Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Departement Elektrotechnik
Institut für Elektronik
Prof. Dr. Gerhard Tröster

WEBCAMERA

Axel Burri und David Reist

Semesterarbeit WS 00/01

Betreuung: Thomas Schwere

9. Februar 2001

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Zusammenfassung

Im Rahmen dieses Projektes wurde eine Schwarzweiss-Kamera an einen Mikroprozessor mit integriertem Netzwerk-Kontroller angebunden. Als Grundlage diente ein kommerzielles Development Board mit RAM, ROM und Ethernet-Anschluss. Als Betriebssystem läuft Linux, welches sämtliche Funktionen eines vollständigen Embedded Webservers wahrnimmt. Die von uns geschriebenen Software-Treiber und einige Tools erlauben es, die Kamera zu konfigurieren und maximal 50 Bilder pro Sekunde mit minimaler Belastung aller Ressourcen über ein 100Mbit Ethernet zu versenden.

Dieser Bericht zeigt den Weg auf von der Erstellung eines Pflichtenheftes über die Webserver-Evaluation zum Systemdesign bis hin zur funktionierenden Hard- und Software.

Kapitel 2

Einleitung

2.1 Motivation

Die Firma Supercomputing Systems AG (SCS) ist daran interessiert, von ihr entwickelte Geräte in kurzer Zeit mit minimalem zeitlichem und finanziellem Aufwand an Computernetze anzubinden. Gründe hierfür sind beispielsweise eine einfache Fernverwaltung, Datenaustausch mit dem Gerät und eine einfache Möglichkeit, die Systemsoftware zu aktualisieren. Dazu sollen in dieser Arbeit einige heute verfügbare Lösungen aufgezeigt und verglichen werden. Als Anwendungsbeispiel wurde eine Webkamera vorgegeben.

2.2 Kurzbeschreibung

Wir entwickelten eine Anbindung einer CMOS-Kamera über das standard ATA-Interface an das Axis Development Board für den ETRAX-100 Prozessor[1]. Auf dem Prozessor läuft die redimensionierte und auf eingebettete Prozessoren zugeschnittene Linux-Variante uClinux[4]. Der Embedded Webserver kann Bilder der Kamera auf ein LAN (Ethernet) geben, darüber fernverwaltet werden und macht einen zusätzlichen Webserver überflüssig. Die weit verbreitete Infrastruktur des LAN, WAN und des Internets kann so direkt mit der Webkamera verbunden werden, wodurch sie weltweit ansprechbar wird.

Nicht zuletzt sind für eine schnelle Integration einer Netzwerk-Anbindung in kommerzielle Projekte auch die Erfahrungen von Bedeutung, die während unserem Kameraprojekt gesammelt werden konnten.

Kapitel 3

Pflichtenheft

3.1 Übersicht

Ziel des Projektes ist der Entwurf und Aufbau einer Miniaturkamera, bestehend aus folgenden Komponenten:

- CMOS-Bildsensor
- Prozessor mit Ethernet-Anschluss
- Speicher
- Betriebssystem
- Dienste und Applikationen

3.2 Allgemeine Anforderungen

- Klein
- Kostengünstig
- Geringe Leistungsaufnahme
- Geringer Programmieraufwand (Device-Treiber, Netzwerk-Protokolle, Standard-Dienste)
- Einfache Integrationsmöglichkeit der Netzwerkanbindung
- Flexibles Betriebssystem (Dienste, Device-Treiber)
- Performance-Reserven des Prozessors für weitere Dienste und Applikationen.

3.3 Funktionsweise

Einzelbilder werden vom CMOS-Bildsensor zum Prozessor übertragen. Applikationen und Dienste auf dem Betriebssystem nehmen die Funktion eines Embedded Webservers wahr. Die Bilder können via LAN oder Internet (FTP, HTTP) auf eine Workstation heruntergeladen werden oder mittels Client-Applikationen aufbereitet und weiter verarbeitet werden.

Der Embedded Webserver auf dem Kameramodul wird übers Ethernet fernverwaltet.

3.4 Hardware

CMOS-Bildsensor

- Auflösung: 385 x 288 Pixel
- Graustufen: 256 (8 bit)
- Bildrate: 0.5 .. 50 fps

Prozessor mit Ethernet-Port

- Einfache und kostengünstige Anbindung ans Ethernet (100Mbit/s, RJ-45)
- Genügend Performance zur Verarbeitung und Bereitstellung der Bilder auf integriertem Webserver
- Zur Entlastung des Prozessors kann nötigenfalls auch ein zusätzliches FPGA oder ein DSP eingesetzt werden

Memory

- RAM
- Nichtflüchtiger Speicher für Betriebssystem und Applikationen

3.5 Software

3.5.1 Betriebssystem

- Embedded Linux oder vergleichbar
- Flexibel: Einfach anpassbar an neue Erfordernisse (Dienste)
- Genügender Funktionsumfang zum Verwalten, Konfigurieren und Aktualisieren des Webservers übers Ethernet
- Zuverlässig im Dauereinsatz

3.5.2 Dienste und Applikationen

- Aufgabe: Bereitstellung der Bilder im Ethernet, Verwaltung des Webservers
- Protokolle: TCP/IP, HTTP, FTP
- Dienste: Telnet, FTP, Server- und Clientapplikationen

3.6 Ziele und Prioritäten

Die Webcamera soll mittels Aufbau der Hardware und Einsatz der benötigten Software realisiert werden. Ein Prototyp auf einer fertigen Platine (PCB) muss nicht erstellt werden, wäre aber wünschenswert. Falls genügend Zeit zur Verfügung steht, können zusätzliche, noch genauer zu spezifizierende Client- und Serveranwendungen erstellt und getestet werden.

Kapitel 4

Zeitplan

Der anfangs aufgestellte Zeitplan wurde nach der sechsten Semesterwoche über den Haufen geworfen, da sich die erste Lösung nicht bewährte. Deshalb konzipierten wir den Zeitplan neu und mussten alle Phasen verschieben und komprimieren. Für die zweite Lösung war also schon von Beginn an klar, dass der Wochenarbeitsaufwand gross und das Erreichen der Ziele des Pflichtenheftes gerade im Bereich Software kritisch wurde. Für die schriftliche Dokumentation und die Vorbereitung des Schlussvortrages wurde es dann zeitlich auch sehr knapp.

4.1 Arbeitsteilung

Die Evaluation und das System-Design wurde grundsätzlich gemeinsam gemacht. Für die Implementierung teilten wir uns dann auf: Axel Burri widmete sich eher der Software, David Reist eher der Hardware, mit gegenseitiger Absprache. Wir versuchten so weit als möglich parallel zu arbeiten, was nicht immer problemlos möglich ist, z.B. wenn ein Software-Treiber für einen Interface-spezifischen DMA-Transfer gar nicht ohne funktionierende Hardware getestet werden kann.

David Reist zeichnete die Blockschaltbilder, schlug sich mit der Verbindung unterschiedlicher Interfaces herum und zeichnete Timingdiagramme und die elektronischen Schaltungen. Auch die Auswahl, Suche und Bestellung der benötigten Bauteile, den Aufbau einer Prototypenschaltung, die messtechnische Verifikation deren Funktionstüchtigkeit und schliesslich auch deren Optimierung oblag dem Hardware-Verantwortlichen.

Axel Burri arbeitete sich in die Kunst des Schreibens eines Linux-Treibers und dessen Integration in den Kernel ein. Er schrieb den Treiber und die Funktionen für den Kamerazugriff. Weiter programmierte er einen Server-Deamon und einen Java-Client, die zusammen die Bildübertragung übers Ethernet gewährleisten. Als weitere Zugabe realisierte er einen Bewegungsmelder.

Projektplan: Webkamera

Tätigkeit	Aufwand	Bearbeiter	Woche WS 2000/2001															
			43	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6
Evaluationsphase																		
Spezifikationen / Pflichtenheft		DR																
Evaluation Prozessor / Betriebssystem		AB, DR																
Designphase Netsilicon																		
Blockschaltbild		DR																
Einarbeitung in Development Kit und Komponentendokumentation		AB, DR																
Interfaces		AB, DR																
Device Driver																		
Designphase Axis																		
Blockschaltbild		DR																
Einarbeitung in Development Kit und Komponentendokumentation		AB, DR																
Interfaces		AB, DR																
Device Driver		AB																
Implementationsphase																		
Bestellung Hardware- Komponenten		DR																
Aufbau Prototyp		DR																
Programmierung Device Driver		AB																
Test- und Abschlussphase																		
Test Hardware		DR																
Test Device Driver		AB																
Test Software		AB																
Allg. Projektarbeiten																		
Projektmanagement		AB, DR																
Präsentations-Vorbereitung		AB, DR																
Dokumentation / Bericht		AB, DR																
Meilensteine:				1	2								3			4		5

1: Evaluation Komponenten abgeschlossen, Development Kit eingetroffen
2: Vorbereitung Kurzpräsentation IfE 2000-11-10 abgeschlossen
3: Design Hard- und Software abgeschlossen
4: Implementierung Hard- und Software abgeschlossen
5: Projekt abgeschlossen (Vorbereitung Präsentation IfE, Dokumentation, Demoschaltung abgeschlossen)

Abbildung 4.1: Projektplan

Gemeinsam wurde das Zusammenspiel zwischen Treiber und Prototypenboard getestet und mittels Messung des Timings der Signale auf dem Prototypenboard die Konfigurations-Parameter für PIO- und DMA-Transfer optimiert.

Da die Hardware gerade mal zwei Wochen vor dem Abgabetermin fertig wurde, wandte sich David Reist dem Bericht zu, während Axel Burri noch weiter an der Software feilte. Die letzte Semesterwoche wurde reserviert zur Vorbereitung des Vortrages und zum Abschluss des Berichtes.

Kapitel 5

Embedded Webserver

5.1 Marktüberblick Hard- und Software

5.1.1 Prozessoren und Kontroller

Als erstes verschafften wir uns einen Überblick über die erhältlichen Prozessoren und Kontroller mit integriertem 10/100Mbit Ethernet-Kontroller und Linux-Betriebssystem. In der Tabelle 5.1 sind einige geeignete Produkte aufgelistet.

Hersteller	Bezeichnung	uP Core	MIPS	DSPs on Die	DSP MIPS each	DSP RAM	FPGA Gates on Die	RAM on Die	Ethernet Controller on Die	(RT)OS
ATMEL	AT75C310	ARM7TDMI 32bit RISC	20	2 OakDSPCores	40	88KB per DSP	–	–	Software Emulation on 1 DSP	special LINUX
Samsung		ARM7TDMI 32bit RISC	(33MHz)						1 channel, 10/100Mbps, MII	
Netsilicon	NET+5&10T, NET+15, NET+40, NET+50	ARM7TDMI 32bit RISC	5, 10, 15, 40, 50, [120-150]	–	–	–	–	Cache	1x 10/100Mbps	NET+Lx, NET+Works for pSOS+, NET+Works for VxWorks
Intel	SA-1110	StrongARM		–	–	–	–		–	
Motorola	MPC850SE	PowerPC	106 (80MHz), 87 (66MHz)	–	–	–	–	–	2x 10Mbps	
IBM	PowerNP NPe405L oder NPe405H	32bit PowerPC 405	282 (200MHz), 375 (266MHz)	–	–	–	–	–	2x oder 4x 10/100Mbps	
Hitachi	SH7615	SH2-DSP (32bit SuperH RISC)	60	1	?	shared		8KB	1 channel, 10/100Mbps, MII	
Intel / AMD	Embedded 386, 486, Pentium	x86		–	–	–	–	–	–	
Axis	ETRAX 100	Axis RISC CRIS	100	–	–	–	–	Cache		eLinux
SiByte	SB-1 CPU	MIPS-64, MIPS-3D	> 2000 (1GHz)	–	–	–	–	–		
SiByte	SB-1250	2x SB-1, MIPS-64, MIPS-3D	4400 (1GHz), 8GFLOPS, 10M packets/s	–	–	–	–		3x 10/100/1000Mbps	NetBSD, Linux, and VxWorks
ATMEL	AT94K	8bit RISC	30	–	–	–	AT40K, 10k-40k	36KB	–	

Tabelle 5.1: Übersicht Prozessoren

Atmel bietet einen ARM-Prozessor mit zwei integrierten DSPs an; auf einem davon kann ein Ethernet-Kontroller softwaremässig emuliert werden.

Aufgrund der schwachen Performance eignet sich dieser Prozessor eher für Anwendungen, in denen fast nur die Ethernet-Anbindung selbst benötigt wird. Als Webserver eignet er sich demnach schlecht.

Samsung verwendet ebenfalls einen ARM-Core, jedoch zusammen mit einem richtigen Ethernet-Controller (MII). Ein geeignetetes Betriebssystem darf man sich dazu selbst suchen. Auch dieser Prozessor ist langsam, und das fehlende Betriebssystem garantiert eine längere Entwicklungszeit.

Netsilicon - Prozessoren bieten eigentlich alles notwendige, wobei die Performance nicht überragend ist.

IBM - Prozessoren sind für diese Anwendung überdimensioniert: 2 oder gar 4 Ethernet-Controller und 282MIPS sind nicht das, was benötigt wird. Auch hier ist das Betriebssystem selbst zu auszusuchen.

Hitachis Prozessor mit DSP leidet ebenfalls an einem fehlenden Betriebssystem, und für den DSP gibt es keine zwingende Verwendung.

Während der Prozessor-Evaluation übers Internet sind wir auf eine sehr gute Lösung leider nicht gestossen: Den ETRAX 100 von Axis. Erst nach dem abgebrochenen Versuch mit einem Netsilicon Development Board hörten wir davon. Der ETRAX 100 vermag durch 100MIPS und eine reiche Auswahl von Interfaces zu überzeugen.

5.1.2 Betriebssysteme

Linux-Betriebssysteme werden in den verschiedensten Variationen für praktisch alle Cores angeboten. Im Hinblick auf eine kurze Entwicklungszeit und minimalen Aufwand wird natürlich eine Distribution bevorzugt, die der Prozessorhersteller optimal angepasst hat. Die zweite Wahl ist ein vom Prozessor-Hersteller empfohlenes Betriebssystem, wobei man sich so sicher zusätzlichen Aufwand einhandelt um Kompatibilitätsprobleme zu beseitigen, vor allem bei speziellen, in den Prozessor integrierten Komponenten. Eine Übersicht zeigt Tabelle 5.2.

5.2 Kandidaten für Webcam

Zur Zeit werden zwei für uns geeignete Webserver Development Kits auf dem Markt angeboten. Von der Funktionalität und Ausstattung her sind beide vergleichbar, wobei sowohl im Preis, wie auch bei der Brauchbarkeit zur Entwicklung grosse Unterschiede bestehen.

Hersteller	Bezeichnung	uP Cores	Ethernet Controller Support	Target Reference Board	Supported Hosts	Version	Development Tools	Bezug
MontaVista Software	Hard Hat Linux for IBM 405GP Embedded Controller	IBM 405GP und andere 405-Varianten, 405CR; STB03xxx; NPe405H und NPe405L;	10/100Mbps	IBM "Walnut" 405GP Evaluation Kit	Red Hat 6.1/6.2 (Lintel host), Yellow Dog 1.2 (Macintosh or other PPC), Solaris 2.7 (SparcStation)	1.2; Linux Kernel 2.4.0–test2	Ja	IBM (Mit Development Kit); MontaVista
MontaVista Software	Hard Hat Linux	Motorola 7xx, 8xx, 7400. 8240, 8260; Intel x86 und kompatible;, StrongArm 110, 1100, 1110; NEC Vr MIPS 41xx, 43xx, 54xx	10/100Mbps	verschiedene	Red Hat 6.1/6.2 (Lintel host), Yellow Dog 1.2 (Macintosh or other PPC), Solaris 2.7 (SparcStation)	1.2	Ja	MontaVista
Lynuxworks	BlueCat Linux	Intel x86; Motorola PowerPC; PowerQUICC; ARM; ARM7 mit MMU; ARM7 SOC; ARM9; StrongARM; Super-H; MIPS R3000, R4000			Red Hat 6.1/6.2; TurboLinux Workstation 6.0	Linux Kernel 2.2.12		LynuxWorks
QNX	QNX Realtime Platform	x86 Intel, AMD, Cyrix; PowerPC 401, 403, 405, 603e, 604e, 740, 750, 7400, 8240, 8260, MPC860, MPC821, MPC823; MIPS R4000, R5000, NEC VR4300/4102/4111, VR5000, R7000, IDT R4700, QED RM5260/5270/5261/5271	Ja		QNX 4; Windows; Solaris; Linux		Ja; mit CodeWarrior IDE	
ATMEL	special LINUX	ARM7TDMI 32bit RISC	Software Emulation on 1 DSP					
Netsilicon	NET+Lx, NET+Works for pSOS+, NET+Works for VxWorks	ARM7TDMI 32bit RISC	Ja					
Precise Software Technologies Inc.	Precise/MQX RTOS	Motorola 68K, 683xx, ColdFire, MCore, PPC 6xx, PPC 5xx, PPC 7xx, MPC 8xx, MPC 82xx, DSP563xx; TI C3x, C4x, C6x, C54x; Intel x86, StrongARM; ARM7; IBM PPC 40x; Analog Devices ADSP2106x; MIPS R3xxx, R4xxx			Windows 95, 98, NT		Ja	
Pacific Software	Fusion X	AMD 186; ARC; ARM7; Hitachi SH1, SH2, SH3, H8; Hyperstone EI-32; Motorola 360EN v.1/v.2, 860 QUICC/T, 850 ADS, Coldfire 5206, Dragonball; Siemens C165, C166, C167, TRI-Core; ST-20, ST-5500; andere in Entwicklung						
Axis	eLinux	ETRAX 100	10/100Mbps	ETRAX 100	Linux		cris compiler	Axis
LINEO	embedix	x86 Jumptec PC104, ST PC104; AMD SC520, SC400; PowerPC 8260, 823, MBX2000, MVME2400, MVME2700; Hitachi SH3, SH4; Dragonball UcSimm; Coldfire eLia MCF5307; MCore MMC2100, 300 Core	10/100Mbps		Windows NT, Windows 2000	Linux Kernel 2.2.16, 2.4	Ja	
ATI	Nucleus	sehr viele						
WindRiver	verschiedene							
US Software	SuperTask!	verschiedene						

Tabelle 5.2: Übersicht Betriebssysteme

5.2.1 Netsilicon Development Kit

Unsere Absicht war es ursprünglich, die Webcam auf dem Netsilicon NET+ARM 40 [2]aufzubauen.

Die Netsilicon-Lösung überzeugte durch ein übersichtliches, aber grosses Development Board mit Anschlussbuchsen für alle verfügbaren Interfaces wie den Systembus, das ENI Interface, je zwei Parallel und Serial Ports.

Sobald man aber einen Gerätetreiber in den Linux-Kernel einbinden will, ist es jedoch fertig mit komfortablem Entwickeln. Denn jede Änderung des Treibers erzwingt ein Neubeschreiben des Flash-ROMs, und sollte der Kernel mal überhaupt nicht mehr booten (was beim ersten Test eines neuen Treibers unvermeidlich ist), geht ohne teures ARM JTAG ICE Modul gar nichts mehr. Ein solches JTAG-Modul konnte Netsilicon leider nicht zur Verfügung stellen und für USD 3'500 eines käuflich zu erwerben war unverhältnismässig. Aus diesen Gründen wurde nach wenigen Wochen die Übung mit dem Netsilicon-Board abgebrochen und nach einer brauchbaren Alternative gesucht.

Trotzdem wird an dieser Stelle etwas näher auf dieses Board eingegeangen.

Spannungsversorgung

Ein mitgeliefertes Netzteil speist das Board mit 5V DC. Für die Speisung der Komponenten werden daraus stabile 3.3V erzeugt. Für die direkte 5V-

Speisung des Kamera-Moduls müssten die 5V des Netzteils sicherlich noch weiter stabilisiert werden oder eine Wandlerschaltung verwendet werden, die aus 3.3V 5V generiert.

Mikroprozessor NET+ARM 40

Der Net+ARM 40 ist ein 32bit ARM7TDMI RISC Prozessor, wird mit 33MHz getaktet und leistet 40MIPS. Ein integrierter 4KB Cache beschleunigt den Speicherzugriff. Der Systembus (32bit Daten-, 28bit Adressbusbreite) unterstützt 8bit, 16bit und 32bit Devices sowohl mit asynchronem wie auch mit synchronem Timing und unterstützt auch Bursting.

Integriert sind die folgenden Schnittstellen: Zwei Serial Ports, 64KB Shared RAM ENI oder vier Parallel Ports mit DMA-Unterstützung und 24Pin GPIO. Der maximale Leistungsverbrauch des 3.3V Prozessors wird mit 750mW angegeben.

RAM

Je nach Bedarf lässt sich das Board mit 8MB DRAM oder 32MB SDRAM bestücken.

Flash ROM

Als nichtflüchtiger Speicher dient ein 1MB, 2MB oder gar 4MB Flash-Speicher von AMD. Darin liegen komprimiert das Betriebssystem und die Applikationen. Ein zusätzliches 8KB EEPROM wird mit dem ARM JTAG ICE Modul zusammen verwendet, um zu booten, wenn der Inhalt des Flash-ROM beschädigt ist.

Ethernet

Der Prozessor verfügt über ein 10/100Mbit/s Media Independent Interface (MII), um eine externe physikalische Netzwerk-Schnittstelle (PHY) anzuschliessen.

Linux

Netsilicons eigene Linux Portierung NET+Lx unterstützt die Protokolle HTTP, FTP, POP, SMTP, DHCP und BOOTP. Weiter stehen der Apache Webserver, Perl-Unterstützung für cgi-Skripts, SMB und NFS zur Verfügung.

Dokumentation

Die Dokumentation zum Development Board ist sehr knapp gehalten. Die verschiedenen Transfermodi des ENI oder Systembusses gehen nicht genügend ins Detail, insbesondere die Timingdiagramme werfen gerade bezüglich

Waitstates Fragen auf, welche die Dokumentation nicht alle beantworten kann.

Preis

Das komplette Development-Kit umfasst Development Board, Software (inkl. Quellcode), JTAG ICE , Handbücher und ein Review der Schaltung durch Ingenieure von Netsilicon. Der Preis steht in keinem Verhältnis zum Nutzen: USD 19'000. Der NET+ARM 40 Prozessor kostet für 1000 Stück etwa USD 33. Als Studenten erhielten wir zu Spezialkonditionen zwar ein Development Board mit Dokumentation und Software, jedoch leider kein JTAG ICE Module.

5.2.2 Axis Development Kit

Der zweite Versuch, die Kamera an einen Embedded Webserver anzubinden, begann mit einem glücklichen Zufall. Für ein anderes SCS Projekt wurde zu dieser Zeit gerade ein Probeexemplar des Axis Development Boards bestellt. Wir evaluierten dessen Tauglichkeit für die Webcam sofort. Dabei kam heraus, dass es mindestens so gut geeignet ist wie die Netsilicon-Lösung.

Das Axis Development Board[1] besitzt keine vollständige Anschlussmöglichkeit an den Systembus. Zur Verfügung stehen dem Entwickler aber verschiedene Schnittstellen: Zwei serielle RS-232 Ports, ein RS-485/RS-422 und zwei Parallel Ports. Die Buchsen der Parallel Ports können auch umkonfiguriert werden (I/O Application Multiplexed Signals) für ATA, 8bit-SCSI, 16bit-SCSI, Shared RAM(-W), zwei weitere serielle Ports oder General Purpose I/O.

Spannungsversorgung

Die Spannungsversorgung des Axis Development Boards wird durch ein 230V AC Netzteil gewährleistet, das 12V DC Ausgangsspannung erzeugt. Das Development Board hat sowohl einen Anschluss für 9-24V DC als auch einen für 9-24V AC. Die AC-Versorgung wird gleichgerichtet und wie die DC-Versorgung durch einen Spannungswandler auf 3.3V geregelt. Sämtliche Komponenten werden mit 3.3V versorgt.

Mikroprozessor ETRAX 100

Der ETRAX 100 32bit RISC Prozessor (AXIS Code Reduced Instruction Set (CRIS) CPU, 100MHz) leistet 100MIPS (200MIPS/W) und ist als "System-on-a-chip" konzipiert. Er beinhaltet DMA-Kontroller mit einem 64Byte FIFO pro Kanal, 16bit/32bit Daten- und 25bit Adressbus für diverse RAM- und ROM-Typen, Ethernet Controller mit 10/100Mbit/s MMI und 10Mbit NS DP8391A kompatiblem Interface, 8KB Cache und DMA-gesteuerte I/O

Ports. Der Adressraum beträgt 2GB. Unterstützt werden Bootstrap Modes über Netzwerk, PROM, serielle oder parallele Ports. Der typische Leistungsverbrauch des im 0.35um-Prozess hergestellten Prozessors liegt bei 510mW (max. 800mW).

Axis verwendet diesen Prozessor in eigenen Produkten wie Webcams, Print Servers, Storage Servers sowie für Bluetooth Access Points. Mit mehr als 1 Million verkauften ETRAX RISC Prozessoren führt Axis im Bereich der Netzwerkanbindung.

RAM

Das Development Board verfügt über 8MB DRAM, was auch für einige selbstentwickelte Software-Applikationen ausreichen dürfte.

Flash ROM

Als nichtflüchtiger Speicher steht ein 2MB grosses Flash ROM zur Verfügung. Darin befindet sich in komprimierter Form sowohl uClinux[4] als auch sämtliche Applikationen.

Ethernet

Der integrierte Ethernet-Port unterstützt 10/100Mbit und verfügt über ein eigenes DMA- gesteuertes FIFO. Bemerkenswert sind die Zero-Copy Protocol Stacks, die CPU und RAM maximal entlasten und gleichzeitig höchst mögliche Netzwerk-Performance bieten. Eine gelbe LED auf dem Development Board zeigt den Netzwerkverkehr an.

Linux

Für den Entwickler erweist sich eine Möglichkeit, den kompilierten Kernel auf dem Endgerät zu testen, als äusserst attraktiv: Das Development Board kann übers Ethernet von der Entwicklungs-Workstation mit dem Kernel versorgt werden, ohne das Flash-ROM zu beschreiben.

Eine der seriellen Schnittstellen dient dem Entwickler als Debug-Schnittstelle.

Dokumentation

Sämtliche schriftlichen Dokumente inkl. eLinux Source-Code, Schaltpläne und Layout des Development Boards sind übers Internet öffentlich zugänglich und werden regelmässig aktualisiert. Die Dokumentation ist teils zwar etwas knapp gehalten, aber leicht verständlich und übersichtlich. Sie macht einen etwas besseren Eindruck als diejenige von Netsilicon.

Preis

Für USD 299 erhält man ein ETRAX 100 Development Board ohne Support, für USD 449 inkl. Support. Als Support bezeichnet Axis die Garantie, innerhalb einer bestimmen Zeit Antwort auf Fragen zu bekommen. Aber auch ohne Support beantworteten Axis-Mitarbeiter von uns gestellte Fragen innerhalb von einem bis zwei Tagen. Der Einzelpreis des Prozessors liegt bei USD 40 (256Pin PBGA-Gehäuse), 1000 Stück kosten etwa USD 30.

Kapitel 6

Kamera-Modul

Der Schwarzweiss-Bildsensor OmniVision OV5017[3] besitzt einen 8bit breiten Daten- und einen 4bit Adressbus. Darüber lassen sich die integrierten Kontroll- und Statusregister lesen und beschreiben. Das Bild wird als serieller Strom einzelner 8bit-Pixel (256 Graustufen) über ein Register ausgegeben. Dies bedingt, dass der Empfänger oder ein FIFO den Datenstrom in Echtzeit verarbeiten kann. Die Kamera passt in der Standardkonfiguration Helligkeit und Kontrast selbständig an die Lichtverhältnisse an, es besteht auch die Möglichkeit manueller Anpassung. Das verwendete Modul wird von einem 14.318MHz Oszillator getaktet. Die weiteren Daten entsprechen dem Pflichtenheft.

Der Sensor benötigt eine 5V-Speisung, während die High-Pegel der Ein- und Ausgangssignale wahlweise auf 3.3V oder auf 5V gelegt werden dürfen. Um das verwendete Prozessor-Interface direkt mit dem Kamera-Modul verbinden zu können, legen wir sie auf 3.3V.

Kapitel 7

Systemdesign

7.1 Anbindung des Kamera-Moduls an den Mikroprozessor

Wir stellen nun einige Möglichkeiten zur Anbindung der Kamera an den Prozessor vor. Jede Variante hat Vor- und Nachteile. Alle Varianten gelten grundsätzlich sowohl für den Axis- wie auch den Netsilicon-Prozessor, falls nichts anders erwähnt ist.

7.1.1 Pufferung des Video-Streams mittels FIFO

Zwischen den Datenbus des Prozessors und der Kamera wird vor dem Datenbus zusätzlich ein FIFO eingefügt. Um die Kamera-Register beim Konfigurieren direkt zu beschreiben und auszulesen, muss dieses mittels Tristate Buffers umgangen werden können. Der Video-Stream wird in das FIFO geschrieben. Mittels DMA werden die Daten dann vom FIFO ins RAM transferiert. Durch den zusätzlichen Einsatz eines Buskonverters (z.B. mittels Multiplexer) können je 4 8bit-Pixel zu einem 32bit-Wort zusammengefasst werden, wodurch der 32bit-Systembus mit einem 4 mal schnelleren Datentransfer weniger lang besetzt wird. Zusätzlich kann auch noch DMA-Bursting verwendet werden, um die Busbelastung minimal zu halten. Dies bedeutet, dass das FIFO jeweils in grösseren Blöcken ausgelesen wird. Ein FIFO kann an allen weiteren Interfaces Verwendung finden.

- Vorteil: Mittels DMA-Bursts wird der Systembus optimal ausgenutzt. Die Daten müssen auch nicht synchron zum Pixeltakt aus dem FIFO ausgelesen werden. Dabei wird das System, insbesondere das Ethernet, nur minimal beeinträchtig. Das Ansprechen des FIFOs geschieht wie bei einem SRAM-Modul.
- Nachteil: Relativ hohe Kosten des zusätzlichen FIFOs, es sei denn, ein FIFO wäre für diesen Zweck bereits in ein geeignetes Prozessor-Interface integriert. Höherer Aufwand für die Logik.

7.1.2 Die direkte Verbindung mit dem System-Bus

Der Daten- und Adressbus des Prozessors wird direkt mit demjenigen der Kamera verbunden. Der Video-Stream wird mittels DMA oder direkt durch den Prozessor ins RAM kopiert. Die Register der Kamera werden wie normale SRAM-Speicherzellen über virtuelle Adressen beschrieben bzw. gelesen. Beachtung muss aber der Synchronisation des Pixeltaktes auf die Busfrequenz geschenkt werden, um die Bildpunkte auslesen zu können.

- Vorteil: Kostengünstige und schnelle Implementation.
- Nachteil: Lange Belegung des System-Busses.

7.1.3 Die direkte Beschreibung von RAM mittels FPGA

Der Video-Stream wird von einem FPGA direkt in ein (dual-ported) SRAM-Modul geschrieben, das danach durch den Prozessor ausgelesen werden kann. Ein Flag wird im Prozessor gesetzt (z.B. GPIO) oder ein Interrupt ausgelöst, wenn ein vollständiges Bild im SRAM vorliegt. Wenn der Prozessor ein Bild braucht, kann er das Flag pollen, resp. auf den Interrupt warten, um das Bild danach auszulesen. Moderne Bildsensoren bieten diese Möglichkeit oft schon von sich aus an.

- Vorteil: Diese Variante lässt die Kamera von aussen als Speicher erscheinen, aus dem das Bild direkt ausgelesen werden kann.
- Nachteil: (Dual-ported) SRAM ist verhältnismässig teuer.

7.1.4 Die General Purpose I/O-Ports des Prozessors

Das ganze Timing wird in Assembler programmiert; beim Netsilicon Prozessor verbraucht alleine das periodische Umschalten eines einzigen GPIO-Pins 10 Clockzyklen. Beim NET+ARM 40 Prozessor ergäbe das bei einer Frequenz von 33MHz höchstens 3.3MHz. Wird ein ganzes Protokoll in Assembler programmiert, stellt dies einen beträchtlichen Aufwand dar, bis das Timing richtig funktioniert. Während des Auslesens des Video Streams durch den Prozessor muss der System Bus reserviert werden, um die Daten ins RAM schreiben zu können.

- Vorteil: Jegliche zusätzliche Hardware entfällt. Falls Fehler auftreten, können sie durch Änderung des Assembler-Codes behoben werden. Der Pixel-Takt des Kamera-Moduls kann auch in seiner Geschwindigkeit verändert werden, um Software und Hardware optimal aufeinander abzustimmen.
- Nachteil: Was normalerweise ein Memory-Controller macht, muss nun mit grossem Aufwand selbst programmiert werden, wobei schliesslich

kein Geschwindigkeits-Vorteil resultiert. Um das kritische Timing zu gewährleisten darf der Prozessor nicht von anderen Aufgaben gestört werden, womit das System während des Auslesens eines Bildes aus der Kamera faktisch blockiert ist.

7.1.5 Das ENI-Interface des Netsilicon NET+ARM 40

Das Embedded Network Interface ist zur Verbindung zweier Prozessoren gedacht. Dank seines 32Byte FIFOs kann es für unsere Anwendung interessant sein.

- Vorteil: Die Daten können mittels DMA-Transfer über ein integriertes FIFO gepuffert direkt ins RAM geschrieben werden.
- Nachteil: ENI-Interface ist proprietär und die Kameraanbindung somit nicht auf andere Prozessoren ohne dieses Interface portierbar.

7.1.6 Das geeignete Netsilicon Interface

Beim NET+ARM 40 stehen uns zwei brauchbare Varianten zur Verfügung: Der Systembus, mit dem die Kamera als virtueller Speicher angesprochen werden kann, oder das ENI Interface. In beiden Fällen kann DMA benützt werden, wobei das ENI Interface die Daten mittels eines FIFOs zwischenspeichert und so die Belegungszeit des Systembusses leicht vermindert. Allerdings wäre die hohe Pixelrate der Kamera bei 50fps mit etwa 7MHz an der obersten Grenze des ENI Interfaces.

Der Vollständigkeit halber haben wir im Anhang (Kap. D.4.1) einige Skizzen und Entwürfe für die Anbindung der Kamera direkt an den Systembus beigefügt. Sie stellen keine fertige Lösung dar, sondern geben den Projektstand wieder, als wir unser Development Board wegen Untauglichkeit aufgeben mussten.

7.1.7 Das geeignete Axis Interface

Benötigt wird ein Interface mit 4bit Adress- und 8bit Datenbus, welches DMA beherrscht und Interrupt Requests der Kamera behandelt. SCSI und ATA erfüllen diese Bedingungen, wohingegen das Shared RAM Interface an den fehlenden Request-Möglichkeiten (IRQ, DMA Handshaking) scheitert. General Purpose I/O stellt keine erstrebenswerte Lösung dar, da das ganze Protokoll in Software gehandhabt werden muss und ein Datentransfer unabhängig des Prozessors nicht möglich ist.

Nach eingehenden Abklärungen entschieden wir uns für die ATA-Schnittstelle, da das ATA-Protokoll sehr einfach ist und für unseren Zweck völlig ausreicht.

7.2 Das AT Attachment Interface (ATA)

Das AT Attachment Interface, häufig auch als (E)IDE-Schnittstelle bezeichnet, wird heute vor allem in PC-Workstations verwendet, um Festplatten und CD-ROM-Laufwerke kostengünstig anzuschliessen. Es zeichnet sich aus durch eine weite Variationsmöglichkeit der Geschwindigkeit individuell für jedes angeschlossene Gerät sowie durch ein einfaches Bus-Protokoll. Hardwareseitig ist das ATA Interface bereits vollständig in den ETRAX-100 integriert.

Von Vorteil für unsere Anwendung ist, dass ATA zwei verschiedene Betriebsmodi anbietet. Im PIO-Modus können die Kamera-Register direkt adressiert, beschrieben und gelesen werden. Im DMA-Modus strömen die Pixel des Bildes, gepuffert über ein kleines in den ETRAX-100 integriertes FIFO, unter Umgehung der CPU direkt ins RAM. Im Anhang (Kap. D.3.3) finden sich die wichtigsten ATA-Spezifikationen und Protokolle.

7.3 Das ATA/Camera Bus Interface (ACBI)

Um die CMOS-Kamera an den gewählten ATA Bus des Axis ETRAX Prozessors anzuschliessen, wird eine elektronische Schaltung benötigt, welche die Kontroll-Logik, Tristate Buffers und einen Datenpuffer enthält. Diese Schaltung nennen wir ATA/Camera Bus Interface (ACBI).

7.3.1 Datenflussdiagramm der Bildübertragung

Wir betrachten nun prinzipiell, wie die Bilddaten vom CMOS-Sensor ins RAM und schliesslich aufs Netzwerk und Internet gebracht werden (siehe Abb. 7.1). Eine genaue Beschreibung der Protokolle folgt weiter unten.

Ein Bild verlässt den Image Sensor pixelweise über den Camera Bus. Der Bus Converter gewährleistet, dass die Daten auf den ATA Bus gelangen. Von hier werden sie im ATA DMA Mode durch den ATA Controller in ein im Prozessor integriertes FIFO geschrieben. Dies wiederholt sich Pixel für Pixel, während der DMA Controller selbständig die Pixel vom FIFO asynchron zum ATA Bus fortlaufend ins RAM schreibt. Wird das Bild übers Netzwerk verschickt, muss das Bild durch den Netzwerkstack verpackt werden. TCP-, IP- und Netzwerkheader werden von der CPU hinzugefügt und für den Transport geeignete Päckchen generiert. Diese werden dann wieder vom DMA Controller in ein Netzwerk-FIFO geschrieben. Der Netzwerk-Kontroller (in unserem Fall Ethernet) gibt sie weiter an die wählbare physikalische Netzwerkschicht, welche die Pakete je nach Bedarf mit 10Mbit/s oder 100Mbit/s versendet. Auffallend ist, dass mit diesem Prinzip der Mikroprozessor weitestgehend vom DMA-Controller entlastet wird und der von der Kamera generierte Pixelstrom asynchron zum Systembus und zur CPU laufen kann.

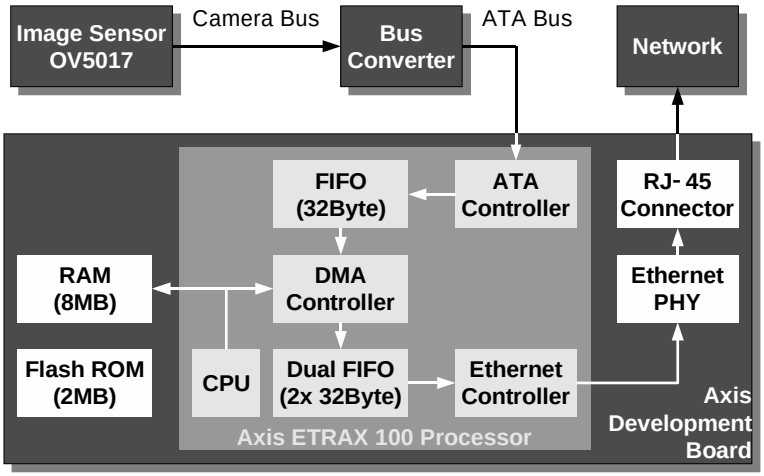

Abbildung 7.1: Datenflussdiagramm ACBI

7.3.2 Der ATA/Camera Bus Converter

Zwischen ATA Controller und Image Sensor liegen verschiedenartige Busse, der ATA Bus und der Camera Bus. Um diese zusammenzuschliessen, muss der Datenbus richtungsgesteuert durchgeschaltet werden können. Um Timing-Bedingungen beim Pixeltransfer im ATA DMA Mode zu erfüllen können die Daten zusätzlich mittels eines Registers zwischengepuffert werden. Durch Logik müssen die 4bit Adressen und Read/Write Kontroll-Signale des Image Sensors aus den 3bit Adressen des ATA Controllers sowie aus den ATA Read/Write Control-Signalen generiert werden.

Für die Bildübertragung sind weiter Synchronisationssignale für den ATA DMA-Transfer wichtig. Um diesen zu initiieren, benötigt der Prozessor vor Beginn jedes Bildes einen Interrupt, welcher vom Vertical Sync Signal erzeugt wird. Der Horizontal Sync steuert dann zusammen mit dem Pixeltakt den ATA Controller, damit dieser immer dann ein Pixel liest, wenn eines gültig auf dem Datenbus anliegt. (siehe Abb. 7.2)

7.3.3 Das Timing

Für die beiden Betriebsmodi PIO und DMA wurden getrennte Timing-Diagramme erstellt. Den Wert für DMA_STROBE mussten wir experimentell optimieren, da der Zeitpunkt, wann die DMA liest, nirgends festgelegt ist und nebenbei auch noch leicht über alle Pixel variiert.

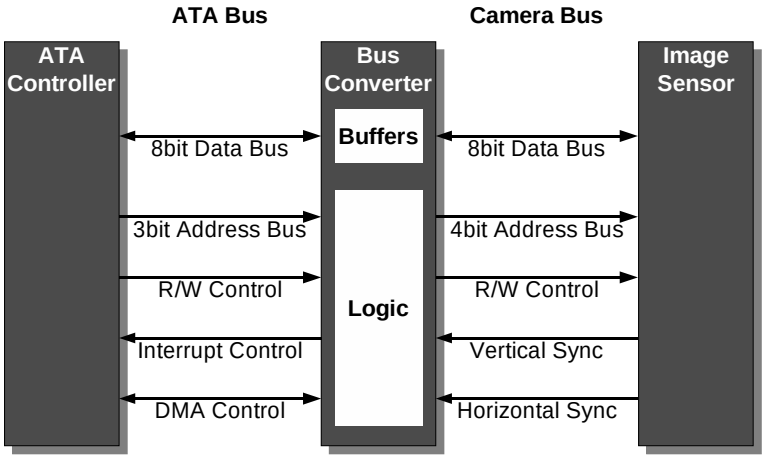

Abbildung 7.2: ATA/Camera Bus Converter Diagramm

7.3.4 Die elektronische Schaltung

Das ACBI wird mit der Komponentenseite nach unten auf den 2x26poligen Stecker des Development Boards aufgesteckt. Am anderen Ende des ACBI lässt sich das Kameramodul mit Objektiv Richtung Verkabelungsseite des ACBI einstecken. Über einen Stiftstecker werden die zusätzlich benötigten Signale DIOR2- und INTRQ2 angeschlossen. Wichtig ist insbesondere, dass die Stecker keinesfalls verschoben oder verdreht eingesteckt werden, da sonst Schäden an der Hardware möglich sind.

Das ACBI enthält alle benötigten Module, um das Kamera-Modul an den ATA-Bus anzuschliessen.

Modul Spannungsversorgung

Um aus den verfügbaren 3.3V die von der Kamera benötigten 5V zu erhalten, wurde eine bewährte Wandlerschaltung eingesetzt. Der MAXIM MAX770 regelt zusammen mit einem N-Kanal MOSFET als Treiber die 5V Ausgangsspannung stabil.

Modul Logik

Benötigt werden zwei AND-ICs (74AC08), ein NOT-IC (74AC04), ein D-Flipflop-IC (74AC74), zwei 8bit Tristate Buffers (74AC244) und ein 8bit Register (74AC374).

Prototyp

Um während der Probe- und Testphase flexibel zu sein wurde eine handverkabelte Schaltung auf einer Prototypenplatine erstellt. Dies erwies sich während den Tests als sehr vorteilhaft, konnten doch schnell Anpassungen vorgenommen werden, um etwa das DMA Timing zu verbessern, bereits auf dem Development Board invertierte Signale wiederherzustellen oder einige Ideen schnell auszuprobieren. Alle ICs sind gesockelt, um sie bei Bedarf schnell wechseln zu können, und besitzen je einen Stabilisierungs-Kondensator für die Versorgungsspannung.

Anstelle des 74AC374 kann auch ein auf einer kleinen Adapter-Platine montierter 74AC244 (ATA/Camera Bus Interface, Schaltung 2 Kap. D.1.4) eingesetzt werden, um 50fps zu erreichen. Den 74AC374 (ATA/Camera Bus Interface, Schaltung 1) war ursprünglich dazu gedacht, die Pixel zu speichern bis der zeitlich nicht festgelegte DMA Read erfolgt. Es erwies sich jedoch in der Praxis, dass der DMA Read bei 50fps die Pixel problemlos direkt abholen kann. Die sichere Methode mit dem 74AC374 lässt aber wegen zu geringer Flankensteilheit von PCLK nur 25fps zu, da das Register nur während einer halben PCLK-Taktperiode ausgelesen werden kann, bevor die nächste DMA-Übertragung initiiert wird. In der Folge beziehen wir uns grundsätzlich auf Schaltung 2.

Aussichten

Als weitere Idee (ATA/Camera Bus Interface, Schaltung 3 Kap. D.1.4)) schlagen wir vor, nur einen Chip Select (CS0-) zur Adressgenerierung zu verwenden und die Bus Tristate Buffers direkt mit den Lese- (DIOR2-) und Schreibsignalen (DIOW2-) anzusteuern. Diese Variante braucht am wenigsten Logik und nur zwei Bus Tristate Buffers (oder einen richtungsgesteuerten bidirektionalen Transceiver). Für Schaltung 3 wurde bis anhin weder ein detailliertes Timing-Diagramm erstellt noch eine Probeschaltung getestet, und es kann somit nicht mit Bestimmtheit gesagt werden, ob das Timing praktisch erfüllt wird.

7.3.5 Notwendige Änderungen am Axis Development Board

Da das Axis Development Board grundsätzlich für zwei Parallel Ports ausgelegt ist, mussten einige kleinere Änderungen vorgenommen werden, um die Signale in der ATA-Konfiguration problemlos zu benutzen. Es sind dies folgende Eingriffe:

- Widerstände R75 und R115 (Sheet 10) überbrückt: Die Spannungsteiler über R52/R75 und R110/R115 verunmöglichten die Benutzung von DMARQ2 und DMACK2-.
- Brücken S18B und S22B entfernt: Abkopplung des RS-422 Ports.

- Anschluss von DIOR2- an S18B und von INTRQ2 an S22B: Die beiden Signale sind nicht auf die Parallel Port Headers geführt.
- U2 (74LCX245SOT) entfernt und Datenbus durchgeschaltet (Brücken über R92 und R93): U2 verhindert den störungsfreien Datenfluss von und zur Kamera.

Schäden: Trotz grösster Vorsicht ist es nicht ganz einfach SMD-Komponenten mit vielen Anschlüssen zu entfernen. Im Zuge der Demontage von U2 lösten sich wärmebedingt zwei Lötpads vom Board. U2 kann somit nicht ersetzt werden. Die Überbrückung mittels R92 und R93 gelang trotzdem fehlerlos.

7.3.6 Der PIO Modus

Bei einem PIO Write und bei einem PIO Read werden die Register der Kamera direkt einzeln beschrieben, resp. gelesen.

PIO Write

Den Schreibzugriff auf ein Register beginnt der Prozessor mit dem Anlegen der Adresse (A[2..0]), der Daten (D[15:8]) und dem Aktivieren des Chip Select Schreib-Signals (CS1-), welches den 8bit Tristate Buffer (PIOW) im Datenbus in Richtung zur Kamera durchschaltet. Nach der Zeit PIO_SETUP aktiviert der Prozessor auch das Schreibsignal (DIOW2-) und somit das Chip Select (CSB-) und Write Enable (WEB-) der Kamera. Auf die steigende Flanke von DIOW2- nach der Zeit PIO_STROBE übernimmt das Register den Wert vom Datenbus. Der Zyklus endet nach PIO_HOLD.

PIO Read

Zum Lesen eines Registers legt der Prozessor die Adresse (A[2..0]) an und aktiviert mit dem Chip Select Lese-Signal (CS0-) den 8bit Tristate Buffer (PIOR) im Datenbus in Richtung zum Prozessor. Nach der Zeit PIO_SETUP aktiviert der Prozessor mit dem Lese-Signal (DIOR2-) das Chip Select (CSB-) und das Output Enable (OEB-) der Kamera. PIO_STROBE später übernimmt der Prozessor die anliegenden Daten auf die steigende Flanke von DIOR2-. Nach PIO_HOLD endet der Zyklus.

Sind CS0- und CS1- beide inaktiv, wird automatisch die richtige Adresse (10xx) für das Auslesen der Pixeldaten angelegt und der Ausgang des Pixelregisters (DMAREG) aktiviert (Schaltung 1), resp. das Tristate DMAR_0 aktiviert (Schaltung 2).

7.3.7 Der DMA Modus

Die DMA-Übertragung beginnt damit, dass die Kamera ein VSYNC Signal kurzzeitig aktiviert, welches im Prozessor einen Interrupt auslöst. Dieser In-

terrupt signalisiert, dass die Kamera nun ein Bild senden möchte. Die Ausgabe jeder Bildzeile beginnt die Kamera mit dem Aktivieren von HREF. Bei jeder steigenden Flanke des Pixel Clocks (PCLK) löst die Kamera über ein D-Flipflop einen DMA Request (DMARQ2) aus und legt das aktuelle Pixel an den Datenbus. Der DMA-Kontroller antwortet mit dem Aktivieren des DMA Acknowledge (DMACK2-). Dies setzt das D-Flipflop zurück und DMARQ2 wird inaktiv. Der DMA-Kontroller reagiert mit dem Deaktivieren von DMACK2-. Der Ausgang des Bus Tristate Buffers ist während des ganzen DMA-Transfers aktiv. In Schaltung 1 (siehe Kap. D.1.4) übernimmt das Pixelregister (DMAREG) auf die fallende Flanke des PCLK die Pixeldaten der Kamera. Der Ausgang des Registers ist ebenfalls während des ganzen DMA-Transfers aktiv und die gespeicherten Daten liegen am ATA-Bus an. Der DMA-Kontroller hat nun bis zur nächsten fallenden Flanke des PCLK Zeit, die Daten ins ATA-FIFO zu speichern. Dies geschieht wie beim PIO Read mit dem Signal DIOR2-. Im Falle der Schaltung 1 darf das Register sogar noch eine weitere halbe PCLK-Periode länger ausgelesen werden. Am Ende der Zeile wird HREF wieder deaktiviert.

7.4 Bild- und Videoübertragung über Ethernet

Ein Rohbild hat eine Grösse von 385x288Byte=110880Byte. Will man 50fps übertragen ergibt das eine Datenrate von 5'544'000Byte/s. 10BaseT erreicht theoretisch maximal 1'250'000Byte/s, was höchstens etwa 11fps zulässt. Erst mit 100BaseT kann die volle Bildrate ausgenützt werden: Bei theoretisch maximal möglichen 12'500'000Byte/s liegt der Flaschenhals nicht mehr bei der Übertragung, sondern allenfalls noch in der Software (Stacks) und in Latenzen durch die Hardware hindurch. Praktisch liegen die maximalen Übertragungsraten je nach Optimierung der Stacks normalerweise um mindestens 20% tiefer als die theoretischen. Denn TCP, IP und Network Header vergrössern das Bild zusammen mit Header und Trailer des verwendeten Bildformates.

Als Übertragungsprotokoll eignet sich UDP/IP nicht (schnell, aber nicht sicher), denn ein Bild muss immer vollständig sein, damit der Empfänger es auch richtig interpretieren kann. Bei UDP wird das Bild aber in mehrere Pakete aufgeteilt, die verloren gehen können und damit das Bild beschädigen. Wir verwenden TCP/IP (langsamer, aber sicher), was uns die vollständige Übertragung eines Bildes garantiert. Bei schlechter oder stark belasteter Verbindung kann die Bildrate zwar beliebig sinken, ohne jedoch die Qualität des Bildes zu beeinträchtigen.

Als Server dient ein kleiner Linux Deamon (siehe auch Kap. 8.1.5). Der Client kann z.B. ein Java-Applet in einer HTML-Page sein, wie unser Demonstrations-Beispiel zeigt. Mit diesem Prinzip kann die Bildübertragung mit einem universellen Server sowohl im Videomodus mit fast beliebiger Bildrate (maximal

Komponente	**Preis** (US$)
Axis ETRAX 100	30
CMOS-Kamera OV5017	15
8MB DRAM (je nach Anwendung auch kleiner)	10
2MB Flash ROM (je nach Anwendung auch kleiner)	25
RJ-45 Buchse mit integriertem PHY	10
SMD-Komponenten	10
Total	**100**
Zusätzliche Komponenten (ohne Preisangabe)	
Netzteil	
Multilayer PCB	
Gehäuse	

Tabelle 7.1: Kostenübersicht

50fps), als auch im Einzelbildmodus betrieben werden. Im Videomodus hängt die Bildrate sowohl von der Bandbreite (effektive Bildübertragung) wie auch von der Latenzzeit (Bildanforderung, Stacks) zwischen Server und Client ab.

Herrscht auf dem Netzwerk zwischen dem Client und dem Server zusätzlicher Verkehr, beeinträchtigt er die Bildrate. Ideal wäre ein Client/Server-Paar, das dynamisch mit der Netzwerkbelastung die Bildrate anpasst und diese je nach Anwendung nach oben begrenzt, um Netzwerk und Internet nicht unnötig zu belasten.

Messungen haben auf einem unbelasteten 10Base-T Netzwerk eine Bildrate von etwa 5fps ergeben, was ungefähr den Erwartungen entspricht. Auf einem 100Base-T Netzwerk erreicht man eine Bildrate von etwa 20 fps.

7.5 Der Einkaufspreis der notwendigen Hardware

Tabelle 7.1 zeigt eine grobe Kostenabschätzung für die vollständige Webcamera-Hardware (ohne PCB). Als Basis wird eine Stückzahl von 1000 Geräten angenommen.

Kapitel 8

Software

Die Software für das Axis Developement Board basiert auf *uClinux*[4], einem Derivat des Linux Kernel, welches ein schlankes UNIX Betriebssystem für Prozessoren ohne Memory Management Unit (MMU) implementiert und im Quelltext verfügbar ist. Axis[1] stellt ein speziell auf das Developement Board abgestimmte Version von uClinux zusammen mit den *cris compiler tools*, der cross-compiler Suite für den ETRAX Prozessor, frei zur Verfügung. Die nachfolgend beschriebene Sofware wurde ausschliesslich mit diesen Tools entwickelt.

8.1 Kernel Treiber

8.1.1 Anforderungen

Der Treiber soll ein Bild nur dann aus der Kamera auslesen, wenn dies von einem Prozess verlangt wird (Bild lesen, Bild aktualisieren, s.u.). Die Konfiguration sollte, wie es für Linux-Kernel üblich ist, mit `make config` bzw. `make menuconfig` möglich sein. Mit dem Befehl `make` wird dann automatisch der Quellcode mit den gewählten Optionen kompiliert.

Die Schnittstelle (für die system calls) des Treibers stellt folgende Funtionen zur Verfügung:

- Register der Kamera lesen/schreiben:
  Start eines PIO read bzw. write auf dem ATA-Bus.
- Lesen eines Bildes:
  Warten auf VSYNC Signal der Kamera; Start eines DMA read auf dem ATA-Bus; Kopieren der Bilddaten vom Kernel-Space in den User-Space.
- Adresse des Bild-Buffers (für direkten Zugriff) ausgeben
- Bild aktualisieren

8.1.2 ETRAX Konfiguration

Da der ETRAX Prozessor verschiedene Schnittstellen unterstützt, diese aber nicht gleichzeitig aktiv sein dürfen (in unserem Fall der serielle Port 2 und der ATA-Bus), muss er unmittelber nach dem Aufstarten konfiguriert werden. Dies geschieht in der Datei `head.S` (Assembler-Code, Seite 69).

8.1.3 Implementierung

In diesem Abschnitt wird nicht der ganze Code beschrieben, sondern vielmehr ein grober Überblick über dessen Funktionsweise verschafft. Der Code selber ist mit zahlreichen Kommentaren versehen und sollte mit den nachfolgenden Erklärungen zu verstehen sein. Der Gesamte Quellcode '`etrax100vis`' ist ab Seite 47 zu finden.

Als Grundlage für die Treiber-Programmierung wurde das Buch 'Linux Kernel Drivers'[6] verwendet.

Register read/write

Für diese Aufgabe sind die Funktionen `vis_pio_read` bzw. `vis_pio_write` zuständig. Sie initiieren einen 'Single PIO Byte read/write' auf dem ATA-Bus.

DMA Data Transfer

Der Bilddatentransfer wird in der Funktion `vis_dma_read` initiiert. Die Anzahl zu transferierenden Bytes stehen in einem DMA-Descriptor, welcher der Funktion übergeben wird. Diese Funktion terminiert direkt nach dem Initiieren des Transfers, alles weitere übernimmt der DMA-Interrupt-Handler `vis_dma_intr`, welcher nach erfolgreichem DMA-Transfer gestartet wird. (siehe Abb. 8.1)

ATA-Interrupt

Das Timing ist in diesem Treiber von zentraler Bedeutung. Ein DMA-Transfer muss initiiert werden, bevor das erste Pixel auf dem Datenbus anliegt, und nachdem das letzte Pixel des vorhergehenden Bildes anliegt. Genau in diesem Zeitfenster wird vom Kamera-Chip ein VSYNC Signal ausgegeben, welches den ATA-Interrupt `vis_ata_intr` auslöst. Der ATA-Interrupt-Handler entscheidet dann anhand der Request-Queue (siehe unten), an welcher Speicherstelle die Bilddaten geschrieben werden müssen und startet dann einen DMA-Transfer. (siehe Abb. 8.1)

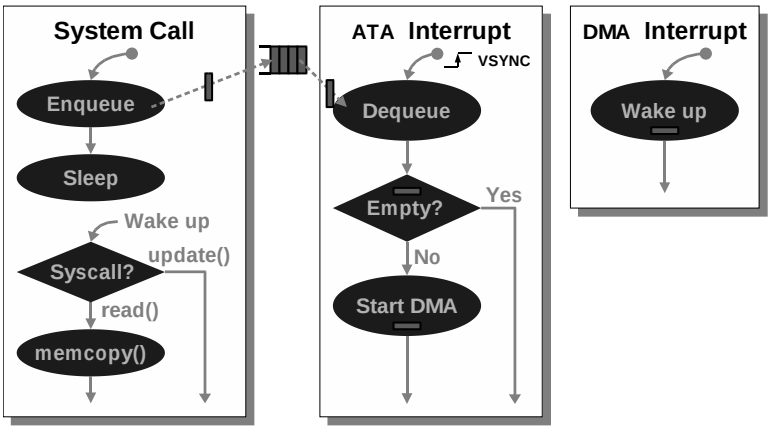

Abbildung 8.1: Funktionsweise des Treibers

Speicher-Management

Vor dem Kompilieren wird in der Konstanten `CONFIG_ETRAX100_OV5017_MAX_READER` festgelegt, wieviele lesende Prozesse gleichzeitig das Device geöffnet haben können. Jeder dieser Prozesse bekommt im Kernel-Space einen Speicherbereich in der Grösse eines Einzelbildes zugewiesen, und er bestimmt selber, wann sein Bild aktualisiert werden soll (z.B. mit einem read() system call).

Die Request-Queue

Jeder Prozess, welcher ein Bild anfordert, wird der Request-Queue angehängt und schlafengelegt. Der vorderste Prozess der Queue bekommt das nächste Bild und wird nach erfolgreichem Datentransfer wieder abgehängt und aufgeweckt. Somit wird verhindert, dass mehrere gleichzeitig lesende Prozesse sich in die Quere kommen. (siehe Abb. 8.1)

read() System Call

Beim Aufruf des Treibers mittels eines read() system calls wird ein Bild im TIFF Format ausgegeben.

ioctl() System Call

Die Funktionen, welche mit dem ioctl() system call aufgerufen werden können, sind der Tabelle 8.1 zu entnehmen.

retval = ioctl(devfd, function, arg);	
Funktion	**Beschreibung**
`VIS_IOC_TEST`	Ruft die Funktion `vis_test` auf. Nur für Testzwecke.
`VIS_IOC_GET_IMAGE`	Kopiert ein Bild (385*288 Pixel) zur Adresse `arg`
`VIS_IOC_GET_BASE_ADR`	Setzt `retval` auf die Adresse des Bildes im Kernel-Space
`VIS_IOC_UPDATE_IMAGE`	Aktualisiert die Bilddaten für den aufrufenden Prozess.
`VIS_IOC_REGISTER_READ`	Liest ein Register des Kamera-Chip (siehe Tab. 8.2)
`VIS_IOC_REGISTER_WRITE`	Beschreibt ein Register des Kamera-Chip (siehe Tab. 8.2)

Tabelle 8.1: ioctl() System Calls

8.1.4 Demo-Applikationen

visreg

Das Programm `visreg` (Code siehe Seite 59) benutzt die ioctl() system calls `VIS_IOC_REGISTER_READ` und `VIS_IOC_REGISTER_WRITE` um die Register des Kamera-Chips zu lesen bzw. zu schreiben.
Aufruf: `visreg r|w reg [value]`

watcher

Das Programm `watcher` (Code siehe Seite 61) ist ein einfacher Bewegungsmelder und demonstriert die Benützung der ioctl() system calls `VIS_IOC_GET_IMAGE`, `VIS_IOC_GET_BASE_ADR` und `VIS_IOC_UPDATE_IMAGE`. Ein Initialbild wird in einer Endlosschleife durch einen einfachen Algorithmus mit dem nächsten verglichen:

1. Helligkeitsanpassung: `new = alpha*new + (1-alpha)*old`
2. Differenzberechnung: `diff` = Differenz `old` zu `new`
3. Falls `diff` grösser als `threshold`: Alarm ausgeben
4. Neues Bild einlesen und weiter mit Punkt 1

Aufruf: `watcher [-r rate] [-t threshold] [-a alpha]`

8.1.5 Image Webserver

Die Programme `visd` und `VisClient` bilden zusammen eine Stream-Applikation für den Einsatz der Webcamera im Internet. (Code siehe Seite 63 bzw. 66)

Register	Zugriff	Beschreibung
VIS_REG_STATUS	R	Status register
VIS_REG_FCTL	W	Single frame flow control system control
VIS_REG_EXCTL	R/W	Auto or manual exposure value
VIS_REG_GCTL	R/W	Gain value
VIS_REG_FRCTL	R/W	Frame rate divider
VIS_REG_MCTL	R/W	Miscellaneous controls
VIS_REG_HWCTL	R/W	Window control
VIS_REG_VWCTL	R/W	Window control
*Achtung:*Ausser den Registern EXCTL, GCTL und MCTL sollten keine anderen beschrieben werden, da dies unweigerlich zu Instabilitäten führt! Einzelheiten entnehmen Sie bitte der OV5017 Dokumentation im Anhang D.2.1, Kap. 2.1		

Tabelle 8.2: OV5017 Register

visd

Dieses Programm stellt den Server dar. Es öffnet lokal einen Port und wartet auf eine Verbindung von aussen. Wurde eine Verbindung geöffnet (z.B. vom `VisClient`) liest er ein neues Bild ein und sendet dieses per TCP-Stream zum anfordernden Prozess.
Aufruf: `visd [-p port] [-b backlog] [-d device] [-q] [-v]`

VisClient

Dies ist die Client-Applikation und ist in Java implementiert, um grösstmögliche Plattformunabhängigkeit zu gewährleisten. Beim Start öffnet sie eine Verbindung zum Port des Servers (`visd`), wartet auf die Ankunft eines Bildes und stellt dieses dar. Nach einer vorgegeben Zeitspanne wird erneut ein Bild angefordert. Aufruf aus HTML-Dokument:[1]

```
<APPLET CODE="VisClient.class" WIDTH="385" HEIGHT="288">
<PARAM NAME="port" VALUE="7777">
<PARAM NAME="delay" VALUE="0">
</APPLET>
```

[1] VisClient.class muss sich in einem Verzeichnis der Webcamera befinden, da es die Sicherheitsvorschriften von Java verlangen, dass sich der geöffnete Port auf demselben Host wie das aufrufende Programm befindet.

Kapitel 9

Projektstand und Ausblick

9.1 Vergleich des Projektstandes mit dem Pflichtenheft

Unsere Lösung erfüllt sämtliche Bedingungen des Pflichtenheftes. Die Zeit reichte trotz dem Zeitverlust mit dem misslungenen Nesilicon-Versuch sogar noch für die Implementierung von Motion Detection und ein Internet Video Transmission Tool, das es ermöglicht, mit einem JAVA-fähigen Web Browser wie etwa Netscape Navigator praktisch ein Echtzeit-Videobild über ein 100Mbit Ethernet zu erhalten.

9.2 Projektstand

Ziel dieser Semesterarbeit war eine funktionierende Webcam (Hardware und Device Driver). Wir bauten einen Prototypen des Interfaces zwischen der Kamera und dem ATA-Bus des Axis Development Boards. Um die Kamera auch ansprechen und steuern zu können, schrieben wir einen Device Driver. Dieser beinhaltet zwei für den Benutzer des Treibers transparente Modi: Im PIO-Modus können die Konfigurations- und Statusregister der CMOS-Kamera beschrieben und ausgelesen werden. Der DMA-Modus erlaubt es, ein ganzes Bild ins RAM zu transferieren. Der Benutzer liest einfach von einem Device oder ruft Funktionen auf, um Kamera-Register zu lesen und zu beschreiben, ohne dass er von den Betriebs-Modi etwas bemerkt. Der Treiber besitzt eine eingebaute Funktion, um das Rohbild mit dem notwendigen Header und Trailer zu versehen und als Datei im TIF-Format auszugeben. Mittels Motion Detection kann einem Empfänger z.B. ein Bild oder eine Meldung gesendet werden, sobald die Kamera eine genügende Veränderung (konfigurierbar) im Sichtbereich feststellt. Das Internet Video Tool bietet schliesslich Echtzeit-Videoübertragung über Netzwerk und Internet.

9.3 Ausblick

In einem Folgeprojekt können weitere Software-Tools zur Steuerung der Kamera sowie zur Verwaltung des eingebetteten Webservers erstellt werden. So könnte z.B. ein Bild ins JPEG-Format konvertiert werden, was eine kürzere Transferzeit übers Ethernet erlaubte. Weiter bietet es sich an, die Konfiguration der Kamera von einer entfernten Workstation aus mittels eines ergonomischen Software-Tools vorzunehmen. Man könnte auch eine Steuerung für Autofokus und Blickrichtung der Kamera implementieren, womit die Kamera dann alle Funktionen besitzt, um vollständig ferngesteuert zu werden.

Ein nächster Schritt wäre sicherlich die Herstellung eines einzigen Prints (PCB) anstelle der Kombination aus Development Board, dem empfindlichen Prototypenboard und dem Kamera-Modul. Löst sich am Prototyp irgendwo ein Kabel, bedarf es einiger Geschicklichkeit und Kenntnis des Prototypenboards, um es zu reparieren. Als kleine Hilfe für die Erstellung des PCB existieren Schaltung 1 bis 3 bereits als Eagle Schema-Dateien (.sch).

Der Mikroprozessor ist leistungsfähig genug, um im Bereich Bildverarbeitung einige nicht allzu komplexe Funktionen auszuführen. Interessant könnte auch die Erweiterung auf zwei Kamera-Module sein, womit sich durch Stereosicht ganz neue Möglichkeiten der Bildverarbeitung ergeben. Auch die Möglichkeiten zur einfachen und schnellen Anbindung eines zusätzlichen DSPs oder FPGAs zu untersuchen, dürfte je nach Einsatzbereich des Webservers interessant sein.

Weiter liesse sich die Eignung des leistungsfähigeren Axis ETRAX 100LX Prozessors für andere Anwendungen untersuchen: Die bald in Stückzahlen erhältliche Nachfolgeversion verfügt zusätzlich über eine integrierte Memory Management Unit (MMU), was die Verwendung eines standard Linux erlaubt. Die LX-Version bietet neu unter anderem auch integrierte USB- und SDRAM-Unterstützung. Die Rückwärtskompatibilität zum ETRAX 100 soll gewährleistet sein.

Kapitel 10

Erfahrungen und Erkenntnisse

10.1 Vergleich Netsilicon - Axis

Grundsätzlich sind beide Prozessoren vergleichbar, obwohl natürlich der Axis-Prozessor mit 100MIPS die einiges leistungsfähigere Variante darstellt. Der Hauptunterschied liegt in der Fähigkeit des ETRAX 100, sich über das Netzwerk Betriebssystem und Applikationen zu holen und direkt damit zu booten. Ein aufwendigeres und riskantes Flashen entfällt damit glücklicherweise. Beim NET+ARM machte bereits das erste Beschreiben des Flashes unser Board wertlos. Da unser erster in den Kernel eingebundener Kamera-Treiber nicht fehlerfrei war und damit Linux nicht erfolgreich starten konnte, konnte man das Flash auch nicht mehr neu beschreiben.

10.2 Eina Kamera am ATA-Bus?

Dass der ATA Bus leistungsfähig ist, beweist uns seine weite Verbreitung im modernen Consumer PC Bereich, seine niedrigen Kosten und seine dauernde Weiterentwicklung. Gerade sein verhältnismässig einfaches Busprotokoll erlaubt es, schnell und mit geringem Aufwand Geräte anzubinden. Im Fall unserer Kamera wird natürlich ein eigener Software-Treiber benötigt, der auf dieses Gerät zugeschnitten ist. Am Axis Development Board ATA Bus können mehrere ATA-Geräte angeschlossen werden. Sollen diese aber gleichzeitig mit unserer Kamera betrieben werden können, bedarf es einer Anpassung des Kamera-Treibers. Veränderungen am Linux ATA-Treiber sollten möglichst vermieden werden. Ideal wäre, wenn der standard ATA-Treiber benutzt werden könnte und die Kamera-spezifischen Teile modular daran angebunden werden könnten. Wie das zu realisieren wäre, wurde nicht betrachtet.

10.3 Eine Semesterarbeit bei SCS

Die Erfahrungen, die man macht, wenn man in einer Firma eine Semester- oder Diplomarbeit macht, sind sehr wertvoll. Im Falle der SCS standen uns eine fast perfekte Infrastruktur und sehr kompetente Mitarbeiter zur Verfügung, die einem mit ihrer Erfahrung auch gerne mal bei einem Problem weiterhalfen. Man knüpft Kontakte, sieht in Projekte hinein und lernt die Probleme im Berufsalltag kennen.

10.4 Persönliche Erfahrungen

Der Misslungene Versuch mit dem Netsilicon Board lehrte uns, dass die Evaluationsphase eines Projektes die wichtigste ist. Übersieht man hier etwas oder klärt man nicht alle noch so kleinen Details genügend ab, verliert man wertvolle Zeit, die später kaum mehr aufgeholt werden kann.

Da wir beide noch nie ein komplettes Systemdesign in der Praxis erarbeitet hatten, mussten wir uns erst einmal mit den Konzepten vertraut machen. Die nächste Hürde war, sich ohne praktische Hardware-Erfahrung mit verschiedenen Interfaces zu befassen. Schliesslich durften wir unsere erste elektronische Schaltung entwerfen und uns überraschen lassen, ob der fertige Prototyp auch so funktionierte, wie wir es uns erhofften. Bei diesem Teil waren wir froh um einige Tips von Daniel Erhardt, der uns z.B. sofort sagen konnte, welche Bauteile-Familie sich für unser Projekt wohl am besten eignet oder wie wir anscheinend zu erwartende Schwingungen vermeiden könnten. Das Zusammenlöten des Prototypen war zwar nicht weiter anspruchsvoll, jedoch etwas zeitaufwendiger als wir es uns gedacht hatten. Auch bei der Porgrammierung eines Linux-Treibers betraten wir Neuland, ohne die Fallen zu kennen die sich in diesem Bereich öffnen würden.

Insgesamt war es ein sehr interessantes Projekt, das nach dem Misserfolg mit dem Netsilicon Board schliesslich innerhalb von acht Wochen doch noch ein gutes Ende gefunden hat! Wir konnten sehr viele Erfahrungen sammeln, sowohl in Eigenverantwortung im Bereich Hardware und Software, als auch im Zweierteam beim Gesamtkonzept oder bei angeregten Diskussionen um Lösungen im Hard- und Softwarebereich. Der Zeitaufwand bewegte sich während der letzten acht Wochen des Semesters insgesamt eher am oberen Limit.

Kapitel 11

Dank

Wir möchten folgenden Personen für ihre wertvolle Hilfe und Unterstützung herzlich danken:

- Thomas Schwere (SCS) betreute uns geduldig und konstruktiv während der ganzen Arbeit. Mit seiner positiven Einstellung vermochte er uns auch dann wieder zu motivieren, wenn wir mit Problemen überhäuft waren.
- Daniel Erhardt (SCS): Bei Hardware-Fragen konnten wir bei ihm Anregungen und Tips erhalten. Brauchte man irgendetwas, was nirgends vorhanden war, hatte er mindestens ein Exemplar davon entweder in seiner berüchtigten Schublade oder zu Hause und leihte es uns grosszügig aus.
- Anton Gunzinger (SCS) stellte das interessante Projekt auf die Beine und gab uns einige Impulse. Er hat auch bereits ein Folgeprojekt ausgeschrieben, um unsere Hard- und Softwarebasis weiter auszubauen. An Ideen mangelt es ihm bestimmt nicht!
- Men Muheim (IfE, D-ELEK, ETHZ) betreute uns seitens der ETH.

Allen Mitarbeitern und Mitarbeiterinnen der SCS danken wir für ihre Hilfe, die sie uns im Zusammenhang mit diesem Projekt leisteten.

Kapitel 12

Rechte

Sämtliche Rechte an den in dieser Arbeit beschriebenen Lösungen und Implementierungen liegen bei den beiden Autoren dieser Arbeit.

Die Rechte für die Schaltpläne der Development Boards liegen bei deren Herstellern.

Literaturverzeichnis

[1] Axis Communications: Produkte-Dokumentation ETRAX 100 Prozessor, Development Board und elinux. Internet-Adresse: `www.axis.com`, `www.developer.axis.com`

[2] Netsilicon: Produkte-Dokumentation NET+ARM 40 Prozessor, Development Board und NET+Lx, Netsilicon Inc. Internet-Adresse: `www.netsilicon.com`

[3] OmniVision: Produkte-Dokumentation OV5017. Internet-Adresse: `www.ovt.com`

[4] uClinux: Internet-Adresse: `www.uclinux.org`

[5] Produkte-Dokumentation zu allen verwendeten Bauteilen via Internet. Internet-Adressen: `www.ti.com`, `www.fairchildsemi.com`

[6] Alessandro Rubini. *Linux Device Drivers*. O'Really & Associates, inc. ISBN 1-56592-292-1

Anhang A

Abkürzungen

Abkürzung	Bedeutung
ATA	AT Attachment Interface
ACBI	ATA/Camera Bus Interface
fps	Frames per second = Bilder pro Sekunde
GPIO	General Purpose I/O
I/O	Input/Output
LAN	Local Area Network
MII	Media Independent Interface
MMU	Memory Management Unit
PIO	Processor Controlled I/O
RISC	Reduced Instruction Set Computer
SCS	Supercomputing Systems AG
uCLinux	Microcontroller Linux
WAN	Wide Area Network

Anhang B

Adressen

Axel Burri
Weitlingweg 63
8038 Zürich
axburri@student.ethz.ch

David Reist
Zidler 13
9057 Weissbad
dreist@ee.ethz.ch

Supercomputing Systems AG
Technoparkstrasse 1
8005 Zürich
www.scs.ch

Anhang C

Source Code

C.1 Device Driver

C.1.1 etrax100vis.h

axis/elinux/drivers/char/etrax100vis.h

```
#include <asm/ioctl.h>


#ifndef VIS_H
#define VIS_H

#define VIS_MAJOR 125  /* drivers major number for local/experimental use */
#define VIS_NAME "ETRAX/100 OV5017 ATA"


/*
 * ioctl commands:
 * ---------------
 *
 * VIS_IOC_GET_IMAGE:
 * argument: pointer to image-buffer (size: VIS_IMAGESIZE)
 *
 * VIS_IOC_GET_BASE_ADR:
 * argument: not used
 * returns : unsigned char *buf[VIS_IMAGESIZE]
 * (WARNING: this is a hack and works only because the ETRAX100
 * does not have a MMU !!!)
 *
 * VIS_IOC_UPDATE_IMAGE:
 * to use with get_base_adr, it just updates the image
 * argument: not used
 *
 * VIS_IOC_REGISTER_WRITE:
 * argument: bit 2..0: register to read (see ov5017 register set below)
 *
 * VIS_IOC_REGISTER_WRITE:
 * argument: bit 19..16: register to set (see ov5017 register set below)
 *           bit   7..0: data
 */
#define VIS_IOC_MAGIC 'o'
#define VIS_IOC_TEST            _IO(VIS_IOC_MAGIC,   0)
#define VIS_IOC_GET_IMAGE       _IO(VIS_IOC_MAGIC,   1)
```

```
#define VIS_IOC_GET_BASE_ADR    _IO(VIS_IOC_MAGIC,   2)
#define VIS_IOC_UPDATE_IMAGE    _IO(VIS_IOC_MAGIC,   3)
#define VIS_IOC_REGISTER_READ   _IO(VIS_IOC_MAGIC,   4)
#define VIS_IOC_REGISTER_WRITE  _IO(VIS_IOC_MAGIC,   5)
#define VIS_IOC_MAXNR 5


/* Image Size (Bytes) */
#define VIS_IMAGE_WIDTH 385
#define VIS_IMAGE_HEIGHT 288
#define VIS_IMAGESIZE VIS_IMAGE_WIDTH * VIS_IMAGE_HEIGHT

/* TIFF Image Size (Bytes) */
#define VIS_TIFF_HEAD_SIZE 8
#define VIS_TIFF_TAIL_SIZE 312
#define VIS_TIFF_IMAGESIZE VIS_TIFF_HEAD_SIZE + VIS_IMAGESIZE + VIS_TIFF_TAIL_SIZE



/*
 * ov5017 register set
 */
#define VIS_REG_STATUS  0  /*  R    */
#define VIS_REG_FCTL    1  /*    W  */
#define VIS_REG_EXCTL   2  /*  R/W  */
#define VIS_REG_GCTL    3  /*  R/W  */
#define VIS_REG_FRCTL   4  /*  R/W  */
#define VIS_REG_MCTL    5  /*  R/W  */
#define VIS_REG_HWCTL   6  /*  R/W  */
#define VIS_REG_VWCTL   7  /*  R/W  */



/*
 * extern procedures
 */
extern int etrax_vis_ov5017_init (void);

#endif /* VIS_H */
```

C.1.2 etrax100vis.c

axis/elinux/drivers/char/etrax100vis.c

```
/*
 * OmniVision ov5017 CMOS-chip driver
 *
 * (c) 2001 Axel Burri <aburri@ee.ethz.ch>
 *
 *
 * Note: "vis" stands for "Video-Image-Sensor"
 *
 * v0.01: - register r/w (using PIO) works
 * v0.02: - dma read works
 * v0.03: - changed data bits from 15..8 to 7..0
 * v0.1 : - multiple access implemented, stable
 * v0.2 : - direct access to memory implemented
 *          (WARNING: this is a hack and works only because the ETRAX100
 *          does not have a MMU !!!)
 *
 * TODO : - real mmap() functions
```

```
 *        - different minor numbers for differnt file formats
 *          (e.g. 0=raw data, 1=tiff, 2=jpeg etc.)
 */

#define VIS_VERSION "0.2"

#define VIS_DEBUG
#undef  VIS_DEBUG
#define VIS_LOWDEBUG
#undef  VIS_LOWDEBUG


#include <linux/errno.h>
#include <linux/malloc.h>

#include <asm/io.h>
#include <asm/irq.h>
#include <asm/svinto.h>

#include "etrax100vis.h"


#define VIS_ATA_IRQ 4    /* (ATA-irq on ETRAX) */
#define VIS_DMA3_IRQ 19  /* (DMA3-irq on ETRAX) */


struct vis_dev {
  int usage;
  void *img_buf;               /* image buffer starts here */
  unsigned char *tiff_img_buf; /* tiff image buffer */
  struct wait_queue *wait_q;
  etrax_dma_descr ata_dma_descr[2];
};
static struct vis_dev vis_devices[CONFIG_ETRAX100_OV5017_MAX_READER];



/*
 * A reader in need of a new image is enqueued to the request-queue.
 * On next ata_intr (VSYNC) the next reader on this queue gets the new image.
 */
struct img_request_queue {
  struct vis_dev *dev;
  struct img_request_queue *next;
};
static struct img_request_queue *img_request_q;



/*
 * state: Current state of the driver.
 * This is used in vis_dma_intr to determine if image is ready or
 * if second part has to be read.
 */
static int state;
#define STATE_IDLE 0
#define STATE_DMA_READ_FIRST 1
#define STATE_DMA_READ_SECOND 2


/*
 * The TIFF header and trailer
```

```
 */
static const unsigned short tiff_head[] =
{18761,   42,   45352,   1};
static const unsigned short tiff_tail[] = {
  17,  254,  4,  1,  0,  0,  0,  256,
  3,  1,  0,  385,  0,  257,  3,  1,
  0,  288,  0,  258,  3,  1,  0,  8,
  0,  259,  3,  1,  0,  1,  0,  262,
  3,  1,  0,  1,  0,  269,  2,  23,
  0,  45562,  1,  270,  2,  22,  0,  45586,
  1,  273,  4,  5,  0,  45608,  1,  274,
  3,  1,  0,  1,  0,  277,  3,  1,
  0,  1,  0,  278,  3,  1,  0,  64,
  0,  279,  4,  5,  0,  45628,  1,  282,
  5,  1,  0,  45648,  1,  283,  5,  1,
  0,  45656,  1,  284,  3,  1,  0,  1,
  0,  296,  3,  1,  0,  2,  0,  0,
  0,  26671,  28015,  12133,  25185,  29743,  29541,  12148,
  25972,  29811,  29742,  26217,  0,  29251,  24933,  25972,
  8292,  26999,  26740,  18464,  28793,  29285,  24899,  109,
  8,  0,  24648,  0,  49288,  0,  8392,  1,
  33032,  1,  24640,  0,  24640,  0,  24640,  0,
  24640,  0,  12320,  0,  0,  18432,  0,  256,
  0,  18432,  0,  256
};


/*
 * enqueue a reader to the request-queue
 */
static void vis_enqueue_reader(struct vis_dev *dev) {
  struct img_request_queue *p;
  struct img_request_queue *q = NULL;
  struct img_request_queue *n;
  unsigned long flags;

  n = (struct img_request_queue *) kmalloc(sizeof(struct img_request_queue), GFP_ATOMIC);
  if(n == NULL) {
    printk("vis_enqueue_reader: kmalloc error\n");
  }
  else {
    n->dev = dev;
    n->next = NULL;

    save_flags(flags); cli(); /* do not disturb! */
    p = img_request_q;
    while(p != NULL) {
      q = p;
      p = p->next;
    }

    if(q == NULL) /* img_request_q == NULL */
      img_request_q = n;
    else
      q->next = p;
    restore_flags(flags);
  }
  /*      *R_IRQ_MASK0_SET = IO_STATE( R_IRQ_MASK0_SET, ata_irq2, set ); */
}
```

```
/*
 * dequeue a reader from the request-queue
 */
static struct img_request_queue * vis_dequeue_reader() {
  struct img_request_queue *q = img_request_q;

  /* since this is used in interrupt handler only, there is no need for cli();sti(); */
  img_request_q = img_request_q->next;
  return q;
}



/*
 * The video data is read in this function (using ATA-DMA)
 */
inline void vis_dma_read(etrax_dma_descr *dma_descr, int count) {
  unsigned long status;

#ifdef VIS_LOWDEBUG
  printk("vis_dma_read: state %i\n", state);
#endif

  /* make sure the DMA channel is available */
  RESET_DMA(3);
  WAIT_DMA(3);
  /*
   * DMA write to buffer (specified in dma_descr)
   */

  /* start the dma channel */
  *R_DMA_CH3_FIRST = (unsigned long)dma_descr;
  *R_DMA_CH3_CMD = IO_STATE(R_DMA_CH3_CMD, cmd, start);

  *R_ATA_TRANSFER_CNT = count;

  /* initiate a multi word dma read using DMA handshaking */
  *R_ATA_CTRL_DATA =
    IO_FIELD(R_ATA_CTRL_DATA, sel,      2        ) | /* ATA bus 2 */
    IO_STATE(R_ATA_CTRL_DATA, cs1,      active   ) |
    IO_STATE(R_ATA_CTRL_DATA, cs0,      active   ) |
    IO_STATE(R_ATA_CTRL_DATA, rw,       read     ) |
    IO_STATE(R_ATA_CTRL_DATA, src_dst,  dma      ) |
    IO_STATE(R_ATA_CTRL_DATA, handsh,   dma      ) |
    IO_STATE(R_ATA_CTRL_DATA, multi,    on       ) |
    IO_STATE(R_ATA_CTRL_DATA, dma_size, byte     );

#ifdef VIS_LOWDEBUG
  printk("vis_dma_read: read complete\n");
#endif

}



/*
 * single byte transfer from ov5017 register
 */
static unsigned char vis_pio_read(unsigned char reg) {
  int status;

  /* make sure the DMA channel is available */
```

```
  RESET_DMA(3);
  WAIT_DMA(3);

  /* invert address (for use on the devboard, it's inverted there) */
  reg = ~reg & 0x7;


  /* wait for busy flag to clear */
  while(*R_ATA_STATUS_DATA & IO_MASK(R_ATA_STATUS_DATA, busy));


  /*
   * initiate PIO single byte read (initiated when register is written)
   */
  *R_ATA_CTRL_DATA = (reg << 25) |
    IO_FIELD(R_ATA_CTRL_DATA, sel,      2        ) | /* ATA bus 2 */
    IO_STATE(R_ATA_CTRL_DATA, cs1,      active   ) |
    IO_STATE(R_ATA_CTRL_DATA, cs0,      inactive ) |
    IO_STATE(R_ATA_CTRL_DATA, rw,       read     ) |
    IO_STATE(R_ATA_CTRL_DATA, src_dst,  register ) | /* output to R_ATA_STATUS_DATA */
    IO_STATE(R_ATA_CTRL_DATA, handsh,   pio      ) |
    IO_STATE(R_ATA_CTRL_DATA, multi,    off      ) |
    IO_STATE(R_ATA_CTRL_DATA, dma_size, byte     );

  /* wait for busy flag to clear */
  while(*R_ATA_STATUS_DATA & IO_MASK(R_ATA_STATUS_DATA, busy));

  /* wait for dav (data available) and read the value from R_ATA_STATUS_DATA */
  while(!((status = *R_ATA_STATUS_DATA) & IO_MASK(R_ATA_STATUS_DATA, dav)));

#ifdef VIS_DEBUG
  printk("vis_pio_read: 0x%x from reg 0x%x\n", status, ~reg & 0x7);
#endif
  return (unsigned char)(status & 0xff) ; /* return bits 0..7 of R_ATA_STATUS_DATA */
}



/*
 * single byte transfer to ov5017 register
 */
static void vis_pio_write(unsigned char reg, unsigned char data) {
  int status;

#ifdef VIS_DEBUG
  printk("vis_pio_write: reg 0x%x, data 0x%x\n", reg & 0x7, data);
#endif

  /* make sure the DMA channel is available */
  RESET_DMA(2);
  WAIT_DMA(2);

  /* invert address (for use on the devboard, it's inverted there) */
  reg = ~reg & 0x7;

  /*
   * initiate PIO single byte read (initiated when register is written)
   */
  *R_ATA_CTRL_DATA = (reg << 25) | (data) |
    IO_FIELD(R_ATA_CTRL_DATA, sel, 2)            | /* ATA bus 2 */
    IO_STATE(R_ATA_CTRL_DATA, cs1, inactive)     | /* cs1 inactive */
    IO_STATE(R_ATA_CTRL_DATA, cs0, active)       | /* cs0 active */
```

```
    IO_STATE(R_ATA_CTRL_DATA, rw, write)         | /* write data */
    IO_STATE(R_ATA_CTRL_DATA, src_dst, register) | /* output to R_ATA_STATUS_DATA */
    IO_STATE(R_ATA_CTRL_DATA, handsh, pio)       | /* PIO mode */
    IO_STATE(R_ATA_CTRL_DATA, multi, off)        | /* stop after this transfer */
    IO_STATE(R_ATA_CTRL_DATA, dma_size, byte);     /* 8 bit transfer */

  /* wait for transmitter ready */
  while(!(*R_ATA_STATUS_DATA & IO_MASK(R_ATA_STATUS_DATA, tr_rdy)));
}



/*
 * The ATA interrupt handler.
 * Called on every VSYNC of the image.
 * Image is updated for the first item in the request-queue .
 */
void vis_ata_intr (int irq, void *dev_id, struct pt_regs *regs) {
#ifdef VIS_LOWDEBUG
  printk("intr\n");
  /*      printk("vis_ata_intr: jiffies %i\n", jiffies); */
#endif
  if (state == STATE_IDLE) {
    if (img_request_q != NULL) {
      /* start reading the image here */
      state = STATE_DMA_READ_FIRST;
      vis_dma_read(&(img_request_q->dev->ata_dma_descr[0]), 65536);
    }
  }
  else {
  }
}



/*
 * Called when a DMA-interrupt occurs.
 * This happens when a DMA transfer is finished and is used:
 * 1) to initiate the second DMA-transfer if state=STATE_DMA_READ_FIRST and
 * 2) to dequeue the reader and wake him up if state=STATE_DMA_READ_SECOND.
 */
void vis_dma_intr (int irq, void *dev_id, struct pt_regs *regs) {
#ifdef VIS_LOWDEBUG
  printk("vis_dma_intr\n");
#endif
  /*      if (*R_IRQ_MASK2_RD & IO_MASK( R_IRQ_MASK2_RD, dma3_eop)) { */
  *R_DMA_CH3_CLR_INTR = IO_STATE( R_DMA_CH3_CLR_INTR, clr_descr, do );
  if(state == STATE_DMA_READ_FIRST) {
    *R_ATA_TRANSFER_CNT = 65536;
    state = STATE_DMA_READ_SECOND;
  }
  else if(state == STATE_DMA_READ_SECOND) {
    /* image ready, wake up waiting process */
    wake_up_interruptible(&(img_request_q->dev->wait_q));
    /* process with next reader in img_request_q (if any)*/
    kfree(vis_dequeue_reader());
    state = STATE_IDLE;
  }
}
```

```
/*
 * general purpose test routine
 */
static int vis_test(unsigned long arg) {
#ifdef VIS_DEBUG
  printk("vis_test called\n");
#endif

#if 0
  *R_ATA_CONFIG = ( IO_FIELD( R_ATA_CONFIG, enable,     1 ) |
    IO_FIELD( R_ATA_CONFIG, dma_strobe,  arg & 0xff) |
    IO_FIELD( R_ATA_CONFIG, dma_hold,    CONFIG_ETRAX100_OV5017_ATA_DMA_HOLD) |
    IO_FIELD( R_ATA_CONFIG, pio_setup,   CONFIG_ETRAX100_OV5017_ATA_PIO_SETUP) |
    IO_FIELD( R_ATA_CONFIG, pio_strobe,  CONFIG_ETRAX100_OV5017_ATA_PIO_STROBE) |
    IO_FIELD( R_ATA_CONFIG, pio_hold,    CONFIG_ETRAX100_OV5017_ATA_PIO_HOLD) );
#endif
  return 0;
}



/*
 * ioctl() system call handler
 */
static int vis_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg) {
  struct vis_dev *dev;
#ifdef VIS_DEBUG
  printk("vis_ioctl called\n");
#endif

  /*  don't decode wrong commands */
  if(_IOC_TYPE(cmd) != VIS_IOC_MAGIC) return -EINVAL;
  if(_IOC_NR(cmd) > VIS_IOC_MAXNR) return -EINVAL;

  switch(cmd) {
  case VIS_IOC_TEST:
    return vis_test(arg);
  case VIS_IOC_GET_IMAGE:
    dev = (struct vis_dev *)filp->private_data;
    vis_enqueue_reader(dev);
    interruptible_sleep_on(&(dev->wait_q));
    memcpy_tofs((void *)arg, dev->img_buf, VIS_IMAGESIZE);
    return 0;
  case VIS_IOC_GET_BASE_ADR:
    /* (WARNING: this is a hack and works only because the ETRAX100
     * does not have a MMU !!!) */
    dev = (struct vis_dev *)filp->private_data;
    return (int)dev->img_buf;
  case VIS_IOC_UPDATE_IMAGE:
    dev = (struct vis_dev *)filp->private_data;
    vis_enqueue_reader(dev);
    interruptible_sleep_on(&(dev->wait_q));
    return 0;
  case VIS_IOC_REGISTER_READ:
    return (int)vis_pio_read((unsigned char)arg);
  case VIS_IOC_REGISTER_WRITE:
    vis_pio_write((unsigned char)(arg >> 16), (unsigned char)arg);
    return 0;
  }

  return 0;
}
```

```
/*
 * open() system call handler
 */
static int vis_open(struct inode *inode, struct file *filp) {
  int i;
  unsigned char *buf;
  struct vis_dev *dev;
  unsigned long flags;
#ifdef VIS_DEBUG
  printk("vis_open called\n");
#endif

  save_flags(flags); cli(); /* do not disturb! */

  /* get a free vis_device */
  i = 0;
  while((i < CONFIG_ETRAX100_OV5017_MAX_READER) && (vis_devices[i].usage))
    i++;

  if (i == CONFIG_ETRAX100_OV5017_MAX_READER) { /* all devices used */
    restore_flags(flags);
    printk("vis_open: No more readers allowed\n");
    return -ENODEV;
  }
  dev = &vis_devices[i];
  dev->usage = 1;

  restore_flags(flags);

  /*
   * get memory and copy TIFF header and trailer
   */
  if((buf =  kmalloc(VIS_TIFF_IMAGESIZE, GFP_DMA | GFP_KERNEL)) == NULL) {
    dev->usage = 0;
    printk("vis_open: kmalloc error\n");
    return -ENODEV;
  }
  memcpy(&buf[0], (void *)tiff_head, VIS_TIFF_HEAD_SIZE);
  memcpy(&buf[VIS_TIFF_HEAD_SIZE + VIS_IMAGESIZE], (void *)tiff_tail, VIS_TIFF_TAIL_SIZE);
#ifdef VIS_DEBUG
  for (i=0; i < VIS_IMAGESIZE; i++)
    buf[i + VIS_TIFF_HEAD_SIZE] = 0;
#endif


  dev->tiff_img_buf = buf;
  dev->img_buf = buf + VIS_TIFF_HEAD_SIZE;


  /*
   * setup DMA descriptors
   */
  dev->ata_dma_descr[0].sw_len = VIS_IMAGESIZE >> 1;
  dev->ata_dma_descr[0].sw_len = (288 - 170) * 385;
  dev->ata_dma_descr[0].ctrl= d_int;
  dev->ata_dma_descr[0].buf = (unsigned long)(dev->img_buf);
  dev->ata_dma_descr[0].next = (unsigned long)&(dev->ata_dma_descr[1]);
  dev->ata_dma_descr[0].hw_len = 0;
  dev->ata_dma_descr[0].status = 0;
  dev->ata_dma_descr[0].fifo_len = 0;
```

```
  dev->ata_dma_descr[1].sw_len = VIS_IMAGESIZE >> 1;
  dev->ata_dma_descr[1].sw_len = 170 * 385;
  dev->ata_dma_descr[1].ctrl= d_eop | d_eol | d_int;
  dev->ata_dma_descr[1].buf = (unsigned long)(dev->img_buf) + (VIS_IMAGESIZE >> 1);
  dev->ata_dma_descr[1].buf = (unsigned long)(dev->img_buf) + (288 - 170) * 385;
  dev->ata_dma_descr[1].next = 0;
  dev->ata_dma_descr[1].hw_len = 0;
  dev->ata_dma_descr[1].status = 0;
  dev->ata_dma_descr[1].fifo_len = 0;

  filp->private_data = dev;
  return 0;
}



/*
 * release() system call handler
 */
static void vis_release(struct inode *inode, struct file *filp) {
  struct vis_dev *dev = filp->private_data;
#ifdef VIS_DEBUG
  printk("vis_release called\n");
#endif
  kfree(dev->tiff_img_buf);
  dev->usage = 0;
}


/*
 * returns a TIFF image
 * Image transfer is initiated if f_pos = 0
 */
static int vis_read(struct inode *inode, struct file *filp, char *buf, int count) {
  /*    struct wait_queue wait = { current, NULL }; */
  unsigned long f_pos = (unsigned long)(filp->f_pos);
  struct vis_dev *dev = filp->private_data;
#ifdef VIS_DEBUG
  printk("vis_read: f_pos %i, count %i\n", f_pos, count);
#endif

  /* check size of transfer */
  if (f_pos > VIS_TIFF_IMAGESIZE)
    return 0;
  if (f_pos + count > VIS_TIFF_IMAGESIZE)
    count = VIS_TIFF_IMAGESIZE - f_pos;

  if (f_pos == 0) {
    /* initiate DMA transfer, sleep until image ready */
    vis_enqueue_reader(dev);
    interruptible_sleep_on(&(dev->wait_q));
  }

  /* copy buffer form kernel space to user space */
  memcpy_tofs(buf, &(dev->tiff_img_buf[f_pos]), count);

  filp->f_pos += count;
  return count;
}
```

```
/*
 * possible file_operations
 */
static struct file_operations vis_fops = {
  NULL,           /* No lseek */
  vis_read,
  NULL,           /* No write */
  NULL,           /* No readdir */
  NULL,           /* No select */
  vis_ioctl,
  NULL,           /* No mmap */
  vis_open,
  vis_release
};


/*
 * initialization of the OV5017 chip
 */
static int vis_ov5017_init(void) {
#ifdef VIS_DEBUG
  printk("vis_ov5017_init: setting register defaults\n");
#endif
  /* write config data to ov5017 registers */
  vis_pio_write(VIS_REG_FCTL, CONFIG_ETRAX100_OV5017_FCTL);
  vis_pio_write(VIS_REG_EXCTL, CONFIG_ETRAX100_OV5017_EXCTL);
  vis_pio_write(VIS_REG_GCTL, CONFIG_ETRAX100_OV5017_GCTL);
  vis_pio_write(VIS_REG_FRCTL, CONFIG_ETRAX100_OV5017_FRCTL);
  vis_pio_write(VIS_REG_MCTL, CONFIG_ETRAX100_OV5017_MCTL);
  vis_pio_write(VIS_REG_HWCTL, CONFIG_ETRAX100_OV5017_HWCTL);
  vis_pio_write(VIS_REG_VWCTL, CONFIG_ETRAX100_OV5017_VWCTL);
}


/*
 * initialization of the ATA-Bus
 * set interrupt handlers
 */
static int vis_ata_init(void) {
  volatile unsigned int dummy;
#ifdef VIS_DEBUG
  printk("vis_ata_init\n");
#endif

  /* setup DMA registers */
  *R_ATA_CTRL_DATA = 0;
  *R_ATA_TRANSFER_CNT = 0;
  *R_ATA_CONFIG = 0;

  genconfig_shadow = (genconfig_shadow &
      ~IO_MASK(R_GEN_CONFIG, dma2) &
      ~IO_MASK(R_GEN_CONFIG, dma3) &
      ~IO_MASK(R_GEN_CONFIG, ata)) |
    ( IO_STATE( R_GEN_CONFIG, dma3, ata    ) |
      IO_STATE( R_GEN_CONFIG, dma2, ata    ) |
      IO_STATE( R_GEN_CONFIG, ata,  select ) );

  *R_GEN_CONFIG = genconfig_shadow;


  /* wait some */
```

```
  dummy = 1;
  dummy = 2;
  dummy = 3;


  /* make a dummy read to set the ata controller in a proper state */
  dummy = *R_ATA_STATUS_DATA;


  /* configure ata transfer times */
  *R_ATA_CONFIG = ( IO_FIELD( R_ATA_CONFIG, enable,     1 ) |
    IO_FIELD( R_ATA_CONFIG, dma_strobe,  CONFIG_ETRAX100_OV5017_ATA_DMA_STROBE) |
    IO_FIELD( R_ATA_CONFIG, dma_hold,    CONFIG_ETRAX100_OV5017_ATA_DMA_HOLD)   |
    IO_FIELD( R_ATA_CONFIG, pio_setup,   CONFIG_ETRAX100_OV5017_ATA_PIO_SETUP)  |
    IO_FIELD( R_ATA_CONFIG, pio_strobe,  CONFIG_ETRAX100_OV5017_ATA_PIO_STROBE) |
    IO_FIELD( R_ATA_CONFIG, pio_hold,    CONFIG_ETRAX100_OV5017_ATA_PIO_HOLD) );

  while(*R_ATA_STATUS_DATA & IO_MASK(R_ATA_STATUS_DATA, busy)); /* wait for busy flag to clear */


  /* clear ata_irq2 (INTRQ2) */
  *R_IRQ_MASK0_CLR = IO_STATE( R_IRQ_MASK0_CLR, ata_irq2, clr );

  /* set interrupt handler for VIS_ATA_IRQ */
  if (request_irq(VIS_ATA_IRQ, vis_ata_intr, SA_INTERRUPT, VIS_NAME, NULL)) {
    printk("unable to assign irq %i\n", VIS_ATA_IRQ);
    panic("irq");
  }

  /* enable ata_irq2 (INTRQ2) */
  *R_IRQ_MASK0_SET = IO_STATE( R_IRQ_MASK0_SET, ata_irq2, set );



  /* clear all pending DMA interrupts */
  *R_DMA_CH3_CLR_INTR = IO_STATE( R_DMA_CH3_CLR_INTR, clr_eop,   do ) |
    IO_STATE( R_DMA_CH3_CLR_INTR, clr_descr, do );
  /* set interrupt handler for VIS_DMA3_IRQ */
  if (request_irq(VIS_DMA3_IRQ, vis_dma_intr, SA_INTERRUPT, VIS_NAME, NULL)) {
    printk("unable to assign irq %i\n", VIS_DMA3_IRQ);
    panic("irq");
  }
  /* enable DMA-descr-irq (descriptor insterupt) */
  *R_IRQ_MASK2_SET = IO_STATE( R_IRQ_MASK2_SET, dma3_descr, set );
#if 0
  *R_IRQ_MASK2_SET = IO_STATE( R_IRQ_MASK2_SET, dma3_eop, set );
#endif

  /* reset the dma channels we will use */
  RESET_DMA(2);
  RESET_DMA(3);
  WAIT_DMA(2);
  WAIT_DMA(3);

  return 0;
}



/*
 * initialize the driver
 * this function is called from mem.c -> chr_dev_init(void)
```

```
 */
int etrax_vis_ov5017_init(void) {
  volatile unsigned int *addr;
  unsigned int val;
  int res;
  unsigned int i;
  unsigned char *buf;
  printk("%s driver v%s (c) 2001 Axel Burri\n", VIS_NAME, VIS_VERSION);

  /*
   * register the device
   */
  if ((res = register_chrdev(VIS_MAJOR, VIS_NAME, &vis_fops)) < 0) {
    printk("unable to get major %d for VIS_OV5017\n", VIS_MAJOR);
    return res;
  }

  /*
   * init variables
   */
  img_request_q = NULL;
  state = STATE_IDLE;

  for (i = 0; i < CONFIG_ETRAX100_OV5017_MAX_READER; i++) {
    vis_devices[i].usage = 0;
    init_waitqueue(&(vis_devices[i].wait_q));
  }

  /*
   * init ATA stuff
   */
  if((res = vis_ata_init()) < 0) {
    printk("error initializing ata port\n");
    return res;
  };

  /*
   * init ov5017 register
   */
  if((res = vis_ov5017_init()) < 0) {
    printk("error initializing ov5017 chip\n");
    return res;
  };
  return 0;
}
```

C.2 Applications

C.2.1 visreg.c

axis/apps/visreg/visreg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/etraxvis.h>
#include <unistd.h>   /*  read() call here */

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

```
#define DEVICE_NAME "/dev/vis"


int openDevice(char* name)
{
    int devfd;

    printf("opening device %s\n", name);
    devfd = open(name, O_RDONLY);
    if (devfd < 1)
    {
        fprintf(stderr, "error %i: failed to open device %s\n", devfd, name);
        exit(-1);
    }
    else
    {
        return devfd;
    }
    return -1;
}


void pioWrite(int devfd, unsigned char reg, unsigned char arg)
{
    unsigned int pioarg;
    int retval;

    pioarg = (reg << 16) | arg;
    fprintf(stdout, "calling ioctl VIS_IOC_REGISTER_WRITE: arg 0x%x\n", pioarg);
    if((retval = ioctl(devfd, VIS_IOC_REGISTER_WRITE, pioarg)) != 0)
    {
        fprintf(stderr, "ioctl failed, err=0x%x\n", retval);
    }
}


int pioRead(int devfd, unsigned char reg)
{
    printf("calling ioctl VIS_IOC_REGISTER_READ: arg 0x%x\n", reg);
    return ioctl(devfd, VIS_IOC_REGISTER_READ, reg);
}


int main(int argc, char *argv[]) {
    int devfd;
    int retval;

    if (argc == 3 && argv[1][0] == 'r') {
        /*
         * PIO read
         */
        devfd = openDevice(DEVICE_NAME);
        retval = pioRead(devfd, (unsigned char)strtol(argv[2], (char **)NULL, 0));
        printf("returned value: 0x%x\n", retval);
    }
    else if (argc == 4 && argv[1][0] == 'w') {
        /*
         * PIO write
         */
        devfd = openDevice(DEVICE_NAME);
```

```
        pioWrite(devfd, (unsigned char)strtol(argv[2], (char **)NULL, 0), (unsigned char)strtol(argv[3], (char **)NULL,
    }
    else {
        fprintf(stderr, "usage: %s [r|w] <register> [value]\n", argv[0]);
    }

    return 0;
}
```

C.2.2 watcher.c

axis/apps/watcher/watcher.c

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>

#include <linux/etraxvis.h>


#define DEVICE_NAME "/dev/vis"
#define DEFAULT_RATE 0  /* seconds */
#define DEFAULT_ALPHA 0.95
#define DEFAULT_PIXEL_THRESHOLD 1000000

unsigned char *imgbuf;
unsigned char *newimg;


int openDevice(char* name) {
  int devfd;

  devfd = open(name, O_RDONLY);
  if (devfd < 1)
    {
      fprintf(stderr, "error %i: failed to open device %s\n", devfd, name);
      exit(-1);
    }
  else
    return devfd;
}


unsigned int getDiffUpdate(unsigned char *old, unsigned char *new, float alpha) {
  float alpha_ = (1 - alpha);
  unsigned int d = 0;
  unsigned int i;
  unsigned char newpix;

  for(i = 0; i < VIS_IMAGESIZE; i++) {
    /* adapt image */
    newpix = alpha * new[i] + alpha_ * old[i];
    /* calc difference */
    d += abs(newpix - old[i]);
    old[i] = newpix;
  }
  return d;
}

void detectChange(int devfd, int rate, unsigned int threshold, float alpha) {
```

```
  unsigned int d;

  /* copy initial new image content */
  if(ioctl(devfd, VIS_IOC_GET_IMAGE, (unsigned long)imgbuf) != 0) {
    perror("get image\n");
    exit(-1);
  }

  while(1) {
    sleep(rate);

    if(ioctl(devfd, VIS_IOC_UPDATE_IMAGE, 0)) {
      perror("update image\n");
      exit(-1);
    }

     if ((d = getDiffUpdate(imgbuf, newimg, alpha)) > threshold) {
      printf("Movement! delta: %i\n", (int)(d - threshold));
    }
  }
}



int main(int argc, char *argv[]) {
  int rate = DEFAULT_RATE;
  unsigned int threshold = DEFAULT_PIXEL_THRESHOLD;
  float alpha = DEFAULT_ALPHA;

  int devfd;
  int ch;


  while ((ch = getopt(argc, argv, "hr:t:a:")) != EOF)
    switch(ch) {
    case 'h':
      printf("usage: %s [-r rate] [-t threshold] [-a alpha]\n", argv[0]);
      exit(0);
      break;
    case 'r':
      rate = atoi(optarg);
      if (rate < 0) rate = DEFAULT_RATE;
      break;
    case 't':
      threshold = atoi(optarg);
      break;
    case 'a':
      alpha = atof(optarg);
      if(alpha > 1 || alpha < 0) alpha = DEFAULT_ALPHA;
      break;
    }


  /* allocate memory */
  if((imgbuf = malloc(VIS_IMAGESIZE)) == NULL) {
    perror("failed to allocate memory\n");
    exit(-1);
  }

  devfd = openDevice(DEVICE_NAME);
  newimg = (unsigned char *)ioctl(devfd, VIS_IOC_GET_BASE_ADR, 0);
```

```
  detectChange(devfd, rate, threshold, alpha);

  return 0;
}
```

C.2.3 visd.c

axis/apps/visd/visd.c

```
/*
 * Internet-Deamon for the OmniVision ov5017 CMOS-chip driver
 *
 * (c) 2001 Axel Burri <aburri@ee.ethz.ch>
 *
 *
 * listens at a given port.
 * accept any connection, get image from the device and send
 * a stream of the image to the caller.
 *
 * v0.2: - added configuration options
 *       - direct access to kernel memory support enabled
 *         (WARNING: this i a hack in the etrax100vis driver!)
 *
 */


#define VISD_VERSION "0.2"


#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <linux/etraxvis.h>


#undef VISD_MEMCPY /* do direct access */

#define DEFAULT_VISDRIVER_PATHNAME "/dev/vis"
#define DEFAULT_PORT 7777
#define DEFAULT_BACKLOG 10  /* how many pending connections queue will hold */



void getImage(int devfd, unsigned char *buf) {
#ifdef VISD_MEMCPY
  if(ioctl(devfd, VIS_IOC_GET_IMAGE, (unsigned long)buf) != 0) {
    perror("get image\n");
```

```
    exit(-1);
  }
#else
  if(ioctl(devfd, VIS_IOC_UPDATE_IMAGE, 0)) {
    perror("update image\n");
    exit(-1);
  }
#endif
}



void serverLoop(int devfd, unsigned char* imgbuf, int port, int backlog, int verbose) {
  int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
  struct sockaddr_in my_addr;    /* my address information */
  struct sockaddr_in their_addr; /* connector's address information */
  int sin_size;
  time_t now;

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  my_addr.sin_family = AF_INET;         /* host byte order */
  my_addr.sin_port = htons(port);       /* short, network byte order */
  my_addr.sin_addr.s_addr = INADDR_ANY; /* automatically fill with my IP */
  bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

  if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
  }

  if (listen(sockfd, backlog) == -1) {
    perror("listen");
    exit(1);
  }
  now = time(NULL);
  printf("visd service started (port %i) at %s", port, ctime(&now));

  while(1) {  /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1) {
      perror("accept");
      continue;
    }
    now = time(NULL);
    if (verbose) printf("%s accept: %s", inet_ntoa(their_addr.sin_addr), ctime(&now));
    if (!fork()) {
      /*
       * this is the child process.
       * get image*/
      getImage(devfd, imgbuf);
      if (send(new_fd, imgbuf, VIS_IMAGESIZE, 0) == -1)
perror("send");
      close(new_fd);
      exit(0);
    }
    close(new_fd);  /* parent doesn't need this */
```

```
    /* clean up all child processes */
    while(waitpid(-1,NULL,WNOHANG) > 0);
  }

}


void printUsage(char* progName) {
  printf("usage: %s [-p port] [-b backlog] [-d device] [-q] [-v]\n", progName);
  printf("Open a port to which images are sent\n\n");
  printf(" -p PORT        set port, default is %i\n", DEFAULT_PORT);
  printf(" -b BACKLOG     set backlog (max # of connections), default is %i\n", DEFAULT_BACKLOG);
  printf(" -d PATHNAME    set device to be used, default is `%s'\n", DEFAULT_VISDRIVER_PATHNAME);
  printf(" -q             turn on quiet mode\n");
  printf(" -v             show version information\n");
}



int main(int argc, char *argv[]) {
  unsigned char *imgbuf;
  int devfd;
  int ch;
  int port = DEFAULT_PORT;
  int backlog = DEFAULT_BACKLOG;
  char* vispathname = DEFAULT_VISDRIVER_PATHNAME;
  int verbose = -1;

  while ((ch = getopt(argc, argv, "hp:c:d:qv")) != EOF)
    switch(ch) {
    case 'h':
      printUsage(argv[0]);
      exit(0);
      break;
    case 'p':
      port = atoi(optarg);
      if (port < 0) {
printUsage(argv[0]);
exit(-1);
      }
      break;
    case 'b':
      backlog = atoi(optarg);
      if (backlog < 1) {
printUsage(argv[0]);
exit(-1);
      }
      break;
    case 'd':
      vispathname = optarg;
      if (vispathname == "") {
printUsage(argv[0]);
exit(-1);
      }
      break;
    case 'q':
      verbose = 0;
      break;
    case 'v':
      printf("%s version %s\n", argv[0], VISD_VERSION);
      exit(-1);
      break;
```

```
    }


  /* open device */
  if ((devfd = open(vispathname, O_RDONLY)) < 1) {
    perror("open device\n");
    exit(-1);
  }


#ifdef VISD_MEMCPY
  /* allocate memory */
  if((imgbuf = malloc(VIS_IMAGESIZE)) == NULL) {
    perror("malloc\n");
    exit(-1);
  }
#else
  imgbuf = (unsigned char *)ioctl(devfd, VIS_IOC_GET_BASE_ADR, 0);
#endif


  serverLoop(devfd, imgbuf, port, backlog, verbose);


  return 0;
}
```

C.2.4 VisClient.java

visclient/VisClient.java

```
/*
 * image request client v0.1
 *
 * (c) 2001 Axel Burri <aburri@ee.ethz.ch>
 *
 *
 * works together with "visd", the image server daemon.
 * requests connection from a given port on the server and
 * get a stream of the image from the server there
 * a stream of the image to the caller.
 *
 *
 */

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.image.*;
import java.applet.Applet;
import java.awt.event.*;


public class VisClient extends Applet implements Runnable {
    int delay=5000;
    int port=7777;
    byte[] buf;
    Thread animatorThread = null;
    boolean frozen = false;
    Image img = null;
    Socket sock = null;
```

```
    public String getAppletInfo() {
return "Webcam server by Axel Burri";
    }


    public void init() {
String str;
System.err.println("%%%  Init  %%%");

// get the parameters from the HTML-page
str = getParameter("delay");
try {
    if (str != null) {
delay = Integer.parseInt(str);
    }
} catch (Exception e) {}
if(delay < 0) delay = 3000;

str = getParameter("port");
try {
    if (str != null) {
port = Integer.parseInt(str);
    }
} catch (Exception e) {}
if(port <= 0) port = 7777;

// allocate memory
buf = new byte[385*288];
    }


    public void getImage() {
int off = 0;
int count = 0;
int size = 385*288;

try {
    System.out.println(getCodeBase().getHost());
    System.out.println( port);
    sock = new Socket(getCodeBase().getHost(), port);
    DataInputStream in = new DataInputStream(sock.getInputStream());

    while (size > 0) {
count = in.read(buf, off, size);
off += count;
size -= count;
    }

    in.close();
    sock.close();
} catch (Throwable e) {
    System.out.println("Error " + e.getMessage());
    e.printStackTrace();
}

ColorModel cm = new DirectColorModel(24, 255, 255, 255);
img = createImage(new MemoryImageSource(385, 288, cm, buf, 0, 385));
    }


    public void paint(Graphics g) {
```

```
if(img != null)
    g.drawImage(img, 0, 0, this);
super.paint(g);
    }


    public void update(Graphics g) {
paint(g);
    }



    public void start() {
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent e) {
    if (frozen) {
frozen = false;
start();
    } else {
frozen = true;
stop();
    }
}
    });

if (!frozen) {
    if (animatorThread == null) {
animatorThread = new Thread(this);
    }
    // start the imgage reader thread
    animatorThread.start();
}
    }


    public void stop() {
animatorThread = null;
    }


    public void run() {
// lower this thread's priority
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);

long startTime;
Thread currentThread = Thread.currentThread();

while (currentThread == animatorThread) {
    startTime = System.currentTimeMillis() + delay;
    getImage();
    repaint();

    //Delay depending on how far we are behind.
    try {
Thread.sleep(Math.max(0, startTime - System.currentTimeMillis()));
    } catch (InterruptedException e) {
break;
    }
}
    }
}
```

C.3 Configuration

C.3.1 head.S

axis/elinux/arch/etrax100/kernel/head.S

```
;; Zeile 185
;; r0 is shadow register
#ifdef CONFIG_ETRAX100_OV5017
;; disable ATA before enabling it in genconfig below
moveq 0,r0
move.d r0,[R_ATA_CTRL_DATA]
move.d r0,[R_ATA_TRANSFER_CNT]
move.d r0,[R_ATA_CONFIG]
#endif

;; Zeile 258
#ifdef CONFIG_ETRAX100_OV5017
;; dma2=ata: bit 10,11=3
;; dma3=ata: bit 12,13=3
;; select ata: bit 1=1
or.d 0x3c02,r0 ; DMA channels 2 and 3 to ATA, ATA enabled
#endif
```

C.3.2 Config.in

axis/elinux/drivers/char/Config.in
Angefügte Zeilen:

```
bool 'Etrax 100 OV5017 support' CONFIG_ETRAX100_OV5017
if [ "$CONFIG_ETRAX100_OV5017" = "y" ]; then
  int '  Max readers allowed'  CONFIG_ETRAX100_OV5017_MAX_READER 4
  hex '  FCTL  Register (hex)' CONFIG_ETRAX100_OV5017_FCTL 00
  hex '  EXCTL Register (hex)' CONFIG_ETRAX100_OV5017_EXCTL ff
  hex '  GCTL  Register (hex)' CONFIG_ETRAX100_OV5017_GCTL 00
  hex '  FRCTL Register (hex)' CONFIG_ETRAX100_OV5017_FRCTL 00
  hex '  MCTL  Register (hex)' CONFIG_ETRAX100_OV5017_MCTL 02
  hex '  HWCTL Register (hex)' CONFIG_ETRAX100_OV5017_HWCTL 00
  hex '  VWCTL Register (hex)' CONFIG_ETRAX100_OV5017_VWCTL 00
  int '  ATA_DMA_STROBE'       CONFIG_ETRAX100_OV5017_ATA_DMA_STROBE 01
  int '  ATA_DMA_HOLD'         CONFIG_ETRAX100_OV5017_ATA_DMA_HOLD  00
  int '  ATA_PIO_SETUP'        CONFIG_ETRAX100_OV5017_ATA_PIO_SETUP 00
  int '  ATA_PIO_STROBE'       CONFIG_ETRAX100_OV5017_ATA_PIO_STROBE 10
  int '  ATA_PIO_HOLD'         CONFIG_ETRAX100_OV5017_ATA_PIO_HOLD  00
fi
```

Anhang D

Beilagen

Die in diesem Anhang vorhandenen Datenblätter und Schemata sind aufs Notwendigste beschränkt. Wer die vollständigen Dokumentationen zu Kamera, Development Board oder Prozessor benötigt, greift via Internet einfach darauf zu. Im Literaturverzeichnis finden sich die Internet-Adressen der Hersteller.

D.1 Bus Converter

D.1.1 PIO Timingdiagramme

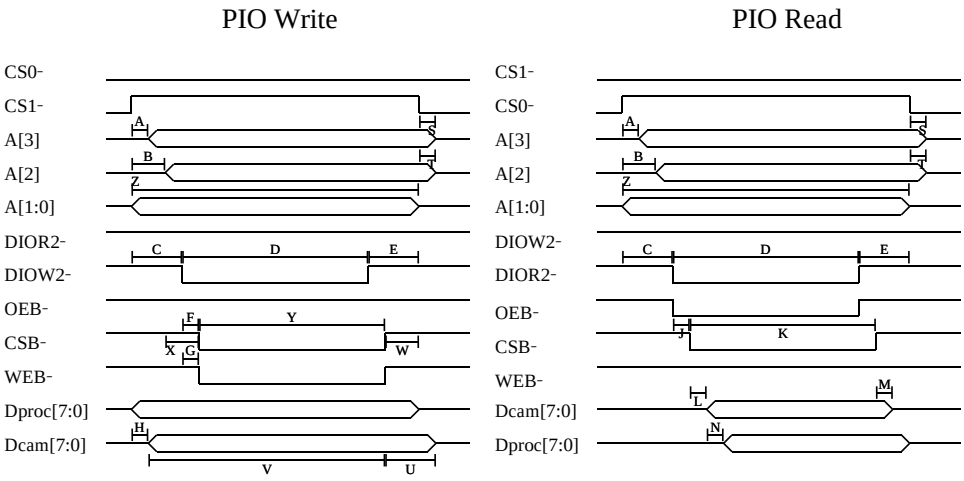

Abbildung D.1: PIO Timing

Signale

Bezeichnung	*Funktion*	*Timing*
CS0-	PIO Read Tristate Enable	Aktiv während PIO Read Cycle
CS1-	PIO Write Tristate Enable	Aktiv während PIO Write Cycle
A[3]	0 während PIO Read / Write (Seite Kamera)	0 falls CS0- oder CS1- aktiv
A[2]	Oberstes bit der Register- Adresse (Seite Kamera)	Frei wählbar während CS0- oder CS1- aktiv
A[1:0]	Untere 2 bits der Register- Adresse (Seite Kamera	Frei wählbar
DIOW2-	Write Timing	Kamera übernimmt Daten bei positiver Flanke von DIOW2-
DIOR2-	Read Timing	Prozessor übernimmt Daten bei positiver Flanke von DIOR2-
OEB-	Output Enable Kamera	Wird direkt von DIOR2- angesteuert, synchron zu CSB-
CSB-	Chip Select Kamera	Aktiv während Read / Write
WEB-	Write Enable Kamera	Wird von DIOW2- angesteuert
Dcam[7:0]	Datenbus Seite Kamera	Write: Daten vom Write Tristate Read: Daten von Kamera
Dproc[7:0]	Datenbus Seite Prozessor	Write: Daten vom Prozessor Read: Daten vom Read Tristate

Zeiten

Bezeichnung	*Beschreibung*	*Wert*
A	CSx- to A[3] Delay	$1*T_{INV} + 1*T_{AND}$ = 2ns..20ns
B	CSx- to A[2] Delay	$2*T_{INV} + 2*T_{AND}$ = 4ns..40ns
C	PIO_SETUP	(0+1)*20ns = 20ns
D	PIO_STROBE	(10+1)*20ns = 220ns
E	PIO_HOLD	(0+1)*20ns = 20ns
F	DIOW2- to CSB- Delay	$1*T_{AND}$ = 1ns..10ns
G	DIOW2- to WEB- Delay	$1*T_{AND}$ = 1ns..10ns
H	Data Write Tristate Delay	MAX(T_{EN}, T_{PROPAG}) = 4.6ns..16.4ns
I	T_{DS}	>=20ns
J	T_{DH}	>=0ns
K	T_{RC}	>=100ns
L	T_{CSA}	<=30ns
M	T_{CSX}	<=15ns
N	Data Read Tristate Delay	MAX(T_{EN} −C −J −L, T_{PROPAG}) = 10.5ns
O	T_{AS}	>=0ns
P	T_{WC}	>=100ns
Q	T_{CS}	>=50ns
R	T_{AH}	>=0ns
S	A[3] Delay to Invalidity	$1*T_{AND}$ + MIN(T_{NAND}, T_{INV}) = 2ns..20ns
T	A[2] Delay to Invalidity	$1*T_{AND + TNAND}$ = 2ns..20ns
U	DIOR2- to CSB- Delay	$1*T_{AND}$ = 1ns..10ns
T_{INV}	Inverter Delay	1ns..10ns
T_{AND}	AND Delay	1ns..10ns
T_{NAND}	NAND Delay	1ns..10ns
T_{EN}	Tristate Enable Delay	4.6ns..16.4ns
T_{PROPAG}	Tristate Propagation Delay	3ns..10.5ns

Legende

SIGNAL	High- aktives Signal	Gemessen direkt am Augang des Erzeugers
SIGNAL-	Low- aktives Signal	Gemessen direkt am Augang des Erzeugers
X	Signalpegel irrelevant	

Abbildung D.2: PIO Legende

D.1.2 DMA Timingdiagramme

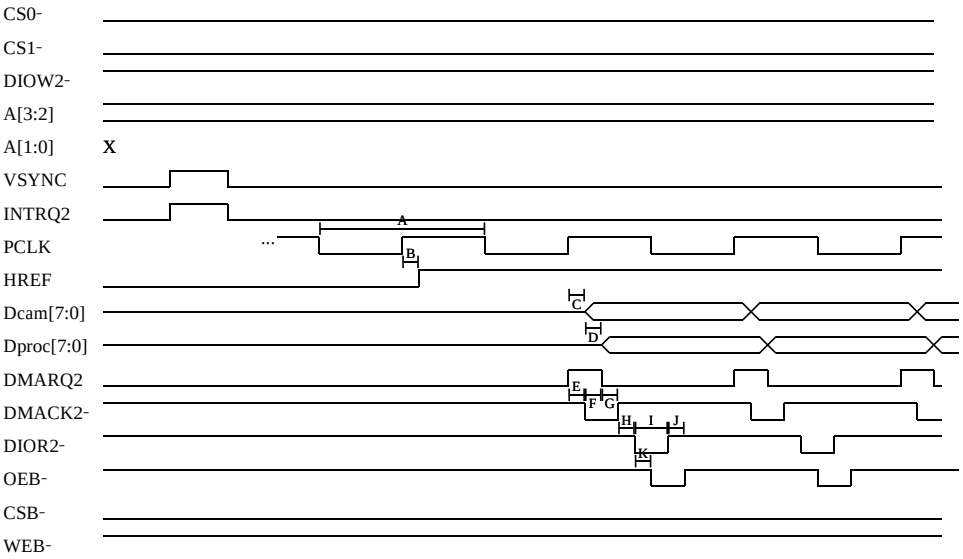

Abbildung D.3: DMA Timing

Signale

Bezeichnung	*Funktion*	*Timing*
CS0-	PIO Read Tristate Enable	Immer inaktiv
CS1-	PIO Write Tristate Enable	Immer inaktiv
DIOW2-	Write	Immer inaktiv
A[3:2]	A[3:0]=10xx = Video Data	10 während CS0- und CS1- inaktiv
A[1:0]	A[3:0]=0xxx = Register Data	Irrelevant während CS0- und CS1- inaktiv
VSYNC	Vertical Synchronization	Kurzer Aktiv- Puls trennt Frames
INTRQ2	Frame Interrupt Request	Wird direkt von VSYNC angesteuert
PCLK	Pixeltakt (im Konfigurations- register invertiert)	14.318MHz/[(FDIV+1)*2), FDIV=[0, 1, 2, ... 63], 7.159MHz bei 50fps
HREF		Aktiv während gültiger Bildzeile
Dcam[7:0]	Datenausgang Kamera. 8bit- Pixel liegen hier an.	Aktualisiert auf positive Flanke von PCLK, gültig auf negative Flanke von PCLK
Dproc[7:0]	Dateneingang am Prozessor.	Daten von Dcam[7:0] erscheinen hier bei aktivem DMA- Tristate
DMARQ2	DMA Request pro Pixel	Synchronisiert DMA auf PCLK
DMACK2-	DMA Acknowledge pro Pixel	Setzt DMARQ2 zurück
DIOR2-	Read	Prozessor übernimmt Daten von Dproc[7:0] bei positiver Flanke von DIOR2-
OEB-	Output Enable Kamera	Wird direkt von DIOR2- angesteuert
CSB-	Chip Select Kamera	Immer aktiv
WEB-	Write Enable Kamera	Immer inaktiv

Zeiten

Bezeichnung	*Beschreibung*	
A	T_{PCLK}	139ns bei 50fps, 278ns bei 25fps
B	T_{PHD}	<25ns
C	T_{PDD}	<25ns
D	Tristate Propagation Delay	T_{PROPAG} = 3ns..10.5ns
E	DMA REQ to ACK Delay	Unbekannt, leicht variierend, Messung nötig
F	DMA REQ Reset Delay	4.1ns..14.6ns
G	DMA Start Delay	Unbekannt, Messung nötig
H	DMA_SETUP	20ns
I	DMA_STROBE	(1+1)*20ns=40ns bei 50fps, (6+1)*20ns=140ns bei 25fps
J	DMA_HOLD	(0+1)*20ns=20ns
K	DIOR2- to OEB- Delay	0ns
T_{INV}	Inverter Delay	1ns..10ns
T_{AND}	AND Delay	1ns..10ns
T_{NAND}	NAND Delay	1ns..10ns
T_{EN}	Tristate Enable Delay	4.6ns..16.4ns
T_{PROPAG}	Tristate Propagation Delay	3ns..10.5ns

Legende

SIGNAL	High- aktives Signal	Gemessen direkt am Augang des Erzeugers
SIGNAL-	Low- aktives Signal	Gemessen direkt am Augang des Erzeugers
X	Signalpegel irrelevant	

Abbildung D.4: DMA Legende

D.1.3 Schaltung 1 bis 3, Spannungsversorgung

D.1.4 Bestückung und Anschlüsse des Prototypenboards

D.2 CMOS-Kamera

D.2.1 Omnivision OV5017

D.3 Axis Development Board

D.3.1 Schaltungs- und Bestückungs-Schemata

D.3.2 Notwendige Änderungen

D.3.3 Auszüge aus der Dokumentation

D.4 Netsilicon Development Board

D.4.1 Skizzen und Entwürfe für Busanbindung

D.5 Datenblätter der elektronischen Komponenten