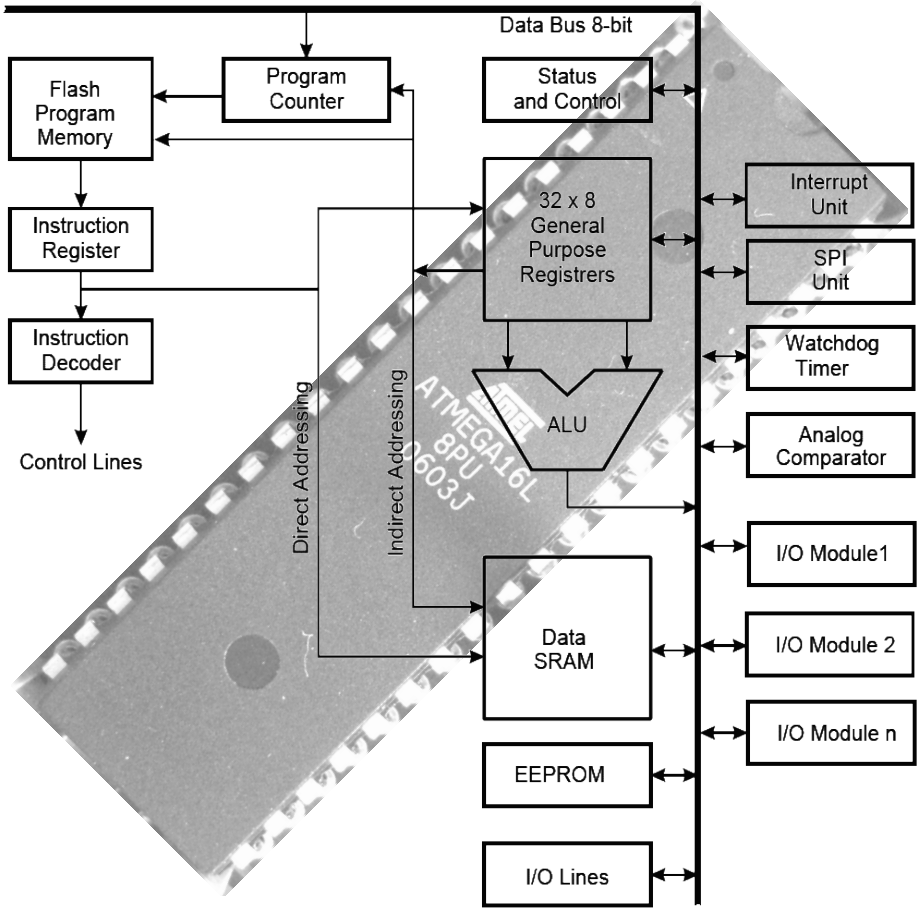
FACHBEREICH PHYSIK
Elektronik–Praktikum

UNIVERSITÄT OSNABRÜCK

Mikrocontroller

atmikro.tex KB 20091124



Die Integration aller wesentlichen Bestandteile eines Computers auf einem einzigen Chip führte in den vergangenen Jahrzehnten zu den uns allen geläufigen beträchtlichen Fortschritten in allen Bereichen, in denen solche *Mikroprozessoren* verwendet werden. Im Vordergrund der Entwicklung stand und steht dort die Erhöhung der Geschwindigkeit – sowohl im Prozessor wie auch zur Peripherie (Speicher, Videocontroller), die Einbeziehung hochverfügbaren Speichers (Cache), die Erhöhung der Wortbreite. Weniger wichtig ist die Integration zusätzlicher Peripherie, insbesondere solcher Peripherie, die eine einfache Kopplung an Mess- und Steuerelektronik ermöglicht. Genau in diese Richtung zielt die Entwicklung der *Mikrocontroller*. Um einen Prozessorkern wird möglichst viele und vielseitige Zusatzhardware angeordnet, um Mess-, Regel- und Steuerungsaufgaben erledigen zu können.

Während die *Mikroprozessor*-Entwicklung sich im Wesentlichen auf einen relativ schmalen High-End-Bereich konzentriert, weist die *Mikrocontroller*-Entwicklung eine beachtliche Bandbreite auf. Das hängt mit der Vielfalt der Anwendungsgebiete zusammen, in denen Mikrocontroller eingesetzt werden. Die Palette reicht von einfachen preisgünstigen Spezialentwicklungen mit einer Wortbreite von 4 oder 8 Bit bis hin zu Controllern, die als Kern einen modernen Prozessor enthalten und damit auch dessen Leistungsklasse erreichen.

Im Praktikum setzen wir einen ATmega16 ein, einen einfachen Mikrocontroller der Firma Atmel mit einer Wortbreite von 8 Bit. Zur Programmierung in Assembler und zur Ablaufkontrolle (*Debugging*) stellt Atmel das *AVR-Studio* als Freeware zur Verfügung. Die Entwicklungsumgebung bietet die Möglichkeit, Programme in einer Simulation ablaufen zu lassen oder – mit einer geeigneten Verbindung (*In-Circuit-Emulator, ICE*) – direkt auf dem Controller. Durch GNU-Plug-Ins (*Cross-Compiler*) kann AVR-Studio für eine Programmierung in C oder C++ erweitert werden.

Inhaltsverzeichnis

1 Ziele

Im Praktikumsversuch lernen Sie die *maschinennahe Programmierung* eines Mikrocontrollers kennen. Grundsätzlich bietet sich dafür Assemblerprogrammierung an, da man dadurch sowohl den Zeitablauf wie auch die Speicherbelegung eines Programms bis in alle Einzelheiten festlegen kann. Sie werden hier jedoch C als Programmiersprache verwenden, da dafür ein sehr guter GNU-Cross-Compiler frei verfügbar ist und da die Programmierung in C wesentlich einfacher als die in Assembler ist.

Bevor Sie eine gewünschte Funktionalität des Mikrocontrollers durch ein Programm realisieren können, müssen Sie den Funktionsumfang des Controllers genau kennen. Wichtig sind insbesondere Kenntnisse über die im Mikrocontroller integrierte zusätzliche *Hardware*.

Das Arbeiten mit einer *Entwicklungsumgebung* ist eine der Voraussetzungen für eine erfolgreiche Programmentwicklung. In dieser Umgebung können Programme mit einem *Editor* geschrieben und mit einem *Cross-Assembler* oder *Cross-Compiler* in Maschinencode übersetzt werden. Die Programme werden dann als *Simulation* oder – nach dem Transfer auf die Zielhardware – mit Hilfe eines *In-Circuit-Emulators* direkt auf dem Zielprozessor getestet. Sie lernen mit dem *Debugger* zu arbeiten, mit dem Sie den *Programmablauf* verfolgen können, das Programm jederzeit *stoppen* können, *Breakpoints* setzen können, *Speicher-* und *Register*inhalte sowie den Status der *Controller-Hardware* ansehen und ändern können. Dies alles sowohl in der Simulation als auch direkt auf der Hardware.

Welche konkreten Aufgaben und Ziele Sie letztlich realisieren, sollten Sie von Ihren Voraussetzungen abhängig machen und in der Vorbereitung und während des Praktikumsversuchs entscheiden.

2 Grundlagen, Literatur

Falls Sie nicht in C-Programmierung fit sind, sollten Sie sich einige Grundlagen aneignen. Für den Versuch wird nicht allzu viel benötigt, insbesondere brauchen Sie fast keine Bibliotheksfunktionen. Sie sollten die wichtigsten Operatoren kennen, Datentypen (die kleinen Integer-Typen reichen aus), Variablen und deren Vereinbarungen, einige Preprozessor-Anweisungen, Funktionen, Ausdrücke und Zuweisungen, Kommentare, einige Kontrollstrukturen. Lesen Sie diese Stichworte in einem C-Lehrbuch nach (z. B. Kernighan und Ritchie: *Programmieren in C*). Sie sollten die Beispielprogramme verstehen und ändern können.

Der Mikrocontroller ATmega16 ist in der von der Herstellerfirma Atmel herausgegebenen ausführlichen Dokumentation in allen Einzelheiten beschrieben. Sie finden alles im Internet, die folgenden Dokumente auch in Stud.IP:

`doc2466.pdf` : Ausführliche Beschreibung des ATmega16

`2466S.pdf` : Zusammenfassende Übersicht zum ATmega16

dip162-d.pdf : Beschreibung der LCD-Anzeige

d_2233S_DFF.pdf : Datenblatt des Gleichstrommotors

d_MT2251S_DFF.pdf : Datenblatt des Tachogenerators

LMD18200.pdf : Beschreibung der H-Brücke für die Motoransteuerung

Ebenfalls im Internet finden Sie eine Vielzahl von Beschreibungen für Anwendungen aller Art (*Application Notes*) für die Controller der Firma Atmel. Für die Versuchsdurchführung könnten die folgenden von Interesse sein, die in Stud.IP bereit liegen:

doc2558.pdf : Beschreibung eines PID-Regler-Programms

AVR221.zip : Quellcode und zugehörige Dokumentation für das Regler-Programm

Download-Links

Die Mikrocontroller von Atmel sind sehr verbreitet, bedingt unter anderem auch durch die Verfügbarkeit von guten Programmierwerkzeugen. Aufgrund der großen Verbreitung finden Sie auch sehr viele Beispiel-Applikationen im Internet, zum Teil mit sehr ausführlichen Beschreibungen. Stichworte für die Suche: *Atmel*, *Microcontroller*, *AVR controller*, *ATmega*. Hier ein paar Adressen, unter denen Sie Werkzeuge und Beispiele, manchmal auch Antworten auf schon gestellte Fragen finden:

www.atmel.com : Herstellerfirma des Mikrocontrollers.

www.atmel.com/products/avr : Einstiegsseite der AVR-Mikrocontroller.

winavr.sourceforge.net : GNU-Werkzeuge für das Arbeiten mit dem Mikrocontroller, unter anderem der C-Cross-Compiler.

www.mikrocontroller.net : Deutschsprachige Seite mit Allgemeinem und Speziellem zu Mikrocontrollern.

www.avrfreaks.net : Wie schon der Name sagt . . .

savannah.nongnu.org/mail/?group=avr : Mailing-Liste zu Programmierung, Bugs etc. Mit Hinweisen und Tipps zu den GNU-Werkzeugen.

www.roboternetz.de/wissen/index.php/Regelungstechnik : Analoge und digitale Regelungstechnik.

www.sprut.de/electronic/switch/schalt.html : Funktionsprinzipien von Schaltreglern.

www.roboternetz.de/wissen/index.php/Pulsweitenmodulation : Beschreibung der Pulsweitenmodulation.

3 Hardware-Komponenten

Zentraler Baustein ist der Mikrocontroller ATmega16, der vom PC aus über ein USB-JTAG-Interface programmiert werden kann. Daran können verschiedene Peripheriemodule für die Ein- und Ausgabe sowie Ver-

bindungsmodule für den direkten Zugang zu den Controller-Ports angeschlossen werden.

3.1 Der Mikrocontroller ATmega16

Kurzcharakterisierung durch die Herstellerfirma:

The ATmega16 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega16 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Während übliche Prozessoren, wie sie beispielsweise in PCs verwendet werden, nur in einem eingeschränkten Taktfrequenzbereich robust arbeiten, lassen CMOS-Prozessoren einen weiten Bereich zu. Im allgemeinen ist nur die maximale Taktfrequenz festgelegt, eine langsamere Taktung bis hin zum völligen Stop ist in der Regel unproblematisch. Damit ist ein äußerst sparsamer Betrieb möglich, je nach Erfordernis lässt man den Prozessor schneller oder langsamer laufen und verbraucht damit nur die unbedingt notwendige Energie. Im Praktikum nutzen wir das jedoch nicht aus, hier ist die Taktfrequenz fest auf 8 MHz eingestellt.

Durch die RISC-Architektur (*R*educed *I*nstruction *S*et *C*PU) arbeitet der Controller mit sehr kurz kodierten Instruktionen, die fast alle in einem oder in zwei Taktzyklen abgearbeitet werden.

Die im Mikrocontroller integrierte Peripherie verwendet 32 Ein/Ausgabe-Leitungen, die in 4 Gruppen mit je 8 Anschlüssen zusammengefasst sind (PortA bis PortD). Zusammen mit den Anschlüssen für die Stromversorgung, Taktfrequenzquarz etc. passt alles in ein 40-poliges DIP-Gehäuse (*D*ual *I*nline *P*ackage). Die Pinbelegung zeigt Abbildung 1.

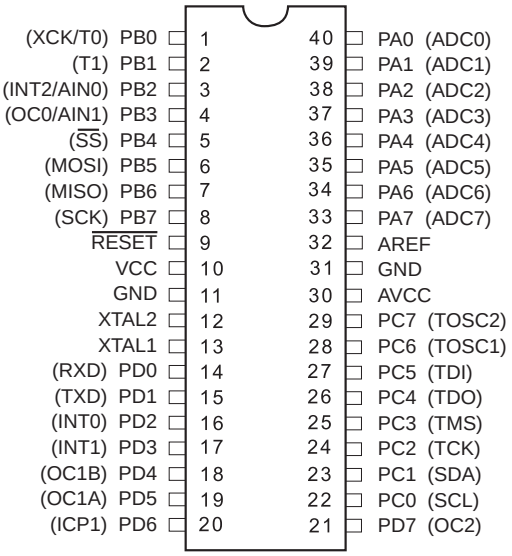


Abbildung 1: Pinbelegung des ATmega16 im 40-poligen DIP-Gehäuse (von oben gesehen).

Die Peripherie-Funktionalität des ATmega16 ist relativ komplex. Einen Überblick darüber geben die Frontseiten der Dokumentation und der Kurzbeschreibung für den ATmega16 (in Stud.IP oder direkt bei der Herstellerfirma Atmel). Um dennoch mit 32 Ein/Ausgabe-Anschlüssen auszukommen, sind die einzelnen Anschlussleitungen mit mehrfacher Funktionalität belegt. So sind beispielsweise die 8 Anschlüsse von Port A entweder als Eingänge für die Analog-Digital-Wandler oder als digitale Ein/Ausgabe-Leitungen verwendbar. In Abbildung 1 sind diese zusätzlichen Funktionen an den Anschlusspins mit angegeben.

Einen Überblick über die Funktionsblöcke der CPU und der auf dem Controller integrierten Peripherie gibt Abbildung 2. Die detaillierte Beschreibung aller Funktionen finden Sie in der Dokumentation des Herstellers.

Die Mehrfachbelegung der Anschlusspins hat den Vorteil, dass die Schaltungs-Layouts im Allgemeinen einigermaßen überschaubar bleiben. Ein großer Nachteil ist jedoch, dass nicht alle Möglichkeiten des Controllers gleichzeitig genutzt werden können. Ein weiterer Nachteil ist, dass man durch fehlerhafte Beschaltung und/oder fehlerhafte Programmierung schwerwiegende Hardwarekonflikte erzeugen kann, die im ungünstigsten Fall zu einer Zerstörung des Controllers oder des angeschlossenen Systems führen kann.

Vermeiden Sie Hardwarekonflikte!

Achten Sie genau darauf, dass Sie Ein/Ausgabe-Pins nicht gleichzeitig in unterschiedlicher Funktionalität benutzen. Achten Sie insbesondere darauf, die durch das JTAG-Interface belegten Pins nicht anderweitig zu verwenden. Schließen Sie das JTAG-Interface nur an der dafür vorgesehenen Steckverbindung an und schließen Sie an dieser Steckverbindung niemals andere Peripherie an.

Für die Verwendung im Praktikum ist der ATmega16 auf einem Modulgehäuse aufgebaut (Abbildungen 3 bis 5). Die notwendigen externen Komponenten und die Anschlussbeschaltung ist mit auf der Platine, das Schaltbild des Moduls finden Sie im Anhang. Die Ein/Ausgabe-Pins der Ports A bis D sind an vier 10-polige Steckverbinder herausgeführt. Deren Anschlussbelegung ist einheitlich: Neben den 8 Bitleitungen stehen Masse und +5 V für periphere Systeme zur Verfügung. Anders beschaltet sind jedoch die weiteren Steckverbinder für Sonderfunktionen wie beispielsweise das JTAG-Interface. Achten Sie darauf, die richtigen Steckverbinder zu benutzen.

Für die Verbindung zu anderen Modulen benutzen Sie bitte nur die Spezialkabel am Versuch (10-polige Flachbandleitungen mit verpolungssicheren Steckern). Achten Sie darauf, sie richtig rum einzustecken (keine Gewalt anwenden).

Zur Spannungsversorgung des ATmega16 wird ein geeignetes einfaches Gleichspannungsnetzteil (8 – 12 V, Steckernetzteil o. ä.) verwendet. Der mit im Modul eingebaute Spannungsregler macht daraus die benötig-

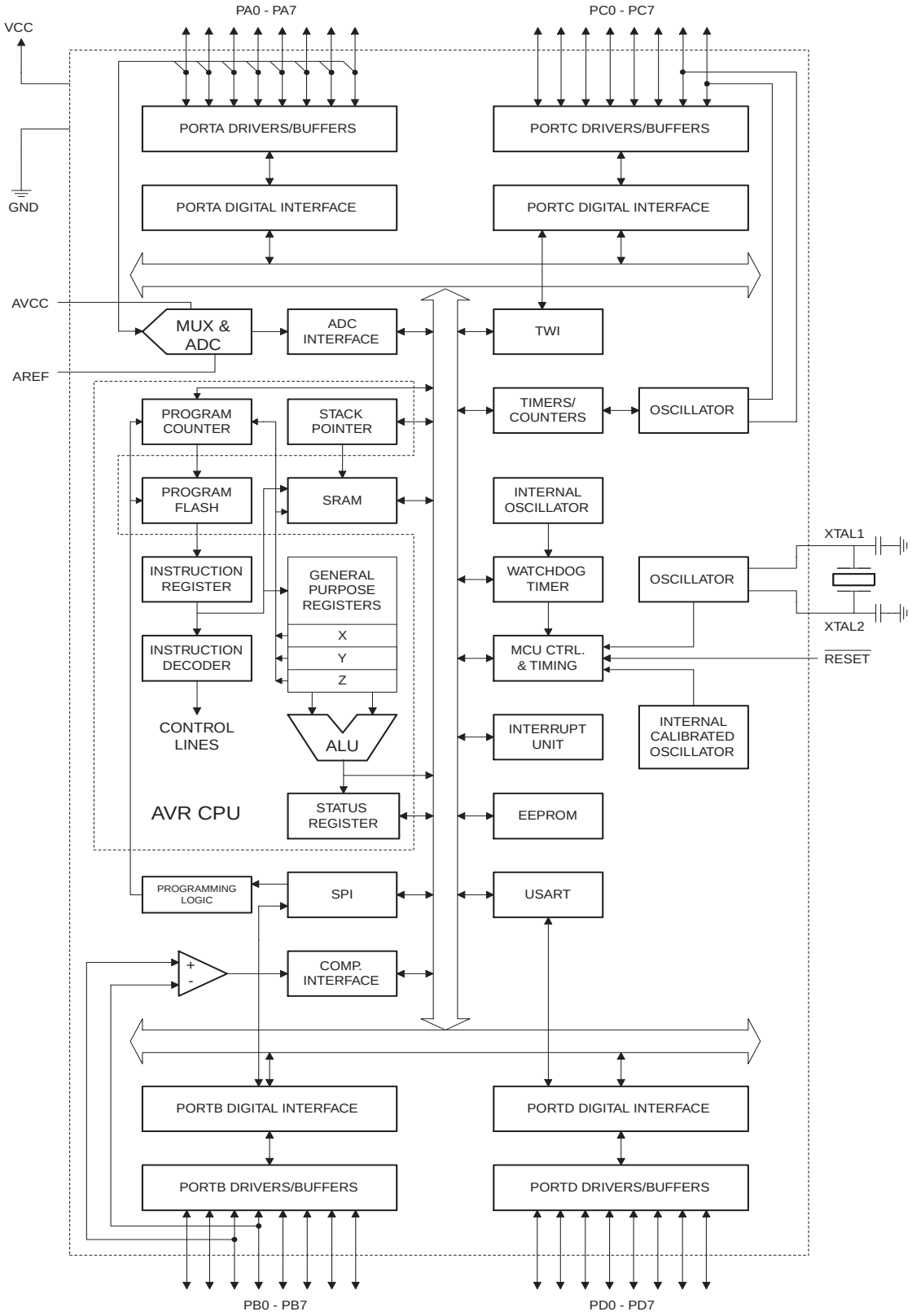


Abbildung 2: Block-Diagramm des ATmega16.

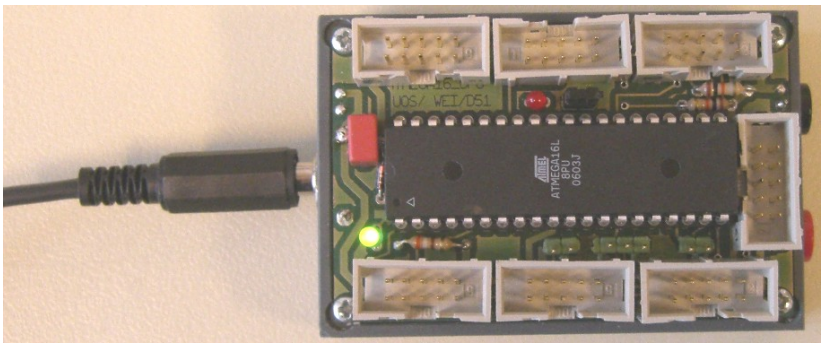


Abbildung 3: ATmega16-Modul, kleine Bauform.

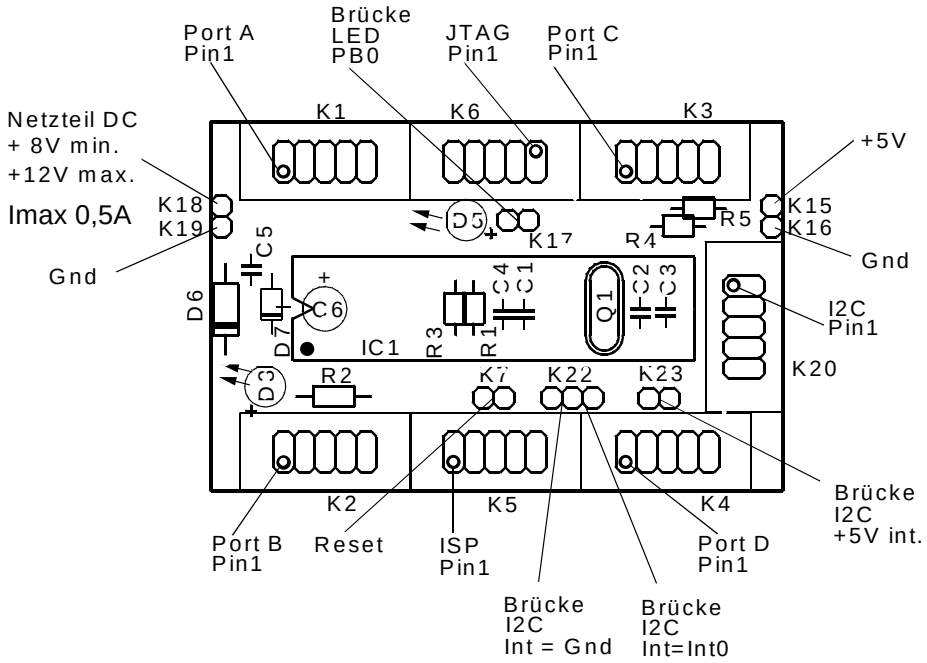


Abbildung 4: Komponenten und Anschlüsse des ATmega16-Moduls.

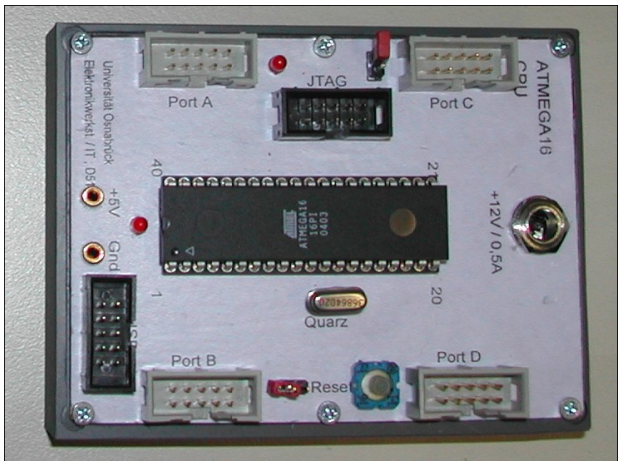


Abbildung 5: ATmega16-Modul, große Bauform.

ten 5V. Stattdessen kann auch eine stabilisierte 5V-Spannungsquelle

direkt an den entsprechenden Buchsen angeschlossen werden.

Die Mikrocontroller werden mit 8 MHz getaktet. Dieser Systemtakt wird in vielen Anwendungen als Basis zur Berechnung von Verzögerungszeiten und Ähnlichem verwendet. Achten Sie bei solchen hardwarenahen Programmen auf die Angabe der richtigen Taktfrequenz. Bei den kleineren Modulen ist der Quarz für die Taktfrequenz fest eingelötet, bei den größeren Modulen in einer Steckfassung und daher bei Bedarf einfach austauschbar.

3.2 JTAG-Interface

Allgemeines zum JTAG-Interface finden Sie bei Wikipedia unter dem Link http://en.wikipedia.org/wiki/JTAG.

Der hier eingesetzte USB-JTAG-Adapter (Abbildung 6) ermöglicht über die JTAG-Schnittstelle des Mikrocontrollers die Programmierung des ATmega16 und die transparente Programmausführung vom PC aus. Der

Abbildung 6: AVR-USB-JTAG-Adapter

Adapter enthält einen USB-Seriell-Konverter als Interface zum PC. Er ist dadurch nicht zum USB-In-Circuit-Emulator vom Atmel (JTAGICE mkII) kompatibel, wohl aber zum älteren JTAG-ICE, der über eine serielle Schnittstelle an den Entwicklungsrechner angeschlossen wird. Zum Betrieb muss am PC zunächst ein Treiber für den Konverter installiert werden, danach ist der USB-Seriell-Konverter als serielle Schnittstelle sichtbar. Passende Treiber können von der Homepage der Herstellerfirma des Konverters Future Technology Devices International (FTDI) heruntergeladen werden: http://www.ftdichip.com/Drivers/VCP.htm. Die richtige Einbindung kann mit dem *Geräte-Manager* des Betriebssystems überprüft werden (*Start* ⇒ *Einstellungen* ⇒ *Systemsteuerung* ⇒ *System* ⇒ *Hardware*). Dort kann, falls notwendig, auch die Schnittstellennummer geändert werden.

Abbildung 7 zeigt den Anschluss des JTAG-Adapters an das ATmega16-Modul (hier die kleinere Modul-Bauform).

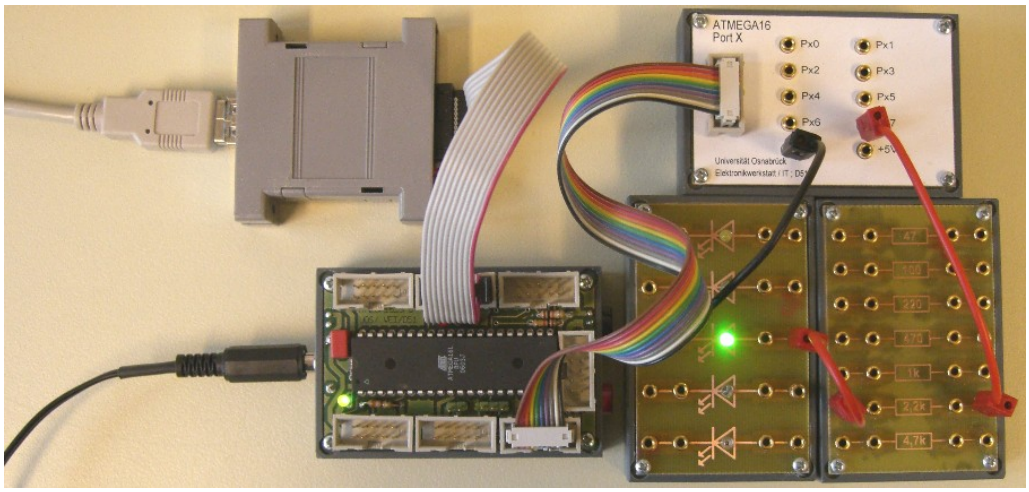


Abbildung 7: JTAG-Adapter am ATmega16-Modul. An Port D ist ein Modul angeschlossen, das die einzelnen Port-Pins auf 2 mm-Steckbuchsen umsetzt. Damit können weitere Module des Praktikums mit den Ports des ATmega16 verbunden werden (hier eine Lumineszenzdiode mit Vorwiderstand).

3.3 LCD-Anzeige

Zur Textausgabe kann ein LCD-Anzeige-Modul an den ATmega16 angeschlossen werden (Abbildung 8). Darauf können 2 Zeilen mit jeweils 16 Zeichen dargestellt werden. Das Datenblatt mit der Beschreibung

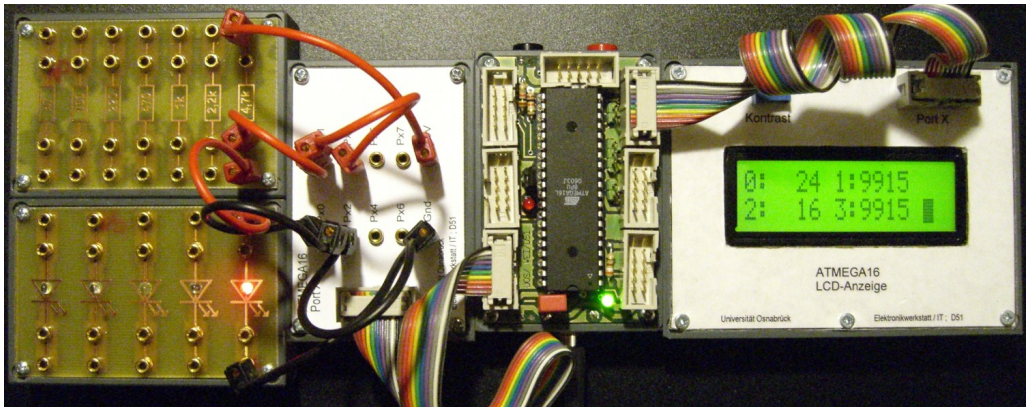


Abbildung 8: LCD-Anzeige-Modul an Port D des ATmega16. Weitere Peripherie an Port A.

des LCD-Moduls finden Sie in Stud.IP (vgl. Abschnitt 2). Das Modul ist hier so beschaltet, dass es nur im so genannten 4-Bit-Modus betrieben werden kann. Damit ist der Betrieb über ein einzelnes Controller-Port möglich. Die Daten-Bytes müssen in 4-Bit-Nibbles aufgeteilt werden, die über 4 der 8 Port-Pins übermittelt werden (Bits 4–7). Drei weitere Pins werden zur Umschaltung zwischen Schreiben und Lesen (Bit 2), zur Umschaltung zwischen Daten und Anweisungen (Bit 1) sowie als Nibble-Clock (“jetzt liegt ein gültiges Nibble an”) benötigt (Bit 0).

3.4 Gleichstrommotor

Das in Abbildung 9 abgebildete Gleichstrommotor-Modul ist so beschaltet (Schaltung im Anhang C), dass es ohne zusätzliche Elektronik direkt an den ATmega16 angeschlossen werden kann. Zur Ansteuerung dient

Abbildung 9: Gleichstrommotor mit Bremslast: Im Modul ist auch die Ansteuerung für den Motor (H-Brücke) eingebaut sowie die Gleichrichtung und Spannungsumsetzung für den Tachogenerator.

ein Logik-IC mit einer so genannten *H-Brücke* als Ausgang, der vom Mikrocontroller mit TTL-Signalen gesteuert wird (Richtungssignal und pulsweitenmodulierte Drehzahlsteuerung). Die der Drehzahl proportionale Tachospannung (-12 . . . +12 V) ist reduziert und unipolar gemacht, somit kompatibel mit den Analog-Digital-Eingängen des Controllers.

Die genauen Daten des Motors und des auf der Motorwelle sitzenden Tachogenerators können den Datenblättern des Herstellers entnommen werden (in Stud.IP, vgl. Abschnitt 2).

Vermeiden Sie eine Überlastung des Motors!

Im Datenblatt finden Sie insbesondere auch den *Thermisch zulässigen Dauerstrom* vermerkt (0.49 A). Diesen sollten Sie unter keinen Umständen überschreiten. Belasten Sie den Motor daher nicht zu stark, es ist ein *Kleinstmotor*. Weder durch die Schaltung noch durch seine Bauart ist der Motor gegen Überstrom geschützt. Die H-Brücke liefert bis zu 3 A, der Anschlusswiderstand des Motors beträgt 9.7 Ω, bei 12 V Betriebsspannung könnten mithin Ströme deutlich über 1 A fließen.

4 Software

Wichtigstes Werkzeug ist das *AVR-Studio*, eine Entwicklungsumgebung zur Softwareentwicklung für die Atmel-Controller, die von der Herstellerfirma Atmel kostenlos zur Verfügung gestellt wird. Ohne weitere Zusätze können damit Assembler-Programme entwickelt werden. GNU-PlugIns ermöglichen die Programmierung in C oder auch C++.

4.1 Entwicklungsumgebung AVR-Studio

Beim Start von AVR-Studio wird entweder ein vorhandenes Projekt geöffnet oder ein neues erstellt. Bei einem neuen Projekt hat man die Wahl

zwischen Assembler und C als Programmiersprachen (wenn das Gnu-PlugIn richtig installiert ist). Im darauf folgenden Fenster kann man sich dann aussuchen, welche Debug-Plattform und welcher Prozessor verwendet wird (Abbildung 10).

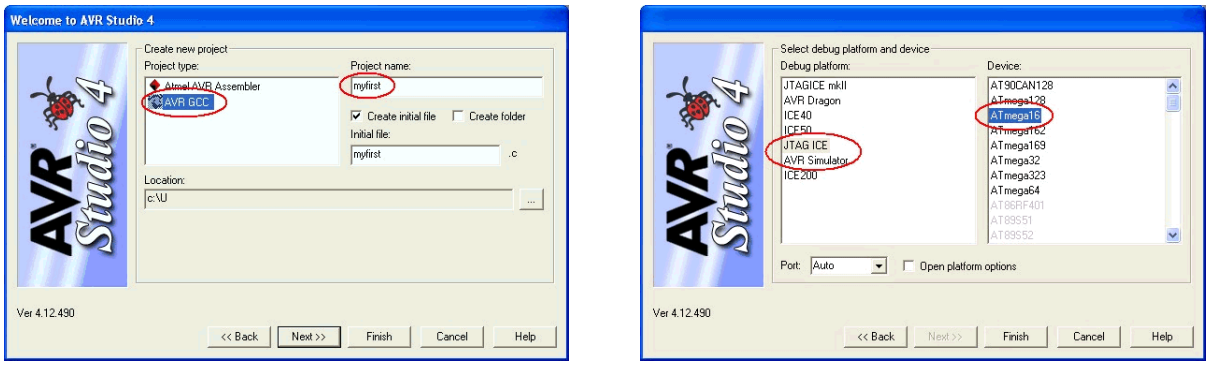


Abbildung 10: Startfenster des AVR-Studio: Links Auswahl des Compilers, rechts Debug-Plattform und Prozessortyp.

4.2 AVR-Studio – Editor

Programme schreiben Sie im Editor-Fenster, dessen Darstellung Sie über *Edit* ⇒ *Font and Color* konfigurieren können (Abbildung 11). Kompiliert wird die aktuelle Datei mit *Build* ⇒ *Compile*, das gesamte Projekt stellen Sie mit den *Build*-Unterpunkten des Menüpunkts zusammen. Im Meldungsfenster am unteren Rand des Hauptfensters werden jeweils die Erfolgs- und Fehlermeldungen ausgegeben.

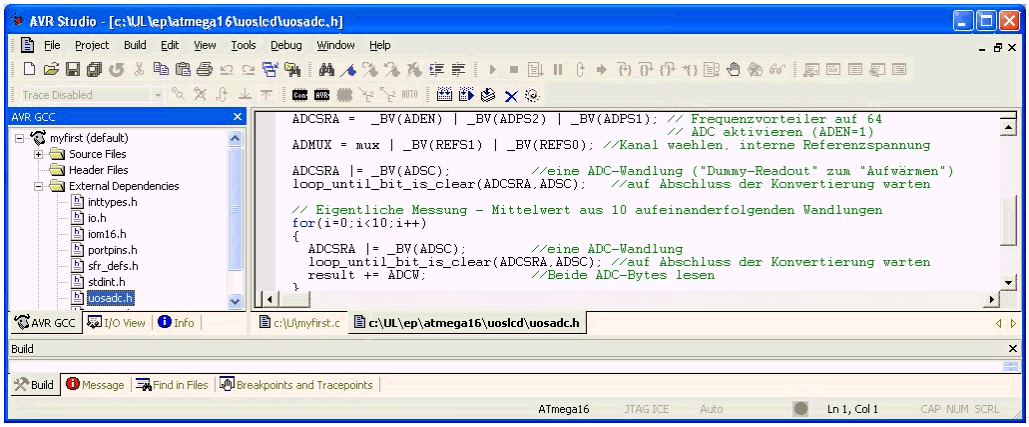


Abbildung 11: Editor-Fenster des AVR-Studio. Links das Datei-Menü-Fenster mit allen Dateien des Projekts.

4.3 AVR-Studio – Debugger

Fertige Programme sollten zunächst mit dem *Debugger* getestet werden. Nach dem Start des Debuggers mit (*Debug* ⇒ *Start Debugging* kann das

Programm schrittweise ausgeführt werden. Über den Menüpunkt *View* können bestimmte Bereiche des Contollers dargestellt werden. In Abbildung 12 ist das der Datenspeicher des Controllers (*View* ⇒ *Memory*).

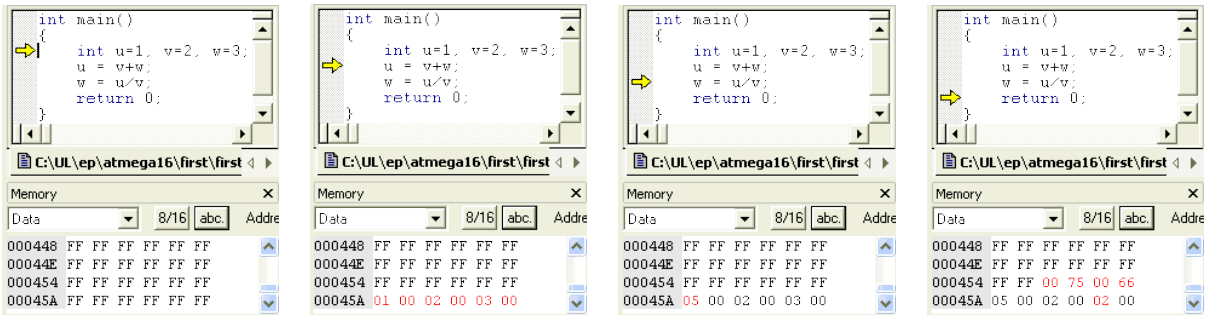


Abbildung 12: Schrittweise Ausführung eines kleinen Programms im Debug-Modus, Veränderung der Variablen im Datenspeicherbereich. Von links nach rechts: ⇒ Programmstart ⇒ nach der ersten Anweisungszeile ⇒ nach der zweiten ⇒ nach der dritten.

Außer dem Speicherbereich kann AVR-Studio auch den gesamten Hardware-Status des Mikrocontrollers darstellen, sowohl in der Simulation wie auch – über das JTAG-Interface – den der konkreten Hardware. Dazu muss, falls noch nicht auf dem Desktop, die Registerkarte I/O über *View* ⇒ *Toolbars* ⇒ *I/O* eingeschaltet werden (Abbildung 13).

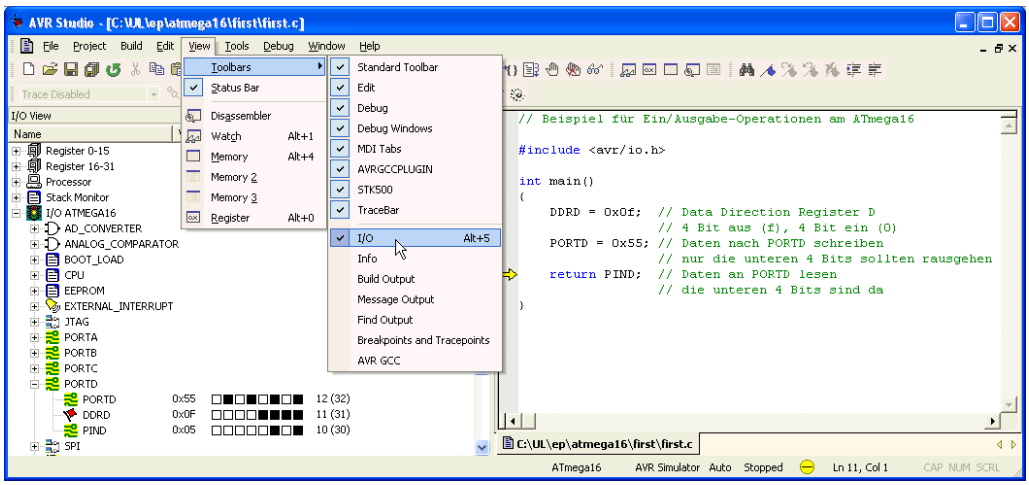


Abbildung 13: Darstellung der Controller-Hardware in der Registerkarte I/O-View *des AVR-Studio.*

Abbildung 14 zeigt, wie sich die schrittweise Ausführung eines kurzen Programms, das auf Port D zugreift, im Port-Status wiederspiegelt.

Im Debugger lassen sich Speicherinhalte, aber auch der Status der Hardware verändern. So können beispielsweise Port-Bits durch einfaches Anklicken umgeschaltet werden. Auf diese Weise kann man die Wirkung auf an den Mikrocontroller angeschlossene Peripherie direkt und

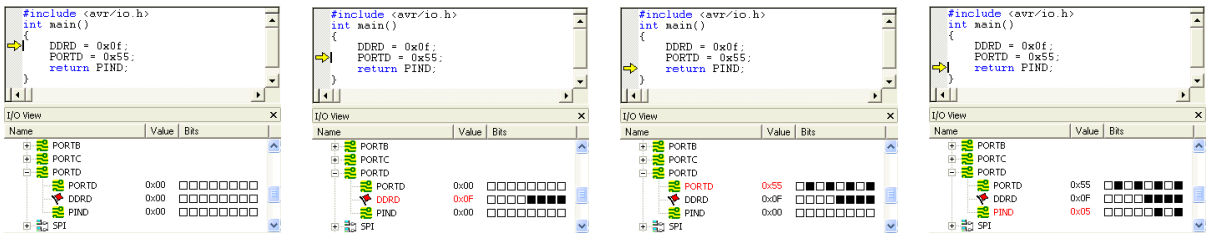


Abbildung 14: Schrittweise Ausführung eines Programms im Debug-Modus, Veränderungen im Status von Port D. Von links nach rechts: ⇒ *Programmstart* ⇒ DDRD, *Bits im* Data Direction Register D *gesetzt* ⇒ PORTD, *Ausgabe auf Port D* ⇒ PIND, *Lesen von Port D.*

schnell testen, bevor man ein Programm in passender Weise modifiziert.

4.4 GNU-Cross-Compiler

Das von Atmel vertriebene AVR-Studio unterstützt nur Assembler-Programmierung. Damit ist ein sehr hardwarenahes Arbeiten möglich (und auch notwendig), das eine exakte Kontrolle über Speicher und Register des Controllers zulässt. Solch eine exakte Kontrolle ist jedoch meist gar nicht nötig, man braucht nicht im Einzelnen zu wissen, welches Controller-Register für welchen Zweck verwendet wird.

Sehr viel komfortabler als Assembler ist das Arbeiten mit einer etwas höheren Programmiersprache. Dies ermöglichen Cross-Compiler wie der GNU-Compiler für die Programmiersprache C. Neben den meist leistungsfähigeren Anweisungen hat eine höhere Programmiersprache den Vorteil größerer Hardwareunabhängigkeit. Assembleranweisungen sind auf die Instruktionen angepasst, die in einem bestimmten Prozessor implementiert sind. Daher gibt es für jede Prozessorfamilie eine darauf abgestimmte Menge von Assemblerbefehlen, die die möglichen CPU-Instruktionen beschreiben. Jede Assembleranweisung wird vom prozessorspezifischen Assembler eindeutig in Maschinencode umgesetzt, umgekehrt kann Maschinencode wieder eindeutig in Assemblercode zurückübersetzt werden (Disassembler). Im Gegensatz dazu ist C weitgehend genormt, daher prozessorunabhängig. Naturgemäß nicht genormt sind spezielle Erweiterungen, insbesondere solche, die Hardwarespezifika betreffen.

Hardwarenahe Programmierung in C kann aufwändig sein, kann aber durch geeignete Makro-Definitionen sehr stark vereinfacht werden. Der GNU-Cross-Compiler hat alle Hardwarefunktionen des Mikrocontrollers auf Makros abgebildet, deren Namen einigermaßen eindeutig eine Funktionszuordnung erlauben. Sie sollten sich diese Art der Ein/Ausgabe-Programmierung angewöhnen, ebenso die Benennung einzelner Pins durch die zugehörigen symbolischen Namen.

Makros für die Ein/Ausgabe-Peripherie

Für die Ein- oder Ausgabe muss eines der Register des Controllers verwendet werden. Eine Assembler-Sequenz, die den ersten beiden Ein/Ausgabe-Anweisungen des Codebeispiels der Abbildung 14 entspricht, könnte etwa so aussehen:

```
.def     temp = r16
   ldi   temp, $0f
   out   DDRD, temp
   ldi   temp, $55
   out   PORTD, temp
```

Gut zu verstehen, aber doch deutlich komplizierter als die entsprechenden Anweisungen in C:

```
DDRD = 0x0f;
PORTD = 0x55;
```

Die Ein/Ausgabe-Anweisungen sind in C als Pseudovariablen implementiert, man kann mit ihnen wie mit richtigen Variablen arbeiten. Wenn man berücksichtigt, dass manche nur zum Lesen und manche nur zum Schreiben gedacht sind. So hat bei reinen Eingabeports (PIND) das Schreiben keine Wirkung, wohl aber kann man Ausgabeports (DDRD) lesen, um den aktuellen Status festzustellen und gegebenenfalls durch Schreiben zu verändern.

Die Makrodefinitionen werden durch `#include <avr/io.h>` am Anfang des Programmcodes eingelesen, die Makros sind jeweils in prozessorspezifischen Dateien definiert (für den ATmega16 ist das `iom16.h`). Sie finden alle zugeladenen Definitionen nach einem Compiler-Lauf unter *External Dependencies* im Datei-Explorer (*View* ⇒ *Toolbars* ⇒ *AVR GCC*) des AVR-Studio.

Makros für die Interrupt-Bearbeitung

Wie für die Ein/Ausgabe sind auch für die Behandlung von Interrupts leistungsfähige Makros definiert, die das Programmieren entsprechender Funktionen deutlich erleichtern (`#include <avr/interrupt.h>`).

So werden mit `sei()` oder `cli()` Interrupts generell erlaubt oder verboten. Eine Funktion zur Bearbeitung eines Interrupts wird mit `ISR(vector)` implementiert, wobei `vector` den symbolischen Namen für den Interrupt enthält (definiert in `iom16.h`, das von `interrupt.h` automatisch zugeladen wird).

Die verschiedenen Interrupt-Kontroll-Register werden wie die Ein/Ausgabe-Register als Pseudovariablen in C behandelt.

Makros zur Bit-Manipulation

Insbesondere bei allen Kontroll-Registern, aber auch bei den Ein/Ausgabe-Registern haben einzelne Bits eine spezielle Bedeutung. Daher ist

es häufig nötig, Einzelbits auf 1 oder 0 zu setzen. Wie schon bei den Registern sollten Sie auch bei den einzelnen Bits stets die aussagekräftigen symbolischen Namen verwenden. Typische Redewendungen in C für die Bearbeitung einzelner Bits sind

```
GICR |= (1<<INT2);
```

für das Setzen von Bit INT2 im Register GICR (der externe Interrupt vom Pin INT2 wird zugelassen, vgl. Abbildung 1) oder

```
GICR &= ~(1<<INT2);
```

für das Rücksetzen des Bits (der Interrupt wird wieder gesperrt).

Da diese Art der Umrechnung von Bitnummern in die entsprechenden Byte-Konstanten sehr häufig vorkommt, ist dafür das Makro `_BV` definiert. Die obigen beiden Anweisungen würde man damit so formulieren:

```
GICR |= _BV(INT2);
GICR &= ~_BV(INT2);
```

Mehrere Bits in einem Register werden geeignet ge*oder*t. So sagt

```
ADCSRA = _BV(ADEN) | _BV(ADPS2) | _BV(ADPS1);
```

dem ADC-Status-Register, was gerade Sache ist.

Um den Zustand einzelner Bits abzufragen oder um auf den Zustand einzelner Bits zu warten, sind die folgenden Makros definiert:

```
bit_is_set(sfr, bit)
bit_is_clear(sfr, bit)
loop_until_bit_is_set(sfr, bit)
loop_until_bit_is_clear(sfr, bit)
```

Sie fragen ab, ob das Bit `bit` im Register `sfr` 1 oder 0 ist, bzw. warten, bis dieser Zustand erreicht ist.

Portierung von und zu anderen Compilern

Die hardwarespezifischen Erweiterungen zu C sind nicht einheitlich, unterschiedliche Compiler-Hersteller verwenden unterschiedliche Funktionen und Makros. Darauf ist zu achten, wenn man Beispielprogramme aus dem Internet verwenden will. Freundliche Programme berücksichtigen dies durch bedingte Preprozessor-Anweisungen wie beispielsweise

```
#if defined(__ICCAVR__)
  #include <iotiny2313.h>
  #include <inavr.h>
#elif __GNUC__ > 0
  #include <avr/io.h>
  #include <avr/interrupt.h>
  #include <avr/wdt.h>
#else
  # error "Unsupported compiler."
```

```
#endif
```

oder

```
#if defined(__ICCAVR__)
  // correct an IAR definition for
  // Watchdog Timer Control Register,
  // may not be needed in later versions.
  #define WDTCSR  WDTCR
#elif __GNUC__ > 0
  // redefine some macros
  #define __C_task int
  #define __enable_interrupt()  sei()
  #define __watchdog_reset()    wdt_reset()
#endif
```

Unfreundliche Programme unterlassen das. Sie sollten dann gegebenenfalls wissen, was Sie von Hand nachbessern müssen.

4.5 Beispielprogramme

Vieles zu den Atmel-Controllern, Beschreibungen typischer Strategien und Beispielprogramme, finden Sie im Internet. Ein paar Anhaltspunkte dazu sind im Abschnitt 2 angegeben. Beispiele zur Programmierung der Analog/Digital-Wandler und des LCD-Anzeigemoduls sind im Anhang zusammengestellt. Im Folgenden noch ein paar kurze Fragmente, die das Arbeiten mit Interrupts und Pulsweitenmodulation erläutern.

Externer Interrupt

Das folgende kurze Programm zeigt die prinzipielle Vorgehensweise:

```
#include <AVR/interrupt.h>

ISR (INT0_vect)
{
  PORTD ^= _BV(0);
}

int main(void)
{
  DDRD = _BV(0);
  MCUCR |= (_BV(ISC01)|_BV(ISC00));
  GICR |= _BV(INT0);
  sei();
  for (;;)
    ;
  return 0;
}
```

In der Interrupt-Service-Routine wird Bit 0 von PORTD bei jedem Aufruf umgeschaltet. Das Hauptprogamm legt dieses Bit als Ausgang fest und definiert als Interrupt-Quelle die ansteigende Flanke des INT0-Eingangspins. Nach dem Setzen des Interrupt-Enable-Flags mit `sei` ergeht es sich in einer leeren Endlosschleife. Machen Sie sich die Bedeutung der verwendeten Makros und der symbolischen Konstanten und Pseudovariablen klar.

Pulsweitenmodulation und externer Interrupt

Die folgende Änderung gibt ein pulsweitenmoduliertes Rechtecksignal auf Pin OC2 aus. Bei jedem Betätigen des an INT0 angeschlossenen Schalters wird der Wert des Vergleichsregisters um 8 erhöht.

```
ISR (INT0_vect)
{
  OCR2 += 8;
}

int main(void)
{
  TCCR2 = _BV(WGM21) | _BV(WGM20) | _BV(COM21) | _BV(CS20);
  DDRD = _BV(DDD7);
  ...
}
```

Pulsweitenmodulation und Timer-Interrupt

Soll das Vergleichsregister bei jedem Zählerüberlauf auf den aktuellen Wert gesetzt werden, könnte das wie folgt aussehen:

```
ISR (TIMER2_OVF_vect)
{
  OCR2 = (PIND&0x0f)<<4;
}

int main(void)
{
  TCCR2 = _BV(WGM21) | _BV(WGM20) | _BV(COM21) | _BV(CS20);
  DDRD = _BV(DDD7);
  TIMSK |= _BV(TOIE2);
  sei();
  ...
```

Die Interrupt-Service-Routine liest die vier niederwertigen Bits von PORTD (die beispielsweise mit dem Schalter-Modul verbunden sind) und setzt die vier oberen Bits des Vergleichsregisters von Timer2 entsprechend.

5 PID-Regelung

Eine relativ ausführliche Beschreibung zur Funktion, Simulation und Realisierung von analogen und digitalen Reglern finden Sie unter dem in Abschnitt 2 angegebenen Link zum Thema Regelungstechnik. Hier ein paar Ergänzungen dazu.

Regelstrecke in LTspice

In dem Artikel zur Regelungstechnik sind einige Modelle für Regelstrecken angegeben (PT1-Glied, Integrator, Verzögerung), die allerdings nicht ohne weiteres zugänglich sind. Sie können sich solche Modelle selbst erstellen, indem sie eine spannungsgesteuerte Spannungsquelle aus LTspice verwenden (E-Source) und dort die gewünschte Übertragungsfunktion direkt angeben (vgl. LTspice-Hilfe zu *Circuit Elements* und dort *E. Voltage Dependent Voltage Source*). Für ein PT2-Glied, das den verwendeten Gleichstrommotor vermutlich beschreiben kann, würde man die dafür zuständige Übertragungsfunktion unter *Value* mit der Syntax *Laplace=* eingeben, z. B.:

```
Laplace=0.8/(1+0.01*s)/(1+0.01*s)   .
```

Damit wären identische Verzögerungszeiten von jeweils 10 ms und eine Verstärkung der Regelstrecke von 0.8 eingestellt. Für eine zusätzliche Totzeit von beispielsweise 5 ms würden Sie die Übertragungsfunktion noch mit der dafür zuständigen Exponentialfunktion multiplizieren

```
Laplace=0.8/(1+0.01*s)/(1+0.01*s)*exp(-0.005*s)   .
```

Open-Loop-Verhalten

Zunächst sollte man das *Open-Loop*-Verhalten des Systems untersuchen, am einfachsten mit einem rechteckförmigen Sollwert und mit unterschiedlicher Belastung, die man in LTspice beispielsweise durch eine unterschiedliche Verstärkung der Regelstrecke simulieren kann.

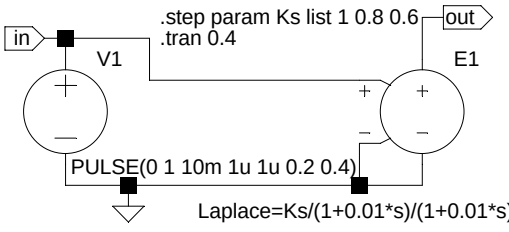


Abbildung 15: Schaltung zur Untersuchung des Open-Loop-Verhaltens eines Regelsystems.

Abhängig von der Belastung weicht die Ist-Drehzahl mehr oder weniger stark von der vorgegebenen Soll-Drehzahl ab. Dies zeigt Abbildung 16.

Aus der Sprungantwort der *Open Loop* erhält man Informationen über die Eigenschaften der Regelstrecke, die man zur Dimensionierung des Reglers heranziehen kann (vgl. Internet-Artikel). Nachstehend zum Vergleich die Sprungantwort bei zusätzlicher Totzeit.

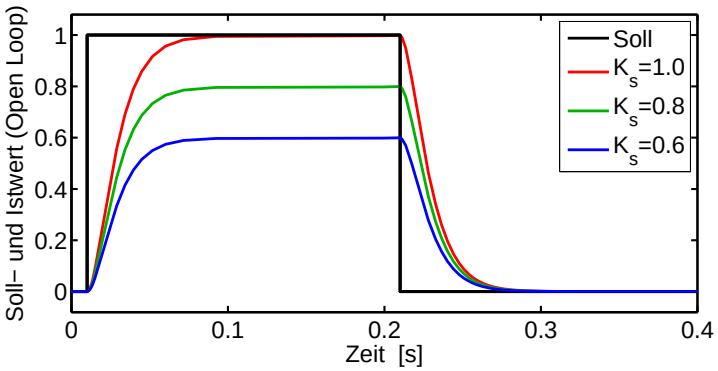


Abbildung 16: Verhalten eines ungeregelten Systems (Open Loop): Die Ist-Drehzahl ist stark lastabhängig. Verschiedene Belastungen werden durch unterschiedliche Verstärkungen $K_s \leq 1$ der Regelstrecke simuliert.

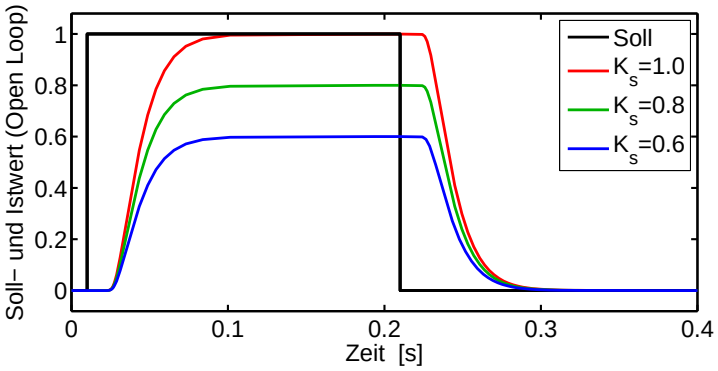


Abbildung 17: Verhalten eines ungeregelten Systems (Open Loop) mit Totzeit: Gegenüber Abbildung 16 tritt noch eine zusätzliche Verzögerung in der Sprungantwort auf.

PI-Regler-Simulation

Abbildung 18 zeigt die komplette Schaltung für einen analogen PI-Regler, wie er in LTspice realisiert werden kann. Es sind *ideale* Operationsverstärker verwendet, im Gegensatz zum genannten Internet-Artikel sind P- und I-Teil durch getrennte OPs modelliert.

Das Regelverhalten mit den gleichen Lastbedingungen wie bei der Open-Loop-Simulation zeigt Abbildung 19. Es wird deutlich, dass beim PI-Regler der Sollwert auch bei Belastung erreicht wird. Das Zeitverhalten ist allerdings lastabhängig.

Digitale Regelung: Pulsweitenmodulation (PWM)

Beim Analogregler wird die Stellgröße meist als Ausgangsspannung eines Operationsverstärkers berechnet (Ausgang von U4 in Abbildung 19). Dies ist in der Simulation unproblematisch und vereinfacht dort die Handhabung. In der realen Ausführung wird an dieser Stelle jedoch

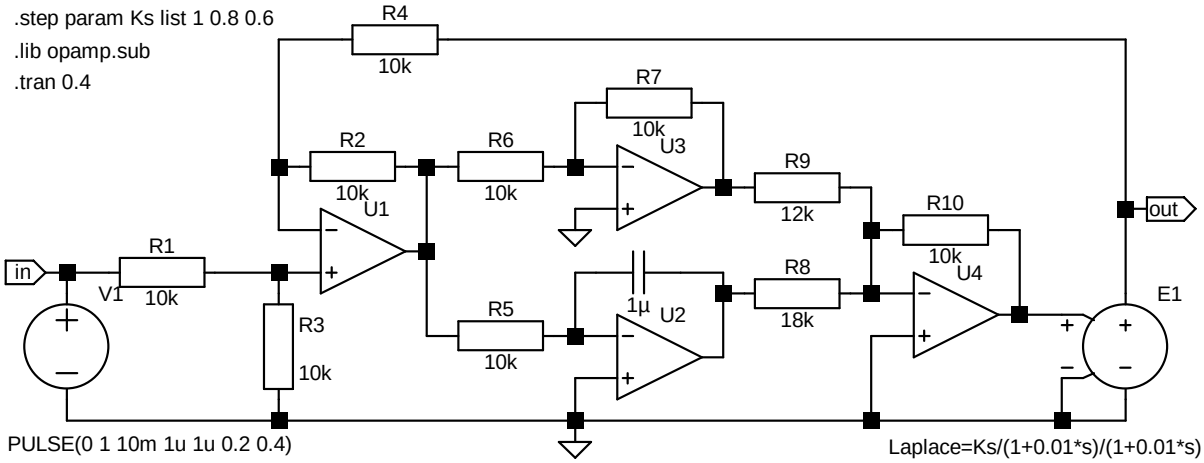


Abbildung 18: PI-Reglerschaltung mit Regelstrecke. Die einzelnen Funktionen sind der Übersichtlichkeit wegen auf verschiedene OPs verteilt: U1 berechnet die Regelabweichung, U2 ist für den Integral-, U3 für den Proportionalteil zuständig, U4 schließlich bildet die gewichtete Summe daraus.

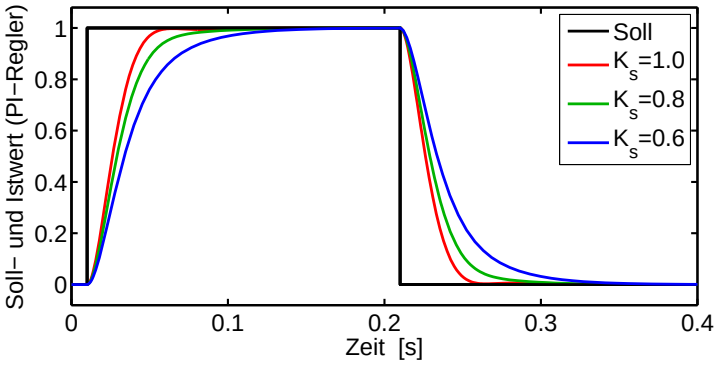


Abbildung 19: Verhalten eines geregelten Systems (PI-Regler): Die Ist-Drehzahl erreicht den Vorgabewert auch bei Belastung, allerdings unterschiedlich rasch.

meist eine beträchtliche elektrische Leistung umgesetzt (Motor, Heizung etc.), so dass ein Leistungsverstärker benötigt wird. In diesem wird dann ein großer Teil der Gesamtleistung nutzlos in Wärme umgewandelt. Solche *Linearen Regler* haben mithin einen sehr schlechten Wirkungsgrad.

Sehr viel bessere Effizienz erreicht man mit so genannten *Schaltreglern*, bei denen zeitlich getaktet die volle Betriebsspannung ein- und ausgeschaltet wird. Die gemittelte Spannung ist dann die Betriebsspannung multipliziert mit dem Verhältnis aus Einschaltzeit und Periodendauer

$$U_\mathrm{m} = U_\mathrm{B} \frac{t_\mathrm{ein}}{t_\mathrm{ein} + t_\mathrm{aus}} \ . \tag{1}$$

U_m kann durch Variation von t_ein (bei konstanter Periodendauer $t_\mathrm{ein} + t_\mathrm{aus}$) auf beliebige Werte zwischen 0 und U_B eingestellt werden.

Weitere Informationen zu Schaltreglern und zur Pulsweitenmodulation finden Sie unter diesen Stichworten im Internet, beispielsweise auch bei den in Abschnitt 2 dazu angegebenen Links.

Das Prinzip der Pulsweitenmodulation hat sich auch bei den Mikrocontrollern voll durchgesetzt, daher enthalten diese zwar Analog-Digital-Wandler, nur selten jedoch Digital-Analog-Wandler. Stattdessen bieten sie meist komfortable Möglichkeiten zur Pulsweitenmodulation. Dies ist auch beim ATmega16 der Fall.

Die pulsweitenmodulierte Ausgangsspannung ist ein Rechtecksignal mit beträchtlicher Amplitude. Benötigt man eine absolut glatte Ausgangsspannung, kombiniert man die PWM-Schaltung mit einem Tiefpass. Das wird bei allen Spannungswandlern gemacht, die nach diesem Prinzip arbeiten. Außerdem verwendet man möglichst hohe Frequenzen, damit der Tiefpass mit möglichst kleinen Kapazitäten oder Induktivitäten auskommt.

Im Gegensatz zu den Spannungswandlern kann man bei anderen technischen Anwendungen meist auf einen speziellen Tiefpass verzichten, da die Regelstrecke selbst ausgeprägtes Tiefpassverhalten aufweist. Das ist bei Motoren, Heizungen, Glühlampen oder ähnlichen trägen Lasten immer gegeben. Darüber hinaus gibt es Anwendungsbereiche, bei denen zwar die Last schnell reagiert, dies aber für die spezielle Anwendung keine Rolle spielt. Das ist unter anderem bei der Helligkeitssteuerung von Lumineszenzdioden durch Puslweitenmodulation der Fall.

6 Aufgaben

Bearbeiten Sie zunächst die Aufgaben 0 bis 4, wählen Sie für die restliche Zeit geeignete Wahlaufgaben aus.

0 **Mikrocontroller ATmega16.** Als Vorbereitung zum Projekt sollten Sie sich mit den wichtigsten Eigenschaften des Mikrocontrollers ATmega16 vertraut machen: Speichertypen auf dem Chip, Ein/Ausgabe-Ports, Analog/Digital-Wandler, Timer für Zeittakt und Pulsweitenmodulation, externe und interne Interrupts.

0 **Software.** Ebenfalls als Vorbereitung sollten Sie sich mit der Software beschäftigen: Die Entwicklungsumgebung *AVR-Studio*, der GNU-Cross-Compiler, die Programmiersprache C, vordefinierte Makros zur einfacheren Programmierung von Mikrocontrollern. Alles ist *Freeware*, Sie können problemlos auf Ihrem eigenen Rechner üben.

!! **Hardwarekonflikte:** Beachten Sie bei allen Beschaltungen des ATmega16, dass das JTAG-Interface 4 Leitungen von Port C verwendet. Es sind die Pins **PC2, PC3, PC4, PC5,** über die die JTAG-Anschlüsse TCK, TMS, TDO, TDI laufen. Vermeiden Sie unter allen Umständen, diese Pins für andere Funktionen zu benutzen. Unvorhersehbare Fehlfunktionen des JTAG-Interfaces sind ansonsten unvermeidlich. Am einfachsten ist es, bei Schaltungen auf Port C komplett zu verzichten.

1 **Programmierung.** Lösen Sie mit der Enwicklungsumgebung zunächst einfache Aufgaben, die noch ohne das Ein/Ausgabe-System des Controllers arbeiten. Beispiel: Addition, Subtraktion, Multiplikation o. ä. Schreiben Sie dazu ein kurzes C-Programm, das Sie zunächst im Simulator testen. Sehen Sie sich an, in welchen Registern die Daten gespeichert werden, wie sind die Ergebnisse? Arbeiten Sie mit *Breakpoints*.

2 **Ein- und Ausgabe.** Benutzen Sie nun zwei Ports des Controllers, z. B. Port A als Eingabeport, Port D als Ausgabeport. Schreiben Sie eine C-Funktion, die als Ergebnis den Status des Eingabeports zurückliefert. Schreiben Sie eine zweite Funktion, die ins Ausgabeport schreibt. Verwenden Sie beide Funktionen in einem Hauptprogramm, das in einer Endlosschleife Port A liest und den Status nach Port D überträgt. Testen Sie das Programm in der Simulation und auf der Hardware.

Schließen Sie geeignete Peripherie an (z. B. vier Schalter am Eingang, vier LEDs am Ausgang). Test.

Erweitern Sie durch logische Verknüpfungen. Beispiele:

- Eine LED leuchtet nur, wenn alle Schalter auf ‘1’ stehen, eine andere, wenn genau ein Schalter auf ‘1’ steht, eine dritte, wenn höchstens zwei Schalter auf ‘1’ stehen.

- Eine LED zeigt das Ergebnis der UND-Verknüpfung von zwei Schaltern, eine andere das Ergebnis der XOR-Verknüpfung (Halbaddierer).
- Mit jeweils zwei Schaltern können Sie 2-Bit-Zahlen darstellen, mit drei LEDs eine 2-Bit-Zahl sowie Überlauf (*Carry*). Bauen Sie einen Addierer für 2-Bit-Zahlen.

3 **LCD-Anzeige.** Verwenden Sie das LCD-Anzeige-Modul zur Ausgabe. Stellen Sie die Ein- und Ausgabebits der vorstehenden Aufgabe in geeigneter Form dar.

4 **Digitalmultimeter.** Sie können mit dem eingebauten A/D-Wandler bis zu acht Spannungen über Port A messen. Mit der eingebauten Referenz ist der Messbereich 0. . . 2.56 V. Schreiben Sie ein Programm, das zwei oder vier Spannungen gleichzeitig digitalisiert und die Messergebnisse in mV auf dem LCD-Display ausgibt. Messbeispiele: Durchlassspannungen an unterschiedlichen Dioden (vgl. Abbildung 8).

Wahlaufgaben

5 **Externe Interrupts.** Machen Sie sich mit der Interrupt-Programmierung des Mikrocontrollers vertraut. Schließen Sie einen der Schalter an einen der externen Interrupt-Eingänge an und schreiben Sie ein Programm, das bei der Betätigung des Schalters (ansteigende Flanke) eine LED abwechselnd ein- und ausschaltet. Erweitern Sie das Programm so, dass eine zweite LED bei jedem Einschalten der ersten umgeschaltet wird. Sie haben damit einen 2-Bit-Zähler mit LED-Anzeige gebaut.

Statt des Schalters können Sie den Funktionsgenerator mit einem Rechtecksignal anschließen. Achten Sie darauf, dass die Rechteckspannung von 0 bis 5 V geht und diesen Bereich auf keinen Fall überschreitet (Test mit dem Oszilloskop). Stellen Sie das Eingangssignal und die Ausgangssignale auf dem Oszilloskop dar (Frequenzteiler). Wählen Sie dazu eine sinnvolle Frequenz.

Messen Sie mit dem Oszilloskop die Zeitverzögerung zwischen Interrupt-Anforderung und Antwort. Was lässt sich daraus über das *Echtzeitverhalten* aussagen?

Erhöhen Sie die Frequenz und testen Sie, wie schnell Interrupts in Ihrem Programm aufeinander folgen dürfen, damit sie noch sicher abgearbeitet werden. Geschätzte Laufzeit der Interrupt-Routine?

6 **Pulsweitenmodulation.** Der Mikrocontroller hat keine Digital/Analog-Wandler; benötigt man diese Funktionalität, wird ein pulsweitenmoduliertes Rechtecksignal (PWM) verwendet. Dazu sind 3 Timer/Counter mit auf dem Controller integriert, zwei mit 8 Bit, einer mit 16 Bit Auflösung. Zu den 8-Bit-Timern gehört jeweils ein Vergleichsregister, zum 16-Bit-Timer dagegen zwei. Insgesamt kann man mithin

4 Vergleichsereignisse festlegen, die sich in den vier *Output-Compare*-Portleitungen OC0, OC1A, OC1B und OC2 wiederspiegeln (vgl. Abbildung 1). Für diese vier Ausgangs-Pins kann man fixe Hardwareabläufe programmieren, z. B. ein Rechtecksignal mit fester Frequenz oder ein PWM-Signal, dessen Charakteristik im Progamm von Zeit zu Zeit verändert wird. Darüber hinaus lassen sich PWM-Signale auch auf anderen Ausgabe-Pins per Software realisieren, indem man Interrupt-Routinen an die Timer bindet.

Machen Sie sich mit der Timer/Counter-Programmierung des Controllers vertraut, insbesondere mit dem 16-Bit-Timer (*Timer/Counter1*). Welche Bits in welchen Registern sind zu setzen, um eine gewünschte Funktion einzustellen? Welche (internen) Interrupts sind mit dem Timer verbunden?

Schreiben Sie ein Programm, das den 16-Bit-Timer im 10-Bit-Modus verwendet. Der Timer soll ein *Fast-PWM*-Signal mit möglichst hoher Grundfrequenz erzeugen. Verwenden Sie den *Timer-Overflow*-Interrupt, um das Vergleichsregister nach jedem Timer-Ablauf neu zu setzen. Dazu soll jeweils der Status des Schaltermoduls gelesen werden, die Schalterstellung übertragen Sie sinnvoll auf die obersten vier Bit des 10-Bit-Vergleichswerts. Schließen Sie eine LED an und zeigen Sie, dass Sie deren Helligkeit nun über die Schalter einstellen können. Dokumentieren Sie einige typische PWM-Signale mit dem Oszilloskop.

7 Analogrechner. Kombinieren Sie zwei Analog/Digital-Wandler und das PWM-Signal, um einfache analoge Rechenoperationen zu realisieren (Addition, Subtraktion, Multiplikation, Quadrierung, Logarithmus, Winkelfunktionen o. ä.). Beachten Sie, dass Sie nicht mit negativen Spannungen arbeiten können. Beschränken Sie sich daher entweder auf positive Größen oder legen Sie einen sinnvollen Nullpunkt im positiven Bereich fest.

Zweckmäßigerweise strukturieren Sie das Programm so, dass die Analog/Digital-Wandlung und die Berechnung des Ergebnisses in einer Endlosschleife im Hauptprogramm abläuft. Wie in der vorstehenden Aufgabe kann das Vergleichsregister des Timers in einer Interrupt-Routine auf den Ergebniswert gesetzt werden.

Führen Sie Rechnungen mit langsam veränderlichen Signalen durch. Beispiele: Multiplikation eines Sinussignals mit einer variablen Konstanten (Potentiometer), Quadrierung eines Dreiecksignals, Logarithmus oder Winkelfunktion eines Dreiecksignals. Überlegen Sie, wie Sie das PWM-Ausgangssignal glätten.

Erhöhen Sie die Signalfrequenzen und beobachten Sie die Wirkung. Messen Sie die *Sprungantwort* mit einem Rechtecksignal – ohne weitere Rechnung und mit einer etwas aufwändigeren mathematischen Funktion.

8 Drehzahlregelung eines Gleichstrommotors. Ein in der Technik häufig auftretendes Problem: Die Drehzahl eines Motors soll in einem weiten Bereich einstellbar sein und die einmal eingestellte Drehzahl soll dann – unabhängig von der Belastung – möglichst konstant bleiben. Darüber hinaus soll der Motor langsamen und schnellen Änderungen des Vorgabewerts prompt folgen, ohne oder mit nur geringer zeitlicher Verzögerung, aber auch ohne über den Zielwert hinaus zu schießen. Ein typisches Regelproblem.

Informieren Sie sich (z. B. unter dem in Abschnitt 2 angegebenen Link) über Grundlagen der Regelungstechnik. Zur Vorbereitung ist es zweckmäßig, das Regelverhalten eines Analogreglers in LTspice zu modellieren (vgl. Abschnitt 5).

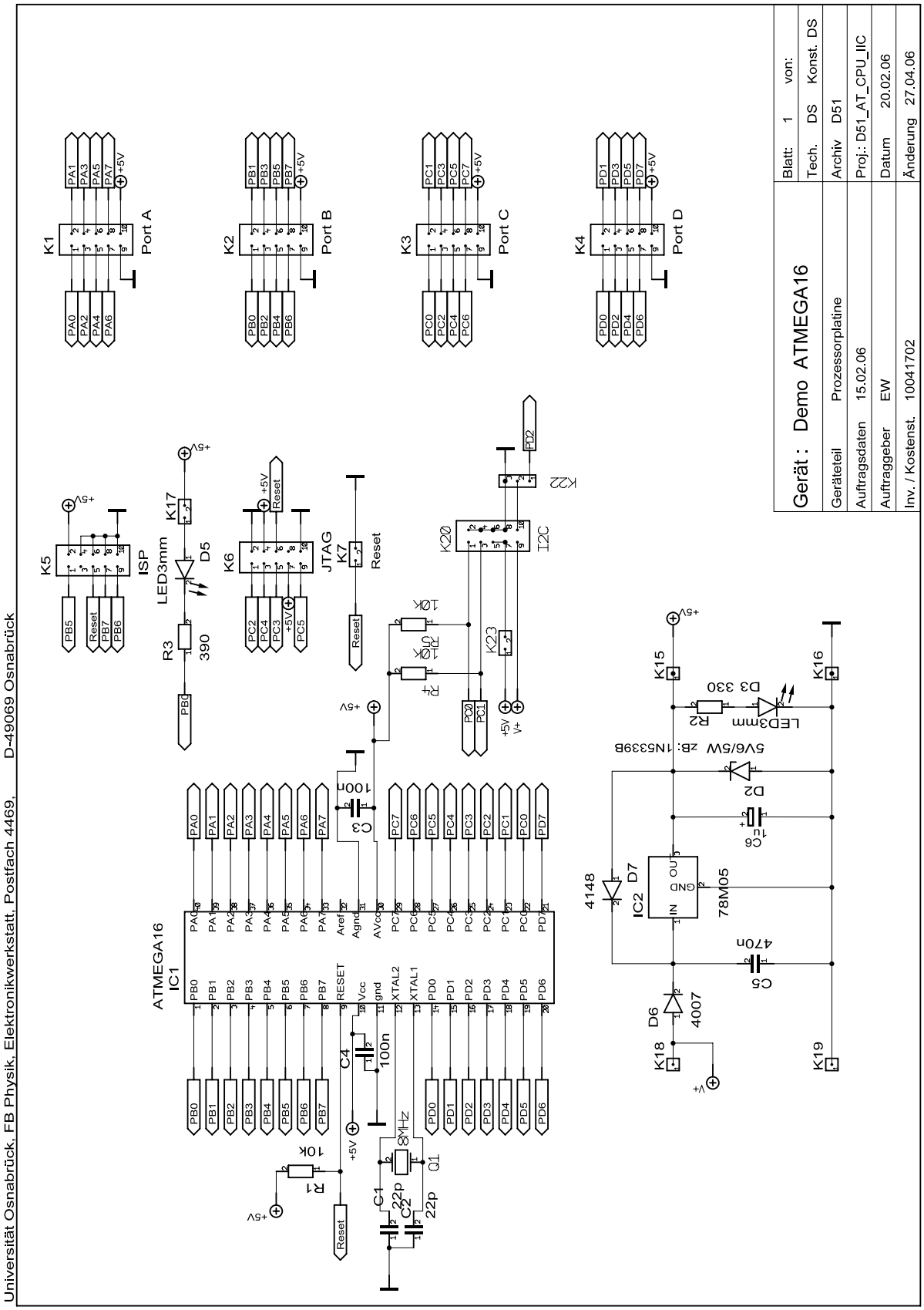
Beginnen Sie mit einem Programm zur *Steuerung* der Drehzahl des Gleichstrommotors (ohne Regelung, *Open-Loop Controller*). Bereiten Sie dabei schon die Algorithmen vor, die Sie später beim Reglerprogramm brauchen. Der Motor wird durch ein pulsweitenmoduliertes Signal angesteuert (10 Bit oder 8 Bit). Die Drehzahl soll durch eine analoge Spannung an einem der Analog-Digital-Wandler-Eingänge vorgegeben werden. Das kann entweder statisch mit Hilfe eines Potentiometers erfolgen oder zeitabhängig durch den Funktionsgenerator. Das Analogsignal wird in einem fest vorgegebenen Zeittakt gemessen (Totzeit). Studieren und dokumentieren Sie das Verhalten (Messung mit dem Tachogenerator): Lastabhängigkeit der Drehzahl, Antwortverhalten auf veränderliche Sollspannungen (Rechteck, Dreieck).

Erweitern Sie das Programm zu einer PI-Regelung. Dazu muss über einen zweiten Analog-Kanal die Tachospannung gemessen werden. Setzen Sie den I-Teil zunächst auf Null und studieren Sie das Verhalten des Proportionalreglers (Lastabhängigkeit und Antwortverhalten bei unterschiedlichen Verstärkungsfaktoren K_p). Nehmen Sie dann den I-Teil dazu und optimieren Sie die Parameter.

Ergänzung: Sie können das zum Versuch gehörende Schaltermodul mit den vier Schaltern dazu verwenden, zwischen unterschiedlichen Programmvarianten umzuschalten, unterschiedliche Parameter einzustellen oder zwischen Rechts- und Linkslauf umzuschalten.

Zusatz: Erweiterung durch einen D-Teil zum PID-Regler.

A Schaltung der ATmega16-Box



B

AVR-USB-JTAG DEVELOPMENT TOOL FOR AVR MICROCONTROLLERS WITH JTAG INTERFACE

Features:

AVR-USB-JTAG (complete analog of ATMEL's AVR JTAG ICE) is development tool for programming, real time emulation and debugging for AVR microcontrollers with JTAG interface (ATmega16, ATmega32, ATMega323, ATmega162, ATmega169, ATmega128 and all other future to come). AVR-USB-JTAG have: JTAG 10 pin connector (Atmel layout), status LED, USB type A connector.

AVR-USB-JTAG works transparently with AVRSTUDIO as it uses USB to RS232 driver inside and AVRSTUDIO just finds ATJTAGICE connected to the virtual COM port which FT232 driver installs. AVR-USB-JTAG allow debugging on all new computers and notebooks which have no RS232 COM port and removes all problems which occur when unknown/uncertified USB to RS232 port converters are used with the original RS232 based ATJTAGICE.

AVR-USB-JTAG is **optically isolated** from USB port, so you can debug directly connected to 120/220VAC mains targets.

AVR-USB-JTAG allows access to all the powerful features of the AVR microcontroller. All AVR resources can be monitored: Flash memory, EEPROM memory, SRAM memory, Register File, Program Counter, Fuse and Lock Bits, and all I/O modules. AVR-JTAG also offers extensive On-chip Debug support for break conditions, including break on change of Program memory flow, Program memory Break Points on single address or address range, and Data memory Break Points on single address or address range.

- AVR Studio Operated
- Full Emulation of All Analog and Digital
- Full JTAG Programming Support
- Supports Multiple Devices in a JTAG Scan Chain
- RS-232 Interface to PC
- Full Support for Assembly and High Level Languages
- Program and Data Breakpoints
- All Operations and Breakpoints are Real Time
- Upgrades are done from AVR Studio
- Target Voltage 3.0-5.0V
- No need for external power supply – power is taken from USB port

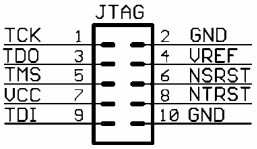
Programming:

To work with AVR-USB-JTAG you need target board with JTAG connector (for instance Olimex's AVR-M16 + AVR-P40B-P8535-8Mhz) and AVRSTUDIO software (you can download from **http://www.avrfreaks.net** in Tools section).

JTAG interface:

The JTAG connector is 2x5 pin with 0,1" step and Atmel's compatible layout. The PIN.1 is marked with square pad on bottom and arrow on top. JTAG signals are: 1- TCK, 2- GND, 3- TDO, 4- VREF, 5- TMS, 6- NSRST, 7- VCC, 8- NTRST, 9- TDI, 10- GND.

JTAG TOP view PCB board layout:



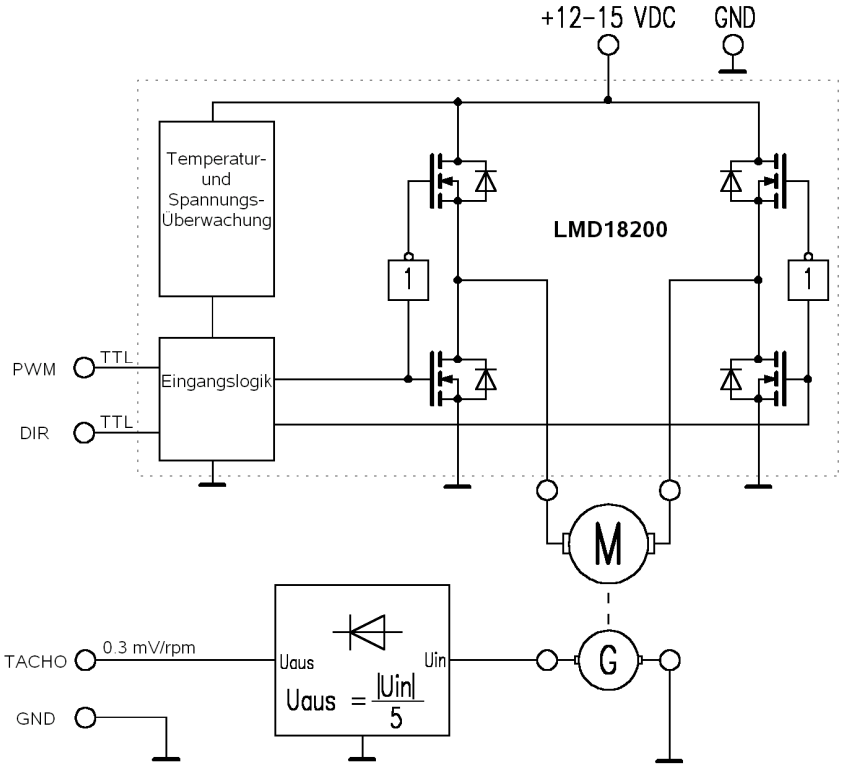
Virtual com port driver:

AVR-USB-JTAG uses FT232 USB to RS232 IC from Future Technology devices, you must download and install proper driver for your computer and OS from:
http://www.ftdichip.com/Drivers/FT232-FT245Drivers.htm

Ordering codes:

AVR-USB-JTAG- assembled and tested

C Schaltung des Gleichstrommotor-Moduls



D Programm-Code Analog-Digital-Wandler (`uosadc.h`)

```
1  #ifndef UOSADC_H
2  #define UOSADC_H
3  /*************************************************************************
4  Analog-Digital-Wandler des ATMega16
5  ReadADC(Kanal) liest einen Wert vom spezifizierten Kanal (0..7)
6  Verwendet die interne Referenzspannung - Ergebnis in mV
7  *************************************************************************/
8
9  #include <avr/io.h>
10
11 uint16_t ReadADC(uint8_t mux)
12 {
13   uint8_t i;
14   uint16_t result = 0;          //Initialisieren wichtig, da lokale Variable
15                                 //sonst zufllige Werte haben.
16   ADCSRA =  _BV(ADEN) | _BV(ADPS2) | _BV(ADPS1); // Frequenzvorteiler auf 64
17                                                  // ADC aktivieren (ADEN=1)
18   ADMUX = mux | _BV(REFS1) | _BV(REFS0); //Kanal waehlen, interne Referenzspannung
19
20   ADCSRA |= _BV(ADSC);          //eine ADC-Wandlung ("Dummy-Readout" zum "Aufwrmen")
21   loop_until_bit_is_clear(ADCSRA,ADSC);   //auf Abschluss der Konvertierung warten
22
23   // Eigentliche Messung - Mittelwert aus 10 aufeinanderfolgenden Wandlungen
24   for(i=0;i<10;i++)
25   {
26     ADCSRA |= _BV(ADSC);        //eine ADC-Wandlung
```

```
27      loop_until_bit_is_clear(ADCSRA,ADSC); //auf Abschluss der Konvertierung warten
28      result += ADCW;             //Beide ADC-Bytes lesen
29    }
30    ADCSRA &= ~_BV(ADEN);         //ADC deaktivieren (ADEN=0)
31
32    return result>>2;             //Ergebnis in mV (Referenzspannung ist 2.56 V)
33  }
34  #endif // UOSADC_H
```

E Programm-Code LCD-Anzeige (`uoslcd.h`)

```
1   #ifndef UOSLCD_H
2   #define UOSLCD_H
3   /************************************************************************
4   Ansteuer-Routinen fr die LCD-Anzeige
5   4bit-Interface ber Port D (Data-Nibble auf oberen 4 Bits)
6   LCD_Init() initialisisert die LCD-Anzeige
7   LCD_Line(Line, String) gibt einen String auf Line (0 oder 1) aus
8   LCD_Clear() lscht die Anzeige
9   ************************************************************************/
10
11  #define XTAL        8       // CPU-Frequenz
12  #define LCDPort     PORTD     // Ausgabeport D
13  #define LCDDDR      DDRD
14  #define PIN_Enable  _BV(0)    // Enable = Bit 0
15  #define PIN_RS      _BV(1)    // RS = Bit 1
16  #define DATAMASK    0xf0      // 4-Bit-Nibble auf DB4-DB7
17
18  #include <avr/io.h>
19  #include "uosdelay.h"
20
21  //erzeugt den Enable-Puls
22  void LCD_Enable()
23  {
24    LCDPort |= PIN_Enable;
25    Delay_us(2);
26    LCDPort &= ~PIN_Enable;
27  }
28
29  // Ausgabe eines 4-Bit_Nibbles
30  static inline void LCD_Nibble(uint8_t N)
31  {
32    LCDPort = N;
33    LCD_Enable();
34  }
35
36  void LCD_Data(uint8_t b)
37  {
38    LCD_Nibble((b&DATAMASK) |  PIN_RS);
39    LCD_Nibble((b<<4) |  PIN_RS);
40    Delay_us(100);
41  }
42
43  void LCD_Command(uint8_t b)
44  {
```

```
45    LCD_Nibble(b&DATAMASK);
46    LCD_Nibble(b<<4);
47    Delay_us(100);
48  }
49
50  static inline void LCD_Clear()
51  {
52    LCD_Command(0b00000001);
53    Delay_ms(3);
54  }
55
56  static inline void LCD_Address(uint8_t addr)
57  {
58    LCD_Command(0x80 | ((addr) & 0x7f));
59  }
60
61  void LCD_Init()
62  {
63    LCDDDR = 0xff;           //Port als Ausgang programmieren
64    Delay_ms(200);
65    LCD_Nibble(0b00110000);  //8bit-Grundmodus einstellen
66    Delay_ms(5);
67    LCD_Enable();            //3 Enable-Pulse !?!?
68    Delay_ms(5);
69    LCD_Enable();
70    Delay_ms(5);
71    LCD_Nibble(0b00100000);  //4bit-Modus einstellen
72    Delay_ms(5);
73    LCD_Command(0b00101000); //2-zeilig
74    LCD_Command(0b00001100); //Display=on Cursor=off blink=off
75    LCD_Clear();
76  }
77
78  void LCD_Line(int line, char * buffer)
79  {
80    if (line==0)
81      LCD_Address(0);
82    else
83      LCD_Address(40);
84    while (*buffer)
85    {
86      LCD_Data(*buffer);
87      buffer++;
88    }
89  }
90  #endif // UOSLCD_H
```

F Programm-Code Verzögerungsschleifen (`uosdelay.h`)

```
1   #ifndef UOSDELAY_H
2   #define UOSDELAY_H
3   /*************************************************************************
4    Verzgerungsroutinen fr Mikrosekunden und Millisekunden
5    Delay_us(__us), Delay_ms(__ms) us bzw. ms Delay (16-bit uint)
6   *************************************************************************/
```

```
 7
 8  #ifndef XTAL
 9  #define XTAL 8
10  #endif
11
12  #include <util/delay.h>
13
14  static inline void Delay_us(uint16_t __us)
15  {
16    _delay_loop_2(XTAL*__us/4);
17  }
18
19  void Delay_ms(uint16_t __ms)
20  {
21    uint16_t i;
22    for(i=0;i<__ms;i++)
23    {
24      Delay_us((uint16_t)1000);
25    }
26  }
27  #endif // UOSDELAY_H
```

G Programm-Code Testprogramm ADC und LCD (`lcdtest.c`)

```
1  /*****************************************************************************
2  Testprogramm fr ADC und LCD-Anzeige
3  *****************************************************************************/
4
5  #include <stdio.h>
6  #include "uoslcd.h"
7  #include "uosadc.h"
8  #include "uosdelay.h"
9
10 int main(void)
11 {
12   uint16_t adc0, adc1;
13   uint8_t k;
14   char buffer[40];
15   LCD_Init();
16   LCD_Line(0,"-- HALLO --");
17   Delay_ms(2500);
18   for(;;)
19     for (k=0;k<2;k++)
20     {
21       adc0 = ReadADC(2*k);
22       adc1 = ReadADC(2*k+1);
23       sprintf(buffer,"%i:%4i %i:%4i",2*k,adc0,2*k+1,adc1);
24       LCD_Line(k,buffer);      //nach Zeile k
25       Delay_ms(250);
26     }
27 }
```