoge CCD-Kameras an das System anschließen zu können.

Abstract – The present thesis describes the operation of a special computer for image processing under the GNU/Linux operating system. A major goal is to keep the resources and power consumption low, as the computer is intented to be an important part of a vision substitution system for blind people. The computer used is a PC/104plus system that includes a custom build module carrying the image processing ASIC. Besides the Linux-specific software development this thesis also covers investigations about the noise behaviour of the image processing ASIC and the development of a frame-grabber for using analog CCD cameras with the system.

Inhaltsverzeichnis

Motivation

Die Arbeitsgruppe *Electronic Vision(s)* am Kirchhoff-Institut beschäftigt sich mit der Untersuchung von Möglichkeiten des künstlichen Sehens. Das Hauptziel dabei ist es, ein Sehersatzsystem für blinde Menschen zu entwickeln. Dabei soll kein künstliches Auge geschaffen werden, sondern visuelle Information über den Tastsinn zugänglich gemacht werden. Prototypen eines taktilen, grafischen Displays (sog. VTD[1] [Mau98]) wurden bereits erfolgreich getestet.

Eine weitere Komponente des Projektes ist eine CMOS[2]-Kamera [Loo99] mit einem Dynamikbereich von etwa sechs Dekaden. Dieser hohe Dynamikbereich ermöglicht es auch bei stark schwankender Beleuchtung (z.B. Gegenlicht) brauchbare Bilder aufzunehmen.

Da die Bilddaten nicht direkt auf dem VTD ausgegeben werden können, müssen sie vorverarbeitet werden, um die Datenmenge zu reduzieren. Die gewählte Methode ist die der Kantenextraktion. Hierbei werden die visuellen Informationen auf die Kontrastkanten im Videobild reduziert. Die Kantenextraktion kann prinzipiell mit bereits bekannten Algorithmen (z.B. Sobel-Filter) geschehen, dies ist jedoch eine rechenintensivere Methode, die auch für diese Problemstellung keine befriedigenden Lösungen liefert. Da aber das System möglichst handlich und stromsparend sein soll, kommt dies nicht in Frage. Es wurde daher der Bildverarbeitungschip EDDA[3] [Sch99] entwickelt. Im Gegensatz zur digitalen Bildverarbeitung geschieht hier die Kantenextraktion durch einen analogen Rechner. EDDA kann so in wenigen μs ein Bildelement von 66×66 Pixeln verarbeiten.

Um nun alle Komponenten zu einem System zu integrieren, wurde auf kommerziell verfügbare Hardware zurückgegriffen. Es wurde ein PC/104 Miniatur-PC angeschafft, für den ein Erweiterungsmodul [Bli00] mit dem EDDA-Chip entwickelt wurde (siehe hierzu auch Kapitel 1.2). Als Betriebssystem des PC/104-Rechners wurde zunächst Windows benutzt. Zur Entwicklung der Software war dies bequemer zu handhaben. Da Windows eine Grafikkarte benötigt und auch schlecht skaliert werden kann (zur Reduktion des nötigen Speicherplatzes bzw. Einsparung der Festplatte), entschied man sich dazu, Linux als Betriebssystem einzusetzen. In der vorliegenden Arbeit geht es nun um die Entwicklung eines fertigen Systems aus Kamera, EDDA-Chip und PC/104-Rechner unter Linux.

[1] **V**irtual **T**actile **D**isplay
[2] **C**omplementary **M**etal-**O**xid-**S**emiconductor
[3] **ED**ge **D**etection **A**rray

Kapitel 1

Technische Voraussetzungen

In den folgenden Unterkapiteln werden wichtige Grundlagen und Begriffe kurz erläutert.

1.1 Der EDDA-Chip

Wie eingangs bereits erwähnt wurde zum Zweck der schnellen Kantenextraktion der EDDA-Chip entwickelt [Sch99]. Das Kernstück des Chips ist ein Netzwerk aus sog. *resistive fuses*. Diese vereinigen die Eigenschaften eines ohm'schen Widerstandes mit dem einer Schmelzsicherung. Innerhalb eines bestimmten Spannungsbereiches verhalten sich diese Schaltelemente wie Widerstände mit dem Widerstand R_{fuse}, wird aber eine bestimmte Spannung V_{off} überschritten, so unterbrechen sie den Stromfluß. Dieser Vorgang ist reversibel, die *resistive fuses* werden dabei nicht zerstört.

Diese *resistive fuses* sind im EDDA-Chip in einem rechtwinkligen, 2-dimensionalen Gitter angeordnet. Einem Pixel des zu verarbeitenden Bildes wird jeweils ein Knoten des Gitters zugeordnet und dort eine zum Grauwert des Pixels äquivalente Spannung V_{pixel} eingeprägt. In Abb. 1.1.a ist dieses Gitter vereinfacht dargestellt, Abb. 1.1.b zeigt das Layout des fertigen Chips.

Übersteigt die Spannung an einer *resistive fuse* den Wert V_{off}, so schaltet diese ab und zwischen den zugehörigen Pixeln wird eine Kante erkannt. Die Information über die erkannten Kanten wird schließlich über Ausleseschaltkreise aus dem Chip ausgelesen.

1.2 Die PC/104-Steckkarte

Da das Bildverarbeitungssystem sehr klein und tragbar sein sollte, wurde eine PCI[1]-Steckkarte in der PC/104-Norm entwickelt, die den EDDA-Chip trägt [Bli00]. Das PC/104-

[1] **P**eripheral **C**omponent **I**nterconnect

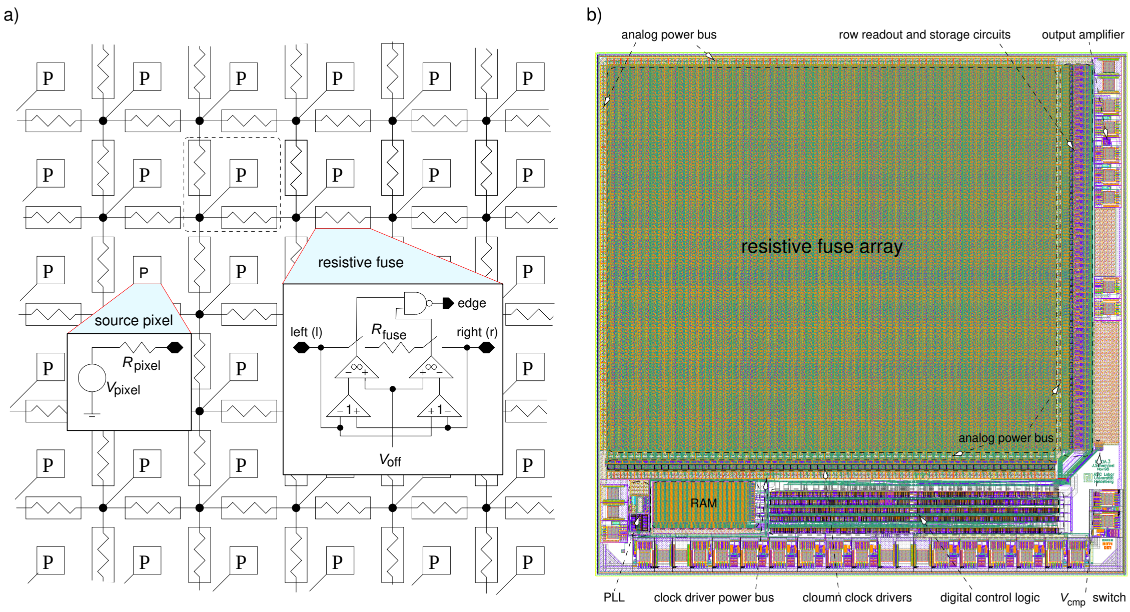

Abb. 1.1: a) Schematischer Aufbau des *resistive fuse* Gitters; b) Layout des EDDA-Chips [Sch99]

System zeichnet sich durch seine geringe Größe (9cm×9,6cm) und Leistungsaufnahme (max. 15W) aus. Abbildung 1.2 zeigt den fertig aufgebauten PC/104-Rechner.

Die EDDA-Karte (siehe Abb. 1.3) trägt neben EDDA noch weitere Komponenten, unter anderem einen FPGA[2], einen PCI-Interface-Chip[3] und RAM[4]. Der FPGA steuert als zentrale Logik die Zugriffe von und zu den einzelnen Komponenten. Durch die freie Programmierbarkeit des FPGAs kann sehr leicht zusätzliche Hardware (z.B. eine Kamera) in das System integriert werden.

1.3 Der Linux-Kernel

Das Konzept des Linux-Kernels[5] [Rus99] geht auf Unix zurück. Er wurde entwickelt, um die Vorzüge eines Unix-Systems auch auf einem PC zur Verfügung zu haben.

Neben den grundlegenden Funktionen eines Betriebssystem-Kernels, wie z.B. Speicherverwaltung, Schnittstelle zum Dateisystem und Prozessverwaltung, waren bei Linux ur-

[2] **F**ield **P**rogrammable **G**ate **A**rray

[3] Dieser stellt die Schnittstelle zwischen dem PCI-Bus des Rechners und der Hardware auf der Steckkarte dar.

[4] **R**andom **A**ccess **M**emory

[5] Es sei hier darauf hingewiesen, daß zwischen dem Linux-Kernel und dem Betriebssystem, das als „Linux“ bezeichnet wird, zu unterscheiden ist. Der Linux-Kernel wurde 1991 von Linus Torvalds begonnen und der Öffentlichkeit zur Verfügung gestellt. Die meisten zusätzlichen Programme, die ein Betriebssystem ausmachen, stammen von GNU (GNU=GNU's Not Unix). Es müßte daher eigentlich „GNU/Linux“ genannt werden.

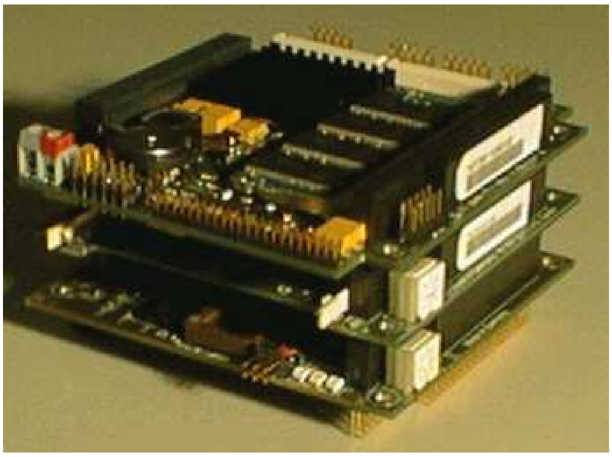

Abb. 1.2: Aufgebauter PC/104-Rechner, bestehend aus Hauptplatine, Grafik- und Netzwerkkarte [Bli00]

sprünglich auch die Gerätetreiber im Kernel enthalten. Diese rein monolithische Architektur hatte einige Nachteile (so können z.B. Gerätetreiber nicht im laufenden Betrieb geladen oder entfernt werden). Daher wurde der Kernel sehr bald modularisiert. Gerätetreiber, die nicht ständig benötigt werden, können so bei Bedarf geladen werden, oder z.B. gegen neuere Versionen ausgetauscht werden. Im folgenden werden einige Teilaspekte des Linux-Kernels, die für die vorliegende Arbeit relevant waren, genauer erläutert.

1.3.1 Die Schnittstelle zu einem Gerätetreiber

Damit ein Gerätetreiber von einem Benutzerprogramm angesprochen werden kann, wird ihm eine bestimmte Nummer zugeteilt, die sogenannte *major number*. Hat ein Gerät (z.B. eine Festplatte) mehrere Untereinheiten (z.B. Partitionen einer Festplatte), so werden diesen Untereinheiten ebenfalls bestimmte Nummern zugeordnet, die sogenannten *minor numbers*. Hat ein Gerät keine Untereinheiten, so ist die *minor number* 0. Um nun dem Benutzer/Programmierer den Zugriff auf ein Gerät bzw. den zugehörigen Treiber zu erleichtern und transparent zu machen, ist im Linux-Verzeichnisbaum das Verzeichnis `/dev` vorhanden. Dort befinden sich spezielle Dateien, die jeweils einem Zahlenpaar der *major* und *minor number* und damit dem zugehörigen Gerät zugeordnet sind. Durch lesen und schreiben in diese Dateien kann ein Benutzerprogramm mit den zugehörigen Geräten kommunizieren.

1.3.2 Speicherverwaltung

Die Aufgabe der Speicherverwaltung ist es, den physikalisch vorhandenen Speicher an die verschiedenen Benutzerprozesse zu verteilen. Dies ist kein triviales Problem. Das bisher erfolgreichste Konzept hierzu ist das des virtuellen Speichers.

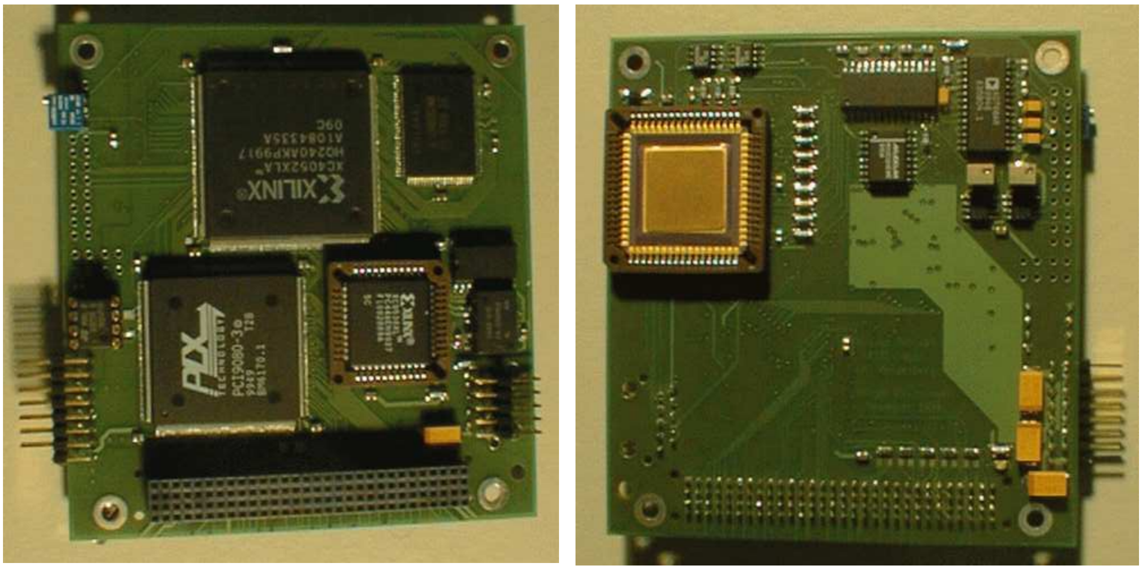

Abb. 1.3: Die PC/104-Karte mit dem EDDA-Chip. Links: Digitalseite mit FPGA und PCI-Interface-Chip. Rechts: Analogseite mit EDDA [Bli00]

1.3.2.1 Virtueller Speicher

Zusätzlich zum normalen, physikalischen RAM kommt meistens noch ein Auslagerungsspeicher auf der Festplatte hinzu, der benutzt wird, wenn das RAM nicht ausreicht. Außerdem stellt die heutige Hardware noch mehrere Ebenen von schnellen Zwischenspeichern (*Cache*) für oft benötigte Daten zur Verfügung.

Das Konzept des virtuellen Speichers bringt all dies unter einen Hut. Jedem Prozess wird zunächst ein virtueller Adressraum zugeteilt, der viel größer als der tatsächlich vorhandene physikalische Speicher sein kann. Die Speicherverwaltung führt nun Buch, wie die virtuellen Adressen dem physikalischen Speicher zugeordnet werden.

Diese Buchführung geschieht in sogenannten *page tables*, die auch direkt von der CPU[6] benutzt werden. Der Speicher wird dabei in größeren Blöcken (*pages*) organisiert. Dies macht Sinn, da der Speicherverbrauch zur Verwaltung der einzelnen Bytes jeweils größer als ein Byte wäre. Ist eine Page allerdings zu groß, wird Speicherplatz verschwendet. Man muß hier also einen Mittelweg finden, um den Speicher effizient auszunutzen. Beim PC mit Intel-Architektur wird z.B. eine Page-Größe von 4096 Bytes benutzt. Abbildung 1.4 stellt das Konzept des virtuellen Speichers schematisch vereinfacht dar.

Die *page tables* sind hierarchisch in mehreren Ebenen organisiert. Sie entsprechen den Ebenen, in denen sie auch in der Hardware implementiert sind (z.B. bei einem PC mit Intel-Architektur sind es zwei, bei Alpha CPUs drei). Linux legt standardmäßig drei Ebenen an. Unterstützt die Hardware weniger Ebenen wird die fehlende durch einen Platzhalter repräsentiert. Dies erleichtert die Portierung auf andere Rechnerarchitekturen.

[6] **C**entral **P**rocessing **U**nit

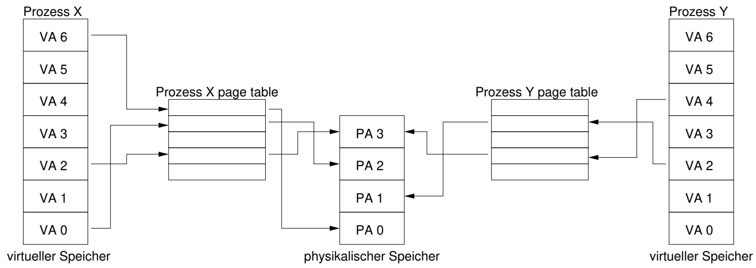

Abb. 1.4: Vereinfachtes Schema des Konzeptes des virtuellen Speichers

1.3.2.2 Das sog. *Demand Paging*

Da es viel weniger physikalischen als virtuellen Speicher gibt muß der physikalische Speicher effizient verwaltet werden. Z.B. macht es wenig Sinn bei einer Datenbankanwendung immer alle Datensätze im Speicher zu halten. Vielmehr lädt man immer nur den Datensatz in den Speicher, der gerade bearbeitet wird. Dieses Verfahren, immer nur gerade benötigte virtuelle *pages* zu laden heißt *demand paging*.

Wird z.B. in einem Programm ein Feld einer bestimmten Länge alloziert, so wird zunächst ein virtueller Speicherbereich hierfür reserviert. Greift das Programm auf Adressen innerhalb dieses Feldes zu, so prüft die Speicherverwaltung der CPU zunächst, ob zu dieser Adresse Einträge in den *page tables* existieren. Ist dies nicht der Fall, so wird eine Meldung (sog. *page fault*) an den Linux-Kernel gegeben. Die Speicherverwaltung des Linux-Kernels, muß nun prüfen, ob es sich dennoch um eine gültige Adresse handelt. Ist die Adresse ungültig, so hat das Programm versucht auf Speicherbereiche zuzugreifen auf die es nicht zugreifen darf. Dies geschieht meist, wenn z.B. das Ende eines Feldes überschritten wird. Das Programm wird dann sofort beendet (→ sog. *segmentation fault*).

Ist die Adresse aber gültig, so muß das Betriebssystem die entsprechenden Daten in den physikalischen Speicher laden bzw. physikalischen Speicher zur Verfügung stellen und die zugehörigen *page tables* aktualisieren.

1.3.2.3 Auflösen einer physikalischen Adresse

Der Anfangspunkt beim Durchlaufen der drei Verwaltungsebenen ist die Datenstruktur `struct mm_struct`. Diese existiert für jeden Benutzerprozess. Sie enthält einen Zeiger auf die zuständige Struktur **P**a**G**e **D**irectory (PGD). Jeder Prozess hat eine solche PGD; sie ist sozusagen die Top-Level Page Table. Mittels der PGD und dem ersten Segment der virtuellen Adresse (siehe Abb. 1.5) kann nun der zuständige Eintrag in der zweiten Instanz errechnet werden. Diese Instanz ist die sog. **P**age **M**id-level **D**irectory (PMD).

Die PMD wird bei Architekturen mit nur zwei Ebenen der Speicherverwaltung wie schon erwähnt durch einen Platzhalter ersetzt, der beim Kompilieren wegoptimiert wird. Mit dem zweiten Segment der virtuellen Adresse gelangt man von der PMD zur **P**age **T**abl**E** (PTE). Die PTE enthält nun die wirklichen physikalischen Adressen des RAM. Allerdings

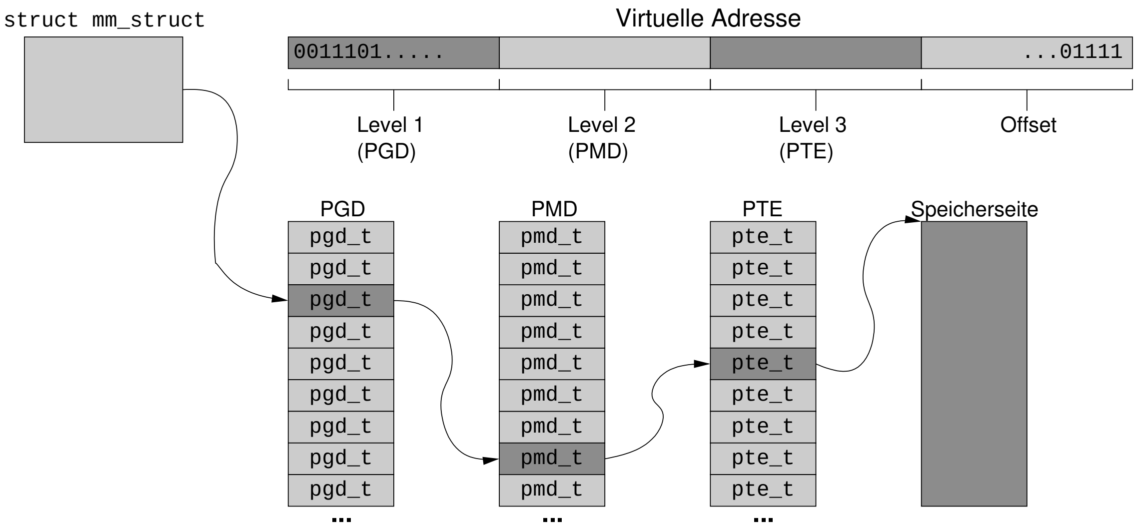

Abb. 1.5: Schematische Darstellung der Auflösung einer physikalischen Adresse anhand einer gegebenen virtuellen Adresse. Die genaue Art und Weise der eingezeichneten Segmentierung der virtuellen Adresse hängt von der jeweiligen Rechnerarchitektur ab.

jeweils nur die Startadresse der jeweiligen Speicherseite. Die Adresse innherhalb einer Speicherseite gibt das letzte Segment der virtuellen Adresse an, das in Abb. 1.5 als Offset bezeichnet ist.

1.4 Direct Memory Access – DMA

Die meisten Computer-Steckkarten sind mit eigenem Speicher ausgestattet, um Daten zwischenzuspeichern. Da ein konventioneller Zugriff auf diesen Speicher über die CPU des Rechners abgewickelt wird, ist die Zugriffszeit recht hoch und der Datendurchsatz für viele Anwendungen nicht ausreichend. Außerdem ist die CPU während der Datenübertragung quasi blockiert und kann in dieser Zeit keine anderen Daten verarbeiten.

Die Lösung stellt **D**irect **M**emory **A**ccess dar. Dabei greift die Logik auf der Steckkarte direkt auf den physikalischen Speicher des Computers zu. Um dies zu ermöglichen, muß der Logik auf der Karte die physikalische Adresse der Daten mitgeteilt werden.

1.4.1 Continuous DMA

Die erste Möglichkeit ist ein sog. DMA-Buffer. Dies ist ein physikalisch zusammenhängendes Speichersegment im RAM des Rechners (daher *Continuous DMA*), dessen Startadresse und Länge der Karte mitgeteilt wird.

Software, die Daten per *Continous DMA* zu einer Steckkarte transferiert, muß diese also zunächst in den DMA-Buffer schreiben und dann den DMA starten. Sollen Daten von der

Karte gelesen werden, müssen diese zunächst per DMA in den Buffer übertragen werden, um dann von der Software weiterverarbeitet zu werden. Die Daten sind so in den meisten Fällen doppelt im RAM vorhanden.

1.4.2 Chain List bzw. Scatter Gather DMA

Da im Laufe des Betriebs das Mapping des physikalische Speicher eines Computers immer unzusammenhängender (fragmentiert) wird, ist es mit zunehmender Laufzeit immer schwieriger einen solchen Buffer dynamisch anzulegen. Beim Starten des Betriebssystems kann zwar ein Teil des Speichers reserviert werden, dies ist aber besonders bei größeren Datenmengen sehr unflexibel. Dieses Problem umgeht man, indem die Logik auf der Karte so ausgelegt wird, daß sie fragmentierten Speicher ansprechen kann. Hierzu muß ihr allerdings dann die Reihenfolge der unzusammenhängenden Speicherblöcke (die sog. *chain list*) mitgeteilt werden. Die *chain list* besteht aus einer Liste von Startadressen und Längen der zusammenhängenden Speicherbereiche. Der Speicher, der zur Übertragung der *chain list* per *Continous DMA* an die Steckkarte benötigt wird, ist relativ klein und kann problemlos beim Starten des Betriebssystems oder später dynamisch reserviert werden.

Kapitel 2

Das Kernel-Modul

Um Hardwarekomponenten wie PCI-Karten in einem Rechner anzusprechen, benötigt man eine Treiber-Software. Um einen solchen Treiber für die EDDA-Steckkarte schnell und sicher zu erstellen wurde die WinDriver Entwicklungssoftware der Firma Jungo benutzt. Diese Software ist ursprünglich dazu entwickelt worden, um Gerätetreiber für Windows-Betriebssysteme zu erstellen, bietet aber auch Linux-Unterstützung. Wie sich aber herausstellte, unterstützt diese aber keine Datenübertragung mit *Scatter Gather DMA*[1]. Das erste Ziel war daher ein Kernel-Modul zu implementieren, das sich in die vorhandene WinDriver-Software einpassen läßt und die fehlende Funktionalität bietet.

WinDriver für Linux besteht aus einem Kernel-Modul für die Hardwarezugriffe und einer C-Quelltextbibliothek für verschiedene Standard-PCI-Hardware. In einem Benutzerprogramm, das WinDriver für Hardwarezugriffe benutzt, müssen stets Teile dieser C-Bibliothek eingebunden sein. Um Verwechslungen zu vermeiden, wird im folgenden das WinDriver Kernel-Modul mit 'WinDriver-Modul', das neu erstellte Kernel-Modul kurz als 'Kernel-Modul' und die C-Quelltextbibliothek als 'WinDriver-Software' bezeichnet.

Das erstellte Kernel-Modul wurde basierend auf Informationen aus [Rub98] und [Pom99] erstellt. Da die Entwicklung des Linux-Kernels aber ständig voranschreitet, waren auch Studien der Quelltexte der verwendeten Kernel-Version (2.2.18) nötig.

2.1 Reservierung des DMA-Buffers

Die Datenübertragung von der Benutzersoftware zur Steckkarte erfolgt zwar per *Scatter Gather DMA*, aber die *chain list* wird über einen *Continuous DMA* an die Steckkarte übertragen. Dafür ist ein DMA-Buffer nötig. Dieser ist zwar recht klein, aber es stellte sich heraus, daß die Reservierung eines solchen Buffers unter Linux nicht-trivial ist. Der schließlich gewählte Lösungsweg wurde in [Pom99] gefunden. Dabei wird dem Linux-Kernel bereits beim Starten des Rechners ein Parameter übergeben, der den benutzbaren

[1] siehe Kapitel 1.4.2

Speicher um ein Megabyte kleiner als den tatsächlich eingebauten Speicher angibt. Der Linux-Kernel benutzt und verwaltet dieses letzte Megabyte nun nicht mehr. Das Kernel-Modul muß daher diesen Speicherbereich selbst verwalten.

Um eine einzelne Steckkarte zu bedienen, benötigt man nur einen DMA-Buffer, eine Speicherverwaltung wäre nicht nötig. Da aber das Modul in späteren Projekten der Arbeitsgruppe auch mehrere Karten bedienen können soll, wurde eine Speicherverwaltung für bis zu 128 DMA-Buffer implementiert. Der reservierte Speicher wird von der selbst implementierten Speicherverwaltung in Stücke von zwei Seitengrößen eingeteilt. Diese Größe ergibt sich aus der Maximalgröße der zu übertragenden *chain list*. Auf einem Intel-PC würde zwar eine Seitengröße ausreichen, auf Architekturen mit kleineren Seitengößen wahrscheinlich aber nicht. Um auf jeden Fall auf der sicheren Seite zu sein, wurde die Größe auf zwei Seitengrößen festgelegt.

Die Verwaltung geschieht mittels eines Feldes von Datenstrukturen[2], das die physikalischen und virtuellen Adressen der einzelnen Stücke bzw. DMA-Buffer, sowie die PID[3] des zugreifenden Prozesses enthält.

Wird ein DMA-Buffer angefordert, wird das obengenannte Feld von Anfang an durchlaufen und ein Element gesucht, dessen PID-Eintrag gleich Null ist. Die zugehörigen physikalischen und virtuellen Adressen werden dann zurückgeliefert und deren Zuordnung der Speicherverwaltung des Kernels mitgeteilt, damit ein Benutzerprogramm in diese Bereiche schreiben kann. Umgekehrt wird ein DMA-Buffer wieder freigegeben, indem der zugehörige PID-Eintrag wieder auf Null gesetzt wird.

2.2 Initialisierung des Moduls

Anders als bei einem normalen C-Programm hat ein Kernel-Modul keine Funktion `main` als oberste Instanz. Es gibt vielmehr eine standardisierte Schnittstelle zum Rest des Kernels. Die Zuordnung der in der Schnittstelle definierten Funktionen zu den Funktionen eines Moduls wird im Modul in einer vordefinierten Datenstruktur festgelegt. Daneben muß jedes Kernel-Modul die Funktion `init_module` enthalten, die direkt nach dem Laden des Moduls ausgeführt wird. Diese Funktion meldet das Modul beim Kernel an. Das Modul gibt seinen Namen an, der z.B. einem Benutzer angezeigt wird, wenn dieser sich eine Liste aller geladenen Module anzeigen läßt und fordert eine *major number* an. Im Falle des erstellten Kernel-Moduls wird außerdem die Speicherverwaltung des DMA-Buffers initialisiert. Dafür wird die Startadresse des reservierten Speichers ermittelt und die Datenstruktur der Speicherverwaltung mit Einträgen der physikalischen und virtuellen Adressen ab diesem Startwert gefüllt.

[2]siehe hierzu Anhang A.1.3
[3]**P**rocess **ID**

2.3 Zusammenarbeit mit der WinDriver-Software

Um das Kernel-Modul in die Abläufe der WinDriver-Software einzufügen muß zunächst eine "Umleitung" existieren, die dafür sorgt, daß entsprechende Anfragen eines Benutzerprogramms nicht an das WinDriver-Modul gehen, sondern an das Kernel-Modul. Die Implementierung der "Umleitung" ist in A.1.1 erläutert.

Das WinDriver-Modul sorgt so lediglich dafür, daß ein Benutzerprogramm Zugriffe auf die Steckkarte durchfühen kann, während die die Speicherverwaltung betreffenden Vorgänge vom Kernel-Modul übernommen werden. Abb. 2.1 stellt diese Abläufe schematisch dar.

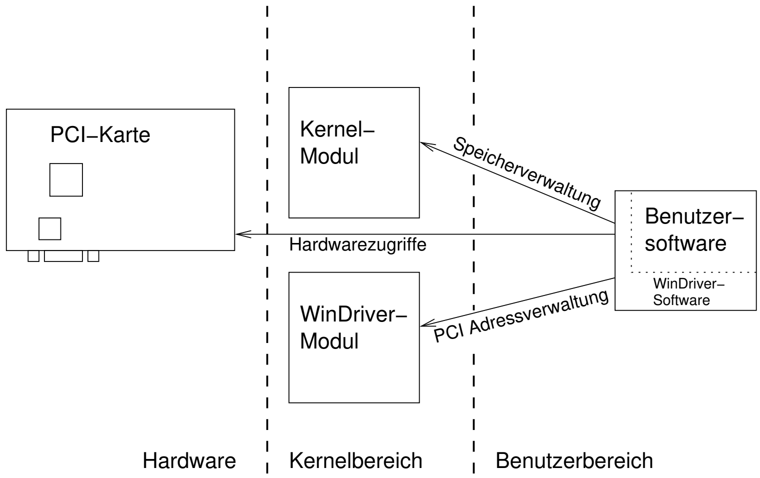

Abb. 2.1: Interaktion eines Benutzerprogramms mit dem Kernel-Modul und dem WinDriver-Modul

2.4 Interaktion mit einem Benutzerprogramm

Ein Benutzerprogramm kann auf das Kernel-Modul zugreifen, indem es die zugehörige Datei[4] im Verzeichnis `/dev` öffnet. Durch schreiben oder lesen von sog. IOCTL[5] Aufrufen in diese Datei kann mit dem Modul kommuniziert werden. Ein IOCTL Aufruf enthält drei Parameter: einen Datei-Zeiger auf die zum Modul gehörende Datei, eine Integer-Zahl (die sog. *IOCTL number*), die das gewünschte Kommando repräsentiert und einen Zeiger auf weitere Daten. Im Falle dieses Modules sind zwei Kommandos implementiert: `IOCTL_DMA_LOCK` und `IOCTL_DMA_UNLOCK`.

[4] `/dev/wd_dma_dev`

[5] **I/O ConTroL**

2.4.1 Das Kommando IOCTL_DMA_LOCK

`IOCTL_DMA_LOCK` weist das Modul an, den im dritten Parameter näher spezifizierten Speicherbereich vor der Auslagerung auf die Festplatte zu schützen. Dieser dritte Parameter, der an das Kernel-Modul übergeben wird, ist ein Zeiger auf eine Datenstruktur[6], die von der WinDriver-Software verwendet wird. Diese enthält die Variable `dwOptions`. Damit wird dem Kernel-Modul die Art des Speicherbereichs (DMA-Buffer oder Benutzerbereich) mitgeteilt, auf die sich ein `IOCTL_DMA_LOCK`-Aufruf beziehen soll. Diese Variable kann vier Werte annehmen, wobei allerdings nur zu zweien entsprechende Funktionen implementiert sind. Die fehlenden Funktionen beziehen sich auf Hardware, die nicht verwendet wurde. Die unterstützten Funktionen werden im folgenden aufgeschlüsselt.

2.4.1.1 Generierung der *chain list* (dwOptions=0)

Zunächst wird der Eintrag in der Speicherverwaltung abgerufen, der sich auf die übergebene, virtuelle Adresse bezieht. Dann wird in diesem Eintrag ein Bit gesetzt, das die Auslagerung der Daten auf die Festplatte verhindert, da sonst kein DMA-Transfer stattfinden kann. Als nächstes wird überprüft, ob die virtuelle Endadresse des Speicherbereichs zu einer gültigen physikalischen Adresse aufgelöst werden kann. Ist dies nicht der Fall sind möglicherweise noch keine entsprechenden Einträge in den *page tables*[7] vorhanden.

Dies kann vorkommen, da zum Lesen von Daten von der EDDA-Karte meist ein neu angelegter Speicherbereich benutzt wird, der zuerst nur virtuell existiert[8]. Bevor dieser Speicherbereich für DMA-Transfers benutzt werden kann muß dafür gesorgt werden, daß die Speicherverwaltung physikalischen Speicher dafür bereithält. Leider gibt es dafür keine fertige Funktion[9]. Die einzige Methode dies schnell zu erreichen ist den Speicherbereich in Schritten der Page-Größe[10] zu durchlaufen und mit einem beliebigen Wert zu initialisieren. Dies zwingt die Speicherverwaltung des Kernels dazu für diese Speicherseiten physikalischen Speicher zur Verfügung zu halten.

Ist nach dieser Prozedur die physikalische Endadresse immer noch nicht ermittelbar, beendet das Modul den Vorgang mit einer Fehlermeldung. Konnte die Adresse aufgelöst werden, dann wird auch der restliche Speicherbereich in Schritten der Page-Größe durchlaufen und die jeweilige physikalische Adresse wird ermittelt. Springt dabei von einem Schritt zum nächsten die physikalische Adresse nicht exakt um eine Page-Größe, so ist der physikalische Speicher an dieser Stelle unzusammenhängend und es wird ein entsprechender Eintrag in die *chain list* vorgenommen.

Wurde der gesamte Speicherbereich ohne Fehler durchlaufen, wird eine Erfolgsmeldung an das aufrufende Benutzerprogramm abgesetzt. Das Benutzerprogramm kann nun einen

[6] siehe hierzu Anhang A.1.2

[7] siehe Kapitel 1.3.2.1

[8] siehe auch Kapitel 1.3.2.2

[9] Laut einer Auskunft auf der Linux-Kernel Mailingliste hat Linus Torvalds die Aufnahme einer entsprechenden Funktion abgelehnt

[10] Beim PC mit Intel-Architektur 4096 Bytes

DMA-Buffer für die *chain list* anfordern und diese dort hineinschreiben.

2.4.1.2 Reservierung eines DMA-Buffers (dwOptions=1)

Wird ein DMA-Buffer angefordert so wird zunächst überprüft, ob die angeforderte Größe nicht größer als zwei Speicherseiten ist, da von der Speicherverwaltung des Kernel-Moduls nur DMA-Buffer bis zu dieser Größe verwaltet werden. Ist dies gegeben, wird die Speicherverwaltung für den DMA-Buffer-Bereich nach einer freien physikalischen Adresse durchsucht. Sobald eine Adresse gefunden wird, wird der zugehörige Eintrag in der Speicherverwaltung als belegt markiert. Schließlich wird diesem physikalischen Speicherbereich eine virtuelle Adresse zugeordnet und die Informationen über Adressen und Länge des Buffers in die WinDriver-Datenstrukturen geschrieben.

2.4.2 Das Kommando IOCTL_DMA_UNLOCK

`IOCTL_DMA_UNLOCK` weist das Modul an, die Sperrung des übergebenen Speicherbereichs rückgängig zu machen. Dabei muß zunächst unterschieden werden, ob es sich um Speicherbereiche eines Benutzers oder des DMA-Buffers handelt. Handelt es sich um eine Adresse im Benutzeradressraum, so wird lediglich das in 2.4.1.1 erwähnte Bit wieder gelöscht. Für Adressen im Bereich des DMA-Buffers wird die Adresse in der Speicherverwaltung des DMA-Buffers wieder als frei markiert, d.h. der Eintrag für die PID des Benutzerprozesses wird auf Null gesetzt.

2.5 Zeitlicher Ablauf bei Zugriffen auf eine Steckkarte

Hier soll kurz in Stichworten die zeitliche Abfolge der einzelnen Schritte bei einem DMA-Zugriff auf eine Steckkarte skizziert werden.

1. Das Benutzerprogramm alloziert Speicher, der Daten enthälten soll, die zur Karte transferiert werden, oder von ihr empfangen werden.
2. Das Benutzerprogramm übergibt die virtuelle Startadresse und Länge des Speichers an das Kernel-Modul und fordert die *chain list* dafür an.
3. Das Kernel-Modul ermittelt die *chain list* und gibt diese an das Benutzerprogramm zurück. Außerdem wird der Speicherbereich vor Auslagerung auf die Festplatte geschützt.
4. Das Benutzerprogramm berechnet den Platzbedarf zur Übertragung der *chain list* und fordert einen entsprechenden DMA-Buffer vom Modul an.
5. Das Kernel-Modul sucht einen freien DMA-Buffer und liefert dem Benutzerprogramm dessen virtuelle Startadresse zurück.

6. Das Benutzerprogramm schreibt die *chain list* in den DMA-Buffer und initiiert den DMA-Transfer, indem es direkt auf die Steckkarte zugreift. Die dafür nötigen Modifikationen in der Speicherverwaltung des Kernels erledigt das WinDriver-Modul.

7. Ist der Transfer abgeschlossen läßt das Benutzerprogramm das Kernel-Modul die verwendeten Speicherbereiche wieder freigeben.

Kapitel 3

Resultate zur Rauschunterdrückung

Das Kantenbild, das aus dem EDDA-Chip ausgelesen wird, ist durch verschiedene Quellen verrauscht. Zum einen rauscht der CCD[1]-Chip der Kamera, die zur Bildaufnahme verwendet wird, zum anderen der EDDA-Chip selbst. Im Kantenbild stellt sich dieses Rauschen als auf ein Pixel begrenzte Kantenstrukturen dar.

Das Rauschen des EDDA-Chips resultiert hauptsächlich aus sog. *fixed pattern noise* in seinem Analogteil.

3.1 Ein-Kanten Rauschen

Da der größte Teil des Rauschens vom EDDA-Chip selbst stammt, treten hauptsächlich einzelne, freistehende Kanten auf ([Sch99] S. 102ff). Ein erster Schritt zur Rauschunterdrückung ist also zu überprüfen, ob eine Kante frei steht bzw. an andere Kanten anschließt und sie dann ggf. zu löschen. Dies wurde von J. Schemmel in der ursprünglichen Software schon implementiert.

3.2 Erkennen von komplexeren Mustern

Rauschen, das schon bei der Bildaufnahme entsteht, erzeugt komplexere Kantenstrukturen als das Rauschen des EDDA-Chips. Um eine Verarbeitung in Echtzeit zu ermöglichen sollten diese Strukturen möglichst schnell erkannt und entfernt werden. Da die Verarbeitung von Bitoperationen am schnellsten erfolgt, werden die Kanteninformationen, die ein bestimmtes Pixel betreffen zunächst in einem Langwort (vier Bytes) binär codiert. Zur Fällung einer Entscheidung müssen die Kanteninformationen der 8 Pixel, die ein zu be-

[1] **C**harge **C**oupled **D**evice

trachtendes zentrales Pixel umgeben, erfasst werden. Für die Codierung dieser Information werden pro umgebendes Pixel zwei Bits verwendet. Horizontale Kante (H) = 01, vertikale Kante (V) = 10. Die horizontale Kante liegt unter dem zugehörigen Pixel, die vertikale rechts neben dem Pixel.

Als Konvention wurde festgelegt, daß die ersten zwei Bits die Information über das zentrale Pixel enthalten, dann folgen vom darunter liegenden Pixel aus die restlichen, indem das zentrale Pixel gegen den Uhrzeigersinn umlaufen wird. Zur Veranschaulichung soll Abb. 3.1 dienen.

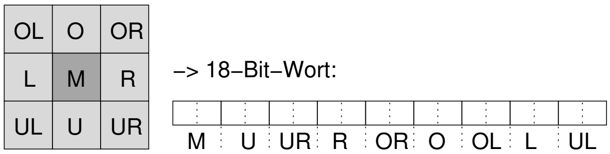

Abb. 3.1: Links sind das zentrale Pixel und seine Umgebung dargestellt, rechts das Schema nach dem die Kanteninformation dieser Umgebung in ein Bitmuster übersetzt werden.

Für ein isoliertes Pixel können 15 verschiedene Kantenmuster auftreten, die in Tabelle 3.1 aufgeführt sind. In den umgebenden 8 Pixeln können jedoch 6 Kanten gesetzt sein,

Nummer	Kantenmuster	Bitmuster
0		010000000000000000
1		100000000000000000
2		000000000001000000
3		000000000000001000
4		110000000000000000
5		100000000001000000
6		000000000001001000
7		010000000000001000
8		110000000000001000
9		110000000001000000
10		100000000001001000
11		010000000001001000
12		110000000001001000
13		010000000001000000
14		100000000000001000

Tabelle 3.1: Liste der möglichen Rauschmuster

die für diese Betrachtung nicht relevant sind bzw. nicht betrachtet werden dürfen, da sie zu einem separaten Kantenverlauf gehören können. Das Pixel rechts unter dem zentralen Pixel fällt aus diesem Grund komplett aus der Betrachtung heraus. Der Vollständigkeit und der besseren Übersicht wegen wird dieses Pixel aber im Bitmuster aufgeführt. Diese Kanten sind in Abb. 3.2 a) dargestellt. Die Kanten, die tatsächlich betrachtet werden müssen sind in Abb. 3.2 b) dargestellt.

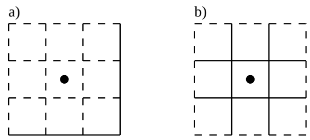

Abb. 3.2: a) Kanten die nicht betrachtet werden dürfen sind als durchgezogene Linien dargestellt. b) Die Kanten, die in die Betrachtung eingehen sind als durchgezogene Linien dargestellt.

Um eine Kantenstruktur als Rauschen zu erkennen vergleicht man nun einfach das zu einem Pixel gehörende Bitmuster nacheinander mit den 15 Bitmustern, die die unerwünschten Bitmuster repräsentieren. Wird eine Übereinstimmung gefunden, werden die betreffenden Kanten gelöscht.

Man muß allerdings bedenken, daß so auch Muster gelöscht werden, die eigentlich kein Rauschen sind. Diese Muster könnten z.B. von sehr kleinen Gegenständen oder feinen Texturen im ursprünglichen Bild stammen. Andererseits kann es wünschenswert sein solche Strukturen zu entfernen und nur die wesentlichen Kanten zu verwerten, da schließlich durch die Kantenextraktion die Informationsmenge reduziert werden soll.

3.3 Statistische Verteilung der Rauschmuster

Wie sich zeigte ist unter realitätsnahen Bedingungen (Verwendung von verschiedenen Bildern, die mit einer Digitalkamera aufgenommen wurden) das Rauschen des EDDA-Chips selbst dominant. D.h. es treten hauptsächlich Muster 0-3 aus Tabelle 3.1 auf. Dieses Ergebnis ist in Abb. 3.3 graphisch dargestellt.

Dies wurde noch zusätzlich verifiziert indem ein gänzlich weißes "Bild" verarbeitet wurde. Hier traten ausschließlich Kanten vom Typ 0-3 auf. Abb. 3.4 zeigt dies.

Bei beiden Grafiken erkennt man eine Asymmetrie bei der Zahl der erkannten einzelnen waagerechten und senkrechten Kanten. Dies läßt sich auf eine nicht ganz korrekte Justierung des EDDA-Chips zurückführen. Prinzipiell läßt sich die Zahl der durch Rauschen erzeugten waagerechten und senkrechten Kanten aneinander angleichen.

Insgesamt liegt der Anteil von Einzelkanten-Rauschen bei etwa 85%. Man muß daher folgern, daß ein Filter für komplexe Kantenmuster nicht sinnvoll ist, da der höhere Aufwand keine wesentlich höhere Bildqualität erwarten läßt. In Abb. 3.5 kann man dies per Augenmaß bestätigt finden. Durch die Gegebenheiten der Software war es allerdings nicht möglich ohne großen Aufwand für die Einzelbilder den gleichen Kanten-Datensatz zu verwenden. Sie stammen also aus verschiedenen EDDA-Durchläufen und unterscheiden sich in feinen Details.

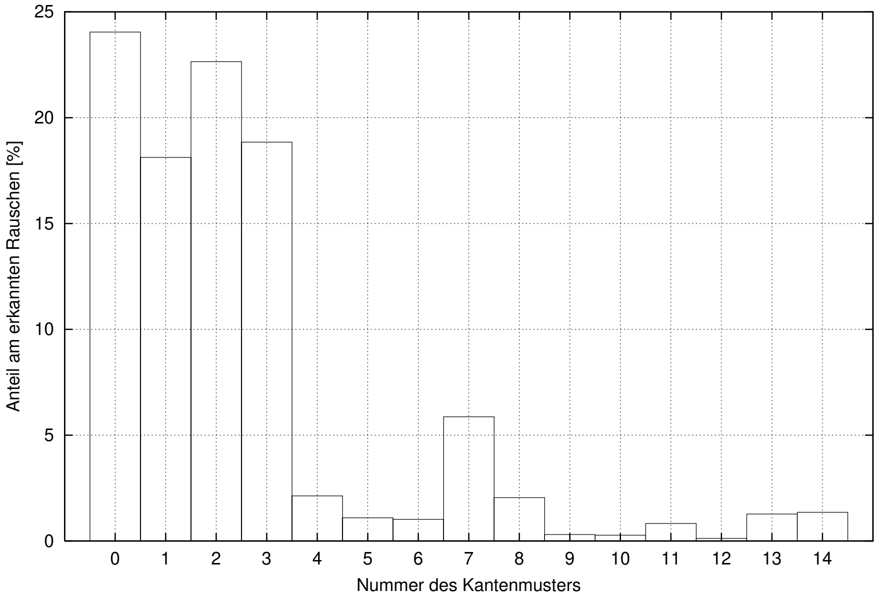

Abb. 3.3: Verteilung der erkannten Rauschmuster. Es wurden drei verschiedene Bilder mehrfach verarbeitet und insgesamt 7227 Kanten herausgefiltert. Die Nummern an der x-Achse entsprechen denen in Tabelle 3.1

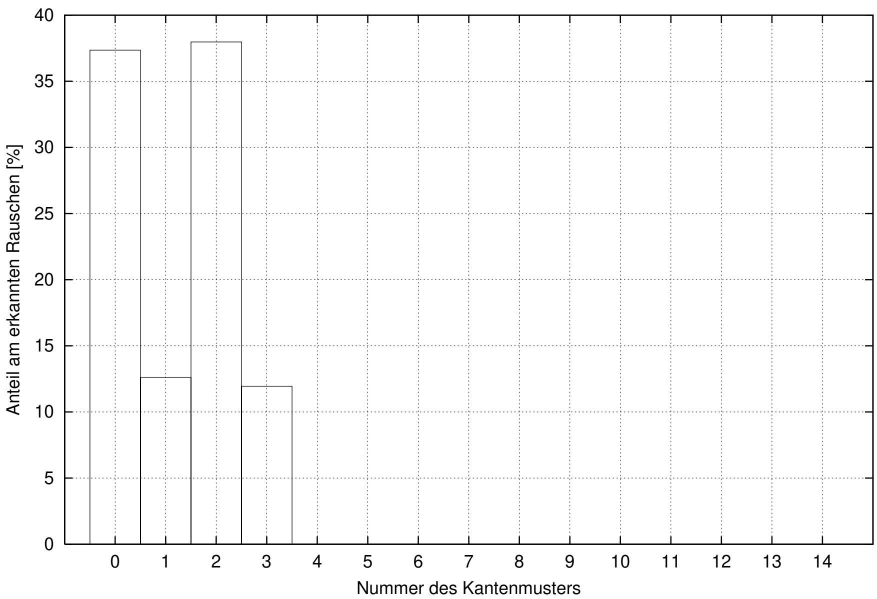

Abb. 3.4: Verteilung der erkannten Rauschmuster bei der Verarbeitung eines leeren Bildes. Die Nummern an der x-Achse entsprechen denen in Tabelle 3.1

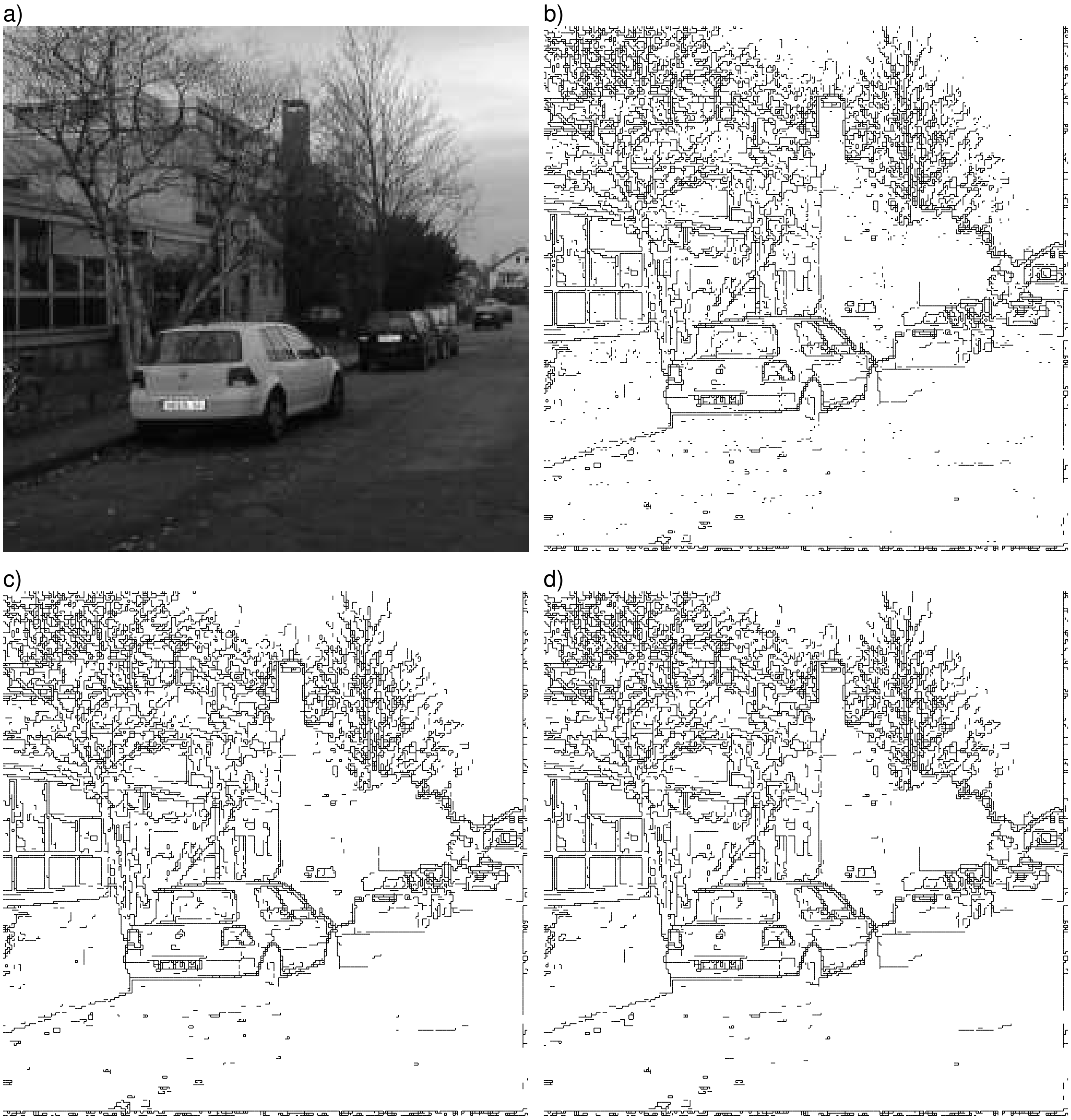

Abb. 3.5: a) Eingabebild b) Kantenbild ohne Rauschunterdrückung c) Filterung von Einzelkanten d) Filterung komplexer Kantenmuster

Kapitel 4

Der Videodigitalisierer

Da das Ziel war, ein komplettes System mit Kamera in Betrieb zu nehmen, wurde entschieden, eine bereits vorhandene analoge CCD-Kamera zu verwenden. Die ursprünglich vorgesehene Kamera [Loo99] erfüllte leider noch nicht ganz die Erwartungen an das System.

Um die analoge Kamera zu verwenden, mußte allerdings ein Videodigitalisierer implementiert werden. Dieser sollte bereits vorhandene Hardware auf der EDDA-Karte benutzen. Zusätzlich nötige Bauteile wurden auf einer kleinen Zusatzplatine untergebracht.

4.1 Das Abtrennen der Synchronisationssignale

Um das Digitalisieren des analogen Videosignals zu vereinfachen, wurde ein sog. Sync Separator verwendet. Dieser Baustein trennt die Signale für die vertikale und horizontale Synchronisation vom analogen Signal ab und gibt sie als Logikpegel aus. Zusätzlich liefert er noch ein Signal, um die geraden bzw. ungeraden Halbbilder zu erkennen. Konkret wurde das Modell EL4583C der Firma élantec verwendet [EL96].

Der Baustein ist zusammen mit seiner externen Beschaltung auf einer separaten Platine untergebracht (siehe Abb. 4.1), die auf eine Stiftleiste auf der EDDA-Karte aufgesteckt wird. Ein Schaltplan findet sich im Anhang B.1.

Die digitalen Ausgangssignale des Sync Separators werden direkt an freie Anschlüsse des FPGA geführt. Das analoge Videosignal gelangt zum Eingang des ADC[1] auf dem EDDA-Board.

[1] **A**nalog **D**igital **C**onverter

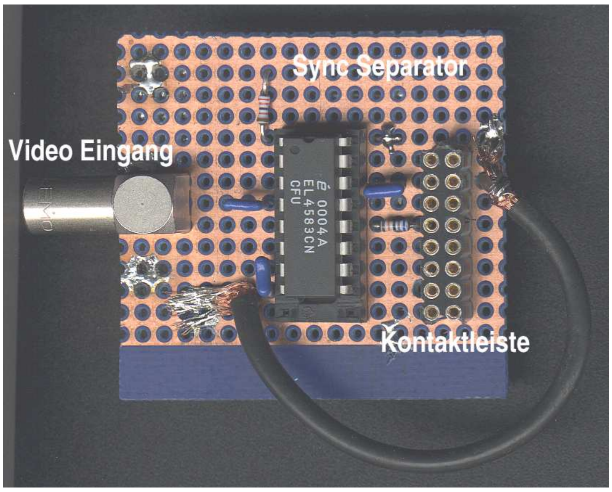

Abb. 4.1: Der Sync Separator zusammen mit seiner äußeren Beschaltung auf einer Lochrasterplatine aufgebaut. Links befindet sich eine Anschlußbuchse für das Videosignal, mittig befindet sich der Baustein selbst und rechts sieht man die Kontaktleiste, mit der man die Platine auf die EDDA-Karte aufstecken kann.

4.2 Die Synchronisationssignale

In einem analogen Videosignal sind mehrere Synchronisationssignale enthalten: ein Signal, das die sog. vertikale Austastlücke[2] (VSYNC) anzeigt, das Signal, das den Beginn einer Zeile (HSYNC) anzeigt und eines, das den Typ (gerade oder ungerade) des nächsten Halbbildes[3] anzeigt. (ODD/EVEN).

Der Sync Separator liefert die Logikpegel für diese Signale mit etwa 0,8μs Verzögerung, was beim Timing zu berücksichtigen ist. In Abb. 4.2 sind die verschiedenen Signale zum Zeitpunkt des Wechsels von einem Bild zum nächsten schematisch dargestellt.

4.3 Das VHDL-Modul

Um das Timing zur Steuerung des ADC und der nötigen RAM-Zugriffe zu bewältigen, wurde das zusätzliches VHDL[4]-Modul "Videc" für den FPGA implementiert. Seine Hauptaufgabe besteht darin, die Synchronisationssignale des Videosignals auszuwerten, zur richtigen Zeit den Wert des ADCs auszulesen und diesen in das RAM auf der EDDA-Karte

[2]Die Austastlücke resultiert aus der Tatsache, daß bei einer Kathodenstrahlröhre die Ablenkeinheit eine gewisse Zeit benötigt, um den Elektronenstrahl vom Ende des Bildes wieder zum Anfang zu bewegen.

[3]Bei allen Videostandards wird das Bild in zwei Halbbildern kodiert. Dabei wird im ersten Halbbild nur jede zweite Zeile angezeigt, im zweiten Halbbild dann die fehlenden Zeilen, so daß sich ein vollständiges Bild ergibt. Dies erhöht die Bildwiederholfrequenz und vermindert somit das vom menschlichen Auge wahrgenommene Flimmern.

[4]**V**HSIC[5]**H**ardware **D**escription **L**anguage

[5]**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit

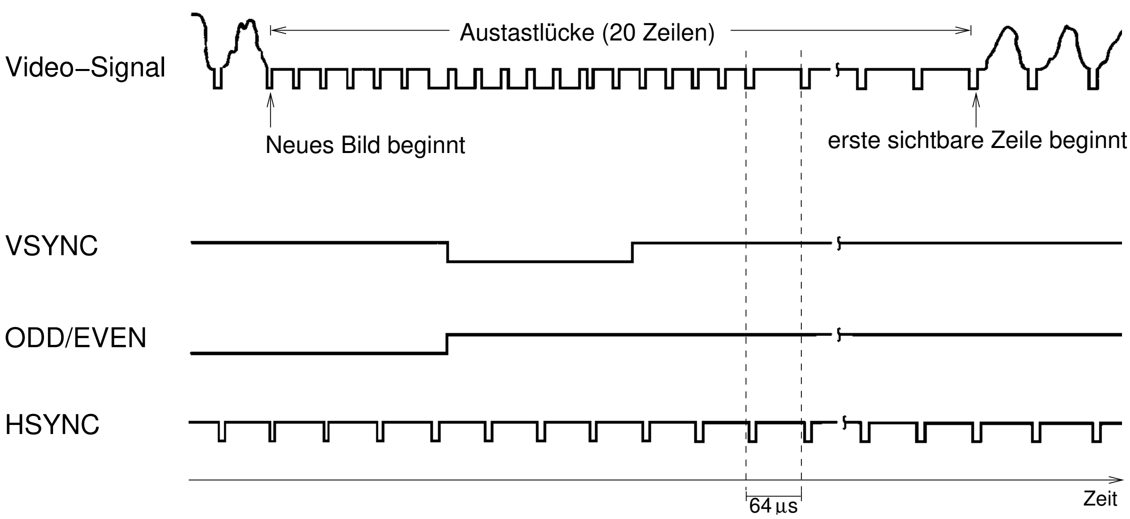

Abb. 4.2: Die Synchronisationssignale: Ganz oben ist das analoge Eingangssignal skizziert, darunter ist das digitale VSYNC-, HSYNC und ODD/EVEN-Ausgangssignal des Sync Separators dargestellt.

zu schreiben.

Da der ADC einen 12 Bit breiten Ausgang hat, EDDA aber analoge Werte erwartet, die erst mit einem 16-Bit DAC[6] erzeugt werden müssen und jeweils zwei davon in einem 32-Bit Wort gelesen werden, mußten mehrere Pipeline- und Arithmetikstufen implementiert werden. Der ADC-Ausgang wird zunächst mit einem einstellbaren 4-Bit Wert multipliziert, um einen 16-Bit Wert zu erhalten. Dies geschieht fortwährend, auch wenn der Wert nicht weiterverarbeitet wird, und das Resultat der Multiplikation wird in ein internes Register geschrieben. In der nächsten Pipeliningstufe wird ein einstellbares Offset zum Ergebnis der Multiplikation addiert und das Ergebnis in ein weiteres Register geschrieben. Dieses Register wird nun erst ausgelesen, wenn tatsächlich ein Grauwert des Videosignals benötigt wird. Immer zwei dieser Werte werden zu einem 32-Bit Wort kombiniert und in das RAM geschrieben.

Das Timing zum Digitalisieren der Bildpunkte erfolgt in einer sog. *Finite State Machine*[7]. Diese hat mehrere definierte Zustände, die durch einen Zustandsspeicher (z.B. ein internes Register) repräsentiert werden. Abhängig vom jeweiligen Zustand und eventuellen Eingangssignalen wechselt die *State Machine* in einen anderen Zustand oder verbleibt im momentanen Zustand. Bei einem Zustandswechsel wird der neue Zustand im Zustandsspeicher festgehalten und evtl. Ausgangssignale ausgelöst. In Abb. 4.3 ist der Aufbau des VHDL-Moduls und seine Anbindung an bestehende VHDL-Module schematisch dargestellt.

Nach dem Initialisieren durch ein externes Reset-Signal wartet sie zunächst auf einen Wechsel des VSYNC-Signals von 0 auf 1. Danach wird eine bestimmte Anzahl von HSYNC-

[6]**D**igital **A**nalog **C**onverter
[7]dt. endlicher Zustandsautomat

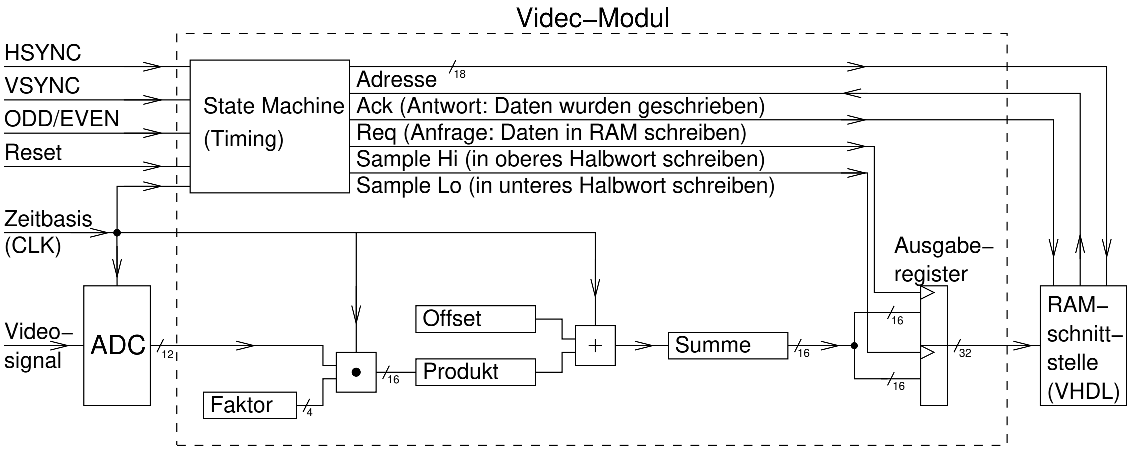

Abb. 4.3: Schematisches Blockdiagramm des VHDL-Moduls "Videc"

Signalen gewartet, bis die erste sichtbare Bildzeile beginnt. Die Parameter für das Timing können mittels mehrerer Register im VHDL-Modul eingestellt werden. Sie sind in Tabelle 4.1 aufgeführt. Zu Beginn jeder Bildzeile muß noch eine gewisse Zeit (die sog. *back porch*) gewartet werden, bis das eigentliche Bildsignal beginnt. Zur Bestimmung dieser Zeit wird die globale Zeitbasis (CLK) des FPGAs benutzt.

Name im VHDL-Code	Beschreibung	Breite [Bit]	Adresse
bpv	Dauer der *back porch* in CLK-Zyklen	16	0000
hsv	Zahl der HSYNCs bis zur ersten aktiven Zeile (=Dauer der Austastlücke)	16	0001
ppl	Zahl der digitalisierten Pixel pro Zeile	16	0010
wait_v	Zahl der CLK-Zyklen zwischen dem Digitalisieren zweier Pixel	16	0011
adc_offset	Summand, der zum multiplizierten ADC-Wert hinzuaddiert wird	16	0100
mfact	Faktor, mit dem der ADC-Wert multipliziert wird	4	0101

Tabelle 4.1: Register im "Videc"-Modul zum Justieren der Timingparameter

Nach dem Ende der *back porch* wird damit begonnen Bilddaten in das RAM zu schreiben, wobei zwischen zwei Pixeln die im Register `wait_v` angegebene Anzahl von CLK-Zyklen abgewartet wird.

Überschreitet der interne Zähler für die Anzahl der digitalisierten Pixel den Wert, der im Register `ppl` angegeben ist, so wird der Zähler für die laufende Zeile um eins erhöht, der Pixelzähler genullt und das nächste HSYNC-Signal bzw. die nächste Zeile abgewartet. Erreicht der Zeilenzähler schließlich den Wert 575 (Anzahl der sichtbaren Zeilen pro Bild laut PAL[8]-Standard [ITU98], so ist das Video-Bild komplett digitalisiert. Die verschiedenen Zähler werden wieder auf ihre Startwerte gesetzt und die *State Machine* wartet nun auf den Beginn des nächsten Bildes.

4.4 Ergebnisse

4.4.1 Simulation

Das VHDL-Modul 'Videc' wurde mit dem Simulationsprogramm ModelSim auf seine Funktionalität hin überprüft. In den Abbildungen 4.4 und 4.5 sind die kritischen Vorgänge dargestellt.

In Abb. 4.4 ist dargestellt, wie der Anfang eines neuen Halbbildes erkannt und dann die erste sichtbare Zeile abgewartet wird.

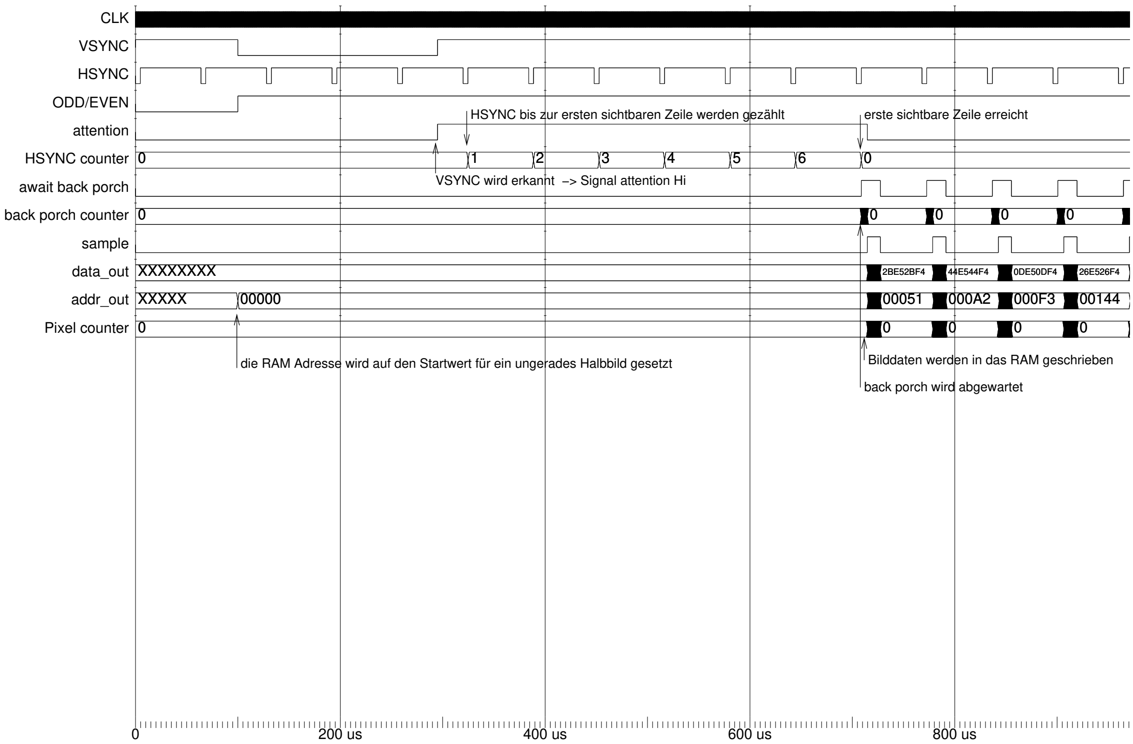

Abb. 4.4: Simulation der Abläufe beim Erkennen des Anfangs eines neuen Halbbildes

Abbildung 4.5 zeigt einen Detailausschnitt der Abläufe beim schreiben der Bilddaten in das RAM. Man erkennt wie nach Ablauf der *back porch* die ersten Pixeldaten in das 32-Bit Register geschrieben werden.

[8]**P**ase **A**lternating **L**ine

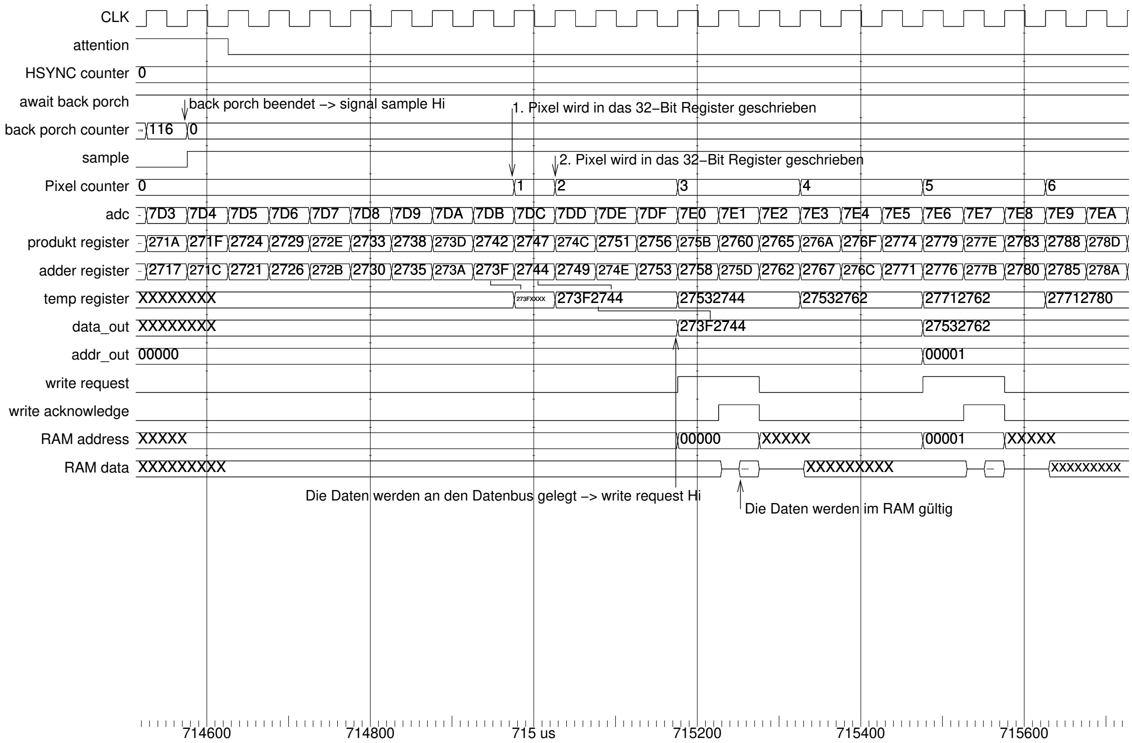

Abb. 4.5: Simulation der Abläufe am Beginn einer Bildzeile

Die Simulation ließ keine Fehler im VHDL-Code erkennen.

4.4.2 Hardware

Zunächst wurde der Sync Separator getestet. Wie Abb. 4.6 zeigt stimmt das Verhalten der Hardware sehr gut mit dem in Abb. 4.2 gezeigten Timing überein.

Wie sich leider herausstellte, hatte die einzige für diesen Zweck taugliche FPGA-Karte einen Hardwaredefekt. Das in Abb. 4.7 gezeigte, aufgenommene Bild ist daher recht undeutlich. Es zeigt aber, daß der Videodigitalisierer prinzipiell funktioniert.

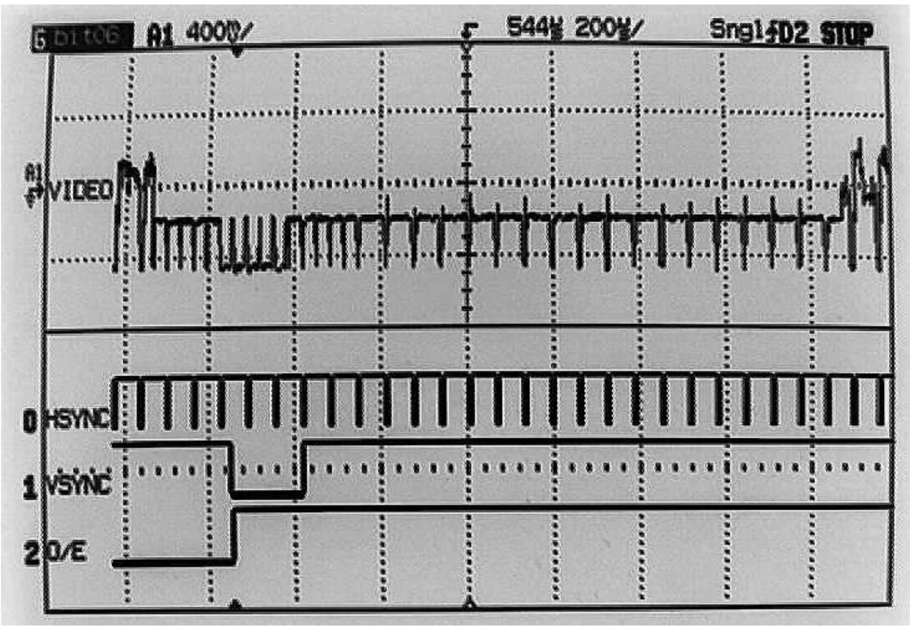

Abb. 4.6: Aufnahme vom Oszilloskop. Es werden das Video-, das VSYNC, das HSYNC und das ODD/EVEN-Signal dargestellt. Die Zeitauflösung beträgt 200μs/DIV.

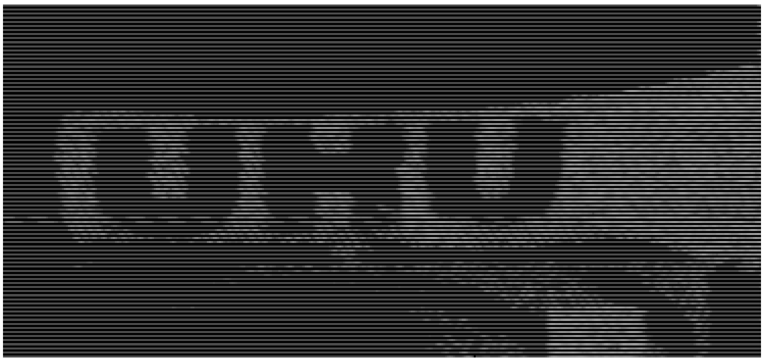

Abb. 4.7: Ein mit dem Videodigitalisierer aufgenommenes Bild

Kapitel 5

Ausblick

Die vorliegende Arbeit ermöglicht nun den Betrieb des EDDA-Chips unter Linux. So wird es ermöglicht bei dem PC/104-System auf die Grafikkarte zu verzichten. Eine weitere Einsparmöglichkeit stellt die Festplatte des Rechners dar. Diese könnte durch ein passendes Flash-Medium ersetzt werden, auf dem ein minimales Linux installiert wird. Durch den Verzicht auf eine Festplatte würde das System auch weitgehend stoßunempfindlich werden.

Die Verwendung von Linux als Betriebssystem hat auch einen finanziellen Vorteil, da auch bei einer Produktion in größerer Stückzahl keine Lizenzkosten anfallen.

Der entwickelte Videodigitalisierer erlaubt den Einsatz von handelsüblichen CCD-Kameras als Bildquelle. Die kompakte Bauweise und geringe Leistungsaufnahme des Gesamtsystems macht es neben dem Einsatz als Hilfssystem für blinde Menschen auch für den Bau mobiler Roboter interessant.

Leider war es bis zur Fertigstellung der Arbeit nicht möglich, den Videodigitalisierer mit voll funktionstüchtiger Hardware zu testen. Es wurde aber gezeigt, daß er prinzipiell funktioniert.

Wie in der Arbeit bereits angedeutet, wird das erstellte Kernel-Modul auch in zukünftigen Projekten Verwendung finden. Die parallel zu dieser Arbeit entwickelte FPGA-Karte 'Darkwing' wird beim Betrieb unter Linux das Kernel-Modul verwenden.

Anhang A

Software

Die erstellte Software befindet sich im CVS-Repository der Arbeitsgruppe Electronic Vision. Programmteile, die spezifisch für EDDA sind sind im Zweig `project/edda_dma` abgelegt. Teile, die auch für andere Projekte verwendet werden können sind im Zweig `project/common` abgelegt.

A.1 Software im Zweig project/common

A.1.1 Veränderungen an den Quelltexten von WinDriver

Um die Aufrufe zur Anforderung der *chain list* und des DMA-Buffers auf das selbst erstellte Kernel-Modul umzuleiten, wurde die Datei `windriver/windrvr.h` verändert. Die Veränderungen sehen wie folgt aus:

```
/*Get our DMA routines in there*/
#ifdef LINUX
#define WD_DMALock(h,pDma)  ioctl_dma_lock(h,pDma)
#else
#define WD_DMALock(h,pDma)\
    WD_FUNCTION(IOCTL_WD_DMA_LOCK, h, pDma, sizeof (WD_DMA), FALSE)
#endif

#ifdef LINUX
#define WD_DMAUnlock(h,pDma)  ioctl_dma_unlock(h,pDma)
#else
#define WD_DMAUnlock(h,pDma)\
    WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, sizeof (WD_DMA), FALSE)
#endif
/*That's all*/
```

Die Funktionen `ioctl_dma_lock` und `ioctl_dma_unlock` sind in der Datei `wd_dma.c` implementiert.

A.1.2 Deklarationen in wd_dma_dev.h

Hier sind die wesentlichen Deklarationen in der Datei `wd_dma_dev.h` aufgeführt. Die eigentliche *chain list* wird in ein Feld des Typs `WD_DMA_PAGE` geschrieben. Dieses Feld ist wiederum Bestandteil der Datenstruktur `WD_DMA`. Diese Datenstruktur enthält außerdem Einträge für die Startadresse der Daten, die übertragen werden sollen (`pUserAddr`), deren Länge in Bytes (`dwBytes`), Optionen für das Kernel-Modul (`dwOptions`) und die Anzahl der Einträge in der *chain list* (`dwPages`).

Auszug aus `wd_dma_dev.h`:

```
typedef unsigned long ULONG;
typedef unsigned short USHORT;
typedef unsigned char BYTE;
typedef void *PVOID;
typedef PVOID HANDLE;
typedef ULONG DWORD;
typedef ULONG BOOL;
typedef USHORT WORD;

enum { WD_DMA_PAGES = 256 };
enum { DMA_KERNEL_BUFFER_ALLOC = 1 };
enum { DMA_KBUF_BELOW_16M = 2 };
enum { DMA_LARGE_BUFFER   = 4 };

typedef struct
{
    PVOID pPhysicalAddr;    // physical address of page
    DWORD dwBytes;          // size of page
} WD_DMA_PAGE;

typedef struct
{
    DWORD hDma;             // handle of dma buffer
    PVOID pUserAddr;        // beginning of buffer
    DWORD dwBytes;          // size of buffer
    DWORD dwOptions;        // allocation options:
    DWORD dwPages;          // number of pages in buffer
    WD_DMA_PAGE Page[WD_DMA_PAGES];
} WD_DMA;
```

A.1.3 Speicherverwaltung des DMA-Buffers

Zur Verwaltung des DMA-Buffers ist wie bereits erwähnt eine eigene Speicherverwaltung implementiert worden. Diese besteht aus einem Feld vom Typ `ch_mm_item` und der Länge `CH_NO_BUFFERS`[1]. Die Struktur `ch_mm_item` enthält die physikalische und virtuelle Startadresse (`ch_phys_start`, `ch_virt_start`) eines Speicherbereichs, sowie die PID[2] (`ch_pid`) des Benutzerprozesses, der den Speicherbereich benutzt. Ist `ch_pid` gleich null, so ist der Speicherbereich nicht belegt.

Dies ist der entsprechende Ausschnitt aus `wd_dma_dev.c`:

```
/*data structure for memory management of the DMA buffer*/
/*The DMA buffer will be chopped in chunks of 8k*/
typedef struct {
  /*phys. start address of a part of the buffer, used by
    a certain process*/
  unsigned long ch_phys_start;
  /*virt. start address of a part of the buffer, used by
    a certain process*/
  unsigned long ch_virt_start;
  /*PID of the process using the buffer. If PID==0 this
    part of the buffer is unused*/
  unsigned int ch_pid;
} ch_mm_item;

/*allocate entries to handle 1M in 8k chunks*/
ch_mm_item ch_mm[CH_NO_BUFFERS];
```

A.1.4 Installation des Kernel-Moduls

Zunächst muß sichergestellt werden, daß die Quelltexte der Kernel-Version, die verwendet werden soll im Pfad `/usr/src/linux` zu finden sind. Die Quelltexte des Moduls befinden sich im CVS-Repository der Arbeitsgruppe Electronic Vision im Zweig `project/common`.

Hat man diesen Zweig ausgecheckt, so findet man dort ein Unterverzeichnis namens `kernel_module`. Nach dem Wechsel in dieses Verzeichnis kann man mit dem Befehl `make` das Modul kompilieren. Mit dem Befehl `make install` wird es installiert. Hierzu benötigt man allerdings Administratorrechte.

Das Modul sollte nun auch bereits geladen worden sein. Dies kann man mit dem Befehl `lsmod` überprüfen. Um das Modul anzusprechen muß noch die zugehörige Datei im Verzeichnis `/dev` erzeugt werden. Dies geschieht mit dem Befehl `mknod /dev/wd_dma_dev c 100 0`. Das Modul ist nun einsatzbereit.

[1] Der Wert `CH_NO_BUFFERS` ist in `wd_dma_dev.h` definiert

[2] **P**rocess **ID**

A.2 Software im Zweig project/edda_dma

A.2.1 Das Programm edda_dma_test

Das Programm 'edda_dma_test' dient zum Testen des Datentransfers von und zur EDDA-Karte. Nach dem Aufrufen führt es zunächst eine Reihe von normalen Lese- und Schreibzugriffe auf das RAM auf der EDDA-Karte aus. Dabei wird jeweils überprüft, ob die übertragenen Daten korrekt sind.

Danach wird eine Reihe von DMA-Zugriffen mit zufälligen Daten und von zufälliger Größe ausgeführt. Auch dabei wird überprüft, ob die Daten korrekt übertragen wurden.

Der anschließende letzte Test ermittelt die Übertragungsgeschwindigkeit beim DMA-Transfer. Dazu wird eine große Datenmenge mehrfach gelesen bzw. geschrieben, um eine zuverlässigere Messung zu erhalten. Mit hilfe der Systemuhr des Rechners wird die benötigte Zeit ermittelt und daraus dann die Transferrate errechnet.

A.2.2 Das Programm eddatest

Das Programm 'eddatest' dient zum Testen der Funktionalität des EDDA-Chips. Wird es ohne zusätzliche Parameter aufgerufen, so öffnet es die Datei `testbild.bmp` im momentanen Verzeichnis und läßt das enthaltene Graustufenbild von EDDA verarbeiten. Das eingelesene Bild wird auf dem Bildschirm dargestellt.

Statt der Datei `testbild.bmp` kann eine andere Datei in der Kommandozeile angegeben werden. Diese muß aber im BMP-Format vorliegen. Wird als erster Parameter der Kleinbuchstabe 's' angegeben, so wird das Bild mit der Simulation des EDDA-Chips verarbeitet.

Das Programm kann leicht so verändert werden, daß es die extrahierten Kanten auf dem Bildschirm, als Bitmapdatei, LaTeX- oder xfig-Grafik ausgibt.

A.2.3 Das Programm framegrabber

Bei dem Programm 'framegrabber' handelt es sich um ein Testprogramm, um die Funktionalität des Videodigitalisierers zu überprüfen. Es lädt den FPGA mit dem entsprechenden Bitstream und liest dann die digitalisierten Videodaten aus dem RAM der EDDA-Karte aus und stellt sie auf dem Bildschirm dar.

A.2.4 Das Programm webedda

Für die Präsentation der Möglichkeiten des EDDA-Chips über das WWW wurde das Programm 'webedda' entwickelt. Es akzeptiert wie 'eddatest' den Namen einer Eingabebzw. Ausgabedatei als Kommandozeilenparameter. Werden keine Parameter dafür angegeben, so ließt es standardmäßig die Datei `testbild.bmp` ein und schreibt das Resultat in die Datei `kantenbild.bmp`. Da es keine Anzeige auf dem Bildschirm öffnet, kann es für WWW-Anwendungen benutzt werden.

Anhang B

Schaltpläne

B.1 Der Sync Separator

Der Sync Separator ist zusammen mit seiner äußeren Beschaltung auf einer kleinen Platine untergebracht. Mittels einer Kontaktleiste wird diese Platine auf eine Reihe Pfostenstecker auf der EDDA-Karte aufgesteckt. Abb. B.1 zeigt die verwendete Beschaltung des Sync Separators.

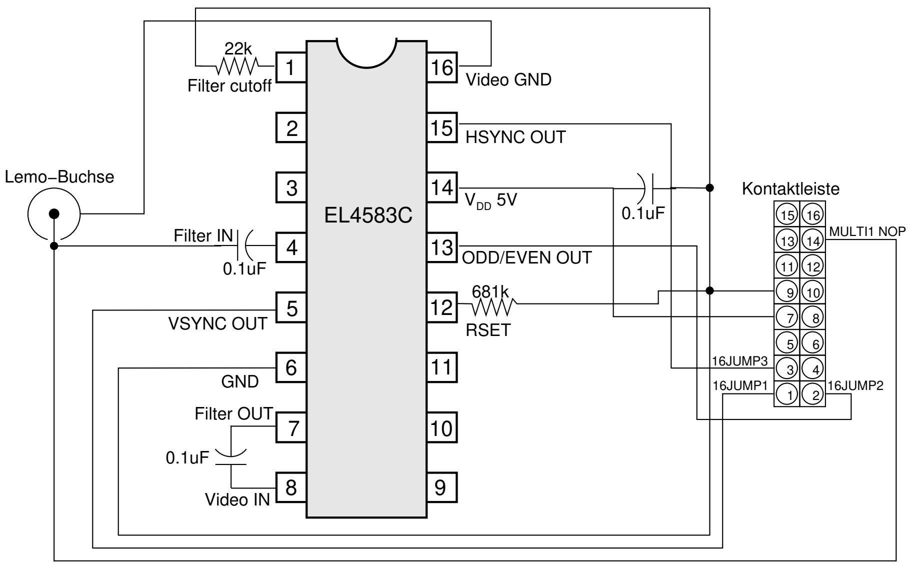

Abb. B.1: Äußere Beschaltung des Signal-Splitter Bausteins; Aufsicht auf die Bestückungsseite (Die Bezeichnungen an der Kontaktleiste entsprechen denen im Schaltplan für die EDDA-Karte)

Die für den Kameraanschluß verwendeten Pins auf der EDDA-Karte sind in Abb. B.2

markiert. Neben je einem Pin für GND[1] (Pin 6) und +5V (Pin 5) wurden noch vier weitere benötigt. Diese werden in Tabelle B.1 zusammen mit ihren Benennungen in Abb. B.2, ihrer Bezeichnung im Schaltplan[2], ihr Signalname im VHDL-Code sowie einer Beschreibung ihrer Verwendung aufgelistet.

Beschreibung	Schaltplan	VHDL-Code	Abb. B.2
Video-Signal	MULTI1 NOP	–	c
HSYNC-Signal	16JUMP3	jump(3)	k
VSYNC-Signal	16JUMP1	jump(1)	n
ODD/EVEN-Signal	16JUMP2	jump(2)	l

Tabelle B.1: Bezeichnungen der für die Kamera benutzten Pins. Hinweis: Das Video-Signal hat keinen Bezeichner im VHDL-Code, da es analog ist, und daher nicht direkt am FPGA anliegt.

[1]=Ground
[2][Bli00] S. 29 ff.

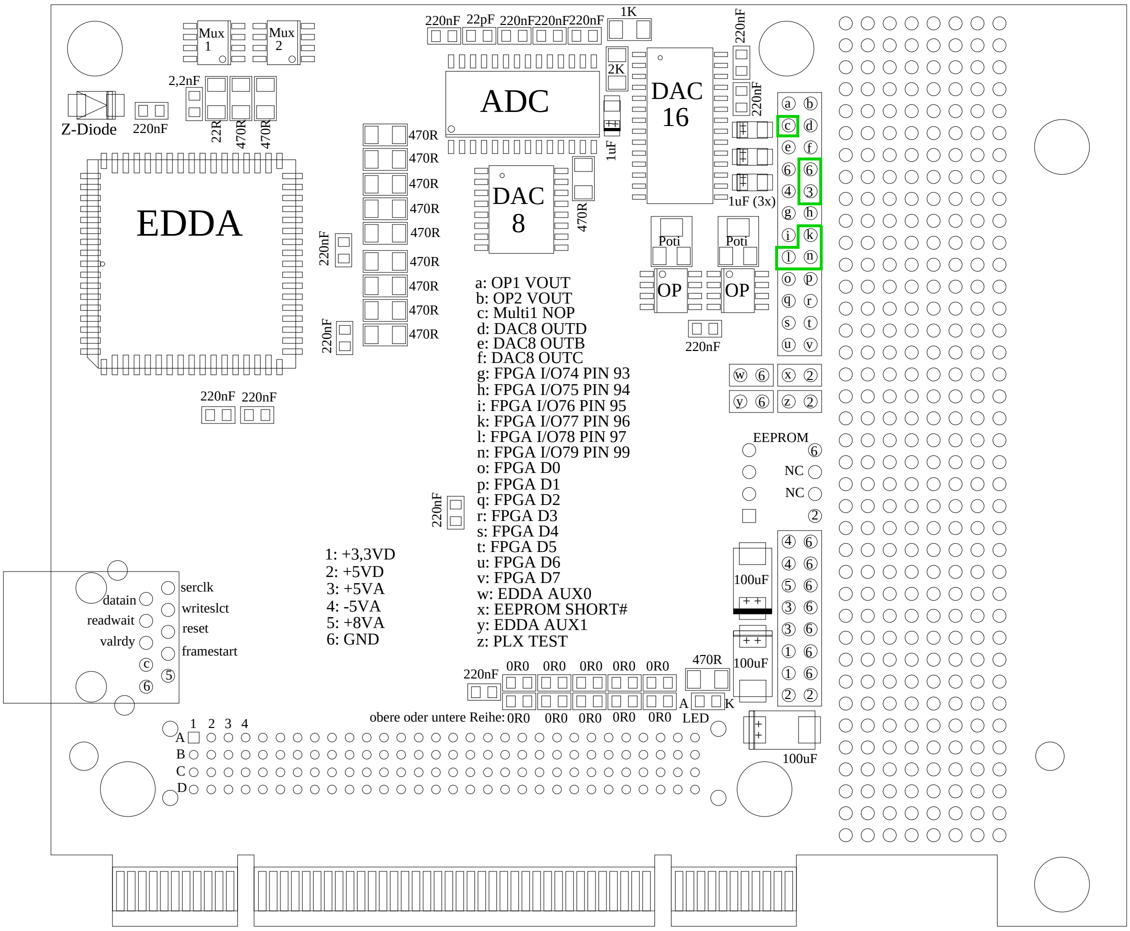

Abb. B.2: Bestückungsplan der EDDA-Steckkarte, Analogseite [Bli00]. Die verwendeten Anschlußpins 5,6,c,k,l und n sind markiert.

Literaturverzeichnis

[Bli00] Blinzinger, Holger: *Aufbau eines kompakten Bildverarbeitungsrechners für ein taktiles Blindenhilfssystem.* Diplomarbeit, Universität Heidelberg, 2000.

[EL96] Elantec Inc.: *EL4583C Sync Separator*, January 1996. Rev. B.

[ITU98] ITU Radiocommunication Assembly: *Recommendation ITU-R BT.470-6, Conventional Television Systems*, 1998.

[Loo99] Loose, Markus: *A Self-Calibrating CMOS Image Sensor with Logarithmic Response.* Dissertation, Universität Heidelberg, 1999.

[Mau98] Maucher, Thorsten: *Aufbau und Test eines taktilen Seh-Ersatzsystems.* Diplomarbeit, Universität Heidelberg, 1998.

[Pom99] Pomeranz, Ori: *Linux Kernel Module Programming Guide.* www.linuxdoc.org, 1999.

[Rub98] Rubini, Alessandro: *Linux Device Drivers.* O'Reilly, 1998.

[Rus99] Rusling, David A.: *The Linux Kernel.* www.linuxdoc.org, 1999.

[Sch99] Schemmel, Johannes: *An Integrated Analog Network for Image Processing.* Dissertation, Universität Heidelberg, 1999.

Danksagung

An dieser Stelle möchte ich allen Menschen danken, die zum Gelingen dieser Arbeit beigetragen haben. Mein besonderer Dank gilt folgenden Personen:

- Herrn Prof. Dr. K. Meier für die interessante Aufgabenstellung und Betreuung der Arbeit
- Herrn Prof. Dr. V. Lindenstruth für die Übernahme der Zweitkorrektur
- Herrn Dr. Johannes Schemmel für das geduldige Beantworten vieler Fragen, manche Debug-Session und für die vielen Tips, Ideen und Hilfestellungen in allen Hard- und Softwarefragen
- Andreas Breidenassel für die Tips bei der VHDL-Programmierung, die Hilfe beim Aufspüren von Fehlern in meinem VHDL-Code und für seine Korrekturen an der Arbeit.
- Thorsten Maucher für die Hilfe bei Problemen mit der QT Bibliothek
- Allen weiteren Mitgliedern des Kirchhoff-Instituts und besonders der Vision-Gruppe für das angenehme Arbeitsklima
- Meinen Eltern, die mir das Physikstudium überhaupt ermöglicht und mich immer unterstützt haben
- Sonja Becker für die moralische Unterstützung und Tips besonders in der Endphase der Arbeit

Erklärung

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den ....