Diplomarbeit

Fachhochschule Augsburg

University of Applied Sciences

Studienrichtung

Informatik

Bianca-Charlotte Liehr

Einführung in die Programmierung

des AVR-Controllers

Eine Anleitung zum Aufbau der Hardware und zum

Einstieg in die Programmierung

Verfasser der Diplomarbeit:
Bianca-Charlotte Liehr
Sandweg 2
86444 Anwalting
Telefon: 08207 962995
bianca.liehr@gmail.com

Fachbereich Informatik
Telefon: +49 821 5586-450
Fax: +49 821 5586-499

Fachhochschule Augsburg
University of Applied
Sciences
Baumgartnerstraße 16
D 86161 Augsburg

Telefon +49 821 5586-0
Fax +49 821 5586-222
www.fh-augsburg.de
poststelle@fh-augsburg.de

Erstprüfer: Prof. Dr. Hubert Högl

Zweitprüfer: Prof. Dr. Gundolf Kiefer

COMMONS DEED

Namensnennung-NichtKommerziell-Weitergabe unter gleichen Bedingungen 2.0 Deutschland

Sie dürfen:

- den Inhalt vervielfältigen, verbreiten und öffentlich aufführen
- Bearbeitungen anfertigen

Zu den folgenden Bedingungen:

Namensnennung. Sie müssen den Namen des Autors/Rechtsinhabers nennen.

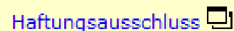
Keine kommerzielle Nutzung. Dieser Inhalt darf nicht für kommerzielle Zwecke verwendet werden.

Weitergabe unter gleichen Bedingungen. Wenn Sie diesen Inhalt bearbeiten oder in anderer Weise umgestalten, verändern oder als Grundlage für einen anderen Inhalt verwenden, dann dürfen Sie den neu entstandenen Inhalt nur unter Verwendung identischer Lizenzbedingungen weitergeben.

- Im Falle einer Verbreitung müssen Sie anderen die Lizenzbedingungen, unter die dieser Inhalt fällt, mitteilen.
- Jede dieser Bedingungen kann nach schriftlicher Einwilligung des Rechtsinhabers aufgehoben werden.

Die gesetzlichen Schranken des Urheberrechts bleiben hiervon unberührt.

Das Commons Deed ist eine Zusammenfassung des Lizenzvertrags in allgemeinverständlicher Sprache.



Inhaltsverzeichnis **Seite**

1 Einleitung ... 6
1.1 Aufgabenstellung ... 6
1.2 Motivation ... 7
2 ATmega16... 8
2.1 Einführung ... 8
2.2 Beschreibung des ATmega16 ... 9
2.2.1 Mikrocontroller ... 9
2.2.2 Bauform ... 9
2.2.3 Ausführung ... 10
2.3 Blockschaltbild und Pinbelegung ... 10
2.4 Fusebits und Lockbits ... 12
2.5 Benötigte Hardware ... 13
2.5.1 ISP ... 13
2.5.2 JTAG ... 13
2.5.3 Evertool ... 14
3 Software ... 15
3.1 WinAVR (Version 2006/01/25) ... 16
3.1.1 avr-libc ... 16
3.1.2 Programmer's Notepad ... 17
3.1.3 Makefiles ... 17
3.1.4 mFile ... 18
3.1.5 AVRDude ... 19
3.1.6 AVaRICE ... 20
3.1.7 avr-gdb ... 21
3.1.8 AVR insight... 21
3.2 AVR Studio 4.12 ... 22
3.2.1 Projekt anlegen ... 23
3.2.2 Programmieren des Chips ... 25
3.2.3 AVR Studio 4.12 Simulator ... 26
3.2.4 Fusebits und Lockbits ... 27
4 Programmierung des Mikrocontrollers ... 30
4.1 Grundlagen ... 30
4.1.1 Programmablauf ... 30
4.1.2 Portregister ... 30
4.1.3 Codebeispiele zur Bitmanipulation ... 32
4.1.4 Ansteuerung von LEDs ... 33
4.1.5 Delay ... 34
4.1.6 Pull-Up und Pull-Down Widerstände... 35
4.1.7 Entprellung von Tastern ... 35

4.2 Speicherzugriffe 36
4.2.1 RAM 36
4.2.2 Flasch 36
4.2.3 EEPROM 37
4.3 Interrupts 38
4.3.1 Interruptverarbeitung 38
4.3.2 Nicht unterbrechbare Interrupts 38
4.3.3 Unterbrechbare Interrupts 39
4.3.4 Interruptvektoren 39
4.3.5 Behandlung eines externen Interrupts........ 40
4.4 Watchdog 41
4.4.1 Aktivierung des Watchdogs 41
4.5 Timer/Counter 42
4.5.1 Timer/Counter0 43
4.5.2 Timer/Counter1 47
4.5.3 Ansteuerung einer LED über PWM 48
4.6 USART 50
4.6.1 Initialisierung des USART 50
4.6.2 Terminal 51
4.7 Analog/Digital Wandler 53
4.7.1 A/D-Wert Berechnung 54
4.7.2 A/D-Wandler Register 54
4.7.3 Initialisierung Free Running Mode 57
4.7.4 Initialisierung Single Conversion Mode 57
4.8 LC Display 58
4.8.1 Pinbelegung eines 4x20 LC Displays 58
4.8.2 LCD Steuerungsfunktionen 59
4.8.3 Initialisierung des Displays 63
4.8.4 Eigene Zeichen definieren 64
4.9 Operationsverstärker 66
4.9.1 Komparator 66
4.9.2 Verstärker 67
4.9.3 Subtrahierer 67
4.6.4 Addierer 68
4.10 Temperatursensoren 69
4.10.1 Sensortypen 69
4.10.2 Anschluss des KTY81-210 70
4.10.3 Kalibrierung des KTY81-210 72
4.11 Summer 73
4.11.1 Piezo-Schallwandler 73

4.12 TWI 74
4.12.1 TWI-Schaltung....74
4.12.2 Adress- und Datenpaket 75
4.12.3 Kommunikationsablauf 77
4.12.4 Bitraten Generator 77
4.12.5 TWI Register 78
4.12.6 TWI Initialisierung 80
4.12.7 TWI Übertragungsarten 80
4.13 Echtzeituhr RTC 4513 86
4.13.1 Register der RTC 4513 86
4.13.2 Initialisierung der RTC 86
4.13.3 Power-On Prozedur 86
4.13.4 Write-Mode 87
4.13.4 Read-Mode 88
4.13.4 Read Flag 89
5 Hardware 90
5.1 Schaltzeichen 90
5.2 Hardware löten 95
5.2.1 Platinenbau 95
5.2.2 Schaltpläne lesen 95
5.2.3 Löten 96
5.3 Parallelport Programmieradapter 99
5.4 Entwicklungsplatine BL1000 100
5.4.1 Beschreibung der BL1000 Platine 101
6 Fazit 102
7 Anhang 103
A Abbildungsverzeichnis 103
B Tabellenverzeichnis 105
C Quellenverzeichnis 106
C.1 Webseiten 106
C.2 Datenblätter 107
D Inhalt der CD 108
E Schaltpläne 109
F Erstellungserklärung 111

1 Einleitung

1.1 Aufgabenstellung

Der Fachbereich Informatik der FH Augsburg bietet seinen Studenten im
5. Semester die Vorlesung Rechnertechnik an. Die Teilnehmer haben dort erstmalig die Chance mit Hardware und deren Programmierung in Kontakt zu kommen. Bisher wurde für diesen Zweck die Entwicklungsplatine NF300 zur Verfügung gestellt. Im Rahmen dieser Diplomarbeit sollte eine neue Basisplatine entwickelt werden, welcher nicht mehr der 68000 Mikrocontroller von Motorola zu Grunde liegt, sondern ein AVR-Controller der Firma ATMEL. Dieser kann sowohl in Assembler als auch in C programmiert werden. Ziel dieser Arbeit soll es sein, den Studenten den Einstieg in die hardwarenahe Programmierung eines aktuellen Mikrocontrollers in der Programmiersprache C zu erleichtern. Zu diesem Zweck wurde die Entwicklungsplatine BL1000 entwickelt. Anhand eines auf die Platine abgestimmten Tutorials wird dem Studenten Schritt für Schritt der Umgang mit Hard- und zugehöriger Software beigebracht. Im Einzelnen beinhaltete die Aufgabenstellung folgende Rahmenpunkte:

- Anleitung zur Bedienung der benötigten Software
- Erklärung des AVR Mikrocontrollers
- Erläuterung der Chipfunktionen anhand von Codebeispielen
- Erstellung der Entwicklungsplatine BL1000
- Einführung in den Hardwarebau

Auf der entwickelten Basisplatine waren folgende Bauteile erwünscht:

- ATmega16
- RS-232 Schnittstelle
- 4 Leuchtdioden
- 2 Taster
- 4 zeiliges LC-Display
- Summer
- Echtzeituhr
- Temperatursensor
- Operationsverstärker
- Spindelpotentiometer
- I²C-Bus[i²C]

Alle auf der Platine vorhandenen Bausteine sollten im Tutorial erklärt werden und durch separate Programme in Betrieb genommen werden können.
Des weiteren beinhaltete die Aufgabenstellung die Entwicklung eines Hauptprogramms, das die einzelnen Programme in einem Modul vereinen und das Zusammenspiel der verschiedenen Funktionalitäten verdeutlichen sollte.

1.2 Motivation

In der Automatisierungstechnik ist es heutzutage nicht mehr möglich, ohne Mikrocontroller auszukommen. Da Embedded-Systeme immer wichtiger werden, bestand die Motivation zur Erstellung dieser Diplomarbeit darin, eine Abhandlung zu entwickeln, die es interessierten Studenten erleichtert, sich mit diesem Thema auseinander zu setzen. Der Einstieg in die hardwarenahe Programmierung sollte durch das Aufzeigen von Fehlerquellen, der Bereitstellung detaillierter Beschreibungen und anhand von Codebeispielen die Hemmschwelle herabsetzen, sich mit dem Thema Mikrocontroller zu beschäftigen. Des weiteren sollte die Möglichkeit geboten werden, Studenten, die noch nie mit Hardware in Berührung gekommen sind, einen Leitfaden zum Bau einer eigenen Platine an die Hand zu geben.

2 ATmega16

2.1 Einführung

Der im Tutorial verwendete Chip ist der ATmega16[ATM16], ein 8Bit RISC Prozessor aus der AVR Reihe von ATMEL. Sollte sich auf der in der Vorlesung verwendeten Platine ein
ATmega32 befinden, dann ändert sich nichts an der Handhabung des Controllers, da der einzige Unterschied zwischen einem ATmega16 und einem ATmega32 die Größe der Speicherbausteine (Flash, EEPROM, RAM) ist.
Die folgenden Seiten sollen es Einsteigern erleichtern, den Mikrocontroller zu programmieren, daher sind in diesem Tutorial vor allem die anfänglichen Stolpersteine ausführlich erklärt. Zum tieferen Verständnis einzelner Funktionen des ATmega16 ist es auf jeden Fall notwendig, das Datenblatt zu konsultieren. Es befindet sich im Verzeichnis "Datenblätter/ATmega16/" auf der CD. Die neueste Version steht unter folgendem Link zum Download zur Verfügung:

http://www.atmel.com/dyn/products/product_card.asp?part_id=2010

Die angegebenen Seitenzahlen in diesem Dokument beziehen sich auf das im Ordner befindliche Datenblatt (Rev. 2466D-09/02).
Das Datenblatt des verwendeten Mikrocontrollers ist das wichtigste Utensil, das bei der Programmierung erforderlich ist, da in diesem der komplette Aufbau des Chips und seine Handhabung erklärt wird. Reicht das Datenblatt nicht aus, um die gewünschte Funktion zu realisieren, findet man auf folgenden Webseiten Hilfe:

http://www.mikrocontroller.net
http://www.roboternetz.de
http://www.avrfreaks.net

Diese Webseiten bieten eine Vielzahl von Artikeln zu unzähligen µC[1]-Themen an. Gerade für Einsteiger bieten diese Artikel meist nützliche Tipps zu anstehenden Problemen. Außerdem lohnt sich immer ein Durchstöbern der Foren, da meist schon einmal jemand mit den gleichen Schwierigkeiten zu kämpfen hatte. Mikrocontroller.net und Roboternetz.de sind deutschsprachige Seiten, die sich mit einer breiten Palette von Mikrocontrollern auseinander setzen. Avrfreaks.net hingegen hat sich auf ATMEL AVR Chips spezialisiert, weshalb man bei controllerspezifischen Problemen wohl hier die beste Anlaufstelle findet. Allerdings ist dies eine rein englischsprachige Seite.

1 µC - Kurzschreibweise für Mikrocontroller

2.2 Beschreibung des ATmega16

2.2.1 Mikrocontroller

Ein Mikrocontroller ist ein Chip, bei dem CPU, Programm- und Arbeitsspeicher, EEPROM, Taktgenerator und Schnittstellen zu Bussystemen auf einem Chip vereint sind.
Alle Mikrocontroller der ATMEL AVR Reihe sind RISC-Prozessoren, die der Harvard-Architektur entsprechen.

RISC (Reduced Instruction Set Computing) [risc]
Bei diesen Prozessoren wird pro Takt ein elementarer Befehl ausgeführt, wobei jeder benutzbare elementare Befehl festverdrahtet ist. Komplexere Befehle werden vom Compiler in elementare Befehle zerlegt. Durch die Verwendung von RISC wird die Ausführung des Codes beschleunigt und es wird Platz auf dem Chip gespart.
Harvard-Architektur [hav]
Mikrocontroller dieser Architektur verfügen über physikalisch voneinander getrennte Haupt- und Programmspeicher. Dadurch ist es möglich Befehle und deren Daten in einem einzigen Takt in das Rechenwerk zu laden. Bei der klassischen Von-Neumann-Architektur[vna] werden dazu mindestens zwei Taktzyklen benötigt.

2.2.2 Bauform

Der Chip ist in 2 verschiedenen Bauformen erhältlich.
DIP oder DIL (Dual-in-line package oder Dual In-line)
Die Pins des Chips liegen an den beiden langen Seiten und werden entweder auf der Unterseite der Platine angelötet oder in einen auf der Platine aufgelöteten Sockel gesteckt.
SMD (Surface Mounted Device)
SMD heißt, dass die Chips des Beinchens nicht mehr durch die Bestückungslöcher der Platine geführt, sondern direkt auf der Oberfläche mit den Leiterbahnen verbunden werden. So ist es zum Beispiel auch möglich, Platinen beidseitig zu bestücken. Als SMD-Version ist der ATmega16 als MLF (*Micro Lead Frame Package*) oder als TQFP (*Thin Quad Flat Pack*) verfügbar. Die TQFP-Version kann man mit viel Geschick noch mit der Hand löten, da seine Größe gerade einmal 12mm auf 12mm beträgt, die Pins nur 0,3mm breit sind und in einem Abstand von 0,8mm voneinander entfernt positioniert sind. Die MLF-Version kann nur maschinell befestigt werden, da die Pins unter dem Gehäuse des Chips verborgen sind. Die Pinbreite beträgt hier 0,18mm und der Pinabstand 0,5mm

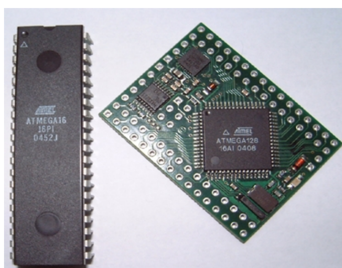


Abb. 01: DIP- und TQFP-Gehäuse

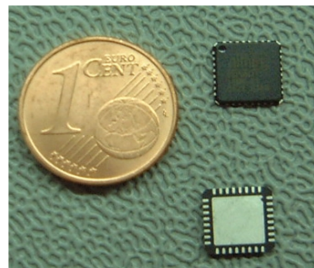


Abb. 02: MLF-Gehäuse

2.2.3 Ausführung

Der ATmega16 wird in zwei Ausführungen angeboten - einmal als ATmega16 und außerdem als ATmega16L.
Die *L-Version* unterscheidet sich dahingehend vom ATmega16, dass der Chip über einen größeren Spannungsbereich (2,7V bis 5,5V anstatt 4,5V bis 5,5V) verfügt, und daher gerade bei Anwendungen, bei denen auf einen geringen Stromverbrauch geachtet werden muss , oder bei Platinen, auf denen Bauteile vorhanden sind, die eine maximale Spannung von 3V vertragen, eher eingesetzt wird. Ein weiterer Unterschied ist, dass die *L-Version* mit einer maximalen Frequenz von 8MHz getaktet werden darf, wohingegen der normale
ATmega16 mit einer Taktfrequenz bis 16MHz betrieben werden kann.

2.3 Blockschaltbild und Pinbelegung

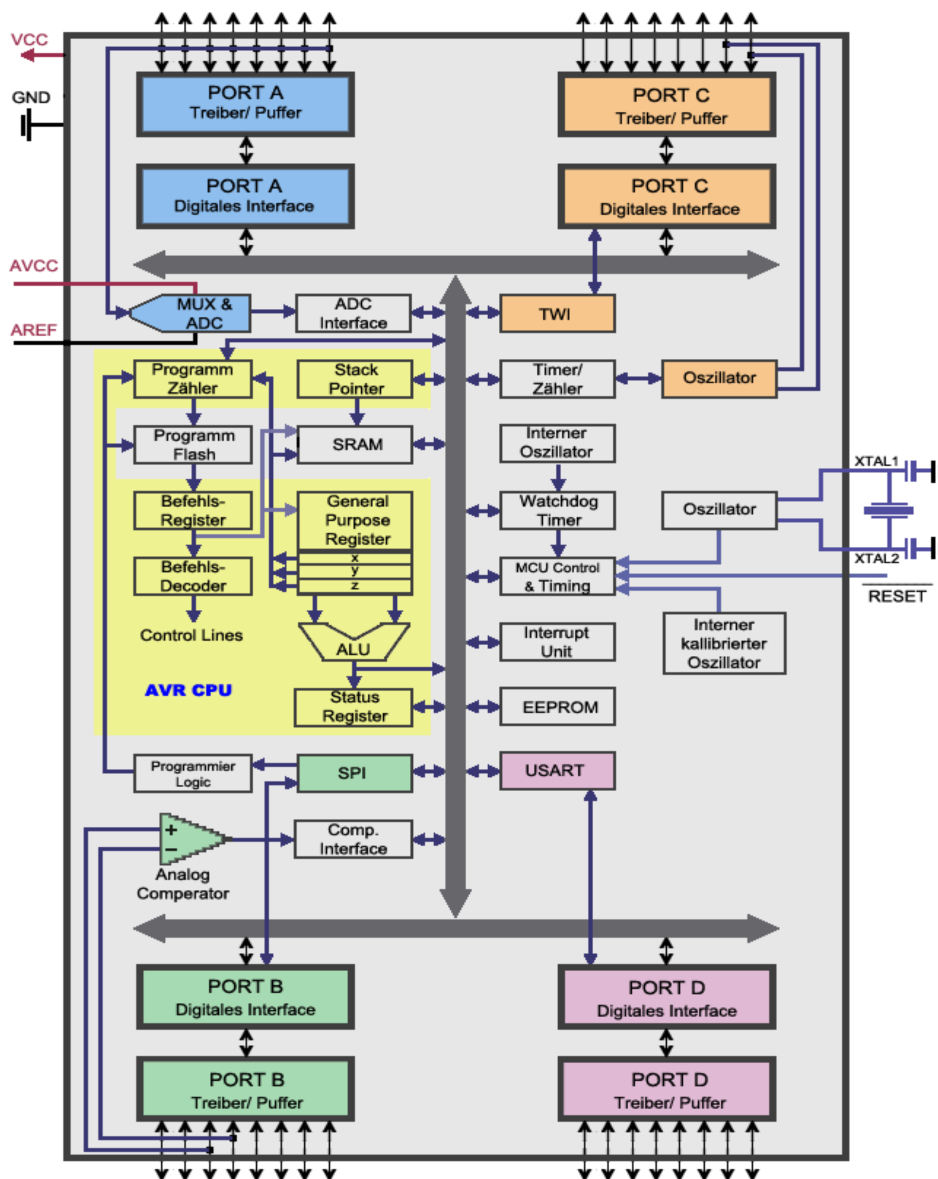


Abb. 03: Blockschaltbild ATmega16

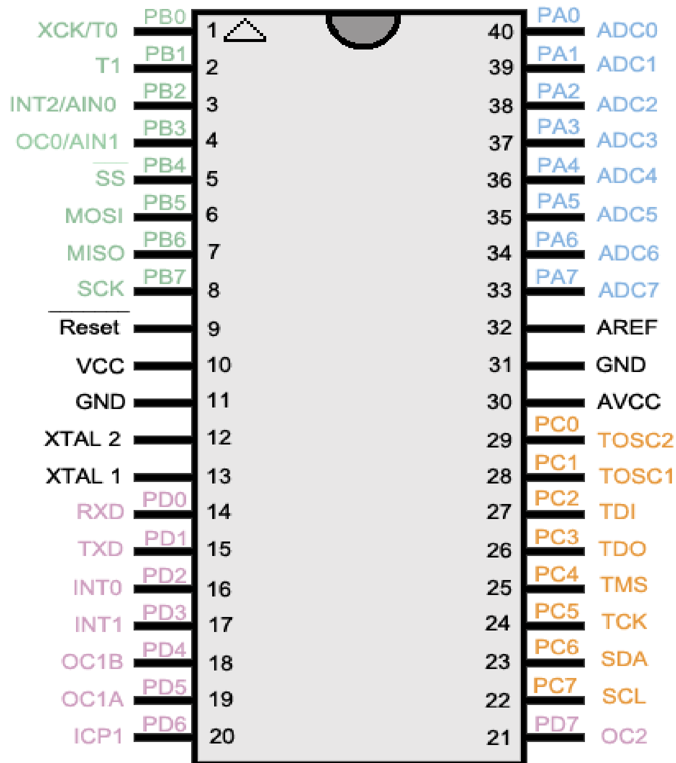


Abb. 04: Blockschaltbild ATmega16

Der DIL-Chip verfügt über 40 Pins. 32 davon können als I/O-Pins verwendet werden. Das heißt, dass jeder dieser 32 Pins als Eingang oder als Ausgang für Signale und Operationen verwendet werden kann.

Die I/O Pins sind in 4 Ports à 8 Pins aufgeteilt und folgendermaßen belegt:

Port A:

Alle 8 Pins können als Eingänge für die Analog-Digital-Wandlung verwendet werden. Wie man im Blockschaltbild sieht, sind sie zum Einen direkt mit dem chipinternen A/D Wandler verbunden, zum Anderen können die Pins 7 bis 0 des Ports auch für andere I/O-Funktionen verwendet werden.

Port B:

Alle Pins des Ports B können für I/O Operationen verwendet werden.

Die Pins 7 bis 4 dienen außerdem als Anschlüsse für den SPI[2]-Bus (SCK, MOSI, MISO, SS), an den der ISP angeschlossen ist.

Der Pin 3 (AIN0/ OC0) wird zum Einen als Ausgang für Timer/Counter0 - Anwendungen und zum Anderen als Eingang für das erste Signal des Analogkomparators verwendet.

Der Pin 2 (INT2) dient als Eingang für das zweite Signal des Analogkomparators und außerdem als Eingang für ein externes Interrupt-Signal.

Pin 1 (T1) und Pin 0 (T0/XCK) können als Eingangspins für externe Zeitgeber verwendet werden. Ein an Pin 0 angeschlossener Quarzoszillator kann außerdem als externer Taktgeber für USART im synchronen Modus fungieren.

2 SPI - Serial Peripheral Interface

Port C:

An die Pins 7 und 6 (TOSC2 und TOSC1)kann ein zweiter Oszillator-Quarz angeschlossen werden.

Pins 5 bis 2 (TDI, TDO, TMS, TCK) werden für den Anschluss der JTAG[3]-Schnittstelle benötigt.

Pin 1 und 0 dienen als Kommunikationspins für die *TwoWire*-Schnittstelle, besser bekannt als I^2C[4].

Port D:

Pin 7, 5, 4 (OC2, OC1A, OC1B) können als Ausgänge für Timer/Counter1 und Timer/Counter2-Operationen verwendet werden.

Pin 6 (ICP1) kann als Eingang für ein externes Signal benutzt werden, das bei Änderung der Flanke ein Ereignis am Timer/Counter2 auslöst.

Pins 3 und 2 (INT1 & INT0) fungieren als Eingänge für externe Interruptsignale.

Die Pins 1 und 0 (TXD & RXD) müssen bei der Verwendung von USART angeschlossen werden.

Die restlichen 8 Pins können nicht für I/O-Operationen verwendet werden, da sie vorbelegte, nicht veränderbare Funktionen erfüllen. Im Einzelnen sind dies:

Pin 9: RESET

Dieser Pin ist "active low", d.h. wenn die anliegende Spannung auf 0V gezogen wird, dann wird ein Reset (Neustart des Programms) ausgelöst.

Pin 10: VCC

Hier wird die Versorgungsspannung (zwischen 3 und 5 Volt - je nach Chipversion) angelegt.

Pin 11: GND

Dieser Pin wird mit der Masse verbunden.

Pin 12: XTAL2 und Pin 13: XTAL1

Wird ein externer Quarz zur Taktgebung verwendet, dann wird dieser mit diesen beiden Pins verbunden. Der ATmega16 stellt aber auch einen internen Oszillator mit 1MHz zur Verfügung.

Pin 30: AVcc Pin31:GND Pin32:AVREF

Diese 3 Pins werden angeschlossen, wenn der A/D Wandler verwendet wird. Allerdings wird im Datenblatt geraten, dass AVCC auch an VCC angeschlossen wird, wenn der A/D-Wandler nicht benutzt wird.

2.4 Fusebits und Lockbits

Diese Bits dienen zur Manipulation der Chip-Einstellungen. Man sollte sie nur anfassen, wenn man wirklich weiß, was man tut, da das Verändern dieser Bits leicht dazu führen kann, dass man sich aus dem Chip aussperrt. Fusebits sind Konfigurationsbits des Chips, die bei der Auslieferung teilweise schon gesetzt sind. Hier lässt sich zum Beispiel die externe Taktfrequenz einstellen. Lockbits werden gesetzt, wenn die Software fehlerfrei funktioniert und der Mikrochip, für welche Aufgabe auch immer, freigegeben wird. Es ist dann möglich, die Lockbits so zu setzen, dass niemand mehr auf den Code zugreifen kann. Eine genaue Beschreibung dieser Bits und ihrer Manipulation ist unter im Kapitel "*AVR Studio - Fusebits und Lockbits* zu finden.

3 JTAG - Joint Test Action Group vgl. Kapitel 2.5.2

4 I^2C - Inter-Integrated Circuit vgl Kapitel 4.12

2.5 Benötigte Hardware

2.5.1 ISP

Um einen Mikrocontroller programmieren zu können, benötigt man spezielle Hardware, nämlich einen so genannten ISP.

Abb. 05: ISP-Adapter für die parallele Schnittstelle

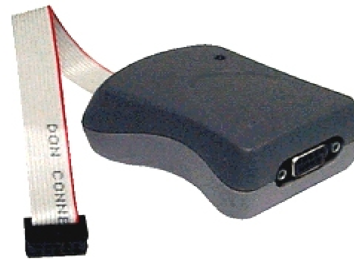


Abb. 06: ATMEL ISP für die serielle Schnittstelle

Der ISP stellt die Verbindung zwischen PC und Mikrocontroller zur Verfügung. Der ISP kann, je nach Hardware sowohl über die serielle, als auch über die parallele Schnittstelle des PCs betrieben werden und eine Verbindung zum Mikrocontroller aufbauen.
Erst durch die Programmierung des Chips über den ISP ist es möglich den Chip direkt auf der Platine zu programmieren - also im System.
Es besteht die Möglichkeit, beide ISPs selbst zu bauen. Der ISP, der über die serielle Schnittstelle betrieben wird, ist zum Beispiel ein Bestandteil des Evertools. Für den Bau eines Parallelport-ISP findet sich eine Anleitung im Kapitel 5.3.

2.5.2 JTAG

Bis vor nicht allzu langer Zeit war das Debuggen von Mikrocontrollern entweder gar nicht oder nur mit Hilfe von teuren Emulatoren möglich. Mitte der 80er Jahre wurde dann JTAG entwickelt, ein standardisiertes Debugverfahren, um die Fehlersuche in IC-Programmen zu erleichtern. JTAG kann neben dem Debuggen auch zum Programmieren des Chips verwendet werden, und fungiert dann praktisch auch noch als ISP. Um JTAG nutzen zu können, benötigt man einen JTAG-Adapter, der 4 Pins des Mikrocontrollers beansprucht. Die Firma ATMEL kombiniert ihr JTAG zusätzlich noch mit einem ICE - *In Circuit Emulator* - zum JTAG ICE. Mit Hilfe des ICE können sämtliche digitale und analoge Chipfunktionen emuliert werden.

Abb. 07: AVR JTAG ICE mk2

Neben der Möglichkeit den Code durch JTAG zu debuggen, sollte die Fehlersuche über das Oszilloskop nicht vernachlässigt werden, da es vorkommen kann, dass der Code einwandfrei funktioniert, allerdings aber eine Verbindungsleitung auf der Platine fehlerhaft ist. Durch das Oszilloskop bietet sich dann die Möglichkeit, nicht-vorhandene oder falsche Signale aufzuspüren.

2.5.3 Evertool

Die Evertool-Platine von Martin Thomas kombiniert den AVR ISP und das AVR JTAG ICE auf einer Platine. Auf dieser Homepage

http://www.siwawi.arubi.uni-kl.de/avr_projects/evertool/

findet man den Schaltplan und eine detaillierte Anleitung zum Aufbau und zur Inbetriebnahme der Evertool-Platine. Der Bau der Platine ist daher interessant, da die aktuelle JTAG-Hardware der Firma ATMEL, das AVR JTAG ICE mk2, ca. 350€ kostet.

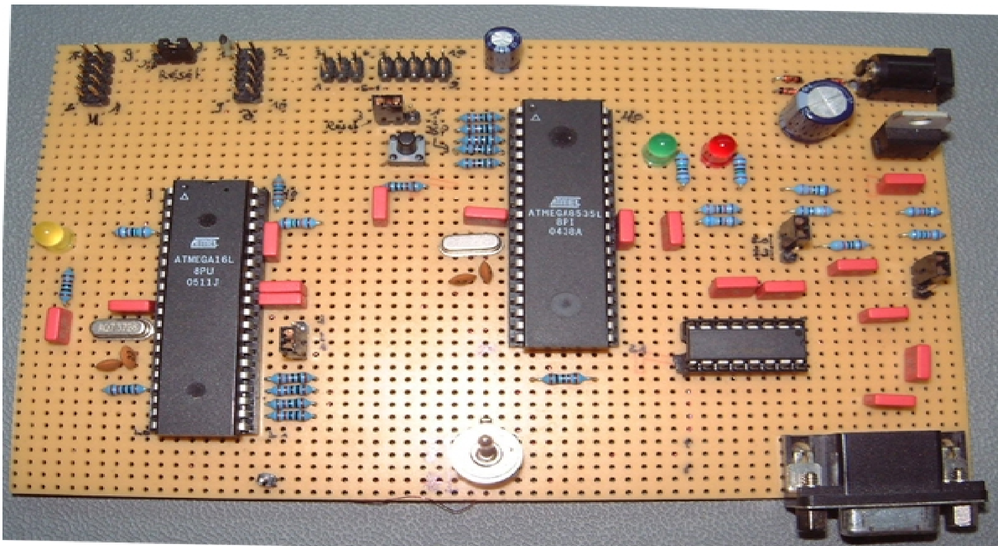


Abb. 08: Evertool

3 Software

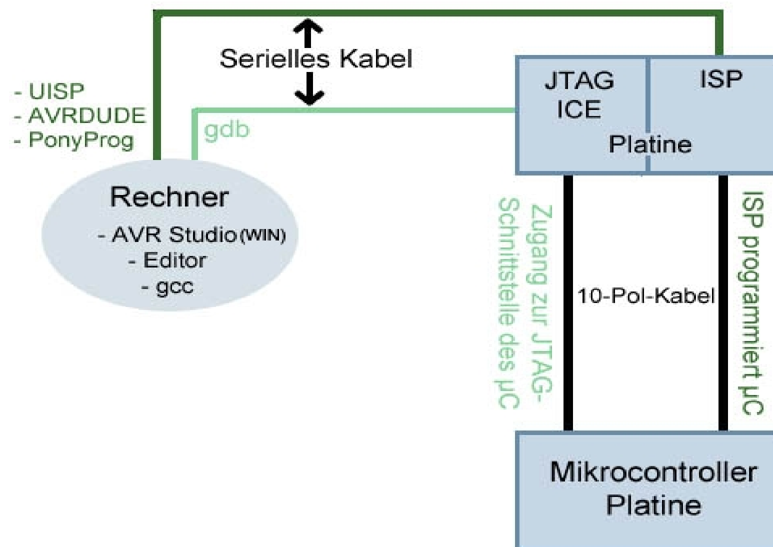


Abb. 09: Verbindung zwischen PC und Mikrocontroller-Platine

Zur Programmierung von Mikrocontrollern wird spezielle Software – die so genannte Toolchain – benötigt. Für AVR-Chips gibt es im Netz zwei frei erhältliche Pakete, die über alle benötigten Komponenten verfügen. Zum Einen ist dies das AVR Studio, zum Anderen WinAVR. Die meisten Softwarekomponenten von WinAVR sind auch für Linux erhältlich.

Funktion	**Name**	**BS**	**Link**
Gesamtpaket	AVR Studio 4	Windows	[avrsw]
Gesamtpaket	WinAVR	Windows	[winavrsw]
Editor	Programmer's Notepad	Windows	[pnsw]
Editor	Emacs	Windows & Linux	[emacs]
C Bibliothek	avr-libc	Windows & Linux	[libcsw]
C Compiler	avr-gcc	Windows & Linux	[gccsw]
Debugger (Konsole)	avr-gdb	Windows & Linux	[gdbsw]
Debugger (mit GUI)	avr-insight	Windows & Linux	[inssw]
Debugger (mit GUI)	DDD (Data Display Debugger)	Windows & Linux	[dddsw]
Programmer (Konsole)	AVRDUDE	Windows & Linux	[dudesw]
Programmer (GUI)	AVRDUDE-GUI	Windows & Linux	[dudeguisw]
Programmer (GUI)	PonyProg	Windows & Linux	[ponysw]
Programmer (Konsole)	UISP	Windows & Linux	[uispsw]

Tab 01: Software

Das AVR Studio 4 ist eine Entwicklungsumgebung, die von der Firma ATMEL direkt zur Verfügung gestellt wird. Darin sind schon ein Debugger und ein Simulator enthalten. Will man den Mikrocontroller nur in Assembler programmieren, dann ist das *AVR Studio 4* die einzige Softwarekomponente, die benötigt wird. Soll der Chip in C programmiert werden, dann ist es auf jeden Fall notwendig, die *avr-libc* und den *avr-gcc* zusätzlich zu installieren. Da im Programmpaket WinAVR diese zwei Komponenten der Toolchain enthalten sind, ist es ratsam unter Windows sowohl das *AVR Studio 4* als auch die *WinAVR*-Entwicklungsumgebung zu installieren.
Wenn man lieber flexibel in der Wahl des Betriebssystems, unter dem programmiert wird, bleiben will, dann ist es natürlich empfehlenswert von Anfang an nur mit den *WinAVR* Toolchainkomponenten zu arbeiten, da diese auch größtenteils für Linux zur Verfügung stehen, und man sich dann nicht mit der Handhabung zweier verschiedener Entwicklungsumgebungen vertraut machen muss.

3.1 WinAVR (Version 2006/01/25)

WinAVR ist ein open-source Programm-Paket, das diverse Tools zur Verfügung stellt, um mit einem Mikrocontroller arbeiten zu können.
Hauptbestandteil ist der avr-gcc Compiler, der speziell an Atmel-Chips angepasste Syntax beinhaltet und zum Beispiel in die Entwicklungsumgebung des AVR Studios eingebunden werden sollte, falls man dort seinen Code in C programmiert.
Sämtliche in *WinAVR* vereinten Programme der Toolchain - außer *Programmer's Notepad* - sind auch für Linux verfügbar und werden genauso gehandhabt.

3.1.1 avr-libc

Hauptbestandteil von WinAVR ist die avr-libc. Das ist eine C-Programm-Bibliothek, die außer vielen gängigen C-Funktionen, auch Funktionen enthält die AVR-spezifisch und für die Programmierung von AVR-Chips essentiell sind.
Außerdem sind in der avr-libc alle Header-Files für alle unterstützten AVR-Mikrocontroller enthalten. Dadurch ist es möglich, nachdem man im Makefile (oder über AVR Studio) angegeben hat, welchen Chip man benutzt, auf Ports und Register über deren Namen aus dem jeweiligen Datenblatt zuzugreifen.
So wird zum Beispiel PORTA beim ATmega16 durch das Headerfile *../WinAVR/avr/include/avr/iom16.h* beim Kompiliervorgang in die Adresse _SFR_IO8(0x1B) übersetzt, wodurch der Mikrocontroller die assemblierte Anweisung, was er an diesem Port tun soll, erst versteht.
In älteren Versionen steht das Handbuch als HTML-Version zur Verfügung, seit *WinAVR* Version 2006/01/25 gibt es eine PDF-Version des Manuals. Diese umfasst 270 Seiten und stellt neben ausführlichen Erklärungen der AVR-spezifischen Funktionen auch viel Beispielcode und eine große FAQ-Sektion zur Verfügung.

3.1.2 Programmer's Notepad

Programmer's Notepad ist ein open-source Editor, der über eine Reihe von Features verfügt:

- Projektverwaltung und Syntaxcoloring für zahlreiche Programmiersprachen
- Möglichkeit des Einbindens von externen Tools
- Möglichkeit des Ein- und Ausklappens von Funktionen zwecks Übersichtlichkeit des Codes
- und vieles mehr...

Eine ausführliche Beschreibung findet man auf der *Programmer's Notepad* Homepage [pnsw].
Nachdem man sein Projekt angelegt und den Code geschrieben hat, ist es erforderlich ein Makefile zu schreiben, um den Code kompilieren zu können. Dazu mehr unter dem Punkt Makefiles.
Das Makefile wird im Projektordner gespeichert.
Unter dem Menü-Eintrag "Tools" klickt man dann auf *WinAVR Make All*. Dadurch wird das Makefile aufgerufen und der Code mittels avr-gcc kompiliert. Im Output-Fenster des *Programmer's Notepad* erscheinen anschließend die Ausgaben des Compilers. Außerdem wird aus dem Code eine .hex Datei erzeugt, die in den Chip übertragen werden kann. Ist der Code fehlerfrei (Process Exit Code: 0) und das Entwicklungsboard angeschlossen, wird *WinAVR Program* ausgewählt. Intern wird nun avrdude.exe aufgerufen - ein Tool, das in der Lage ist, das durch *WinAVR Make All* erstellte .hex File in den Chip zu übertragen. Der dritte Eintrag im Tools-Menü - *WinAVR Make Clean* - ruft rm.exe auf und löscht alle durch *WinAVR Make All* erzeugten Files. *WinAVR Make Clean* wird vor allem dann benötigt, wenn man einen "Rebuild All" ausführen möchte.
Um Tastenshortcuts für die oben beschriebenen Programmaufrufe zu setzen, geht man folgendermaßen vor:

- In der Menüleiste *Tools → Options* auswählen
- Im Options-Popup auf den Menüeintrag *Tools* klicken
- Dann im dazugehörigen Dropdown-Menü oben in der Mitte
  (*None - Global Tools*):
  - Das Tool anklicken, zu dem man den Shortcut will und dann auf *add*
  - Im daraufhin erscheinenden Popup kann man unter *Properties* den Shortcut setzen.

3.1.3 Makefiles

Das Shell-Kommando make benötigt zur Ausführung ein Makefile, in dem steht, welche Dateien kompiliert und gelinkt werden sollen. Das Makefile muss bestimmte Komponenten in einer festgelegten Reihenfolge enthalten, damit es ausgeführt werden kann. [make]

Ziel - Bedingung - Kommando

Ziel
ist der Name der Datei, die erzeugt werden soll, oder ein Befehl der ausgeführt werden soll
Bedingung
ist der Name der Datei(en), die zur Erzeugung des Ziels benötigt wird
Kommando
ist die Aktion, die make ausführen soll.

Es können auch mehrere Kommandos hintereinander aufgerufen werden, wobei allerdings zu beachten ist, dass jedes Kommando in einer eigenen Zeile stehen muss, und diese Kommandozeilen jeweils mit einem TAB beginnen.

➔ Erzeugen eines Object-Files

```
main.o : main.c main.h
	cc -c main.c
```

➔ Erzeugen der dazugehörigen exe

```
main : main.o
	cc -o main main.o
```

3.1.4 mFile

mFile ist ein kleines Programm, welches es Einsteigern ermöglicht, schnell und einfach Makefiles selbst zu erstellen. Es stellt das Gerüst eines Makefiles zur Verfügung, in das der Benutzer nur noch die relevanten Daten anhand einer Liste im Menü einfügen muss.

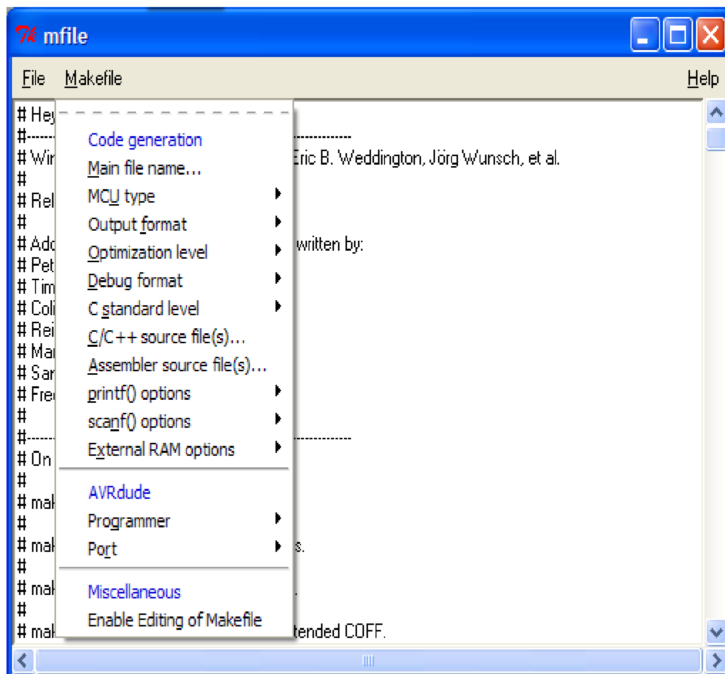


Abb. 10: mFile

1. Makefile/Main file name
   ➔ Name des c-Files, das die Hauptfunktion enthält
2. MCU type
   ➔ Chip Bezeichnung (ATmega -> atmega16)
3. Output format
   ➔ File-Typ der in den Chip übertragen werden soll (hex)
4. Optimization level
   ➔ Optimierung bei der Kompilierung (siehe avr-libc UserManual S.237)

5. Debug format
   ➔ Auswahl des Debug-Fileformats, das durch den Compiler erzeugt werden soll.
6. C standard level
   ➔ Welcher C-Standard soll verwendet werden.
7. C/C++ source file(s)
   ➔ Source Files, die neben dem Haupt-Source-File mitkompiliert werden sollen
8. Assembler source file(s)
   ➔ Assembler Files, die neben dem Haupt-Source-File mitkompiliert werden sollen
   Der Pfad wird automatisch komplett übernommen. Leider kommt make nicht
   mit Backslashes zurecht, weshalb der Pfad, wenn er nicht benötigt wird
   eher ganz weggelassen werden sollte oder es werden alle Backslashes in Slashes
   umformatiert.
9. printf() und scanf() options
   ➔ Welche Bibliothek wird für printf()/ scanf()-Befehle im Code benötigt
   ➔ Zur Ausgabe/ Zum Einlesen von floats wird eine zusätzliche Library benötigt, das
   muss dann im Makefile vermerkt sein
10. External RAM options
    ➔ Wichtig, wenn man nicht nur den RAM im Chip verwendet. Näheres dazu steht in der
    Hilfe von mFile
11. Programmer
    ➔ Auswahl des ISP-Adapters
12. Port
    ➔ COM- oder LPT-Port an den die Platine angeschlossen ist.

Nachdem das Makefile komplett ist, wird es in das Verzeichnis gespeichert, in dem auch die dazugehörigen Files liegen.

3.1.5 AVRDude

AVRDude ist die Komponente der Toolchain, die es ermöglicht über den ISP den Chip zu programmieren und zu löschen und auch die Fuse- und Lockbits zu setzen und zu verändern. Man hat bei diesem Programm die Möglichkeit, sowohl über die Konsole als auch über eine grafische Oberfläche zu agieren.
Im Feld *-p Device* wird der Mikrocontrollertyp ausgewählt. Im Feld *-c Programmer* wird bei einem seriellen ISP das *STK 500* und bei einem Parallelport Programmieradapter STK 200 aktiviert. Anschließend wird der entsprechende COM- oder LPT-Port festgelegt. Der Pfad der .hex Datei, die in den Mikrocontroller programmiert werden soll, wird im Editfeld *Flash* eingetragen.
In der aktuellen Firmware des seriellen ISP ist das Kommunikationsprotokoll ein anderes als in älteren Firmware-Versionen, weshalb in der *Command Line* das stk500 in stk500v2 umgeändert werden muss (das ist Feld ist editierbar), da in der Version 0.2.0 von *avrdude gui* dieser Programmer leider nicht im oberen Dropdown Menü *-c programmer* vorhanden ist.

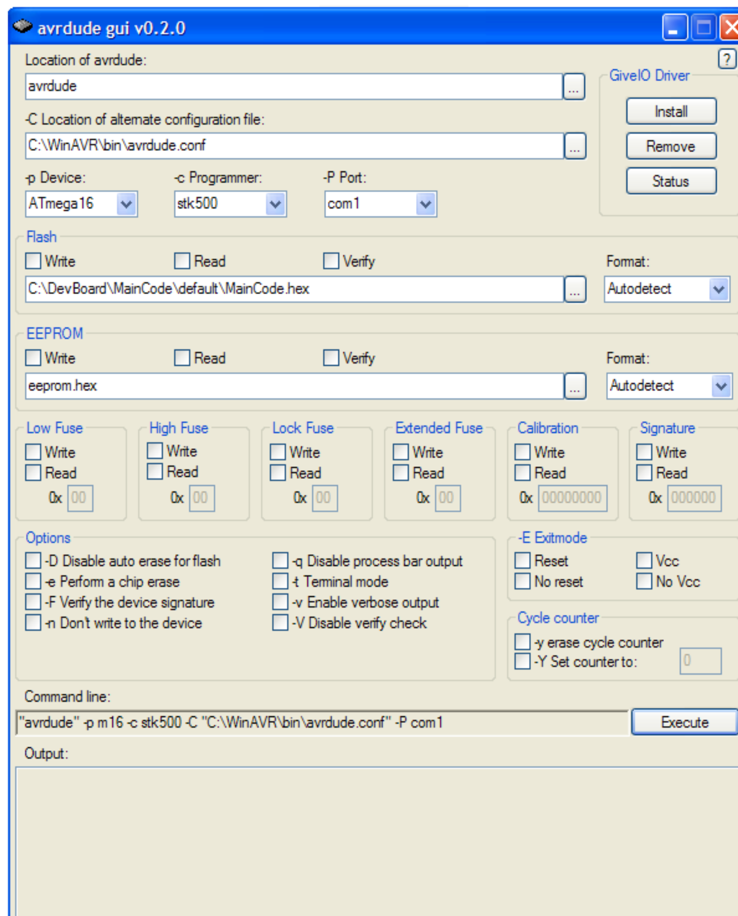


Abb. 11: AVRDude-GUI

3.1.6 AVaRICE

AVaRICE ist ein Konsolenprogramm, dass benötigt wird, um den Chip mit dem AVR-GDB zu debuggen. Über einen TCP-Socket wird die Verbindung zwischen AVaRice und GDB hergestellt (daher kann der Chip dann auch von einem anderen Rechner aus debugged werden). Das Konsolenprogramm fungiert dann als Übersetzer zwischen AVR JTAG ICE und dem avr-gdb.

Start von AVaRICE:

- ➔ Konsole öffnen
- ➔ avarice -j (oder --jtag) /dev/comport :port   z.B. avarice -j /dev/com1 :1024

```
C:\>avarice -j /dev/com1 :1024
AVaRICE version 2.4, Jan 23 2006 12:31:17

Defaulting JTAG bitrate to 1 MHz. Make sure that the target
frequency is at least 4 MHz or you will likely encounter failures
controlling the target.

JTAG config starting.
Hardware Version: 0xc1
Software Version: 0x7f
Reported JTAG device ID: 0x9403
Configured for device ID: 0x9403 atmega16
JTAG config complete.
Preparing the target device for On Chip Debugging.

Disabling lock bits:
  LockBits -> 0xff

Enabling on-chip debugging:
  Extended Fuse byte -> 0xcd
      High Fuse byte -> 0x19
       Low Fuse byte -> 0xcd
Waiting for connection on port 1024.
```

Abb. 12: AVaRICE

3.1.7 avr-gdb

Nun da AVaRICE aktiv ist, wird der avr-gdb gestartet. Als erstes wird dem avr-gdb der Dateiname des zu debuggenden Files übergeben:
cd Verzeichnis des elf-Files (das elf-File wurde beim Kompilieren erstellt)

<gdb> file file.elf

Dann wird die Verbindung zwischen AVaRICE und dem avr-gdb hergestellt. Dazu gibt man beim avr-gdb folgendes ein:

<gdb> remote localhost:port

z.B.: <gdb> remote localhost:1024
Sämtliche Befehle zum Debuggen und Tracen finden sich in der gdb-Hilfe

3.1.8 AVR Insight

AVR Insight ist das grafische Interface für den avr-gdb. Hier muss auch vorher AVaRICE gestartet werden, um den Chip debuggen zu können.
Nach dem Start, gilt es zuerst das zu debuggende File auszuwählen.

➔ File/Open (.elf Datei)

Der komplette Code erscheint dann im Hauptfenster. Anschließend muss die Verbindung zu AVaRICE hergestellt werden. Hierzu wählt man unter *File - Target Settings* aus und gibt im Popup-Fenster den Hostnamen (wenn es der gleiche Rechner ist, dann localhost) und den Port (derselbe, wie bei AVaRICE) an. Außerdem ist es ratsam *Set breakpoint at 'main'* gecheckt zu lassen.
Jetzt wird die Verbindung über *Run - Connect to Target* erzeugt. Es erscheint dann ein Popup, das die Nachricht *"Successfully connected"* enthält. Nun können im Code Breakpoints gesetzt werden und zwar nur an den Stellen, wo vor der Zeilennummer ein '-' steht. Dazu einfach einmal auf das '-' klicken.
Unter dem Menüpunkt *View* können Fenster mit jeglichen zur Verfügung stehenden Informationen über Registerinhalte, Speicherinhalte usw. ausgewählt werden.
Außerdem ist es möglich rechts oben auszuwählen, ob der Code in C, in Assembler oder gemischt angezeigt werden soll.

3.2 AVR Studio 4.12

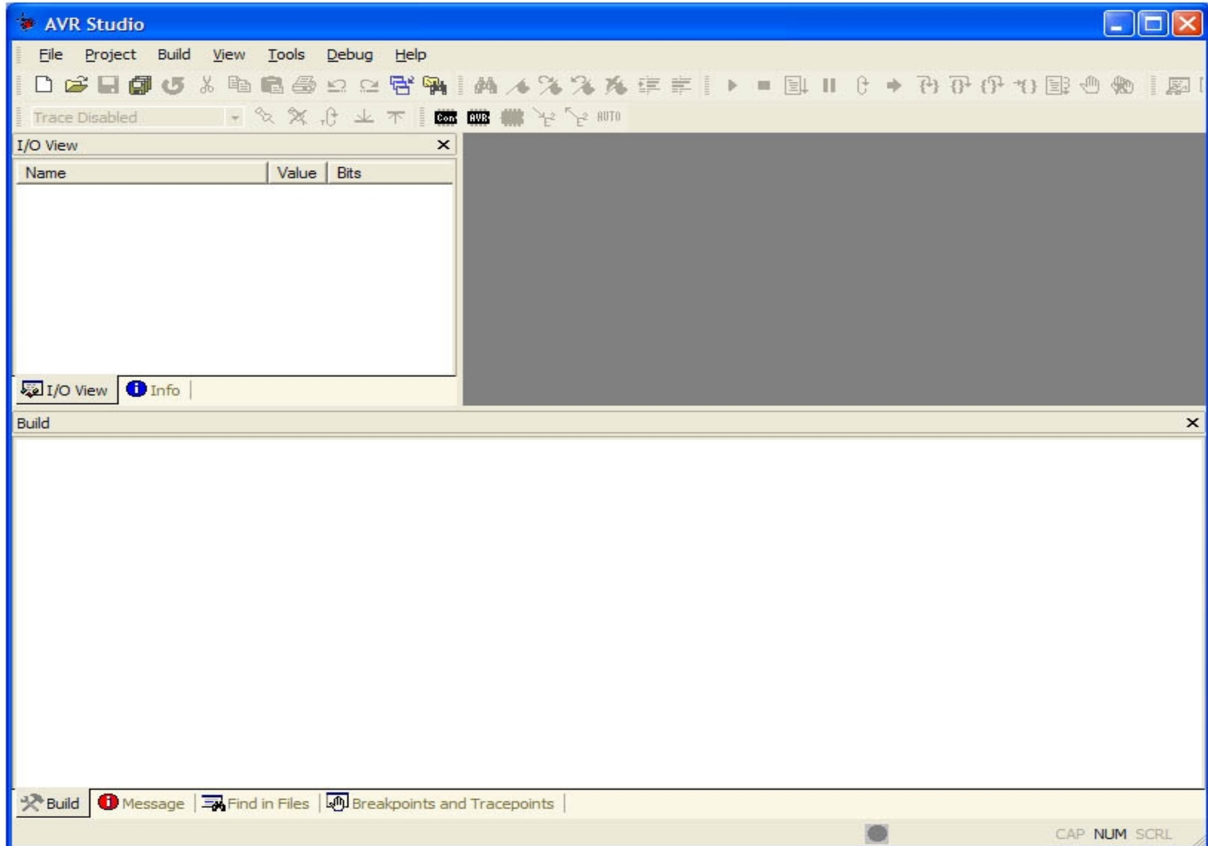


Abb. 13: AVR Studio 4.12

Das Entwickeln, Testen und Debuggen von Code mit Hilfe des AVR Studios gestaltet sich gerade für Einsteiger recht einfach, da in diesem Paket alle Komponenten vereinigt sind, die zur Handhabung eines AVR-Mikrocontrollers nötig sind und man sich nicht erst mit verschiedenen Programmen für die einzelnen Schritte auseinandersetzen muss.

Die neueste Version, das AVR Studio 4.12 beinhaltet eine Projektverwaltung, einen Editor, in den man den GCC-Compiler einbinden kann und einen Debugger.

Außerdem enthält das Paket einen Simulator, der alle Funktionen der AVR-Chips unterstützt. So ist es möglich, den selbstgeschriebenen Code mit Hilfe des Debuggers zuerst einmal im Simulator zu testen, und so vor dem Einspielen in den Chip auf logische Fehler zu überprüfen. Es ist äußerst ratsam dieses Feature zu nutzen, da es bei der Programmierung von Mikrocontrollern durchaus möglich ist, Hardware-Komponenten auf der Platine, oder sogar den Chip selbst, durch fehlerhaften Code zu beschädigen.

Die Installationsdatei des AVR Studios 4.12 findet sich im Verzeichnis *Software/AVR Studio4.12/aStudio4b460.exe* auf der CD. Nach deren Ausführung muss anschließend das SP *aStudio412SP2b472.exe*, das sich im gleichen Verzeichnis befindet, installiert werden.

3.2.1 Projekt anlegen

Hier wird ein Neues Projekt angelegt oder, falls vorhanden, ein aktuelles aus der Liste gewählt.

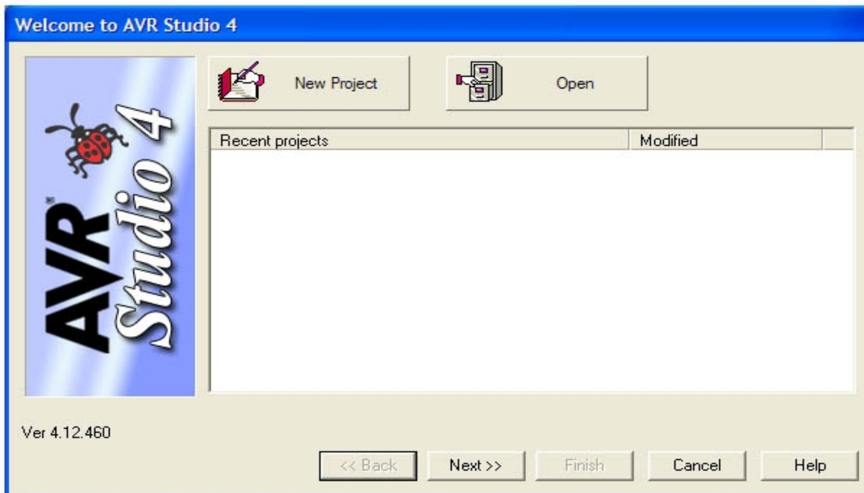


Abb. 14: AVR Studio 4.12 - Projekt anlegen

Wenn in Assembler programmiert werden soll, dann wird der programmeigene Assemblierer verwendet, wenn der Code in C geschrieben wird, wird der *avr-gcc* eingebunden. (Voraussetzung dafür ist die installierte WinAVR Toolchain.)

Abb. 15: AVR Studio 4.12 - Sprache wählen

Das AVR Studio bietet die Möglichkeit, den Code vorab im Simulator zu testen und zu debuggen. Als Device ist der verwendete Mikrocontroller anzugeben.

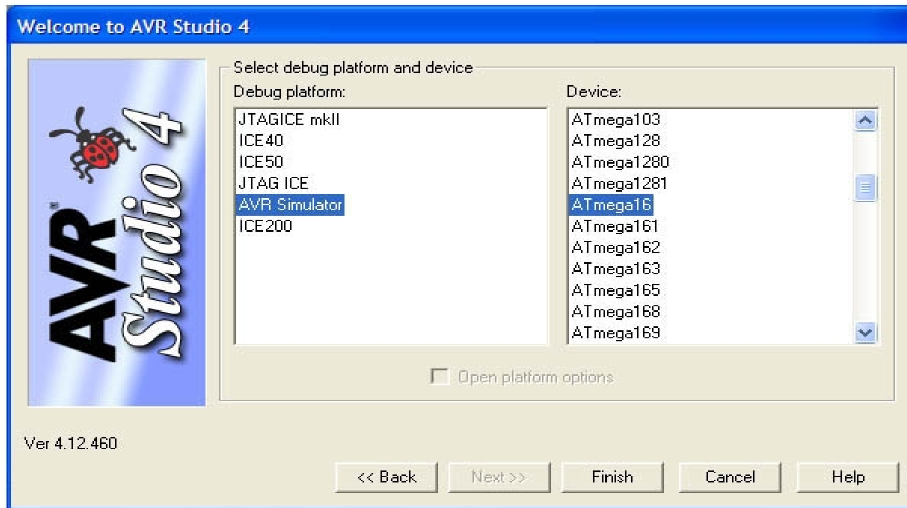


Abb. 16: AVR Studio 4.12 - Debugging über Simulator

JTAGICE ist der Debugger auf der, an das Entwicklungsboard angeschlossenen, Platine. Falls beim Anlegen des Projektes der Simulator als Debugger ausgewählt wurde, kann später im Menü *Debug/Select Platform and Device* JTAG ICE im Nachhinein ausgewählt werden. Als Device ist der verwendete Mikrocontroller anzugeben.

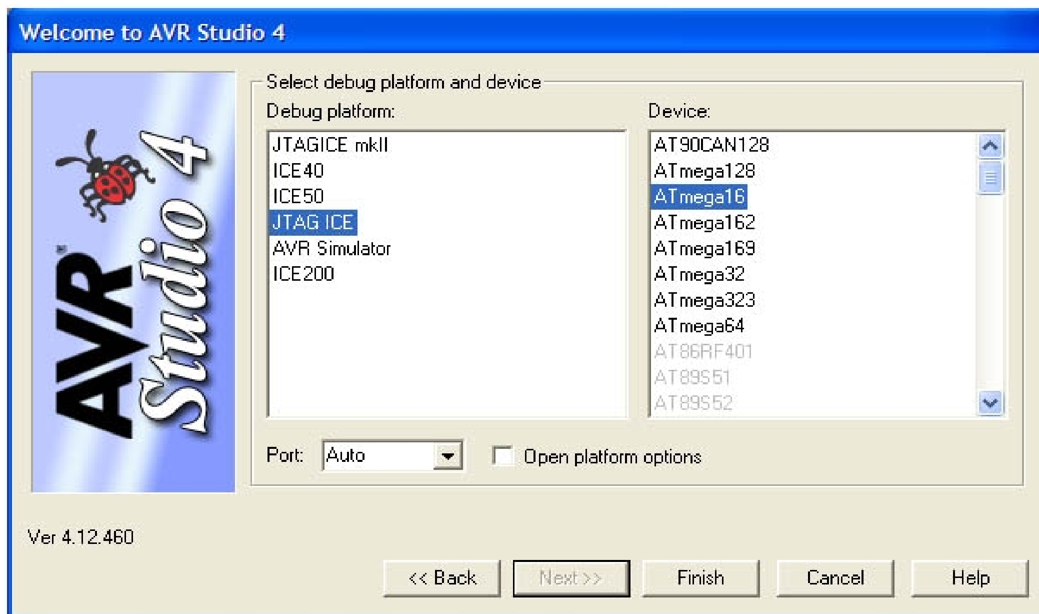


Abb. 17: AVR Studio 4.12 - Debugging über JTAG

3.2.2 Programmieren des Chips

Es gibt zwei unterschiedliche Wege, den geschriebenen Code in den Mikrocontroller zu schreiben. Zum Einen kann man den ISP-Teil, zum Anderen den JTAG-Teil der Evertool-Platine als AVR-Programmer verwenden.
Will man seinen Code vorerst mit dem Simulator debuggen, und kennt außerdem den COM-Port nicht namentlich, an den die Platine angeschlossen ist, empfiehlt es sich, über den *"CON"*-Button die Einstellungen *"STK500 or AVRISP"* als Quelle für den ISP und *"Auto"* für die Com-Port-Suche zu aktivieren.

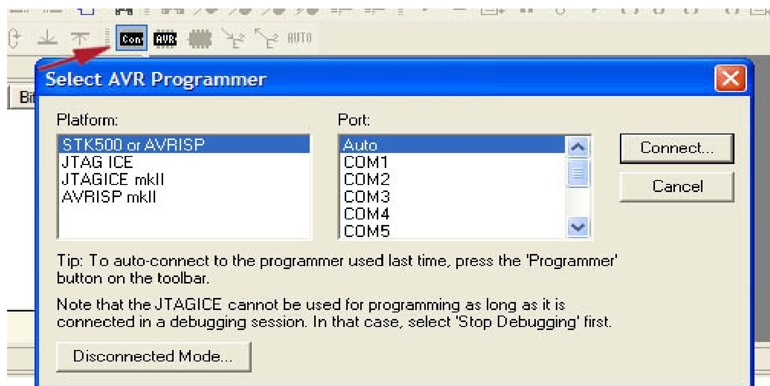


Abb. 18: AVR Studio 4.12 - Auswahl des Programmers: ISP

Wenn der Code direkt im Chip über JTAG ICE debugged werden soll, dann wird als ISP *"JTAG ICE"* und der entsprechende COM-Port (oder "Auto") gewählt. Bei dieser Einstellung wird dann, sobald man "Start Debugging" unter dem Menü-Punkt "Debug" auswählt, der Code in den Chip gespielt.

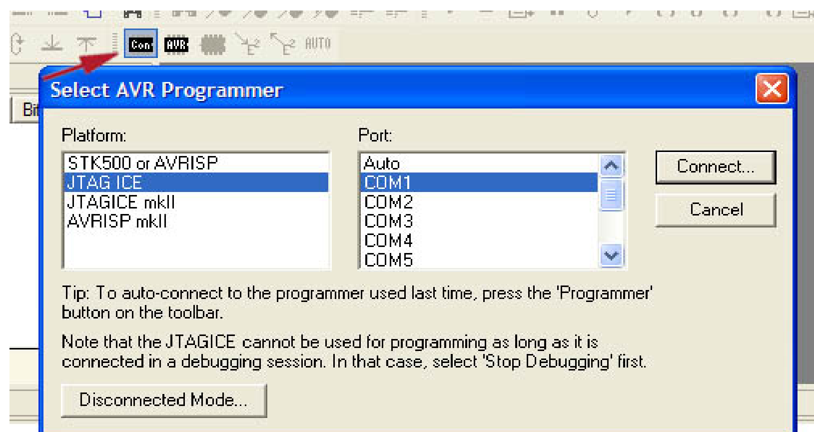


Abb. 19: AVR Studio 4.12 - Auswahl des Programmers: JTAG

Nachdem die Verbindung zwischen ISP und Mikrocontroller hergestellt wurde, wird über den Button *"AVR"* dieser Dialog aufgerufen. Im Reiter *Program* kann der Mikrocontroller gelöscht und neu bespielt werden. Leider wird der Pfad bei *Flash - Input HEX File* nicht aktualisiert, wenn während des Betriebs des AVR Studios 4.12 das aktuelle Projekt geschlossen und ein anderes Projekt geöffnet wird. Daher sollte immer überprüft werden, ob der Pfad dem gewünschten HEX-File entspricht.

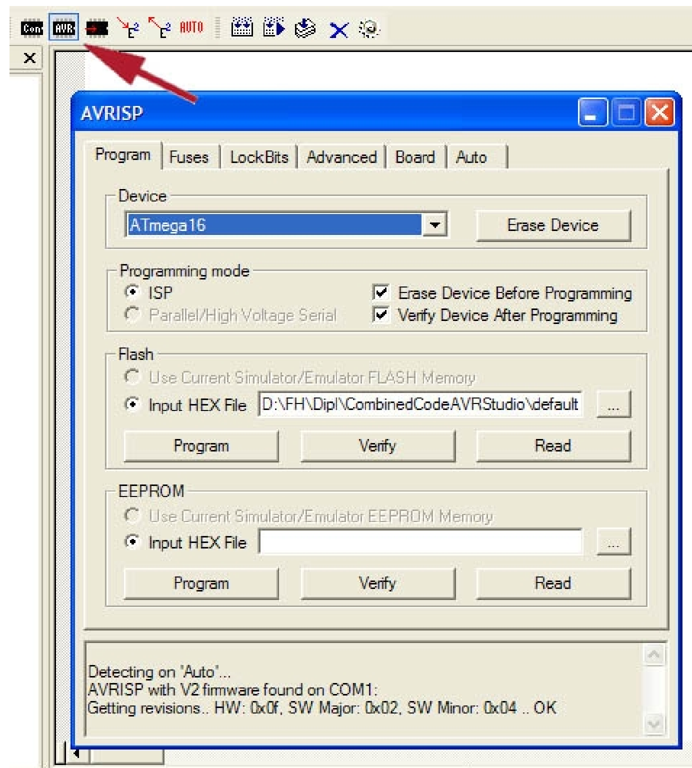


Abb. 20: AVR Studio 4.12 - ISP Dialog

3.2.3 AVR Studio 4.12 Simulator

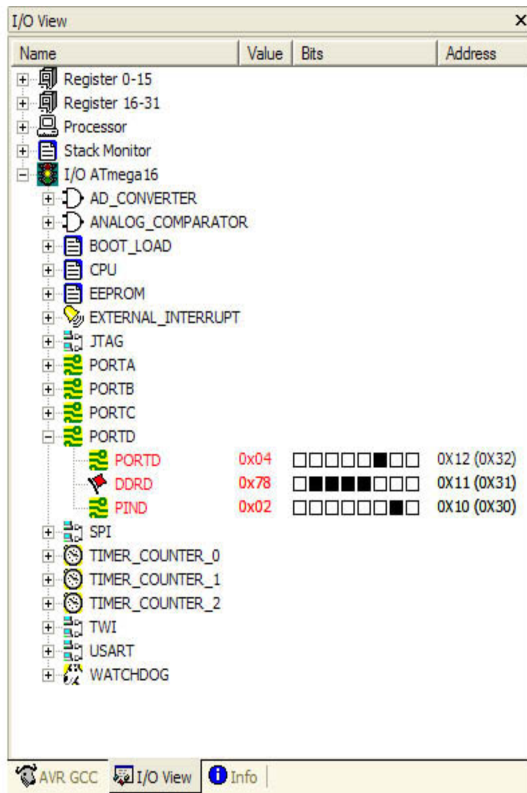


Abb. 21: AVR Studio 4.12 - Simulator

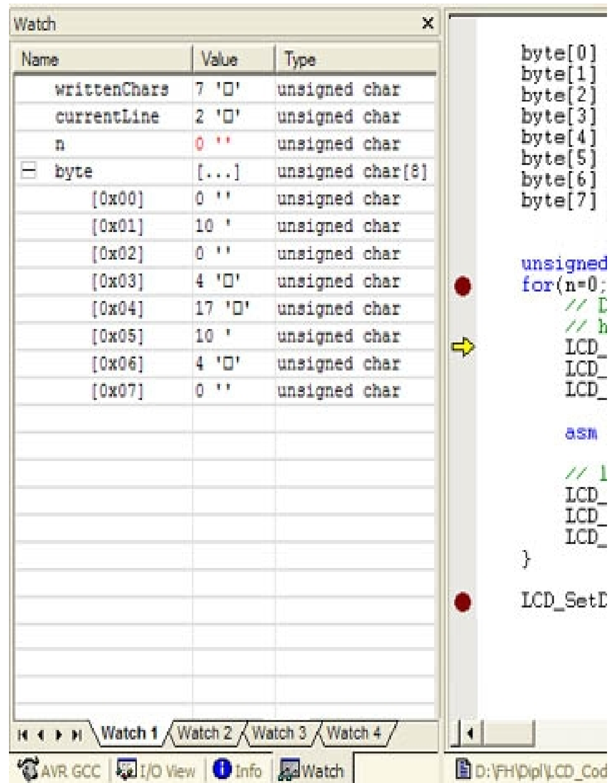


Abb. 22: AVR Studio 4.12 - Watch-Fenster

Der Simulator des AVR Studios bietet die Möglichkeit, den Code vor dem Einspielen in den Chip zu debuggen und zu testen. Alle Funktionen des Mikrocontrollers können simuliert und entsprechend debugged werden. Mit Hilfe des Watch-Fensters können Variablenwerte verfolgt werden (rechter Mausklick auf Variable *Add Watch: "dummy"* fügt die Variable dem Watch-Fenster hinzu).

3.2.4 Fusebits und Lockbits

Wie in der Einleitung angesprochen, verfügen Mikrocontroller über so genannte Fusebits, die für die Chipeinstellungen zuständig sind, und Lockbits, mit denen man zum Beispiel ein weiteres Programmieren des Chips verhindern kann. In diesem Abschnitt soll nun näher auf die Funktion dieser Bits und deren Handhabung eingegangen werden, da unbedachte Veränderungen der Fuses und Locks unerwünschte Auswirkungen haben können.
Im schlimmsten Fall ist der Mikrocontroller danach nicht mehr ansteuerbar.

3.2.4.1 Fusebits

Der ATmega16 verfügt über 16 Fusebits - also 2 Fusebytes.
Es ist darauf zu achten, dass aktivierte Fusebits den Wert 0 haben, und deaktivierte den Wert 1.

7	6	5	4	3	2	1	0
OCDEN	JTAGEN	SPIEN	CKOPT	EESAVE	BOOTSZ1	BOOTSZ2	BOOTRST

Abb. 23: Fusebits - High Byte

OCDEN: On Chip Debugging aktivieren

In der Voreinstellung ist dieses Fusebit nicht gesetzt, also 1. Es muss aktiviert werden, wenn über die JTAG Schnittstelle debugged werden soll.

JTAGEN: JTAG aktiviert

In der Voreinstellung ist dieses Fusebit gesetzt, also 0.
Die JTAG-Schnittstelle belegt 4 Bits des Mikrocontrollers (TDI/PC5, TDO/PC4, TMS/PC3, TCK/PC2). Benötigt man keine JTAG-Schnittstelle, da man zum Beispiel den Mikrocontroller nicht debuggen will, deaktiviert man JTAGEN und hat dann am Chip 4 zusätzliche I/O-Pins zur Verfügung.

SPIEN: Programmierung über SPI aktiviert

In der Voreinstellung ist dieses Fusebit gesetzt.
Wird dieses Fusebit auf 1 gesetzt (deaktiviert) dann lässt sich der Chip nicht mehr über den ISP programmieren sondern nur noch über einen Adapter der den Mikrocontroller nicht seriell sondern parallel programmiert (z.B. ATMEL STK500).
Dieses Bit kann unter AVR Studio nicht verändert werden, wenn man den Chip seriell mit einem ISP programmiert.

CKOPT: Verstärkung der Oszillator-Frequenz

Wird der Quarz nicht direkt neben dem µC platziert sondern etwas weiter entfernt, dann kann es sein, dass das Signal der Frequenz zu schwach am Chip ankommt. Tritt dieser Fall auf, kann man das Signal über dieses Fusebit verstärken, wodurch allerdings der Stromverbrauch erhöht wird.

EESAVE: EEPROM wird beim Chip-Erase nicht beeinflusst

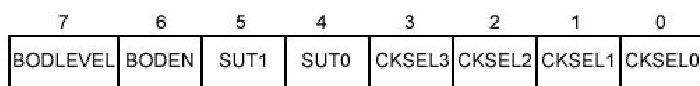
In der Voreinstellung ist dieses Fusebit nicht gesetzt. Das heißt, dass bei jedem Chip-Erase das EEPROM auch mit gelöscht wird. Hat man Daten darin gespeichert, die auch über einen Chip-Erase hinweg erhalten werden sollen, oder greift man gar nicht auf das EEPROM zu, sollte dieses Bit gesetzt ("0") werden.

BOOTSZ1 und BOOTSZ0: Größe des Bootloader[5]-Bereichs

Die Größe des Bootloaderbereichs liegt zwischen 128 und 1024 Words und kann über diese beiden Fusebits festgelegt werden. Da in der Voreinstellung diese beiden Bits gesetzt sind, liegt die Größe des Bootloaderbereichs bei 1024 Words.

BOOTRST: Auswahl des Resetvektors

In der Voreinstellung ist dieses Fusebit nicht gesetzt. Führt man am Chip einen Reset durch, dann liegt die Wiedereinsprungadresse normalerweise bei 0x00. Wird dieses Fusebit gesetzt, dann startet die Ausführung nicht an dieser Adresse sondern im Bootloader-Bereich.



7	6	5	4	3	2	1	0
BODLEVEL	BODEN	SUT1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0

Abb. 24: Fusebits - Low Byte

BODLEVEL: auslösendes Spannungslevel für den Brown-Out[6]-Detector

In der Voreinstellung ist dieses Fusebit nicht gesetzt, das heißt der Reset wird bei 2,7V ausgelöst. Wird das Bit auf 0 gesetzt, also aktiviert, dann findet der Reset bei 4V statt.

BODEN: Brown-Out ist aktiviert

In der Voreinstellung ist dieses Fusebit nicht gesetzt.

Das Bit sollte aktiviert werden, wenn das EEPROM verwendet wird, da dieses ohne Brown-Out bei starken Spannungsschwankungen unter Umständen gelöscht werden kann.

SUT1 und SUT0: Vorlaufzeit des Mikroprozessors

In der Voreinstellung ist SUT1 nicht gesetzt und SUT0 gesetzt, wodurch eine maximale Vorlaufzeit von 65ms gewährleistet ist.

CKSEL0 bis CKSEL3: Clock Select

Über diese 4 Fusebits wird die Taktquelle bestimmt.

Die Fusebits können im AVR Studio über *"Connect to AVR Programmer"* im Reiter *Fuses* verändert werden.

3.2.4.2 Lockbits

Der ATmega16 stellt 6 Lockbits zur Verfügung, die, wie auch die Fusebits, bei 0 gesetzt und bei 1 nicht gesetzt sind. Lockbits finden dann Verwendung, wenn der Chip fertig programmiert ist, und sein Inhalt danach nicht mehr verändert werden darf. Werden diese Bits gesetzt, ist kein Schreibzugriff auf den Chip mehr möglich. Die 6 Lockbits sind in 3 Gruppen aufgeteilt. Im Auslieferungszustand sind alle Bits auf 1 gesetzt und damit nicht aktiviert.

---

5 Bootloader: Lädt das eigentliche Programm in den Flash

6 Brown-Out: Überwachungsschaltung, die eine zu geringe Versorgungsspannung erkennt und dann einen Reset ausführt

LB2-LB1: Lockbit Mode

Wird LB2 gesetzt (0), dann ist es nicht mehr möglich den Chip zu programmieren.

Werden beide Bits gesetzt, dann kann auf den Mikrocontroller nicht mehr schreibend zugegriffen werden, und er kann auch nicht mehr verifiziert werden.

BLB02-BLB01: Boot0 Lockbit Mode

Diese Bits betreffen den Zugriff über die Assemblerbefehle

SPM (Store Program Memory) und LPM (Load Program Memory) auf den Speicherbereich des Flash in dem das Anwendungsprogramm liegt.

BLB12-BLB11: Boot1 Lockbit Mode

Diese Bits betreffen den Zugriff über die Assemblerbefehle

SPM (Store Program Memory) und LPM (Load Program Memory) auf den Speicherbereich des Flash in dem der Bootloader liegt.

Die Lockbits können im AVR Studio über *"Connect to AVR Programmer"* im Reiter *LockBits* verändert werden.

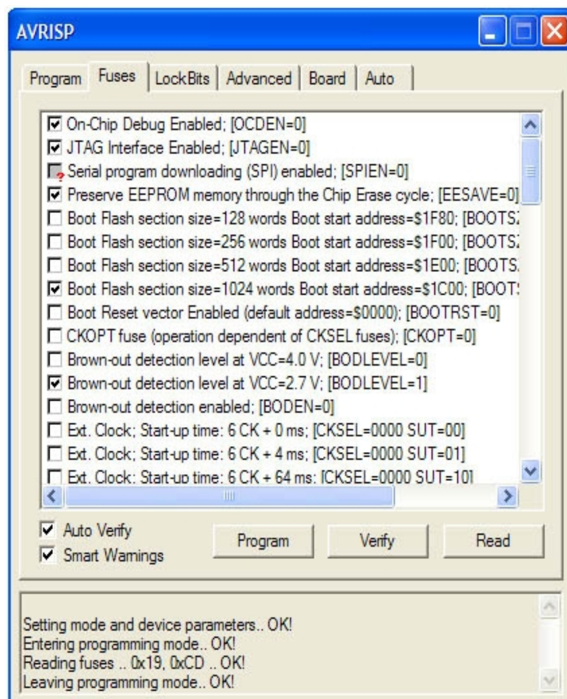


Abb. 25: AVR Studio 4.12 Fusebits

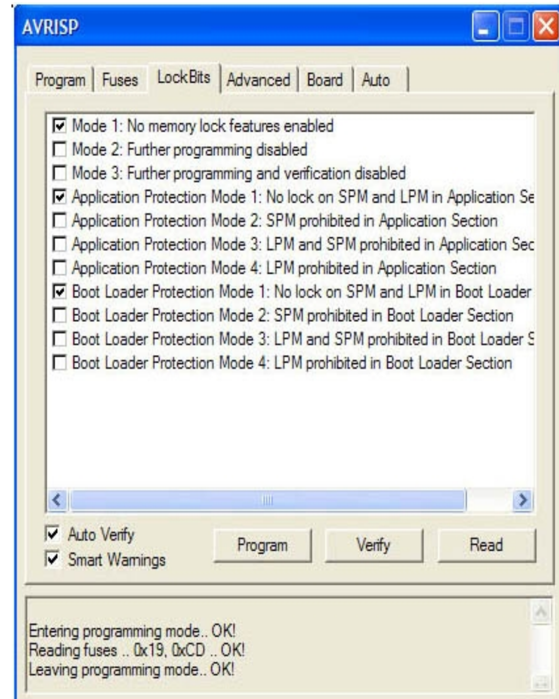


Abb. 26: AVR Studio 4.12 Lockbits

4 Programmierung des Mikrocontrollers

Dieses Kapitel widmet sich der Programmierung des ATmega16. Es werden die wichtigsten Chip-Funktionalitäten und deren Bedienung besprochen und außerdem die Ansteuerung von externen Bauteilen auf der Platine, wie zum Beispiel das LCD oder die Echtzeituhr. Durch Codebeispiele und Register-erklärungen sollen die auftretenden Probleme reduziert und die Inbetriebnahme von Hardwarekomponenten erleichtert werden.

4.1 Grundlagen

4.1.1 Programmablauf

Grundsätzlich gibt es zwei Arten, um Mikrocontroller-Programme zu schreiben.
Beim sequentiellen Programmablauf wird der auszuführende Code in eine Endlosschleife gepackt und so immer wieder ausgeführt, bis der Chip entweder keinen Strom mehr bekommt oder ein Reset ausgelöst wird.
Beim interruptgesteuerten Programmablauf werden beim Start des Programms alle Interruptquellen aktiviert und erst dann folgt eine Endlosschleife, in der nicht-zeitkritischer Code abgearbeitet wird. Erfolgt ein Interrupt, wird zur entsprechenden Interrupt-Routine gesprungen und der dortige Code ausgeführt.

```
//////////////////////////
// Sequentieller Ablauf
int main (void){
   //Init
   //Endlosschleife
   while(1){
        //...do something
   }
}
```

```
/////////////////////////////////
// Interruptgesteuerter Ablauf
          #include <avr/interrupt.h>
          ISR(vectorname)
   {
       // do something
   }
          int main (void){
              //Init
              .......
              //Endlosschleife
              while(1){
                    // do something
                    // ISR-Auslöser
             }
        }
```

4.1.2 Portregister

DDRx = Datenrichtungsregister für Port x

Alle I/O Pins des Mikrocontrollers sind Bidirektional, d.h. sie können auf zwei verschiedene Artenbetrieben werden:
Ein Pin kann als Ausgang geschaltet sein, wenn das entsprechende Bit auf *1 -> high* gesetzt ist und
ein Pin kann als Eingang geschaltet sein, wenn das entsprechende Bit auf *0 -> low* gesetzt ist.

Beispiel:

PINS 1,2,4 und 7 des Ports A werden explizit als Ausgänge gesetzt, die restlichen Pins automatisch als Eingänge.

(Die folgenden Ausdrücke sind gleichwertig und unterscheiden sich nur in der Notation)

```
DDRA |= (1 << DDA1) | (1 << DDA2) | (1 << DDA4) | (1 << DDA7);
DDRA |= (1 << PIN1) | (1 << PIN2) | (1 << PIN4) | (1 << PIN7);
DDRA |= (1 << PA1) | (1 << PA2) | (1 << PA4) | (1 << PA7);
DDRA |= (1 << 1) | (1 << 2) | (1 << 4) | (1 << 7);
```

Beispiel:

PINS 0,3,5 und 6 des Ports A werden explizit als Eingänge gesetzt, die restlichen Pins automatisch als Ausgänge.

(Die folgenden Ausdrücke sind gleichwertig und unterscheiden sich nur in der Notation)

```
DDRA &= ~( (1 << DDA0) | (1 << DDA3) | (1 << DDA5) | (1 << DDA6));
DDRA &= ~( (1 << PIN0) | (1 << PIN3) | (1 << PIN5) | (1 << PIN6));
DDRA &= ~( (1 << PA0) | (1 << PA3) | (1 << PA5) | (1 << PA6));
DDRA &= ~( (1 << 0) | (1 << 3) | (1 << 5) | (1 << 6));
```

Es ist auch möglich einen Ausgang so zu setzen:

```
DDRA = (1 << PIN3);
```

Allerdings werden bei dieser Notation alle Pins, außer Pin 3, als Eingänge geschaltet. Hat man vorher noch andere Ausgänge definiert, werden diese durch obigen Code neu gesetzt, und zwar als Eingänge. Ist das im darauf folgenden Code nicht gewollt, treten bei der Ausführung Probleme auf.

PORTx = Datenregister für Port x

Das Datenregister eines Ports wird verwendet, um dessen Ausgänge anzusteuern, oder um interne Pull-Up Widerstände[7] an Eingängen zuzuschalten. Wurde ein Pin des Ports mittels DDRX-Manipulation als Ausgang definiert, kann nun durch PORTX die anliegende Spannung am Pin durch 1 auf "high" und durch 0 auf "low" gesetzt werden.

Beispiel:

(Die folgenden Ausdrücke sind gleichwertig und unterscheiden sich nur in der Notation)

```
PORTA |= (1 << PIN3);
PORTA |= (1 << PA3);
PORTA |= (1 << 3);
```

PINx = Register der Eingangsadresse für PORT x

Dieses Register dient zur Abfrage des Signalzustands des Ports x, d.h. wenn an einem als Eingang definierten Pin Spannung anliegt, dann wird "1" zurückgegeben, wenn keine Spannung anliegt, wird "0" zurückgegeben.

```
uint8_t zustandPA;
.....
zustandPA = PINA;
```

7 Pull-Up Widerstand: vgl. Kapitel 4.1.6

4.1.3 Codebeispiele zur Bitmanipulation

Die folgenden Codezeilen dienen dazu, zu veranschaulichen, wie viele unterschiedliche Arten es gibt, ein Portregister mit dem gleichen Wert zu befüllen. Die gängigsten Syntax sind dabei rot hervorgehoben. Da die Ausdrücke äquivalent sind, bleibt es dem Programmierer überlassen, welchen Stil er bevorzugt. In den Beispielen gilt es jeweils die Pins 6, 5, 4 und 3 im Datenrichtungsregister des Ports D als Ausgang zu setzen.

Übergabe eines expliziten Werts

```
// HEX-Wert
DDRD = 0x78;
// Dezimal-Wert
DDRD = 120;
// Binär-Wert
DDRD = 0b01111000;
```

Festlegen des Bitmusters durch Bitshifts

```
// Einzelne Pins werden explizit als Ausgang gesetzt
DDRD = (1<<PD6) | (1<<PD5) | (1<<PD4) | (1<<PD3);
// 120 als Bitmuster 1111000 wird ab Bit 0 in das Register geshiftet
DDRD = (120<<0);
// 30 als Bitmuster 11110 wird ab Bit 2 in das Register geshiftet
DDRD = (30<<2);
```

UND-Verknüpfung - bestimmte Bits löschen, ohne andere zu verändern

```
// Alle Pins von PORTD werden als Ausgänge gesetzt,
// anschließend werden die nicht benötigten Pins als Eingänge gesetzt  - also gelöscht
//   11111111           --> Initialisierung
// & 10000111           --> UND-Verknüpfung
// &~01111000           --> invertierte UND-Verknüpfung
// = 01111000           --> Ergebnis
DDRD = 0xff;
DDRD &= ~((1<<PD7) | (1<<PD2) | (1<<PD1) | (1<<PD0));
// Wie oben, nur dass der invertiere-UND-Verknüpfte Wert als HEX
// dargestellt wird
DDRD = 0xff;
DDRD &= ~0x87;
```

ODER-Verknüpfung --> bestimmte Bits setzen, ohne andere Bits zu verändern

```
// PIN 5 wird als Ausgang gesetzt und alle anderen Pins behalten
// ihren Zustand
DDRD = 0x58;  // -> 01011000
DDRD |= (1<<PD5); // -> 01111000

// Hier muss aufgepasst werden, da keine ODER-Verknüpfung stattfindet
// Das Register erhält im 1. Schritt einen expliziten Wert.
// Dieser wird von einem 2. expliziten Wert überschrieben
// der nur PIN5 als Ausgang setzt
DDRD = 0x58; // -> 01011000
DDRD = (1<<PD5); // -> 00100000
```

4.1.4 Ansteuerung von LEDs

Codebeispiel: CD-Verzeichnis Code/LED

Das Ansteuern von Bauteilen ist abhängig von ihrem Anschluss auf der Platine. Zur Verdeutlichung werden hier die Anschlussmöglichkeiten einer LED dargestellt, und in Abhängigkeit davon der dazugehörige Code. Voraussetzung für die dargestellten Codeausschnitte ist die Initialisierung des Pins im Datenrichtungsregister des Ports als Ausgang.

4.1.4.1 Active Low

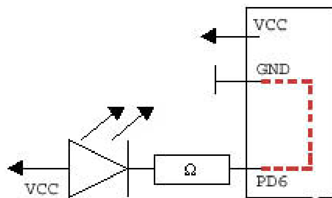


Abb. 27: LED als Active Low angeschlossen

Die LED ist zwischen Mikrocontroller und Versorgungsspannung angeschlossen und leuchtet nur, wenn der dazugehörige Pin auf Masse gezogen wird, damit ein geschlossener Stromkreis entsteht.

```
// Ausschalten der LED
PORTD |= (1<<PD6);
// Anschalten der LED
PORTD &= ~(1<<PD6);
```

4.1.4.2 Active High

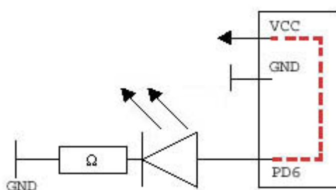


Abb. 28: LED als Active High angeschlossen

Die LED ist zwischen Mikrocontroller und Masse angeschlossen und leuchtet nur, wenn der dazugehörige Pin an der Versorgungsspannung anliegt, damit ein geschlossener Stromkreis entsteht.

```
// Ausschalten der LED
PORTD &= ~(1<<PD6);
// Anschalten der LED
PORTD |= (1<<PD6);
```

Der Anschluss der LED kann außer Acht gelassen werden, wenn man sie per XOR (Bit--Toggle) ansteuert. Ob der Zustand der LED Active-High oder Active-Low ist, kann dann nur daran erkannt werden, ob die LED nach der Initialisierung durch die folgenden Zeile Code an- oder ausgeschaltet ist.

```
PORTD ^= (1<<PD6);
```

4.1.5 Delay

In der avr-libc sind bereits Funktionen für die Verwendung von Delays definiert. Der Zugriff auf diese ist erst nach dem Inkludieren von <util/delay.h> möglich. Delays benötigt man zum Beispiel dann, wenn man eine LED an- und ausschaltet. Programmiert man dies ohne Delay, dann ist das An-und Ausschalten nicht mehr erkennbar, da man normalerweise mindestens eine Taktfrequenz von 1MHz am Chip anliegen hat. Da sowohl das An- als auch das Ausschalten nur einen Taktzyklus benötigt, ist die LED um die 500.000 mal in der Sekunde an. Das ist nicht einmal mehr annähernd für das menschliche Auge erkennbar.
Der erste Schritt ist, entweder im Makefile oder im Code festzulegen, welche Taktfrequenz am Chip anliegt. Erzeugt man sein Makefile mit mFile, ist die definierte Standardfrequenz 8MHz. Die sollte man dann in die tatsächlich anliegende Frequenz ändern.

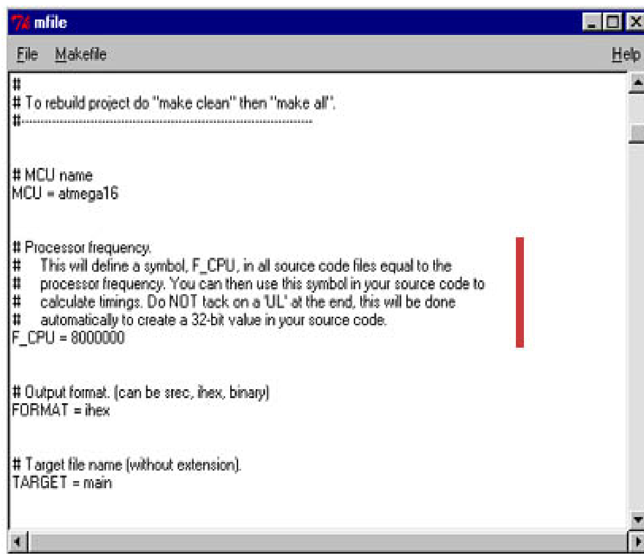


Abb. 29: mFile - F_CPU ändern

Will man die Taktfrequenz im Code festlegen, dann funktioniert das so
(bei einem externen Quarz mit 7.3728MHz):

```
#define F_CPU 7372800UL
```

Das #define muss vor der inkludierten delay.h im Code stehen.
Es gibt 4 verschiedene Delay-Funktionen:

```
// bei 1MHz max. Delay bei einem Aufruf von 768us
// bei 8MHz max. Delay bei einem Aufruf von 96us
_delay_loop_1(uint8_t __count)
_delay_us(double __us)
// bei 1MHz max. Delay bei einem Aufruf von 262ms
// bei 8MHz max. Delay bei einem Aufruf von 32ms
_delay_loop_2(uint16_t__count)
_delay_ms(double __ms)
```

Eine genaue Beschreibung der Delay-Funktionen und der Berechnung des maximal möglichen Delays in Abhängigkeit des CPU-Taktes findet sich im avr-libc Manual ab Seite 113.

4.1.6 Pull-Up und Pull-Down Widerstände

Der ATmega16 hat an allen I/O-Pins zuschaltbare Pull-Up Widerstände [pull]. Pull-Up Widerstände dienen dazu, den Pegel am Eingangspin stabil zuhalten. Hat man zum Beispiel ein Schaltelement, das nur bei Active-Low reagieren soll, also wenn am Pin, das als Eingang geschaltet ist, nur etwas passieren soll, wenn keine, bzw. vernachlässigbare Spannung anliegt, dann dient der verwendete Pull-Up Widerstand dazu, den Pegel bei logisch 1 zu halten, wenn keine Aktion durchgeführt wird. Hat man ein umgekehrt funktionierendes Schaltelement, also eines, das nur bei Active-High reagieren soll, dann nennt man den verwendeten Widerstand, Pull-Down Widerstand, da dieser den Pegel bei logisch 0 hält.
Eine typische Anwendung für Pull-Up Widerstände wäre zum Beispiel ein Taster. Dieser soll ja schließlich nur dann eine Aktion auslösen, wenn er gedrückt wird. Würde man hier den Pull-Up Widerstand nicht zuschalten, könnte die Aktion unter Umständen auch dann ausgelöst werden, wenn der Taster gar nicht gedrückt wird, da zum Beispiel Spannungsschwankungen auftreten.
Um einen Pull-Up Widerstand zuzuschalten, spricht man das Datenregister eines Ports an.
Wurde mittels DDRx ein Pin des Ports x als Eingang definiert, wird dann über PORTx der Pull-Up-Widerstand dieses Pins gesetzt.

```
DDRA &= ~(1 << PA0);  // Pin 0 von Port A wird als Eingang gesetzt.
PORTA |= (1 << PA0);  // Pull-Up-Widerstand für Eingangspin 0 aktivieren
PORTA &= ~(1 << PA0); // Pull-Up-Widerstand für Eingangspin 0 deaktivieren
```

Per Hardware zuschaltbare Pull-Ups benötigen zusätzlich Strom, weshalb man sie im Sleep-Mode deaktivieren sollte. Das SFIOR-Register verfügt über das Bit PUD - Pull-Up Disable. Wird das Bit gesetzt, dann sind die Pull-Ups an ALLEN Pins des Mikrocontrollers deaktiviert. Weckt man den Chip wieder auf, dann muss PUD wieder auf Null zurückgesetzt werden.

4.1.7 Entprellung von Tastern

Die Anwendung von Pull-Up Widerständen soll nun am Beispiel der Entprellung von Tastern erläutert werden.
Taster liefern, wenn sie gedrückt werden, leider kein einzelnes Signal (Pegelwechsel von 0 auf 1, oder umgekehrt, je nach Anschluss), sondern lassen unter Umständen den Pegel innerhalb von bis zu 100ms mehrmals hin- und herschwingen. Dies würde gleichzeitig natürlich bedeuten, dass die Funktionalität, die durch einen Tastendruck ausgelöst werden soll, mehrmals hintereinander aufgerufen wird. Um diesen unerwünschten Effekt zu verhindern, entprellt man den Taster (debouncing).
Dies geschieht durch eine simple Funktion, die, innerhalb eines bestimmten Zeitraums (bis zu 100ms) nach dem ersten Pegelwechsel, weitere vom Taster erkannte Pegelwechsel unterdrückt.

Im Codebeispiel ist der Taster an Pin 2 des Ports D angeschlossen

```
inline uint8_t debounce (volatile uint8_t *port, uint8_t pin)
{
        // Vergleich, ob Taster gedrückt wurde
        // ist dieser als low active geschaltet und wird gedrückt,
        // dann wird 1 zurückgegeben
        if(!(*port &(1<<pin)){
             for(i=0; i<20; i++) _delay_ms(32);
             return 1;
        }
        return 0;
}

int main (void){
        //Initialisierung des Tasters als Eingang mit Pull-Up-Widerstand
        DDRD &= ~(1<<PD2);
        PORTD |= (1<<PD2);
        while (1){
            if(debounce (&PIND, PD2)){
                for(i=0; i<20; i++) _delay_ms(32);
            }
        }
}
```

4.2 Speicherzugriffe

Der ATmega16 verfügt über 3 voneinander unabhängige Speicherbausteine.

4.2.1 RAM

Größe des RAM-Speichers: 1 KByte
Der RAM-Speicher (Random Access Memory) ist flüchtig, d.h. dass der Inhalt des RAMs bei Zusammenbruch oder Abschalten der Stromversorgung verloren geht. Im RAM wird der Stack abgelegt und der Compiler reserviert in diesem Speicherbaustein Platz für Variablen.

4.2.2 Flash

Größe des Flash-Speichers: 16 KByte
Der Flash-Speicher ist seitenweise beschreibbar, d.h. er kann zwar byteweise ausgelesen werden, aber nur ganze Seiten können gelöscht werden. Er ist außerdem in zwei Teile zerlegt: der eine Teil beinhaltet das Boot-Programm, der andere Teil das Anwendungsprogramm. Eigentlich ist es so, dass man sich nur in zwei speziellen Fällen um den Flash-Speicher kümmern muss. Zum Einen, wenn man ein Boot-Loader-Programm schreibt, zum Anderen, wenn man Hauptspeicher einsparen muss. Dann besteht die Möglichkeit Konstanten, deren Wert zum Kompillierzeit festliegt und die daraufhin nicht mehr geändert werden können, im Flash abzulegen.
Die Header-Datei <avr/pgmspace.h> muss dafür inkludiert werden.

Syntax für Speicherung konstanter Variablen im Flash:

```
// Globale Variablen
const uint8_t myInt PROGMEM = 20;
// Variablen in einer Funktion
static uint8_t myFuncInt PROGMEM = 30;
```

Syntax für Lesen von Variablen aus dem Flash:

```
uint8_t myReadInt;
myReadInt = pgm_read_byte(&myInt);
```

4.2.3 EEPROM

Größe des EEPROMs: 512 Byte

Das EEPROM ist ein nicht-flüchtiger Speicher, d.h. die gespeicherten Werte bleiben auch nach der Trennung von der Stromversorgung erhalten. Die Zellen im Speicher können 100.000 mal beschrieben werden, danach wird die Funktionstüchtigkeit nicht mehr garantiert. Das EEPROM kann byteweise beschrieben und ausgelesen werden. Dafür muss die Header-Datei
<avr/eeprom.h> inkludiert werden.

4.2.3.1 EEARH & EEARL

Dies sind die zwei Adressregister des EEPROM.

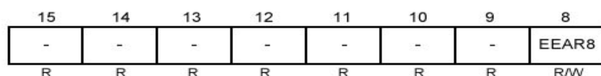


Abb. 30: EEPROM - EEARH

Die oberen 7 Bit des EEARH sind vom Chip reserviert und werden immer als 0 gelesen.

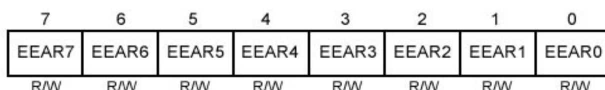


Abb. 31: EEPROM - EEARL

Aus den Bits 8 bis 0 ergibt sich die Adresse, die zwischen 511 und 0 liegt.

4.2.3.2 EEDR

8 Bit breites Datenregister des EEPROM

Hier werden die Daten, die in das EEPROM geschrieben oder gelesen werden sollen, gehalten.

EEPROM-Zugriff

```
// Byte in EEPROM schreiben
// eeprom_write_byte(uint8_t* adr, uint8_t val)
// Byte aus EEPROM lesen
// eeprom_read_byte(uint8_t* adr)
eeprom_write_byte((uint8_t *)0x01, 5); // Schreibt eine 5 in Byte 1 des EEPROM
eeprom_read_byte((uint8_t *)0x01);      // Liest Wert von Byte 1 des EEPROM
```

4.3 Interrupts

Wenn man komplexere Programme schreibt, ist es fast nicht mehr möglich den Programm-Ablauf sequentiell zu gestalten, weshalb man irgendwann auf Interrupts zurückgreifen muss. Implementiert man Interrupt-Routinen in den Code, und tritt bei der Ausführung des Codes ein Ereignis auf, das einen der vorher definierten Interrupts auslöst, so wird das Hauptprogramm gestoppt, und die entsprechende Interruptroutine aufgerufen. Erst nach ihrer Abarbeitung wird zurück in das Hauptprogramm gesprungen.

ISR = Interrupt Service Routine (Interruptroutine)

ISRs sollen klein, handlich und übersichtlich sein, weshalb es zu vermeiden ist, aufwendige Berechnung oder Schleifen darin auszuführen. Packt man in eine ISR zum Beispiel Code, dessen Abarbeitung 1 Sekunde dauert, dann steht das Hauptprogramm 1 Sekunde lang und in dieser Zeit passiert nichts. Außerdem sollte man es unterlassen, andere ISRs oder die gleiche, in einer Interruptroutine aufzurufen. Die Möglichkeit, einen solchen Code zu schreiben besteht zwar, kann aber zu unerwünschten Nebeneffekten führen.
Es gibt 2 verschiedene Arten von Interrupts - Interne und Externe Interrupts. Externe Interrupts werden erzeugt, wenn an den Pins INT0, INT1 oder INT2 ein auslösendes Ereignis auftritt, selbst dann, wenn der jeweilige Pin als Ausgang geschaltet ist. Externe Ereignisse können durch Schalter, Taster, Sensoren oder auch andere Mikrocontroller an obigen Pins einen Interrupt auslösen. Interne Interrupts werden durch Ereignisse im Chip ausgelöst (z.B. Timer, USART).

4.3.1 Interruptverarbeitung

Sobald man Interruptroutinen im Code implementiert, muss <avr/interrupt.h> eingebunden werden. Interrupts werden durch Aufruf von *sei();* aktiviert. Durch diesen Befehl das Global Interrupt Enable-Bit im Status Register des Mikrocontrollers gesetzt wird.
Will man die Behandlung von Interrupts wieder unterbinden, muss *cli();* aufgerufen werden. Dieser Befehl löscht das Global Interrupt Enable-Bit im Status Register.

4.3.2 Nicht unterbrechbare Interrupts

```
ISR(vectorname)
{
        // do something
}
```

Die Verwendung von ISR führt dazu, dass während der Ausführung des ISR-Codes keine weiteren Interrupts zugelassen werden. Erst wenn die Funktion beendet wird, wird das Global Interrupt Enable-Bit automatisch erneut gesetzt, und Interrupts sind wieder möglich.

4.3.3 Unterbrechbare Interrupts

```
void vectorname (void) __attribute__((interrupt);
...
void vectorname (void)
{
        .....
}
```

Hier muss bei der Programmierung aufgepasst werden, da diese Art von Interrupt-Aufruf nicht dazu führt, dass das Global Interrupt Enable-Bit gelöscht wird. Das heißt, beim Ausführen des Codes in der Funktion können andere Interrupts auftreten, die dann auch sofort ausgeführt werden, was zu erheblichen Problemen führen kann.

4.3.4 Interruptvektoren

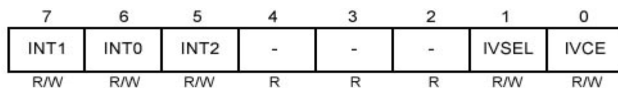
In der folgenden Tabelle sind sämtliche, durch den ATmega16 unterstützte Interrupt-Routinen, inklusive des dazugehörenden Enable-Bits und bei Ausführung des Interrupts gesetzten Flags, aufgelistet.

Vektorname	***Beschreibung***	***Interruptregister und Interruptbit***	***Interruptflag***
ISR (INT0_vect)	Externer Interrupt an Pin INT0	GICR - INT0	GIFR - INTF0
ISR (INT1_vect)	Externer Interrupt an Pin INT1	GICR - INT1	GIFR - INTF1
ISR (INT2_vect)	Externer Interrupt an Pin INT2	GICR - INT2	GIFR - INTF2
ISR (TIMER0_COMP_vect)	Timer/Counter0 Compare Match	TIMSK - OCIE0	TIFR - OCF0
ISR (TIMER0_OVF_vect)	Timer/Counter0 Overflow	TIMSK - TOIE0	TIFR - TOV0
ISR (TIMER1_CMPA_vect)	Timer/Counter1 Compare Match 1A	TIMSK - OCIE1A0	TIFR - OCF1A
ISR (TIMER1_CMPB_vect)	Timer/Counter1 Compare Match 1B	TIMSK - OCIE1B	TIFR - OCF1B
ISR (TIMER1_OVF_vect)	Timer/Counter1 Overflow	TIMSK - TOIE1	TIFR - TOV1
ISR (TIMER1_CAPT_vect)	Timer1 Capture Event	TIMSK - TICIE1	TIFR - ICF1
ISR (TIMER2_COMP_vect)	Timer/Counter2 Compare Match	TIMSK - OCIE2	TIFR - OCF2
ISR (TIMER2_OVF_vect)	Timer/Counter2 Overflow	TIMSK - TOIE2	TIFR - TOV2
ISR (SPI_STC_vect)	Serial Transfer Complete	SPCR - SPIE	SPSR - SPIF
ISR (TWI_vect)	2-Wire Serial Interface	TWCR - TWIE	TWCR - TWINT
ISR (SPM_RDY_vect)	Store Program Memory Ready	SPMCR - SPMIE	
ISR (ADC_vect)	ADC Conversion Complete	ADCSRA - ADIE	ADCSRA - ADIF
ISR (ANA_COMP_vect)	Analog Comparator	ACSR - ACIE	ACSR - ACI
ISR (USART_RXC_vect)	USART RX Complete	UCSRB - RXCIE	UCSRA - RXC
ISR (USART_TXC_vect)	USART TX Complete	UCSRB - TXCIE	UCSRA - TXC
ISR (USART_UDRE_vect)	USART Data Register Empty	UCSRB - UDRIE	UCSRA - UDRE
ISR (EE_RDY_vect)	EEPROM Ready	EECR - EERIE	

Tab 02: Interruptvektoren

4.3.5 Behandlung eines externen Interrupts

Wenn man einen Interrupt durch einen der Pins INT0, INT1 oder INT2 auslösen will, also einen externen Interrupt, dann muss in der Initialisierung das entsprechende Interrupt-Enable-Flag gesetzt sein, und es muss festgelegt werden, bei welcher Signaländerung am Pin der Interrupt ausgelöst werden soll. Angenommen man hat einen Taster an INT0 angeschlossen, und will einen externen Interrupt auslösen, sobald der Taster gedrückt wird. Dafür muss in GICR - General Interrupt Control Register - INT0 auf 1 gesetzt werden.

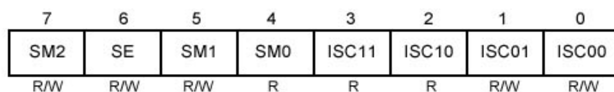


7	6	5	4	3	2	1	0
INT1	INT0	INT2	-	-	-	IVSEL	IVCE
R/W	R/W	R/W	R	R	R	R/W	R/W

Abb. 32: GICR - General Interrupt Control Register

```
GICR = (1<<INT0);
```

Außerdem muss noch definiert werden, wann der Interrupt ausgelöst wird. Wird der Taster einmal gedrückt kommt es (wenn man ihn entprellt hat) auf jeden Fall zu 2 Zustandsänderungen - also zwei Flankenwechseln. Wann der Interrupt dann ausgelöst werden soll, wird im MCUCR - MCU Control Register - über die ISC - Interrupt Sense Control - Bits für externe Interrupts an den Pins INT0 und INT1 bestimmt.



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

Abb. 33: MCUCR - MCU Control Register

Für einen externen Interrupt an INT2 muss das entsprechende ISC2 Bit im Register MCUCSR - MCU Control and Status Register - gesetzt werden.

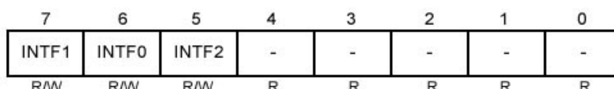
7	6	5	4	3	2	1	0
JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W

Abb. 34: MCUCSR - MCU Control and Status Register

Die Einstellungen zu den auslösenden Ereignissen aller externen Interrupts findet man im ATmega16 Datenblatt auf den Seiten 66 und 67. Soll der Interrupt bei steigender Flanke an Pin INT0 ausgelöst werden, dann müssten ISC01 und ISC00 gesetzt werden.

```
MCUCR = (1<<ICS01) | (1<<ISC00);
```

Wenn der Interrupt ausgelöst wurde (also der Taster an INT0 gedrückt wurde), dann reagiert der Mikrocontroller dementsprechend darauf, dass er im GIFR - General Interrupt Flag Register - das 6. Bit (INTF0) auf 1 setzt.



7	6	5	4	3	2	1	0
INTF1	INTF0	INTF2	-	-	-	-	-
R/W	R/W	R/W	R	R	R	R	R

Abb. 35: GIFR - General Interrupt Flag Register

Das hat zur Folge, dass die implementierte ISR - in diesem Fall *ISR (INT0_vect)* - aufgerufen, und der darin stehende Code abgearbeitet wird. Anschließend sollte das entsprechende Interrupt-Flag wieder zurückgesetzt werden, wenn das nicht automatisch von der Hardware erledigt wird.

4.4 Watchdog

Der Watchdog ist ein wirksames Mittel, um bei unvorhergesehenen Zuständen einen Reset auszuführen. Dies können Bugs sein, die das Programm zum Stillstand bringen, oder auch Endlosschleifen.
Der Watchdog wird zum Beispiel so programmiert, dass er jedes mal, wenn die Ausführung wieder im Hauptprogramm landet, ein Signal bekommt. Dieses Signal teilt dem Watchdog mit: alles ok, das Programm funktioniert. Landet die Ausführung in einer Endlosschleife, oder taucht ein unvorhergesehener Fehler auf, dann wird kein Signal mehr an den Watchdog geschickt. Je nachdem, auf welchen Schwellenwert man den Watchdog eingestellt hat, bekommt dieser innerhalb seiner festgelegten Zeitspanne kein OK-Signal mehr. Der Watchdog reagiert dann, indem er einen Reset auslöst.
Der Watchdog hat einen eigenen Timer/Counter und wird über einen eigenen eingebauten Takt (1MHz) im Chip gesteuert. Um den Watchdog implementieren zu können, muss die Header-Datei
<avr/wdt.h > eingebunden werden.

4.4.1 Aktivierung des Watchdogs

Als erstes wird festgelegt, in welchem Zeitfenster der Watchdog das Signal bekommt, dass er keinen Reset ausführen soll. Dieses Zeitfenster wird über einen Prescaler festgelegt, durch den die folgenden Timeouts vordefiniert sind:

Konstante	Timeout
WDTO_15MS	15 ms
WDTO_30MS	30 ms
WDTO_60MS	60 ms
WDTO_120MS	120 ms
WDTO_250MS	250 ms
WDTO_500MS	500 ms
WDTO_1S	1 s
WDTO_2S	2 s

Tab. 03: Watchdogkonstanten

➔ Watchdog wird aktiviert über:

```
wdt_enable(uint8_t timeout);
```

➔ Beispiel:

```
wdt_enable(WDTO_500MS);
```

➔ Watchdog-Timer zurücksetzen, damit kein Reset erfolgt

(muss vor Ablauf des Timeouts geschehen!)

```
wdt_reset();
```

➔ Watchdog deaktivieren

```
wdt_disable();
```

4.5 Timer/Counter

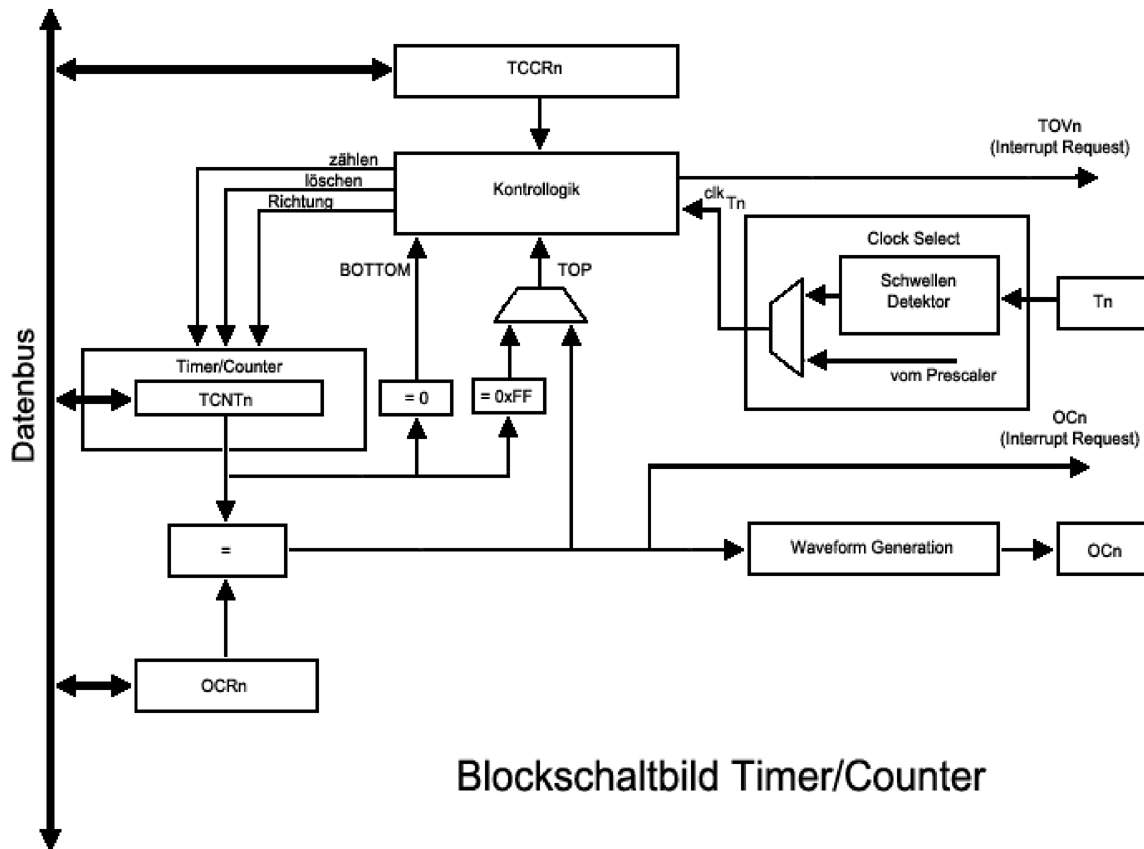


Abb. 36: Blockschaltbild Timer/Counter

Der ATmega16 verfügt über zwei 8bit Timer/Counter und einen 16bit Timer/Counter. Diese unterscheiden sich nicht nur in der Registerbreite, weshalb separat auf beide Timer/Counter-Module des Mikrocontrollers eingegangen wird. Im Prinzip sind Timer und Counter das gleiche - sie werden nur zum Zählen unterschiedlicher Ereignisse eingesetzt. Der Hardwarebaustein wird Timer genannt, wenn sich das Zählen auf einen intern oder extern anliegenden periodischen Takt bezieht. Kennt man dessen Frequenz, kann man nach einer vergangenen Zeitspanne Rückschlüsse auf den Wert des Timers ziehen. Zählt man ein Signal, das zu unterschiedlichen Zeiten auftritt, also nicht periodisch ist, dann spricht man von einem Counter. Da beide Anwendungen durch den gleichen Schaltungsbaustein im Chip realisiert werden, spricht man von Timer/Counter und unterscheidet die Bezeichnung erst dann, wenn man eine spezifizierte Anwendung beschreibt.

4.5.1 Timer/Counter0

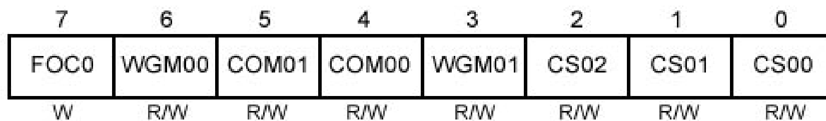
8bit Timer/Counter Modul

Der Timer/Counter0 kann über den internen Takt (veränderbar durch den Prescaler) oder durch einen an Pin T0 anliegenden externen Takt gesteuert werden. Ist kein Takt ausgewählt, ist Timer/Counter0 inaktiv. Timer/Counter0 besitzt zwei 8bit Register, in denen der aktuelle Zähler-Wert (TCNT0) und dessen Vergleichswert (OCR0) gespeichert ist. Diese Register werden kontinuierlich (einmal pro Takt) miteinander verglichen. In jedem Taktzyklus wird TCNT0 entsprechend des gewählten Modus verändert. Die Beschreibung der Modi erfolgt weiter unten. Der Inhalt des Vergleichsregisters OCR0 kann folgende Werte haben:

BOTTOM:	0x00 (0)
MAX:	0xFF (255)
TOP:	0x00 bis 0xFF (0 bis 255)

Stimmen die Werte der beiden Register TCNT0 und OCR0 überein, wird ein *Compare Match* ausgelöst und im darauf folgenden Takt das *Output Compare Flag (OCF0)* gesetzt. Wurde diese Option aktiviert, kann beim Setzen des OCF0 ein Interrupt ausgelöst werden. Dieser Interrupt löscht das OCF0-Bit sobald er ausgeführt wird. Erfolgt eine Verarbeitung ohne Interrupt, muss OCF0 per Hand zurückgesetzt werden, indem man das Bit auf 1 setzt. Je nach Operationsmodus des Timer/Counter0 wird dann das am Ausgangspin OC0 anliegende Signal verändert.

4.5.1.1 Timer/Counter0 Control Register TCCR0



7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Abb. 37: TCCR0

FOC0: Force Output Compare

Sobald dieses Bit gesetzt wird, wird die Ausführung eines *Compare Match* erzwungen, obwohl das Zählerregister TCNT0 und das Vergleichsregister OCR0 verschiedene Werte aufweisen. Es muss darauf geachtet werden, dass kein Interrupt ausgelöst und der Wert in TCNT0 wird nicht verändert wird.

high (1):

*FOC0 d*arf nur 1 sein, wenn der Betriebsmodus für Timer/Counter0 NICHT PWM ist.

low (0) :

*FOC0 m*uss 0 sein, wenn der Modus PWM ist

Beispiel:

Modus ist CTC, Compare Output Mode ist *Set OC0 on Compare Match*, Clock Select ist in diesem Beispiel nicht relevant

```
 TCCR0 = (0<<FOC0) | (0<<WGM00) | (1<<COM01) | (1<<COM00) |
    (1<<WGM01) | (X<<CS02) | (X<<CS01) | (X<<CS00);
...
// Funktion in der FOC0 gesetzt wird:
 void FOC0_Funktion (void)
 {
       TCCR0 |= (1<<FOC0);

       .....
 }
```

Angenommen in TCNT0 steht der Wert 34 und der Vergleichswert in OCR0 ist 128, dann wird, sobald FOC0 auf high gesetzt wird, ein Compare Match ausgeführt, obwohl der Counterwert und der Vergleichswert nicht übereinstimmen.

WGM01-00: Waveform Generation Mode

Bestimmt den Betriebsmodus, in dem der Timer/Counter0 verwendet wird

0-0: Normaler Modus

TCNT0 wird inkrementiert.

Wird der TOP-Wert überschritten, wird TCNT0 im nächsten Takt auf 0 zurückgesetzt, und gleichzeitig das Timer Overflow Flag (TOV0) gesetzt.

Wird der Timer/Counter0 Overflow Interrupt verwendet, wird TOV0 bei dessen Ausführung auf 0 zurückgesetzt. Ohne Interrupt-Behandlung behält TOV0 den Wert 1 bis zum nächsten Überlauf, wodurch der Counter auf 9 Bit erweitert werden kann.

Der 9bit Counterwert kann folgendermaßen ausgelesen werden:

```
// 16bit Variable anlegen, da 9bit Wert ausgelesen werden soll
uint16_t counterValue;
// der 9bit Wert muss nun zusammengesetzt werden aus
// dem Overflow-Flag TOV0 und dem Zählerwert aus TCNT0
// Dafür wird nur das Bit 0 (TOV0) aus dem TIFR Register
// um 8 Stellen nach links geshiftet
// und dann mit dem TCNT0-Wert ver-odert.
counterValue = (TIFR&0x01) << 8 | TCNT;
```

1-0: CTC - Clear Timer on Compare Match Modus

TCNT0 wird inkrementiert.

Kommt es bei einem Vergleich von TCNT0 und OCR0 zur Übereinstimmung (Compare Match), wird TCNT0 im nächsten Takt auf 0 zurückgesetzt. Gleichzeitig kann ein Interrupt ausgelöst werden, da das OCF0 (Output Compare) Flag gesetzt wird, wenn der Compare Match auftritt.

Das Interrupt-Flag TOV0 wird immer dann gesetzt, wenn der Zähler zurück auf 0x00 gesetzt wird.

0-1: phasenkorrekte PWM - Phase Correct PWM

TCNT0 wird bis 0xFF (MAX) inkrementiert, und nach Erreichen des MAX-Wertes bis 0x00 (BOTTOM) dekrementiert.

Die phasenkorrekte PWM kann auf zwei verschiedene Arten die Signale am Ausgangspin OC0 darstellen:

nicht-invertierter Modus:

OC0 wird auf low gesetzt, wenn der Compare Match zwischen TCNT0 und OCR0 auftritt, während TCNT0 inkrementiert wird.

OC1 wird auf high gesetzt, wenn der Compare Match zwischen TCNT0 und OCR0 auftritt, während TCNT0 dekrementiert wird.

invertierter Modus:

Die Ansteuerung von OC0 verhält sich genau umgekehrt.

Das Interrupt-Flag TOV0 wird immer dann gesetzt, wenn der Zähler 0x00 erreicht. Dadurch kann dann die entsprechende Interrupt-Routine ausgelöst werden.

1-1: schnelle PWM - Fast PWM

TCNT0 wird inkrementiert, bis 0xFF (MAX) erreicht ist.

Im darauf folgenden Taktzyklus wird der Registerinhalt auf 0 zurückgesetzt.

Auch dieser PWM-Modus kann invertiert oder nicht-invertiert betrieben werden.

Das Interrupt-Flag TOV0 wird immer dann gesetzt, wenn MAX erreicht wird.

COM01:COM00: Compare Match Output Mode

Über diese zwei Bits wird das Ausgangssignal (0 oder 1) am OC0-Pin gesteuert. Je nachdem in welchem Modus der Timer/Counter0 betrieben wird, verhält sich die Ansteuerung des Pins unterschiedlich.

0-0: Alle Modi

Es wird kein Signal an OC0 gesendet. Der Pin kann dann für normale I/O-Operationen verwendet werden.

0-1:

Normaler oder CTC Modus

Invertierung des jeweiligen Signalzustandes von OC0 (Toggle) bei Auftreten des Compare Match

Phasenkorrekte PWM oder schnelle PWM

Die beiden Bits dürfen in diesen Modi nicht gesetzt werden, da sie reserviert sind!

1-0:

Normaler oder CTC Modus

OC0 = 0 bei Compare Match

Schnelle PWM

OC0 = 0 bei Compare Match

OC0 = 1 bei TCNT0 = TOP

Wenn TOP 0xFF entspricht, wird der Compare Match ignoriert. Folglich erhält der Pin nur noch ein high-Signal.

Phasenkorrekte PWM

OC0 = 0 bei Compare Match während TCNT0 inkrementiert wird

OC0 = 1 bei Compare Match während TCNT0 dekrementiert wird

1-1:

Normaler oder CTC Modus

OC0 = 1 bei Compare Match

Schnelle PWM - invertiert

OC0 = 0 bei TCNT0 = TOP

OC0 = 1 bei Compare Match

Wenn TOP 0x00 entspricht, wird der Compare Match ignoriert. Folglich erhält der Pin nur noch ein low-Signal

Phasenkorrekte PWM - invertiert

OC0 = 0 bei Compare Match während TCNT0 dekrementiert wird

OC0 = 1 bei Compare Match während TCNT0 inkrementiert wird

CS02:CS00: Clock Select

Durch die drei Clock Select Bits wählt man den Prescaler der Mikrocontroller-CPU für das Timer/Counter0 Modul. Es kann entweder der Takt verwendet werden, mit dem der Chip betrieben wird (interner Takt), oder ein Taktsignal, das an Pin T0 anliegt (externer Takt).

0-0-0: Das Timer/Counter Modul ist inaktiv - es liegt keine Taktquelle an

0-0-1: Chiptakt ohne Prescaler

interner Oszillator mit 1MHz

pro Sekunde wird TCNT0 1.000.000 mal verändert

externer Oszillator mit 7,3728 MHz

pro Sekunde wird TCNT0 7.372.800 mal verändert

0-1-0: Chiptakt mit Prescaler = 8

interner Oszillator mit 1MHz

pro Sekunde wird TCNT0 125.000 mal verändert

externer Oszillator mit 7,3728 MHz

pro Sekunde wird TCNT0 921.600 mal verändert

0-1-1: Chiptakt mit Prescaler = 64

1-0-0: Chiptakt mit Prescaler = 256

1-*0-1*: Chiptakt mit Prescaler = 1024

1-1-0: externes Signal an T0 mit Taktsignal während der fallenden Flanke

1-1-1: externes Signal an T0 mit Taktsignal während der steigenden Flanke

(z.B. Quarzoszillator, Oszillatorschaltung ...)

4.5.1.2 Timer/Counter0 Interrupts

Soll ein Interrupt ausgelöst werden, wenn beim Vergleich von TCNT0 und OCR0 ein Compare Match auftritt, dann muss zuerst im TIMSK Register (Timer/Counter Interrupt Mask Register) das Bit OCIE0 (Timer/Counter0 Output Compare Match Interrupt Enable) gesetzt sein.
Erst dann kann die entsprechende Interrupt-Routine aufgerufen werden, wenn beim Auftreten des Compare Match das OCF0 Flag (Timer/Counter0 Output Compare Flag) im TIFR Register (Timer/Counter Interrupt Flag Register) gesetzt wird.
Beim Ausführen der Interrupt-Routine ISR (TIMER0_COMP_vect) wird OCF0 automatisch wieder auf 0 zurückgesetzt. Soll OCF0 per Hand auf 0 zurückgesetzt werden (zum Beispiel wenn keine Interrupt-Routine ausgeführt wird) dann muss OCF0 auf 1 gesetzt werden.

```
TIFR |= (1<<OCF0)
```

Soll ein Interrupt ausgelöst werden, wenn es zu einem Überlauf des Zählers kommt, dann muss zuerst im TIMSK Register (Timer/Counter Interrupt Mask Register) das Bit TOIE0 (Timer/Counter0 Overflow Interrupt Enable) gesetzt sein. Erst dann kann die entsprechende Interrupt-Routine aufgerufen werden, wenn beim Auftreten des Zähler-Überlaufs das TOV0 Flag (Timer/Counter0 Overflow Flag) im TIFR Register (Timer/Counter Interrupt Flag Register) gesetzt wird. Beim Ausführen der Interrupt-Routine ISR (TIMER0_OVF_vect) wird TOV0 automatisch wieder auf 0 zurückgesetzt.
Soll TOV0 per Hand auf 0 zurückgesetzt werden (zum Beispiel wenn keine Interrupt-Routine ausgeführt wird) dann muss TOV0 auf 1 gesetzt werden.

```
TIFR |= (1<<OCF0)
```

4.5.2 Timer/Counter1

16bit Timer/Counter Modul
Alle Register dieses Moduls sind 16Bit breit, weshalb der Timer/Counter in einer größeren Genauigkeit angesteuert werden kann. Im Gegensatz zu 8Bit Registern, muss man beim Zugriff auf 16Bit-Register auf die Vorgehensweise achten.

Um schreibend auf ein 16Bit Register des Timer/Counter1 zuzugreifen, muss das Highbyte zuerst geschrieben werden, danach das Lowbyte. Beim Lese-Zugriff muss zuerst das Lowbyte, danach das Highbyte ausgelesen werden.

```
// Beispiel für das Lesen eines 16Bit Wertes aus dem Zählerregister
// TCNT1 von Timer/Counter1
tcnt1_Value = TCNT1L | (TCNT1H<<8);
```

Der Timer/Counter1 bietet zusätzlich auch die Möglichkeit, über die *Input Capture Unit* ein externes Ereignis abzufangen. Dazu wird entweder ein Signal am Pin *ICP1* (PortD Pin 6) oder ein Ereignis am Analog Komparator abgefragt, um einen Zeitstempel zu setzen mit dem dann weitergearbeitet wird.

4.5.3 Ansteuerung einer LED über PWM

Codebeispiel: CD-Verzeichnis Code/PWM

PWM[pwm] ist die Modulation eines Rechtecksignals in seinem Tastverhältnis bei konstanter Frequenz.

$$T = \frac{t_{ein}}{t_{ein} + t_{aus}}$$

Der Mittelwert der Spannung beträgt dann:

$$U_m = U_{aus} + (U_{ein} - U_{aus}) * T$$

Im folgenden Codebeispiel wird *Fast PWM* auf eine LED angewendet, die am Pin3 des Ports B (OCR0) anliegt. Das Vergleichsregister *OCR0* wird nach einer bestimmten Anzahl von Timer/Counter0 Compare Match Interrupts erhöht, bis ein maximale Wert erreicht ist, danach wird *OCR0* dekrementiert bis 0 erreicht wird. Dieses auf-und abzählen von OCR0 wird kontinuierlich wiederholt.

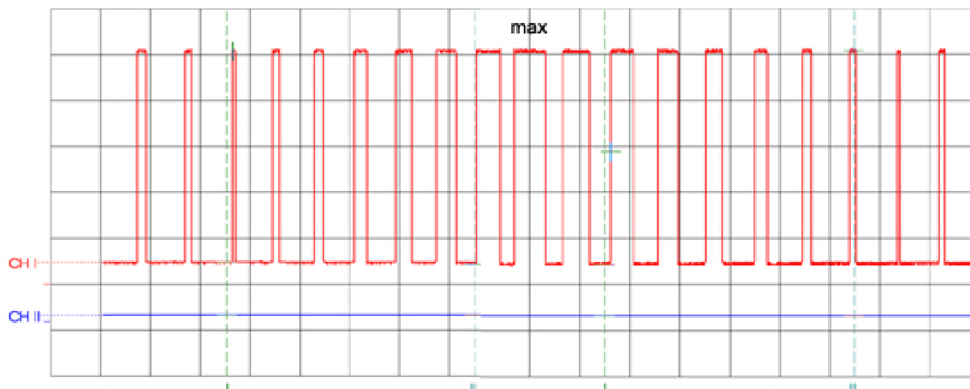


Abb. 38: Darstellung von Fast PWM mit Hilfe eines Oszilloskops

Diese Grafik wurde mittels eines Oszilloskops erstellt und zeigt die Funktionsweise des unteren Codebeispiels. Das dritte Spannungssignal ist der Beginn der Hochdimmphase. Das Rechtecksignal wird in jedem Puls um einen bestimmten Faktor verbreitert, bis der maximal Wert des Vergleichsregister erreicht ist. Danach erfolgt während des Herunterdimmens eine Verschmälerung jedes Rechtecksignals, bis der minimalste Wert von OCR0 erreicht wird.

```
 // Globale Variablen
volatile uint8_t counter;
bool dimmUp;          // Dimmrichtung
// Timer/Counter0 Compare Match Interrupt Routine
ISR (TIMER0_COMP_vect)
{
    counter++;
}
```

```
int main (){
      // init PWM-LED
      DDRB = (1<<PB3);
    // init Timer/Counter0 Control Register
      // WGM01:WGM00 -  Waveform Generation Mode  -  FastPWM
      // COM01:COM00 -  Compare Match Output Mode -
      // Set OC0 on compare match
      // CS02 : CS00 -  Clock Select
      TCCR0 = (1<<WGM01) | (1<<WGM00) | (1<<COM01) | (1<<COM00) | (1<<CS02);
      // init Timer/Counter0 Zählregister
      TCNT0 = 0;
      // init Timer/Counter0 Interrupt Mask - Bit für Interrupt bei
      // Compare Match setzen
      TIMSK = (1<<OCIE0);
      // init OCR0 Vergleichsregister
      OCR0 = 0;
      // Enable Interrupts
      sei();
      // init globals
      dimmUp = true;            // am Anfang wird hochgedimmt
      // trotz eingesetztem Prescaler soll nur nach einer best. Anzahl
      // von Interrupts OCR0 hochgesetzt werden, daher ein zusätzlicher
      // counter als Delay-Ersatz
      counter = 0;
      // Endlosschleife
      while(1){
              // während hochgedimmt wird, soll OCR0 inkrementiert werden,
              // sobald der Interrupt 10 Mal aufgerufen wurde
              if (dimmUp){
                      if(counter == 10){
                              OCR0+=20;   // inkrementieren von OCR0
                      // Wenn OCR0 255 erreicht hat,
                              // wird wieder heruntergedimmt
                              if(OCR0 == 240) dimmUp = false;
                              counter = 0;
                      }
              }
              // während heruntergedimmt wird, soll OCR0 dekrementiert werden,
              // sobald der Interrupt 10 Mal aufgerufen wurde
              else if (!dimmUp){
                      if(counter == 10){
                      OCR0-=20;         // dekrementieren von OCR0
                      if(OCR0 == 0) dimmUp = true;
                      counter = 0;
                      }
              }
      }
}
```

4.6 USART

Codebeispiel: CD-Verzeichnis Code/USART

Universal Synchronous and Asynchronous Receiver Transmitter

= Serielle (bitweise) synchrone oder asynchrone Übertragung zwischen Mikrocontroller und RS232-Schnittstelle eines Rechners.

asynchrone Übertragung:

Die Taktinformation wird nicht mitübertragen, wodurch mit Hilfe von Start- und Stopbits ständig neu synchronisiert werden muss.

synchrone Übertragung:

Die Taktinformation wird auf Pin1 (XCK) mitübertragen..

Vollduplex-Fähigkeit:

Es kann gleichzeitig gesendet und empfangen werden

Standardmäßig verläuft die Übertragung zwischen Sender und Empfänger nach folgendem Muster:
Startbit ------- 5 bis 8 Datenbits (LSB zuerst) ------- optional: Paritybit ------- Stopbit
Die Datenübertragungsrate wird in Baud angegeben, wobei bit/s eigentlich der korrekte Begriff ist.

4.6.1 Initialisierung des USART

Beim Definieren des externen Taktes muss man Folgendes beachten:
Arbeitet man mit einem Editor, der das Makefile nicht selber anlegt, dann muss im Makefile die Taktfrequenz explizit mit F_CPU = *Frequenz* angegeben werden.
Wurde das Makefile mit mFile erzeugt, muss man den zum ausgewählten Default-Wert von F_CPU anpassen.
Arbeitet man mit dem AVR Studio, dann muss man die Taktfrequenz entweder im Source-File per *#define F_CPU Frequenz* festlegen, oder unter *Project/Configuration Options/General/Frequency* eintragen.

```
// F_CPU definieren
#define F_CPU 7372800
// Festlegen der Baud-Rate
#define BAUD 9600
// Festlegen der Einstellung des UBRR (USART Baud Rate Register)
// F_CPU = externer Takt des Chips
#define MYUBRR F_CPU/16/BAUD-1
void init_USART (uint8_t ubrr)
{
   // Festlegen der Baudrate in UBRR
   UBRRH = (unsigned char) (ubrr << 8);
   UBRRL =   (unsigned char) ubrr;
   // Transmitter und Receiver aktivieren
   UCSRB = (1<< RXEN) | (1 << TXEN);
```

```
    // USART Control and Status Register
    // Klassiker: 8n1 (8Datenbits - no parity - 1Stopbit)
    // USBS :StopBitSelect
    // UCSZ1:0: Anzahl der Datenbits
    UCSRC = (1 << URSEL)  |  (0 << USBS)  | (1 << UCSZ0)  |  (1 << UCSZ1);
}
```

Wenn man Bits im UCSRC-Register verändern will, muss man immer das URSEL-Bit setzen, da sich UCSRC und UBRRH dieselbe I/O Adresse teilen und ansonsten die Bits im UCSRC-Register nicht veränderbar sind.

4.6.2 Terminal

Um die Daten zu sehen, die über USART zwischen Mikrocontroller und PC ausgetauscht werden, benötigt man ein so genanntes Terminal-Programm. Davon gibt es unzählige freie Versionen im Netz, sowohl für Windows als auch für Linux. Anhand des Terminals von Br@y++ soll die grundsätzliche Funktionsweise kurz vorgestellt werden. Das Terminal liegt im Ordner *Software/Terminal* der CD oder ist unter http://bray.velenje.cx/avr/terminal/ zum Download verfügbar.

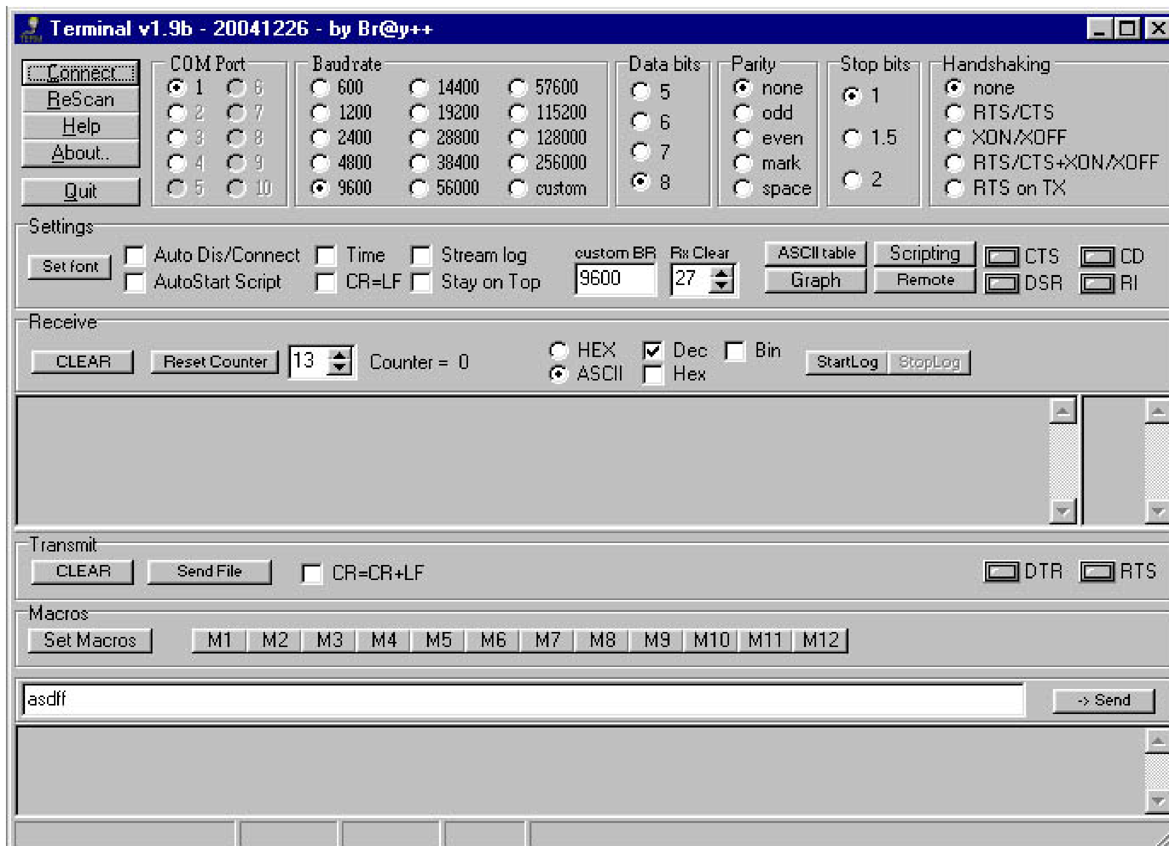


Abb. 39: Terminal

COM Port:

Es sind nur die aktiven COM-Ports auswählbar. Hier wird der COM-Port aktiviert, über den der Rechner durch RS232 mit dem Mikrocontroller verbunden ist.

Baud Rate:

Diese Einstellung muss mit der Definition im Code übereinstimmen, da ansonsten nur Datenmüll gesendet und empfangen wird.

Data Bits:

Standardmäßig wären das 8 Bit

Parity:

Für den ATmega16 stehen hier nur *none*, *even* oder *odd* zur Verfügung. Standardmäßig wird none gewählt

Stop Bits:

Standardmäßig wird ein 1 Stopbit übertragen

Handshaking:

Früher waren die Übertragungsraten so gering, dass man den Kommunikationspartnern durch Handshaking-Signale die Möglichkeit gab, die Übertragung zu unterbrechen, bis die angekommenen Daten verarbeitet waren. Diese Signale wurden allerdings auf gesonderten Leitungen übertragen, die heutzutage nicht mehr von modernen Mikrocontrollern unterstützt werden. Aufgrund der mittlerweile mehr als ausreichenden Übertragungsraten ist das Handshaking-Verfahren bei der Kommunikation über USART überflüssig geworden.

Nachdem alle Einstellungen vorgenommen wurden und das Terminal per *Connect* mit dem µC verbunden wurde, sollten eingehende Daten im *Receive-Fenster* sichtbar sein und Daten die über das *Transmit-Fenster* an den Mikrocontroller gesendet werden, an diesem ankommen.

4.7 Analog/Digital Wandler

Codebeispiel: CD-Verzeichnis Code/ADC

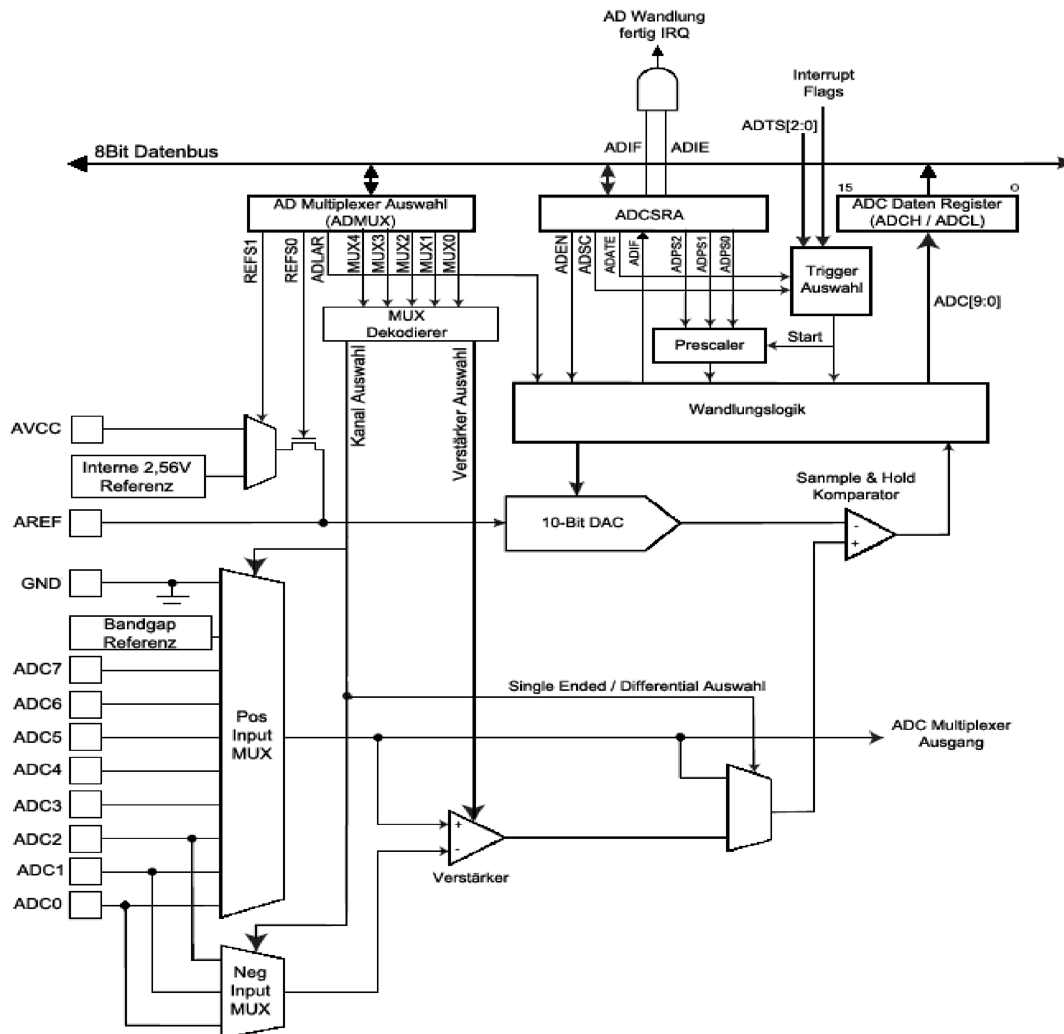


Abb. 40: Blockschaltbild A/D-Wandler

Alle 8 Pins des Ports A dienen als Eingänge einer analogen Spannung für den chipinternen A/D-Wandler. Dieser hat eine Auflösung von 10Bit, und kann somit die referenzierte analoge Spannung mit Werten von 0 (0 Volt) bis 1023 (VCC) als digitales Signal darstellen. Um den A/D-Wandler benutzen zu können, muss AVCC mit VCC verbunden sein (und darf nicht mehr als 0.3V von VCC abweichen) und GND (Pin 31) mit der Masse verbunden sein. Der Pin AREF dient für die analoge Referenzspannung und kann entweder an VCC angeschlossen sein oder unverbunden bleiben.

Anforderungen:

Wenn AREF nicht verbunden wird, dann kann über ein bestimmtes Bit eine interne Referenzspannung von 2,56 Volt oder auch VCC gewählt werden. Allerdings wäre es vorteilhaft, AREF nicht vollkommen unbeschaltet zu lassen, sondern einen Kondensator zwischen AREF und GND zu schalten, da so Störsignale abgeschwächt werden.

Wenn AREF verbunden wird, kann entweder VCC als maximale Referenzspannung oder 2 Volt als minimale Referenzspannung verwendet werden.

AVCC darf 5,5V nicht über- und 2,7V nicht unterschreiten.

Eine Besonderheit des ADC-Port ist, dass man die Pins PA0 und PA1 (bzw. PA2 und PA3) als Eingänge für einen eingebauten Verstärker[8] verwenden kann, der die Möglichkeit bietet, den ADC-Wert um 10 oder 200 zu verstärken.

4.7.1 A/D-Wert Berechnung

Der normale ADC-Wert, der das Resultat einer an einem der ADC-Pins anliegenden Spannung ist:

$$ADC_{val} = \frac{V_{pin} * 1024}{V_{ref}}$$

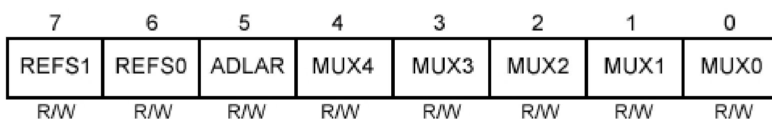
Werden an die zur Verstärkung nutzbaren Pins zwei unterschiedliche Spannungen angelegt, dann errechnet sich der ADC-Wert wie folgt:

$$ADC_{val} = \frac{(V_{PA1} - V_{PA0}) * 512 * Verstärkungsfaktor}{V_{ref}}$$

4.7.2 A/D-Wandler Register

4.7.2.1 ADMUX - AD Multiplexer Selection Register

Eine genaue Beschreibung der Bitfunktionen findet sich im ATmega16 Datenblatt auf Seite 215.



7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Abb. 41: ADMUX

REFS1:0 Reference Selection Bit

Quelle der Referenzspannung (extern von PIN AREF, oder intern entweder VCC oder 2,56V)

ADLAR Ausrichtung des Wandlungsergebnisses

1: Das Ergebnis der Wandlung ist links-ausgerichtet

0: Das Ergebnis der Wandlung ist rechts-ausgerichtet

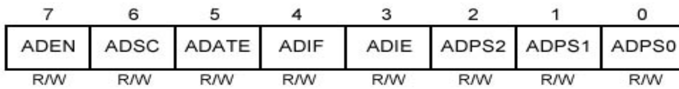
Mehr dazu unter der Überschrift *ADC Data Register*

MUX4:0 Durch diese 4 Bits wird die Quelle der analogen Eingangsspannung festgelegt.

8 vgl. Kapitel 4.9

4.7.2.2 ADCSRA - ADC Control and Status Register

Eine genaue Beschreibung der Bitfunktionen findet sich im ATmega16 Datenblatt auf Seite 217.



7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Abb. 42: ADCSRA

ADEN ADC Enable

1: Aktivierung des AD-Wandlers

0: Deaktivierung des AD-Wandlers. Geschieht dies während einer Wandlung, wird diese abgebrochen.

ADSC ADC Start Conversion

1: Wandlung wird gestartet

0: ADSC wird automatisch auf 0 gesetzt, wenn die Wandlung fertig ist.

ADATE ADC Auto Trigger Enable

1: Dieses Bit muss gesetzt sein, wenn man den ADC im Free Running Mode betreibt, oder wenn man im Single Conversion Mode Interrupts verwendet, um eine neue Wandlung auszulösen. Die Trigger-Quelle wird im SFIOR-Register gesetzt.

0: Ist ADATE nicht gesetzt, dann wird die Wandlung nicht durch einen Trigger gestartet (z.B. normaler Funktionsaufruf)

ADIF ADC Interrupt Flag

1: Bit wird automatisch gesetzt, wenn die Wandlung abgeschlossen, und der neue Wert ins Datenregister ADC geschrieben wurde

0: Bit ist 0, wenn die Wandlung gerade stattfindet

ADIE ADC Interrupt Enable

1: Wird als Trigger-Quelle im Single Conversion Mode der *ADC Conversion Complete Interrupt* gewählt, dann muss dieses Bit gesetzt sein.

0: Trigger-Quelle ist ein anderer Interrupt, oder es wird kein Auto Triggering verwendet

ADPS2:0 ADC Prescaler Select Bits

Der chipinterne AD-Wandler benötigt einen eigenen Takt, den er sich aus dem für den Chip verwendeten Takt generiert. Dieser darf beim ATmega16 zwischen 50kHz und 1MHz liegen.

Die wählbaren Prescaler werden nach folgender Formel ausgerechnet:

minimaler Prescaler = Chip-Taktfrequenz / max. Taktfrequenz des ADC

minimaler Prescaler (ATmega16) = 7.372800 / 1.000000 = 7,37

➔ 8

maximaler Prescaler = Chip-Taktfrequenz / min. Taktfrequenz des ADC

maximaler Prescaler (ATmega16) = 7.372800 / 50.000 = 147,46 --> 128

➔ 128

Wählt man für den Prescaler einen Wert, der nicht innerhalb dieser Grenzen liegt, dann wird der ADC-Wert sehr ungenau.

4.7.2.3 ADC - ADC Data Register

Eine genaue Beschreibung der Bitfunktionen findet sich im ATmega16 Datenblatt auf Seite 218.
Im ADC-Register wird das Ergebnis der AD-Wandlung gespeichert. Da der AD-Wandler eine Auflösung von 10bit hat, reicht 1 Byte für die Darstellung des Ergebnisses nicht aus. Daher ist das ADC-Register 2 Byte groß und teilt sich in ADCH (High-Byte) und ADCL (Low-Byte). Wird das ADLAR-Bit in ADMUX nicht gesetzt (also ADLAR = 0), dann wird das 10bit-Ergebnis der Wandlung von rechts nach links in ADC gespeichert:

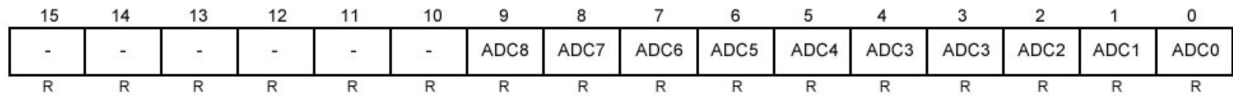


Abb. 43: ADLAR = 0

Wenn das Ergebnis in ADC rechts ausgerichtet ist, MUSS ADCL vor ADCH ausgelesen werden, da erst ein neuer Wert in ADC geschrieben werden kann, wenn ADCH ausgelesen wurde.

```
uint16_t adcValue;
adcValue = ADCL | (ADCH<<8); // Richtiger Code
adcValue = (ADCH<<8) | ADCL; // Falscher Code
```

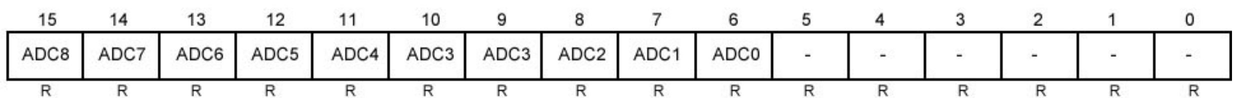


Abb. 44: ADLAR = 1

ADLAR wird in ADMUX auf 1 gesetzt, wenn es ausreichend ist, das Ergebnis der AD-Wandlung in 8bit darzustellen. Dann wird nur das High-Byte des ADC-Registers ausgelesen.

```
uint16_t adcValue;
adcValue = ADCH;
```

4.7.2.4 SFIOR - Special Function I/O Register

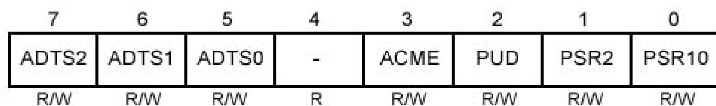


Abb. 45: SFIOR

Durch die Bits ADTS2, ADTS1 und ADTS0 im SFIOR Register wird der Wandlungsmodus bestimmt, bzw. im Single Conversion Mode die Quelle angegeben, durch die die nächste Wandlung ausgelöst werden soll.
Es ist zu beachten, dass Veränderungen dieser Bits nicht berücksichtigt werden, wenn das Bit ADATE im Register ADCSRA NICHT gesetzt wurde, da die Wandlungen weder durch den Free Running Modus, noch durch eine Trigger Quelle ausgelöst werden.

0-0-0: Free Running Mode ist aktiv

0-0-1: Trigger Quelle - Analoger Komparator

0-1-0: Trigger Quelle - externer Interrupt am Pin INT0

0-1-1: Trigger Quelle - Compare Match an Timer/Counter0

1-0-0: Trigger Quelle - Overflow an Timer/Counter0

1-0-1: Trigger Quelle - Compare Match an Timer/Counter1B

1-1-0: Trigger Quelle - Overflow an Timer/Counter1

1-0-1: Trigger Quelle - Externes Ereignis an ICP1

4.7.3 Initialisierung: Free Running Mode

Im Free Running Mode wird das ADSC-Bit nur einmal gesetzt, um die erste Wandlung zu starten. Danach wird während der gesamten Zeit, in der der Free Running Mode aktiv ist, sofort nach Beendigung der einen Wandlung die nächste gestartet. Dieser Modus wirkt sich nicht auf andere Funktionen aus, die das Programm abzuarbeiten hat, allerdings leidet die Genauigkeit der Wandlungsergebnisse.

```
// Initialisierung für Free Running Mode
void initFreeRunningConversion (void)
{
    ADMUX = (1<<REFS0);
    ADCSRA = (1<<ADEN) | (1<<ADSC) | (1<<ADPS1) | (1<<ADATE);
}
```

4.7.4 Initialisierung: Single Conversion Mode

Im Single Conversion Mode wird die A/D-Wandlung durch Auftreten eines Interrupts oder durch explizites Setzen des ADSC-Bits ausgelöst. Der Interrupt kann extern sein, oder durch eines der Timer/Counter Register ausgelöst werden. Dies wird im Register SFIOR festgelegt. Im Gegensatz zum Free Running Mode ist das Ergebnis dieser Wandlung sehr genau.

```
// Initialisierung für Single Conversion Mode mit Interrupt
void initSingleConversion (void)
{
    ADMUX = (1<<REFS0);
    ADCSRA = (1<<ADEN) | (1<<ADSC) | (1<<ADATE) | (1<<ADPS0) | (1<<ADPS1);
    // Auslöser der A/D-Wandlung ist ein externer Interrupt an INT0
    SFIOR = (1<<ADTS1);
}

// Initialisierung für Single Conversion Mode ohne Interrupt
// Die Wandlung wird im Code durch explizites Setzen des Bits ADSC gestartet
void initSingleConversion (void)
{
    ADMUX = (1<<REFS0);
    //ADATE ist nicht gesetzt
    ADCSRA = (1<<ADEN) | (1<<ADSC) | (1<<ADPS0) | (1<<ADPS1);
}
```

4.8 LC Display

Codebeispiel: CD-Verzeichnis Code/LC Display

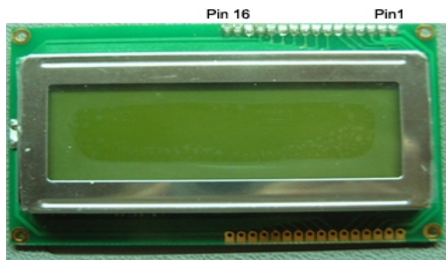


Abb. 46: 4 zeiliges LC-Display

LCD - Liquid Crystal Display (Flüssigkristall Display)

Trotz der großen Anzahl verfügbarer Punkt-Matrix-LCDs ist es nicht notwendig, sich bei jedem Display auf ein Neues mit dem Anschluss und der Bedienung zu beschäftigen, da sich der LCD Controller-Chip HD44780 als Standard durchgesetzt hat, und entweder dieser oder aber kompatible Chips in allen Zeilendisplays verbaut wurden.
Displays mit 4 Zeilen à 40 Zeichen bilden eine kleine Ausnahme, da bei diesen die Funktionalität über 2 Controller geregelt wird, wodurch man die Pin-Belegung und auch die Steuerung geringfügig anpassen muss. Im Folgenden soll der Anschluss und Umgang mit einem 4x20 Punkt-Matrix LCD beschrieben werden. [lcd]

4.8.1 Pinbelegung eines 4x20 LCD mit Hintergrundbeleuchtung

Es gibt sowohl Displays mit Hintergrundbeleuchtung (16 Pins) als auch ohne Hintergrundbeleuchtung (14 Pins). Das Datenblatt des Displays, auf das sich dieses Kapitel bezieht, befindet sich auf der CD im Ordner *Datenblätter/LCD/LCD204B_DIS.pdf*[LCD].

Pin 1: Masse (Vss)

Pin 2: VCC (Vdd)

Pin 3: Masse oder Schleifer-Beinchen eines Potentiometers um den Kontrast der Zeichen einzustellen.

Pin 4: RS - Verbunden mit einem Portpin des µC

Wird der Pin am Mikrocontroller auf *high (1)* gezogen, dann setzt der Display-Chip die an den Datenbit-Pins anliegenden Signale als Daten um, die auf dem Display dargestellt werden.
Wird der Pin am Mikrocontroller auf *low (0)* gezogen, dann werden die anliegenden Signale als Befehle an den Display-Controller interpretiert.

Pin 5: R/W - Verbunden mit einem Portpin des µC oder an Masse angeschlossen

Wird dieser Pin mit der Masse verbunden, dann ist er dauerhaft auf *low (0)*, d.h. es können nur Daten oder Befehle an das Display gesendet, aber keine Informationen aus dem Display-Chip gelesen werden, da für eine Read-Operation der Pin auf *high* gezogen werden müsste.

Pin 6: E - Chip Enable Signal - Verbunden mit einem Pin des µC

Will man Daten oder Befehle an den Display-Controller schicken, dann kann dieser die

Eingangssignale nur korrekt verarbeiten, wenn man den Pin auf *high (1)* setzt, einige Taktzyklen wartet und anschließend den Pin wieder auf *low (0)* zieht.

Pin 7 bis Pin 14: Datenleitungen

Hat man genügend Pins am Mikrocontroller zur freien Verfügung, dann kann man alle 8 Datenbit-Pins des Display-Controllers mit dem Mikrocontroller verbinden. Will man Pins sparen, genügt es auch, nur 4 Datenbit-Pins anzuschließen. Um den Aufwand bei der Programmierung in Grenzen zu halten, sollte man die Datenleitungen 4 bis 7 des Display-Controllers mit den oberen 4 Pins eines Mikrocontroller-Ports verbinden, da man dann die ersten vier Bit *(high nibble)* direkt übertragen kann und anschließend nur einmal um 4 Bit shiften muss, um das l*ow nibble* an das LCD zu senden.

Optional:

Pin 15: VCC der Hintergrundbeleuchtung

Pin 16: Masse der Hintergrundbeleuchtung

Im Stromkreis der Hintergrundbeleuchtung muss entweder an der Versorgungsspannung oder an der Masse ein Widerstand (10Ω) eingelötet werden, da die anliegende Spannung 4,2V nicht übersteigen darf.

4.8.2 LCD Steuerungsfunktionen

Durch den obigen Anschluss an den Mikrocontroller können nun folgende Display-Pins manipuliert werden:

RS	***R/W***	***DB7***	***DB6***	***DB5***	***DB4***	***DB3***	***DB2***	***DB1***	***DB0***

RS - L:

Solange der Display-Controller Funktionen ausführen soll, ist RS *low (0)*.

Erst wenn Daten auf dem Display dargestellt werden sollen, wird RS auf *high (1)* gesetzt

R/W - L:

Soll in den Display-Speicher geschrieben werden, ist dieses Bit *low (0)*.

Wenn aus dem Speicher gelesen werden soll, wird R/W auf *high (1)* gesetzt.

Ist der Pin mit der Masse verbunden, dann können KEINE Daten aus dem Displa<peicher gelesen werden!

Set Function

Dieses Kommando wird zum Initialisieren des LCDs benötigt.

Nach der Initialisierung können diese Einstellungen nicht mehr verändert werden.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	H	DL	N	F	X	X

DB7-DB6 - Low:

müssen *low (0)* sein

DB5 - High:

'Set Function' Kommandobit

DB4 - DL:

Anzahl der Datenleitungen, die an den Mikrocontroller angeschlossen sind.

high (1): alle 8 Datenbits stehen für die Übertragung zur Verfügung

low (0) : nur 4 Datenleitungen stehen zur Verfügung

zuerst wird das *high nibble* übertragen danach das *low nibble*

DB3 - N:

Anzahl der Display-Zeilen.

high (1): für 1 bis 4 zeilige Displays (bei einzeiligen Displays mit mehr als 8 Zeichen)

low (0) : für einzeilige Displays mit maximal 8 Zeichen

DB2 - F:

Anzahl der Punkte, durch die die einzelnen Zeichen dargestellt werden.

high (1): Zeichendarstellung mit Auflösung von 10x5 Punkten

4 Zeilen können auf dem LCD angezeigt werden

low (0) : Zeichendarstellung mit Auflösung von 7x5 Punkten

2 Zeilen können auf dem LCD dargestellt werden

DB1:DB0 - X:

Diese Bits können *high (1)* oder *low (0)* sein, da sie nicht relevant sind.

Display Clear

Kommando zum Löschen des Displays

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	L	L	L	L	L	H

DB7-DB1 - Low:

müssen *low (0)* sein

DB0 - High:

'Display Clear' Kommandobit

Display On/Off

Kommando, durch welches das Display an oder aus geschaltet wird und ein (blinkender) Cursor gesetzt werden kann. Die Hintergrundbeleuchtung ist vom Ausschalten des Displays nicht betroffen!

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	L	L	H	D	C	B

DB7-DB4 - Low:

müssen *low (0)* sein

DB3 - High:

'Display On/Off' Kommandobit

DB2 - D:

An- und Ausschalten des LCDs

high (1): Display ist eingeschaltet

low (0) : Display ist ausgeschaltet

DB1 - C:

An- und Ausschalten des Cursors

high (1): Cursor ist eingeschaltet

low (0) : Cursor ist ausgeschaltet

DB0 - D:

Blinken aller 5x7 oder 5x10 Punkte an der Cursor-Position

high (1): Blinken ist eingeschaltet

low (0) : Blinken ist ausgeschaltet

Entry Mode Set

Kommando, durch das bestimmt wird, ob sich der Cursor nach links oder rechts bewegt und ob das Display geshiftet werden soll.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	L	L	L	H	I/D	SH

DB2 - High:

'Entry Mode Set' Kommandobit

DB1 - I/D:

Richtung, in die sich der Cursor bewegen soll

high (1): Cursor bewegt sich nach rechts

low (0) : Cursor bewegt sich nach links

DB1 - SH:

Shift des Displays: 1. Zeichen wird an letzte Stelle der 1. Zeile gesetzt und dann wird nach links geshiftet

high (1): Display wird geshiftet

low (0) : Display wird nicht geshiftet

Return Home

Cursor wird an die Adresse 0x00 gesetzt

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	L	L	L	L	H	X

DB1 - High:

'Return Home' Kommandobit

DB0 - X:

Dieses Bit kann *high (1)* oder *low (0)* sein, da es nicht relevant ist.

Shift

Kommando durch das festgelegt wird, ob und in welche Richtung das Display oder der Cursor geshiftet werden soll

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	L	L	H	S/C	R/L	X	X

DB4 - High:

'Shift' Kommandobit

DB3 - S/C:

Auswahl, ob Display oder Cursor bewegt werden soll

high (1): Auswahl des Displays

low (0) : Auswahl des Cursors

DB2 - R/L:

Richtung, in die Display oder Cursor bewegt werden soll

high (1): nach rechts

low (0) : nach links

DB1:DB0 - X

Diese Bits können *high (1)* oder *low (0)* sein, da sie nicht relevant sind.

Set DD RAM Address

Kommando, durch das die Cursor-Adresse festgelegt wird

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	H	DD6	DD5	DD4	DD3	DD2	DD1	DD0

DB7 - High:

'Set DD RAM Address' Kommandobit

DB6 - DB0:

Festlegen der Adresse, an der der Cursor stehen soll

Addressmap

	Zeichen 1	...	Zeichen 20
Zeile 1	0b1000 0000	...	0b1001 0011
	0x80	...	0x93
	128	...	147
Zeile 2	0b1100 0000	...	0b11010011
	0xC0	...	0xD3
	192	...	211
Zeile 3	0b1001 0100	...	0b1010 0111
	0x94	...	0xA7
	148	...	167
Zeile 4	0b1101 0100	...	0b11100111
	0xD4	...	0xE7
	212	...	231

Tab. 04: Addressmap

4.8.3 Initialisierung des Displays

Nachdem das LCD an die Versorgungsspannung angeschlossen wurde, durchläuft der LCD-Controller eine Initialisierungssequenz. Im Anschluss daran muss das Display von Hand initialisiert werden. Dies läuft folgendermaßen ab:

```
// ca. 20ms warten, damit das LCD nach dem Stromanstecken
// Zeit hat warmzulaufen
_delay_ms(20);

// Benötigte Port-Pins auf Ausgang setze, damit darüber Signale an das LCD // gesendet werden können
LCD_DDR = (1<<RS) | (1<<CE) | (1<<DB7) | (1<<DB6) | (1<<DB5) | (1<<DB4);

// dreimaliges Aufrufen von Set Function mit DB5 und DB4 auf high (1)
for(n=0; n<3; n++){
    // 8bit Mode
    LCD_PORT = (0<<RS) | (1<<DB5) | (1<<DB4);
    // Enable Pin (CE) auf high (1) setzen mindestens 220ns warten, dann
    // Enable Pin zurück auf low (0) setzen
    // wird ab hier -> SetEnable(); genannt
    LCD_PORT |= (1<<CE);
    _delay_us(1);
    LCD_PORT &= ~(1<<CE);
    _delay_ms(5);
}
// Umschalten auf 4 Datenleitungen - DB4 wird auf low (0) gesetzt
LCD_PORT = (0<<RS) | (1<<DB5);
SetEnable();

// Ab hier werden alle Signale (Kommandos und Daten) über 4 Datenleitungen
// übertragen daher nun immer die Aufteilung in high-nibble und low-nibble
// Festlegen der Zeilenanzahl
// high-nibble
LCD_PORT = (0<<RS) | (1<<DB5);
SetEnable();
// low nibble
LCD_PORT = (0<<RS) | (1<<DB3);
SetEnable();

// Festlegen der Zeichendarstellung - wenn 5x10 Punkte verwendet werden sollen
  ...
// Display auf AUS stellen
// high nibble
LCD_PORT = (0<<RS);
SetEnable();
// low nibble
LCD_PORT = (0<<RS) | (1<<DB3);
SetEnable();

// Display Clear
// high nibble
```

```
LCD_PORT = (0<<RS);
SetEnable();
// low nibble
LCD_PORT = (0<<RS) | (1<<DB0);
SetEnable();
// Entry Mode Set - DB2 und DB1 setzen - DB0 auf 0
// high nibble
LCD_PORT = (0<<RS);
SetEnable();
// low nibble
LCD_PORT = (0<<RS) | (1<<DB2) | (1<<DB1);
SetEnable();
```

Nun kann man Daten auf dem Display anzeigen lassen, selbst Zeichen definieren und auch im Nachhinein Darstellungsfunktionen des LCDs (z.B. Shiften) verändern.

4.8.4 Eigene Zeichen definieren

Bestandteil des Displays ist ein ROM-Speicher, in dem alle verfügbaren Zeichen (192 Stück) als Bitmuster gespeichert sind (siehe Zeichentabelle im Datenblatt des LCDs). Im
RAM des Controllers werden 64 Byte Speicher zur Verfügung gestellt, um eigene Zeichen zu definieren - dies ist der so genannte *Character Generator RAM.*

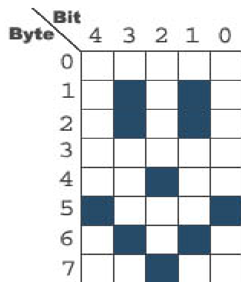


Abb. 47: Selbstdefiniertes LCD-Zeichen

Um ein Zeichen im CG RAM selbst zu definieren, muss dem LCD-Controller zuerst gesagt werden, dass er auf diesen Speicherbereich zugreifen soll. Dies geschieht über das Kommando *Set CG RAM Address*.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
L	L	L	H	CG5	CG4	CG3	CG2	CG1	CG0

DB6 - High:

'Set CG RAM Address' Kommandobit

DB5 - DB0:

Festlegen der Adresse, ab der das Zeichen gespeichert werden soll.

```
// CG RAM Adresse an den Controller übergeben
// Es stehen 64 Byte zur Verfügung - also können 8 Zeichen definiert werden.
// Das 1.Zeichen steht an der Speicherstelle 0, das 2. an der Stelle 8 usw.
```

```
// Selbstdefiniertes Zeichen an der Speicherstelle 0 verknüpft mit
// Kommandobit für Set CG RAM Address.
cdAdr = 0b01000000;

LCD_PORT = (0<<RS);
// UND-Verknüpfung um die unteren 4bit zu maskieren
LCD_PORT |= (cgAdr&0xf0);
SetEnable();
LCD_PORT = (0<<RS);
LCD_PORT |= (cgAdr<<4);
SetEnable();
// Array der Größe 8 definieren
unsigned char byte[8];
// Array füllen
byte[0] = 0b00000000;
byte[1] = 0b00001010;
byte[2] = 0b00001010;
byte[3] = 0b00000000;
byte[4] = 0b00000100;
byte[5] = 0b00010001;
byte[6] = 0b00001010;
byte[7] = 0b00000100;

// Array in den CG RAM schreiben
unsigned char n;
for(n=0; n<8;n++){
   // Data to be send
   // high nibble
   LCD_PORT = (1<<RS);
   LCD_PORT |= (byte[n]&0xf0);
   SetEnable();
   // low nibble
   LCD_PORT = (1<<RS);
   LCD_PORT |= (byte[n]&0xf0);
   SetEnable();
}
// CG RAM verlassen, indem man Set DD RAM Address oder Display Clear aufruft
...
// Zeichen ausgeben
// high nibble
LCD_PORT = (1<<RS);
LCD_PORT |= (0&0xF0);
SetEnable();
// low nibble
LCD_PORT = (1<<RS);
LCD_PORT |= sentVal<<4;
SetEnable();
```

4.9 Operationsverstärker

Operationsverstärker[op1][op2] sind das Arbeitspferd und die Grundkomponente von fast jeder Analogschaltung. Was das Gatter im Digitalbereich ist, kann man vom OP in der analogen Welt sagen.
Das Grundprinzip des Operationsverstärkers ist, dass die Eingangsspannungen verglichen, und ihre Differenz durch Rückkoppelung am Ausgang verstärkt wird. Allerdings sind mit diesem Bauteil auch andere Verschaltungen, wie zum Beispiel Addierer und Subtrahierer oder Filter aller Art möglich. Im Folgenden werden ausgewählte Operationsverstärker-Schaltungen kurz vorgestellt.

4.9.1 Komparator

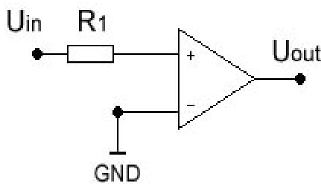


Abb. 48: nicht-invertierender Komparator

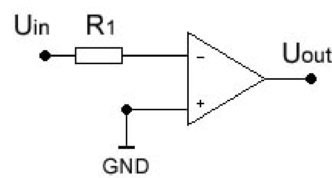


Abb. 49: invertierender Komparator

Der Komparator ist die einfachste Operationsverstärker-Schaltung.
Das linke Schaltbild zeigt einen *nicht-invertierenden Komparator*, dessen Ausgangsspannung gegen die positive Betriebsspannung tendiert, da die Eingangsspannung am nicht-invertierenden Eingang anliegt und der invertierende Eingang auf Masse gezogen wird. Wird die Eingangs-spannung am invertierenden Eingang angelegt, und der nicht-invertierende Eingang auf Masse gezogen, dann liegt am Ausgang die negative Betriebsspannung an. In dieser Schaltung wirkt die größtmögliche Verstärkung (*Leerlaufverstärkung*), da keine Gegenkoppelung am Ausgang anliegt.
Die erzeugte Ausgangsspannung wird nach folgender Formel berechnet:

$$U_{Ausgang} = V_{max} * U_{Eingang}$$

Da die Ausgangsspannung die Betriebsspannung (maximal 5V) nicht übersteigen kann, sollte für die Leerlaufverstärkung erst die maximale Eingangsspannung ausgerechnet werden, bei der die Ausgangsspannung die Betriebsspannung erreicht.

$$U_{Eingang} = \frac{U_{VCC}}{V_{max}}$$

Das würde heißen, dass bei einer Versorgungsspannung von 5V der Komparator bereits bei einer Eingangsspannung von 50mV die maximale Ausgangsspannung erreicht.

4.9.2 Verstärker

Durch Gegenkopplung[kopp] wird die Ausgangsspannung dem invertierenden Eingang des OPs zugeführt, wodurch es zu einer Verstärkung der Eingangsspannung um den Faktor V am Ausgang des Operationsverstärkers kommt. Je nachdem, welche Verschaltung den Operationsverstärker umgibt, wird er zu einem invertierenden Verstärker, an dessen Ausgang die invertierte verstärkte Spannung anliegt, oder zu einem nicht-invertierenden Verstärker, an dessen Ausgang die nicht-invertierte verstärkte Spannung anliegt.

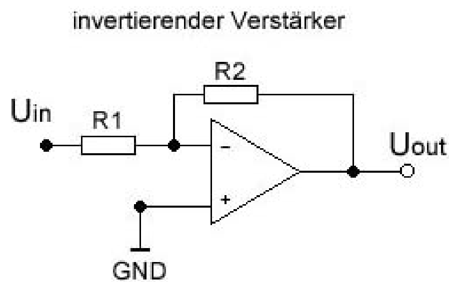


Abb. 50: invertierender Verstärker

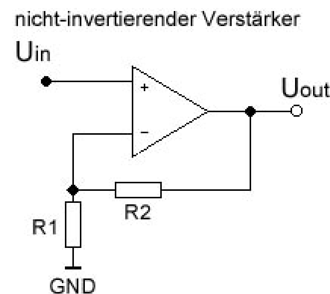


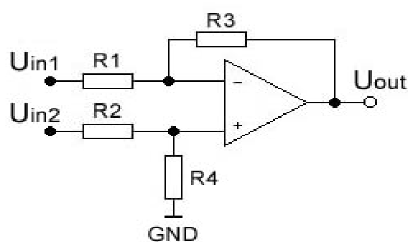
Abb. 51: nicht-invertierender Verstärker

$$-U_{Ausgang} = V * U_{Eingang} \quad mit \quad V = \frac{R_1}{R_2}$$

$$U_{Ausgang} = V * U_{Eingang} \quad mit \quad V = 1 + \frac{R_1}{R_2}$$

4.9.3 Subtrahierer

Ein Operationsverstärker wird zum Subtrahierer, wenn die äußere Beschaltung gleichzeitig einem invertierendem und einem nicht-invertierendem Verstärker entspricht.



Wenn gilt: $R_1 = R_2 = R_3 = R_4$

dann ist

$$U_{Ausgang} = U_{E2} - U_{E1}$$

Abb. 52: Subtrahierer

Der OP fungiert als reiner Subtrahierer.

Wenn gilt:

$$\frac{R_1}{R_2} = \frac{R_3}{R_4}$$

dann ist

$$U_{Ausgang} = (U_{E2} - U_{E1}) * \frac{R_1}{R2}$$

Der OP fungiert als Subtrahierer, und das Ausgangssignal wird gleichzeitig um den Faktor R_1/R_2 verstärkt.

Gelten diese Sonderfälle nicht, da $R_1 \neq R_2 \neq R_3 \neq R_4$ gilt,

dann liegt am Ausgang folgende Spannung an:

$$U_{Ausgang} = U_{E2} * (1 + \frac{R_3}{R_1}) * \frac{R_4}{(R_2 + R_4)} - \frac{R_3}{R_1} * U_{E1}$$

4.9.4 Addierer

Wenn ein Operationsverstärker als Addierer beschaltet wird, dann besteht die Schaltung aus einem invertierenden Verstärker, und zusätzlichen Eingangsspannungen die durch einen Vorwiderstand unterschiedlich gewichtet werden können, und anschließend aufsummiert werden. Es können beliebig viele Eingangsspannungen an den Summierverstärker angeschlossen werden.

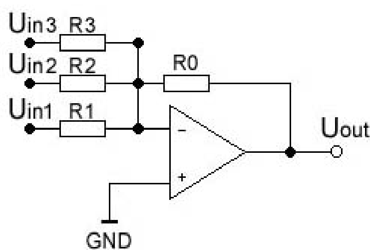


Abb. 53: Addierer

Wenn gilt: $R_0 \neq R_1 \neq R_2 \neq _{...} \neq R_x$

dann ist

$$U_{Ausgang} = -R_0 * (\frac{U_{E0}}{R_0} + \frac{U_{E1}}{R_1} + \frac{U_{E2}}{R_2} + ... + \frac{U_{Ex}}{R_x})$$

Wenn gilt: $R_1 = R_2 = R_3 = ... R_x$

dann ist

$$U_{Ausgang} = \frac{R_0}{R_x} * (U_{E0} + U_{E1} + U_{E2...} + U_{Ex})$$

4.10 Temperatursensor

Codebeispiel: CD-Verzeichnis Code/TemperaturSensor

Temperatursensoren[temp] sind elektronische Bauteile, die die anliegende Temperatur in eine elektrische Größe umwandeln. Es gibt verschiedene Typen von Sensoren, die sich in der Funktionsweise, in der Ansteuerung und auch im Preis unterscheiden, weshalb auf die einzelnen Gruppen nun näher eingegangen werden soll.

4.10.1 Sensortypen

4.10.1.1 Heißleiter und Kaltleiter

Diese Temperatursensoren sind nichts anderes als veränderliche Widerstände in einem Plastikgehäuse. Kaltleiter, auch PTCs (Positive Temperature Coefficient) genannt, leiten den anliegenden Strom besser bei kalten Umgebungstemperaturen, da dann der Widerstand kleiner ist. Mit steigenden Temperaturen steigt auch der Widerstand. Der Widerstand von Heißleitern, oder NTCs (Negative Temperature Coefficient), ist klein, wenn die Umgebungstemperatur hoch ist, und leitet schlechter bei niedrigen Temperaturen. Beide Arten von Temperatursensoren zeigen ein nichtlineares Verhalten bei Veränderung der Temperatur, wodurch es schwierig ist, diese Sensoren für Messungen über große Temperaturbereiche zu verwenden.

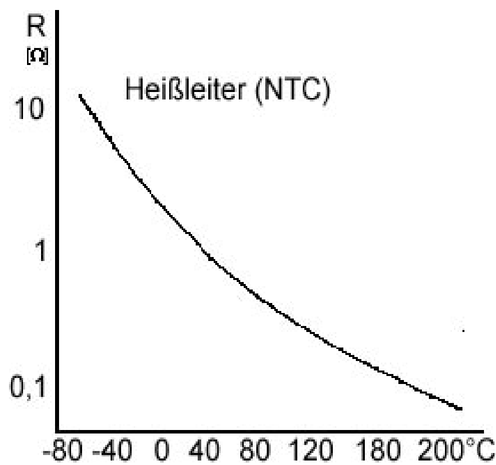


Abb. 54: NTC

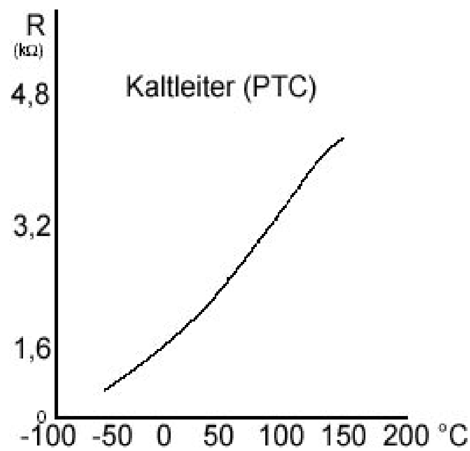


Abb. 55: PTC

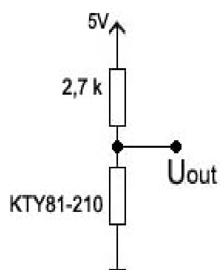
NTCs und PTCs sind veränderliche Widerstände. Um sie zu messen muss in irgendeiner Weise eine veränderliche Spannung erzeugt werden. Der einfachste Weg, diese Spannung zu messen, ist, den AD Wandler des Mega16 zu verwenden. Danach ist das Signal digital und kann damit auch leicht manipuliert werden.

4.10.1.2 Digitale Temperatursensoren

Bei digitalen Temperatursensoren entfällt der Zwischenschritt, die Messergebnisse erst an den ADC des Mikrocontrollers zu schicken, da diese Sensoren bereits einen AD-Wandler eingebaut haben. Das Messergebnis wird bei diesen Sensoren in digitaler Form über 1-Wire, 3-Wire oder I^2C direkt an den µC übertragen und kann dort weiterverarbeitet werden. Die Genauigkeit digitaler Temperatursensoren ist höher als die von NTCs/PTCs, außerdem sind sie bereits kalibriert. Allerdings ist die Kommunikation zwischen diesen ICs und dem Mikrocontroller aufwendiger zu handhaben, als die Auswertung eines Eingangssignals am ADC Pin, wie es bei NTCs/PTCs der Fall ist.

4.10.2 Anschluss des KTY81-210

Der KTY81-210[KTY] ist ein herkömmlicher PTC mit einer nichtlinearen Temperatur-Widerstandskurve. Um diese Kurve linear zu bekommen, wird ein Spannungsteiler mit dem Temperatursensor und einem 2,7 kΩ Widerstand aufgebaut.



$$U_{Ausgang} = U_{Eingang} * (\frac{R_2}{R_1} + R_2)$$

Abb. 56: Spannungsteiler

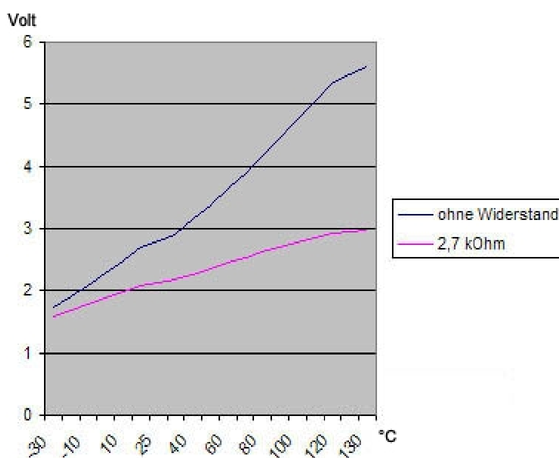


Abb. 57: Temperatur-Spannung-Kurve

Diese Kurve zeigt deutlich die Verbesserung, die man durch eine Spannungsteilerschaltung erreicht. Der Spannungsbereich, der am Spannungsteiler (U_{out})anliegt, liegt zwischen 1,33 Volt bei -50°C und 3,06 Volt bei 150°C. Da Temperaturen unter -30°C schwer zu erzielen sind, ist der folgende Schaltungsaufbau für die Temperaturspanne zwischen -30°C und 150°C ausgelegt. Da die Messung des Sensors an den AD-

Wandler gelegt und von diesem ausgewertet wird, bietet es sich an, den Spannungsbereich des Wandlers komplett auszunutzen (0V bis 2,56, da die interne Referenzspannung verwendet wird) und dann die Genauigkeit durch eine Verstärkung des Eingangssignals zu erhöhen.

Im ersten Schritt wird deshalb dem Spannungsteiler ein OP nachgeschaltet, der als Subtrahierer fungiert. So wird der Spannungsbereich von 1,57 Volt bis 3,06 Volt auf einen Spannungsbereich von 0,1 Volt bis 1,51 Volt reduziert. Da Temperaturen von über 110 °C schlecht auf einer normalen Platine zu erzielen sind, da die Abkühlung durch Abstrahlung zu groß ist, wird die obere Temperaturgrenze auch reduziert. Dadurch schrumpft der Spannungsbereich am oberen Ende nach der Subtrahiererschaltung auf 1,29 Volt.

Im zweiten Schritt wird das durch den Subtrahierer veränderte Spannungssignal um den Faktor 2 verstärkt, um eine Verdoppelung der Genauigkeit der gemessenen Spannung und der daraus resultierenden gemessenen Temperatur zu erreichen.

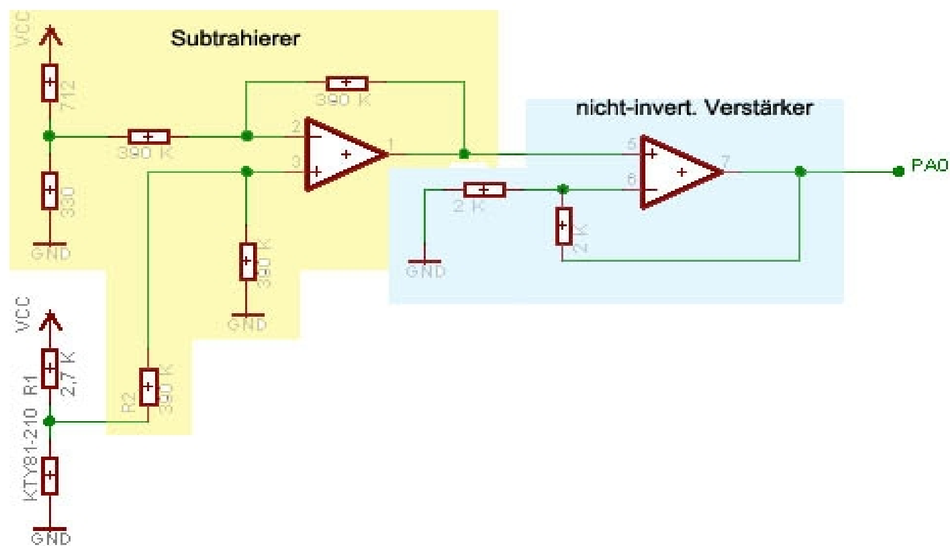


Abb. 58: Schaltungsaufbau des Temperatursensors[9]

Beispiel:

Messung von T = 0°C

Spannungsteiler Ust_temp:

$$U_{st_temp} = U_{in} * (R_2 / (R_1 + R_2)) = 5V * (1{,}63k\Omega / 2{,}7k\Omega + 1{,}63k\Omega) = 1{,}88V$$

Spannungsteiler

$$Ust = U_{in} * (R_2 / (R_1 + R_2)) = 5V * (330\Omega / 712\Omega + 330\Omega) = 1{,}58V$$

Subtrahierer:

$$U_{out} = U_{st_temp} - U_{st} = 1{,}88V - 1{,}58V = 0{,}30V$$

Verstärker:

$$U_{out} = U_{in} * (1 + (R_2 / R_1) = 0{,}30V * (1 + 2k\Omega*2\Omega) = 0{,}60V$$

9 Schaltbild des Temperatursensors: CD - Schaltbild/Temperatursensor.bmp

4.10.3 Kalibrierung des KTY81-210

Die Kalibrierung des Temperatursensors ist auf jeden Fall notwendig, da er im Lieferzustand eine Abweichung von 2-3°C erreichen kann. Außerdem ist es so, dass die Verstärkerschaltung die gemessenen Werte zusätzlich verfälscht und mit zunehmender Temperatur das Ergebnis noch mehr von der tatsächlichen Temperatur abweicht. Zur Kalibrierung verwendet man Wasser - einmal in Form von Eis (am Besten zerstoßen) und einmal kochend. Dann hat man als Referenztemperatur 0°C und 100°C. Um mit den Werten des AD-Wandlers und des Sensors besser arbeiten zu können, bietet es sich an, mit den *typischen* Widerstandswerten aus dem Datenblatt und den durch die Verstärkerschaltung berechneten Soll-Werte, eine Temperaturabhängige Tabelle aufzustellen.
Im ersten Schritt wird der durch die Temperaturmessung erzeugt A/D-Wert abgefangen.
Da der KTY ein Kaltleiter ist, ist das Ergebnis bei niedrigen Temperaturen unverfälschter, weshalb man bei der Kalibrierung mit dem Eiswasser anfangen sollte. Angenommen der Temperatursensor misst im Eis 0°C. Die Eingangsspannung am ADC sollte dann bei ca. 0,6Volt liegen. Der korrespondierende ADC-Wert dazu ist 252. Den tatsächlichen ADC-Wert sollte man an die 252 angleichen, da die Verstärkerschaltung das ADC-Ergebnis etwas verfälscht. Dividiert durch den Faktor 8 ergibt sich ein Wert zwischen 30 und 31, von dem jetzt noch die 30°C, die der Sensor im Minusbereich messen soll, abgezogen werden. Der zweite Kalibrierungsschritt findet mit kochendem Wasser statt. Da der Temperatursensor mit zunehmender Temperatur ungenauer wird und das Ergebnis außerdem durch die Eigenerwärmung verfälscht wird, sucht man hier nun den Faktor, der die im kochenden Wasser gemessene Temperatur auf 100°C +/- 1 bis 2 °C einstellt. Selbst nach der Kalibrierung muss man trotzdem mit einer Ungenauigkeit zwischen 1 und 2 Grad leben.

4.11 Summer

Codebeispiel: CD-Verzeichnis Code/Piezo

Summer basieren auf dem *piezoelektrischen Effekt* [piz]. Der Piezokristall eines Summers verformt sich, wenn elektrische Spannung angelegt wird, und gibt dadurch einen Ton von sich.

Summer gliedern sich in zwei Gruppen. Zum Einen gibt es diejenigen, die beim Anschluss an die Spannung sofort einen nicht-veränderbaren Dauerton von sich geben. Diese nennt man *Piezo-Summer*.

Zum Anderen gibt es Piezo-Schallwandler. Wird ihnen Strom zugeführt, dann ertönt nur ein kurzes Klacken. Erst wenn man die anliegende Spannung in einer bestimmten Frequenz ein- und ausschaltet, wird ein Dauerton hörbar, wobei dessen Tonhöhe von der Frequenz des Ein-Ausschaltvorgangs abhängt. Je kürzer die Zeit zwischen Ein- und Ausschalten, desto höher ist der Ton.

4.11.1 Piezo-Schallwandler

Im folgenden Codebeispiel wird der Piezo-Schallwandler EPM 121[EMP] verwendet. Dieser wird über Pin7 des Ports D gesteuert, wodurch der Piepser auch über den Timer/Counter2 betrieben werden könnte. Im einfachsten Fall schaltet man die Versorgungsspannung des Pins mit einem Delay.

```
 // Ansteuerung eines Summers über Delay
// Der Summer ist active-low geschaltet
  int main (){
    // PD7 als Ausgang schalten
    DDRD |= (1<<PD7)
    PORTD |= (1<<PD7);      // Summer ist aus

    while(1){
        PORTD |= (0<<PD7); // Summer ist an
        _delay_ms(32);      // Delay von 32 Millisekunden
        PORTD |= (1<<PD7); // Summer ist aus
        _delay_ms(32);      // Delay von 32 Millisekunden
    }
}
```

4.12 TWI

Codebeispiel: CD-Verzeichnis Code/TWI

TWI [twi1][ATM315] steht für *Two Wire Serial Interface* und ist das ATMEL-Äquivalent zu I^2C (gesprochen I Quadrat C oder I square C). TWI darf aus patentrechtlichen Gründen nicht I^2C genannt werden. I^2C ist ein serieller Bus der Anfang der 80er Jahre von der Firma Phillips entwickelt wurde. Die Besonderheit des Busses liegt darin, dass ein Master eine Vielzahl von Slaves über kurze Entfernungen hinweg, durch nur zwei Leitungen kontrollieren kann. Die Übertragungsraten lagen anfangs bei 100 kBit/s, wurden 1992 auf 400 kBit/s erweitert. Der neueste I^2C Standard erlaubt sogar Übertragungsraten von bis zu 3,4MBit/s, allerdings wird dies vom ATmega16 nicht unterstützt. Der Adressraum des Bussystems ist 7Bit breit, wodurch maximal 128 verschiedene I^2C-Komponenten angeschlossen werden könnten.
TWI wird über zwei bidirektionale, synchrone Leitungen betrieben:

- *SCL* (Serial Clock Line) als Taktleitung
- *SDA* (Serial Data Line) als Datenleitung

Außer dem normalen Master-Slave-Modus ist es auch möglich ein Multi-Master-System aufzubauen, dessen Stabilität durch Arbitrierung und Kollisionserkennung gewährleistet wird.

4.12.1 TWI-Schaltung

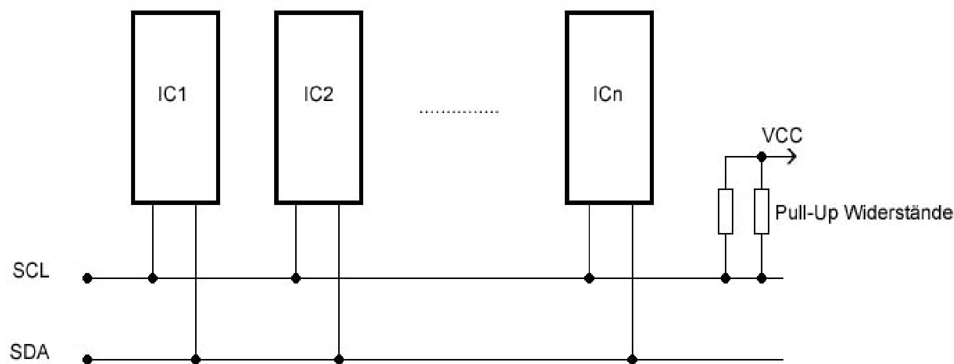


Abb. 59: Anschluss von Slaves an TWI

Es ist essentiell, dass die TWI-Leitungen mit Pull-Up Widerständen versehen werden. Beim ATmega16 müssen diese nicht extern verbaut werden, sondern es können die in der Hardware integrierten Pull-Ups softwaremäßig zugeschaltet werden.

4.12.2 Adress- und Datenpaket

Adresspaket:

Das Adresspaket ist 9Bit lang.

Bei der Übertragung des Pakets wird immer zuerst das MSB (Most Significant Bit) gesendet.

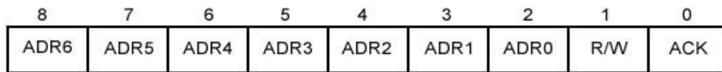


Abb. 60: TWI Adresspaket

Jeder Teilnehmer im System hat eine einzigartige Adresse, die 7 Bit lang ist.

Die Adresse 0000 000 ist für den *General Call* reserviert, und kann deshalb nicht an einen Slave vergeben werden. Ein General Call wird dann vom Master gesendet, wenn er ein Datenpaket an alle Slaves schicken möchte, also eine *Write-Operation* ausführen möchte. Ein General Call in Kombination mit einer Read-Operation ist nicht möglich, da diese nur zwischen zwei Teilnehmern des Systems stattfinden kann.

Außerdem wird im Adresspaket noch ein Bit verwendet, um zu definieren, ob eine Schreib-oder Lese-Operation auf dem Bus stattfinden soll. Findet eine Lese-Operation statt, dann ist dieses Bit gesetzt, für eine Schreib-Operation wird es gelöscht.

Es sollten keine Adressen des Aufbaus 1111 xxxx vergeben werden, da sie für zukünftige Zwecke reserviert sind. Effektiv bedeutet das, dass anstatt den vorher genannten 128 Adressen nur noch 118 zur freien Verfügung stehen. Das 9. Bit dient der Bestätigung und wird Acknowledge-Bit, kurz *ACK* genannt.

Wenn ein Slave erkennt, dass er durch das Adresspaket angesprochen wird, dann zieht er beim 9.Takt der SCL-Leitung seine SDA-Leitung auf Masse um zu signalisieren, dass er bereit ist. Verarbeitet der angesprochene Slave gerade Daten und kann nicht auf die Anfrage des Masters reagieren, dann bleibt seine SDA-Leitung auf high. Bei einem General Call empfangen alle Slaves, die ihre SDA-Leitung in der ACK-Phase auf Masse ziehen, das folgende Datenpaket.

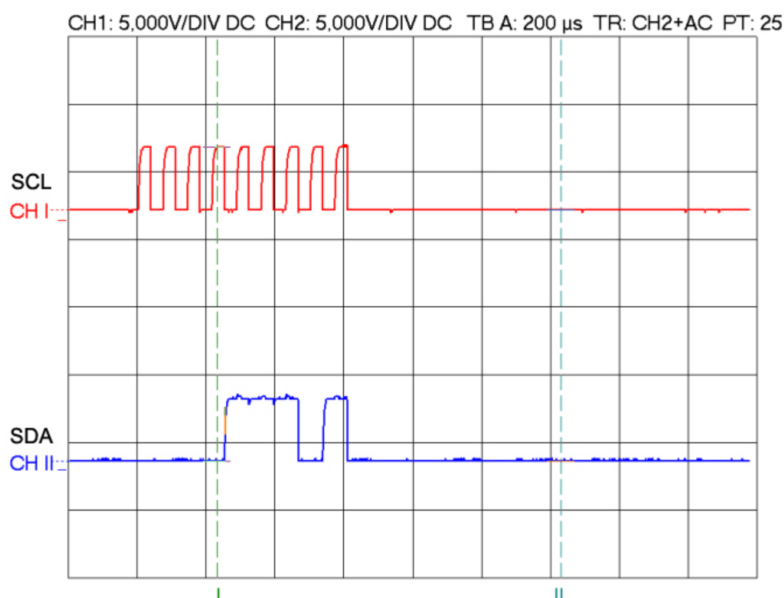
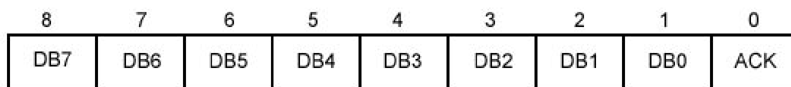


Abb. 61: TWI-Adresspaket - dargestellt mit Hilfe eines Oszilloskops

Die obige Grafik stellt die Übertragung eines Adresspakets mit der Slaveadresse 0000111 und einer SLA+W-Anweisung dar. Auf den ersten 4 Takten der Clockleitung werden vier Nullen übertragen, auf den nächsten 3 Takten folgen drei Einsen. Im nächsten Takt folgt eine Null für die SLA+W-Anweisung. Im letzten dargestellten Takt wird abschließend eine Eins übertragen. Dies geschieht, da der Slave die Adresse 0000111 als seine eigene erkannt hat, die SLA+W-Anweisung akzeptiert und die SDA-Leitung somit im neunten Takt auf high zieht, um ein Acknowledge an den Master zu senden.

Datenpaket:

Das Datenpaket ist 9 Bit lang und besteht aus einem Byte Daten und einem Bit, das für das ACK zuständig ist. Bei der Übertragung des Paketes wird immer zuerst das MSB (Most Significant Bit) gesendet.



8	7	6	5	4	3	2	1	0
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	ACK

Abb. 62: TWI Datenpaket

Das ACK wird wieder über das Ziehen der SDA-Leitung im 9.SCL Zyklus signalisiert. Wenn der Slave keine weiteren Daten mehr empfangen kann, dann muss er dies über ein NACK (not-acknowledged) deutlich machen, indem er die SDA-Leitung nicht auf Masse zieht.

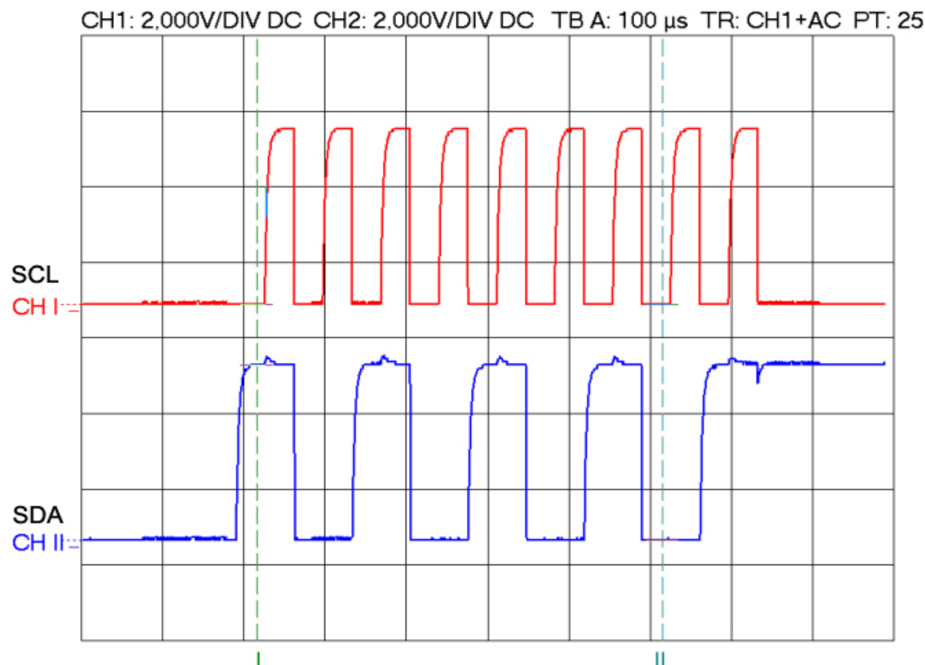


Abb. 63: TWI-Datenpaket - dargestellt mit Hilfe eines Oszilloskops

Die obige Grafik zeigt die Übertragung des Datenpakets 10101010 vom Master an den vorher adressierten Slave. Der Slave bestätigt den Eingang des Datenpakets im neunten Taktzyklus mit einem Acknowledge, indem er die SDA-Leitung auf high zieht.

4.12.3 Kommunikationsablauf

Die Pegeldiagramme zu den folgenden Aussagen befinden sich im Datenblatt des ATmega16 auf den Seiten 172 und 173. Die Übertragung von Daten über den TWI-Bus im Master-Slave-Modus verläuft nach folgendem Schema:

START - Bedingung, während die SCL-Leitung auf high ist und die SDA-Leitung von high in low übergeht

SLA-R/W - Bedingung: Adresse, Read-oder Write-Bit und ACK

DATENPAKET - 1 Datenbyte und ACK (ein oder mehrere Datenpakete)

STOP - Bedingung, während die SCL-Leitung auf high ist und die SDA-Leitung von low in high übergeht

Ein Adress- oder Datenbit ist nur gültig, wenn die SCL-Leitung sich im high-Zustand befindet und sich SDA währenddessen nicht ändert.
Eine "1" wird übertragen, wenn SCL und SDA high sind, eine "0" wird übertragen, wenn SCL high und SDA low ist.

4.12.4 Bitraten Generator

Der Two-Wire Bus kann mit zwei verschiedenen Übertragungsraten [twi2] betrieben werden. Im Standard-Mode sind 100 kBit/s möglich, im Fast-Mode 400 kBit/s. Die SCL-Frequenz ist abhängig vom CPU-Takt des I^2C-Bausteins und berechnet sich nach folgender Formel:

$$F_{SCL} = \frac{F_{CPU}}{16 + 2(TWBR) * 4^{TWPS}}$$

Es gilt zu beachten, dass die Übertragung auf größeren Strecken besser ist, wenn die Übertragungsrate geringer ist. Mit 400 kBit/s ist die Fehlerfreiheit nur noch im Zentimeterbereich garantierbar. Daher sind hohe Datenraten eher für I^2C-Bausteine in unmittelbarer Nähe zum Master zu empfehlen. Will man Distanzen von bis zu einem Meter erreichen, dann sollte die Übertragungsrate eher bei 100 kBit/s oder noch weniger liegen.
Beispiel:
Angenommen es soll eine TWI-Verbindung zwischen zwei Mikrocontrollern aufgebaut werden. Die Master-CPU taktet mit 7.3728 MHz und die Frequenz der SCL-Leitung soll 100 kBit/s betragen. Da der Prescaler erst dann nötig ist, wenn der Master mit über 20 MHz getaktet wird, ist er beim ATmega16 überflüssig (4^0 = 1), da dieser mit maximal 16 MHz getaktet werden kann. Laut obiger Formel ist es nun noch nötig, dem Mikrocontroller zur Berechnung der SCL-Frequenz eine Konstante zu übergeben, die im TWBR - TWI Bit Rate Register - steht. Diese wird folgendermaßen berechnet:

$$TWBR = \frac{\frac{F_{CPU}}{F_{SCL}} - 16}{2 * 4^{TWPS}}$$

Es ergäbe sich demnach der Wert 28,9 also 29, für das TWBR-Register.

4.12.5 TWI Register

TWBR - TWI Bit Rate Register

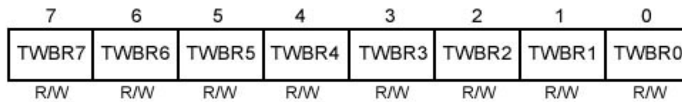


Abb. 64: TWBR - TWI Bit Rate Register

Das TWB-Register enthält den Divisionsfaktor, der zur Berechnung der Taktrate der SCL-Leitung erforderlich ist. Eine Tabelle mit Standardwerten findet sich in der TWI-Application Note (doc2564_TWI_AppNote.pdf) im Ordner Datenblätter.

TWDR - TWI Daten Register

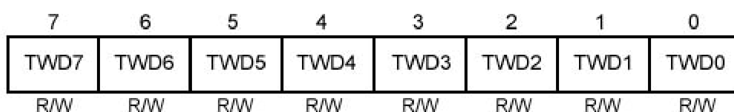


Abb. 65: TWDR - TWI Daten Register

Dieses Register enthält im *Transmit Mode* das Byte, das als nächstes gesendet werden soll, und im *Receive Mode* das Byte, das als letztes empfangen wurde.

TWAR - TWI Address Register

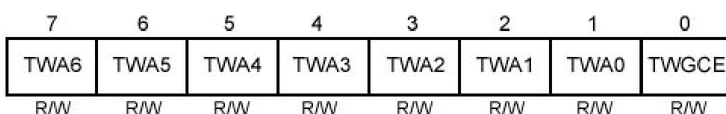


Abb. 66: TWAR - TWI Address Register

In den 7 high Bits wird die Adresse des Slaves hinterlegt. Anhand des TWGCE-Bit - TWI General Call Enable - wird bestimmt, ob der Slave auf einen General Call antwortet (TWGCE = 1) oder nicht.

TWCR - TWI Control Register

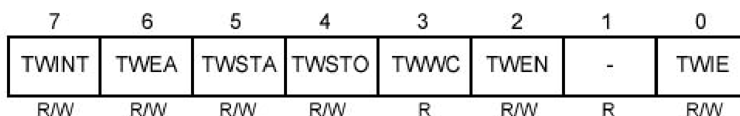


Abb. 67: TWCR - TWI Control Register

TWINT: TWI Interrupt Flag

Dieses Interruptflag wird automatisch gesetzt, sobald eine Aufgabe vom TWI abgearbeitet wurde und eine Reaktion der Software erwartet wird. Das Bit kann nur gelöscht werden, indem man es auf logisch 1 setzt. Wenn die TWI-Interruptroutine verwendet wird, ist darauf zu achten, dass das TWINT-Bit nach Verlassen der Routine nicht automatisch zurückgesetzt wird, sondern explizit im Code gelöscht werden muss. Sobald das TWINT-Bit gelöscht wird, ist der TWI-Bus wieder aktiv, daher müssen zu diesem Zeitpunkt alle vorherigen TWI-Aufgaben abgeschlossen sein.

TWEA: TWI Enable Acknowledge

Ist der Wert des Bits 1, dann ist TWEA dafür zuständig, dass der Slave zum richtigen Zeitpunkt seine Bestätigung (ACK) schickt, wenn er seine Adresse oder einen General Call erkannt hat, oder sich schon im Datenaustausch mit einem Master befindet. Setzt man TWEA auf 0, dann ist die Verbindung zwischen diesem Slave und dem Master getrennt, da der Slave nicht mehr bestätigen kann.

TWSTA: TWI Start Condition

Der Master zeigt mit Setzen dieses Bits an, dass er eine neue Übertragung starten will. Daraufhin prüft die Hardware, ob die Leitung gerade benutzt wird. Ist dies nicht der Fall, dann wird die START-Sequenz ausgeführt, ansonsten wartet das TWI auf die nächste STOP-Bedingung und führt anschließend erst das START aus. Wurde die START-Bedingung ausgeführt, muss das TWSTA danach im Code gelöscht werden.

TWSTO: TWI Stop Condition

Durch Setzen der TWSTO-Bits durch den Master wird die STOP-Bedingung ausgelöst. Wurde die Bedingung erfolgreich ausgeführt, wird das Bit automatisch wieder gelöscht.

TWWC: TWI Write Collision Flag

Erkennt das TWI-Modul eine Schreibkollision, wird dieses Bit gesetzt. Dieser Fehler tritt auf, wenn ein Zugriff auf das TWI Daten Register ausgeführt wird, obwohl das TWINT-Bit nicht gesetzt ist. Das Bit kann nicht im Code gesetzt oder gelöscht werden.

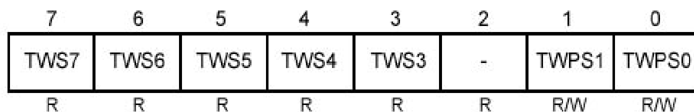
TWEN: TWI Enable

Erst wenn TWEN auf 1 gesetzt wird, können Operationen auf dem TWI-Bus ausgeführt werden. Wird das Bit gelöscht, dann werden sämtliche Operationen des TWI-Moduls sofort abgebrochen und TWI wird deaktiviert.

TWIE: TWI Interrupt Enable

Ist dieses Bit auf high, kann der TWI-Interrupt auftreten, solange TWINT gesetzt ist.

TWSR - TWI Status Register



7	6	5	4	3	2	1	0
TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
R	R	R	R	R	R	R/W	R/W

Abb. 68: TWSR - TWI Status Register

TWPS1:TWPS0 TWI Prescaler

Diese zwei Bits dienen der Einstellung des Prescalers für den Takt der auf SCL anliegt. Da der maximale CPU-Takt des ATmega16 bei 16MHz liegt, ist ein Prescaler nicht nötig.

TWS7:TWS3 TWI Status

In diesen 5 Bits legt das TWI-Modul den jeweiligen Status fest. Anhand der gesetzten Bits kann man erkennen, ob ein Fehler bei der Ausführung der TWI-Operation aufgetreten ist, oder ob der Befehl ohne Probleme ausgeführt wurde. Die Status-Codes sind abhängig von dem aktiven TWI-Modus (siehe Kommunikationsarten).

4.12.6 TWI Initialisierung

Die Pull-Up Widerstände der SCL und SDA Pins müssen gesetzt werden. Außerdem muss das TWBR-Register befüllt werden, um eine Übertragungsrate festzulegen.

```
// Pull-Up Widerstände von SCL und SDA setzen
DDRC &= ~((1<<SCL) | (1<<SDA));
PORTC |= (1<<SCL) | (1<<SDA);
// benötigten Wert zur Datenratenerzeugung in das TWBR schreiben
TWBR = TWBR_VAL;
```

4.12.7 TWI Übertragungsarten

Um TWI-spezifische Definitionen wie zum Beispiel TW_WRITE, im Code verwenden zu können, muss <util/twi.h> inkludiert werden.

4.12.7.1 Master Transmitter (MT)

Der Master sendet Daten an den oder die Receiver. Es kann entweder ein einzelner Slave anhand dessen eindeutiger Adresse zur Kommunikation aufgefordert werden, oder es werden alle Slaves über einen General Call angesprochen.
Die Kommunikation läuft nach folgendem Schema ab:

Die TWI-Schnittstelle wird aktiviert, TWINT wird gelöscht und das das Bit für die START-Bedingung wird gesetzt.

```
TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
```

War die Ausführung der START-Bedingung erfolgreich, wird TWINT automatisch gesetzt und die nächste Anweisung kann ausgeführt werden.

```
while (!(TWCR & (1<<TWINT)));
```

Der Master sendet das Adresspaket, das die Adresse des Slaves und das MT-Kommando SLA+W enthält. Die Adresse muss um 1 nach links geshiftet werden, da die SLA+W Anweisung in Bit 0 steht.

```
TWDR = (SLAVEADR<<1) | TW_WRITE;
```

Um den nächsten Befehl senden zu können, muss TWINT gelöscht werden.

```
TWCR = (1<<TWINT) | (1<<TWEN);
```

Wurde die Adresse erfolgreich übertragen und hat der Slave mit Acknowledge geantwortet, wird TWINT durch die Hardware wieder gesetzt. Daher sollte vor der Übertragung des Datenpakets TWINT überprüft werden.

```
while (!(TWCR & (1<<TWINT))) ;
TWDR = Wert;
```

War die Übertragung des Datenpakets erfolgreich, antwortet der Slave mit Acknowledge.
Der Master kann nun weitere Datenpakete schicken, oder die Kommunikation mit einer STOP-Bedingung beenden.

```
TWCR = (1<<TWINT) | (1<<TWSTO) | (1<<TWEN);
```

Anhand der Statuscodes (die komplette Liste findet sich im ATmega16 Datenblatt Seite 186) lässt sich überprüfen, ob die Kommunikation reibungslos verläuft. Da die IDs der Statuscodes in den Bits 7 bis 3 gespeichert werden, muss man die Bits 2 bis 0 bei der Überprüfung maskieren.

```
// Überprüfung, ob START-Bedingung erfolgreich war
if ((TWSR & 0xF8) == 0x08){
        // do something
}
else  if ((TWSR & 0xF8) == 0x08){
        // do something different
}
```

Master Transmitter Statuscodes:

0x08

START-Bedingung wurde verschickt.

0x18

Adresspaket das Slaveadresse bzw. General Call und SLA+W enthält wurde übertragen und von dem adressierten Slave bzw. von allen General Call-akzeptierenden Slave mit ACK bestätigt.

0x28

Der im TWDR-Register gespeicherte Wert wurde verschickt, und der Slave hat dessen Eingang durch ACK quittiert.

4.12.7.2 Slave Receiver (SR)

Jeder Slave bekommt nach der Initialisierung eine eindeutige Adresse zugewiesen, und es muss entschieden werden, ob dieser Slave auf einen General Call antwortet. Diese Informationen werden in das TWI-Adressregister geschrieben. Es ist darauf zu achten, dass die Adresse um 1 nach links geshiftet werden muss.

```
TWAR = (SLAVEADR<<1) | (1<<TWGCE);
```

Anschließend wird die TWI-Schnittstelle aktiviert, das TWINT-Bit gelöscht, der TWI-Interrupt aktiviert und das TWEA-Bit gesetzt, das es dem Slave ermöglicht, ACK an den Master zu senden.

```
TWCR = (1<<TWEN)|(1<<TWINT)|(1<<TWEA)|(1<<TWIE);
```

Der Aufruf der Funktion, die ausgeführt werden soll, wenn der Master diesen Slave zur Kommunikation auffordert, ist in der Interruptroutine ISR(TWI_vect) zu platzieren.

```
ISR(TWI_vect)
{
        SlaveReceiver();
}
```

Schickt der Master dann das Datenpaket mit der Adresse des Slaves bzw. einen General Call und die SLA+W Anweisung, dann bestätigt der Slave mit einem Acknowledge. Danach muss der Eingang der Daten

```
wert = TWDR;
```

wiederum durch ein Acknowledge bestätigt werden. Diese Bestätigung erfolgt, solange Daten eintreffen, bis schlussendlich vom Master eine STOP-Bedingung eintrifft, oder der Slave keine Daten mehr empfangen kann - er sendet dann ein NACK. Es muss darauf geachtet werden, dass nach jeder Bestätigung des Slaves das TWINT-Bit durch Setzen auf 1 gelöscht werden muss.

Anhand der Statuscodes (eine Auflistung aller SR-Statuscodes findet sich im ATmega16 Datenblatt Seite 192) die im TWSR-Register hinterlegt werden, kann überprüft werden, ob die Kommunikation reibungslos verläuft oder ob Fehler auftreten.

Slave Receiver Statuscodes:

0x60

Der Slave erkennt seine Adresse und erhält die SLA+W Anweisung des Masters. Daraufhin bestätigt er mit ACK

0x70

Ein General Call wurde erkannt und die Kommunikationsbereitschaft durch ACK bestätigt.

0x80

Der explizit adressierte Slave empfängt das vom Master gesendete Datenbyte und bestätigt durch ACK. Dies wird solange wiederholt, bis die STOP-Bedingung eintrifft.

0x90

Zu Beginn der Kommunikation schickte der Master einen General Call. Die erfolgreiche Datenübertragung wird vom General Call-fähigen Slave durch ACK bestätigt.

0xA0

STOP-Bedingung wurde erkannt

4.12.7.3 Master Receiver (MR)

In diesem Kommunikationsmodus fordert der Master einen Slave dazu auf, Daten zu übertragen. Hier ist kein General Call möglich, da dann alle reagierenden Slaves gleichzeitig Daten an den Master schicken würden, und dieser nicht darauf reagieren könnte. Die Kommunikation zwischen Master und Slave verläuft nach folgendem Schema:

Die TWI-Schnittstelle wird aktiviert, TWINT wird gelöscht, durch Setzen von TWEA wird sichergestellt, dass der Master ACK und NACK an den Slave schicken kann und das das Bit für die START-Bedingung wird gesetzt.

```
TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN) | (1<<TWEA);
```

War die Ausführung der START-Bedingung erfolgreich, wird TWINT automatisch gesetzt und die nächste Anweisung kann ausgeführt werden.

```
while (!(TWCR & (1<<TWINT)));
```

Der Master sendet das Adresspaket, das die Adresse des Slaves und das MR-Kommando SLA+R enthält. Die Adresse muss um 1 nach links geshiftet werden, da die SLA+R Anweisung in Bit 0 steht.

```
TWDR = (SLAVEADR<<1) | TW_WRITE;
```

Um den nächsten Befehl senden zu können, muss TWINT gelöscht werden.

```
TWCR = (1<<TWINT) | (1<<TWEN);
```

Antwortet der Slave mit ACK wird das TWINT-Bit von der Hardware gesetzt, daher sollte der Zustand dieses Bits vor der nächsten Anweisung überprüft werden.

```
while (!(TWCR & (1<<TWINT)));
```

Nun muss TWINT wieder gelöscht werden.

```
TWCR = (1<<TWINT) | (1<<TWEN);
```

Im nächsten Schritt empfängt der Master die vom Slave gesendeten Daten.

```
wert= TWDR;
```

Der Master sendet ein ACK zum Slave.
Das TWINT-Bit wird wieder von der Hardware gesetzt und es findet die Überprüfung statt, ob der Slave Daten geschickt hat. TWINT wird in der Software gelöscht und der Master kann das nächste Datenpaket aus dem TWDR-Register lesen. Dies wird solange wiederholt, bis der Master das letzte Datenpaket empfangen hat. Dessen Eingang muss er dann durch ein NACK (Not Acknowledged) bestätigen.
Danach sendet der Master die STOP-Bedingung und der Datenaustausch wird beendet.

```
TWCR = (1<<TWINT) | (1<<TWSTO) | (1<<TWEN);
```

Anhand der Statuscodes (eine Auflistung aller MR-Statuscodes findet sich im ATmega16 Datenblatt Seite 189) kann überprüft werden, ob die Kommunikation zwischen empfangendem Master und sendendem Slave erfolgreich war.

Master Receiver Statuscodes:

0x08

START-Bedingung wurde verschickt.

0x40

SLA+R und Adresse des gewünschten Slaves wurde übertragen und mit ACK durch den Slave bestätigt.

0x50

Das Datenpaket wurde empfangen und ACK an den Slave geschickt

0x58

Das letzte Datenpaket wurde empfangen und NACK an den Slave gesendet.

4.12.7.4 Slave Transmitter (ST)

Der Slave überträgt, nachdem er vom Master durch SLA+R und die spezifische Slaveadresse dazu aufgefordert wurde, Daten an eben diesen.
Nachdem das TWI für den Slaveteilnehmer initialisiert wurde, bekommt dieser eine eindeutige Adresse, die zusammen mit dem Bit für General Call-Akzeptanz in das TWI-Adressregister gespeichert wird.
Es ist darauf zu achten, dass die Adresse um 1 nach links geshiftet werden muss.

```
TWAR = (SLAVEADR<<1) | (1<<TWGCE);
```

Anschließend wird die TWI-Schnittstelle aktiviert, das TWINT-Bit gelöscht, der TWI-Interrupt aktiviert und das TWEA-Bit gesetzt, das es dem Slave ermöglicht, ACK an den Master zu senden.

```
TWCR = (1<<TWEN)|(1<<TWINT)|(1<<TWEA)|(1<<TWIE);
```

Der Aufruf der Funktion, die ausgeführt werden soll, wenn der Master den Slave zur Kommunikation auffordert, ist in der Interruptroutine ISR(TWI_vect) zu platzieren.

```
ISR(TWI_vect)
{
        SlaveTransmitter();
}
```

Erkennt der Slave, dass er zum Senden von Daten durch den Master aufgefordert wird, bestätigt er den Empfang seiner Adresse und der SLA+R Bedingung durch ACK. Die Hardware setzt daraufhin das TWINT-Bit. Bevor die Kommunikation wieder aufgenommen werden kann, muss dieses Bit wieder gelöscht werden.

```
TWCR=(1<<TWEN)|(1<<TWINT)|(1<<TWEA)|(1<<TWIE);
```

Das TWI-Datenregister wird mit dem zu sendenden Byte gefüllt und übertragen.

```
TWDR = wert;
```

Anschließend erhält der Slave eine Bestätigung (ACK) durch den Master. Wird während der Übertragung des Datenbytes das TWEA-Bit des Slaves gelöscht, signalisiert dies das Senden des letzten Datenpakets. Der Master sollte daraufhin mit NACK anstatt ACK antworten. Tut er dies nicht, wird der Master anschließend vom Slave ignoriert. Die Kommunikation wird beendet, wenn der Slave die vom Master transferierte STOP-Bedingung erkennt. Anhand der Statuscodes (eine Auflistung aller ST-Statuscodes findet sich im ATmega16 Datenblatt Seite 195) kann überprüft werden, ob die Kommunikation zwischen empfangendem Master und sendendem Slave erfolgreich war.

Slave Transmitter Statuscodes:

0xA8

Der Slave erkennt seine Adresse und die SLA+R Anweisung und bestätigt durch ACK.

0xB8

Das in TWDR gespeicherte Byte wurde übertragen und ACK wurde vom Master empfangen.

0xC0

Das in TWDR gespeicherte Byte ist das letzte das übertragen wurde. Der Master hat dies erkannt und bestätigt durch NACK.

0xC0

Das in TWDR gespeicherte Byte ist das letzte das übertragen wurde. Der Master hat dies nicht erkannt und bestätigt durch ACK.

4.13 Echtzeituhr RTC 4513

Die RTC 4513[RTC] hat einen eingebauten 32KHz Quarz, weshalb es nicht mehr notwendig ist, einen externen Quarz zuzuschalten. Die Echtzeituhr wird über 3 Pins gesteuert:

CLK - vom Mikrocontroller eingehendes Taktsignal
CE - Aktivierung der RTC
DATA - bidirektionaler Datenaustausch zwischen Mikrocontroller und RTC

Die Echtzeituhr wurde auf der Entwicklungsplatine nicht mit einer Batterie versehen, wodurch alle Daten bei Ausbleiben der Stromzufuhr verloren gehen.

4.13.1 Register der RTC 4513

Address	Register name	D3 (MSB)	D2	D1	D0 (LSB)	Count range	Remarks
0	S_1	s8	s4	s2	s1	0~9	1-second digit register
1	S_{10}	fo	s40	s20	s10	0~5	10-second digit register
2	MI_1	mi8	mi4	mi2	mi1	0~9	1-minute digit register
3	MI_{10}	fr	mi40	mi20	mi10	0~5	10-minute digit register
4	H_1	h8	h4	h2	h1	0~9	1-hour digit register
5	H_{10}	fr	PM/AM	h20	h10	0~1,2	10-hour digit register
6	D_1	d8	d4	d2	d1	0~9	1-day digit register
7	D_{10}	fr	*	d20	d10	0~3	10-day digit register
8	MO_1	mo8	mo4	mo2	mo1	0~9	1-month digit register
9	MO_{10}	fr	*	*	mo10	0~1	10-month digit register
A	Y_1	y8	y4	y2	y1	0~9	1-year digit register
B	Y_{10}	y80	y40	y20	y10	0~9	10-year digit register
C	W	fr	w4	w2	w1	0~6	Day-of-the-week register
D	C_D	30ADJ.	IRQ-F	CAL/HW	HOLD	–	Control register D
E	C_E	t1	t0	INT/STND	MASK	–	Control register E
F	C_F	TEST	24/12	STOP	RESET	–	Control register F

Tab. 05: RTC 4513 Register

4.13.2 Initialisierung der RTC

In der Initialisierung werden die Pins, an denen CE und CLK angeschlossen sind auf Ausgang gesetzt. Der Zustand des DATA-Pins wird nicht gesetzt, da dieser sich ändert, je nachdem, ob der Mikrocontroller Daten an die Echtzeituhr sendet, oder Daten von dieser empfängt. Anschließend wird die Power-on Prozedur ausgeführt.

4.13.3 Power-on Prozedur

Um die Echtzeituhr ansteuern zu können, muss nach dem Anschließen der Stromzufuhr eine Initialisierung durchgeführt werden, da sich zu diesem Zeitpunkt alle Register in einem undefinierten Zustand befinden. Diese verläuft nach folgendem Muster:

- Register C_F füllen:

  D0 und D1 = 1

  D2 = 0 oder 1 (12h oder 24h Modus)

  D3 = 0 (darf NIE 1 sein)
- Register S_1 bis W füllen
- Register C_F füllen:

  D0 und D1 = 0

  D2 = 0 oder 1 (12h oder 24h Modus)

  D3 = 0 (darf NIE 1 sein)
- Register C_D füllen:

  D0 = 0

  D1 = 1 oder 0

  (wenn 0, dann werden die Register D1 bis Y10 nicht verwendet und können als RAM benutzt werden)

  D2 und D3 = 0

4.13.4 RTC Write-Mode

Um Daten an die Echtzeituhr senden zu können, müssen folgende Schritte im Code implementiert werden:

- CE (im Portregister) auf high setzen
- DATA (im DDR-Register) als Ausgang setzen, da der µC Daten sendet
- Nibble = 0011 auf DATA (im Portregister) abbilden, das der Echtzeituhr signalisiert, dass der Write-Modus gestartet werden soll. Es muss darauf geachtet werden, dass immer zuerst das LSB übertragen wird. Die RTC empfängt das Signal der DATA-Leitung sobald CLK in den high-Zustand wechselt.
  - DATA (im Portregister) setzen
  - CLK (im Portregister) auf high ziehen
  - CLK (im Portregister) auf low ziehen
- Adresse des Registers der RTC festlegen, das die Daten speichern soll.
  Die Adresse wird nach obigem Muster als 4-bit Nibble, mit dem Least Significant Bit zuerst, an die Uhr übertragen.
- Anschließend folgt das Datennibble, das in das entsprechend adressierte Register gespeichert werden soll.
- Die Adresse des aktuellen Registers wird jetzt um 1 inkrementiert, und dieser Vorgang mit Abschluss jedes 4-Bit Datenpakets wiederholt, weshalb mehrere Register innerhalb der gleichen Write-Anweisung beschrieben werden können.
- Sollen keine Daten mehr an die RTC gesendet werden, wird die CE-Leitung (im Portregister) auf low gezogen.

4.13.5 RTC Read-Mode

Um Daten von der Echtzeituhr empfangen zu können, müssen folgende Schritte im Code implementiert werden.

- CE (im Portregister) auf high setzen
- DATA (im DDR-Register) als Ausgang setzen, da der µC Daten sendet
- Nibble = 1100 auf DATA (im Portregister) abbilden, das der Echtzeituhr signalisiert, dass der Read-Modus gestartet werden soll. Es muss darauf geachtet werden, dass immer zuerst das LSB übertragen wird.
- Adresse des Registers der RTC festlegen, aus dem das erste Datenpaket gelesen werden soll.
- DATA (im DDR-Register) als Eingang setzen, da der µC ab jetzt Daten empfängt
- Der Zustand der DATA-Leitung wird über das entsprechende PIN-Register abgefragt:
    - CLK (im Portregister) auf high ziehen
    - DATA (im Pinregister) auslesen
    - CLK (im Portregister) auf low ziehen
- Dieser Vorgang wird wiederholt, bis das empfangene Datennibble vollständig ist.
- Da auch im Read-Mode die Registeradressen inkrementiert werden, können die Inhalte mehrerer, aufeinander folgender Register in einer Read-Anweisung ausgelesen und gespeichert werden.
- Sollen keine Daten mehr von der RTC an den Mikrocontroller gesendet werden, wird die CE-Leitung (im Portregister) auf low gezogen.

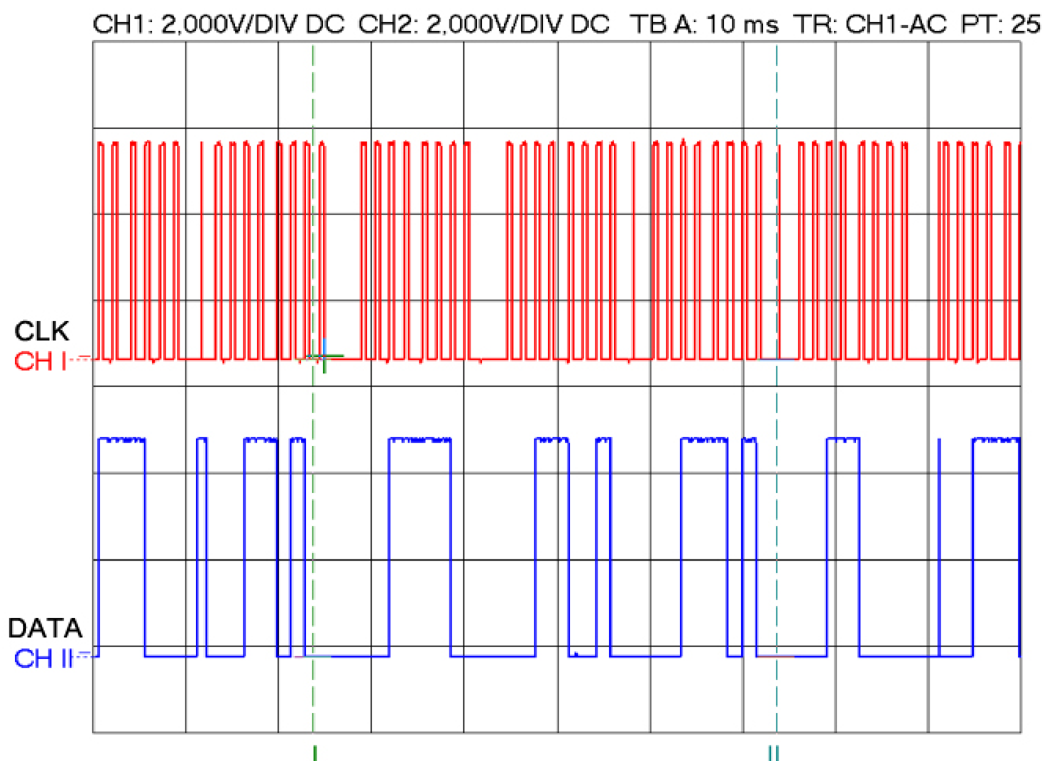


Abb. 69: Darstellung des Auslesens der Register S1 bis W mit Hilfe eines Oszilloskops

4.13.6 RTC Read-Flag

In der im Kapitel 4.13.1 dargestellten Registertabelle sieht man, dass das MSB der Register
S10, MI10, H10, D10, MO10 und W als Statusbit (Read Flag = fr) verwendet wird. Es wird automatisch auf 1 gesetzt, wenn im Sekundenregister während des Lesens des Registerinhalts ein Übertrag stattfand. Dieser Übertrag hat zur Folge, dass alle bis dahin gelesenen Register erneut ausgelesen werden müssen, um die Genauigkeit der Zeitanzeige zu gewährleisten.

5 Hardware

5.1 Schaltzeichen

Um Platinen bauen zu können, ist es nötig Schaltpläne lesen zu können. Wenn man noch nie in Kontakt damit gekommen ist, dann können die Schaltzeichen und Leiterbahnen ziemlich verwirrend sein. Daher werden in diesem Kapitel die wichtigsten elektronischen Bauteile dargestellt und erläutert.

Versorgungsspannung und Masse

Abb. 70: Versorgungsspannung

Abb. 71: Masse

Elektronische Bauteile müssen immer in einem geschlossenen Stromkreis aufgebaut sein. Dies ist entweder bei einem direkten Anschluss der Fall oder wenn einer der beiden Anschlüsse über einen Pin des Mikrocontrollers repräsentiert wird.

Kondensatoren

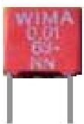


Abb. 72: Folienkondensator

Abb. 73: Keramikkondensator

Abb. 74:Schaltzeichen ungepolter Kondensatoren

Abb. 75: Elektrolytkondensator

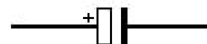


Abb. 76: Schaltzeichen gepolter Kondensatoren

Elektrolytkondensatoren haben eine definierte Polarität! Das heißt, sie können nur in eine Richtung angeschlossen werden, bei der das kurze Beinchen an der Masse hängen muss. Allerdings haben diese Bauteile auch einen nicht zu übersehenden Aufdruck mit Minuszeichen an der Seite des kurzen Beinchens. Wird das beim Einbau nicht beachtet, besteht beim Anschließen der Versorgungsspannung akute Explosionsgefahr des Kondensators. [kond]

Widerstände

Abb. 77: Widerstand

Abb. 78: Schaltzeichen Widerstand

Widerstände[wi] sind in jeder Schaltung unerlässlich. Beim Einbau von Widerständen sollte man immer zweimal überprüfen, ob man auch den richtigen in der Hand hat. Es gibt mehrere Möglichkeiten herauszufinden, welchen Wert ein Widerstand hat. Zum Einen, durch ein Tool auf dieser Homepage: http://www.okaphone.nl/calc/widerstand.shtml
Oder, was noch leichter ist, mit einem Messgerät, das man auf Widerstandsmessung einstellt, so denn man eines greifbar hat. Dann gibt es noch die Möglichkeit, sich beim Elektronikversand ein Vitrometer (Widerstandsuhr) zu bestellen.

Potentiometer

Abb. 79: Cermet-Potentiometer

Abb. 80: Drehpotentiometer

Abb. 81: Schaltzeichen Potentiometer

Ein Potentiometer[poti] ist ein veränderlicher Widerstand, auch Spannungsteiler genannt. Ein Schleifkontakt im Inneren des Potis teilt den Gesamtwiderstand elektrisch in zwei Teilwiderstände. Typischer Einsatz für ein Poti wäre ein Lautstärkenregler.

Dioden

Abb. 82: Diode

Abb. 83: Schaltzeichen Diode

Dioden[dio] sind Halbleiter, in denen der Strom nur in eine Richtung fließen kann. Darum muss auch hier beim Einbau wieder auf die Polung geachtet werden. Wird die Diode falsch herum in den Stromkreis eingebaut, dann fließt durch dieses Bauteil einfach kein Strom mehr.
Ein typisches Schaltungselement, bei dem Dioden verwendet werden ist der Brückengleichrichter. Hier wird die anliegende Wechselspannung in pulsierende Gleichspannung umgewandelt, die anschließend, durch einen nachgeschalteten Kondensator noch geglättet wird.

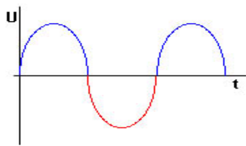


Abb. 84: Wechselspannung

Abb. 85: Schaltzeichen Brückengleichrichter

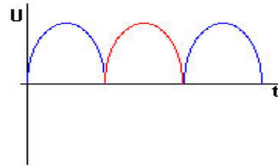


Abb. 86: pulsierende Gleichspannung

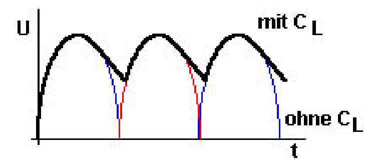


Abb. 87: durch Kondensator veränderte Gleichspannung

Abb. 88: Brückengleichrichter

Leuchtdioden

Abb. 89: rote Leuchtdiode

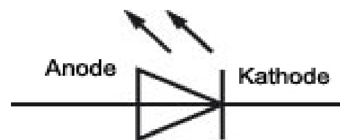


Abb. 90: Schaltzeichen Leuchtdiode

Die Leuchtdiode hat eine Durchlassrichtung. Das heißt, sie muss richtig angeschlossen werden, um auch zu funktionieren. Die LED hat zwei unterschiedlich lange Beinchen - das lange ist der Anschluss zur Anode, das kurze der Anschluss zur Kathode. Das heißt also in der Schaltung erfolgt folgender Aufbau:

VCC ➔ Anode (langes Beinchen)   Kathode (kurzes Beinchen) ➔GND

Irgendwo in diesem Aufbau muss der Vorwiderstand eingebaut werden.
Jede Leuchtdiode braucht einen Vorwiderstand. Wird dieser vergessen, dann wird die LED heiß, verändert ihre Farbe, und brennt durch. Es ist egal, ob der Vorwiderstand zwischen Versorgungsspannung und LED oder zwischen LED und Masse liegt.

Spannungsregler

Abb. 91: Spannungsregler

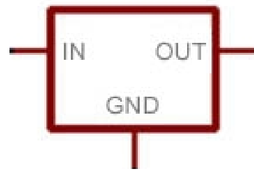


Abb. 92: Schaltzeichen Spannungsregler

Spannungsregler dienen dazu eine anliegende Spannung auf einen bestimmten Wert zu beschränken. Da es verschiedene Bautypen gibt, ist es immer ratsam, die Pinbelegung anhand des entsprechenden Datenblattes zu überprüfen..

Quarze

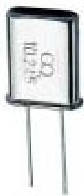


Abb. 93: Quarz

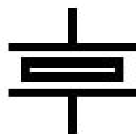


Abb. 94: Schaltzeichen Quarz

Abb. 95: Quarzoszillator

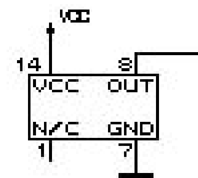


Abb. 96: Schaltzeichen Quarzoszillator

Quarze dienen dazu, dem Chip eine exakte Taktfrequenz zu geben. Zwar haben die meisten Chips einen eingebauten Takt, doch entspricht dieser oft nicht den später benötigten Anforderungen und ist außerdem zu ungenau..

Neben normalen Quarzen gibt es noch Quarzoszillatoren. Diese benötigen im Unterschied zu normalen Quarzen nicht mehr die Zuschaltung von Kondensatoren zwischen Quarz und Masse, sondern können, wie man am Schaltzeichen sieht, direkt über ihre Beinchen mit VCC, Masse und Chip verbunden werden. Das N/C Beinchen ist Not Connected - also mit nichts verbunden und dient nur zur Stabilisierung des Bauteils.

Verwendet man in der Schaltung keinen Quarzoszillator ,sollte der Quarz immer über 2 Stützkondensatoren (22pF) mit der Masse verbunden sein, da ansonsten Störungen in der Schwingung auftreten können.

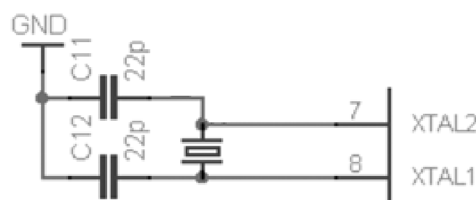


Abb 97: Zuschaltung von Stützkondensatoren zum Taktgeber

Operationsverstärker

Eine ausführliche Beschreibung über die Funktionsweise von Operationsverstärkern findet sich im Tutorial unter dem Überpunkt "Programmierung - Operationsverstärker".

Abb. 98: Operationsverstärker

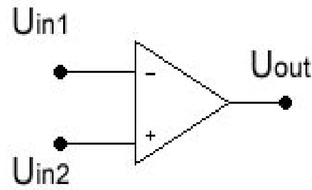


Abb. 99: Schaltzeichen Operationsverstärker

Temperatursensor

Eine ausführliche Beschreibung der verschiedenen Sensortypen findet sich im Tutorial unter dem Überpunkt "Programmierung - Temperatursensor".

Abb. 100: Heißleiter

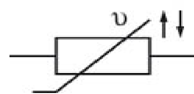


Abb. 101: Schaltzeichen Heißleiter

Abb. 102: Kaltleiter

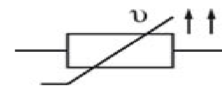


Abb. 103: Schaltzeichen Kaltleiter

Summer

Piezo-Summer erzeugen einen konstanten nicht-veränderbaren Dauerton.
Der Ton von Piezo-Schallwandlern ist veränderbar. Piezo-Schallwandler besitzen eine Polung!

Abb. 104: Piezo-Summer

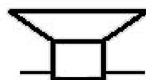


Abb. 105: Schaltzeichen Piezo-Summer

Abb. 106: Piezo-Schallwandler

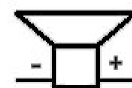


Abb. 107: Schaltzeichen Schallwandler

5.2 Hardware löten

5.2.1 Platinenbau

Unter *Schaltpläne/Evertool* finden sich der Schaltplan und die Bauteilliste für die Evertool-Platine. Eine detaillierte Beschreibung des Aufbaus und der Inbetriebnahme des Evertools ist auf der Seite

http://www.siwawi.arubi.uni-kl.de/avr_projects/evertool/

abrufbar.

Der Bau der Platine sollte gut überlegt sein, da der Aufbau für Anfänger ca. 20h dauert und die benötigten Teile bei ca. 20€ liegen. Kann man auf das Debuggen verzichten, dann bietet es sich an, einen *AVR ISP* für ca. 15-40€ zu kaufen. Will man seinen Code debuggen, dann gibt es außer der Evertool-Platine noch die Möglichkeit sich das *ATMEL JTAG ICE MK2* für ca. 330€ zuzulegen. Das MK2 hat trotz des hohen Preises einige Vorteile gegenüber der Evertool-Platine. Man hat keine Arbeit mit dem Aufbau der Hardware, sondern schließt die MK2 Steckverbindung einfach an die Entwicklungsplatine an. Es fungiert dann als ISP und Debugger. Das Debuggen ist außerdem um einiges schneller als mit dem Evertool. Im direkten Vergleich braucht das Evertool für einen Step im Code etwas weniger als 1 Sekunde. Das MK2 führt dagegen den Step *sofort* aus. Ein weiteres Feature des MK2 ist das Debuggen über Debug Wire. Die neueren Chips der Firma ATMEL verfügen über Debug Wire, d.h. die Debugleitung wird mit dem Reset-Pin des Chips verbunden und darüber wird der Code debugged.

5.2.2 Schaltpläne lesen

Schaltpläne können auf den ersten Blick verwirrend sein, gerade, wenn man sich noch nie damit auseinander gesetzt hat. Daher finden sich im Kapitel "Schaltzeichen" sämtliche, zum Bau der Entwicklungsplatine benötigten Bauteile, samt einer kurzen Beschreibung und deren Schaltzeichen. Bevor man mit dem löten beginnt, sollte man sich zuerst mit dem Schaltplan vertraut machen.
Leider sind die meisten Pläne nicht so aufgebaut, dass man die Anordnung der Bauteile direkt für die Platine übernehmen kann, weshalb es im ersten Schritt hilfreich ist, sich eine grobe Skizze der Platine zu machen, auf der andeutungsweise die Bauteile eingezeichnet sind. Dann sollte man sich einen Anfangspunkt suchen, an dem man mit dem Aufbau beginnt. Dafür bietet sich meistens die Stromversorgung an, da man dann Schrittweise testen kann, ob die gelöteten Bauteile funktionieren.

5.2.3 Löten

5.2.3.1 Grundlagen

Zum Löten benötigt man zunächst einen Lötkolben und Lötzinn. Beim Lötzinn ist es wichtig, dass dieser ein Flussmittel enthält. Bis vor kurzem war Lötzinn noch bleihaltig, aber seit März 2006 sind giftige Substanzen in elektronischen Bauteilen laut dem ElektroG §5 [elge] verboten. Der Lötkolben sollte mindestens auf 300°C erhitzt werden können, besser 340°C.

Tipps:

- Bevor ein Bauteil angelötet wird, sollte man die Lötkolbenspitze mit einem Hauch Lötzinn betupfen. Dadurch schmilzt das Lötzinn, das man mit dem Beinchen verbinden will, schneller.
- Den Lötzinn an das Beinchen des zu lötenden Teiles halten und dann die Lötkolbenspitze so halten, dass sowohl der Zinn schmilzt, als auch das Beinchen erhitzt wird. So verbindet sich beides besser.
- Das Lötzinn glänzt solange es heiß ist und wird matt sobald es erkaltet.
- Das Löten muss schnell gehen, da viele Bauteile solch hohe Temperaturen nicht über einen längeren Zeitraum hinweg vertragen.
- Der Lötzinn und das Beinchen sollten nicht mit der Spitze des Lötkolbens, sondern mit dessen Breitseite erhitzt werden, da die Spitze einen nicht ausreichend großen Wärmeübertrag bietet.
- Nach jedem Lötvorgang sollte die Spitze des Kolbens an einem nassen Schwamm abgestrichen werden.

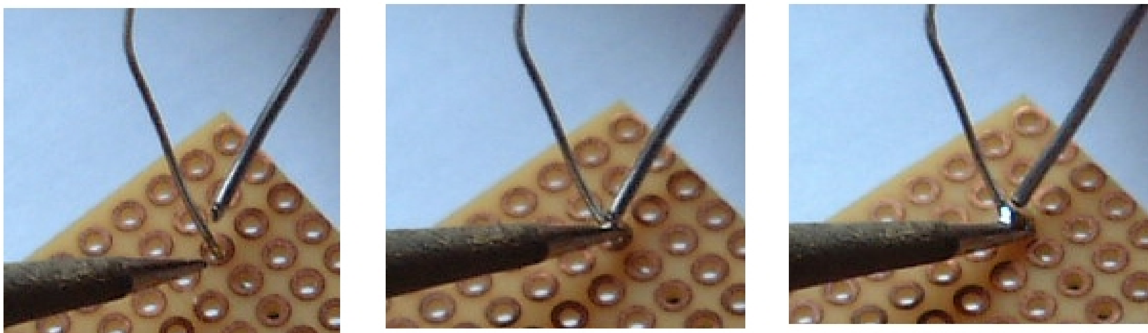


Abb. 108 Lötvorgang

Die abgebildeten Fotos zeigen das Anlöten eines Widerstandsbeinchens. Zuerst erhitzt man das Beinchen ein wenig mit dem Lötkolben, dann wird der Zinn zugleich an Beinchen und Lötkolben gehalten, wodurch sich das heiße Lötzinn mit dem Beinchen und dem Lötauge der Platine verbindet, wie auf dem letzten Bild zu sehen ist.

Die Bauteile einer Selbstbauplatine werden normalerweise auf einer einseitig beschichteten Lochrasterplatine mit einem Lochdurchmesser von 1/10 inch, also 2,54mm angebracht. Jedes Loch auf der Platine ist von einem Kupferring umgeben, dem so genannten Lötauge. Erst durch das Kupfer kann sich das

Lötzinn mit der Platine verbinden, das Bauteil sicher befestigt und einen leitenden Kontakt hergestellt werden. Leider ist es auch bei den Lötaugen so, dass sie hohe Temperaturen über längere Zeiträume nicht vertragen. Zwar sind sie nicht so sensibel wie einige elektronische Komponenten, doch können sie sich nach einer zu langen Erhitzung durch den Lötkolben durchaus von der Platine lösen. Bei vielen Bauteilen kann man Verbindungen schaffen, indem man deren Beinchen festlötet, und anschließend Beinchen verbundener Bauteile, die nebeneinander liegen, direkt miteinander verlötet. Wenn man zum Beispiel den Stützkondensator am Mikrocontroller zwischen VCC und Masse anbringt, dann sollte man den Kondensator ohne Zwischenraum neben dem Chip anbringen, so dass man zum Beispiel das VCC-Beinchen des µC direkt an eines der Kondensatorbeinchen löten kann. Leider liegt das andere Beinchen dann nicht mit dem Masse-Pin des Mikrocontrollers auf einer Höhe. Hier kommt dann der Fädeldraht zum Einsatz.
Fädeldraht ist mit einem nicht-hitzefesten isolierenden Lack beschichtet. Der Lack wird durch das Erhitzen mit dem Lötkolben und Lötzinn geschmolzen. Dieser Vorgang dauert ein paar Sekunden und man erkennt den schmilzenden Lack daran, dass er dunkler wird, und anfängt Bläschen zu werfen. Diese abisolierte Stelle verbindet man durch Erhitzen der befestigenden Lötstelle mit dem Beinchen des Bauteiles. Man sollte dabei den Fädeldraht ca. 2-3 cm von der Lötstelle entfernt festhalten, da er sehr leitfähig ist, und dementsprechend sehr heiß wird. Wurden zwei Endpunkte mit dem Draht verbunden, wird der Fädeldraht mit kreisenden Bewegungen von der letzten Lötstelle abgetrennt.

Tipps:

- Der Lötzinn sollte das Lötauge komplett abdecken
- Lötzinn, der eher als glänzende Kugel auf dem Lötauge erscheint, sollte noch einmal erhitzt werden, da die Verbindung unter Umständen nicht perfekt ist.
- Bauteile, die im Schaltplan direkt miteinander verbunden sind, möglichst so platzieren, dass man die verbundenen Beinchen direkt miteinander verlöten kann, und keinen Draht benötigt.
- Der Fädeldraht sollte ca. 2-3 cm von der Lötstelle entfernt festgehalten werden, da er sehr leitfähig ist, und dementsprechend sehr heiß wird.
- Nachdem das eine Ende des Fädeldrahtes verbunden ist, den Draht so verlegen, dass man ihn in einer Schleife um die Verbindungsstelle legen kann. Die Lötkolbenspitze kann durch die Schleife geschoben werden, der Draht kann fester an den Lötkolben gepresst werden, wodurch sich der Lack schneller erhitzt und deutlich schneller das Metall freigibt. Anschließend die Schleife um das betroffene Beinchen legen, fest anziehen und festhalten, während der Lötzinn des Beinchens wieder erhitzt wird.
- Es ist hilfreich im Voraus die Leiterbahnen zu planen. Hat man zum Beispiel an 2 Bauteilen je ein Beinchen das auf Masse liegt, dann bietet es sich an, den Fädeldraht mit der Masse zu verbinden, zum Massepin des ersten Bauteils eine Verbindung zu schaffen, hier den Draht in der Schleife abzuisolieren, anzulöten und den Fädeldraht dann gleich mit dem nächsten Massepin zu verbinden.
- Jede Verbindung sollte nach der Lötung auf ihre Richtigkeit überprüft werden. Das funktioniert am einfachsten mit einem Durchgangsprüfer, der in den meisten Messgeräten integriert ist. Lässt man diese Zwischenprüfungen weg, dann ist der Aufwand, eine fehlerhafte Verbindung auf der fertigen Platine zu finden, sehr groß.

- Der Durchgangsprüfer sollte auch verwendet werden, um Verbindungen zwischen Beinchen, die nicht verbunden gehören, auszuschließen. Wird so ein Fehler übersehen, dann kann es schlimmstenfalls zu einem Kurzschluss und damit zur Beschädigung des betroffenen Bauteils kommen.

5.2.3.2 Stecker löten

Leider sind die meisten Stecker (Stromsteckerbuchse, RS232-Stecker usw.) nicht konform mit dem Lochabstand und -durchmesser der Platine. Dann muss man die entsprechenden Löcher auf der Platine aufbohren, aber möglichst so, dass man im nahen Umfeld intakte Lötaugen hat, mit denen die Verbindung zu den Anschlusspins hergestellt werden kann. Der Stecker sollte vor der Bohrung in Position gebracht und die Bohrpunkte markiert werden. Liegt der Stecker nach der Bohrung gut auf der Platine auf, sollten die Lötanschlüsse so umgebogen werden, dass sie dem Stecker zusätzliche Stabilität verleihen. Das Festlöten geschieht hier nach dem Prinzip "Je mehr desto besser", da Stecker oft einer hohen Belastung standhalten müssen. Daher sollten mindestens zwei Lötaugen mit dem Beinchen des Steckers und sehr viel Zinn verbunden werden.
Beim RS232-Stecker tritt noch eine Besonderheit auf. Der 9-Pol Stecker hat zwei Reihen mit einmal 5 und einmal 4 Beinchen. Diese Reihen liegen aber nicht auf einer Höhe, sondern die obere Reihe ist so versetzt, dass die Beinchen auf einer Standard-Lochrasterplatine genau zwischen den Löchern liegen. Für die Entwicklungsplatine bietet es sich an die Beinchen 6 bis 9 so zu verbiegen, dass sie durch die Löcher passen. Da diese Beinchen nicht relevant für die Kommunikation sind, wäre es nicht gravierend, falls eines der Beinchen den Kontakt verliert. Das sollte aber trotzdem vermieden werden.

5.2.3.3 Einsatz von Sockeln

Etwas, das es auf jeden Fall zu beachten gilt, und das im Nachhinein Ärger, Nerven und Arbeitszeit spart, ist, dass Chips immer gesockelt werden. Das heißt, ein Sockel dessen Anschlüsse der Anzahl der Chipbeinchen entsprechen wird auf die Platine gelötet, dann wird der Chip in den Sockel gesteckt. Dadurch ist die Austauschbarkeit bei Mängeln oder selbst verschuldeten Beschädigungen ausgeschlossen. Außerdem geht man nicht das Risiko ein, einen Bereich des Chips beim Löten zu überhitzen und damit zu zerstören. Wenn man den Chip wieder vom Sockel lösen will, muss das sanft und vorsichtig geschehen, damit die Anschlüsse der Pins nicht beschädigt werden. Das geschieht am einfachsten mit einem kleinen flachen Schraubenzieher, der auf einer Seite zwischen Sockel und Chip geschoben wird. Anschließend hebelt man den Chip ein wenig heraus. Das gleiche wird auf der gegenüberliegenden Seite gemacht, und der ganze Vorgang so lange wiederholt, bis sich der Chip ohne Kraftaufwand aus dem Sockel heben lässt.

5.2.3.4 Entlöten

Leider kann man beim Löten nicht verhindern, dass der Lötzinn eine Verbindung zwischen zwei Beinchen schafft, wo keine sein sollte. Besonders häufig passiert das bei elektronischen Bausteinen mit drei nebeneinander liegenden langen Anschlussbeinchen, wie zum Beispiel Spannungsreglern. Die Verbindung zwischen den Beinchen muss dann wieder gelöst werden, wofür es einerseits Entlötlitze gibt, die sich besser für punktgenaues und vollständiges Entlöten eignet und andererseits Entlötpumpen, die man für einfache Entlötungen verwendet. In den meisten Fällen reicht die Entlötpumpe aus.

5.3 Parallelport Programmieradapter

Die einfachste Möglichkeit einen Mikrocontroller zu programmieren besteht darin einen Adapter für den Parallelport [pport] zu bauen.
Dieser besteht in der simpelsten Variante aus einem 25-poligen D-Sub
Stecker (männlich), 2 1kΩ Widerständen und einem 10-poligen Stecker
der die Verbindung zur Entwicklungsplatine herstellt. Es sei darauf hingewiesen, dass diese Version des Adapters nicht stabil funktioniert, und es durchaus vorkommen kann, dass man entweder den LPT-Port des PCs oder den Mikrocontroller beschädigt.

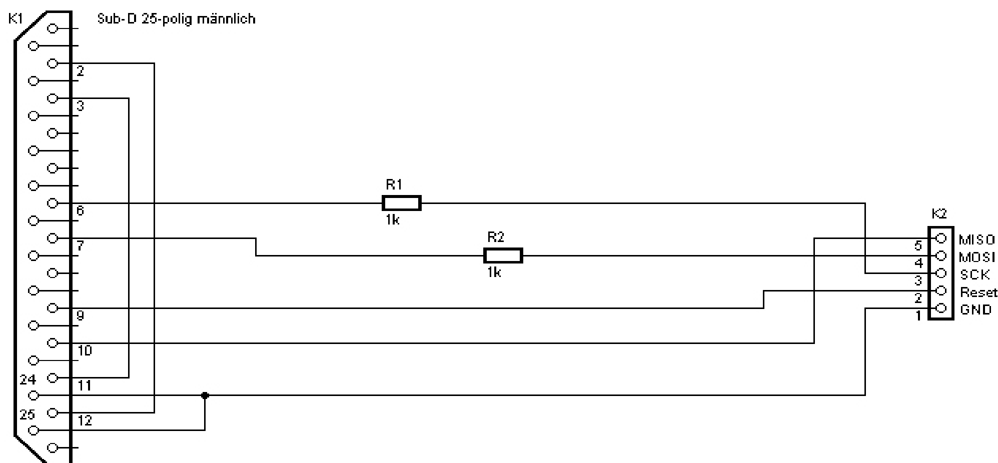


Abb. 109: einfacher Parallelport Programmieradapter

Um dieses Risiko zu verringern sollte die Schaltung um einen Puffer-IC erweitert werden.
Eine Anleitung dieses Parallelport Programmieradapters findet sich auf folgender Seite:

http://www.kreatives-chaos.com/index.php?seite=parallel

5.4 Entwicklungsplatine BL1000

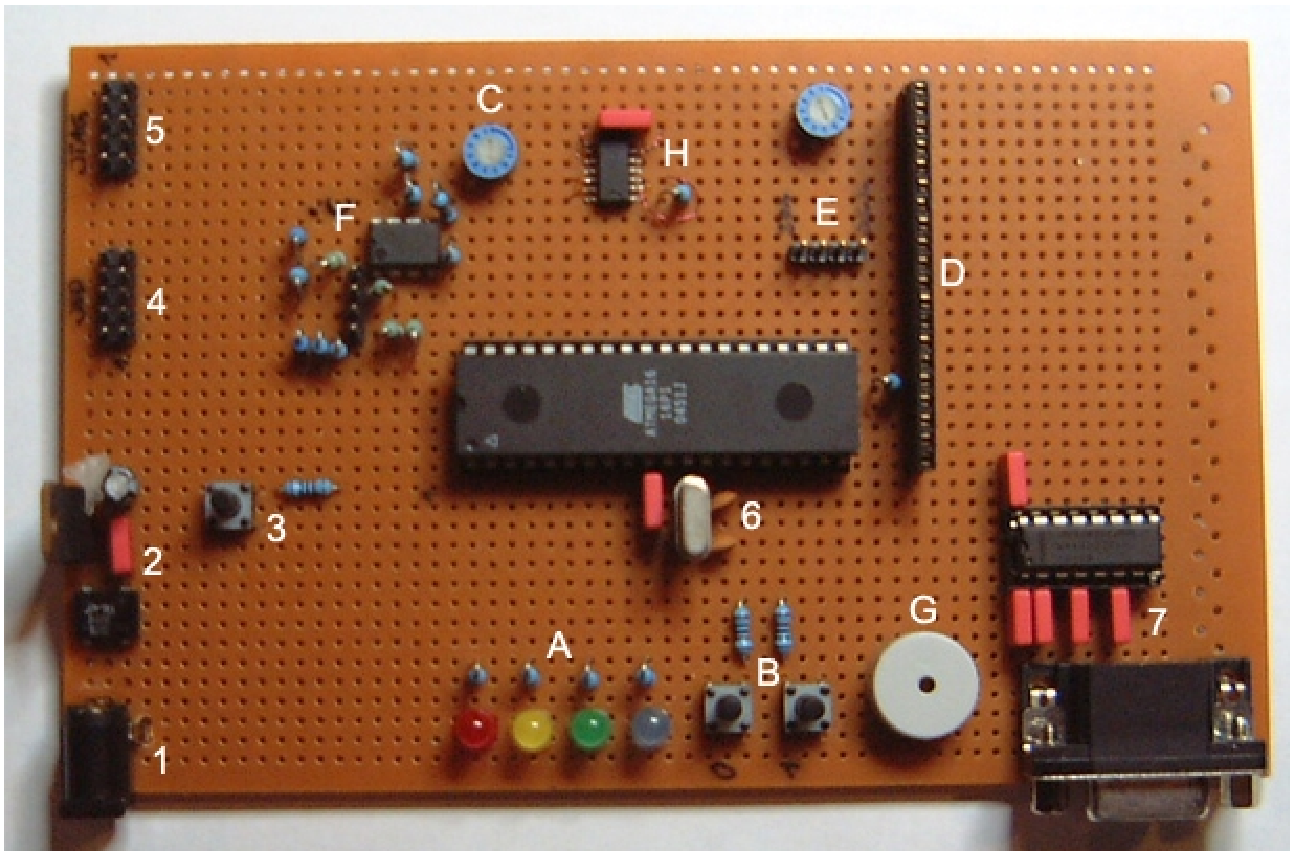


Abb. 110: BL1000 ohne LCD

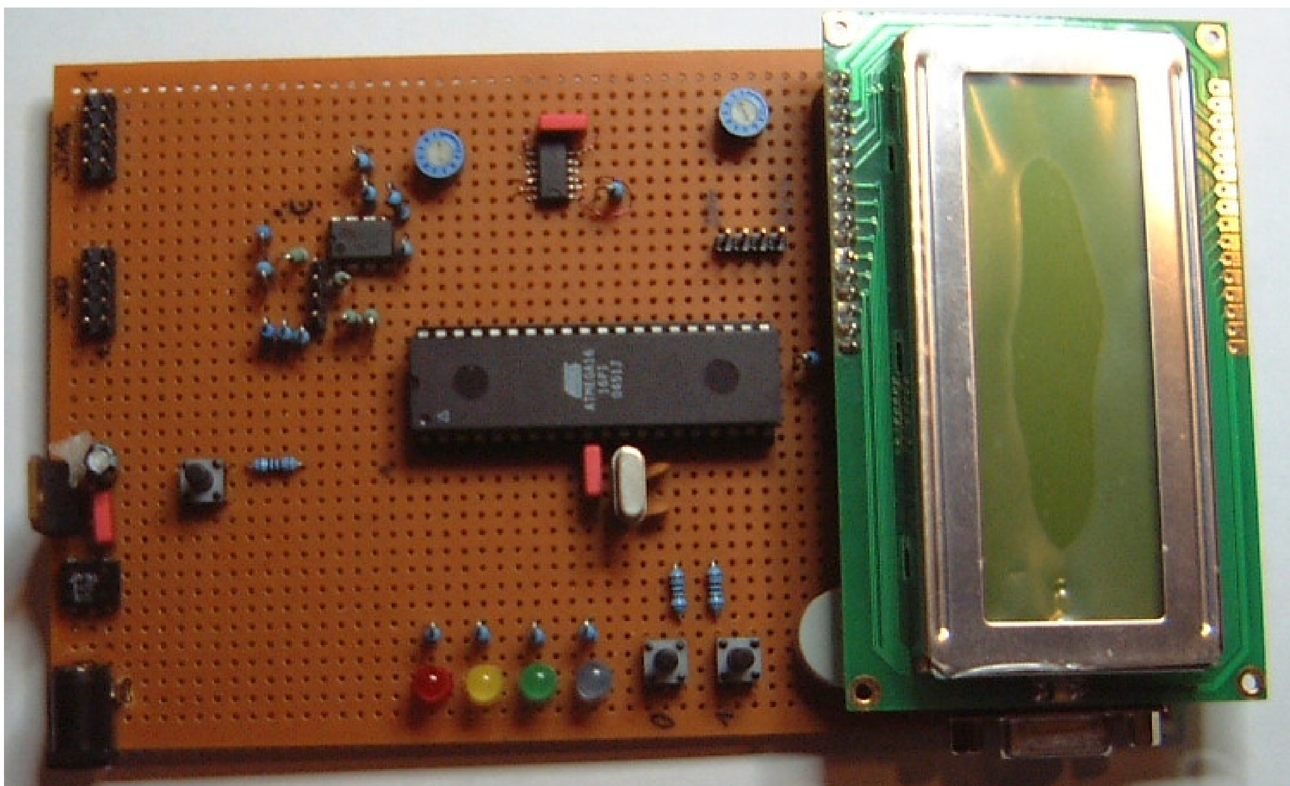


Abb. 111: BL1000 mit LCD

5.4.1 Beschreibung der BL1000 Platine

Die folgende Komponentenbeschreibung bezieht sich auf die Abbildung 110 auf Seite 100.

1 Hohlstecker, an den ein Netzteil angeschlossen werden kann. Alternativ kann die Platine auch über das Evertool mit Strom versorgt werden.

2 Spannungsversorgung der Platine bestehend aus den Hauptkomponenten Brückengleichrichter und Spannungsregler.

3 Taster, der über einen 1kΩ Widerstand zwischen Masse und RESET-Pin des Controllers angeschlossen ist.

4 10pol Pfostenstecker für den Anschluss eines JTAG-Adapters

5 10pol Pfostenstecker für den Anschluss eines ISP-Adapters

6 7.3738 MHz Quarz als Taktgeber für den ATmega16

7 D-Sub 9pol Stecker und MAX3232 Pegelwandler für die Realisierung der RS-232 Schnittstelle

A 4 Leuchtdioden die mit Hilfe von Vorwiderständen zwischen VCC und Pins des µC geschaltet sind

B 2 Taster die an die externen Interruptpins INT0 und INT1 des ATmega16 angeschlossen sind

C 10kΩ Potentiometer am Pin ADC1 des Mikrocontrollers

D 16pol Buchsenleiste für den Anschluss des LC Displays (vgl. Abb. 111 Seite 100)

E 4pol Stiftleiste für den Anschluss eines dreiadrigen Kabels zur Interchipkommunikation über TWI. Zwei Pins der Leiste werden für die SCL- und SDA- Leitungen verwendet.
Der dritte Pin des Steckers wird mit der Masse verbunden, der vierte Pin dient zur Stabilisierung.

F Temperatursensorschaltung (vgl. Abb. 58 Seite 71) bestehend aus Operationsverstärker und dessen Widerständen und einem 4pol Stiftleiste. 2 Pins der Stiftleiste werden für die Verbindung zum Sensor benötigt, ein Pin wird mit der Masse verbunden, der vierte dient der Stabilisierung
Der Temperatursensor wurde nicht direkt auf die Platine gelötet, sondern an das eine Ende eines geschirmten dreiadrigen Kabels (z.B. Audiokabel, das früher beim Zubehör von Mainboards dabei war) gelötet. Das andere Ende des Kabels wird mit einem 4pol Stecker versehen.

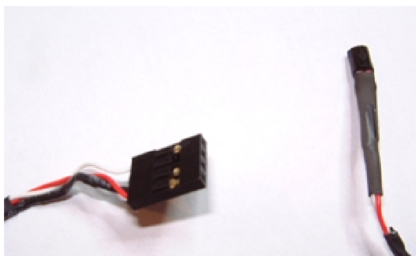


Abb. 112: Temperatursensor an einem dreiadrigen Kabel

G Piezo-Schallwandler

H RTC 4513

6 Fazit

Zum Abschluss dieser Arbeit sollen kurz deren Ergebnisse zusammengefasst werden.
Es wurde eine Platine entwickelt, die auf dem ATMEL AVR ATmega16 Mikrocontroller basiert. Hierbei war es der Autorin wichtig, dass sowohl Bauteile der in der Vorlesung verwendeten NF300 Platine, als auch zusätzliche Elemente, wie z.B. die TWI-Schnittstelle, realisiert wurden.
Neben LEDs, Tastern und einem LC Display befinden sich auf der Platine außerdem ein Schallwandler, eine TWI-Schnittstelle zur Interchipkommunikation, eine RS-232 Schnittstelle um die Kommunikation per USART zu ermöglichen, eine Echtzeituhr und ein Temperatursensor.
Die im Tutorial besprochenen Themen sind so aufgebaut, dass der Schwierigkeitsgrad mit den Kapiteln ansteigt, und es so den Studenten, die mit diesem Tutorial arbeiten, ermöglicht wird, Schritt für Schritt in das Thema der hardwarenahen Programmierung einzusteigen.
In der HTML-Version des Tutorials wurde darauf Wert gelegt, dass gängige Fehlerquellen besonders hervorgehoben werden.
Zu allen im Tutorial besprochenen Themen entwickelte die Autorin Codebeispiele, die so programmiert wurden, dass nur noch die Pindefinitionen geändert werden müssen, wenn die verwendete Platine nicht dem Aufbau der BL1000 Platine entspricht. Es wurde außerdem auf eine ausführliche Kommentierung des Codes geachtet, damit dieser auch ohne das Tutorial verständlich ist.
Das letzte Kapitel des Tutorials beschäftigte sich mit dem Einstieg in den Platinenbau. Ziel dieses Kapitels sollte es sein, das Interesse am Bau der eigenen Platine zu wecken.
Die gesamte Diplomarbeit wurde unter die Creative Commons Lizenz gestellt, daher ist es möglich und auch erwünscht, dass, unter den Lizenzbedingungen, sowohl das Tutorial als auch der von der Autorin erstellte Sourcecode, verwendet, verändert und auch verbessert wird,

7 Anhang

A Abbildungsverzeichnis Seite

Abb. 01: DIP- und TQFP-Gehäuse 09
Abb. 02: MLF-Gehäuse 09
Abb. 03: Blockschaltbild des ATmega16 10
Abb. 04: Pinbelegung des ATmega16 11
Abb. 05: ISP-Adapter für die parallele Schnittstelle 13
Abb. 06: ATMEL ISP für die serielle Schnittstelle 13
Abb. 07: AVR JTAG ICE mk2 13
Abb. 08: Evertool 14
Abb. 09: Verbindung zwischen PC und Mikrocontroller-Platine 15
Abb. 10: mFile Oberfläche 18
Abb. 11: AVRDude-GUI 20
Abb. 12: AVaRICE 20
Abb. 13: AVR Studio 4.12 22
Abb. 14: AVR Studio 4.12 - Projekt anlegen 23
Abb. 15: AVR Studio 4.12 - Sprache wählen 23
Abb. 16: AVR Studio 4.12 - Debugging über Simulator 24
Abb. 17: AVR Studio 4.12 - Debugging über JTAG 24
Abb. 18: AVR Studio 4.12 - Auswahl des Programmers: ISP 25
Abb. 19: AVR Studio 4.12 - Auswahl des Programmers: JTAG 25
Abb. 20: AVR Studio 4.12 - ISP Dialog 26
Abb. 21: AVR Studio 4.12 - Simulator 26
Abb. 22: AVR Studio 4.12 - Watch-Fenster 26
Abb. 23: Fusebits - High Byte 27
Abb. 24: Fusebits - Low Byte 28
Abb. 25: AVR Studio 4.12 Fusebits 29
Abb. 26: AVR Studio 4.12 Lockbits 29
Abb. 27: LED als Active-Low angeschlossen 33
Abb. 28: LED als Active-High angeschlossen 33
Abb. 29: mFile - F_CPU ändern 34
Abb. 30: EEPROM - EEARH 37
Abb. 31: EEPROM - EEARL 37
Abb. 32: GICR - General Interrupt Control Register 40
Abb. 33: MCUCR - MCU Control Register 40
Abb. 34: MCUCSR - MCU Control and Status Register 40
Abb. 35: GIFR - General Interrupt Flag Register 40
Abb. 36: Blockschaltbild Timer/Counter 42

Seite

Abb. 37: TCCR0 43
Abb. 38: Darstellung von Fast PWM mit Hilfe eines Oszilloskops 48
Abb. 39: Terminal 51
Abb. 40: Blockschaltbild A/D-Wandler 53
Abb. 41: ADMUX 54
Abb. 42: ADCSRA 55
Abb. 43: ADLAR = 0 56
Abb. 44: ADLAR = 1 56
Abb. 45: SFIOR 56
Abb. 46: 4 zeiliges LC-Display 58
Abb. 47: Selbstdefiniertes LCD-Zeichen 64
Abb. 48: nicht-invertierender Komparator 66
Abb. 49: invertierender Komparator 66
Abb. 50: invertierender Verstärker 67
Abb. 51: nicht-invertierender Verstärker 67
Abb. 52: Subtrahierer 67
Abb. 53: Addierer 68
Abb. 54: NTC 69
Abb. 55: PTC 69
Abb. 56: Spannungsteiler 70
Abb. 57: Temperatur-Spannung-Kurve 70
Abb. 58: Schaltungsaufbau des Temperatursensors 71
Abb. 59: Anschluss von Slaves an TWI 74
Abb. 60: TWI Adresspaket 75
Abb. 61: TWI-Adresspaket - dargestellt mit Hilfe eines Oszilloskops 75
Abb. 62: TWI Datenpaket 76
Abb. 63: TWI-Datenpaket - dargestellt mit Hilfe eines Oszilloskops 76
Abb. 64: TWBR - TWI Bit Rate Register 78
Abb. 65: TWDR - TWI Daten Register 78
Abb. 66: TWAR - TWI Address Register 78
Abb. 67: TWCR - TWI Control Register 78
Abb. 68: TWSR - TWI Status Register 80
Abb. 69: Darstellung des Auslesens der Register S1 bis W mit Hilfe eines Oszilloskops 88
Abb. 70: Versorgungsspannung 90
Abb. 71: Masse 90
Abb. 72: Folienkondensator 90
Abb. 73: Keramikkondensator 90
Abb. 74: Schaltzeichen ungepolter Kondensatoren 90
Abb. 75: Elektrolytkondensator 90

Seite

Abb. 76: Schaltzeichen gepolter Kondensatoren 90
Abb. 77: Widerstand 91
Abb. 78: Schaltzeichen Widerstand 91
Abb. 79: Cermet-Potentiometer 91
Abb. 80: Drehpotentiometer 91
Abb. 81: Schaltzeichen Potentiometer 91
Abb. 82: Diode 91
Abb. 83: Schaltzeichen Diode 91
Abb. 84: Wechselspannung 92
Abb. 85: Schaltzeichen Brückengleichrichter 92
Abb. 86: pulsierende Gleichspannung 92
Abb. 87: durch Kondensator veränderte Gleichspannung 92
Abb. 88: Brückengleichrichter 92
Abb. 89: rote Leuchtdiode 92
Abb. 90: Schaltzeichen Leuchtdiode 92
Abb. 91: Spannungsregler 93
Abb. 92: Schaltzeichen Spannungsregler 93
Abb. 93: Quarz 93
Abb. 94: Schaltzeichen Quarz 93
Abb. 95: Quarzoszillator 93
Abb. 96: Schaltzeichen Quarzoszillator 93
Abb. 97: Zuschaltung von Stützkondensatoren zum Taktgeber 93
Abb. 98: Operationsverstärker 94
Abb. 99: Schaltzeichen Operationsverstärker 94
Abb. 100: Heißleiter 94
Abb. 101: Schaltzeichen Heißleiter 94
Abb. 102: Kaltleiter 94
Abb. 103: Schaltzeichen Kaltleiter 94
Abb. 104: Piezo-Summer 94
Abb. 105: Schaltzeichen Piezo-Summer 94
Abb. 106: Piezo-Schallwandler 94
Abb. 107: Schaltzeichen Schallwandler 94

B Tabellenverzeichnis

Tab. 01: Software 15
Tab. 02: Interruptvektoren 39
Tab. 03: Watchdogkonstanten 41
Tab. 04: Addressmap 62
Tab. 05: RTC 4513 Register 86

C Quellenverzeichnis

C.1 Webseiten

[I^2C] http://www.semiconductors.philips.com/acrobat_download/literature/9398/39340011.pdf

[risc] http://de.wikipedia.org/wiki/RISC

[hav] http://de.wikipedia.org/wiki/Harvard-Architektur

[vna] http://de.wikipedia.org/wiki/Von-Neumann-Architektur

[avrsw] http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725

[winavrsw] http://winavr.sourceforge.net/

[pnsw] http://www.pnotepad.org

[emacs] http://www.gnu.org/software/emacs/emacs.html

[libcsw] http://www.nongnu.org/avr-libc

[gccsw] http://www.avrfreaks.net/index.php?module=FreaksTools&func=viewItem&item_id=145

[gdbsw] http://www.gnu.org/software/gdb/gdb.html

[inssw] http://sources.redhat.com/insight/

[dddsw] http://www.gnu.org/software/ddd/

[dudesw] http://www.nongnu.org/avrdude/

[dudeguisw] http://sourceforge.net/projects/avrdude-gui/

[ponysw] http://www.lancos.com/prog.html

[uispsw] http://www.nongnu.org/uisp/index.html

[make] http://www.gnu.org/software/make/manual/make.html

[isp] http://de.wikipedia.org/wiki/In-System-Programmierung

[pull] http://de.wikipedia.org/wiki/Pullup_Widerstand#Pull_Up

[pwm] http://www.mikrocontroller.net/articles/Pulsweitenmodulation

[op1] http://www.mikrocontroller.net/articles/Operationsverstärker-Grundschaltungen

[op2] http://de.wikipedia.org/wiki/Opamp

[kopp] http://de.wikipedia.org/wiki/Gegenkopplung

[piz] http://de.wikipedia.org/wiki/Piezo

[poti] http://de.wikipedia.org/wiki/Potentiometer

[wi] http://de.wikipedia.org/wiki/Elektrischer_Widerstand

[kond] http://de.wikipedia.org/wiki/Kondensator_%28Elektrotechnik%29

[dio] http://de.wikipedia.org/wiki/Diode

[elge] http://www.bmu.de/files/pdfs/allgemein/application/pdf/elektrog.pdf

[lcd] http://www.sprut.de/electronic/lcd/index.htm

[temp] http://www.sprut.de/electronic/temeratur/temp.htm

[twi1] http://www.roboternetz.de/wissen/index.php/I2C

[twi2] http://www.roboternetz.de/wissen/index.php/TWI

[pport1] http://s-huehn.de/elektronik/avr-prog/avr-prog.htm

C.1 Datenblätter

[ATM16] ATMEL Corp., 8-bit AVR Mikrocontroller with 16K Bytes In-System Programmable Flash - ATmega16 (Rev. 2466D-09/02)

[ATM315] ATMEL Corp., 8-bit AVR Mikrocontrollers Application Note Using the TWI module as I²C master (Rev. 2564B-AVR-09/04)

[LCD] Displaytech Ltd, LCD Module 204B Series (Version 1.1)

[EMP] Deltron Components GmbH, emp121_IMP

[KTY] Discrete Semiconductors, Datasheet KTY81-2 series, (1996 Dec 06)

[RTC] SEIKO EPSON Corp., Application Manual RTC-4513 (MQ222-02)

D CD

Inhalt der CD:

1. Sourcecode

    /Code/Hauptprogramm

        Programm, das einige Unterprogramme vereint

    /Code/AD Wandler/FreeRunningMode

        Programm zur Ansteuerung des ADC Free Running Modes

    /Code/AD Wanlder/SingleConversionMode

        Programm zur Ansteuerung des ADC Single Conversion Modes

    /Code/LC Display

        Funktionen zur Ansteuerung eines LC Displays

    /Code/LED

        Programm zur Ansteuerung von 4 LEDs

    /Code/Piezo

        Programm zur Ansteuerung eines Piezo Schallwandlers

    /Code/PWM

        Programm zur Ansteuerung einer LED mittels Fast PWM

    /Code/RTC

        Funktionen zur Ansteuerung der Echtzeituhr RTC4513

    /Code/Temperatursensor

        Programm zur Ansteuerung des Sensors KTY81-210

    /Code/TWI/Master

        Funktionen zur Ansteuerung des Masters einer TWI-Kommunikation

    /Code/TWI/Slave

        Funktionen zur Ansteuerung des Slaves einer TWI-Kommunikation

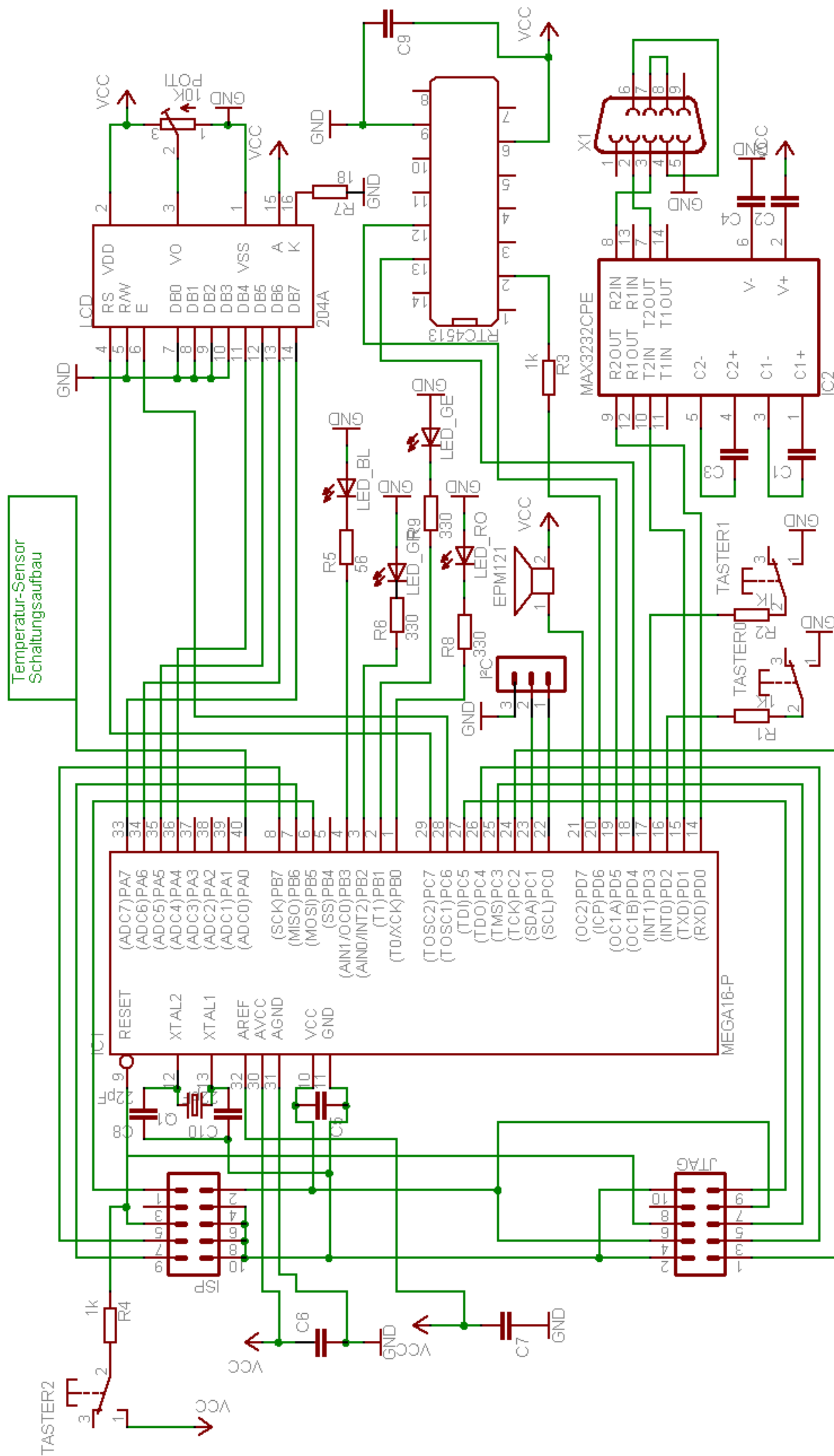
    /Code/USART

        Funktionen zur Realisierung der Kommunikation über USART

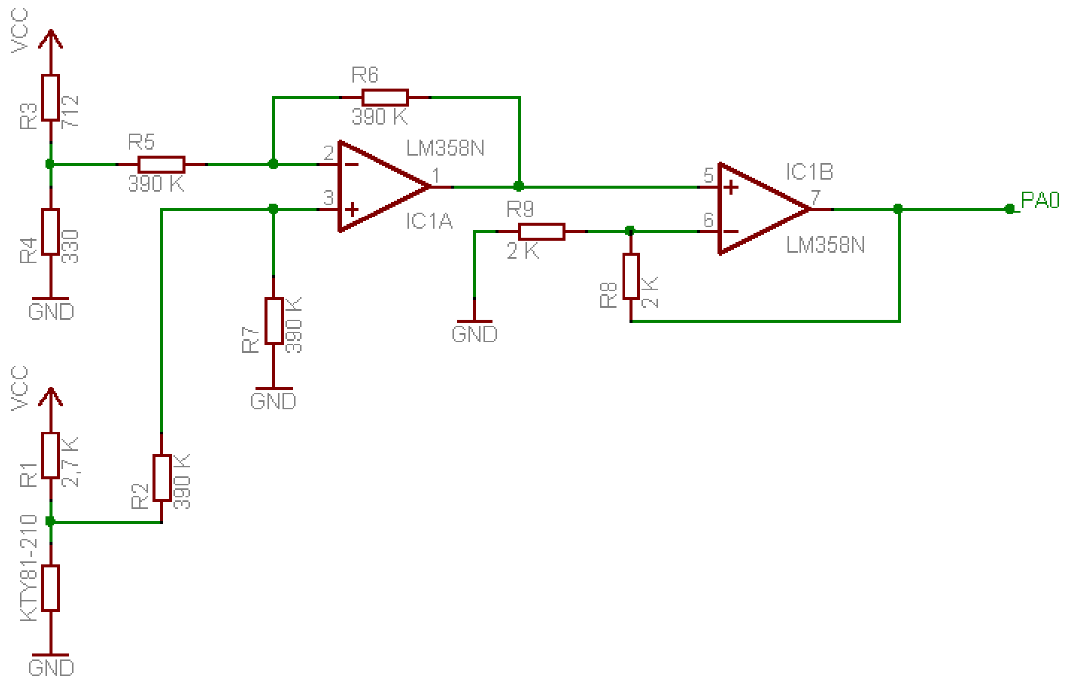
2. Datenblätter
3. Schaltpläne
    - Entwicklungsplatine BL1000
    - Evertool
4. Software
    - AVR Studio 4.12
    - WinAVR 2006/01/25
    - Terminalprogramm
5. Tutorial
6. Diplomarbeit

E Schaltpläne BL1000

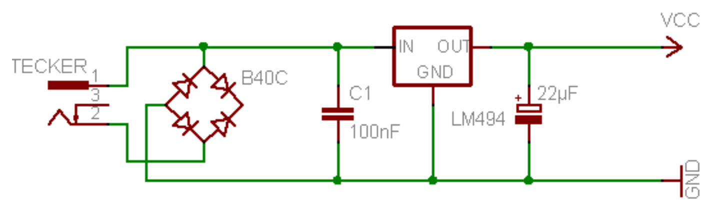
Entwicklungsplatine BL1000



Temperatursensor-Schaltung



Netzanschluss



Erstellungserklärung

Die Diplomarbeit ist gemäß §31 der Rahmenprüfungsordnung für die Fachhochschulen in Bayern (RaPO) vom 18.09.1997 mit Ergänzung durch die Prüfungsordnung (PO) der Fachhochschule Augsburg vom 15.12.1994 erstellt worden.

Ich versichere, dass ich die Diplomarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Anwalting, den 09.07.2006

____

Bianca-Charlotte Liehr