Projekt-Praktikum

Mikrocontroller-Einführung

Nikolai Helwig, Marco Schüler, Johannes Matthieu

Lehrstuhl für Messtechnik
Univ.-Prof. Dr. rer. nat. Andreas Schütze
Universität des Saarlandes

Inhaltsverzeichnis

1. Motivation 1
2. Hardware 2
2.1 Aufbau Experimentierboard 2
2.2 Mikrocontrollerschaltung 5
2.2.1 Mikrocontroller 5
2.2.2 Spannungsversorgung 5
2.2.3 Schwingquarz 6
2.2.4 ISP-Schnittstelle 7
2.3 Ein- und Ausgabe 7
2.3.1 Taster und Interrupt-Eingangspins 7
2.3.2 LED-Ausgabe 9
2.4 Wandlung zwischen digitalen und analogen Signalen 10
2.4.1 Analog-Digital-Converter 10
2.4.2 Digital-Analog-Converter und Pulsweitenmodulation 13
2.5 LCD Ansteuerung 16
2.6 Datenaustausch zwischen PC und Mikrocontroller per USB 18
3. Software 20
3.1 Schreiben eines Programms in C und Übertragen zum Mikrocontroller 20
3.1.1 C-Einführungsbeispiel 20
3.1.2 Makefile 22
3.1.3 Kompilieren mit WinAVR 23
3.1.4 Programmieren des ATmega-16 mit der Software AVR-Studio 23
3.1.4.1 Mikrocontroller konfigurieren 23
3.1.4.2 Mikrocontroller-Programmspeicher beschreiben in AVR-Studio 26
3.2 C 28
3.2.1 Header-Dateien 28
3.2.2 Datentypen in C (inkl. stdint.h) 29
3.2.3 Kontrollstrukturen in C 29
3.2.4 Funktionen 30
3.2.4.1 Nützliche Funktionen 31
3.2.5 Definition, Deklaration und Initialisierung 33
3.2.6 Sequentieller und interruptbasierter Programmablauf 34
3.3 Register 35
3.3.1 Bitmanipulation 35
3.3.1.1 Bitweise Operatoren 35
3.3.1.2 Register konfigurieren 35

3.3.2 Die Register des ATmega-16 37
3.3.2.1 Ein- und Ausgänge lesen und schalten 37
3.3.2.2 Externe Interrupts konfigurieren 38
3.3.2.3 ADC initialisieren und Wandlung starten 41
3.3.2.4 Timer und Pulsweitenmodulation konfigurieren und starten 44
3.3.2.5 USART konfigurieren und Daten senden 47
3.3.2.6 LCD initialisieren und Text ausgeben 51
4. Beispielprojekt Temperaturabhängige Lüfterregelung 53
4.1 Funktionsbeschreibung 53
4.1.1 Benutzereingabe und Menü 54
4.1.2 Temperaturmessung 55
4.1.3 Regler 57
4.1.4 Graphische Ausgabe am PC mit LogView 58
4.2 Quellcode 60
4.2.1 main.c 60
4.2.2 ADC 61
4.2.3 LCD 62
4.2.4 Regler 68
4.2.5 PWM 69
4.2.6 Taster 70
4.2.7 USART 71
5. Anhang 73
5.1 Software-Bezugsquellen 73
5.2 Stückliste Experimentierboard 74
5.3 Abkürzungsverzeichnis 75
5.4 Abbildungsverzeichnis 75
5.5 Literatur 76
5.6 Boardlayout 77

1. Motivation

Mikrocontroller (µC) sind aus modernen technischen Systemen nicht mehr wegzudenken. Unbemerkt verrichten sie in den Bereichen Unterhaltungselektronik, Mobiltelefone, Chipkarten und PC-Peripherie-Geräte ihren Dienst. In einem aktuellen Mittelklassewagen sorgen bis zu 80 Mikrocontrollersysteme unter anderem für die korrekte Funktion von Fahrassistenzsystemen, Multimediaanwendungen und der Klimaautomatik. Innovationen und Neuheiten im Automobilbereich basieren zum großen Teil auf mikrocontrollerbasierten Sensor-Aktor-Systemen.

Im Rahmen dieses Praktikums sollen mithilfe eines Mikrocontroller-Experimentierboards verschiedene Versuche durchgeführt werden.

Um die unterschiedlichen Funktionen eines µC's exemplarisch zu zeigen, wurde ein einfaches Sensor-Aktor-System entwickelt. Eine Regelung, welche die Drehzahl eines Lüfters abhängig von der Umgebungstemperatur variiert und Messwerte an einen PC übermittelt.

Ziel dieser kurzen Einführung ist es, den Einstieg in den Bereich eingebettete Systeme zu erleichtern, eine Übersicht über benötigte Hard- und Software aufzuzeigen und Interessierten einen Ausgangspunkt für eigene Mikrocontroller-Entwicklungen zu bieten.

2. Hardware

2.1 Aufbau Experimentierboard

Mittelpunkt des Experimentierboards ist der Mikrocontroller, welcher Eingänge liest, Daten verarbeitet und Ausgänge setzt. Dieser ist auf einem 40-poligen Sockel fixiert und kann bei Bedarf einfach ausgewechselt werden.

Der µC, ein Atmega-16, benötigt 5V Versorgungsspannung, welche mithilfe eines Linearreglers und externen Netzteils zur Verfügung gestellt wird.

Um den Programmspeicher des Mikrocontrollers mit dem kompilierten Quellcode zu beschreiben und Fusebit-Register zu ändern, finden sich rechts neben dem ATmega zwei Programmierschnittstellen, das 10-polige JTAG- und das 6-polige ISP-Interface.

Insgesamt verfügt das Board über neun Ein- und Ausgänge (davon 4 ADC- und 3 PWM-fähig), die auf der linken Seite über den 20-poligen Wannenstecker nach außen geführt werden. Daran kann beispielsweise eine Testplatine mit Sensoren angeschlossen werden.

Zwei Taster erlauben Benutzereingaben, der dritte ist für Reset zuständig und startet den Mikrocontroller neu.

Über die vier LED's auf der linken Seite können Zustände und Debugging-Hinweise angezeigt werden; das LCD informiert per Textausgabe über Sensordaten und Systemparameter.

Mit der USB-Schnittstelle ist eine Kommunikation zwischen PC und Mikrocontroller möglich, es lassen sich Messdaten empfangen, speichern und Befehle senden.

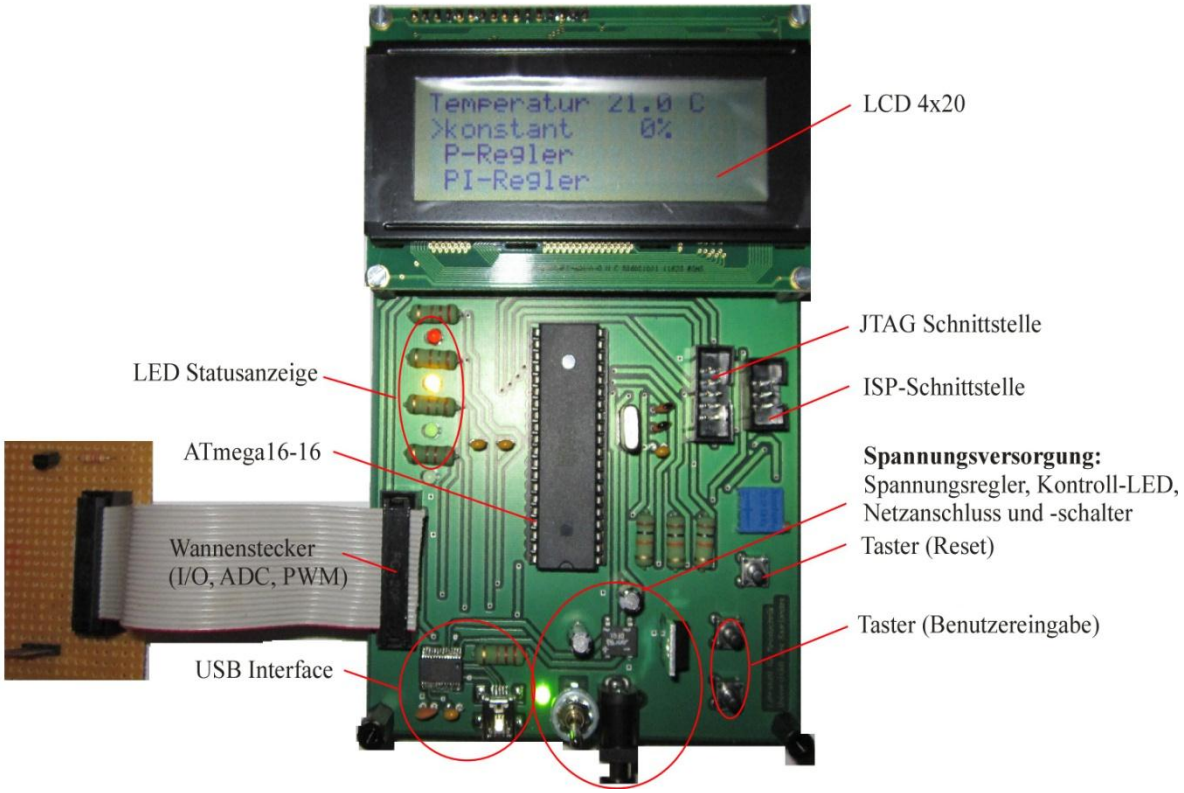


Abbildung 1: Aufbau des Experimentierboards

Übersicht über die Ein- und Ausgänge des Mikrocontroller-Experimentierboards:

Typ	**Bezeichnung**	**Beschreibung**
Input	ISP-Interface	• Programmierschnittstelle
Input	JTAG-Interface	• Programmierschnittstelle mit erweiterter Debug-Funktion
Input	Taster 0, Taster 1	• Benutzereingabe
Input	Reset	• Reset-Taster zum Programmneustart
Input	Spannungs-versorgung	• 12 V Netzteil benötigt • Gleich- oder Wechselstrom • An/Aus-Kippschalter
I/O	USB-Schnittstelle	• Kommunikation zwischen Mikrocontroller und PC
I/O	Wannenstecker	• 3 Hardware-PWM-Pins (auch konfigurierbar als Standard I/O Pins) • 4 ADC-Pins (auch konfigurierbar als Standard I/O Pins) • 3 I/O Pins • 5 V Spannungsversorgung (Vcc) • 12 V Spannungsversorgung
Output	LED-Statusanzeige	• LED rot • LED gelb • LED grün • LED blau
Output	LCD-Display	• Textausgabe von Daten und Informationen

Tabelle 1: Schnittstellen des Experimentierboards

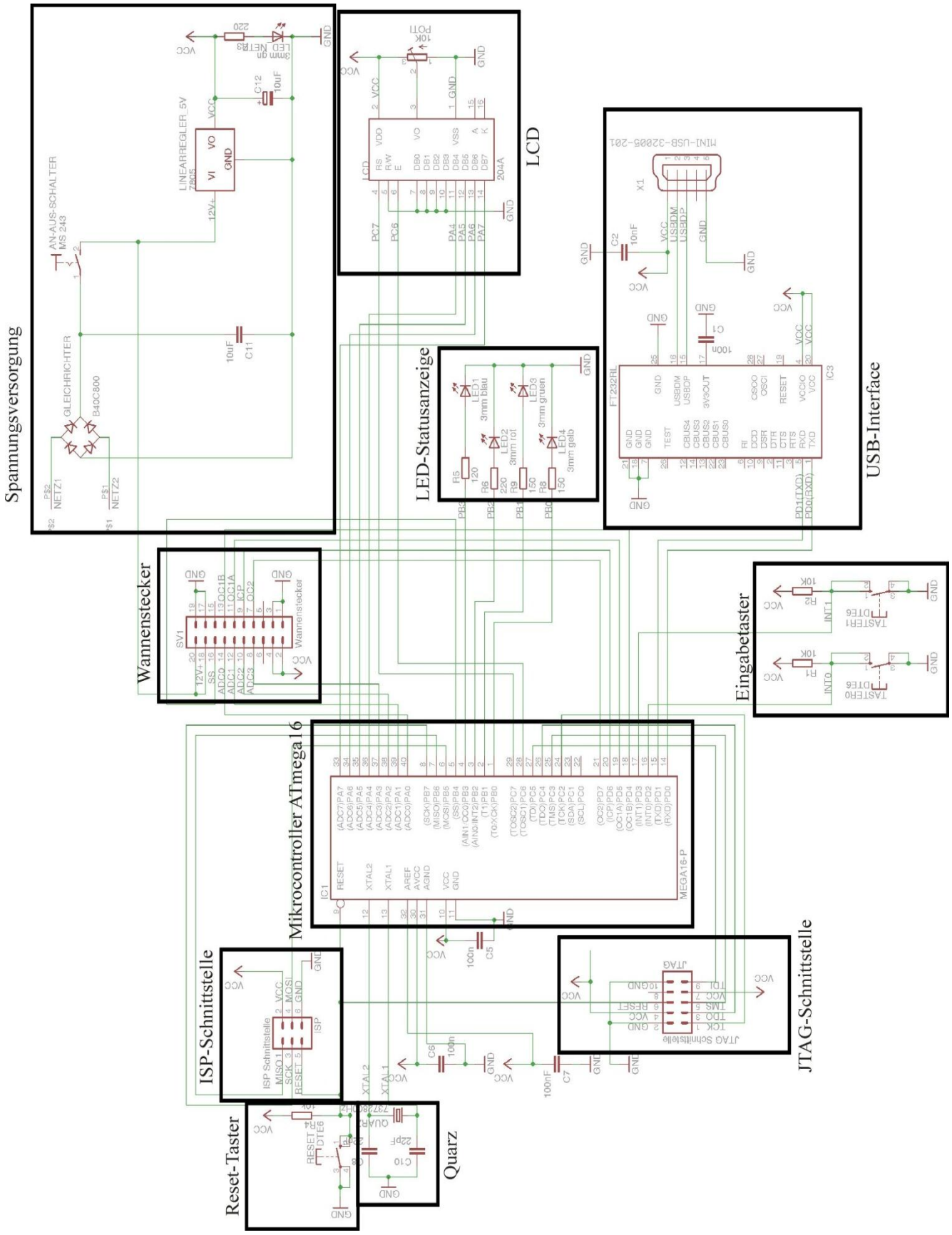


Abbildung 2: Gesamtschaltplan des Experimentierboards

2.2 Mikrocontrollerschaltung

2.2.1 Mikrocontroller

Als Mikrocontroller bezeichnet man einen Chip, auf dem neben einem Mikroprozessor weitere Module mit unterschiedlichen Funktionen integriert sind. Häufig zählen dazu Programm- und Arbeitsspeicher, digitale und analoge Ein- und Ausgänge, Timer und Schnittstellen zur Kommunikation. Der Vorteil eines solchen System-on-a-Chip's (SoC) ist, dass zum Betrieb nur wenige externe Bauteile benötigt werden. Im einfachsten Fall arbeitet ein programmierter µC bereits mit Versorgungsspannung, möchte man weitere Peripheriefunktionen nutzen, erhöht sich entsprechend der Bauteil- und Verschaltungsaufwand.

Für das Experimentierboard wurde ein Mikrocontroller der AVR-Familie von Atmel verwendet, der *ATmega-16*. Die AVR-Mikrocontroller sind nach der Harvard-Architektur aufgebaut, besitzen somit getrennte Busse für Arbeitsspeicher (RAM) und Programmspeicher (Flash-Rom) und können deswegen gleichzeitig auf beide Speicherarten zugreifen.

Der ATmega-16 ist ein 8-Bit-Controller, das bedeutet, dass er bei einer internen Rechenoperation ganzzahlige Werte bis maximal 2^8 -1, also 255 (0 ist 256. Zustand), darstellen kann. Werden Befehle mit größeren Zahlen durchgeführt, müssen mehrere 8-Bit Operationen nacheinander abgearbeitet werden, was die Rechenzeit erhöht.

Für den Betrieb des Mikrocontrollers wird ein Taktgeber benötigt. Hierfür kann der interne 1 MHz-Oszillator des ATmega's oder ein externer Quarz verwendet werden.

Wird der ATmega-16 mit einem 16 MHz Quarz (Maximalfrequenz) betrieben, kann er theoretisch 16 MIPS (Million instructions per second), also 16 Millionen Maschinenbefehle pro Sekunde ausführen.

Als weitere Peripheriefunktionen bietet er:

- 8-Bit-Timer (2x)
- 16-Bit-Timer
- Real-Time-Counter
- PWM-Kanäle (4x)
- 10-Bit ADC-Kanäle (8x)
- USART-Schnittstelle
- TWI-Schnittstelle (I2C)
- Watchdog Timer

In Abbildung 3 sind die 40 Anschluss-Pins des AVR's zu sehen, 32 davon besitzen eine Hardware-Funktion (z.B ADC-Eingänge PA0-PA7, PWM-Ausgänge PD4 und PD5, usw.), können aber im Quellcode auch als reguläre Eingangs- oder Ausgangspins konfiguriert werden.

Damit kann der ATmega flexibel den jeweiligen Anforderungen angepasst werden.

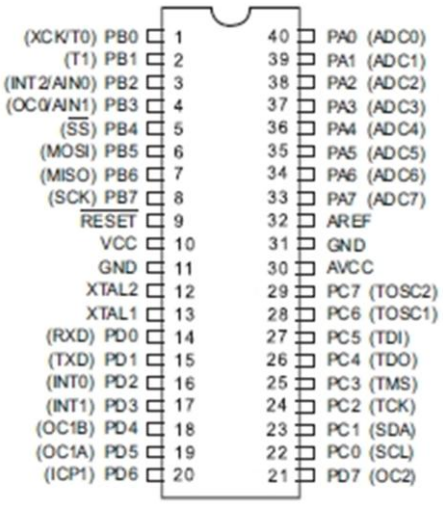


Abbildung 3: ATmega-16 Pinbelegung
(Quelle: ATmega 16 Datenblatt)

[zurück zu Aufbau Experimentierboard]

2.2.2 Spannungsversorgung

Um sowohl Wechsel- als auch Gleichspannungsnetzteile als Versorgung für die Schaltung verwenden zu können, wird der Eingang zunächst mit einer Diodenbrücke gleichgerichtet und die resultierende Welligkeit der Spannung mithilfe eines Elkos geglättet. Der Linearregler 7805 generiert daraus eine stabile und größtenteils lastunabhängige Spannung von 5V für die elektronischen Komponenten und verträgt maximal 35V.

Somit kann ein externes Netzteil mit 12 V Ausgangsspannung verwendet werden. Alternativ wird die

Schaltung auch von einem PC über USB mit 5 V versorgt.

Eine grüne LED zeigt an, ob die Spannungsversorgung korrekt angeschlossen und der Netzschalter angeschaltet ist.

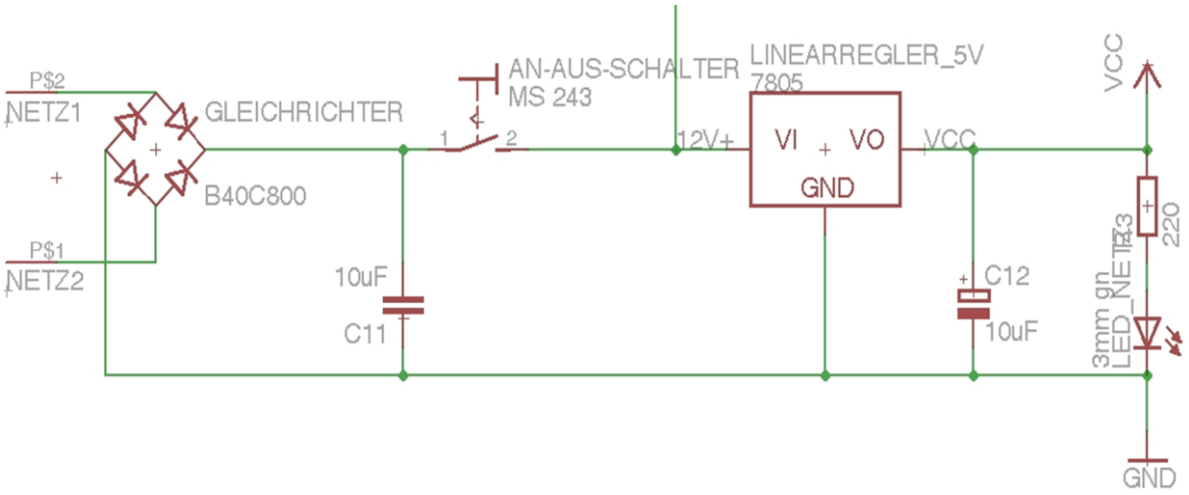


Abbildung 4: Spannungsversorgung

2.2.3 Schwingquarz

Der Schwingquarz ist das taktgebende Herz des Mikrocontrollers und bestimmt, wie schnell dieser Prozesse abarbeitet. Für den Betrieb eines Controllers ist der Quarz nicht unbedingt notwendig, da er über einen internen Taktgeber, einen RC-Oszillator (Genauigkeit ± 3%) verfügt. Sobald allerdings eine gewisse Genauigkeitsanforderung an den Taktgeber gestellt wird, etwa wenn der Mikrocontroller mit dem PC kommunizieren soll, ist der Einsatz eines Quarzes (Genauigkeit 10-100ppm) unerlässlich.

Standardmäßig wird die ATmega-Familie mit aktiviertem internem RC-Oszillator ausgeliefert. Die Frequenz des verwendeten Quarzes wird im Makefile unter F_CPU angegeben, in diesem Fall 7,3728 Mhz.

Abbildung 5 zeigt die Verschaltung des Quarzes, welcher an die Pins XTAL 1 und 2 angeschlossen wird, zusammen mit zwei Anschwingkondensatoren gegen Masse.

[Zum Kapitel Software: Makefile]
[Zum Kapitel Software: Konfigurieren des µC]

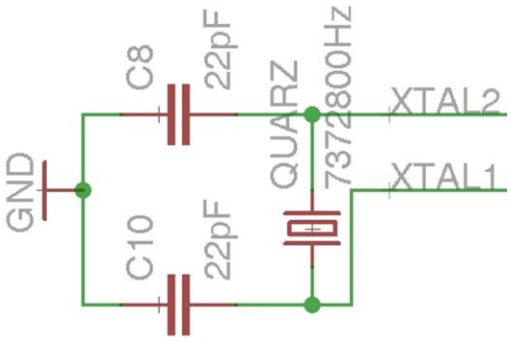


Abbildung 5: Quarz mit Anschwingkondensatoren

2.2.4 ISP-Schnittstelle

Für das Übertragen des kompilierten Programms auf den Mikrocontroller wird die ISP-Schnittstelle (In-System-Programmierung) und der Atmel AVRISP-Programmer (Abbildung 6) verwendet. Möchte man sich mit dem Mikrocontroller verbinden, müssen zunächst in der Programmierumgebung am PC (z.B. Atmel AVR-Studio) der Programmer und Mikrocontroller eingestellt und nach dem Verbinden die Fusebits konguriert werden. Danach lässt sich das hex-File in den Flash-Speicher des Controllers schreiben.

In Abbildung 7 ist die Pinbelegung der ISP-Buchse zu sehen. Neben der Versorgungsspannung muss sie mit den MISO-, MOSI-, SCK- und RESET-Pins des µC verbunden werden.

[Zum Kapitel Software: Programmierung des ATmega16 mit AVR-Studio]

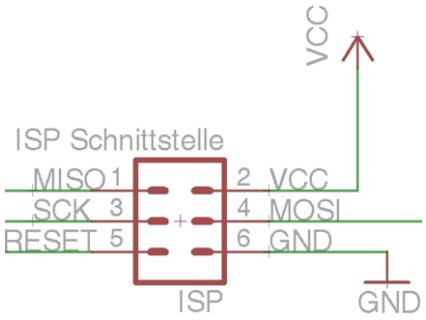


Abbildung 6: Beschaltung der ISP-Buchse

Abbildung 7: ISP Programmiergerät Atmel AVRISP mkII (Quelle: Wikipedia)

2.3 Ein- und Ausgabe

2.3.1 Taster und Interrupt-Eingangspins

Die drei Mikrocontrollerpins INT0, INT1 und INT2 des ATmega besitzen eine Hardware-Interrupt-Funktion. Sie liefern intern Rückmeldung bei einem bestimmten Ereignis, ohne dass sie zyklisch vom µC abgefragt werden müssen (Polling). Diese Nebenläufigkeit wirkt sich positiv auf die Performanz aus.

Ist im Quelltext ein Interrupt bei steigender Flanke aktiviert, wird der sequentielle Programmdurchlauf bei einem Ansteigen des Eingangssignals von logisch 0 auf 1 unterbrochen und die Interrupt-Service-Routine (ISR) ausgeführt. Das Ereignis (Interrupt Request IRQ) wird durch den ISR-Vektor im Quellcode als Argument an die ISR übergeben.

Ist die ISR durchgelaufen, springt der µC wieder zurück an die Stelle des Programms, an der er vom Interrupt unterbrochen wurde.

µC-Pin	**ISR-Vektor**	**Beschreibung**
PD2 (INT0)	INT0_vect	Interrupt-Eingangspin 0
PD3 (INT1)	INT1_vect	Interrupt-Eingangspin 1
PB2 (INT2)	INT2_vect	Interrupt-Eingangspin 2

Tabelle 2: Interrupt-Eingangspins des ATmega-16

Mit einem Taster kann auf diese Weise eine Benutzereingabe realisiert werden. Beim Drücken des Tasters sind die beiden Pin-Paare des Tasters leitend verbunden, ansonsten isoliert.

Abbildung 8 zeigt die Beschaltung der Taster auf dem Experimentierboard nach dem sogenannten *Active-low* Prinzip. Wird der Taster nicht gedrückt, zieht der Pullup-Widerstand das Potential des Interrupt-Eingangspins des µCs auf Vcc und somit logisch 1. Bei gedrücktem Taster liegt der Pin direkt an Masse, logisch 0, über den Pullup-Widerstand fließt ein vernachlässigbar kleiner Strom.

Durch diese Verschaltung ist am Eingangspin des Mikrocontrollers immer ein definiertes Potential gegeben.

Bei mechanischen Tastern und Schaltern tritt häufig das unerwünschte Verhalten des *Prellens* auf. Bevor beim Drücken oder Loslassen ein stabiler Zustand erreicht wird, springt der Schaltkontakt mehrmals zwischen An- und Aus-Zustand. Um dies zu vermeiden, kann ein Taster sowohl hard- als auch softwareseitig entprellt werden.

[Zum Kapitel Software: Externe Interrupts konfigurieren]

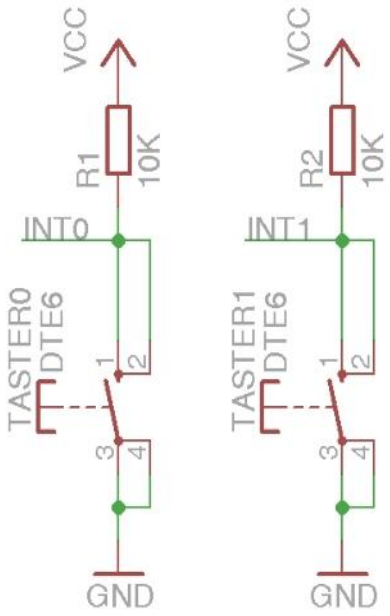


Abbildung 8: Active-low-Schaltung mit 10k-Pullup-Widerständen

2.3.2 LED-Ausgabe

Mithilfe von Leuchtdioden (LEDs) können Zustände und Informationen zum Programmablauf ausgegeben werden. Wird ein Ausgangs-Pin des µC's im Quellcode auf logisch 1 gesetzt, leuchtet die daran angeschlossene LED.

Abbildung 9 zeigt die Verschaltung der vier Status-LEDs. Die Dimensionierung der Vorwiderstände, die als Strombegrenzer dienen, hängt von der anliegenden Versorgungsspannung und der Durchlassspannung/-stromstärke der jeweiligen Leuchtdiode (siehe Datenblatt!) ab.

LED-Farbe	**Durchlass-spannung**	**Vorwiderstand bei 5V Betriebsspannung und 20mA Durchlass-Strom**	**Verlustleistung des Vorwiderstands**	**Verbunden mit µC-Pin**
IR	1,5 V	220 Ω	70 mW	-
rot	1,6 V	220 Ω	68 mW	PB2
gelb	2,1 V	150 Ω	58 mW	PB0
grün	2,2 V	150 Ω	58 mW	PB1
blau	2,9 V	120 Ω	42 mW	PB3
weiss	4,0 V	56 Ω	20 mW	-

Tabelle 3: LED Kenngrößen

Nach Tabelle 3 können Standard-Widerstände mit 0,25 W maximaler Verlustleistung zur LED-Strombegrenzung bei 5V Betriebsspanunng genutzt werden.

[Zum Kapitel Software: Ausgänge schalten]

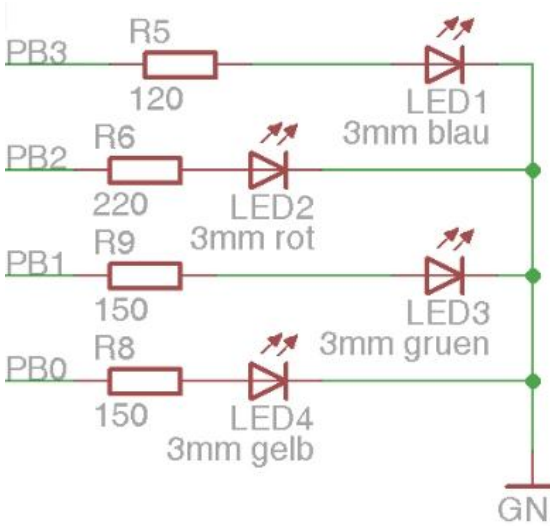


Abbildung 9: LEDs mit Vorwiderständen

2.4 Wandlung zwischen digitalen und analogen Signalen

2.4.1 Analog-Digital-Converter

Der Analog-Digital-Converter (ADC) ist die Brücke von der analogen zur digitalen Welt. Er misst die am Eingang anliegende Spannung und wandelt diese in einen digitalen Wert um.

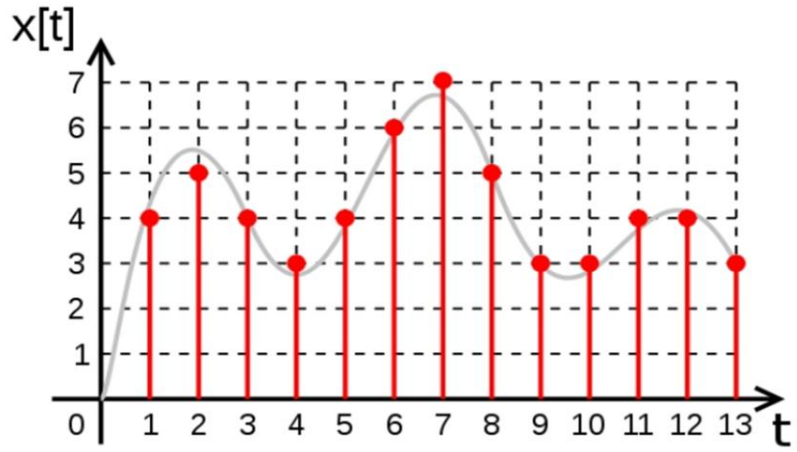


Abbildung 10: Zeitliche und wertmäßige (3-Bit) Quantisierung eines Analogsignals (Quelle: Wikipedia)

Bei idealerweise gleichen Zeitabständen wird die Amplitude eines Eingangssignal diskretisiert (Abbildung 10). Wie fein dies geschieht, bestimmt die Anzahl der Quantisierungsstufen. Häufig besitzen µC einen 10-Bit ADC. Damit können 1024 Zustände unterschieden werden. Die Abtastfrequenz des ADC ist so zu wählen, dass sie mindestens doppelt so groß wie die höchste zu messende Frequenz des Messsignals ist, um eine eindeutige Rekonstruktion des Signals zu gewährleisten (Nyquist-Shannon-Abtasttheorem).

Man unterscheidet drei Wandlungsverfahren: Flash-Wandler, Slope-Wandler und *Sukzessive Approximation (SAR)*. Letztere ist ein guter Kompromiss aus Geschwindigkeit und Hardwareaufwand und deswegen weit verbreitet, unter anderem in der ATmega-Serie.

Der SAR-Wandler (Abbildung 11) vergleicht die mittels eines internen Digital-Analog-Wandlers erzeugte Spannung mit der zu messenden Spannung am ADC-Eingang. Beginnend mit dem Most Significant Bit (höchstwertiges Bit, MSB) werden nacheinander die Bitstellen des DA-Wandlers (DAC) auf 1 gesetzt und die nachfolgenden auf 0. Befindet sich die DAC-Spannung unter der ADC-Spannung, wird das nächsttiefere Bit auf 1 gesetzt. Ist die DAC-Spannung größer als die ADC-Spannung, wird das aktuelle Bit wieder zurückgesetzt und das nächsttiefere Bit auf 1 gesetzt. Diese Intervallschachtelung wird solange fortgesetzt bis man den ADC-Spannungswert eindeutig eingegrenzt hat und beim Least Significant Bit (LSB) angelangt ist.

In dem Beispiel aus Tabelle 4 wird der Eingangsspannung von 2,45 V somit nach fünf Schritten der digitale Wert 15 zugeordnet.

Damit sich die Spannung am ADC-Eingang während des Wandlungsvorgangs nicht ändert, wird sie durch ein Halteglied konstant gehalten (sample & hold).

Schritt	ADC-Eingangs-spannung	Erzeugter DAC-Vergleichswert	DAC-Vergleichswert < ADC-Wert ?	DAC Bit 1 (16)	DAC Bit 2 (8)	DAC Bit 3 (4)	DAC Bit 4 (2)	DAC Bit 5 (1)
1	2,45 V	2,58 V	f	1	0	0	0	0
2	2,45 V	1,29 V	w	0	1	0	0	0
3	2,45 V	1,94 V	w	0	1	1	0	0
4	2,45 V	2,26 V	w	0	1	1	1	0
5	2,45 V	2,42 V	w	0	1	1	1	1

Tabelle 4: Schritte bei 5-Bit SAR-Verfahren bei 2,45 V Eingangsspannung und 5V Referenz

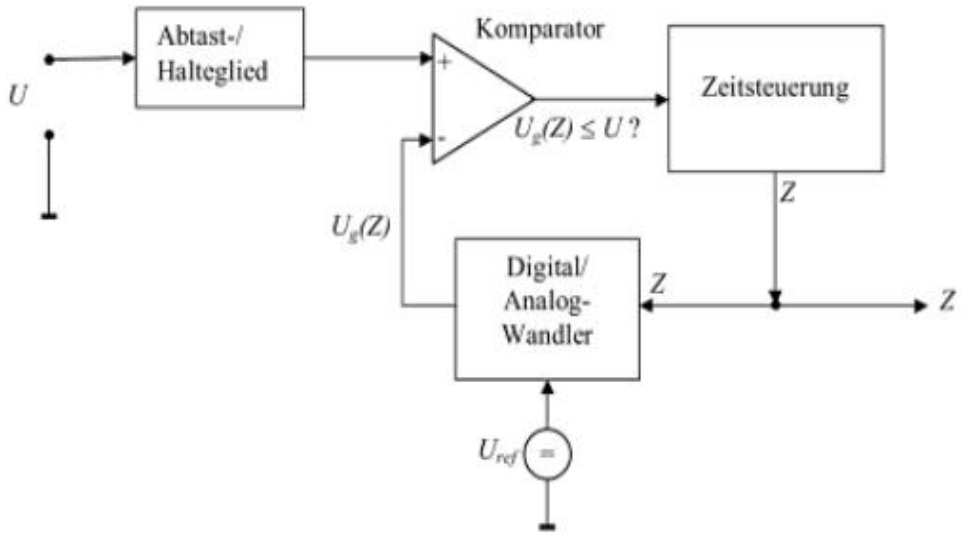


Abbildung 11: Prinzip Sukzessive Approximation
(Quelle: Mikrocontroller und Mikroprozessoren (Springer Verlag, Brinkschulte)

Der Eingangswiderstand der ADC-Pins liegt im MΩ-Bereich, somit wird die anliegende Spannung durch die Messung praktisch nicht verfälscht.

Die zu messende Spannung muss im Bereich zwischen GND und der Referenzspannung (meist Versorgungsspannung 5V oder 2,56V) liegen. Weist das Messsignal größere Spannungen auf, kann es durch einen Spannungsteiler in den Messbereich transformiert werden.

Die ADC Pin-Belegung am Beispiel eines Atmega 16:

µC-Pin	**Beschreibung**
ADC0-ADC7	ADC-Eingangspins; bei der ADC-Initialisierung muss im ADMUX-Register der verwendete Pin aktiviert werden. *Hinweis*: Der ATmega-16 verfügt zwar über 8 ADC-Kanäle, aber nur einen Analog-Digital-Wandler. Über einen Multiplexer können die verschiedenen Eingänge nacheinander abgefragt werden.
AREF	Auswählbare externe ADC Referenzspannung *Hinweis*: Minimalbeschaltung mit 100nF Pufferkondensator gegen AGND.
AVCC	Interne ADC Referenzspannung (5V oder daraus erzeugte 2,56V auswählbar) *Hinweis*: Ist im Allgemeinen der externen Referenzspannung vorzuziehen. Minimalbeschaltung mit 100nF Pufferkondensator gegen AGND.
AGND	Interne ADC Referenzmasse

Tabelle 5: ADC Pins ATmega16

[Zum Kapitel Software: ADC Konfiguration]

Weiterführende Links:

mikrocontroller.net Tutorial–AD-Wandlung

Mikrocontroller und Mikroprozessoren-Kapitel AD-Wandlung

Wikipedia-Analog-Digital-Umsetzer

2.4.2 Digital-Analog-Converter und Pulsweitenmodulation

Damit ein Mikrocontroller andere Bauteile und Aktoren mit flexibler Spannung ansteuern kann, benötigt er einen *Digital-Analog-Converter* (DAC). Da das Erzeugen von regulierbaren analogen Spannungen technisch aufwendig ist, wird die *Pulsweitenmodulation* (PWM) verwendet. Das Ausgangssignal wird dabei vom µC sehr schnell kontinuierlich an- und ausgeschaltet, also abwechselnd auf Versorgungsspannung und Masse gesetzt. Die relative Anschaltdauer beeinflusst dabei die effektive Ausgangsspannung.

Schaut man sich das Signal im Oszilloskop an, erkennt man eine konstante Periode mit einer variablen Pulsbreite, bei der das Signal auf Vcc gesetzt wird. Aus deren Zeitverhältnis wird der Tastgrad (Duty-Cycle) berechnet:

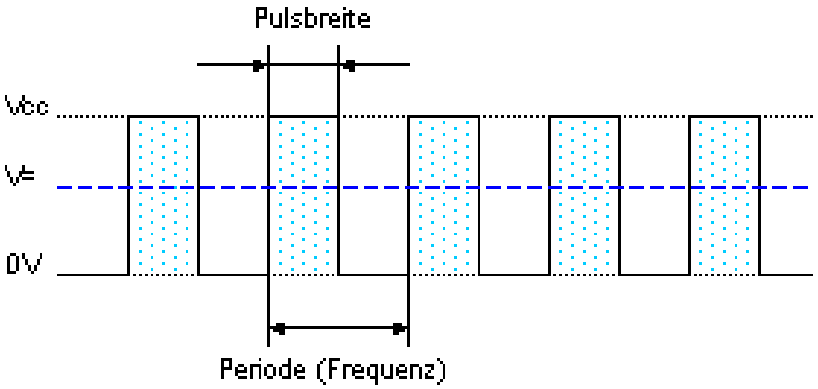


Abbildung 12: Prinzip PWM
(Quelle: mikrocontroller.net)

$$DutyCycle = \frac{Pulsbreite}{Periodendauer} = \frac{\tau}{T}$$

In Abbildung 12 beträgt die Pulsbreite gerade die Hälfte der Periodendauer, der Tastgrad ist somit 50%.

Der Duty-Cycle ist proportional zum arithmetischen zeitlichen Mittelwert der PWM. Durch Modulation der Pulsbreite lässt sich somit der Mittelwert der PWM-Ausgangsspannung variieren:

$$\overline{U_{PWM}} = \frac{1}{T}\int_{t=0}^{t=T} u(t)\,dt = \frac{1}{T}\sum_{i=0}^{i=T} u_i = \frac{\tau}{T} \cdot Vcc$$

Das PWM-Signal wird durch einen Tiefpass demoduliert und geglättet und entspricht dann näherungsweise einem analogen konstanten Signal (Abbildung 13, 15).

Manche Bauteile benötigen keinen Filter, da sie aufgrund ihrer Trägheit (mechanisch, thermisch, usw.) selbst wie ein Tiefpass wirken. Wird eine LED mittels Pulsweitenmodulation angesteuert, nimmt das träge menschliche Auge ein gleichmäßiges Leuchten wahr, obwohl sie eigentlich mit hoher Frequenz flackert.

Die Einsatzgebiete einer PWM sind beispielsweise das Dimmen von LEDs, die Ansteuerung einer Heizung und Motoransteuerungen.

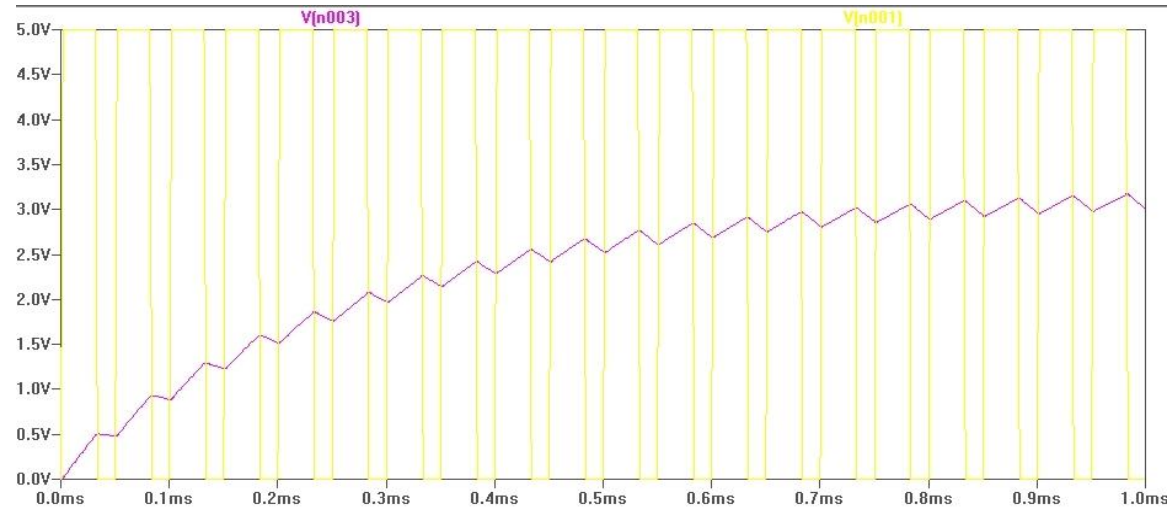


Abbildung 13: PWM-Signal ohne (gelb) und mit Tiefpass-Filter 2. Ordnung (violett), Simulation in LTspice

Beim Erzeugen eines 8-Bit PWM-Signals (Abbildung 14) im invertierenden *Fast-PWM-Modus* zählt ein interner Timer in jeder Periode von 0 (BOTTOM) bis 255 (TOP) und vergleicht seinen Zählerstand bei jedem Schritt mit dem Wert im Output Compare Register (OCR) , das mit dem gewünschten PWM-Wert beschrieben werden kann. Sind Register- und aktueller Zählwert gleich (Match), wird das PWM-Signal auf 1 gesetzt. Beim Erreichen des Zählwerts 255 fällt das PWM-Signal auf low und das Hochzählen beginnt wieder bei 0.

Im nicht-invertierenden Modus ist das PWM-Signal zu Periodenbeginn auf high und wird bei Match zu 0 gesetzt.

Der *Phase-Correct-PWM-Modus* arbeitet ähnlich, allerdings unterscheidet sich die Zählweise des Timers. Er beginnt innerhalb jeder Periode bei TOP, zählt bis BOTTOM und von dort wieder hoch bis TOP. Dadurch ist der phasenkorrekte Modus zwar nur halb so schnell wie der Fast-PWM-Modus, liefert aber ein symmetrisches Ausgangssignal.

Bei *Clear Timer on Compare (CTC)* zählt der Timer hoch bis er mit dem OCR-Wert übereinstimmt, setzt das Ausgangssignal und beginnt dann wieder bei Null von Vorne.

Die Frequenz und somit die Periodendauer der PWM hängt vom verwendeten Quarz, dem im Quellcode eingestellten Frequenzteiler, dem Modus und der Auflösung der PWM ab.

Mit einem 8 MHz Quarz und einem Frequenzteiler von 1 beispielsweise inkrementiert der PWM-Timer alle 125 ns. Bei einer 8-Bit Fast-PWM ergibt sich damit eine Periodendauer von 32 µs und eine Frequenz von 31,25 kHz, bei der ein neuer PWM-Ausgangswert gesetzt werden kann.

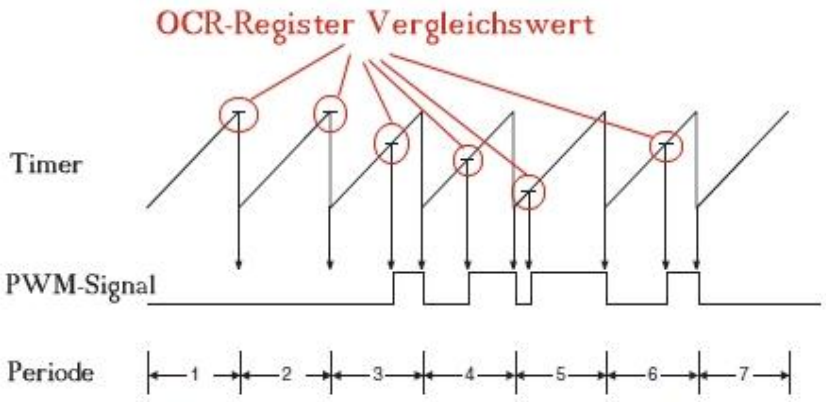


Abbildung 14: Erzeugung des PWM-Signals (invertierendes Fast PWM)
(Quelle:ATmega 16 Datenblatt)

Die PWM-Pinbelegung am Beispiel des ATmega-16:

µC-Pin	**Beschreibung**
PB3 (OC0)	8-Bit PWM-Ausgang, gesteuert durch Timer/Counter 0
PD5 (OC1A)	16-Bit PWM-Ausgang, gesteuert durch Timer/Counter 1
PD4 (OC1B)	16-Bit PWM-Ausgang, gesteuert durch Timer/Counter 1
PD7 (OC2)	8-Bit PWM-Ausgang, gesteuert durch Timer/Counter 2

Tabelle 6: PWM Pins ATmega16

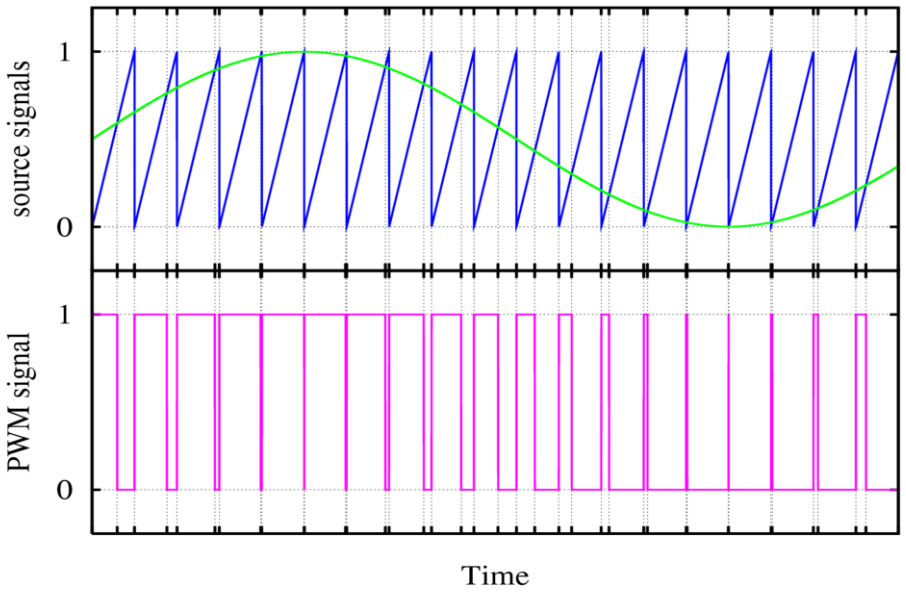


Abbildung 15: Erzeugen einer „Sinusschwingung“ per nicht-invertierender Fast-PWM
(Quelle: Wikipedia)

[Zum Kapitel Software: PWM Konfiguration]

Weiterführende Links:

Mikrocontroller und Mikroprozessoren-Kapitel Pulsweitenmodulator

http://www.mikrocontroller.net/articles/AVR_PWM

Wikipedia–Pulsweitenmodulation

mikrocontroller.net Tutorial–PWM

2.5 LCD Ansteuerung

Ein LCD kann beim Betrieb eines Mikrocontrollers äußerst nützlich sein, zum einen für den Entwickler, um sich Debug-Informationen ausgeben zu lassen, zum anderen für den Benutzer der fertigen Schaltung, der so direkt aktuelle Zustände und Werte ablesen kann.

Abbildung 16: 4x20 Zeichen Text-Display

Die meisten Text-LCDs verfügen über einen HD44780 kompatiblen Controller und besitzen folgende Pinbelegung (sollte anhand des Datenblatts verifiziert werden):

Pin	**Bezeichnung**	**Funktion**
1	Vss	GND
2	Vcc	5V
3	Vee	Kontrastspannung
4	RS	Register Select
5	RW	Read/Write
6	E	Enable
7	DB0	Datenbit 0
8	DB1	Datenbit 1
9	DB2	Datenbit 2
10	DB3	Datenbit 3
11	DB4	Datenbit 4
12	DB5	Datenbit 5
13	DB6	Datenbit 6
14	DB7	Datenbit 7
15	A	Anode LED Beleuchtung
16	K	Kathode LED Beleuchtung

Tabelle 7: Standard-Pinbelegung eines HD44780 Displays, grau markierte Pins werden nicht verwendet

Das Display des Experimentierboards wird im 4-Bit Modus betrieben, das heißt die LCD-Datenbit-Pins DB0 bis DB3 bleiben ungenutzt und die Textinformationen werden über die Pins DB4 bis DB7 gesendet. Das hat den Vorteil, dass sich der Beschaltungsaufwand reduziert und man sich vier µC-Ausgänge spart, die anderweitig genutzt werden können. Die zu sendenden Bytes werden jeweils in zwei 4-bit Datenpakete aufgespalten und nacheinander geschickt. In der Praxis ergeben sich durch den Betrieb im 4-bit Modus keinerlei Nachteile gegenüber dem 8-bit Modus.

Neben den Datenleitungen und der Betriebsspannung an den Pins 1 und 2 müssen noch die Pins 4 (RS) und 6 (E) mit dem ATmega verbunden werden. RS bestimmt, ob der Mikrocontroller gerade Daten oder Befehle an das LCD schickt. Pin E signalisiert, ob Daten oder Befehle versandfertig sind. Pin 5 (RW) legt die Datenrichtung fest und wird auf Masse gelegt, da das LCD nur Daten lesen soll. An Pin 3 wird die Kontrastspannung des LCDs angelegt, mithilfe des angeschlossenen Potis lässt sich somit der Kontrast den Displays verändern.

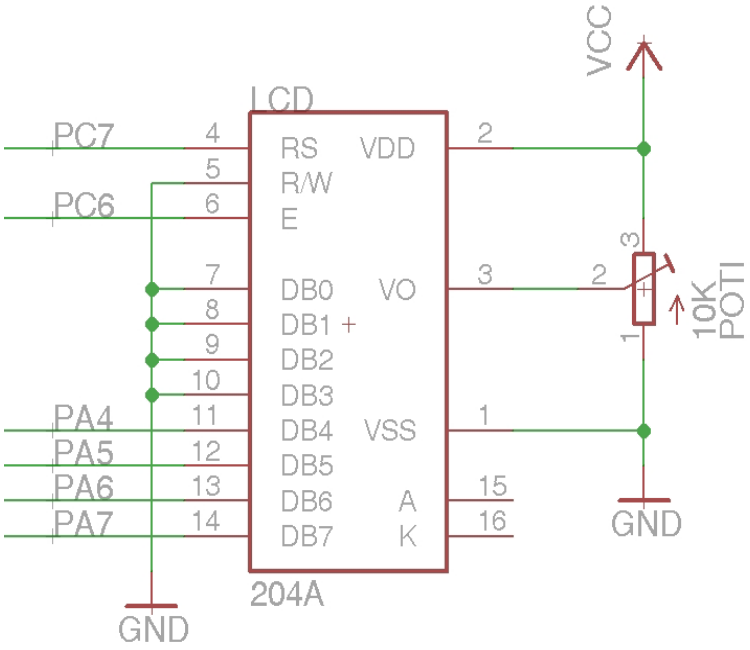


Abbildung 17: 4bit-Beschaltung eines HD44780 LCD

Anstelle der Pins PA4-7 und PC6/7 (Abbildung 17) können auch andere Mikrocontroller-Ausgänge zur Ansteuerung des Displays verwendet werden, dafür muss die Header-Datei lcd-routines.h angepasst werden.

[Zum Kapitel Software: LCD-Ansteuerung]

Weiterführende Links:

mikrocontroller.net Tutorial-LCD Ansteuerung

2.6 Datenaustausch zwischen PC und Mikrocontroller per USB

Oftmals ist es erwünscht, Daten vom Mikrocontroller an andere IC's oder einen PC zu senden und Befehle zu empfangen.

Für den Datenaustausch zwischen IC's gibt es eine Vielzahl von Schnittstellen und Bussystemen, z.B.:

- RS-232
- Serial Peripheral Interface (SPI)
- USB
- I²C (Inter IC Bus)
- Bluetooth Funkübertragung

Das Experimentierboard verfügt über ein USB-Interface, um mit dem PC kommunizieren zu können. Auf eine RS232-Schnittstelle wurde verzichtet, da sie an modernen PCs immer seltener vorkommt und die USB-Schnittstelle im Consumerbereich zum Standard geworden ist. Somit ist eine gute Kompatibilität gewährleistet.

Da der ATmega-16 keinen Hardware-USB Ausgang besitzt, wird ein weiterer IC benötigt, der mittels USART vom Controller angesteuert wird und die empfangenen Daten auf die USB-Schnittstelle ausgibt. Dafür eignet sich beispielsweise der FT232RL von FTDI (Abbildung 18). Dieser ist einfach zu verschalten und wird im einfachsten Fall nur durch die RXD und TXD Pins des Mikrocontrollers angesteuert. Der FTDI verfügt über eine interne Taktquelle und benötigt keinen externen Quarz.
Die Ausgänge USBDM und USBDP werden über die USB-Buchse an den PC geführt und liefern die transformierten Daten.
Windows und Linux Betriebssysteme benötigen einen Treiber, um mit dem FT232 kommunizieren zu können. Neuere Windows Versionen wie Vista/7 haben diesen bereits in ihrer Bibliothek und installieren ihn automatisch, sobald man die Schaltung mit dem PC verbindet. Bei älteren Windows Versionen muss der VCD-Treiber (Virtual COM-Port Driver) von der Webseite des Herstellers heruntergeladen und manuell installiert werden. Er simuliert den COM-Port einer RS232-Schnittstelle (virtueller COM-Port), von dem aus Daten empfangen und gesendet werden können.
Der Benutzer bestimmt durch die PC-seitige Software (z.B. ein Terminalprogramm) Baudrate, Stoppbits und Anzahl der Datenbits der Verbindung. Diese Konfiguration wird automatisch an den FTDI übermittelt und abgeglichen. Die Kommunikation unterscheidet sich somit für den Mikrocontroller und die PC-Software nicht von der gewöhnlichen seriellen Schnittstelle.

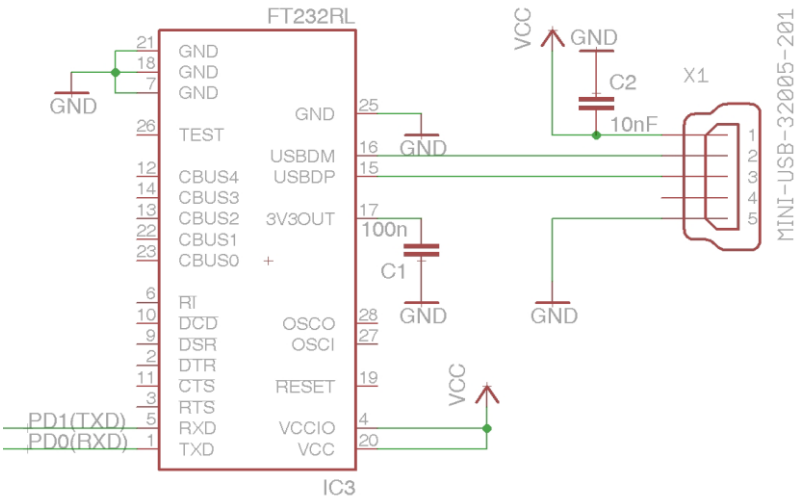


Abbildung 18: UART Ansteuerung des USB-Interfaces FT232

Pin	Bezeichnung	Typ	Funktion
1	TXD	Output	Transmit Asynchronous Data Output.
2	DTR#	Output	Data Terminal Ready Control Output / Handshake Signal.
3	RTS#	Output	Request to Send Control Output / Handshake Signal.
4	VCCIO	Power	+1.8V to +5.25V supply to the UART Interface and CBUS group pins (1...3, 5, 6, 9...14, 22, 23).
5	RXD	Input	Receiving Asynchronous Data Input.
6	RI#	Input	Ring Indicator Control Input.
7	GND	Power	Device ground supply pins
8	NC		No internal connection
9	DSR#	Input	Data Set Ready Control Input / Handshake Signal.
10	DCD#	Input	Data Carrier Detect Control Input.
11	CTS#	Input	Clear To Send Control Input / Handshake Signal.
12	CBUS4	I/O	Configurable CBUS output only Pin.
13	CBUS2	I/O	Configurable CBUS I/O Pin.
14	CBUS3	I/O	Configurable CBUS I/O Pin.
15	USBDP	I/O	USB Data Signal Plus, incorporating internal series resistor and 1.5kΩ pull up resistor to 3.3V.
16	USBDM	I/O	USB Data Signal Minus, incorporating internal series resistor.
17	3V3OUT	Power Output	+3.3V output from integrated LDO regulator. This pin should be decoupled to ground using a 100nF capacitor.
18	GND	Power	Device ground supply pins
19	RESET#	Input	Active low reset pin. This can be used by an external device to reset the FT232R. If not required can be left unconnected, or pulled up to VCC.
20	VCC	Power	+3.3V to +5.25V supply to the device core.
21	GND	Power	Device ground supply pins
22	CBUS1	I/O	Configurable CBUS I/O Pin.
23	CBUS0	I/O	Configurable CBUS I/O Pin.
24	NC		No internal connection
25	AGND	Power	Device analogue ground supply for internal clock multiplier
26	TEST	Input	Puts the device into IC test mode.
27	OSCI	Input	Input 12MHz Oscillator Cell. Optional – Can be left unconnected for normal operation.
28	OSCO	Output	Output from 12MHZ Oscillator Cell. Optional – Can be left unconnected for normal operation if internal Oscillator is used.

Tabelle 8: Pin-Belegung des FTDI FT232 (Auszug aus dem Datenblatt, grau markierte Pins werden nicht verwendet)

[Zum Kapitel Software: USART]

3. Software

Mit dem Aufbau der Hardware ist der Mikrocontroller betriebsbereit und wartet darauf, Befehle zu erhalten. Zwei Programmiersprachen sind im Bereich eingebettete Systeme weit verbreitet, der hardwarenahe Assembler und die Hochsprache C.

In diesem Praktikum wird C verwendet, es hat neben der Möglichkeit der direkten Speicherzugriffe auch den Vorteil, ohne großen Aufwand externe Bibliotheken in das Programm integrieren zu können und somit den Funktionsumfang zu erweitern.

Die verwendeten Programme WinAVR, AVR Studio und Logview können kostenlos heruntergeladen werden (siehe Anhang: Bezugsquellen).

3.1 Schreiben eines Programms in C und Übertragen zum Mikrocontroller

3.1.1 C-Einführungsbeispiel

Der Einstieg erfolgt direkt mit dem Quellcode der Datei main.c, mit der der µC eine LED ansteuert:

C-Code	Beschreibung
`#include <avr/io.h>`	Präprozessor: • wird beim Kompilieren als erstes aufgerufen • Einbinden externer Header-Dateien mit #include • Ersetzen von Ausdrücken mit #define oder bedingtes Ersetzen mit #ifndef (falls nicht schon definiert, definiere...) zur besseren Lesbarkeit des Codes • io.h enthält eine Beschreibung der ATmega-Register und muss deswegen bei der Mikrocontroller-Programmierung eingebunden werden
	Definitionen (Beispiel S. 29): • Hier können **globale** Variablen definiert werden. Diese können von allen Funktionen verwendet werden. Auf sie können auch extern eingebundene *.c-Files zugreifen, wenn die Variablen dort im Quellcode mit *extern* deklariert sind • Definition von Funktionen: Zur besseren Lesbarkeit können hier größere Funktionen definiert werden. In der Main-Funktion muss diese Funktionen nur noch aufgerufen werden. Dies verbessert die Übersichtlichkeit und verringert die Code-Länge.
`int main(void)` `{`	Main-Funktion: • Beginn des eigentlichen Programms • hier: vom Typ int (Integer) • (void) bedeutet, dass ihr kein Wert übergeben wird • muss in jedem C-Programm aufgerufen werden

`DDRB=(1<<PB3);` `PORTB=(1<<PB3);`	Anweisungen: • Wertezuweisungen werden mit ; abgeschlossen • DDRB und PORTB sind 8-Bit Register • Registerbits sind alle mit 0 initialisiert • Mit DDRB=(1<<PB3) wird dem Bit PB3 des Registers DDRB der Wert 1 zugeordnet. Das bedeutet, dass der Pin PB3 als Ausgang verwendet wird. Alle übrigen Bits des Registers sind weiterhin gelöscht (0) und damit als Eingänge definiert • Analog wird das Bit PB3 im Register PORTB gesetzt. Damit wird der Pin PB3, der zuvor als Ausgang definiert wurde, intern auf Versorgungsspannung gelegt • Die an PB3 angeschlossene LED beginnt zu leuchten

`while(1)`
`{`
`  ;`
`}`
	Main-Schleife:
	• da das Argument der while-Schleife immer 1 (= true) ist, läuft sie zyklisch unendlich oft durch
	• das Programm ist in der Schleife „gefangen“
	• in diesem Beispiel soll nichts getan werden, deswegen die leere Anweisung „ ; “

`return 0;` `}`	Return-Wert • gehört zum Formalismus • main-Funktion ist vom Typ Integer, deswegen muss sie auch einen Integerwert zurückgeben, in diesem Fall 0 • wird nie erreicht und zurückgegeben, da das Programm in der Main-Schleife bleibt

Kommentare im C-Quelltext werden mit dem Ausdruck „`//`“ eingeleitet oder können alternativ so geschrieben werden: `/* Dies ist ein Kommentar*/`. Kommentare dienen der besseren Verständlichkeit des Quelltextes, werden vom Compiler ignoriert und sind im Programmer's Notebook [WinAVR] (siehe Abbildung 19) grün markiert.

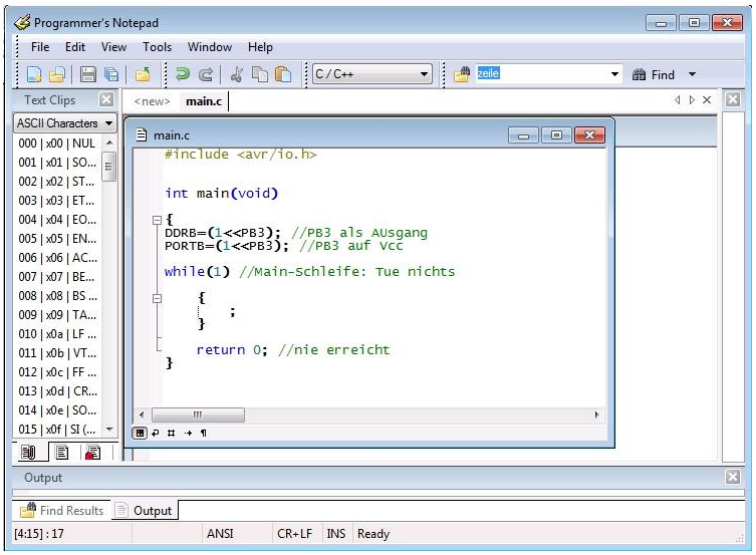


Abbildung 19: main.c: Software Programmer's Notebook [WinAVR]

3.1.2 Makefile

Mit dem Einführungsbeispiel ist das erste kleine Programm schon fertig. Allerdings fehlen dem Compiler noch wichtige Informationen, die er für die Umwandlung in Maschinensprache benötigt. Diese werden im *Makefile* definiert. Das Makefile besitzt keine Datei-Endung und muss sich im selben Verzeichnis wie das Hauptprogramm befinden. Im Verzeichnis \Sample der Win-AVR Software ist ein Makefile zu finden (oder alternativ hier bei mikrocontroller.net), das nach Anpassungen folgender Werte benutzt werden kann (Abbildung 20):

- Mikrocontroller Typ (MCU=atmega16)
- Quarzfrequenz (F_CPU=7372800)
- Name des *.c-Files mit der Main-Funktion (Beim Einführungsbeispiel: TARGET=main)
- Auflisten aller *.c-Files, die in das Hauptprogramm eingebunden werden sollen (für das Einführungsbeispiel: SRC= $(TARGET).c)
  → Wenn externe *.c-Dateien in das Projekt eingebunden werden sollen, müssen sie sich im selben Ordner befinden und im Makefile bei SRC eingetragen werden. (Beispiel: SRC= $(TARGET).c adc.c pwm.c taster.c)
- Gewünschtes Ausgangsformat (FORMAT=ihex)

Nach dem Speichern der Änderungen und dem Kopieren des Makefiles in das Verzeichnis der main.c kann das Einführungsbeispiel kompiliert werden.

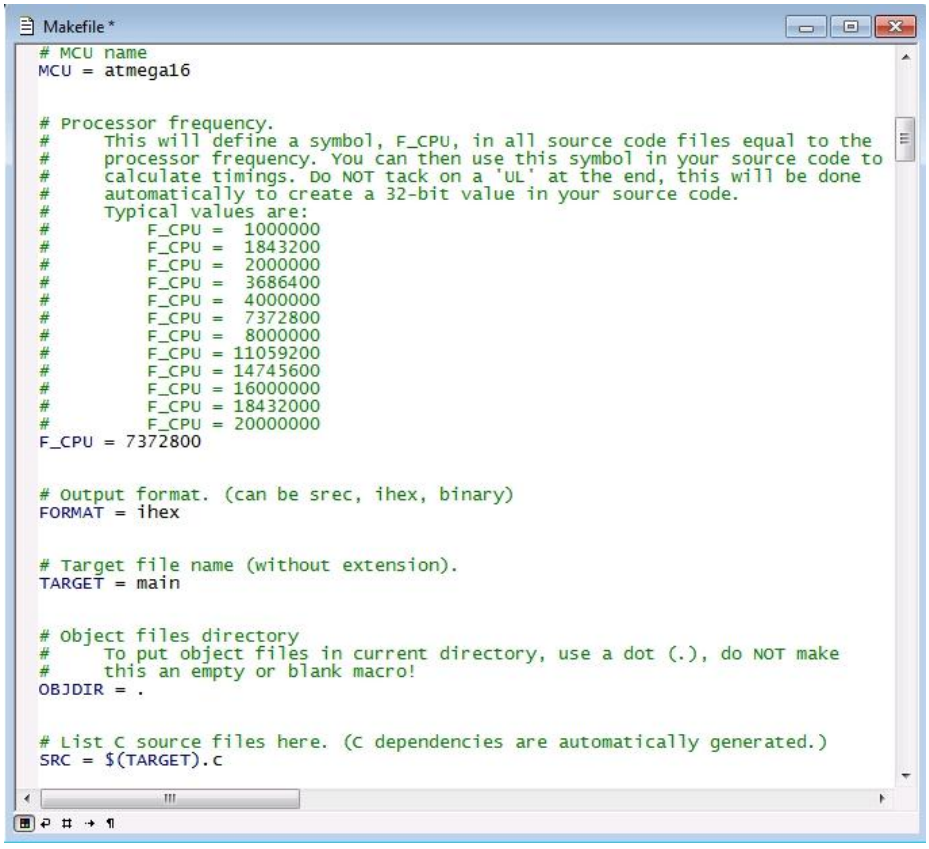


Abbildung 20: Makefile in WinAVR

3.1.3 Kompilieren mit WinAVR

Das Kompilieren mit dem Programmer's Notepad erfolgt unter dem Reiter *Tools → [WinAVR]Make All.*

Das Fenster Output gibt Rückmeldung über den Kompilierungsvorgang (siehe Abbildung 21) und zeigt Warnungen und Fehler im Programm. Wurde der Befehl erfolgreich ausgeführt, befindet sich nun die Datei main.hex im Ordner der main.c und des Makefiles. Diese Datei in Maschinensprache kann in den Programmspeicher des ATmega-16 geladen werden (siehe Kapitel Programmspeicher beschreiben).

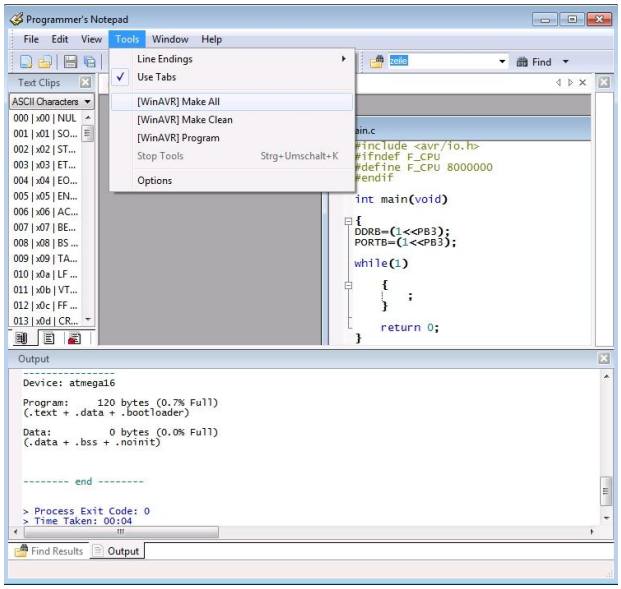


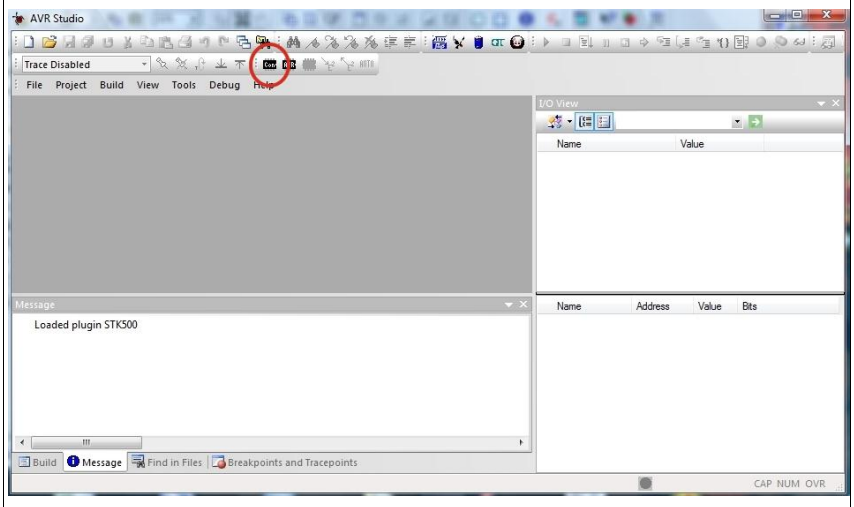
Abbildung 21: Kompilieren unter Programmer's Notepad [WinAVR]

3.1.4 Programmieren des ATmega-16 mit der Software AVR-Studio

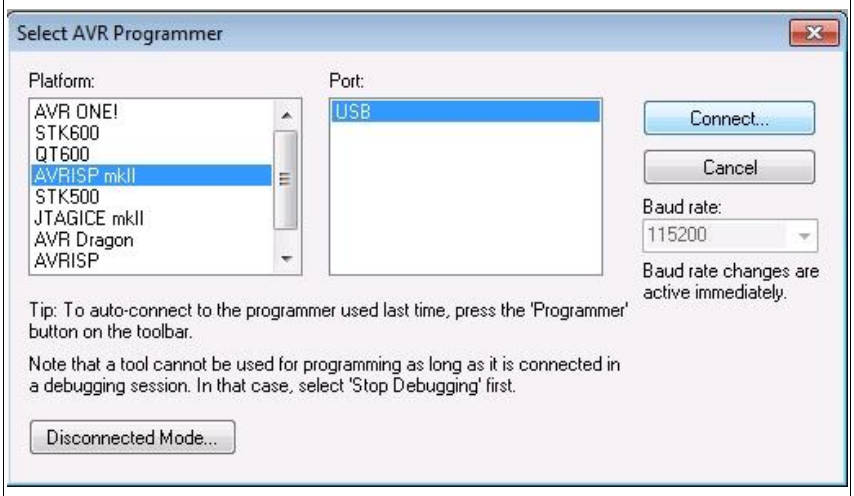
3.1.4.1 Mikrocontroller konfigurieren

Zum Beschreiben des Programmspeichers des ATmega wird die Software AVR-Studio benutzt. Nach der Installation muss zunächst das verwendete Programmiergerät ausgewählt werden. Daraufhin kann sich das Programm mit dem µC verbinden. Über die Fusebits kann die Taktgeberquelle vom internen Oszillator auf den Quarz umgestellt werden. Die Fusebit-Änderungen sind dauerhaft gespeichert, können aber jederzeit geändert werden.

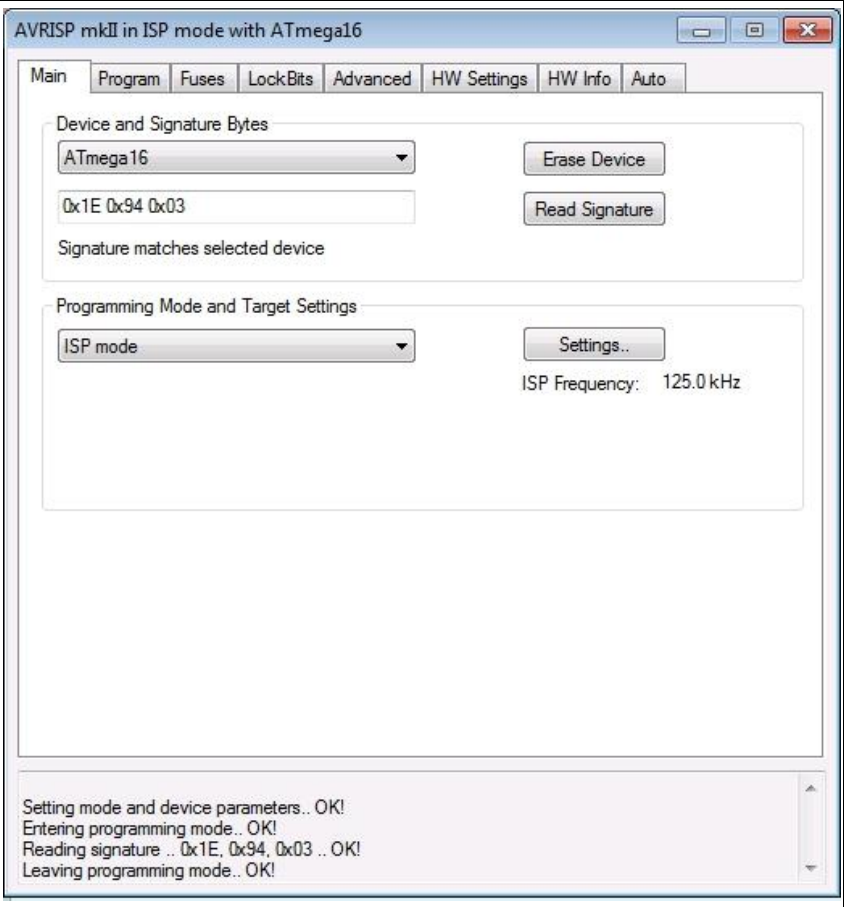
Hier die einzelnen Schritte für die Konfiguration bei der ersten Benutzung:

• Icon *Con* öffnet das Auswahl-Menü für den AVR-Programmer	

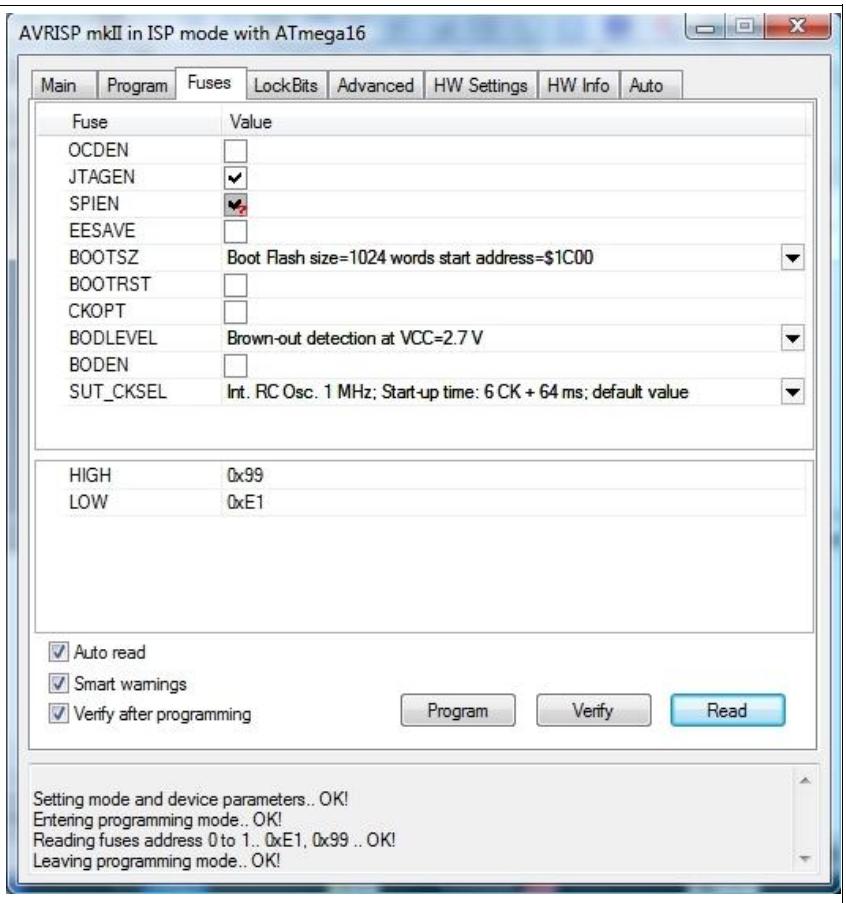
- Auswahl des *AVRISP mkII*
- Bestätigen mit Connect



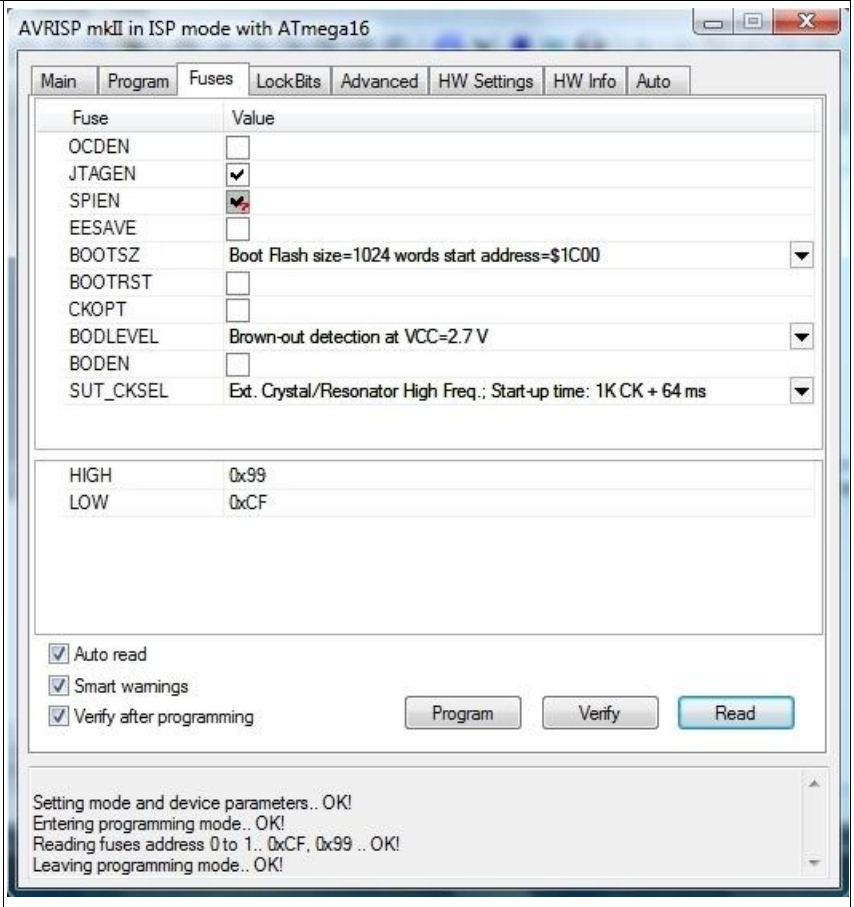
- Es öffnet sich das Fenster *AVRISP mkII*
- Im Main-Register unter *Device and Signature Bytes* Auswählen des ATmega16



- Im Register *Fuses* ist unter SUT_CKSEL die Werkseinstellung für den Taktgeber eingestellt (interner RC-Oszi 1 MHz).



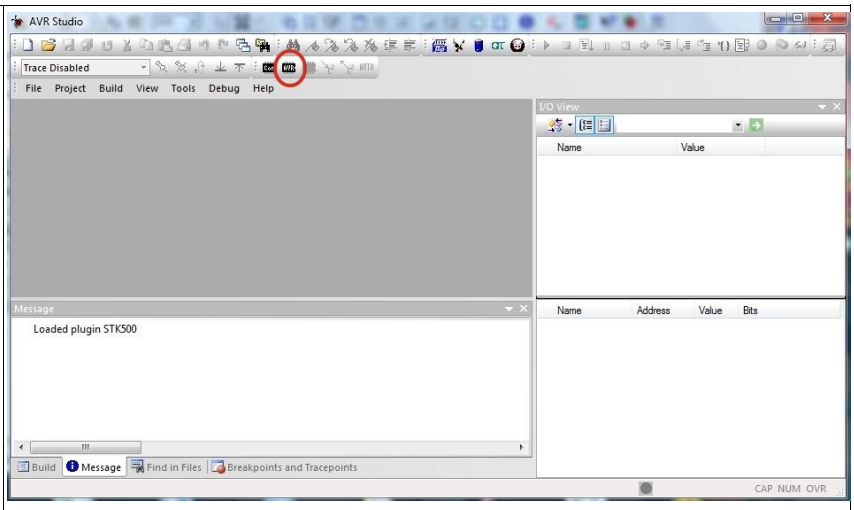
- Für die Verwendung des genaueren externen 7,3728 MHz Quarzes: Auswählen *Ext. Crystal/Resonator High Frequ. Start-up tome 1K CK + 64 ms* aus der Liste
- Speichern der Änderungen mit *Program*



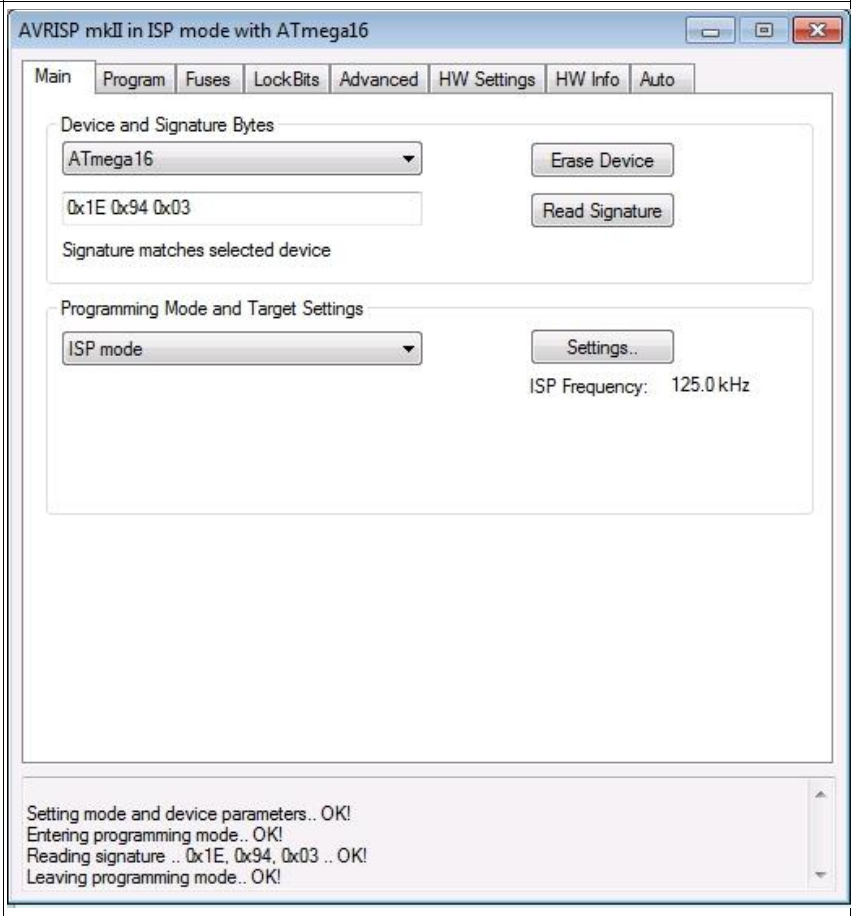
3.1.4.2 Mikrocontroller-Programmspeicher beschreiben in AVR-Studio

Als nächstes kann der Flash-Speicher des µC mit einem kompilierten Programm beschrieben werden:

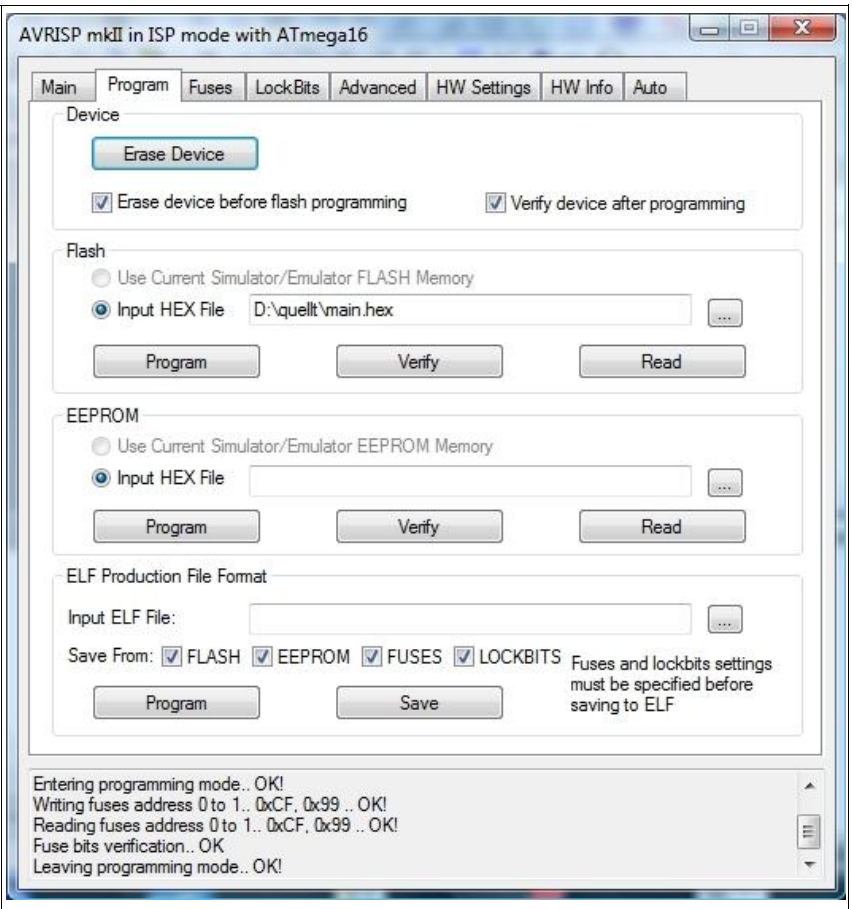
- Icon *AVR* öffnet das Programmier-Menü für den AVR-Programmer



- Im Programmiermenü auf Reiter *Program* klicken



- Im Abschnitt *Flash* Laden des kompilierten *.hex-Files mit ...-Button
- Übertragen mit *Program*
- Die Konsole im unteren Bereich gibt Rückmeldung ob Beschreiben erfolgreich war



3.2 C

3.2.1 Header-Dateien

Je nachdem, welche Funktionen/Definitionen im Quelltext benötigt werden, müssen die entsprechenden Header-Dateien im Präprozessor mit #include <*.h> eingebunden werden.

Hier eine kleine Auswahl:

avr/io.h	Die io.h enthält eine Beschreibung und Zuordnung der Mikrocontroller-Register. Wird für die Programmierung eines ATmega-16 benötigt.
util/delay.h	Bietet die Funktion _delay_ms(Wartezeit [ms]). Wird sie aufgerufen, bleibt das Programm an der Stelle solange stehen, bis die der Funktion übergebene Zeit abgelaufen ist. Solange ist der Programmablauf blockiert. Beispiel: _delay_ms(1000); // Warte eine Sekunde
avr/interrupt.h	Muss bei interrupt-gesteuertem Programmablauf eingebunden werden, also wenn die Interrupt-Service-Routine verwendet werden soll. Im Quellcode werden die Interrupts mit *sei()* aktiviert und mit *cli()* deaktiviert
stdlib.h	Liefert u.a. Funktionen (atof, atoi, atol), um Strings in int, double, usw. umzuwandeln
`stdio.h`	Liefert u.a. die printf-Funktion, die unterschiedliche Datentypen in eine Stringkette schreiben kann. Die scanf-Funktion liest umgekehrt Informationen aus einer Stringkette (die beispielsweise über USB an den ATmega gesendet wurde) aus.
stdint.h	Definition verschiedener Integer-Datentypen (vorzeichenbehaftet, vorzeichenlos, 8/16/32/64-Bit)
math.h	Bietet mathematische Funktionen wie sin(x), sqrt(x), exp(x),etc.

Tabelle 9: Header-Dateien

3.2.2 Datentypen in C (inkl. stdint.h)

Datentyp	Bezeichnung	Größe	Wertebereich	Beispieldefinition/-initialisierung
char	Character (Zeichen)	8 Bit	256 verschiedene Zeichen	char test= '!';
char []	String (Character-Array)	(Anzahl der char +1)*8 Bit	Beliebige Anzahl von Zeichen	char string[]="Das ist ein Test!"
float	Einfache Fließkommazahl	32 Bit	1.5E-45... 3.4E38 (7-8 Stellen)	float Pi=3.14;
double	Doppelt genaue Fließkommazahl	64 Bit	5.0E-324...1.7E308 (15-16 Stellen)	
int8_t	Vorzeichenbehafteter 8-Bit Integer	8 Bit	-128...127	int8_t x= -35;
uint8_t	Vorzeichenloser 8-Bit Integer	8 Bit	0...255	uint8_t y= 213;
int16_t	Vorzeichenbehafteter 16-Bit Integer	16 Bit	-32768...32767	
uint16_t	Vorzeichenloser 16-Bit Integer	16 Bit	0...65535	
int32_t	Vorzeichenbehafteter 32-Bit Integer	32 Bit	-2147483648...2147483647	
uint32_t	Vorzeichenloser 32-Bit Integer	32 Bit	0...4294967295	
int64_t	Vorzeichenbehafteter 64-Bit Integer	64 Bit	−9223372036854775808... 9223372036854775807	
uint64_t	Vorzeichenloser 64-Bit Integer	64 Bit	0 ... 18446744073709551615	

Tabelle 10: Datentypen

Strings werden automatisch mit \0 abgeschlossen. D.h., es besteht ein Unterschied zwischen char c='a' und char c[]="a". Während der Charakter nur aus einem einzigen Zeichen 'a' besteht, beinhaltet der String „a", das Zeichen 'a' und die Endmarkierung \0, ist also in dem Fall doppelt so groß wie der Character.

3.2.3 Kontrollstrukturen in C

Kontrollstukturen führen den Ablauf des Programms im Quellcode. Dazu gehören Bedingungen und Schleifen.

Im Einführungsbeispiel ist bereits eine solche Kontrollstruktur zu finden, die while-Schleife. Sie durchläuft die Anweisungen in geschweiften Klammern solange, bis der Bedingungs-Ausdruck, der ihr übergeben wird, falsch, also gleich 0 ist. Da sie bei jedem Durchgang aber eine 1 (steht für wahr) bekommt, springt das Programm nicht aus der Schleife heraus.

Die wichtigsten Kontrollstrukturen:

- if-Anweisung
- for-Schleife
- while-Schleife
- break (aus Schleife herausspringen)

- continue (Sprung zum Schleifenkopf)
- switch-Anweisung

Link: Wikibooks - Kontrollstrukturen in C (Erklärung und Beispiele)

3.2.4 Funktionen

Um eine Reihe von Anweisungen zusammenzufassen und modularisieren zu können, werden Funktionen definiert. Dadurch wird der Quellcode verständlicher und kürzer, Fehler lassen sich einfacher finden.

Die Definition von Funktionen sind folgendermaßen aufgebaut:

```
Rückgabetyp Funktionsname
(Übergabeparameter)
{
Anweisungen
}
```

Hierzu eine Beispielfunktion *kugeloberflaeche*, die den Radius einer Kugel übergeben bekommt und daraus ihre Oberfläche berechnet:

```
float kugeloberflaeche(float radius)//Rückgabetyp,Übergabetyp
float
{                                    //Beginn der Anweisungen
     float oberflaeche;              //lokale Variable
oberflaeche
     oberflaeche=4*3.14159*radius^2;

     return oberflaeche;              //Rückgabewert vom Typ
float
}                                    //Ende der
Funktionsdefinition
```

Ein Programm mit dieser Funktion könnte dann beispielsweise so aussehen:

```
#include <avr/io.h>

#define Pi 3.14159

float r=13.7; //globale Variable r vom Typ float
float A;         //globale Variable A vom Typ float

float kugeloberflaeche(float radius) //Start
Funktionsdefinition
{
     float oberflaeche;//lokale Variable „oberflaeche“
     oberflaeche=4*Pi*radius^2;

     return oberflaeche;
}                                    //Ende Funktionsdefinition

int main()
{
```

```
A=kugeloberflaeche(r);                    //Funktionsaufruf,
Übergabe

     while(1)
     {
          ;
     }

return 0;
)
```

Auch das Anschalten der LED aus dem Einführungsbeispiel (siehe Kapitel 3.1) ließe sich als Funktion schreiben. Da sie auf die globalen ATmega16-Register zugreift, die in der io.h definiert sind, muss sie keine Variable übergeben bekommen. Auch die Ausgabe eines Rückgabewertes entfällt. In solch einem Fall wird die Funktion mit dem Rückgabetyp *void* (engl. nichtig, leer) versehen, sie gibt also nichts zurück. Auch der Übergabeparameter ist void, somit bekommt die Funktion nichts übergeben.

Die Definition der led_an-Funktion aus dem Einführungsbeispiel sähe so aus:

```
void led_an(void)
{
DDRB|=(1<<PB3);
PORTB|=(1<<PB3);
}
```

Der Aufruf dieser Funktion in der Main-Funktion:

```
led_an();
```

Links: Wikibooks – Funktionen in C (Erklärungen und Beispiele)

3.2.4.1 Nützliche Funktionen

Hier eine Auswahl an nützlichen C-Funktionen:

Funktion	**Header-Datei**	**Beschreibung**
`_delay_ms (uint16_t x)`	util/delay.h	Hält den Programmablauf für die übergebene Zeitspanne (in Millisekunden) an. **Beispiel:** `_delay_ms(1000); //Warte 1 s`

`sprintf(char *buffer, const char *format,...)`	stdio.h	Schreibt eine oder mehrere beliebige Variablen in einen String. Nützlich für die uart_puts(*String) und die lcd_string(*String) Ausgabe.
		Beispiel:
		`char Buffer[20];`
		`int x=3;`
		`int y=9;`
		`sprintf(Buffer,"%i mal %i ergibt %i\r\n",x,y, x*y);`
		`lcd_string(Buffer);`

<table>
<tr><td></td><td></td><td><table>
<tr><th>Variablentyp</th><th>Parameter Syntax</th></tr>
<tr><td>Integer (dezimal)</td><td>%i</td></tr>
<tr><td>Vorzeichenloser Integer (dezimal)</td><td>%u</td></tr>
<tr><td>Float</td><td>%f</td></tr>
<tr><td>Character</td><td>%c</td></tr>
<tr><td>Character-String</td><td>%s</td></tr>
</table></td></tr>
<tr><td><code>sscanf(char *buffer, const char *format,...)</code></td><td>stdio.h</td><td>Untersucht einen String, interpretiert ihn und speichert die ausgelesenen Werte in Variablen. Die Paramter-Syntax ist wie bei sprintf.
Beispiel:
char Buffer[20]=(„PI ist 3.14159“);
float kreiszahl;
sscanf(Buffer, „PI ist %f“, kreiszahl);</td></tr>
<tr><td><code>dtostrf(double Gleitkommazahl, char Anzahl_Gesamtstellen, char Anzahl_Nachkommastellen, *Zielstring)</code></td><td>stdlib.h</td><td>Wandelt eine double- oder float-Variable in einen String um.
Beispiel:
<code>char Buffer[6];</code>
<code>float pi=3.14159;</code>
<code>dtostrf(pi,6 ,5,Buffer);</code></td></tr>
<tr><td><code>double atof(const char *s)
int atoi(const char *s)
long atol(const char *s)</code></td><td>stdlib.h</td><td>Wandelt String *s in double (atof), Integer (atoi), long Integer (atol) um</td></tr>
<tr><td><code>strtol(const char * str, char ** endptr, int base)</code></td><td>stdlib.h</td><td>Wandelt String in long Integer mit beliebiger Basis. Speichert Zeiger auf Reststring.
Beispiel:
<code>buffer[6]="32000";</code>
<code>long int x;</code>
<code>x=strtol(buffer, NULL, 10);</code></td></tr>
</table>

Links:

Wikipedia – Input/Output-Functions in C (stdio.h)

FH Fulda – C Standard Bibliothek (stdlib.h)

3.2.5 Definition, Deklaration und Initialisierung

Häufig werden die Begriffe Definition und Deklaration verwechselt, bzw. herrscht Unklarheit, wann Funktionen und Variablen deklariert werden müssen.

Definition:

- Bei der Definition einer **Variable** wird ihr ein Bereich im Speicher zugeteilt
- Beispiel: `uint32_t i;`
- Bei der Definition einer **Funktion** werden Name, Datentyp, Übergabewerte und die Anweisungen festgelegt, die bei ihrem Aufruf abgearbeitet werden sollen
- Beispiel:

```
void init_outputs(void)
{
      DDRB|=(1<<PB1);
}
```

Deklaration:

- sagt dem Compiler, dass eine Variable oder Funktion bereits irgendwo im Quelltext definiert ist und man sich auf sie beziehen möchte
- wird in größeren Projekten mit mehreren Dateien benötigt um auf globale Variablen und Funktionen zugreifen zu können
- Um eine globale **Variable** in einer Header-Datei zu deklarieren (die beispielsweise in einer anderen Datei des Projekts definiert ist), benutzt man das Schlüsselwort extern
- Beispiel: `extern uint32_t i;`
- Um eine **Funktion** zu deklarieren wird ihr Name, Datentyp und die Übergabewerte mit dem Compiler bekannt gemacht
- Beispiel: `void init_outputs(void);`

Initialisierung:

- Die Initialisierung weist einer **Variable** im Zuge der Definition einen Anfangswert zu
- Beispiel: `uint32_t i=14253;`

3.2.6 Sequentieller und interruptbasierter Programmablauf

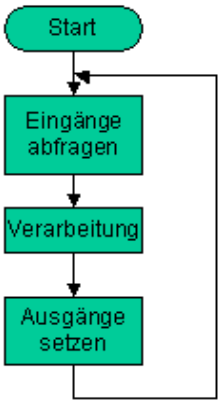
<table>
<tr><th>Sequentieller Ablauf</th><th>Interruptgesteuerter Ablauf</th></tr>
<tr><td> 

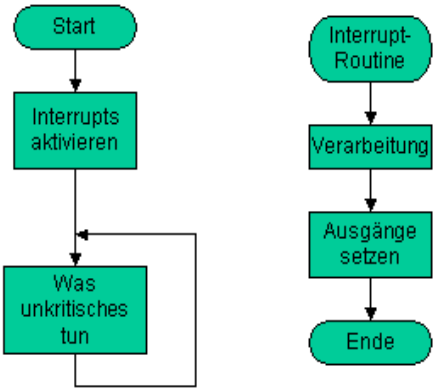
Abbildung 22: Sequentieller Programmablauf
(Quelle: mikrocontroller.net)</td><td> 

Abbildung 23: Programm mit Interrupten
(Quelle: mikrocontroller.net)</td></tr>
<tr><td>Prinzip:
Programm wird Schritt für Schritt abgearbeitet, keine Sprünge
Andauerndes Abfragen der Eingänge (Polling) ineffizient</td><td>Prinzip:
In der Main-Schleife wird sequentiell ein Programm abgearbeitet
Beim Auftreten eines Interrupts: Sprung in die Interrupt-Routine
Nach Durchlaufen der Routine: Sprung zurück an die unterbrochene Programmstelle
Vorgehensweise:
Definieren der Interrupt Service Routine (ISR), Übergabe der ISR-Vektoren
Einbinden der interrupt.h im Präprozessor
Aktivieren der Interrupts im µC-Register
Aktivieren der Interrupts mit sei(); im Programm
Hinweis:
ISR-Funktionen sollten so kurz wie möglich sein (keine Schleifen, kein printf, usw.)
Rechenintensive Operationen können in die Main-Schleife ausgelagert werden und durch ein in der ISR gesetztes Flag aufgerufen werden</td></tr>
</table>

3.3 Register

Um dem ATmega mitzuteilen was er wie zu tun hat, muss man sich mit seinen Registern auseinandersetzen. Register sind thematisch geordnete Bereiche im Speicher des µC. Jedes Register ist einer Mikrocontroller-Funktion zugeordnet und wird durch Setzen oder Löschen seiner 8 Bits konfiguriert. Das Ansprechen des Analog-Digital-Wandlers, Aktivieren von Interrupts und Schalten von Ausgängen – all das erreicht der Programmierer durch Manipulation der Bits im entsprechenden Register.

3.3.1 Bitmanipulation

3.3.1.1 Bitweise Operatoren

Mit den bitweisen Operatoren steht ein Werkzeug zur Verfügung, mit dem Register geändert und somit die Funktionen des µC durch das Setzen der entsprechenden Bits konfiguriert werden können.

Bitweises NICHT (Operator „~“)

Bit	**~ Bit**
0	1
1	0
Byte	**~ Byte**
10110100	01001011

Bitweises ODER (Operator „|“)

Bit 1	**Bit 2**	**Bit 1 | Bit 2**
0	0	0
0	1	1
1	0	1
1	1	1
Byte 1	**Byte 2**	**Byte 1 | Byte 2**
10010100	01011010	11011110

Bitweises UND (Operator „&“)

Bit 1	**Bit 2**	**Bit 1 & Bit 2**
0	0	0
0	1	0
1	0	0
1	1	1
Byte 1	**Byte 2**	**Byte 1 & Byte 2**
10010100	01011010	00010000

3.3.1.2 Register konfigurieren

Initialisiert sind alle Registerbits bei Programmstart mit 0. Im Datenblatt des ATmega-16 sind die Bedeutungen der einzelnen Registerbits detailliert aufgeführt. Das heißt der Programmierer sollte sich zunächst informieren welche Bits er für welche Funktion setzen muss.

Das gezielte Setzen eines Bits in einem Register:

`PORTB=PORTB|(1<<PB0);` oder kürzer: `PORTB|=(1<<PB0);`

Syntax	**Bitmaske**
PORTB	1-0-0-0-0-0-0-0
(1<<PB0)	0-0-0-0-0-0-0-1
PORTB|=(1<<PB0)	1-0-0-0-0-0-0-1

Durch das bitweise ODER werden nur Änderungen am PB0-Bit von PORTB vorgenommen, die übrigen Bits bleiben unangetastet. Die Syntax (1<<n) bewirkt, dass eine 1 in der Bitmaske n mal nach links geschoben wird. PB0 ist eine Umschreibung für Bit 0 des Registers, (1<<PB0) bewirkt also eine 0-malige bitweise Verschiebung der 1 nach links und erzeugt damit die Bitmaske 0-0-0-0-0-0-0-1.

Das gezielte Löschen eines Bits in einem Register:

`PORTB=PORTB&~(1<<PB0);` oder kürzer: `PORTB&=~(1<<PB0);`

Syntax	**Bitmaske**
PORTB	0-1-1-0-0-0-0-1
(1<<PB0)	0-0-0-0-0-0-0-1
~(1<<PB0)	1-1-1-1-1-1-1-0
PORTB&=~(1<<PB0)	0-1-1-0-0-0-0-0

Durch das bitweise UND der negierten Bitmaske (1<<PB0) wird nur PB0 im Register PORTB gelöscht, alle anderen Bits behalten ihren ursprünglichen Zustand.

Das komplette Überschreiben eines Registers:

`PORTB=(1<<PB0);`

Syntax	**Bitmaske**
PORTB	0-1-0-1-0-0-0-0
(1<<PB0)	0-0-0-0-0-0-0-1
PORTB=(1<<PB0)	0-0-0-0-0-0-0-1

Mit dieser Schreibweise werden alle Bits vom Register PORTB überschrieben, alle ursprünglichen Zustände gehen verloren.

Ziel	Syntax	Ergebnis Register PORTA							
		Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
		PA7	**PA6**	**PA5**	**PA4**	**PA3**	**PA2**	**PA1**	**PA0**
		0	0	0	0	0	0	0	0
Setzen von PA0 ohne übrige Bits zu manipulieren	PORTA|=(1<<PA0);	0	0	0	0	0	0	0	1
Setzen von PA3 und 5 ohne übrige Bits zu manipulieren	PORTA|=(1<<PA3)|(1<<PA5);	0	0	1	0	1	0	0	1
Alle PORTA-Bits negieren	PORTA~=PORTA;	1	1	0	1	0	1	1	0
Löschen von PA1 ohne übrige Bits zu manipulieren	PORTA&=~(1<<PA1);	1	1	0	1	0	1	0	0
Löschen von PA7 und 6 ohne übrige Bits zu manipulieren	PORTA&= ~(1<<PA6)&(~(1<<PA7));	0	0	0	1	0	1	0	0

Tabelle 11: Beispiel für Registeroperationen an PORTA

3.3.2 Die Register des ATmega-16

3.3.2.1 Ein- und Ausgänge lesen und schalten

Die Pins eines Mikrocontrollers können sowohl als Ein- oder Ausgänge konfiguriert werden. Der Zustand von Eingangspins kann im PIN-Register gelesen werden und ist zu logisch 1 gesetzt wenn die anliegende Spannung größer als 0,7*Vcc und zu logisch 0 wenn sie kleiner als 0,2*Vcc ist. Die Spannung der Ausgangspins kann im PORT-Register entweder auf Vcc oder GND gelegt werden. Damit können zum Beispiel LEDs an- und ausgeschaltet werden.

Der ATmega-16 besitzt 32 I/O-Pins, die in vier Ports mit jeweils 8 Pins unterteilt werden: A, B, C und D (siehe Kapitel Mikrocontroller).Ein Register bezieht sich auf einen solchen Port (letzter Buchstabe) und besteht aus 8 Bits, die die einzelnen Pins repräsentieren.

Register	Beschreibung
DDRA / DDRB / DDRC / DDRD	• **Richtungsregister**: Setzt Pins als Ein- oder Ausgang • Initialisiert mit 0-0-0-0-0-0-0-0 • Bit gesetzt (1): als Ausgang konfiguriert • Bit gelöscht (0): als Eingang gesetzt
PINA / PINB / PINC / PIND	• Zugriffsregister für **Eingangspins** • Abfragen des Pinzustands • Bit gesetzt (1): Potential von Pin auf high • Bit gelöscht (0): Potential von Pin auf low
PORTA / PORTB / PORTC / PORTD	• Schreibregister für **Ausgangspins** • Setzen und Löschen von Ausgängen • Bit gesetzt (1): Setze Ausgangspin auf high (Vcc) • Bit gelöscht (0): Setze Ausgangspin auf low (Masse)

Tabelle 12: Ein-/Ausgangsregister Atmega-16

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0	0	0	0	1	1	1	1

Tabelle 13: Beispiel für Register DDRB: Pins PB0-3 als Ausgangspins

Schalten von LEDs (Quelltext Beispiele/LED/leds.c)

```
/*Beispiel für LED-Ansteuerung: Lauflicht*/

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

int main (void)
{

DDRB=(1<<PB0)|(1<<PB1)|(1<<PB2)|(1<<PB3);//Schalte Pins PB0,1,2,3 als
Ausgang

while(1)

    {
    PORTB=(1<<PB2);
    _delay_ms(100);
    PORTB=(1<<PB0);
    _delay_ms(100);
    PORTB=(1<<PB1);
    _delay_ms(100);
    PORTB=(1<<PB3);
    _delay_ms(100);

    }

}
```

[Zum Kapitel Hardware: LED]

3.3.2.2 Externe Interrupts konfigurieren

Die Eingangspins INT0, 1 und 2 bieten die Möglichkeit, Interrupts bei einer bestimmten Spannung oder Potentialänderung auszulösen und die Interrupt-Service-Routine aufzurufen. Über externe Signale lassen sich damit Programmfunktionen aufrufen, die beispielsweise LEDs schalten oder Variablen ändern.

Register	Funktion
MCUCR	• Konfiguriert Interrupt-Pins INT1,0
MCUCSR	• Konfiguriert Interrupt-Pin INT2
GICR	• Aktiviert externe Interrupts
GIFR	• Interrupt-Flag Register • wird automatisch gesetzt und bei der Übergabe des Interrupt-Vektors gelöscht • Zugriff vom Programmierer i.d.R. nicht notwendig

Tabelle 14: Die Register für externe Interrupts

MCUCR-Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00

Tabelle 15: Micro Controller Unit Control Register (MCUCR)

ISC11	ISC10	Beschreibung
0	0	Interrupt bei Pin INT1 auf low Level
0	1	Interrupt bei logischer Eingangsänderung von INT1
1	0	Interrupt bei fallender Flanke von INT1
1	1	Interrupt bei steigender Flanke von INT1

Tabelle 16: ISC11..0 Bits des MCUCR

ISC01	ISC00	Beschreibung
0	0	Interrupt bei Pin INT0 auf low Level
0	1	Interrupt bei logischer Eingangsänderung von INT0
1	0	Interrupt bei fallender Flanke von INT0
1	1	Interrupt bei steigender Flanke von INT0

Tabelle 17: ISC01..0 Bits des MCUCR

MCUCSR-Register

Das MCU Control und Status Register (MCUCSR):

Bit	Bezeichnung	Beschreibung
6	ISC2	• Bit gesetzt (1): Interrupt bei steigender Flanke von INT2 • Bit gelöscht (0): Interrupt bei fallender Flanke von INT2

Tabelle 18: ISC2 Bit des MCUCSR

GICR-Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT1	INT0	INT2	-	-	-	IVSEL	IVCE

Tabelle 19: General Interrupt Control Register (GICR)

Bit	Bezeichnung	Beschreibung
7	INT1	Aktiviert den externen Interrupt an INT1
6	INT0	Aktiviert den externen Interrupt an INT0
5	INT2	Aktiviert den externen Interrupt an INT2

Tabelle 20: INT2..0 Bits des GICR

Interrupt -Vektoren für ISR

Die Interrupt -Vektoren der externen Interrupts für die Interrupt-Service-Routine:

Vektor	Beschreibung
INT0_vect	Interrupt-Vektor für Pin INT0
INT1_vect	Interrupt-Vektor für Pin INT1
INT2_vect	Interrupt-Vektor für Pin INT2

Tabelle 21: Interrupt Vektoren INT2..0_vect

Schalten von LEDs mittels Taster 0 und 1 (Quelltext Beispiele/Taster/taster.c)

```
/*Beispiel für Interrupt-gesteuerten Programmablauf: LEDs mit Taster
umschalten*/

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

void taster_init(void)
{
MCUCR=(1<<ISC11)|(1<<ISC10)|(1<<ISC01)|(1<<ISC00); // INT1,0 Interrupt bei
steigender Flanke
GICR=(1<<INT1)|(1<<INT0); //aktiviere INT1,0
}

ISR(INT1_vect) //Interrupt Taster 1
{
PORTB=(1<<PB1); //Schalte grüne LED an
}

ISR(INT0_vect) //Interrupt Taster 1
{
PORTB=(1<<PB2); //Schalte rote LED an
}

int main(void)
{
sei(); //Aktiviere Interrupts
DDRB=(1<<PB1)|(1<<PB2)|(1<<PB0); // Schalte Pins (mit LED) als Ausgang
taster_init();
PORTB=(1<<PB0);

while(1)
{}
}
```

[Zum Kapitel Hardware: Taster]

3.3.2.3 ADC initialisieren und Wandlung starten

Die Analog-Digital-Wandlung lässt sich über zwei Register des Atmega im Quellcode konfigurieren und starten, das Ergebnis kann im Datenregister ausgelesen werden:

ADC-Register	**Funktion**
ADMUX	Bestimmt ADC-Eingangspin und ADC-Spannungsreferenz
ADCSRA	Steuert ADC, bestimmt Wandlungsmodus und Frequenz
ADCW	Datenregister in welches der 10-Bit ADC-Wert geladen wird (besteht aus den zwei Registern ADCH und ADCL)

ADMUX-Register

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0

Tabelle 22: ADC-Register ADMUX

REFS1	**REFS0**	**Spannungsreferenz**
0	0	AREF, interne Vergleichsspannung deaktiviert
0	1	AVCC
1	0	Reserviert
1	1	Intern erzeugte 2,56V Spannungsreferenz

Tabelle 23: Bedeutung der REFS0/1 Bits des ADMUX-Registers

MUX4..0	**Eingang**
OOOOO	ADC0
OOOO1	ADC1
OOO1O	ADC2
OOO11	ADC3
OO1OO	ADC4
OO1O1	ADC5
OO11O	ADC6
OO111	ADC7

Tabelle 24: MUX4..0 des ADMUX-Registers: Auswahl des ADC-Eingangspins

ADCSRA-Register

Das ADCSRA steuert den Wandlungsvorgang:

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Tabelle 25: ADC Control and Status Register A (ADCSRA)

Bit	Bezeichnung	Bedeutung
7	ADEN	• Bit gesetzt (1): ADC aktiviert • Bit gelöscht (0) ADC deaktiviert
6	ADSC	• Single Conversion Mode: - Bit setzen startet einmalige Wandlung - Nach ADC-Wandlung wird Bit gelöscht • Free Running Mode: - Bit setzen startet ADC-Wandlung in Endlosschleife - ADC-Datenregister ADCW wird laufend automatisch aktualisiert
5	ADATE	• Bit gesetzt (1): Free Running Mode • Bit gelöscht (0): Single Conversion Mode
4	ADIF	• ADC Interrupt Flag - gesetzt wenn ADC Wandlung abgeschlossen - wird gelöscht wenn zugehöriger ADC_vect der ISR übergeben wird
3	ADIE	• Aktiviert das ADC Interrupt Flag, wenn zu 1 gesetzt
2..0	ADPS2..0	• Bestimmt Teilungsfaktor zwischen Quarz- und ADC-Frequenz

Tabelle 26: Bedeutung der Bits des ADCSRA-Registers

ADPS2..0	Teilungsfaktor
OOO	2
OO1	2
O1O	4
O11	8
1OO	16
1O1	32
11O	64
111	128

Tabelle 27: AD Prescaler Select 2..0 Bits des ADCSRA-Registers

Konfigurieren und Durchführen einer AD-Wandlung (Quelltext Beispiele/ADC/adc.c)

```
/*Beispiel für ADC-Polling*/

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

uint16_t adc_value;

void ADC_Init(void) //ADC initialisieren
{
   ADMUX = (0<<REFS1) | (1<<REFS0)|(1<<MUX0)|(1<<MUX1);      // AVCC
Referenzspannung(5V), ADC3 aktiv
  ADCSRA = (1<<ADPS1) | (1<<ADPS0);     // Frequenzvorteiler 8 -> ADC
Frequenz ca. 922kHz
  ADCSRA |= (1<<ADEN);                  // ADC aktivieren
}
```

```
uint16_t ADC_Read(void) //ADC Einzelmessung
{
  // Kanal waehlen, ohne andere Bits zu beeinflußen
     _delay_ms(10);
  ADCSRA |= (1<<ADSC);            // eine Wandlung "single conversion"
  while (ADCSRA & (1<<ADSC) )     // auf Abschluss der Konvertierung warten
    ;

  return ADCW;                    // ADC auslesen und zurückgeben
}


int main(void)
{

      ADC_Init();

            while(1)
            {
                  adc_value=ADC_Read();
            }
      return 0;
}
```

[Zum Kapitel Hardware: ADC]

3.3.2.4 Timer und Pulsweitenmodulation konfigurieren und starten

Timer sind aus den meisten Mikrocontroller-Anwendungen nicht wegzudenken. Sie gehören zu den Peripheriefunktionen des Controllers und laufen nebenläufig zu dem eigentlichen Programm. Durch die Verwendung von Timer Interrupten können auf diese Weise im Quellcode flexible Wartezeiten realisiert werden ohne dass das gesamte Programm in dieser Zeit angehalten wird (wie z.B. bei Verwendung der _delay_ms()-Funktion). Eine weitere wichtige Anwendung des Timers ist die Pulsweitenmodulation.

Die Erläuterung der Timer-Funktionen erfolgt anhand des 8-Bit Timer/Counters 0. Sollte ein 16-Bit Timer (Timer/Counter 1) benötigt werden, hilft das ATmega-16-Datenblatt ab Seite 89 weiter. Mit Timer/Counter 2 steht ein zweiter 8-Bit Timer zur Verfügung (Datenblatt ab S. 117).

Register	**Beschreibung**
TCCR0	• Konfiguration des Timers
TCNT0	• Aktueller 8-Bit Zählerstand des Timers 0 • kann gelesen oder gesetzt werden
OCR0	• 8-Bit Timer-Vergleichswert (Output Compare Register) • Beschreiben des Registers weist PWM-Wert zu
TIMSK	• Aktiviert Timer Interrupts
TIFR	• Interrupt-Flag Register • wird automatisch gesetzt und bei der Übergabe des Interrupt-Vektors gelöscht • Zugriff vom Programmierer i.d.R. nicht notwendig

Tabelle 28: Register für Timer/Counter 0

TCCR0-Register

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

Tabelle 29: Timer/Counter Control Register (TCCR0)

Bit	**Bezeichnung**	**Funktion**
7	FOC0	• Bit wird gesetzt (1) im Nicht-PWM-Timer Betrieb
3,6	WGM01..0	• Konfiguriert PWM-Modus
4,5	COM01..0	• Konfiguriert das Verhalten des Output Compare Pins OC0 • Bei normalem Betrieb gelöscht (00) • für abweichende Konfiguration siehe Datenblatt S.84
0,1,2	CS02..00	• Auswahl des Taktgebers und Frequenzteilers

Tabelle 30: Beschreibung TCCR0

WGM01	WGM00	Timer/Counter Betriebsmodus
0	0	Normal (Nicht PWM)
0	1	PWM, phasenkorrekt
1	0	CTC
1	1	Fast-PWM

Tabelle 31: WGM01..0 Bits des TCCR0 Registers

COM01	COM00	Beschreibung
0	0	Ausgangspin OC0 deaktiviert
0	1	Reserviert
1	0	Nicht-invertierter PWM-Modus
1	1	Invertierter PWM-Modus

Tabelle 32: COM01..0 Bits des TCCR0 Registers

CS02	CS01	CS00	Beschreibung
0	0	0	Timer gestoppt (Keine Taktgeber zugeordnet)
0	0	1	Frequenzteiler 1
0	1	0	Frequenzteiler 8
0	1	1	Frequenzteiler 64
1	0	0	Frequenzteiler 256
1	0	1	Frequenzteiler 1024
1	1	0	Externer Taktgeber an T0 Pin (fallende Flanke)
1	1	1	Externer Taktgeber an T0 Pins (steigende Flanke)

Tabelle 33: CS02..0 Bits des TCCR0 Registers

TIMSK-Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	OCIE0	TOIE0

Tabelle 34: Timer/Counter Interrupt Mask Register (TIMSK)

Bit	Bezeichnung	Beschreibung
1	OCIE0	• Bit gesetzt (1): Aktiviert Interrupt bei Output Compare Match (Timerwert im TCNT0 Register stimmt mit dem im OCR0 Vergleichswert überein)
0	TOIE0	• Bit gesetzt (1): Aktiviert Timer0 Overflow-Interrupt (Zählwert des 8-Bit Timers läuft bei 255 über und beginnt wieder bei 0 zu zählen)

Tabelle 35: Beschreibung TIMSK-Register

Timer-Interrupt -Vektoren für die ISR:

Vektor	Beschreibung
TIMER0_OVF_vect	Overflow Interrupt-Vektor Timer 0 (Überlaufen von TCNT0)
TIMER0_COMP_vect	Compare Match Interrupt-Vektor Timer 0

Tabelle 36: Timer 0 Interrupt-Vektoren

Erzeugen eines Dreiecksignals am Ausgangspin OC0 (Quelltext Beispiele/PWM/pwm.c)

```
/*Beispiel für PWM-Ausgabe an blauer LED: Dreieckssignal*/

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

void PWM_init(void)
{
      TCCR0 = (1<<WGM01)|(1<<WGM00)|(1<<COM01)|(0<<CS02)|(1<<CS00);//Starte
Timer mit f=f_CPU, Fast-PWM Mode, nicht-invertiert
      DDRB|=(1<<PB3);// Setzen des OC0 Pins als Ausgang
}

int main(void)
{
      PWM_init();
      uint8_t a=0;
      int8_t b=1;


      while(1)
      {
            a=a+b;
            OCR0=a; //Erzeuge Dreiecksignal am OC0 Pin, gestreckt auf den
Bereich GND-Vcc
            _delay_ms(5);

            if (a==255)
            {b=-1;}
            if (a==0)
            {b=1;}

      }

}
```

Manche Pins besitzen mehrere Hardware-Funktionen (siehe Kapitel Mikrocontroller). PB3, der hier als PWM-Ausgangspin OC0 benutzt wird, besitzt außerdem eine Komparator-Funktion (AIN1). Die Registerkonfiguration bestimmt den Verwendungszweck des Pins.

[Zum Kapitel Hardware: PWM]

3.3.2.5 USART konfigurieren und Daten senden

Auch wenn die USART-Register anfangs ein wenig kompliziert und unübersichtlich wirken, sind im Grunde nur wenige Einstellungen für den Standard-Asynchronbetrieb nötig. Für die Initialisierung im UCSRB-Register müssen Empfänger und Sender (RXEN, TXEN) aktiviert werden und die Interrupts RXCIE („Ungelesene Daten empfangen“-Flag) und UDRIE („Schreibbuffer bereit“-Flag) gesetzt werden, die beim Senden und Empfangen verwendet werden. Im UCSRC-Register wird Framegröße (UCSZ) und Anzahl der Stoppbits (USBS) festgelegt und im Register UBRR die Geschwindigkeit konfiguriert. Danach ist die USART-Schnittstelle bereit zur Kommunikation.

Register	**Beschreibung**
UDR	• Empfangs-/Sendedatenbuffer Register
UCSRA	• Interrupt-Flag-Register
UCSRB	• Konfiguriert Interrupts und Framegröße, aktiviert Sender und Empfängerpin
UCSRC	• Konfiguriert Stoppbits, Betriebsmodus, usw.
UBRR	• Konfiguriert Baudrate des USART

Tabelle 37: Register für USART-Kommunikation

UCSRA

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
RXC	**TXC**	**UDRE**	**FE**	**DOR**	**PE**	**U2X**	**MPCM**

Tabelle 38: USART Control and Status Register A (UCSRA)

Bit	**Bezeichnung**	**Beschreibung**
7	RXC	• Flag-Bit: Gesetzt wenn ungelesene Daten im USART-Empfangs-Buffer • wird gelöscht wenn Daten ausgelesen wurden
6	TXC	• Flag-Bit: Gesetzt wenn Daten im USART-Sendebuffer komplett übertragen wurden und neue Daten versendet werden können
5	UDRE	• Flag-Bit: Gesetzt wenn USART-Sendebuffer bereit ist mit neuen Daten beschrieben zu werden
4	FE	• Flag-Bit: Gesetzt bei Frame Error des aktuell empfangenen Char
3	DOR	• Data-Overrun-Flag: Gesetzt wenn Empfangsbuffer überschrieben wurde, obwohl die alten Daten noch nicht ausgelesen wurden
2	PE	• Parity-Error-Flag: Gesetzt wenn im aktuellen Char des Empfangsbuffer ein Parity Error detektiert wurde
1	U2X	• Synchroner Betrieb: Setze Bit zu Null • Asynchroner Betrieb: - Gesetzt: Halbiert Baudraten-Teilungsfaktor auf 8 → Verdoppelt USART-Transferrate - Gelöscht: Standard Baudraten-Teilungsfaktor 16
0	MPCM	• Gesetzt: Aktiviert Multi-Prozessor-Modus • alle empfangenen Datenframes, die nicht eine Adressinformation enthalten, werden ignoriert

Tabelle 39: Beschreibung des UCSRA

UCSRB

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

Tabelle 40: USART Control and Status Register B (UCSRB)

Bit	Bezeichnung	Beschreibung
7	RXCIE	• Wenn gesetzt: RXC Interrupt (UCSRA-Register) aktiviert
6	TXCIE	• Wenn gesetzt: TXC Interrupt (UCSRA-Register) aktiviert
5	UDRIE	• Wenn gesetzt: UDR Interrupt (UCSRA-Register) aktiviert
4	RXEN	• Bit gesetzt: USART-Empfangspin RX aktiviert
3	TXEN	• Bit gesetzt: USART-Sendepin TX aktiviert
2	UCSZ2	• Konfiguriert zusammen mit den UCSZ1..0 Bits im UCSRC-Register die Größe von Empfangs-/Sendeframes
1	RXB8	• Falls 9-Bit Frame konfiguriert: 9. Empfangsbit (MSB) • muss dann vor dem Hauptbuffer ausgelesen werden
0	TXB8	• Falls 9-Bit Frame konfiguriert: 9. Sendebit (MSB) • muss dann vor dem Hauptbuffer beschrieben werden

Tabelle 41: Beschreibung des UCSRB-Registers

UCSRC

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

Tabelle 42: USART Control and Status Register C (UCSRC)

Bit	Bezeichnung	Beschreibung
7	URSEL	• Auswahl-Bit zwischen UCSRC und UBRRH • Muss gesetzt werden wenn UCSRC geändert/beschrieben werden soll
6	UMSEL	• Konfiguriert USART-Modus • Bit gesetzt (1): Synchroner Betrieb • Bit gelöscht (0): Asynchroner Betrieb
5,4	UPM1..0	• Konfiguriert und aktiviert Parity-Check
3	USBS	• Konfiguriert Stop-Bits • Bit gesetzt (1): 2 Stoppbits • Bit gelöscht (0): 1 Stoppbit
2,1	UCSZ1..0	• Konfiguriert Anzahl der Datenbits in einem Sende-/Empfangsframe
0	UCPOL	• Konfiguriert USART-Taktgeber im synchronen Betriebsmodus • Datenblatt S. 167

Tabelle 43: Beschreibung des UCSRC-Registers

UBRR

UBRR ist ein 12-Bit Wert und verteilt sich auf zwei Register. Es beschreibt die Baudrate der USART-Kommunikation, also wie viele Symbole pro Sekunde versendet oder empfangen werden.

Aus der Baudrate (typische Werte z.B. 9600, 19200 oder 115200 Symbole/s) lässt sich der Wert für UBRR ermitteln:

$$UBRR = \frac{Taktfrequenz[Hz]}{16 \cdot Baudrate} - 1$$

Da bei dieser Berechnung auf ganze Zahlen „gerundet“ wird (d.h. es werden die Nachkommastellen weggestrichen), muss außerdem überprüft werden, ob der Abrundungsfehler kleiner als 1% ist, damit es keine Übertragungsfehler gibt:

$$Rundungsfehler = \left(\frac{UBBR_{gerundet} + 1}{UBBR_{genau} + 1} - 1\right) \cdot 100$$

Ist dies sichergestellt, kann UBRR beschrieben werden, z.B so:

```
#define BAUD        115200
#define UBRR_BAUD   ((F_CPU/(16*BAUD))-1)
UBRRH = (unsigned char) (UBRR_BAUD>>8);
UBRRL = (unsigned char) (UBRR_BAUD);
```

UBRRH	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
	URSEL	-	-	-	UBRR			
UBRRL	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 3**	**Bit 1**	**Bit 0**
UBBR								

Tabelle 44: USART Baudrate Register (UBBR), unterteilt in die zwei 8-Bit Register UBRRH/L

Initialisieren des USART und Schreiben eines Strings (Quelltext Beispiele/UART/uart.c)

```
/*Beispiel für Senden eines Strings über UART*/

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

#define BAUD        115200UL
#define UBRR_BAUD   ((F_CPU/(16UL*BAUD))-1)

void uart_init(void) // USART initialisieren
{
      DDRD &= ~(1<<PD0);//RXD als Eingang
      DDRD |= (1<<PD1);//TXD auf Ausgang

    /*Baudrate einstellen ( Normaler Modus )*/
    UBRRH = (unsigned char) (UBRR_BAUD>>8);
    UBRRL = (unsigned char) (UBRR_BAUD);

    UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);// Aktivieren des Empfängers,
des Senders und des "Daten empfangen"-Interrupts
```

```
      UCSRC=(1<<URSEL)|(1<<USBS)|(1<<UCSZ0)|(1<<UCSZ1);// Einstellen des
Datenformats: 8 Datenbits, 1 Stoppbit
}


void uart_putc(unsigned char c)
{
    while (!(UCSRA & (1<<UDRE)))  /* warten bis Senden moeglich */
            ;

    UDR = c;                      /* sende Zeichen */
}

void uart_puts (char *s) //Sende String ueber UART
{
    while (*s)//so lange *s != '\0' also ungleich dem "String-Endezeichen"
    {
        uart_putc(*s);
        s++;
    }
}

int main (void)
{
uart_init();
      while(1)
            {
            uart_puts("Hallo Welt! \r\n");
            _delay_ms(200);
            }
      return 0;
}
```

[Zum Kapitel Hardware: USB/USART]

3.3.2.6 LCD initialisieren und Text ausgeben

Für die Ansteuerung des Displays wurde auf eine fertige C-Bibliothek der Webseite www.mikrocontroller.net zurückgegriffen. Da diese Bibliothek auf eine andere Mikrocontroller-Pinbelegung ausgelegt ist, müssen manche Werte angepasst werden. Die Bibliothek, bestehend aus zwei Dateien (lcdroutines.c und lcd-routines.h), wird über das Makfile und den Präprozessor der main.c integriert.

Bei der lcd-routines.h müssen die Richtungsregister DDRx, die Ausgangsregister PORTx und die Pin-Bits des Mikrocontrollers für die Datenleitungen 4-7 und die RS- und EN- Anschlüsse definiert werden.

Bei der Verwendung des Experimentierboards sollte dies dann so aussehen:

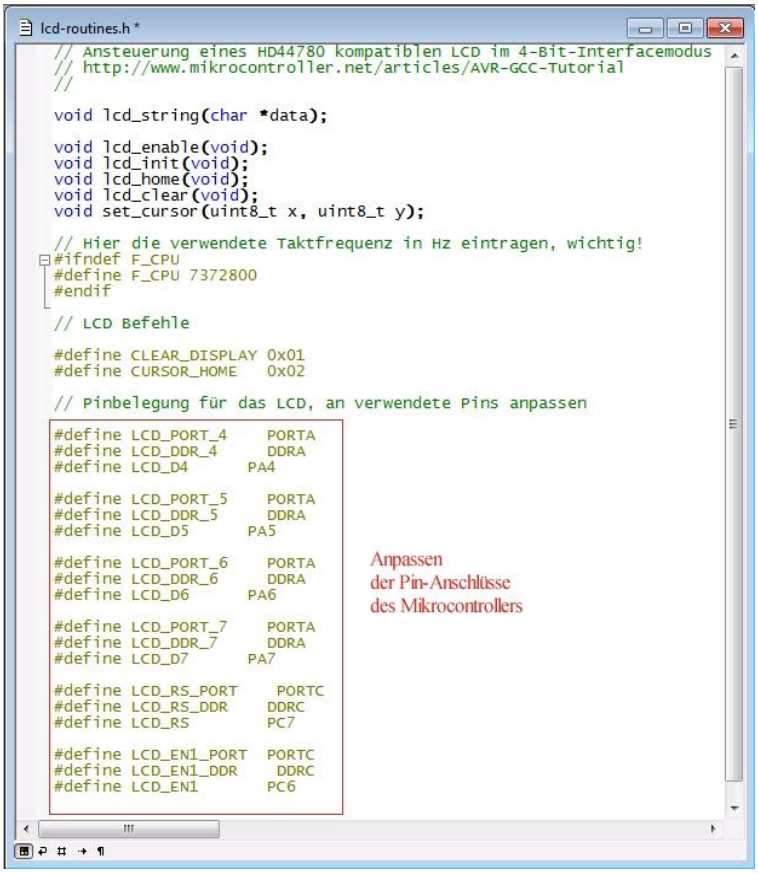


Abbildung 24: lcd-routines.h: Pindefinition für Experimentierboard

Außerdem muss die lcd-routines.c im Makefile eingebunden werden:

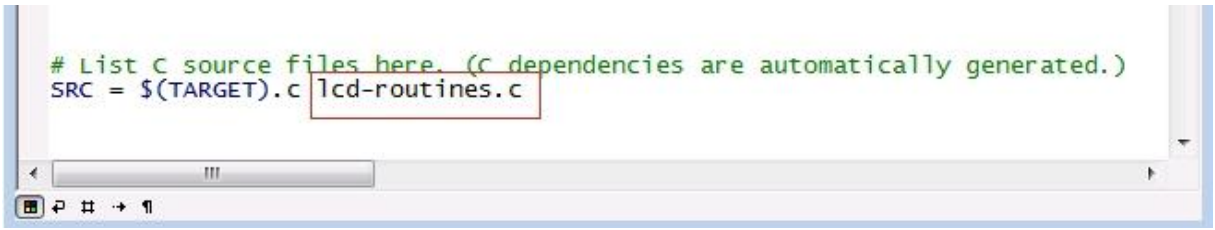


Abbildung 25: Makefile mit eingebundener lcd-routines.c

Damit stehen nun folgende Funktionen zur Verfügung:

Funktion	**Beschreibung**
void lcd_init(void);	• Initialisiert das Display • muss einmalig aufgerufen werden bevor Text ausgegeben werden kann
void lcd_clear(void);	• Löscht die komplette LCD-Ausgabe
void set_cursor(uint8_t x, uint8_t y);	• Setzt das LCD an die Stelle, ab welcher der String geschrieben werden soll • Funktion bekommt uint8_t x und uint8_t y übergeben • uint8_t x: Stelle einer Zeile, beginnt bei 0 • uint8_t y: Zeilennummer, beginnt bei 1
void lcd_string(char *data);	• Schreibt einen String an die mit set_cursor(x,y) ausgewählte Stelle des Displays • z.B. `lcd_string(„Test 123“);`

Tabelle 45: LCD-Funktionen

Schreiben eines Strings auf das LCD (Quelltext Beispiele/LCD/main.c)

```
/* Beispiel für Text-Ausgabe auf LCD*/

#include <avr/io.h>
#include <lcd-routines.h>

int main(void)
{
lcd_init(); //Initialisieren des Displays (lcd-routines.c)
set_cursor(0,1);//Setze LCD auf erste Zeile, erste Spalte (lcd-routines.c)
lcd_string("Hallo Welt!");//Schreibe String (lcd-routines.c)

while(1)
{;}

return 0;
}
```

[Zum Kapitel Hardware: LCD-Ansteuerung]

4. Beispielprojekt Temperaturabhängige Lüfterregelung

Mit dem Experimentierboard können nun beliebige Projekte realisiert werden. Im Folgenden werden Kapitel 2 und 3 dieses Skriptes anhand des Beispiels temperaturabhängige Lüfterregelung verdeutlicht. Eine solche Lüfterregelung ist heutzutage in vielen PCs zu finden und liefert einen guten Kompromiss aus Kühlung und Lautstärke, da sie flexibel auf die Ist-Temperatur reagiert und die Drehzahl des Lüfters entsprechend anpasst.

4.1 Funktionsbeschreibung

Die Lüfteransteuerung hat drei umschaltbare Betriebszustände, eine Steuerung mit manuell einstellbarer Lüfterdrehzahl und zwei temperaturabhängige Regler. Der Temperatursensor befindet sich auf einer externen Lochrasterplatine, die über ein Flachbandkabel mit dem Experimentierboard verbunden ist, und wird in einem zeitlichen Abstand von 0,5 Sekunden abgefragt.

Nach dem Erfassen der Temperatur wird der Reglerwert berechnet und dem Lüfter die entsprechende Drehzahl zugewiesen.

Auf dem LCD erfolgt die Ausgabe der aktuellen Temperatur, des aktiven Betriebsmodus und der prozentualen Lüfterdrehzahl. Vom Benutzer lassen sich die Reglerparameter während des Betriebs über die Taster anpassen. Am PC wird der zeitliche Verlauf der Temperatur und der Lüfterdrehzahl mit dem Tool LogView graphisch ausgegeben.

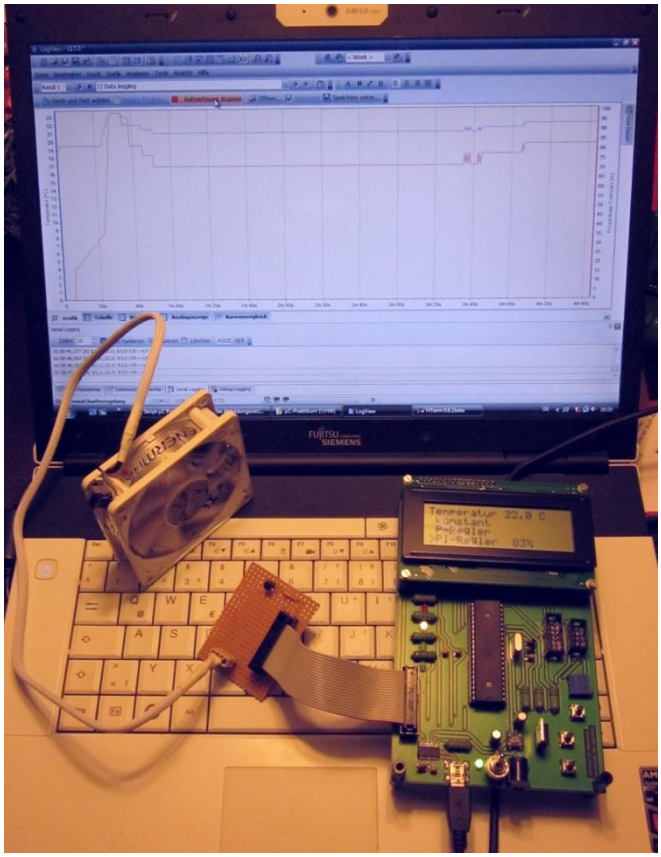


Abbildung 26: Gesamtsystem: Experimentierboard, externer Temperatursensor, Lüfter und graphische Ausgabe

4.1.1 Benutzereingabe und Menü

Für die Benutzereingabe werden die beiden Taster (Taster 0, Taster 1) des Experimentierboards verwendet. Mit Taster 1 lassen sich die 3 Hauptmodi Lüftersteuerung, P-Regler und PI-Regler umschalten, Taster 0 öffnet die Einstellungen des momentan aktiven Modus. Hat der Benutzer mit Taster 1 die Lüfterregelung ausgewählt, kann durch Drücken von Taster 0 die Drehzahl des Lüfters erhöht werden. Im P-/PI-Regler-Modus öffnet Taster 0 das Einstellmenü für die Grenztemperaturen T_{min} und T_{max}. In diesem Menü wird durch Taster 0 die untere Grenztemperatur T_{min} schrittweise um 0,5° C erhöht, mit Taster 1 die obere Grenztemperatur T_{max}. Nach 2 Sekunden springt das Programm aus dem Optionsmenü in den Hauptmodus zurück. Graphisch ist die Struktur in Abbildung 27 dargestellt.

[Zum Quelltext Benutzereingabe]

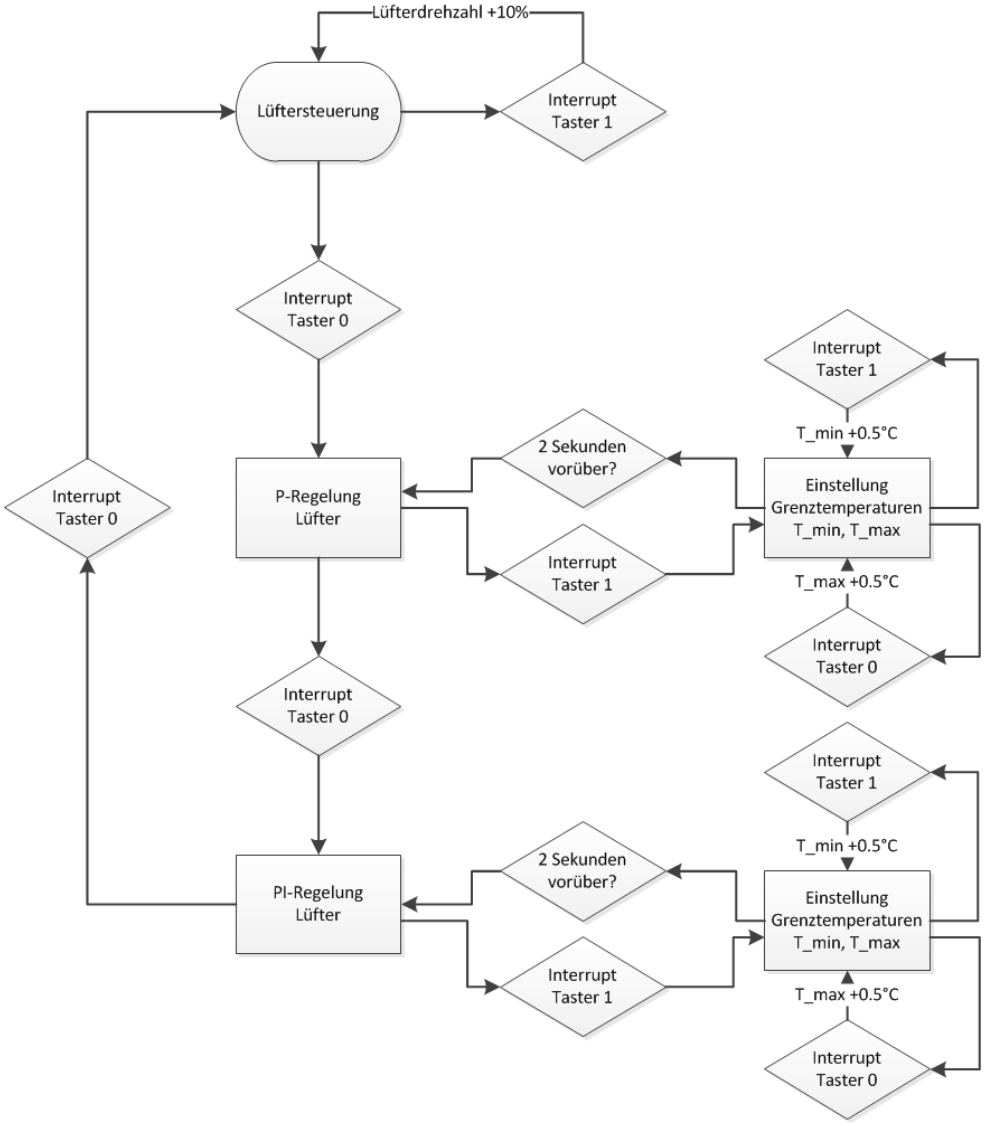


Abbildung 27: Flussdiagramm Benutzereingabe

4.1.2 Temperaturmessung

Der für die Temperaturmessung verwendete Sensor ist ein KTY-81-110, ein preisgünstiger PTC auf Siliziumbasis. In Abbildung 28 lässt sich erkennen, dass sich bei steigender Temperatur der Widerstand exponentiell erhöht und somit, im Gegensatz zu einem PT100, ein nichtlinearer Zusammenhang besteht.

Die Aufgabenstellung beschränkt sich nur auf einen kleinen Temperaturbereich, in dem die Regelung arbeiten soll. Deswegen ist es erlaubt, einen Arbeitsbereich festzulegen, in dem die Umrechnung der ADC-Sensorwerte in einen Temperaturwert mittels eines linearen Zusammenhangs nur eine geringe Abweichung gegenüber der tatsächlichen nichtlinearen Funktion aufweist. Außerhalb dieses Arbeitsbereichs kann die Abweichung vom tatsächlichen Wert groß werden (bis zu 18% bei 150°C).

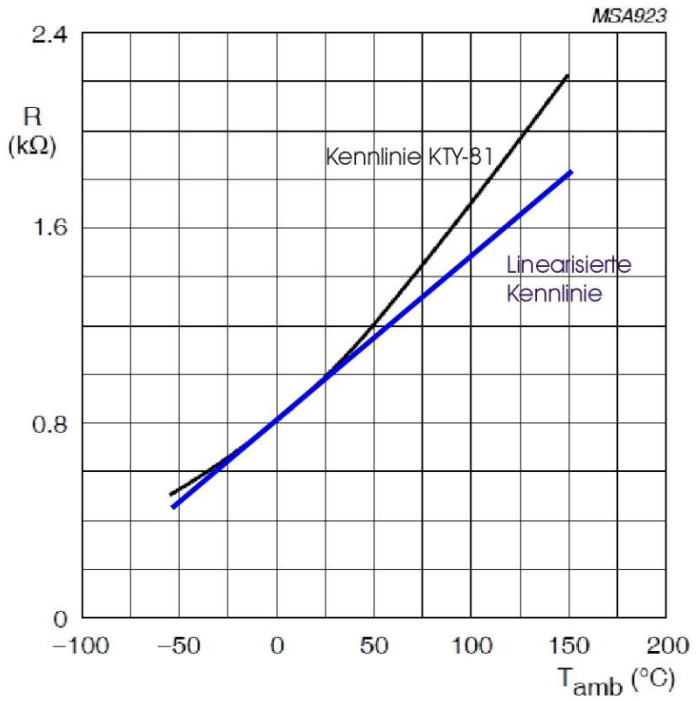


Abbildung 28: Kennlinie $R(T_{amb})$ des KTY-81 und Linearisierung
(Quelle: Datenblatt KTY-81)

Um den temperaturabhängigen Widerstand des Sensors in eine für den ADC messbare Spannung zu wandeln, wird ein *Spannungsteiler* verwendet, der KTY mit einem 1k Widerstand in Reihe geschaltet (Abbildung 29) und der Spannungsabfall über dem PTC gemessen. Es ist darauf zu achten, dass der Gesamtwiderstand des Spannungsteilers nicht zu klein ist, damit die Eigenerwärmung des Thermistors durch den Messstrom (sollte 3mA nicht überschreiten) die Temperaturmessung nicht verfälscht.

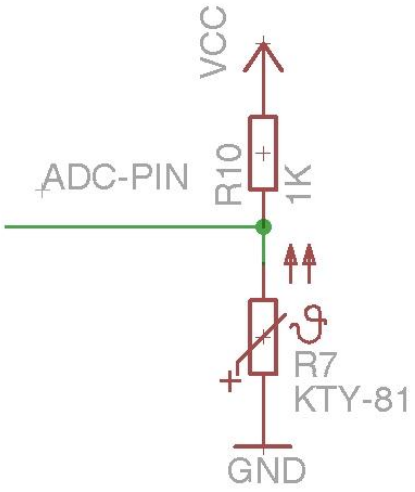


Abbildung 29: Spannungsteilerschaltung

Für die über dem Thermistor abfallende Spannung gilt:

$$U_{ADC} = R(Temp) \cdot I$$

$$I = \frac{U_{Vcc}}{R_{10} + R(Temp)}$$

$$\rightarrow U_{ADC} = U_{Vcc} \frac{R(Temp)}{R_{10}+R(Temp)}$$

Für den daraus gewonnenen idealen digitalisierten ADC-Wert (Nachkommastelle wird verworfen) gilt analog:

$$ADC = ADC_{10-Bit\_max} \frac{R(Temp)}{R_{10} + R(Temp)} = 1023 \cdot \frac{R(Temp)}{R_{10} + R(Temp)}$$

Der Gesamtwiderstand ändert sich im Bereich zwischen 10° und 30°C und damit auch der Messstrom. Diese Änderung ist jedoch so gering, dass sie keinen merklichen Einfluss auf die Erwärmung des Sensors hat:

$$\Delta I = I(10°C) - I(30°C) = U_{VCC} \cdot \left(\frac{1}{R_{10} + R(10°C)} - \frac{1}{R_{10} + R(30°C)}\right)$$
$$= U_{VCC} \cdot \left(\frac{1}{1886\ \Omega} - \frac{1}{2040\ \Omega}\right) = 2{,}65\ mA - 2{,}45\ mA = 200\mu A$$

Um eine softwareseitige Kalibrierung durchzuführen (auf das Anfertigen eine Messprotokolls wird verzichtet), vergleicht man mit einem Referenzthermometer die Temperatur mit dem gelieferten ADC-Wert des µC. Bei 10°C und 30°C wurde der Spannungsabfall über dem PTC gemessen und die digitalisierten Werte 497 und 537 aufgenommen.

Mithilfe der allgemeinen Geradengleichung lässt sich der ADC-Wert auf die Temperatur abbilden:

- allgemein: Temperatur = m · ADC + n
- 10°C = m · 497 + n
- 30°C = m · 537 + n

Man erhält daraus die Geradengleichung:

$$Temperatur = \frac{ADC - 477}{2}\,°C = \frac{\left(1023 \cdot \frac{R(Temp)}{R_{10}+R(Temp)}\right) - 477}{2}\,°C$$

[Zum Quelltext Temperaturmessung]

4.1.3 Regler

Der Enermax-Lüfter UCL8 mit PWM-Eingang (und Duty-Cycle abhängiger Drehzahl) hat die Aufgabe, den Temperaturwert am PTC-Sensor auf einen Sollwert abzukühlen.

Für die Lüfterregelung kommt ein P-Regler zum Einsatz, welcher mathematisch folgendermaßen beschrieben werden kann:

$y = k_p \cdot e = k_p \cdot (T - T_{soll})$

y bezeichnet dabei die Stellgröße, k_p die Proportional-Konstante des Reglers, e den Temperaturfehler $T - T_{soll}$, T_{soll} den Temperatursollwert und T den Istwert.

Der Sollwert der Temperatur, also der Grenzwert bei welcher der Lüfter zu drehen beginnt wird im Folgenden mit T_{min} bezeichnet. Die zweite Grenztemperatur T_{max} beschreibt die Temperatur bei der der Lüfter mit maximaler Drehzahl arbeitet. Im Bereich zwischen T_{min} und T_{max} steigt die Stellgröße Lüfterdrehzahl (bzw. Duty-Cycle der PWM) linear an (Abbildung 30). Beide Temperaturen lassen sich über die Taster vom Benutzer je nach Anwendungszweck einstellen, aus ihnen werden der Temperatursollwert und die Proportional-Konstante des Reglers berechnet.

$$PWM(T) = \begin{cases} 0\% & \text{falls } T < T_{min}, \\ 100\% \cdot \left(\frac{T}{T_{max}-T_{min}} - \frac{T_{min}}{T_{max}-T_{min}}\right), & \text{falls } T_{min} < T < T_{max}, \\ 100\% & \text{falls } T > T_{max} \end{cases}$$

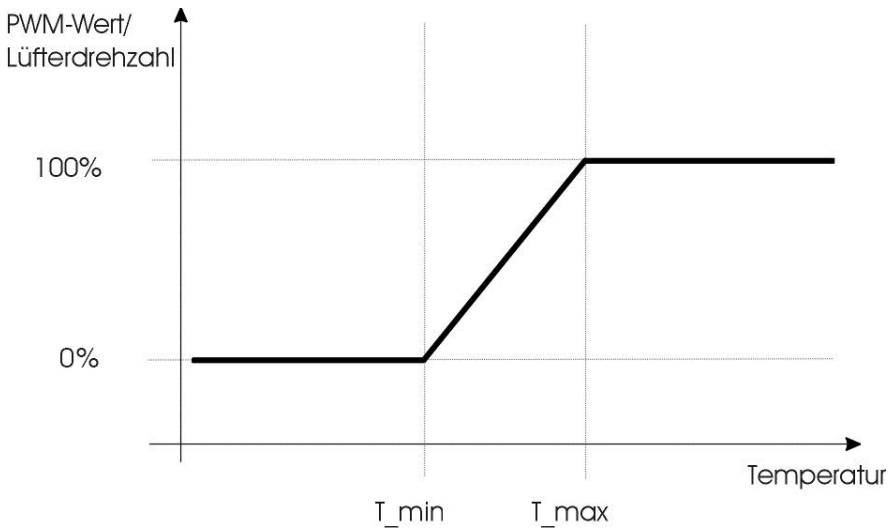


Abbildung 30: Kennlinie P-Regler

Im dritten Regelmodus (Abbildung 31) sorgt ein zusätzlicher Integralteil, in welchem die vergangenen Fehlerwerte T - T_{soll} aufsummiert werden, für ein Beseitigen von dauerhaften (Offset-) Fehlern. Dabei ist zu beachten, dass der I-Teil begrenzt werden muss, damit er nicht unkontrolliert anwächst (Anti-Windup).

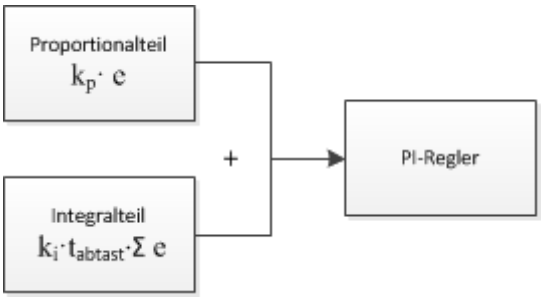


Abbildung 31: PI-Regler Zusammensetzung

[Zum Quelltext Regler]

Link: RN-Wissen – Digitaler Regler

4.1.4 Graphische Ausgabe am PC mit LogView

Nachdem die Kommunikation zwischen PC und dem ATmega hergestellt ist und mit einem Terminal-Programm Daten empfangen und versendet werden können, bietet sich zur besseren Übersichtlichkeit und Visualisierung der Sensordaten eine graphische Ausgabe an.

Dafür wurde auf das Open-Source Programm LogView zurückgegriffen, welches die am virtuellen COM-Port ankommenden Daten loggt und daraus einen oder mehrere Graphen erstellt. Damit LogView etwas mit den empfangenen Werten vom Mikrocontroller anfangen kann, müssen sie in einem festgelegten Protokoll als String versendet werden:

```
„$Kanalnummer;Zustandsnummer;Time-Stamp;Wert Typ 1;Wert Typ 2;...;Wert Typ n;Checksumme Endzeichen“
```

In dem Beispielprojekt wird nur ein Kanal und ein Zustand verwendet, beide Werte sind somit fest 1. Auf einen Time-Stamp wurde verzichtet, LogView generiert dann automatisch eine Zeitskala. Unterschieden werden zwei Arten von Werten, Temperatur- und Reglerdaten. Auf einen Checksummen-Test wurde ebenfalls verzichtet, da die Übertragung unkritisch ist (Checksumme fest zu 0), das Endzeichen ist „/r/n“.

Somit ergibt sich:

```
„$1;1;;Temperatur;Drehzahl;0 /r/n“
```

In einer *.ini-Datei im /Geraete/OpenFormat Ordner von Logview müssen die Einstellungen der Kommunikation und des Protokolls mittels eines Texteditors gespeichert werden:

Luefterregelung.ini

```
[Gerät]
Name                            = Luefterregelung
Hersteller                      = hw
Gruppe                          = 07
Device_ID                       = ID_OPENFORMAT
Used                            = 1
Abbildung                       = OpenFormat.jpg
ChangeSettings                  = 1
HerstellerLink1                 =
HerstellerLink2                 =
LogViewLink                     = www.logview.info
TimeStep_ms                     = 500
TimeGiven                       = 0
```

```
KanalAnzahl                     = 8
WerteFormat                     = ASCII
Prüfsummenberechnung            = keine
PrüfsummenFormat                = ASCII
AutoStart                       = 0
AutoOpenToolbox                 = 0
DateTimeFormat                  = R_%hh"h" %nn"m" %ss"s"

[Stati]
StatiAnzahl                     = 1
001                             = Data logging

[serielle Schnittstelle]
Port                            = COM10
Baudrate                        = 460800
Datenbits                       = 8
Stopbits                        = 1
Parität                         = 0
Flusskontrolle                  = 0
ClusterSize                     = -50
SetDTR                          = 0
SetRTS                          = 0

[Schnittstelle TimeOuts]
RTOCharDelayTime                = 290
RTOExtraDelayTime               = 100
WTOCharDelayTime                = 290
WTOExtraDelayTime               = 100

[Anzeige Einstellungen Kanal 01]
Zeitbasis                       = Zeit
Einheit                         = s
Symbol                          = t
WerteAnzahl                     = 3

Messgröße1                      = Temperatur
Einheit1                        = °C
Symbol1                         = T
Faktor1                         = 1.0
OffsetWert1                     = 0.0
OffsetSumme1                    = 0.0

Messgröße2                      = Prozentuale Drehzahl
Einheit2                        = %
Symbol2                         = n
Faktor2                         = 1.0
OffsetWert2                     = 0.0
OffsetSumme2                    = 0.0
```

Zum Aufzeichnen der Daten muss nach dem Starten von LogView unter *Gerät→ Geräte und Port wählen* die erstellte *.ini-Datei und der COM-Port ausgewählt werden. Danach kann mit dem Button *Aufzeichnung starten* die graphische Ausgabe beginnen (Abbildung 32).

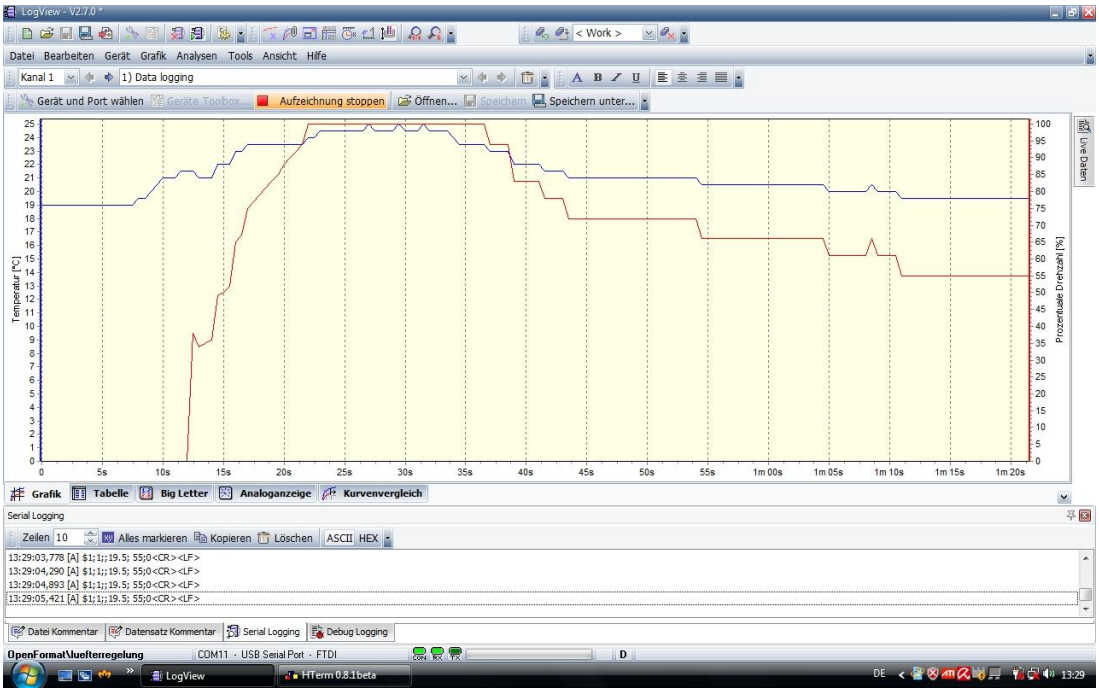


Abbildung 32: Ausgabe der Daten in LogView

[Zum Quelltext USB/USART Ausgabe]

4.2 Quellcode

4.2.1 main.c

In der main.c werden die Header-Dateien der ausgelagerten c-Files eingebunden (adc.h, pwm.h, uart.h, cd-routines.h) und alle globalen Variablen definiert, auf die von den externen Dateien adc.c, pwm.c, uart.c, lcd-routines.c, regler.c und taster.c zugegriffen wird.

adc.c liefert die Funktionen `ADC_Init()` und `get_temperature()`, lcd-routines.c die Funktionen `lcd_init()`, `lcd_write_temperature(temp)`, `lcd_write_menue()` und `lcd_clear()`. Die Regelungsfunktion `regler(temp,tempmax,tempmin,state)` ist in der regler.c zu finden und die Ausgabe der Daten an LogView `usb_logview()` in der uart.c.

main.c

```
#include <avr/io.h>
#include <stdlib.h>
#include <util/delay.h>
#include <inttypes.h>
#include <avr/interrupt.h>
#include "adc.h"
#include "pwm.h"
#include "uart.h"
#include "lcd-routines.h"



/*Definition globaler Variablen*/
volatile float tempmax=27; //obere Grenztemp
volatile float tempmin=18;    //untere Grenztemp
volatile uint8_t state=1;     // Betriebsmodus ( Zustand 1,2 oder 3)
volatile uint8_t zeile;       //Hilfsvariable für LCD
volatile uint8_t konstpwm=0; //konstante Lüfterdrehzahl
volatile float ersum=0;      //Integralteil des Reglers
```

```
volatile uint8_t edit_flag=0; // Optionsmenue-Flag
char Buffer2[30];
float tempval;
float temp;                   //aktuelle Ist-Temperatur
uint8_t pwm;
char Buffer[30];
char Buffer1[30];
float error=0;                //Fehler T-Tmin


int main (void)
{

DDRB=(1<<PB0)|(1<<PB1)|(1<<PB2)|(1<<PB3); //Pins als LED-Ausgang setzen

      lcd_init();
      ADC_Init();
      PWM_init();
      uart_init();
      taster_init();

      sei(); //aktiviere Interrupts


      while(1)

            {

            temp=get_temperature();//weise temp die Temperatur zu

            lcd_write_temperature(temp); //Gebe Temperatur auf LCD aus

            regler(temp,tempmax,tempmin,state); //Fuehre Regler aus

            usb_logview();    //Sende alle wichtigen Daten an Logview

            lcd_write_menue(); //Gebe Menue auf LCD aus

            _delay_ms(500);
            lcd_clear();

            }

   return 0; // nie erreicht
}
```

4.2.2 ADC

Die Funktion `get_temperature()` führt eine ADC-Wandlung durch und berechnet aus dem 16-Bit unsigned Integer ADC-Wert den Temperaturwert. Dieser wird als float von der Funktion zurückgegeben.

adc.h

```
extern float tempval;
extern float temp;

uint16_t ADC_Read(void);
void ADC_Init(void);
float get_temperature (void);
```

adc.c

```
#include <avr/io.h>
#include "adc.h"
#include <inttypes.h>
#include <stdlib.h>
#include <util/delay.h>

void ADC_Init(void) //ADC initialisieren
{
  uint16_t result;

  ADMUX = (0<<REFS1) | (1<<REFS0)|(1<<MUX0)|(1<<MUX1);       // AVCC
Referenzspannung(5V), ADC3 aktiv
ADCSRA = (1<<ADPS1) | (1<<ADPS0);     // Frequenzvorteiler 8 -> ADC
Frequenz ca. 922kHz
  ADCSRA |= (1<<ADEN);                  // ADC aktivieren

  /* nach Aktivieren des ADC wird ein "Dummy-Readout" empfohlen, man liest
    also einen Wert und verwirft diesen, um den ADC "warmlaufen zu lassen"
*/
  ADCSRA |= (1<<ADSC);                  // eine ADC-Wandlung
  while (ADCSRA & (1<<ADSC) );          // auf Abschluss der Konvertierung
warten
   result = ADCW;
}

uint16_t ADC_Read(void) //ADC Einzelmessung
{
  // Kanal waehlen, ohne andere Bits zu beeinflußen
  ADMUX |=(1<<MUX0); //ADC1 aktiv
  _delay_ms(10);
  ADCSRA |= (1<<ADSC);            // eine Wandlung "single conversion"
  while (ADCSRA & (1<<ADSC) )     // auf Abschluss der Konvertierung
warten
    ;

  return ADCW;                    // ADC auslesen und zurückgeben
}

float get_temperature (void)
{
float adc_value;
float temperature;
adc_value=ADC_Read(); //fuehre ADC-Wandlung durch
temperature=(adc_value-477)/2; //rechne ADC-Wert in Temperatur um
return temperature;
}
```

4.2.3 LCD

`lcd_write_temperature()` bekommt den Temperaturwert als float übergeben, wandelt ihn in einen String um und schreibt diesen in die erste Zeile des Displays, beispielsweise „Temperatur 18.5 C“. `lcd_write_menue()` schreibt das Auswahlmenue in die zweite bis vierte Zeile. Dabei wird entweder der Betriebsmodus mit zugehörigem PWM-DutyCycle (0-100%) ausgegeben, oder, wenn das Optionsmenue-Flag edit_flag durch Taster 1 auf TRUE gesetzt wurde, die Grenztemperaturen des aktiven Modus in die ensprechende Zeile geschrieben.

lcd-routines.h

```
// Ansteuerung eines HD44780 kompatiblen LCD im 4-Bit-Interfacemodus
// http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial

extern volatile float tempmax; //obere Grenztemp
```

```
extern volatile float tempmin;      //untere Grenztemp
extern volatile uint8_t state;      // Zustand
extern volatile   uint8_t zeile;
extern volatile uint8_t konstpwm; //konstante Lüfterdrehzahl
extern volatile float ersum;  //Integralteil des Reglers
extern volatile uint8_t edit_flag; // Optionsmenue-Flag
extern char Buffer2[30];
extern float temp;
extern char Buffer1[30];

void lcd_string(char *data);

void lcd_enable(void);
void lcd_init(void);
void lcd_home(void);
void lcd_clear(void);
void set_cursor(uint8_t x, uint8_t y);
void lcd_write_temperature(float x);
void lcd_write_options(void);
void lcd_write_menue(void);

// Hier die verwendete Taktfrequenz in Hz eintragen, wichtig!
#define F_CPU 7372800

// LCD Befehle

#define CLEAR_DISPLAY 0x01
#define CURSOR_HOME   0x02

// Pinbelegung für das LCD, an verwendete Pins anpassen

#define LCD_PORT_4    PORTA
#define LCD_DDR_4     DDRA
#define LCD_D4      PA4

#define LCD_PORT_5    PORTA
#define LCD_DDR_5     DDRA
#define LCD_D5      PA5

#define LCD_PORT_6    PORTA
#define LCD_DDR_6     DDRA
#define LCD_D6      PA6

#define LCD_PORT_7    PORTA
#define LCD_DDR_7     DDRA
#define LCD_D7      PA7

#define LCD_RS_PORT    PORTC
#define LCD_RS_DDR    DDRC
#define LCD_RS        PC7

#define LCD_EN1_PORT  PORTC
#define LCD_EN1_DDR    DDRC
#define LCD_EN1       PC6
```

lcd-routines.c

```
// Ansteuerung eines HD44780 kompatiblen LCD im 4-Bit-Interfacemodus
// http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial
// Die Pinbelegung ist über defines in lcd-routines.h einstellbar

#include <avr/io.h>
```

```
#include "lcd-routines.h"
#include <util/delay.h>
#include <stdlib.h>

// sendet ein Datenbyte an das LCD

void lcd_data(unsigned char temp1)
{
   unsigned char temp2 = temp1;

   LCD_RS_PORT |= (1<<LCD_RS);         // RS auf 1 setzen

   temp1 = temp1 >> 4;
   temp1 = temp1 & 0x0F;

   LCD_PORT_4 &= ~(1<<LCD_D4);
   LCD_PORT_5 &= ~(1<<LCD_D5);
   LCD_PORT_6 &= ~(1<<LCD_D6);
   LCD_PORT_7 &= ~(1<<LCD_D7);

   if(temp1 & 0x01) LCD_PORT_4 |=(1<<LCD_D4) ;
   if(temp1 & 0x02) LCD_PORT_5 |=(1<<LCD_D5) ;
   if(temp1 & 0x04) LCD_PORT_6 |=(1<<LCD_D6) ;
   if(temp1 & 0x08) LCD_PORT_7 |=(1<<LCD_D7) ;
   lcd_enable();



   temp2 = temp2 & 0x0F;

   LCD_PORT_4 &= ~(1<<LCD_D4);
   LCD_PORT_5 &= ~(1<<LCD_D5);
   LCD_PORT_6 &= ~(1<<LCD_D6);
   LCD_PORT_7 &= ~(1<<LCD_D7);

   if(temp2 & 0x01) LCD_PORT_4 |=(1<<LCD_D4) ;
   if(temp2 & 0x02) LCD_PORT_5 |=(1<<LCD_D5) ;
   if(temp2 & 0x04) LCD_PORT_6 |=(1<<LCD_D6) ;
   if(temp2 & 0x08) LCD_PORT_7 |=(1<<LCD_D7) ;
lcd_enable();


   _delay_us(42);
}

// sendet einen Befehl an das LCD

void lcd_command(unsigned char temp1)
{
   unsigned char temp2 = temp1;

   LCD_RS_PORT &= ~(1<<LCD_RS);         // RS auf 0 setzen

   temp1 = temp1 >> 4;               // oberes Nibble holen
   temp1 = temp1 & 0x0F;

   LCD_PORT_4 &= ~(1<<LCD_D4);
   LCD_PORT_5 &= ~(1<<LCD_D5);
   LCD_PORT_6 &= ~(1<<LCD_D6);
   LCD_PORT_7 &= ~(1<<LCD_D7);             // maskieren

   if(temp1 & 0x01) LCD_PORT_4 |=(1<<LCD_D4) ;// setzen
```

```
   if(temp1 & 0x02) LCD_PORT_5 |=(1<<LCD_D5) ;
   if(temp1 & 0x04) LCD_PORT_6 |=(1<<LCD_D6) ;
   if(temp1 & 0x08) LCD_PORT_7 |=(1<<LCD_D7) ;
   lcd_enable();

   temp2 = temp2 & 0x0F;

LCD_PORT_4 &= ~(1<<LCD_D4);
   LCD_PORT_5 &= ~(1<<LCD_D5);
   LCD_PORT_6 &= ~(1<<LCD_D6);
   LCD_PORT_7 &= ~(1<<LCD_D7);        // unteres Nibble holen und maskieren

if(temp2 & 0x01) LCD_PORT_4 |=(1<<LCD_D4) ;// setzen
   if(temp2 & 0x02) LCD_PORT_5 |=(1<<LCD_D5) ;
   if(temp2 & 0x04) LCD_PORT_6 |=(1<<LCD_D6) ;
   if(temp2 & 0x08) LCD_PORT_7 |=(1<<LCD_D7) ;
   lcd_enable();

   _delay_us(42);
}

// erzeugt den Enable-Puls

void lcd_enable(void)
{
   // Bei Problemen ggf. Pause gemäß Datenblatt des LCD Controllers
einfügen

   // http://www.mikrocontroller.net/topic/81974#685882
   LCD_EN1_PORT |= (1<<LCD_EN1);


    _delay_us(1);                    // kurze Pause
   // Bei Problemen ggf. Pause gemäß Datenblatt des LCD Controllers
verlängern
   // http://www.mikrocontroller.net/topic/80900
   LCD_EN1_PORT &= ~(1<<LCD_EN1);


}
// Initialisierung:
// Muss ganz am Anfang des Programms aufgerufen werden.

void lcd_init(void)
{
   // Ports auf Ausgang schalten
   LCD_DDR_4 |=(1<<LCD_D4);
   LCD_DDR_5 |=(1<<LCD_D5);
   LCD_DDR_6 |=(1<<LCD_D6);
   LCD_DDR_7 |=(1<<LCD_D7);

   LCD_EN1_DDR |= (1<<LCD_EN1);

   LCD_RS_DDR  |= (1<<LCD_RS);
   // muss 3mal hintereinander gesendet werden zur Initialisierung

_delay_ms(15);

   LCD_PORT_4 |=  (1<<LCD_D4);
   LCD_PORT_5 |=  (1<<LCD_D5);
   LCD_PORT_6 &= ~(1<<LCD_D6);
   LCD_PORT_7 &= ~(1<<LCD_D7);
```

```
   LCD_RS_PORT &= ~(1<<LCD_RS);      // RS auf 0
   lcd_enable();

   _delay_ms(5);
   lcd_enable();

   _delay_ms(1);
   lcd_enable();
   _delay_ms(1);

// 4 Bit Modus aktivieren


   LCD_PORT_4 &=  ~(1<<LCD_D4);
LCD_PORT_5 |=   (1<<LCD_D5);
   LCD_PORT_6 &=  ~(1<<LCD_D6);
   LCD_PORT_7 &=  ~(1<<LCD_D7);

   lcd_enable();
_delay_ms(1);

   // 4Bit / 2 Zeilen / 5x7
   lcd_command(0x28);

   // Display ein / Cursor aus / kein Blinken
   lcd_command(0x0C);

   // inkrement / kein Scrollen
lcd_command(0x06);

   lcd_clear();
}

// Sendet den Befehl zur Löschung des Displays

void lcd_clear(void)
{
   lcd_command(CLEAR_DISPLAY);
_delay_ms(5);
}

// Sendet den Befehl: Cursor Home

void lcd_home(void)
{
   lcd_command(CURSOR_HOME);
_delay_ms(5);
}

// setzt den Cursor in Zeile y (1..4) Spalte x (0..15)

void set_cursor(uint8_t x, uint8_t y)
{
  uint8_t tmp;

  switch (y) {
case 1: tmp=0x80+0x00+x;  break;    // 1. Zeile
case 2: tmp=0x80+0x40+x;  break;    // 2. Zeile
case 3: tmp=0x80+0x14+x;  break;    // 3. Zeile
case 4: tmp=0x80+0x54+x;  break;    // 4. Zeile
  }
```

```
  lcd_command(tmp);
}

// Schreibt einen String auf das LCD

void lcd_string(char *data)
{
    while(*data) {
        lcd_data(*data);
        data++;
    }
}

void lcd_write_temperature(float x)
{
      char Buf[6];
      set_cursor( 0, 1 );//setze LCD an den Anfang
      lcd_string("Temperatur"); //schreibe String "Temperatur"
      dtostrf(x, 3, 1, Buf); //wandle Temp.-wert in String
      set_cursor( 11, 1 ); //setze LCD an richtige Stelle
      lcd_string(Buf); //schreibe Temperatur auf LCD
      set_cursor( 15, 1 );
      lcd_string(" C");

}

void lcd_write_options(void)
{
set_cursor(0,zeile);
                  lcd_string("Tmn ");
                  dtostrf(tempmin, 3, 1, Buffer2);
                  set_cursor(4,zeile);
            lcd_string(Buffer2);
            set_cursor(8,zeile);
            lcd_string("C ");
            set_cursor(10,zeile);
                  lcd_string("Tmx ");
                  dtostrf(tempmax, 3, 1, Buffer2);
                  set_cursor(14,zeile);
            lcd_string(Buffer2);
            set_cursor(18,zeile);
            lcd_string("C");

}


void lcd_write_menue(void)
{
/* Ausgabe verschiedener Werte auf das LCD*/
            set_cursor(1,2);
            lcd_string("konstant");
            set_cursor(1,3);
            lcd_string("P-Regler");
            set_cursor(1,4);
            lcd_string("PI-Regler");

            zeile=state+1;
            set_cursor(0,zeile);
            lcd_string(">"); //markiere aktiven Modus
            set_cursor(11,zeile);
            lcd_string(Buffer2); //Schreibe PWM-Wert des aktiven Modus
            set_cursor(14,zeile);
```

```
            lcd_string("%");


            if (edit_flag==1) //falls im Regler-Options-Menue -> Ausgabe
der Grenztemperaturen in der aktuellen Zeile
            {

            int i;

            for(i=0;i<50;i++)
            {
            lcd_write_options();
            _delay_ms(100);// warte 5 s
            }
            edit_flag=0; // springe aus Optionsmenue heraus
            PORTB|=(1<<PB3);
            }

}
```

4.2.4 Regler

Die Reglerfunktion regler() bekommt die aktuelle Ist-Temperatur temp, die beiden Grenztemperaturen tempmax, tempmin und den aktuellen Betriebsmodus state übergeben. Nur im PI-Modus (state == 3) wird der Integralteil berücksichtigt, ansonsten ist er zu Null gesetzt. Befindet sich die Ist-Temperatur außerhalb des Intervalls T_{min} und T_{max} wird der Stellwert auf Maximum bzw Minimum gesetzt, dazwischen ist der Regler aktiv. Im Modus state==1 (Lüftersteuerung) wird das PWM-Register OCR0 mit dem Wert konst_pwm beschrieben, das der Benutzer durch Taster 1 anpassen kann.

regler.h

```
extern volatile float tempmax; //obere Grenztemp
extern volatile float tempmin;      //untere Grenztemp
extern volatile uint8_t state;      // Zustand
extern volatile   uint8_t zeile;
extern volatile uint8_t konstpwm; //konstante Lüfterdrehzahl
extern volatile float ersum;  //Integralteil des Reglers
extern volatile uint8_t edit_flag; // Optionsmenue-Flag
extern char Buffer2[30];
extern float temp;
extern uint8_t pwm;
extern float error;

void regler(float temp, float tempmax, float tempmin, uint8_t state);
```

regler.c

```
#include <avr/io.h>
#include <stdlib.h>
#include <util/delay.h>
#include <inttypes.h>
#include "regler.h"


void regler(float temp, float tempmax, float tempmin, uint8_t state)

{

error=-tempmin+temp;//aktuelle Sollabweichung

if (state==3) // falls PI-Regler
            {
```

```
                  ersum+=error;// Integralteil: Sollabweichung aufaddieren

                        if (ersum>100) //Begrenzung des I-Teils
                              {ersum=100;}
            }

            pwm=(255/(tempmax-tempmin))*temp-(255/(tempmax-
tempmin))*tempmin+ersum; // P/PI-Regler

            if ((255/(tempmax-tempmin))*temp-(255/(tempmax-
tempmin))*tempmin+ersum>255) //falls PWM Wert ueber 100%
                  {pwm=255;}  // PWM Wert Begrenzung


            if (temp>tempmin && temp<tempmax) //falls T_min<T<T_max
            {
                  PORTB=(1<<PB0); //gelbe LED anschalten
                  OCR1A = pwm; //weise PWM-Register Reglerwert zu
                  dtostrf(pwm*100/255, 3, 0, Buffer2); //Schreibe PWM-
Tastgrad in String Buffer2
            }

            if (temp<=tempmin) //falls T<T_min
            {
                  ersum=0; //setze I-Teil zurueck
                  PORTB=(1<<PB1); //gruene LED anschalten
                  pwm=0; //PWM-Tastgrad auf 0% setzen
                  OCR1A =pwm; //PWM-Register zuweisen
                  dtostrf(pwm, 3, 0, Buffer2); //Schreibe PWM-Tastgrad in
String Buffer2
            }

            if (temp>=tempmax)//falls T>T_max
            {
                  PORTB=(1<<PB2);//rote LED anschalten
                  pwm=255; //PWM-Tastgrad auf Maximun
                  OCR1A =pwm;//PWM-Register zuweisen
                  dtostrf(pwm*100/255, 3, 0, Buffer2);//Schreibe PWM-
Tastgrad in String Buffer2
            }

            if (state==1)// falls Luefterregelung aktiv
            {
                  OCR1A=konstpwm; //weise PWM-Register manuellen Wert zu
                  dtostrf(konstpwm*100/255, 3, 0, Buffer2);//Schreibe PWM-
Tastgrad in String Buffer2
            }

}
```

4.2.5 PWM

In der pwm.c wird PWM_init() definiert:

pwm.h

```
void PWM_init(void);
```

pwm.c

```
#include <avr/io.h>
#include "pwm.h"
```

```
void PWM_init(void) //Initialisiere PWM Ausgang
{
TCCR1A = (1<<WGM10)|(1<<WGM12)|(1<<COM1A1);//8bit, PWM Mode 3: Fast-PWM
TCCR1B = (1<<CS11)|(1<<CS10);//PWM-Frequenz=f_CPU/64=115,2kHz
DDRD|=(1<<PD5); //setze PWM-Pin auf Ausgang
}
```

4.2.6 Taster

Die taster.c besteht aus der Interrupt-Service-Routine für die Eingänge INT0 und INT1, an denen Taster 0 und 1 angeschlossen sind. Taster 0 schaltet die Betriebszustände (state-Variable) um, Taster 1 öffnet das Optionsmenü (setzt edit_flag auf TRUE). Darin wiederum lässt sich durch Tastendruck eine Anpassung der Grenztemperaturen vornehmen (siehe Abbildung 27).

taster.h

```
void taster_init(void);
ISR(INT0_vect);
ISR(INT1_vect);

extern volatile uint8_t state;
extern volatile uint8_t zeile;
extern volatile uint8_t edit_flag;
extern volatile float tempmax;
extern volatile float tempmin;
extern volatile uint8_t konstpwm;
extern char Buffer2;
extern volatile float ersum;
```

taster.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <inttypes.h>
#include "taster.h"

void taster_init(void)
{
      MCUCR=(1<<ISC11)|(0<<ISC10)|(1<<ISC01)|(0<<ISC00); // INT1,0
Interrupt bei fallender Flanke
      GICR=(1<<INT1)|(1<<INT0); //aktiviere INT1,0
}

ISR(INT1_vect) //Interrupt Taster 1
{
if (edit_flag<1) //falls nicht im Optionsmenue
{state++;         //naechster Regelmodus

      if (state >3 )
            {state=1;}

ersum=0; // setze Integralteil des PI-Reglers zurueck
}

if (edit_flag==1) //falls im Optionsmenue
      {
                  tempmax+=.5; //erhoehe obere Grenztemp.
                  zeile=state+1;
      }

if (tempmax>30)
```

```
      {tempmax=23.5;}

}

ISR(INT0_vect) //Interrupt Taster 0
{

      if (state==1) //falls Luefterregelung
      {konstpwm+=26;} // erhoehe Luefterdrehzahl

      if (state>1) //falls P/PI Regler
      {
            tempmin+=.5; //erhoehe T_min
            edit_flag=1; //aktiviere Optionsmenue

            zeile=state+1;
      }
      if (tempmin>23)
      {tempmin=18;}
}
```

4.2.7 USART

Die Funktion `usb_logview()` schreibt einen LogView-konformen String „`$1;1;;Temperatur;Drehzahl;0 /r/n`" per `sprintf()` in die Variable Buffer und sendet ihn per USB an den PC.

uart.h

```
#define BAUD        115200UL

#define UBRR_BAUD   ((F_CPU/(16UL*BAUD))-1)

extern char Buffer[30];
extern char Buffer1[30];
extern char Buffer2[30];

void uart_init(void);

void uart_putc(unsigned char c);

uint8_t uart_receive(void);

void uart_puts (char *s);

void uart_flush( void );

void usb_logview(void);
```

uart.c

```
#include <avr/io.h>
#include <stdlib.h>
#include <util/delay.h>
#include <inttypes.h>
#include "uart.h"

void uart_init(void) // USART initialisieren
{
      DDRD &= ~(1<<PD0);//RDD als EIngang
      DDRD |= (1<<PD1);//TXD auf Ausgang
```

```
    /*Baudrate einstellen ( Normaler Modus )*/
UBRRH = (unsigned char) (UBRR_BAUD>>8);
UBRRL = (unsigned char) (UBRR_BAUD);

UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);// Aktivieren des Empfängers, des
Senders und des "Daten empfangen"-Interrupts

      UCSRC=(1<<URSEL)|(1<<USBS)|(1<<UCSZ0)|(1<<UCSZ1);// Einstellen des
Datenformats: 8 Datenbits, 1 Stoppbit
}

void uart_putc(unsigned char c)
{
    while (!(UCSRA & (1<<UDRE)))  /* warten bis Senden moeglich */
            ;

    UDR = c;                      /* sende Zeichen */
}

uint8_t uart_receive(void) //Empfange String ueber UART
{while (! (UCSRA & (1<<RXC)));
return UDR;
}


void uart_puts (char *s) //Sende String ueber UART
{
    while (*s)//so lange *s != '\0' also ungleich dem "String-Endezeichen"
{
        uart_putc(*s);
        s++;
    }
}

void uart_flush( void )
{
unsigned char dummy;
while ( UCSRA & (1<<RXC) ) dummy = UDR;
}

void usb_logview(void)
{
sprintf(Buffer,"$1;1;;%s;%s;0\r\n",Buffer1, Buffer2);//Schreibe
Temp./Reglerwert in String
uart_puts(Buffer);      //USB-Ausgabe Temp./Reglerwert im LogView Protokoll
}
```

5. Anhang

5.1 Software-Bezugsquellen

Software	Beschreibung	Link
WinAVR	Programmierumgebung für Atmel AVRs (enthält Programmers Notebook, Compiler, usw.)	http://winavr.sourceforge.net/
AVR Studio 4	Entwicklungsumgebung für Atmel AVRs (Fusebits setzen, Programme in µC laden, usw.)	http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
LogView	Tool um serielle Daten (RS232, USB) von Geräten aufzuzeichnen und grafisch auszuwerten	http://www.logview.info/
FTDI-Treiber	Treiber für USB-Interface FT232 unter Windows / Linux	http://www.ftdichip.com/FTDrivers.htm

5.2 Stückliste Experimentierboard

Anbieter	Best.-Nr.	Beschreibung	Einzelpreis	Anz.
Reichelt	HEBW 21	Hohlstecker-Einbaubuchse	0,12 €	1
Reichelt	µA 7805	Spannungsregler	0,29 €	1
Reichelt	B40C800DIP	DIL-Gleichrichter	0,18 €	1
Reichelt	7,3728-HC18	Standardquarz, Grundton, 7,3728 MHz	0,18 €	1
Reichelt	USB BWM SMD	USB-Einbaubuchse, M-Mini gew. , SMD	0,24 €	1
Reichelt	ATMEGA 16-16 DIP	ATMega AVR-RISC-Controller, DIL-40	3,65 €	1
Reichelt	FT 232 RL	USB UART IC, SSOP 28	3,15 €	1
Reichelt	GS 40	IC-Sockel, 40-polig, doppelter Federkontakt	0,10 €	1
Reichelt	64P-10K	Präzisionspoti, 25 Gänge, liegend, 10 K-Ohm	0,30 €	1
Reichelt	TASTER 33301B	Kurzhubtaster 6x6mm, Höhe 9.5mm, 12V, vertikal	0,09 €	3
Reichelt	LED 3MM GN	LED, 3mm, Low Cost, grün	0,12 €	2
Reichelt	LED 3MM RT	LED, 3mm, Low Cost, rot	0,06 €	1
Reichelt	LED 3MM GE	LED, 3mm, Low Cost, gelb	0,07 €	1
Reichelt	LED 3MM BL	LED, 3mm, Low Cost, blau	0,28 €	1
Reichelt	WSL 6G	Wannenstecker 6-polig, gerade	0,22 €	1
Reichelt	WSL 10G	Wannenstecker 10-polig, gerade	0,08 €	1
Reichelt	WSL 20G	Wannenstecker 20-polig, gerade	0,26 €	1
Reichelt	MS 243	Kippschalter, 3A-125VAC, Ein-Aus	1,30 €	1
Reichelt	KERKO 100N	Keramik-Kondensator 100N	0,05 €	4
Reichelt	KERKO 10N	Keramik-Kondensator 10N	0,05 €	1
Reichelt	KERKO 22P	Keramik-Kondensator 22P	0,05 €	2
Reichelt	RAD 10/100	Elektrolytkondensator, 6,3x11mm, RM 2,5mm	0,04 €	2
Reichelt	2W METALL 10K	Metalloxidschicht-Widerstand 2W, 5% 10K-Ohm	0,10 €	3
Reichelt	2W METALL 220	Metalloxidschicht-Widerstand 2W, 5% 220 Ohm	0,10 €	2
Reichelt	2W METALL 150	Metalloxidschicht-Widerstand 2W, 5% 150 Ohm	0,10 €	2
Reichelt	2W METALL 120	Metalloxidschicht-Widerstand 2W, 5% 120 Ohm	0,10 €	1
Reichelt	SL 1X36G 2,54	Stiftleiste, 1-reihig, 36-polig, gerade	0,14 €	1
Reichelt	BL 1X20G 2,54	20pol. Buchsenleiste, gerade, RM 2,54	0,67 €	1
Farnell	1671507	POWERTIP - PC2004ARS-AWA-A-Q - LCD MODUL, STN REFLECT 20X4	11,87 €	1

5.3 Abkürzungsverzeichnis

µC - Mikrocontroller
JTAG – Joint Test Action Group
ISP – In-System Programmer
ADC - Analog Digital Converter
PWM - Pulsweitenmodulation
LED - Leuchtdiode
LCD - Liquid Crystal Display
USB – Universal Serial Bus
I/O – Input/Output
SoC – System on a Chip
RAM – Random Access Memory
ROM – Read Only Memory
MIPS – Million instructions per second
USART – Universal Synchronous Asynchronous Receiver Transmitter
TWI – Two wire interface
IRQ – Interrupt Request
ISR – Interrupt Service Routine
IR - Infrarot
SAR – Sukzessive Approximation
DAC – Digital Analog Converter
GND - Ground
Vcc - Versorgungsspannung
OCR – Output Compare Register
CTC – Clear Timer on compare
SPI – Serial Peripheral Interface
I^2C – Inter IC Bus

5.4 Abbildungsverzeichnis

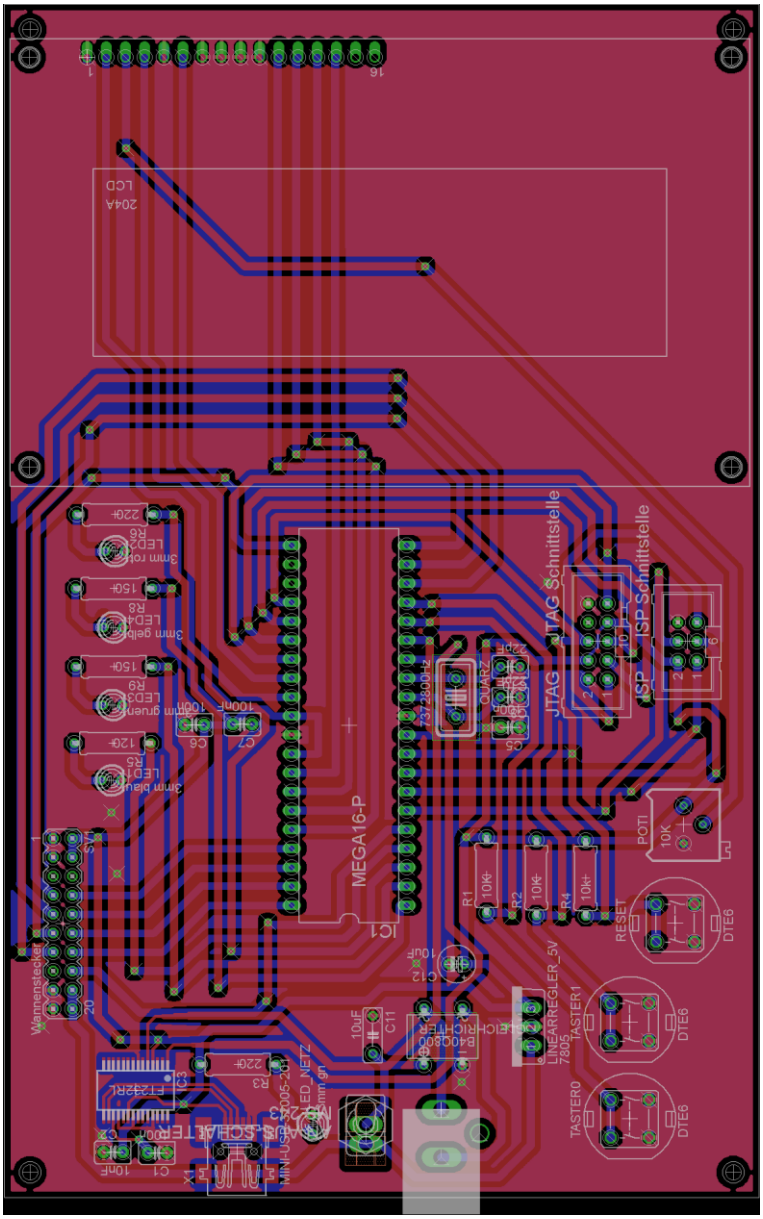
Abbildung 1: Aufbau des Experimentierboards 2
Abbildung 2: Gesamtschaltplan des Experimentierboards 4
Abbildung 3: ATmega-16 Pinbelegung (Quelle: Datenblatt) 5
Abbildung 4: Spannungsversorgung 6
Abbildung 5: Quarz mit Anschwingkondensatoren 6
Abbildung 6: ISP Programmiergerät Atmel AVRISP mkII (Quelle: Wikipedia) 7
Abbildung 7: Beschaltung der ISP-Buchse 7
Abbildung 8: Active-low-Schaltung mit 1k-Pullup-Widerständen 8
Abbildung 9: LEDs mit Vorwiderständen 9
Abbildung 10: Zeitliche und wertmäßige (3-Bit) Quantisierung eines Analogsignals 10
Abbildung 11: Prinzip Sukzessive Approximation 11
Abbildung 13: PWM-Signal ohne (gelb) und mit Tiefpass-Filter 2. Ordnung (violett), Simulation in LTspice 14
Abbildung 12: Prinzip PWM 13
Abbildung 14: Erzeugung des PWM-Signals (invertierendes Fast PWM) 14
Abbildung 15: Erzeugen einer „Sinusschwingung“ per nicht-invertierender Fast-PWM 15
Abbildung 16: 4x20 Zeichen Text-Display 16
Abbildung 17: 4bit-Beschaltung eines HD44780 LCD 17
Abbildung 18: UART Ansteuerung des USB-Interfaces FT232 18
Abbildung 19: main.c: Software Programmer's Notebook [WinAVR] 21
Abbildung 20: Makefile in WinAVR 22

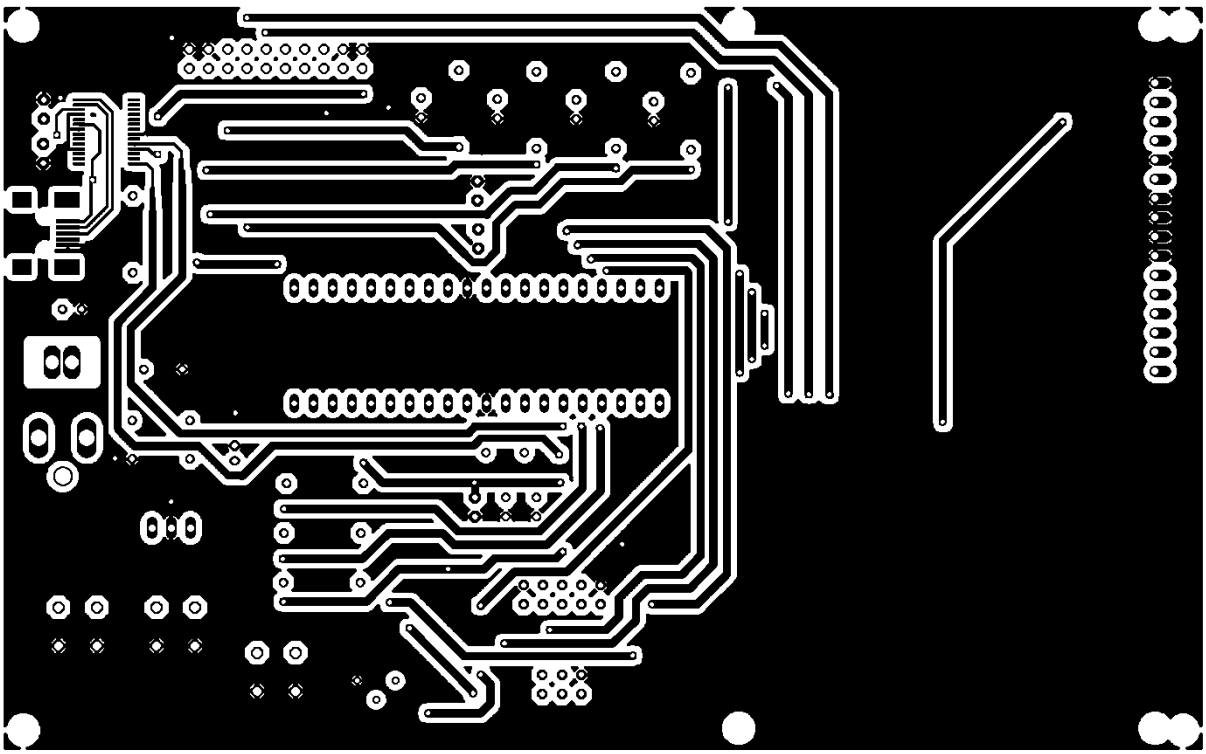
Abbildung 21: Kompilieren unter Programmer's Notepad [WinAVR] 23
Abbildung 22: Sequentieller Programmablauf 34
Abbildung 23: Programm mit Interrupten 34
Abbildung 24: lcd-routines.h: Pindefinition für Experimentierboard 51
Abbildung 25: Makefile mit eingebundener lcd-routines.c 51
Abbildung 26: Gesamtsystem: Experimentierboard, externer Temperatursensor, Lüfter und graphische Ausgabe 53
Abbildung 27: Flussdiagramm Benutzereingabe 54
Abbildung 28: Kennlinie R(T_{amb}) des KTY-81 und Linearisierung 55
Abbildung 29: Spannungsteilerschaltung 56
Abbildung 30: Kennlinie P-Regler 57
Abbildung 31: PI-Regler Zusammensetzung 58
Abbildung 32: Ausgabe der Daten in LogView 60

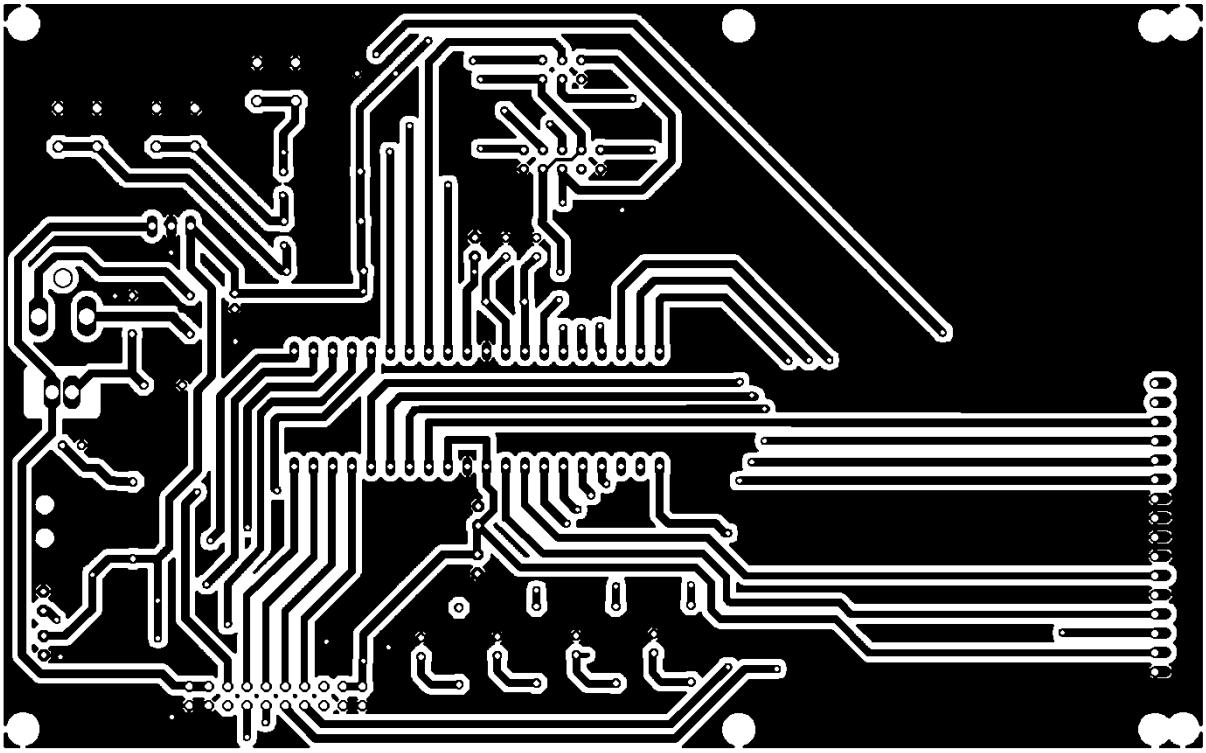
5.5 Literatur

Atmel. *Datenblatt ATmega 16.*
Brinkschulte. *Mikrocontroller und Mikroprozessoren.* Springer-Verlag.
FTDI. *Datenblatt FT232R.*
Hofmann, M. *Mikrocontroller für Einsteiger.* Franzis.
Krüger, G. *Programmieren in C.* Addison-Weslay.
Mikrocontroller.net Tutorial. (2011). Von http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial abgerufen
Philips. *Datenblatt KTY 81.*
Pont, M. *Embedded C.* Pearson Education limited.
Powertip. *Datenblatt LCD PC2004-A.*
Roboter Netz. (2011). Von http://www.rn-wissen.de/ abgerufen
Schäffer, F. *AVR Hardware und C-Programmierung in der Praxis.* Elektor Verlag.
Spanner, D. G. *AVR-Mikrocontroller in C programmieren.* Franzis Verlag.
Wiegelmann, J. *Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller: C-Programmierung für Embedded-Systeme.* Hüthig Verlag.
Wikipedia. (2011).

5.6 Boardlayout







5.8.1 Programmieren des ATmega-16 mit Eclipse und dem AVR Plugin

Im Folgenden wird die Benutzung der IDE Eclipse erklärt. Diese bietet den Vorteil, dass sie wesentlich leichter und plattformunabhängig ist. Der wesentliche Nachteil besteht darin, dass sie nicht speziell für AVR Mikrocontroller optimiert ist und die Installation des Programmieradapters AVR ISP mkII unter Umständen problematisch sein kann. Installationsanleitungen sowie die aktuelle Version von Eclipse für C/C++ findet man im Internet.

Durch die Benutzung von Eclipse als IDE entfällt die Erstellung eines Makefiles. Dieses wird von Eclipse automatisch generiert und sollte nicht manuell editiert werden. Alle Einstellungen lassen sich über die Project Properties, wie im nächsten Kapitel beschrieben, einstellen.

5.8.1.1 Mikrocontroller konfigurieren

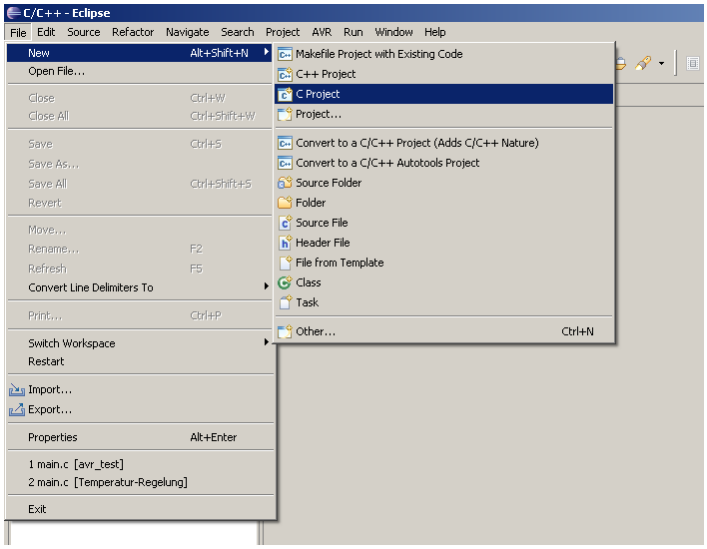
Zum Beschreiben des Programmspeichers des ATmega wird die Software Eclipse IDE for C/C++ Developers benutzt. Nach der Installation muss zunächst das AVR Plugin installiert werden. Eine Anleitung findet man per Google.

Hier die einzelnen Schritte für die Erstellung eines neuen Projektes:

<table>
<tr>
<td>

- *File → New → C Project*

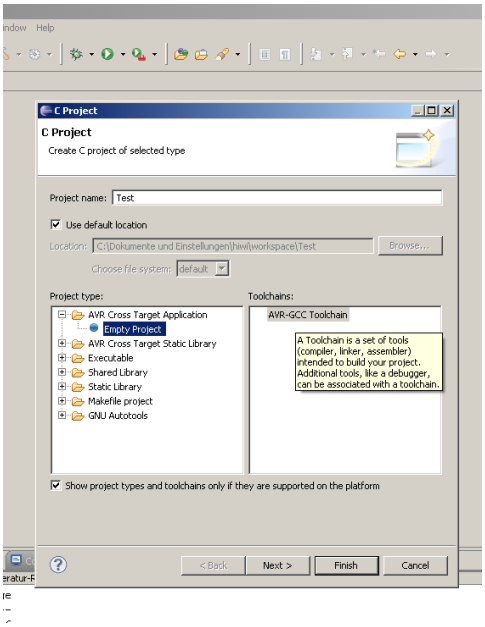
</td>
<td>



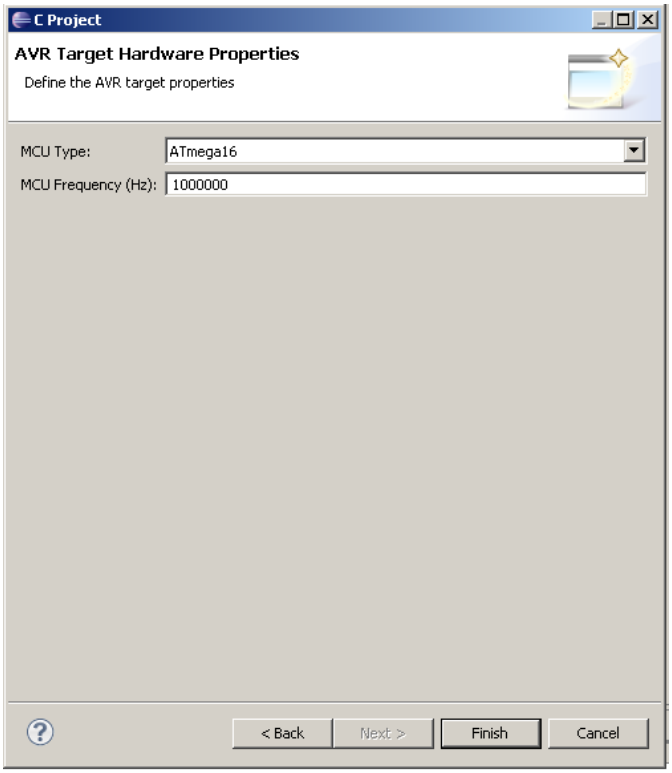
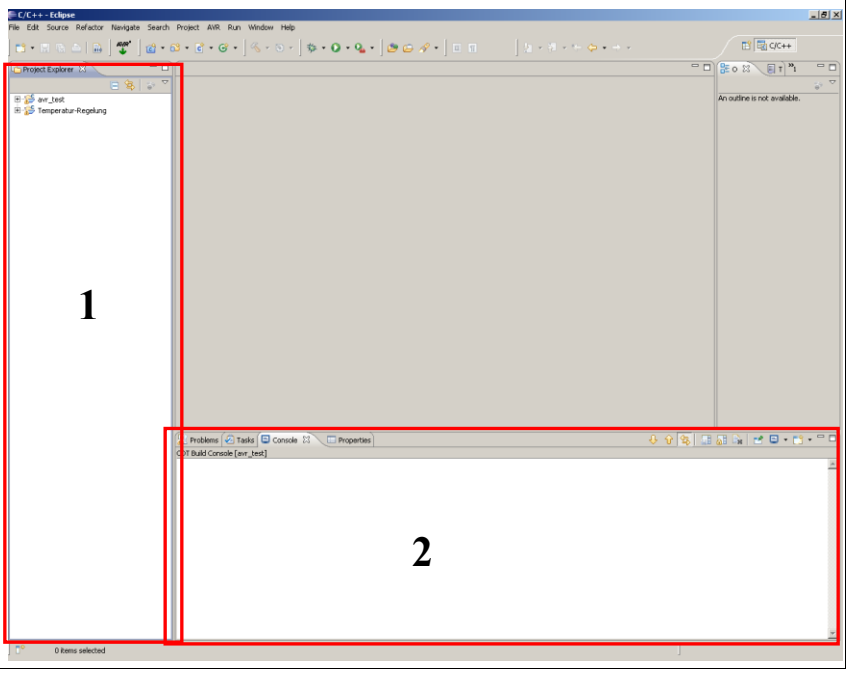
</td>
</tr>
<tr>
<td>

- Projektname angeben
- *AVR Cross Target Application → Empty Project*
- Weiter, nächste Seite unverändert belassen und Weiter

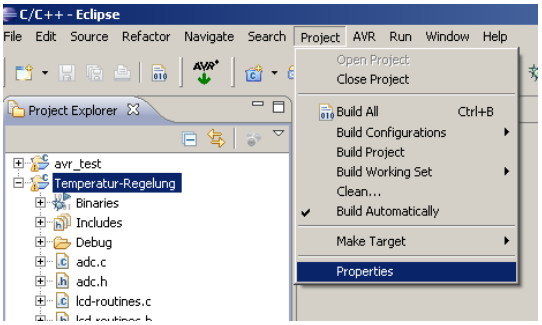
</td>
<td>



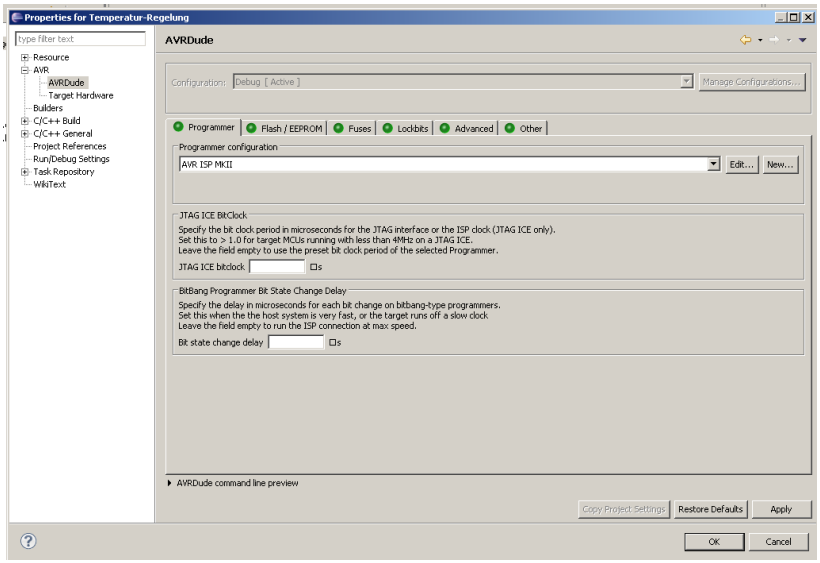
</td>
</tr>
</table>

• µC Typ setzen • Taktfrequenz einstellen	
• Rechts sieht man den Workspace (für Quelltext…) • 1: *Project Explorer* alle aktuell vorhandenen Projekte • 2: *Console* Ausgaben des Compilers etc.	

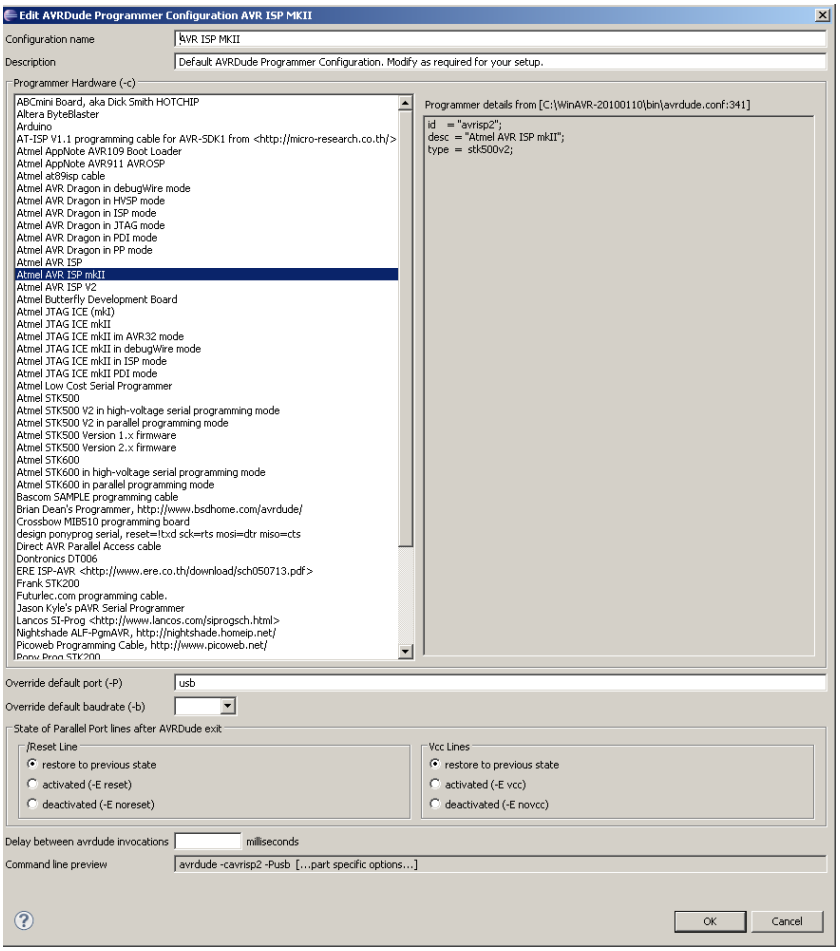
- *Project*
  → *Properties*



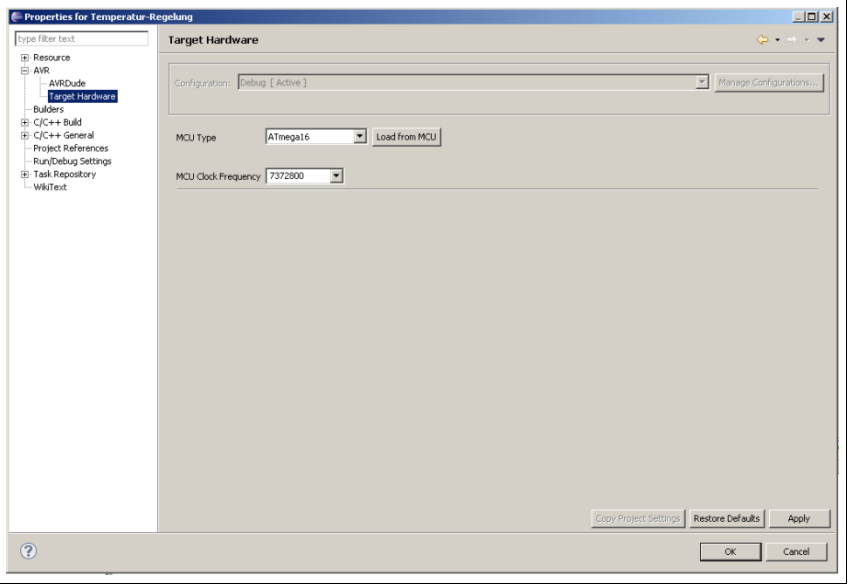
- Einstellungen zu dem aktuell im *Project Explorer* markierten Projekt
- *AVR* → *AVRDude*
- Über *New* lässt sich ein neuer Programmieradapter einstellen, über *Edit* ein vorhandener bearbeiten



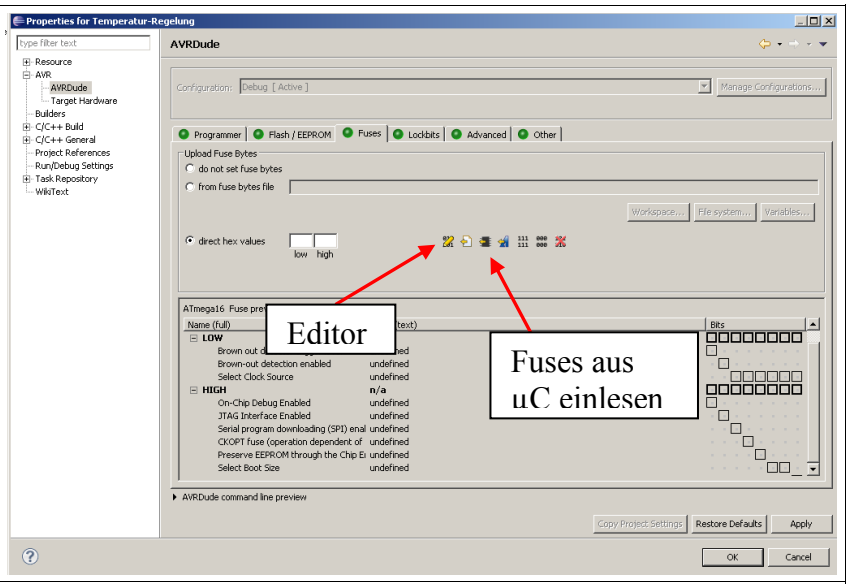
- *Configuration Name* eingeben (beliebig)
- *Programmer Hardware* auswählen (hier: Atmel AVR ISP mkII)
- *Override default port* auf *usb* setzen
- Mit OK bestätigen



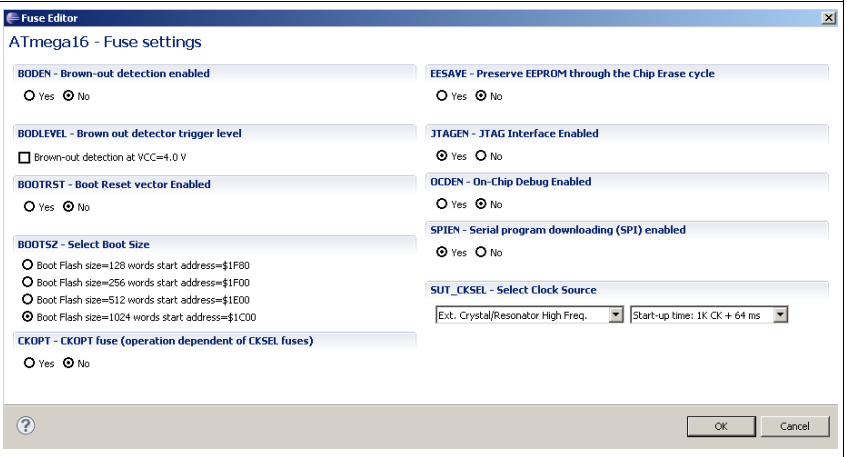
- Unter dem Punkt *Target Hardware* den verwendeten µC auswählen (bzw. automatisch auslesen)
- Die *MCU Clock Frequency* wurde beim Erstellen des Projektes schon gesetzt



- Zum Setzen der Fusebits unter *AVRDude* auf den Reiter *Fuses* gehen
- *Direct hex values* aktivieren
- Zuerst die Fuses auslesen
- Editor starten



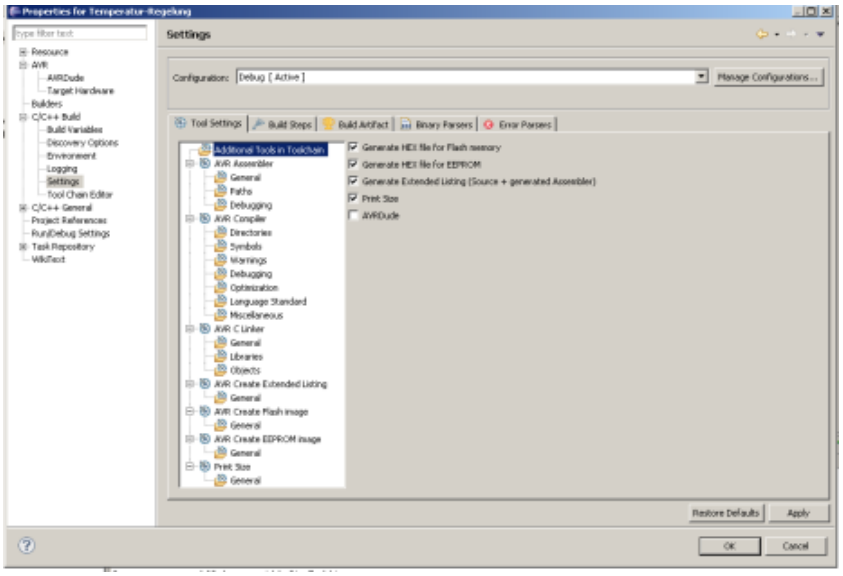
- *Select Clock Source* auf *Ext. Crystal/Resonator High Freq.* stellen → aktiviert den externen Quarz
- NIEMALS *SPIEN* auf No setzen → der Controller ließe sich nicht mehr programmieren!
- Beim nächsten *Upload* auf den Controller werden die Fuses mitgeschrieben



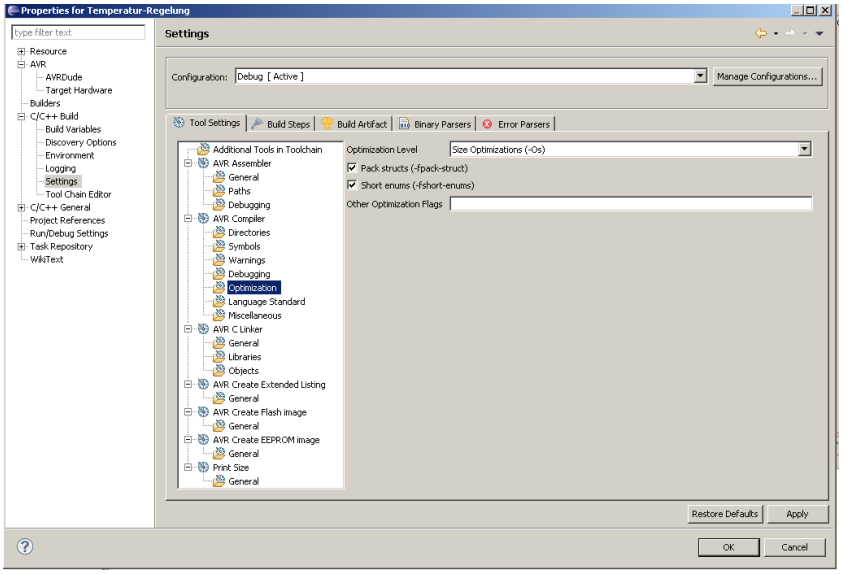
5.8.1.2 Mikrocontroller-Programmspeicher beschreiben in Eclipse

Als nächstes kann der Flash-Speicher des µC mit einem kompilierten Programm beschrieben werden:

- *Project Properties*
- *C/C++-Build*
  → *Settings*
- *Additional Tools in Toolchain*:
  Generate HEX File for Flash anwählen



- Unter *Optimization*:
  Optimization Level
  → *Size Optimization (-Os)*



- Kompilieren und Linken mit dem *Build-Button*
- Übertragen auf den µC mit dem *Upload-Button*