Fachhochschule Brandenburg

Konzeption und Implementierung einer Videodigitalisierung und Videoausgabe unter Embedded Linux

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker(FH)

vorgelegt von

Frank Schwanz

Brandenburg 2002

Eingereicht am 21. Oktober 2002

1. Gutachter: Dipl. Informatiker Ingo Boersch

2. Gutachter: Dipl. Ing. Henry Westphal

Erklärung

Hiermit erkläre ich, da ich diese Arbeit selbständig und nur mit den zugelassenen und aufgeführten Hilfsmitteln erstellt habe.

Brandenburg, 21. Oktober 2002

Frank Schwanz

Aufgabe

Im Rahmen des Projektes „Initiative Intelligente Autonome Systeme“ wird an der FH Brandenburg ein Kernel für intelligente autonome Systeme entwickelt. Eine Teilkomponente des R-Cube-Systems ist eine eigenständige Bildverarbeitungskarte mit StrongARM-Prozessor und ARM-Linux.

Zielstellung des Themas ist die Konzeption und Implementierung der Videodigitalisierung und Videoausgabe der Bildverarbeitungskarte. Hierbei sind Schnittstellen zum Auslesen der Bilddaten (vorzugsweise Video4Linux) und Ausgabe von Bilddaten (vorzugsweise Framebuffer-Device) zu entwickeln. Die Umsetzung soll besonderen Wert auf die mögliche Nutzung in kommenden Linux-Kernelversionen sowie die Performance der Lösung legen. Die Funktionsfähigkeit des Systems ist durch geeignete Teststellungen zu evaluieren.

Inhaltsverzeichnis

1 Einleitung **8**
- 1.1 Aufgaben des BV-Boards . . . 9
- 1.2 Aufbau der Arbeit . . . 11

2 Hardware **12**
- 2.1 Das BV-Board . . . 13
- 2.2 LART-Board . . . 14
  - 2.2.1 Die Schnittstellen des LART Boards . . . 14
  - 2.2.2 Prozessor . . . 16
  - 2.2.3 Speicher . . . 19
- 2.3 Videoerweiterung LartVio . . . 27
  - 2.3.1 Analoge und digitale Videoübertragung . . . 28
  - 2.3.2 Der Videodecoder SAA7113 . . . 29
  - 2.3.3 Der Videoencoder ADV7171 . . . 30
  - 2.3.4 I^2C Bus . . . 31
  - 2.3.5 Der Videocontroller ACEX1K . . . 34
- 2.4 SA-1100 Multimedia Development Board . . . 38

3 Betriebssystem und Gerätetreiber **43**
- 3.1 Das Betriebssystem des LART-Board . . . 43
- 3.2 Linuxkernel . . . 44
- 3.3 Gerätetreiber und Geräteklassen . . . 45
  - 3.3.1 Adressräume . . . 47
  - 3.3.2 Interrupts . . . 47
- 3.4 Framebuffer . . . 48
  - 3.4.1 Das aktuelle Framebuffer Subsystem . . . 48
  - 3.4.2 Das Framebuffer Subsystem im Kernel 2.5 . . . 49
- 3.5 Video4Linux . . . 50
  - 3.5.1 Die Video4Linux-API . . . 50
  - 3.5.2 Die Video4Linux2-API . . . 52
- 3.6 Das I^2C Subsystem . . . 53

4 Anforderungen an das BV-Board **55**
4.1 Videoeingabe . . . 55
4.2 Videoausgabe . . . 58

5 Entwurf **60**
5.1 Grundentwurf . . . 60
5.2 Busmasterbetrieb . . . 63
5.3 Module des Videocontrollers . . . 65
5.3.1 DMA-Modul . . . 66
5.3.2 Adressgenerator . . . 66
5.3.3 Datenpuffer . . . 68
5.3.4 Videoeingabe . . . 70
5.3.5 Videoausgabe . . . 73
5.3.6 StrongARM Interface . . . 73
5.4 Gerätetreiber . . . 74
5.4.1 Framebuffer . . . 76
5.4.2 Video4Linux . . . 76
5.4.3 I^2C -Treiber . . . 78
5.5 Testverfahren . . . 79

6 Implentierung **81**
6.1 FGPA . . . 81
6.1.1 Das DMA-Modul . . . 81
6.1.2 Wandlung der Farbbereiche . . . 84
6.1.3 Austastrahmen für die Videoausgabe . . . 86
6.1.4 Compilerprobleme . . . 87
6.2 Gerätetreiber . . . 87
6.2.1 Framebuffer . . . 88
6.2.2 Video4Linux2 . . . 90
6.2.3 I^2C Treiber . . . 93
6.3 Schnittstellen zu Applikationen . . . 94
6.4 Nachweis der Funktionalität . . . 95

7 Zusammenfassung und Ausblick **99**

A Analyse **102**
A.1 Überprüfung der Speichertransferleistung . . . 102
A.1.1 Übertragung mit memcpy() . . . 102
A.1.2 Übertragung mit asmcpy() . . . 102
A.1.3 Speicheradressen bündig . . . 104
A.1.4 Das Programm mem2mem . . . 105

B Listings **109**
B.1 AHDL-Quellcode 109
B.1.1 YCbCr_YCbCr.tdf (Auszug) 109
B.2 Auszüge aus lartviofb.c (Framebuffertreiber) 111
B.2.1 Die Funktion lartviofb_check_var bzw. lartviofb_decode_var 111
B.2.2 Die Funktion lartviofb_set_par 113
B.2.3 Die Funktion lartviofb_init_fpga 116
B.2.4 Die Funktion lartviofb_encode_fix 117
B.2.5 Die Funktion lartviofb_encode_var 117
B.2.6 Die Funktion lartviofb_blank 118
B.3 Auszüge aus lartvio_vin.c (Video4Linux2-Treiber) 120
B.3.1 Die Funktion mmap_request_buffers 120
B.3.2 Die Funktion capture_queuebuffer 122
B.3.3 Die Funktion mmap_unrequest_buffers 123

C Testprotokolle **124**
C.1 Framebuffer-API Test 124
C.2 Video4Linux2 Test 126
C.3 Wellenformsimulation 127
C.4 Langzeittests 129

D CD-Inhalt **134**

Abbildungsverzeichnis

2.1 Blockschaltbild des BV-Boards 13
2.2 LART-Board: Oberseite aus [LARak] 14
2.3 StrongARM Blockschaltbild (aus [Int99, Functional Description]) 17
2.4 Zuordnung der CPU-Adressbits zu Speichermatrix 20
2.5 Fast Page Mode nach [Oet97] 21
2.6 Zeilen- und Spaltensignale eines *Burst of Eight* in der aktuellen Konfiguration 24
2.7 Blockschaltbild des LartVIO 27
2.8 I^2C Bus Bit-Transfer nach [Phi01, Fig.4] 32
2.9 I^2C Bus Daten-Transfer nach [Phi01, Fig.6] 33
2.10 I^2C Schreiboperation bei SAA7113 und ADV7110/7111 33
2.11 I^2C Leseoperation bei SAA7113 und ADV7110/7111 34
2.12 Logikelement in der ACEX 1K Architektur von Altera (aus [Alt01]) 35
2.13 Blockschaltbild des SA-1100 Multimedia Development Board [Int98, Figure 4-1] 38
2.14 Blockschaltbild der Videodigitalisierung des SA-1100 Multimedia Development Board (Ausschnitt aus [Int98, Figure 4-2]) 39
2.15 Blockschaltbild der Videoausgabe des SA-1100 Multimedia Development Board (Ausschnitt aus [Int98, Figure 4-6]) 40
2.16 Funktionsdiagramm SA1100 und SA1101 Developement Board [Int98, Figure 4-13] 41

3.1 Monolithische Architektur 45
3.2 Mikrokern Architektur 45
3.3 Prozess- und Interruptebene (nach [Mar99]) 47
3.4 einfaches Overlay 51
3.5 Overlay mit Clipping oder Chromakey 51
3.6 Chromakey Verfahren: die Farbe definiert den transparenten Bereich 52

3.7 Clipping: um die angegebenen Bereiche herum wird das Bild geschrieben 52
3.8 Struktur des I^2C Subsystems 54

5.1 Datenfluss der abstrahierten Videomodule. Der Prozessor verarbeitet den Videostrom 61
5.2 aktive Bild- und Austastzeiten eines Halbbildes 62
5.3 Datenfluss der abstrahierten Videomodule mit Puffer zur Nutzung der horizontalen Austastphase. Ein DMA-Modul koordiniert die Speicherzugriffe. 64
5.4 Entwurf des FGPA 67
5.5 Interlacing und Deinterlacing 68
5.6 Abhängigkeit der Füllzeit von FIFO-Größe und Füllsignal . . 71
5.7 Datenfluss: Videoeingabe 72
5.8 Die Schnittstelle: Videobearbeitung 72
5.9 Datenfluss: Videoausgabe 73
5.10 Übersicht über das Treiberdesign 75

6.1 Zustandsmaschine fuer den Speicherzugriff 82
6.2 Zustandsmaschine fuer den Zugriff auf den DMA 83
6.3 Zwischenzustände auf den Adress- und Steuerleitungen 84
6.4 Vorgang des Bildeinlesens 92

A.1 Steuerleitungen nRAS und nCAS bei Datentransfer mit memcpy() 103

Tabellenverzeichnis

1.1 Anwendungsszenarien für das BV-Board nach [BH02, Anwendungsszenarien] 10

2.1 Schnittstellen des LART-Boards 15
2.2 Alternative Funktionen der GPIOs am High-Speed-Connector 18
2.3 DRAM Konfigurationregister [Int99, 10.2.1 DRAM Configuration Register (MDCNFG)] 23
2.4 DRAM Wellenform Register im Kernel 2.4.18 24
2.5 Datenübertragung des 4:2:2 Digital Component Video (8 Bit Übertragung) 29
2.6 Funktion einer Lookup Table (LUT) 34
2.7 Merkmale der möglichen Bestückungsvarianten des ACEX 1K 36

5.1 Bildverändernde Funktionalität von Videoein- und Videoausgabe 65

6.1 Zuordnung der Farbkanäle RGB zu den eingelesenen 32Bit Werten 85

A.1 Datentransferraten mit memcpy() 103
A.2 Datentransferraten mit asmcpy() 104
A.3 Datentransferraten mit angepassten Startadressen (beide Aufrufe mit -align) 104

Kapitel 1

Einleitung

An der FH Brandenburg wird eine intelligente Roboterplattform für sehende autonome Systeme entwickelt, die sich im privaten Bereich, als Forschungs- und Ausbildungsplattform an Hochschulen oder auch als Basis für industrielle Anwendungen eignet. Das RCube Projekt verbindet eigenständige Module, deren Kombination verschiedenste Anwendungsmöglichkeiten eröffnen. Zu diesen Modulen gehören :

CPU-Board Die zentrale Steuerungseinheit kommuniziert mit den anderen Boards über eine CAN-Bus Schnittstelle.

Aksen-Board Das Aktoren- und Sensorenboard besitzt verschiedene analoge und digitale Ein- und Ausgänge. Integriert ist ein 8Bit Microprozessor, der die Steuerung der Ein- und Ausgänge übernimmt.

BV-Board Die Bildverarbeitungskarte soll Videoströme aufnehmen und verarbeiten. Zusätzlich wurde eine Videoausgabe integriert, die die Ergebnisse graphisch darstellen kann.

Jedes dieser Boards zeichnet sich durch seine Autonomie aus. Alle Boards sind mit Mikroprozessoren bestückt, die auf das entsprechende Board abgestimmt sind. Bereits für eigenständige Verwendung sind Anwendungsfälle vorhanden. In Kombination mit anderen Boards eröffnen sich weitere Anwendungsszenarien.

Das LART-Board, ein StrongARM-basiertes Embedded Linux Board der Universität Delft, ist die Grundlage von CPU-Board und Bildverarbeitungskarte. Die um eine Videodigitalisierung und -ausgabe erweiterte Bildverarbeitungskarte, im weiteren BV-Board genannt,

> ... soll die Bildaufnahme bis zur semantischen Analyse des Bilderstroms leisten und über eine CAN-Schnittstelle mit dem restlichen System kommunizieren

[BH02, Das Auge des Systems: BV-Board].

Das Unternehmen Tigris Elektronik GmbH entwickelte eine Erweiterungskarte für Videodigitalisierung und -ausgabe für das LART-Board, im folgenden kurz „LartVIO“ genannt. Diese Diplomarbeit umfasst die Konzeption und Entwicklung der Treibersoftware für die Videodigitalisierung und Videoausgabe, sowie die Integration der LartVIO-Hardware im Umfeld des LART-Board. Die CAN-Busschnittstelle des LartVIO ist kein Bestandteil dieser Diplomarbeit.

Haupteinsatzgebiet des BV-Boards ist die Bildaufnahme und -verarbeitung eines Bilderstroms. Die Aufgabe der Arbeit war, neben der Konzeption und Umsetzung der Software, eine umfassende Analyse der Hardware. Als Ergebnis sollte eine performante Lösung der Bildaufnahme und -wiedergabe entstehen, die künftigen Nutzern des BV-Boards Raum für die Implementation verschiedenster Anwendungen, insbesondere im Gebiet der Bildverarbeitung, bietet.

Die Schnittstellen zur Hardware sollten sich an im Betriebssystem vorhandene Schnittstellen anlehnen, um Portierung vorhandener Anwendungen auf das BV-Board zu erleichtern. In der Aufgabestellung wird besonderer Wert auf die Nutzung in kommenden Linuxversionen gelegt. Eine Betrachtung aktueller Entwicklungen für das Betriebssystem ist daher unerläßlich.

1.1 Aufgaben des BV-Boards

Für die Entwicklung eines Produktes ist es notwendig, herauszufinden, was entwickelt werden soll. Neben der reinen Funktionalität muss das Produkt auf die Nutzergruppe und deren Anforderungen zugeschnitten sein. Im letzten Abschnitt wurden bereits die anvisierten Nutzergruppen benannt:

- private Anwender
- Forschungs- und Ausbildungsplattform an Hochschulen
- Basis für industrielle Anwendungen

Private Anwender und die Ausbildung an Hochschulen benötigen eine leichte Bedienbarkeit des Produktes. Die Einarbeitungszeit in die Bedienung des Produktes muss bei privaten Anwendern und Ausbildungsplattformen an Hochschulen kurz sein, sonst wird das Produkt nicht angenommen. Die Ausbildung, deren Hauptaugenmerk die Wissensvermittlung ist, wird das Produkt als Hilfsmittel nutzen. Die Bedienung und in diesem speziellen Fall die Ansteuerung der Treiber muss für Standardanwendungen einfach gehalten

sein. Forschung und Entwicklung an Hochschulen und Industrie benötigen ein Produkt, das Raum für komplexe und anspruchsvolle Entwicklungen bietet. Diese Zielgruppen wünschen Erweiterbarkeit und Flexibilität des Produktes. Hard- und Software sollen erweiterbar und veränderbar sein. Das Treiberdesign sollte diese Ansprüche berücksichtigen.

In [BH02, Anwendungsszenarien] werden Einsatzgebiete für verschiedene Kombinationen der Module des RCube benannt. Die Anwendungsgebiete, in denen das BV-Board zum Einsatz kommen soll, sind:

BV	Autonomes Bildverarbeitungs-System
BV + CPU	Intelligente Kamera
Aksen + CPU + BV	R-Cube: Roboterkernel für Serviceroboter

Tabelle 1.1: Anwendungsszenarien für das BV-Board nach [BH02, Anwendungsszenarien]

In der Aufgabenstellung enthalten ist das Haupteinsatzgebiet Bildverarbeitung. Bildverarbeitungen stellen hohe Ansprüche an Prozessor und Datenbus. Es wird zu prüfen sein, welche Möglichkeiten die Hardware in Bezug auf Bildgrösse, Farbformat und Performance bietet.

Das BV-Board besitzt einen Videoausgang, der Video- und Grafikdaten auf den Bildschirm bringen soll. Die Versuchung liegt nahe, diesen Videoausgabeport als Grafikkarte zu implementieren und die Einsatzmöglichkeit auf Standardanwendungen auszudehnen. In wie weit sich sich diese Ideen umsetzen lassen, hängt vor allem von der Hardware ab. Von Bedeutung ist auch das Zusammenspiel von Videoein- und -ausgabe.

Entsprechend der Aufgabenstellung muss das Betriebssystem in der aktuellen Version betrachtet und die derzeit in Entwicklung befindlichen Veränderungen und Entwicklungen berücksichtigt werden. Von Interesse sind neben dem Framebuffertreiber und dem Video4Linux-Treiber auch allgemeine Änderungen an der Treiberarchitektur.

Als mobiles Gerät darf die Leistungsaufnahme der Hardware nicht unberücksichtigt bleiben. Dies gilt sowohl für den Hardware- als auch für den Softwarebereich.

Abschließende Tests sollen die Funktionalität der Software untermauern. Dieses Verfahren ist neben der Verifikation der Entwicklung üblich. Welche Testverfahren verwandt werden, wird in der Entwurfsphase zu prüfen sein.

1.2 Aufbau der Arbeit

Die Arbeit gliedert sich in fünf Hauptkapitel und soll den Entwicklungsprozess von der Bestandsaufnahme und den allgemeinen Betrachtungen über den Entwurf und die Implementierung, sowie der Testumgebung bis zu einem abschliessenden Kapitel auf künftige Erweiterungsmöglichkeiten und Anwendungen darstellen.

Wie im letzten Abschnitt angesprochen, stellt eine Bildverarbeitung hohe Ansprüche an die vorhandene Hardware. Eine Optimierung der Performance setzt Wissen über die vorhanden und benötigten Resourcen voraus. Desweiteren werden umfangreiche Kenntnisse der Hardware, insbesondere des Prozessors und dessen Datenbus, notwendig sein. Die Betriebsystemumgebung und deren Programmierung spielt für die Optimierung ebenfalls eine gewichtige Rolle. In Kapitel 2 wird des LART-Board, sowie die geplante Videoerweiterungskarte LartVIO untersucht. Nach einer kurzen Vorstellung der beiden Platinen wird auf einzelne, für die Entwicklung wichtige Baugruppen eingegangen. Am Ende des Kapitels wird eine Referenzimplementation einer StrongARM-basierten Multimediakarte der Firma Intel vorgestellt.

Kapitel 3 stellt das Betriebssystem des LART-Board vor und untersucht die für den Entwurf notwendigen Subsysteme des Kernels. Dabei wird die aktuelle Version des Betriebssystems mit den Entwicklungen für spätere Versionen verglichen. Als Ergebnis dieses Kapitels soll eine Entscheidung möglich sein, welche Subsysteme für das Treiberdesign verwendet werden. Die Subsysteme sollen mit der aktuellen Version des Betriebssystems funktionieren, aber auch mit der nächsten stabilen Version des Kernel verwendbar sein.

Die Anforderungen an die Bildverarbeitungskarte werden in Kapitel 4 spezifiziert. Das Kapitel berücksichtigt die aus der Analyse der Hardware und des Betriebssystems gewonnen Erkentnisse. Auf der Grundlage des Anforderungskataloges wird in Kapitel 5 eine geeignete Architektur der Treiber für das BV-Board entworfen. Es werden die Aufgaben der Hard- und Software spezifiziert und Überlegungen über notwendige Tests vorgenommen. Ein abschliessender Abschnitt wird den Entwurf mit der Aufgabenstellung verifizieren.

Das Kapitel 6 beschreibt die Implementierung der entworfenen Komponenten und deren Zusammenspiel. Ein weiterer Teil geht auf die Tests ein und zeigt die Schritte zum Nachweis der Funktionalität auf.

Die Ergebnisse der Arbeit werden schließlich in Kapitel 7 zusammengefasst und ein Ausblick auf mögliche Erweiterungen und weiterführende Entwicklungen geboten.

Kapitel 2

Hardware

Die Videodigitalisierung und -ausgabe LartVIO ist eine Erweiterungskarte für das LART-Board. Nach der Vorstellung des Gesamtmodules werden hier ausgewählte Teile der Hardware vorgestellt. Vom LART-Board werden der Prozessor und der Speicher näher betrachtet. Danach werden die Bauelemente des LartVio vorgestellt, da diese Bauelemente in das Betriebssystem eingebunden werden sollen. Die Spannungsversorgung wird hier nicht betrachtet, da sie für die Treiberentwicklung nicht von Bedeutung ist.

Als Abschluß wird eine Referenzimplementation der Firma Intel vorgestellt. Das „SA1100 Multimedia Development Board“ zeigt die Implementation einer Videodigitalisierung und enthält eine Beispielimplementierung für eine Videoausgabe im NTSC-Format.

2.1 Das BV-Board

Die Bildverarbeitungskarte LartVio ist über einen Erweiterungstecker, den High-Speed-Connector, mit dem LART-Board verbunden. Auf diesem Stecker sind die für den Buszugriff notwendigen Signalleitungen vorhanden. Alle Leitungen des Steckers sind an den Videocontroller angeschlossen. Hinter dem Videocontroller verbirgt sich ein rekonfigurierbarer Schaltkreis. Es handelt sich um einen Field Programmable Gate Arrays, im weiteren kurz FPGA. Videoencoder und -decoder sind ebenfalls mit dem Videocontroller verbunden.

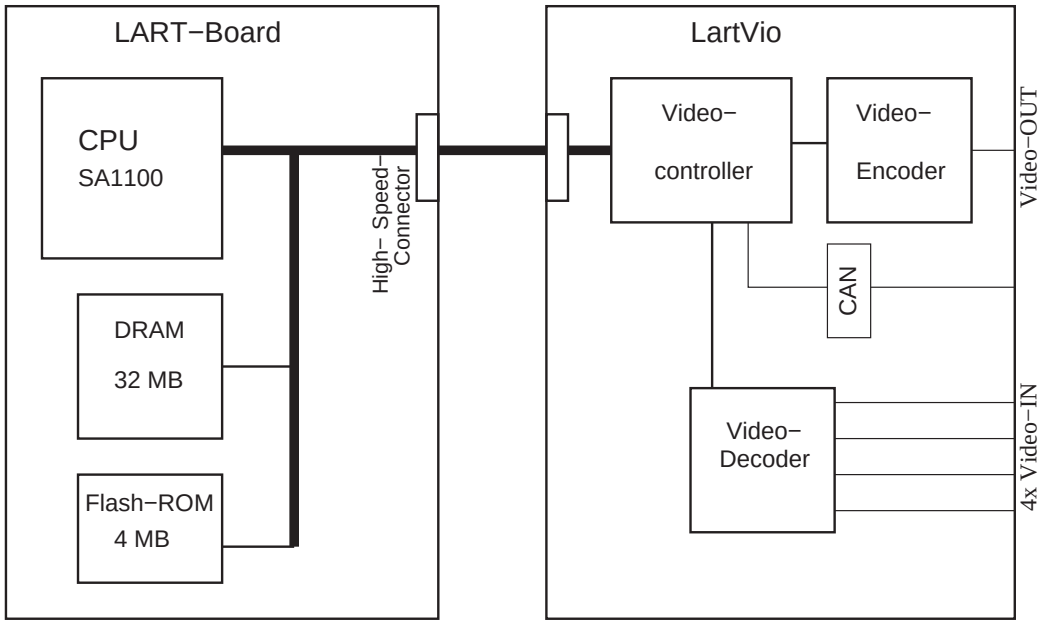

Abbildung 2.1: Blockschaltbild des BV-Boards

Durch die Verwendung eines rekonfigurierbaren Schaltkreises als Videocontroller kann das Hardwaredesign dem Anwendungsgebiet angepasst werden. Der Vollständigkeit halber soll der CAN-Controller des LartVIO nicht unerwähnt bleiben, der ebenfalls mit dem Videocontroller verbunden ist. Er wird jedoch nicht Gegenstand in dieser Arbeit sein.

Wie im Bild 2.1 ersichtlich, läuft die Kommunikation zwischen dem LART-Board und dem LartVio vollständig über den Videocontroller.

2.2 LART-Board

Das LART-Board ist ein StrongARM-basiertes Embedded-Linux-Board, das an der Universität Delft entwickelt wurde. Die Schaltpläne [Lar00] sind als Open Hardware frei verfügbar. Boards können entsprechend selbst gebaut werden. Das LART-Board wurde speziell für hohe Leistung (mehr als 200 MIPS), bei kleinem Stromverbrauch (weniger als 1 W), entwickelt. Das LART-Board ist ein 10x7 cm großes Mainboard mit StrongARM CPU, 32 MB RAM und 4 MB Flash Speicher.

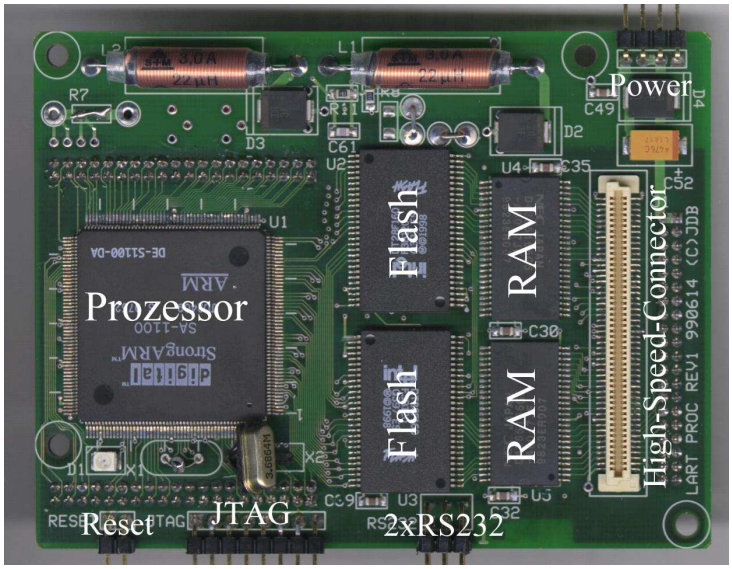

Abbildung 2.2: LART-Board: Oberseite aus [LARak]

2.2.1 Die Schnittstellen des LART Boards

Fast alle verfügbaren Signale des Prozessors wurden für externe Erweiterungen zugänglich gemacht. Daher residieren neben den standardisierten Schnittstellen vier weitere Steckverbinder auf dem LART-Board. Der High-Speed-Connector befindet sich auf der Oberseite, die Verbinder J2 - J4 auf der Unterseite des LART-Boards. Über zwei serielle Schnittstellen kann je eine Terminalverbindung mit dem LART-Board aufgebaut werden. Tabelle 2.1 gibt einen Überblick über die Schnittstellen des LART-Boards. Im folgenden werden diese Schnittstellen näher beschrieben.

Connector	Schnittstellen	Größe, Lage
Power	Stromversorgung	4 polig, seitlich
Reset	Resetschalter	2 polig, seitlich
RS232	2 x serielle Schnittstelle	6 polig, seitlich
JTAG	JTAG Interface (Hardwaredebugging)	8 polig, seitlich
J1, High-Speed-Connector	Adress- und Datenleitungen, PCMCIA, GP21-GP27	100 polig, oben
J2	Adress- und Datenbus Connector	40 polig, unten
J3	RS232, USB, IRDA, PCMCIA, JTAG Signale	40 polig, unten
J4	GPIO, LCD, sowie weitere PCMCIA-Signale	40 polig unten

Tabelle 2.1: Schnittstellen des LART-Boards

Powerconnector und Reset

Das LART-Board kann laut [LARak] mit 3.5 bis 16 V betrieben werden. In der LART-Mailing-Liste wurde vor dem Betreiben über 7.5V gewarnt, da einige Teile der Spannungsversorgung nicht für so hohe Spannungen spezifiziert sind und die Gefahr besteht das LART Board zu zerstören. Der Powerstecker ist so konzipiert, dass ein Vertauschen der Spannungspotentiale durch falschen Aufstecken nicht möglich ist. Werden die Spannungspotentiale an der Stromquelle vertauscht, kommt es über D4 [Lar00, LART Power suply] zu einem Kurzschluss. Die Stromquelle sollte daher eine Strombegrenzung besitzen.

Der Reset-Stecker führt neben dem Reset-Signal noch eine Masseanschluss, da das Reset-Signal Low-Aktiv ist. Hier kann ein einfacher Taster angeschlossen werden.

Die seriellen Schnittstellen

Das LART-Board verfügt über fünf serielle Schnittstellen [Int99, Peripheral Control Module 11], wovon zwei RS232 Schnittstellen (Serial Port1, Serial Port3) seitlich herausgeführt wurden. Die Übertragungsraten lassen sich von 56.24 bps[1] bis 230.4 Kbps einstellen. Daneben können über J3 eine USB Schnittstelle (max. 12 Mbps), ein IRDA Port(max. 4 Mbps), sowie einen „Multimedia Communication Port“(MCP) abgegriffen werden.

[1] bps = Bits per second

Das JTAG-Interface

Das JTAG Interface ist eine standardisierte Schnittstelle zum Testen von Hardware. Mit JTAG kompatiblen ICs ist es möglich, durch seriellen Zugriff über den JTAG Test Access Port (TAP) auf jeden einzelnen I/O Pin des Chips sowohl schreibend als auch lesend zuzugreifen (Boundary Scan). Der Zugriff auf den unprogrammierten Flash ist ein Beispiel für die Nutzung des JTAG-Interfaces – unerlässlich für die Erstprogrammierung des Bootloaders. Desweiteren ermöglicht es den Zugriff auf andere periphere Elemente zur Laufzeit, wie zum Beispiel auf den DRAM Bereich unter Umgehung des laufenden Betriebsystems.

Der High-Speed-Connector

Der 100-polige High-Speed-Connector enthält Adress-, Daten- und Steuerleitungen für den Zugriff auf ROM und DRAM-Bereiche. Desweiteren sind einige General Purpose I/O Pins verfügbar, die in Abschnitt 2.2.2 beschrieben werden. Da über den High-Speed Connector das LartVio mit dem LART Board verbunden wird, werden die Signale detailliert aufgeführt :

- alle prozessorexternen Adressleitungen A0-A25
- die Datenleitungen D0-D32
- DRAM Steuerleitungen nRAS0-nRAS3
- Byteauswahlleitungen nCAS0-nCAS3
- ROM Steuerleitungen nCS2 und nCS3
- Read und Write Signalleitung
- die General Purpose I/O Pins GP21-GP27
- sowie Signale zur Ansteuerung von PCMCIA-Geräten

2.2.2 Prozessor

Das Herz des Lart Boards ist der StrongARM SA1100. Es handelt sich dabei um einen Microcontroller für Embedded-Geräte. Abbildung 2.3 zeigt ein Blockschaltbild des Prozessors. Neben der zentralen Verarbeitungseinheit integriert der Microcontroller unter anderem mehrere serielle Schnittstellen,

einen LCD-Controller und digitale Ein-/Ausgabeports. Die internen Peripheriemodule werden durch zwei DMA-Controller unterstützt. Die DMA-Controller unterstützten nur interne Peripheriemodule. Zum Kopieren von Speicherbereichen kann der DMA-Controller nicht verwendet werden. Die Intel Application Note „Memory to Memory Transfer using the SA-1100 DMA“ [Int00b] beschreibt eine Methode unter Nutzung der IRDA Schnittstelle. Die maximale Transferrate liegt bedingt durch die serielle Schnittstelle bei 4 MBit/s.

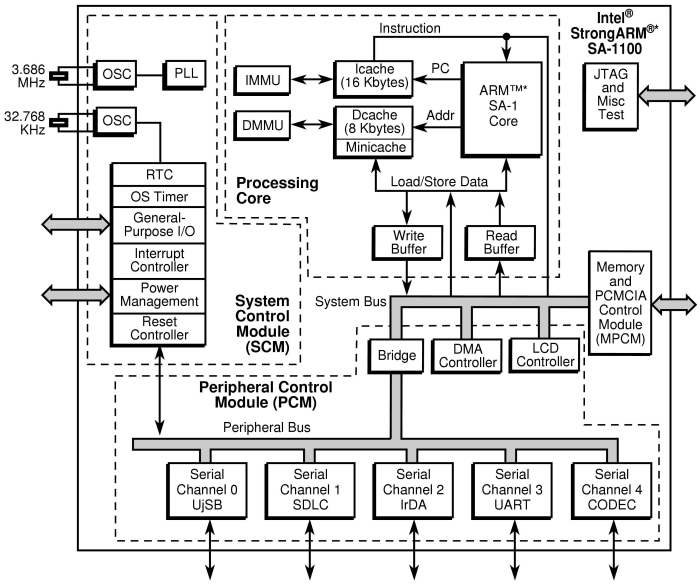

Abbildung 2.3: StrongARM Blockschaltbild (aus [Int99, Functional Description])

Ein 16 KB Instruktions- und ein 8 KB Datencache reduzieren die Zugriffe auf den Speicherbus. Ein Readpuffer und ein Writepuffer unterstützen Schreib- und Lesezugriffe auf den Speicherbus. Der SA-1100 besitzt 28 digitale Ein-/Ausgabeports (*GPIOs*). Jedes der 28 Bits ist einzeln programmierbar und kann folgende Funktionalitäten bereitstellen:

- digitaler Eingabepin
- externe Interruptleitung
- digitaler Ausgabepin

- alternative Funktion [2] (siehe [Int99, 11.5])

Am High-Speed-Connector, der Verbindung zur Videoerweiterung, sind die GPIOs 21-27 verfügbar. Die alternativen Funktionen dieser GPIOs sind in Tabelle 2.2 aufgelistet.

GPIO	**Signalname**	**Alternative Funktion**
21	MBGNT	Busanforderung (Abschnitt 2.2.3)
22	MBREQ	Busgewährung (Abschnitt 2.2.3)
23	TREQB	Testcontroller Signal
24	-	keine alternative Funktion
25	RTC clock	Echtzeittakt 1Hz
26	RCLK_OUT	Bustakt – CPU-Takt / 2
27	32KHZ_OUT	32 kHz Oszillator Takt

Tabelle 2.2: Alternative Funktionen der GPIOs am High-Speed-Connector

Der StrongARM implementiert die ARM V4 Architektur und dessen Befehlssatz [ARM01]. RISC Prozessor typisch ist die Load/Store Architektur. Er besitzt einen internen 32 Bit Adressbus und einen 32 Bit breiten Datenbus.

Der Prozessor besitzt 37 interne Register, die in sich teilweise überlappenden Bänken angeordnet sind. Dazu gehören:

- 30 allgemeine 32 Bit Register
- ein Programmcounter
- das aktuelle Statusregister
- fünf Sicherungstatusregister

Von den 30 allgemeinen Registern sind nur jeweils 15 als r0 ... r14 zu einem Zeitpunkt erreichbar. r13 wird als Stackpointer genutzt. Im User-Mode wird r14 als *link register* (lr) zur Speicherung der Rücksprungadresse in Unterroutinen benutzt. Auf den Programmcounter kann über r15 oder pc zugegriffen werden.

[2] nicht alle GPIO Pins besitzen alternative Funktionen

2.2.3 Speicher

Neben dem Prozessor befinden sich auf dem Lart Board 32 MByte RAM. Verwendet wurden zwei EDO-RAM Bausteine µPD426165 der Firma NEC auf der Oberseite des Lartbords und zwei EDO-RAM Bausteine MT4LC4M16F5 der Firma Micron auf der Unterseite des Lart Boards. In den elektrischen Kennwerten unterscheiden sich die Chips kaum.

Adressierung

Der StrongARM Prozessor arbeitet mit einem internen 32-Bit Adressbereich. Davon stehen 26 Adressleitungen (A0-A25) am externen Bus zur Verfügung. Den Adressbereich teilt der Prozessor in mehrere Speicherbänke. Die Signale für die Steuerleitungen der Speicherbänke (nRAS0-nRAS3) werden über die Adressbits A27-A31 gebildet. Es sind 2^{26} Bytes (64 MByte) in jeder Bank adressierbar [Int99, 2.4 Memory Map].

Der Adressbereich für die DRAM Bänke [Int99, 2.4 Memory Map] erstreckt sich von :

```
Bank0 0xC000 0000 - 0xC7FF FFFF
Bank1 0xC800 0000 - 0xCFFF FFFF
Bank2 0xD000 0000 - 0xD7FF FFFF
Bank3 0xD800 0000 - 0xDFFF FFFF
```

Auf dem LART-Boord werden nur zwei DRAM-Bänke genutzt. Die Steuerleitungen nRAS0 und nRAS1 entsprechen den DRAM-Bänken 0 und 1 und sind folgenden Teiladressen zugeordnet.

```
      A31 A30 A29 A28 A27
nRAS0 1   1   0   0   0
nRAS1 1   1   0   0   1
nRAS2 1   1   0   1   0
nRAS3 1   1   0   1   1
```

An den Adresseingängen der DRAM-Bausteine liegen die Adressleitungen A10-A20 und A24 des Prozessorbusses an [Lar00, LART Memory]. Entsprechend [Int99, 10.3.1 DRAM Overview] werden für die Zeilenadresse die Adressleitungen A24 und A20-10 benutzt, für die Spaltenadresse A23-A21 und A9-A2. Die Adressleitungen A0 und A1 entsprechen den Steuerleitungen nCAS0-nCAS3. Der µPD426165 nutzt in der Spaltenadresse nur die unteren 10 Adressbits [NEC97, S.4].

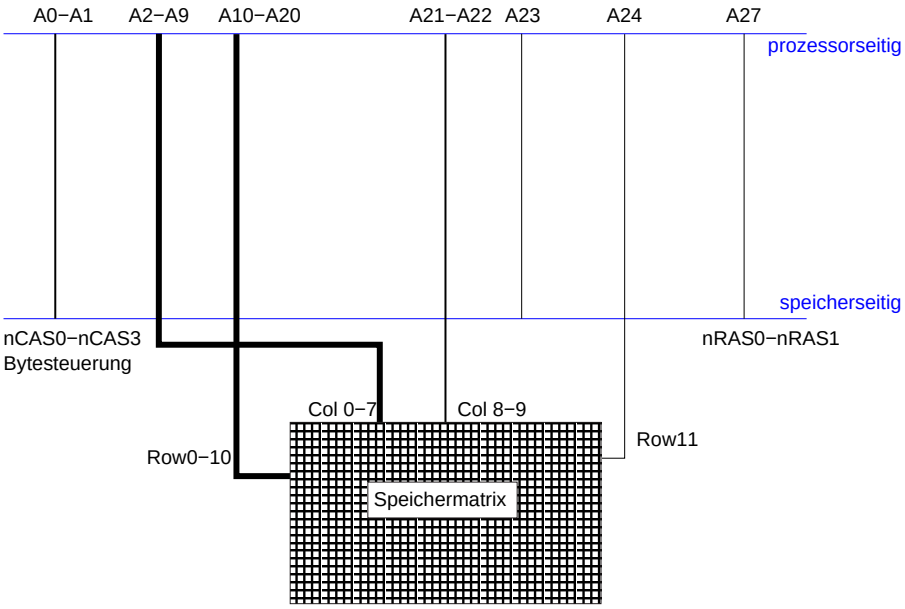

Abbildung 2.4: Zuordnung der CPU-Adressbits zu Speichermatrix

Die Adresse setzt sich somit aus den Adressleitungen nCAS0-nCAS3 (entspricht A0 und A1), A2-A22 und A24, sowie A27 (nRAS0 und nRAS1) zusammen. Damit ergibt sich ein maximaler Adressbereich von 2^{25} addressierbaren Speicherzellen oder 32 MByte addressierbarer Speicher. Abbildung 2.4 verdeutlicht die Zuordnung der Adressleitungen des Prozessors zu den Zeilen- und Spaltenadressen des Speichers.

Die verwendeten EDO-RAM Bausteine können im Fast Page Mode betrieben werden. Bei dieser Zugriffsart wird die Zeilenadresse nur einmal angelegt. Danach können wahlfrei Spaltenadressen angelegt werden ohne erneut die Zeilenadresse zu übermitteln. Mit der fallenden Flanke von RAS bzw. CAS wird die Adresse übernommen. Der Extended Data Output (EDO) vereinfacht den Lesezugriff. Mussten die Daten bei älteren DRAM-Bausteinen in der aktiven Zeit von nCAS gelesen werden, stehen die Daten bei einem Lesezugriff nach der Zeit t_{CAS} bis zur nächsten fallenden Flanke von CAS an. Diese Zugriffsart wird in [Int99, 10.1 Overview of Operation] als Burstmode bezeichnet.

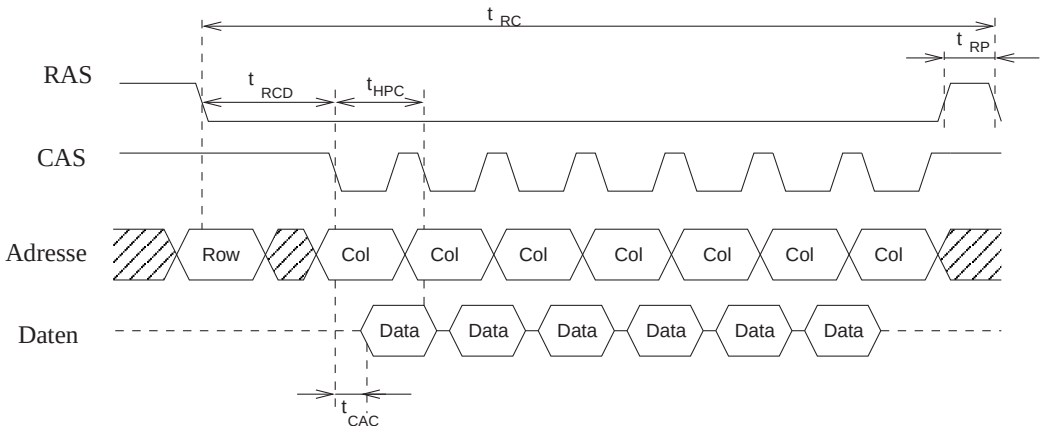

Abbildung 2.5: Fast Page Mode nach [Oet97]

Speichertransferleistung von DRAM und Prozessor

Der externe Speicherbus des SA-1100 kann theoretisch mit 400 MBit/s Daten vom und zum Prozessor übertragen, wenn pro Bustakt [3] ein 32 Bitwort gelesen wird. Möglich ist diese Geschwindigkeit mit entsprechenden SRAM Bausteinen und PCMCIA-Geräten, die in einem Bustakt addressiert und ein 32 Bit-Datenwort lesen oder schreiben können. Doch mit welcher Geschwindigkeit kann der verbaute DRAM Speicher den Prozessor mit Daten versorgen?

Die Übertragung zwischen EDO-RAM Speicher und Prozessor erfolgt asynchron. Der Takt spielt nur prozessorseitig eine Rolle. [NEC97] spezifiziert die Zugriffe für den verbauten DRAM. Grundlage für die weiteren Berechnungen sind die Spezifikationen für die µPD4265165-50 Bausteine.

Ein einfacher Schreib- oder Lesezyklus benötigt $t_{RC} = 84ns$. Die Schreib- und Lesezyklen im Burstmode ergeben sich aus den Zeiten für die Übertragung von Zeilen- und Spaltenadressen. Die Übertragung der Daten erfolgt innerhalb des Spaltenzyklus.

Die Zeit für die Übernahme der Zeilenadresse t_{RCD} beträgt minimal 11 ns. In dieser Zeit wird die Zeilenadresse im DRAM-Baustein zwischengespeichert und liegt nun für weitere Spaltenzugriffe an. Ein vollständiger Spaltenzyklus t_{HPC} dauert mindestens 20 ns. Der erste Spaltenzyklus wird begrenzt durch die $nCAS$ Haltezeit $t_{CSH} = 38ns$. Nach einem Burstzyklus wird eine Zeitspanne $t_{RP} >= 30ns$ benötigt, die die Gültigkeit der Zeilenadresse beendet.

[3] $Bustakt = CPUTakt/2$

Ein Burstzyklus [4] ergibt sich aus

$$t_{BUSRT} = t_{RCD} + t_{CSH} + (n-1) * t_{HPC} + t_{RP} \tag{2.1}$$

wobei n die Anzahl der Spaltenzugriffe innerhalb eines Burstzyklus sind.

Für die Auffrischung der DRAM Speicherzellen sind 4096 Refreshzyklen innerhalb von 64 ms notwendig (64000 Zyklen/s). Ein *RAS before CAS* Zyklus [5] ist 89 ns ($t_{CSR} + t_{RC}$ [NEC97]) lang. 5,696 ms pro Sekunde werden demnach für die Auffrischung der DRAM Speicherzellen benötigt. In dieser Zeit steht der Bus für die Datenübertragungen nicht zur Verfügung.

Abzüglich der Zeit für Refreshzyklen sind

$$\frac{1s - 5,696 * 10^{-3}s}{84 * 10^{-9}s} = 11.836.952$$

einfache Schreib-/Lesezugiffe möglich. Die Datenübertragungsleistung liegt bei

$$45.15 MByte/s = 11.836.952 \frac{Zugriffe}{s} * 4 \frac{Bytes}{Zugriff}.$$

Der SA-1100 unterstützt Burstzugriffe bei DMA-Operationen, sowie beim Füllen und Leeren der Cacheline mit 4 und 8 Spaltenzugriffen in einem Burst. Ein „Burst of Eight“ benötigt [6]

$$11ns + 38ns + (8-1) * 20ns + 30ns = 219ns.$$

In der theoretisch freien Buszeit sind 4.540.200 Burstzyklen innerhalb einer Sekunde möglich. Das entspricht einer maximalen Datentransferleistung von 138 MByte/s. Werden „Burst of four“ für den Speicherzugriff genutzt, sinkt die Übertragungsleistung auf 109 MByte/s.

Der Prozessor arbeitet synchron mit dem Prozessortakt, mit 221MHz [7]. Der Bustakt beträgt jeweils $CPUTakt/2$, also 110,5 Mhz. Ein Bustakt ist 9,05 ns lang.

Die DRAM Bänke werden in den StrongARM-Registern MDCNFG und MDCAS0 - MDCAS3 konfiguriert (siehe auch [Int99, 10.2 Memory Configuration Register]). Eine individuelle Konfiguration der einzelnen Speicherbänke ist nicht möglich. In den Betriebssystemsourcen ist die Konfiguration in der Datei arch/arm/mach-sa1100/cpu-sa1100.c zu finden.

[4] Einige Parameter wurden zur Vereinfachung nicht berücksichtigt. Sie beeinflussen das Ergebnis nur geringfügig

[5] Refreshart des SA-1100

[6] Einschränkungen des StrongARM wurden in dieser Berechnung nicht berücksichtig

[7] Einstellbar zwischen 56 MHz und 280 Mhz. Für die Berechnungen wird die Standardeinstellung von 221MHz herangezogen

Name	Konfiguration	Bedeutung
DE	0x3	Bank 0 und 1 aktiv
DRAC	0x2	11 Bit für die Adresszeile, siehe Abbildung 2.4
CDB2	0x0	CAS Wellenform wird jeden CPU Takt verschoben
TRP	0x04	5 Bustakte zwischen zwei Zugriffen
TRASR	0x07	8 nRAS-Bustakte für Refresh
TDL	0x3	3 CPU-Takte nach Deaktivierung von nCAS werden Daten übernommen
DRI	0x1A7	$(423 * 4) * 9,05ns = 15,3\mu s$ zwischen zwei Refreshzyklen

Tabelle 2.3: DRAM Konfigurationregister [Int99, 10.2.1 DRAM Configuration Register (MDCNFG)]

Tabelle 2.3 zeigt die Konfiguration des StrongARM-Register MDCNFG. Zwischen zwei DRAM Zugriffen werden 5 Bustakte eingefügt, $t_{RP} = 45.25ns$. 4,85 ns nach der steigenden Flanke von nCAS werden die Daten bei einem Lesezugriff in den Prozessor übernommen (TDL).

Der Refreshcounter DRI wird synchron mit dem Bustakt heruntergezählt. Mit jedem vierten Überlauf wird ein internes Refreshflag gesetzt, das den Prozessor veranlasst, nach Beendigung des aktuellen Buszugriffes einen Refreshzyklus auszuführen. Dazu werden alle nCAS Leitungen aktiviert. Nach zwei weiteren Bustakten werden die nRAS Leitungen aller DRAM-Bänke für $TRASR+1$ Bustakte [8] aktiv. Die nCAS Leitungen werden zwei weitere Takte später deaktiviert. Der nächste Buszugriff erfolgt dann frühestens TRP+1 Bustakte später. Die Zeit für einen Refreshzyklus kann mit

$$t_{Refresh} = 2 + (TRASR + 1) + 2 + (TRP + 1)$$

berechnet werden. Entsprechend der Konfiguration des LART-Boards dauert ein Refreshzyklus 154 ns. Alle 15,3 µs wird ein Refreshzyklus generiert. Für die Generierung Refreshzyklen belegt der StrongARM den Speicherbus 10ms pro Sekunde.

[8] Die Dokumentation hierzu ist widersprüchlich. [Int99, 10.2.1 DRAM Configuration Register (MDCNFG)] spricht von $TRASR-1$, [Int99, 10.3.3 DRAM Refresh] von $TRASR+1$.

Die Register MDCAS0, MDCAS1 und MDCAS2 enthalten die nCAS-Wellenform für einen *Burst of Eight* Speicherzugriff. Einen CPU-Takt nach dem Anlegen der Zeilenadresse wird nRAS aktiviert. Gleichzeitig beginnt die nCAS Wellenform. Über ein Schieberegister werden die Inhalte der Bitstellen der Register MDCAS0 bis MDCAS3 an nCAS-Steuerleitungen gelegt. Bis zum Abschluss des Zugriffs wird das Schieberegister mit jeden Takt um eine Bitstelle verschoben. Das Konfigurationsbit CDB2 des Registers MDCNFG gibt an, ob die Verschiebung mit jedem CPU-Takt oder jedem Bustakt erfolgt. Eine 1 in der Bitstelle entspricht einem High-Zustand der nCAS-Leitung, eine 0 dem Low-Zustand. Die Spaltenadresse wird einen CPU-Takt vor der Aktivierung von nCAS angelegt.

MDCAS0	1100	0111	0001	1100	0111	0000	0011	1111
MDCAS1	1111	1111	1100	0111	0001	1100	0111	0001
MDCAS2	1111	1111	1111	1111	1111	1111	1111	1111

Tabelle 2.4: DRAM Wellenform Register im Kernel 2.4.18

Der StrongARM nutzt diese Methode für alle Schreib- und Lesezugriffe auf den DRAM. Der Speicherzugriff endet, wenn die Anzahl der $nCAS$ Pulse mit der Burstgröße übereinstimmt. Dabei wird ein einfacher Speicherzugriff als „Burst of One“ angesehen.

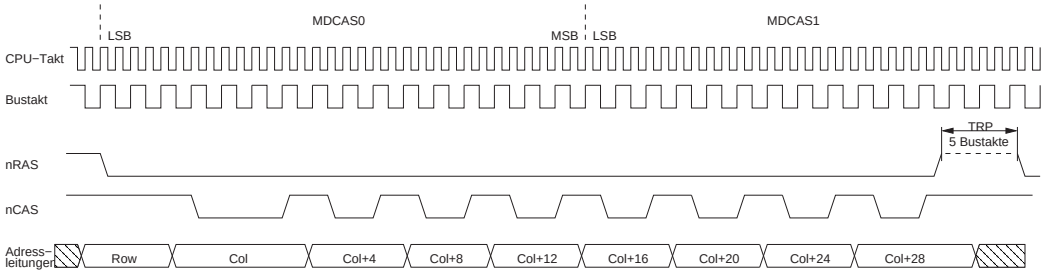

Abbildung 2.6: Zeilen- und Spaltensignale eines *Burst of Eight* in der aktuellen Konfiguration

Die Zugriffszeit für einen *Burst Of Eight* ergibt sich aus 54 CPU-Takte der Wellenform, einem Takt zwischen der Deaktivierung von nCAS und nRAS, sowie 5 $(TRP + 1)$ Bustakte zwischen zwei nRAS-Zyklen,

$$BurtOfEight = 294ns = (54 + 1) * 4,525ns + (TRP + 1) * 9,05ns.$$

Abzüglich der Refreshzeiten sind bei ständiger Generierung von *Burst Of*

Eight Zyklen

$$\frac{\frac{1s-10*10^{-3}s}{294*10^{-9}s} * 8Zugriffe * 4\frac{Byte}{Zugriff}}{1s} = 102,76MByte/s$$

möglich. Da *Burst Of Four* Zyklen nur 30 Bits der Wellenform verwenden, ist ein Zyklus $195ns$ lang. Die Datentransferleistung liegt noch bei $77.5MByte/s$.

Neben diesen theoretischen Überlegungen wurden einige Tests durchgeführt, die die Datentransferleistung unter realen Bedingungen wiederspiegeln sollen. Das komplexe Design des Prozessors und der Overhead des Betriebssystems müssen bei der Übertragungsleistung berücksichtigt werden. Dazu wurde zwei Datenbereiche angelegt, die über die Cacheline eingelesen und geschrieben werden. Die 8 KByte Datencache sollten keinen Einfluss auf die Transferleistung haben, daher wurden 150 KByte - 1,2 MByte grosse Datenpuffer verwendet. Die Tests sind in Anhang A.1 dokumentiert.

Für die Übertragung wurden die Standardkopiermethode *memcpy* und eine optimierte Assemblerroutine verwendet. Bei der Optimierung wurde davon ausgegangen, dass die Daten immer in 1 KByte Blöcken gelesen werden.

Die ARM4v Architektur stellt Multiple Load and Store Befehle für das Füllen und Speichern von Registern zur Verfügung. Bei 8 verwendeten 32 Bit Registern können mit einem Befehl 32 Byte gelesen bzw. geschrieben werden.

memcpy nutzt diese Möglichkeit bereits, muss jedoch auch abweichende Grössen (nicht durch 8 teilbare) berücksichtigen. In Tabelle A.1 sind die Ergebniss für *memcpy* und die optimierte Assemblerroutine aufgeführt. Die Assemblerroutine greift bei Lesezugriffen mit *Burst Of Eight* und bei Schreibzugriffen mit *Burst Of Four* auf den Speicher zu. Die Übertragung mit *Burst Of Eight* kann über das Verhältnis zur berechneten Übertragsleistung mit *Burst Of Eight* näherungsweise ermittelt werden.

$$BurstOfEight_{real} = 98MByte/s = \frac{86MByte/s * 102,76MByte/s}{\frac{1}{2}(102,76MByte/s + 77.5MByte/s)}$$

Die maximale real erreichbare Übertragungsleistung liegt in der Region der berechneten Transferleistung.

Externe Busmaster

Der SA1100 unterstützt externe Geräte, die den Speicher selbst ansprechen können. Der *alternative Busmastermode* nutzt die GPIOs 21 und 22 als Steuerleitungen. Dazu müssen die alternativen Funktionen der GPIOs aktiviert und der Testcontroller abgeschalten werden. Die GPIOs 21 und 22 besizten

zwei alternative Funktionen, die über das Register *TUCR* ausgewählt werden. GPIO 21 übernimmt die MBREQ Funktion, die mit einem aktiven Signal eine Anfrage des externen Busmaster signalisiert. Der Prozessor beendet eventuelle Buszugriffe und schaltet alle Address-, Daten- und Steuerleitungen in den hochohmigen Zustand. *MBGNT* (GPIO 22) signalisiert die Freigabe des Busses. Der externe Busmaster kann die Steuerung des Speicherbusses übernehemen. Nach Beendigung seiner Aktivitäten auf dem Speicherbus deaktiviert der externe Busmaster das *MBREQ* Signal wieder.

Wurde in der Zeit der externen Busaktivitäten im StrongARM das Refreshflag gesetzt, führt der Prozessor nach Rückgabe des Busses einen Refresh durch.

2.3 Videoerweiterung LartVio

Das LartVIO wird das LART-Board um eine Videoein- und Videoausgabe erweitern. Ein CAN-Buscontroller soll die Kommunikation mit anderen Modulen des R-Cube ermöglichen. Auf dem LartVIO werden drei verschiedene Spannungen benötigt. Aus $3.3V$ des Lart Board werden $2.5V$ und $5V$ erzeugt, da die verschiedenen Schaltkreise mit unterschiedlichen Spannungen arbeiten. Abbildung 2.7 zeigt ein Blockschaltbild des LartVIO, wobei die Spannungsversorgungen nicht berücksichtigt sind.

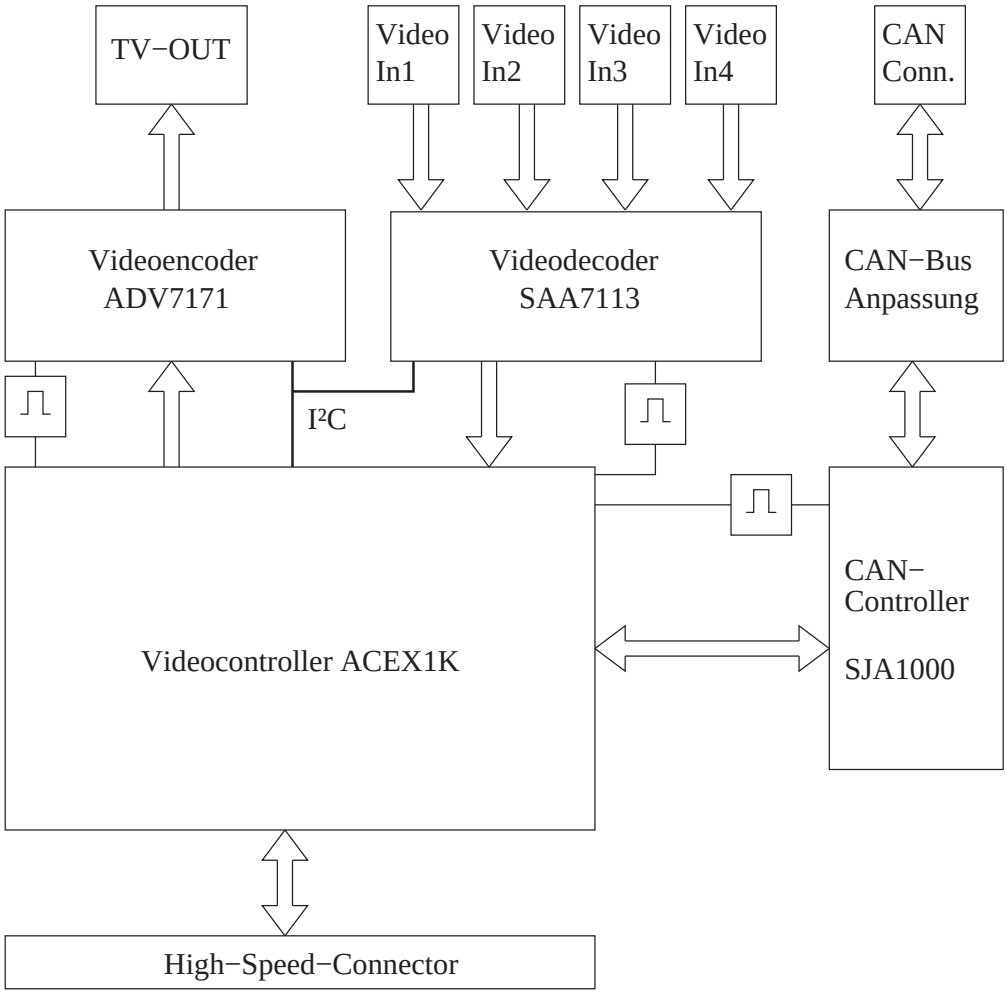

Abbildung 2.7: Blockschaltbild des LartVIO

Die Konfiguration der Encoder- und Decoderschaltkreise erfolgt über einen I^2C-Bus. Encoder-, Decoder- und CAN-Schaltkreis haben eine eigene Takt-

versorgung, die über den Videocontroller einzeln an- und abschaltbar sind. Die Abschaltung des Taktes versetzt die Schaltkreise in einen Ruhezustand.

2.3.1 Analoge und digitale Videoübertragung

Für das Verständnis der Kodierung und Dekodierung von Videosignalen ist ein kurzer Einblick in die Videosignaltechnik und digitale Videostandards notwendig. Die Techniken werden in [Sto95], [BHK96] und [Jac96] ausführlich beschrieben.

Videosignale enthalten Bildinhalt- und Zeitinformationen. Zu den Zeitinformationen gehören die horizontalen und vertikalen Synchronisationsimpulse. Sie markieren den Anfang einer Zeile bzw. den Beginn eines Halbbildes. Die Bilder werden in der Fernsehtechnik im Zeilensprungverfahren (interlaced) übertragen. Dabei wird das Bild in zwei Halbbilder geteilt. Das erste Halbbild enthält alle ungeraden Zeilen, das zweite Halbbild alle geraden Zeilen. Nach CCIR Norm besteht ein Bild aus 625 Zeilen, wovon 575 Zeilen Bildinformationen enthalten. Die Bildwiederholrate liegt bei 25Hz. Eine Zeile ist 64µs lang. Davon werden 52µs für die Übertragung der Bilddaten einer Zeile genutzt. Die verbleibenden 12µs werden für die horizontale Synchonistation verwendet.

Das amerikanische RS170-Videosignal enthält 525 Zeilen. Die Bildwiederholrate liegt bei 30Hz. Das Seitenverhältnis von 4:3 gilt für beide Standards. Das Seitenverhältnis gibt an, wie die Auflösung eines Bildes aufgeteilt ist. Bei CCIR liegt diese Auflösung bei 768 Pixel x 576 Zeilen im darstellbaren Bereich. RS170 hat einen darstellbaren Bereich von 640 Pixel x 480 Zeilen.

NTSC (National Television System Committee) ist ein Farbvideostandard in den USA, Kanada, Japan und in Teilen von Südamerika. PAL ist eine Abwandlung von NTSC, vor allem in Europa und Australien verbreitet. PAL und NTSC definieren die Farbübertragung auf einem BAS-Signal. Das entstehende Signal wird FBAS-Signal genannt. Bei beiden Verfahren werden zusätzlich zum Y-Signal oder Luminanzsignal zwei Farbdifferenzsignale übertragen. Es sind die Signale R-Y und B-Y, wobei R und B für die Farben aus dem RGB-Farbsystem stehen. Es entspricht dem YUV Farbsystem.

$$
\begin{aligned}
Y &= 0,3R + 0,59G + 0,11B \\
U = R - Y &= 0,7R - 0,59G - 0,11B \\
V = B - Y &= -0,3R - 0,59G + 0,89B
\end{aligned}
$$

Für die digitale Videoübertragung wurde von der CCIR (Comittee Consultatif International des Radio-Communications) eine Empfehlung ausgear-

beitet – die ITU-R BT.601 (ehemals CCIR601). Als Farbformat wurde das YCbCr Farbsystem entwickelt. Das YCbCr Format ist eine skalierte und nullpunktverschobene Variante des YUV-Farbsystems. Die Helligkeitswerte definiert das Y-Signal zwischen 16 und 235. Die Farbsignale Cb und Cr sind zwischen 16 und 240 definiert, wobei der Nullpunkt bei 128 liegt.

$$Y = 0,592(Y - 16)$$
$$U = 0,503(Cb - 128)$$
$$V = 0,710(Cr - 128)$$

Das 4:2:2 YCbCr Format tastet für jeweils zwei Luminanzwerte (Y) ein Cb und ein Cr Signal ab. Üblicherweise werden die Signale mit 8 bis 10 Bit Auflösung abgetastet. Ein 8 Bit abgetasteter Doppelpixel wird als Y2-Cr-Y1-Cb 32 Bitwert gespeichert.

In den nicht definierten Bereichen von 0 bis 15 und 236(241 für Farbsignale) bis 255 bei einer 8 Bit Speicherung können zusätzliche Informationen übertragen werden.

Für die Übertragung der Zeitinformationen dienen die Start-Of-Video (SAV) und End-Of-Video (EAV) Werte. Im SA und EA Statuswort sind horizonale und vertikale Synchronisationsinformationen, sowie die Informationen über das Halbbild (ODD/EVEN) enthalten.

Austastphase			Zeitreferenzcode				720 Pixel YCbCr 4:2:2 Daten						Zeitreferenzcode			
...	80	10	FF	00	00	SA	Cb0	Y0	Cr0	Y1	...	Y719	FF	00	00	EA

Tabelle 2.5: Datenübertragung des 4:2:2 Digital Component Video (8 Bit Übertragung)

Von den 720 übertragenen Pixel sind nach CCIR nur 704 wirklich aktive Pixel. Dadurch ergibt sich ein kleiner schwarzer Rand an der rechten und linken Seite des Videobildes.

2.3.2 Der Videodecoder SAA7113

Der Videodecoder [Phi99] besitzt vier Videoeingänge, die als zwei Y/C-Eingänge, vier FBAS-Eingänge oder als Mischform ein Y/C-Eingang und zwei FBAS-Eingänge geschalten werden können. Das Y/C Format trennt das Helligkeits- und Farbsignal. Für dieses Format werden zwei Eingänge benötigt. Das FBAS – Farbbild-Austast-Synchron-Signal, auch als CVBS oder Composite bekannt – enthält Farb- und Helligkeitsinformationen und Synchronimpulse in einem Signal. Das FBAS-Format ist das bei Kameras

am häufigsten verwendete Format. Die maximal vier Videoeingänge werden über einen analogen Schalter ausgewählt. Es wird nur jeweils ein Videostrom gleichzeitig verarbeitet.

Unterstützt werden die Videostandards CCIR und RSA170A und die Farbübertragungssysteme PAL, NTSC und SECAM. Der Decoder erkennt automatisch Videostandard und Farbübertragunssystem und schaltet in den entsprechenden Modus. Die Videonormen können bei abgeschaltener automatischer Erkennung auch manuell eingestellt werden. Helligkeit, Kontrast, Sättigung und Farbverschiebung sind ebenfalls veränderbar.

Die Videodaten werden über die Ausgänge VPO[7..0] im YCbCr 4:2:2 Format (siehe Abschnitt 2.3.1) übertragen. Tabelle 2.5 zeigt die Folge der Daten am VPO-Bus. Mit Beginn der Zeile wird der Zeilenreferenzcode übertragen, gefolgt von 720 Pixelwerten im YCbCr 4:2:2 Format. Die Daten liegen mit der steigenden Flanke des Pixeltakts LCC an. Für Videosynchronsignale stehen zwei Steuerleitungen RTS0 und RTS1 zur Verfügung, die auf verschiedene Signale eingestellt werden können.

Der SAA7113 wird über einen $24,576 MHz$ Quarzgenerator mit dem nötigen Takt versorgt. Über PLL Schaltungen wandelt der SAA7113 den Takt in die 27 Mhz Pixelfrequenz. Ein fehlender Takt am Schaltkreis initiert einen Reset. Alle Ausgänge werden in den hochohmigen Zustand versetzt. Der Schaltkreis muss danach neu initialisiert werden.

Die Konfiguration des Videodecoders erfolgt über einen I^2C -Bus, über den der Registersatz des SAA7113 geschrieben und gelesen werden kann. Philips gibt als Slaveadresse zwei 8 Bit Adressen 0x48 und 0x49 an. Das entspricht gemäß I^2C Spezifikation der 7 Bit Adresse 0x24. Die kleinste Bitstelle gibt die Schreib- / Leserichtung an. Der I^2C Bus wird in Abschnitt 2.3.4 besprochen.

2.3.3 Der Videoencoder ADV7171

Die Generierung der Videosignale für die Bild- und Grafikausgabe übernimmt der ADV7171 [Ana01] von Analog Devices. Als Videosignale können FBAS (CVBS), S-Video (Y/C), YUV, RGB ausgegeben werden. Auf dem Lart-VIO stehen von den vier DACs (Digital to Analog Converter) des ADV7171 nur DAC A als Ausgang zur Verfügung. Diese Hardwarebeschränkung besteht aufgrund der hohen Leistungsaufnahme der DACs [9]. Somit steht als Ausgabeformat nur FBAS (Farbbild-Austast-Signal) zur Verfügung, da alle anderen Formate mehr als eine Übertragungsleitung benötigen.

[9] ca. 37mA pro DAC

Die Grafikdaten übernimmt der ADV7171 im YCbCr 4:2:2 Format über einen 16 Bit Datenbus. Im 8 Bit Modus werden nur die Datenleitungen P[7..0] für die Übertragung der Bilddaten genutzt. Auf dem LartVIO sind nur die Datenleitungen P[7..0] mit dem FPGA verbunden.

Die Videosynchronsignale können extern in den ADV7171 eingespeist oder vom ADV7171 selbst erzeugt werden. Liegen Videosignale als Datenstrom vor, wird die erste Methode benötigt. Eine direkte Anzeige der Videodaten des Videodecoders unter Umgehung von Hauptspeicher und Prozessor ist möglich, in dem die Videosynchronsignale und Videodaten des Videodecoders direkt an den Videoencoder übertragen werden. Der ADV7171 arbeitet dann im Slavemodus. Sind die Grafikdaten wahlfrei verfügbar, kann man auf die erzeugten Videosynchronsignale des ADV7171 reagieren und die entsprechenden Grafikdaten dem Videoencoder übermitteln. Der Videoencoder muss dazu in den Mastermodus versetzt werden.

Implementiert wurde ein dreistufiges Powermanagement. Der Low-Power-Mode reduziert die Leistung der DACs um 45%. Im Sleep-Mode wird der Stromverbrauch auf 200nA reduziert. Alle Funktionen bis auf den I^2C -Bus werden abgeschaltet.

Für die Konfiguration der internen Register bringt der ADV7171 einen I^2C -Bus mit. Über das Pin ALSB kann die 7Bit Slaveadresse auf 0x25 oder 0x26 eingestellt werden.

2.3.4 I^2C Bus

Videoencoder und Videodecoder nutzen für die Konfiguration der internen Register den I^2C -Bus. Alle Register mit Ausnahme des Adressierungsregisters können gelesen und geschrieben werden. Die Adressierung der Register ist bei beiden Chips gleich und wird im letzten Teil dieses Abschnitts vorgestellt.

Der I^2C Bus wurde von Philips [Phi01] für die Kommunikation zwischen integrierten Schaltkreisen entwickelt. Eingesetzt wurde er zuerst in der Fernseh- und Videotechnik. Die Stärken dieses seriellen Zweidrahtbusses liegen in der einfachen Kommunikationsschicht und der geringen Anzahl an Leitungen. Die Geschwindigkeit liegt je nach Implementation zwischen

0-100 kbit/s im Standard Mode,

0-400 kbit/s im Fast Mode und

0-3.4 Mbit/s High Speed Mode, ab Version 2.0.

Für die Ansteuerung des Encoder- und Decoderchips ist die Übertragungsgeschwindigkeit nicht relevant, da keine zeitkritische Steuerung über den I^2C Bus erfolgt. Es wird daher nicht näher auf die Unterschiede zwischen den verschiedenen Modi eingegangen. Durch eine separate Taktleitung (*SCL*) ist die Datentransferrate dynamisch, kann also zwischen 0 und Maximalgeschwindigkeit variieren. Geräte mit verschiedenen Geschwindigkeiten können dabei an einem Bus betrieben werden. Die Spezifikation teilt die Geräte in zwei Klassen nach dem Master/Slave Prinzip:

Master Der Master initiiert und beended den Transfer. Er erzeugt die Signale der Taktleitung.

Slave Jedes Slave-Gerät erhält eine eindeutige Adresse. Es reagiert auf die Anforderungen des Master.

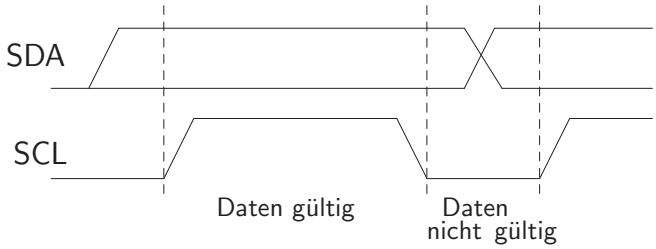

Abbildung 2.8: I^2C Bus Bit-Transfer nach [Phi01, Fig.4]

Die Spezifikation beschreibt ebenfalls den Betrieb mit mehreren Master-Geräten. Da die oben beschriebene Hardware nur einen Master, das Lart-Board, besitzt, wird auf den Betrieb mit mehreren Mastern nicht näher eingegangen.

Takt- und Datenleitung sind bidirektionale Open-Drain Leitungen. Im High-Zustand der Taktleitung (*SCL*) wird der Zustand der Datenleitung (*SDA)* als gültig betrachtet, während sich im Low-Zustand von SCL der Zustand von SDA ändern darf. Die Daten werden dem entsprechend mit der steigenden Flanke von SCL übernommen. Eine Ausnahme bilden die Start- und Stopbedingung. Hier ändert sich die Datenleitung, während die Taktleitung sich im High-Zustand befindet. Die Start- (S) und Stopbedingung (P) werden nur vom Master erzeugt. Sie leiten ein Datentransfer ein bzw. beenden ihn. Eine Stopbedingung gibt den Bus wieder frei. Die Startbedingung kann auch wiederholt auftreten (Sr) ohne Beendigigung des Transfers durch eine Stopbedingung.

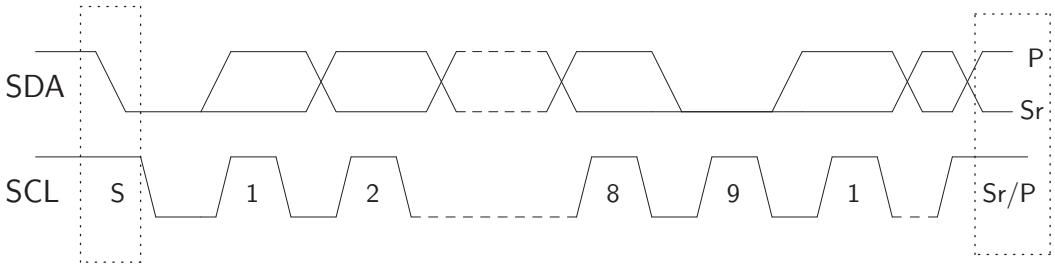

Abbildung 2.9: I²C Bus Daten-Transfer nach [Phi01, Fig.6]

Ein Datenwort besteht aus 8 Bit und einem Acknowledge-Bit. Während der Übertragung kann der Slave die Taktleitung auf Low ziehen. Der Master wartet dann bis die Taktleitung wieder auf High ist. Nach dem achten Datenbit folgt das Acknowledge-Bit. Der Sender gibt die Datenleitung frei (High-Zustand). Während der High-Periode des Taktimpules muss der Empfänger den Empfang durch den Low-Zustand der Datenleitung bestätigen.

Eine Übertragung beginnt mit der Adressierung des Slave-Gerätes – der 7-Bit breiten Slaveadresse. Das achte Bit des Datenwortes ist das Read/Write Flag, welches die Datenrichtung angibt. Das entsprechende Slave-Gerät quitiert die Übertragung im Acknowledge-Bit und bleibt bis zur nächsten Stopbedingung aktiv. In den Datenblättern werden häufig je eine Adresse zum Lesen und zum Schreiben angegeben. Diese entsprechen der 7 Bit-Adresse und dem Read/Write Flag. Neben der 7-Bit Adressierung existiert eine Erweiterung für 10-Bit Adressen.

Encoder- und Decoderchip interpretieren das erste Datenbyte nach der Slaveadresse als Subadresse. Mit dieser Erweiterung werden die internen Register des jeweiligen Chips addressiert.

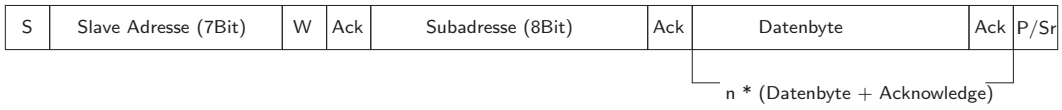

Abbildung 2.10: I²C Schreiboperation bei SAA7113 und ADV7110/7111

Die Subadresse wird in einem chipinternen Register gespeichert und nach der Übertragung eines Datenbytes inkrementiert.

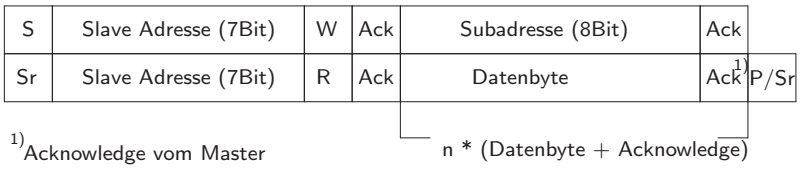

Abbildung 2.11: I²C Leseoperation bei SAA7113 und ADV7110/7111

2.3.5 Der Videocontroller ACEX1K

Die Verbindung zwischen Speicherbus des Lartboards und den Videobausteinen, sowie dem CAN-Bus Controller übernimmt ein FPGA *ACEX 1K* [Alt01] der Firma Altera. FPGAs – *Field Programmable Gate Arrays* – sind komplexe programmierbare Logikbausteine, deren Architektur auf einzelnen Funktionsblöcken beruht, die in einer regelmäßigen Struktur angeordnet sind. Die Funktionalität von FPGAs wird im Wesentlichen durch die Programmierung der Verbindungsleitungen zwischen den Basiszellen erreicht.

Adresse		**Daten/Funktion**			
a[1]	a[0]	OR	XNOR	XOR	NAND
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

Tabelle 2.6: Funktion einer Lookup Table (LUT)

Altera bezeichnet diese Funtionsblöcke als Logikelemente. Die Funktion der Logikelemente [10] wird durch Lookup Tables (LUT) realisiert. Sie sind aufgebaut wie eine normaler Speicher, wobei für jede der 2^n Kombinationen an n Eingängen eine Speicherzelle adressiert wird. Der Inhalt der Speicherzellen bestimmt die logische Funktion. Tabelle 2.6 zeigt beispielhaft unterschiedliche Funktionen in Abhängikeit von den Inhalten der Speicherzelle.

Der ACEX 1K implementiert in jedem Logikelement eine LUT mit vier Eingängen. Neben der LUT besitzt jedes Logikelement eine Registerstufe, die als D-, T-, JK- oder RS-Flipflop konfiguriert werden kann. Das Register kann über den *Clock* und den *Clock Enable* synchron betrieben werden. Die Setz- und Rücksetzeingänge *PRN* und *CLRN* erlauben den asynchronen Betrieb des Registers, unterstützt von der *Clear & Preset Logic Control* Einheit. Der *Carry-Chain* Block ermöglicht die Verknüpfung mehrerer Logikelemente zu

[10] Die Beschreibung der Logikelemente bezieht sich auf die Altera-Architektur

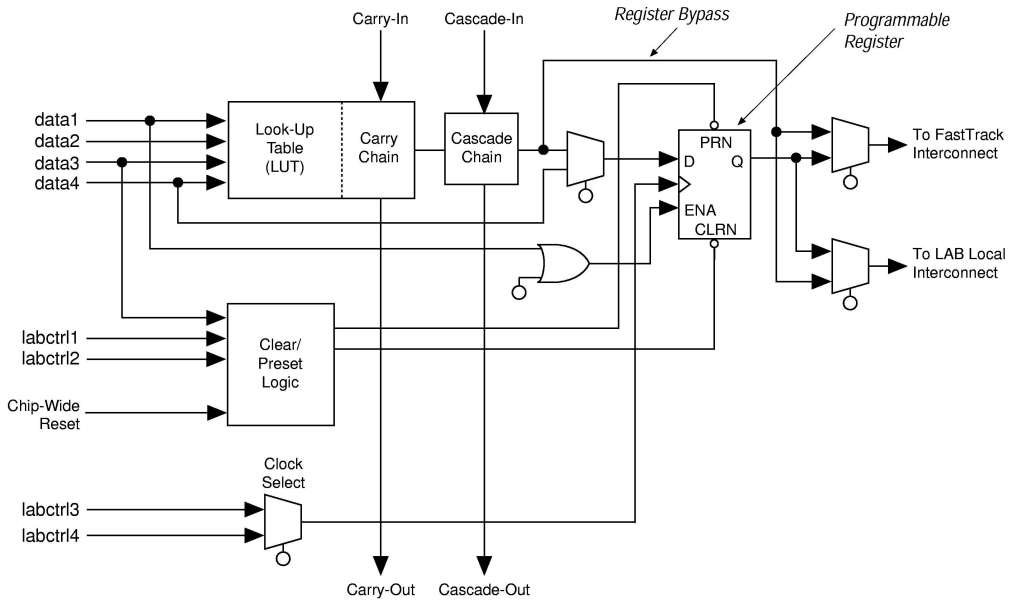

Abbildung 2.12: Logikelement in der ACEX 1K Architektur von Altera (aus [Alt01])

Zählern und Addierern, während die *Cascade* Signale bei Funktionen mit mehreren Eingängen kurze Verzögerungszeiten garantieren sollen.

Jeweils acht Logikelemente sind zu einem *Logic Array Block* (LAB) zusammengeschalten. Die Carry und Cascade Signale der einzelnen Logikelemente in einem *Logic Array Block* sind mit dem jeweils nachfolgenden Logikelement verbunden. Diese Verschaltung ermöglicht eine höhere Integration und schnellere Signallaufzeiten.

Neben den Logikblöcken sind im ACEX 1K flexible RAM Blöcke (EABs) integriert. Diese können wie Lookup Tables für die Generierung komplexer Funktionen oder als Speicher genutzt werden. Der Speicher kann von zwei Seiten mit unterschiedlichem Takt angesprochen werden (dual-port RAM). Die Konfiguration ermöglicht die Nutzung des Speichers als Speicherblock oder FIFO (First-In-First-Out), jeweils mit der Option single-port oder dual-port. Die einzelnen Blöcke können zu Größen von 256x16 Bits, 512x8 Bits, 1024x4 Bits oder 2048x2 Bits geschalten werden.

Auf dem LartVIO wurde ein ACEX 1K im 208-Pin PQFP Gehäuse vorgesehen. Die Platine kann mit den pinkompatiblen Versionen EP1K30, EP1K50 und EP1K100 bestückt werden. Die Unterschiede zwischen den Versionen sind in Tabelle 2.7 aufgeführt.

Feature	EP1K30	EP1K50	EP1K100
Logikelemente	1728	2880	4992
EABs	6	10	12
Total RAM Bits	24.576	40.960	49.152

Tabelle 2.7: Merkmale der möglichen Bestückungsvarianten des ACEX 1K

Stromverbrauch

Die Leistungsaufnahme des ACEX 1K kann mit

$$P = P_{INT} + P_{IO} = (I_{CCSTANDBY} + I_{CCACTIVE}) * V_{CC} + P_{IO}$$

berechnet werden. Die Leistungsaufnahme für die IO-Pins wird beeinflusst durch die angeschlossenen Geräte. Die aktive Stromaufnahme entsteht im Umschaltzeitpunkt der Logikzellen. Die aktive Stromaufnahme $I_{CCACTIVE}$ ist daher abhängig von der Anzahl der Logikzellen und der Häufigkeit der Umschaltzustände, die durch den Takt beeinflusst werden.

$$I_{CCACTIVE}(\mu A) = K * f_{MAX} * N * tog_{LC}$$

f_{MAX} die maximale Taktfrequenz

N die Anzahl der verwendeten Logikelemente

tog_{LC} die typische Anzahl der Logikelemente, die jeden Takt schalten

Nicht benutzte Logikelemente sollten daher über den *Clock Enable* Eingang gesperrt oder der Takt für ganze Bereiche abgeschalten werden.

Entwicklungsumgebung

Für die Programmung stellt Altera zwei Entwiclungsumgebungen, MAX Plus II und Quartus II, zur Verfügung. Beide Entwicklungsumgebungen können von Alteras Webseite heruntergeladen werden. Sie unterscheiden sich von den kommerziell vertriebenen Versionen durch einen eingeschränkten Funktionsumfang. Die Entwicklungsumgebungen erlauben das Design über Schaltpläne und Programmiersprachen wie Verilog-HDL, VHDL oder Altera-HDL. Eine Funktionsbibliothek mit vorgefertigten Standardlogikelementen unterstützt die Entwicklung. Für die Verifikation steht eine statische Wellenform-Simulation zur Verfügung. Beide Produkte bieten Schnittstellen für Produkte von Fremdanbietern. Für die komplexe Verifikation stellt Altera ebenfalls das

Produkt ModelSim-Altera, eine eingeschränkte Version von ModelSim der Firma Model Technology, zur Verfügung. ModelSim-Altera kann nur mit den kostenpflichtigen Versionen von MAX Plus II und Quartus II genutzt werden.

2.4 SA-1100 Multimedia Development Board

Für den SA1100 wurde von der Firma Intel bis Mai 2000 ein Multimedia Development Board für die Evaluierung eigener StrongARM-Entwicklungen vertrieben und diente gleichzeitig als Referenzdesign. Das Benutzerhandbuch [Int98] beschreibt Aufbau und Funktionsweise der Komponenten des Boards. Im Folgenden werden die Videoeingabe und -ausgaben des SA1100 Multimedia Development Board vorgestellt.

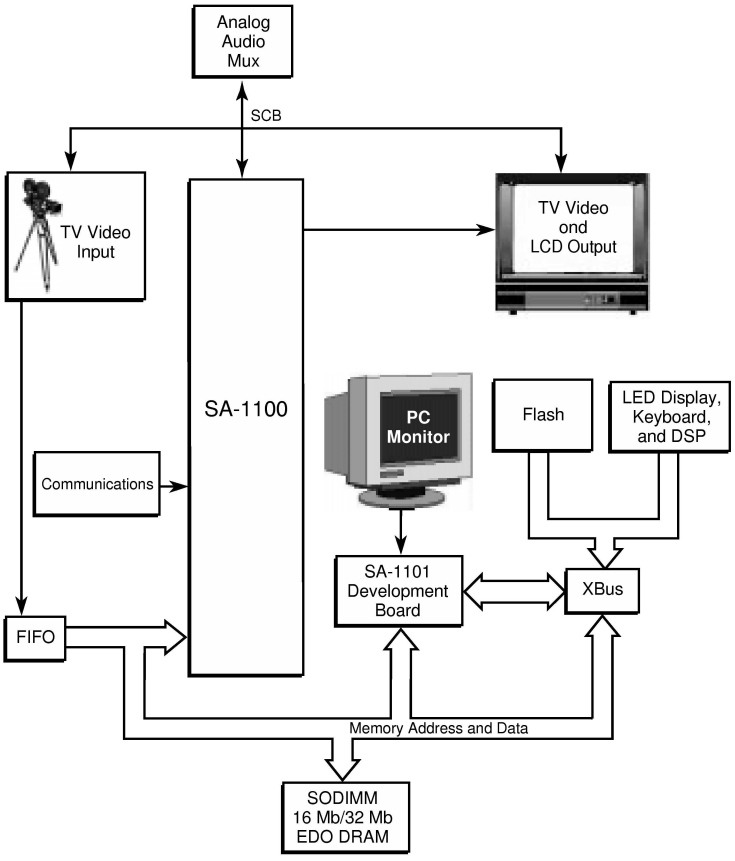

Abbildung 2.13: Blockschaltbild des SA-1100 Multimedia Development Board [Int98, Figure 4-1]

Das SA1100 Multimedia Development Board besteht aus einer Hauptplatine und der Erweiterungsplatine SA1101 Development Board. Auf der Hauptplatine sind neben dem Prozessor unter anderem zwei Videoeingänge, eine Videoausgabe und der Hauptspeicher untergebracht. Auf der Erweiterungsplatine ist die zweite Videoausgabe von Interesse.

Als Hauptspeicher verwendet Intel EDO-RAMs, die über einen SODIMM-Sockel gesteckt werden. Der Prozessor wird mit 220MHz getaktet. Der Bustakt ist mit dem Bustakt des LART-Boards vergleichbar. Intel gibt die Bandbreite des Busses mit 120 MByte/s an. Im Verhältnis zu den berechneten Übertragsraten des LART-Board müsste die Bandbreite bei ca. 102 MByte/s liegen. Intel hat hier möglicherweise eine bessere Anbindung des Speichers erreicht.

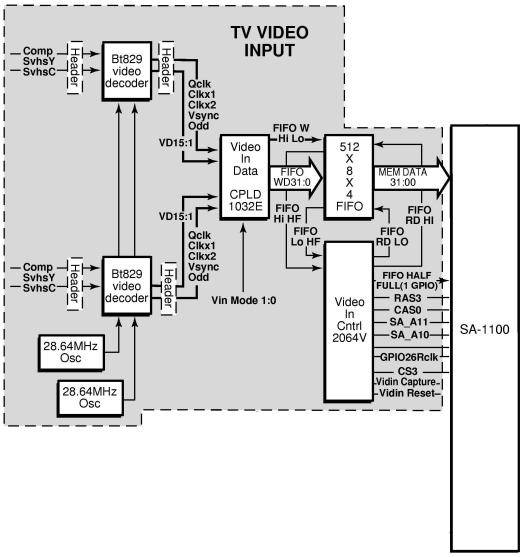

Abbildung 2.14: Blockschaltbild der Videodigitalisierung des SA-1100 Multimedia Development Board (Ausschnitt aus [Int98, Figure 4-2])

Die Videodecodierung übernehmen zwei Bt829. Diese Videodecoder wandeln das analoge Videosignal in YCbCr 4:2:2 Daten, die über einen 16 Bit breiten, optional auch 8 Bit breiten, Bus übertragen werden. Die Bt829 können das Videobild skalieren und beschneiden, interessant für ROI in der Bildverarbeitung. Die mittlere Leistungsaufnahme ist angegeben mit 170mA bei 5.0V, ca. 850mW.

Über einen CPLD – ein programmierbarer Logikbaustein – werden die Daten in zwei 512x16Bit breite FIFOs übertragen. Wählbar ist die Übertragung eines Videos mit 32Bit Übertragungsbreite oder zwei 16Bit breite Videoströme. Ein weiter CPLD übernimmt die prozessorseitige Ansteuerung der FIFOs und die Übertragung des FIFO *HalfFull* Signals. Das *HalfFull* Signal ist mit dem GPIO1 verbunden. Die FIFOs arbeiten mit der gleichen Geschwindigkeit wie der Hauptspeicher und wird in über die DRAM Bank3 in den Hauptspeicher eingeblendet.

Auf die Übertragung von Videosynchronsignalen wurde verzichtet. Die Synchronisation des Videostromes kann über Kontrollcodes ermittelt werden, die am Anfang jeder Zeile und bei der vertikalen Syncronisation auftreten (Vgl. Abbildung 2.5) . Der HalfFull-Interrupt liest den Videostrom bis zur Framestartkennung und beginnt dann mit der Übertragung der Videodaten. Der FIFO wird mit *Bursts of Eight* ausgelesen und in den Hauptspeicher geschrieben. Bei der Berechnung der Bandbreite rechnet Intel dabei mit $32 * 8 * 30ns = 7680ns$ für das Leeren eines FIFO, 30ns als Zeit für einen CAS Zyklus. Diese Zeit liegt ca. 2.5ns über dem technisch machbaren (vlg. Abschnitt 2.2.3).

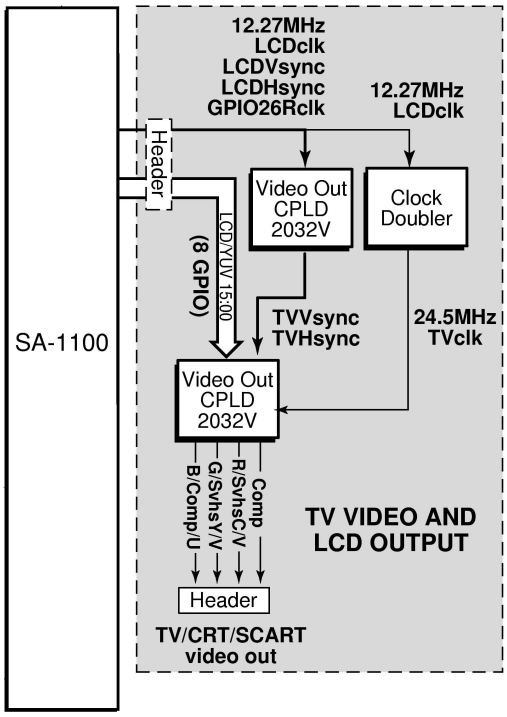

Abbildung 2.15: Blockschaltbild der Videoausgabe des SA-1100 Multimedia Development Board (Ausschnitt aus [Int98, Figure 4-6])

Für die Videoausgabe auf dem SA-1100 Multimedia Development Board wurde der LCD Controller genutzt. Dieser liest die Videodaten aus dem Hauptspeicher über einen integrierten DMA-Controller und stellt diese einschließlich horizontalen und vertikalen Synchronisationssignalen an den GPIO-Pins zur Verfügung. Diese Lösung belastet den Speicherbus nur mit Lesezugriffen. Die Synchronisationssignale für LCD-Displays werden in Video-

synchronisationsignale gewandelt und an einen Videoencoder übertragen. Die Arbeit erledigen zwei CPLD Schaltkreise. Beschrieben wurde ebenfalls die Nutzung eines Videoencoders ADV7175, statt des zu programierenden CPLD Schaltkreises.

Das Videobild muss im Framebuffer bereits interlaced vorliegen, da der LCD-Controller eine solche Funktionalität nicht besitzt. Entsprechend müssen Anwendungen ebenfalls ein Interlaced Bild erzeugen, will man neben Videodaten auch Grafik ausgeben.

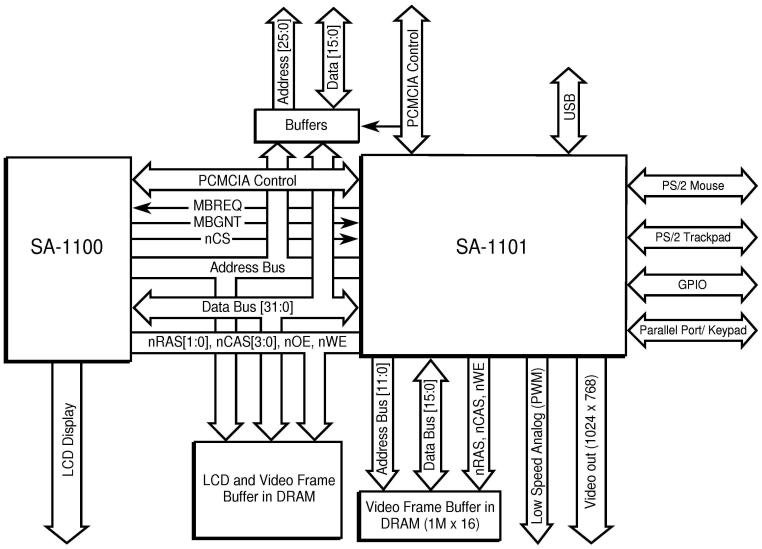

Abbildung 2.16: Funktionsdiagramm SA1100 und SA1101 Developement Board [Int98, Figure 4-13]

Für die Grafikausgabe auf einem Monitor befindet sich auf dem SA-1101 Development Board eine Grafikkarte. Die Erweiterungskarte nutzt für den Zugriff auf den Hauptspeicher den Busmastermode, der in Abschnitt 2.2.3 beschrieben wurde. Die Grafikausgabe auf der Erweiterungskarte wurde für den Anschluss von Monitoren entwickelt. Über das Farbformat des Grafikspeichers wurde keine Aussage getroffen. Da ein PC-Monitor RGB-Signale benötigt, muss bei nicht-RGB Farbformaten eine Farbraumkonvertierung im CPLD erfolgen, der die Übertragung der Daten an den VGA-Controller übernimmt. Beschrieben wurden zwei Varianten, die sich in der Lage des Grafikspeichers unterscheiden.

Im Haupspeicher untergebracht benötigt das Erweiterungsboard keinen eigenen Grafikspeicher. Nachteilig ist die zusätzliche Belastung des Speicherbusses, der bereits Video- und TV-Ausgabestrom bewältigen muss. Für

höhere Auflösungen ist diese Variante nicht geeignet.

Externer Grafikspeicher auf der Erweiterungsplatine wird ebenfalls über den CPLD angesteuert, der den Busmasterzugriff auf den Hauptspeicher vornimmt. Die Übertragung der Grafikdaten für den Bildaufbau belasten den Hauptspeicherbus in dieser Konfiguration nicht.

Kapitel 3

Betriebssystem und Gerätetreiber

Dieses Kapitel befasst sich eingehend mit dem Betriebssystem des LART-Boards. Der erste Teil dieses Kapitels stellt das Betriebssystem und Grundlagen für die Treiberentwicklung vor. Im zweiten Teil werden das Framebuffer Subsystem und Video4Linux vorgestellt. In Abschnitt 2.3 wurde als Kommunikationslayer zu Encoder und Decoder der I^2C Bus vorgestellt. Da bereits ersichtlich ist, dass für den I^2C Bus ebenfalls Treiber erstellt werden müssen, befasst sich der letzte Abschnitt mit dem I^2C Subsystem im Linuxkernel. Entsprechend der Aufgabenstellung werden sowohl aktuelle als auch in Entwicklung befindliche Schnittstellen untersucht.

3.1 Das Betriebssystem des LART-Board

Auf dem LART-Board startet Linux als Betriebssystem. Es entspricht dem Standardlinuxkernel, erweitert um einen Patch von Russel King. Der Patch beinhaltet Änderungen und Erweiterungen für die ARM-Architekturen, die bisher noch nicht in den Kernel eingeflossen sind. Für den StrongArm Prozessor werden eine ganze Reihe von Implementierungen unterstützt. Neben dem LART-Board gehören dazu unter anderem das Itsy Board, eine Beispielimplementierung von Compaq und Personal Digital Assistants(PDAs) von Compaq, Hewlett Packard und Sharp.

Der Bootloader *blob* kopiert das Kernelimage, das im Flash-ROM gespeichert ist, in den RAM-Bereich und startet das gepackte Kernelimage. Beim Bootvorgang entpackt der Kernel eine ebenfalls auf dem Flash-ROM befindliche komprimierte Ramdisk. Die Ramdisk enthält das Dateisystem. Dateien, die während des Betriebs auf das LART-Board übertragen werden, existieren

auf dem LART-Board daher nur bis zum nächsten Reset.

Eine weitere Methode den Flash als Festplatte anzusprechen ermöglicht das Jornaling Flash File System. Vorteil dieser Methode ist die persistente Speicherung der Daten. Die 4 MB Flash-ROM des LART-Boards abzüglich Bootloader und Kernelimage erlauben dann nur ein minimales Linuxsystem, da im Gegensatz zur Ramdisk das Journaling Flash File System die Daten bisher nicht komprimiert. Diese Option soll in einer der nächsten Versionen hinzukommen.

3.2 Linuxkernel

Der Linuxkernel stellt den eigentlichen Betriebssystemkern, die Schnittstelle zur Hardware dar. Begonnen als Programmierübung des Informatikstudenten Linus Torwald stellt der aktuelle Kernel mit knapp 140 MB Sourcen ein beachtliches Projekt dar. Die Zahl der Entwickler ist kaum mehr überschaubar.

Die Entwicklung des Betriebssystems läuft in mehreren Strängen.

stable kernel werden von Linus Torvald für die allgemeine Nutzung als genügend stabil erachtet. Stable Kernel erhalten immer eine gerade Minor-Nummer, also 2.0 oder 2.4.

development kernel entspricht den Entwicklerversionen. Die Minor-Nummern sind immer ungerade. Die Entwicklungsreihe ist häufig weitreichenden Änderungen unterzogen.

Viele Entwicklungen neuerer Kernelversion werden auf ältere Versionen portiert, sofern die Änderungen nicht die Stabilität des Kernel gefährden und die Änderungen in den Schnittstellen vorhandener Treiber nicht zu umfangreich sind. Insbesondere Gerätetreiber, die neuere Hardware unterstützen, werden auf ältere Versionen portiert. Bisher wurden die jeweils letzen beiden stabilen Kernel weiter gepflegt.

Linux beruht auf einer monolithischen Architektur. Im Gegensatz zur Mikrokernelarchitektur, in der die Gerätetreiber in einem geschützten Benutzerbereich residieren und mit einem minimalen Kern kommunizieren, sind in einer monolithischen Architektur auch die Gerätetreiber in den Kern integriert. Sie arbeiten im priviligierten Modus der CPU und im Kerneladressraum. Der Vorteil gegenüber einem Microkernel ist der minimierte Kommunikationsaufwand und erweiterten Möglichkeiten der Optimierung. Nachteil ist die geringere Resistenz gegenüber fehlerhaften Gerätetreibern.

Applikationen kommunizieren mit dem Kernel über Systemaufrufe (*system calls*). Ein Systemaufruf ist der Übergang eines Prozesses vom Nutzer-

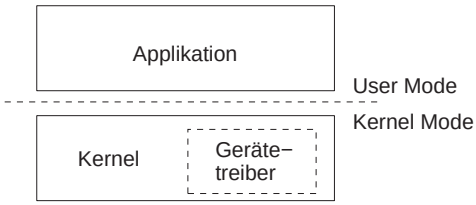

Abbildung 3.1: Monolithische Architektur

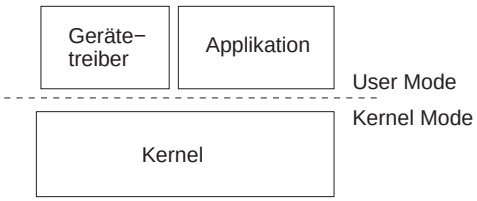

Abbildung 3.2: Mikrokern Architektur

zum Systemmodus. Unter Linux ist es üblich, bekannte Systemaufrufe als Bibliotheksfunktionen zur Verfügung zu stellen.

3.3 Gerätetreiber und Geräteklassen

Gerätetreiber stellen eine Softwareschicht zwischen der Hardware und der Applikationsschicht dar. Sie abstrahieren die Hardwaresteuerung und stellen sie über eine definierte Schnittstelle der Applikationsschicht zur Verfügung. Dabei verhindert der Treiber gleichzeitig unbefugten und missbräuchlichen Zugriff auf die Hardware. Rubini und Corbert beschreiben Gerätetreiber in [RC01, An Introduction to Device Drivers] :

> Device drivers take on a special role in the Linux kernel. They are distinct „black boxes“that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works.

Der Linuxkernel erlaubt es, Gerätetreiber zur Laufzeit als Modul zu laden oder den Gerätetreiber mit dem Kernel zu kompilieren und ins Kernelimage zu integrieren. Bei letztgenannter Methode wird der Treiber während des Bootvorganges initialisiert.

Auf Gerätetreiber wird in Unix-Systemen über das Dateisystem zugegriffen. Dazu werden virtuelle Dateien im Verzeichnisbaum – üblicherweise im Verzeichnis /dev – angelegt. Die Unterscheidung der Gerätetreiber erfolgt über sogenannte Major-Nummern. Ein Teil dieser Major-Nummern sind fest bestimmten Gerätetypen zugeordnet [T$^+$02, devices.txt]. Dem Framebuffertreiber ist die Major-Nummer 29, dem Video4Linux-Treiber die Major-Nummer 81 zugeordnet.

Die Minor-Nummer hat für die jeweiligen Gerätetreiber spezifische Bedeutung. Während bei Blockgeräten, wie Festplatten die einzelnen Partitionen über die Minor-Nummer angesprochen werden, erhält bei Framebuffer-

treibern jede Graphikkarte eine eigene Minor-Nummer. Der Bereich beider Nummern läuft von 0 bis 255.

Seit der Kernelversion 2.4 können diese Einträge beim Laden des jeweiligen Treibers durch das Modul devFS erzeugt werden. Bei der Standardinstallation des LART-Boards ist diese Option deaktiviert. Die Einträge müssen manuell mit dem Programm „mknod“angelegt werden. Für den ersten Framebuffer wäre der Eintrag /dev/fb0 mit der Major-Nummer 29 und der Minor-Nummer 0 anzulegen.

```
%mknod /dev/fb0 c 29 0
```

Die Option *c* weist auf ein Zeichengerät hin. Für Blockgeräte muss die Option b angegeben werden. Gerätetreiber werden in zwei Grundklassen eingeteilt:

Zeichengeräte arbeiten mit einem Zeichenstrom. Die Daten werden sequentiell gelesen oder geschrieben. Der Zugriff auf solche Geräte erfolgt über die Systemaufrufe *open*, *close*, *read* und *write*. Ein Beispiel für ein Zeichengerät ist der Consoletreiber. Die unterstützten Operationen und deren Einsprungspunkte werden mit der Struktur *file_operations* definiert. Der Treiber übergibt diese Struktur mit der Registrierung an den Kernel.

Blockgeräte arbeiten ähnlich den Zeichengeräten, verarbeiten jedoch Blöcke fester Grösse. Typische Blockgrössen sind Zweierpotenzen von 512 bis 64536. Die Blöcke können ähnlich dem Zeichengeräten gelesen und geschrieben werden. Jeder Block kann dabei adressiert werden. Dafür steht der Systemaufruf *seek* zur Verfügung. Beispiele für Blockgeräte sind die SCSI- und IDE-Treiber. Die unterstützten Operationen werden in der Struktur *block_device_operations* definiert.

Subsysteme stellen für verschiedene Geräte einer Klasse eine Grundfunktionalität bereit und definieren eine einheitliche Schnittstelle für den Zugriff auf die Geräte. Das SCSI-Subsystem unterstützt die Treiber für die verschiedenen Chipsätze mit einer gemeinsamen Schnittstelle und einer Verwaltungsebene. Für die Applikation bzw. hier die nachfolgende Treiberschicht CDROM, Disk usw. ist der Zugriff transparent, auch wenn mehrere SCSI-Treiber bedient werden. Häufig werden in der Schnittstelle Kommandos und deren Argumente, die an den Gerätetreiber über den Systemaufruf *ioctl* übergeben werden, spezifiziert.

3.3.1 Adressräume

Multitaskingbetriebssysteme nutzen für die Adressierung virtuelle Adressen. Für den Zugriff auf I/O Adressen müssen die realen Adressen in den virtuellen Adressraum eingeblendet werden.

Linux unterscheidet dabei zwischen (Vgl. [T+02, IO-mapping.txt]) :

Physische Adressen, wie Sie auf dem Speicherbus außerhalb des Prozessors existieren. Mit diesen Adressen greift die CPU auf den Speicher zu.

Virtuelle Adressen, die über die MMU des Prozessors in physische Adressen gewandelt werden. Diese Adressen existieren nur innerhalb des Prozessors.

Busadressen sind die Adressen der externen Geräte, mit denen Sie auf den Bus zugreifen. Für den Strongarmspeicherbus sind physische Adressen und Busadressen gleich.

3.3.2 Interrupts

Ein Gerätetreiber behandelt Anfragen von Applikationen *(system calls)* und Hardwareunterbrechnungen *(interrupts)*. Anhand der Richtung der Anfragen werden die Anforderungen unterteilt in *top halfs*, den Systemaufrufen des Betriebssystems, und *bottom halfs*, den Hardwareanforderungen. Die Treiberfunktionalität kann so in zwei Schichten – *Prozess und Interrupt-Ebene* – unterteilt werden. Die Schichten arbeiten meist asynchron. Da sich beide Schichten einen gemeinsamen Status teilen, müssen Zugriffe auf den *shared* Bereich synchronisiert werden.

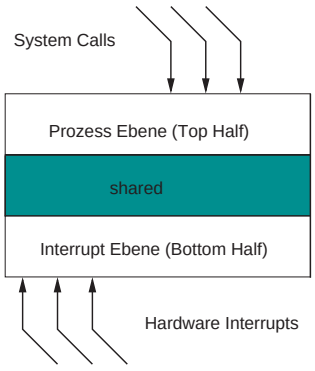

Abbildung 3.3: Prozess- und Interruptebene (nach [Mar99])

Für die Synchronisation sind Warteschlangen, Semaphoren und Unterbrechungssperren oft ausreichende Mechanismen. Werden mehrere *top half* Zugriffe vom Gerätetreiber zugelassen, müssen ebenfalls entsprechende Synchronisationsmethoden implementiert werden.

Linux verwendet in diesem Zusammenhang den sogenannten *Bottom-Half-Handler* – Softwareinterrupts, die nach der Abarbeitung zeitkritischer Hardwareinterrupts zeitunkritische Funktionen auslösen können.

3.4 Framebuffer

Grafikkarten besitzen meist einen eigenen Grafikspeicher, in dem die Bilddaten für das Bild – einen *Frame* – liegen. Für diese framebuffer-basierten Ausgabegeräte wurde das Framebuffer Device Subsystem geschaffen, das seit der Version 2.1.107 fester Bestandteil des Linuxkernel ist. Applikationen können über eine hardwareabstrahierte Schnittstelle auf die Ausgabegeräte zugreifen. Ein Framebuffer stellt sich als linearer Speicherbereich mit standardisierter Kodierung der Pixelwerte dar.

3.4.1 Das aktuelle Framebuffer Subsystem

Die Implementation des Framebuffer Device Subsystem entspricht dem seit der Version 2.1.107 im Kernel befindlichen Subsystem.

Ein Framebuffertreiber ist ein Zeichengerät. Es unterstützt dem entsprechend *read* und *write* Operationen auf dem Framebufferspeicher. Über die Systemoperation *ioctl* können die Parameter des Gerätes gelesen und verändert werden. Zusätzlich bietet die Schnittstelle die Möglichkeit den Grafikspeicher in den Adressraum der Applikation mittels *mmap* einzublenden. Diese Methode ist effizienter als *read* und *write*, da die Daten nicht von und zum Userspace kopiert werden müssen.

Das Framebuffer Device Subsystem wurde im ersten Schritt für die Anbindung des Console Subsystems an framebuffer-basierte Geräte geschaffen. Das Console Subsystem greift über den den Framebuffer Console Treiber auf die Framebuffergeräte zu. Da die Console nur max. 16 Farben benutzt, wurden Farbtabellen eingeführt, die die 16 Farben auf einzelne darzustellende Farben des Framebuffergerätes abbildet. Ein geringer Anteil der Geräte unterstützt solche Farbtabellen hardwareseitig.

Beim Start des Framebuffersubsystem werden die Consoletreiber automatisch mit eingebunden. Die Implementierung des Subsystems lässt eine Funktion ohne Consoletreiber nicht zu. Dieser Entwurf ist unglücklich gewählt, da vor allem Embedded Devices keine Verwendung für diese Consolen haben.

Es werden lediglich Resourcen verbraucht. Problematisch ist die notwendige Implementation von Funktionsaufrufen für die Consoletreiber in den Hardwaretreibern des Framebuffersubsystems. Die Funktionsaufrufe sind nicht optional und in den ersten Versionen des Framebuffersubsystem gab es keine generischen Funktionen im Subsystem. Als Folge verwenden alle Hardwaretreiber fast den gleichen Code für diese Funktionsaufrufe. Eventuelle Fehler sind dadurch in vielen verschiedenen Treibern zu korrigieren, wovon einige nicht im Standardkernel übernommen wurden. Nachträglich wurden einige generische Funktionen in der Datei fbcon.c geschaffen, die dieses Manko beheben soll. Bisher verwendet kein Treiber des Kernelbaums diese Funktionen.

Die Consoletreiber verwenden eine eigene Struktur *display* für die Hardwareparameter. Applikationen bedienen sich der Strukturen *fb_fix_screeninfo* und *fb_var_screeninfo* für das Lesen und Setzen der Parameter, die alle Parameter der Struktur *display* enthalten. Diese Redundanzen sind Fehlerquellen und haben keinen praktischen Nutzen.

Die Strukturen *fb_fix_screeninfo* enthält durch den Nutzer nicht veränderbare Eigenschaften, wie die Framebufferadresse und Zeilenlänge. Die Struktur *fb_var_screeninfo* enthält veränderbare Eigenschaften des Framebuffertreiber. Der Treiber unterscheidet zwischen sichtbaren Bildbereich und virtuellen Bildbereich. Der Grafikspeicher wird nach dem virtuellen Bildbereich aufgeteilt. Die in *fb_fix_screeninfo* angegebene Zeilenlänge bezieht sich daher auf den virtuelle Bildbereich.

3.4.2 Das Framebuffer Subsystem im Kernel 2.5

Die im letzten Abschnitt angesprochenen Mängel veranlassten James Simmons eine bessere Trennung der einzelnen Subsysteme zu entwerfen. In einer E-Mail vom 23.11.2001 schreibt er auf der Framebuffer-Mailingliste [Sim02a] :

> ... Well it ended up as a total rewrite of the tty/console layer. In the new design the the tty/console layer is composed of seperate subsystems which can exist independently outside of the tty layer. Thus the tty layer is constructed from these subsystems. This makes for a cleaner mor emodular design. Some of things done are:
>
> 1) New framebuffer api. This new api allows the fbramebuffer layer to exist without a framebuffer console. This makes for a

> much simpler api and much smaller code. Plus on embedded devices like a iPAQ have a VT console doesn't make sense. ...

Seit der Version 2.5.2 befindet sich das neue Subsystem im Kernel. Neben überarbeiteten generischen Funktionen wurde die Schnittstelle um Operation für die Nutzung von hardwarebeschleunigten 2D Funktionen erweitert. Die Framebuffer-API wird vollständig von dem Consoletreiber-Subsystem getrennt, so dass der Framebuffertreiber ohne Consoletreiber lauffähig ist.

Die Änderungen am Design sind noch nicht vollständig abgeschlossen. In den Headerdateien existiert bereits ein Hinweis darauf, dass die Struktur *display* entfernt wird. Zur Zeit muss sie aber noch benutzt werden. Einige Erweiterungen für Embedded Geräte werden in [Sim02b] beschrieben.

3.5 Video4Linux

Der Name Video4Linux ist ein wenig irreführend. Video4Linux soll eine API für Multimediageräte sein. Dazu gehören neben TV- und Videograbbertreiber auch Treiber für Radiokarten, Komprimierer, Dekomprimierer, Effekthardware und Teletext. Video4Linux ist kein Subsystem, das eigene Funktionalität oder Protokolle implementiert. Es werden lediglich die Treiber verwaltet und Applikationaufrufe an die Treiber weitergeleitet. Es stellt mehr eine Spezifkation der Schnittstellen dar.

Aufgrund der historischen Entwicklung der API stehen zur Zeit zwei inkompatible Versionen zur Verfügung, die in den nächsten beiden Abschnitten erörtert werden. Dabei konzentriert sich der Blick auf Videoeingabegeräte.

3.5.1 Die Video4Linux-API

Die erste Version der API entstand aus der Notwendigkeit den bttv-Treiber, einen Treiber für TV-Karten mit Booktree Chipsatz, in den Kernel zu integrieren. Künfigen Applikationen sollte eine einheitliche Schnittstelle zu den Treibern bereitgestellt werden. Auf Grundlage des bttv-Treibers wurde die Video4Linux-API entworfen. Gerd Knorr schreibt dazu auf [Kno02] :

> Late in the 2.1.x cycle Alan Cox stepped in with the idea to create a common API for that kind of hardware. video4linux was born. The idea is great. But it turned out later that the actual implementation has a number of flaws. Basically the API was desiged too much along the lines of the existing bttv driver.

Diese API ([T+02, video4linux/API.html]) ist im derzeitig aktuellen Kernel 2.4.18 implementiert. Als Zeichengerät kann auf die Treiber über *read* und *write* zugegriffen werden, wobei je nach Art des Gerätes eine der beiden Operationen keinen Sinn machen kann. Die Operation gibt dann als Rückgabewert den Fehler *Invalid Value* zurück. Mit der Operation *mmap* wird Applikationen der direkte Zugriff auf den Framebufferspeicher ermöglicht, indem der Framebuffer in den Nutzeradressraum eingeblendet wird. Es kann Speicher für mehrere Frames angefordert werden, wobei die maximale Anzahl der Frames von der Hardware und dem Video4Linux-Treiber abhängig ist. Während der Verarbeitung eines Frames können bereits weitere Frames angefordert werden. Hier offenbart sich ein gravierender Designfehler. Nach der Anforderung der Frames und Einblenden in den Applikationsadressraum dürfen die Eigenschaften des Bildes verändert werden. Einstellung an Grösse und Auflösung verändern aber meist auch die Grösse des benötigten Speichers für einen Frame.

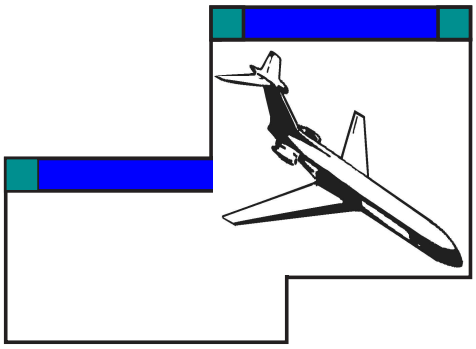

Abbildung 3.4: einfaches Overlay

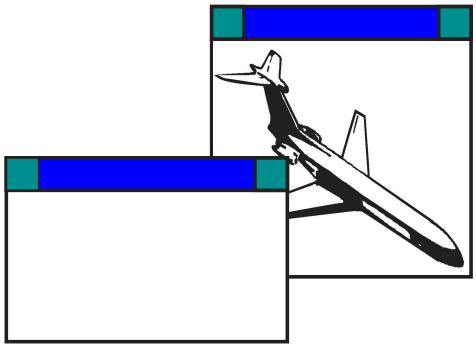

Abbildung 3.5: Overlay mit Clipping oder Chromakey

Eine weitere Methode ist das Einblenden des Videobildes der Grafikkarte. Für diese Methode gibt es je nach Hardware verschiedene Ansätze. Beim *Overlaying* Verfahren überlagert die Grafikkartenhardware das Videobild aus dem Video4Linux-Treiber bei der Videoausgabe. Overlaying muss von der Grafikkarte unterstützt werden.

Der Speicher der Grafikkarte kann auch an den Video4Linux-Treiber übergeben werden. Die Videograbberhardware schreibt die Daten dann direkt in den Grafikkartenspeicher. Hier muss die entsprechende Funktionalität durch den Video4Linux-Treiber oder die Videohardware gestellt werden.

Bei der einfachen Variante des Overlaying bleibt das Videobild immer im Vordergrund (Abbildung 3.4). Es existieren Erweiterungen, das Videobild von anderen GUI-Fenstern überlagern zu lassen (Abbildung 3.5).

Clipping definiert über eine Liste von Rechtecken Bereiche, die durch das Video nicht überschrieben werden dürfen.

Chromakey definiert den transparenten Bereich mit einer festgelegten Farbe. Dieses Verfahren kommt aus der Videotechnik. Die Videotechnik bevorzugt als Farbe ein kräftiges Blau. In der Computertechnik hat sich ein grelles Purpur durchgesetzt.

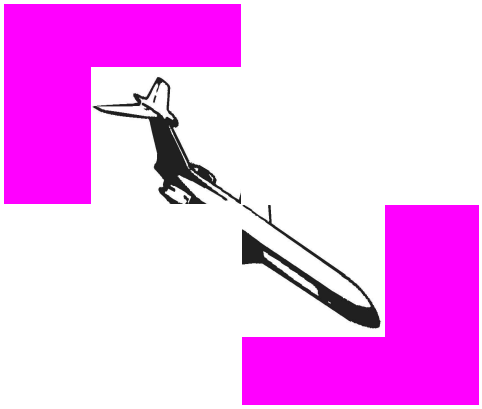

Abbildung 3.6: Chromakey Verfahren: die Farbe definiert den transparenten Bereich

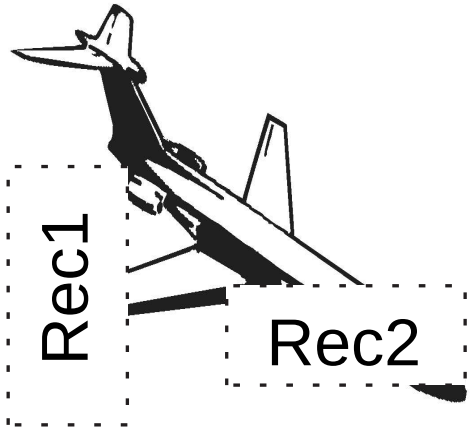

Abbildung 3.7: Clipping: um die angegebenen Bereiche herum wird das Bild geschrieben

Über *ioctl* Kommandos sind Informationen über das Gerät, Eingabekanäle und Bildeigenschaften erhältlich. Eingabekanal und Bildeigenschaften können verändert werden.

Wie bereits erwähnt, ist diese API seit der stabilen Version 2.2 Bestandteil des Kernel. Viele Applikationen greifen über diese API auf Multimediageräte zu.

3.5.2 Die Video4Linux2-API

Bill Dirks überarbeitete die Video4Linux-API bereits 1999. Die Unterstützung der verschiedenen Multimediageräte wurde erweitert. Die neue API unterteilt die Geräte in folgende Schnittstellen

- Video Capture Interface - dazu gehören alle Arten von Videograbbern aber auch Audio Karten.
- Video Effect Interface - für die Bildmanipulation.

- Video Codec Interface - Komprimierer und Dekomprimierer, und Geräte die das Video in anderer Weise verändern.
- Video Output Interface - Ausgabe von Bildströmen, die auch als Overlay im Framebuffer angezeigt werden können.
- Data Service Interface - Schnittstelle für Teletext, RDS etc.

Für die Videodigitalisierung ist das *Video Capture Interface* von besonderem Interesse. Auf die weiteren Schnittstellen wird im folgenden nicht weiter eingegangen.

Auffälligste Änderung ist die Möglichkeit, das Gerät von mehreren Applikationen zu öffnen. Die Operationen werden I/O und no-I/O Operationen unterschieden. Ein-/Ausgabe Operationen können weiterhin nur von einer Applikation aufgerufen werden. Die Eigenschaften des Gerätes *(no-IO Operationen)* können durch weitere Applikationen durchgeführt werden, sofern die Änderungen die Ein-/Ausgabe Operationen nicht tangieren. Für das Videograbbergerät bedeutet die Erweiterung, dass während des Einlesen von Bildern die Parameter, wie Kontrast, Helligkeit aber auch der Eingabekanal geändert werden können. Die Bildeigenschaften, wie Grösse und Farbformat, können nicht geändert werden, solange Ein-/Ausgabe Operationen laufen.

Der Zugriff auf die Bilder erfolgt wie bei der Video4Linux-API über *read*, *mmap* oder Overlay. Eine Erweiterung erfuhr das *mmap* Verfahren. Die Applikation beantragt weiterhin eine Anzahl von Bildpuffer, die über die Operation *mmap* in den Applikationsadressbereich eingeblendet werden. Die Puffer werden zum Füllen in eine Queue gestellt. Gefüllte Puffer enthalten dann zusätzliche Parameter, wie Zeitstempel, Bildzähler usw.

Die einstellbaren Eigenschaften der Hardware sind nicht mehr in einer fest definierten Struktur untergebracht. Die Informationen über die Eigenschaften müssen erst aus dem Treiber ausgelesen werden. Damit werden zusätzliche Einstellungen für die verschiedenen Geräte ermöglicht.

Weitere Neuerungen sind Informationsstrukturen für angeschlossene Geräte (Kamera), Zoom und Performanceinformationen.

Die Video4Linux2-API ist im Vergleich zur Video4Linux-API wesentlich umfangreicher und aufwendiger zu implementieren. Auch die Schnittstelle zu den Applikationen ist aufwendiger. Offensichtliche Designschwächen traten bei der Evaluierung nicht hervor.

3.6 Das I²C Subsystem

In Abschnitt 2.3.4 wurde das I²C-Protokoll erörtert. I²C ist ein Kommunikationsprotoll, das über einen Zweidrahtbus übertragen wird. Die Taktleitung

SCL sorgt für die Synchronistation von Master und Slave. Die Daten werden seriell über die Datenleitung SDA übertragen. Der Master kann eine Kommunikation mit einem beliebigen Slave initiieren. Angesprochen wird der Slave über eine 7Bit I^{2}C-Adresse.

Das I^{2}C Subsystems unterscheidet zwischen dem I^{2}C Businterface und dem Treiberinterface. Das I^{2}C Businterface stellt den Zugriff auf den I^{2}C Bus zur Verfügung. In Fall des LartVio erfolgt dieser Zugriff über den FPGA. Dabei wird zusätzlich zwischen verschiedenen Zugriffsalgorithmen unterschieden. Zur Veranschaulichung soll der i2c-algo-bit Treiber dienen. Dieser Treiber stellt die Grundfunktionalität für Adaptertreiber bereit, die die Takt- und Datenleitung (*SCL* und *SDA*) als Bitzugriffe ermöglichen. Der Adaptertreiber implementiert in vier Methoden die bitweisen Schreib- und Lesezugriffe auf die Leitungen. Der i2c-core Treiber enthält die Funktionalität des I^{2}C Protokolls und stellt Methoden für den Zugriff auf die Busadapter und Clienttreiber zur Verfügung.

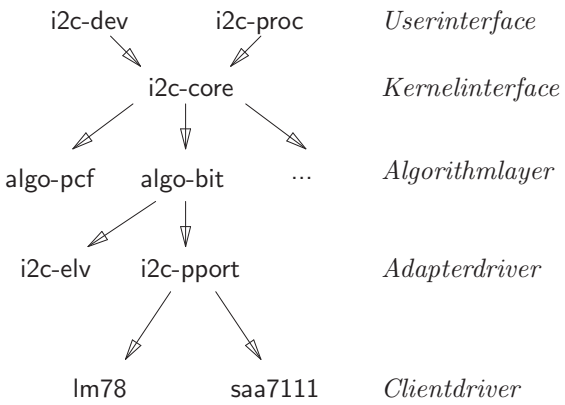

Abbildung 3.8: Struktur des I^{2}C Subsystems

Clienttreiber abstrahieren die Funktionalität des jeweiligen Chips. Diese Treiber werden über das Treiberinterface angesprochen. Wird ein Clienttreiber geladen übergibt der i2c-core Treiber jedem registrierten Busadapter den Clienttreiber. Der Busadapter sucht dann eine Kommunikation mit dem Chip auf seinem Bus. Kann eine Kommunikation aufgebaut werden, wird der Clienttreiber am Busadapter registriert.

Kapitel 4

Anforderungen an das BV-Board

Durch die aus den Kapiteln 1, 2 und 3 gewonnenen Kenntnisse von Hard- und Software sind die Möglichkeiten und Grenzen bekannt. Für den Framebuffer und die Videoeingabe wird die zu entwickelnde Funktionalität festgelegt.

In Abschnitt 1.1 wurden bereits einige Anforderungen an das Produkt gestellt. Ausgehend von den Zielgruppen, die das Gerät benutzen, können folgende Aussagen getroffen werden.

Private Anwender und Ausbildung

- Die Grundfunktionalität muss für Anwender ohne tiefe Einarbeitung erreichbar sein.
- Die Treiber müssen kompatibel zu Standardsoftware sein, um fertige Produkte ohne Anpassungen nutzen zu können.

Forschung und Industie

- Forschung und Industie wünschen zusätzlich Möglichkeiten der Performanceoptimierung Ihrer Produkte.
- Die Funktionalität der Treiber muss erweiterbar sein.
- Für Portierungen bereits vorhandener Software ist die bereits bei privaten Anwendern und Ausbildung erwähnte Kompatibilität zu Standardsoftware notwendig.

4.1 Videoeingabe

Bereits in der Aufgabenstellung enthalten ist das Haupteinsatzgebiet Bildverarbeitung. Im Unterschied zu einfachen Multimediaanwendungen wird das

Bild in mehreren Stufen verarbeitet. In Bildvorverarbeitung, Segmentierung und Merkmalsextraktion werden die Bilder bearbeitet. Der Datenstrom entspicht einem Vielfachen der Bildgrösse, zumal bei jeder Verarbeitung das Bild eingelesen wird und die Ergebnisse geschrieben werden. Die Verarbeitung PAL-Bild mit der Auflösung 720x576 Bildpunkten und einer Farbtiefe von 24 Bit soll diese Zusammenhänge verdeutlichen. Dieses Bild ist ca. 1,3 MByte gross. Wird für die Bildvorverarbeitung und Segmentierung nur ein Filter, die üblicherweise mit 3x3 Matrizen arbeiten, benutzt, sind je Filter

$$(3 * 3) * 720 * 576 = 3.732.480$$

Pixel[1] einzulesen. Bei einer Farbtiefe von 24 Bit ist ein Pixel 3Byte groß. Es werden also mindestens

$$3Byte * 3.732.480 = 11.197.440Byte = 10,68MByte$$

an Daten für einen Filterprozess eingelesen. Ausgegeben werden

$$768 * 576 = 1,3MByte.$$

Für einen Filterprozess müssen ca. 12MByte an Daten übertragen werden. Diese Berechnung ist stark vereinfacht und gilt nicht für jeden Verarbeitungsprozess, zeigt aber das hohe Datenvolumen, das eine Bildverarbeitung zu bewältigen hat. Aus dieser Berechnung wird auch klar welchen Einfluss die Bildgröße auf die Berechnung hat. Bei einem Bild mit der Auslösung 360*288 veringert sich das Datenvolumen auf ein Viertel. Die Hardware sollte den Ansprüchen der Bildverbeitung entsprechend unterschiedliche Bildgrössen zur Verfügung stellen können.

Die Bildverarbeitung löst dieses Problem mit Regions of Interest (ROI), grenzt also den Bildbereich ein. Neben der Bildgrösse stellt sich die Frage, welches Farbformat für Bildverarbeitungssoftware am geeignetsten ist. Im Computerbereich dominiert das RGB-Format. Es nutzt die Farben Rot, Grün und Blau zur Beschreibung von Farbe und Helligkeit für einen Bildpunkt. In der Videotechnik ist das YUV-Format verbreitet. Es beschreibt einen Bildpunkt über die Helligkeit und zwei Farbdifferenzen. Der Vorteil für die Bildverarbeitung liegt in der Trennung von Helligkeit und Farbe. Unterschiedlich ausgeleuchtete Objekte sind in den Farbkanälen von den Helligkeitsschwankungen nicht so stark betroffen wie das RGB-Format. Für die Bildverarbeitung werden häufig die Farbmodelle HSI und HSV eingesetzt. Der Farbton wird in dem Hue-Kanal bespeichert, der den Winkel im Farbkreis angibt. Die Sättigung S gibt an, wie stark der jeweilige Farbton ausgeprägt ist. V bzw.

[1]Pixel = Bildpunkt

I entsprechen dem Helligkeitswert. Die vollständige Trennung des Farbtons erleichtert die Segmentierung ohne grösseren Aufwand bei der Vorverarbeitung. Das Farbformat hängt stark vom Einsatzgebiet der Bildverarbeitung ab. So ist für die Erkennung von Schrift ein Graustufenbild ausreichend. Es verringert gegenüber anderen Farbformaten das zu bearbeitende Datenvolumen.

Eine Aussage zu treffen, welches Farbformat für die Bildverarbeitung geeignet ist, muss der jeweiligen Implementation überlassen werden. Obwohl das RGB-Format die Segmentierung erschwert, ist es doch das in der Computerindustrie verbreitete Format. Für dieses Format spricht ebenfalls, dass der Framebuffer mit diesem Format arbeitet.

Ein weiterer Unterschied besteht in der Speicherung der Farbkanäle. Die Farbkanäle können als Folge für jeden Pixel oder in Feldern (Planes) abgelegt werden. Bei der Speicherung in Planes werden die Farbwerte separiert. Ein Bildpunkt i erhält seine Information aus dem $i$ten Wert der jeweiligen Farbkanalfelder. Diese Technik vereinfacht die Segmentierung, da die Farbkanäle getrennt vorliegen. Nachteilig wirkt sich der Speicherzugriff auf die Lese- und Schreiboptimierung des Prozessors aus. Da in verschiedenen Bereichen gelesen und geschrieben wird, sind längere Burstzugriffe nicht möglich.

Folgende Funktionalität wird für die Videoeingabe zu implementieren sein:

Bildformate

- 720x576 und QSIF 360x288 bei 25 Bildern/s
- Deinterlacing der Videobilder.
- optional Cropping (Bildausschnitt).

Farbformate

- RGB24, andere Formate optional.
- Es muss eine Schnittstelle entwickelt werden, die die Implementierung zusätzlicher Farbformate in den Treibern vereinfacht.
- optional Speicherung in Planes

Performance

- Der Prozessor muss genügend Leistungsreserven für die Verarbeitung der Bilder haben.

Bildeigenschaften

- Helligkeit, Kontrast, Sättigung sollten über den Treiber einstellbar sein.

Zugriff auf Bilder

- Eine einfache Methode Bilder einzulesen. Sie soll den Einstieg mit wenigen Codezeilen Erfolge ermöglichen.
- Für performanceintensive Programme werden schnelle Methoden für den Bildzugriff benötigt. Sofern die einfache Methode dies nicht gewährleistet, sind entsprechende Algorithmen zu entwicklen.
- optional das direkte Einblenden des Videostroms in den Framebuffer.
- die Hardware muss ausserhalb des Betriebes in einen Schlafmodus geschaltet werden.

4.2 Videoausgabe

Die Videoausgabe soll Bild- und Grafikdaten auf den Bildschirm bringen. Der Framebuffertreiber stellt unter Linux eine Schnittstelle für Standardsoftware zur Verfügung. Zu dieser Kategorie zählen unter anderem XFree86 und QT-Embedded. Der Framebuffertreiber stellt jedoch lediglich Informationen über den Aufbau des Grafikspeichers zur Verfügung. Die Software muss mit den vorgefundenen Einstellungen den Grafikspeicher schreiben können. Für das jeweilige Farbformat müssen entsprechende Algorithmen vorhanden sein. QT-Embedded als Beispiel kann mit den Farbformaten RGB8, RGB16, RGB24 und RGB32 umgehen. Die Speicherung in Planes beherscht QT-Embedded nicht. Die Information wird von QT-Embedded ignoriert und das Bild weiterhin in fortlaufenden RGB-Werten gespeichert.

Der Framebuffer sollte mindestens das grösste Bild der Videoeingabe darstellen können. Da die Übertragung eines vollen PAL-Bildes den Bus entsprechend stark belastet, ist eine kleinere Darstellung ebenfalls sinnvoll. Desweiteren muss der Framebuffer abschaltbar sein. Im Gegensatz zur Videoeingabe definiert sich die Dauer Hardwarenutzung nicht über die Nutzung des Treibers. Wenn eine Applikation den Zugriff auf den Framebuffer beendet, muss dieser weiterhin das Bild anzeigen. Zusammengefasst werden folgende Anforderungen an die Videoeingabe gestellt:

Bildformate

- mindestens die Maximalgrösse der Videoeingabe, also 720x576

- Interlacing des Videobilder.
- optional QSIF 360x288

Farbformate

- RGB24
- optional Speicherung in Planes, sofern die Videoeingabe diese Funktionalität erhält.

Performance

- Der Prozessor muss genügend Leistungsreserven für die Verarbeitung der Bilder haben. Die Anforderung verschärft sich gegenüber der Videoeingabe, da sowohl Videoeingabe, Videoausgabe und Bildverarbeitung gleichzeitig arbeiten.

Bildeigenschaften

- optional Helligkeit, Kontrast, Sättigung einstellbar. Das Framebuffersubsystem stellt keine Schnittstelle für solche Einstellungen zur Verfügung.

Zugriff auf Bilder

- eine einfache Methode Bilder auszugeben und zurückzulesen.
- wahlfreien Zugriff auf den Grafikspeicher.

Leistungsaufnahme

- die Videoausgabe muss an- und abschaltbar sein.
- optional eine automatische Abschaltung nach einer definierten Zeitspanne.

Kapitel 5

Entwurf

Nachdem die Entwicklungsumgebung in den letzten beiden Kapitel beleuchtet wurde, wird in diesem Kapitel der Entwurf von Soft- und Hardware beschrieben. In den ersten Abschnitten werden einige Grundüberlegungen zu den Grenzen der Hardware und das Anwendungsgebiet angestellt, bevor der eigentliche Entwurf dargelegt wird. Die Soft- und Hardware wird in einzelne Module geteilt und deren Aufgaben definiert. Im letzten Abschnitt werden notwendige Testumgebungen für Soft- und Hardware vorgestellt.

5.1 Grundentwurf

Für die Betrachtung wurde der Videocontroller des LartVio in zwei unabhängige Funktionsgruppen geteilt, der Videoeingabe und Videoausgabe. Beide Module verbinden die jeweiligen Chips mit dem Speicherbus des Lart-Board. Die im Hauptspeicher des Lart-Board befindlichen Grafikdaten müssen von bzw. zum Modul des Videocontrollers übertragen werden. Der Videodecoder liefert im aktiven Bildbereich alle 37ns ein Byte. Nach 148ns steht jeweils einen Doppelpixel im YCbCr Format (32 Bit) zur Verfügung. Der Videoencoder erwartet im gleichen Abstand jeweils ein Doppelpixel. Für die folgenden Betrachtungen ist die Datenrichtung in vielen Fällen unerheblich. Die Videoeingabe und die Videoausgabe werden im folgenden als zwei Videomodule behandelt, die Daten einlesen. Abbildung 5.1 zeigt den Datenfluss auf dem Speicherbus.

Die Videomodule liefern einen YCbCr-Datenstrom, der im Prozessor in das entsprechende Farbformat gewandelt und im Hauptspeicher abgelegt wird. Der Prozessor konvertiert den Videostrom in das Farbformat RGB24. Die Busbelastung liegt in dieser Konstellation bei 128.75 MByte/s.

Ein Bild der Grösse 720x576 Bildpunkten mit 16Bit Pixelbreite benötigt

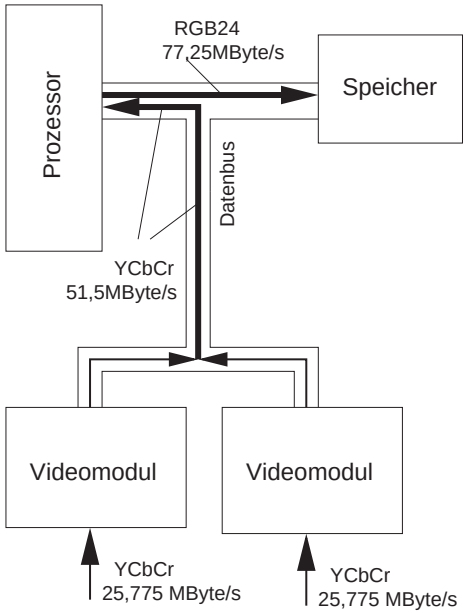

Abbildung 5.1: Datenfluss der abstrahierten Videomodule. Der Prozessor verarbeitet den Videostrom

810 KByte/Bild. 25 Bilder/s, die bei einer Bildwiederholrate von 25Hz übertragen werden, erzeugen einen durchschnittlichen Datenstrom von 19.8 MByte/s. Die Differenz zur in Abbildung 5.1 angegebenen Transferleistung ergibt sich durch die horizontalen und vertikalen Austastzeiten. Horizontal werden in jeder Bildzeile ca. 52 µs Bilddaten – 704 wirklich aktive Bildpunkte – übertragen. Innerhalb der 12 µs horizontalen Austastzeit, bestehend aus vorderer Schwarzschulter, Zeilensynchronimpuls und hinterer Schwarzschulter, werden keine Bildinhalte übertragen. Der Encoder empfängt in einer Zeile 720 Bildpunkte im Abstand von 74ns.

$$\frac{720 * 2Byte}{720 * 74 * 10^{-9}s} = 25,775MByte/s$$

Der durch den Prozessor in das Farbformat RGB24 konvertierte Datenstrom ist $\frac{2}{3}$ größer als der YCbCr-Datenstrom. Der gesamte Datestrom ist demnach

$$2 * 25,775MByte/s + \frac{2}{3} * 2 * 25,775MByte/s = 128,875Mbyte/s.$$

Von den 625 Zeilen einen Bildes nach CCIR werden 576 Zeilen für das aktive Bild verwendet. Die Zeit des aktiven Bildbereichs eines Halbbildes beträgt

$$288 * 64\mu s = 18.43\mu s.$$

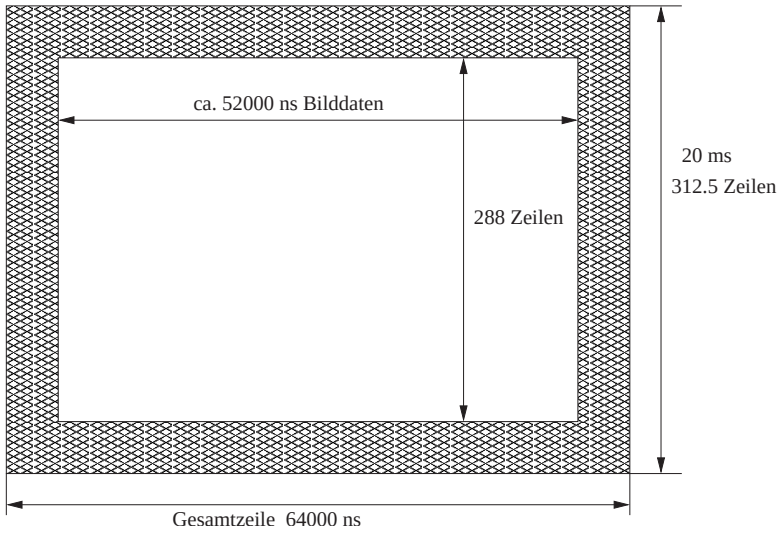

Abbildung 5.2: aktive Bild- und Austastzeiten eines Halbbildes

Abbildung 5.2 veranschaulicht diese zeitlichen Zusammenhänge.

Werden die Austastzeiten für die Übertragung mit genutzt, reduziert sich die maximale Datentransferrate durch die zeitliche Verteilung des Datenstroms. Die notwendige Grösse des Puffers ergibt sich aus

t_{leer}	der Zeit die bisher für die Übertragung nicht genutzt wurde.
t_{ges}	die Gesamtzeit die für die Übertragung zur Verfügung steht.
n_U	die Anzahl der Bytes die in der Zeit t_{ges} zu übertragen sind.
n_{buf}	der notwendige Puffer, die Übertragung auf t_{leer} auszuweiten.

$$n_{buf} = \frac{t_{leer} * n_U}{t_{ges}}.$$

Für eine Pufferung der Daten in der horizontalen Austastzeit müssen ca.

$$\frac{(64000ns - 720 * 74ns) * 720 * 2Byte}{64000ns} = 242Byte$$

zwischengespeichert werden. Die Datentransferrate zwischen Prozessor und Videomodulen reduziert sich auf

$$2 * \frac{720 * 2}{64 * 10^{-6}s} = 42,92MByte/s.$$

Zwischen Prozessor und Speicher wird der Speicherbus noch mit

$$\frac{3}{2} * 42,92MByte/s = 64.38MByte/s$$

belastet. Die Maximalbelastung des Speicherbusses liegt bei 107,3 MByte/s.

Soll die vertikale Austastphase ebenfalls zur Reduzierung der maximalen Datentransferrate genutzt werden, wird ein Puffer der Grösse

$$\frac{((312.5 * 64\mu s - 288 * 64\mu s) * 720 * 288 * 2Byte}{312.5 * 64\mu s} = 32514Byte$$

benötigt. Die Datentransferrate eines Videostroms reduziert sich damit auf 19.77 MByte/s. Die Last auf dem Speicherbus beträgt noch

$$2 * 19,77MByte/s = 39.55MByte/s$$

für den Datenstrom zwischen Prozessor und Videomodulen. Der im Prozessor nach RGB24 gewandelten Videostrom belastet den Speicherbus mit

$$39.55MByte/s * \frac{3}{2} = 59.33MByte/s.$$

Die Maximalbelastung des Speicherbusses reduziert sich auf 98.88 MByte/s.

Der Videocontroller ACEX 1K besitzt in der größten Ausführung 6 KByte Ram. Für die Pufferung der vertikalen Austastphase werden jedoch ca. 2 * 32 KByte RAM benötigt. Eine vollständige Pufferung vertikalen Austastzeit ist nicht möglich. Ausgehend von der in Abschnitt 2.2.3 ermittelten Datentransferleistung von ca. 102 MByte/s sind die bisherigen Ergebnisse für die Videoeingabe und Videoausgabe nicht zufriedenstellend. Die benötigten Datentransferraten übersteigen die real erreichbaren Datentransferraten.

5.2 Busmasterbetrieb

In Abschnitt 2.4 wurde das SA-1100 Multimedia Development Board vorgestellt. Die Grafikausgabe auf der Erweiterungplatine *SA-1101 Development Board* steuert den Speicher im Busmastermodus an. Der Prozessor ist bei den Speicherzugriffen unbeteiligt. Diese Methode reduziert den Datenstrom um den Datenfluss zum Prozessor. In Abbildung 5.3 wurde diese Methode auf die Videomodule angewandt. Ein Puffer für die Nutzung der horizontalen Austastphase wurde bereits berücksichtigt. Der Busmaster verwaltet die Zugriffsanfragen beider Videomodule auf den Speicher und legt die Datenelemente im Speicher ab.

Die Datentransferrate reduziert sich auf $64.38MByte/s$, sofern die Busmasterimplementierung die Datentransferraten des StrongARM erreicht. Geht man von den Speicherzugriffen des Prozessors aus, stehen ca. 30% der gesamten Datentransferleistung für den Prozessor zur Verfügung, wenn beide Videomodule aktiv sind.

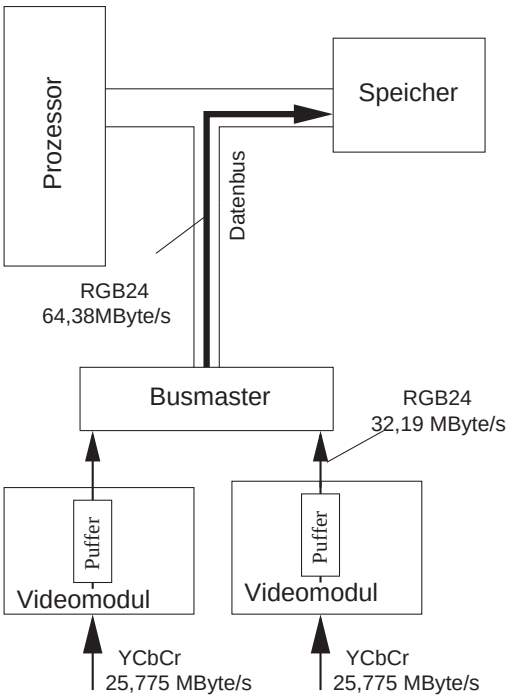

Abbildung 5.3: Datenfluss der abstrahierten Videomodule mit Puffer zur Nutzung der horizontalen Austastphase. Ein DMA-Modul koordiniert die Speicherzugriffe.

5.3 Module des Videocontrollers

Die Implementierung eines Busmasters zieht weitreichende Folgen mit sich. Da der Prozessor an der Datenübertragung nicht mehr beteiligt ist, muss jegliche Funktionalität, die den Datenstrom verarbeitet, in die Videomodule integriert werden. In Tabelle 5.1 werden die in Abschnitt 4 ausgeführten funktionalen Anforderungen zusammengefasst. Die Tabelle beschränkt sich auf die den Videostrom verändernden Funktionen.

Videoeingabe	**Videoausgabe**
Adresssteuerung	Adresssteuerung
Deinterlacing	Interlacing
Skalierung Bild/2 (optional: stufenlose Skalierung)	optionale Skalierung Bild/2
optional: Cropping	
Farbkonvertierung nach RGB24 (optional weitere Konverter)	Farbkonvertierung von RGB24 nach YCbCr
Schnittstelle für weitere Farbkonverter	
optional: Speicherung in Planes	optional: Lesen von Planes

Tabelle 5.1: Bildverändernde Funktionalität von Videoein- und Videoausgabe

Die Resourcen des FGPA ACEX1K sind begrenzt. Wie viele Resourcen die einzelnen Funktionen beanspruchen, war schwer abzuschätzen. Das Design wurde dementsprechend so entworfen, dass weiterere Funktionen nachträglich, möglichst ohne Änderungen an der vorhandenen Funktionalität, implementiert werden können.

Die Speicherung in Planes, also im Hauptspeicher getrennten Farbbereichen, wurde verworfen. Diese Funktion würde die Implementierung von Burstzugriffen erschweren, da ständig verschiedenen Speicherbereiche angesprochen werden. Der ACEX1K bietet nicht genügend Pufferspeicher um die Farbkanäle einzeln zu puffern.

Abbildung 5.4 zeigt den Datenfluss und die Anordnung der Module. Eine stufenlose Skalierung auf der Videoeingabeseite wurde weiterhin als optional angesehen, sollte beim Entwurf aber berücksichtigt werden. Stehen nach der Implementierung noch genügend Resourcen zur Verfügung, kann die Skalierung nachträglich implementiert werden. Diese Aussage trifft auch auf das Cropping zu. Die Skalierung auf halbe Bildgrösse in der Videoausgabe wurde mit aufgenommen. Sie reduziert den Datenstrom der Videoausgabe auf

dem Hauptspeicherbus um die Hälfte. Die einzelnen Pixel müssen nur als Doppelpixel an den Encoder übergeben werden.

Im Weiteren wird die Funktion der einzelnen Module des Entwurfs besprochen. Die Schnittstellen der Module sind in [KS02] ausführlich beschrieben.

5.3.1 DMA-Modul

Das DMA-Modul muss die Speicherzugriffsanfragen des Videoeingabemoduls und des Videoausgabemoduls verwalten. Es muss sicherstellen, dass jedes der beiden Module genügend Übertragungsresourcen erhält. Andernfalls wäre Datenverlust – und damit der Verlust des aktuellen Bildes – die Folge.

Neben der Koordinierung der Zugriffsanforderungen, muss das DMA-Modul die DRAM Bausteine im Schreibenden und Lesenden Modus ansteuern. Da die Adressen in einer Bildzeile fortlaufend sind, kann der Burst auf bis zu einer Bildschirmzeile verlängert werden. Dazu wird das Signal `newRow` eingeführt, das veranlassen soll den Burst zu beenden. Danach beginnt ein neuer Burst, der mit der Übertragung der Zeilenadresse eingeleitet wird.

Die Schnittstellen zu Videoeingabe und Videoausgabe sind sehr ähnlich. Lediglich die Datenrichtung ist unterschiedlich zu behandeln. Es wird im folgenden nur die Schnittstelle zur Videoeingabe beschrieben. `DEC_dma_req` fordert eine Datenübertragung zum Hauptspeicher an. Das Signal muss nur einen Takt lang aktiv sein und kann nur über `DEC_dma_exit_req` zurückgesetzt werden. Die Anforderung wird je nach Zustand des DMA-Moduls sofort bearbeitet oder die Anfrage notiert. Damit ein DMA-Zugriff eingeleitet wird, muss `DEC_dma_en` aktiv sein. Das Deaktivieren von `DEC_dma_en` beendet sofort die Datenübertragung.

5.3.2 Adressgenerator

Die Adressierung des Speichers wird über das Busmastermodul erfolgen. Daher erscheint es sinnvoll, den Adressgenerator der beiden Videomodule direkt an das Busmastermodul zu koppeln. Bei der Adressierung des Grafikspeichers wird von einem linearen Speicherbereich ausgegangen.

Interlacing und Deinterlacing kann durch die Zuordnung der Startadresse einer Zeile gelöst werden. Die Startadresse einer Zeile kann aus der Zeilennummer n_{Zeile} und der Anzahl der Bytes einer Zeile l_{Zeile} berechnet werden

$$Adr_{Zeile} = Adr_{FB} + n_{Zeile} * l_{Zeile}.$$

Adr_{FB} enspricht der Startadresse des Grafikspeichers. Diese Berechnung kann der Adressgenerator ebenfalls übernehmen. Da die Halbbilder einzeln

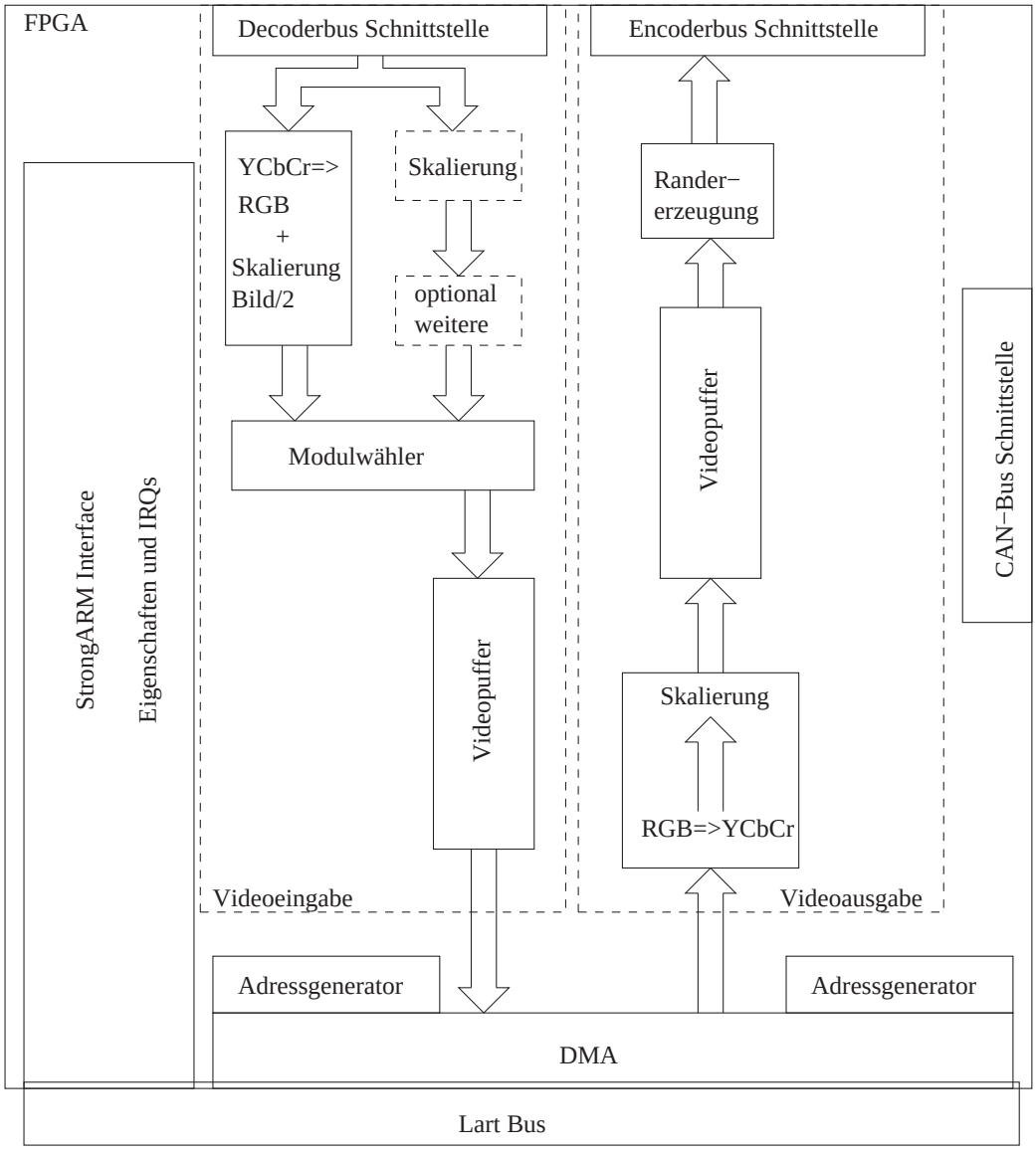

Abbildung 5.4: Entwurf des FGPA

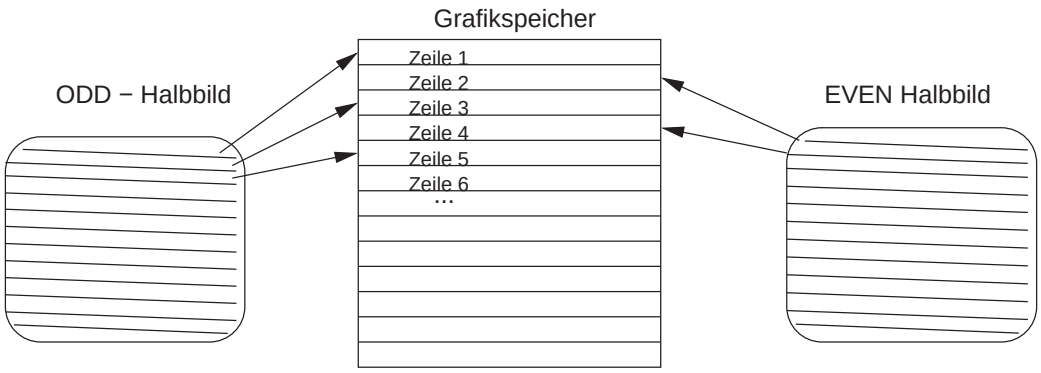

Abbildung 5.5: Interlacing und Deinterlacing

übertragen werden, ist es sinnvoll die Startadresse für das Halbbild mit geraden Zeilennummern und für das Halbbild mit ungeraden Zeilennummern vorzugeben. Dies ermöglicht die Speicherung der Halbbildern in verschiedenen Grafikspeichern.

Der Linux-Frambuffertreiber unterscheidet zwischen sichtbaren und virtuellen Auflösungen. Die virtuelle Auflösung ist mindestens so gross wie die sichtbare Auflösung sein. Die sichtbare Auflösung kann daher einen Zeilenoffset zur nächsten Zeile haben. Der Offset für das Interlacing und Deinterlacing muss durch die Softwaretreiber berechnet werden. Das Ende des Grafikspeichers muss ebenfalls berücksichtigt werden, um ein Überschreiben fremder Bereiche zu verhindern.

Für jedes Videomodul wird eine Adresssteuerung benötigt.

5.3.3 Datenpuffer

Da die Speicheransteuerung durch das Busmastermodul vorgenommen wird, können längere Burstzyklen als die *Burst Of Eight* Zyklen des Prozessor generiert werden. Die Datenpuffer müssen, neben der notwendige Grösse für die Pufferung der horizontalen Austastzeit, auch die Zeit für den Datentransfer zum Speicher berücksichtigen. Das gilt sowohl für das eigene Videomodul als auch für das beim Zugriff auf den Speicher konkurierende Videomodul.

Als Datenpuffer wurden nach dem FIFO-Prinzip Puffer vorgesehen. Die Daten die auf der einen Seite des Datenpuffer hineinfliessen, kommen in der gleichen Reihenfolge aus dem Datenpuffer wieder heraus. Im weiteren werden die Datenpuffer kurz FIFOs genannt. In diesem Abschnitt wird die notwendige Grösse der FIFOs betrachtet. Für die Betrachtung ist die Datenrichtung wieder unerheblich. Es wird davon ausgegangen, das die FIFOs die gleiche

Datenbusbreite wie der Speicherbus haben. Für die weiteren Berechnungen müssen einige Variablen definiert werden.

n_{FIFO}	Anzahl der Elemente im FIFO
n_{POP}	Anzahl der Elemente die bis zum Zeitpunkt n_{SIG} entfernt werden müssen
n_{SIG}	Anzahl der Elemente bei dem das Signal zum Leeren des FIFOs gegeben wird
n_{PSIG}	Anzahl der zu lesenden Elemente bei Signal n_{sig}
t_{POP}	Zeit für das Füllen von n_{tf} Elementen
t_{PX}	Zeit für das Auslesen eines Pixels (word)
n_{BCYC}	max. Anzahl der Col Zylken in einem Burst
t_{HPC}	Zeit für einen Spaltenzyklus (siehe auch Abschnitt 2.2.3)
t_{RCD}	Zeit für die Übergabe der Zeilenadresse (siehe auch Abschnitt 2.2.3)
t_{RP}	Zeit zwischen zwei Zeilenzyklen (siehe auch Abschnitt 2.2.3)
t_{MBREQ}	Zeit zwischen Anforderung des Busses(MBREQ) bis zur Freigabe (MGBGNT) (siehe Abschnitt 2.2.3)

FIFO Variablen erhalten zur Unterscheidung der beiden FIFOs eine Nummerierung 1 und 2. Wird der FIFO gefüllt muss ab einer bestimmten Marke n_{sig} das Busmastermodul den Datentransfer zum Hauptspeicher beginnen, um die aufgelaufenen Daten aus dem FIFO zu entfernen. Da zwei FIFOs parallel arbeiten, kann im ungünstigsten Fall n_{SIG_1} und n_{SIG_2} gleichzeitig eintreten. Der Busmaster muss diese Situation regeln, und einen der beiden FIFOs bevorzugen. In der Zeit des Datentransfer werden weiterhin Daten in die FIFOs geschrieben. t_{POP_1} ist die Zeit für das Leeren des ersten FIFOs und t_{POP_2} die Zeit für das Leeren des zweiten FIFOs.

Folgende Bedingungen müssen beim Betrieb des FIFO jederzeit erfüllt sein:

- Es darf nicht zu einem Unter- oder Überlauf des FIFO kommen, die Folge wäre Datenverlust.
- Bedingt durch den zweiten FIFO muss t_{POP_2} kleiner als der Abstand zwischen zwei Refresh-Zyklen des DRAM sein, da sonst Refresh-Zyklen verloren gehen würden. [1]

Es wird davon ausgegangen, dass

$$n_{PSIG} = n_{PSIG_1} = n_{PSIG_2}$$

[1] Es wird davon ausgegangen, dass die Refreshlogik des StrongARM genutzt wird.

sind. Die Anzahl der in den Hautpspeicher zu schreibenden Elemente ist danach n_{PSIG} zuzüglich der Zeit die für das Schreiben benötigt wird.

$$n_{POP_1} = n_{PSIG} + \frac{t_{POP_1}}{2t_{PX}} \tag{5.1}$$

t_{POP_1}, die Zeit für das Leeren des FIFO ergibt sich aus der Summe den Zeiten von Busanforderung, sowie den Zeiten für Zeilenzyklen und Spaltenzyklen.

$$t_{POP_1} = t_{MBREQ} + (\frac{n_{POP_1}}{n_{BCYC}} + 1)(t_{RCD} + t_{RP}) + n_{POP_1} t_{HPC}$$

oder nach Einsetzen von n_{POP_1}

$$t_{POP_1} = \frac{t_{MBREQ} + (\frac{n_{PSIG}}{n_{BCYC}} + 1)(t_{RCD} + t_{RP}) + n_{PSIG} t_{HPC}}{1 - \frac{1}{2t_{PX} n_{BCYC}} - \frac{1}{2t_{PX}}} \tag{5.2}$$

Der zweite FIFO wird in der Zeit t_{POP_1} noch entleert. Somit ist

$$n_{POP_2} = n_{PSIG} + \frac{t_{POP_1}}{2t_{PX}} + \frac{t_{POP_2}}{2t_{PX}}$$

und

$$t_{POP_2} = \frac{t_{MBREQ} + (\frac{n_{PSIG}}{n_{BCYC}} + \frac{t_{POP_1}}{2t_{PX} n_{BCYC}} + 1)(t_{RCD} + t_{RP}) + (n_{PSIG} + \frac{t_{POP_1}}{2t_{PX}}) t_{HPC}}{1 - \frac{1}{2t_{PX} n_{BCYC}} - \frac{1}{2t_{PX}}} \tag{5.3}$$

Abbildung 5.6 zeigt Werte für n_{FIFO} und n_{PSIG} zwischen 0 und 512 mit $t_{RP} = t_{HPC} = t_{RCD} = 2 * 18ns$ [2]. Die Anzahl der Spaltenzyklen in einem Burst t_{BCYC} wurde auf 128 gesetzt. t_{px} entspricht dem 2 * 37ns. Die FIFO-Größe n_{FIFO} von 512 Einträgen bei einem Signal n_{SIG} zwischen 128 und 256 Einträgen zeigt sich als guter Kompromiss zwischen maximalen Burstzeiten und den oben definierten Beschränkungen.

5.3.4 Videoeingabe

Die Videoeingabe erhält vom SAA7113 die Videodaten über einen 8Bit-Bus. Diese müssen ein ein 32Bit-Format gewandelt werden. Bevor die Videodaten in den FIFO gelangen, können Sie beliebig verarbeitet werden. Dafür wurde eine Schnittstelle entworfen, die für verschiedene Funktionen genutzt werden kann. Um die unterschiedlichen Vorverarbeitungsmodule ansprechen

[2] 18ns = Bustakt/2

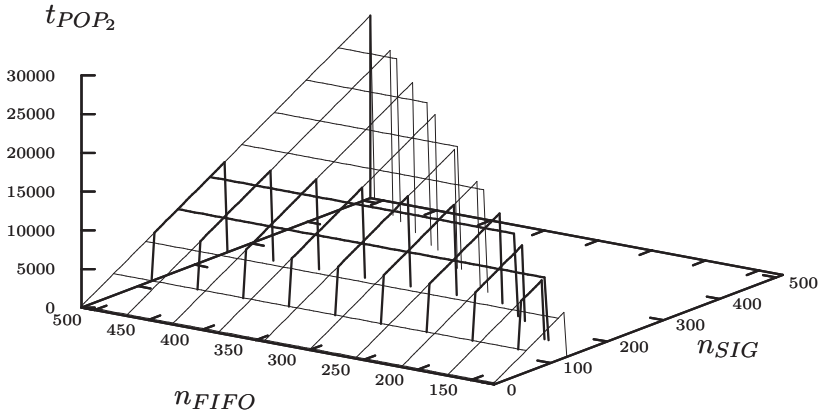

Abbildung 5.6: Abhängigkeit der Füllzeit von FIFO-Größe und Füllsignal

zu können wird ein Multiplexer in die Videoeingabe integriert, der über die Softwaretreiber gesteuert werden kann. Abbildung 5.7 zeigt den Datenfluss der Videoeingabe.

Die Schnittstelle der Videoeingabe wurde zusammen mit der Videoausgabeschnittstelle so abstrahiert, dass für das DMA-Modul die Anbindung beider Module einfach zu realisieren ist. Die Schnittstellen unterscheiden sich nur in der Datenrichtung.

Videodecoderbus

Der Videodecoder übergibt jeweils 8 Bit Werte. Diese müssen zu jeweils einem 32Bit Doppelpixel zusammengefasst und an die nachfolgenden Module weitergeleitet werden. Neben dem Datenbus kommen vom Videodecoder noch einige Steuerleitungen. Die Signale der Steuerleitungen sind intern zu synchronisiern und gegebenenfalls für eine einheitliche interne Steuerung zu bearbeiten.

Zusätzlich kann über den FPGA die Resetleitung des SAA7113 gesteuert werden und der Taktgenerator des Videodecoder ein- und ausgeschalten werden. Der SAA7113 bietet keine Stromsparfunktion. Diese wird durch die Abschaltung des Taktes und der Aktivierung des Resetsignals emuliert.

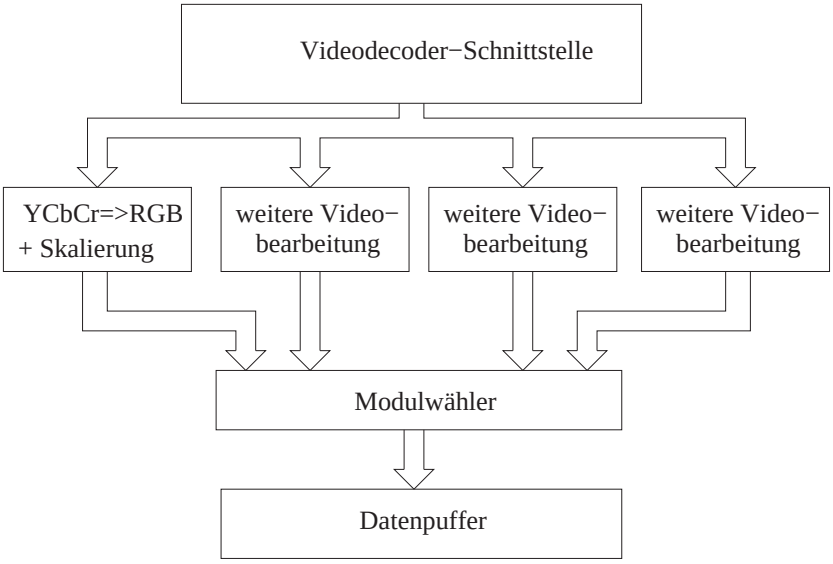

Abbildung 5.7: Datenfluss: Videoeingabe

Videobearbeitung

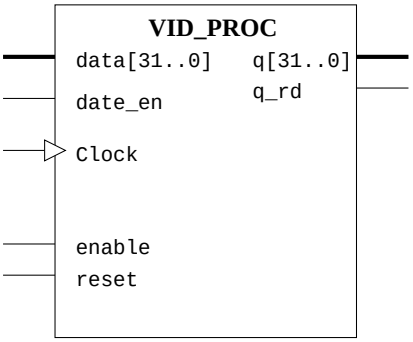

Abbildung 5.8: Die Schnittstelle: Videobearbeitung

Für die Manipulation des Videostroms wurde eine Schnittstelle entwickelt, die von der Videodecoderschnittstelle Daten empfangen und den veränderten Videostrom in den Datenpuffer schieben kann. Die Konvertierung des YCbCr-Formates nach RGB24 wird ein Modul sein, das diese Schnittstelle nutzt. Die Schnittstelle definiert alle notwendigen Signale, kann aber, je nach Modul, um zusätzliche Eigenschaften erweitert werden. Die entwickelte Schnittstelle kann ebenfalls bei einer mehrstufigen Verarbeitung verwendet werden. So ist es möglich eine Skalierung vor die Farbraumkonverter zu setzen.

In der Schnittstelle ist die mögliche Pipelinetiefe nicht definiert. Diese muss für den entsprechenden Einsatz definiert werden. Damit ist es unter anderem möglich diese Schnittstelle ebenfalls für die Videoverarbeitung auf der Videoausgabeseite zu verwenden.

5.3.5 Videoausgabe

Der Datenbus des ADV7171 ist 8 Bit breit. Die auszugebenden 32Bit-Videodaten müssen in Bytes gesplittet an den Datenbus des Encoder-Chip übergeben werden. Der ADV7171 erzeugt im Mastermodus die Videosynchronsignale. Die Videoeingabe kann sich darauf beschränken auf die Synchronsignale zu reagieren.

Die Farbkonvertierung liegt wie bei der Videoeingabe in Datenflussrichtung vor dem FIFO. Nachdem der FIFO das `full` Signal setzt, dürfen keine weiteren Daten mehr in den FIFO geschrieben werden. Sollten für die Farbkonvertierung eine oder mehrere Pipelinestufen nötig sein, muss das `full` Signal entsprechend der Pipelinetiefe früher gesetzt werden.

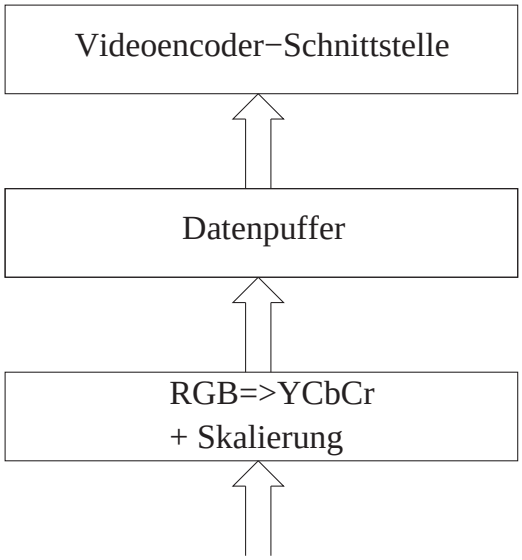

Abbildung 5.9: Datenfluss: Videoausgabe

Abbildung 5.9 zeigt den Datenfluss und die Bestandteile der Videoausgabe. Die Skalierung wurde in das Videobearbeitungsmodul integriert, da keine weiteren Module zur Veränderung des Videostromes vorgesehen sind.

5.3.6 StrongARM Interface

Für die Kommunikation mit dem Prozessor und die Einstellung der Eigenschaften wird ein Modul benötigt, das diese Kommunikation zum Prozessor übernimmt. Die Eigenschaften werden für den Prozessor als sichtbarer Speicher in der statischen Speicherbank 2 ansprechbar sein. Die Beschreibung der

Register kann [KS02] entnommen werden.

Neben den Eigenschaftsregistern benötigen die Treiber einige Interruptsignale für die Behandlung von Ereignissen. Diese werden ebenfalls durch das StrongARM Interface bereitgestellt. Dazu zählen die Videoeingabeinterrupts `PIC_START` und `PIC_STOP`, die den Beginn und das Ende eines eingelesenen Bildes signalisieren. Der Interrupt des CAN-Controllers muss an den Prozessor weitergereicht werden. Für den Interruptbetrieb stehen die GPIOs `LART_FORCE`, `LART_SPARE` und `LART_1HZ` zur Verfügung.

5.4 Gerätetreiber

Entsprechend der Aufgabenstellung wurde bei der Auswahl der Schnittstellen besonderer Wert auf die Funktion in zukünftigen Kernelversionen gelegt. Zum Entwurfszeitpunkt war die aktuelle Version des stabilen Kernels 2.4.18. Der Entwickerkernel war bei Version 2.5.20 angelangt, wobei dieser auf dem Lart Board nicht funktionsfähig war. Meldungen von Nachrichtenagenturen zuvolge sollen ab Oktober 2002 in den Entwicklerkernel 2.5.x keine neuen Funktionen mehr aufgenommen werden. Die Schnittstellen der Subdesigns werden ab diesem Zeitpunkt nach Möglichkeit nicht mehr geändert. Bis zu einer stabilen Version des Kernel vergingen bei den Versionen 2.1.x und 2.3.x etwa ein Jahr. Mit einer funktionsfähigen Version des Kernels kann ab etwa Januar 2003 gerechnet werden.

Um den Anforderungen einer leichten Portierbarkeit auf zukünftige Kernelversionen gerecht zu werden, wurden einige Grundsätze aufgestellt die bei der Entwicklung und Implementierung berücksichtigt wurden:

- Die Kernelsourcen dürfen nicht verändert werden.
- Eine Splittung in mehrere Treiberversionen für verschiedene Kernelversionen sollte vermieden werden, da es später den Aufwand erhöht, die Sourcen auf dem aktuellen Stand zu halten.
- Als Codingstyle wird der in [T$^+$02, CodingStyle.txt] beschriebene Stil verwendet, um die eventuelle Integration in den Kernelbaum zu erleichtern.
- Kommentare werden durchgehend in Englisch geschrieben.

Die Implementierung der Treiber kann in zwei Varianten erfolgen. Der Treiber kann als Patch in den Kernelbaum eingebunden werden oder als eigenständige Sourcen vorliegen. Ein Patch bietet Vorteile bei der Integration

der Sourcen in den Kernel. So können im Kernel vorhandene Sourcen angepasst werden und die Treiber in die Konfigurationstools des Kernel aufgenommen werden. Nachteilig ist der erhöhte Aufwand bei der Pflege der Patches, da sie an jede Kernelversion angepasst werden müssen. Eigenständige Sourcen bieten eine höhere Portabilität zwischen den einzelnen Kernelversionen. Sie integrieren sich jedoch nicht in die Konfigurationstools. Insbesondere Abhängigkeiten zu anderen Treibern und Subsystemen können nicht abgebildet werden.

Neben dem Framebuffertreiber und dem Video4Linux-Treiber sind weitere Gerätetreiber für die Ansteuerung des Encoder- und Decoderchips über den I^2C-Bus notwendig. Diese Treiber abstrahieren die möglichen Einstellungen der Chips und sind nicht mit dem Framebuffertreiber und dem Video4Linux-Treiber verbunden. Ein weiterer Treiber wird den Zugriff auf den I^2C-Bus des LartVIO gewährleisten. Abbildung 5.10 zeigt das Zusammenspiel der zu implementierenden Treiber untereinander und zu den Subsystemen.

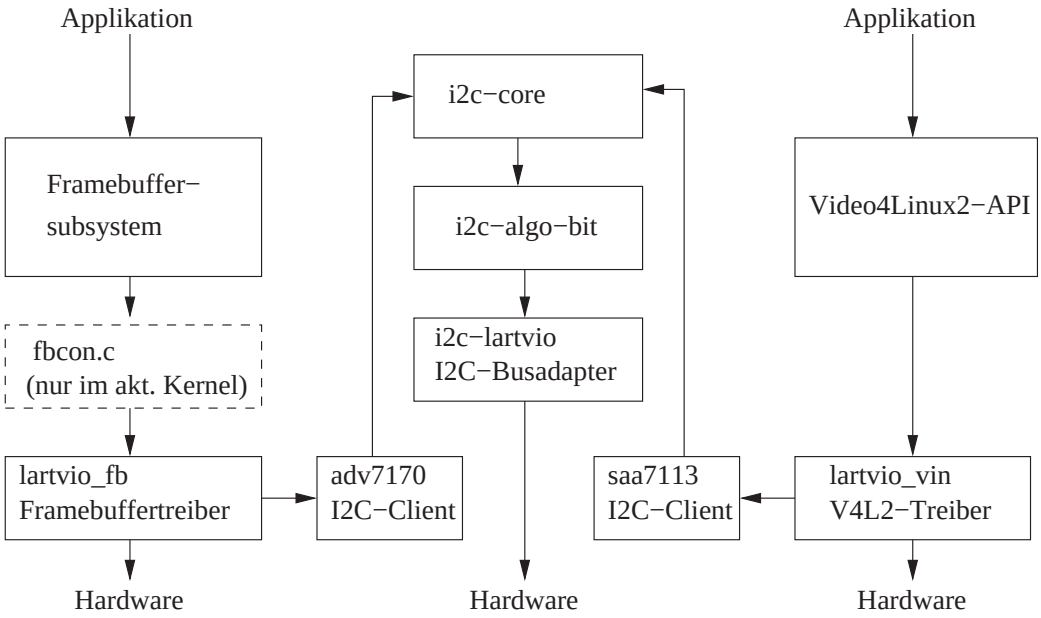

Abbildung 5.10: Übersicht über das Treiberdesign

Da ein Großteil der Funktionalität in den FPGA verschoben wurde, liegt die Hauptaufgabe der Treiber in der Abstraktion der Hardware, sowie den allgemeinen Gerätetreiberanforderungen wie in Abschnitt 3.3 beschrieben.

5.4.1 Framebuffer

In Abschitt 3.4.1 wurde das Framebuffer-Subsystem der aktuellen, stabilen Kernelversion untersucht. Die generischen Funktionen aus fbcon.c entsprechen in der Struktur in weiten Teilen dem Design in der aktuellen Entwickerversion des Framebuffersubsystems. Unterschiede liegen vor allem in den Übergabeparametern der Funktionen. Viele Funktionsaufrufe für die Console sind in das Subsystem integriert, können aber auch durch eigene Implementierungen ersetzt werden. Das Framebuffersystem der stabilen Kernelversion ist somit dem des Entwicklerkernel ähnlich.

Die Dokumentation der Hardwaretreiber für das Framebuffersubsystem reduziert sich auf die Kommentare des Skeletontreiber skeleton.c im Kernelbaum. Sie stellen ein Grundgerüst für die Implementierung eigener Treiber mit Quelltextdokumentation dar. Die Funktionen für Hardwarebeschleunigung können durch generische Funktionen ersetzt werden, da das FPGA-Design keine Hardwarebeschleunigung vorsieht.

Berücksichtigt werden muss weiterhin die Forderung zur Abschaltung der Videoausgabe. Die Schnittstelle zum Subdesign sieht dafür die Funktion *blank* vor.

5.4.2 Video4Linux

Die Video4Linux2-API wird seit 1999 entwickelt. Sie kann mit der stabilen Kernelversion 2.4.x compiliert und verwendet werden. Die Video4Linux2-API soll bis zur Veröffentlichung des stabilen Kernel 2.6 die alte Video4Linux-API ersetzen und stellt bereits deren Funktionalität über Kompatibilitätstreiber zur Verfügung. Ältere Treiber, die für die Video4Linux-API geschrieben wurden können somit weiter verwendet werden.

Die Video4Linux2-API ist in 3.5.2 ausführlich dokumentiert. Es müssen nur die API-Funktionen implementiert werden, die die Hardware unterstützt. Alle anderen Funktionen kehren mit Fehlerwert zurück. Für die Umsetzung der Funktionalität entsprechend den Anforderungen müssen folgende Funktionen der API realisiert werden:

- ***read*** soll den einfachen Zugriff auf Bilder ermöglichen.
- ***mmap*** bietet in der Video4Linux2-API im Zusammenhang mit den ioctl-Kommandos

  VIDIOC_REQBUFS und

  VIDIOC_QUERYBUF

die Möglichkeit mehrere Bildpuffer zu durch den Treiber verwalten zu lassen. Die Bildpuffer werden zum Füllen dem Treiber über das ioctl-Kommando

VIDIOC_QBUF gestellt und nach vollständer Bildübertragung mit **VIDIOC_DQBUF** wieder entnommen.

VIDIOC_QBUF und VIDIOC_DQBUF verwenden dabei Strukturen die den ensprechenden Bildpuffer und das eingelesene Bild beschreiben. Gestartet und gestoppt wird die Übertragung mittels

VIDIOC_STREAMON und
VIDIOC_STREAMOFF ioctl-Kommandos.

Die Applikation kann sich über die ***poll*** Funktion den Stand des Einleseprozesses abfragen oder warten bis das nächste Bild übertragen wurde.

- Die optionale Einblendung des Bildes in den Framebuffer wird über die ioctl-Kommandos

  VIDIOC_G_FBUF, VIDIOC_S_FBUF,
  VIDIOC_G_WIN, VIDIOC_S_WIN und
  VIDIOC_PREVIEW implementiert.

  Die Eigenschafen Framebuffer muss mit dem ioctl-Kommando VIDIOC_S_FBUF gesetzt werden. VIDIOC_S_WIN beschreibt die Grösse und Lage des Videobildes im Framebuffer. VIDIOC_PREVIEW startet und stoppt die Einblendung des Videobildes. Für die Einblendung wird das einfachste Verfahren des direkten Überschreibens des Framebufferbereiches gewählt. Overlaying oder Clipping müssten im FPGA zusätzlich integriert werden.

- Die Einstellung von Bildgrösse und Farbformat für die Funktionen *read* und *mmap* erfolgt über die ioctl-Kommandos

  VIDIOC_G_FMT und
  VIDIOC_S_FMT.

  Die Previewfunktion besitzt eigene Funktionen für die Einstellung der Bildgrösse und das Farbformat. Diese sollten auch getrennt gespeichert werden, um ein einfaches Umschalten zwischen *read* / *mmap* und Preview zu ermöglichen. Damit bietet sich dann eine schnelle Methode verschieden Grosse Bilder parallel zu bearbeiten.

- Die Einstellungen der Bildeigenschaften wie Helligkeit, Kontrast, Sättigung und Farbton sind in der Video4Linux-API als sogenannte Controls bzw. Bedienelemente. Diese Bedienelemente können von unterschiedlichen Typen wie Integer, Boolean, Menu oder Button sein. Der Wertebereich und die Schrittweite der Bedienelemente kann ebenfalls den Hardwareanforderungen angepasst werden. Das ioctl-Kommando

  VIDIOC_QUERYCTRL

  gibt die verfügbaren Bedienelemente zurück. Es sind bereits eine Anzahl von Standardbedienelementen vordefiniert. Bedienelemente können aber auch spezielle Hardwareeigenschaften abgebilden. Die ioctl-Kommandos

  VIDIOC_G_CTRL und
  VIDIOC_S_CTRL

  ermöglichen das Auslesen und Setzten der Hardwareeigenschaften.

- Die Videoeingabe kann zwischen vier Videoeingängen wählen. Das ioctl-Kommando

  VIDIOC_ENUMINPUT

  listet die Eingänge und deren Leistungsmerkmale auf.

  VIDIOC_G_INPUT und
  VIDIOC_S_INPUT

  erlauben das Lesen und Setzen des aktuellen Eingangs.

Die Schnittstelle bietet keine Powermanagementfunktion. Der Einleseprozess durch den FPGA muss jedoch nur erfolgen, wenn mindesten eine Applikation den Gerätetreiber geöffnet hat. Benutzt keine Applikation den Gerätetreiber kann der Decoderchip und die Funktionalität im FPGA abgeschalten werden.

5.4.3 I^2C -Treiber

Die Kommunikation mit den Encoder- und Decoderchips wird über den I^2C-Bus abgewickelt. In Abschnitt 2.3.4 wurde das Protokoll vorgestellt. Das Subsystem, beschrieben in Abschnitt 3.6, teilt die Treiber in I^2C Busadapter und Chiptreiber. Das Businterface des LartVIO wird den Algo-Bit-Treiber

nutzen. Dieser Layer nutzt vier Einsprungsroutinen um die Bitzustände der Clockleitung SCL und der Datenleitung SDA lesen und schreiben zu können.

Jeder Chip besitzt eine Anzahl von Adressen über die er auf dem I²C Bus adressiert werden kann. Wird der Chiptreiber geladen, registriert er sich beim i2c_core Treiber. Den registrierten Busadaptern wird der Chiptreiber der Reihe nach übergeben. Auf dem entsprechenden Bus wird dann versucht den Chip anzusprechen. Der Chiptreiber muss daher die für den Chip relevanten Adressen in einer Struktur an den Bustreiber übergeben. Gleichzeitig müssen die Chips zum Zeitpunkt der Registrierung auf I²C Busanfragen reagieren. Der Busadapter sollte daher den FPGA initialisieren und die Chips einschalten. Die Chiptreiber hingegen sollten so allgemein wie möglich gehalten werden. Unter Linux ist es üblich für jeden Chip nur einen Treiber zu verwenden. Der Chiptreiber kann so auch in anderen Architekturen eingesetzt werden.

Beide Treiber sollten mit den in Europa üblichen Videostandards CCIR und PAL initialisiert werden. Der Chiptreiber wird ADV7170 benannt. Der auf dem LartVIO verwendete ADV7171 ist eine abgespeckte Variante des ADV7170 und bis auf einige nicht vorhandene Funktionen vollständig kompatibel zu diesem. Da der Framebuffertreiber keine Unterstützung für die Einstellung der Bildeigenschaften Helligkeit, Kontrast usw. bietet, ist die Implementierung dieser Funktionen nicht notwendig.

Im Kernelbaum existiert bereits ein Treiber für den SAA7111, der eine ähnliche Funktionalität bietet wie der SAA7113. Der Treiber wurde jedoch für eine ältere Version des I²C Bustreiber geschrieben, bevor das I²C Subsystem in den Kernel implementiert wurde. Der Aufwand für die Anpassungen des Treiber auf das neue Subsystem schienen gegenüber einer Neuimplementierung ungleich grösser. Die Test für den angepassten Treiber hätten auf die bereits vorhandenen Treiber ausgedehnt werden müssen, die diesen Treiber nutzen.

Es wurde daher ein neuer Treiber entwickelt der neben der Initialisierung des SAA7113 Eintstellungen für Helligkeit, Kontrast, Sättigung und Farbton zulässt. Der Videoeingang wird ebenfalls über den I²C Bus ausgewählt und muss im Treiber berücksichtigt werden.

5.5 Testverfahren

Ein grosser Teil der Funktionalität wurde in den FPGA verschoben. Als Testverfahren sind für die Module Überprüfungen mit Wellenform-Simulationen vorgesehen.

Die Treiber müssen ensprechend den Schnittstellen der Subsysteme funk-

tionieren. Daher werden die Schnittstellen zum Subsystem zu prüfen sein. Insbesondere beim Framebuffersubsystem ist die Schnittstelle des Subsystems nicht vollständig spezifiziert. Zoon berscheibt in [Zoo01] einen Teil der Console und Framebufferschnittstellen. Da die Console-Treiber mit dem Framebuffersubsystem verbunden sind, müssen auch einige Tests mit den Console-Schnittstellen durchgeführt werden. Die Video4Linux2-API ist in 3.5.2 spezifiziert.

Neben eigenen Testprogrammen stehen vorhandene Anwendungen für die Videoeingabe und die Videoausgabe zur Verfügung. Im Bereich der grafischen Oberflächen sind der X-Server XFree86 und die QT-Bibliothek von Trolltech zwei bekannte Vertreter, die das Framebufferdevice nutzen. Weiterhin sind verschiedene kleinere Anwendungen verfügbar, die das Framebuffergerät ansprechen. So ist das Kommando *fbset* bereits auf in der Lart Distribution enthalten. fbset verändert die Einstellungen des Framebuffergerätes.

Anwendungen, die den Video4Linux-Treiber nutzen, sind reichhaltig vorhanden. Viele Programme werden zur Zeit auf die neue Video4Linux2-API angepasst. XawTv ist ein Programm, das bereits die Video4Linux2-API nutzt. Es setzt jedoch einen X-Server voraus. Mit einer Auswahl dieser Programme sollen die Funktionen und Schnittstellen der Treiber getestet werden.

Kapitel 6

Implentierung

Nachdem die Schwerpunkte der Entwicklung im letzten Kapitel beschrieben und die Schnittstellen zwischen Hard- und Software definiert wurden, beschäftigt sich dieses Kapitel mit der Implementierung der Treiber und des Hardwaredesigns. Es werden einige vom Autor als interessant empfundene Teile der Implementierung näher beschrieben. Ein sich anschließender Abschnitt wird auf aufgetretene Probleme und deren Lösungen eingehen. Die Ergebnisse der Testumgebung stellen den Abschluss dieses Kapitels dar.

6.1 FGPA

Für die Implementierung des FPGA-Designs nutzte Tigris Elektronik als Programmiersprache Altera HDL in Verbindung mit der Integrierten Entwicklungsoberfläche Quartus II Webedition. Herr Guido Kuhlmann entwarf ein Gundgerüst, das an den eigenen Entwurf angepasst wurde.

6.1.1 Das DMA-Modul

Eine Teilaufgabe des DMA Moduls liegt in der Übertragung der Daten zwischen Hauptspeicher und Videomodulen. Dazu muss das DMA-Modul den Prozessor vom Speicherbus trennen und die Steuersignale für den Speicher selbst generieren. Für die Abkopplung des Prozessors vom peripheren Bus stellt der SA1100 zwei Steuerleitungen zur Verfügung, die als alternative Funktionen der Pins GPIO21 und GPIO22 auch am High-Speed-Connector, der Verbindung zwischen LART Board und LartVio, zur Verfügung stehen. Mit MBREQ wird der periphere Bus angefordert. Der StrongARM beendet seine Buszugriffe und signalisiert über MBGNT die Freigabe des Busses. Der Prozessor arbeitet mit seinem internen Cache weiter bis zu einem notwen-

digen Zugriff auf den Hauptspeicher. Danach gelangt der Prozessor in einen Wartezustand.

Die Erzeugung der notwendigen Refreshzyklen für den DRAM-Speicher übernimmt weiterhin der Prozessor. Die Notwendigkeit eines Refresh stellt er über einen internen Counter MDCONFIG (siehe [Int99, 10.1.2 Memory Configuration Register]) fest, dessen Überlauf in einem Refreshflag gespeichert wird. Der Refresh wird dann beim nächstmöglichen Zugriff ausgeführt. Erhält der Prozessor den Bus mit dem Lowsignal von MBREQ zurück, wird der DRAM mit dem nächsten Buszugriff aufgefrischt. Der FPGA kann den Bus daher maximal die Zeit zwischen zwei Refreshzyklen beanspruchen. Danach muss der Prozessor mindestens für kurze Zeit den Buszugriff zurückbekommen, um notwendige Refreshzyklen zu generieren.

Das Busmasterdesign teilt sich in die Verwaltung der Busanforderungen und die Erzeugung der Steuersignale für den Speicher. Videoeingabe und Videoausgabe besitzen je einen Zwischenspeicher, die ab einem vom Zwischenspeicher bestimmten Punkt bedient werden müssen.

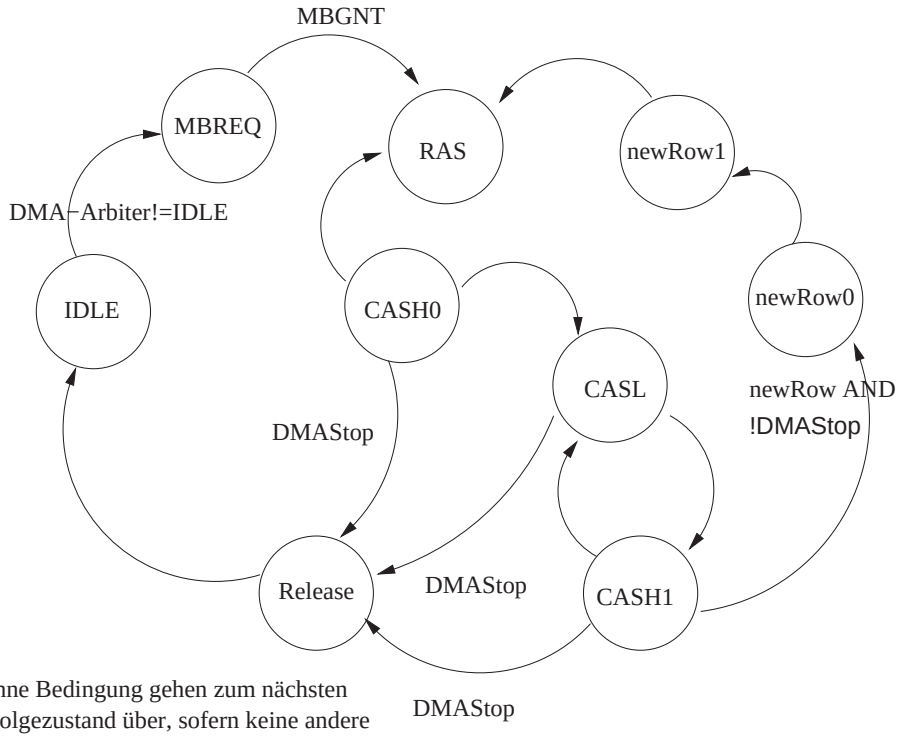

Abbildung 6.1: Zustandsmaschine fuer den Speicherzugriff

Die Signale für den Hauptspeicher wurden mit einer endlichen Zustands-

maschine erzeugt. Die Zustandsmaschine arbeitet synchron zum Takt. Dabei wurde die Taktlänge mit 18ns (Lartbustakt/2) angenommen. Wie bereits im Abschnitt 2.2.3 beschrieben, muss beim Fastpagemode die Zeilenadresse nur einmal für die gesamte Zeile der Speichermatrix geschrieben werden. Der Speicherzugriff reduziert sich damit auf

$$t_{Speicherzugriff} = n * t_{HPC} + \frac{t_{RCD}}{n}$$

wobei n die Gesamtzahl der Speicherzugriffe in derselben Zeile sind.

Der Fastpagemode des DRAM wurde in dieser Zustandsmaschine berücksichtigt. Abbildung 6.1 zeigt die möglichen Zustände und die Übergangsbedingungen. Die Zustände newRow0 und newRow1 sichern die Erholzeit zwischen zwei RAS-Zyklen. Im Zustand CASL werden die Daten geschrieben. Bei EDO-RAMs können die Daten bis zur nächsten fallenden Flanke von nCAS gelesen werden. Die Daten werden daher erst im Zustand CASH1 übernommen. Die Bedingungen für einen Zustandswechsel sind am Übergang notiert. Bei Übergängen die keine Bedingung haben, wird mit dem nächsten Takt in den neuen Zustand gewechselt, sofern keine andere Übergangsbedingung zutrifft. Bedingt durch die Anordnung der Adressleitungen (Abbildung 2.4) ist ein Burstzyklus auf 256 Spalten begrenzt.

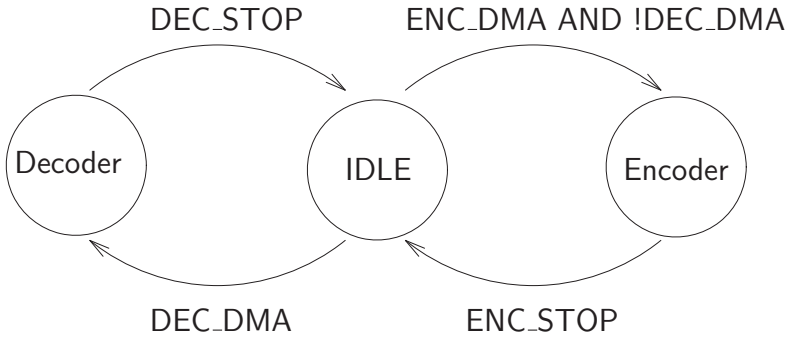

Abbildung 6.2: Zustandsmaschine fuer den Zugriff auf den DMA

Eine weitere Zustandsmaschine verwaltet DMA-Anforderungen von Videoein- und Videoausgabe. Die Signale ENC_STOP und ENC_DMA, sowie DEC_STOP und DEC_DMA schliessen sich aus. Treten die Signale ENC_DMA und DEC_DMA zeitgleich auf, wird das Videoausgabemodul zurückgestellt. Abbildung 6.2 verdeutlicht Arbeitsweise der Zustandsmaschine für die DMA-Anfragen.

Die erste Version der Implentierung zeigte häufig fehlerhaft eingelesene und geschriebene Werte. Messungen an den Adress- und Steuerleitungen

zeigten kurze Zwischenimpulse wie in Abbildung 6.3 zu sehen. Diese Impulse treten durch unterschiedliche Verzögerungszeiten einzelner Logikeinheiten auf. Es ergaben sich kurzzeitig unerwünschte Zustände, die auf die Ausgangsleitungen geschalten wurden. Die Adress-, Daten- und Steuerleiten die mit dem High-Speed-Bus verbunden sind, wurden über D-Flipflops gepuffert. Die Signale `dec_read` und `enc_write` mussten entsprechend um einen Takt verzögert werden.

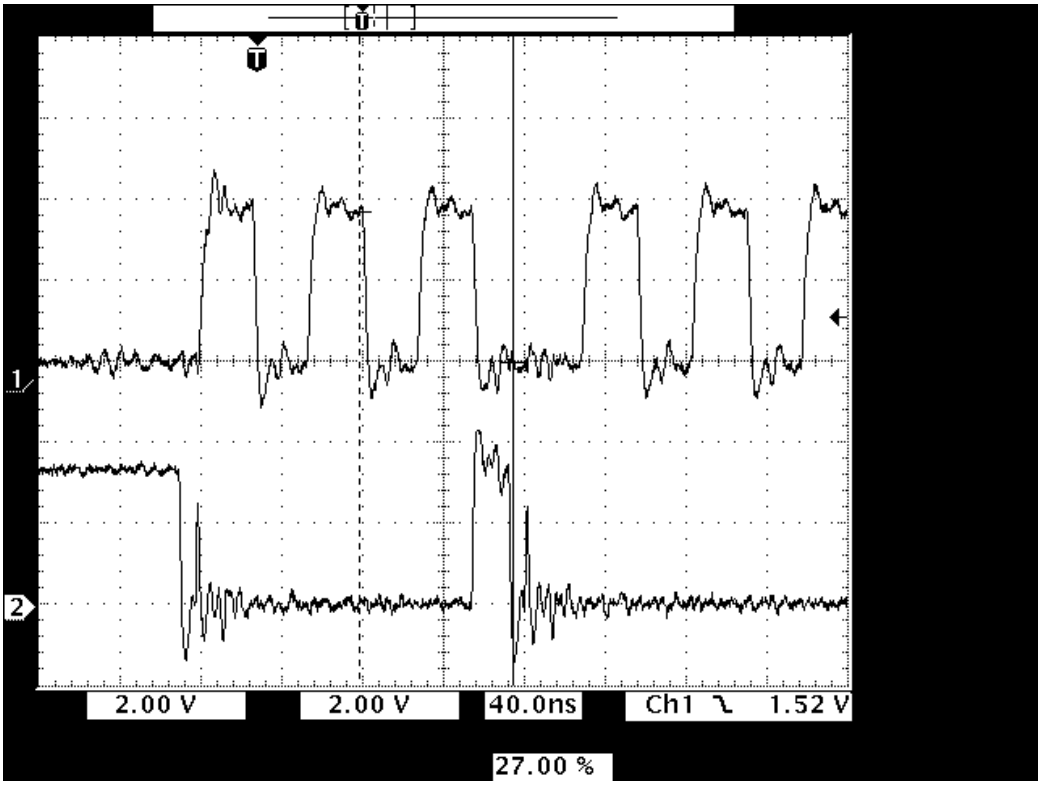

Abbildung 6.3: Zwischenzustände auf den Adress- und Steuerleitungen

6.1.2 Wandlung der Farbbereiche

Beispielhaft für die Videobearbeitungsmodule wird die Konvertierung von RGB nach YCbCr beschrieben. Jack beschreibt die Konvertierung in [Jac96, YCbCr Color Space] mit den Formeln

$$Y = 0,257R + 0,504G + 0,098B + 16$$
$$Cb = -0,148R - 0,291G + 0,439B + 128$$
$$Cr = 0,439R - 0,368G - 0,071B + 128$$

in denen bereits eine Angleichung an die Wertebereiche des RGB24 Formats enthalten ist. Für die Konvertierung von RGB nach YCbCr sind zwei Pixel RGB = 48 Bit erforderlich, die in einen 16Bit YCbCr Werte gewandelt werden. Das DMA-Modul übergibt jeweils 32Bit-Werte, in denen die RGB Werte unterschiedlich angeordnet sind. Teilt man die 32Bit-Werte in je 8Bit-Werte ergeben sich die in Tabelle 6.1 aufgeführte Konstellationen. Die Vorkommastellen bezeichnen die 32Bit-Werte in der Reihenfolge des Einlesens, also 0 für den ersten 32Bit-Wert und 2 für den letzten 32Bit-Wert. Die Nachkommastellen geben den 8Bit-Wert innerhalb der 32Bit-Werte an, wobei 0 das niederwertigste Byte beschreibt.

RGB-Wert	R	G	B
1.	0.0	0.1	0.2
2.	0.3	1.0	1.1
3.	1.2	1.3	2.0
4.	2.1	2.2	2.3

Tabelle 6.1: Zuordnung der Farbkanäle RGB zu den eingelesenen 32Bit Werten

Mit drei 32Bit Werten werden vier RGB Werte übernommen, die sich in zwei YCbCr Werte übertragen lassen. Die RGB Werte 2. und 3. sind über zwei 32Bit Werte verteilt. Der aktuelle Wert wird bei der Übergabe zwischengespeichert. Über eine Zustandsmaschine erfolgt die Zuordnung der RGB Wert zu den 32Bit Werten entsprechend Tabelle 6.1. Mit der Übernahme des dritten 32Bit Wertes stehen zwei RGB Werte zur Verarbeitung an, der dritte und vierte RGB Wert. Die Berechnung der YCbCr-Werte beginnt mit der Übernahme des 32Bit Wertes. Der vierte RGB Wert wird einen Takt nach der Übernahme des dritten 32Bit Wertes in die Berechnungspipeline übergeben.

Wie aus den obigen Gleichungen ersichtlich, sind für die Konvertierung eines RGB-Wertes neun Multiplikationen und neun Additionen erforderlich. Für die Berechnung wurden ausschließlich Festkommaeinheiten verwendet, um die Anzahl der verwendeten Logikeinheiten des FPGA so gering wie möglich zu halten.

Als Ergebnis der Konvertierung werden Y1, Cb1, Cr1 und Y2 für einen vollständigen YCbCr Wert benötigt. Das Ergebnis wird in zwei Stufen gebildet:

Stufe 1: RGB1 => Y1,Cb,Cr

Stufe 2: RGB2 => Y2

Die Multipliziereinheiten enthalten eine zweistufige Pipeline. Der Timinganalysator von Quartus II ermittelte eine maximale Taktfrequenz von ca. 60 MHz für den EP1k100QC208-2. Soll der FPGA mit vollem LART-Bustakt arbeiten, müssen weitere Pipelinestufen eingefügt und die Signale YResultready und q_rd angepasst werden. Die nachfolgenden Additionen wurden ohne Latenz implementiert. Bei höherer Taktfrequenz kann sich die Notwendigkeit ergeben hier ebenfalls eine Pipelinestufe einzufügen. Die Ergebnisse des ersten RBG Wertes werden zwischengespeichert. Mit dem Ergebnis des Y Wertes für den zweiten Pixel werden die Werte an die nachfolgende Einheit übergeben.Bei eingeschalteter Skalierung wird jeder RGB-Wert als YCbCr-Doppelpixel ausgegeben.

Im Videoeingabemodul wurde ein zweiter Farbkonverter implementiert, der das YCbCr-Signal weiterleitet. Er soll als einfaches und vom Code übersichtliches Beispiel für weitere Implementierungen von Videobearbeitungsmodulen dienen. Die Skalierung des Konverters verwirft jeden zweiten Doppelpixel. Das Listing ohne Kommentarkopf ist in Anhang B.1.1 zu finden.

6.1.3 Austastrahmen für die Videoausgabe

Ein Bild nach CCIR601 enthält 704 aktive Bildpunkte in der Horizontalen. 720 Bildpunkte werden an den Encoderchip übertragen. Eine erste Implementierung zeigte bei verschiedenen Handelsüblichen TV-Geräten einen wesentlich breiteren Rahmen als acht Bildpunkte ($\frac{720-704}{2}$). Der sichtbare Bereich lag horizontal bei ca. 640 Bildpunkten. 80 Bildpunkte einer Zeile wurden aus dem Hauptspeicher übertragen und ohne effektiven Nutzen an den Encoder weitergeleitet. Zwischen FIFO und Encoderbus wurde eine Randerzeugung implementiert. Die Randerzeugung entspricht einem Umschalter der in selbstdefinierten aktiven Bereichen die Daten vom FIFO an den Encoderbus überträgt und in den inaktiven Bereichen schwarze YCbCr-Werte erzeugt. Als angenehmer Nebeneffekt kann die Größe des Videobildes nun zwischen 140x80 Bildpunkten und 720x576 Bildpunkten verändert werden, wobei die Anzahl der Bildpunkte ein Vielfaches von 4 sein muss. Die Einschränkung nur Vielfache von 4 zuzulassen, ergibt sich aus der Datenwortgröße und der Umwandlung der Werte von RGB nach YCbCr.

6.1.4 Compilerprobleme

Im Laufe der Implementierung der FPGA-Funktionen traten mit zunemehmender Komplexität der Sourcen immer häufiger nicht erklärbare Fehler auf. Zunächst wurden diese Fehler fälschlicherweise auf Programmierfehler zurückgeführt. Das Fehlermuster war schließlich sehr unterschiedlich. Verschiedene Kompilierungen mit minimalen Änderungen brachten unterschiedliche Ergebnisse. Diese reichten von einem funktionierenden Resultat und einem fehlerhaften Resultat bis zu zwei fehlerhaften Resultaten mit Fehlern in vollständig unabhängigen Modulen.

Ein Fehler, das Blockieren des Farbraumkonverters YCbCr_RGB auf der Videoeingabeseite, wurde näher untersucht. Das Modul enthält eine Zustandsmaschine, die die Zuordnung der RGB-Werte zu den 32-Bit Datenworten übernimmt. Diese Zustandsmaschine geriet nach einiger Zeit in einem nicht definierten Zustand. Da für diesen nicht definierten Zustand keine Überleitung in einen anderen Zustand existierte, blockierte die Zustandsmaschine in dem Zustand. Der Compiler bildet Zustandsmaschinen als Registerwerte ab. Die beschriebene Zustandsmaschine hatte vier Zustände, die der Compiler in einem 4Bit-Wert abbildete. Das Problem des Blockierens wurde durch die Definition eines Zustandes als allgemeiner Zustand gelöst. Dafür wurde der Anfangszustand verwendet. Die Zustandsmaschine enthält dann $n - 1$ definierte Zustände und einen sonstigen Zustand, der über alle restlichen Zustände, einschließlich nicht definierter Zustände gilt. Dem Autor ist bewusst, das diese Problembehandlung keine vollständige Lösung ist. Warum die Zustandsmaschine in einen nicht definierten Zustand gerät, konnte nicht vollständig geklärt werden. Der Autor vermutet Einflüsse von anderen naheliegenden Verbindungsleitungen, die die Logikeinheiten des FPGA verbinden.

In weiteren Tests wurden ähnliche Fehler in anderen Zustandsmaschinen entdeckt. Die Probleme wurden, wenn auch nicht vollständig, nach dem gleichen Muster wie bei der oben beschriebenen Zustandsmaschine umgangen. Eine Besonderheit stellt die Zustandsmaschine zur Speicheransteuerung dar. Diese muss in den IDLE-Zustand zurückkehren, da der IDLE-Zustand das Ende einer Übertragung kennzeichnet. Gerät die Zustandsmaschine in einen nicht definierten Zustand, geht sie mit dem nächsten Takt in den IDLE-Zustand über.

6.2 Gerätetreiber

Applikationen greifen auf die Hardware über die Gerätetreiber Framebuffer und Video4Linux2 zu. Beide Gerätetreiber kommunizieren mit den Encoder-

bzw. Decoderchips über I^2C-Treiber. Die Implementierung von Framebuffertreiber und Videl4Linux2-Treiber erfolgte in zwei Schritten. Im ersten Schritt wurden ein Grundgerüst der Treiber und die Schnittstellen implementiert. Im zweiten Schritt wurde Hardware angebunden. Diese Zweiteilung wurde aufgrund von Verzögerungen in der Hardwareherstellung vorgenommen. Die Schnittstellen konnten so bereits vor der vollständigen Implementierung teilweise getestet und die Funktion mit Standardapplikationen nachgewiesen werden.

6.2.1 Framebuffer

Die in diesem Abschnitt besprochenen Funktionen und Strukturen sind der Datei lartviofb.c entnommen. Listings besprochener Funktionen können in Anhang B.2 nachgeschlagen werden.

Der Framebuffertreiber basiert auf der BeispielImplementierung skeleton.c der Kernelversion 2.5.20 aus dem Verzeichnis drivers/video. In der BeispielImplementierung sind die notwendigen Funktionen für die Anbindung an das Framebuffersubsystem dokumentiert. In dieser Version ist die Struktur display, auf die in Abschnitt 3.4 eingegangen wurde, noch notwendig. Im Laufe der Entwicklung soll diese Struktur bis zur Fertigstellung der Kernelversion 2.4.6 entfernt werden. Die Funktionen in der Datei skeleton.c entsprechen in weiten Teilen den Funktionen in fbcon.c der aktuellen Kernelversion 2.4.18. Lediglich die Übergabeparameter sind unterschiedlich. Um redundanten Code zu vermeiden, wurde für beide Kernelversionen nur ein Treiber geschrieben. Unterschiede wurden über Preprozessorbefehle eingebettet. Das folgende Codebeispiel zeigt die Anpassung der Übergabeparameter anhand der Funktion lartviofb_set_par, die weiter unten erläutert wird. Das Makro KERNEL_VERSION setzt die übergebene Version in eine Integerzahl um.

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
static int lartviofb_set_par(struct fb_info *info)
{
    struct lartvio_par *par = (struct lartvio_par *) info->par;
    struct fb_var_screeninfo *var = &info->var;
#else
static void lartviofb_set_par(const void *vpar, struct fb_info_gen *gen)
{
    struct lartvio_par *par = (struct lartvio_par *) vpar;
    struct fb_var_screeninfo *var = &par->mvar;
#endif
```

In der Initialisierungphase wird die Funktion

```
int lartviofb_init_fpga(struct lartvio_par *par)
```

aufgerufen. In der Funktion `lartviofb_init_fpga` wird zunächst die Speicherbank2 initialisiert. In dieser Speicherbank liegen die Register des LartVIO. Der Bustakt des LART-Board wird auf das LartVIO geschalten. Ab jetzt können die Register des LartVIO gelesen und geschrieben werden. Das Vorhandensein der Hardware wird über das Register IDSTR geprüft. Die Register des LartVIO sind im „LART-Video Hardware & Programmable Logic Manual“[KS02] erläutert. Für die Kommunikation mit dem Encoderchip ADV7171 ist der I²C-Treiber *adv7170* zuständig. Bevor der Treiber initialisiert werden kann, muss der I²C-Adaptertreiber i2c_lartvio geladen sein. Der I²C-Adaptertreiber gewährleistet den Zugriff auf den I²C-Bus des LartVIO. Beide Module werden über die Funktion `request_module` geladen. Der Kernel läd dann automatisch die Module nach, sofern sie nicht bereits geladen sind.

Anschließend wird ein linearer, nicht gecachter Speicherbereich angelegt, der später die Bilddaten enthält. Der Speicherbereich wird entsprechend der maximalen Bildgröße angelegt. Das sind derzeit $720 * 576 * 3Byte = 1,3MByte$[1]. Dieser Speicherbereich ändert sich bei Veränderungen der Bildgröße nicht. Arbeiten auf dem LART-Board verschiedene Applikationen, kann der Speicher nach einer gewissen Zeit stark fragmentiert sein. Das Erneute Anlegen eines linearen Speicherbereiches ist dann nicht mehr gewährleistet.

Zwei Funktionen sind für Veränderungen der Bildeigenschaften notwendig. In der Funktion

```
int lartviofb_check_var(struct fb_var_screeninfo *var,
                        struct fb_info *info)
```

werden die neuen Bildeigenschaften geprüft. Die Eigenschaften werden in der Struktur fb_var_screeninfo übergeben. Die Struktur fb_info ist eine interne Struktur des Framebuffertreiber, die zum Zeitpunkt der Registrierung an das Framebuffersubsystem übergeben wurde. Unterstützt die Hardware die angeforderten Eigenschaften nicht, wird der negative Fehlerwert EINVAL (Invalid Value) zurückgegeben. `lartviofb_check_var` unterscheidet sich gegenüber der Funktion `lartviofb_decode_var` aus der aktuellen Kernelversion in der Funktionsweise. `lartviofb_check_var` prüft die Struktur fb_var_screeninfo und kann diese bei Bedarf verändern. Applikationen können sich nicht darauf verlassen, dass alle Eigenschaften übernommen wurden. Die Funktion

[1] 3Byte = 24Bit / 8

`lartviofb_decode_var` darf fb_var_screeninfo nicht verändern. Die Eigenschaften für die Hardware werden über die interne Struktur lartvio_par übergeben.

```
int lartviofb_decode_var(const struct fb_var_screeninfo *cvar,
                                                    void *vpar,
                                        struct fb_info_gen *gen)
```

Gibt die jeweilige Funktion den Wert Null zurück, wird durch das Framebuffersubsystem die Funktion `lartviofb_set_par` aufgerufen. Aufgabe der Funktion ist die Veränderung der Hardwareeinstellungen. In der Kernelversion 2.5 wird hier die geprüfte Struktur fb_var_screeninfo in der Struktur fb_info übergeben. Die Funktion im aktuellen Kernel erhält als Übergabeparameter die in `lartviofb_decode_var` erzeugte interne Hardwarestruktur. Die Differenzen in der Funktionsweise wurden über Hardwarestruktur lartvio_par gelöst. Sie enthält die Struktur fb_var_screeninfo, die in der Funktion `lartviofb_decode_var` gefüllt wird. In der Anpassung der Übergabeparameter der Funktion `lartviofb_set_par` wird über einen Zeiger auf die entsprechende Struktur fb_var_screeninfo verwiesen.

Die Funktionen `lartviofb_encode_fix` und `lartviofb_encode_var` geben aktuellen Einstellungen der jeweiligen Struktur zurück. In der Kernelversion 2.5.x wurde dafür generische Funktionen geschaffen.

Die Framebufferhardware kann über die Funktion

```
static int lartviofb_blank(int blank_mode, struct fb_info *info)
```

in den Ruhezustand versetzt und wieder angeschalten werden. Das Framebuffersubsystem sieht dafür vier verschiedene Zustände vor, wovon die Hardware nur zwei unterstützt. Alle Schlafmodi schalten die Videoausgabehardware, einschließlich der FPGA-Logik für die Videoausgabe vollständig ab. Beim Wiedereinschalten wird die Hardware neu initialisiert.

Die weiteren Funktionen werden durch den Consoletreiber aufgerufen und sind für die Funktion des Framebuffertreiber nicht relevant.

6.2.2 Video4Linux2

Alle Funktionen und Strukturen in diesem Abschnitt sind der Datei lartvio_vin.c entnommen. Listings besprochener Funktionen können in Anhang B.3 nachgeschlagen werden.

Die Initialisierung des Video4Linux2-Treibers entspricht in weiten Teilen der des Framebuffertreibers. Nach der Einrichtung der Speicherbank wird das

Vorhandensein der Hardware geprüft. Ein linearer, nicht gecachter Speicherbereich der Größe eines RGB24-Bildes mit der Auflösung 720x576 Bildpunkte wird ebenfalls eingerichtet. Der Videodecoderchip wird initialisiert, um die Funktion des I^2C-Bus zu prüfen. Danach wird die Hardware deaktiviert. Wenn eine Applikation auf den Treiber zugreift, wird die Hardware erneut initialisiert. Solange bleibt die Hardware abgeschalten. Beendet die letzte Applikation die Verbindung zum Treiber, wird die Hardware wieder schlafen gelegt. Diese Maßnahmen machen ein zusätzliches Powermanagment überflüssig.

Während das Framebuffersubsystem den Zugriff auf den Bildspeicher übernimmt, müssen die Funktionen für den Zugriff im Video4Linux-Treiber implementiert werden. Die Funktion *mmap* für das Einblenden des Bildspeichers in den Applikationsbereich wird im Zusammenhang mit den Streamingfunktionen benötigt. `mmap_request_buffers` teilt den Bildspeicher in mehrere Puffer.

```
int mmap_request_buffers(struct capture_device *dev,
     struct v4l2_requestbuffers *req)
```

In der Implementierung werden maximal vier Puffer angelegt. Die Hardwareeinstellungen, repräsentiert durch die LartVIO-Register, werden beim Anlegen berechnet und in den Pufferstrukturen abgespeichert. Durch dieses Verfahren wird die Interruptroutine entlastet, die sonst für jedes Bild erneut die Hardwareeigenschaften berechnen müsste. Die Puffer werden durch die Applikation in eine Warteschlange zum Füllen gestellt

```
int capture_queuebuffer(struct capture_device *dev,
    struct v4l2_buffer   *vidbuf)
```

und durch die Interruptroutine nach der Übertragung des Bildes in eine weitere Warteschlange eingereiht, die die fertigen Bilder repräsentiert. In der Interruptroutine wird die Struktur des Puffers mit einem Zeitstempel und eine laufenden Bildnummer versehen. Sind keine weiteren Puffer vorhanden schaltet die Interruptroutine die Übertragung ab.

Die Methode *read* soll die einfache Methode für den Zugriff auf Bilder darstellen. Sie nutzt die eben beschrieben Streamingalgorithmen für den Bildzugriff. Es werden beim ersten Zugriff zwei Puffer angelegt und in die Warteschlange zum Füllen gestellt. Dabei wird davon ausgegangen, dass eine Applikation wahrscheinlich mehr als ein Bild abholen wird. *read* prüft daher im ersten Schritt, ob bereits Bilder bereitliegen. Die Warteschlange wird dabei entleert und nach einem Bild gesucht, dass nicht älter als 20ms

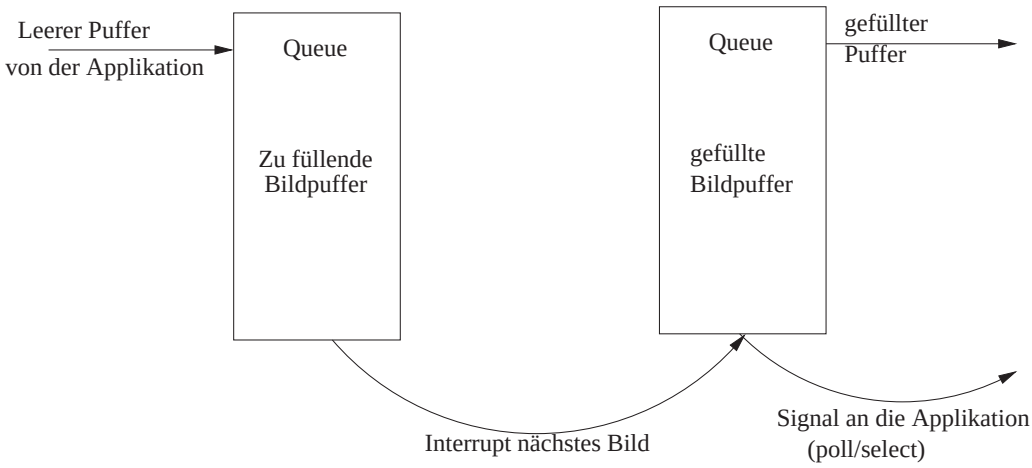

Abbildung 6.4: Vorgang des Bildeinlesens

[2] ist. Wird ein Bild gefunden übergibt *read* ohne Verzögerung das Bild an die Applikationen. Da *read* die Bilder in einen Applikationspuffer kopiert, kann der Bildpuffer danach wieder in die Warteschlange zum Füllen gestellt werden. Der Bildspeicher kann nur ein Bild der Größe 720x576 fassen. Daher ist die eben beschriebene Funktion nur für Bilder mit dem Format 360x288 Bildpunkte gültig. Wird der Bildspeicher auf die Größe zweier Bilder im Format 720x576x24Bit erweitert, ist die Funktion auch für diese Bildgrößen gewährleistet. Es darf daher nicht verwundern, dass nur 12,5Bilder/s in der Auflösung 720x576 eingelesen werden. Die Interruptroutine stellt nach jedem Bildeinzug fest, dass keine weiteren Bildpuffer zur Verfügung stehen und unterbricht die Übertragung. Der Vorgang kann frühestens mit zum übernächsten Bild wieder gestartet werden.

Bei Veränderungen der Bildeigenschaften werden die Bildpuffer über die Prozedur

```
void mmap_unrequest_buffers(struct capture_device *dev)
```

gelöscht, da Änderungen an den Bildeigenschaften auch die Bildpuffergröße verändern können. Die Overlayfunktion, die das Videobild in den Framebuffer schreibt, speichert die Bildeigenschaften in eigenen Variablen. Der Aufruf der Previewfunktion beeinflußt die Bildpuffer nicht. Diese Impelementation ermöglicht Applikationen in schneller Folge Bilder mit unterschiedlichen Eigenschaften einzulesen. Grundsätzlich wäre es auch möglich Puffer

[2] 20ms entsprechen der Übertragungszeit eines Halbbildes

mit verschiedenen Bildeigenschaften zu erzeugen und füllen zu lassen. Die Video4Linux2-API unterstüzt aber keine verschiedenen Bildpuffer.

In Abschnitt 5.3.4 wurde auf die Implementierung eigener Farbkonverter und anderer Videobearbeitungsmodule im FPGA eingegangen. Eigenimplementierungen muss der Video4Linux2-Treiber natürlich ebenfalls unterstützten. Für reine Farbkonverter wurde eine Struktur

```
static struct v4l2_fmtdesc capfmt[] =
{
     /*     colfmt, colorstring, */
     /*     Pixel format  (see videodev.h), Format flags ,  */
     /*     depth, {reserved} */
     {    0, {"RGB-24 (R-G-B)"},
          V4L2_PIX_FMT_RGB24,   0, 24, {0, 0},
     },
     {    1, {"YUV 4:2:2 (Y-U-Y-V)"},
          V4L2_PIX_FMT_YUYV,    V4L2_FMT_CS_601YUV, 16, {0, 0},
     },
/* could be in one of the next releases of FPGA
     {    2, {"Greyscale-8"},
          V4L2_PIX_FMT_GREY,    0, 8, {0, 0},
     },
*/
};
```

angelegt. Die Struktur muss um das neue Farbformat erweitert werden. Ein Beispieleintrag für Graustufenbilder liegt als Kommentar vor. Über die Einträge der Struktur werden die Prüfungen und Berechnungen für die Hardwareeinstellungen vorgenommen. Die erste Variable der Struktur *colfmt* muss mit der Nummer des Multiplexereingangs im FPGA entsprechen. Die Farbkonverter müssen ebenfalls die Skalierung auf halbgroße Bilder unterstützen. Eine Anpassung des Treibers an eigene Farbraumkonverter sollte keine Probleme bereiten.

6.2.3 I^2C Treiber

Für die Kommunikation mit den Encoder- und Decoderchips mussten drei Treiber implementiert werden. Der Adaptertreiber stellt die Kommunikation mit dem I^2C-Bus bereit. Für die Abstraktion der Chipeigenschaften wurde für jeden Chip ein eigener Treiber implementiert.

Die beiden Leitungen des I^2C-Bus SCL und SDA können über das Register COMC1 gelesen und geschrieben werden. Dieses Verfahren unterstützt

der im Kernel vorhandene Bit-Algorithm-Layer. Im Adaptertreiber für das LartVIO mussten lediglich vier Funktionen implementiert werden, die die Zustände der Leitungen SCL und SDA lesen und schreiben können. In Initialisierung des Treibers wird der FPGA initialisiert und Encoder- sowie Decoderchip resetet. Danach bleiben beide Chips im eingeschalteten Zustand.

Die Chiptreiber sind unabhängig von den Besonderheiten des LartVIO. Sie können daher auch von anderen Treibern genutzt werden. Im Linuxkernel ist es üblich für jeden Chip nur einen Treiber zu implementieren, der von verschiedenen Hardwaretreibern genutzt wird. Die Kommunikation der Hardwaretreiber erfolgt über die Funktion

```
int command(struct i2c_client *client,
  unsigned int cmd, void *arg),
```

die der ioctl Funktion herkömmlicher Treiber entspricht. Die Kommandos des jeweiligen Treibers sind in den Headerdateien der Treiber definiert.

6.3 Schnittstellen zu Applikationen

Der Zugriff auf den Framebuffertreiber und die Video4Linux2-API wird für die einfachen Funktionen über die Standardzugriffe auf Treiber, also `open`, `read`, `write` und `close`, realisiert. Die meisten erweiterten Funktionen können durch die Applikation über ioctl-Kommandos aufgerufen werden. Insbesondere die Video4Linux2-API ist sehr umfangreich. Die applikationsseitige Nutzung der Treiber ist in [Sch02] ausführlich beschrieben. In diesem Abschnitt werden nur die einfachen Zugriffe auf ein Bild und die Übergabe an die Videoausgabe beschrieben.

Die Strukturen für den Framebuffertreiber sind in der Inlcudedatei fb.h defniniert. Applikationen müssen daher

```
#include <linux/fb.h>
```

einfügen. Die Strukturen und Kommandos der Video4Linux2-API sind in der Datei videodev.h definiert.

```
#include <linux/videodev.h>
```

Für den Zugriff auf die Treiber muss der Treiber geöffnet werden. Der Video4Linux2-Treiber ist über "/dev/video0"erreichbar.

```
int fd;
fd = open("/dev/fb0", O_RDWR);
```

Die Strukturen fb_var_screeninfo und fb_fix_screeninfo werden über ioct-Kommandos gelesen.

```
struct fb_fix_screeninfo finfo;
ioctl(fd, FBIOGET_FSCREENINFO, &finfo)
```

bzw.

```
struct fb_var_screeninfo vinfo;
ioctl(fd, FBIOGET_VSCREENINFO, &vinfo)
```

Die Struktur fb_var_screeninfo kann über das ioctl-Kommando `FBIOPUT_VSCREENINFO` mit veränderten Bildparametern an den Framebuffertreiber übergeben werden.

Der Grafikspeicher wird mit der Funktion `mmap` in den Applikationsspeicher eingeblendet. Bei Verwendung der Methoden `read` und `write` werden die Grafikdaten immer in den Grafikspeicher kopiert. Beim Zugriff auf den eingeblendeten Grafikspeicher entfällt der zusätzliche Kopiervorgang.

```
char *data;
data = (char *) mmap(0, finfo.smem_len, PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
```

Der folgende read-Befehl schreibt ein Bild vom Video4Linux2-Treiber in den Grafikspeicher des Framebuffertreibers. Das Bild wird dann auf der Grafikausgabe ausgegeben.

```
read(vid, data, vinfo.yres *  vinfo.xres * vinfo.bits_per_pixel / 8);
```

Wie bereits zu Beginn dieses Abschnitts erwähnt, ist die Schnittstelle zu den Treibern weit umfangreicher. Eine vollständige Auflistung aller Funktionen würde den Rahmen dieser Arbeit sprengen. Die Schnittstellen beider Treiber werden mit direktem Bezug auf das LartVIO in [Sch02] erläutert. Die Vollständige Video4Linux2-API wird in [Dir02] beschrieben. Da das Framebuffergerät direkt mit den Consoletreibern interagiert, kann auch über die Consoletreiber auf das Framebuffergerät zugegriffen werden.

6.4 Nachweis der Funktionalität

Entsprechend den Testanforderungen wurden die verschiedene Bereiche von Soft- und Hardware mit unterschiedlichen Mitteln getestet. Zu Testen waren:

- die Schnittstellen der Subsysteme,

- die Module des FGPA,
- sowie das funktionale Zusammenspiel von Hard- und Software.

Die Schnittstellen der Subsysteme wurden mit funktionalen Äquivalenzklassentests geprüft. Es entstanden dabei zwei Testprogramme *fbtest* und *v4l2test*, die im Quellcode auf der beiliegenden CD enthalten sind. Das Programm *fbtest* überprüft neben der korrekten Funktion der Schnittstellen gleichzeitig die Funktion der Hardware. Bei allen korrekten Hardwareeinstellungen wird eine Farbprüfung, die Funktion des Consoletreibers und die Anordnung der Bildpunkte im Speicher getestet. Aufgrund vielfältigen Einstellungsmöglichkeiten der Video4Linux2-Schnittstelle beschränkt sich das Programm v4l2test auf die Schnittstellenprüfung. Die Testprotokolle sind in gekürzter Form in Anhang C zu finden. Die Funktionstest des Video4Linux2-Treibers wurden in mehrere Programme ausgegliedert, die gleichzeitig der Softwaredokumentation [Sch02] als Beispielapplikationen dienen.

vtread liest über die read-Funktion der Video4Linux-Schnittstelle Bilder ein und gibt sie auf dem Framebuffergerät aus. *vtread* basiert auf dem Programm *vtcat*. *vtcat* ist ein Testprogramm, dass von der Video4Linux2 Homepage heruntergeladen werden kann. Es ist explizit zum Testen der Video4Linux2-Schnittstelle gedacht.

vtmmap prüft die Streamingfunktionen. Wie vtread kopiert vtmmap das eingelesene Bild in den Framebuffer.

vtpreview nutzt die direkte Übertragung des Bildes in den Framebuffer durch die Hardware.

vtctrl, ebenfalls ein Testprogramm von der Video4Linux Homepage, wurde um einige Funktionen erweitert. Es verändert Eingabekanal, Hardwareeinstellungen wie Helligkeit und die Bildauflösung. vctrl greift auf den Treiber im non-IO Modus zu. Der gleichzeitige Zugriff mehrerer Applikationen auf den Video4Linux2-Treiber konnte durch das Programm ebenfalls überprüft werden.

Die Schnittstellen wurden entsprechend der Dokumentationen geprüft. Programmierung der Treiber und der Tests lagen in der Hand einer Person. Eine falsche Interpretation der Schnittstellendokumentation konnte nicht ausgeschlossen werden. Standardapplikationen sollten bisher durchgeführte Tests untermauern. Für den Framebuffertreiber wurde die Bibliothek für graphische Oberflächen QT-Embedded der Firma Trolltech kompiliert und auf der Ramdisk des LART-Board installiert. Die Bibliothek enthält eine Reihe von

Beispielprogrammen, aus denen das Programm *hello* ausgewählt wurde. Das Programm öffnet ein Fenster und stellt einen sich bewegenden Text dar. Der XServer XFree86 war zu groß für die 8MByte Ramdisk des LART-Board. Für die Video4Linux2-Schnittstelle wurde kein Programm gefunden, das ohne XServer oder weitere Bibliotheken funktioniert.

Die Funktion der Module des FPGA wurde über Wellenformsimulationen überprüft. Die Eingangssignale des jeweiligen Moduls werden in der Wellenformsimulation definiert. Ein Simulator berechnet dann die Ausgangssignale. Der in Quartus II integrierte Simulator kann die logischen Funktionen und die zeitliche Verschiebung der Signale anhand des Kompilats berechnen und anzeigen. Anhang C.3 zeigt einen kleinen Teil der Wellenformsimulation des Farbkonverters RGB_YCbCr. Auf eine vollständige Auflistung der Wellenformsimulationen wurde aufgrund der Größe verzichtet. Die Funktion vieler Module musste aus Performancegründen in mehreren Schritten getestet werden. Das StrongARM-Interface wurde nicht simuliert. Es hat eine geringe Komplexität bei einer großen Anzahl von Ausgabesignalen. Die Funktion des Modules wurde über die Schnittstelle zum LART-Board und die Funktionstest nachgewiesen.

In Abschnitt 6.1.4 wurden bereits Probleme angesprochen, die auf den Kompiliervorgang zurückgeführt wurden. Die Zustandsmaschinen blockierten zwar nicht mehr in undefinierten Zuständen, die Fehler waren damit aber nicht behoben. So zeigte die Bildausgabe bei einem Kompilat eine Folge wiederkehrender Streifen. Die Übertragung von Daten zum Hauptspeicher lief nicht. Der FIFO der Videoausgabe musste also leer sein. Ein leerer FIFO gibt jedoch immer das letzte im FIFO befindliche Datenwort zurück. Der FIFO aus den Altera-Bibliotheken hat eine Unterlauf- und Überlaufsicherung. Die Ausgabe hätte ein einfarbiges Bild liefern müssen. Die Streifen deuteten auf ein ständiges Durchlaufen des FIFO-Speicher hin. Das Problem konnte nicht weiter verfolgt werden, da das Ergebnis nicht reproduzierbar war. Nachdem das Empty-Signal des FIFO zur Analyse auf einen Testpunkt gelegt wurde, war der Fehler verschwunden.

Zu Testzwecken wurde der Farbkonverter der Videoeingabe entfernt. Er benötigt ca. 300 Logikeinheiten des FPGA. Danach traten mit verschiedensten Compilereinstellungen keine Fehler mehr auf. Die Module des FPGA-Designs wurden danach auf möglichst geringen Resourcenverbrauch an Logikeinheiten optimiert. Die Fehlerrate der Kompilate sank mit der Veringerung benutzer Logikeinheiten. Nun stellte sich die Frage, wie der Nachweis der vollständigen Funktion eines Kompilates zu führen ist. Fehler traten mit einigen Kompilaten erst nach einem oder mehreren Tagen auf. Die Funktionstests wurden auf mehrere Wochen ausgedehnt. Für diese Tests zeigte sich insbesondere das Programm *xvtmmap* geeignet, da es den Prozessor,

den Speicherbus und den FPGA stark belastet. *xvtmmap* wurde von Ingo Boersch entwickelt. Es ist ein, um eine Bildanalyse erweitertes, vtmmap. *xvtmmap* verarbeitet einen RGB-Bildstrom und extrahiert die Rotanteile des Bildes. Ein Fadenkreuz zeigt den Schwerpunkt der ermittelten Rotanteile. Das Ergebnis wird auf dem Videoausgabegerät angezeigt.

Die Protokolle der Langzeittests sind in Anhang C.4 aufgeführt. Ein entgültiger Nachweis der uneingeschränkten Funktion ist über diese Tests nicht gegeben.

Die Treiber wurden für zwei Kernelversionen implementiert. Bis zum Abschluss der Arbeit veränderten sich die Schnittstellen von Framebuffersubsystem und Video4Linux2-API. Das neue Framebuffersubsystem ist seit der Version 2.5.6 im Kernel integriert. Es waren klare Aussagen über noch zu erwartende Änderungen seit der Version 2.5.20 in der Schnittstellendokumentation enthalten, die bei der Implementierung berücksichtigt wurden. Die Änderungen im Framebuffersubsystem entsprachen bis zur Version 2.5.30 den in der Dokumentation bereits angekündigten Änderungen. Die Video4Linux2-API ist trotz mehrfacher Ankündigung bis jetzt nicht in den Kernel aufgenommen worden. Es sind lediglich einige Vorbereitungen getroffen, die auf eine baldige Aufnahme in den Kernel hindeuten. Innoffizielle Patches sind auf [Kno02] verfügbar, die unter anderem Änderungen in der Registrierung der Hardwaretreiber enthält. Aufgrund fehlender Dokumentation zu den Änderungen war eine Anpassung bisher nicht möglich. Bis zur Version 2.5.30 konnte kein Kernel in einer für das LART-Board lauffähigen Version kompiliert werden. Über die Funktion der Treiber mit dem Entwicklerkernel kann keine Aussage getroffen werden.

Kapitel 7

Zusammenfassung und Ausblick

Die Ziele des BV-Board waren hoch gesteckt. Bildverarbeitung auf Embedded Geräten ist heute noch eine Herausforderung. Um keine unrealistischen Anforderungen an das BV-Board zu stellen wurde zunächst die Hardware des LART-Board und des LartVIO untersucht. Die Untersuchungen zeigten insbesondere Resourcenengpässe beim Speicherbus. Der Hauptspeicherbus wurde deshalb genauer untersucht.

Die Treiberarchitektur sollte unter Berücksichtigung aktueller Entwicklungen im Betriebssystemkernel entworfen werden. Die Aufgabenstellung verwies für die Umsetzung der Treiber auf die Kernelsubsysteme Framebuffer und Video4Linux. Die Subsysteme wurden in der aktuell stabilen Version und der in Entwicklung befindlichen Version betrachtet. Beide Subsysteme waren für die Umsetzung einer Videodigitalisierung und Videoausgabe geeignet.

Unter Berücksichtigung der anvisierten Nutzergruppen und der Ergebnisse aus der Untersuchung von Hard- und Software wurden die Anforderungen an die Videodigitalisierung und -ausgabe ausgearbeitet. Hauptanforderungen waren:

- 720x576 und QSIF 360x288 bei 25 Bildern/s
- Deinterlacing der Videobilder.
- Farbformet RGB24, andere Formate optional.
- Der Prozessor muss genügend Leistungsreserven für die Verarbeitung der Bilder haben.

Anhand der Anforderungen wurde die Übertragung der Videodaten über den Hauptspeicherbus untersucht. Der erste Ansatz ging davon aus, dass die Funktionalität in den Betriebssystemtreibern integriert wird. Die benötigten Übertragunsraten für den Videostrom lagen über den maximal möglichen

Übertragensraten des Hauptspeicherbus. Das Konzept wurde so überarbeitet, dass der auf dem LartVIO integrierte Videocontroller die Übertragung ohne Beteiligung des Prozessors erledigt. Die maximale Übertragunsrate konnte mit dieser Lösung um zwei Drittel reduziert werden. Da der Prozessor an der Übertragung der Videodaten nicht mehr beteiligt sein sollte, mussten alle den Videostrom verarbeitenden Funktionen ebenfalls in den Videocontroller integriert werden. Die Gerätetreiber übernahmen nur noch die Initialiserung der Hardware.

Die Treibersubsysteme für die Version des Betriebssystems waren zum Zeitpunkt der Diplomarbeit in einem frühen Entwicklungsstadium. Die Schnittstellen zu den Hardwaretreibern wurden häufig geändert. Das Framebuffersubsystem änderte vor allem die Schnittstellen zu den Gerätetreibern. Die Funktionalität des Treibers änderte sich nur geringfügig. Für das Video4Linux Subsystem ist bereits seit 1999 eine zweite, stark erweiterte Version verfügbar, die die alte Version des Video4Linux Subsystems in der nächsten stabilen Betriebssystemversion ablösen soll. Da Video4Linux2 auch für den derzeit aktuellen Kernel kompiliert werden kann, wurde Video4Linux2 als Subsystem für den Videoeingabetreiber gewählt.

In der Anforderungsanalyse fiel die Auswahl der notwendigen Farbformate schwer. Jedes Farbformat hat seine Stärken und Schwächen. Je nach Anwendungsfall sind unterschiedliche Anforderungen an das Farbformat zu stellen. Aus diesem Grund wurde eine einfache Schnittstelle zur Implementierung eigener Farbraumkonverter und anderer videoverarbeitenden Einheiten entworfen. Bis zu vier verschiedene Farbraumkonverter können vom Treiber ausgewählt werden. Denkbar sind aber auch verschiedene Videofilter, die zugeschalten werden können.

Das Design des Videocontroller ist modular aufgebaut. Damit läßt sich das LartVIO an veränderte Hardware leicht anpassen und erweitern.

Das Ergebnis nach der Implementierung war überwältigend. Bilder mit einer Auflösung von 720x768 Bildpunkten und 16 Millionen Farben konnten mit 25 Bilder pro Sekunde von der Kamera übertragen und wieder auf einem TV-Gerät ausgegeben werden. Der Videocontroller belastet den Hauptspeicherbus dabei mit ca. 65%. Mehr als ein Drittel steht damit noch Anwendungen zur Verfügung. Wird die Videoausgabe abgeschalten, stehen Anwendungen 60% zur Verfügung. Kleinere Anwendungen mit Bildanalyse konnten 25 Bilder pro Sekunde der Größe 360x288 mit 16 Millionen Farben verarbeiten.

Getrübt wird das über positive Gesamtbild von einigen Kompilierfehlern, die den Videocontroller betreffen. Es wurden verschiedene Maßnahmen getroffen, die die Fehler minimierten. Bis zum Abschluß der Arbeit konnte die Ursache der Fehler nicht vollständig geklärt werden. Bedingt durch die-

se Fehler konnte der Nachweis der Funktionstüchtigkeit des Produktes nicht vollständig erbracht werden.

Die leichte Erweiterbarkeit des Designs ist dadurch beeinträchtigt. Zum Einen ist der Funktionsnachweis sehr langwierig, zum Anderen war eine Maßnahme zur Minimierung der Kompilerfehler die Reduzierung der Codegröße. Ist die Ursache für die Kompilerfehler gefunden, kann das Produkt auch für den industriellen Einsatz uneingeschränkt empfohlen werden.

Anhang A

Analyse

A.1 Überprüfung der Speichertransferleistung

In Abschnitt 2.2.3 wurde die Speichertransferleistung rechnerisch ermittelt. Die Berechnungen sollen unter realen Bedingungen getestet werden. Das Programm sollte reale Bildgrössen von einem Speicherbereich in einen anderen Speicherbereich kopieren, mit dem Versuch *Burst Of Eight* Zugriffe auf den Speicher zu generieren. Die Zugriffe auf den Speicher müssen verifiziert werden.

A.1.1 Übertragung mit memcpy()

In der ersten Version des Programmes mem2mem wurde der Speicherbereich mit der Funktion memcpy() übertragen. memcpy ist bereits assembleroptimiert (Kernel-2.4.18-rmk6: /arch/arm/lib/memcpy.S).

Die Speicherzugriffe wurden mit einem Zweistrahl-Oszilloskop an den Steuerleitungen nRAS und nCAS überprüft. Die Übertragung erfolgte mit *Burst Of Four* Speicherzugriffen. Die Datentransferrate liegt ca. 35% unter der berechneten Transferrate. Wahrscheinlich verlangsamt der Zugriff auf die Funktion die Geschwindigkeit.

A.1.2 Übertragung mit asmcpy()

Da die Übertragungsleistung von memcpy() nicht mit *Burst Of Eight* erfolgte, wurde eine Assemblerroute in das Programm mem2mem integriert. Die Assemblerroutine nutzt Multiple-Load und Multiple-Store Befehle. Mit einem Befehl werden acht 32Bit Register gleichzeitig geschrieben oder gelesen. Die Funktion wird durch Übergabe des Parameters `-asm` gestartet.

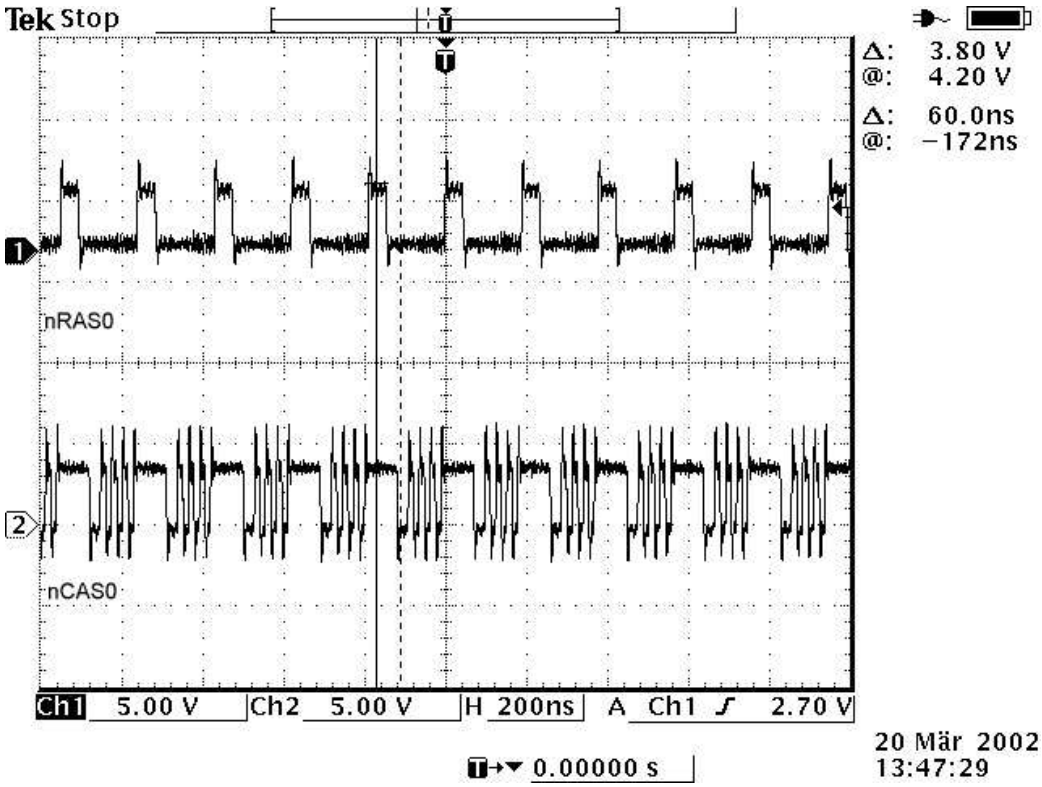

Abbildung A.1: Steuerleitungen nRAS und nCAS bei Datentransfer mit memcpy()

Framegrösse	memcpy	
	Frames/s	MByte/s
320, 240, 2	172.4	50.5
320, 240, 3	115.2	50.6
640, 480, 2	43.1	50.5
640, 480, 3	28.8	50.6
768, 576, 2	29.9	50.6
768, 576, 3	19.9	50.6

Tabelle A.1: Datentransferraten mit memcpy()

Im Vergleich zur Funktion memcpy() sind die Übertragungsraten um 50% höher. Die Übertragung erfolgt mit *Burst Of Eight* bei Lesezugriffen und zwei *Burst Of Two* und einem *Burst Of Four* bei Schreibzugriffen.

Framegrösse	memcpy -asm	
	Frames/s	MByte/s
320, 240, 2	263.2	77.1
320, 240, 3	65.8	77.4
640, 480, 2	43.2	77.1
640, 480, 3	43.9	77.2
768, 576, 2	45.8	77.3
768, 576, 3	30.5	77.3

Tabelle A.2: Datentransferraten mit asmcpy()

A.1.3 Speicheradressen bündig

Die unterschiedlichen Zugriffsarten ließen vermuten, dass die Startadressen der Puffer nicht durch 64 teilbar sind [Int99, 10.1.5 Transaction Summary]. Eine Überprüfung der angeforderten Pufferadressen bestätigte die Vermutung. Das Programm mem2mem wurde um den Parameter `-align` erweitert, der die Startadressen der Puffer auf durch 64 teilbare Adressen legt.

Framegrösse	memcpy		memcpy -asm	
	Frames/s	MByte/s	Frames/s	MByte/s
320, 240, 2	173.6	50.8	294.1	86.2
320, 240, 3	115.2	50.6	195.3	85.8
640, 480, 2	43.2	50.7	73.3	85.9
640, 480, 3	28.8	50.6	48.9	86.0
768, 576, 2	30.0	50.7	51.0	86.0
768, 576, 3	20.0	50.7	33.9	86.0

Tabelle A.3: Datentransferraten mit angepassten Startadressen (beide Aufrufe mit -align)

Bei Leseoperationen wurden weiterhin mit *Burst Of Eight* auf den Speicher zugegriffen. Schreiboperationen erzeugten zwei *Burst Of Four*. In [Int99, 6.3 Write Buffer (WB)] wird die Grösse der Write Buffer Blöcke mit bis zu 16 Byte angegeben, was bei einem Flush *Burst Of Four* erzeugt. *Burst Of Eight* werden beim Zurückkopieren des Cache durch den Write Buffer erzeugt. Experimente mit dem Leeren des Datencaches zeigten *Burst Of Eight* Perioden.

Da aber der ganze Cache zurückgeschrieben wurde, beinflusste das Leeren des Datencache das Ergebnis negativ. Da die Ergebnisse nicht reproduzierbare Werte ergaben, wurden sie nicht aufgenommen.

Die Speichertests wurden an dieser Stelle abgebrochen, da die Ergebnisse bereits Aussagen im Vergleich zu den berechneten Speichertransferraten zulassen.

A.1.4 Das Programm mem2mem

```
/** Test to get the maximum transfer rate on a system
  * moving frames from a memory area to an other.
  * You can add your own frame size in structur frames[].
  * Be shure the defined frames in structur frames[] are
  * smaller than MAX_FRAMESIZE
  *
  * Author: Frank Schwanz    EMail: schwanz-at-fh-brandenburg.de
  * Version: 1.2             Date:  2002-03-21
  *
  * Revision:  1.0    2002-03-18    initial                    10
  *            1.1    2002-03-20    assembler memcpy added
  *            1.2    2002-03-21    align buffer
  */

#include <stdio.h>
#include <string.h>
#include <time.h>


                                                              20

#define MAXFRAMESIZE 1024 * 1024 * 2 /* 2 MB */
#define BUFSIZE 1024
#define FRAMES_PER_SECOND 25

/** to get better results we need to
 * run more than a second
 */
#define SECONDS 10
                                                              30

struct frame {
        int height;
        int width;
        int colorbytes;
};
/** Different frame sizes to test.
```

```
 *      { height, width, bytes per pixel }
 */
struct frame frames[] = {                                              40
        {320, 240, 2},
        {320, 240, 3},
        {640, 480, 2},
        {640, 480, 3},
        {768, 576, 2},
        {768, 576, 3}
};


int main(int argc, char **argv)                                        50
{
        char *fb_in, *fb_out;
        char *fb_in_align, *fb_out_align;
        char *bufincnt, *bufoutcnt;
        int i, j, k, asmcpy = 0, memalign = 0, framesize, buffs_for_frame;
        clock_t cl1, cl2;
        float time;

        printf("%s [-asm] [-align]\n", argv[0]);
        printf("[-asm] using assembler memcpy\n");                     60
        printf("[-align] using align memory address\n\n");


        for (i = 1; i < argc; i++) {
                if (strcmp(argv[i], "-asm") == 0) {
                        asmcpy = 1;
                        printf("Using assembler memcpy\n");
                }
                if (strcmp(argv[i], "-align") == 0) {
                        memalign = 1;                                  70
                        printf("Using align memoryaddress\n");
                }
        }

        /*
         * create in and out buffers
         */
        fb_in = (char *) malloc(MAXFRAMESIZE + BUFSIZE);
        fb_out = (char *) malloc(MAXFRAMESIZE + BUFSIZE);
        /*                                                             80
         * make address align to 1024, to enable Burst Of Eight
         */
        if (memalign) {
                fb_in_align =
                    (char *) (((unsigned long) (fb_in + BUFSIZE - 1)) &
                              ~((unsigned long) BUFSIZE - 1));
```

```
            fb_out_align =
                (char *) (((unsigned long) (fb_out + BUFSIZE - 1)) &
                        ~((unsigned long) BUFSIZE - 1));
        } else {                                                        90
            fb_in_align = fb_in;
            fb_out_align = fb_out;
        }

        /*
         * How many frames to test (the number of entries in frames[])
         */
        for (j = 0; j < sizeof(frames) / sizeof(struct frame); j++) {
            /* height * width * bytes per pixel */
            framesize =                                                 100
                frames[j].height * frames[j].width *
                frames[j].colorbytes;

            /* frame will copied in blocks of BUFFSIZE */
            buffs_for_frame = framesize / BUFSIZE;

            /* get time before work */
            cl1 = clock();
            for (k = 0; k < FRAMES_PER_SECOND * SECONDS; k++) {
                bufincnt = fb_in_align;                                 110
                bufoutcnt = fb_out_align;

                for (i = 0; i < buffs_for_frame; i++) {
#ifdef ARM_LINUX
                    if (asmcpy) {

        asm volatile(
            ".align 4
            stmfd  sp!,{r0-r11}  /* push register r0-r11 */
            ldr r0, %0           /* r0 := wordbufin  */                 120
            ldr r1, %1           /* r1 := wordbufout */
            mov r3, #32          /* 1024 / 4 Bytes / 8 words */
        copy:
            LDMIA r0!, {r4-r11} /* read 8 words to register */
            STMIA r1!, {r4-r11} /* write 8 words to memory */
            SUBS r3, r3, #1     /* dec r3             */
            BNE copy            /* loop if r3 not 0 */

            ldmfd  sp!,{r0-r11} /* pop register r0-r11 */
            "::"m" (bufincnt), "m"(bufoutcnt)                           130
        );

                    } else {
                        memcpy(bufoutcnt, bufincnt,
                            BUFSIZE);
```

```
                        }
#else
                        memcpy(bufoutcnt, bufincnt, BUFSIZE);
#endif
                        bufoutcnt += BUFSIZE;                                    140
                        bufincnt += BUFSIZE;

                    }
                }
                /* get time after work */
                cl2 = clock();
                /* time of work in seconds */
                time = ((float) (cl2 - cl1)) / (float) CLOCKS_PER_SEC;
                printf
                    ("FRAMESIZE[%d, %d, %d] = %d Bytes\n",                       150
                     frames[j].height, frames[j].width,
                     frames[j].colorbytes,
                     frames[j].height * frames[j].width *
                     frames[j].colorbytes,);
                printf("TIME to copy %d FRAMES = %f\n",
                       FRAMES_PER_SECOND * SECONDS, time);
                printf("%f FRAMES/PER SECOND\n",
                       ((float) FRAMES_PER_SECOND * SECONDS) / time);
                printf("%f MB/PER SECOND\n",
                       (((float) FRAMES_PER_SECOND * SECONDS) * framesize /      160
                        time) / 1024 / 1024 * 2);
                printf("==================================\n");
        }
        free(fb_in);
        free(fb_out);
        return 0;
}
```

Anhang B

Listings

B.1 AHDL-Quellcode

B.1.1 YCbCr_YCbCr.tdf (Auszug)

```
-- Subdesign Section
SUBDESIGN YCbCr_YCbCr
(
        data[31..0]:    INPUT; -- YCbCr Daten
        data_en:        INPUT; -- Signal für neue Daten am Eingang

        reset:          INPUT; -- Rücksetzten der Logik
        -- Parameter
        FULLSIZE:       INPUT; -- =0 : Am Ausgang soll nur jeder zweite Pixel
                               -- ausgegeben werden (Halfsize ==High)
        -- Misc signals
        enable:         INPUT; -- Enable Decoder Logik
        Clock:          INPUT; -- Takt

        q_rd:           OUTPUT; -- Neues Datenwort liegt bereit (1 Takt);
        q[31..0]:       OUTPUT; -- Datenwort in der Form (YCbCrYCbCr)

)

-- Variable declaration
VARIABLE
        Y1[7..0]:       NODE;
        Y2[7..0]:       NODE;
        Cr[7..0]:       NODE;
        Cb[7..0]:       NODE;
```

```
        TOGGLE_SM: MACHINE WITH STATES ( FIRST,
                                         SECOND);
                                                                        30
-- Implementing design
BEGIN
        --************ YCbCr => YCbCr Konvertierung ************_
        -- Der Decoder füllt das Schieberegister in 4 Takten,
        -- in dieser Zeit müssen beide Y und Cb,Cr in der Verarbeitungs-
        -- pipeline sein.
        -- Die Werte werden ohne Verzögerung berechnet. Bei der
        -- Halbierung der Daten wird jedes 2. Doppelpixel
        -- verworfen. Die Reduzierung der Daten ist verbesserungswürdig.
                                                                        40

        Cb[] = data[7..0];
        Y1[] = data[15..8];
        Cr[] = data[23..16];
        Y2[] = data[31..24];

        q[7..0]   = Cb[];
        q[15..8] = Y1[];
        q[23..16] = Cr[];
        q[31..24] = Y2[];                                               50

        TOGGLE_SM.clk = Clock;
        CASE (TOGGLE_SM) IS
                WHEN SECOND =>
                        q_rd = data_en AND enable AND FULLSIZE;
                        IF ((data_en) OR (reset)) THEN
                                TOGGLE_SM = FIRST;
                        END IF;
                WHEN OTHERS %FIRST% =>
                        q_rd = data_en AND enable;                      60
                        IF ((data_en) AND (!reset)) THEN
                                TOGGLE_SM = SECOND;
                        END IF;
        END CASE; -- TOGGLE State Machine


END;
```

B.2 Auszüge aus lartviofb.c (Framebuffertreiber)

B.2.1 Die Funktion lartviofb_check_var bzw. lartviofb_decode_var

```
/**
 *      lartviofb_check_var - Validates a var passed in.
 *      var: frame buffer variable screen structure
 *      info: frame buffer structure that represents a single frame buffer
 *
 * Checks to see if the hardware supports the state requested by
 * var passed in. This function does not alter the hardware state!!!
 * This means the data stored in struct fb_info and struct lartvio_par do
 *      not change. This includes the var inside of struct fb_info.
 * Do NOT change these. This function can be called on its own if we
 * intent to only test a mode and not actually set it. The stuff in
 * modedb.c is a example of this. If the var passed in is slightly
 * off by what the hardware can support then we alter the var PASSED in
 * to what we can do. If the hardware doesn't support mode change
 * a -EINVAL will be returned by the upper layers. You don't need to
 * implement this function then.
 *
 * Returns negative errno on error, or zero on success.
 */

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
static int lartviofb_check_var(struct fb_var_screeninfo *var, struct fb_info *info)
{
    struct lartvio_par *par = (struct lartvio_par *) info->par;
    int line_length;
#else
static int lartviofb_decode_var(const struct fb_var_screeninfo *var, void *vpar,
            struct fb_info_gen *gen)
{
    struct lartvio_par *par = (struct lartvio_par *) vpar;
    struct lartvio_par *gpar = (struct lartvio_par *) gen->info.par;
    int line_length;
    memcpy (&par->mvar, var, sizeof(struct fb_var_screeninfo));
    par->videomemorysize = gpar->videomemorysize;
    par->videomemory = gpar->videomemory;
    par->map_dma    = gpar->map_dma;
    par->map_size   = gpar->map_size;
    par->encaddr    = gpar->encaddr;
    par->adv        = gpar->adv;
```

```
#endif
        if ((var->xres < 144) || (var->yres < 80) /* min 144x80 Pixel */
       || (var->xres > 720) || (var->yres > 576) /* max PAL size */
           || (var->xres%4) || (var->yres%4)
           || (var->xres_virtual%4) || (var->yres_virtual%4)
           || (var->xoffset%4) || (var->yoffset%4))    /* size must be modulo 4 */
    {
        return -EINVAL;
    }

        /* we have only one fullsizeflag for both vertical and
           horizontal, so both must be fullsize or halfsize */
        if (((var->xres>360) && (var->yres <= 288))
           || ((var->xres<=360) && (var->yres > 288)))
            return -EINVAL;

    if (var->xres > var->xres_virtual)
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,5,6)
        var->xres_virtual = var->xres;
#else
            return -EINVAL;
#endif
    if (var->yres > var->yres_virtual)
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,5,6)
        var->yres_virtual = var->yres;
#else
            return -EINVAL;
#endif
    if (var->xres_virtual < var->xoffset + var->xres)
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,5,6)
        var->xres_virtual = var->xoffset + var->xres;
#else
            return -EINVAL;
#endif

    if (var->yres_virtual < var->yoffset + var->yres)
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,5,6)
        var->yres_virtual = var->yoffset + var->yres;
#else
            return -EINVAL;
#endif
    /*
     * Memory limit
     */
    line_length =
        get_line_length(var->xres_virtual, var->bits_per_pixel);
    if (line_length * var->yres_virtual > par->videomemorysize)
        return -ENOMEM;
```

```
    /*                                                                    90
     * Now that we checked it we alter var. The reason being is that the video
     * mode passed in might not work but slight changes to it might make it
     * work. This way we let the user know what is acceptable.
     */
    switch (var->bits_per_pixel) {
    case 24:        /* RGB 888 */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,5,6)
        var->red.offset = 0;
        var->red.length = 8;
        var->green.offset = 8;                                            100
        var->green.length = 8;
        var->blue.offset = 16;
        var->blue.length = 8;
        var->transp.offset = 0;
        var->transp.length = 0;
#endif
        break;
    default:
        return -EINVAL;
    }                                                                     110
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,5,6)
    var->red.msb_right = 0;
    var->green.msb_right = 0;
    var->blue.msb_right = 0;
    var->transp.msb_right = 0;
#endif
    return 0;
}
```

B.2.2 Die Funktion lartviofb_set_par

```
/**
 *      lartviofb_set_par -   Alters the hardware state.
 *      info: frame buffer structure that represents a single frame buffer
 *
 *      Using the fb_var_screeninfo in fb_info we set the resolution of the
 *      this particular framebuffer. This function alters the par and the
 *      fb_fix_screeninfo stored in fb_info. It doesn't not alter var in
 *      fb_info since we are using that data. This means we depend on the
 *      data in var inside fb_info to be supported by the hardware.
 *      lartviofb_check_var is always called before lartviofb_set_par to ensure this.   10
 *
 *      In 2.4.x useing var from vpar.
 */
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
static int lartviofb_set_par(struct fb_info *info)
{
        struct lartvio_par *par = (struct lartvio_par *) info->par;
        struct fb_var_screeninfo *var = &info->var;
#else

static void lartviofb_set_par(const void *vpar, struct fb_info_gen *gen)
{
        struct lartvio_par *par = (struct lartvio_par *) vpar;
        struct fb_var_screeninfo *var = &par->mvar;
#endif
        unsigned int ressize, rowoffset, rowcount;
        unsigned int xborder, yborder, yvideolen, xvideolen;
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,5,6)
        memcpy(&lartviofb_var, &par->mvar,
               sizeof(struct fb_var_screeninfo));
#endif

        /* PAL, if NTSC will be added you will need to change it depends on
           the standard */
        yvideolen = 287;
        xvideolen = 359;

        *(par->encaddr + LARTVIO_EADRSO) =
            par->map_dma + var->xoffset * (var->bits_per_pixel / 8)
            + var->yoffset * get_line_length(var->xres_virtual,
                                             var->bits_per_pixel);

        if (var->yres > 288) {
                *(par->encaddr + LARTVIO_EADRSE) =
                    *(par->encaddr + LARTVIO_EADRSO)
                    + get_line_length(var->xres_virtual,
                                      var->bits_per_pixel);
                rowoffset = (2 * var->xres_virtual -
                             var->xres) * (var->bits_per_pixel / 8);
                /* xvideolen - var->xres = available border
                   half for a border left and same rigth
                   second half comes from technical manual LartVIO */
                xborder = (2 * (xvideolen + 1) - var->xres) / 2 / 2;
                /* xvideolen - var->xres = available border
                   half for a border top and same buttom
                   second half comes from technical manual LartVIO */
                yborder = (2 * (yvideolen + 1) - var->yres) / 2 / 2;
                *(par->encaddr + LARTVIO_ENCC) |= ENCC_FULLSIZE;
        } else {
                *(par->encaddr + LARTVIO_EADRSE) =
                    *(par->encaddr + LARTVIO_EADRSO);
```

```
                *(par->encaddr + LARTVIO_ENCC) &= ~ENCC_FULLSIZE;
                rowoffset = (var->xres_virtual -
                        var->xres) * (var->bits_per_pixel / 8);
                /* xvideolen - var->xres = available border
                   half for a border left and same rigth
                   second half comes from technical manual LartVIO */
                xborder = (xvideolen + 1 - var->xres) / 2;
                /* xvideolen - var->xres = available border
                   half for a border top and same buttom
                   second half comes from technical manual LartVIO */
                yborder = (yvideolen + 1 - var->yres) / 2;
        }

        ressize = var->xres * var->yres * (var->bits_per_pixel / 8);
        if (ressize > par->map_size) {
                printk(KERN_ERR
                      "lartviofb: Size for framebuffer too small\n");
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
                return -EINVAL;
#else
                return;
#endif
        }

        rowcount = (var->xres) * (var->bits_per_pixel / 8);

        *(par->encaddr + LARTVIO_EBORDER) =
            (((yvideolen << 23) & LARTVIO_EBORDER_ENCYL) |
             ((yborder << 16) & LARTVIO_EBORDER_ENCYB) | ((xvideolen << 7) &
                                                  LARTVIO_EBORDER_ENCXL)
             | (xborder & LARTVIO_EBORDER_ENCXB));

        *(par->encaddr + LARTVIO_ERCNTO) =
            ((((rowoffset / 4) << 16) & LARTVIO_ERCNTO_ROWOFFSET) |
             ((rowcount / 4) & LARTVIO_ERCNTO_ROWCNT));

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
        return 0;
#endif

}
```

B.2.3 Die Funktion lartviofb_init_fpga

```
/**
 *   lartviofb_init_fpga - Setup communikation to LartVIO..
 *   par:  hardware parameter structure
 *
 *      Setup membank2, check LartVIO and setup i2c-driver
 *
 *   Returns negative errno on error, or zero on success.
 */
static int lartviofb_init_fpga(struct lartvio_par *par)                    10
{
        int arg = 0;

        setup_membank2();
        enable_busclock();
        /* FPGA available ? */
        if (lartvio_available())
                return -ENODEV;
        par->encaddr =
            ioremap_nocache(LARTVIO_ENC_ADR, LARTVIO_ENC_ADR_SIZE);        20
        if (!par->encaddr)
                return -ENOMEM;

        /* enable and reset encoder */
        *(par->encaddr + LARTVIO_ENCC) |= (ENCC_LOGIC_ENABLE |
                                           ENCC_RESET | ENCC_CLOCK_ENABLE);
        udelay(1000);
        *(par->encaddr + LARTVIO_ENCC) &= ~ENCC_RESET;
        udelay(500);
        /* init the i2c_driver */                                          30
        /* load needed modules */
        request_module("i2c_lartvio");
        request_module("adv7170");
        par->adv =
            i2c_get_client(I2C_DRIVERID_ADV7170,
                           I2C_HW_B_LARTVIO | I2C_ALGO_BIT, par->adv);
        if (!par->adv) {
                printk(KERN_ERR "no ADV7170 driver found");
                return -EINVAL;
        }                                                                  40
        if (i2c_use_client(par->adv) < 0) {
                printk(KERN_ERR "unable to use ADV7170 driver");
                return -EINVAL;
        }

        /* we reseted the encoder so we need to init it again */
```

```
        par->adv->driver->command(par->adv, ADV7170_INIT, &arg);
        arg = VIDEO_MODE_PAL;
        par->adv->driver->command(par->adv, SET_VIDEO_NORM, &arg);
        return 0;                                                          50
}
```

B.2.4 Die Funktion lartviofb_encode_fix

```
/**
 *      lartviofb_encode_fix - fill in the 'fix' structure based on the values
 *      in the 'par' structure.
 *      fix: fb_fix_screeninfo which will be filled
 *      vpar: parameter structure
 *      gen: fbinfogen. includes all parameter structures
 *
 *      Returns negative errno on error, or zero on success.
 */
static int lartviofb_encode_fix(struct fb_fix_screeninfo *fix,             10
                                const void *vpar, struct fb_info_gen *gen)
{
        struct lartvio_par *par = (struct lartvio_par *) gen->info.par;
        memset(fix, 0, sizeof(struct fb_fix_screeninfo));
        strcpy(fix->id, MODULENAME);
        fix->line_length =
            get_line_length(lartviofb_var.xres_virtual,
                            lartviofb_var.bits_per_pixel);
        fix->smem_start = par->map_dma;
        fix->smem_len = par->videomemorysize;                              20
        fix->type = FB_TYPE_PACKED_PIXELS;
        fix->type_aux = 0;
        fix->visual = FB_VISUAL_TRUECOLOR;
        fix->ywrapstep = 0;
        fix->xpanstep = 0;
        fix->ypanstep = 0;
        fix->accel = FB_ACCEL_NONE;
        return 0;
}
```

B.2.5 Die Funktion lartviofb_encode_var

```
/**
 *      lartviofb_encode_var - return current var structure.
 *      var: fb_var_screeninfo which will be filled
 *      vpar: parameter structure
```

```
 *      gen: fbinfogen. includes all parameter structures
 *
 *      Returns negative errno on error, or zero on success.
 */
static int lartviofb_encode_var(struct fb_var_screeninfo *var,
                                const void *vpar, struct fb_info_gen *gen)
{
        var = (struct fb_var_screeninfo *) &gen->info.var;
        return 0;
}
```

B.2.6 Die Funktion lartviofb_blank

```
/**
 *      lartviofb_blank - Blanks the display.
 *      blank_mode: the blank mode we want.
 *      info: frame buffer structure that represents a single frame buffer
 *
 *      Blank the screen if blank_mode != 0, else unblank. Return 0 if
 *      blanking succeeded, != 0 if un-/blanking failed due to e.g. a
 *      video mode which doesn't support it. Implements VESA suspend
 *      and powerdown modes on hardware that supports disabling hsync/vsync:
 *      blank_mode == 2: suspend vsync
 *      blank_mode == 3: suspend hsync
 *      blank_mode == 4: powerdown
 *
 *      LartVIO will powered down if blank_mode !=0.
 *
 *      Returns negative errno on error, or zero on success.
 *
 */

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
static int lartviofb_blank(int blank_mode, struct fb_info *info)
#else
static void lartviofb_blank(int blank_mode, struct fb_info *info)
#endif
{
        int arg;
        int nblank = !blank_mode;
        struct lartvio_par *par = (struct lartvio_par *) info->par;

        /* disable and reset encoder */

        if (blank_mode) {
                par->adv->driver->command(par->adv, ENABLE_OUTPUT,
```

```
                                        &nblank);
                *(par->encaddr + LARTVIO_ENCC) |= ENCC_RESET;
                *(par->encaddr + LARTVIO_ENCC) &=
                    ~(ENCC_CLOCK_ENABLE | ENCC_LOGIC_ENABLE);
        } else {
                /* enable and reset encoder */
                *(par->encaddr + LARTVIO_ENCC) |= (ENCC_LOGIC_ENABLE |
                                                   ENCC_RESET |
                                                   ENCC_CLOCK_ENABLE);


                udelay(1000);
                *(par->encaddr + LARTVIO_ENCC) &= ~ENCC_RESET;
                udelay(1000);
                /* we reseted the encoder so we need to init it again */
                par->adv->driver->command(par->adv, ADV7170_INIT, &arg);
                arg = VIDEO_MODE_PAL;
                par->adv->driver->command(par->adv, SET_VIDEO_NORM, &arg);
                par->adv->driver->command(par->adv, ENABLE_OUTPUT,
                                        &nblank);
        }
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,6)
        return 0;
#endif
}
```

B.3 Auszüge aus lartvio_vin.c (Video4Linux2-Treiber)

B.3.1 Die Funktion mmap_request_buffers

```
/**
 *      mmap_request_buffers -
 *              setup streaming buffers. it trys to setup req->count buffers.
 *              but if no such many buffers available it change req->count.
 *
 *      dev:            device structure
 *      req:            request structure  v4l2_requestbuffers
 *
 *      Returns 1 = success; 0 = failed.
 */
static int mmap_request_buffers(struct capture_device *dev,
                                struct v4l2_requestbuffers *req)
{
        int i, lineoffset;
        u32 buflen;
        u32 type;

        if (dev->stream_buffers_mapped)
                return 0;       /* can't make requests if buffers are mapped */
        if (req->count < 1)
                req->count = 1;
        if (req->count > MAX_CAPTURE_BUFFERS)
                req->count = MAX_CAPTURE_BUFFERS;
        type = V4L2_BUF_TYPE_CAPTURE;

        /* The buffer length needs to be a multiple of the page size */
        buflen = (dev->clientfmt.sizeimage + PAGE_SIZE - 1)
            & ~(PAGE_SIZE - 1);
        if (req->count * buflen > dev->map_size)
                req->count = dev->map_size / buflen;

        for (i = 0; i < req->count; ++i) {
                dev->stream_buf[i].requested = 1;
                dev->stream_buf[i].vidbuf.index = i;
                dev->stream_buf[i].vidbuf.type = type;
                dev->stream_buf[i].map_dma = dev->map_dma + (i * buflen);
                /* offset must be unique for each buffer, and a multiple */
                /* of PAGE_SIZE on 2.4.x */
                dev->stream_buf[i].vidbuf.offset = PAGE_SIZE * (i + 1);
                dev->stream_buf[i].vidbuf.length = buflen;
                dev->stream_buf[i].vidbuf.bytesused = 0;
```

```
                dev->stream_buf[i].vidbuf.flags = 0;
                dev->stream_buf[i].vidbuf.timestamp = 0;
                dev->stream_buf[i].vidbuf.sequence = 0;
                memset(&dev->stream_buf[i].vidbuf.timecode, 0,
                        sizeof(struct v4l2_timecode));

                /* we calc here the fpga register, so we are faster while interrupt */

                dev->stream_buf[i].dadrso = dev->stream_buf[i].map_dma;
                dev->stream_buf[i].dadre = dev->stream_buf[i].map_dma
                    +
                    (dev->clientfmt.height * dev->clientfmt.bytesperline);
                if ((dev->clientfmt.height == 288)) {
                        dev->stream_buf[i].dadrse =
                            dev->stream_buf[i].map_dma;
                        lineoffset =
                            dev->clientfmt.bytesperline -
                            dev->clientfmt.width * dev->clientfmt.depth /
                            8;
                } else {
                        dev->stream_buf[i].dadrse =
                            dev->stream_buf[i].map_dma +
                            dev->clientfmt.bytesperline;
                        lineoffset =
                            2 * dev->clientfmt.bytesperline -
                            dev->clientfmt.width * dev->clientfmt.depth /
                            8;
                }
                dev->stream_buf[i].drcnto =
                    /* linebytes / 4 */
                    (((dev->clientfmt.width * dev->clientfmt.depth) >> 5)
                     & LARTVIO_DRCNTO_ROWCNT) |
                    /* lineoffset /4 */
                    (((lineoffset / 4) << 16) & LARTVIO_DRCNTO_ROWOFFSET);
        }
        for (i = req->count; i < MAX_CAPTURE_BUFFERS; ++i)
                dev->stream_buf[i].requested = 0;
        dev->stream_buffers_requested = req->count;

        return 1;
}
```

B.3.2 Die Funktion capture_queuebuffer

```
/**
 *      capture_queuebuffer - Add a stream buffer to capture queue
 *
 *      dev:            device structure
 *      vidbuf: stream buffer
 *
 *      Returns 1 = success; 0 = failed.
 */
static int capture_queuebuffer(struct capture_device *dev,
                               struct v4l2_buffer *vidbuf)                    10
{
        int i = vidbuf->index;
        struct stream_buffer *buf = NULL;

        if (vidbuf->type != V4L2_BUF_TYPE_CAPTURE) {
                printk(KERN_INFO "QBUF wrong type\n");
                return 0;
        }
        if (i < 0 || i >= MAX_CAPTURE_BUFFERS
            || !dev->stream_buf[i].requested) {                               20
                printk(KERN_INFO "QBUF buffer index %d is out of range\n",
                       i);
                return 0;
        }

        buf = &dev->stream_buf[i];

        if ((buf->vidbuf.flags & V4L2_BUF_FLAG_QUEUED)) {
                printk(KERN_INFO "QBUF buffer %d is already queued\n", i);
                return 0;                                                     30
        }

        buf->vidbuf.flags &= ~V4L2_BUF_FLAG_DONE;
        v4l2_q_add_tail(&dev->stream_q_capture, &buf->qnode);
        buf->vidbuf.flags |= V4L2_BUF_FLAG_QUEUED;
        dev->ready_to_capture = 1;
        /* if streaming and buffers was emtpy the grabber will be disabled
         * to get more space on the bus. Now we have to enable it again.
         */
        if ((dev->state == STREAMING) || (dev->state == READING))             40
                if (!dev->grabber_enabled) {
                        capture_grab_frame(dev);
                }
        return 1;
}
```

B.3.3 Die Funktion mmap_unrequest_buffers

```
/**
 *      mmap_unrequest_buffers - unrequest all mapped buffers
 *
 *      dev:            device structure
 *
 */
static void mmap_unrequest_buffers(struct capture_device *dev)
{
        int i;
        if (dev->grabber_enabled)                                          10
                grabbing_enable(dev, 0);
        if (dev->stream_buffers_requested == 0 ||
            dev->stream_buffers_mapped)
                return;
        for (i = 0; i < MAX_CAPTURE_BUFFERS; ++i)
                dev->stream_buf[i].requested = 0;
        dev->stream_buffers_requested = 0;
}
```

Anhang C

Testprotokolle

C.1 Framebuffer-API Test

Testbeschreibung	**Ergebnis**	**Sichtprüfung**	
		Farbtest	**Auflösung**
PAL 720x576 Grösse normal, RGB24	OK	OK	OK
Überschreitung der Maximalbreite der sichtbaren Auflösung 1024x576	OK	nicht durchgeführt	nicht durchgeführt
Überschreitung der Maximalbreite der virtuellen Auflösung 1024x576	OK	nicht durchgeführt	nicht durchgeführt
Virtueller Bereich 640x576 kleiner als die sichtbare Auflösung 720x576	OK	nicht durchgeführt	nicht durchgeführt
Überschreitung der Maximalhöhe des sichtbaren Auflösung 720x768	OK	nicht durchgeführt	nicht durchgeführt
Virtueller Bereich 720x575 kein Teiler von 4	OK	nicht durchgeführt	nicht durchgeführt
Virtueller Bereich kleiner als die sichtbare Auflösung 720x768	OK	nicht durchgeführt	nicht durchgeführt
Kleinstmögliche Auflösung 144x80, RGB24	OK	OK	OK
Standardauflösung 320x240, RGB24	OK	OK	OK
Falsche Farbtiefe übergeben 320x240, RGB16	OK	nicht durchgeführt	nicht durchgeführt

Testbeschreibung	**Ergebnis**	**Sichtprüfung**	
		Farbtest	**Auflösung**
Nicht darstellbares Farbformat übergeben übergeben 320x240, BGR24. Anmerkung: Farbformat sollte korrigiert werden	OK	Farbformat nicht BGR	OK
Offset Übergeben 40x40 bei 320x240, offset 40x40	OK	OK	OK
Offset nicht durch 4 teilbar, offset 50x50	OK	nicht durchgeführt	nicht durchgeführt
Nicht darstellbare Auflösung verwendet 364x288	OK	nicht durchgeführt	nicht durchgeführt
Nicht darstellbare Auflösung verwendet 320x576	OK	nicht durchgeführt	nicht durchgeführt

C.2 Video4Linux2 Test

Testbeschreibung	Ergebnis	Sichtprüfung
Setzen des Eingabekanals 0	OK	OK
Read(YUV) 360x288	OK	OK
Read (RGB24) 720x756	OK	OK
Mmap (RGB24) 360x288	OK	OK
Mmap (RGB24) 720x576	OK	OK
Overlay (RGB24) 320x288	OK	OK
Test mit 2 Eingabekanälen xvtmmap -two -loop	OK	OK
Setzen des Eingabekanals 2 (vctrl -i 3)	OK	OK
Setzen des Eingabekanals 3 (vctrl -i 3)	OK	OK
Setzen eines nicht definierten Farbformates (vctrl -d 15)	OK	nicht durchgeführt
Setzen einer nicht definierten Bildgröße 640x480	OK	nicht durchgeführt
Auslesen der Controls (vctrl -l)	OK	nicht durchgeführt
Setzen der Helligkeit (vctrl -b 255)	OK	OK
Setzen der Helligkeit (vctrl -b 0)	OK	OK
Setzen des Kontrasts (vctrl -c 0)	OK	OK
Setzen des Kontrasts (vctrl -c 128)	OK	OK
Setzen der Farbe (vctrl -h 0)	OK	OK
Setzen der Farbe (vctrl -h 64)	OK	OK
Setzen der Sättigung (vctrl -s 0)	OK	OK
Setzen der Sättigung (vctrl -s 64)	OK	OK

C.3 Wellenformsimulation

Seite austauschen

C.4 Langzeittests

Seite austauschen

Seite austauschen

Seite austauschen

Seite austauschen

Anhang D

CD-Inhalt

Verzeichnis	**Inhalt**
driver-src	Treiberquelltexte
fgpa-src	Hardwaresourcen der FPGA-Programmierung
testing	Testprogramme die auf dem Lartboard lauffähig sind. Den Programmen liegt der Sourcecode bei.
tools	Beispielprogramme für die Verwendung des LartVIO. Den Programmen liegt der Sourcecode bei.
literatur	Literatur aus dem Literaturverzeichnis, sofern Sie aus dem Internet geladen wurde.
fpga-test	Wellenformsimulationen der FPGA-Module

Literaturverzeichnis

[Alt01] Altera Coorperation. *ACEX 1K Programmable Logic Device Family*, 2001. http://www.altera.com/literature/ds/acex.pdf.

[Ana01] Analog Devices Inc. *Digital PAL/NTSC Video Encoder with 10-Bit SSAF ™and Advanced Power Management ADV7170/ADV7171*, 2001. http://www.analogdevices.com/productSelection/pdf/ADV7170_1_a.pdf.

[ARM01] ARM Limited. *ARM Developer Suite Version 1.2, Assembler Guide*, 2001. http://www.arm.com/support/574FKU/$FILE/ADS_AssemblerGuide_B.pdf.

[BBD+01] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Claus Schröter, and Dirk Verworner. *Linux Kernelprogrammierung, Algorithmen und Strukturen der Version 2.4*. Addison Weslay Verlag, 2001.

[BH02] Ingo Boersch and Prof. Dr. Joachim Heinsohn. Projekt R-Cube - Initiative Ias - Intelligente Autonome Systeme. Technical report, Fachhochschule Brandenburg, 2002. http://ots.fh-brandenburg.de/mod.php?mod=userpage&menu=1301&page_id=6.

[BHK96] Klaus Beuth, Richard Hanebuth, and Günter Kurz. *Nachrichtentechnik - Elektronik 7*. Vogel Verlag und Druck GmbH, Würzburg, 1996.

[Dir02] Bill Dirks. Video for linux two, 2002. http://www.thedirks.org/v4l2/.

[Hüb00] Kai Hübner. Techniken der farbsegmentierung. Technical report, Technischen Fakultät der Universität Bielefeld, 2000. http://www.kaihuebner.de/RoboCup/main.htm.

[Int98] Intel. *Intel® StrongARM® SA-1100 Multimedia Development Board with Companion SA-1101 Development Board*, 1998. http://developer.intel.com.

[Int99] Intel. *Intel® StrongARM® SA-1100 Microprocessor Developer's Manual*, 1999. http://www.lart.tudelft.nl/278088.pdf.

[Int00a] Intel. *Intel® StrongARM® SA-1100 Microprocessor Specification Update*, 2000. http://www.lart.tudelft.nl/27810525.pdf.

[Int00b] Intel. *Memory to Memory Transfer using the SA-1100 DMA*, 2000. http://www.intel.com/design/strong/applnots/mem2mem.pdf.

[Jac96] Keith Jack. *Video Demystified - A Handbook for the Digital Engeneer*. Harris Semiconductors, Virginia, second edition edition, 1996.

[Ker02] Kernelnewbies.org, 2002. http://www.kernelnewbies.org.

[Kno02] Gerd Knorr. video4linux hq, 2002. http://bytesex.org/v4l/.

[KS02] Guido Kuhlmann and Frank Schwanz. *LART-Video Hardware & Programmable Logic Manual*. Tigris Elektronik GmbH, Berlin, 2002.

[Lar00] *The schematics for revision 4 of the LART main board*, 2000. http://www.lart.tudelft.nl/lartware/plint/Lart-rev-4.pdf.

[LARak] *LARTware - Mainboard*, 2000 . http://www.lart.tudelft.nl/lartware/plint/.

[Mar99] Kevin Van Maren. The fluke device driver framework. master's thesis. Master's thesis, Univerity of Utha, 1999.

[NEC97] NEC. *Datasheet MOS integrated circuit μPD42S65165, 4265165*, 1997. http://www.eecg.toronto.edu/ lemieux/z50/NEC-D4265165.pdf.

[Oet97] Frank Oettel. Der RAM - Speicher. Technical report, TU-Chemnitz, 1997. http://www.tu-chemnitz.de/informatik/RA/kompendium/vortraege_97/ram/index.html.

[Phi99] Philips Semiconductors. *Datasheet SAA 7113 - 9 Bit video input processor*, 1999. http://www.semiconductors.philips.com/acrobat/datasheets/SAA7113H_1.pdf.

[Phi01] Philips Semiconductors. *The I^2C-Bus Spezification Version 2.1*, 2001. http://www.semiconductors.philips.com/acrobat/various/I2C_BUS_SPECIFICATION_3.pdf.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly & Associates Inc., second edition edition, 2001.

[Sch02] Frank Schwanz. *LartVIO Softwareinstallation und Programmierung*. Tigris Elektronik GmbH, Berlin, 2002.

[Sik01] Axel Sikora. *Programmierbare Logikbauelemente - Architekturen und Anwendungen*. Carl Hanser Verlag, München, 2001.

[Sim01] James Simmons. *Linux Framebuffer Driver Writing HOWTO*, 2001. http://linuxconsole.sourceforge.net/fbdev/HOWTO/.

[Sim02a] James Simmons. First new fbdev driver. Mailingliste: linuxconsole-dev, 2002. http://sourceforge.net/mailarchive/forum.php?thread_id=262694&forum_id=5379.

[Sim02b] James Simmons, editor. *Reconstruction of the TTY layer for linux to deal with Embedded techology*. Ottowa Linux Symposium, 2002. http://linuxconsole.sourceforge.net/paper/.

[Sto95] Dieter Stotz. *Computergestützte Audio- und Videotechnik - Multimediatechnik in der Anwendung*. Springer Verlag, Berlin, 1995.

[Sty02] Linux Kernel Coding Style. Kernel documentations, 2002. http://www.purists.org/linux/.

[T+02] Linus Torwald et al. Kernel documentations, 2002. http://www.kernel.org.

[Urb99] Peter Urbanek. *Mikrocomputertechnik*. B.G.Teubner Stuttgart, Leipzig, 1999.

[Wan98] Markus Wannemacher. *Das FPGA-Kochbuch*. International Thomson Publishing GmbH, Bonn, 1998.

[Zoo01] Wiebe Zoon. *Console programming HOWTO*, 2001. http://ibiblio.org/gferg/ldp/Console-Programming-HOWTO.