Fachhochschule Bielefeld
University of Applied Sciences
Fachbereich Ingenieurwissenschaften und Mathematik

Thema: EVALUIERUNG UND IMPLEMENTIERUNG EINES I^2C BUS-SYSTEMS

Name: Dipl.-Ing. (FH) Christian Benjamin Ries

Datum: 27. Dezember 2009

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung liefert ein mögliches Konzept für die Verwendung der I^2C Bustechnologie.

Es soll die Kommunikation des I^2C an einem Beispiel veranschaulicht werden. Dafür werden verschiedene Prinzipien der Hardware- und Softwaretechnik verwendet. Es soll mit dem Prinzip der Analog-Digital-Wandlung (*Analog-Digital-Converter*, ADC) (s. Abs. 2.5) eine Leuchtdiode (LED) gedimmt werden. Das Dimmen wird durch eine Pulsweitenmodulation (PWM) umgesetzt (s. Abs. 2.4). Die Kommunikation zwischen diesen zwei Komponenten soll über ein Bussystem erfolgen, dem sogenannten I^2C Bus (s. Abs. 2.2). Abbildung 1 veranschaulicht diese Umsetzung.

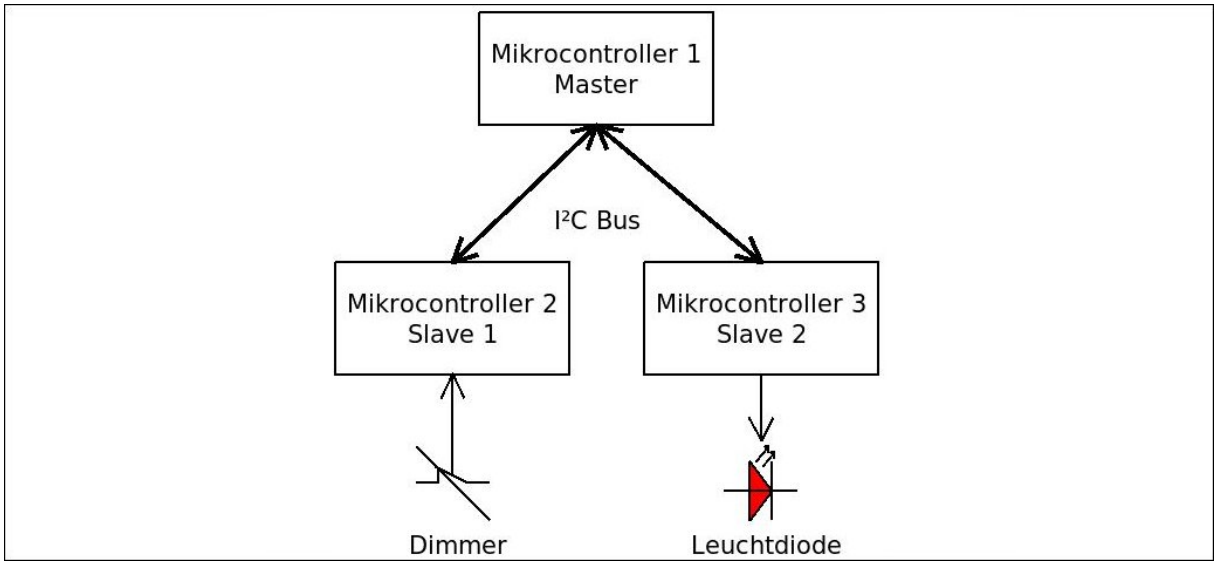

Abbildung 1.1: Prinzipdarstellung der Aufgabenstellung

2 Grundlagen

Heutige Rechnersysteme sind entweder nach der Von-Neumann (s. Abs. 2.1.1) oder Harvard (s. Abs. 2.1.2) Struktur aufgebaut. In beiden Architekturen müssen die verbauten Komponenten miteinander Informationen austauschen. Dies geschieht mit Bussystemen.

2.1 Rechnerstrukturen

2.1.1 Von-Neumann-Rechnerstruktur

Die Von-Neumann-Rechnerstruktur besitzt drei Busse, die nachfolgend erläutert werden:

Adressbus Jede Komponente besitzt eine Adresse um diese Komponente ansteuern zu können. Der Adressbus ist für die Adressierung zuständig. In 32-bit Systemen besteht dieser Bus aus 32 Leitungen. Bei 64-bit Systemen dementsprechend aus 64 Leitungen.

Steuerbus Der Steuerbus ist für die Steuerrung der Komponenten zuständig. Ein Speicherbaustein ist für die Bereitstellung von Speicherplatz zuständig. Dieser Speicherplatz kann lesend oder schreibend angesprochen werden. Der Steuerbus regelt den Modus der Verarbeitung von Speicherstellen.

Datenbus Mit dem Datenbus werden Daten zwischen den Komponenten transferiert.

Durch diesen Aufbau werden ausführende oder informationshaltende Datenblöcke über den selben Bus übertragen, dem Datenbus. Es ist dem Programm überlassen wie die Daten gehandhabt werden. Die Von-Neumann Rechnerstruktur wird bei Prozessoren der x86 Familie eingesetzt, sprich den handelsüblichen Prozessoren von Intel oder AMD.

2.1.2 Harvard-Rechnerstruktur

Bei der Harvard-Rechnerstruktur werden die auszuführenden Datenblöcke von den informationsbehafteten Datenblöcken getrennt übertragen. Diese seperat über einen Bus angebunden. Abbildung 2.1.2 verdeutlicht dies. Der Programmspeicher, sowie der Datenspeicher befinden sich an einzelnen Bussen. Dies hat zur Folge das beide Komponenten jeweils adressiert werden müssen. Es muss im ersten Schritt die Komponente addressiert werden, in einem weiteren werden die Daten übertragen. Der Steuerbus ist wie bei der Von-Neumann-Rechnerstruktur vorhanden und regelt die Ansteuerung der Komponenten. Sollen Daten gespeichert werden, dann wird die Datenspeicherkomponente durch den Adressbus adressiert, der Steuerbus regelt die Komponente auf die Funktionalität *Abspeichern*, daraufhin wird der Adressbus als Datenbus verwendet und die Daten zum Abspeichern übertragen.

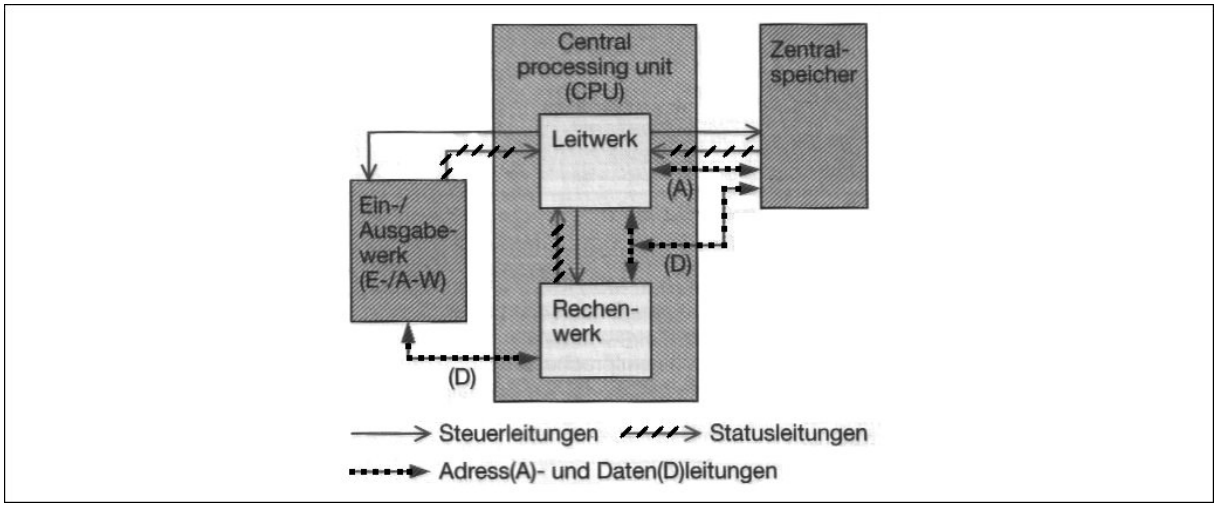

Abbildung 2.1: Grundelemente der Von-Neumann-Rechnerstruktur [ITBuch]

Die Harvard-Rechnerstruktur wird in den AVR-Mikrocontrollern der Firma Atmel angewandt, die in dieser Arbeit Verwendung finden.

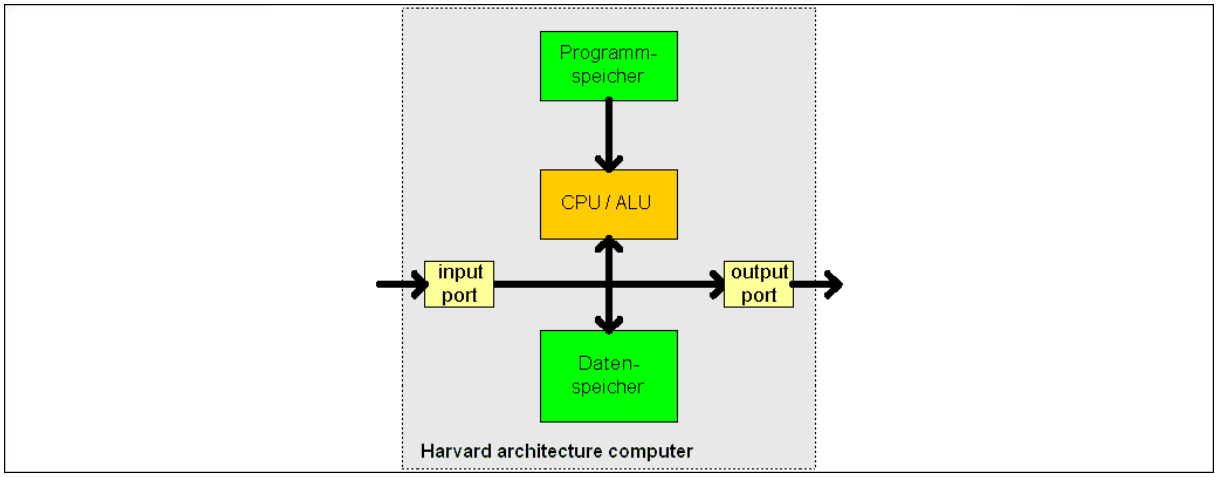

Abbildung 2.2: Die Harvard-Rechnerstruktur [Sprut]

2.2 I^2C Bustechnologie

2.2.1 Aufbau

Die vorherigen genannten Busse benötigen eine bestimmte Anzahl an Leitungen, in heutigen Rechnerarchitekturen sind dies mindestens 32 bzw. 64 Leitungen pro Bus (Adress-/Steuer- und Datenbus). Dem gegenüber stehen Bustechnologien zur Verfügung die mit zwei Leitungen auskommen. Zu dieser Bustechnologie gehört der I^2C Bus. Dieser Bus wird auch Zweidrahtbussystem genannt und ist als *Master-Slave*-Bus konzipiert. Der *Master* ist

für die Ansteuerung und Kommunikation aller Komponenten zuständig. Es können mehr als ein *Master* am Bus angeschlossen sein.

Die zwei Leitungen ersetzen die zuvor genannten Busse und vereinen die Funktionalitäten zu einem Ganzen. Eine Leitung dient der Kommunikation und die zweite Leitung der Signaltaktung[1]. Die Leitung zur Kommunikation wird *Serial Data Line* (SDA) und die Leitung zur Signaltaktung *Serial Clock Line* (SCL) genannt.

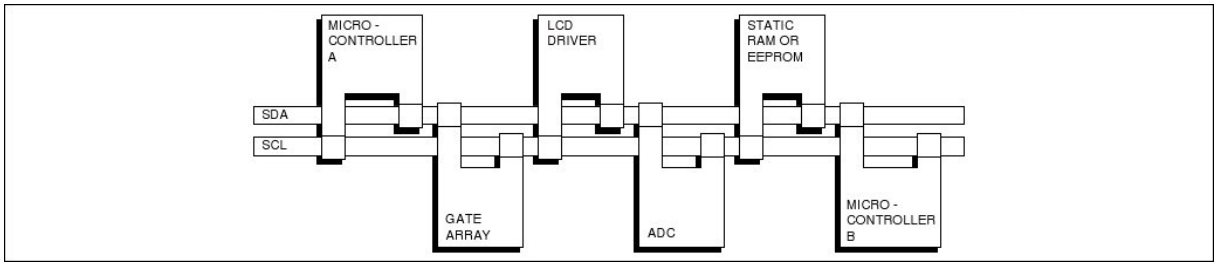

Abbildung 2.3: Beispiel eines I^2C Aufbaus mit zwei Mikrocontrollern [UM10204]

In Abbildung 2.2.1 ist ein Beispielaufbau veranschaulicht. An die zwei Leitungen *SCL* und *SDA* können in dieser Arbeit bis zu 128 Komponenten angeschlossen werden. Diese Anzahl resultiert daraus, dass ein Mikrocontroller von Atmel verwendet wird, der eine Adressierung mit 7-Bit-Adressen erlaubt (s. Abs. 2.3).

Die Leitungen *SDA* und *SCL* müssen mit einem Widerstand (s. Abb. 2.2.1) an die Versorgungsspannung angeschlossen werden.

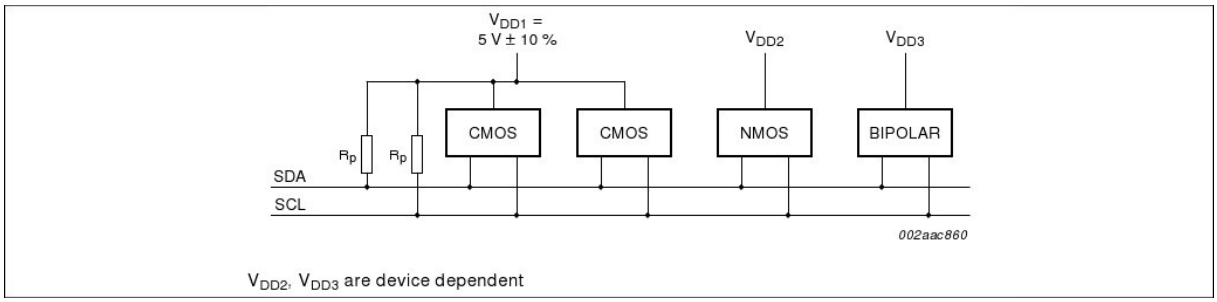

Abbildung 2.4: Beispielschaltung mit *Pull-Up* Widerständen R_p (s. [UM10204, Seite 8])

Die *Pull-Up* Widerstände liegen zwischen einem maximalen und minimalen Wert (s. [UM10204, Seite 42]) und können durch die Diagramme in Abbildung 2.2.1 gewählt werden.

Für das Verständnis dieses Diagramm sind folgende Informationen wichtig. Durch die Weiterentwicklung des I^2C Standards haben sich mehrere Übertragungsmodus entwickelt, die eine höhe Übetragungsgeschwindigkeit besitzen. Der Mikrocontroller in dieser Arbeit kann maximal mit einer Geschwindigkeit von 400 kHz (s. [ATmega8, Seite 163]) arbeiten. Folgende Auflistung liefert die im Standard definierten Modus:

***Standard-mode* (Sm)** mit einer Geschwindigkeit von bis zu 100 kbit/s.

[1] Die Signaltaktung dient zur synchronen Verarbeitung der Befehle, die über die zweite Leitung übertragen werden. Dadurch ist sichergestellt, dass jede mit dem Bus verbundene Komponenten zur selben Zeit schaltet und angesteuert wird.

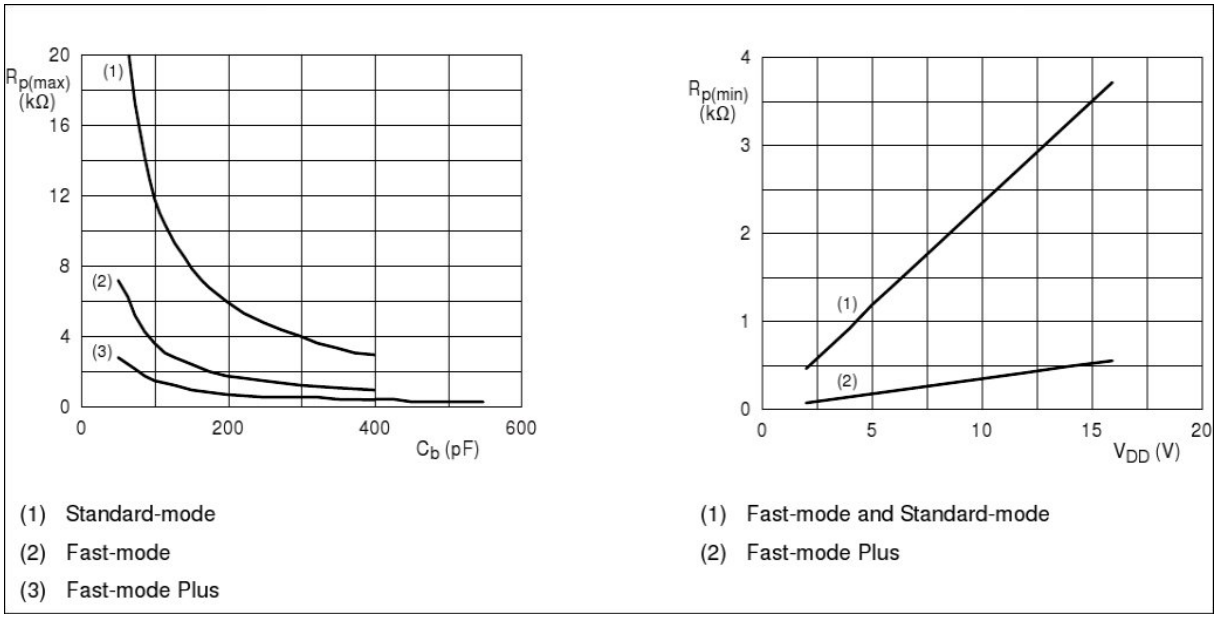

Abbildung 2.5: Ermittlung des Widerstandwertes R_p (s. [UM10204, Seite 43])

Fast-Mode **(Fm)** mit einer Geschwindigkeit von bis zu 400 kbit/s.

Fast-mode Plus **(Fm+)** mit einer Geschwindigkeit von bis zu 1 Mbit/s.

High-speed mode **(Hs-mode)** mit einer Gechwindigkeit von bis zu 3.4 Mbit/s.

Der Mikrocontroller in dieser Arbeit unterstützt somit den *Sm* und *Fm* Modus. Ein R_p Wert zwischen $0.5k\Omega$ und 7kΩ unterstützt somit beide Modus.

2.2.2 Kommunikationsdefinition

Die Kommunikation wird durch vier Modus abgedeckt, die im folgenden aufgelistet sind (s. [ATmega8, Seite 179-190]):

Master Transmitter Mode **(MT)** In diesem Modus sendet der *Master* Daten an einen *Slave*, der *Slave* muss dafür im SR Modus sein.

Master Receiver Mode **(MR)** Der *MR* Modus dient zum Empfangen von Daten eines *Slave* der im *ST* Modus ist.

Slave Receiver Mode **(SR)** Wenn der *Slave* Daten von einem *Master* empfängt wird in den *SR* Modus beim *Slave* gewechselt.

Slave Transmitter Mode **(ST)** Der *Slave* sendet im *ST* Modus Daten an einem *Master* im *MR* Modus.

Weiterhin existieren einige Statuswerte die den Zustand der Kommunikation wiederspiegeln. Diese sind im Anhang zu finden (s. A.1, A.2, A.3 und A.4).

In Abbildung 2.2.2 wird beispielhaft die Initialisierung und Datenübertragung eines *Slave* erläutert.

Der *Slave* erhält vom *Master* die Nachricht zum Starten einer neuen Kommunikation, dies wird durch **S** eingeleitet. Darauf folgt die eindeutige Adresse des *Slave* mit dem Kommuniziert werden soll. Das **R** symbolisiert den Modus in den gewechselt werden soll, hier *ST* Modus, somit werden Daten an den *Master* im *MR* Modus gesendet. Wenn diese Initialisierung erfolgreich war, wird eine Bestätigung vom *Slave* an den *Master* gesendet, **A**. Daraufhin sendet der *Slave* die Daten und der *Master* bestätigt diese. Dies geht so lange bis der *Master* die Kommunikation mit `P` beendet.

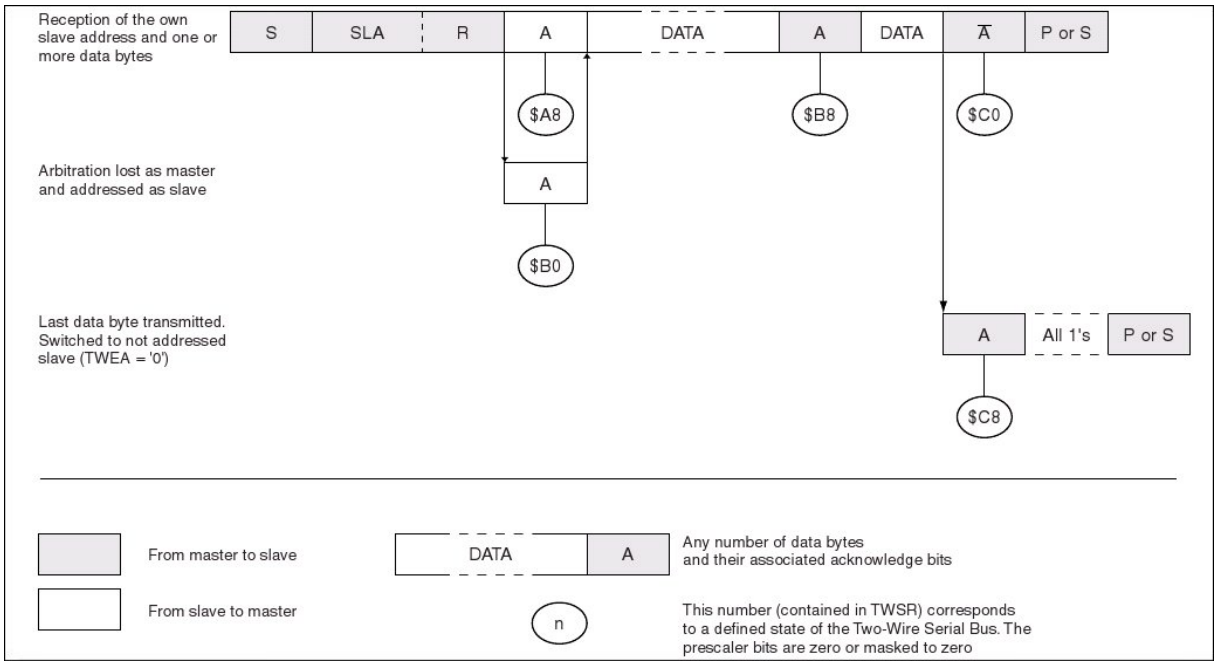

Abbildung 2.6: Kommunikationsverlauf eines *Slave* im *ST* Modus

2.3 Mikrocontroller

Diese Arbeit verwendet mehrere Mikrocontroller der Firma Atmel. Fast alle von Atmel gelieferten Mikrocontroller besitzen eine I^2C Unterstützung. Aus lizenzrechtlichen Gründen wird dieses System *Two-wire Serial Interface* (TWI) genannt, das allerdings vollkompatibel zum I^2C Standard [UM10204] ist. Im Abschnitt 3 wird auf die Implementierung tiefer eingegangen und beschrieben welche Mikrocontroller zum Einsatz kommen bzw. zusätzlich verwendet werden können.

2.4 Pulsweitenmodulation

Die Ausgänge eines Mikrocontrollern können nur zwei Status einnehmen, An (1) und Aus (0). Eine Eins bedeutet ein Spannungswert in Höhe der Versorgungsspannung, hier 5 Volt und eine Null ist gleichbedeutend mit der Masse von 0 Volt. In Abbildung 2.4 sind verschiedene *LED* Kennlinien abgebildet. Eine *LED* hat keinen linearen Verlauf und leuchtet mit einer Versorgungsspannung zwischen $2 - 3.5$ Volt. Bei einer Spannungsversorgung unter 2 Volt

ist die *LED* aus und über 3.5 Volt kann die *LED* je nach verträglicher Stromstärke durchbrennen. Die 5 Volt, die am Ausgang des Mikrocontrollers anliegen, müssen in den Bereich von $2 - 3.5$ Volt skaliert werden.

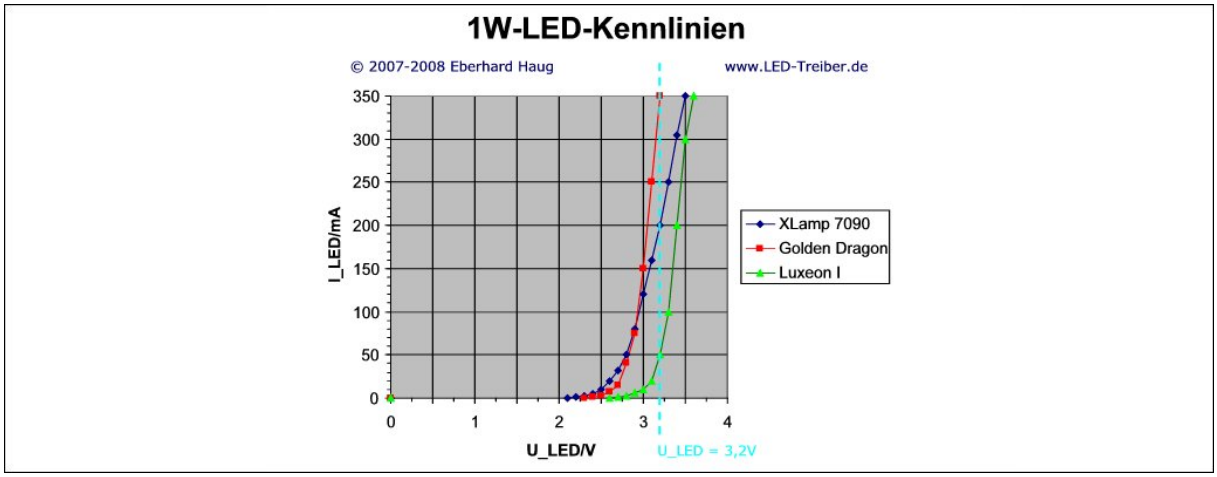

Abbildung 2.7: LED-Kennlinien im Vergleich

Bei der Pulsweitenmodulation wird die Ein- und Ausschaltzeit eines Rechtecksignals bei fester Grundfrequenz variiert. Das Verhältnis $\frac{t_{ein}}{t_{ein}+t_{aus}}$ wird als Tastverhältnis (*Duty Cycle*, DC) bezeichnet. Das Tastverhältnis ist ein Wert zwischen 0 und 1 (s. [MCWeb, Pulsweitenmodulation]).

Folgende Herleitung verdeutlicht, dass der Mittelwert eine wichtige Größe bei der Pulsweitenmodulation spielt.

$$U_{Mittelwert} = \frac{1}{T} \int_0^T u(t)dt \tag{2.1}$$

$$U_{Mittelwert} = \frac{1}{T} \int_0^{t_{ein}} U_{ein}dt + \frac{1}{T} \int_{t_{ein}}^T U_{aus}dt \tag{2.2}$$

$$U_{Mittelwert} = U_{aus} + (U_{ein} - U_{aus}) * \frac{t_{ein}}{t_{ein} + t_{aus}} \tag{2.3}$$

$U_{aus} := 0$ Volt und $U_{ein} := 5$ Volt. Daraus resultiert folgende Kurzschreibweise:

$$U_{Mittelwert} = V_{CC} * DutyCycle \tag{2.4}$$

Durch das zeitlich abhängige Ein- und Ausschalten der *LED* kann ein Dimmen dieser moduliert werden.

2.5 Analog-Digital-Wandler

Der Mikrocontroller in dieser Arbeit besitzt einen Analog-Digital-Wandler (*Analog-Digital-Converter*, ADC), der einen analogen Spannungswert in einen Zahlenwert umwandelt. Für diesen Zweck ist ein 10 Bit (s. [ATmega8, Seite 196]) breiter Speicherbereich vorhanden, d.h. dass eine Spannung von 5 Volt in 2^{10} Werte unterteilt wird. Bei 5 Volt entspricht ein Wert den Bereich von $4,88 * 10^{-3}$ Volt. Diese Unterteilung wird Quantisierung genannt.

Für eine genaue *ADC* Umwandlung muss eine Referenzspannung (V_{Ref}) angegeben werden. Die zu messende Spannung (V_{In}) kann im Vorfeld mit der Formel 2.5 (s. [ATmega8, Seite 205] berechnet werden.

$$ADC = \frac{V_{In} * 1024}{V_{Ref}} \tag{2.5}$$

3 Umsetzung

3.1 Konzept

Abbildung 3.1 zeigt den Aufbau der elektronischen Schaltung, die in dieser Arbeit entwickelt wurde. Die Widerstände R_4 und R_5 entsprechen dem Widerstand R_p. In Abbildung 3.1 ist das Schema auf einem Reizbrett aufgebaut und im Betrieb. Der *Master* übernimmt die Kommunikation zwischen *Slave* 1 und 2. Der *Slave* 1 steuert eine *LED* an und dimmt diese dem quantisierten Wert von *Slave* 2 nach. Ein höherer Wert bedeutet eine heller leuchtende *LED*.

In dieser Arbeit wird der ATMega8 verwendet.

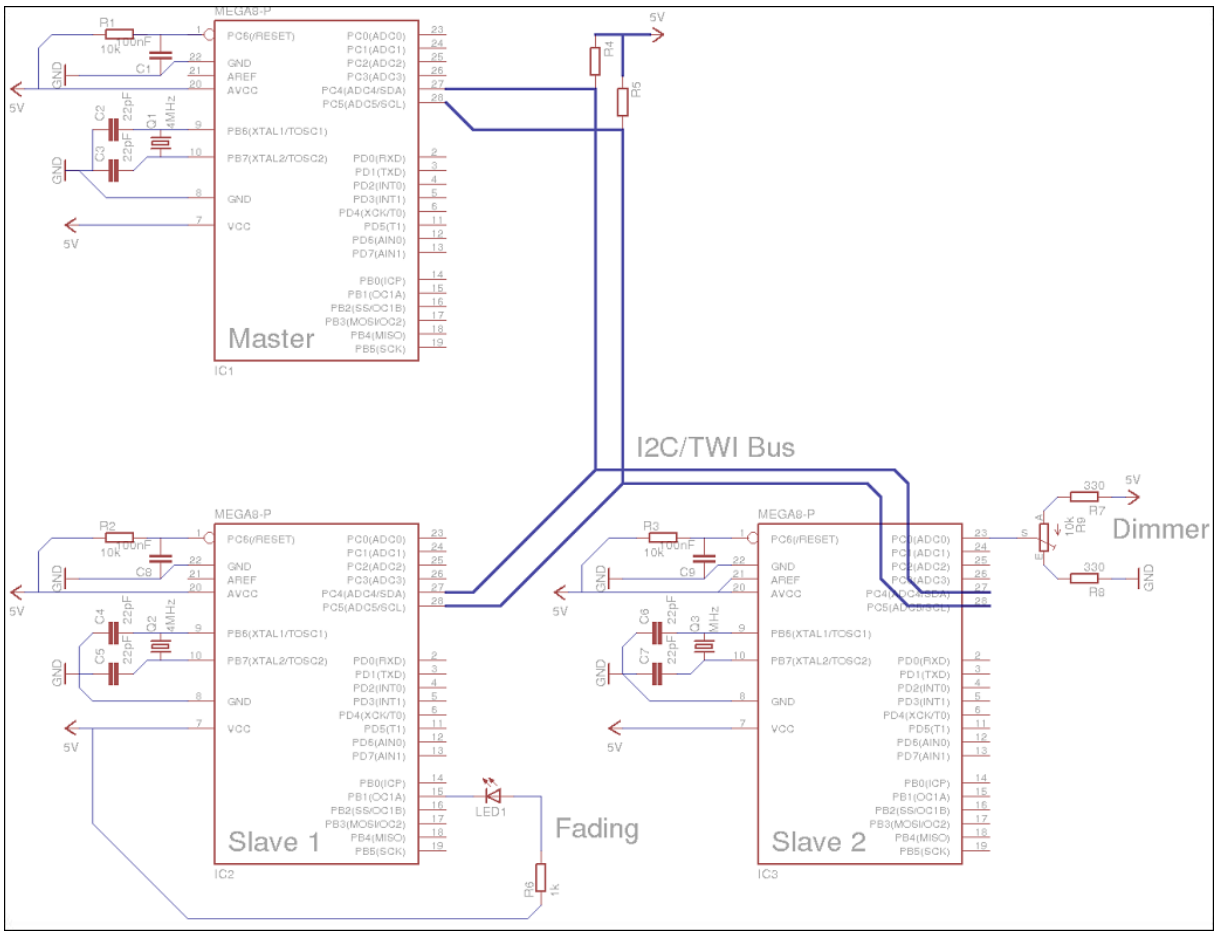

Abbildung 3.1: Schema der elektrischen Verbindungen

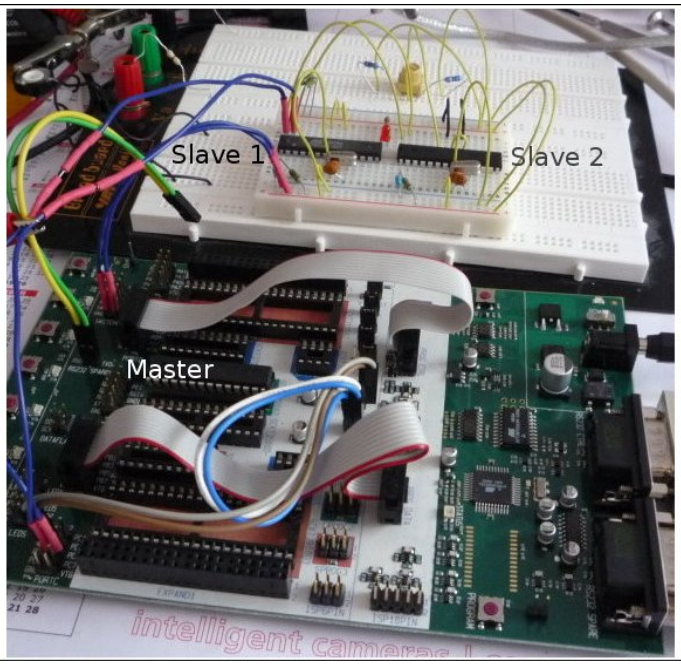

Abbildung 3.2: Reizbrettaufbau der Schematik mit dem STK-500

3.2 Implementierung

Diese Arbeit umfasst die Softwareentwicklung von drei Programmem, jeweils ein Programm für einen Mikrocontroller. Die Mikrocontroller haben verschiedene Aufgaben die auf den selben Funktionsumfang zurückgreifen.

Folgende Aufgaben werden von den Mikrocontrollern abgedeckt.

3.2.1 Master

Der Master übernimmt die Kommunikationssteuerung auf dem I^2C Bus und transferiert die Informationen zwischen *Slave* 1 und 2.

3.2.2 Slave 1

Dieser *Slave* empfängt Daten vom *Master*. Die Daten beinhalten den Wert der *ADC* Umwandlung zum Dimmen der *LED*. *Slave* 1 übernimmt die *PWM* der *LED*.

In Listing B.10 ist eine Software basierende PWM implementiert. In Zeile 27 wird der *Timer0* des ATmega8 initialisiert. Folgende Einstellungen werden im Listing B.6 vorgenommen:

Zeile 10 *Timer0* einschalten

Zeile 11 Den *Prescaler*[1] des *Timer0* auf 8 einstellen.

Die Schaltung in dieser Arbeit ist mit einer Frequenz von $f = 4MHz$ getaktet. Somit benötigt ein Takt

$$T = \frac{1}{f} \tag{3.1}$$

$$T = \frac{1}{4MHz} = 250ns \tag{3.2}$$

Durch den *Prescaler* (P) mit dem Wert 8 erhöht sich dieser Wert auf

$$T(P) = \frac{1 * P}{4MHz} \tag{3.3}$$

$$T(8) = \frac{8}{4MHz} = 2\mu s \tag{3.4}$$

Der *Timer0* vom ATmega8 besitzt ein 8 Bit großes Register, das sich nach jedem Takt um Eins erhöht. Durch den *P* mit 8, geschieht dies erst alle 8 Takte. Nachdem das 8 Bit Register 255 erreicht hat, wird es beim nächsten Hochzählen auf Null zurückgesetzt. Dieses Zurücksetzen wird mit einem Interrupt signalisiert, dem sogenannten *Overflow*-Interrupt (s. Zeile 16 im Listing B.10. Diese Interrupt-Routine übernimmt die *PWM* Umsetzung. Sobald der Zähler in Zeile 18 größer oder gleich dem Maximalwert ist, wird die *LED* eingeschaltet, ansonsten ausgeschaltet. Durch den Maximalwert kann die *LED* gedimmt werden.

Die Zeit zwischen einem *Overflow* kann mit der Formel 3.4 wie folgt errechnet werden

$$T_{Overflow} = \frac{P}{f} * 2^8 \tag{3.5}$$

$$T_{Overflow} = \frac{8}{4MHz} * 2^8 = 512\mu s \tag{3.6}$$

Durch die Formel 3.6 ist zu erkennen, dass die *LED* mit voller Kraft leuchtet, wenn der Ausgang zur *LED* die komplette Zeit lang auf 1 gesetzt ist. Eine Halbierung dieser Zeit, lässt die *LED* nur die halbe Kraft leuchten.

Im Schaltungsschema 3.1 ist an der *LED* ein Widerstand mit dem Wert von $1k\Omega$ angeschlossen. Da der ATmega8 in der Regel nur $20mA$ an Ausgangsstrom liefert (s. [ATmega8, Seite 242]), für kurze Zeit sind auch $40mA$ möglich (kann zu Schäden führen) und so der Strom begrenzt wird. Es muss bei einer Reihenschaltung von Diode und Widerstand der Arbeitspunkt der Diode ermittelt werden. Die Kennlinie der Diode ist in Abbildung 3.2.2 zu sehen.

Die Versorgungsspannung beträgt $U_0 = 5$ Volt. Der Widerstand $1k\Omega$, daraus resultiert mit Gleichung 3.7 die grafische Lösung in Abbildung 3.2.2.

[1] Ein Prescaler teilt die Taktrate durch einen Faktor der Basis 2.

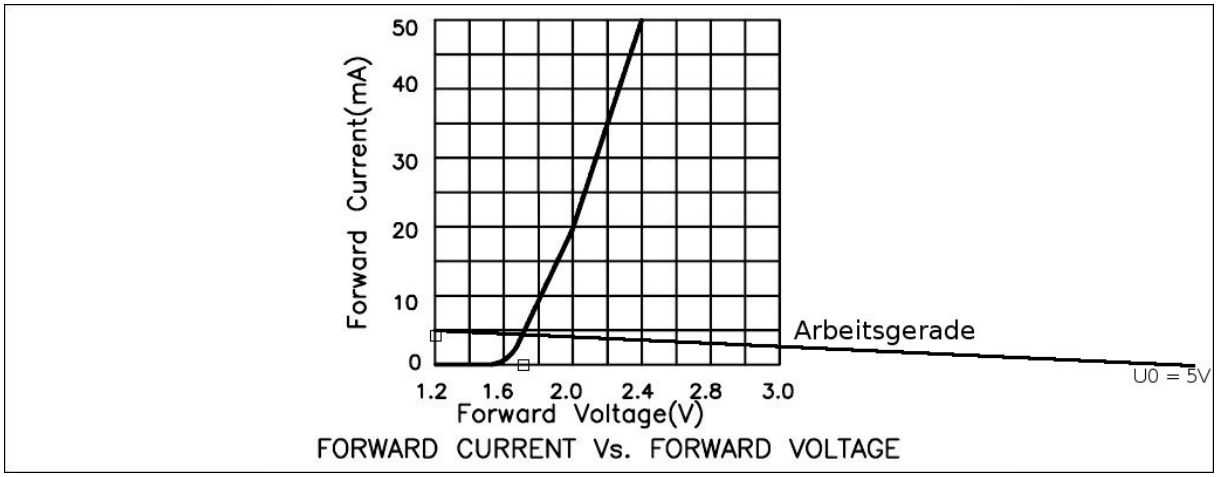

Abbildung 3.3: Diodenkennlinie der roten *LED* BrightRed L-934HD

$$I_{ArbeitspunktDiode} = \frac{U_0}{R} = \frac{5Volt}{1k\Omega} = 5mA \tag{3.7}$$

Aus Abbildung 3.2.2 ist zu entnehmen, dass die Diode zwischen einem Spannungswert von 1.6 und 1.9 Volt leuchtet, wenn über die volle Zeit aus Gleichung 3.6 die *LED* eingeschaltet wird.

Mit der Gleichung 2.3 lässt sich das *DC* Verhältnis berechnen. Die *LED* ist bei einem Spannungswert von $U_{Mittelwert} = 1,6Volt$ ausgeschaltet, daraus ergibt sich für *DC*

$$DC = \frac{U_{Mittelwert}}{U_{Aus} + (U_{Ein} - U_{Aus}} = \frac{1.6}{0 + (5 - 0)} = 0.32 = 32\% \tag{3.8}$$

Das Verhältnis zwischen angeschaltet und ausgeschaltet beträgt 0.32. Der Ausgang der LED muss über den Zeitraum von $t_{5Volt} = 512\mu s$, 68% dieser deaktiviert sein, damit die LED nicht mehr leuchtet.

$$T_{Komplett} = 512\mu s \hat{=} 5Volt \tag{3.9}$$

$$T_{Aus} = T_{Komplett} * (100 - DC) = 512\mu s * (100 - 0.32) = 348\mu s \hat{=} 1.6Volt \tag{3.10}$$

Durch die Gleichungen 3.9 und 3.10 kann ein Dreisatz hergeleitet werden. Dadurch kann der Wert für die *PWM* berechnet und im Listing B.10 verwendet werden

$$\frac{U_0}{U_x} = \frac{Counter_{An}}{Counter_{Aus}} \tag{3.11}$$

3.2.3 Slave 2

Der *Slave* 2 ist für die Ermittlung des digitalen Wertes der *ADC* Umwandlung zuständig. Der Wert wird an den *Master* über den I^2C Bus übertragen.

Der ATmega8 unterstützt zwei Modus für die *ADC* Umwandlung. Dies sind der *Free Running* (FR) oder *Single Conversion* (SC) Modus. Beim *FR* Modus wird kontinuierlich nach der Initialisierung ein Interrupt ausgelöst, sobald eine neue Umwandlung von einem analogen zu einem digitalen Wert vorliegt. Der *SC* Modus wird für jede Umwandlung neugestartet und es wird solange gewartet bis ein Ergebnis vorliegt. In Listing B.8 befindet sich die Funktion $adc\_readChannel(uint8\_t)$ mit der eine Umwandlung im *SC* Modus angeworfen wird. Der Rückgabewert dieser Funktion ist ein gültiger *ADC* Wert. Die Funktion übernimmt die Initialisierung des *ADC* Wandlers und stellt folgende Konfiguration ein:

Zeile 13 Einstellung des Kanals, es sind acht Eingänge vorhanden, allerdings nur einer zu einem bestimmten Zeitpunkt nutzbar.

Zeile 14 Das Ergebnis der Umwandlung soll linksbündig in den zwei Registern *ADCL* und *ADCH* stehen (L = *low*, H = *high*), dadurch wird es möglich das 10 Bit Ergebnis als 8 Bit Ergebnis zu nutzen.

Zeile 15 Den *ADC* Umwandler einschalten.

Zeile 17 Der *Prescaler* für die *ADC* Umwandlung wird auf 32 gestellt.

Zeile 20 Den Modus auf *SC* einstellen.

Zeile 21 Solange Warten bis ein Ergebnis vorliegt.

Es werden insgesamt 32 Messungen durchgeführt, um einem Rauschen vorzubeugen. Eine Mittellung dieser 32 Werte liefert ein ausreichend gutes Ergebnis.

Literaturverzeichnis

[ATmega8] ATmega8 Datasheet: **8-bit AVR with 8K Bytes In-System Prorgammable Flash**, 2009

[ITBuch] Hübscher, Petersen, Rathgeber, Richter, Scharf: **IT-Handbuch, IT-Systemelektroniker/-in, Fachinformatiker/-in**, 2. Auflage, Westermann Schulbuchverlag GmbH, Braunschweig, 2001

[MCWeb] Andreas Schwarz, http://www.mikrocontroller.net

[Sprut] Bredendiek: **PIC-Prozessoren - Grundlagen**, http://www.sprut.de

[UM10204] NXP founded by Philips: I^2C**-bus specification and user manual**, Rev. 03, 2007

A Anhang - I^2C Statuswerte

A.1 Master Transmitter Mode

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+W	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+W or	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
		Load SLA+R	0	0	1	X	SLA+R will be transmitted; Logic will switch to Master Receiver mode
0x18	SLA+W has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x20	SLA+W has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x28	Data byte has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x30	Data byte has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x38	Arbitration lost in SLA+W or data bytes	No TWDR action or	0	0	1	X	Two-wire Serial Bus will be released and not addressed Slave mode entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free

A.2 Master Receiver Mode

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					
		To/from TWDR	To TWCR				Next Action Taken by TWI Hardware
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+R	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+R or	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
		Load SLA+W	0	0	1	X	SLA+W will be transmitted Logic will switch to Master Transmitter mode
0x38	Arbitration lost in SLA+R or NOT ACK bit	No TWDR action or	0	0	1	X	Two-wire Serial Bus will be released and not addressed Slave mode will be entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free
0x40	SLA+R has been transmitted; ACK has been received	No TWDR action or	0	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	0	0	1	1	Data byte will be received and ACK will be returned
0x48	SLA+R has been transmitted; NOT ACK has been received	No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x50	Data byte has been received; ACK has been returned	Read data byte or	0	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	0	0	1	1	Data byte will be received and ACK will be returned
0x58	Data byte has been received; NOT ACK has been returned	Read data byte or	1	0	1	X	Repeated START will be transmitted
		Read data byte or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		Read data byte	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset

A.3 Slave Receiver Mode

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x60	Own SLA+W has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x68	Arbitration lost in SLA+R/W as Master; own SLA+W has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x70	General call address has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x78	Arbitration lost in SLA+R/W as Master; General call address has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x80	Previously addressed with own SLA+W; data has been received; ACK has been returned	Read data byte or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	X	0	1	1	Data byte will be received and ACK will be returned
0x88	Previously addressed with own SLA+W; data has been received; NOT ACK has been returned	Read data byte or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		Read data byte or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		Read data byte or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		Read data byte	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0x90	Previously addressed with general call; data has been received; ACK has been returned	Read data byte or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	X	0	1	1	Data byte will be received and ACK will be returned
0x98	Previously addressed with general call; data has been received; NOT ACK has been returned	Read data byte or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		Read data byte or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		Read data byte or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		Read data byte	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0xA0	A STOP condition or repeated START condition has been received while still addressed as Slave	No action	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
			0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
			1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
			1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free

A.4 Slave Transmitter Mode

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0xA8	Own SLA+R has been received; ACK has been returned	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received
		Load data byte	X	0	1	1	Data byte will be transmitted and ACK should be received
0xB0	Arbitration lost in SLA+R/W as Master; own SLA+R has been received; ACK has been returned	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received
		Load data byte	X	0	1	1	Data byte will be transmitted and ACK should be received
0xB8	Data byte in TWDR has been transmitted; ACK has been received	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received
		Load data byte	X	0	1	1	Data byte will be transmitted and ACK should be received
0xC0	Data byte in TWDR has been transmitted; NOT ACK has been received	No TWDR action or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		No TWDR action or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		No TWDR action or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		No TWDR action	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0xC8	Last data byte in TWDR has been transmitted (TWEA = "0"); ACK has been received	No TWDR action or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		No TWDR action or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		No TWDR action or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		No TWDR action	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free

B Anhang - Quelltexte

B.1 I²C/TWI Master

```
#ifndef _I2CMASTER_H
#define _I2CMASTER_H   1
/*************************************************************************
* Title:    C include file for the I2C master interface
*           (i2cmaster.S or twimaster.c)
* Author:   Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
* File:     $Id: i2cmaster.h,v 1.10 2005/03/06 22:39:57 Peter Exp $
* Software: AVR-GCC 3.4.3 / avr-libc 1.2.3
* Target:   any AVR device
* Usage:    see Doxygen manual
*************************************************************************/

#ifdef DOXYGEN
/**
 @defgroup pfleury_ic2master I2C Master library
 @code #include <i2cmaster.h> @endcode

 @brief I2C (TWI) Master Software Library

 Basic routines for communicating with I2C slave devices. This single master
 implementation is limited to one bus master on the I2C bus.

 This I2c library is implemented as a compact assembler
 software implementation of the I2C protocol which runs on any AVR
 (i2cmaster.S) and as a TWI hardware interface for all AVR with built-in
 TWI hardware (twimaster.c). Since the API for these two implementations is
 exactly the same, an application can be linked either against the software
 I2C implementation or the hardware I2C implementation.

 Use 4.7k pull-up resistor on the SDA and SCL pin.

 Adapt the SCL and SDA port and pin definitions and
 eventually the delay routine in the module i2cmaster.S to your target
 when using the software I2C implementation !

 Adjust the  CPU clock frequence F_CPU in twimaster.c or in
 the Makfile when using the TWI hardware implementaion.

 @note
    The module i2cmaster.S is based on the Atmel Application
    Note AVR300, corrected and adapted
    to GNU assembler and AVR-GCC C call interface.
    Replaced the incorrect quarter period delays found in AVR300 with
    half period delays.

 @author Peter Fleury pfleury@gmx.ch  http://jump.to/fleury

 @par API Usage Example
  The following code shows typical usage of this library, see example test_i2cmaster.c

 @code

 #include <i2cmaster.h>


 #define Dev24C02  0xA2      // device address of EEPROM 24C02, see datasheet

 int main(void)
```

```
59  {
60      unsigned char ret;
61
62      i2c_init();                             // initialize I2C library
63
64      // write 0x75 to EEPROM address 5 (Byte Write)
65      i2c_start_wait(Dev24C02+I2C_WRITE);     // set device address and write mode
66      i2c_write(0x05);                        // write address = 5
67      i2c_write(0x75);                        // write value 0x75 to EEPROM
68      i2c_stop();                             // set stop conditon = release bus
69
70
71      // read previously written value back from EEPROM address 5
72      i2c_start_wait(Dev24C02+I2C_WRITE);     // set device address and write mode
73
74      i2c_write(0x05);                        // write address = 5
75      i2c_rep_start(Dev24C02+I2C_READ);       // set device address and read mode
76
77      ret = i2c_readNak();                    // read one byte from EEPROM
78      i2c_stop();
79
80      for(;;);
81  }
82  @endcode
83
84 */
85 #endif /* DOXYGEN */
86
87 /**@{*/
88
89 #if (__GNUC__ * 100 + __GNUC_MINOR__) < 304
90 #error "This␣library␣requires␣AVR-GCC␣3.4␣or␣later,␣update␣to␣newer␣AVR-GCC␣compiler␣!"
91 #endif
92
93 #include <avr/io.h>
94
95 /** defines the data direction (reading from I2C device) in i2c_start(),i2c_rep_start() */
96 #define I2C_READ    1
97
98 /** defines the data direction (writing to I2C device) in i2c_start(),i2c_rep_start() */
99 #define I2C_WRITE   0
100
101
102 /**
103  @brief initialize the I2C master interace. Need to be called only once
104  @param  void
105  @return none
106  */
107 extern void i2c_init(void);
108
109
110 /**
111  @brief Terminates the data transfer and releases the I2C bus
112  @param void
113  @return none
114  */
115 extern void i2c_stop(void);
116
117
118 /**
119  @brief Issues a start condition and sends address and transfer direction
120
121  @param    addr address and transfer direction of I2C device
122  @retval   0   device accessible
123  @retval   1   failed to access device
124  */
125 extern unsigned char i2c_start(unsigned char addr);
```

```
126
127
128 /**
129  @brief Issues a repeated start condition and sends address and transfer direction
130
131  @param   addr address and transfer direction of I2C device
132  @retval  0 device accessible
133  @retval  1 failed to access device
134  */
135 extern unsigned char i2c_rep_start(unsigned char addr);
136
137
138 /**
139  @brief Issues a start condition and sends address and transfer direction
140
141  If device is busy, use ack polling to wait until device ready
142  @param    addr address and transfer direction of I2C device
143  @return   none
144  */
145 extern void i2c_start_wait(unsigned char addr);
146
147
148 /**
149  @brief Send one byte to I2C device
150  @param    data  byte to be transfered
151  @retval   0 write successful
152  @retval   1 write failed
153  */
154 extern unsigned char i2c_write(unsigned char data);
155
156
157 /**
158  @brief    read one byte from the I2C device, request more data from device
159  @return   byte read from I2C device
160  */
161 extern unsigned char i2c_readAck(void);
162
163 /**
164  @brief    read one byte from the I2C device, read is followed by a stop condition
165  @return   byte read from I2C device
166  */
167 extern unsigned char i2c_readNak(void);
168
169 /**
170  @brief    read one byte from the I2C device
171
172  Implemented as a macro, which calls either i2c_readAck or i2c_readNak
173
174  @param    ack 1 send ack, request more data from device<br>
175                0 send nak, read is followed by a stop condition
176  @return   byte read from I2C device
177  */
178 extern unsigned char i2c_read(unsigned char ack);
179 #define i2c_read(ack)  (ack) ? i2c_readAck() : i2c_readNak();
180
181
182 /**@}*/
183 #endif
```

Listing B.1: twimaster.h

```
1 /*************************************************************************
2 * Title:    I2C master library using hardware TWI interface
3 * Author:   Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
4 * File:     $Id: twimaster.c,v 1.3 2005/07/02 11:14:21 Peter Exp $
5 * Software: AVR-GCC 3.4.3 / avr-libc 1.2.3
6 * Target:   any AVR device with hardware TWI
```

```
* Usage:    API compatible with I2C Software Library i2cmaster.h
************************************************************************/
#include <inttypes.h>
#include <compat/twi.h>

#include "i2cmaster.h"

/* define CPU frequency in Mhz here if not defined in Makefile */
#ifndef F_CPU
#define F_CPU 4000000UL
#endif

/* I2C clock in Hz */
#ifndef SCL_CLOCK
#define SCL_CLOCK  100000L
#endif


/*************************************************************************
 Initialization of the I2C bus interface. Need to be called only once
*************************************************************************/
void i2c_init(void)
{
  /* initialize TWI clock: 100 kHz clock, TWPS = 0 => prescaler = 1 */

  TWSR = 0;                         /* no prescaler */
  TWBR = ((F_CPU/SCL_CLOCK)-16)/2;  /* must be > 10 for stable operation */

}/* i2c_init */


/*************************************************************************
  Issues a start condition and sends address and transfer direction.
  return 0 = device accessible, 1= failed to access device
*************************************************************************/
unsigned char i2c_start(unsigned char address)
{
    uint8_t   twst;

        // send START condition
        TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_START) && (twst != TW_REP_START)) return 1;

        // send device address
        TWDR = address;
        TWCR = (1<<TWINT) | (1<<TWEN);

        // wail until transmission completed and ACK/NACK has been received
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_MT_SLA_ACK) && (twst != TW_MR_SLA_ACK) ) return 1;

        return 0;
}/* i2c_start */

/*************************************************************************
 Issues a start condition and sends address and transfer direction.
 If device is busy, use ack polling to wait until device is ready

```

```
 Input:   address and transfer direction of I2C device
*****************************************************************************/
void i2c_start_wait(unsigned char address) {
    uint8_t   twst;
    while ( 1 )
    {
            // send START condition
            TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_START) && (twst != TW_REP_START)) continue;

        // send device address
        TWDR = address;
        TWCR = (1<<TWINT) | (1<<TWEN);

        // wail until transmission completed
        while(!(TWCR & (1<<TWINT)));

        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst == TW_MT_SLA_NACK )||(twst ==TW_MR_DATA_NACK) )
        {
            /* device busy, send stop condition to terminate write operation */
                TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

                // wait until stop condition is executed and bus released
                while(TWCR & (1<<TWSTO));

            continue;
        }
        //if( twst != TW_MT_SLA_ACK) return 1;
        break;
     }
}/* i2c_start_wait */


/*****************************************************************************
 Issues a repeated start condition and sends address and transfer direction

 Input:   address and transfer direction of I2C device

 Return:  0 device accessible
          1 failed to access device
*****************************************************************************/
unsigned char i2c_rep_start(unsigned char address)
{
    return i2c_start( address );
}/* i2c_rep_start */


/*****************************************************************************
 Terminates the data transfer and releases the I2C bus
*****************************************************************************/
void i2c_stop(void)
{
    /* send stop condition */
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

        // wait until stop condition is executed and bus released
        while(TWCR & (1<<TWSTO));

}/* i2c_stop */
```

```
141
142
143 /*************************************************************************
144   Send one byte to I2C device
145
146   Input:    byte to be transfered
147   Return:   0 write successful
148             1 write failed
149 *************************************************************************/
150 unsigned char i2c_write( unsigned char data )
151 {
152     uint8_t   twst;
153
154         // send data to the previously addressed device
155         TWDR = data;
156         TWCR = (1<<TWINT) | (1<<TWEN);
157
158         // wait until transmission completed
159         while(!(TWCR & (1<<TWINT)));
160
161         // check value of TWI Status Register. Mask prescaler bits
162         twst = TW_STATUS & 0xF8;
163         if( twst != TW_MT_DATA_ACK) return 1;
164         return 0;
165
166 }/* i2c_write */
167
168
169 /*************************************************************************
170  Read one byte from the I2C device, request more data from device
171
172  Return:  byte read from I2C device
173 *************************************************************************/
174 unsigned char i2c_readAck(void)
175 {
176         TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
177         while(!(TWCR & (1<<TWINT)));
178
179     return TWDR;
180
181 }/* i2c_readAck */
182
183
184 /*************************************************************************
185  Read one byte from the I2C device, read is followed by a stop condition
186
187  Return:  byte read from I2C device
188 *************************************************************************/
189 unsigned char i2c_readNak(void)
190 {
191         TWCR = (1<<TWINT) | (1<<TWEN);
192         while(!(TWCR & (1<<TWINT)));
193
194     return TWDR;
195
196 }/* i2c_readNak */
```

Listing B.2: twimaster.c

B.2 I²C/TWI Slave

```
1 /**
2  * Christian Benjamin Ries, Christian.Ries@linux-sources.de
3  * 29. July 2009
4  */
5 #ifndef __TWISLAVE_H__
```

```
6  #define __TWISLAVE_H__
7
8  /**
9   * USER CONFIGURATION
10  * Size of the buffer.
11  */
12 #define BUFFER_SIZE 8
13 #define BUFFER_MAX_SIZE 0xff
14
15 /**
16  * The rxbuffer and txbuffer are used as a I2C-EEPROM simulation.
17  * You should define the buffer size with the buffer_size definition above.
18  */
19 volatile uint8_t rxbuffer[BUFFER_SIZE];
20 volatile uint8_t txbuffer[BUFFER_SIZE];
21
22 /**
23  * Address value of the data.
24  */
25 volatile uint8_t buffer_addr;
26
27 #include <stdint.h>
28
29 void init_twi_slave (uint8_t addr);
30
31 #endif // __TWISLAVE_H__
```

Listing B.3: twislave.h

```
1  /**
2   * Initialized by Uwe Grosse-Wortmann (uwegw)
3   * Status: Testphase, keine Garantie fuer ordnungsgemaesse Funktion!
4   * letze Aenderungen:
5   * 23.03.07 Makros fuer TWCR eingefuegt. Abbruch des Sendens,
6   * wenn der TXbuffer komplett gesendet wurde.
7   * 24.03.07 verbotene Buffergroessen abgefangen
8   * 25.03.07 noetige externe Bibliotheken eingebunden
9   * Abgefangene Fehlbedienung durch den Master:
10  * - Lesen ueber die Grenze des txbuffers hinaus
11  * - Schreiben ueber die Grenzen des rxbuffers hinaus
12  * - Angabe einer ungueltigen Schreib/Lese-Adresse
13  * - Lesezuggriff, ohne vorher Leseadresse geschrieben zu haben
14  *
15  * Modifications
16  * Christian Benjamin Ries, Christian.Ries@linux-sources.de
17  * 29. July 2009, standard text formatting
18  *                changed german to english
19  */
20 #include <util/twi.h>
21 #include <avr/interrupt.h>
22
23 #include "twislave.h"
24
25 #if (__GNUC__ * 100 + __GNUC_MINOR__) < 304
26         #error "This␣library␣requires␣AVR-GCC␣3.4.5␣or␣later,␣" \
27                                 "update␣to␣newer␣AVR-GCC␣compiler␣!"
28 #endif
29
30 #if (BUFFER_SIZE >= BUFFER_MAX_SIZE)
31         #error Buffer zu gross gewaehlt! Maximal BUFFER_MAX_SIZE Bytes erlaubt.
32 #endif
33
34 #if (BUFFER_SIZE < 2)
35         #error Buffer muss mindestens zwei Byte gross sein!
36 #endif
37
38 /**
```

```
39  * Initializes the TWI-Interface.
40  * Should be called before other TWI operations used.
41  * @param addr Address of the slave.
42  */
43 void init_twi_slave (uint8_t addr) {
44         TWAR = addr;
45         TWCR &= ~(1<<TWSTA)|(1<<TWSTO);
46         TWCR |= (1<<TWEA) | (1<<TWEN)|(1<<TWIE);
47         buffer_addr = BUFFER_MAX_SIZE;
48 }
49
50 // ACK nach empfangenen Daten senden/ ACK nach gesendeten Daten erwarten
51 #define TWCR_ACK TWCR   = (1<<TWEN) | (1<<TWIE) | (1<<TWINT) \
52         | (1<<TWEA) | (0<<TWSTA) | (0<<TWSTO) | (0<<TWWC);
53 // NACK nach empfangenen Daten senden/ NACK nach gesendeten Daten erwarten
54 #define TWCR_NACK TWCR  = (1<<TWEN) | (1<<TWIE) | (1<<TWINT) \
55         | (0<<TWEA) | (0<<TWSTA) | (0<<TWSTO) | (0<<TWWC);
56 // switched to the non adressed slave mode...
57 #define TWCR_RESET TWCR = (1<<TWEN) | (1<<TWIE) | (1<<TWINT) \
58         | (1<<TWEA) | (0<<TWSTA) | (0<<TWSTO) | (0<<TWWC);
59
60 /**
61  * Interrupt routine for TWI handling.
62  * It depends on the values in the TWSR
63  * register which functionally will execute.
64  * @param TWI_vect Interrupt vector of the TWI interface
65  */
66 ISR (TWI_vect) {
67         uint8_t data = 0;
68
69         switch (TW_STATUS) {
70                 // SLA+W received, ACK returned
71                 case TW_SR_SLA_ACK:
72                         TWCR_ACK;
73                         buffer_addr = BUFFER_MAX_SIZE;
74                 break;
75
76                 // 0x80, data received, ACK returned
77                 case TW_SR_DATA_ACK:
78                         // TWDR, data register
79                         data = TWDR;
80
81                         if (buffer_addr == BUFFER_MAX_SIZE) {
82                                 if(data <= BUFFER_SIZE) {
83                                         buffer_addr = data;
84                                 }
85
86                                 TWCR_ACK;
87                         } else {
88                                 rxbuffer[buffer_addr] = data;
89
90                                 buffer_addr++;
91
92                                 if(buffer_addr < (BUFFER_SIZE-1)) {
93                                         // send ACK to retrieve more data
94                                         TWCR_ACK;
95                                 } else {
96                                         // send NACK, when buffer is full
97                                         TWCR_NACK;
98                                 }
99                         }
100                 break;
101
102                 // 0xA8 SLA+R received, ACK returned
103                 case TW_ST_SLA_ACK:
104
105                 // 0xB8 data transmitted, ACK received
```

```
106                 case TW_ST_DATA_ACK:
107                         if (buffer_addr == BUFFER_MAX_SIZE) {
108                                 buffer_addr = 0;
109                         }
110                         // set tx data to the TWI data register
111                         TWDR = txbuffer[buffer_addr];
112 
113                         buffer_addr++;
114 
115                         if(buffer_addr < (BUFFER_SIZE-1)) {
116                                 TWCR_ACK;
117                         } else {
118                                 TWCR_NACK;
119                         }
120                 break;
121 
122                 // 0xC0 data transmitted, NACK received
123                 case TW_ST_DATA_NACK:
124                 // 0x88 data received, NACK returned
125                 case TW_SR_DATA_NACK:
126                 // 0xC8  last data byte transmitted, ACK received
127                 case TW_ST_LAST_DATA:
128                 // 0xA0 stop or repeated start condition received while selected
129                 case TW_SR_STOP:
130                 default:
131                         TWCR_RESET;
132         }
133 }
```

Listing B.4: twislave.c

B.3 Pulsweitenmodulation Initialisierung

```
1 /**
2  * @date 29. July 2009
3  */
4 #ifndef __TIMER_H__
5 #define __TIMER_H__
6 
7 void timer_init();
8 
9 #endif // __TIMER_H__
```

Listing B.5: timer.h

```
1 /**
2  * @date 29. July 2009
3  */
4 #include "defines.h"
5 
6 #include <avr/io.h>
7 
8 void timer_init() {
9         // init timer
10         TIMSK |= (1<<TOIE0);
11         TCCR0 |= (0<<CS02) | (0<<CS01) | (1<<CS00);
12 }
```

Listing B.6: timer.c

B.4 Analog-Digital-Conversion Implementierung

```
1 /**
2  * @date 29. July 2009
```

```
 */
#ifndef __ADC_H__
#define __ADC_H__

uint16_t adc_readChannel(uint8_t mux);

#endif // __ADC_H__
```

Listing B.7: adc.h

```
/**
 * @date 29. July 2009
 */
#include "defines.h"

#include <avr/io.h>

uint16_t adc_readChannel(uint8_t mux) {
        int sample, i;

        sample = 0;

        ADMUX = mux;
        ADMUX |= (1 << ADLAR);
        ADCSRA |= (1 << ADEN);

    ADCSRA |= (1<<ADPS2) | (0<<ADPS1) | (1<<ADPS0);

    for(i=0; i<32; i++) {
          ADCSRA |= (1<<ADSC);
      while ( ADCSRA & (1<<ADSC) )
        ;

      sample += ADCH;
        }

    return sample/32;  // Aritmethisches Mittel der Samplewerte
}
```

Listing B.8: adc.c

B.5 Master - Hauptprogramm

```
/**
 * @date 29. July 2009
 */
#define SCL_CLOCK 400000L
#define SLAVE1_ADDRESS 0x50
#define SLAVE2_ADDRESS 0x60
#define STEP 5
#define MAX 250

#include <avr/io.h>
#include <util/delay.h>

#include "i2cmaster.h"

int main(void) {

        uint8_t adcvalue = 0;

        i2c_init();

        for(;;) {
                /**
                 * Sends the value for the LED.
```

```
24                  */
25                 i2c_start_wait(SLAVE1_ADDRESS+I2C_WRITE);
26                 i2c_write(0x00);
27                 i2c_write(adcvalue);
28                 i2c_stop();
29
30                 /**
31                  * Einlesen des POTI Wertes des Slave 2.
32                  */
33                 if(!(i2c_start(SLAVE2_ADDRESS+I2C_WRITE))) {
34                 i2c_write(0x00);
35                 i2c_rep_start(SLAVE2_ADDRESS+I2C_READ);
36                                         adcvalue = i2c_readNak();
37                                         i2c_stop();
38         }
39
40                                 adcvalue -= 6;
41                         _delay_ms(50);
42         }
43 }
```

Listing B.9: Hauptprogramm des *Master* in der I^2C Kommunikation

B.6 Slave 1 - Hauptprogramm

```
1  /**
2   * @date 29. July 2009
3   */
4  #define SLAVE_ADDRESS 0x50
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <util/delay.h>
9
10 #include "twislave.h"
11 #include "timer.h"
12
13 volatile uint8_t counter = 0;
14 volatile uint8_t max = 128;
15
16 ISR(TIMER0_OVF_vect) {
17         counter++;
18         if(counter >= max) {
19                 PORTB = (1<<PB1);
20         } else {
21                 PORTB = ~(1<<PB1);
22         }
23 }
24
25 int main(void) {
26         // PWM
27         timer_init();
28
29         DDRB |= (1<<PB1);
30
31         // TWI
32         init_twi_slave(SLAVE_ADDRESS);
33
34         sei();
35
36         for(;;) {
37                 max = rxbuffer[0];
38         }
39 }
```

Listing B.10: Hauptprogramm des *Slave* 1 mit der Pulsweitenmodulation

B.7 Slave 2 - Hauptprogramm

```
/**
 * @date 29. July 2009
 */
#define SLAVE_ADDRESS 0x60

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#include "twislave.h"
#include "adc.h"

int main(void) {

        uint16_t adcval = 0;

        // TWI
        init_twi_slave(SLAVE_ADDRESS);

        sei();

        for(;;) {
                adcval = adc_readChannel(0);
                txbuffer[0] = adcval;
                _delay_ms(100);
        }
}
```

Listing B.11: Hauptprogramm des *Slave* 2 mit Analog-Digital-Wandler