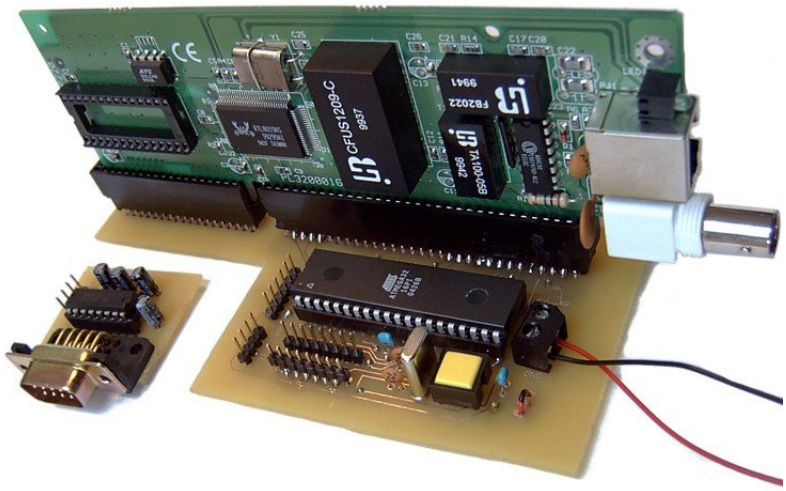
Studienarbeit:
Netzwerk-Kommunikation mit einem Mikrocontroller

Thomas Wilbert und Daniel Pähler
thowil@uni-koblenz.de, tulkas@uni-koblenz.de

Betreuer: Dr. Merten Joost

20. September 2005



Zusammenfassung

Diese Studienarbeit behandelt den Anschluss eines AVR-Mikrocontrollers der Firma Atmel an ein Ethernet-Netzwerk. Die dazu notwendige Hardware – Grundaufbauten sowie diverse zusätzliche Peripherie wie z.B. MultiMediaCard als externes Speichermedium und der Anschluss eines CCD-Chips – werden erläutert. Die zur Realisierung benötigte Software wird besprochen und anhand von Beispielen erklärt.

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 Einführung und Zielsetzung

Mikrocontroller finden im alltäglichen Leben immer häufiger Verwendung. Sei es in Autos, Waschmaschinen oder sogar Sportschuhen – Mikrocontroller sind derart klein, kostengünstig, robust und zugleich leistungsfähig, dass sie praktisch jede (nicht allzu umfangreiche) Steuerungs- bzw. Regelungsaufgabe übernehmen können.
Gleichzeitig hat sich auch die Rolle von Rechnernetzen verändert: In Unternehmen haben Client-Server-Systeme[1] zentrale Rollen angenommen, was der Vernetzung von immer mehr Rechnern Vorschub leistet. Auch für Heimanwender sind Netzwerklösungen so preiswert und einfach geworden, dass sie in immer mehr Haushalten zu finden sind.
Es bietet sich also an, diese beiden bewährten Technologien zu kombinieren, um für Steuerungsaufgaben ein Höchstmaß an Flexibilität zu erreichen; die Umsetzung dieser Idee und damit das Ziel dieser Studienarbeit ist ein Mikrocontroller, der über ein Netzwerk kommunizieren kann.

Bei der Suche nach Möglichkeiten, einen Mikrocontroller netzwerkfähig zu machen, gibt es einige wichtige Faktoren zu beachten:

- Es wird in aller Regel zusätzliche Hardware benötigt. Ein Aufbau, um einen gängigen Mikrocontroller wie solche aus der Atmel AVR-Serie, PIC-Controller oder auch Geräte der 8051-Serie für eine vollständige Teilnahme am Netzwerkverkehr in irgendeiner Art direkt an ein Ethernet-System anzuschließen, ist nicht möglich. Zwar existieren Konzepte, um zumindest rudimentär Daten versenden zu können (siehe auch [Prob]); hierzu ist bei einem Ethernet der 10Mbps-Klasse das Generieren eines Signals einer Maximalfrequenz von 10^7Hz notwendig[2], was dem Anlegen eines Spannungswertes alle 50ns bzw. $2 * 10^7$ mal pro Sekunde entspricht. Ein Controller jedoch, der Daten von einem Ethernet der 10Mbps-Klasse lesen soll, müsste nach dem Abtasttheorem ein Minimum von $2 * 10^7$ Samples des Spannungsverlaufes auf der Leitung nehmen. Ein Senden mit ordnungsgemäßer Kollisionskontrolle setzt wiederum das parallele Erzeugen und Lesen des Signals voraus – dies sind Aufgaben, die auch die leistungsstärksten low-cost-Controller mit 16MHz Taktfrequenz nicht mehr bewerkstelligen können.[3] Kurzum:

[1] Systeme, in denen ein zentraler Server eine Funktionalität anbietet, die von mehreren Clients in Anspruch genommen werden kann.

[2] 10Mbps, wobei der Manchestercode angewandt wird

[3] Schon das rudimentäre Senden in [Prob] ist nur durch Übertaktung auf 20MHz zu erreichen.

der Anschluss des Controllers an das Netzwerk kann nur mit zusätzlicher Hardware geschehen. Ein hierfür verwendbarer IC wird auch NIC (Network Interface Controller) genannt, solche Controller sind in großer Auswahl auf dem Markt verfügbar. Die Notwendigkeit der Verwendung eines zusätzlichen Netzwerkcontrollers führt zum nächsten Punkt:

- Über den eingesetzten Netzwerk-Controller müssen ausreichende Informationen verfügbar sein; zwar ist es möglich, aus beliebigen Ethernet- oder sogar WLAN-Karten die Chips herauszulöten, um sie für eigene Zwecke zu gebrauchen, wenn jedoch nicht wenigstens ein Datenblatt des Herstellers zum jeweiligen Chip vorliegt, ist es praktisch unmöglich, diesen sinnvoll einzusetzen. Dies ist auch der Grund, warum sich die Autoren trotz der offensichtlichen Vorteile von WLAN gegenüber dem kabelgebundenen Ethernet für letzteres entschieden haben: Die Beschaffung eines WLAN-Chips samt Dokumentation war zum Zeitpunkt des Beginns dieser Studienarbeit nicht möglich.

- Die Ethernet-Anbindung ist kein Selbstzweck, der Mikrocontroller soll natürlich imstande sein, zusätzlich zur Ansteuerung des NICs noch weitere Aufgaben aus den Bereichen Messen, Steuern und Regeln zu übernehmen. Daher sollte die Lösung so flexibel wie möglich sein. Vorgefertigte Hard- bzw. Software, die kaum Möglichkeiten der Erweiterung lassen, sind also wenig wünschenswert.

- Die Kosten müssen gering bleiben. Da es je nach praktischem Einsatz, für das die Mikrocontroller/Ethernet-Kombination vorgesehen ist, durchaus zu physischen Schäden kommen kann, ist natürlich die Möglichkeit wünschenswert, Einzelteile auszutauschen oder sogar die komplette Platine neu zu fertigen, ohne dass allzu große Kosten entstehen.

1.2 Verfügbare Lösungsansätze

Die Recherche im Internet nach bereits vorhandenen Lösungen liefert zahlreiche Ansätze, die mehr oder weniger gut den gestellten Ansprüchen genügen. Die interessantesten sind hierbei:

Ethernut: Bei Ethernut handelt es sich um eine Kombination aus Hardwaredesign und darauf aufsetzender Software[4]; Die Hardware ist im wesentlichen eine Platine, die mit einem Atmel ATmega128 Mikrocontroller und wahlweise einem von zwei möglichen Ethernet-Controllern bestückt ist (RTL8019AS von Realtek oder LAN91C111 von SMSC).

[4] für detaillierte Informationen siehe auch [Hara]

Da das Hardwarelayout schon feststeht (tatsächlich ist es möglich, die Hardware komplett zu bestellen), wurde die Platine mit einigen Erweiterungen versehen (z.B. mehr RAM etc.), die den Anreiz erhöhen sollen, mit diesem Board zu arbeiten. Die Software ist wie das Hardwaredesign "Open Source" und bietet das Echtzeit-Betriebssystem Nut/OS sowie TCP/IP-Unterstützung.
Trotz der zahlreichen Vorteile, die diese Lösung hat (z.B. 100Mbit Ethernet im Falle des SMSC-Chips), spricht ein gravierender Nachteil dagegen, sie einzusetzen: die Kosten. Selbst die einfachste Version kostet noch mindestens 95 Euro – die Platine hingegen selbst zu bestücken empfehlen die Entwickler nur echten Profis. Die fast ausschließlich in SMD-Technik zu verbauenden Elemente erschweren das Löten von Hand ebenso wie die in der Regel im QFP-Package gelieferten ICs.

EDTP Packet Whacker: Der EDTP Packet Whacker ist eine einfache Platine, die außer einem RTL8019AS noch die zum physischen Anschluss an das Netzwerk notwendigen Bauteile mitbringt (z.B. RJ45-Buchse, Entkopplung usw.). Der Aufbau der Hardware entspricht in etwa dem, der auch auf einer mit dem RTL8019AS bestückten ISA-Ethernetkarte zu finden ist. Die benötigten Anschlüsse des NICs sowie auch für die Spannungsversorgung führt das Packet Whacker Modul als durchkontaktierte Bohrungen im Standard-2,54mm-Raster heraus, wodurch die Platine einfach auf entsprechende Pinhead-Reihen auf der den Mikrocontroller beherbergenden Platine aufgesteckt werden kann. An Software werden einige von EDTP selbst als "Starter-Code" bezeichnete Codebeispiele mitgeliefert. Das Packet Whacker Modul ist zwar eine elegante Lösung, die Ähnlichkeit zur ISA-Karte sowie deren wesentlich (zum Zeitpunkt der Erstellung dieser Arbeit um 2 Größenordnungen) niedrigerer Preis legt jedoch die im nächsten Punkt beschriebene Option nahe.

ISA-Karte mit RTL8019AS-Chip & uIP: Diese Kombination aus Hard- und Software hat verschiedene Vorteile:

Hardware: Eine alte ISA-Netzwerkkarte wird verwendet. Nicht nur sind solche Karten sehr preiswert, es ist auch schon an Bauteilen alles auf der Platine vorhanden, was sonst in Handarbeit an den Netzwerkchip angeschlossen werden müsste (Netzwerkbuchse, Pegelwandler, etc.). Da also an der Karte selbst kaum etwas verändert werden muss, stellt ihr Austausch im Falle eines Defekts

kein Problem dar. Zum Anschluss des Mikrocontrollers an die Karte gibt es bereits diverse Informationsquellen im Internet, zudem existiert zum RTL8019AS-Chip eine umfassende Dokumentation von Realtek selbst. Dass seine Funktionsweise schon lange bekannt und ausreichend dokumentiert ist, ist vermutlich der Grund, warum dieser Netzwerkchip in verschiedenen Projekten eingesetzt wird, die mit Mikrocontrollern zu tun haben.

Software: Der Open Source TCP/IP-Stack uIP[5] ist eine in C geschriebene frei erhältliche Software, auf die eigene Programme aufgesetzt werden können, die auf dem Mikrocontroller laufen und über TCP/IP kommunizieren sollen. Hierbei bieten sich mannigfaltige Konfigurationsmöglichkeiten, viele Features lassen sich beispielsweise zugunsten der Performanz deaktivieren oder den individuellen Bedürfnissen anpassen. Ein Hardwaretreiber für den RTL8019AS ist vorhanden, sodass dessen Verwendung für uIP kein Problem darstellen sollte.

Die niedrigen Kosten sowie die hohe Flexibilität gaben schließlich den Ausschlag, die letztgenannte Lösung einzusetzen. Auf die verwendete Hardware wird detaillierter in Abschnitt 3 eingegangen. Die Software und ihre Konfigurationsmöglichkeiten werden darauf folgend in Abschnitt 4.2 betrachtet.

[5] siehe auch [Sofd]; eine speziell auf AVR Mikrocontroller angepasste Version wird im Rahmen des uIP-AVR-Projektes verwaltet, siehe [Sofb]

2 Netzwerkgrundlagen

In diesem Abschnitt werden kurz einige Grundlagen zum Thema Netzwerk erläutert. Er dient ausdrücklich nicht einer detaillierten Beschreibung der verschiedenen Protokolle und Funktionsweisen, sondern soll vielmehr einen Überblick über die beteiligten Komponenten und Protokolle ermöglichen sowie einen Einblick darin geben, welche Aufgaben Hard- und Software des in dieser Arbeit aufgebauten Gerätes zu bewältigen haben, um eine Netzwerkkommunikation zu erreichen.

2.1 Ethernet

Ethernet beschreibt eine Technologie zur Vernetzung von Geräten ("Hosts") zu sog. LANs (local area networks). Der Ethernetstandard definiert nicht nur Kabeltypen sowie Signalpegel und -formen, um einzelne Bits zu übertragen, er regelt auch die Zusammenfassung von einzelnen Daten zu Paketen und sieht Mechanismen vor, den gemeinsamen Zugriff auf das verwendete Vernetzungsmedium zu ermöglichen. Damit deckt Ethernet etwa die Aufgaben der Bitübertragungs- sowie der Sicherungsschicht ab, wie sie im OSI-Modell definiert sind. Jedes am Netzwerk beteiligte Gerät verfügt über eine weltweit eindeutige 48-Bit-Adresse (MAC bzw. Media Access Control), über welche Sender und Empfänger von Daten identifiziert werden. Für das Versenden von Daten sieht Ethernet zunächst eine Segmentierung in Pakete vor, deren Länge nicht größer als 1500 Byte sein darf. Anschließend werden die Pakete mit einem sog. Link-Level-Header versehen, der zumindest aus Ziel und Quelladresse sowie einem zwei Byte langen Typfeld, welches das in der nächst höheren Schicht verwendete Protokoll angibt, besteht[6]. An die Payload angehängt wird eine vier Byte lange CRC32 Checksumme, die sich über das gesamte zu sendende Frame erstreckt. Mit dieser kann der Empfänger feststellen, ob die Daten auf dem Leitungsweg korrumpiert wurden und ggf. das Frame verwerfen. Eine der ersten gängigen Methoden zur Vernetzung mittels Ethernet war, alle am LAN beteiligten Geräte mit einem Koaxialkabel zu verbinden. Da diese Methode sehr anfällig für Ausfälle war und zudem nicht gut mit der Anzahl der angeschlossenen Geräte skalierte, ist heute die Mehrzahl aller Ethernet verwendenden Systeme mittels Twisted-Pair-Kabel an das Netzwerk angeschlossen. Am Zugriffssystem auf das Medium selbst hat das jedoch nichts geändert. Die Zugriffskontrolle übernimmt der sog. CSMA/CD-Algorithmus (*Carrier Sense Multiple Access/Collision Detection*). Sollen Daten versandt werden, so wird zunächst gewartet, bis die

[6]Dies bezieht sich auf das heute fast ausschließlich verwendete Ethernet II Frameformat. Vorher war anstelle des Typenfeldes ein Längenfeld vorgesehen.

Leitung von keinem anderen Netzteilnehmer zum Senden verwendet wird. Dann beginnt er, seinen Datenstrom auf die Leitung zu bringen. Gleichzeitig lauscht er auf die Signale, die auf der Leitung auftreten. Wird ein anderes als das zu versendende Signal festgestellt, so wird dies als Kollision gewertet (zwei Hosts versuchten annähernd gleichzeitig zu senden) und das Senden wird abgebrochen. Nach einer zufällig gewählten Zeit wird der Sendevorgang wiederholt, bis entweder Erfolg oder, bei zu häufig auftretender Kollision, Fehlschlag an die darüber liegende Schicht gemeldet wird.

Das verwendete Softwaresystem muss zumindest das Segmentieren der zu sendenden Daten sowie das Generieren der Headerdaten beherrschen. Erstellen und Prüfen der CRC32-Checksummen sowie das Ausführen von CSMA/CD kann der NIC erledigen.

2.2 Internet Protocol

Um vom zur Datenübertragung genutzten Medium und der physikalischen Art der Übertragung unabhängig zu sein und auch den Datenaustausch zwischen in verschiedenen physikalischen Netzen liegenden Hosts zu ermöglichen, wird das sog. Internet Protocol (IP) eingesetzt. IP ist ein in der Vermittlungsschicht des OSI-Modells angesiedeltes Protokoll, welches zur Adressierung der einzelnen Hosts logische Adressen in Form von 32-Bit-Zahlen einsetzt. Zusätzlich sind die Adressen in einzelne Subnetze eingeteilt. Die zu versendenden Daten werden in geeignet große Teilpakete zerlegt und mit einem gewöhnlicherweise 20 Byte umfassenden Header versehen, der z.B. Ziel- und Quell-IP-Adresse, Versionsnummer usw. enthält. Über den Header wird eine Prüfsumme gebildet, um so bei der Übertragung "beschädigte" Pakete erkennen zu können. Wird nun ein Paket an einen Rechner gesendet, der sich nicht im gleichen physischen Netzwerk befindet, so kann das Paket von einem Rechner, der Verbindung zu mehreren Netzen unterhält, entgegengenommen und über ein anderes Netz weiter Richtung Zielhost gelenkt werden (Routing). Dieser Vorgang ist für den Benutzer von IP vollkommen transparent, das Netz kümmert sich also praktisch selbständig um die Lieferung der Paket ans Ziel. Jedoch wird ein vollständiges Ankommen der Nachricht beim Empfänger bei IP nicht garantiert und auch über die zeitliche Reihenfolge des Eintreffens einzelner Pakete wird keine Zusicherung gemacht. Ebenso kann es vorkommen, dass einzelne Datenfragmente doppelt ihr Ziel erreichen. Die auf dem Mikrocontroller laufende Software muss in der Lage sein, die als Payload eines Ethernetframes dienenden IP-Pakete zum einem mit kompletttem Header zu erstellen zum anderen auch ankommende Pakete richtig zu interpretieren. Anders als auf Ethernetebene müssen hier alle Aufgaben vom Controller übernommen werden, der NIC leistet keinerlei Hilfestellung mehr.

2.3 Address Resolution Protocol

Da auf IP-Ebene alle Hosts nur noch mit logischen Adressen angesprochen werden, muss ein Mechanismus zur Verfügung stehen, der eine Zuordnung von physikalischen zu logischen Adressen erlaubt. Dieses Bindeglied zwischen IP und Ethernet stellt das Address Resolution Protocol, kurz ARP, dar. Mittels dieses Protokolls tauschen in einem Ethernet agierende Rechner Informationen über die Adresszuordnungen aus. Die einzelnen Hosts können Adressen anderer erfragen und auf Anfragen nach ihrer eigenen Adresse antworten. Jeder Host verwaltet einen sog. ARP-Cache, eine Art Tabelle, in der die Adresszuordnungen gespeichert werden. Diese Aufgabe muss natürlich auch vom Mikrocontroller übernommen werden. Es ist insbesondere darauf zu achten, dass die Einträge in der Tabelle nicht veralten, da sonst kein ordnungsgemäßer Netzwerkverkehr mehr stattfinden kann.

2.4 Internet Control Message Protocol

Um Informationen über den Status einzelner IP-Pakete auszutauschen, bedienen sich die Hosts des Internet Control Message Protocol (ICMP). Über dieses selbst auf IP aufsetzende Protokoll kann z.B. ein Sender benachrichtigt werden, wenn der gewünschte Empfänger nicht erreichbar ist oder auch wenn ein IP-Paket schon zu lange im Netz unterwegs ist und deshalb vernichtet wurde. Auch das bekannte *ping* ist ein ICMP-Paket. Zumindest eine rudimentäre Unterstützung von ICMP muss durch die auf dem μC laufende Software gegeben sein.

2.5 User Datagram Protocol

Beim User Datagram Protocol (UDP), handelt es sich um eines der einfachsten ungefähr in der Transportschicht des OSI-Modells angesiedelten Protokolle. Im Wesentlichen erweitert es IP nur um die Möglichkeit, Paketen einen sog. Quell- und Zielport mit zu geben. Dies sind 16-Bit-Zahlen, die die das Paket behandelnden Prozesse auf den einzelnen Hosts identifizieren. Auf diese Art wird unterschieden, welcher Prozess sich um die gerade auf IP-Basis empfangenen Daten kümmern muss. Ebenso wie IP arbeitet UDP ohne Garantien über das Ankommen der Daten beim Empfänger. Auch Pakete, die zwar ihr Ziel erreichen, dann jedoch aufgrund eines Prüfsummenfehlers verworfen werden, werden nicht neu gesendet. Die Nutzlast ist somit verloren. Eine UDP-Unterstützung ist in der in dieser Arbeit verwendeten Software nur rudimentär enthalten. Wesentlich mehr Wert wurde auf die Unterstützung des im Folgenden kurz beschriebenen TCP-Protokolls gelegt.

2.6 Transmission Control Protocol

Für viele Anwendungen ist es von Nöten, eine Möglichkeit der garantierten Übersendung von Daten zur Verfügung zu haben. Diese Möglichkeit liefert das Transmission Control Protocol (TCP). TCP ist wie UDP in der Transportschicht des OSI-Modells angesiedelt, ist jedoch etwas komplexer aufgebaut. Es handelt sich hierbei im Kontrast zu IP oder auch UDP um ein verbindungsbehaftetes Protokoll, d.h. bevor eine Kommunikation zwischen zwei Hosts stattfindet, wird eine Verbindung mittels des sog. *Three-Way-Handshakes* initiiert. Danach können beide Host gleichberechtigt senden und empfangen. Um die Verbindung zu beenden kommt es abschließend mittels eines *Four-Way-Handshakes* zum Verbindungsabbau. Eine Verbindung kann also verschiedene Zustände besitzen. Pakete, die den Empfänger erreichen, werden von diesem quittiert – bleibt dieses "ACK" eine gewisse Zeit lang aus, so nimmt der Sender die Daten als verloren an und sendet das fragliche Paket erneut. Ein weiterer wichtiger Punkt ist, dass TCP mittels des sog. *Sliding Windows* die Datenmenge, die sich auf dem Weg zum Empfänger befinden kann, bevor den Sender eine Empfangsquittung für vorangegangene Pakete erreicht, regulieren kann. Hiermit wird die Geschwindigkeit der Übertragung optimiert, auch wenn das zur Übertragung genutzte Netzsystem die einzelnen Pakete nur langsam propagiert.

Wie auch UDP unterstützt TCP das Konzept der Portnummern. Ohne näher auf das Paketformat einzugehen sei noch erwähnt, dass sich die in den Paketen befindlichen Prüfsummen bei TCP auf das gesamte zu übertragende Paket beziehen. Damit sind auch Übertragungsfehler in der Payload erkennbar. Die Software des Controllers muss nicht nur in der Lage sein, TCP-Pakete korrekt zusammenzusetzen und zu interpretieren, sie muss auch den Zustandsautomaten von TCP implementieren. Sliding Window oder andere Spezialitäten des TCP-Protokolls müssen jedoch nicht unbedingt vollständig implementiert sein. Befindet sich beispielsweise immer nur ein Paket auf dem Weg zwischen Sender und Empfänger auf der Leitung, so schränkt dies zwar die Geschwindigkeit der Übertragung ein, verhindert aber nicht die ordnungsgemäße und gesicherte Kommunikation.

2.7 Protokolle der Anwendungsschicht

Auf TCP bzw. UDP aufsetzend existieren viele bekannte Protokolle, die einzelnen spezifischen Diensten zugeordnet sind. Als Beispiele seien an dieser Stelle nur das Hypertext Transfer Protocol (HTTP) sowie das File Transfer Protocol (FTP) genannt. Diese Protokolle müssen nachher anwendungsspezifisch für den Mikrocontroller implementiert werden. Dabei werden i.d.R.

ebenfalls wieder nur genau die benötigten Teile des jeweiligen Protokolls umgesetzt. Beispielhaft wird später ein Programm vorgestellt, welches Webseiten und andere Dateien auf mittels HTTP gestellte Anfragen ausliefert. Nur ein sehr begrenzter Teil des HTTP-Protokolls muss hierfür umgesetzt werden, die vollständige Unterstützung aller Möglichkeiten eines Protokolls der Anwendungsschicht würde in vielen Fällen die Leistungsfähigkeit des μC überschreiten.

3 Hardware

3.1 Die Familie der AVR-Mikrocontroller

Bei den Mikrocontrollern der AVR-Serie der Firma Atmel[7] handelt es sich um in RISC-Architektur aufgebaute 8-Bit-μCs. Das AVR-Design stammt ursprünglich von den Studenten Alf und Vegard der Universität Trondheim, die den AVR-Kern im Rahmen ihrer Diplomarbeit entwarfen und die Rechte anschließend an Atmel verkauften. Ein AVR-Controller verfügt über 32 general-purpose-Register[8] (kein gesonderter Akkumulator) und ist speziell auf die Ausführung von aus Hochsprachen wie C erstelltem Code optimiert. Die AVR-μCs unterteilen sich weiter in 3 Gruppen: ATmega, AT90 und ATtiny. Die einzelnen Mikrocontroller unterscheiden sich vor allem in der Anzahl zur Verfügung stehender IO-Ports, der Größe des RAMs, des EEPROMS und des Flash-Speichers. Wegen ihrer einfachen Handhabung und dem gerade im Verhältnis zur Leistung sehr günstigen Preis von jeweils nur wenigen Euro haben sie eine weite Verbreitung auch im privaten Bereich gefunden. Die meisten Controller dieser Serie sind in verschiedenen Bauformen (z.B. PDIP, TQFP usw.) und für verschiedene Betriebsspannungen (i.d.R. zw. 2,7 und 5,5V) verfügbar, so dass sich praktisch für jede denkbare Aufgabe ein passendes Modell finden lässt. Eine breite Palette an freier und kommerzieller Software erleichtert ferner die Entwicklung für diese μCs.
Für die hier vorliegende Aufgabe scheint im speziellen der ATmega32 geeignet zu sein. Dieser soll nun im Folgenden etwas näher beschrieben werden.

3.2 Mikrocontroller Atmel ATmega32

Um eine zügige Verarbeitung und Versendung von Daten über Ethernet zu ermöglichen, benötigt der zu verwendende Controller neben moderater Taktrate vor allem einen ausreichend groß dimensionierten Arbeitsspeicher. Komplette zu sendende bzw. empfangene Pakete müssen in diesem vorhaltbar sein. Außerdem sollte der Controller über genügend IO-Ports verfügen. Sind nach dem Anschluss ans Netzwerk alle Ports belegt, so ist der Aufbau weitgehend sinnlos, da dann keine weiteren Steueraufgaben mehr übernommen werden können. Ein weiterer Gesichtspunkt zur Wahl des μCs ist der Aufwand, der durch seine Handhabung beim Löten und Ähnlichem verursacht wird. Hier erscheint ein in der PDIP-Bauform verfügbarer Controller besonders geeignet.

[7] siehe http://www.atmel.com

[8] Auf 16 dieser Register sind jedoch einige wenige Befehle nicht anwendbar.

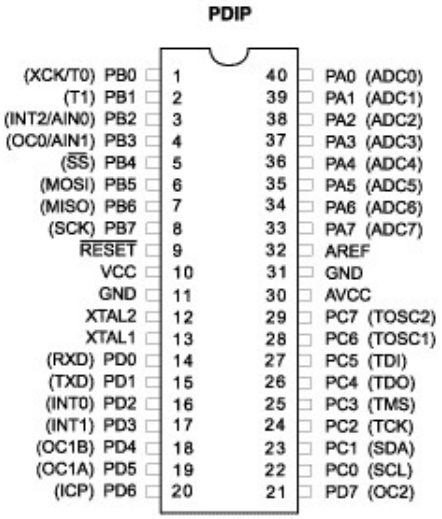


Abbildung 1: Übersicht über die Pinbelegung des ATmega32

Diese Anforderungen sind mit dem ATmega32 nahezu optimal zu erfüllen. Er ist der zum Zeitpunkt der Erstellung dieser Arbeit leistungsstärkste AVR-μC, der in PDIP-40-Bauform verfügbar ist. Mit 32KByte Flash-Speicher für ausführbaren Code, 2KByte SRAM und einer Taktrate von bis zu 16MHz bietet er die für den Versand von Daten per Ethernet benötigten Ressourcen. Er verfügt über vier acht Bit breite IO-Ports, von denen einer auch mittels eingebautem A/D-Wandler zum Einlesen von analogen Signalen genutzt werden kann. Ferner verfügt er über eine USART zur seriellen Kommunikation, beispielsweise über die RS232-Schnittstelle mit einem PC oder ähnlichem, und ist wie fast alle AVR-Controller in-circuit-programmierbar. Zum einfacheren Debuggen ist auch eine JTAG-Schnittstelle vorhanden, von der hier jedoch nicht weiter Gebrauch gemacht wird. TWI-Pins (für I^2C) sind ebenso vorhanden wie die Möglichkeit zur Ausgabe von pulsweiten-modulierten Signalen. Neben den internen Interrupts von 8- und 16-Bit-Timer, Reset, USART u.ä. können auch zwei externe Quellen zum Auslösen eines Interrupts genutzt werden. Der ATmega32 ist wie viele andere Modelle der ATmega-Baureihe neben der Standardvariante mit einer Betriebsspannung von 5V auch in einer L-Version, ausgelegt für eine Betriebsspannung von 3,3V, verfügbar. Da der verwendete NIC jedoch mit 5V betrieben wird, bietet sich die Verwendung des 5V-Modells an. Abbildung 1 gibt einen Überblick über die Pinbelegung des ATmega32, Abbildung 2 zeigt den Aufbau des ATmega32 als Blockbild.

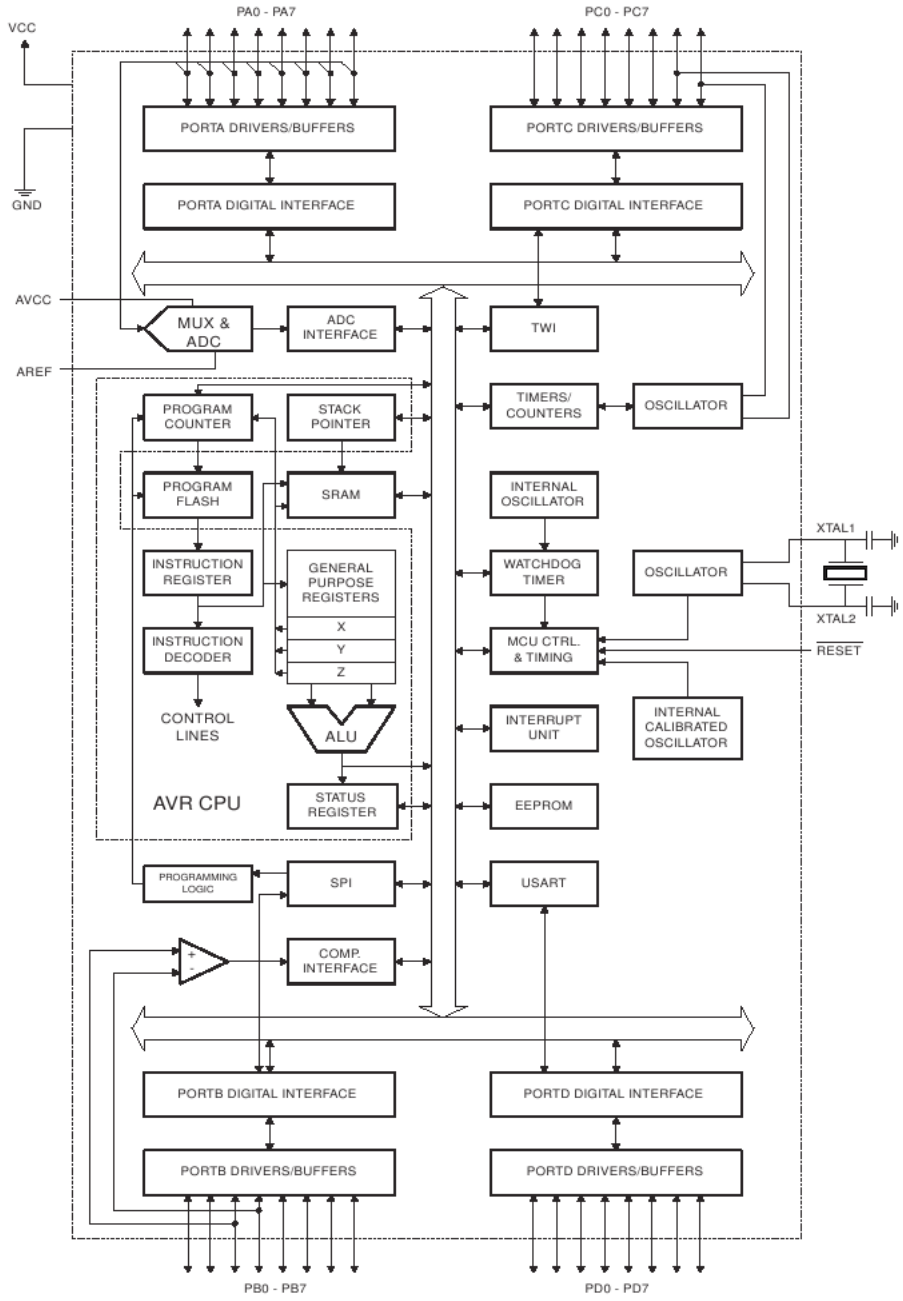


Abbildung 2: Aufbau des ATmega32 als Blockbild

3.3 Grundaufbau für die Arbeit mit dem ATmega32

Zunächst soll ein Blick auf den Aufbau geworfen werden, der generell notwendig ist, um einen Mikrocontroller der ATmega-Serie zu betreiben. Hierzu zählen neben dem Board, in welchem der Controller selbst nachher seinen Dienst verrichtet, auch eine Möglichkeit der seriellen Kommunikation mit einem PC über die RS232-Schnittstelle sowie ein Aufbau, mittels dessen der μC vom PC aus programmiert werden kann.

3.3.1 Basisplatine

Um einen ATmega-Mikrocontroller in Betrieb zu nehmen, sollte die Basisplatine neben Elementen für die Stromversorgung, einem Sockel zur Aufnahme des μCs selbst und den zur Takterzeugung nötigen Elementen[9], wie z.B. einem Quarz, auch noch geeignete Anschlüsse (z.B. Pinheader oder Buchsen) enthalten, auf die die Ports und Pins zur Programmierung geführt werden können. Auch ein Taster zum Rücksetzen (*Reset*) des ATmega sollte verbaut werden.

Zunächst muss der Controller mit geeigneter Spannung von 5V versorgt werden. Zur Spannungsstabilisierung kann beispielsweise ein Spannungsregler 7805 verbaut werden. Zur Entstörung sollte in unmittelbarer Nähe zum Controller ein 100nF Kondensator zwischen VCC und GND geschaltet werden. Dieser glättet evtl. in der Versorgungsspannung auftretende Schwankungen, die sonst die Funktion des Controllers negativ beeinflussen könnten. Ein solcher Entstörkondensator ist im Übrigen auch vor anderen eingesetzten ICs sinnvoll. Bei Einsatz des 7805 ist darauf zu achten, dass die auf Primärseite anliegende Spannung mindestens 6V beträgt, da sonst auf Sekundärseite die benötigten 5V nicht erreicht werden.

Der Taster für den manuellen Reset kann einfach zwischen den lowaktiven Reseteingang des ATmega und GND geschaltet werden, so dass beim Betätigen des Tasters RESET auf Masse gezogen wird.

Was den taktgebenden Aufbau angeht, so genügt ein zwischen XTAL1 und XTAL2 geschalteter Quarz der gewünschten Frequenz (hier 16MHz), wobei die Pins jeweils mit 22pF-Kondensatoren gegen Masse verbunden werden. Es ist später darauf zu achten, dass der Controller auch tatsächlich den externen Quarz als Taktquelle nutzt und nicht den internen Oszillator. Dazu sind die Fusebits des Controllers entsprechend zu setzen.

Programmiert wird der ATmega-Controller nachher in-circuit, hierfür

[9]Die Modelle der ATmega-Serie weisen zwar bereits einen internen 1MHz Oszillator auf, dieser ist jedoch zum einen zu ungenau um beispielsweise eine fehlerfreie Kommunikation über die USART zu gewährleisten und zum anderen schlichtweg sehr langsam.

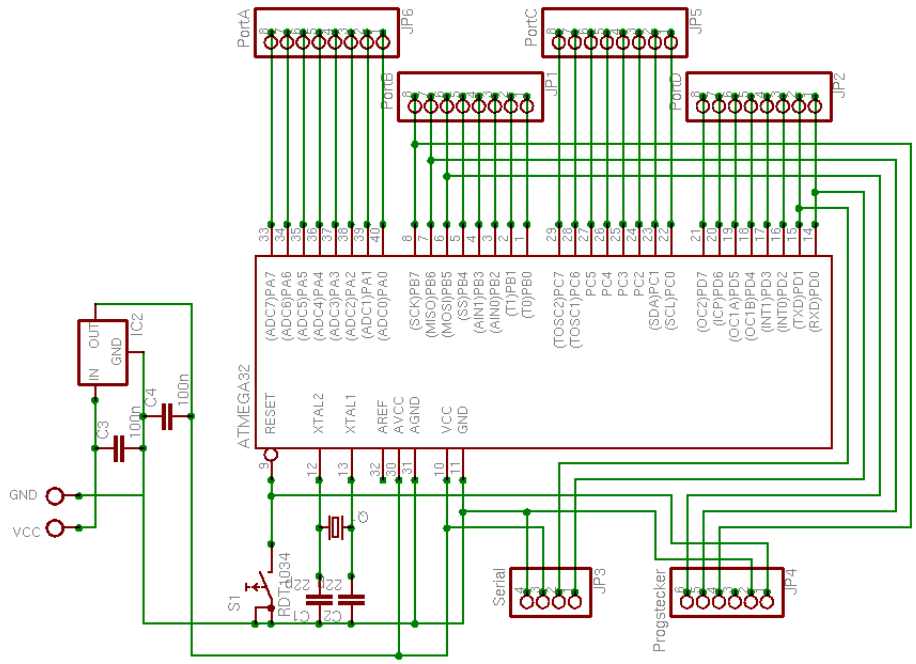


Abbildung 3: Schemazeichnung einer Basisplatine für den ATmega32

werden die Pins MISO, MOSI, SCK und RESET genutzt. Zusätzlich wird noch ein Masseausgleich mit dem PC benötigt. Für den Steckeraufbau wurde in dieser Arbeit ein verbreitetes Format gewählt, an welches sich nachher auch der Programmieradapter halten wird.

Ein Schema der Grundplatine kann Abbildung 3 entnommen werden.

3.3.2 Programmieradapter

Die Programmierung des Mikrocontrollers läuft im Folgenden über die parallele Schnittstelle des PCs. Hierzu wird ein 25-poliger SubD-Stecker entsprechend einer Belegung verlötet, die gemeinhin als “Steve Bolt’s Programmer” oder auch kurz *sp12* bekannt ist. Die gängigen Programmiertools auf PC-Seite (z.B. `avrdude`) können mit dieser Pinbelegung umgehen. Die Beschaltung kann Abbildung 4 entnommen werden. Sollte es beim Programmieren wiederholt zu Problemen kommen, so kann nach Erfahrung der Autoren ein Weglassen der Widerstände helfen.

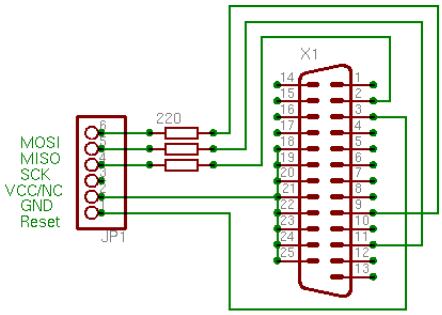


Abbildung 4: Schemazeichnung des *sp12*-Programmiersteckers

3.3.3 Adapter für die serielle Kommunikation

Dem ATmega-Controller ist es möglich, seriell mit einem PC gemäß dem RS232-Standard zu kommunizieren. Hierzu ist neben dem Anschluss eines 9-poligen SubD-Steckers noch eine Schaltung nötig, die die Signale mit TTL-Pegel (zw. 0 und 5V) des Controllers entsprechend auf die Pegel umsetzt, welche der PC an der Com-Port-Schnittstelle nutzt. Typischerweise wird also eine Pegelwandlung von TTL nach etwa -12 und +12V (und umgekehrt) benötigt. Eine Möglichkeit hierzu ist der Einsatz eines MAX232 Bausteins, der entsprechend der Schaltung in Abbildung 5 verbaut wird. Die verbauten Elkos weisen jeweils eine Kapazität von 1μF auf und wurden entsprechend der Beispielschaltungen aus dem Datenblatt zum MAX232N angeschlossen.

3.4 Realtek RTL8019AS

Nachdem nun eine funktionsfähige Grundschaltung für den ATmega32 gefertigt wurde, müssen Eigenschaften und Anschlussmöglichkeiten des zu verwendenden NICs untersucht werden. Der Realtek RTL8019AS ist ein NE2000-kompatibler in PQFP-100-Bauform verfügbarer Network Interface Controller, welcher zur Kommunikation mit Netzwerken nach Ethernet II und IEEE802.3 10Base5, 10Base2 und 10BaseT geeignet ist. Damit erlaubt er das Senden und Empfangen von Daten über das bekannte Ethernet mit maximal 10Mbit/s. Der NIC verfügt standardmäßig über 16KByte SRAM und wird hauptsächlich als Controller für ISA-Netzwerkkarten im PC-Bereich eingesetzt. Mit der Möglichkeit im 8-Bit-Modus zu operieren eignet er sich aber auch hervorragend für die Anbindung an einen Mikrocontroller. Neben der Anschlussmöglichkeit von vier Kontroll-LEDs bietet er auch die Option, Konfigurationsdaten aus einem 9346-EEPROM (64*16Bit) zu lesen. Der Chip benötigt eine 5V-Spannungsversorgung und wird per Quarz mit einer Tak-

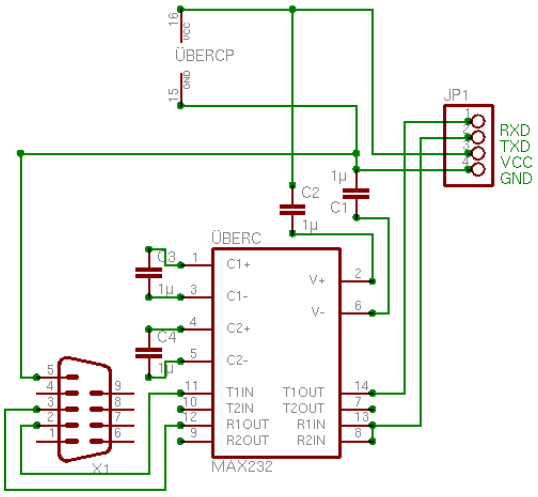


Abbildung 5: Schemazeichnung des Adapters mit Pegelwandlung für RS232

trate von 20MHz betrieben. Abbildung 6 zeigt die Pinbelegung des NIC. Wie schon erwähnt soll hier jedoch nicht der "nackte" Chip verbaut werden, sondern eine komplette ISA-Netzwerkkarte. Auf dieser sind zusätzlich zum NIC noch einige weitere benötigte Baugruppen vorhanden. Hierzu zählen z.B. Quarz, eine LED, RJ45-Buchse usw., kurzum alles, um den RTL8019AS an sich zu betreiben sowie ans Ethernet anzuschließen. Dies erleichtert den Aufbau erheblich, der Mikrocontroller übernimmt nun praktisch die Rolle des PCs mit ISA-Bus. Der NIC wird über den Standard-ISA-Slot mit dem Controller verbunden, wichtig für das Funktionieren des Aufbaus sind auf Hardwareseite also außer dem Anschließen von VCC und GND nur wenige Pins, mit denen der NIC an den ISA-Bus angebunden ist. Diese werden im Folgenden näher erläutert.

Pin 29 / IORB (Input) *I/O Host Read*. Der Host signalisiert dem NIC, dass er Daten lesen will, indem dieser Pin auf *low* gezogen wird. Dieser Pin muss direkt mit einem der Pins des ATmega32 verbunden werden.

Pin 30 / IOWB (Input) *I/O Host Write*. Der Host signalisiert dem NIC, dass er Daten schreiben will, indem dieser Pin auf *low* gezogen wird. Dieser Pin muss direkt mit einem der Pins des ATmega32 verbunden werden.

Pin 31 / SMEMRB (Input) *Host Memory Read Command*. Lowaktiver Eingang zum Zugriff auf das Bootrom. Wird nicht genutzt und kann

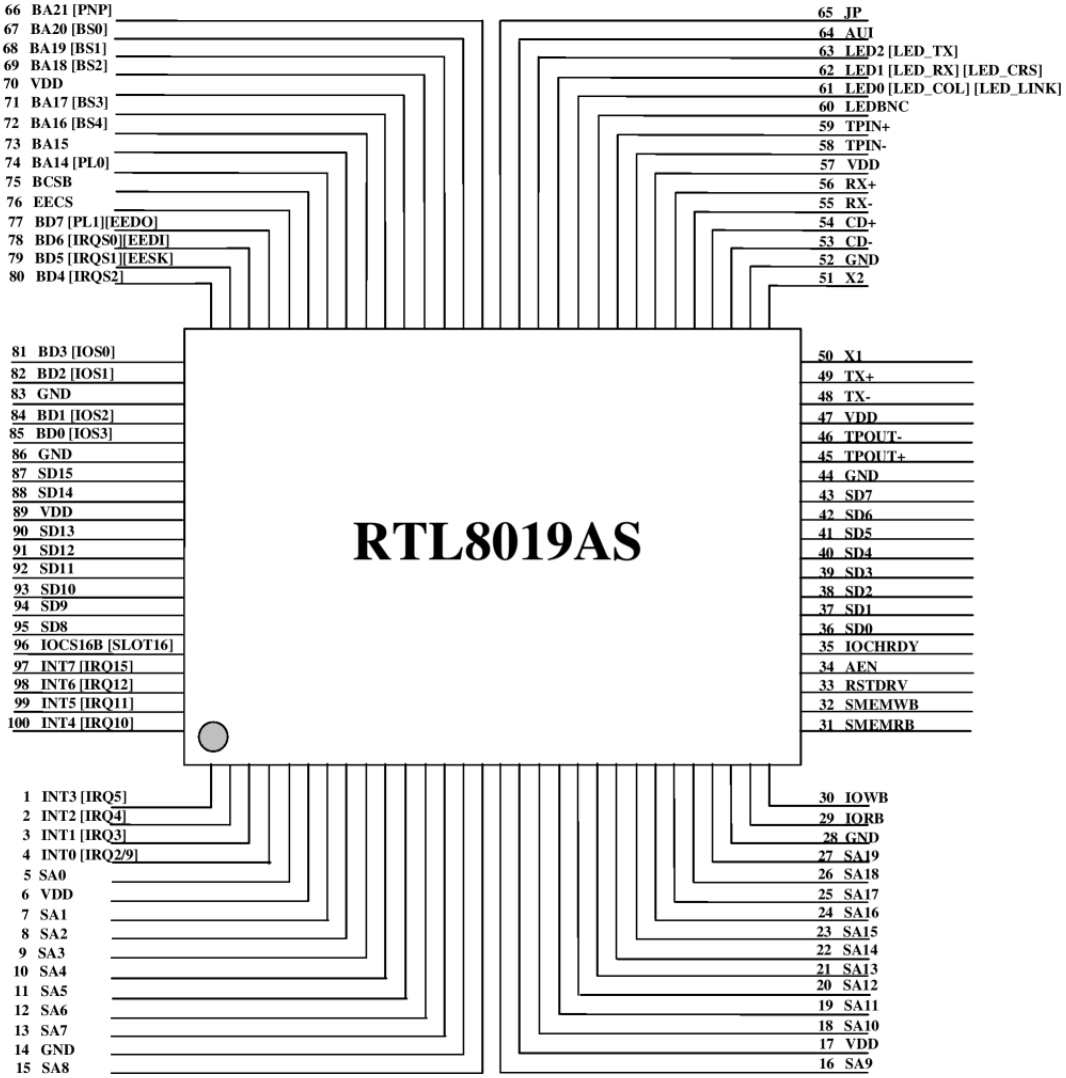


Abbildung 6: Pinbelegung des NIC

daher fest an VCC angeschlossen werden.

Pin 32 / SMEMWB (Input) *Host Memory Write Command.* Lowaktiver Eingang zum Schreiben von Flashspeicher (im Bootromsockel). Wird nicht genutzt und kann daher fest an VCC angeschlossen werden.

Pin 33 / RSTDRV (Input) *Reset.* Highaktives Reset-Signal. Dieser Pin muss direkt mit einem Pin des ATmega32 verbunden werden.

Pin 34 / AEN (Input) *Address Enable.* Der ISA-Standard sieht vor, dass AEN auf *low* gesetzt sein muss, während die Karte Eingabe- und Ausgabeoperationen vornimmt. Der Aufbau wird den eigentlich als Bus gedachten ISA-Anschluss jedoch nur als Port-Verbindung nutzen, d.h. der NIC muss sich keine I/O-Pins des Mikrocontrollers mit anderer Peripherie teilen. Damit sind Ein- und Ausgaben immer möglich, AEN kann statisch mit GND verbunden werden.

Pin 35 / IOCHRDY (Output) Fügt nach dem ISA-Standard Wartezyklen in die momentan in Abwicklung befindliche Lese- bzw. Schreiboperation des Host ein, wenn das Signal auf *low* gezogen wird. Dieses Signal wird von uIP nicht beachtet und kann somit einfach unverbunden gelassen werden. Dies spart u.a. auch I/O-Pins auf Seiten des Controllers.

Pin 97–100,1–4 / IRQ7–0 (Output) Diese Pins werden auf die Interrupts des ISA-Bus geschaltet. Da uIP keinen Interruptmechanismus unterstützt, bleiben diese Anschlüsse unbeschaltet.

Pin 96 / IOCS16B (Output) Eigentlich als Output-Signal genutzt, welches bei 16-Bit-Datentransfer auf *low* gesetzt wird, dient dieser Pin während der Einschaltphase (power-on reset) als Input-Pin, um festzustellen, ob sich die Karte in einem 16-Bit-ISA-Slot befindet. Liegt am Pin dann GND an, so wird von einem kürzeren 8-Bit-Slot ausgegangen und der NIC arbeitet im 8-Bit-Modus. Dies ist der für den 8-Bit-AVR geeignete Modus, daher wird im Aufbau IOCS16B fest mit GND verbunden.

Pin 87,88,90–95,43–36 / SD15–0 (Input/Output) *Host Data Bus.* Bei diesen Pins handelt es sich um die Datenpins, über welche der NIC mit dem μC kommuniziert. Da der NIC im 8-Bit-Modus betrieben wird, werden nur die unteren acht Bit direkt mit einem der Ports des Mikrocontrollers verbunden. Die restlichen Pins werden im Aufbau einfach auf Masse geschaltet.

Pin 27–18,16,15,13–7,5 / SA19–0 (Input) *Host Address Bus.* Mit Hilfe dieser Adressleitungen werden die einzelnen Register des RTL8019AS angesprochen. Die relevanten (NE2000-kompatiblen) Register werden in vier Speicherseiten (auch "Pages" oder "Kacheln" genannt) mit jeweils bis zu 16 Registern verwaltet. Die Selektion der einzelnen Pages erfolgt über die beiden niederwertigsten Bits im sog. *CR*-Register. Dieses ist über die jeweils erste Adresse (0x00) jeder Seite erreichbar. Eine vollständige Registerliste ist dem Datenblatt zum RTL8019AS (siehe [Harb]) entnehmbar. Die Register der vier Seiten sind somit alle durch Schreiben ihrer Seitennummer in die niederwertigen Bits von *CR* (immer Adresse 0x00) und anschließendem Anlegen der Registernummer innerhalb der Seite erreichbar. Da pro Seite maximal 16 Register vorhanden sind, erstrecken sich die Nummern der Register von 0x00 bis 0x0F, sind also mit vier Bit darstellbar. Darüber hinaus werden über die Adressen 0x10-0x17 noch der sog. *Remote DMA Port* und über 0x18-0x1F der sog. *Reset Port* angesprochen. Ersterer wird vom Device-Driver von uIP genutzt, daher müssen Adressen von fünf Bit Länge an den NIC anlegbar sein. Die Adressen werden alle relativ zu einem Offset (auch *I/O-Base* genannt) gerechnet. Die für den RTL8019AS nutzbaren I/O-Bases sind ebenfalls dem Datenblatt zu entnehmen. In Abhängigkeit der bei der verwendeten ISA-Karte eingestellten I/O-Base ist sind nun die Adress-Pins zu beschalten. Alle verwendbaren Basisadressen (z.B. 0x300, 0x320, 0x340 etc.) haben die Eigenschaft, dass die fünf niederwertigsten Bits in ihrer Binärdarstellung *0* sind. Somit können die Adressleitungen SA4-SA0 direkt mit einem der Ports des ATmega32 verbunden werden. Nun müssen noch die restlichen Pins so belegt werden, dass sich bei Anlegen von *0* bzw. GND an SA4-SA0 die Basisadresse ergibt. Bei der in dieser Arbeit verwendeten Karte wurde die Basisadresse mittels eines PCs auf 0x300 (binär 11 0000 0000) eingestellt. Somit ergibt sich für die Pins SA8 und SA9 eine Beschaltung von *1* (VCC). Die restlichen Pins werden mit GND verbunden.

Auf ein genaueres Eingehen auf die Funktionsweise des RTL8019AS kann in der weiteren Beschreibung verzichtet werden, da der Treiber von uIP dessen Funktionalität vor dem Programmierer verbirgt. Das API von uIP abstrahiert vollkommen von der darunter liegenden Hardware.

3.5 ISA-Bus-Connector

Die eben beschriebenen Pins sind alle, genormt nach dem ISA-Standard, auf den sog. ISA-Bus-Connector geführt. Eine Pinbelegung für den ISA-Slot, in

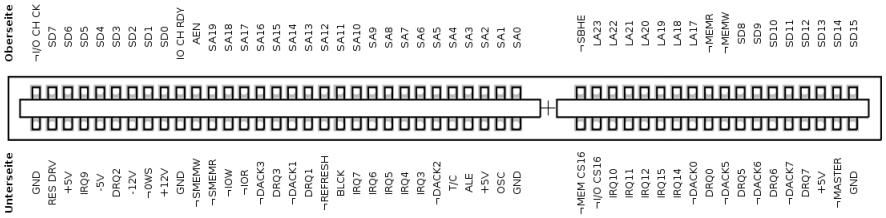


Abbildung 7: Pinbelegung des ISA-Slots

den die verwendete Netzwerkkarte später eingeschoben wird, kann Abbildung 7 entnommen werden. Der ISA-Slot weist ein Rastermaß von 2,54mm auf und ist somit bei der Platinenbestückung relativ einfach zu verarbeiten. Diejenigen Signalpins, welche kein Gegenstück beim RTL8019AS aufweisen, bleiben im Aufbau einfach offen. Theoretisch könnten auch alle im hinteren kurzen Teil liegenden Anschlüsse offen bleiben, da die Karte wie in einem 8-Bit-Slot betrieben werden soll. Da sich damit bei dem in dieser Arbeit verwendeten Kartenmodell jedoch nicht immer der gewünschte Effekt einstellte, werden auch die in der 16-Bit-Erweiterung befindlichen Pins mit GND verbunden.

3.6 Aufbau für Controller und Netzwerkkarte

An dieser Stelle kann das endgültige Design der Platine incl. ATmega-Controller und ISA-Slot für die Netzwerkkarte festgelegt werden. Wie eben gesehen, werden insgesamt zwei 8-Bit-I/O-Ports für den Anschluss der Netzwerkkarte benötigt. Ein Port dient als Datenport, auf den anderen Port werden die fünf Adressbit sowie die drei Steuersignale RSTDRV, IOWB und IORB gelegt. Der hier erstellte Aufbau verwendet PortA des μC als Datenport und die unteren fünf Bit von PortC (PinC4-PinC0) als Schnittstelle für Adressen. PinC5 wird auf IORB des NICs geschaltet, was bei der Beschriftung des ISA-Slots IOR entspricht. PinC6 geht auf IOWB des NICs, beim Schema des ISA-Slots aufgeführt als IOW. Das RESET-Signal wird durch PinC7 gegeben werden.

Dieser Aufbau hält PortB mit den zur Programmierung benötigten Pins sowie auch PortD, über welchen die serielle Kommunikation läuft, frei. Auch die Möglichkeit, den Controller mit externen Interrupts zu versorgen, bleibt erhalten, da die hierfür benötigten Pins ebenfalls innerhalb von PortD liegen.

Abbildung 8 zeigt schematisch den fertigen Aufbau incl. Beschaltung des ISA-Slots. Eine doppelseitig gelayoutete Version des Aufbaus, wie sie auch zum Ätzen der Platine für diese Arbeit genutzt wurde, kann Abbildung 17 im Anhang entnommen werden.

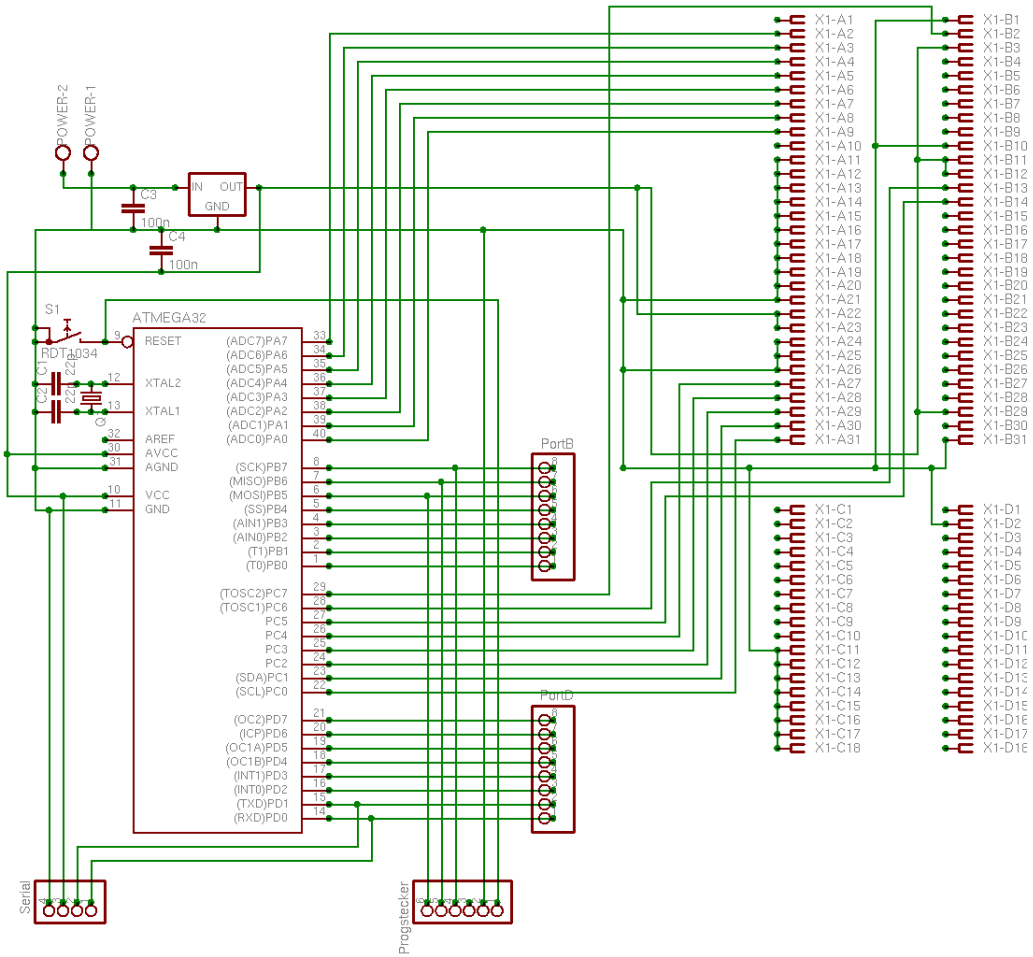


Abbildung 8: Schemazeichnung der Platine für Controller und Netzwerkkarte

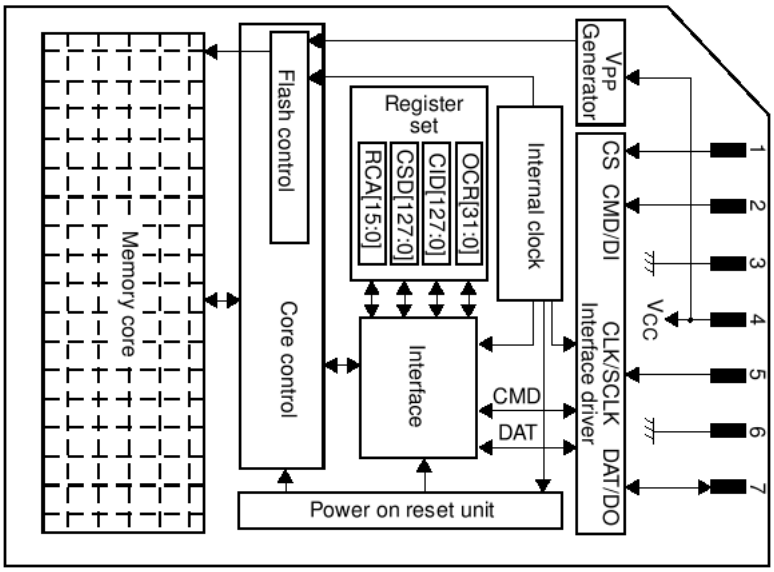


Abbildung 9: Blockbild einer MultiMediaCard

3.7 MultiMediaCard

Soll der Aufbau nachher beispielsweise als kleiner Webserver betrieben werden, so ergibt sich recht schnell ein Problem bezüglich des zur Verfügung stehenden Speicherplatzes für auszuliefernde Daten. Es wird also ein externes Medium benötigt, auf dem die zu versendenden Daten, wie z.B. HTML-Dateien, Bilder u.ä. gespeichert sind. Ein gut mit einem μC verwendbares Medium stellt eine MultiMediaCard (MMC) oder auch eine zu dieser abwärtskompatible Secure-Digital (SD) Karte dar. Bei diesen Kartentypen handelt es sich um leichte, kleine Flash-Speicher, welche heute in vielen Geräten, von tragbaren MP3-Player bis zu Digitalkameras, Verwendung finden. Sie sind kostengünstig und leicht in einer großen Auswahl an Speicherkapazitäten verfügbar. Bei der hier verwendeten Karte handelt es sich beispielsweise um eine 32MByte fassende MMC-Karte, die ursprünglich in einem Mobiltelefon Verwendung fand. Abbildung 9 zeigt das Blockschaltbild einer solchen Karte. Mit ihr kann mittels einer SPI-Schnittstelle kommuniziert werden. Problematisch ist zunächst die Tatsache, dass die Speicherkarte mit einer Spannung zwischen 2,7 und 3,6V betrieben werden muss, während der Mikrocontroller sowie die Netzwerkkarte ja mit 5V versorgt werden. Auch die Signaleingänge können ebenfalls nicht mit 5V beschaltet werden, sondern bedürfen eines Pegels im Bereich der Versorgungsspannung der Karte. Eine Versorgungsspannung von 3,6V kann einfach aus der vorhandenen 5V-

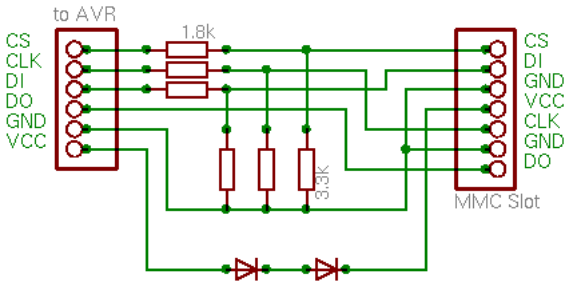


Abbildung 10: Schaltung für den Anschluss einer MultiMediaCard an den AVR-Controller

Spannung gewonnen werden, in dem zwei Dioden mit einer Diffusionsspannung von 0,7V verbaut werden. Die Ausgangspins des AVR-Controllers werden nun nicht unmittelbar, sondern jeweils über Spannungsteilerschaltungen (hier 1,8kOhm/3,3kOhm) an die Karte geführt, womit sich die benötigten Pegel ergeben. Der μC selbst benötigt auf Eingangsseite keine besonderen Vorkehrungen, die von der Karte erzeugte Spannung am Kontakt *DO* reicht aus, um dem Controller Highpegel zu signalisieren. Abbildung 10 zeigt die benötigte Schaltung. Als Slot zur Aufnahme der MMC-Karte kann ein Teil eines Steckplatzes mit 2,54mm Rasterabstand genutzt werden, wie er auch für die Netzwerkkarte zum Einsatz kommt.

Zum Betrieb der Karte existiert bereits frei verfügbarer Sourcecode aus diversen Quellen. Auch die Behandlung des normalerweise auf solchen Medien zum Einsatz kommenden FAT-Dateisystems kann gut mit frei verfügbarem Code erledigt werden. Da es in dieser Arbeit primär weder um die Medien MMC/SD-Card noch um die Interna des FAT-Dateisystems geht, wird an dieser Stelle von einem tieferen Einstieg in diese Themengebiete abgesehen. Zur Verwendung kam hier ein von Ulrich Radig geschriebener Treiber für die Speicherkarte sowie der dazugehörige Treiber für das FAT-Dateisystem. Beides ist auf Radigs Webseite ([Prod]) verfügbar. Kleine Modifikationen, die die Zusammenarbeit mit uIP begünstigen, werden später unter 4.2.3 beschrieben. Für nähere Informationen zu MultiMedia- wie auch SD-Karten empfiehlt sich ein Blick in die entsprechenden Datenblätter.

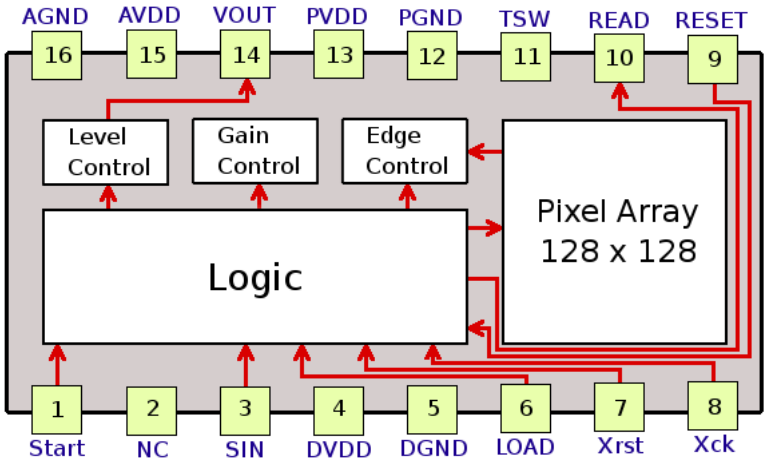


Abbildung 11: Blockbild des M64282FP

3.8 M64282FP Artificial Retina Chip

Beim M64282FP handelt es sich um einen 128x128 Pixel CMOS-Sensor, der simultan zur Aufnahme des Bildes dieses auch noch verarbeiten kann[10] – hierzu zählen u.a. diverse Modi zur Kantenerkennung. Die Versorgungsspannung des ICs muss 5V betragen, so dass der Anschluss an die Stromversorgung des μC kein Problem darstellt. Die Verbindung mit dem Mikrocontroller selbst erfolgt mit sechs Signalleitungen plus einer Leitung für die Daten der einzelnen erfassten Pixel, die der Chip dort als Analogsignal ausgibt[11]. Hierbei entsprechen z.B. 1,5V einem komplett schwarzen Pixel, 3,5V einem komplett weißen. Dazwischen liegende Werte wirken sich entsprechend linear auf die Ausgangsspannung aus. Somit ist der M64282FP in der Lage, ein Graustufenbild seiner Umgebung zu liefern. Abbildung 11 zeigt ein Blockbild des Bildsensors. Im Folgenden eine genauere Beschreibung der Pins:

Pin 4,13,15 / DVDD, AVDD1, AVDD2 Anschlüsse für die Spannungsversorgung von digitalen und analogen Elementen des Chips. Alle drei werden auf VCC geschaltet.

Pin 5,12,16 / DGND, AGND1, AGND2 Masseanschlüsse für digitale und analoge Elemente des Chips. Alle drei werden auf GND geschaltet.

[10] Daher rührt auch der Name *Artificial Retina*, da das menschliche Auge ebenfalls Aufnahme und Verarbeitung des Bildes zur selben Zeit beherrscht.

[11] Dieser einen Leitung auf Seite des Kamerachips entsprechen acht Leitung auf Mikrocontrollerseite, da dieser seine Daten von einem zwischengeschalteten A/D-Wandler bezieht.

Pin 2,11 / NC, TSW (n/a) Not Connected bzw. Reserved. Diese Pins werden nicht angeschlossen.

Pin 8 / Xck (Input) An diesem Pin muss ein extern erzeugtes Taktsignal für den Chip im TTL-Pegel anliegen. Dieses darf nicht höher als 500kHz sein. Zu dessen Generierung wird hier natürlich der μC genutzt, daher wird dieser Pin direkt mit einem der Pins des AVR verbunden.

Pin 7,9 / Xrst, RESET (Input) Diese lowaktiven Pins dienen dem Reset des Systems bzw. der Speichereinheit des Chips. Sie können zusammen auf einen Pin des Controllers geschaltet werden.

Pin 6 / LOAD (Input) Wird dieser Pin gesetzt, so signalisiert dies dem Chip den Abschluss einer Schreiboperation in eines der auf dem Chip vorhandenen Register. Dieser Pin wird direkt mit dem Controller verbunden (siehe auch SIN).

Pin 3 / SIN (Input) An diesem Pin werden Eingabedaten angelegt. Die mit Daten zu versorgenden acht Bit fassenden Register des M64282FP können allesamt mittels einer 3-Bit-Adresse angesprochen werden. Zur Datenübergabe wird zunächst die Adresse bitweise seriell an SIN angelegt, gefolgt vom ebenfalls serialisierten Byte, welches in das Register geschrieben werden soll. Hierbei wird jeweils das höchstwertige Bit zuerst übertragen. Abschließend wird das LOAD Signal gesetzt. Ein Schreibvorgang besteht also immer aus dem taktweisen seriellen Anlegen von elf Bit an SIN gefolgt von einem abschließenden setzten von LOAD. Dieser Pin wird daher direkt mit einem der Pins des μC verbunden.

Pin 1 / START (Input) Ein Anlegen von Highpegel an diesen Pin startet die Bildaufnahme. Nach Aufnahme und Übertragung des ersten Bildes generiert der Chip dieses Signal nach fünf Takten automatisch, so dass dieser Pin im Standardbetrieb tatsächlich nur einmal benötigt wird. Danach generiert die Kamera automatisch Bilddaten bis zum Abschalten der Stromversorgung oder dem nächsten Reset. Dieser Pin wird direkt mit einem der Pins der Controllers verbunden.

Pin 10 / READ (Output) Solange an READ Highpegel feststellbar ist, gibt der M64282FP taktweise Pixeldaten an Vout aus. Dieser Pin wird direkt mit einem der Pins des AVR verbunden.

Pin 14 / Vout (Output) Wie schon weiter oben angesprochen legt der Chip hier die Farbwerte der einzelnen aufgenommenen Pixel im Takt

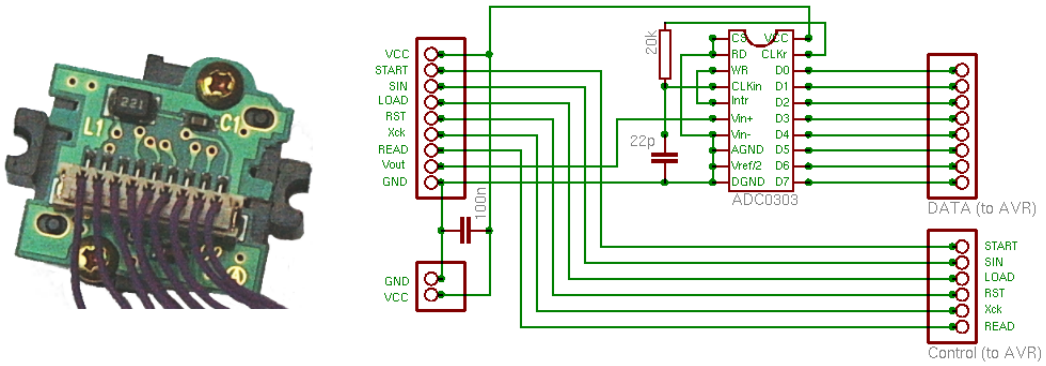


Abbildung 12: Links: Rückansicht der ausgebauten Platine mit Stecker. Die Belegung von links nach rechts lautet: GND (5,12,16), Vout (14), READ (10), Xck (8), RST (7,9), LOAD (6), SIN (3), START (1), VCC (4,13,15). Rechts: Anschlussschema für den M64282FP Retina-Chip an den AVR mit Hilfe eines ADC0803 A/D-Wandlers. Einzelheiten zum ADC0803 sind dessen Datenblatt zu entnehmen.

von Xck in analoger Form an. Hierbei wird zeilenweise vorgegangen. Dieser Pin wird für diese Arbeit mit einem ADC0803 8-Bit-A/D-Wandler verbunden, der seinerseits mit einem kompletten Port des Controllers verbunden ist. Der AD-Wandler wird im sog. *Free-Running-Mode* betrieben, d.h. er wandelt ständig mit der ihm maximal möglichen Geschwindigkeit.[12] Somit können die Bytewerte für die einzelnen Pixel einfach durch Lesen vom entsprechenden Port des μC festgestellt werden.

Der hier verwendete Chip befand sich ursprünglich in einer Kamera, welche als Zubehör zur Spielkonsole “Gameboy” von Nintendo verkauft wurde. Von der dort verbauten Platine lassen sich die notwendigen Signale mittels eines neunpoligen Steckers abgreifen. Abbildung 12 zeigt eine Rückansicht der ausgebauten Platine und gibt die Belegung des Steckers an, das Schema für den Anschluss des M64282FP an den AVR-Controller ist ebenfalls gegeben. Weitere Einzelheiten zum Betrieb des CCD-Chips folgen in Abschnitt 4.2.4, wo ein Treiber für den Betrieb des M64282FP entwickelt wird.

[12]Der externe Wandler wird verwendet, da er zum einen schneller ist als die im ATmega vorhandenen Wandler und zum anderen diese im verwendeten Aufbau bereits durch den NIC belegt sind.

4 Software

4.1 Toolchain zur Programmentwicklung

Hier nun ein Überblick über die für die Entwicklung von in C geschriebener Software für den AVR-Controller nötigen Werkzeuge. Generell werden – natürlich zusätzlich zu einem geeigneten Quelltexteditor – folgende Programmpakete benötigt:

avr-gcc Hierbei handelt es sich um eine speziell zur Erzeugung von auf dem AVR-Controller lauffähigem Code übersetzte Version des bekannten GCC-C-Compilers. Das Programmpaket kann z.B. unter [Tood] bezogen werden. Statt des GCC könnte auch ein anderer C-Compiler verwendet werden, uIP lässt sich beispielsweise auch mit dem ImageCraft-Compiler übersetzen. Verschiedene Compiler nutzen jedoch unter Umständen unterschiedliche Mechanismen zur Behandlung von Interrupts und ähnlichem, so dass hier bei anderen Compilern Vorsicht geboten ist.

avr-binutils Eine für die Erstellung von Binaries für den AVR-Controller verwendbare Version der GNU-binutils. Das Paket enthält unter anderem den Linker *avr-ld* sowie den Assembler *avr-as*. Die binutils können bei [Tooc] heruntergeladen werden.

avr-libc Version der libc, die speziell für die Benutzung mit dem AVR ausgelegt wurde. Hier finden sich unter anderem praktische Funktionen wie *printf*, aber auch Definitionen zum Umgang mit Ports und Pins des μC. Unter [Toob] kann der Quelltext der avr-libc bezogen werden. Leider nutzt uIP noch einige inzwischen aus der libc entfernten Befehle, so dass es mit einer aktuellen Version nicht mehr compilefähig ist. Die entsprechenden nötigen Anpassungen werden später in Abschnitt 4.2.2 beschrieben.

avrdude Um den mit den bereits erwähnten Tools erstellten Code in den Flashspeicher des Controllers zu laden, wird ein weiteres Werkzeug benötigt. Für diese Arbeit findet dazu das Programm *avrdude* Verwendung, welches unter [Toof] zu beziehen ist; weitere Informationen finden sich auch auf der Homepage des Entwicklers Brian S. Dean ([Tooa]).

Die Entwicklung für diese Arbeit fand komplett unter Linux statt. Können die nötigen Programmpakete nicht über das Installationssystem der Distribution eingespielt werden, so sind sie relativ leicht aus den Quellen zu erzeugen. Für Windows stellt das WinAVR-Projekt ([Tooe]) fertige Binaries der kompletten Toolchain zur Verfügung.

4.2 uIP

Bei dem bereits mehrfach erwähnten uIP handelt es sich um ein in C geschriebenes "Softwaregerüst", welches es ermöglicht, eigene Programme für den Mikrocontroller zu schreiben, die über TCP-Verbindungen Daten senden und empfangen.
Der Entwickler von uIP, Adam Dunkel, legte auf folgende Punkte besonderen Wert:

- uIP geht sehr sparsam mit Speicher um (dies gilt besonders für den RAM, aber auch für den Platz, den der Code im Flash-Speicher belegt).
- uIP unterstützt die wichtigsten Standards zur Netzwerkkommunikation – Dunkel hat sich hierbei an RFC 1122[13] orientiert, die die Anforderungen zusammenfasst, welche ein Host im Internet erfüllen muss. Dies bedeutet, dass theoretisch beinah beliebige Netzwerk-Software mit uIP realisiert werden kann – in der Praxis wird eher die Hardware die Limitation sein.

In der uIP-Dokumentation[14] erläutert Dunkel, dass er bei der Implementation der o.g. Anforderungen besonderes Gewicht auf jene gelegt hat, die die Host-zu-Host-Kommunikation betreffen, damit der Host, auf dem uIP läuft, nach außen hin "korrektes" TCP/IP verwendet. Um den Code klein und performant zu halten, hat er allerdings einige Anforderungen bewusst nicht erfüllt, die sich auf die Kommunikation zwischen Applikation und Protokollstack beziehen. Als Beispiel führt er Fehler an, die in der TCP-Schicht auftreten können: laut RFC müssen diese der Applikationsschicht mitgeteilt werden können – die RFC besagt allerdings auch, dass die meisten Applikationen diese Fehlermeldungen getrost ignorieren können. Dunkel hat hier den Code stark vereinfacht, indem er den geforderten Mechanismus weggelassen hat.

Die Speicherplatzersparnis erreicht Dunkel, indem er für jegliche eingehenden TCP-Pakete sowie die Header der ausgehenden Pakete genau einen zentralen Pufferspeicher einsetzt. Die Größe dieses Speichers kann zur Compilezeit eingestellt werden, jedoch muss hier abgewogen werden zwischen einem kleineren Puffer, der mehr RAM für andere Aufgaben übrig lässt und einem großen, der höhere Datendurchsatzraten bedeutet: je größer der Puffer, desto weniger müssen eingehende TCP-Pakete segmentiert werden, und desto mehr Daten können auf einmal empfangen werden. Der Programmierer muss sich überdies darum kümmern, dass empfangene Daten sofort verarbeitet oder an

[13] siehe [RFCa]
[14] siehe [Sofc]

eine andere Speicherstelle kopiert werden, da das nächste empfangene Paket wiederum im Puffer abgelegt wird und somit das vorige überschreibt. Zu sendende Daten hingegen müssen nicht im Puffer vorgehalten werden, da uIP diesen nur zum Zusammensetzen des jeweiligen TCP-Headers benötigt. In der Praxis ist es allerdings möglich (und je nach Größe des RAM sogar nötig), dass der Programmierer die Daten, die gesendet werden sollen, ebenfalls im Puffer ablegt.

Neben den bereits erwähnten Einschränkungen ist auch das API von uIP für jemanden, der bereits Erfahrungen mit BSD-Sockets[15] gesammelt hat, ungewöhnlich. In modernen Betriebssystemen kann man durchaus Sockets verwenden, die blockieren, also die Prozessausführung solange unterbrechen, bis eine bestimmte Menge an Daten gesendet bzw. empfangen wurde. Da besagte Betriebssysteme fast immer multitaskingfähig sind, können andere Prozesse weiterarbeiten, während solche, die für die Netzwerkkommunikation verantwortlich sind, blockieren. Auf einem Mikrocontroller jedoch, der üblicherweise ohne Scheduler auskommen muss und auf dem somit nur ein Prozess zur Zeit aktiv sein kann, würde allein das Blockieren der Sende- oder Empfangsaktivität schon ausreichen, um das ganze System stillzulegen.
Daher verfolgt Dunkel einen anderen, nämlich den "ereignisgesteuerten" Ansatz: Jede Aktion, die aus uIP heraus angestoßen wird, ist immer eine Reaktion auf irgendein Ereignis. Die auf uIP aufsetzende Applikation kann also z. B. auf ein eingehendes Datenpaket reagieren, indem sie zunächst prüft, ob es sich um eine Anfrage handelt (beispielsweise ein "GET" im Falle eines HTTP-Servers). Unmittelbar darauf kann sie dann die passende Antwort senden[16]. Dunkel hebt hervor, dass auch die gesendeten Daten in uIP anders als in üblichen TCP-Stacks behandelt werden: um Speicherplatz zu sparen, werden sie nicht gepuffert – sollte das zuletzt übertragene Paket verloren gehen und somit eine Neuübertragung nötig werden, so muss die Applikation die Daten neu generieren. Auch hierfür hat der Programmierer zu sorgen.

Damit die Ausführung des Benutzerprogramms schließlich nicht alleinig an den Netzwerkverkehr gebunden ist (z. B. wenn es eigenständig Verbindungen aufbauen soll), wird es auch noch periodisch angestoßen[17]. Der Programmierer muss also im Prinzip nur eine einzige Funktion implementieren, von

[15] Die aus BSD-Unix stammenden Sockets stellen heute die am weitesten verbreitete Softwareschnittstelle für Netzwerkverbindungen dar.

[16] Dieses Beispiel lässt sich noch weiter verfolgen: Ist die zu sendende Antwort größer als die maximale Paketgröße für ausgehende Pakete, so sendet die Applikation zunächst nur ein TCP-Paket. Erst wenn uIP ein TCP-Acknowledge erhalten hat, kann sie prüfen, zu welcher Verbindung dieses ACK gehört, und gegebenenfalls weitere Daten versenden, die für diese Verbindung noch anstehen.

[17] Die Standardeinstellung sieht hierfür eine Periode von 500ms vor.

der er weiß, dass sie jedes Mal ausgeführt wird, wenn irgendein Ereignis auftritt, spätestens jedoch nach Ablauf der Periode. Die Funktion selbst muss dann prüfen, weshalb sie aufgerufen wurde, und entsprechend reagieren.

4.2.1 Einfaches Anwendungsbeispiel

Folgendes an die uIP-Dokumentation angelehnte Beispiel zeigt eine sehr einfache Applikation:

```
1 void example_init(void) {
2    uip_listen(1234);
3 }
4 void example_app(void) {
5    if(uip_newdata() || uip_rexmit()) {
6       uip_send("ok\n", 3);
7    }
8 }
```

Die Funktion `example_init(void)` dient der Initialisierung des Programms, ein Aufruf der Funktion muss dementsprechend frühzeitig in uIPs `main(void)` Funktion erfolgen. `uip_listen(1234)` bedeutet, dass die die Applikation an TCP-Port 1234 auf eingehende Verbindungen warten soll. uIP prüft fortan für jedes eingehende TCP-Paket, an welchen Port es gerichtet ist, und ruft ggf. die Applikation auf; diese ist (von der Initialisierung abgesehen) komplett in `example_app(void)` implementiert. Kommt also ein an Port 1234 gerichtetes Paket an, so wird die Funktion aufgerufen und die Abfrage, ob neue Daten anliegen, liefert `true`[18]. Die Applikation reagiert, indem sie in `uip_send("ok\n", 3)` eine einfache Antwort aus 3 Bytes zurücksendet; Auf welche TCP-Verbindung sich der Sendebefehl bezieht weiß uIP durch den Kontext, aus dem heraus `example_app(void)` aufgerufen wurde. Schließlich ist die Beispielapplikation noch imstande, auf den Verlust eines Paketes zu reagieren: kommt als Antwort auf das “ok”-Paket kein ACK in einem gewissen Zeitraum, so geht uIP davon aus, dass das letzte Paket nochmals übertragen werden muss und ruft wiederum die Applikation auf. Wie bereits erwähnt, muss diese die zuletzt gesendeten Daten neu generieren, was allerdings im Beispiel besonders einfach ausfällt; `uip_rexmit()` liefert `true`, wenn das letzte Paket neu übertragen werden muss, und die Applikation sendet dasselbe “ok”, dass sie schon zuvor gesendet hat.

[18] Genaugenommen ist der Rückgabewert von `uip_newdata()` nur `true` für Pakete, die Nutzlast enthalten – Verbindungsauf- und abbau sowie ACK-Pakete des Clients werden von dieser Applikation also nicht beachtet.

4.2.2 Grundkonfiguration von uIP

Damit mit uIP Applikationen für unterschiedlichsten Hardwareplatformen entwickelt werden können, hat Dunkel viele von uIP benötigten Parameter variabel gehalten. So muss der Programmierer z.B. die Taktrate des μC sowie die Belegung der Pins, die mit dem NIC verbunden sind, als Konstanten definieren. Auch die Größe des Paketpuffers und ähnliche relevante Daten können angepasst werden.

Um uIP für den in Abschnitt 3.6 beschriebenen Aufbau von μC und Netzwerkkarte nutzbar zu machen, sollten folgende Manipulationen an uIP vorgenommen werden[19]:

- Die enthaltenen Dateien rtl8019.c und rtl8019.h müssen durch die ebenfalls auf der uIP-AVR Webseite ([Sofb]) verfügbaren, gleichnamigen Dateien ersetzt werden, welche Leseoperationen auf dem EEPROM der Netzwerkkarte ermöglichen. Damit kann später die zu verwendende MAC-Adresse von der Netzwerkkarte selbst bezogen werden.
  Leider ist zum Entstehungszeitpunkt dieser Arbeit ein Fehler in der Datei rtl8019.h enthalten, der noch behoben werden muss: in Zeile 159 muss hinter der Deklaration der Funktion `rtl8019_read_eeprom_word()` ein Semikolon hinzugefügt werden.

- In rtl8019.h müssen ab Zeile 205 die Definitionen der Ports bzw. Pins, die der Gerätetreiber verwendet, an die Hardware angepasst werden; im konkreten Fall bedeutet dies folgende Belegung:

```
// RTL8019 address port
#define RTL8019_ADDRESS_PORT        PORTC
#define RTL8019_ADDRESS_DDR         DDRC
#define RTL8019_ADDRESS_MASK        0x1F

// RTL8019 data port
#define RTL8019_DATA_PORT           PORTA
#define RTL8019_DATA_DDR            DDRA
#define RTL8019_DATA_PIN            PINA

// RTL8019 control port
#define RTL8019_CONTROL_PORT        PORTC
#define RTL8019_CONTROL_DDR         DDRC
#define RTL8019_CONTROL_READPIN     5
#define RTL8019_CONTROL_WRITEPIN    6

// ...

// RTL8019 RESET pin
#define RTL8019_RESET_PORT      PORTC
#define RTL8019_RESET_DDR       DDRC
```

[19] Dies bezieht sich auf uIP-AVR in der Version 0.90.0

```
#define RTL8019_RESET_PIN        7
```

Dann am Besten noch das darunter liegende `#define MAC_FROM_EEPROM` auskommentieren – es wird aus Konsistenzgründen später in der uipopt.h bei der Konfiguration der MAC-Adresse gesetzt.

- In der Datei nic.h muss in Zeile 25 angegeben werden, für welche Art von NIC uIP kompiliert werden soll:

```
#define NIC_CHOICE RTL8019
```

- In der Datei delay.c muss in Zeile 4 die Taktfrequenz des μC angegeben werden:

```
#define  F_CPU  16000000   /* The AVR clock frequency in Hertz */
```

- Das `#define` in Zeile 45 von main.c muss auskommentiert werden, da `F_CPU` bereits in delay.h definiert wurde.
- Wiederum in main.c in Zeile 53 darf beim Schreiben auf das Konfigurationsregister von Timer0 (TCCR0) nicht der Wert 7 gesetzt werden. Die richtige Einstellung für einen Vorteiler von 1024 liegt hier bei 5, wie in der Dokumentation zum ATmega32 nachzulesen ist:

```
outp( 5, TCCR0 ) ;
```

- Ebenfalls in main.c sollte der Konsistenz halber in Zeile 91 der Aufruf der Initialisierungsfunktion der Applikation über eine Konstante erfolgen (z.B. `UIP_APPINIT`), die später in app.h definiert werden kann.
- In der Datei compiler.h müssen ab Zeile 27 Definitionen eingefügt werden für Befehle, die neuere Versionen der libc nicht mehr enthalten, die aber in uIP noch Verwendung finden:

```
#define outp(val, reg)   (reg = val)
#define inp(reg)         (reg)
#define cbi(reg, bit)    (reg &= ~_BV(bit))
#define sbi(reg, bit)    (reg |= _BV(bit))
```

Zusätzlich sollte noch in den in dieser Datei stehenden `#include`-Anweisungen die Verzeichnistrenner von Backslash auf Slash geändert werden. So kann die Datei auch unter Linux kompiliert werden.

- In uipopt.h müssen ab Zeile 133 die IP-Adresse, die uIP verwenden soll, sowie Netzmaske und Default-Gateway (byteweise) gesetzt werden. Außerdem muss ca. bei Zeile 182 folgende Anweisung eingefügt werden, die dafür sorgt, dass die MAC-Adresse des NIC vom EEPROM der ISA-Karte bezogen wird:

```
#define MAC_FROM_EEPROM 1
```

Die anschließenden Definitionen, die eigentlich die MAC-Adresse festlegen, werden damit bedeutungslos.

Die Basiskonfiguration ist nun abgeschlossen; weitere anwendungsspezifische Einstellungen werden später im jeweiligen Kontext erläutert. Eine mittels diesen Änderungen erzeugte Version von uIP ist der Arbeit beigefügt.

4.2.3 Webserver

Während das Beispiel aus Abschnitt 4.2.1 eher der Veranschaulichung als dem praktischen Nutzen diente, ist es auch möglich, mit uIP einen HTTP-Server zu realisieren. Zwar existieren bereits Lösungen, um mit uIP einen solchen Server aufzusetzen, jedoch besitzen diese verschiedene Schwächen:

- Sie sind alle darauf angewiesen, dass die Daten, die sie ausliefern sollen, auf dem μC selbst abgelegt sind. Das bedeutet nicht nur, dass der Server sehr unflexibel ist, da der μC jedes Mal neu programmiert werden muss, wenn sich die Datenbasis ändert. Auch müssen die Dateien in ein spezielles Format gebracht werden, um sie im Quellcode einzubetten.
- Sie sind nicht (ausreichend) HTTP-konform: mit modernen Browsern, deren Anforderungen an den Server umfangreicher sind als sie es noch vor einigen Jahren waren, funktionieren sie nicht zusammen.

Um das erste Problem zu lösen, bot sich, wie schon in Abschnitt 3.7 beschrieben, der Einsatz eines wechselbaren Speichermediums an. Lesegeräte für MMC-Karten, die ein einfaches Editieren der Daten am PC ermöglichen, sind zudem kostengünstig zu haben und weit verbreitet. Um das zweite Problem zu lösen, war es nötig, sich näher mit HTTP bzw. der Arbeitsweise einiger Browser auseinanderzusetzen; einige Fallstricke waren hierbei:

Die "Content-Length" im HTTP-Header: Der Server muss, bevor er eine vom Client angeforderte Datei ausliefert, einen HTTP-Header versenden, in dem das Element "Content-Length" gesetzt und die Länge der auszuliefernden Datei enthalten ist. Die syntaktische Korrektheit des Headers ist Voraussetzung für eine erfolgreiche HTTP-Kommunikation.

Von Client gesendete Metadaten: Zusätzlich zur HTTP-GET-Anfrage, die für den Server bereits ausreicht, um eine Datei auszuliefern, senden moderne Browser eine Vielzahl von (optionalen) Metadaten[20] im HTTP-Header mit, in dem die Anfrage steht. Zwar kann der Server diese Metadaten ignorieren, jedoch führt die große Menge an Daten auch dazu, dass die Anfrage des Clients in mehrere TCP-Pakete aufgespalten wird; Wartet der Server nicht das letzte dieser Pakete ab, sondern fängt schon nach dem ersten Paket (das die eigentliche Anfrage enthält) an, die Datei zu versenden, so ignoriert der Browser dies. Da das Paket nichtsdestotrotz auf TCP-Ebene vom Client korrekt empfangen und bestätigt wurde, gibt es für den Server keine Veranlassung, es erneut zu übertragen. Nun befinden sich beide Parteien im Wartezustand, bis letztendlich bei einer Partei ein Timeout für einen Verbindungsabbruch sorgt. Es ist also zwingend notwendig, dass der Server mit HTTP-Anfragen umgehen kann, die aus mehreren TCP-Paketen bestehen.

Zur detaillierten Erläuterung der Funktionsweise des Servers folgt nun der Quellcode, in dem jedes Codestück durch den unmittelbar darüber stehenden Kommentar erläutert wird[21]. "include"-Anweisungen und ähnliches, das zwar im Quellcode steht, aber nicht unmittelbar für das Verständnis der Applikation notwendig ist, wurden hierbei der Übersichtlichkeit halber entfernt. Zunächst ein Codesegment, das die Deklarationen enthält, die sich in der Datei app.h befinden:

[20] z.B. "Accept-Language" für die vom Browser bevorzugte Sprache oder "Accept-Charset" für den bevorzugten Zeichensatz

[21] Diese Art der Dokumentierung wurde gewählt, um Code und Kommentar jeweils auf einen Blick verfügbar zu machen – umständliche Verweise auf Zeilennummern entfallen damit. Überdies ist somit auch der Quellcode quasi selbsterklärend.

```
1  /*
2      Dieses struct hält Statusinformationen einer HTTP-Verbindung fest.
3      Auf das struct kann zugegriffen werden über das uip_conn->appstate
        der TCP-Verbindung, zu der die HTTP-Verbindung gehört.
4  */
5  struct httpd_state {
6      /*
7          (Eindeutige) Verbindungsnummer.
8          Dieser Wert wird für unvollständige Dateitransfers verwendet,
           damit unterschieden werden kann, welche Daten zu welcher HTTP-
           Verbindung gehören.
9      */
10     unsigned char connumber;
11
12     /*
13         Beschreibt den aktuellen Verbindungsstatus.
14         state kann die Werte STATE_IDLE und STATE_GET annehmen:
           STATE_GET wird angenommen sobald eine HTTP GET Anfrage
           empfangen wurde, STATE_IDLE wird nach der erfolgreichen
           Dateiübertragung angenommen.
15     */
16     unsigned char state;
17
18     /*
19         Zeiger auf die Daten, die an den Client gesendet werden sollen
           .
20     */
21     unsigned char *msg;
22
23     /*
24         Erster Cluster der Datei, die gesendet werden soll.
25         Beim Zugriff auf das FAT Dateisystem über U. Radigs Bibliothek
            braucht man den ersten Cluster einer Datei, um diese lesen zu
            können.
26     */
27     unsigned int Cluster;
28
29     /*
30         Wie viele Bytes der Datei bereits gesendet worden sind.
31     */
32     unsigned long datasent;
33
34     /*
35         Wie viele Bytes noch übrig sind.
36     */
37     unsigned long dataleft;
38
39     /*
40         Der Name der Datei, die gesendet werden soll.
41         filename wird geschrieben, wenn die GET Anfrage gelesen wird
           und anschließend verwendet, um die Datei im FAT zu finden. In
           dem Moment, in dem der Anfangscluster bekannt ist (s.o.), wird
            filename nutzlos.
42     */
43     char filename[13];
44 };
```

Die Elemente des in der Header-Datei definierten structs `httpd_state` werden von der Webserver-Applikation verwendet, wie im Folgenden Ausschnitt der Datei app.c zu sehen ist:

```
1  /*
2      Die Werte, die das state-Element aus httpd_state annehmen kann.
3  */
4  #define STATE_IDLE    0
5  #define STATE_GET     1
6
7  /*
8      Dieses Makro liefert true, falls das TCP-Paket, das im Moment im
       Puffer ist, das Ende eines HTTP-Headers enthält;
9      Dies ist an den vier Bytes "\r\n\r\n" am Paketende zu erkennen.
10 */
11 #define HEADER_END (uip_appdata[uip_datalen()-4]=='\r' && uip_appdata[
   uip_datalen()-3]=='\n' && uip_appdata[uip_datalen()-2]=='\r' &&
   uip_appdata[uip_datalen()-1]=='\n')
12
13 /*
14     Dieses Makro liefert true, falls die ersten vier Bytes des Paketes
       , das gerade im Puffer liegt, "GET " sind - dies ist genau bei
       einer HTTP-GET-Anfrage der Fall.
15 */
16 #define REQUEST_IS_GET (uip_appdata[0]=='G' && uip_appdata[1]=='E' &&
   uip_appdata[2]=='T' && uip_appdata[3]==' ')
17
18 /*
19     Zeiger für den einfacheren Zugriff auf das httpd_state der
       momentan aktiven Verbindung.
20 */
21 struct httpd_state *hs;
22
23 /*
24     char-Array, das später verwendet wird, um darin den Header einer
       HTTP-Antwort zusammenzusetzen. Mit Ausnahme von "HTTP/1.1 " am
       Anfang wird der Inhalt jedesmal neu generiert, diese
       Initialisierung dient nur dem Zweck, ein ausreichend großes Array
       zu erzeugen.
25 */
26 char *header = "HTTP/1.1 ccc hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh\r\nContent
   -Length: 0123456789\r\n\r\n";
27
28 /*
29     Puffer, in dem die Dateisegmente abgelegt werden, die von der MMC-
       Karte geladen wurden.
30 */
31 unsigned char Buffer[512];
32
33 /*
34     Wird zum Speichern der Dateiattribute verwendet, die die FAT-
       Funktionen Search_File() und Read_Dir_Ent() zurückgeben.
35 */
36 unsigned char Dir_Attrib = 0;
37
38 /*
39     Gibt an, welche HTTP-Verbindung zuletzt auf den Speicherkarten-
       Puffer zugegriffen hat, bzw. welcher Verbindung er "gehört". Ist
       also für eine Verbindung (connumber aus httpd_state) =
```

```
    bufferOwnedBy, so kann diese Verbindung Daten aus dem Puffer
    direkt versenden, ohne vorher Daten von der MMC-Karte neu laden zu
     müssen.
*/
unsigned char bufferOwnedBy = 0;

/*
    Wird verwendet, um neuen Verbindungen (fortlaufend) eindeutige
    Verbindungsnummern zu geben.
*/
unsigned char connectionNumber = 1;

/*
    Konstruiert den HTTP-Header.
    Dieser beginnt in jedem Fall mit "HTTP/1.1 ", gefolgt von einem
    dreistelligen Statuscode (z.B. 200 für "Anfrage erfolgreich").
    Danach kommt eine textuelle Beschreibung des Status (z.B. "OK").
    Anschließend ist in HTTP noch die Angabe der Länge der Nutzlast
    nötig - dies geschieht mittels "Content-Length: ", gefolgt von der
     Länge der Nutzlast in Byte. Der Header wird mit einem
    Zeilenumbruch und einer Leerzeilen beendet ("\r\n\r\n"). Die
    Funktion erhält als Parameter die Statusinformationen als char-
    Array sowie die Nutzlast-Länge als long und gibt einen Zeiger auf
    den erzeugten Header zurück.
*/
char *http_constHeader(char *type, long len){

    uint8_t j=0;

    /*
        Kopiert den Inhalt von type in den Header, ab dessen neuntem
        Byte
    */
    while(type[j]){
        header[9+j]=type[j];
        j++;
    }

    /*
        Fügt die Zeichenkette "Content-Length: " hinzu
    */
    j+=9;
    header[j++]='\r';
    header[j++]='\n';
    header[j++]='C';
    header[j++]='o';
    header[j++]='n';
    header[j++]='t';
    header[j++]='e';
    header[j++]='n';
    header[j++]='t';
    header[j++]='-';
    header[j++]='L';
    header[j++]='e';
    header[j++]='n';
    header[j++]='g';
    header[j++]='t';
    header[j++]='h';
    header[j++]=':';
    header[j++]=' ';

    /*
```

```
88          Wandelt den long-Wert len in einen ASCII-Text um und trägt ihn
             in den Header ein
89      */
90      ultoa(len, &(header[j]), 10);
91
92      /*
93          Hängt hinter der Nutzlast-Länge noch den Header-Schluss an
94      */
95      strcat(header, "\r\n\r\n");
96
97      return header;
98  }
99
100
101 /*
102     Die eigentliche Applikation, die von uIP aufgerufen wird (z.B.
        beim Eintreten eines Ereignisses).
103 */
104 void httpd_app(void){
105
106         /*
107             hs soll auf den Applikationsstatus der momentan aktiven
                Verbindung zeigen
108         */
109         hs = (struct httpd_state *)(uip_conn->appstate);
110
111         /*
112             Zählvariable, um den Namen einer angeforderten Datei
                auszulesen
113         */
114         int i = 0;
115
116         /*
117             Neue Verbindung wurde aufgebaut.
118             Unmittelbar nach dem 3-Way-Handshake wird die Applikation
                aufgerufen
119         */
120         if( uip_connected() ){
121
122             /*
123                 Setzt den Verbindungsstatus vorläufig auf IDLE, bis
                    der Dateitransfer beginnt
124             */
125             hs->state = STATE_IDLE;
126
127             /*
128                 Noch sind keine Daten versendet worden
129             */
130             hs->datasent=0;
131
132             /*
133                 Gib dieser HTTP-Verbindung eine Nummer und setze
                    connectionNumber für die nächste Verbindung hoch
134             */
135             hs->connumber = connectionNumber++;
136         }
137
138         /*
139             In einer bestehenden Verbindung sind neue Daten empfangen
                worden.
140             Die Nutzlast des eingegangenen TCP-Paketes ist nun
                verfügbar
```

```
141         */
142         if( uip_newdata() ){
143
144             /*
145                 Prüft, ob das eingegangene Paket eine HTTP-GET-Anfrage
                    ist
146             */
147             if( REQUEST_IS_GET ) {
148
149                 /*
150                     Setzt den Applikationsstatus entsprechend
151                 */
152                 hs->state = STATE_GET;
153
154                 /*
155                     Aus der Anfrage wird der Name der angeforderten
                        Datei gelesen und in das filename-Element aus
                        httpd_state eingetragen
156                 */
157                 i=0;
158                 while ( i<12 && uip_appdata[i+5]!=' '){
159                     hs->filename[i] = uip_appdata[i+5];
160                     i++;
161                 }
162
163                 /*
164                     Das char-Array wird mit '\0' terminiert
165                 */
166                 hs->filename[i]='\0';
167             }
168
169             /*
170                 Liegt eine GET-Anfrage vor, so muss noch auf das Ende
                    der Anfrage gewartet werden (s.o.), bevor die
                    Dateiübertragung beginnen kann
171             */
172             if (hs->state==STATE_GET && HEADER_END){
173
174                 /*
175                     Cluster muss gesetzt werden, damit die Funktion
                        Search_File weiß, wo sie mit der Suche nach der
                        angeforderten Datei beginnen soll. Cluster wird
                        hier statisch auf 0 gesetzt, was dem
                        Wurzelverzeichnis des FAT-Dateisystems entspricht;
                         Dementsprechend kann der Webserver nicht mit
                        Dateianfragen umgehen, die sich auf
                        Unterverzeichnisse beziehen.
176                 */
177                 hs->Cluster = 0;
178
179                 /*
180                     Die Search_File-Funktion aus der FAT-Bibliothek
                        sucht hier im Wurzelverzeichnis nach der Datei mit
                         dem Namen, der in der Anfrage stand. Falls die
                        Datei gefunden wurde, liefert sie true und setzt
                        überdies Cluster auf den Anfangscluster der Datei.
                         Außerdem setzt sie dataleft auf die Dateigröße
                        und speichert in Dir_Attrib die Dateiattribute der
                         gefundenen Datei (die allerdings in dieser
                        Applikation keine weitere Verwendung finden)
181                 */
```

```
182                 if (Search_File(hs->filename,&(hs->Cluster),&(hs->
                    dataleft),&Dir_Attrib,Buffer) == 1){
183
184                     /*
185                         Falls die angeforderte Datei gefunden wurde,
                            kann für die Antwort an den Client der Header
                            generiert werden, der eine Erfolgsmeldung
                            sowie die Nutzlast-Länge enthält, die der
                            Client zu erwarten hat.
186                     */
187                     hs->msg = http_constHeader("200 OK", hs->dataleft)
                        ;
188
189                 } else { // Datei wurde nicht gefunden
190
191                     /*
192                         Setzt Cluster wieder auf 0, da die Search_File
                            -Funktion es umgesetzt hat
193                     */
194                     hs->Cluster = 0;
195
196                     /*
197                         Um dem Client eine adäquate Fehlermeldung
                            auszuliefern, auch wenn die geforderte Datei
                            nicht verfügbar ist, wird nach einer
                            speziellen "Fehler"-Seite mit Namen "404.HTM"
                            gesucht. Um das Funktionieren des Webservers
                            zu gewährleisten, sollte diese Datei also auf
                            der MMC-Karte vorhanden sein.
198                     */
199                     if (Search_File("404.HTM",&(hs->Cluster),&(hs->
                        dataleft),&Dir_Attrib,Buffer)==1){
200
201                         /*
202                             Falls die "Fehler"-Seite vorhanden ist,
                                wird diese an den Header angehängt, der
                                dem Client den Misserfolg meldet
203                         */
204                         hs->msg = http_constHeader("404 Not Found", hs
                            ->dataleft);
205
206                     } else { // Datei mit Namen "404.HTM" ist nicht
                        vorhanden
207
208                         /*
209                             Die Verbindung wird direkt abgebrochen
210                         */
211                         uip_abort();
212                         return;
213                     }
214                 }
215
216                 /*
217                     An diesem Punkt liegen Daten vor, die an den
                        Client gesendet werden können. Da bislang hs->msg
                        nur den selbsterzeugten Header enthält, der sicher
                         '\0'-terminiert ist, kann als Länge der zu
                        übertragenden Daten einfach strlen(hs->msg)
                        verwendet werden
218                 */
219                 uip_send(hs->msg, strlen(hs->msg));
220
```

```
221             } // Ende if (hs->state==STATE_GET && HEADER_END)
222
223         } // Ende if( uip_newdata() )
224
225
226         /*
227             Ein an den Client gesendetes Paket wurde mit TCP-
                Acknowledge bestätigt. Das gesendete Paket kann der HTTP-
                Header gewesen sein, ein Teil aus der Datei, oder genau
                das Dateiende. Anhängig davon wird unterschiedlich auf das
                 ACK reagiert
228         */
229         if( uip_acked() ){
230
231             /*
232                 Beim Senden des letzten Paketes wurden bereits so
                    viele Daten übertragen, wie von der Datei noch übrig
                    waren, und das letzte Paket war nicht der Header. Dies
                     bedeutet, dass die Datei fertig übertragen wurde und
                    die Verbindung beendet werden kann.
233             */
234             if(hs->dataleft <= uip_conn->len && ((int)hs->msg != (int)
                header) ) {
235
236                 /*
237                     Setzt den Status der HTTP-Verbindung entsprechend.
                         Auf HTTP-Ebene weiß der Client, dass die
                        Verbindung zu Ende ist, da die "Content-Length"
                        erreicht wurde.
238                 */
239                 hs->state = STATE_IDLE;
240
241                 /*
242                     Schließt die TCP-Verbindung
243                 */
244                 uip_close();
245                 return;
246             }
247
248             /*
249                 Falls die zuletzt gesendete Nachricht nicht der Header
                     war, muss es ein Teil der Datei gewesen sein. In
                    diesem Fall müssen die Elemente datasent und dataleft
                    aus httpd_state angepasst werden.
250             */
251             if ( (int)hs->msg != (int)header ){
252                 hs->datasent += uip_conn->len;
253                 hs->dataleft -= uip_conn->len;
254             }
255
256             /*
257                 Wenn dieser Punkt des Codes erreicht wurde, muss die
                    Dateiübertragung noch im Gange sein - es muss also das
                     nächste Teilstück der Datei versendet werden.
258                 Hier wird geprüft, ob der Puffer zum Zwischenspeichern
                     der (512 Bytes großen) Dateiteile im Moment einer
                    anderen HTTP-Verbindung gehört oder ob beim letzten
                    Senden der Pufferinhalt "aufgebraucht" wurde. In
                    beiden Fällen muss ein neuer Dateiteil von der MMC-
                    Karte geladen werden.
259             */
```

```
260             if ((bufferOwnedBy!=hs->connumber) || (hs->datasent
                %512==0)){
261
262                 /*
263                     Ergreift Besitzt vom Puffer
264                 */
265                 bufferOwnedBy = hs->connumber;
266
267                 /*
268                     Liest die Datei von der MMC-Karte in den Puffer.
269                     Der dritte Parameter gibt hierbei an, welcher
                        Dateiblock ausgelesen werden soll: War der Puffer
                        vorher im Besitz einer anderen Verbindung, so
                        enthält der zu ladende Block auch bereits
                        gesendete Daten, d.h. datasent % 512 > 0. Durch
                        die Ganzzahldivision wird dennoch der korrekte
                        Block (der gleiche wie zuvor) angefordert. Ist der
                         Puffer zuvor bereits komplett gesendet worden, so
                         ist hs->datasent/512 um eins größer als beim
                        letzten Senden, und es wird der nachfolgende
                        Dateiblock angefordert.
270                 */
271                 Read_File (hs->Cluster,Buffer,hs->datasent/512);
272
273             }
274
275             /*
276                 Um beim anstehenden Senden von Daten die
                    Anfangsadresse der Nachricht zu kennen, wird der msg-
                    Zeiger aus httpd_state auf den Puffer gerichtet und "
                    weitergeschoben" um die Zahl Bytes, die aus diesem
                    Dateisegment evtl. schon übertragen wurden.
277             */
278             hs->msg = Buffer + hs->datasent % 512;
279
280             /*
281                 Falls dies der letzte Dateiblock ist, kann es sein,
                    dass das Dateiende mitten im Puffer ist. Um nicht mehr
                     Daten auszusenden, als zur Datei gehören, Wird dieser
                     Fall hier gesondert behandelt (512-(hs->datasent%512)
                     ist hierbei genau die Anzahl Bytes, die momentan
                    zwischen msg-Zeiger und Pufferende liegen).
282             */
283             if ( hs->dataleft < 512-(hs->datasent%512) ){
284
285                 /*
286                     Überträgt nur so viele Bytes aus dem Puffer, wie
                        noch zur Datei gehören
287                 */
288                 uip_send(hs->msg, hs->dataleft);
289
290             } else { // Dateiende noch nicht erreicht
291
292                 /*
293                     Überträgt alles bis zum Ende des Puffers
294                 */
295                 uip_send(hs->msg, 512-(hs->datasent%512));
296             }
297
298         } // Ende if( uip_acked() )
299
```

```
300
301         /*
302             Die Applikation wird von uIP zur Neuübertragung
                aufgefordert.
303             Dieser Fall ist analog zu uip_acked() mit dem Unterschied,
                 dass hier weder geprüft werden muss, ob die Übertragung
                zu Ende ist, noch müssen datasent bzw. dataleft geändert
                werden (beides wurde ja schon in uip_acked() erledigt).
304         */
305         if(uip_rexmit()){
306
307             /*
308                 Zunächst wird geprüft, ob das letzte versendete Paket
                    der Header war (dieser wird ggf. neu gesendet).
309             */
310             if ((int)hs->msg == (int)header){
311                 uip_send(hs->msg, strlen(hs->msg));
312                 return;
313             }
314
315             if ((bufferOwnedBy!=hs->connumber)){
316                 bufferOwnedBy = hs->connumber;
317                 Read_File (hs->Cluster,Buffer,hs->datasent/512);
318             }
319
320             hs->msg = Buffer + hs->datasent % 512;
321
322             if ( hs->dataleft < 512-(hs->datasent%512) ){
323                 uip_send(hs->msg, hs->dataleft);
324             } else {
325                 uip_send(hs->msg, 512-(hs->datasent%512));
326             }
327
328         } // Ende if(uip_rexmit())
329
330 } // Ende httpd_app(void)
331
332
333 /*
334     Die Initialisierungfunktion für die Applikation.
335     Diese Funktion wird einmal kurz nach dem Start des uC aufgerufen
        und dient hier im Wesentlichen dazu, sicherzustellen, dass der
        Zugriff auf die MMC-Karte funktioniert. Darüber hinaus wird uIP
        noch angewiesen, auf eingehende Verbindungen an TCP-Port 80 zu
        warten.
336 */
337 void httpd_init(void) {
338
339     /*
340         Initialisierung der MMC-Karte (sh. FAT/MMC-Bibliothek).
341         Da die Hardware eine kurze Anlaufzeit benötigen kann, bevor
            der Zugriff gelingt, wird mmc_init() einfach solange
            wiederholt, bis es 0 (= Erfolg) zurückliefert. Solange die MMC
            -Karte sich nicht initialisieren lässt, kann auch der
            Webserver nicht korrekt arbeiten, und der uC ist hier
            korrekterweise wartend blockiert.
342     */
343     while ( mmc_init() !=0){
344         // Statusausgabe
345     }
346
347     /*
```

```
348         Von der Bibliothek benötigte Funktion, um korrekte Zugriffe
            auf die Karte zu gewährleisten.
349         In der Original-Bibliothek von U. Radig ist diese Funktion
            parameterlos; sie legt einen lokal verwendeten 512-Byte-Puffer
             an, auf dem sie arbeitet. Da diese große Speichermenge auf
            dem Webserver nicht zur Verfügung steht, wurde die Funktion
            dahingehend verändert, dass sie nun den Puffer als Parameter
            erhält. Sie wird hier mit dem globalen Lesepuffer aufgerufen,
            der sowieso noch ungenutzt ist.
350         Die Initialisierung der MMC-Karte ist mit diesem
            Funktionsaufruf abgeschlossen.
351     */
352     Cluster_Data_Store(Buffer);
353
354     /*
355         Weist uIP an, fortan für alle eingehenden Pakete, die an Port
            80 gerichtet sind, die Applikation aufzurufen
356     */
357     uip_listen(HTONS(80));
358
359 }
```

Wie in Abschnitt 4.2.2 erwähnt, kommen in uipopt.h noch einige Einstellung zur Grundkonfiguration hinzu, um den Webserver funktionsfähig zu machen:

- uIP muss nicht in der Lage sein, aktiv Verbindungen aufzubauen, daher kann `UIP_ACTIVE_OPEN` auf 0 gesetzt werden
- Da pro Verbindung, die uIP verwaltet, über 60 Bytes Speicher gebraucht werden, empfiehlt es sich, die maximale Verbindungsanzahl `UIP_CONNS` von standardmäßig 10 auf einen kleinen Wert zu reduzieren – ein erfahrungsgemäß guter Wert ist hier 2.
- Speicher lässt sich ebenfalls sparen durch das Reduzieren der Anzahl der Ports `UIP_LISTENPORTS`, auf denen uIP auf eingehende Verbindungen wartet. Der Wert 1 ist hier ausreichend – der Webserver läuft ohnehin nur auf Port 80.
- TCP-Pakete, bei denen das URG-Flag[22] gesetzt ist, könnten sowieso nicht anders behandelt werden als solche, bei denen es nicht gesetzt ist, daher kann `UIP_URGDATA` auf 0 gesetzt werden, was die Speichernutzung reduziert.

[22]Ein gesetztes “Urgent”-Flag bedeutet, dass der empfangende TCP-Stack diese Pakete direkt an die darüberliegende Applikationsschicht weitergeben soll, um sie bevorzugt behandeln zu können.

- Ebenfalls nicht benötigt wird das Führen einer Statistik über die von uIP durchgeführten Aktionen, `UIP_STATISTICS` wird daher auf 0 gesetzt.

- Wie in Abschnitt 4.2 erwähnt, ist der Puffer in erster Linie zum Empfangen von eingehenden Paketen nötig, außerdem werden in ihm die Header der ausgehenden Pakete zusammengesetzt. Bei einer Applikation wie dem Webserver könnte dies dazu verleiten, den Puffer klein zu halten, da die eingehenden Pakete bei der HTTP-Verbindung volumenmäßig nur einen sehr geringen Anteil an der Kommunikation haben. Eine geringe Puffergröße könnte so mehr RAM für andere Zwecke lassen (z.B. um mehrere Dateiblöcke auf einmal von der Karte zu laden). Allerdings wird in uIP die Puffergröße auch dazu verwendet, die MSS[23] zu berechnen, die ihrerseits wieder zu Rate gezogen wird, bevor ein Paket versendet wird – auch ein ausgehendes Paket darf nicht größer als die MSS sein. Folglich ist es eine gute Idee, die Puffergröße `UIP_BUFSIZE` größer als (Größe eines Dateiblocks + TCP-Overhead + Link-Layer-Overhead)[24] zu wählen, damit ein kompletter Dateiblock auf einmal versendet werden kann.

Damit uIP beim Kompilieren den Namen der Applikationsfunktion bzw. den der Initialisierungfunktion kennt[25], müssen die Konstanten `UIP_APPCALL` und `UIP_APPINIT` als die entsprechenden Funktionsnamen definiert werden. Dies sind hier `httpd_app` bzw. `httpd_init`; die Definitionen müssen in der Datei app.h stehen.

4.2.4 Server mit Kamera-Anbindung

Um auch die in Abschnitt 3.8 vorgestellte "Artificial Retina" in Verbindung mit uIP nutzen zu können, mussten zwei Teilaufgaben bewältigt werden: Ein Gerätetreiber sowie eine uIP-Applikation, die unter Verwendung des Treibers die Bilddaten bezieht und per TCP versendet, mussten geschrieben werden – beide Lösungen werden im Folgenden vorgestellt. Während der Treiber nur nach intensivem Studium des Datenblattes und durch ebenso intensives Testen zu realisieren war, fiel die Applikation recht einfach aus. Dies liegt zum einen daran, dass die Kamera-Funktionalität optimal im Treiber gekapselt ist und von der Applikation in wenigen Funktionsaufrufen angesteuert werden

[23] Maximum Segment Size: die maximale Größe, die ein TCP-Paket auf einer konkreten Leitung von Endpunkt zu Endpunkt haben darf.

[24] 512 + 40 + 14 = 566

[25] uIP selbst sieht inkonsistenterweise nur eine Definition für ersteren vor, daher wurde in Abschnitt 4.2.2 angeraten, eine zusätzliche Konstante einzuführen.

kann. Zum anderen musste kein bereits existierendes Protokoll der Applikationsschicht (wie z.B. HTTP) eingehalten werden, sondern es konnte ein eigenes (sehr einfaches) Protokoll auf TCP aufgesetzt werden:
Der Client sendet zwei Bytes an den Server, die die Belichtungszeit enthalten[26], mit der die Aufnahme erfolgen soll. Der Server sendet dieselben zwei Bytes an den Client zurück (der sie höchstens für Debug-Zwecke braucht), und beginnt danach, das Bild zu versenden.

Um die Bilddaten von Seite des Clients aus anfordern und auch empfangen zu können, musste eine Software erstellt werden, die diese Aufgaben erledigt. Da hierzu Java eingesetzt wurde, ist die Client-Software nicht nur auf unterschiedlichen Plattformen lauffähig, auch konnte durch das (gemessen an C) sehr hohe Abstraktionsniveau von Java das Programm problemlos um die Möglichkeiten erweitert werden, das aufgenommenen Bild direkt (vergrößert) anzuzeigen oder im PNG-Bildformat abzuspeichern. Da die Client-Software zwar gebraucht wird, um die aufgenommenen Bilder zu empfangen und anzuzeigen, selbst jedoch nicht direkt zum Kern dieser Arbeit gehört, ist der Quellcode mit erklärenden Kommentaren im Anhang zu finden.

Besonders zu beachten gilt bei dieser Softwarelösung die Art, wie mit dem sehr begrenzten Speicherplatz des μC umgegangen wird; Da die Größe eines einzelnen Bildes 128 * 123 = 15744 Bytes beträgt, ist es offensichtlich unmöglich, das komplette Bild im Speicher des Controllers abzulegen, um es dann von dort aus an den Client zu senden. Die naheliegende Lösung wäre hier, einfach eine geeignete Menge Pixel von der Kamera zu lesen, diese Pixel an den Client zu senden und erst danach wieder mit dem Einlesen von Bilddaten fortzufahren. Es stellte sich jedoch heraus, dass die durch das Senden entstehende Verzögerung den Betrieb der Kamera empfindlich stört und die nachfolgenden Bilddaten unbrauchbar macht. Die einzig praktikable Lösung ist daher die, für jedes neu an den Client zu sendende Paket die Kamera von Neuem zu initialisieren und eine neue Aufnahme zu machen. Soll ein Paket übertragen werden, das weiter hinten im Bild liegende Pixel enthält, so werden zunächst die Daten von der Kamera "im Schnelldurchlauf" gelesen, ohne sie zu verwerten. Beim relevanten Bildausschnitt werden dann die Bilddaten zusätzlich gespeichert, danach wird das Einlesen abgebrochen. Die so gewonnenen Daten können nun an den Client gesendet werden. Aus dieser Lösung resultieren jedoch auch Nachteile, welche in Abschnitt 5.3 erläutert werden.

Im Folgenden wird zunächst die Funktionsweise des Kameratreibers anhand der bekannten Paarung von Code und Kommentaren veranschaulicht:

[26]Genaugenommen enthalten sie die Belichtungszeit * 1000 als Ganzzahlwert, um Problemen bei der Konvertierung von Fließkommazahlen aus dem Weg zu gehen.

```
1  /*
2      Definiert, welche Ports für die Kommunikation mit der Kamera
       verwendet werden sollen.
3      EYE_DATA muss hierbei an der A/D-Wandler angeschlossen sein,
       EYE_CONTROL direkt an die Kamera.
4  */
5  #define EYE_DATA PINB
6  #define EYE_DATA_DDR DDRB
7  #define EYE_CONTROL PORTD
8  #define EYE_CONTROL_IN PIND
9  #define EYE_CONTROL_DDR DDRD
10
11 /*
12     Definiert, welches Bit am Port EYE_CONTROL welche Aufgabe hat.
13     Ein Bit des Ports EYE_CONTROL bleibt hierbei ungenutzt.
14 */
15 #define EYE_START 2
16 #define EYE_SIN 3
17 #define EYE_LOAD 4
18 #define EYE_RST 5
19 #define EYE_XCK 6
20 #define EYE_READ 7
21
22 /*
23     Definiert, welche Bits von Port EYE_CONTROL (zur Ausgabe) genutzt
       werden.
24 */
25 #define EYE_CONTROL_DDR_MASK 0x7F
26
27 /*
28     Diese Makros setzen jeweils die Taktleitung der Kamera auf 1 bzw.
       0 und erzeugt Verzögerungen, die für eine Stabilisierung sorgen.
29 */
30 #define setEyeClockHigh() delay_us(2); EYE_CONTROL |= _BV(EYE_XCK);
   delay_us(2);
31 #define setEyeClockLow() delay_us(2); EYE_CONTROL &= ~_BV(EYE_XCK);
   delay_us(2);
32
33 /*
34     Erleichtert das Setzen eines einzelnen Bits am gegebenen Port.
35 */
36 #define setBit(port, bit, value) if (value) { port |= _BV(bit); } else
    { port &= ~_BV(bit); }
```

Die dazugehörige .c-Datei sieht folgendermaßen aus:

```
1  /*
2      Für alle Funktionen gilt die Konvention, dass vor dem Aufrufen 0
       auf der Taktleitung liegen muss, und dass die Funktion die
       Taktleitung auch wieder auf 0 hinterlässt.
3  */
4
5  /*
6      Führt einen Reset der Kamera aus.
7  */
8  void eye_reset(){
```

```
9
10          /*
11              Legt 0 an die RST-Leitung an (RST ist low-aktiv).
12          */
13          setBit(EYE_CONTROL, EYE_RST, 0);
14
15          /*
16              Setzt den Takt auf 1 und ermöglicht so der Kamera, das
                Signal auf der RST-Leitung zu lesen (das Einlesen erfolgt
                bei steigender Taktflanke).
17          */
18          setEyeClockHigh();
19
20          /*
21              Setzt die RST-Leitung wieder zurück.
22          */
23          setBit(EYE_CONTROL, EYE_RST, 1);
24
25          /*
26              Setzt die Taktleitung wieder auf 0, um sie korrekt zu
                hinterlassen.
27          */
28          setEyeClockLow();
29  }
30
31
32  /*
33      Beschreibt das Konfigurationsregister regnum der Kamera mit dem
        Wert data.
34      Bei jedem Schreibvorgang wird zuerst die Adresse des zu
        beschreibenden Registers als 3-Bit-Wert seriell an SIN angelegt (
        höchstwertiges Bit zuerst). Danach werden 8 Bit angelegt, die das
        angegebene Register beschreiben. Die Übernahme der einzelnen Bits
        durch die Kamera wird durch Setzen des Taktes auf high
        gewährleistet; zudem muss der Schreibvorgang durch Setzen von LOAD
         auf 1 abgeschlossen werden.
35  */
36  void eye_writeToRegister(unsigned char regnum, unsigned char data){
37
38      // Zählvariable
39      int i;
40
41      /*
42          Setzt LOAD auf low, um den Schreibvorgang zu beginnen.
43      */
44      setBit( EYE_CONTROL, EYE_LOAD, 0 );
45
46      /*
47          Legt regnum bitweise an die SIN-Leitung.
48      */
49      for (i=2; i>=0; i--){
50          setEyeClockLow();
51
52          /*
53              Legt das i-te Bit der Adresse an.
54          */
55                  setBit( EYE_CONTROL, EYE_SIN , ((regnum>>i) & 0x01) );
56                  setEyeClockHigh();
57          }
58
59      /*
60          Analog zur Adresse wird hier bitweise data an die Kamera
```

```
            übergeben.
61      */
62      for (i=7; i>=0; i--){
63                  setEyeClockLow();
64                  setBit( EYE_CONTROL, EYE_SIN , ((data>>i) & 0x01) );
65                  setEyeClockHigh();
66          }
67
68          /*
69          Beendet den Schreibvorgang durch Setzen von LOAD auf 1
70      */
71          setBit( EYE_CONTROL, EYE_LOAD, 1 )
72
73          setEyeClockLow();
74
75      /*
76          Setzt LOAD wieder zurück
77      */
78          setBit( EYE_CONTROL, EYE_LOAD, 0 )
79
80  } // Ende eye_writeToRegister()
81
82
83
84  /*
85      Konfiguriert die Kamera, um für Aufnahmen die Belichtungszeit msec
           zu verwenden.
86      Die Kamera sieht hierzu zwei Konfigurationsregister vor, die
        zusammen die Belichtungszeit angeben:
87      Belichtungszeit = Register_c0 * 16 us + Register_c1 * 4.096 ms
88  */
89  void eye_setExposure(double msec){
90
91      /*
92          Um aus einer gegebenen Zahl msec die Register zu berechnen,
            wird zunächst msec / 4.096 berechnet, was die grobe Körnung
            und somit dem Wert von c1 angibt.
93      */
94          unsigned char c1 = (unsigned char)floor(msec/4.096);
95
96      /*
97          Der Rest aus obiger Division wird durch 0.016 geteilt, was die
             feine Körnung angibt.
98      */
99      unsigned char c0 = (unsigned char)floor((msec - 4.096*c1)/16e-3);
100
101     /*
102         Die entsprechenden Konfigurationsregister der Kamera werden
            mit den errechneten Werten beschrieben.
103     */
104         eye_writeToRegister(2, c1);
105         eye_writeToRegister(3, c0);
106
107 }
108
109 /*
110     Initialisiert die Kamera durch Setzen aller Konfigurationsregister
        .
111 */
112 void eye_init(double exposureTime){
113     /*
114         Setzt den EYE_DATA-Port als Eingabeport.
```

```
115     */
116     EYE_DATA_DDR= 0x00;
117     EYE_DATA = 0xff;
118
119     /*
120         Setzt die Maske für EYE_CONTROL.
121     */
122     EYE_CONTROL_DDR = EYE_CONTROL_DDR_MASK;
123
124     /*
125         Initialisiert alle Bits an Port EYE_CONTROL.
126     */
127     EYE_CONTROL &= 0x00;
128
129     /*
130         Entsprechend dem Flussdiagramm im Datenblatt des Artificial
            Retina Chips wird nun das Setup des Chips durchgeführt.
131         Zunächst sind ein Reset sowie das Setzen der Belichtungszeit
            nötig.
132     */
133
134     eye_reset();
135
136     eye_setExposure(exposureTime);
137
138     /*
139         Die Register 0 und 1 sowie 4-7 werden hier statisch
            beschrieben, da die besondere Filter- sowie
            Kantenerkennungseigenschaften des Chips nicht benötigt werden.
140
141         Die Register 4, 5 und 6 werden dem Datenblatt entsprechend für
             die Aufnahme eines Positivbildes gesetzt.
142     */
143     eye_writeToRegister(4, 0x01);
144     eye_writeToRegister(5, 0x00);
145     eye_writeToRegister(6, 0x01);
146
147
148     /*
149         Kalibriert die Kamera für ein positives Bild;
150         Voltage Offset = 0V
151     */
152     eye_writeToRegister(0, 0x80);
153
154     /*
155         Setzt die Verstärkung des Ausgangssignals auf 24.5 dB (diese
            Einstellung ist ein Erfahrungswert).
156     */
157     eye_writeToRegister(1, 0x07);
158
159     /*
160         Vref = 0V;
161         Non-inverted Output;
162         Edge Enhancement = 50% (kleinstmöglicher Wert)
163     */
164     eye_writeToRegister(7, 0x00);
165
166 } // Ende eye_init()
167
168
169 /*
170     Sendet den Start-Befehl an die Kamera.
```

```
171     Die Kamera beginnt daraufhin mit der Aufnahme von Bildern. Der
        Start-Befehl wird durch Anlegen von high an der START-Leitung
        realisiert.
172 */
173 void eye_start(){
174
175     setEyeClockLow();
176     setBit(EYE_CONTROL, EYE_START, 1);
177     setEyeClockHigh();
178     setEyeClockLow();
179     setBit(EYE_CONTROL, EYE_START, 0);
180
181 }
182
183 /*
184     Diese Funktion blockiert, bis die Kamera meldet, dass ein Bild
        fertig aufgenommen ist.
185 */
186 void eye_waitImageReady(){
187
188         /*
189             Prüft wiederholt, ob EYE_READ gesetzt ist.
190         */
191         while ( ! ((1<<EYE_READ)&EYE_CONTROL_IN) ){
192
193             /*
194                 Gibt Taktimpulse, damit die Kamera weiterarbeitet.
195             */
196             setEyeClockHigh();
197             setEyeClockLow();
198         }
199 }
200
201 /*
202     Liest einen blocksize großen Bildausschnitt beginnend mit Pixel #
        offset in den Puffer data und liefert die Menge der tatsächlich
        gelesenen Pixel zurück.
203 */
204 int eye_getImageBlock(unsigned char *data, int blocksize, int offset){
205
206     int bytesRead = 0;
207     int total = 0;
208
209     /*
210         Liest Daten aus, solange die Kamera ein valides Ausgangssignal
             liefert, noch nicht das Bildende erreicht wurde und auch noch
             nicht blocksize Bytes gelesen wurden.
211     */
212     while (((1<<EYE_READ)&EYE_CONTROL_IN) && total<15744 && bytesRead<
        blocksize){
213
214         setEyeClockHigh();
215
216         /*
217             Damit der Kameratakt nicht allzu ungleichmäßig wird, wird
                für die Pixel, die zwar von der Kamera gelesen, dann aber
                nicht in den Puffer geschrieben werden, eine künstliche
                Verzögerung eingebaut.
218             Da ein plötzlicher Taktwechsel (der eintritt, sobald die
                Verzögerung hinzukommt, die Daten auch tatsächlich zu
                lesen), zu Problemen bei der Bildqualität führt, wird
                rechtzeitig (250 Pixel vor dem eigentlich zu lesenden
```

```
                Datum) eine künstliche Verzögerung eingeführt, sodass die
                Bilddaten ab offset wieder akkurat sind. Der Wert von
                insgesamt (ca.) 220 us ist ein Erfahrungswert, er
                entspricht dem zweifachen der Zeit, die der uC auf den A/D
                -Wandler warten muss (das Taktsignal wird symmetrisch
                gehalten).
219         */
220         if (total>=offset-250) delay_us(110);
221
222         /*
223             Wenn offset erreicht wurde, kann nach 110 us davon
                ausgegangen werden, dass der A/D-Wandler fertig ist, und
                ein Byte an Bilddaten kann gelesen werden.
224         */
225         if (total>=offset) {
226             data[bytesRead++] = EYE_DATA;
227         }
228
229         total++;
230
231         setEyeClockLow();
232         if (total>=offset-250) delay_us(110);
233     }
234
235     return bytesRead;
236 }
```

Wie bereits erwähnt, fällt der Quellcode der dazugehörigen Applikation vergleichsweise einfach aus, was sich auch in der Länge des Codes niederschlägt; so ist die Header-Datei schnell zu überblicken:

```
1  /*
2      Dieses struct hält Statusinformationen des Servers fest.
3  */
4  struct camserv_state {
5
6      /*
7          Die Anzahl der Pixel, die bereits übertragen wurden (
               entspricht dem Index des Pixels, der als nächstes übertragen
               werden soll).
8      */
9      int pixel;
10
11     /*
12         Die Belichtungszeit, mit der Aufnahmen gemacht werden sollen (
               wird beim Ansteuern der Kamera durch 1000 geteilt).
13     */
14     int exposure;
15 };
```

Auch die .c-Datei ist nicht allzu komplex:

```
1  /*
2      Legt fest, wie viele Bytes an Bilddaten in einem Paket übertragen
       werden sollen.
3  */
4  #define BLOCKSIZE 1440
5
6  /*
7      Um Speicher zu sparen, werden die zu sendenden Daten direkt in den
        Paketpuffer geschrieben, wodurch uIP Ethernet- und TCP-Header
       hinzugefügt werden. Damit die Header nicht die Nutzlast
       überschreiben, werden 54 Bytes (Länge der beiden Header) Platz
       gelassen.
8      Diese Konstante gibt also das Offset im Speicher an, an dem die
       Nutzlast beginnen kann.
9  */
10 #define PAYLOAD_START &(uip_buf[UIP_LLH_LEN + 40])
11
12 /*
13     Zeiger für den einfacheren Zugriff auf das Status-struct der
       momentan aktiven Verbindung.
14 */
15 struct camserv_state *cs;
16
17 /*
18     Hält die Anzahl Bytes fest, die in Form von Bilddaten von der
       Kamera gelesen wurden.
19 */
20 int length;
21
22 /*
23     Die Funktion, die die eigentliche Applikation ausmacht.
24 */
25 void camserv_app(void){
26
27     /*
28         cs soll auf den Applikationsstatus der momentan aktiven
           Verbindung zeigen.
29     */
30     cs = (struct camserv_state *)(uip_conn->appstate);
31
32     /*
33         Neue Verbindung wurde aufgebaut.
34     */
35     if( uip_connected() ){
36
37         /*
38             Der Startpixel wird auf 0 gesetzt, da eine neue
               Bildübertragung ansteht.
39         */
40         cs->pixel=0;
41
42         /*
43             Die Belichtungszeit wird mit 0 initialisiert.
44             Die tatsächlich verwendete Zeit wird im uip_newdata()-
               Zweig gesetzt.
45         */
46         cs->exposure = 0;
47
48
49         /*
50             Vom neu zu sendenden Bild sind noch keine Daten gelesen
```

```
            worden.
51          */
52          length = 0;
53
54      }
55
56      /*
57          Es liegen neu empfangene Daten vor.
58          Da der Client nur einmal, nämlich zu Beginn der Verbindung,
            Daten an den Server sendet, muss auch nur dieser Fall hier
            behandelt werden. Dem Protokoll zufolge sendet der Client
            genau zwei Bytes Nutzlast, die die gewünschte Belichtungszeit
            enthalten; es kann davon ausgegangen werden, dass daher auch
            nur ein TCP-Paket vom Server empfangen wird.
59      */
60      if ( uip_newdata() ) {
61
62          /*
63              Liest die Belichtungszeit (als Ganzzahlwert) aus dem
                Empfangspuffer und speichert sie im entsprechenden Element
                 aus camserv_state.
64          */
65          cs->exposure = htons(*((int*)uip_appdata));
66
67          /*
68              Um die eigentliche Bildübertragung vollständig im
                uip_acked()-Zweig behandeln zu können, sendet der Server
                im ersten Paket an den Client noch keine Bilddaten,
                sondern nur die soeben empfangene Belichtungszeit. Der
                Client kann diese Informationen zwar ignorieren, jedoch
                können sie das Debugging erleichtern.
69          */
70          uip_send(PAYLOAD_START, sizeof(int));
71      }
72
73      /*
74          Ein an den Client gesendetes Paket wurde bestätigt.
75          Da es der begrenzte Speicherplatz nicht zulässt, dass einfach
            ein Foto gemacht und dieses dann Stück für Stück übertragen
            wird, wird für jedes an den Client zu sendende Paket die
            Kamera neu initialisiert und ein neues Foto gemacht,
            übertragen wird dann jeweils nur ein Bildausschnitt. Für
            diesen Funktionszweig bedeutet dies, dass der Bildanfang
            genauso behandelt wird wie alle anderen Bildteile.
76      */
77      if( uip_acked() ){
78
79          /*
80              Falls die Anzahl der bereits gesendeten Pixel 15744 ist,
                wurde bereits das komplette Bild gesendet.
81          */
82          if (cs->pixel>=15744){
83
84              /*
85                  Die Verbindung kann geschlossen werden.
86              */
87              uip_close();
88
89          } else { // Das Bild wurde noch nicht komplett gesendet
90
91              /*
92                  Initialisiert die Kamera; exposure / 1000.0 ist
```

```
                    hierbei die Belichtungszeit, mit der die Aufnahme
                    gemacht werden soll. Für Details zu den eye_ -
                    Funktionen siehe auch die Beschreibung des
                    Kameratreibers.
93              */
94              eye_init((double)cs->exposure/1000.0);
95
96
97              /*
98                  Startet die Aufnahme.
99              */
100             eye_start();
101
102             /*
103                 Diese Funktion blockiert solange, bis die Kamera
                    meldet, dass die Aufnahme gemacht wurde (frühestens
                    nach exposure / 1000.0 ms).
104             */
105             eye_waitImageReady();
106
107             /*
108                 Lädt die Bilddaten von der Kamera.
109                 Die Daten (BLOCKSIZE Bytes) werden hierbei im
                    Paketpuffer abgelegt (sh. PAYLOAD_START); bei welchem
                    Pixel die Übertragung beginnen soll wird der Funktion
                    im dritten Parameter mitgeteilt. length speichert, wie
                     viele Bytes tatsächlich gelesen wurden (am Bildende
                    ist length < BLOCKSIZE).
110             */
111             length = eye_getImageBlock(PAYLOAD_START, BLOCKSIZE, cs->
                pixel);
112
113             /*
114                 Damit beim nächsten Durchlauf der nachfolgende
                    Bildausschnitt übertragen wird, wird das pixel-Element
                     aus camserv_state entsprechend inkrementiert.
115             */
116             cs->pixel += length;
117
118             /*
119                 Sendet die Anzahl Bytes an den Client, die soeben von
                    der Kamera bezogen wurde.
120             */
121             uip_send(PAYLOAD_START, length);
122         }
123
124     } // Ende if ( uip_newdata() )
125
126     /*
127         Eine Neuübertragung des letzten Paketes wird notwendig.
128         Da in der Zeit seit dem letzten Sendevorgang nicht in den
            Puffer geschrieben wurde, sind die Bilddaten dort noch intakt;
             auch hat length noch den Wert, den es beim letzten Senden
            hatte. Es kann also einfach der Sendebefehl aus uip_acked()
            wiederholt werden.
129     */
130     if ( uip_rexmit() ){
131         uip_send(PAYLOAD_START, length);
132     }
133
134 } // Ende camserv_app()
135
```

```
136 /*
137     Initialisierungfunktion für die Applikation.
138     Es muss nur uIP mitgeteilt werden, dass es auf TCP-Port 6666 auf
        eingehende Verbindungen warten soll.
139 */
140 void camserv_init(void) {
141
142     uip_listen(HTONS(6666));
143 }
```

Damit uIP optimal mit der gezeigten Applikation zusammenarbeitet, ist es wiederum nötig, einige Einstellungen anzupassen:

- Die Anzahl der Verbindungen `UIP_CONNS`, die uIP verwenden kann, muss auf 1 gesetzt werden – das Aufnehmen und Versenden von Fotos ist schließlich schon für eine (nicht unterbrochene) Verbindung zeitaufwendig genug.
- Da dieser Server wie auch der HTTP-Server nur auf einem Port lauschen soll, kann `UIP_LISTENPORTS` auf 1 gesetzt werden.
- `UIP_BUFSIZE` kann auf den Maximalwert 1500 gesetzt werden, da zum einen die Applikation außerhalb des Puffers kaum Speicher benötigt und zum anderen umso mehr Pixel pro TCP-Paket versendet werden, was die Bildübertragung beschleunigt.
- Analog zum HTTP-Server können auch hier `UIP_STATISTICS` wie auch `UIP_URGDATA` auf 0 gesetzt werden.

4.2.5 Verwendung von UDP

Über den bereits zur Genüge erläuterten Gebrauch von TCP hinaus bietet uIP auch (rudimentäre) Unterstützung von UDP. Diese sei hier kurz in einem Beispiel gezeigt, wurde aber im Rahmen dieser Arbeit nur zur Bestimmung der maximalen Übertragungsrate, zu der uIP fähig ist, genutzt. Eine Verwendung von UDP durch den Kamera-Server, bei dem schließlich das Protokoll frei gewählt werden konnte, brachte im Experiment keinen nennenswerten Geschwindigkeitszuwachs, da hier das langsamste Glied der A/D-Wandler war. Die gesicherte Übertragung mit TCP ist hier der höher einzuschätzende Vorteil.

Vorbereitung für UDP

- `UIP_UDP` muss selbstverständlich auf 1 gesetzt werden, die Anzahl der möglichen UDP-"Verbindungen" `UIP_UDP_CONNS` kann angepasst werden, im Beispiel genügt der Wert 1.
- `UIP_UDP_APPCALL` auf den gewünschten Funktionsnamen setzen.
- Der in main.c vorgegebene Code ist für den Aufruf der UDP-Applikation nicht ausreichend; hier empfiehlt es sich, in der Hauptschleife ab Zeile 115 folgenden Codeschnipsel einzufügen, der die Applikation bei jedem Schleifendurchlauf aufruft. So ist es möglich, bei jedem Durchlauf ein Paket zu versenden.

```
for(i = 0; i < UIP_UDP_CONNS; i++){
    uip_udp_periodic(i);
    if(uip_len > 0){
        uip_arp_out();
        nic_send();
    }
}
```

Applikationscode Die Header-Datei wird hier nicht weiter erläutert, in ihr werden nur die benötigten Funktionen deklariert. Der Applikationscode selbst sieht folgendermaßen aus:

```
1  u16_t ipaddr[2];
2
3  //  Initialisierungsfunktion
4  void throughput_init(void){
5
6      /*
7          In ipaddr wird die IP-Adresse abgelegt, zu der Pakete
           versendet werden sollen.
8      */
9      uip_ipaddr(ipaddr, 192,168,99,104);
10
11     /*
12         Erzeugt eine neue "Verbindung" zur gegebenen Adresse an Port
           12345.
13     */
14     uip_udp_new(ipaddr, 12345);
15 }
16
17 //  Applikationsfunktion
18 void throughput_app(void){
19
20     /*
```

```
21          Sendet ein UDP-Paket der maximal möglichen Größe.
22          Die versendeten Daten werden dabei dem Paketpuffer entnommen -
             da sie nie gezielt dort hinein geschrieben wurden, sind sie
            quasi "zufällig".
23          Für Messzwecke ist dies optimal, da der uC keine Zeit mit dem
            Generieren von Daten "vergeudet".
24      */
25      uip_udp_send(UIP_BUFSIZE - (UIP_LLH_LEN + 28));
26  }
```

Damit ist die Beschreibung der im Rahmen dieser Arbeit mit uIP entwickelten Software abgeschlossen. Im nächsten Abschnitt werden einige der mittels ihr erzielten Ergebnisse vorstellen.

5 Ergebnisse

Bei der Arbeit mit uIP stellte sich heraus, dass durchaus akzeptable Netzwerk-Übertragungsraten erzielt werden können, wie im Folgenden aufgezeigt wird. Zum Testen wurde hierbei der in Abschnitt 3.6 beschriebene Aufbau mittels eines 10/100 MBit Switching Hubs mit zwei PCs verbunden, auf denen in beiden Fällen gentoo Linux[27] mit Kerneln der 2.6er Reihe zum Einsatz kam.

5.1 Generelles

5.1.1 Antwortzeit beim Ping

Auf einen "ICMP Echo Request" antwortete das Controller-Netzwerkkarten-Gespann im Versuchsaufbau wie folgt:

```
$ ping 192.168.99.66
PING 192.168.99.66 (192.168.99.66) 56(84) bytes of data.
64 bytes from 192.168.99.66: icmp_seq=1 ttl=64 time=0.858 ms
64 bytes from 192.168.99.66: icmp_seq=2 ttl=64 time=0.856 ms
64 bytes from 192.168.99.66: icmp_seq=3 ttl=64 time=0.856 ms
64 bytes from 192.168.99.66: icmp_seq=4 ttl=64 time=0.857 ms
64 bytes from 192.168.99.66: icmp_seq=5 ttl=64 time=0.857 ms
64 bytes from 192.168.99.66: icmp_seq=6 ttl=64 time=0.855 ms

--- 192.168.99.66 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 4999ms
rtt min/avg/max/mdev = 0.855/0.856/0.858/0.029 ms
```

Zum Vergleich die Antwortzeit(en) zwischen den beiden PCs:

```
$ ping 192.168.99.104
PING 192.168.99.104 (192.168.99.104) 56(84) bytes of data.
64 bytes from 192.168.99.104: icmp_seq=1 ttl=64 time=0.147 ms
64 bytes from 192.168.99.104: icmp_seq=2 ttl=64 time=0.150 ms
64 bytes from 192.168.99.104: icmp_seq=3 ttl=64 time=0.142 ms
64 bytes from 192.168.99.104: icmp_seq=4 ttl=64 time=0.144 ms
64 bytes from 192.168.99.104: icmp_seq=5 ttl=64 time=0.135 ms
64 bytes from 192.168.99.104: icmp_seq=6 ttl=64 time=0.155 ms

--- 192.168.99.104 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5000ms
rtt min/avg/max/mdev = 0.135/0.145/0.155/0.013 ms
```

[27] siehe [Toog]

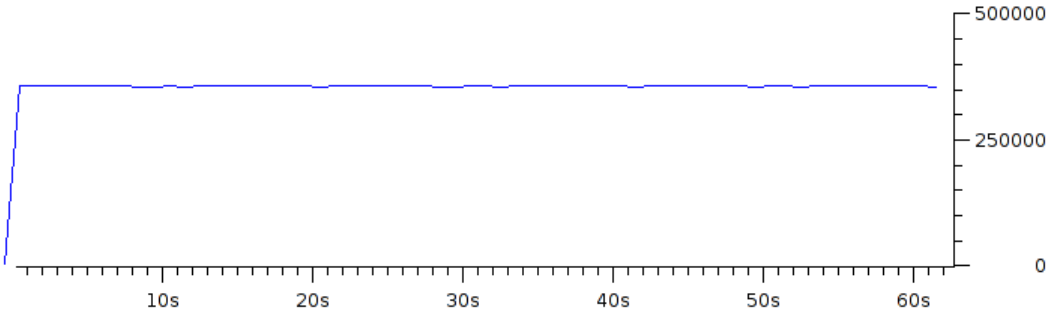


Abbildung 13: Mit Ethereal erstellter Graph, der den UDP-Verkehr zeigt. Die x-Achse beschreibt die Zeit, die y-Achse die Anzahl Bytes

5.1.2 Maximaler Durchsatz

Ein Mitschnitt des Netzwerkverkehrs, den die in Abschnitt 4.2.5 vorgestellte UDP-Applikation verursacht, ist in Abb. 13 zu sehen. Wie dem Graphen zu entnehmen ist, beträgt die maximale Senderate für die im Aufbau verwendete Hardware ca. 310 kB/s. Da dabei der Netzwerkchip noch nicht an seine Grenzen stößt, könnte ein schneller getakteter μC wohl noch höhere Durchsatzraten erzielen. Beim Betrachten der hier aufgezeigten Ergebnisse muss auch bedacht werden, dass der μC außer dem Senden keinerlei Aufgaben übernimmt – in der Praxis wird ein Teil der Rechenzeit darauf entfallen, die zu sendenden Daten zuvor in irgendeiner Weise zu generieren, wodurch wiederum zwischen den Sendeoperationen mehr Zeit vergeht und die Übertragungsrate sinkt.

5.2 Webserver

Der in Abschnitt 4.2.3 gezeigte Webserver zeigt (erwartungsgemäß) eine wesentlich geringere Übertragungsrate als die eben gezeigte. Dies liegt zum einen an den Zeitverlusten, die das Lesen der Daten von der MMC-Karte verursacht sowie der vergleichsweise kleinen Paketlänge. Zum anderen ist die Übertragung langsamer, da TCP verwendet wird: Gerade dadurch, dass ein Paket erst gesendet wird, wenn ein ACK für das vorangegangene empfangen wurde, treten lange Wartezeiten zwischen den einzelnen Sendevorgängen auf, wohingegen der UDP-Server ununterbrochen senden kann. Beim Warten auf ACKs ist (zumindest in der verwendeten Testumgebung) die Latenz des Netzwerks weniger kritisch; die Ursache für Probleme liegt darin, dass normalerweise Pakete nicht mit ACK-Takt versendet werden, sondern solange Pakete auf die Leitung gelegt werden, bis die Fenstergröße des Empfängers

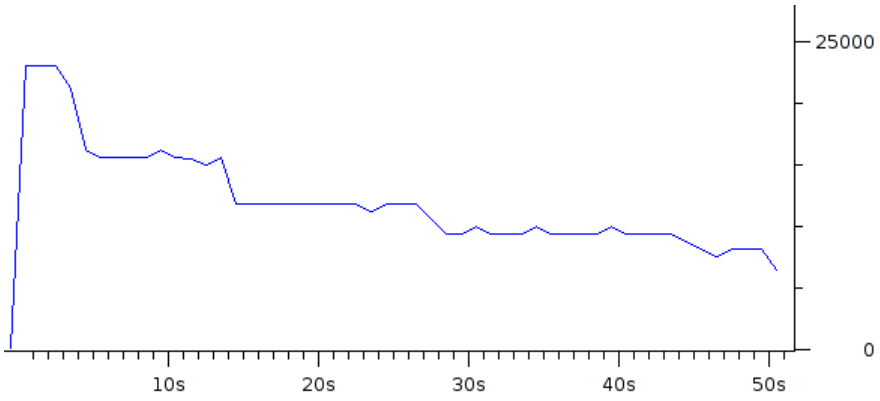


Abbildung 14: Mit Ethereal erstellter Graph, der den Datendurchsatz beim Abruf einer 500kB-Datei per HTTP aufzeigt.

erreicht ist. Dementsprechend geht der Empfänger davon aus, eine gewisse Anzahl eingehender Pakete anhäufen zu können, bevor er diese mit einem einzigen (akkumulativen) ACK quittiert. Der in RFC 2581 vorgeschlagene "delayed ACK"-Algorithmus[28] sieht nun vor, dass der Empfänger zwischen zwei gesendeten ACKs bis zu 500ms verstreichen lassen kann. Die tatsächliche Wartezeit wird in den meisten TCP-Stacks heuristisch festgelegt, so dass sich bei den verwendeten PCs die Wartezeit zwischen zwei ACKs mit steigender Paketanzahl erhöht. Da jedoch der Server in dieser Zeit kein weiteres Paket sendet, sinkt mit steigender Zeit die Durchsatzrate, wie in Abb. 14 zu sehen ist. Bei Dateien von mehreren Megabyte Größe erreicht die Wartezeit zwischen zwei ACKs in Testaufbau die erwähnten 500ms, was zu einer Übertragungsrate von unter 1kB/s führt.

Dem Problem der langsamer werdenden Verbindung kann begegnet werden, indem auf Serverseite die Wartezeit für einen Retransmit-Timeout sehr gering gewählt wird: Wartet der Server zu lange (beispielsweise länger als 200ms) auf ein ACK, überträgt er das letzte Paket erneut. Ein optimal RFC-konformer Client nimmt dieses zweite seit dem letzten ACK eingetroffene Paket zum Anlass, sofort ein ACK zu senden. Überdies setzten die Versuchsrechner die Wartezeit für das delayed ACK zurück auf den Minimalwert, sodass für eine gewisse Zeit wieder jedes Paket sofort bestätigt wurde.

Dieser Ansatz birgt jedoch seinerseits wieder Probleme wie die erhöhte Netzwerklast und eine nach wie vor langsamer werdende Übertragung. Zudem ist davon abzuraten, den Retransmit-Mechanismus zur Verbesserung der Durchsatzrate zu "missbrauchen".

[28] sh. [RFCb, Abschnitt 4.2]

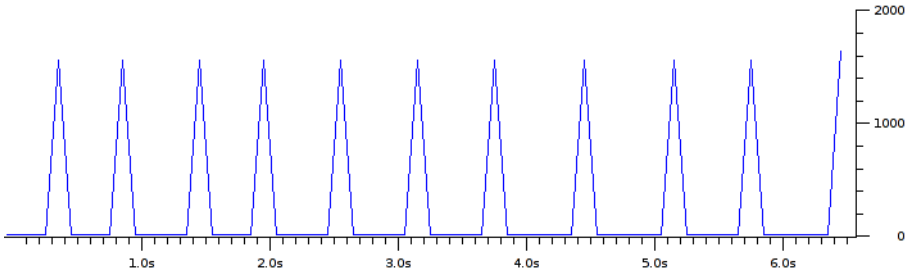


Abbildung 15: Mit Ethereal erstellter Graph, der den Datendurchsatz beim Abruf eines Bildes vom Kameraserver zeigt.

5.3 Kameraserver

Anders als beim Webserver sind in diesem Aufbau die wiederholten Zugriffe auf die Kamera bzw. der langsame A/D-Wandler der Flaschenhals, der den Datendurchsatz limitiert. Abb. 15 zeigt den zeitlichen Verlauf des Netzwerkverkehrs bei Übertragung eines Bildes. Die elf Spitzen resultieren daraus, dass die Gesamtmenge von 15744 Bytes Daten in Pakete à 1440 Bytes Nutzlast aufgeteilt wird[29]. Von den insgesamt etwa 6.5 Sekunden, die zur Übertragung eines Bildes benötigt werden, entfallen alleine ca. 4.1 Sekunden auf Wartezeiten für A/D-Wandlung (bzw. Taktstabilisierung). Hinzu kommt noch der Overhead für das Initialisieren der Kamera, das Belichten und das "Durchlaufen" bis zum gerade betrachteten Bildausschnitt. Dass letzteres länger dauert, je mehr sich die Übertragung dem Bildende nähert, spiegelt sich auch in den zunehmenden zeitlichen Abständen zwischen den einzelnen Paketen wider. Für viele praktische Einsatzgebiete ist die Arbeitsweise der Applikation leider ungeeignet, da nur solche Motive einigermaßen akkurat aufgenommen werden können, die 6.5 Sekunden lang unbewegt sind.

Zu Veranschaulichung der Qualität der mit der Kamera aufgenommenen Bilder sind hier zwei Aufnahmen abgebildet, die die Autoren zeigen.

Abbildung 16: Die Autoren, aufgenommen mit der "Artificial Retina"

[29] $\lceil 15744/1440 \approx 10.93 \rceil = 11$

6 Ausblick

Unter Beibehaltung der erläuterten μC/Netzwerkkarten-Kombination lassen sich bereits mannigfaltige Aufgaben bewältigen. So können die zur Verfügung stehenden Ressourcen in vielen Arten für Steuerung oder Diagnose eingesetzt werden, welche dann über Ethernet zugreif- bzw. konfigurierbar sind. Internet-Communities wie z.B. mikrocontroller.net ([Forb]) lassen schnell erkennen, dass die Einsatzmöglichkeiten für μCs im Allgemeinen fast unbegrenzt sind – viele bereits existente Aufbauten würden durch einen Ethernetanschluss an Komfort und Funktionalität hinzugewinnen.

Leistungsstärkere Controller bzw. zusätzlicher (externer) RAM eröffnen überdies die Möglichkeit, Systeme wie Contiki ([Sofa]) einzusetzen, womit dann ein vollwertiges Betriebssystem (samt TCP-Stack) zum Einsatz käme. Insbesondere die Möglichkeit, mehrere Prozesse parallel laufen und das Scheduling vom Betriebssystem übernehmen zu lassen, verlangt dem Programmierer ein geringeres Maß an Anpassung an die Hardware ab und ließe somit eine effektivere Art der Softwareentwicklung zu.

Der eingesetzte uIP-AVR-Code liegt zum Zeitpunkt der Erstellung dieser Arbeit schon seit mehreren Monaten brach, lässt aber dem geneigten Entwickler durchaus Raum für Verbesserungen: So könnte z.B. ein Interrupt-Mechanismus implementiert werden, so dass der μC nicht mehr ununterbrochen, sondern nur beim Auftreten von Netzwerkanfragen aktiv wird und in der übrigen Zeit beispielsweise im Energie sparenden Sleep-Modus verweilen kann.

Ferner wäre der Anschluss des Mikrocontrollers an einen WLAN-Adapter wünschenswert. Dies würde auch den mobilen Einsatz des Aufbaus ermöglichen. Fahr- oder Fluggeräte könnten so gesteuert werden oder autonom agierende Roboter den Aufbau zur Kommunikation mit der Außenwelt nutzen.

Abschließend bleibt festzuhalten, dass der Aufbau, obschon bereits in der jetzigen Form vielseitig einsetzbar, noch ein hohes Potential an Möglichkeiten der Weiterentwicklung birgt.

Literatur

[Fora] *avrfreaks.net – Internationale AVR-Community.* http://www.avrfreaks.net/.

[Forb] *Mikrocontroller.net – Deutsche µC Community.* http://www.mikrocontroller.net/.

[Hara] *Ethernut Homepage.* http://www.ethernut.de/en/index.html.

[Harb] *Realtek Homepage.* http://www.realtek.com.tw.

[Proa] *AndyBits.* http://pages.zoom.co.uk/andyc/.

[Prob] *Implementation UDP into microcontroller Igor Atmel-UDP device.* http://cesko.euro-science.net/IgorPlugUDP/IgorPlug-UDP

[Proc] *Implementing Vision Using a GAME BOY Camera on the MRM.* http://www.seattlerobotics.org/encoder/200205/gbcam.html.

[Prod] *Webseiten von Ulrich Radig.* http://www.ulrichradig.de.

[RFCa] *RFC 1122 – Requirements for Internet Hosts.* http://www.faqs.org/rfcs/rfc1122.html.

[RFCb] *RFC 2581 – TCP Congestion Control.* http://www.ietf.org/rfc/rfc2581.txt.

[Sofa] *The Contiki Operating System.* http://www.sics.se/ adam/contiki/.

[Sofb] *uIP-AVR Homepage.* http://www.laskater.com/projects/uipAVR.htm.

[Sofc] *uIP-Dokumentation.* http://www.sics.se/∼adam/download/uip-0.9-refman.pdf.

[Sofd] *uIP Homepage.* http://www.sics.se/∼adam/uip/.

[Tooa] *Brian S. Dean's website – Private Webseite des Initiators von avrdude.* http://www.bsdhome.com/.

[Toob] *Homepage der avr-libc.* http://www.nongnu.org/avr-libc/.

[Tooc] *Homepage der binutils.* http://www.gnu.org/software/binutils/.

[Tood] *Homepage des GCC.* http://www.gnu.org/software/gcc/gcc.html.

[Tooe] *Homepage des WinAVR-Projekts.* http://winavr.sourceforge.net/.

[Toof] *Homepage von avrdude*. http://www.nongnu.org/avrdude/.

[Toog] *Homepage von gentoo Linux*. http://www.gentoo.org.

7 Anhang

7.1 Ätzvorlage

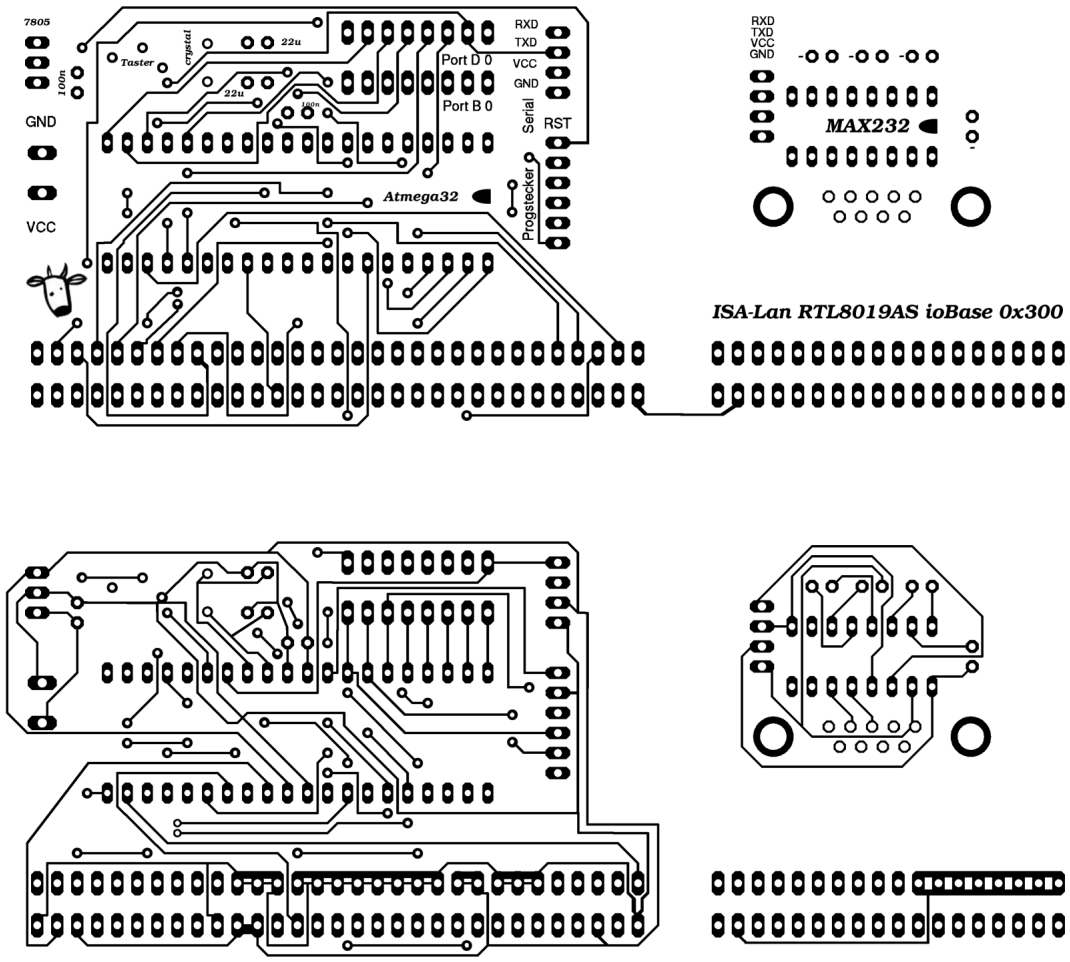


Abbildung 17: Ätzvorlage für den Aufbau incl. Slot für die ISA-Netzwerkkarte und Adapterplatine für den Anschluss des Controllers an die serielle Schnittstelle. Oben: Top-Layer; Unten: Bottom-Layer. In der oberen rechten Ecke liegt das Layout für den Adapter zum Anschluss an die serielle Schnittstelle des PCs. Dieser kann nachher von der Hauptplatine abgetrennt werden.

7.2 Client-Quellcode (sh. Seite 49)

7.2.1 Datei TCPReceiver.java

```
1  package base;
2  import java.io.IOException;
3  import java.net.InetAddress;
4  import java.net.Socket;
5  import java.net.UnknownHostException;
6
7  /**
8      Diese Klasse kapselt die Funktionalität, um mit dem Kameraserver
       zu kommunizieren.
9      Damit das Empfangen der Daten nicht zwangsläufig das GUI blockiert
       , erbt TCPReceiver von Thread und kann somit nebenläufig die
       Bilddaten empfangen.
10  */
11 public class TCPReceiver extends Thread {
12
13         /*
14         Die vorgegebenen Werte für die IP-Adresse bzw. den Port, auf
           dem der Server läuft. Diese Werte sind nur relevant, falls der
            Benutzer keine eigenen Werte einstellt.
15     */
16         public static final byte[] DEFAULT_SERVER_IP = {(byte)192,(
           byte)168,99,66};
17         public static final int DEFAULT_SERVER_PORT = 6666;
18
19
20         /**
21         Die (vorzeichenbehafteten) Rohdaten, wie sie vom Server
           empfangen wurden.
22         signedRawdata wird verwendet, um die verschiedenen Threads zu
           synchronisieren, d.h. dass es nicht möglich ist, dass
           gleichzeitig der TCPReceiver neue Daten empfängt und dem
           ImageCreator Bilddaten liefert (was schließlich zu defekten
           Bildern führen würde).
23          */
24         private byte[] signedRawdata;
25
26         private GBcamclient camclient;
27
28         /*
29         Die IP-Adresse und der Port des Kamera-Servers.
30     */
31     private InetAddress serverAddress;
32         private int serverPort;
33
34         private Socket socket;
35
36         /**
37         Erzeugt einen neuen TCPReceiver.
38          */
39         public TCPReceiver(GBcamclient camclient) {
40                 signedRawdata = new byte[GBcamclient.IMAGEWIDTH *
                   GBcamclient.IMAGEHEIGHT];
41
42                 this.camclient = camclient;
43
```

```
44                  socket = null;
45          }
46
47      //-------------------------------------------------------
48      // Getter & Setter
49      //-------------------------------------------------------
50
51          /**
52          Liefert die IP-Adresse des Servers zurück.
53           */
54          public InetAddress getServerAddress() {
55                  return serverAddress;
56          }
57
58          /**
59          Setzt die IP-Adresse des Servers.
60           */
61          public void setServerAddress(InetAddress serverAddress) {
62                  this.serverAddress = serverAddress;
63          }
64
65          /**
66          Liefert den Port des Servers zurück.
67           */
68          public int getServerPort() {
69                  return serverPort;
70          }
71
72          /**
73          Setzt den Port des Servers.
74           */
75          public void setServerPort(int serverPort) {
76                  this.serverPort = serverPort;
77          }
78
79          /**
80          Liefert die (vorzeichenbehafteten) Rohdaten zurück.
81          Diese Methode nutzt signedRawdata zur Synchronisation: falls
            gerade neue Daten vom Server empfangen werden, blockiert der
            aufrufende Thread, bis der Empfang abgeschlossen ist.
            Praktisch bedeutet dies, dass das GUI blockiert, sobald man
            neue Daten vom Server anfordert.
82           */
83          public byte[] getSignedRawdata() {
84                  synchronized (signedRawdata) {
85                          return signedRawdata;
86                  }
87          }
88
89      //-------------------------------------------------------
90      // Netzwerk-Methoden
91      //-------------------------------------------------------
92
93          /**
94          Öffnet einen Socket zum Server.
95          Falls setServerAddress() und setServerPort() noch nicht
            aufgerufen wurden, werden die Default-Werte verwendet.
96          Diese Methode wird in run() aufgerufen.
97           */
98          private void openSocket(){
99
100         /*
```

```
101             Falls die Servereinstellungen noch nicht gesetzt wurden,
                sollen die Default-Werte genutzt werden.
102          */
103
104                 if (getServerAddress() == null){
105                         try {
106                                 setServerAddress(InetAddress.
                                    getByAddress(DEFAULT_SERVER_IP));
107                         } catch (UnknownHostException e1) {
108                                 e1.printStackTrace();
109                         }
110                 }
111
112                 if(getServerPort() == 0){
113                         setServerPort(DEFAULT_SERVER_PORT);
114                 }
115
116                 try {
117                         socket = new Socket(serverAddress, serverPort)
                            ;
118                 } catch (IOException e) {
119                         e.printStackTrace();
120                 }
121         }
122
123         /**
124         Schließt den Socket.
125         Diese Methode wird in run() aufgerufen.
126          */
127         private void closeSocket(){
128                 try {
129                         socket.close();
130                 } catch (IOException e) {
131                         e.printStackTrace();
132                 }
133         }
134
135         /**
136         Sendet ein TCP Paket an den Server, welches die
            Belichtungszeit * 1000 enthält.
137          */
138         private void sendStartCommand(){
139                 try {
140             /*
141                 Um Probleme mit Fliekommazahlen auf verschiedenen
                    Plattformen zu vermeiden, wird hier ein short-Wert
                    erzeugt, der dann auf Seiten des uC wieder
                    zurückkonvertiert wird.
142             */
143                         short shortTime = (short)(camclient.
                            getExposureTime() * 1000);
144
145             /*
146                 Umwandlung in "Network byte order", welche der uC
                    erwartet.
147             */
148                         byte[] firstAndSecondByte = new byte[2];
149
150             firstAndSecondByte[0] = (byte)((shortTime & 0xff00)>>8);
151             firstAndSecondByte[1] = (byte)(shortTime & 0xff);
152                         socket.getOutputStream().write(
                            firstAndSecondByte);
```

```
153
154                         // Debug-Ausgabe
155                         System.out.println("Exposure time " +
                            camclient.getExposureTime() + " sent to server
                             as short: "
156                                         + shortTime);
157
158                 } catch (IOException e) {
159                         e.printStackTrace();
160                 }
161         }
162
163
164     /**
165          Diese Methode erledigt die Hauptarbeit: sie empfängt und
             speichert die vom Server gesendeten Daten.
166          Die ersten Zwei Byte werden eingelesen, in "Host Byte Order"
             gebracht und angezeigt, der Rest wird als Bilddaten in
             signedRawdata gespeichert. Diese Methode benutzt
             signedRawdata zur Synchronisation: solange Daten empfangen
             werden, blockiert getSignedRawdata().
167      */
168         private void receiveData(){
169                 synchronized (signedRawdata) {
170
171                         byte[] returnedExposureTime = new byte[2];
172                         try {
173                                 int bytesReadForExposureTime = socket.
                                    getInputStream().read(
                                    returnedExposureTime);
174                                 if (bytesReadForExposureTime != 2){
175                                         System.err.println("Warning:
                                            returned exposure time from
                                            the server could" +
176                                                         "not be read!"
                                                         );
177                                 }
178
179                 /*
180                      Der nachfolgende Code ist die einzige Möglichkeit
                        , korrekte int- (bzw. Bit-) Werte zu erhalten.
181                      Das Verhalten von Java ist hier irreführend: 0
                        xFFFF & byteWert liefert einen short-Wert, dessen
                         höherwertiges Byte 0xFF ist; Es werden also Bits
                        , die in einem bestimmten Datentyp gar nicht
                        existieren, mit einem Wert von 1 angenommen, wenn
                         man bitweises & verwendet.
182                  */
183                 int hibyte = ((int)0xFF) & returnedExposureTime[0];
184                 int lobyte = ((int)0xFF) & returnedExposureTime[1];
185                 int time = (hibyte <<8) | lobyte;
186
187                                 /*
188                                      Debug-Ausgabe
189                                  */
190                                 System.out.println("Exposure time was
                                    received: " + time);
191                 System.out.println("This equals: hibyte " +
                    returnedExposureTime[0] + " | lobyte " +
                    returnedExposureTime[1]);
192                 System.out.println("Or casted to ints: hibyte " +
                    hibyte + " | lobyte " + lobyte);
```

```
193
194                 /*
195                      Kommentare können für Debugging-Zwecke entfernt
                         werden, um eine Fortschrittsanzeige zu erhalten.
196                  */
197                 //System.out.println("Progress (one dot for each 50
                    pixels): ");
198                                 for (int i = 0; i < GBcamclient.
                                    IMAGEWIDTH * GBcamclient.IMAGEHEIGHT;
                                    i++){
199                                         socket.getInputStream().read(
                                            signedRawdata,i,1);
200                                         // if (i % 50 == 0) System.out
                                            .print(".");
201                                 }
202                         } catch (IOException e) {
203                                 e.printStackTrace();
204                         }
205                 }
206         }
207
208     //-------------------------------------------------------
209     // Thread-Methoden
210     //-------------------------------------------------------
211
212         /**
213              Überschreibt die run()-Methode aus Thread.
214          Für jeden Aufruf von run() wird ein neuer Socket zum Server
             geöffnet, das Startkommando gesendet, die Bilddaten empfangen
              und der Socket anschließend wieder geschlossen.
215          */
216         public void run(){
217                 openSocket();
218                 sendStartCommand();
219                 receiveData();
220                 closeSocket();
221         }
222
223 }
```

7.2.2 Datei ImageCreator.java

```
1  package base;
2
3  import java.awt.Image;
4  import java.awt.image.BufferedImage;
5  import java.awt.image.RenderedImage;
6  import java.io.IOException;
7  import java.io.RandomAccessFile;
8  import java.util.Iterator;
9  import java.util.NoSuchElementException;
10
11 import javax.imageio.ImageIO;
12 import javax.imageio.ImageWriter;
13 import javax.imageio.stream.ImageOutputStream;
```

```
14
15 /**
16     Diese Klasse ist verantwortlich für die Konvertierung der Rohdaten
       , die von der Kamera kommen, in ein anzeigbares Bild.
17  */
18 public class ImageCreator{
19
20     /*
21         Da die Kamera Spannungen zwischen 1.5V und 3.5V liefert, der A
           /D-Wandler jedoch zwischen 0V und 5V wandelt, müssen die vom A
           /D-Wandler bezogenen Werte zunächst verkleinert werden. Danach
            werden sie aufgespreizt, um schließlich Graustufen zwischen 0
            und 255 zu haben.
22         ADJUSTVALUESBY und SPLAYVALUESBY sind hierbei konstante Werte,
            die erfahrungsgemäß gute Endergebnisse erzielen.
23     */
24         public static final short ADJUSTVALUESBY = 60;
25         public static final float SPLAYVALUESBY = 2.9f;
26
27
28     /*
29         Das Bild als Rohdaten.
30         Da Java kein "unsigned byte" kennt, muss hier der
           nächstgrößere Datentyp gewählt werden, um Werte zwischen 0 und
            255 speichern zu können.
31     */
32         private short[] rawdata;
33
34     /*
35         Das Bild als BufferedImage.
36         In diesem Format kann es leicht von Java weiterverarbeitet/
           angezeigt werden.
37     */
38         private BufferedImage image;
39
40     /*
41         Eine skalierte Ausgabe von image.
42         Wird hier von ImageCreator vorgehalten, um den
           Berechnungsaufwand des Skalierens für die Anzeige niedrig zu
           halten.
43     */
44     private Image scaledImage;
45
46     /**
47         Erzeugt einen neuen ImageCreator.
48     */
49         public ImageCreator(){
50                 rawdata = new short[GBcamclient.IMAGEHEIGHT *
                   GBcamclient.IMAGEWIDTH];
51         }
52
53         //-------------------------------------------------------
54         // Hilfsmethoden
55         //-------------------------------------------------------
56
57         /**
58         Diese Methode wandelt die vorzeichenbehafteten Werte in
           signedRawdata in vorzeichenlose short-Werte um. Darüberhinaus
           werden die Werte durch Verringerung und Aufspreizung auf den
           Farbraum von 0 bis 255 verteilt.
59          */
```

```
60          private short[] adjustAndSplayValues(byte[] signedRawdata,
            short lowerBy, float splayBy){
61
62                  short[] unsignedRawdata = new short[signedRawdata.
                    length];
63                  for (int i = 0; i < signedRawdata.length; i++){
64                          short loweredValue = (short)(Utils.
                            toUnsignedShort(signedRawdata[i]) - lowerBy);
65                          if (loweredValue < 0) loweredValue = 0;
66                          short splayedValue = (short)(loweredValue *
                            splayBy);
67                          if (splayedValue > 255) splayedValue = 255;
68                          unsignedRawdata[i] = splayedValue;
69                  }
70                  return unsignedRawdata;
71          }
72
73          //------------------------------------------------------
74          // Getter & Setter
75          //------------------------------------------------------
76
77      /**
78          Liefert das Rohdaten-Array rawdata zurück.
79      */
80          public short[] getRawdata() {
81                  return rawdata;
82          }
83
84      /**
85          Setzt rawdata auf den gegebenen Wert.
86      */
87          public void setRawdata(short[] rawdata) {
88                  this.rawdata = rawdata;
89          }
90
91      /**
92          Liefert das BufferedImage dieses ImageCreator zurück.
93          Falls noch kein BufferedImage existiert, erzeugt die
            Hilfsmethode generateBufferedImage() es aus den Rohdaten.
94      */
95          public BufferedImage getImage() {
96                  if (image == null){
97                          generateBufferedImage();
98                  }
99                  return image;
100         }
101
102     /**
103         Setzt das BufferedImage auf image.
104     */
105         public void setImage(BufferedImage image) {
106                 this.image = image;
107         }
108
109         //------------------------------------------------------
110         // Methoden, die die Ein-/Ausgabe der Rohdaten regeln.
111         //------------------------------------------------------
112
113     /**
114         Veranlasst den ImageCreator, den übergebenen TCPReceiver
            anzustoßen, sodass dieser Daten vom Netzwerk bezieht.
115         Diese Daten werden angepasst und in rawdata abgelegt.
```

```
116     */
117         public void readRawdataFromTCPReceiver(TCPReceiver receiver){
118                 receiver.run();
119                 rawdata = adjustAndSplayValues(receiver.
                    getSignedRawdata(),ADJUSTVALUESBY,SPLAYVALUESBY);
120         }
121
122         //-------------------------------------------------------
123         // Methoden, die direkt auf dem BufferedImage arbeiten.
124         //-------------------------------------------------------
125
126         /**
127         Kapselt die Funktionalität, die Java bietet, um ein
            RenderedImage in der Datei filename im gegebenen Format format
             abzuspeichern.
128          */
129         private static void writeImageToFile(RenderedImage img, String
             filename, String format) throws IOException{
130                 Iterator writers = ImageIO.getImageWritersByFormatName
                    (format);
131                 ImageWriter writer = null;
132
133         try {
134                         writer = (ImageWriter) writers.next();
135                 } catch (NoSuchElementException e){
136                         e.printStackTrace();
137                 }
138
139                 RandomAccessFile outfile = new RandomAccessFile(
                    filename, "rw");
140                 ImageOutputStream outStream = ImageIO.
                    createImageOutputStream(outfile);
141                 writer.setOutput(outStream);
142
143                 writer.write(img);
144                 outfile.close();
145         }
146
147     /**
148         Ruft writeImageToFile() für das BufferedImage dieses
            ImageCreator auf.
149         Der Dateiname muss hierbei als Parameter übergeben werden, das
             Dateiformat ist statisch auf PNG gesetzt.
150     */
151     public void writeImageToFileAsPNG(String filename){
152         try {
153             writeImageToFile(image, filename, "png");
154         } catch (IOException e) {
155             e.printStackTrace();
156         }
157     }
158
159     /**
160         Erzeugt aus den Rohdaten ein BufferedImage und speichert
            dieses in image ab.
161     */
162         public void generateBufferedImage(){
163
164                 image = new BufferedImage(GBcamclient.IMAGEWIDTH,
                    GBcamclient.IMAGEHEIGHT, BufferedImage.TYPE_INT_ARGB);
165
166                 for (int j = 0; j < GBcamclient.IMAGEHEIGHT; j++){
```

```
167                             for (int i = 0; i < GBcamclient.IMAGEWIDTH; i
                                ++){
168                                     short colorValue = rawdata[j*
                                        GBcamclient.IMAGEWIDTH + i];
169                                     int argbColorValue = (0xFF<<24) | (
                                        colorValue<<16) | (colorValue<<8) | (
                                        colorValue);
170                                     image.setRGB(i,j, argbColorValue);
171                             }
172                     }
173             }
174 }
```

7.2.3 Datei GBcamclient.java

```
1  package base;
2
3  /**
4      Dies ist die Hauptklasse des Clients.
5   */
6  public class GBcamclient {
7
8          /*
9          Bilddimensionen
10     */
11         public static final int IMAGEWIDTH = 128;
12         public static final int IMAGEHEIGHT = 123;
13
14
15     /*
16         Belichtungszeit.
17         Dieser Wert sollte (Short.MAX_VALUE/10000) nie überschreiten,
           da sonst die Kommunikation mit dem uC scheitern wird.
18     */
19         private float exposureTime;
20
21     /*
22         GBcamclient muss die beiden Klassen kennen, die die
           Hauptarbeit des Clients verrichten.
23     */
24         private TCPReceiver receiver;
25         private ImageCreator creator;
26
27
28         /**
29         Erzeugt einen neuen GBcamclient.
30          */
31         public GBcamclient() {
32                 receiver = new TCPReceiver(this);
33                 creator = new ImageCreator();
34         }
35
36         //-------------------------------------------------------
37         // Getter und Setter
38         //-------------------------------------------------------
```

```
39
40          /**
41             Liefert den ImageCreator dieses GBcamclient zurück.
42           */
43          public ImageCreator getCreator() {
44                  return creator;
45          }
46
47          /**
48          Liefert den TCPReceiver dieses GBcamclient zurück.
49           */
50          public TCPReceiver getReceiver() {
51                  return receiver;
52          }
53
54          /**
55          Liefert die Belichtungszeit zurück, die dieser GBcamclient
            verwendet.
56           */
57          public float getExposureTime() {
58                  return exposureTime;
59          }
60
61          /**
62          Setzt die Belichtungszeit.
63           */
64          public void setExposureTime(float exposureTime) {
65                  this.exposureTime = exposureTime;
66          }
67  }
```

7.2.4 Datei GBcamclientGUI.java

```
1   package gui;
2
3   import java.awt.BorderLayout;
4   import java.awt.FlowLayout;
5   import java.awt.event.ActionEvent;
6   import java.awt.event.ActionListener;
7   import java.awt.event.ItemEvent;
8   import java.awt.event.ItemListener;
9   import java.io.File;
10  import java.net.InetAddress;
11  import java.net.UnknownHostException;
12  import java.util.Hashtable;
13
14  import javax.swing.JCheckBox;
15  import javax.swing.JFileChooser;
16  import javax.swing.JLabel;
17  import javax.swing.JMenuItem;
18  import javax.swing.JOptionPane;
19  import javax.swing.JSlider;
20  import javax.swing.border.TitledBorder;
21
22  import base.GBcamclient;
```

```
23 import base.Utils;
24
25 /**
26      Dies ist die Klasse, die das GUI sowie die Anbindung zum Backend
        enthält.
27  */
28 public class GBcamclientGUI extends javax.swing.JFrame {
29
30
31     private javax.swing.JPanel imagePanel;
32
33     private javax.swing.JMenuBar jMenuBar1;
34     private javax.swing.JMenu fileMenu;
35     private JMenuItem saveFileMenuItem;
36     private javax.swing.JMenu optionsMenu;
37     private JMenuItem serverOptionsMenuItem;
38
39     private javax.swing.JPanel controlPanel;
40     private javax.swing.JPanel controlPanelLeft;
41     private JSlider exposureTimeSlider;
42     private JCheckBox useAutoScalingCheckbox;
43     private javax.swing.JPanel controlPanelRight;
44     private javax.swing.JButton acquireImageButton;
45
46
47         /*
48              Der GBcamclient, der die eigentliche Arbeit erledigt.
49          */
50         private GBcamclient camclient;
51
52     /**
53          Erzeugt ein neues GBcamclientGUI.
54      */
55     public GBcamclientGUI() {
56         setTitle("GBcamclient");
57
58         camclient = new GBcamclient();
59
60         /*
61              Stellt sicher, dass bereits zum Programmstart Bilddaten
                vorhanden sind, die angezeigt werden können. Hierzu wird
                ein "Testbild" generiert.
62          */
63         camclient.getCreator().setRawdata(Utils.generateRawdataSample
            ());
64
65         initComponents();
66     }
67
68     /**
69          Liefert den GBcamclient dieses GUIs zurück.
70      */
71         public GBcamclient getCamclient() {
72                 return camclient;
73         }
74
75     /**
76          Liefert true, wenn die automatische Skalierung des
             angezeigten Bildes aktiviert ist, false sonst.
77      */
78     public boolean useAutoScaling(){
79         return useAutoScalingCheckbox.isSelected();
```

```
80      }
81
82      /**
83           Diese Methode wird im Konstruktor aufgerufen, um alle GUI-
             Elemente zu initialisieren.
84       */
85      private void initComponents() {
86
87          imagePanel = new ImagePanel(this);
88
89          jMenuBar1 = new javax.swing.JMenuBar();
90          fileMenu = new javax.swing.JMenu();
91          saveFileMenuItem = new JMenuItem("Save File ...");
92          optionsMenu = new javax.swing.JMenu();
93          serverOptionsMenuItem = new JMenuItem("Server Options ...");
94
95          controlPanel = new javax.swing.JPanel();
96          controlPanelLeft = new javax.swing.JPanel();
97          exposureTimeSlider = new JSlider(0,150);
98          useAutoScalingCheckbox = new JCheckBox("Use Auto-Scaling",
            true);
99          controlPanelRight = new javax.swing.JPanel();
100         acquireImageButton = new javax.swing.JButton();
101
102         getContentPane().setLayout(new BorderLayout(5,15));
103
104         setDefaultCloseOperation(javax.swing.WindowConstants.
            EXIT_ON_CLOSE);
105         addWindowListener(new java.awt.event.WindowAdapter() {
106             public void windowClosing(java.awt.event.WindowEvent evt)
                {
107                 exitForm(evt);
108             }
109         });
110
111         //---------------------------------------------------------
112         // imagePanel
113         //---------------------------------------------------------
114             getContentPane().add(imagePanel,BorderLayout.CENTER);
115         //---------------------------------------------------------
116
117
118             //-----------------------------------------------------
119             // controlPanel
120             //-----------------------------------------------------
121                 controlPanel.setLayout(new java.awt.GridLayout(1, 2,
                    10, 10));
122
123                 //--- controlPanelLeft ----
124                         controlPanelLeft.setLayout(new java.awt.
                            GridLayout(2, 1, 5, 1));
125
126                         //--- exposureTimeSlider --
127                                 Hashtable labelTable = new Hashtable
                                    (3);
128                                 labelTable.put(new Integer(0), new
                                    JLabel("0.0"));
129                     labelTable.put(new Integer(50), new JLabel("0.5"))
                        ;
130                                 labelTable.put(new Integer(100), new
                                    JLabel("1.0"));
```

```
131                     labelTable.put(new Integer(150), new JLabel("1.5")
                        );
132                                 exposureTimeSlider.setLabelTable(
                                    labelTable);
133                                 exposureTimeSlider.setPaintLabels(true
                                    );
134                     exposureTimeSlider.setMajorTickSpacing(25);
135                     exposureTimeSlider.setMinorTickSpacing(5);
136                     exposureTimeSlider.setPaintTicks(true);
137                                 exposureTimeSlider.setToolTipText("The
                                     exposure time which the camera shall
                                    use" +
138                             " (best leave this as it is)");
139                     exposureTimeSlider.setBorder(new TitledBorder("
                        Exposure time"));
140                                 controlPanelLeft.add(
                                    exposureTimeSlider);
141                             //--------------------------
142
143                 //--- useAutoScalingCheckbox --
144                     useAutoScalingCheckbox.setToolTipText("If Auto-
                        Scaling is on, the image size will" +
145                             " adapt to the window size");
146                     useAutoScalingCheckbox.addItemListener(new
                        ItemListener(){
147                         public void itemStateChanged(ItemEvent event){
148                             useAutoScalingCheckboxItemStateChanged(
                                event);
149                         }
150                     });
151                     controlPanelLeft.add(useAutoScalingCheckbox);
152                 //-------------------------------
153
154                             controlPanel.add(controlPanelLeft);
155                         //-------------------------------
156
157
158                     //--- controlPanelRight ----
159                             controlPanelRight.setLayout(new FlowLayout
                                (FlowLayout.CENTER, 10, 10));
160
161                             //--- acquireImageButton --------------
162                                 acquireImageButton.setText("Acquire
                                    Image");
163                                 acquireImageButton.addActionListener(
                                    new java.awt.event.ActionListener() {
164                                     public void actionPerformed(java.
                                        awt.event.ActionEvent evt) {
165                                         acquireImageButtonActionPerformed
                                            (evt);
166                                     }
167                                 });
168
169                                 controlPanelRight.add(
                                    acquireImageButton);
170                             //--------------------------
171
172                             controlPanel.add(controlPanelRight);
173                         //--------------------------
174
175                 getContentPane().add(controlPanel, BorderLayout.SOUTH)
                    ;
```

```
176             //-------------------------------------------------------
177
178
179         //-------------------------------------------------------
180         // jMenuBar1
181         //-------------------------------------------------------
182             //--- fileMenu ----------------------
183                 fileMenu.setText("File");
184
185                 //--- saveFileMenuItem ---------------
186                     saveFileMenuItem.addActionListener(new
                        ActionListener() {
187                         public void actionPerformed(ActionEvent event)
                            {
188                             saveFileMenuItemActionPerformed(event);
189                         }
190                     });
191                     fileMenu.add(saveFileMenuItem);
192                 //-------------------------------------
193                 jMenuBar1.add(fileMenu);
194             //-------------------------------------
195
196
197             //--- optionsMenu ---------------------
198                 optionsMenu.setText("Options");
199
200                 //--- serverOptionsMenutItem ---------
201                     serverOptionsMenuItem.addActionListener(new
                        ActionListener(){
202                         public void actionPerformed(ActionEvent event){
203                             serverOptionsMenutItemActionPerformed(event
                                );
204                         }
205                     });
206
207                     optionsMenu.add(serverOptionsMenuItem);
208                 //-------------------------------------
209
210                 jMenuBar1.add(optionsMenu);
211             //-------------------------------------
212
213             setJMenuBar(jMenuBar1);
214         //-------------------------------------------------------
215
216         pack();
217     }
218
219     /**
220          Wird aufgerufen, wenn im "Options"-Menü der Unterpunkt "
             Server Options" gewählt wird.
221          Es wird ein Dialog angezeigt, in dem IP-Adresse sowie Port
             des Servers eingestellt werden können.
222      */
223     protected void serverOptionsMenutItemActionPerformed(ActionEvent
        event) {
224         String result;
225         String message = "Please enter the IP address of the server
            and the TCP port";
226         String address = "192.168.99.66:6666";
227         boolean isCorrect = false;
228
229         /*
```

```
230                 Folgende Schleife prüft die im Dialogfenster
                    eingegebenen Werte auf Korrektheit, und öffnet es
                    solange erneut, bis der Dialog abgebrochen wird ("Cancel
                    ") oder die Werte korrekt sind.
231             */
232             do {
233                 result = JOptionPane.showInputDialog(this, message,
                    address);
234
235                 /*
236                      Wenn der Dialog abgebrochen wurde, müssen auch die
                         Einstellungen nicht geprüft werden.
237                  */
238                 if (result == null) break;
239
240                 /*
241                      Prüft mittels regulärem Ausdruck, ob das Format des
                         eingegebenen Strings syntaktisch korrekt ist.
242                  */
243                 isCorrect = result.matches("(\\d{1,3}\\.){3}\\d{1,3}:\\d
                    {1,5}");
244
245                 /*
246                     Falls schon klar ist, dass der eingegebene String
                        falsch ist, muss nicht weiter überprüft werden.
247                     Andernfalls wird der String in 5 Ganzzahlwerte zerlegt
                         und jeweils überprüft, ob der zulässige Wertebereich
                        eingehalten wurde (also max. 255 in der IP-Adresse und
                         max. 65535 für den Port).
248                  */
249                 if (isCorrect){
250                     String[] addressAndPort = result.split("\\.|:");
251                     byte[] newAddress = new byte[4];
252                     for (int i=0; i< 4; i++){
253                         int v = Integer.parseInt(addressAndPort[i]);
254                         if (v > 255) {
255                             isCorrect = false;
256                             break;
257                         }
258                         newAddress[i] = (byte)v;
259                     }
260                     int newPort = Integer.parseInt(addressAndPort[4]);
261                     isCorrect &= newPort <= 0xFFFF;
262
263                     /*
264                          Falls bis hierhin kein Fehler gefunden wurde,
                             werden die Einstellungen übernommen.
265                      */
266                     if (isCorrect){
267                         try {
268                             camclient.getReceiver().setServerAddress(
                                InetAddress.getByAddress(newAddress));
269                         } catch (UnknownHostException e) {
270                             e.printStackTrace();
271                         }
272                         camclient.getReceiver().setServerPort(newPort);
273                     }
274                 }
275
276                 /*
277                      Falls die Eingabe inkorrekt war, soll beim nächsten
                         Öffnen des Dialogs der Benutzer darauf hingewiesen
```

```
                    werden. Dementsprechend wird Aufforderung zur Eingabe
                     angepasst; die fehlerhafte Eingabe wird erneut im
                    Eingabefeld gezeigt, sodass der Benutzer sie leicht
                    auf Fehler prüfen und ggf. ändern kann.
278                 */
279                 if (!isCorrect){
280                     message = "You entered an invalid address/port!\
                    nPlease use \"a.b.c.d:e\" notation and try again";
281                     address = result;
282                 }
283             } while (!isCorrect);
284
285             repaint();
286         }
287
288         /**
289              Wird aufgerufen, wenn im "File"-Menü der Unterpunkt "Save
                 File" gewählt wird.
290              Es wird ein Dialog angezeigt, in dem der Benutzer Verzeichnis
                 und Dateiname für das zu speichernde Bild auswählen kann.
                 Das Dateiformat ist hierbei stets PNG.
291          */
292         private void saveFileMenuItemActionPerformed(ActionEvent event) {
293             JFileChooser fileChooser = new JFileChooser((File)null);
294             fileChooser.setDialogTitle("Save File ...");
295             fileChooser.setDialogType(JFileChooser.SAVE_DIALOG);
296             int returnValue = fileChooser.showSaveDialog(this);
297             if (returnValue == JFileChooser.APPROVE_OPTION){
298                 System.out.println("Datei: " + fileChooser.getSelectedFile
                    () );
299                 camclient.getCreator().writeImageToFileAsPNG(fileChooser.
                    getSelectedFile().getAbsolutePath());
300             }
301
302         }
303
304         /**
305              Diese Methode wird aufgerufen, wenn sich der Status der
                 Checkbox "Use Auto-Scaling" ändert (wenn also ein Häkchen
                 gesetzt oder entfernt wurde).
306              Beim Neuzeichnen des Anzeigefensters wird der Wert der
                 Checkbox ausgelesen und ggf. das angezeigte Bild skaliert oder
                 nicht. Damit die Änderung des Checkbox-Status sich sofort
                 bemerkbar macht, muss also nur ein Neuzeichnen des kompletten
                 Fensters ausgelöst werden.
307          */
308         private void useAutoScalingCheckboxItemStateChanged(ItemEvent
            event) {
309             this.repaint();
310         }
311
312         /**
313              Diese Methode wird aufgerufen, wenn der Button "Acquire Image
                 " gedrückt wurde.
314              Es wird der momentane Wert des Sliders "Exposure Time"
                 ausgelesen und als Startkommando an den Server geschickt. Das
                  Bild wird empfangen und sofort umgewandelt und angezeigt. Da
                  das Umwandeln erst beginnt, nachdem das komplette Bild
                 empfangen wurde, blockiert der GUI-Thread solange.
315          */
316         private void acquireImageButtonActionPerformed(java.awt.event.
            ActionEvent evt) {
```

```
317             /*
318                  Der Kommentar kann entfernt werden, um Debug-Ausgaben zu
                     erhalten.
319              */
320             /*
321                 System.out.println("exposureTimeSlider hat den Wert " +
                    exposureTimeSlider.getValue());
322                 System.out.println("als tatsächlicher float-wert wird
                    angenommen: " + (float)exposureTimeSlider.getValue()/100);
323             */
324             camclient.setExposureTime((float)exposureTimeSlider.getValue()
                /100);
325             camclient.getCreator().readRawdataFromTCPReceiver(camclient.
                getReceiver());
326             camclient.getCreator().generateBufferedImage();
327             repaint();
328         }
329
330
331         /**
332              Wird aufgerufen, wenn das Hauptfenster geschlossen wird und
                 schließt dann das Programm.
333          */
334         private void exitForm(java.awt.event.WindowEvent evt) {
335             System.exit(0);
336         }
337
338
339         /**
340              Startet das Programm.
341          */
342         public static void main(String args[]) {
343             new GBcamclientGUI().show();
344         }
345     }
```

7.2.5 Datei ImagePanel.java

```
1  package gui;
2
3  import java.awt.Color;
4  import java.awt.Dimension;
5  import java.awt.FlowLayout;
6  import java.awt.Graphics;
7  import java.awt.Image;
8
9  import javax.swing.JPanel;
10
11 import base.Utils;
12
13 /**
14     Diese Klasse dient der Darstellung des Bildes in einem separaten
       JPanel.
15  */
16 public class ImagePanel extends JPanel {
```

```
17
18          private GBcamclientGUI clientGUI;
19
20      Image imageToPaint;
21
22      float scaleFactor = 2f;
23
24
25          /**
26               Erzeugt ein neues ImagePanel.
27           */
28          public ImagePanel(GBcamclientGUI clientGUI) {
29                  this.clientGUI = clientGUI;
30
31                  clientGUI.getCamclient().getCreator().setRawdata(Utils
                    .generateRawdataSample());
32          imageToPaint = clientGUI.getCamclient().getCreator().getImage
            ();
33
34                  setLayout(new FlowLayout(FlowLayout.CENTER));
35
36          setPreferredSize(new Dimension((int)(imageToPaint.getWidth(
            this)*scaleFactor),(int)(imageToPaint.getHeight(this)*
            scaleFactor)));
37
38          setVisible(true);
39          }
40
41      /**
42           Überschreibt die paint()-Methode von JPanel.
43           Das Bild wird vom ImageCreator bezogen und angezeigt.
44       */
45          public void paint(Graphics g){
46                  Image imageToPaint = clientGUI.getCamclient().
                    getCreator().getImage();
47
48
49          /*
50               Falls automatisches Skalieren aktiv ist:
51           */
52          if (clientGUI.useAutoScaling()){
53
54              /*
55                   Bestimmt, ob die angezeigte Bildgröße durch die Höhe
                     oder die Breite des anzeigendes ImagePanels
                     eingeschränkt ist.
56               */
57              float percentageOfPanelWidth = (float)imageToPaint.
                getWidth(this)/(float)getSize().getWidth();
58              float percentageOfPanelHeight = (float)imageToPaint.
                getHeight(this)/(float)getSize().getHeight();
59
60              /*
61                   Passt die Bildgröße an die o.g. beschränkende
                     Dimension an.
62               */
63              if (percentageOfPanelWidth > percentageOfPanelHeight){
64                  scaleFactor = 1/percentageOfPanelWidth;
65              } else {
66                  scaleFactor = 1/percentageOfPanelHeight;
67              }
68          } else { // automatisches Skalieren deaktiviert
```

```
69              scaleFactor = 1f;
70          }
71
72          /*
73               Verschiebt das Bild in die Mitte des Panels
74           */
75                  g.translate((int)(getSize().getWidth()/2 -
                    imageToPaint.getWidth(this)/2*scaleFactor),(int)(
                    getSize().getHeight()/2 - imageToPaint.getHeight(this)
                    /2*scaleFactor));
76
77          /*
78               Das Bild wird (skaliert) gezeichnet.
79           */
80                  g.drawImage(imageToPaint, 0, 0,(int)(imageToPaint.
                    getWidth(this)*scaleFactor), (int)(imageToPaint.
                    getHeight(this)*scaleFactor),Color.black, this);
81          }
82  }
```

7.2.6 Datei Utils.java

```
1   package base;
2
3   /**
4       Diese Klasse enthält einige Hilfsfunktionen, die sonst nirgends
        untergebracht werden konnten.
5    */
6   public abstract class Utils {
7
8           /**
9            Wandelt einen (als unsigned interpretierten) byte-Wert in
             einen short-Wert um.
10           */
11          public static short toUnsignedShort(byte value){
12                  return (short)((value & 0x7f) + (value < 0 ? 128 : 0))
                    ;
13          }
14
15      /**
16           Erzeugt ein "Testbild" (sh. ImageCreator).
17       */
18          public static short[] generateRawdataSample(){
19                  short[] rawdatasample = new short[GBcamclient.
                    IMAGEWIDTH * GBcamclient.IMAGEHEIGHT];
20                  for(int i=0; i < rawdatasample.length; i++){
21                          rawdatasample[i] = (short) (i % 127);
22                  }
23                  return rawdatasample;
24          }
25  }
```

7.3 Erklärung

Hiermit erklären wir, Daniel Pähler (Matrikelnr. 200210851) und Thomas Wilbert (Matrikelnr. 201210251), die vorliegende Studienarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Koblenz, den _____

_____	_____
Daniel Pähler	Thomas Wilbert