AVR-Tutorial

Aus dem Microcontroller.net

Was ist ein Mikrocontroller?

Ein Mikrocontroller ist einem Prozessor ähnlich. Der Unterschied zu PC-Prozessoren besteht darin, dass bei einem Mikrocontroller Speicher, Digital- und Analog-Ein- und -Ausgänge etc. meist auf einem einzigen Chip integriert sind, so dass eine Mikrocontroller-Anwendung oft mit wenigen Bauteilen auskommt.

Mikrocontroller werden als erstes an der Bit-Zahl des internen Datenbusses unterschieden: 4bit, 8bit, 16bit und 32bit. Diese Bit-Zahl kann man als die Länge der Daten interpretieren, die der Controller in einem Befehl verarbeiten kann. Die größte in 8 Bit (= 1 Byte) darstellbare Zahl ist die 255, somit kann ein 8-Bit-Mikrocontroller z. B. in einem Additionsbefehl immer nur Zahlen kleiner-gleich 255 verarbeiten. Zur Bearbeitung von größeren Zahlen werden dann jeweils mehrere Befehle hintereinander benötigt, was natürlich länger dauert. Ein Mikrocontroller braucht zum Betrieb, wie jeder andere Prozessor auch, einen Takt. Die maximale Taktfrequenz mit der ein Controller betrieben werden kann, reicht von 1 MHz bei alten Controllern bis hin zu über 100 MHz bei teuren 32-Bittern. Diese Taktfrequenz sagt jedoch noch nichts über die tatsächliche Geschwindigkeit eines Prozessors aus. So wird z. B. bei den meisten 8051-Controllern die Frequenz intern durch 12 geteilt, ein mit 24 MHz getakteter 8051 arbeitet also eigentlich nur mit 2 MHz. Benötigt dieser dann für einen Befehl durchschnittlich 2 Taktzyklen, so bleiben "nur" noch 1 Mio. Befehle pro Sekunde übrig - ein AVR, der ungeteilt mit 8MHz arbeitet und für die meisten Befehle nur einen Zyklus braucht, schafft dagegen fast 8 Mio. Befehle pro Sekunde.

Wozu ist ein Mikrocontroller gut?

Hier ein paar Beispiele, für welche Aufgaben Mikrocontroller verwendet werden (können):

- Ladegeräte, komplexere Stromversorgungen, Umrichter
- Motorsteuerungen, CNC-Maschinen
- Roboter
- Messwerterfassung (z. B. Drehzahlmessung im Auto), Datenlogger
- Temperaturregler, Drehzahlregler
- MP3-Player, Uhren, Armbanduhren (etwa mit ausgefallenen Extras)
- Schaltuhren, SPS-Ersatz
- Alarmanlagen
- LED-Matrizen (Blinkmuster etc.)
- Zur Steuerung und Regulierung von Flüssigkeiten
- „Konsumgüter“-Steuerungen, etwa im Eigenbau-Radio oder Amateurfunkgerät, Fernbedienungen, PC-Zubehör
- ...

Welchen Mikrocontroller soll ich verwenden?

Typische Anforderungen an einen Mikrocontroller für Hobbyanwender (einige davon konkurrieren miteinander):

- Gute Beschaffbarkeit und geringer Preis
- Handliche Bauform: Ein Controller mit 20 Pins ist leichter zu handhaben als einer mit 128
- Flash-ROM: Der Controller sollte mindestens 1000 mal neu programmiert werden können
- In-System-Programmierbarkeit (ISP): Man benötigt kein teures Programmiergerät und muss den Controller zur Programmierung nicht aus der Schaltung entfernen
- Kostenlose Software verfügbar: Assembler bekommt man praktisch immer kostenlos

Weitere Entscheidungskriterien sind im Artikel Entscheidung Mikrocontroller zusammengefasst.

Viele dieser Anforderungen werden von den 8-bit-AVR-Controllern von Atmel erfüllt. Deshalb werde ich einen AVR, genauer gesagt den ATmega8, in diesem Tutorial einsetzen.

Und damit kein Missverständnis aufkommt: So etwas wie den "besten" Controller gibt es nicht. Es hängt immer von der Aufgabenstellung ab, welcher Controller **gut** dafür geeignet ist. Natürlich haben sich einige Controller als Standardtypen in der Praxis durchgesetzt, mit denen man in vielen Fällen ein gutes Auslangen hat und die mit ihrer Leistungsfähigkeit einen weiten Bereich abdecken können. Der ATmega8 ist z. B. so einer. Aber daneben gibt es noch viele andere.

In welcher Sprache soll programmiert werden?

Je nach Anforderungsfall bieten sich verschiedene Sprachen an:

Vorbemerkung

Warum ist dieses Tutorial für Assembler geschrieben, wo es doch einen kostenlosen C-Compiler (WinAVR, AVR-GCC) und einen billigen Basic-Compiler gibt?

Assembler ist für den Einstieg "von der Pike auf" am besten geeignet. Nur wenn man Assembler anwendet, lernt man den Aufbau eines Mikrocontrollers richtig kennen und kann ihn dadurch besser nutzen; außerdem stößt man bei jedem Compiler irgendwann mal auf Probleme, die sich nur oder besser durch das Verwenden von Assemblercode lösen lassen und sei es nur, dass man das vom Compiler generierte Assemblerlisting studiert, um zu entscheiden, ob und wie man eine bestimmte Sequenz im C-Code umschreiben soll, um dem Compiler das Optimieren zu ermöglichen/erleichtern.

Allerdings muss auch erwähnt werden, dass das Programmieren in Assembler besonders fehleranfällig ist und dass es damit besonders lange dauert, bis das Programm erste Ergebnisse liefert. Genau aus diesem Grund wurden "höhere" Programmiersprachen erfunden, weil man damit nicht immer wieder "das Rad neu erfinden" muss. Das gilt besonders, wenn vorbereitete Programmblöcke zur Verfügung stehen, die man miteinander kombinieren kann. Auch der Geschwindigkeitsvorteil ist selten und nur bei kritischen Anwendungen von Bedeutung. Heutige Compiler generieren zudem oft schnelleren oder kleineren Code als handgeschriebene Assemblerroutinen. Wer regelmäßig programmieren und auch längere Programme schreiben

möchte, dem sei deshalb geraten, nach diesem Assembler-Tutorial C zu lernen, zum Beispiel mit dem AVR-GCC-Tutorial.

Wer C schon kann, für den bietet es sich an, das Tutorial parallel in C und Assembler abzuarbeiten. Die meisten hier vorgestellten Assemblerprogramme lassen sich relativ einfach in C umsetzen. Dabei sollte großes Augenmerk darauf gelegt werden, dass die dem Programm zugrunde liegende Idee verstanden wurde. Nur so ist ein vernünftiges Umsetzen von Assembler nach C (oder umgekehrt) möglich. Völlig verkehrt wäre es, nach sich entsprechenden 'Befehlen' zu suchen und zu glauben, damit hätte man dann ein Programm von Assembler nach C übersetzt.

Assembler

Die Vorteile von Assembler wurden bereits genannt:

- direkter Einstieg in die Hardware
- keine Abhänhigkeit von Compilern und deren Fehlern, bzw. Misinterpretation, keine Seiteneffekte
- optimaler Code erzeugbar
- sehr schnell in der Ausführung
- Feintuning der Geschwindigkeitsreserven
- kurzer Weg zu hardwarenahen Funktionen
- sehr langer Weg zu komplexeren Funktionen

Basic

Basic bietet den einfachsten Einstieg, wenn man bereits eine höhere Programmiersprache beherrscht und wenig Kenntnisse über die Hardware hat und sich zudem auf komplexere Steuerungen ohne optimale Ausschöpfung der HW-Resourcen beschränkt.

- direkter Einstieg in komplizierte Abläufe
- einfacher Einstieg in die Programmiersprache
- Abhängigkeit von Compilern und deren Fehlern
- Code ist schnell erzeugbar
- sehr langsam in der Ausführung
- kurzer Weg zu komplexeren Funktionen
- keine hardwarenahen Funktionen verfügbar

Fast immer beziehen sich BASIC-Programmierer auf BASCOM, eine kostenpflichtige Softwarelösung zum einfachen Erstellen von Mikrocontroller-Programmen. Während damit gewisse Standardaufgaben in null-komma-nix erschlagen sind, muss man für alles andere die Materie genauso durchdringen wie der Assembler- oder C-Programmierer. Im Gegensatz zum Assembler-Programmierer muss man mit Seiteneffekten rechnen.

C / C++

C bietet den optimalen Kompromiss, da man durch Funktionen und Prozeduren sehr leicht hochsprachliche Strukturen und Datentypen nutzen kann und dennoch sehr effektiven Code

produzieren (lassen) kann. Allerdings ist C strukturell am schwierigsten zu verstehen.

- schwieriger Einstieg in die Programmiersprache
- Abhängigkeit von Compilern und deren Fehlern, allerdings verifizierbar
- Code ist automatisch erzeugbar, manuell aber kompliziert
- sehr schnell in der Ausführung
- akzeptabler Weg zu komplexeren Funktionen
- hardwarenahe Funktionen verfügbar
- mit Assembler kombinierbar (zumeist mit Inline-Assembler, ist jedoch nicht C-standardkonform)

Generell kann man zu C++ raten, um dessen Vorteile zu nutzen, aber dann sollte man keinesfalls alle Fähigkeiten von C++ ausreizen!

Nutzbar sind:

- Variablendeklaration an beliebigen Stellen
- bool-Datentyp
- Initialisierung von Strukturen mit Nicht-Konstanten
- Klassen mit Einfachvererbung
- Überladene Funktionen
- Einfache Templates
- Statische Konstruktoren

Nicht unbedingt nutzen sollte man (oder man muss nachsehen, wie blöd sich der Compiler anstellt):

- STL (Standard Template Library)
- Mehrfachvererbung
- Virtuelle Methoden (muss man fallweise entscheiden, zumindest bei avr-gcc umständlich)
- Allzu verrückte oder neue Features wie
    - Verschachtelte Klassen
    - Automatische Typerkennung (auto-Schlüsselwort)
    - Laufzeit-Typprüfung
    - Anonyme Funktionen (Lambda-Ausdrücke)
- new- und delete-Operator (Speicherverwaltung allgemein)

Anfängern ist *nicht* zu C++ zu raten, weil die Fehlermeldungen (bei Tippfehlern) wesentlich unverständlicher sind.

Deutsch oder englisch?

Die Entscheidung über die Sprache der **Kommentierung** und **Ausgabetexte** ist je nach Problem und Audienz zu fällen. In der Regel kommentiert man deutsch, damit sich Franzosen über die Unlesbarkeit beschweren, oder englisch, damit man sich selbst schwer tut und Briten sich über das unverständliche Englisch beschweren:-)

Gerade als Anfänger macht man den Fehler, in den Kommentar hineinzuschreiben, *was* eine

Standardfunktion tut (das steht in der Dokumentation und nach 10 Aufrufen im Kopf), aber nicht, *warum* man diese aufgerufen hat. Bei eigenen Funktionen sollten stets alle Parameter in ihrer Wirkung erläutert werden. Zu nichts nütze ist ein Kommentar wie „*bool silentmode* switches the function to silent mode“.

Nichtsdestotrotz muss man sich mit dem verwendeten **Zeichensatz** auseinander setzen. Denn früher oder später will man Umlaute, „°C“ oder „kΩ“ ausgeben, und hier sind alle klassischen 1-Byte-Zeichensätze am Ende. Am zukunftssichersten benutzt man die Mehrbyte-Kodierung UTF-8 und implementiert geeignete Transkodierungsroutinen bspw. für HD44780-kompatible LC-Anzeigen. Die Beschränkung auf BMP (< 65000 Zeichen) genügt für Mikrocontroller-Projekte, sodass ein UTF-8-Zeichen maximal 3 Bytes lang ist.

Mit Betriebssystem oder ohne?

Betriebssysteme erfreuen sich auch auf Mikrocontrollern immer größerer Beliebtheit. Multitasking- und Echtzeitanwendungen lassen sich so manchmal viel einfacher implementieren, da standardisierte Schnittstellen und Zugriffsmethoden verwendet werden und die zur Verfügung stehenden Bibliotheken und Software-Pakete genutzt werden können. Es ist jedoch stets abzuwägen, ob der Mehraufwand der Einarbeitung in ein Multitasking-Betriebssystem und der damit in Verbindung stehende Programmieraufwand mit dem potenziellen Ersparnis an Denk- und Planungszeit, die zur Realisierung der benötigten Funktionen „zu Fuß“ benötigt würde, im Verhältnis steht. Oftmals wird in der Industrie nach wie vor aus Kostengründen auf ein Betriebssystem verzichtet, weil es das Risiko eines nicht zu vertretenden Mehraufwands birgt und die Ausnutzung der Rechenleistung — gerade kleiner Controller — stark herabsetzt, was wiederum die Echtzeitfähigkeit verschlechtert.

Mit Betriebssystem

- Einarbeitung kostet Zeit und Geld
- Echtzeitfunktion im Millisekundenbereich einfacher möglich, da das Multitasking die parallele Reaktion des Systems auf äußere Einflüsse erleichtert
- Multitaskingfunktion ist vorimplementiert - muss nur genutzt werden
- Implementierung des Multitaskings kostet weniger Zeit
- Die Multitasking-relevanten Teile des Codes sind betriebssystem-spezifisch und nicht übertragbar
- Der gesamte Code ist weniger gut auf andere Controller portierbar (wieso?)
- Der Controller ist pauschal mit viel nicht nutzbarem Overhead belastet
- Es muss tendenziell ein teuerer Controller eingesetzt werden

Zu einem Betriebssystem greift man gern wenn TCP/IP zu implementieren ist. Das ist typischerweise ein Controller mit Ethernet-Anschluss. Oder der Zugriff auf ein Dateisystem, etwa USB-Sticks. Das wiederum erfordert einen USB-Hostcontroller. (Die meisten Mikrocontroller implementieren nur den einfacheren USB-Devicecontroller.) Auch für den Zugriff auf eine SD-Karte bietet sich ein Betriebssystem an; will man jedoch nur FAT16/FAT32 *lesen*, gibt es einfachere Lösungen. Auf den Punkt gebracht: Betriebssysteme nimmt man, wenn man hochgradig

softwarelastige Standardaufgaben lösen will.

Linux eignet sich praktisch nur bei Controllern mit externem RAM und Flash und eignet sich daher kaum für selbst gebastelte Mikrocontroller-Boards. Für kleinere Controller gibt es angepasste Betriebssysteme, etwa NutOS. Als grobe Schranke gilt: Unter 64 KByte Flash braucht man über ein Betriebssystem nicht nachzudenken.

Hieran sieht man auch, warum sich Toaster und Kühlschränke mit Internetanschluss nicht durchsetzen: Der hierfür benötigte Controller ist einfach viel größer und energiehungriger als wenn nur eine Temperatur zu regeln bzw. der Bräunungsgrad der Toastscheibe zu messen ist!

Will man neben dem Betriebssystem Echtzeitanforderungen im Mikrosekundenbereich realisieren benötigt man ein tiefes Verständnis in das Betriebssystem sowie den Quellkode, um an den richtigen Stellen eingreifen zu können, etwa um Interruptsperrzeiten im Task-Switcher zu minimieren. Im Allgemeinen verzichtet man auf solche Extras und erledigt solche Aufgaben in einem separaten Mikrocontroller ohne Betriebssystem.

Das Betriebssystem legt die Wahl der Programmiersprache fest. Dies ist ausnahmslos C oder C++.

Ohne Betriebssystem

- Kein zusätzlicher Einarbeitungsaufwand
- Multitasking muss selbst implementiert werden (ist nicht so schwer, aber man benötigt etwas Übung)
- Implementierung des Multitaskings kostet mehr Zeit
- die Multitasking-relevanten Teile des Codes sind voll übertragbar
- Weniger Overhead, da nur benötigte Funktionen eingebaut werden
- Echtzeitfunktion ebenfalls möglich, insbesondere im Mikrosekundenbereich, wenn auch schwieriger zu implementieren
- Es kann tendenziell ein preiswerterer Controller eingesetzt werden

Die meisten Projekte werden ohne Betriebssystem erstellt. Auch für Anfänger im Bereich der Mikrocontroller ist dies der bessere Weg. Denn je kleiner der verwendete Controller ist, desto weniger kann er, und desto besser gelingen bspw. Echtzeitfunktionen im Mikrosekunden-Bereich.

Weitere Informationen

Weiterführende Informationen u. A. zu den berüchtigten Fuse-Bits, zu Programmier-Hard- und Software, dem AVR Softwarepool und einer Checkliste mit Hinweisen zur Lösung üblicher Probleme finden sich im Hauptartikel AVR.

AVR-Tutorial - Benötigte Ausrüstung

Hardware

Ein Mikrocontroller alleine ist noch zu nichts nütze. Damit man etwas damit anfangen kann braucht man eine Schaltung in die der Controller eingesetzt wird. Dazu werden bei Elektronikhändlern Platinen angeboten, die alles nötige (Taster, LEDs, Steckverbinder...) enthalten. Häufig enthalten diese Platinen nicht nur Platz für den Mikroprozessor, sondern auch einen ISP-Programmierer (Näheres dazu später)

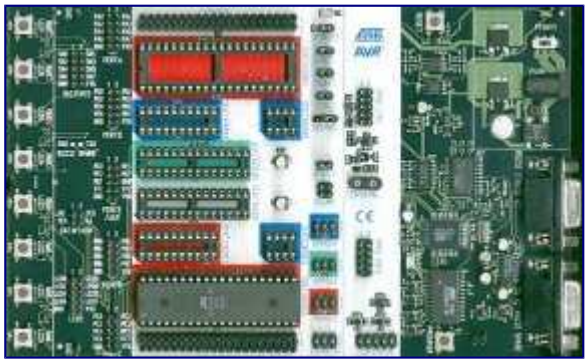
Da es eine Unmenge verschiedener Mikrocontroller allein aus der AVR-Familie gibt, ist der Anfänger mit der Auswahl bereits überfordert. Dieses Tutorial (und viele andere auch) beginnt mit dem *mittelgroßen* Controller **ATmega8**. Die 8 steht für 8 Kilobyte Flash-Speicher. Er gilt als ideal für Einsteiger, und auch von „alten Hasen“ wird dieser Typ gern verwendet. Es gibt ihn im 28-poligen DIL-Gehäuse, ideal für Bastelschaltungen auf dem Steckbrett, als auch in verschiedenen SMD-Gehäusen wenn das Endgerät klein werden soll.

Fertige Evaluations-Boards und Starterkits

AVR Starterkit aus dem Mikrocontroller.net-Shop

Sehr gut für dieses Tutorial geeignet ist das Olimex AVR-P28-Board, welches zusätzlich noch den Controller ATmega8, einen USB-ISP-Programmieradapter und ein Steckernetzteil benötigt. Die Platine enthält eine Fassung für den Controller, einen Spannungswandler, die Beschaltung für die serielle Schnittstelle und einen Anschluss für den Programmieradapter. Die restliche Hardware wie LEDs und Taster kann man sich selber nach Belieben auf das Lochrasterfeld löten.

STK500



Das STK500 ist das Standard-Board für AVR Entwicklung, direkt von Atmel. Es enthält auch einen ISP-Programmer und ist fertig aufgebaut. Es ist unter Entwicklern sehr beliebt und wird natürlich von Atmel unterstützt. Es gilt allgemein als gute Investition wenn man ernsthaft in das Thema einsteigen möchte.

Das STK500 kostet bei Reichelt ca. 115 Euro (Stand November 2015). Ein geeignetes Netzteil muss zusätzlich erworben werden.

Pollin ATMEL Evaluations-Board Version 2.x

Bei Pollin Elektronik gibt es für ca. 15 Euro ein Evaluations-Board als Bausatz zum Selbstlöten. Im Bausatz sind die Aufbauanleitung, die Platine und Bauteile enthalten. Der/die Mikrocontroller und eine Stromversorgung müssen separat beschafft werden. Auf dem Board ist ein einfacher ISP-Programmer (serielles *bit-banging*) integriert.

Siehe:

- Pollin ATMEL Evaluations-Board
- http://www.pollin.de

Pollin Funk-AVR-Evaluationsboard v1.x

Bei diesem Board besteht die Möglichkeit, Funkmodule wie das RFM12, RFM01 oder RFM02 auf dem Board aufzulöten.

Siehe:

- Pollin Funk-AVR-Evaluationsboard
- http://www.pollin.de

Rumpus Board von lochraster.org

Lochraster.org bietet ein Entwicklungsboard namens *rumpus* an. Es kommt als Bausatz mit allen Teilen und Microcontroller (Atmega 168). Auf dem Microcontroller ist bereits ein USB Bootloader installiert, so dass man nach dem Zusammenbau sofort starten kann. Das Board wird direkt über USB mit Strom versorgt und programmiert. Es kann auch selbst als Programmer für AVR Microcontroller benutzt werden. Das Board ist mit recht umfangreicher Peripherie ausgestattet, so dass sich von sehr einfachen Anwendungen wie dem Blinken einer LED bis hin zu komplexen Aufgaben wie Senden und Empfangen von Infrarot Signalen eine Vielzahl von Anwendungen realisieren lassen. Mit 45 Euro gehört es sicher nicht zu den ganz billigen Einsteigerboards, für den ambitionierten Amateur bietet die reichhaltige Peripherie den Vorteil, das Board während des gesamten Lernprozesses zu nutzen ohne für die Realisierung komplexerer Aufgaben neue Hardware auflöten zu müssen. Auch relativiert sich dieser Preis wieder dadurch, dass kein ISP Programmer benötigt wird. Beim Umstieg auf ein anderes Board, für welches man dann einen ISP Programmer benötigt, kann der Rumpus diese Aufgabe übernehmen anstatt zum alten Eisen geworfen zu werden (s. Infos im Forumbeitrag von Sebastian Noack).

Weitere Infos unter http://www.lochraster.org/ und http://wiki.lochraster.org/

RN-Control

Die Forengemeinde von Roboternetz hat ebenfalls ein Evaluierungsboard entwickelt das mittlerweile sehr ausgereift ist und viele Erweiterungsmöglichkeiten bietet.

Siehe:

- http://robotikhardware.de/
- http://www.roboternetz.de/

Arduino

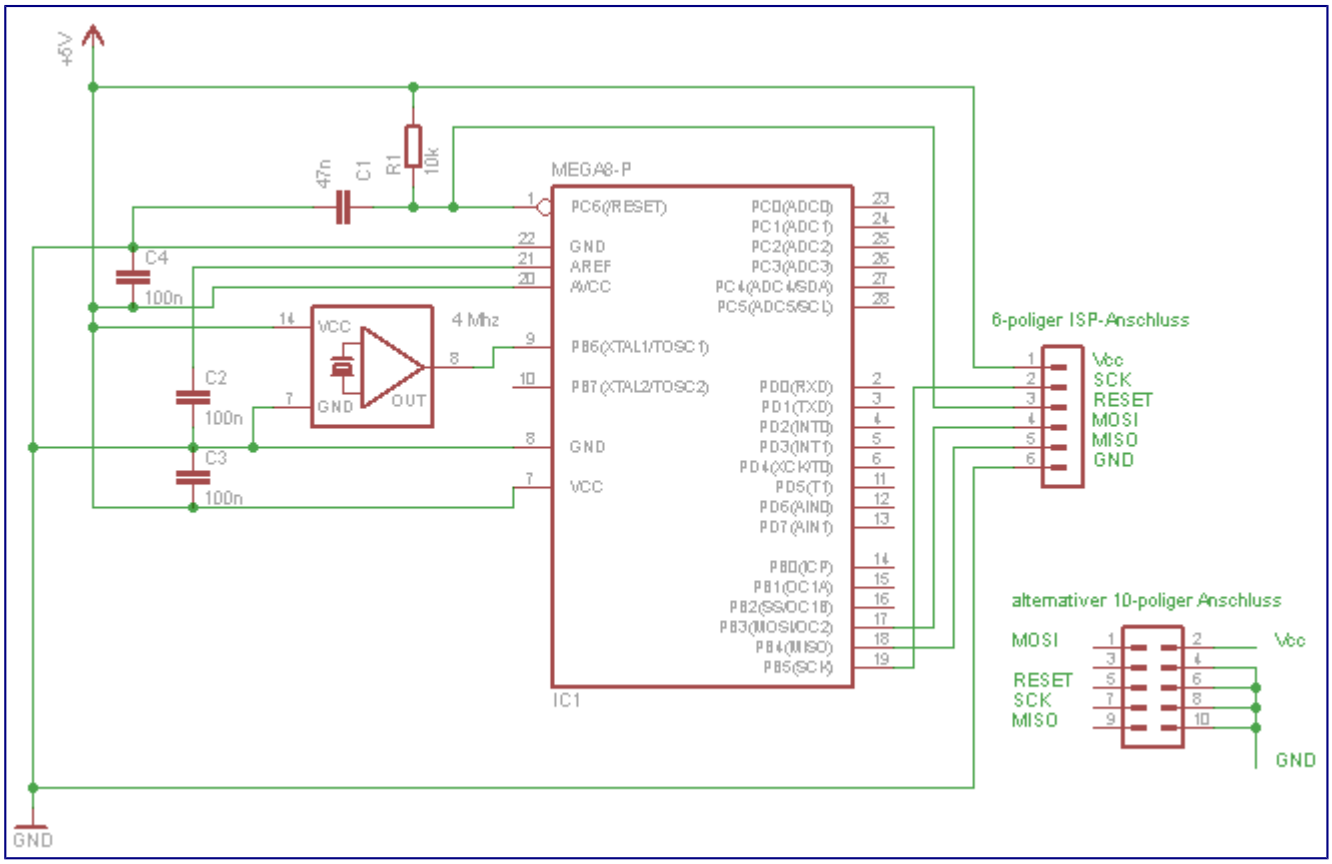
Die Boards der Arduino-Familie bieten z.B. einen ATmega328P mit 16MHz und lassen sich über einen integrierten USB-seriell-Wandler und Bootloader programmieren. Die Ports sind auf Buchsenleisten herausgeführt. Arduino-Boards können auch unabhängig von der Arduino-Entwicklungsumgebung (Arduino-IDE) als AVR-Entwicklungsboard genutzt werden.

Andere

Das Angebot an AVR-Evaluationboards, -Experimentierplatinen, -Entwicklerplatinen oder wie die jeweiligen Hersteller ihre Produkte auch immer bezeichnen, ist mittlerweile recht groß geworden. Sie alle zu bewerten ist unmöglich geworden.

Selbstbau

Ein fertiges Board ist gar nicht nötig, man kann die benötigte Schaltung auch selbst auf einem kleinen Steckbrett oder einer Lochrasterplatine aufbauen. So kompliziert wie das STK500 wird es nicht, es reichen eine Handvoll Bauteile. Wie man das macht wird im Folgenden beschrieben. Steckbretter (Breadboards) gibt's z. B. bei Reichelt, ConeleK, ELV oder Conrad.



Die Grundschaltung eines ATmega8/ATmega8A.

ACHTUNG: Die Pinbelegung der 6-poligen ISP-Verbindung weicht von den ATMEL Angaben ab! Wenn ATMEL oder ATMEL-kompatible ISP-Adapter benutzt werden, die Pinbelegung aus AVR042 (PDF) bzw. AVR_In_System_Programmer#Pinbelegung benutzen

Über den Takteingang **XTAL1** ist der Mikrocontroller mit dem **Quarzoszillator** verbunden, der den

benötigten Takt von 4 MHz liefert (siehe unten). Achtung: die Pins werden, wenn man den Oszillator mit der Schrift nach oben vor sich liegen hat, von unten links aus abgezählt. Unten links ist Pin 1, unten rechts Pin 7, oben rechts Pin 8 und oben links Pin 14 (natürlich hat der Oszillator nur 4 Pins. Die Nummerierung kommt daher, dass bei einem normalen IC dieser Größe an den gleichen Positionen die Pins Nr. 1, 7, 8 und 14 wären). Zu den Pins Datenblatt beachten [1].

PD0-PD7 und **PB0-PB5** sind die **IO-Ports** des Mikrocontrollers. Hier können Bauteile wie LEDs, Taster oder LCDs angeschlossen werden. Der **Port C (PC0-PC5)** spielt beim Atmega8 eine Sonderrolle: mit diesem Port können Analog-Spannungen gemessen werden. Aber dazu später mehr! An **Pin 17-19** ist die Stiftleiste zur Verbindung mit dem ISP-Programmer angeschlossen, über den der AVR vom PC programmiert wird (Achtung: Pins in Abbildung entsprechen nicht der Belegung des AVRISP mkII. Die korrekte Pin-Belegung kann im Handbuch des AVRISP mkII eingesehen werden). Die Resetschaltung, bestehend aus **R1** und **C1**, sorgt dafür, dass der Reseteingang des Controllers standardmäßig auf Vcc=+5V liegt. Zum Programmieren zieht der ISP-Adapter die Resetleitung auf Masse (GND), die Programmausführung wird dadurch unterbrochen und der interne Speicher des Controllers kann neu programmiert werden. Zwischen Vcc und GND kommen noch jeweils ein 100nF Keramik- oder Folienkondensator C3 und C4, um Störungen in der Versorgungsspannung zu unterdrücken. Diese Abblockkondensatoren sollten so nah wie möglich am Controller platziert werden. An den Ausgang ARef wird ebenfalls ein 100nF Kondensator angeschlossen. Dieser wird allerdings erst benötigt, wenn der Analog/Digital Konverter des µC in Betrieb genommen wird.

Für den Anschluss des ISP-Programmiergerätes kann man im Grunde jede beliebige Pin-Belegung des ISP Steckers benutzen, solange nur alle benötigten Leitungen mit dem Programmiergerät verbunden sind. In der Praxis haben sich allerdings bestimmte Belegungen durchgesetzt. Im Schaltbild ist eine **eigene** Belegung des 6-poligen Steckers gezeigt. Die alternative Pinbelegung eines 2-reihigen/10-poligen Steckers ist eine übliche Belegung. Benutzt man so eine übliche Belegung, so reicht normalerweise ein 10-poliges Flachbandkabel, um den vorhandenen ISP-Programmer so mit der Schaltung zu verbinden, dass alle Signale am richtigen Prozessorpin ankommen. Siehe auch AVR_In_System_Programmer.

Hier die Liste der benötigten Bauteile:

- R1 Widerstand 10 kOhm
- C1 Keramikkondensator 47 nF
- C2, C3, C4 Keramik- oder Folienkondensator 100 nF
- Stiftleiste 6-polig
- Mikrocontroller ATmega8 oder ATmega8A (kann u.a. auf http://shop.mikrocontroller.net/ bestellt werden)
- Quarzoszillator 4 MHz

Beim Steckbrett ist darauf zu achten, dass man die parallellaufenden Schienen für GND (blau) und Vcc (rot) jeweils mit Drähten verbindet (nicht Vcc und GND miteinander!).

Eine Zusammenstellung der benötigten Bauteile befindet sich in der Bestellliste.

Eine weitere Beschreibung für ein Minimalsystem gibt es hier.

Ergänzende Hinweise zur Taktversorgung (kann übersprungen werden)

Ein Mikrocontroller benötigt, wie jeder Computer, eine Taktversorgung. Der Takt ist notwendig, um die internen Abläufe im Prozessor in einer zeitlich geordneten Reihenfolge ausführen zu können. Die Frequenz des Taktes bestimmt im Wesentlichen, wie schnell ein Mikrocontroller arbeitet. Bei einem ATMega8 gibt es viele Möglichkeiten zur Taktversorgung, die Wichtigsten sollen hier gezeigt werden.

- interner RC-Oszillator, das ist der Auslieferungszustand
- Keramikresonator
- Quarz
- Quarzoszillator

Vergleich der AVR-Taktquellen

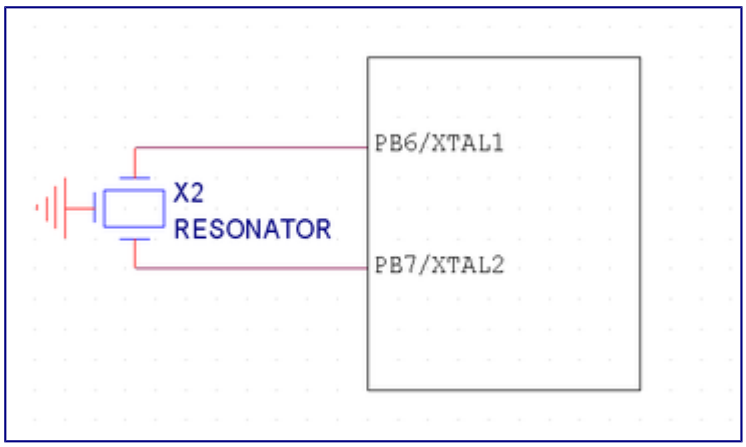
Typ	Genauigkeit	Vorteile	Nachteile
interner RC-Oszillator	1-5%	Xtal1/2 Pins verfügbar kostenlos kein Platzbedarf schnellstes Einschwingen (wenige Takte)	ungenau Betrieb der UART riskant
Keramikresonator	0,5-1%	ausreichend genau für UART in sehr hohen Stückzahlen billiger als Quarz schnelleres Einschwingen als Quarz (ca. 1ms)	XTAL1/2 Pins nicht verfügbar Platzbedarf
Quarz	10-100ppm	sehr genau temperaturstabil	XTAL1/2 Pins nicht verfügbar Platzbedarf Kosten bei sehr hohen Stückzahlen (1000++) langsames Anschwingen (ca. 10ms) 2 zusätzliche Ballast-Kondensatoren typ. 22 pF
Quarzoszillator	1-100ppm	hochgenau sehr temperaturstabil liefert selbst ein Signal, kann dadurch verfuste AVRs retten kann mehrere Takteingänge treiben	XTAL1 Pin nicht verfügbar Platzbedarf Kosten bei sehr hohen Stückzahlen (1000++) langsames Anschwingen (ca. 10ms)

1ppm = 0,0001% (engl. one **p**art **p**er **m**illion, der millionste Teil)

Achtung: Ein ATMega8 wird mit aktiviertem internen RC-Oszillator ausgeliefert. Um eine andere Taktquelle zu aktivieren, müssen die AVR Fuses des Prozessors verändert werden. Das muss jedoch sehr vorsichtig gemacht werden, siehe Artikel.

Keramikresonator

Die Anbindung des Keramikresonators sieht so aus:

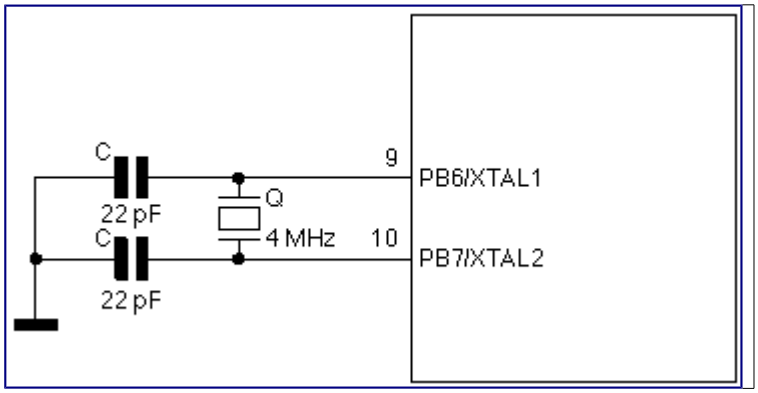


Resonator Standardbeschaltung

Es werden keine Kondensatoren benötigt, diese sind schon eingebaut, daher ist der Anschluss eines Keramikschwingers kinderleicht. Achtung: Keramikresonatoren gibt es mit zwei oder drei Pins. Nur die mit drei Pins besitzen interne Kondensatoren.

Quarz

Die Anbindung des Quarzes sieht so aus:



Quarz Standardbeschaltung

Die beiden Kondensatoren **C3** und **C4** sind zum Betrieb des Quarzes notwendig. Ihre Größe ist abhängig von den Daten des Quarzes. Zur Berechnung ihrer Größe gibt es die folgende Formel:

C=2xCL-(CP+CI)

- CP: Leiterbahnen bedingte Kapazität
- CI: Portbedingte Kapazität
- CL: Datenblatt des Quarzes

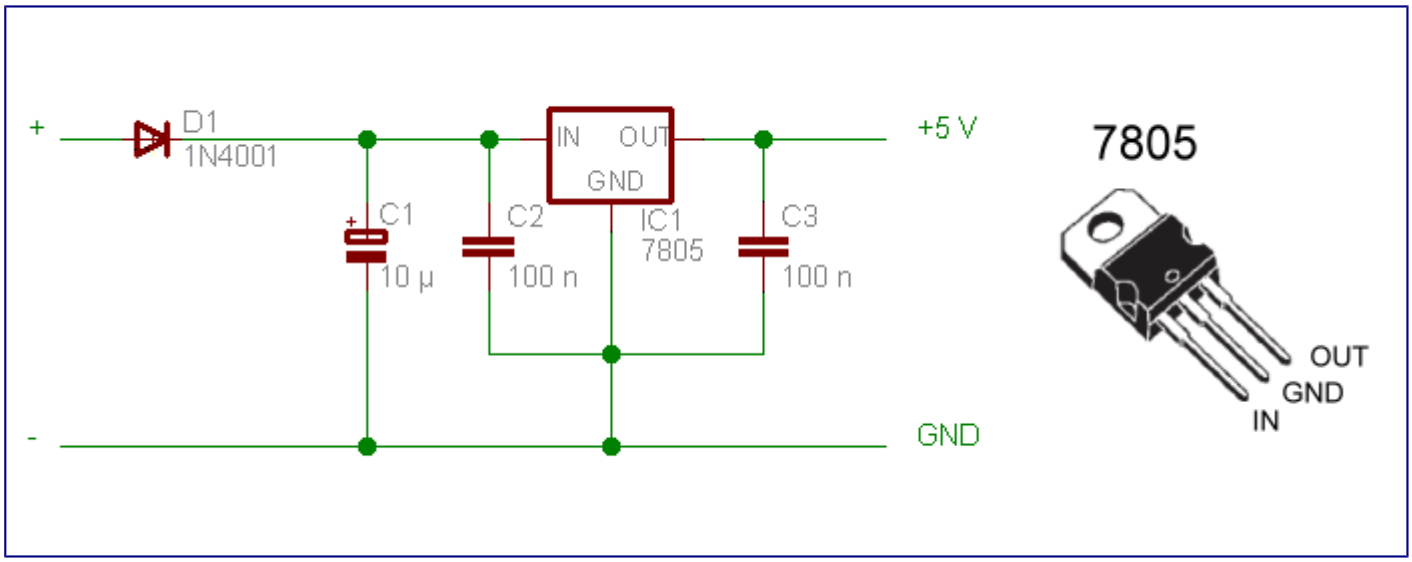
CP+CI ca. 5pF ?

Am Beispiel von CL = 32pF:

C = 2x32pF-5pF = 59pF

Spannungsversorgung

Die Versorgungsspannung $\mathbf{U_{CC}}$ beträgt 5V und kann z. B. mit der in diesem Kapitel beschriebenen Schaltung erzeugt werden. Falls zum Programmieren des Mikrocontrollers ein ISP-Programmiergerät verwendet wird, das an die USB-Schnittstelle angeschlossen ist, kann man die Schaltung auch darüber mit Strom versorgen und dieses Kapitel überspringen.



Standard-Netzteilbeschaltung eines 7805

Bauteile:

- IC1: 5V-Spannungsregler 7805
- C1: Elko 10µF (Polung beachten!)
- C2,C3: 2x Kondensator 100nF (kein Elektrolyt)
- D1: Diode 1N4001

Hauptelement der Schaltung ist das IC 7805. Seine Aufgabe ist es aus der Versorgungsspannung stabile 5 V zu erzeugen. Dieses IC gibt es seit vielen Jahren und wird von vielen Chipherstellern produziert. Er stellt die einfachste Möglichkeit dar, aus einer vorhandenen Gleichspannung definierte 5 V zu erzeugen. Den 7805 gibt es in verschiedenen Ausführungen, was seine maximale Strombelastung angeht. Für die Zwecke dieses Tutorials ist die Standard-Variante, welche maximal 1 A abgeben kann, völlig ausreichend. Der 7805 enthält eine Übertemperatursicherung, so dass er abschaltet, wenn es ihm zu heiß wird. Auch ansonsten ist dieser Spannungsregler-IC kaum kaputt zu bekommen.

Die beiden 100-nF-Kondensatoren haben die Aufgabe, eine mögliche Schwingneigung des 7805 zu unterdrücken. Sie müssen so nahe wie möglich an den Anschlusspins des 7805 angeschlossen werden, um ihre Wirkung zu entfalten.

An den Eingang (+ und – im Schaltplan) wird ein Steckernetzteil mit einer Spannung von 7 .. 12 V angeschlossen. Der 7805 benötigt an seinem Eingang eine Gleichspannung, die mindestens 7 V beträgt. Auf der anderen Seite ergibt es auch keinen Sinn, wesentlich über 12 V Eingangsspannung hinauszugehen. Der 7805 ist ein Linearregler. Salopp gesagt, wird die überschüssige Spannung in Form von Wärme vernichtet. Deshalb die Kühlfahne zum thermischen Anschluss an eine

Wärmesenke. Liegt die Eingangsspannung weit über 12 V, so wird schon wesentlich mehr Energie in Form von Wärme umgesetzt, als am Ausgang entnommen werden kann. Mal ganz davon abgesehen, dass der 7805 davon brennend heiß wird. Je nach Strombedarf der Gesamtschaltung kann es zweckmäßig sein, einen Schaltregler vorzusehen, wenn die Eingangsspannung stark schwanken kann, etwa für die alternative Speisung mit 24 V aus einem Schaltschrank oder dem LKW-Bordnetz. Für Anfänger führt das zu weit, und ein Kühlblech am 7805 tut es dann auch.

Hier ein paar kleine Rechenbeispiele: 12 V Eingangsspannung – 5 V Ausgangsspannung = 7 V Differenz x 0,1 A Strombedarf der Schaltung ergibt die Verlustwärme die abgeführt werden muss.

7 V × 0,1 A = 0,7 Watt

Wenn man jetzt eine Eingangsspannung von 7 V nimmt, so dass die Mindestdifferenz von 2V noch eingehalten wird kommen wir zu diesen Werten

2 V × 0,1 A = 0,2 Watt Abwärme 2 V × 0,35 A = 0,7 Watt Abwärme oder anders gesagt, können wir der Schaltung 350 mA entnehmen und haben die gleiche Abwärme wie im oberen Beispiel mit nur 100 mA Stromentnahme.

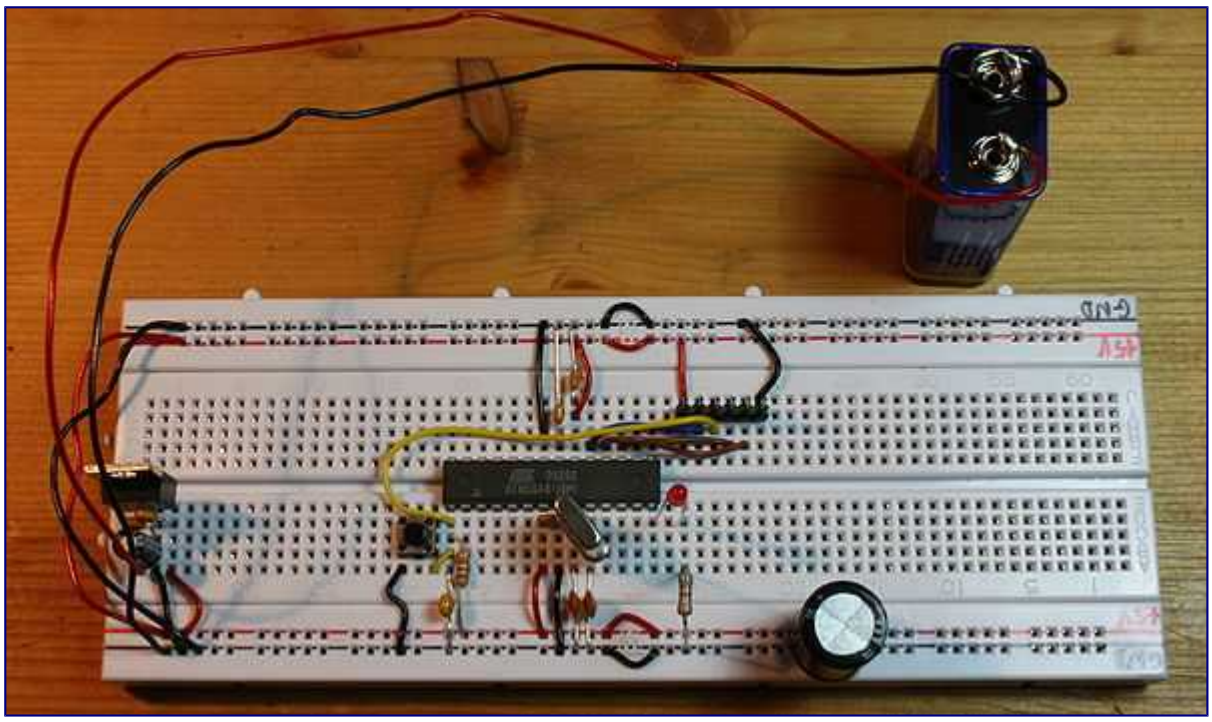
Man sieht, dass man die Eingangsspannung so klein wie möglich wählen sollte um dadurch die Verluste in Grenzen halten zu können. Außerdem ist es meist so, dass für eine geringere Stromentnahme auch eine niedrigere Differenzspannung ausreicht. In manchen Datenblätter ist z.B. angegeben 0,5 A = 1 V Spannungsgefälle und bei 1 A = 2 V Gefälle …

Weiterhin sei gesagt, dass es so genannte Low-Drop-, Ultra-Low-Drop- usw. -Regler gibt, die mit einer viel kleineren Differenz zwischen Ein- und Ausgangsspannung zurechtkommen, wodurch man die Verluste noch weiter drücken kann.

Eine Stromversorgung mit Batterien ist grundsätzlich auch möglich, wenn die elektrischen Grenzdaten des Mikrocontrollers eingehalten werden (max. Spannung, min. Spannung). Bei der geregelten Stromversorgung oben sollte die Batteriespannung ca. 1,5 .. 2,5 V (Dropout-Spannung des Linearreglers) größer sein als die Versorgungsspannung des Controllers. Die Versorgung aus einer Zelle ist ein Thema für Fortgeschrittene. In der Regel verzichtet man dabei auf die Spannungsregelung, da der Controller in einem weiten Speisespannungsbereich betriebsfähig ist.

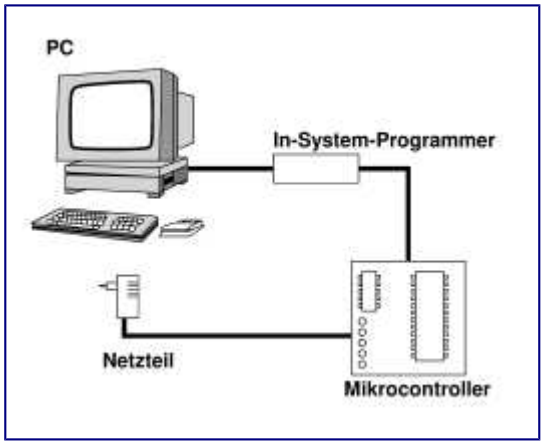
Beispielhafter Aufbau auf einem Steckbrett

Auf einem Steckbrett könnte eine Schaltung etwa so aussehen:



Hier ist die oben beschriebene Selbstbauschaltung zu sehen. Spannungsversorgung (links), 6-poliger ISP-Anschluss (rechts hinter dem µC), Quarz mit 2 Kondensatoren statt Oszillator, erweitert um eine LED mit Vorwiderstand an PB0 (rechts vor dem µC), einem Resettaster (links vor dem µC) und einem Stützkondensator zwischen +5V und GND (rechts unten).

Der ISP-Programmierer (In-System-Programmer)



ISP Programmierer

Dann braucht man nur noch den **ISP-Programmieradapter**, über den man die Programme vom PC in den Controller übertragen kann. Eine Übersicht über mögliche ISP-Programmer Varianten findet sich im Artikel AVR_In_System_Programmer.

Fertige ISP-Programmer zum Anschluss an den Parallelport oder USB gibt es z. B. auf http://shop.mikrocontroller.net/.

Eine Bauanleitung gibt es u.a. auf http://www.rn-wissen.de/index.php/AVR-ISP_Programmierkabel oder http://rumil.de/hardware/avrisp.html.

Den ISP-Adapter schließt man an den Parallelport an und verbindet ihn mit der Stiftleiste SV1 über

ein 6-adriges Kabel (siehe Schaltplan).

Sonstiges

Wer vorausschauend kauft, kauft mehr als einen Mikrocontroller. Bis der erste Controller defekt ist, oder man durch Austauschen sicher gehen möchte, ob der Fehler im Programm oder im Controller ist, vergeht nur wenig Zeit.

Tipp: Die Preise für Mikrocontroller haben eine deutliche Spannweite, nicht selten ist ein und derselbe Typ für 3 oder 8 Euro zu haben. Oft sind neuere oder größere Modelle billiger (ATmega8A statt ATmega8, ATmega328 statt ATmega8A). Eine Suche im Internet lohnt sich. Das Gleiche gilt für den Kauf von ISP-Programmierern.

Für weitere Kapitel dieses Tutorials sollte man sich noch die folgenden Bauteile besorgen:

Teil 2 (I/O-Grundlagen)

- 6 LEDs 5mm (Standard-LED, ruhig auch in unterschiedlichen Farben, rot/gelb/grün)
- 5 Taster
- 6 Widerstände 1k
- 5 Widerstände 10k

Teil 6 (LC-Display)

- 1 Potentiometer 10k
- 1 HD44780-kompatibles LCD, z. B. 4x20 oder 2x16 Zeichen
- besitzt das LCD eine Hintergrundbeleuchtung, dann noch einen Vorwiderstand dafür. Details dazu stehen im Datenblatt des LCD. Ein Wert von 50Ω sollte aber in jedem Fall passen. Schlimmstenfalls ist die Hintergrundbeleuchtung dann etwas zu dunkel.

Teil 10 (Der UART)

- 1 Pegelwandler MAX232, MAX232**A** oder MAX202
- 5 Kondensatoren
    - Bei einem MAX232: je 1µF Elektrolytkondensator
    - Bei einem MAX202 oder MAX232**A**: je 100nF Keramik- oder Elektrolytkondensator

  Die Kondensatoren dürfen auch größer sein. Ist man sich nicht sicher, welchen MAX232 man hat (A oder nicht A), dann die größeren Kondensatoren 1µF nehmen, die funktionieren auch beim MAX232A oder MAX202.

- 1 9-polige SUBD-Buchse (female)
- 1 dazu passendes Modem(nicht Nullmodem!)-Kabel

Teil 14 (ADC)

- 1 Kondensator 100n
- 1 Potentiometer 10k
- nach Lust und Laune temperatur- oder lichtabhängige Widerstände und jeweils einen Widerstand in der gleichen Größenordnung wie der Sensor

---

Teil 17 (Schieberegister)

- 2 Schieberegister 74HC595
- einige LED, damit man an die Schieberegister auch etwas anschließen kann, samt passenden Vorwiderständen

---

Teil 19 (7-Segmentanzeige)

- 4 7-Segmentanzeigen mit gemeinsamer Anode
- 4 PNP-Transistoren BC328
- 4 Widerstände 1k
- 7 Widerstände 100Ω

Für weitere Bauteile, die man als angehender µC Bastler auch des Öfteren mal benötigt, empfiehlt sich ein Blick in die Liste der Standardbauelemente bzw. in die Grundausstattung. Wenn Ihr Händler Großpackungen (zb. 100 Stück) von 100n Kondensatoren, 10k, 1k oder 100Ω Widerständen anbietet, sollten Sie deren Erwerb in Erwägung ziehen. Diese Bauteile benötigt man oft, und derartige Großpackungen sind meist nicht teurer, als wenn man einige wenige Exemplare einzeln kauft. Dies hängt damit zusammen, dass das Herauszählen von 9 Bauteilen für den Verkäufer teurer kommt, als 100 Bauteile abgepackt aus dem Regal zu nehmen.

Software

In diesem Tutorial wird nur auf die Programmierung in Assembler eingegangen, da Assembler für das Verständnis der Hardware am besten geeignet ist.

Assembler

Zuerst braucht man einen **Assembler**, der in Assemblersprache geschriebene Programme in Maschinencode übersetzt. Windows-User können das AVR-Studio von Atmel verwenden, das neben dem Assembler auch einen Simulator enthält, mit dem sich die Programme vor der Übertragung in den Controller testen lassen; für Linux gibt es tavrasm, avra und gavrasm.

Um die vom Assembler erzeugte ".hex"-Datei über den ISP-Adapter in den Mikrocontroller zu programmieren, kann man unter Windows z. B. das Programm yaap verwenden, für Linux gibt es uisp, für beide avrdude.

C

Wer in C programmieren möchte, kann den kostenlosen GNU-C-Compiler AVR-GCC (unter Windows "WinAVR") ausprobieren. Dieser C-Compiler kann auch in das für Assembler-Programmierung notwendige AVR-Studio integriert werden. In der Artikelsammlung gibt es ein umfangreiches Tutorial zu diesem Compiler;

Wer unter Windows und Linux gleichermassen kostenlos entwickeln will, der sollte sich die IDE Eclipse for C/C++ Developers und das AVR-Eclipse Plugin ansehen, beide sind unter Windows und Linux einfach zu installieren. Hier wird auch der AVR-GCC benutzt. In der Artikelsammlung gibt es ein umfangreiches AVR Eclipse Tutorial zu dieser IDE. Ebenfalls unter Linux und Windows verfügbar ist die Entwicklungsumgebung Code::Blocks (aktuelle, stabile Versionen sind als Nightly Builds regelmäßig im Forum verfügbar). Innerhalb dieser Entwicklungsumgebung können ohne die Installation zusätzlicher Plugins "AVR-Projekte" angelegt werden.

Fragen dazu stellt man am besten hier im GCC-Forum.

Pascal

Wer in Pascal programmieren muss, kann AVRPascal ausprobieren.
Dieser Pascalcompiler ist kostenfrei bis 4kb Code und bietet viele ausgereifte Bibliotheken für Servoansteuerung, Serielle Schnittstellen (COM, TWI, SPI), PWM, Timernutzung, LC-Displays usw.
Außerdem gibt es eine kostenfreie Version für den Mega8 und den Mega88. E-LAB.

Basic

Auch Basic-Fans kommen nicht zu kurz, für die gibt es z. B. Bascom AVR ($69, Demo verfügbar) oder das kostenlose [LunaAVR].

Forth

Wer einen direkten und interaktiven Zugang zum Controller haben will, sollte sich Forth anschauen. Voraussetzung ist ein serieller Anschluß (Max232), also etwas mehr als die Minimalbeschaltung.

Bereit für das erste Programm?

Zum nächsten Kapitel geht es erst, wenn man Hard- und Software in einer Weise zum Funktionieren gebracht hat, dass:

- die Programmiersoftware das Programmiergerät erkennt
- die Programmiersoftware den Mikrocontroller erkennt (alle heute üblichen Controller haben eine typspezifische Selbsterkennung), dazu muss der Controller mit Strom versorgt sein
- die verwendete Erstell-Software (bspw. der Assembler) aus einem leeren oder Beispiel-Projekt eine Programmier-Datei erstellt (meistens eine .HEX-Datei), die man in die Programmiersoftware laden kann (noch nicht in den Controller programmieren!)

Diesen arbeitsfähigen Zustand zu erreichen ist im Eigenbau (etwa dem Steckbrett) schwieriger als mit einem Starterkit — und der wichtigste Grund warum viele das Geld für das ansonsten nutzlose

Starterkit ausgeben.

Literatur

Bevor man anfängt, sollte man sich die folgenden PDF-Dateien runterladen und zumindest mal reinschauen:

- Datenblatt des ATmega8 (6,3 MB)
- Befehlssatz der AVRs (1,4 MB)
- oder Befehlssatz in deutscher Übersetzung online
- oder Datenblatt des ATmega8 in deutscher Übersetzung online

Das Datenblatt eines Controllers ist das wichtigste Dokument für einen Entwickler. Es enthält Informationen über die Pinbelegung, Versorgungsspannung, Beschaltung, Speicher, die Verwendung der IO-Komponenten und vieles mehr.

Im Befehlssatz sind alle Assemblerbefehle der AVR-Controllerfamilie aufgelistet und erklärt.

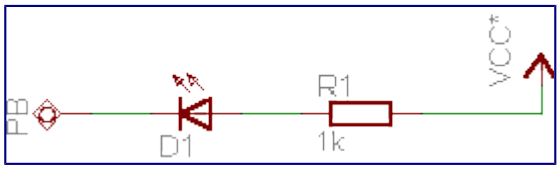
AVR-Tutorial: IO-Grundlagen

Inhaltsverzeichnis

- 1 Hardware
- 2 Zahlensysteme
- 3 Ausgabe
  - 3.1 Assembler-Sourcecode
  - 3.2 Assemblieren
  - 3.3 Hinweis: Konfigurieren der Taktversorgung des ATmega8
  - 3.4 Programmerklärung
- 4 Eingabe
  - 4.1 mögliche Zeitverzögerungen
- 5 Pullup-Widerstand
- 6 Zugriff auf einzelne Bits
- 7 Zusammenfassung der Portregister
- 8 Ausgänge benutzen, wenn mehr Strom benötigt wird

Hardware

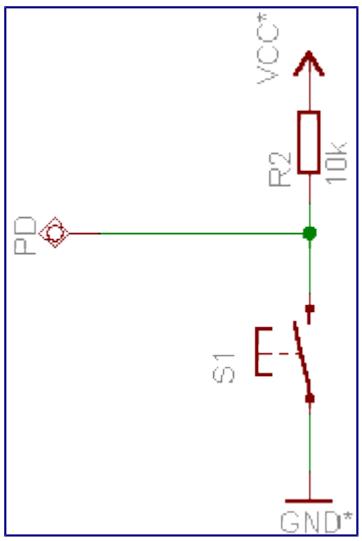
Für die ersten Versuche braucht man nur ein paar Taster und LEDs an die IO-Ports des AVRs anzuschließen. An **PB0-PB5** schließt man 6 LEDs über einen Vorwiderstand von je 1 kΩ gegen Vcc (5V) an. In der Praxis ist es unerheblich, ob der Widerstand vor oder nach der Diode liegt, wichtig ist nur, dass er da ist. Weitere Details zu LEDs und entsprechenden Vorwiderständen findet ihr im Artikel über LEDs und in diesem Thread im Forum.



Standard Led Anschluss

Dass die LEDs an den gleichen Pins wie der ISP-Programmer angeschlossen sind, stört übrigens normalerweise nicht. Falls wider Erwarten deshalb Probleme auftreten sollten, kann man versuchen, den Vorwiderstand der LEDs zu vergrößern.

An **PD0-PD3** kommen 4 Taster mit je einem 10 kΩ Pullup-Widerstand:



Standard Taster Anschluss

Zahlensysteme

Bevor es losgeht, hier noch ein paar Worte zu den verschiedenen Zahlensystemen.

Binärzahlen werden für den Assembler im Format **0b00111010** geschrieben, Hexadezimalzahlen als **0x7F**. Umrechnen kann man die Zahlen z. B. mit dem Windows-Rechner. Hier ein paar Beispiele:

Dezimal	Hexadezimal	Binär
0	0x00	0b00000000
1	0x01	0b00000001
2	0x02	0b00000010
3	0x03	0b00000011
4	0x04	0b00000100
5	0x05	0b00000101
6	0x06	0b00000110
7	0x07	0b00000111
8	0x08	0b00001000
9	0x09	0b00001001
10	0x0A	0b00001010
11	0x0B	0b00001011
12	0x0C	0b00001100
13	0x0D	0b00001101
14	0x0E	0b00001110
15	0x0F	0b00001111
100	0x64	0b01100100
255	0xFF	0b11111111

"0b" und "0x" haben für die Berechnung keine Bedeutung, sie zeigen nur an, dass es sich bei dieser

Zahl um eine Binär- bzw. Hexadezimalzahl handelt.

Wichtig dabei ist es, dass Hexadezimal- bzw. Binärzahlen bzw. Dezimalzahlen nur unterschiedliche Schreibweisen für immer das Gleiche sind: eine Zahl. Welche Schreibweise bevorzugt wird, hängt auch vom Verwendungszweck ab. Je nachdem kann die eine oder die andere Schreibweise klarer sein.

Auch noch sehr wichtig: Computer und µCs beginnen immer bei 0 zu zählen, d.h. wenn es 8 Dinge (Bits etc.) gibt, hat das erste die Nummer 0, das zweite die Nummer 1, ..., und das letzte (das 8.) die Nummer 7(!).

Ausgabe

Assembler-Sourcecode

Unser erstes Assemblerprogramm, das wir auf dem Controller laufen lassen möchten, sieht so aus:

```
.include "m8def.inc"         ; Definitionsdatei für den Prozessortyp einbinden

         ldi r16, 0xFF       ; lade Arbeitsregister r16 mit der Konstanten 0xFF
         out DDRB, r16       ; Inhalt von r16 ins IO-Register DDRB ausgeben

         ldi r16, 0b11111100 ; 0b11111100 in r16 laden
         out PORTB, r16      ; r16 ins IO-Register PORTB ausgeben

ende:    rjmp ende           ; Sprung zur Marke "ende" -> Endlosschleife
```

Assemblieren

Das Programm muss mit der Endung ".asm" abgespeichert werden, z. B. als "leds.asm". Diese Datei können wir aber noch nicht direkt auf den Controller programmieren. Zuerst müssen wir sie dem Assembler füttern. Bei wavrasm funktioniert das z. B., indem wir ein neues Fenster öffnen, den Programmtext hineinkopieren, speichern und auf "assemble" klicken. Wichtig ist, dass sich die Datei "m8def.inc" (wird beim Atmel-Assembler mitgeliefert) im gleichen Verzeichnis wie die Assembler-Datei befindet. Der Assembler übersetzt die Klartext-Befehle des Assemblercodes in für den Mikrocontroller verständlichen Binärcode und gibt ihn in Form einer sogenannten "Hex-Datei" aus. Diese Datei kann man dann mit der entsprechenden Software direkt in den Controller programmieren.

Hinweis: Konfigurieren der Taktversorgung des ATmega8

Beim **ATmega8** ist vom Hersteller der interne 1 MHz-Oszillator aktiviert; weil dieser für viele Anwendungen (z. B. das UART, siehe späteres Kapitel) aber nicht genau genug ist, soll der Mikrocontroller seinen Takt aus dem angeschlossenen 4 MHz-Quarzoszillator beziehen. Dazu müssen ein paar Einstellungen an den **Fusebits** des Controllers vorgenommen werden. Am besten und sichersten geht das mit dem Programm yaap. Wenn man das Programm gestartet hat und der ATmega8 richtig erkannt wurde, wählt man aus den Menüs den Punkt "Lock Bits & Fuses" und

klickt zunächst auf "Read Fuses". Das Ergebnis sollte so aussehen: Screenshot. Nun ändert man die Kreuze so, dass das folgende Bild entsteht: Screenshot und klickt auf "Write Fuses". Vorsicht, wenn die Einstellungen nicht stimmen, kann es sein, dass die ISP-Programmierung deaktiviert wird und man den AVR somit nicht mehr programmieren kann! Die FuseBits bleiben übrigens nach dem Löschen des Controllers aktiv, müssen also nur ein einziges Mal eingestellt werden. Mehr über die Fuse-Bits findet sich im Artikel AVR Fuses.

Nach dem Assemblieren sollte eine neue Datei mit dem Namen "leds.hex" oder "leds.rom" vorhanden sein, die man mit yaap, PonyProg oder AVRISP in den Flash-Speicher des Mikrocontrollers laden kann. Wenn alles geklappt hat, leuchten jetzt die ersten beiden angeschlossenen LEDs.

Programmerklärung

In der ersten Zeile wird die Datei m8def.inc eingebunden, welche die prozessortypischen Bezeichnungen für die verschiedenen Register definiert. Wenn diese Datei fehlen würde, wüsste der Assembler nicht, was mit "PORTB", "DDRD" usw. gemeint ist. Für jeden AVR-Mikrocontroller gibt es eine eigene derartige Include-Datei, da zwar die Registerbezeichnungen bei allen Controllern mehr oder weniger gleich sind, die Register aber auf unterschiedlichen Controllern unterschiedlich am Chip angeordnet sind und nicht alle Funktionsregister auf allen Prozessoren existieren. Für einen ATmega8 beispielsweise würde die einzubindende Datei m8def.inc heißen. Normalerweise ist also im Namen der Datei der Name des Chips in irgendeiner Form, auch abgekürzt, enthalten. Kennt man den korrekten Namen einmal nicht, so sieht man ganz einfach nach. Alle Include-Dateien wurden von Atmel in einem gemeinsamen Verzeichnis gespeichert. Das Verzeichnis ist bei einer Standardinstallation am PC auf C:\Programme\Atmel\AVR Tools\AvrAssembler\Appnotes\. Einige Include-Dateien heißen

```
AT90s2313:  2313def.inc
ATmega8:    m8def.inc
ATmega16:   m16def.inc
ATmega32:   m32def.inc
ATTiny12:   tn12def.inc
ATTiny2313: tn2313def.inc
```

Um sicher zu gehen, dass man die richtige Include-Datei hat, kann man diese mit einem Texteditor (AVR-Studio oder Notepad) öffnen. Der Name des Prozessors wurde von Atmel immer an den Anfang der Datei geschrieben:

```
;*************************************************************************
;* A P P L I C A T I O N   N O T E   F O R   T H E   A V R   F A M I L Y
;*
;* Number               :AVR000
;* File Name            :"2313def.inc"
;* Title                :Register/Bit Definitions for the AT90S2313
;* Date                 :99.01.28
;* Version              :1.30
;* Support E-Mail       :avr@atmel.com
;* Target MCU           :AT90S2313
...
```

Aber jetzt weiter mit dem selbstgeschriebenen Programm.

In der 2. Zeile wird mit dem Befehl **ldi r16, 0xFF** der Wert 0xFF (entspricht 0b11111111) in das Register r16 geladen (mehr Infos unter Adressierung). Die AVRs besitzen 32 Arbeitsregister, r0-r31, die als Zwischenspeicher zwischen den I/O-Registern (z. B. DDRB, PORTB, UDR...) und dem RAM genutzt werden. Zu beachten ist außerdem, dass die ersten 16 Register (r0-r15) nicht von jedem Assemblerbefehl genutzt werden können. Ein Register kann man sich als eine Speicherzelle direkt im Mikrocontroller vorstellen. Natürlich besitzt der Controller noch viel mehr Speicherzellen, die werden aber ausschließlich zum Abspeichern von Daten verwendet. Um diese Daten zu manipulieren, müssen sie zuerst in eines der Register geladen werden. Nur dort ist es möglich, die Daten zu manipulieren und zu verändern. Ein Register ist also vergleichbar mit einer Arbeitsfläche, während der restliche Speicher eher einem Stauraum entspricht. Will man arbeiten, so muss das Werkstück (= die Daten) aus dem Stauraum auf die Arbeitsfläche geholt werden und kann dann dort bearbeitet werden. Der Befehl Befehl **ldi** lädt jetzt einen bestimmten konstanten Wert in so ein Arbeitsregister. In diesem Fall kommt der zu ladende Wert also nicht aus dem Stauraum, sondern der Programmierer kennt ihn bereits. Auch Assemblerbefehle sind nicht einfach willkürlich gewählte Buchstabenkombinationen, sondern sind oft Abkürzungen für eine bestimmte Aktion. **ldi** bedeutet in Langform **Load immediate**. Load ist klar - laden. Und immediate bedutet, dass der zu ladende Wert beim Befehl selber angegeben wurde (engl. immediate - unmittelbar).

Die Erklärungen nach dem Semikolon sind Kommentare und werden vom Assembler nicht beachtet.

Der 3. Befehl, **out**, gibt den Inhalt von r16 (=0xFF) in das Datenrichtungsregister für Port B aus. Das Datenrichtungsregister legt fest, welche Portpins als Ausgang und welche als Eingang genutzt werden. Steht in diesem Register ein Bit auf 0, wird der entsprechende Pin als Eingang konfiguriert, steht es auf 1, ist der Pin ein Ausgang. In diesem Fall sind also alle 6 Pins von Port B Ausgänge. Datenrichtungsregister können ebenfalls nicht direkt beschrieben werden, daher muss man den Umweg über eines der normalen Register r16 - r31 gehen.

Der nächste Befehl, **ldi r16, 0b11111100** lädt den Wert 0b11111100 in das Arbeitsregister r16, der durch den darauffolgenden Befehl **out PORTB, r16** in das I/O-Register PORTB (und damit an den Port, an dem die LEDs angeschlossen sind) ausgegeben wird. Eine 1 im PORTB-Register bedeutet, dass an dem entsprechenden Anschluss des Controllers die Spannung 5V anliegt, bei einer 0 sind es 0V (Masse).

Schließlich wird mit **rjmp ende** ein Sprung zur Marke **ende:** ausgelöst, also an die gleiche Stelle, wodurch eine Endlosschleife entsteht. Sprungmarken schreibt man gewöhnlich an den Anfang der Zeile, Befehle in die 2. und Kommentare in die 3. Spalte. Ein Marke ist einfach nur ein symbolischer Name, auf den man sich in Befehlen beziehen kann. Sie steht stellvertretend für die Speicheradresse des unmittelbar folgenden Befehls. Der Assembler, der den geschriebenen Text in eine für den µC ausführbare Form bringt, führt über die Marken Buch und ersetzt in den eigentlichen Befehlen die Referenzierungen auf die Marken mit den korrekten Speicheradressen.

Bei Kopier- und Ladebefehlen (ldi, in, out...) wird immer der 2. Operand in den ersten kopiert:

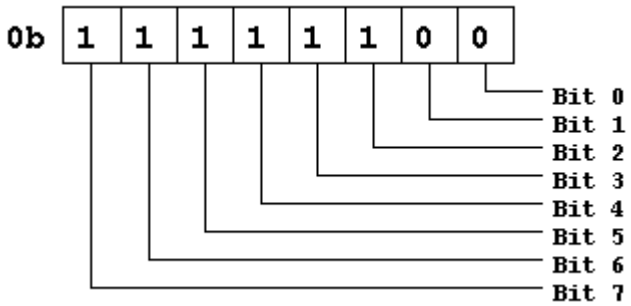
```
ldi r17, 15     ; das Register r17 wird mit der Konstanten 15 geladen
mov r16, r17    ; das Register r16 wird mit dem Inhalt des Registers
```

```
r17 geladen
         out PORTB, r16  ; das IO-Register "PORTB" wird mit dem Inhalt des
Registers r16 geladen
         in r16, PIND    ; das Register 16 wird mit dem Inhalt des IO-Registers
"PIND" geladen
```

Wer mehr über die Befehle wissen möchte, sollte sich die PDF-Datei Instruction Set (1,27 MB) runterladen (benötigt Acrobat Reader oder in der Hilfe von Assembler oder AVR-Studio nachschauen. Achtung: nicht alle Befehle sind auf jedem Controller der AVR-Serie verwendbar!

Nun sollten die beiden ersten LEDs leuchten, weil die Portpins PB0 und PB1 durch die Ausgabe von 0 (low) auf Masse (0V) gelegt werden und somit ein Strom durch die gegen Vcc (5V) geschalteten LEDs fließen kann. Die 4 anderen LEDs sind aus, da die entsprechenden Pins durch die Ausgabe von 1 (high) auf 5V liegen.

Warum leuchten die beiden ersten LEDs, wo doch die beiden letzen Bits auf 0 gesetzt sind? Das liegt daran, dass man die Bitzahlen von rechts nach links schreibt. Ganz rechts steht das niedrigstwertige Bit ("LSB", Least Significant Bit), das man als Bit 0 bezeichnet, und ganz links das höchstwertige Bit ("MSB", Most Significant Bit), bzw. Bit 7. Das Prefix "0b" gehört nicht zur Zahl, sondern sagt dem Assembler, dass die nachfolgende Zahl in binärer Form interpretiert werden soll.



Das LSB steht für PB0, und das MSB für PB7... aber PB7 gibt es doch z. B. beim AT90S4433 gar nicht, es geht doch nur bis PB5? Der Grund ist einfach: Am Gehäuse des AT90S4433 gibt es nicht genug Pins für den kompletten Port B, deshalb existieren die beiden obersten Bits nur intern.

Eingabe

Im folgenden Programm wird Port B als Ausgang und Port D als Eingang verwendet:

Download leds+buttons.asm

```
.include "m8def.inc"

         ldi r16, 0xFF
         out DDRB, r16      ; Alle Pins am Port B durch Ausgabe von 0xFF ins
                            ; Richtungsregister DDRB als Ausgang konfigurieren
         ldi r16, 0x00
         out DDRD, r16      ; Alle Pins am Port D durch Ausgabe von 0x00 ins
                            ; Richtungsregister DDRD als Eingang konfigurieren
loop:
         in r16, PIND       ; an Port D anliegende Werte (Taster) nach r16
einlesen
         out PORTB, r16     ; Inhalt von r16 an Port B ausgeben
```

```
rjmp loop         ; Sprung zu "loop:" -> Endlosschleife
```

Wenn der Port D als Eingang geschaltet ist, können die anliegenden Daten über das IO-Register **PIND** eingelesen werden. Dazu wird der Befehl **in** verwendet, der ein IO-Register (in diesem Fall PIND) in ein Arbeitsregister (z. B. r16) kopiert. Danach wird der Inhalt von r16 mit dem Befehl **out** an Port B ausgegeben. Dieser Umweg ist notwendig, da man nicht direkt von einem IO-Register in ein anderes kopieren kann.

rjmp loop sorgt dafür, dass die Befehle **in r16, PIND** und **out PORTB, r16** andauernd wiederholt werden, so dass immer die zu den gedrückten Tasten passenden LEDs leuchten.

Achtung: Auch wenn es hier nicht explizit erwähnt wird: Man kann natürlich jeden Pin eines jeden Ports einzeln auf Ein- oder Ausgabe schalten. Dass hier ein kompletter Port jeweils als Eingabe bzw. Ausgabe benutzt wurde, ist reine Bequemlichkeit.

In komplexeren Situationen als der einfachen Verbindung eines Port-Pins mit einem Taster, der zuverlässig auf GND-Potential zieht, ist die **Schaltschwelle** des Eingangstreibers zu beachten. Diese liegt bei etwa 50 % der Versorgungsspannung. In dieser Testschaltung wird dieser Aspekt genauer untersucht.

mögliche Zeitverzögerungen

Vorsicht! In bestimmten Situationen kann es passieren, dass scheinbar Pins nicht richtig gelesen werden.

Speziell bei der Abfrage von Matrixtastaturen kann der Effekt auftreten, dass Tasten scheinbar nicht reagieren. Typische Sequenzen sehen dann so aus:

```
ldi r16,0x0F
out DDRD,r16    ; oberes Nibble Eingang, unteres Ausgang
ldi r16,0xFE
out PORTD,r16   ; PD0 auf 0 ziehen, PD4..7 Pull ups aktiv
in  r17,PIND    ; Pins lesen schlägt hier fehl!
```

Warum ist das problematisch? Nun, der AVR ist ein RISC-Microcontroller, welcher die meisten Befehle in einem Takt ausführt. Gleichzeitig werden aber alle Eingangssignale über FlipFlops abgetastet (synchronisiert), damit sie sauber im AVR zur Verfügung stehen. Dadurch ergibt sich eine Verzögerung (Latenz) von bis zu 1,5 Takten, mit der auf externe Signale reagiert werden kann. Die Erklärung dazu findet man im Datenblatt unter der Überschrift "I/O Ports - Reading the Pin Value".

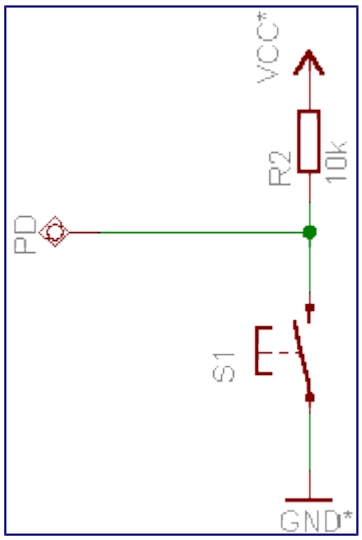
Was tun? Wenn der Wert einer Port-Eingabe von einer unmittelbar vorangehenden Port-Ausgabe abhängt, muss man wenigstens einen weiteren Befehl zwischen beiden einfügen, im einfachsten Fall ein **nop**. **nop** bedeutet in Langform **no operation**, und genau das macht der Befehl auch: nichts. Er dient einzig und alleine dazu, dass der Prozessor einen Befehl abarbeitet, also etwas zu tun hat, aber ansonsten an den Registern oder sonstigen Internals nichts verändert.

```
ldi r16,0x0F
out DDRD,r16    ; oberes Nibble Eingang, unteres Ausgang
ldi r16,0xFE
out PORTD,r16   ; PD0 auf 0 ziehem, PD4..7 Pull ups aktiv
NOP             ; Delay der Synchronisations-FlipFlops ausgleichen
```

```
        in  r17,PIND    ; Pins lesen ist hier OK.
```

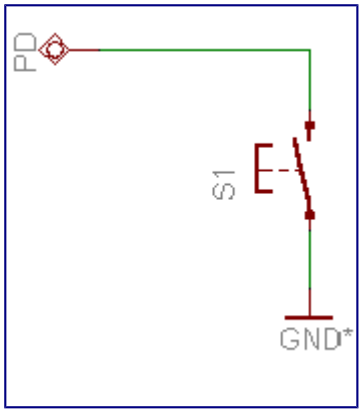
Ein weiteres Beispiel für dieses Verhalten bei rasch aufeinanderfolgenden **out** und **in** Anweisungen ist in einem Forenbeitrag zur Abfrage des Busyflag bei einem LCD angegeben. Dort spielen allerdings weitere, vom LCD-Controller abhängige Timings eine wesentliche Rolle für den korrekten Programmablauf.

Pullup-Widerstand



Standard Taster Anschluss

Bei der Besprechung der notwendigen Beschaltung der Ports wurde an einen Eingangspin jeweils ein Taster mit einem Widerstand nach Vcc vorgeschlagen. Diesen Widerstand nennt man einen **Pullup**-Widerstand. Wenn der Taster geöffnet ist, so ist es seine Aufgabe, den Eingangspegel am Pin auf Vcc zu ziehen. Daher auch der Name: 'pull up' (engl. für hochziehen). Ohne diesen Pullup-Widerstand würde ansonsten der Pin bei geöffnetem Taster in der Luft hängen, also weder mit Vcc noch mit GND verbunden sein. Dieser Zustand ist aber unbedingt zu vermeiden, da bereits elektromagnetische Einstreuungen auf Zuleitungen ausreichen, dem Pin einen Zustand vorzugaukeln, der in Wirklichkeit nicht existiert. Der Pullup-Widerstand sorgt also für einen definierten 1-Pegel bei geöffnetem Taster. Wird der Taster geschlossen, so stellt dieser eine direkte Verbindung zu GND her und der Pegel am Pin fällt auf GND. Durch den Pullup-Widerstand rinnt dann ein kleiner Strom von Vcc nach GND. Da Pullup-Widerstände in der Regel aber relativ hochohmig sind, stört dieser kleine Strom meistens nicht weiter.



Taster bei Benutzung des interen Pullup

Anstelle eines externen Widerstandes wäre es auch möglich, den Widerstand wegzulassen und stattdessen den in den AVR eingebauten Pullup-Widerstand zu aktivieren. Die Beschaltung eines Tasters vereinfacht sich dann zum einfachst möglichen Fall: Der Taster wird direkt an den Eingangspin des µC angeschlossen und schaltet nach Masse durch.

Das geht allerdings nur dann, wenn der entsprechende Mikroprozessor-Pin auf Eingang geschaltet wurde. Ein Pullup-Widerstand hat nun mal nur bei einem Eingangspin einen Sinn. Bei einem auf Ausgang geschalteten Pin sorgt der Mikroprozessor dafür, dass ein dem Port-Wert entsprechender Spannungspegel ausgegeben wird. Ein Pullup-Widerstand wäre in so einem Fall kontraproduktiv, da der Widerstand versucht, den Pegel am Pin auf Vcc zu ziehen, während eine 0 im Port-Register dafür sorgt, dass der Mikroprozessor versuchen würde, den Pin auf GND zu ziehen.

Ein Pullup-Widerstand an einem Eingangspin wird durch das **PORT**-Register gesteuert. Das **PORT**-Register erfüllt also 2 Aufgaben. Bei einem auf Ausgang geschalteten Port steuert es den Pegel an den Ausgangspins. Bei einem auf Eingang geschalteten Port steuert es, ob die internen Pullup-Widerstände aktiviert werden oder nicht. Ein 1-Bit aktiviert den entsprechenden Pullup-Widerstand.

DDRx	PORTx	IO-Pin-Zustand
0	0	Eingang ohne Pull-Up (Resetzustand)
0	1	Eingang mit Pull-Up
1	0	Push-Pull-Ausgang auf LOW
1	1	Push-Pull-Ausgang auf HIGH

```
.include "m8def.inc"

         ldi r16, 0xFF
         out DDRB, r16      ; Alle Pins am Port B durch Ausgabe von 0xFF ins
                            ; Richtungsregister DDRB als Ausgang konfigurieren
         ldi r16, 0x00
         out DDRD, r16      ; Alle Pins am Port D durch Ausgabe von 0x00 ins
                            ; Richtungsregister DDRD als Eingang konfigurieren

         ldi r16, 0xFF      ; An allen Pins vom Port D die Pullup-Widerstände
         out PORTD, r16     ; aktivieren. Dies geht deshalb durch eine Ausgabe
                            ; nach PORTD, da ja der Port auf Eingang gestellt
ist.
loop:
         in r16, PIND       ; an Port D anliegende Werte (Taster) nach r16
einlesen
         out PORTB, r16     ; Inhalt von r16 an Port B ausgeben
         rjmp loop          ;  zu "loop:" -> Endlosschleife
```

Werden auf diese Art und Weise die AVR-internen Pullup-Widerstände aktiviert, so sind keine externen Widerstände mehr notwendig und die Beschaltung vereinfacht sich zu einem Taster, der einfach nur den µC-Pin mit GND verbindet.

Zugriff auf einzelne Bits

Man muss nicht immer ein ganzes Register auf einmal einlesen oder mit einem neuen Wert laden. Es gibt auch Befehle, mit denen man einzelne Bits abfragen und ändern kann:

- Der Befehl **sbic** ("skip if bit cleared") überspringt den darauffolgenden Befehl, wenn das angegebene Bit 0 (low) ist.
- **sbis** ("skip if bit set") bewirkt das Gleiche, wenn das Bit 1 (high) ist.
- Mit **cbi** ("clear bit") wird das angegebene Bit auf 0 gesetzt.
- **sbi** ("set bit") bewirkt das Gegenteil.

Achtung: Diese Befehle können nur auf die IO-Register angewandt werden!

Der große Vorteil, vor allen Dingen der **cbi** bzw. **sbi** Instruktionen ist es, dass sie tatsächlich nur ein einziges Bit am Port manipulieren. Dies ist insbesonders dann interessant, wenn an einem Port mehrere LED hängen, die unterschiedliche Dinge anzeigen. Will man dann eine bestimmte LED ein bzw. ausschalten, dann sollen sich ja deswegen die anderen LED nicht verändern. Greift man mittel **out** auf den kompletten Port zu, dann muss man dies berücksichtigen. Wird die eine LED aber mittels **cbi** bzw. **sbi** manipuliert, dann braucht man sich um die anderen LED an diesem Port nicht kümmern - deren Zustand verändert sich durch **cbi** bzw. **sbi** nicht.

Am besten verstehen kann man das natürlich an einem Beispiel:

Download bitaccess.asm

```
.include "m8def.inc"

         ldi r16, 0xFF
         out DDRB, r16       ; Port B ist Ausgang

         ldi r16, 0x00
         out DDRD, r16       ; Port D ist Eingang


         ldi r16, 0xFF
         out PORTB, r16      ; PORTB auf 0xFF setzen -> alle LEDs aus

loop:    sbic PIND, 0        ; "skip if bit cleared", nächsten Befehl
überspringen,
                             ; wenn Bit 0 im IO-Register PIND =0 (Taste 0
gedrückt)
         rjmp loop           ; Sprung zu "loop:" -> Endlosschleife

         cbi PORTB, 3        ; Bit 3 im IO-Register PORTB auf 0 setzen -> 4. LED
an

ende:    rjmp ende           ; Endlosschleife
```

Dieses Programm wartet so lange in einer Schleife ("**loop:**"..."**rjmp loop**"), bis Bit 0 im Register **PIND** 0 wird, also die erste Taste gedrückt ist. Durch **sbic** wird dann der Sprungbefehl zu "**loop:**" übersprungen, die Schleife wird also verlassen und das Programm danach fortgesetzt. Ganz am Ende schließlich wird das Programm durch eine leere Endlosschleife praktisch "angehalten", da es ansonsten wieder von vorne beginnen würde.

Zusammenfassung der Portregister

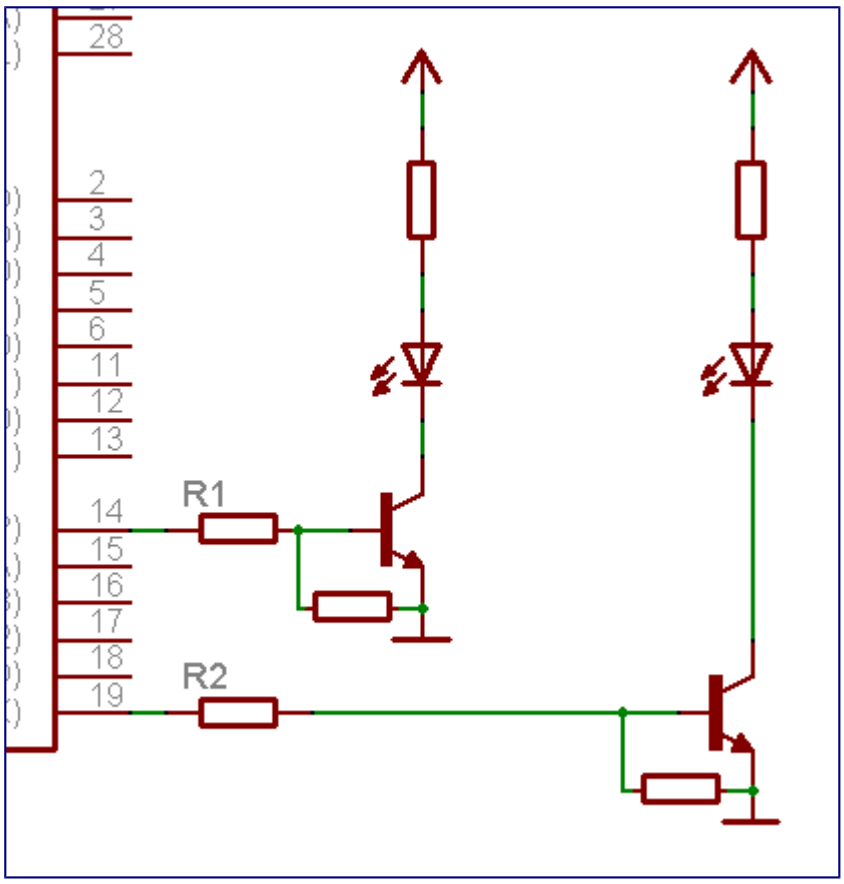
Für jeden Hardwareport gibt es im Mikroprozessor insgesamt 3 Register:

- Das Datenrichtungsregister **DDRx**. Es wird verwendet um die Richtung jedes einzelnen Mikroprozessor-Pins festzulegen. Eine 1 an der entsprechenden Bit Position steht für Ausgang, eine 0 steht für Eingang.
- Das Einleseregister **PINx**. Es wird verwendet um von einem Mikroprozessor-Pin den aktuellen, extern anliegenden Zustand einzulesen. Dazu muss das entsprechende Datenrichtungsbit auf Eingang geschaltet sein.
- Das Ausgangsregister **PORTx**. Es erfüllt 2 Funktionen, je nachdem wie das zugehörige Datenrichtungsbit geschaltet ist.
    - Steht es auf Ausgang, so wird bei einer entsprechenden Zuweisung an das **PORTx** Register der entsprechende Mikroprozessor-Pin auf den angegebenen Wert gesetzt.
    - Steht es auf Eingang, so beeinflusst das **PORTx**-Bit den internen Pullup-Widerstand an diesem Mikroprozessor-Pin. Bei einer 0 wird der Widerstand abgeschaltet, bei einer 1 wird der Widerstand an den Eingangs-Pin zugeschaltet.
- Bei den neueren AVR (wie z. B. *ATtiny13*, *ATtiny2313*, *ATtiny24/44/84*, *ATtiny25/45/85*, *ATmega48/88/168*, usw.) kann man als Ausgang konfigurierte Pins toggeln (**PORTx** zwischen 0 und 1 „umschalten“), indem man eine 1 an die entsprechende Bit Position des **PINx** Register schreibt.

Ausgänge benutzen, wenn mehr Strom benötigt wird

Man kann nicht jeden beliebigen Verbraucher nach dem LED-Vorbild von oben an einen µC anschließen. Die Ausgänge des ATMega8 können nur eine begrenzte Menge Strom liefern, so dass der Chip schnell überfordert ist, wenn eine nachgeschaltete Schaltung mehr Strom benötigt. Die Ausgangstreiber des µC würden in solchen Fällen den Dienst quittieren und durchbrennen.

Abhilfe schafft in solchen Fällen eine zusätzliche Treiberstufe, die im einfachsten Fall mit einem Transistor als Schalter aufgebaut wird.



Transistor Treiberstufe

Die LED samt zugehörigen Widerständen dienen hier lediglich als Sinnbild für den Verbraucher, der vom µC ein und ausgeschaltet werden soll. Welcher Transistor als Schalter benutzt werden kann, hängt vom Stromverbrauch des Verbrauchers ab. Die Widerstände **R1** und **R2** werden als **Basiswiderstände** der Transistoren bezeichnet. Für ihre Berechnung siehe z. B. hier. Um eine sichere Störfestigkeit im Resetfall des Mikrocontrollers zu gewähren (wenn der µC daher die Ausgänge noch nicht ansteuert), sollte man noch einen Pulldown Widerstand zwischen Basis und Emitter schalten oder einen digitalen Transistor (z. B. BCR135) mit integriertem Basis- und Basisemitterwiderstand benutzen.

Um ein Relais an einen µC-Ausgang anzuschließen, siehe hier.

AVR-Tutorial: Logik

In weiterer Folge werden immer wieder 4 logische Grundoperationen auftauchen:

- UND
- ODER
- NICHT
- XOR (Exlusiv oder)

Was hat es mit diesen Operationen auf sich?

Inhaltsverzeichnis

- 1 Allgemeines
- 2 Die Operatoren
  - 2.1 UND
  - 2.2 ODER
  - 2.3 NICHT
  - 2.4 XOR (Exlusives Oder)
- 3 Weblinks

Allgemeines

Die logischen Operatoren werden mit einem Register und einem zweiten Argument gebildet. Das zweite Argument kann ebenfalls ein Register oder aber eine direkt angegebene Zahl sein.

Da ein Register aus 8 Bit besteht, werden die logischen Operatoren immer auf alle 8 Bit Paare gleichzeitig angewendet.

Mit den logischen Grundoperationen werden die beiden Argumente miteinander verknüpft und das Ergebnis der Verknüpfung im Register des ersten Argumentes abgelegt.

Die Operatoren

UND

Wahrheitstabelle UND

A	B	Ergebnis
0	0	0
0	1	0

1	0	0
1	1	1

Das Ergebnis ist genau dann 1, wenn A **und** B 1 sind.

Verwendung

- gezielt einzelne Bits auf 0 setzen
- dadurch auch die Verwendung um einzelne Bits auszumaskieren

AVR Befehle

```
        and  r16, r17             ; Verknüpfung zweier Register
        andi r16, 0b01011010      ; Verknüpfung eines Registers mit einer
Konstanten
```

Die beiden Operanden werden mittels **UND** miteinander verknüpft, wobei jeweils die gleichwertigen Bits der Operanden laut Wahrheitstabelle unabhängig voneinander verknüpft werden.

Sei der Inhalt des Registers r16 = 0b11001100, so lautet die Verknüpfung **andi r16, 0b01011010**

```
      0b11001100
      0b01011010   und
     -----------
      0b01001000
```

Das Ergebnis wird im ersten Operanden (r16) abgelegt.

Im Ergebnis haben nur diejenigen Bits denselben Wert den sie im ersten Argument hatten, bei denen im zweiten Argument (in der Maske) eine 1 war. Alle anderen Bits sind auf jeden Fall 0. Da in der Maske

```
     0b01011010
```

die Bits 0, 2, 5, 7 eine 0 aufweisen, ist auch im Ergebnis an diesen Stellen mit Sicherheit eine 0. Alle andern Bits (diejenigen bei denen in der Maske eine 1 steht), werden aus der Ursprungszahl so wie sie sind übernommen.

ODER

Wahrheitstabelle ODER

A	B	Ergebnis
0	0	0
1	0	1
0	1	1
1	1	1

Das Ergebnis ist genau dann 1, wenn A **oder** B **oder beide** 1 sind.

Verwendung

- gezielt einzelne Bits auf 1 setzen

AVR Befehle

```
        or   r16, r17           ; Verknüpfung zweier Register
        ori  r16, 0b01011010    ; Verknüpfung eines Registers mit einer
Konstanten
```

Die beiden Operanden werden miteinander **ODER** verknüpft, wobei jeweils die jeweils gleichwertigen Bits der Operanden laut Wahrheitstabelle unabhängig voneinander verknüpft werden.

Sei der Inhalt des Registers r16 = 0b11001100, so lautet die Verknüpfung **ori r16, 0b01011010**

```
      0b11001100
      0b01011010     oder
     -----------
      0b11011110
```

Das Ergebnis wird im ersten Operanden (r16) abgelegt.

Im Ergebnis tauchen an den Bitpositionen an denen in der Maske eine 1 war auf jeden Fall ebenfalls eine 1 auf. In den restlichen Bitpositionen hängt es vom ersten Argument ab, ob im Ergebnis eine 1 auftaucht oder nicht.

Da in der Maske

```
      0b01011010
```

an den Bitpositionen 1, 3, 4, 6 eine 1 steht, ist an diesen Bitpositionen im Ergebnis ebenfalls mit Sicherheit eine 1. Alle andern Bits werden so wie sie sind aus der Ursprungszahl übernommen.

NICHT

Wahrheitstabelle NICHT

A	Ergebnis
0	1
1	0

Das Ergebnis ist genau dann 1, wenn A **nicht** 1 ist.

Verwendung

- alle Bits eines Bytes invertieren.

AVR Befehle

```
        com  r16
```

Sei der Inhalt des Registers r16 = 0b11001100, so lautet die Verknüpfung **com r16**

```
      0b11001100     nicht
     -----------
```

```
0b00110011
```

Das Ergebnis wird im ersten und einzigen Operanden (r16) abgelegt.

XOR (Exlusives Oder)

Wahrheitstabelle XOR

A	B	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	0

Das Ergebnis ist genau dann 1, wenn A **oder** B, aber **nicht beide** 1 sind.

Verwendung

- gezielt einzelne Bits umdrehen

AVR Befehle

```
eor  r16, r17           ; Verknüpfung zweier Register
```

Die beiden Operanden werden miteinander **XOR** verknüpft, wobei die jeweils gleichwertigen Bits der Operanden laut Wahrheitstabelle unabhängig voneinander verknüpft werden.

Sei der Inhalt des Registers r16 = 0b11001100 und der Inhalt des Registers r17 = 0b01011010, so lautet die Verknüpfung **eor r16, r17**

```
 0b11001100
 0b01011010     xor
-----------
 0b10010110
```

Das Ergebnis wird im ersten Operanden (r16) abgelegt.

Im Ergebnis werden diejenigen Bits umgedreht, an deren Bitposition in der Maske eine 1 vorhanden ist.

Da in der Maske

```
0b01011010
```

an den Bitpositionen 1, 3, 4, 6 jeweils eine 1 steht, enthält das Ergebnis an eben diesen Bitpositionen die umgedrehten Bits aus der Ursprungszahl. Alle anderen Bits werden so wie sie sind aus der Ursprungszahl übernommen.

Weblinks

- GATE: The Logic Game - Eine kleine Abwechselung für die Tutorialpause. (ab Windows XP, Mac OS X)

AVR-Tutorial: Arithmetik8

Eine der Hauptaufgaben eines Mikrokontrollers bzw. eines Computers allgemein, ist es, irgendwelche Berechnungen anzustellen. Der Löwenanteil an den meisten Berechnungen entfällt dabei auf einfache Additionen bzw. Subtraktionen. Multiplikationen bzw. Divisionen kommen schon seltener vor, bzw. können oft durch entsprechende Additionen bzw. Subtraktionen ersetzt werden. Weitergehende mathematische Konstrukte werden zwar auch ab und an benötigt, können aber in der Assemblerprogrammierung durch geschickte Umformungen oft vermieden werden.

Inhaltsverzeichnis

- 1 Hardwareunterstützung
- 2 8 Bit versus 16 Bit
- 3 8-Bit Arithmetik ohne Berücksichtigung eines Vorzeichens
  - 3.1 Die Umwandlung von Binär in das Dezimalystem
  - 3.2 Die Umwandlung von Dezimal in das Binärsystem
- 4 8-Bit Arithmetik mit Berücksichtigung eines Vorzeichens
  - 4.1 Problem der Kodierung des Vorzeichens
  - 4.2 2-er Komplement
- 5 Arithmetikflags
  - 5.1 Carry
  - 5.2 Signed- und Overflowflag
  - 5.3 Weitere Flags
  - 5.4 Übersicht über die arithmetischen Flags
- 6 Inkrementieren / Dekrementieren
  - 6.1 AVR-Befehle
- 7 Addition
  - 7.1 AVR-Befehle
- 8 Subtraktion
  - 8.1 AVR-Befehle
- 9 Multiplikation
  - 9.1 Hardwaremultiplikation
  - 9.2 Multiplikation in Software
- 10 Division
  - 10.1 Division in Software
- 11 Arithmetik mit mehr als 8 Bit

Hardwareunterstützung

Praktisch alle Mikroprozessoren unterstützen Addition und Subtraktion direkt in Hardware, das heißt: Sie haben eigene Befehle dafür. Einige bringen auch Unterstützung für eine Hardwaremultiplikation mit (so zum Beispiel der ATmega8), während Division in Hardware schon seltener zu finden ist.

8 Bit versus 16 Bit

In diesem Abschnitt des Tutorials wird gezielt auf 8 Bit Arithmetik eingegangen, um zunächst die Grundlagen des Rechnens mit einem µC zu zeigen. Die Erweiterung von 8 Bit auf 16 Bit Arithmetik ist in einigen Fällen wie Addition und Subtraktion trivial, kann sich aber bei Multiplikation und Division in einem beträchtlichen Codezuwachs niederschlagen.

Der im Tutorial verwendete ATmega8 besitzt eine sog. 8-Bit Architektur. Das heißt, dass seine Rechenregister (mit Ausnahmen) nur 8 Bit breit sind und sich daher eine 8-Bit Arithmetik als die natürliche Form der Rechnerei auf diesem Prozessor anbietet. Berechnungen, die mehr als 8 Bit erfordern, müssen dann durch Kombinationen von Rechenvorgängen realisiert werden. Eine Analogie wäre z. B. das Rechnen, wie wir alle es in der Grundschule gelernt haben. Auch wenn wir in der Grundschule (in den Anfängen) nur die Additionen mit Zahlen kleiner als 10 auswendig gelernt haben, so können wir dennoch durch die Kombination von mehreren derartigen Additionen beliebig große Zahlen addieren. Das gleiche gilt für Multiplikationen. In der Grundschule musste wohl jeder von uns das 'Kleine Einmaleins' auswendig lernen, um Multiplikationen im Zahlenraum bis 100 quasi 'in Hardware' zu berechnen. Und doch können wir durch Kombinationen solcher Einfachmultiplikationen und zusätzlichen Additionen in beliebig große Zahlenräume vorstoßen.

Die Einschränkung auf 8 Bit ist also keineswegs eine Einschränkung in dem Sinne, dass es eine prinzipielle Obergrenze für Berechnungen gäbe. Sie bedeutet lediglich eine obere Grenze dafür, bis zu welchen Zahlen in einem Rutsch gerechnet werden kann. Alles, was darüber hinausgeht, muss dann mittels Kombinationen von Berechnungen gemacht werden.

8-Bit Arithmetik ohne Berücksichtigung eines Vorzeichens

Die Bits des Registers besitzen dabei eine Wertigkeit, die sich aus der Stelle des Bits im Byte ergibt. Dies ist völlig analog zu dem uns vertrauten Dezimalsystem. Auch dort besitzt eine Ziffer in einer Zahl eine bestimmte Wertigkeit, je nach dem, an welcher Position diese Ziffer in der Zahl auftaucht. So hat z. B. die Ziffer 1 in der Zahl 12 die Wertigkeit 'Zehn', während sie in der Zahl 134 die Wertigkeit 'Hundert' besitzt. Und so wie im Dezimalsystem die Wertigkeit einer Stelle immer das Zehnfache der Wertigkeit der Stelle unmittelbar rechts von ihr ist, so ist im Binärsystem die Wertigkeit einer Stelle immer das 2-fache der Stelle rechts von ihr.

Die Zahl 4632 im Dezimalsystem kann also so aufgefasst werden:

```
4632  =     4 * 1000          ( 1000 = 10 hoch 3 )
        +  6 * 100           (  100 = 10 hoch 2 )
        +  3 * 10            (   10 = 10 hoch 1 )
        +  2 * 1             (    1 = 10 hoch 0 )
```

Die Umwandlung von Binär in das Dezimalystem

Völlig analog ergibt sich daher folgendes für z. B. die 8 Bit Binärzahl **0b10011011** (um Binärzahlen von Dezimalzahlen zu unterscheiden, wird ein **0b** vorangestellt):

```
0b10011011     =    1 * 128    ( 128 = 2 hoch 7 )
                  +  0 * 64     (  64 = 2 hoch 6 )
                  +  0 * 32     (  32 = 2 hoch 5 )
                  +  1 * 16     (  16 = 2 hoch 4 )
                  +  1 * 8      (   8 = 2 hoch 3 )
                  +  0 * 4      (   4 = 2 hoch 2 )
                  +  1 * 2      (   2 = 2 hoch 1 )
                  +  1 * 1      (   1 = 2 hoch 0 )
```

Ausgerechnet (um die entsprechende Dezimalzahl zu erhalten) ergibt das 128 + 16 + 8 + 2 + 1 = 155. Die Binärzahl **0b10011011** entspricht also der Dezimalzahl **155**. Es ist wichtig, sich klar zu machen, dass es zwischen Binär- und Dezimalzahlen keinen grundsätzlichen Unterschied gibt. Beides sind nur verschiedene Schreibweisen für das Gleiche: Eine Zahl. Während wir Menschen an das Dezimalsystem gewöhnt sind, ist das Binärsystem für einen Computer geeigneter, da es nur aus den 2 Ziffern 0 und 1 besteht, welche sich leicht in einem Computer darstellen lassen (Spannung, keine Spannung).

Welches ist nun die größte Zahl, die mit 8 Bit dargestellt werden kann? Dabei handelt es sich offensichtlich um die Zahl **0b11111111**. In Dezimalschreibweise wäre das die Zahl

```
0b11111111   =   1  *  128
               + 1  *  64
               + 1  *  32
               + 1  *  16
               + 1  *  8
               + 1  *  4
               + 1  *  2
               + 1  *  1
```

oder ausgerechnet: **255**

Wird also mit 8 Bit Arithmetik betrieben, wobei alle 8 Bit als signifikante Ziffern benutzt werden (also kein Vorzeichenbit, dazu später mehr), so kann damit im Zahlenraum **0** bis **255** gerechnet werden.

Binär	Dezimal	Binär	Dezimal
0b00000000	0	0b10000000	128
0b00000001	1	0b10000001	129
0b00000010	2	0b10000010	130
0b00000011	3	0b10000011	131
0b00000100	4	0b10000100	132
0b00000101	5	0b10000101	133
...		...	
0b01111100	124	0b11111100	252
0b01111101	125	0b11111101	253
0b01111110	126	0b11111110	254
0b01111111	127	0b11111111	255

Die Umwandlung von Dezimal in das Binärsystem

Aus dem vorhergehenden ergibt sich völlig zwanglos die Vorschrift, wie Binärzahlen ins Dezimalsystem umgewandelt werden können (nicht vergessen: Die Zahl selber wird ja gar nicht verändert. Binär- und Dezimalsystem sind ja nur verschiedene Schreibweisen): Durch Anwendung der Vorschrift, wie denn eigentlich ein Stellenwertsystem aufgebaut ist. Aber wie macht man den umgekehrten Schritt, die Wandlung vom Dezimal ins Binärsystem. Der Weg führt über die Umkehrung des vorhergehenden Prinzips. Fortgesetzte Division durch 2

Es sei die Zahl 92 ins Binärsystem zu wandeln.

```
92 / 2   =   46   Rest 0
46 / 2   =   23   Rest 0
23 / 2   =   11   Rest 1
11 / 2   =    5   Rest 1
 5 / 2   =    2   Rest 1
 2 / 2   =    1   Rest 0
 1 / 2   =    0   Rest 1
```

Die Division wird solange durchgeführt, bis sich ein Divisionergebnis von 0 ergibt. Die Reste, von unten nach oben gelesen, ergeben dann die Binärzahl. Die zu 92 gehörende Binärzahl lautet also 1011100. Es wird noch eine führende 0 ergänzt um sie auf die standardmässigen 8-Bit zu bringen: **0b01011100**.

8-Bit Arithmetik mit Berücksichtigung eines Vorzeichens

Soll mit Vorzeichen (also positiven und negativen Zahlen) gerechnet werden, so erhebt sich die Frage: Wie werden eigentlich positive bzw. negative Zahlen dargestellt? Alles was wir haben sind ja 8 Bit in einem Byte.

Problem der Kodierung des Vorzeichens

Die Lösung des Problems besteht darin, dass ein Bit zur Anzeige des Vorzeichens benutzt wird. Im Regelfall wird dazu das am weitesten links stehende Bit benutzt. Von den verschiedenen Möglichkeiten, die sich hiermit bieten, wird in der Praxis fast ausschließlich mit dem sog. 2-er Komplement gearbeitet, da es Vorteile bei der Addition bzw. Subtraktion von Zahlen bringt. In diesem Fall muß nämlich das Vorzeichen einer Zahl überhaupt nicht berücksichtigt werden. Durch die Art und Weise der Bildung von negativen Zahlen kommt am Ende das Ergebnis mit dem korrekten Vorzeichen heraus.

2-er Komplement

Das 2-er Komplement verwendet das höchstwertige Bit eines Byte, das sog. MSB (= **M**ost **S**ignificant **B**it) zur Anzeige des Vorzeichens. Ist dieses Bit 0, so ist die Zahl positiv. Ist es 1, so handelt es sich um eine negative Zahl. Die 8-Bit Kombination **0b10010011** stellt also eine negative Zahl dar, **wenn und nur wenn diese Bitkombination überhaupt als vorzeichenbehaftete Zahl aufgefasst werden soll**. Anhand der Bitkombination alleine ist es also nicht möglich, eine definitive Aussage zu treffen, ob es sich um eine vorzeichenbehaftete Zahl handelt oder nicht. Erst wenn durch den Zusammenhang klar ist, dass man es mit vorzeichenbehafteten Zahlen zu tun hat, bekommt das MSB die Sonderbedeutung des Vorzeichens.

Um bei einer Zahl das Vorzeichen zu wechseln, geht man wie folgt vor:

- Zunächst wird das 1-er Komplement gebildet, indem alle Bits umgedreht werden. Aus 0 wird 1 und aus 1 wird 0
- Danach wird aus diesem Zwischenergebnis das 2-er Komplement gebildet, indem noch 1 addiert wird.

Diese Vorschrift kann immer dann benutzt werden, wenn das Vorzeichen einer Zahl gewechselt werden soll. Er macht aus positiven Zahlen negative und aus negativen Zahlen positive.

Beispiel: Es soll die Binärdarstellung für -92 gebildet werden. Dazu benötigt man zunächst die Binärdarstellung für +92, welche **0b01011100** lautet. Diese wird jetzt nach der Vorschrift für 2-er Komplemente negiert und damit negativ gemacht.

```
0b01011100            Ausgangszahl
0b10100011            1-er Komplement, alle Bits umdrehen
0b10100100            noch 1 addieren
```

Die Binärdarstellung für -92 lautet also **0b10100100**. Das gesetzte MSB weist diese Binärzahl auch tatsächlich als negative Zahl aus.

Beispiel: Gegeben sei die Binärzahl **0b00111000**, welche als vorzeichenbehaftete Zahl anzusehen ist. Welcher Dezimalzahl entspricht diese Binärzahl?

Da das MSB nicht gesetzt ist, handelt es sich um eine positive Zahl und die Umrechnung kann wie im Fall der vorzeichenlosen 8-Bit Zahlen erfolgen. Das Ergebnis lautet also +56 (= 0 * 128 + 0 * 64 + 1 * 32 + 1 * 16 + 1 * 8 + 0 * 4 + 0 * 2 + 0 * 1)

Beispiel: Gegeben sei die Binärzahl **0b10011001**, welche als vorzeichenbehaftete Zahl anzusehen ist. Welcher Dezimalzahl entspricht diese Binärzahl?

Da das MSB gesetzt ist, handelt es sich um eine negative Zahl. Daher wird diese Zahl zunächst negiert um dadurch eine positive Zahl zu erhalten.

```
0b10011001       Originalzahl
0b01100110       1-er Komplement, alle Bits umdrehen
0b01100111       2-er Komplement, noch 1 addiert
```

Die zu 0b10011001 gehörende positive Binärzahl lautet also 0b01100111. Da es sich um eine positive Zahl handelt, kann sie wiederum ganz normal, wie vorzeichenlose Zahlen, in eine Dezimalzahl umgerechnet werden. Das Ergebnis lautet 103 (= 0 * 128 + 1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1). Da aber von einer negativen Zahl ausgegangen wurde, ist **0b10011001** die binäre Darstellung der Dezimalzahl -103.

Beispiel: Gegeben sei dieselbe Binärzahl **0b10011001**. Aber diesmal sei sie als vorzeichenlose Zahl aufzufassen. Welcher Dezimalzahl entspricht diese Binärzahl?

Da die Binärzahl als vorzeichenlose Zahl aufzufassen ist, hat das MSB keine spezielle Bedeutung. Die Umrechnung erfolgt also ganz normal: 0b10011001 = 1 * 128 + 0 * 64 + 0 * 32 + 1 * 16 + 1 * 8 + 0 * 4 + 0 * 2 + 1 * 1 = 153.

Beispiel: Wie lautet die Binärzahl zu -74?

Da es sich hier offensichtlich im eine vorzeichenbehaftete Zahl handelt, müssen die Regeln des 2-er Komplemnts angewendet werden. Zunächst ist also die Binärrepräsentierung von +74 zu bestimmen, welche dann durch Anwendung des 2-er Komplements negiert wird.

```
 74 / 2 = 37  Rest 0
 37 / 2 = 18  Rest 1
 18 / 2 =  9  Rest 0
  9 / 2 =  4  Rest 1
  4 / 2 =  2  Rest 0
  2 / 2 =  1  Rest 0
  1 / 2 =  0  Rest 1
```

Die Binärdarstellung für +74 lautet daher **0b01001010**

```
   0b01001010     +74
   0b10110101     1-er Komplement, alle Bits umdrehen
   0b10110110     noch 1 addieren
```

Die Binärdarstellung für -74 lautet daher **0b10110110**

Arithmetikflags

Im Statusregister des Prozessors gibt es eine Reihe von Flags, die durch Rechenergebnisse beeinflusst werden, bzw. in Berechnungen einfließen können.

Carry

Das Carry-Flag **C** zeigt an, ob bei einer Berechnung mit vorzeichenlosen Zahlen ein Über- oder Unterlauf erfolgt ist, d.h. das Ergebnis der Berechnung liegt außerhalb des darstellbaren Bereiches 0...255.

Wie das?

Angenommen es müssen zwei 8-Bit-Zahlen addiert werden.

```
   10100011
 + 11110011
  ---------
  110010110
```

Das Ergebnis der Addition umfasst neun Bit und liegt außerhalb des in einem Register darstellbaren Zahlenbereiches 0...255; die Addition *ist übergelaufen*.

Werden dieselben Zahlen subtrahiert,

```
   10100011
 - 11110011
  ---------
   10110000
```

verbleibt an der höchstwertigsten Stelle ein "geborgtes" Bit, welches durch das Carry angezeigt wird.

Signed- und Overflowflag

Wird mit vorzeichenbehafteten Zahlen gerechnet, so wird das Verlassen des 8-Bit-Zahlenbereiches

-128...+127 durch die Flags **S** und **V** angezeigt. Der Prozessor selbst unterscheidet nicht zwischen vorzeichenlosen und -behafteten Zahlen. Der Programmierer legt durch Wahl der ausgewerteten Flags (C bei vorzeichenlosen bzw. S/V bei vorzeichenbehafteten) die Interpretation der Zahlen fest.

Weitere Flags

Das Zero-Flag **Z** wird gesetzt, wenn das 8-Bit-Ergebnis der Berechnung null ist. Dies kann bei der Addition auch durch Überlauf geschehen, was durch ein zusätzliches Carryflag angezeigt wird.

Das Negative-Flag **N** wird gesetzt, wenn im Ergebnis das höchstwertige Bit gesetzt ist. Bei vorzeichenbehafteter Arithmetik ist dies als negative Zahl zu interpretieren, sofern nicht durch das V-Flag ein Verlassen des Zahlbereichs angezeigt wird.

Das Half-Carry-Flag **H** zeigt, analog zum Carry-Flag, einen Übertrag zwischen Bit 3 und 4 an. Dies kann in speziellen Anwendungen (z. B. zwei simultane Vier-Bit-Berechnungen mit einem Befehl) nützlich sein.

Übersicht über die arithmetischen Flags

Ergebnis des Befehls **ADD Rd, Rr**

```
         Rr|    0 |    1 |   64 |  127 |  128 |  129 |  192 |  255
 Rd        |(  +0)|(  +1)|( +64)|(+127)|(-128)|(-127)|( -64)|(  -1)
----------+------+------+------+------+------+------+------+------
  0 (  +0)|    Z |      |      |      |S N   |S N   |S N   |S N
  1 (  +1)|      |      |      | VN   |S N   |S N   |S N   |   CZ
 64 ( +64)|      |      | VN   | VN   |S N   |S N   |   CZ |   C
127 (+127)|      | VN   | VN   | VN   |S N   |   CZ |   C  |   C
128 (-128)|S N   |S N   |S N   |S N   |SV CZ |SV C  |SV C  |SV C
129 (-127)|S N   |S N   |S N   |   CZ |SV C  |SV C  |SV C  |S NC
192 ( -64)|S N   |S N   |   CZ |   C  |SV C  |SV C  |S NC  |S NC
255 (  -1)|S N   |   CZ |   C  |   C  |SV C  |S NC  |S NC  |S NC
```

Man erkennt: C=1 genau dann wenn die Addition Rd + Rr mit vorzeichenlosen Zahlen überläuft. V=1 genau dann wenn die Addition mit vorzeichenbehafteten Zahlen überläuft.

Ergebnis des Befehls **SUB Rd, Rr** bzw. **CP Rd, Rr**

```
         Rr|    0 |   63 |   64 |  127 |  128 |  191 |  192 |  255
 Rd        |(  +0)|( +63)|( +64)|(+127)|(-128)|( -65)|( -64)|(  -1)
----------+------+------+------+------+------+------+------+------
  0 (  +0)|    Z |S NC  |S NC  |S NC  | VNC  |   C  |   C  |   C
 63 ( +63)|      |    Z |S NC  |S NC  | VNC  | VNC  |   C  |   C
 64 ( +64)|      |      |    Z |S NC  | VNC  | VNC  | VNC  |   C
127 (+127)|      |      |      |    Z | VNC  | VNC  | VNC  | VNC
128 (-128)|S N   |SV    |SV    |SV    |    Z |S NC  |S NC  |S NC
191 ( -65)|S N   |S N   |SV    |SV    |      |    Z |S NC  |S NC
192 ( -64)|S N   |S N   |S N   |SV    |      |      |    Z |S NC
255 (  -1)|S N   |S N   |S N   |S N   |      |      |      |    Z
```

Man erkennt: C=1 genau dann wenn die Subtraktion Rd - Rr mit vorzeichenlosen Zahlen unterläuft; äquivalent dazu ist Rd < Rr (vorzeichenlos). S=1 genau dann wenn die Subtraktion mit vorzeichenbehafteten Zahlen unterläuft bzw. Rd > Rr (vorzeichenbehaftet).

Inkrementieren / Dekrementieren

Erstaunlich viele Operationen in einem Computer-Programm entfallen auf die Operationen 'Zu einer Zahl 1 addieren' bzw. 'Von einer Zahl 1 subtrahieren'. Dementsprechend enthalten die meisten Mikroprozessoren die Operationen *Inkrementieren* (um 1 erhöhen) bzw. *Dekrementieren* (um 1 verringern) als eigenständigen Assemblerbefehl. So auch der ATmega8.

AVR-Befehle

```
inc  r16
```

bzw.

```
dec  r16
```

Die Operation ist einfach zu verstehen. Das jeweils angegebene Register (hier wieder am Register r16 gezeigt) wird um 1 erhöht bzw. um 1 verringert. Dabei wird die Zahl im Register als vorzeichenlose Zahl angesehen. Enthält das Register bereits die größtmögliche Zahl (0b11111111 oder dezimal 255), so erzeugt ein weiteres Inkrementieren die kleinstmögliche Zahl (0b00000000 oder dezimal 0) bzw. umgekehrt dekrementiert 0 zu 255.

Addition

Auf einem Mega8 gibt es nur eine Möglichkeit, um eine Addition durchzuführen: Die beiden zu addierenden Zahlen müssen in zwei Registern stehen.

AVR-Befehle

```
add  r16, r17      ; Addition der Register r16 und r17. Das Ergebnis wird
                   ; im Register r16 abgelegt
adc  r16, r17      ; Addition der Register r16 und r17, wobei das Carry-Bit
                   ; noch zusätzlich mit addiert wird.
```

Bei der Addition zweier Register wird ein möglicher Überlauf in allen Fällen im Carry Bit abgelegt. Daraus erklärt sich dann auch das Vorhandensein eines Additionsbefehls, der das Carry-Bit noch zusätzlich mitaddiert: Man benötigt ihn zum Aufbau einer Addition die mehr als 8 Bit umfasst. Die niederwertigsten Bytes werden mit einem **add** addiert und alle weiteren höherwertigen Bytes werden, vom Niederwertigsten zum Höchstwertigsten, mittels **adc** addiert. Dadurch werden eventuelle Überträge automatisch berücksichtigt.

Subtraktion

Subtraktionen können auf einem AVR in zwei unterschiedlichen Arten ausgeführt werden. Entweder es werden zwei Register voneinander subtrahiert oder es wird von einem Register eine konstante Zahl abgezogen. Beide Varianten gibt es wiederum in den Ausführungen mit und ohne Berücksichtigung des Carry Flags

AVR-Befehle

```
     sub  r16, r17      ; Subtraktion des Registers r17 von r16. Das Ergebnis
wird
                        ; im Register r16 abgelegt
     sbc  r16, r17      ; Subtraktion des Registers r17 von r16, wobei das
Carry-Bit
                        ; noch zusätzlich mit subtrahiert wird. Das Ergebnis
wird
                        ; im Register r16 abgelegt
     subi r16, zahl     ; Die Zahl (als Konstante) wird vom Register r16
subtrahiert.
                        ; Das Ergebnis wird im Register r16 abgelegt
     sbci r16, zahl     ; Subtraktion einer konstanten Zahl vom Register r16,
wobei
                        ; zusätzlich noch das Carry-Bit mit subtrahiert wird.
                        ; Das Ergebnis wird im Register r16 abgelegt.
```

Multiplikation

Multiplikation kann auf einem AVR je nach konkretem Typ auf zwei unterschiedliche Arten ausgeführt werden. Während die größeren ATMega Prozessoren über einen Hardwaremultiplizierer verfügen, ist dieser bei den kleineren Tiny Prozessoren nicht vorhanden. Hier muß die Multiplikation quasi zu Fuß durch entsprechende Addition von Teilresultaten erfolgen.

Hardwaremultiplikation

Vorzeichenbehaftete und vorzeichenlose Zahlen werden unterschiedlich multipliziert. Denn im Falle eines Vorzeichens darf ein gesetztes 7. Bit natürlich nicht in die eigentliche Berechnung mit einbezogen werden. Statt dessen steuert dieses Bit (eigentlich die beiden MSB der beiden beteiligten Zahlen) das Vorzeichen des Ergebnisses. Die Hardwaremultiplikation ist auch dahingehend eingeschränkt, dass das Ergebnis einer Multiplikation immer in den Registerpärchen *r0* und *r1* zu finden ist. Dabei steht das LowByte (also die unteren 8 Bit) des Ergebnisses in *r0* und das HighByte in *r1*.

AVR-Befehl

```
    mul   r16, r17      ; multipliziert r16 mit r17. Beide Registerinhalte
werden
                        ; als vorzeichenlose Zahlen aufgefasst.
                        ; Das Ergebnis der Multiplikation ist in den Registern
r0 und r1
                        ; zu finden.

    muls  r16, r17      ; multipliziert r16 mit r17. Beide Registerinhalte
werden
                        ; als vorzeichenbehaftete Zahlen aufgefasst.
                        ; Das Ergebnis der Multiplikation ist in den Registern
r0 und r1
                        ; zu finden und stellt ebenfalls eine
vorzeichenbehaftete
                        ; Zahl dar.

    mulsu r16, r17      ; multipliziert r16 mit r17, wobei r16 als
vorzeichenbehaftete
```

```
                        ; Zahl aufgefasst wird und r17 als vorzeichenlose Zahl.
                        ; Das Ergebnis der Multiplikation ist in den Registern
r0 und r1
                        ; zu finden und stellt eine vorzeichenbehaftete Zahl
dar.
```

Multiplikation in Software

Multiplikation in Software ist nicht weiter schwierig. Man erinnere sich daran, wie Multiplikationen in der Grundschule gelehrt wurden: Zunächst stand da das kleine Einmal-Eins, welches auswendig gelernt wurde. Mit diesen Kenntnissen konnten dann auch größere Multiplikationen angegangen werden, indem der Multiplikand mit jeweils einer Stelle des Multiplikators multipliziert wurde und die Zwischenergebnisse, geeignet verschoben, addiert wurden. Die Verschiebung um eine Stelle entspricht dabei einer Multiplikation mit 10.

Beispiel: Zu multiplizieren sei 3456 * 7812

```
      3456     * 7812
      ---------------
     24192     <-+|||
 +    27648    <--+||
 +      3456   <---+|
 +       6912  <----+
     --------
     26998272
```

Im Binärsystem funktioniert Multiplikation völlig analog. Nur ist hier das kleine Einmaleins sehr viel einfacher! Es gibt nur 4 Multiplikationen (anstatt 100 im Dezimalsystem):

```
   0 * 0   = 0
   0 * 1   = 0
   1 * 0   = 0
   1 * 1   = 1
```

Es gibt lediglich einen kleinen Unterschied gegenüber dem Dezimalsystem: Anstatt zunächst alle Zwischenergebnisse aufzulisten und erst danach die Summe zu bestimmen, werden wir ein neues Zwischenergebnis gleich in die Summe einrechnen. Dies deshalb, da Additionen von mehreren Zahlen im Binärsystem im Kopf sehr leicht zu Flüchtigkeitsfehlern führen (durch die vielen 0-en und 1-en). Weiters wird eine einfache Tatsache benutzt: 1 mal eine Zahl ergibt wieder die Zahl, während 0 mal eine Zahl immer 0 ergibt. Dadurch braucht man im Grunde bei einer Multiplikation überhaupt nicht zu multiplizieren, sondern eigentlich nur die Entscheidung treffen: Muss die Zahl geeignet verschoben addiert werden oder nicht?

```
   0b00100011         * 0b10001001
   ---------------------------------
     00100011          <--+||||||
+     00000000         <---+||||||
     ---------              ||||||
     001000110              ||||||
+      00000000        <----+|||||
     ----------              |||||
     0010001100              |||||
+       00000000       <-----+||||
     -----------              ||||
     00100011000              ||||
+        00100011      <------+|||
```

```
     ------------              |||
     001001010011              |||
+         00000000     <-------+||
     -------------              ||
     0010010100110              ||
+          00000000    <--------+|
     --------------             |
     00100101001100             |
+           00100011   <---------+
     ---------------
     001001010111011
```

Man sieht auch, wie bei der Multiplikation zweier 8 Bit Zahlen sehr schnell ein 16 Bit Ergebnis entsteht. Dies ist auch der Grund, warum die Hardwaremultiplikation immer 2 Register zur Aufnahme des Ergebnisses benötigt.

Ein Assembler Code, der diese Strategie im Wesentlichen verwirklicht, sieht z. B. so aus. Dieser Code wurde nicht auf optimale Laufzeit getrimmt, sondern es soll im Wesentlichen eine 1:1 Umsetzung des oben gezeigten Schemas sein. Einige der verwendeten Befehle wurden im Rahmen dieses Tutorials an dieser Stelle noch nicht besprochen. Speziell die Schiebe- (**lsl**) und Rotier- (**rol**) Befehle sollten in der AVR Befehlsübersicht genau studiert werden, um ihr Zusammenspiel mit dem Carry Flag zu verstehen. Nur soviel als Hinweis: Das Carry Flag dient in der **lsl** / **rol** Sequenz als eine Art Zwischenspeicher, um das höherwertigste Bit aus dem Register r0 beim Verschieben in das Register r1 verschieben zu können. Der **lsl** verschiebt alle Bits des Registers um 1 Stelle nach links, wobei das vorhergehende MSB ins Carry Bit wandert und rechts ein 0-Bit nachrückt. Der **rol** verschiebt ebenfalls alle Stellen eines Registers um 1 Stelle nach links. Diesmal wird aber rechts nicht mit einem 0-Bit aufgefüllt, sondern an dieser Stelle wird der momentane Inhalt des Carry Bits eingesetzt.

```
    ldi  r16, 0b00100011  ; Multiplikator
    ldi  r17, 0b10001001  ; Multiplikand
                          ; Berechne r16 * r17

    ldi  r18, 8          ; 8 mal verschieben und gegebenenfalls addieren
    clr  r19             ; 0 wird für die 16 Bit Addition benötigt
    clr  r0              ; Ergebnis Low Byte auf 0 setzen
    clr  r1              ; Ergebnis High Byte auf 0 setzen

mult:
    lsl  r0              ; r1:r0 einmal nach links verschieben
    rol  r1
    lsl  r17             ; Das MSB von r17 ins Carry schieben
    brcc noadd           ; Ist dieses MSB (jetzt im Carry) eine 1?
    add  r0,r16          ; Wenn ja, dann r16 zum Ergebnis addieren
    adc  r1,r19

noadd:
    dec  r18             ; Wurden alle 8 Bit von r17 abgearbeitet?
    brne mult            ; Wenn nicht, dann ein erneuter Verschiebe/Addier
Zyklus

                         ; r0 enthält an dieser Stelle den Wert 0b10111011
                         ; r1 enthält 0b00010010
                         ; Gemeinsam bilden r1 und r0 also die Zahl
                         ; 0b0001001010111011
```

Division

Anders als bei der Multiplikation, gibt es auch auf einem ATMega-Prozessor keine hardwaremässige Divisionseinheit. Divisionen müssen also in jedem Fall mit einer speziellen Routine, die im wesentlichen auf Subtraktionen beruht, erledigt werden.

Division in Software

Um die Vorgangsweise bei der binären Division zu verstehen, wollen wir wieder zunächst anhand der gewohnten dezimalen Division untersuchen wie sowas abläuft.

Angenommen es soll dividiert werden: 938 / 4 (938 ist der Dividend, 4 ist der Divisor)

Wie haben Sie es in der Grundschule gelernt? Wahrscheinlich so wie der Autor auch:

```
 938 : 4 = 234
---
-8
----
 1
 13
-12
 ---
  1
  18
 -16
  --
   2 Rest
```

Der Vorgang war: Man nimmt die erste Stelle des Dividenden (9) und ruft seine gespeicherte Einmaleins Tabelle ab, um festzustellen, wie oft der Divisor in dieser Stelle enthalten ist. In diesem konkreten Fall ist die erste Stelle 9 und der Divisor 4. 4 ist in 9 zweimal enthalten. Also ist 2 die erste Ziffer des Ergebnisses. 2 mal 4 ergibt aber 8 und diese 8 werden von den 9 abgezogen, übrig bleibt 1. Aus dem Dividenden wird die nächste Ziffer (3) heruntergezogen und man erhält mit der 1 aus dem vorhergehenden Schritt 13. Wieder dasselbe Spiel: Wie oft ist 4 in 13 enthalten? 3 mal (3 ist die nächste Ziffer des Ergebnisses) und 3 * 4 ergibt 12. Diese 12 von den 13 abgezogen macht 1. Zu dieser 1 gesellt sich wieder die nächste Ziffer des Dividenden, 8, um so 18 zu bilden. Wie oft ist 4 in 18 enthalten? 4 mal (4 ist die nächste Ziffer des Ergebnisses), denn 4 mal 4 macht 16, und das von den 18 abgezogen ergibt 2. Da es keine nächste Ziffer im Dividenden mehr gibt, lautet also das Resultat: 938 : 4 ergibt 234 und es bleiben 2 Rest.

Die binäre Division funktioniert dazu völlig analog. Es gibt nur einen kleinen Unterschied, der einem sogar das Leben leichter macht. Es geht um den Schritt: Wie oft ist x in y enthalten? Dieser Schritt ist in der binären Division besonders einfach, da das Ergebnis dieser Fragestellung nur 0 oder 1 sein kann. Das bedeutet aber auch: Entweder ist der Divisior in der zu untersuchenden Zahl enthalten, oder er ist es nicht. Das kann aber ganz leicht entschieden werden: Ist die Zahl größer oder gleich dem Divisior, dann ist der Divisor enthalten und zum Ergebnis kann eine 1 hinzugefügt werden. Ist die Zahl kleiner als der Divisior, dann ist der Divisior nicht enthalten und die nächste Ziffer des Ergebnisses ist eine 0.

Beispiel: Es soll die Division 0b01101100 : 0b00001001 ausgeführt werden.

Es wird wieder mit der ersten Stelle begonnen und die oben ausgeführte Vorschrift angewandt.

```
  0b01101101 : 0b00001001 = 0b00001100
                             ^^^^^^^^
                             ||||||||
    0                ---------+|||||||   1001 ist in 0 0-mal enthalten
   -0                          |||||||
   --                          |||||||
    0                          |||||||
    01               ----------+||||||   1001 ist in 1 0-mal enthalten
   - 0                           ||||||
    --                           ||||||
    01                           ||||||
    011              -----------+|||||   1001 ist in 11 0-mal enthalten
   -  0                           |||||
    ---                           |||||
    011                           |||||
    0110             ------------+||||   1001 ist in 110 0-mal enthalten
   -   0                           ||||
    ----                           ||||
    0110                           ||||
    01101            -------------+|||   1001 ist in 1101 1-mal enthalten
   - 1001                           |||
    -----                           |||
     0100                           |||
     01001           --------------+||   1001 ist in 1001 1-mal enthalten
    - 1001                           ||
     -----                           ||
     00000                           ||
     000000          ---------------+|   1001 ist in 0 0-mal enthalten
   -      0                          |
     ------                          |
     0000001         ----------------+   1001 ist in 1 0-mal enthalten
    -      0
     -------
           1 Rest
```

Die Division liefert also das Ergebnis 0b00001100, wobei ein Rest von 1 bleibt. Der Dividend 0b01101101 entspricht der Dezimalzahl 109, der Divisor 0b00001001 der Dezimalzahl 9. Und wie man sich mit einem Taschenrechner leicht überzeugen kann, ergibt die Division von 109 durch 9 einen Wert von 12, wobei 1 Rest bleibt. Die Binärzahl für 12 lautet 0b00001100, das Ergebnis stimmt also.

```
    ldi  r16, 109   ; Dividend
    ldi  r17,   9   ; Divisor

                    ; Division r16 : r17

    ldi  r18,   8   ; 8 Bit Division
    clr  r19        ; Register für die Zwischenergebnisse / Rest
    clr  r20        ; Ergebnis

divloop:
    lsl  r16        ; Zwischenergebnis mal 2 nehmen und das
    rol  r19        ; nächste Bit des Dividenden anhängen

    lsl  r20        ; das Ergebnis auf jeden Fall mal 2 nehmen,
                    ; das hängt effektiv eine 0 an das Ergebnis an.
                    ; Sollte das nächste Ergebnis-Bit 1 sein, dann wird
                    ; diese 0 in Folge durch eine 1 ausgetauscht

    cp   r19, r17   ; ist der Divisor größer?
    brlo div_zero   ; wenn nein, dann bleibt die 0
```

```
    sbr  r20, 1     ; wenn ja, dann jetzt die 0 durch eine 1 austauschen ...
    sub  r19, r17   ; ... und den Divisor abziehen

div_zero:
    dec  r18        ; das Ganze 8 mal wiederholen
    brne divloop

                    ; in r20 steht das Ergebnis der Division
                    ; in r19 steht der bei der Division entstehende Rest
```

Arithmetik mit mehr als 8 Bit

Es gibt eine Sammlung von Algorithmen zur AVR-Arithmetik mit mehr als 8 Bit, deren Grundprinzipien im wesentlichen identisch zu den in diesem Teil ausgeführten Prinzipien sind.

AVR-Tutorial: Stack

Inhaltsverzeichnis

- 1 Motivation
- 2 Aufruf von Unterprogrammen
- 3 Sichern von Registern
- 4 Sprung zu beliebiger Adresse
- 5 Weitere Informationen (von Lothar Müller):

Motivation

Bisher war es so, dass wenn die Programmausführung an einer anderen Stelle fortgesetzt werden soll, als sich durch die Abfolge der Befehle ergibt, mittels **rjmp** an diese andere Stelle gesprungen wurde. Das ist aber oft nicht ausreichend. Oft möchte man den Fall haben, dass man aus der normalen Befehlsreihenfolge heraus eine andere Sequenz von Befehlen ausgeführt wird und wenn diese abgearbeitet ist, genau an die Aufrufstelle zurückgesprungen wird. Da diese eingeschobene Sequenz an vielen Stellen aufrufbar sein soll, gelingt es daher auch nicht, mittels eines **rjmp** wieder zur aufrufenden Stelle zurück zu kommen, denn dann müsste ja dieser Rücksprung je nachdem von wo der Hinsprung gekommen ist entsprechend modifiziert werden.

Die Lösung dieses Dilemmas besteht in einem eigenen Befehl **rcall**. Ein **rcall** macht prinzipiell auch den Sprung zu einem Ziel, legt aber gleichzeitig auch noch die Adresse von wo der Sprung erfolgt ist in einem speziellen Speicherbereich ab, so dass sein 'Gegenspieler', der Befehl **ret** (wie Return) anhand dieser abgelegten Information wieder genau zu dieser Aufrufstelle zurückspringen kann. Diesen speziellen Speicherbereich nennt man den "Stack". Stack bedeutet übersetzt soviel wie Stapel. Damit ist ein Speicher nach dem LIFO-Prinzip ("last in first out") gemeint. Das bedeutet, dass das zuletzt auf den Stapel gelegte Element auch zuerst wieder heruntergenommen wird. Es ist nicht möglich, Elemente irgendwo in der Mitte des Stapels herauszuziehen oder hineinzuschieben. Ein Stack (oder Stapel) funktioniert wie ein Stapel Teller. Der Teller, welcher zuletzt auf den Stapel gelegt wird, ist auch der erste, welcher wieder vom Stapel heruntergenommen wird. Und genau das wird in diesem Fall ja auch benötigt: jeder **rcall** legt seine Rücksprungadresse auf den Stack, so dass alle nachfolgenden **ret** jeweils in umgekehrter Reihenfolge wieder die richtigen Rücksprungadressen anspringen.

Bei allen aktuellen AVR-Controllern wird der Stack im RAM angelegt. Der Stack wächst dabei von oben nach unten: Am Anfang wird der Stackpointer (Adresse der aktuellen Stapelposition) auf das Ende des RAMs gesetzt. Wird nun ein Element hinzugefügt, wird dieses an der momentanen

Stackpointerposition abgespeichert und der Stackpointer um 1 erniedrigt. Soll ein Element vom Stack heruntergenommen werden, wird zuerst der Stackpointer um 1 erhöht und dann das Byte von der vom Stackpointer angezeigten Position gelesen.

Aufruf von Unterprogrammen

Dem Prozessor dient der Stack hauptsächlich dazu, Rücksprungadressen beim Aufruf von Unterprogrammen zu speichern, damit er später noch weiß, an welche Stelle zurückgekehrt werden muss, wenn das Unterprogramm mit **ret** oder die Interruptroutine mit **reti** beendet wird.

Das folgende Beispielprogramm (AT90S4433) zeigt, wie der Stack dabei beeinflusst wird:

Download stack.asm

```
.include "4433def.inc"     ; bzw. 2333def.inc

.def temp = r16

         ldi temp, RAMEND  ; Stackpointer initialisieren
         out SP, temp

         rcall sub1        ; sub1 aufrufen

loop:    rjmp loop


sub1:
                           ; hier könnten ein paar Befehle stehen
         rcall sub2        ; sub2 aufrufen
                           ; hier könnten auch ein paar Befehle stehen
         ret               ; wieder zurück

sub2:
                           ; hier stehen normalerweise die Befehle,
                           ; die in sub2 ausgeführt werden sollen
         ret               ; wieder zurück
```

.def temp = r16 ist eine Assemblerdirektive. Diese sagt dem Assembler, dass er überall, wo er "temp" findet, stattdessen "r16" einsetzen soll. Das ist oft praktisch, damit man nicht mit den Registernamen durcheinander kommt. Eine Übersicht über die Assemblerdirektiven findet man hier.

Bei Controllern, die mehr als 256 Byte RAM besitzen (z. B. ATmega8), passt die Adresse nicht mehr in ein Byte. Deswegen gibt es bei diesen Controllern das Stack-Pointer-Register aufgeteilt in **SPL** (Low) und **SPH** (High), in denen das Low- und das High-Byte der Adresse gespeichert wird. Damit es funktioniert, muss das Programm dann folgendermaßen geändert werden:

Download stack-bigmem.asm

```
.include "m8def.inc"

.def temp = r16

         ldi temp, HIGH(RAMEND)            ; HIGH-Byte der obersten RAM-Adresse
         out SPH, temp
         ldi temp, LOW(RAMEND)             ; LOW-Byte der obersten RAM-Adresse
         out SPL, temp
```

```
        rcall sub1                          ; sub1 aufrufen

loop:   rjmp loop


sub1:
                                            ; hier könnten ein paar Befehle
stehen
        rcall sub2                          ; sub2 aufrufen
                                            ; hier könnten auch Befehle stehen
        ret                                 ; wieder zurück

sub2:
                                            ; hier stehen normalerweise die
Befehle,
                                            ; die in sub2 ausgeführt werden
sollen
        ret                                 ; wieder zurück
```

Natürlich ist es unsinnig, dieses Programm in einen Controller zu programmieren. Stattdessen sollte man es mal mit dem AVR-Studio simulieren, um die Funktion des Stacks zu verstehen.

Als erstes wird mit **Project/New** ein neues Projekt erstellt, zu dem man dann mit **Project/Add File** eine Datei mit dem oben gezeigten Programm (stack.asm) hinzufügt. Nachdem man unter **Project/Project Settings** das **Object Format for AVR-Studio** ausgewählt hat, kann man das Programm mit Strg+F7 assemblieren und den Debug-Modus starten.

Danach sollte man im Menu **View** die Fenster **Processor** und **Memory** öffnen und im Memory-Fenster **Data** auswählen.

Das Fenster **Processor**

- *Program Counter*: Adresse im Programmspeicher (FLASH), die gerade abgearbeitet wird
- *Stack Pointer*: Adresse im Datenspeicher (RAM), auf die der Stackpointer gerade zeigt
- *Cycle Counter*: Anzahl der Taktzyklen seit Beginn der Simulation
- *Time Elapsed*: Zeit, die seit dem Beginn der Simulation vergangen ist

Im Fenster **Memory** wird der Inhalt des RAMs angezeigt.

Sind alle 3 Fenster gut auf einmal sichtbar, kann man anfangen, das Programm (in diesem Fall "stack.asm") mit der Taste F11 langsam Befehl für Befehl zu simulieren.

Wenn der gelbe Pfeil in der Zeile **out SPL, temp** vorbeikommt, kann man im Prozessor-Fenster sehen, wie der Stackpointer auf 0xDF (*ATmega8*: 0x45F) gesetzt wird. Wie man im Memory-Fenster sieht, ist das die letzte RAM-Adresse.

Wenn der Pfeil auf dem Befehl **rcall sub1** steht, sollte man sich den Program Counter anschauen: Er steht auf 0x02 (*ATmega8*: 0x04).

Drückt man jetzt nochmal auf F11, springt der Pfeil zum Unterprogramm sub1. Im RAM erscheint an der Stelle, auf die der Stackpointer vorher zeigte, die Zahl 0x03 (*ATmega8*: 0x05). Das ist die Adresse im ROM, an der das Hauptprogramm nach dem Abarbeiten des Unterprogramms fortgesetzt wird. Doch warum wurde der Stackpointer um 2 verkleinert? Das liegt daran, dass eine Programmspeicheradresse bis zu 2 Byte breit sein kann, und somit auch 2 Byte auf dem Stack

benötigt werden, um die Adresse zu speichern.

Das gleiche passiert beim Aufruf von sub2.

Zur Rückkehr aus dem mit rcall aufgerufenen Unterprogramm gibt es den Befehl **ret**. Dieser Befehl sorgt dafür, dass der Stackpointer wieder um 2 erhöht wird und die dabei eingelesene Adresse in den "Program Counter" kopiert wird, so dass das Programm dort fortgesetzt wird.

Apropos Program Counter: Wer sehen will, wie so ein Programm aussieht, wenn es assembliert ist, sollte mal die Datei mit der Endung ".lst" im Projektverzeichnis öffnen. Die Datei sollte ungefähr so aussehen:

```
                .def temp = r16

               ;Stackpointer initialisieren
000000  ed0f        ldi temp, RAMEND
000001  bf0d        out SPL, temp

000002  d001        rcall sub1

            loop:
000003  cfff        rjmp loop


            sub1:
000004  d001          rcall sub2
000005  9508        ret

            sub2:
000006  0000          nop
000007  9508        ret
```

Im blau umrahmten Bereich steht die Adresse des Befehls im Programmspeicher. Das ist auch die Zahl, die im Program Counter angezeigt wird, und die beim Aufruf eines Unterprogramms auf den Stack gelegt wird. Der grüne Bereich rechts daneben ist der OP-Code des Befehls, so wie er in den Programmspeicher des Controllers programmiert wird, und im roten Kasten stehen die "mnemonics": Das sind die Befehle, die man im Assembler eingibt. Der nicht eingerahmte Rest besteht aus Assemblerdirektiven, Labels (Sprungmarkierungen) und Kommentaren, die nicht direkt in OP-Code umgewandelt werden. Der grün eingerahmte Bereich ist das eigentliche Programm, so wie es der µC versteht. Die jeweils erste Zahl im grünen Bereich steht für einen Befehl, den sog. OP-Code (OP = Operation). Die zweite Zahl codiert Argumente für diesen Befehl.

Sichern von Registern

Eine weitere Anwendung des Stacks ist das "Sichern" von Registern. Wenn man z. B. im Hauptprogramm die Register R16, R17 und R18 verwendet, dann ist es i.d.R. erwünscht, dass diese Register durch aufgerufene Unterprogramme nicht beeinflusst werden. Man muss also nun entweder auf die Verwendung dieser Register innerhalb von Unterprogrammen verzichten, oder man sorgt dafür, dass am Ende jedes Unterprogramms der ursprüngliche Zustand der Register wiederhergestellt wird. Wie man sich leicht vorstellen kann ist ein "Stapelspeicher" dafür ideal: Zu

Beginn des Unterprogramms legt man die Daten aus den zu sichernden Registern oben auf den Stapel, und am Ende holt man sie wieder (in der umgekehrten Reihenfolge) in die entsprechenden Register zurück. Das Hauptprogramm bekommt also wenn es fortgesetzt wird überhaupt nichts davon mit, dass die Register inzwischen anderweitig verwendet wurden.

Download stack-saveregs.asm

```
.include "4433def.inc"            ; bzw. 2333def.inc

.def temp = R16

         ldi temp, RAMEND         ; Stackpointer initialisieren
         out SP, temp

         ldi temp, 0xFF
         out DDRB, temp           ; Port B = Ausgang

         ldi R17, 0b10101010      ; einen Wert ins Register R17 laden

         rcall sub1                ; Unterprogramm "sub1" aufrufen

         out PORTB, R17           ; Wert von R17 an den Port B ausgeben

loop:    rjmp loop                ; Endlosschleife


sub1:
         push R17                 ; Inhalt von R17 auf dem Stack speichern

         ; hier kann nach belieben mit R17 gearbeitet werden,
         ; als Beispiel wird es hier auf 0 gesetzt

         ldi R17, 0

         pop R17                  ; R17 zurückholen
         ret                      ; wieder zurück zum Hauptprogramm
```

Wenn man dieses Programm assembliert und in den Controller lädt, dann wird man feststellen, dass jede zweite LED an Port B leuchtet. Der ursprüngliche Wert von R17 blieb also erhalten, obwohl dazwischen ein Unterprogramm aufgerufen wurde, das R17 geändert hat.

Auch in diesem Fall kann man bei der Simulation des Programms im AVR-Studio die Beeinflussung des Stacks durch die Befehle **push** und **pop** genau nachvollziehen.

Sprung zu beliebiger Adresse

Dieser Abschnitt ist veraltet, da nahezu alle ATmega/ATtiny Typen IJMP/ICALL unterstützen.

Kleinere AVR besitzen keinen Befehl, um direkt zu einer Adresse zu springen, die in einem Registerpaar gespeichert ist. Man kann dies aber mit etwas Stack-Akrobatik erreichen. Dazu einfach zuerst den niederen Teil der Adresse, dann den höheren Teil der Adresse mit **push** auf den Stack legen und ein **ret** ausführen:

```
        ldi ZH, high(testRoutine)
        ldi ZL, low(testRoutine)
```

```
        push ZL
        push ZH
        ret

        ...
testRoutine:
        rjmp testRoutine
```

Auf diese Art und Weise kann man auch Unterprogrammaufrufe durchführen:

```
        ldi ZH, high(testRoutine)
        ldi ZL, low(testRoutine)
        rcall indirectZCall
        ...


indirectZCall:
        push ZL
        push ZH
        ret

testRoutine:
        ...
        ret
```

Größere AVR haben dafür die Befehle `ijmp` und `icall`. Bei diesen Befehlen muss das Sprungziel in ZH:ZL stehen.

Weitere Informationen (von Lothar Müller):

- Der Stack - Funktion und Nutzen (pdf)
- Der Stack - Parameterübergabe an Unterprogramme (pdf)
- Der Stack - Unterprogramme mit variabler Parameteranzahl (pdf)

(Der in dieser Abhandlung angegebene Befehl *MOV ZLow, SPL* muss *IN ZL, SPL* heißen, da SPL und SPH I/O-Register sind. Ggf ist auch SPH zu berücksichtigen --> 2byte Stack-Pointer)

AVR-Tutorial: LCD

Kaum ein elektronisches Gerät kommt heutzutage noch ohne ein LCD daher. Ist doch auch praktisch, Informationen im Klartext anzeigen zu können, ohne irgendwelche LEDs blinken zu lassen. Kein Wunder also, dass die häufigste Frage in Mikrocontroller-Foren ist: "Wie kann ich ein LCD anschließen?"

Inhaltsverzeichnis

- 1 Das LCD und sein Controller
- 2 Anschluss an den Controller
- 3 Ansteuerung des LCDs im 4-Bit-Modus
- 4 Initialisierung des Displays
  - 4.1 Initialisierung für 4 Bit Modus
  - 4.2 Initialisierung für 8 Bit Modus
- 5 Routinen zur LCD-Ansteuerung im 4-Bit-Modus
- 6 Anwendung
- 7 ASCII
- 8 Welche Befehle versteht das LCD?
  - 8.1 Clear display: 0b00000001
  - 8.2 Cursor home: 0b0000001x
  - 8.3 Entry mode: 0b000001is
  - 8.4 On/off control: 0b00001dcb
  - 8.5 Cursor/Scrollen: 0b0001srxx
  - 8.6 Konfiguration: 0b001dnfxx
  - 8.7 Character RAM Address Set: 0b01aaaaaa
  - 8.8 Display RAM Address Set: 0b1aaaaaaa
- 9 Einschub: Code aufräumen
  - 9.1 Portnamen aus dem Code herausziehen
  - 9.2 Registerbenutzung
  - 9.3 Lass den Assembler rechnen
- 10 Ausgabe eines konstanten Textes
- 11 Zahlen ausgeben
  - 11.1 Dezimal ausgeben
  - 11.2 Unterdrückung von führenden Nullen
  - 11.3 Hexadezimal ausgeben
  - 11.4 Binär ausgeben

    - 11.5 Eine 16-Bit Zahl aus einem Registerpärchen ausgeben
    - 11.6 Eine BCD Zahl ausgeben
- 12 Benutzerdefinierte Zeichen
- 13 Der überarbeitete, komplette Code:

Das LCD und sein Controller

Die meisten Text-LCDs verwenden den Controller **HD44780** oder einen kompatiblen (z. B. KS0070) und haben 14 oder 16 Pins.
Die Pinbelegung ist meist (Ausnahme z. B. TC1602E (Pollin 120420): V_{DD} und V_{SS} vertauscht) folgendermaßen:

ACHTUNG: Es gibt Displays mit abweichender Anschluss-Belegung, falscher Anschluss kann zur Zerstörung führen! Daher immer das zugehörige Datenblatt zu Rate ziehen!

Einzelheiten unter Artikel zum Controller HD44780

Pin #	Bezeichnung	Funktion
1	V_{SS} (selten: V_{DD})	GND (selten: +5 V)
2	V_{DD} (selten: V_{SS})	+5 V (selten: GND)
3	V_{EE}, V0, V5	Kontrastspannung (-5 V / 0 V bis 5 V)
4	RS	Register Select (0=Befehl/Status 1=Daten)
5	RW	1=Read 0=Write
6	E	0=Disable 1=Enable
7	DB0	Datenbit 0
8	DB1	Datenbit 1
9	DB2	Datenbit 2
10	DB3	Datenbit 3
11	DB4	Datenbit 4
12	DB5	Datenbit 5
13	DB6	Datenbit 6
14	DB7	Datenbit 7
15	A	LED-Beleuchtung, meist Anode
16	K	LED-Beleuchtung, meist Kathode

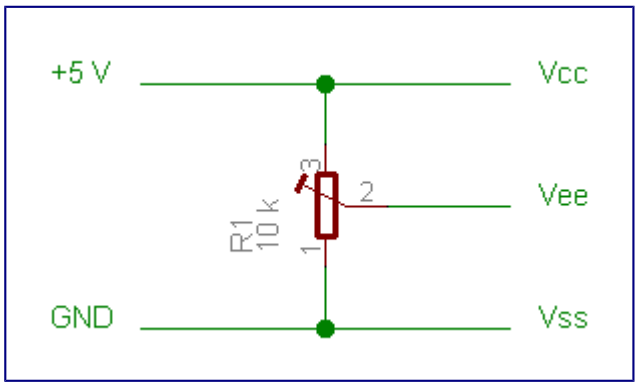
Achtung: Unbedingt von der richtigen Seite zu zählen anfangen! Meistens ist das Pin1-Pad eckig oder daneben eine kleine 1 auf der LCD-Platine, ansonsten im Datenblatt nachschauen.

Bei der DIL-Version (2x7, 2x8 Kontakte) auch darauf achten, auf welcher Platinen-Seite der Stecker montiert wird: auf der falschen (meist hinteren) Seite sind dann die Flachbandleitungen 1 und 2, 3 und 4 usw. vertauscht. Das kann man kompensieren, indem man es auf der anderen

Kabelseite genauso permutiert oder es auf dem Layout bewusst so legt (Stecker auf der Bottom-Seite plazieren). Man kann es NICHT kompensieren, indem man das Flachbandkabel auf der anderen Seite in den Stecker führt.

Bei LCDs mit 16-poligem Anschluss sind die beiden letzten Pins für die Hintergrundbeleuchtung reserviert. Hier unbedingt das Datenblatt zu Rate ziehen. Die beiden Anschlüsse sind je nach Hersteller verdreht beschaltet. Falls kein Datenblatt vorliegt, kann man mit einem Durchgangsprüfer feststellen, welcher Anschluss mit Masse (GND) verbunden ist.

V_{SS} wird ganz einfach an GND angeschlossen und V_{CC}=V_{DD} an +5 V. V_{EE} = V0 = V5 kann man testweise auch an GND legen. Wenn das LCD dann zu dunkel sein sollte, muss man ein 10kΩ-Potentiometer zwischen GND und 5 V schalten, mit dem Schleifer an V_{EE}. Meist kann man den +5 V-Anschluss am Poti weglassen, da im Display ein Pull-up-Widerstand ist:



Gewinnung der Kontrastspannung

Wenn der Kontrast zu schwach sein sollte (z.B. bei tiefen Temperaturen oder bei Betrieb mit 3.3V), kann man anstelle von GND eine negative Spannung ans Kontrast-Poti legen. Diese kann bis -5 V gehen und kann leicht aus einem Timerpin des µC, einem Widerstand, zwei Dioden und zwei Kondensatoren erzeugt werden. So wird auch ein digital einstellbarer Kontrast mittels PWM ermöglicht. ACHTUNG: Es gibt jedoch auch Displaycontroller wie den Epson SED1278, die zwar Software-kompatibel sind, aber keine negativen Kontrastspannung verkraften. Wird der Kontrast also bei negativer Spannung schlechter oder geht das Display ganz aus, ist davon auszugehen, dass der Controller diesen Betriebsmodus nicht unterstützt.

Es gibt zwei verschiedene Möglichkeiten zur Ansteuerung eines solchen Displays: den **8-Bit**- und den **4-Bit**-Modus.

- Für den **8-Bit-Modus** werden (wie der Name schon sagt) alle acht Datenleitungen zur Ansteuerung verwendet, somit kann durch einen Zugriff immer ein ganzes Byte übertragen werden.
- Der **4-Bit-Modus** verwendet nur die oberen vier Datenleitungen (**DB4-DB7**). Um ein Byte zu übertragen, braucht man somit zwei Zugriffe, wobei zuerst das höherwertige **"Nibble"** (= 4 Bits), also Bit 4 bis Bit 7 übertragen wird und dann das niederwertige, also Bit 0 bis Bit 3. Die unteren Datenleitungen des LCDs, die beim Lesezyklus Ausgänge sind, lässt man offen (siehe Datasheets, z. B. vom KS0070).

Der 4-Bit-Modus hat den Vorteil, dass man 4 IO-Pins weniger benötigt als beim 8-Bit-Modus. 6

bzw. 7 Pins (eines Portes) reichen aus.

Neben den vier Datenleitungen (DB4, DB5, DB6 und DB7) werden noch die Anschlüsse **RS**, **RW** und **E** benötigt.

- Über **RS** wird ausgewählt, ob man einen Befehl oder ein Datenbyte an das LCD schicken möchte. Beim Schreiben gilt: ist RS Low, dann wird das ankommende Byte als Befehl interpretiert; Ist RS high, wird das Byte auf dem LCD angezeigt (genauer: ins Data-Register geschrieben, kann auch für den CG bestimmt sein).
- **RW** legt fest, ob geschrieben oder gelesen werden soll. High bedeutet lesen, low bedeutet schreiben. Wenn man RW auf lesen einstellt und RS auf Befehl, dann kann man das **Busy-Flag** an DB7 lesen, das anzeigt, ob das LCD den vorhergehenden Befehl fertig verarbeitet hat. Ist RS auf Daten eingestellt, dann kann man z. B. den Inhalt des Displays lesen - was jedoch nur in den wenigsten Fällen Sinn macht. Deshalb kann man RW dauerhaft auf low lassen (= an GND anschließen), so dass man noch ein IO-Pin am Controller einspart. Der Nachteil ist, dass man dann das Busy-Flag nicht lesen kann, weswegen man nach jedem Befehl ca. 50 µs (beim Return Home 2 ms, beim Clear Display 20 ms) warten sollte, um dem LCD Zeit zum Ausführen des Befehls zu geben. Dummerweise schwankt die Ausführungszeit von Display zu Display und ist auch von der Betriebsspannung abhängig. Für professionellere Sachen also lieber den IO-Pin opfern und Busy abfragen.
- Der **E** Anschluss schließlich signalisiert dem LCD, dass die übrigen Datenleitungen jetzt korrekte Pegel angenommen haben und es die gewünschten Daten von den Datenleitungen bzw. Kommandos von den Datenleitungen übernehmen kann. Beim Lesen gibt das Display die Daten / Status so lange aus, wie E high ist. Beim Schreiben übernimmt das Display die Daten mit der fallenden Flanke.

Anschluss an den Controller

Jetzt, da wir wissen, welche Anschlüsse das LCD benötigt, können wir das LCD mit dem Mikrocontroller verbinden:

ACHTUNG: Es gibt Displays mit abweichender Anschluss-Belegung (z. B. TC1602E (Pollin 120420): Vdd und Vss vertauscht), falscher Anschluss kann zur Zerstörung führen! Daher immer das zugehörige Datenblatt zu Rate ziehen.

Einzelheiten unter Artikel zum Controller HD44780

Pinnummer LCD	Bezeichnung	Anschluss
1	V_{SS}	GND (beim TC1602E: V_{CC})
2	V_{CC}	+5 V (beim TC1602E: Gnd)
3	V_{EE}	GND , Poti oder PWM am AVR
4	RS	PD4 am AVR
5	RW	GND
6	E	PD5 am AVR

7	DB0	nicht angeschlossen
8	DB1	nicht angeschlossen
9	DB2	nicht angeschlossen
10	DB3	nicht angeschlossen
11	DB4	PD0 am AVR
12	DB5	PD1 am AVR
13	DB6	PD2 am AVR
14	DB7	PD3 am AVR
15	A	Vorsicht! Meistens nicht direkt an +5 V anschließbar, sondern nur über einen Vorwiderstand, der an die Daten der Hintergrundbeleuchtung angepasst werden muss.
16	K	GND

Ok. Alles ist verbunden. Wenn man jetzt den Strom einschaltet, sollten ein oder zwei schwarze Balken auf dem Display angezeigt werden.

Doch wie bekommt man jetzt die Befehle und Daten in das Display? Dazu muss das LCD initialisiert werden und man muss Befehle (Commands) und seine Daten an das LCD senden. Weil die Initialisierung ein Spezialfall der Übertragung von Befehlen ist, im Folgenden zunächst die Erklärung für die Übertragung von Werten an das LCD.

Ansteuerung des LCDs im 4-Bit-Modus

Um ein Byte zu übertragen, muss man es erstmal in die beiden Nibbles zerlegen, die getrennt übertragen werden. Da das obere Nibble (Bit 4 - Bit 7) als erstes übertragen wird, die 4 Datenleitungen jedoch an die vier unteren Bits des Port D angeschlossen sind, muss man die beiden Nibbles des zu übertragenden Bytes erstmal vertauschen. Der AVR kennt dazu praktischerweise einen eigenen Befehl:

```
swap r16                ; vertauscht die beiden Nibbles von r16
```

Aus 0b00100101 wird so z. B. 0b01010010.

Jetzt sind die Bits für die erste Phase der Übertragung an der richtigen Stelle. Trotzdem wollen wir das Ergebnis nicht einfach so mit **out PORTD, r16** an den Port geben. Um die Hälfte des Bytes, die jetzt nicht an die Datenleitungen des LCDs gegeben wird auf null zu setzen, verwendet man folgenden Befehl:

```
andi r16, 0b00001111    ; Nur die vier unteren (mit 1 markierten)
                        ; Bits werden übernommen, alle anderen werden
null
```

Also: Das obere Nibble wird erst mit dem unteren vertauscht, damit es unten ist. Dann wird das obere (das wir jetzt noch nicht brauchen) auf null gesetzt.

Jetzt müssen wir dem LCD noch mitteilen, ob wir Daten oder Befehle senden wollen. Das machen

wir, indem wir das Bit, an dem RS angeschlossen ist (PD4), auf 0 (Befehl senden) oder auf 1 (Daten senden) setzen. Um ein Bit in einem normalen Register zu setzen, gibt es den Befehl sbr (Set Bits in Register). Dieser Befehl unterscheidet sich jedoch von sbi (das nur für IO-Register gilt) dadurch, dass man nicht die Nummer des zu setzenden Bits angibt, sondern eine Bitmaske. Das geht so:

```
            sbr r16, 0b00010000     ; Bit 4 setzen, alle anderen Bits bleiben
gleich
```

RS ist an PD4 angeschlossen. Wenn wir r16 an den Port D ausgeben, ist RS jetzt also high und das LCD erwartet Daten anstatt von Befehlen.

Das Ergebnis können wir jetzt endlich direkt an den Port D übergeben:

```
            out PORTD, r16
```

Natürlich muss vorher der Port D auf Ausgang geschaltet werden, indem man 0xFF ins Datenrichtungsregister DDRD schreibt.

Um dem LCD zu signalisieren, dass es das an den Datenleitungen anliegende Nibble übernehmen kann, wird die E-Leitung (Enable, an PD5 angeschlossen) auf high und kurz darauf wieder auf low gesetzt. Ein Puls an dieser Leitung teilt also dem LCD mit, das die restlichen Leitungen jetzt ihren vom Programm gewollten Pegel eingenommen haben und gültig sind.

```
            sbi PORTD, 5              ; Enable high
            nop                       ; 3 Taktzyklen warten ("nop" = nichts tun)
            nop
            nop
            cbi PORTD, 5              ; Enable wieder low
```

Die eine Hälfte des Bytes wäre damit geschafft! Die andere Hälfte kommt direkt hinterher: Alles, was an der obenstehenden Vorgehensweise geändert werden muss, ist, das "swap" (Vertauschen der beiden Nibbles) wegzulassen.

Initialisierung des Displays

Allerdings gibt es noch ein Problem. Wenn ein LCD eingeschaltet wird, dann läuft es zunächst im 8 Bit Modus. Irgendwie muss das Display initialisiert und auf den 4 Bit Modus umgeschaltet werden, und zwar nur mit den 4 zur Verfügung stehenden Datenleitungen.

Wenn es Probleme gibt, dann meistens an diesem Punkt. Die "kompatiblen" Kontroller sind gelegentlich doch nicht 100% identisch. Es lohnt sich, das Datenblatt (siehe Weblinks im Artikel LCD) genau zu lesen, in welcher Reihenfolge und mit welchen Abständen (Delays) die Initialisierungbefehle gesendet werden. Eine weitere Hilfe können Ansteuerungsbeispiele in Forenbeiträgen geben z. B.

- (A) KS0066U oder Ähnliche --- LCD Treiber

Initialisierung für 4 Bit Modus

Achtung: Im Folgenden sind alle Bytes aus Sicht des LCD-Kontrollers angegeben! Da LCD-seitig

nur die Leitungen DB4 - DB7 verwendet werden, ist daher immer nur das höherwertige Nibble gültig. Durch die Art der Verschaltung (DB4 - DB7 wurde auf dem PORT an PD0 bis PD3 angeschlossen) ergibt sich eine Verschiebung, so dass das am Kontroller auszugebende Byte nibblemässig vertauscht ist!

Die Sequenz, aus Sicht des Kontrollers, sieht so aus:

- Nach dem Anlegen der Betriebsspannung muss eine Zeit von mindestens ca. 15ms gewartet werden, um dem LCD-Kontroller Zeit für seine eigene Initialisierung zu geben
- $3 ins Steuerregister schreiben (RS = 0)
- Mindestens 4.1ms warten
- $3 ins Steuerregister schreiben (RS = 0)
- Mindestens 100µs warten
- $3 ins Steuerregister schreiben (RS = 0)
- $2 ins Steuerregister schreiben (RS = 0), dadurch wird auf 4 Bit Daten umgestellt
- Ab jetzt muss für die Übertragung eines Bytes jeweils zuerst das höherwertige Nibble und dann das niederwertige Nibble übertragen werden, wie oben beschrieben
- Mit dem Konfigurier-Befehl $20 das Display konfigurieren (4-Bit, 1 oder 2 Zeilen, 5x7 Format)
- Mit den restlichen Konfigurierbefehlen die Konfiguration vervollständigen: Display ein/aus, Cursor ein/aus, etc.

Eine Begründung, warum die ersten Befehle dreifach geschickt werden sollen, findet sich im Forum.

Initialisierung für 8 Bit Modus

Der Vollständigkeit halber hier noch die notwendige Initialiserungssequenz für den 8 Bit Modus. Da hier die Daten komplett als 1 Byte übertragen werden können, sind einige Klimmzüge wie im 4 Bit Modus nicht notwendig. Begründung für die anfänglichen Wiederholungen siehe oben.

- Nach dem Anlegen der Betriebsspannung muss eine Zeit von mindestens ca. 15ms gewartet werden, um dem LCD-Kontroller Zeit für seine eigene Initialisierung zu geben
- $30 ins Steuerregister schreiben (RS = 0)
- Mindestens 4.1ms warten
- $30 ins Steuerregister schreiben (RS = 0)
- Mindestens 100µs warten
- $30 ins Steuerregister schreiben (RS = 0)
- Mit dem Konfigurier-Befehl 0x30 das Display konfigurieren (8-Bit, 1 oder 2 Zeilen, 5x7 Format)
- Mit den restlichen Konfigurierbefehlen die Konfiguration vervollständigen: Display ein/aus, Cursor ein/aus, etc.

Routinen zur LCD-Ansteuerung im 4-Bit-Modus

Im Folgenden werden die bisherigen Grundroutinen zur LCD-Ansteuerung im 4-Bit-Modus

zusammengefasst und kommentiert. Die darin enthaltenen Symbole (temp1, PORTD,...) müssen in einem dazugehörenden Hauptprogramm definiert werden. Dies wird nächsten Abschnitt *Anwendung* weiter erklärt.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                 LCD-Routinen                ;;
;;                 ============                ;;
;;              (c)andreas-s@web.de            ;;
;;                                             ;;
;; 4bit-Interface                              ;;
;; DB4-DB7:       PD0-PD3                      ;;
;; RS:            PD4                          ;;
;; E:             PD5                          ;;
;;                                             ;;
;; Takt:          4 MHz                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


 ;sendet ein Datenbyte an das LCD
lcd_data:
           mov temp2, temp1             ; "Sicherungskopie" für
                                        ; die Übertragung des 2.Nibbles
           swap temp1                   ; Vertauschen
           andi temp1, 0b00001111       ; oberes Nibble auf Null setzen
           sbr temp1, 1<<4              ; entspricht 0b00010000 (Anm.1)
           out PORTD, temp1             ; ausgeben
           rcall lcd_enable             ; Enable-Routine aufrufen
                                        ; 2. Nibble, kein swap da es schon
                                        ; an der richtigen stelle ist
           andi temp2, 0b00001111       ; obere Hälfte auf Null setzen
           sbr temp2, 1<<4              ; entspricht 0b00010000
           out PORTD, temp2             ; ausgeben
           rcall lcd_enable             ; Enable-Routine aufrufen
           rcall delay50us              ; Delay-Routine aufrufen
           ret                          ; zurück zum Hauptprogramm

 ; sendet einen Befehl an das LCD
lcd_command:                            ; wie lcd_data, nur RS=0
           mov temp2, temp1
           swap temp1
           andi temp1, 0b00001111
           out PORTD, temp1
           rcall lcd_enable
           andi temp2, 0b00001111
           out PORTD, temp2
           rcall lcd_enable
           rcall delay50us
           ret

 ; erzeugt den Enable-Puls
 ;
 ; Bei höherem Takt (>= 8 MHz) kann es notwendig sein,
 ; vor dem Enable High 1-2 Wartetakte (nop) einzufügen.
 ; Siehe dazu http://www.mikrocontroller.net/topic/81974#685882
lcd_enable:
           sbi PORTD, 5                 ; Enable high
           nop                          ; mindestens 3 Taktzyklen warten
           nop
           nop
           cbi PORTD, 5                 ; Enable wieder low
           ret                          ; Und wieder zurück
```

```
 ; Pause nach jeder Übertragung
delay50us:                              ; 50µs Pause (bei 4 MHz)
           ldi  temp1, $42
delay50us_:dec  temp1
           brne delay50us_
           ret                          ; wieder zurück

 ; Längere Pause für manche Befehle
delay5ms:                               ; 5ms Pause (bei 4 MHz)
           ldi  temp1, $21
WGLOOP0:   ldi  temp2, $C9
WGLOOP1:   dec  temp2
           brne WGLOOP1
           dec  temp1
           brne WGLOOP0
           ret                          ; wieder zurück

 ; Initialisierung: muss ganz am Anfang des Programms aufgerufen werden
lcd_init:
           ldi  temp3,50
powerupwait:
           rcall  delay5ms
           dec  temp3
           brne powerupwait
           ldi temp1, 0b00000011        ; muss 3mal hintereinander gesendet
           out PORTD, temp1             ; werden zur Initialisierung
           rcall lcd_enable             ; 1
           rcall delay5ms
           rcall lcd_enable             ; 2
           rcall delay5ms
           rcall lcd_enable             ; und 3!
           rcall delay5ms
           ldi temp1, 0b00000010        ; 4bit-Modus einstellen
           out PORTD, temp1
           rcall lcd_enable
           rcall delay5ms
           ldi temp1, 0b00101000        ; 4Bit / 2 Zeilen / 5x8
           rcall lcd_command
           ldi temp1, 0b00001100        ; Display ein / Cursor aus / kein
Blinken
           rcall lcd_command
           ldi temp1, 0b00000100        ; inkrement / kein Scrollen
           rcall lcd_command
           ret

 ; Sendet den Befehl zur Löschung des Displays
lcd_clear:
           ldi temp1, 0b00000001   ; Display löschen
           rcall lcd_command
           rcall delay5ms
           ret

 ; Sendet den Befehl: Cursor Home
lcd_home:
           ldi temp1, 0b00000010   ; Cursor Home
           rcall lcd_command
           rcall delay5ms
           ret
```

Anm.1: Siehe Bitmanipulation

Weitere Funktionen (wie z. B. Cursorposition verändern) sollten mit Hilfe der Befehlscodeliste

nicht schwer zu realisieren sein. Einfach den Code in temp laden, lcd_command aufrufen und ggf. eine Pause einfügen.
Natürlich kann man die LCD-Ansteuerung auch an einen anderen Port des Mikrocontrollers "verschieben": Wenn das LCD z. B. an Port B angeschlossen ist, dann reicht es, im Programm alle "PORTD" durch "PORTB" und "DDRD" durch "DDRB" zu ersetzen.
Wer eine höhere Taktfrequenz als 4 MHz verwendet, der sollte daran denken, die Dauer der Verzögerungsschleifen anzupassen.

Anwendung

Ein Programm, das diese Routinen zur Anzeige von Text verwendet, kann z. B. so aussehen (die Datei lcd-routines.asm muss sich im gleichen Verzeichnis befinden). Nach der Initialisierung wird zuerst der Displayinhalt gelöscht. Um dem LCD ein Zeichen zu schicken, lädt man es in temp1 und ruft die Routine "lcd_data" auf. Das folgende Beispiel zeigt das Wort "Test" auf dem LCD an.

Download lcd-test.asm

```
.include "m8def.inc"

; .def definiert ein Synonym (Namen) für ein µC Register
.def temp1 = r16
.def temp2 = r17
.def temp3 = r18

           ldi temp1, LOW(RAMEND)      ; LOW-Byte der obersten RAM-Adresse
           out SPL, temp1
           ldi temp1, HIGH(RAMEND)     ; HIGH-Byte der obersten RAM-Adresse
           out SPH, temp1

           ldi temp1, 0xFF    ; Port D = Ausgang
           out DDRD, temp1

           rcall lcd_init     ; Display initialisieren
           rcall lcd_clear    ; Display löschen

           ldi temp1, 'T'     ; Zeichen anzeigen
           rcall lcd_data

           ldi temp1, 'e'     ; Zeichen anzeigen
           rcall lcd_data

           ldi temp1, 's'     ; Zeichen anzeigen
           rcall lcd_data

           ldi temp1, 't'     ; Zeichen anzeigen
           rcall lcd_data

loop:
           rjmp loop

.include "lcd-routines.asm"            ; LCD-Routinen werden hier eingefügt
```

Für längere Texte ist die Methode, jedes Zeichen einzeln in das Register zu laden und "lcd_data" aufzurufen natürlich nicht sehr praktisch. Dazu später aber mehr.

Bisher wurden in Register immer irgendwelche Zahlenwerte geladen, aber in diesem Programm

kommt plötzlich die Anweisung

```
ldi temp1, 'T'
```

vor. Wie ist diese zu verstehen? Passiert hier etwas grundlegend anderes als beim Laden einer Zahl in ein Register?

Die Antwort darauf lautet: Nein. Auch hier wird letztendlich nur eine Zahl in ein Register geladen. Der Schlüssel zum Verständnis beruht darauf, dass zum LCD, so wie zu allen Ausgabegeräten, für die Ausgabe von Texten immer nur Zahlen übertragen werden, sog. Codes. Zum Beispiel könnte man vereinbaren, dass ein LCD, wenn es den Ausgabecode 65 erhält, ein 'A' anzeigt, bei einem Ausgabecode von 66 ein 'B' usw. Naturgemäß gibt es daher viele verschiedene Code-Buchstaben Zuordnungen. Damit hier etwas Ordnung in das potentielle Chaos kommt, hat man sich bereits in der Steinzeit der Programmierung auf bestimmte Codetabellen geeinigt, von denen die verbreitetste sicherlich die ASCII-Zuordnung ist.

ASCII

ASCII steht für *American Standard Code for Information Interchange* und ist ein standardisierter Code zur Zeichenumsetzung. Die Codetabelle sieht hexadezimal dabei wie folgt aus:

	x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Die ersten beiden Zeilen enthalten die Codes für einige Steuerzeichen, ihre vollständige Beschreibung würde hier zu weit führen. Das Zeichen **SP** steht für ein *Space*, also ein Leerzeichen. **BS** steht für *Backspace*, also ein Zeichen zurück. **DEL** steht für *Delete*, also das Löschen eines Zeichens. **CR** steht für *Carriage Return*, also wörtlich: der Wagenrücklauf (einer Schreibmaschine), während **LF** für *Line feed*, also einen Zeilenvorschub steht.

Der Assembler kennt diese Codetabelle und ersetzt die Zeile

```
ldi temp1, 'T'
```

durch

```
ldi temp1, $54
```

was letztendlich auch der Lesbarkeit des Programmes zugute kommt. Funktional besteht kein

Unterschied zwischen den beiden Anweisungen. Beide bewirken, dass das Register temp1 mit dem Bitmuster 01010100 (= hexadezimal 54, = dezimal 84 oder eben der ASCII Code für **T**) geladen wird.

Das LCD wiederrum kennt diese Code-Tabelle ebenfalls und wenn es über den Datenbus die Codezahl $54 zur Anzeige empfängt, dann schreibt es ein **T** an die aktuelle Cursorposition. Genauer gesagt, weiss das LCD nichts von einem **T**. Es sieht einfach in seinen internen Tabellen nach, welche Pixel beim Empfang der Codezahl $54 auf schwarz zu setzen sind. 'Zufällig' sind das genau jene Pixel, die für uns Menschen ein **T** ergeben.

Welche Befehle versteht das LCD?

Auf dem LCD arbeitet ein Controller vom Typ HD44780. Dieser Kontroller versteht eine Reihe von Befehlen, die allesamt mittels lcd_command gesendet werden können. Ein Kommando ist dabei nichts anderes als ein Befehlsbyte, in dem die verschiedenen Bits verschiedene Bedeutungen haben:

Bitwert	Bedeutung
0	dieses Bit muss 0 sein
1	dieses Bit muss 1 sein
x	der Zustand dieses Bits ist egal
sonstige Buchstaben	das Bit muss je nach gewünschter Funktionalität gesetzt werden. Die mögliche Funktionalität des jeweiligen Bits geht aus der Befehlsbeschreibung hervor.

Beispiel: Das Kommando 'ON/OFF Control' soll benutzt werden, um das Display einzuschalten, der Cursor soll eingeschaltet werden und der Cursor soll blinken. Das Befehlsbyte ist so aufgebaut:

```
0b00001dcb
```

Aus der Befehlsbeschreibung entnimmt man:

- Display ein bedeutet, dass an der Bitposition d eine 1 stehen muss.
- Cursor ein bedeutet, dass an der Bitposition c ein 1 stehen muss.
- Cursor blinken bedeutet, dass an der Bitposition b eine 1 stehen muss.

Das dafür zu übertragende Befehlsbyte hat also die Gestalt 0b00001111 oder in hexadezimaler Schreibweise $0F.

Clear display: 0b00000001

Die Anzeige wird gelöscht und der Ausgabecursor kehrt an die Home Position (links, erste Zeile) zurück.

Ausführungszeit: 1.64ms

Cursor home: 0b0000001x

Der Cursor kehrt an die Home Position (links, erste Zeile) zurück. Ein verschobenes Display wird auf die Grundeinstellung zurückgesetzt.

Ausführungszeit: 40µs bis 1.64ms

Entry mode: 0b000001is

Legt die Cursor Richtung sowie eine mögliche Verschiebung des Displays fest

- i = 1, Cursorposition bei Ausgabe eines Zeichens erhöhen
- i = 0, Cursorposition bei Ausgabe eines Zeichens vermindern
- s = 1, Display wird gescrollt, wenn der Cursor das Ende/Anfang, je nach Einstellung von i, erreicht hat.

Ausführungszeit: 40µs

On/off control: 0b00001dcb

Display insgesamt ein/ausschalten; den Cursor ein/ausschalten; den Cursor auf blinken schalten/blinken aus. Wenn das Display ausgeschaltet wird, geht der Inhalt des Displays nicht verloren. Der vorher angezeigte Text wird nach wiedereinschalten erneut angezeigt. Ist der Cursor eingeschaltet, aber Blinken ausgeschaltet, so wird der Cursor als Cursorzeile in Pixelzeile 8 dargestellt. Ist Blinken eingeschaltet, wird der Cursor als blinkendes ausgefülltes Rechteck dargestellt, welches abwechselnd mit dem Buchstaben an dieser Stelle angezeigt wird.

- d = 0, Display aus
- d = 1, Display ein
- c = 0, Cursor aus
- c = 1, Cursor ein
- b = 0, Cursor blinken aus
- b = 1, Cursor blinken ein

Ausführungszeit: 40µs

Cursor/Scrollen: 0b0001srxx

Bewegt den Cursor oder scrollt das Display um eine Position entweder nach rechts oder nach links.

- s = 1, Display scrollen
- s = 0, Cursor bewegen
- r = 1, nach rechts
- r = 0, nach links

Ausführungszeit: 40µs

Konfiguration: 0b001dnfxx

Einstellen der Interface Art, Modus, Font

- d = 0, 4-Bit Interface
- d = 1, 8-Bit Interface
- n = 0, 1 zeilig
- n = 1, 2 zeilig

- f = 0, 5x7 Pixel
- f = 1, 5x11 Pixel

Ausführungszeit: 40µs

Character RAM Address Set: 0b01aaaaaa

Mit diesem Kommando werden maximal 8 selbst definierte Zeichen definiert. Dazu wird der Character RAM Zeiger auf den Anfang des Character Generator (CG) RAM gesetzt und das Zeichen durch die Ausgabe von 8 Byte definiert. Der Adresszeiger wird nach Ausgabe jeder Pixelspalte (8 Bit) vom LCD selbst erhöht. Nach Beendigung der Zeichendefinition muss die Schreibposition explizit mit dem Kommando "Display RAM Address Set" wieder in den DD-RAM Bereich gesetzt werden.

aaaaaa 6-bit CG RAM Adresse

Ausführungszeit: 40µs

Display RAM Address Set: 0b1aaaaaaa

Den Cursor neu positionieren. Display Data (DD) Ram ist vom Character Generator (CG) Ram unabhängig. Der Adresszeiger wird bei Ausgabe eines Zeichens ins DD Ram automatisch erhöht. Das Display verhält sich so, als ob eine Zeile immer aus 40 logischen Zeichen besteht, von der, je nach konkretem Displaytyp (16 Zeichen, 20 Zeichen) immer nur ein Teil sichtbar ist.

aaaaaaa 7-bit DD RAM Adresse. Auf 2-zeiligen Displays (und den meisten 16x1 Displays), kann die Adressangabe wie folgt interpretiert werden:

1laaaaaa

- l = Zeilennummer (0 oder 1)
- a = 6-Bit Spaltennummer

```
--------------------------------
DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
--- --- --- --- --- --- --- ---
 1   A   A   A   A   A   A   A
```

Setzt die DDRAM Adresse:

Wenn N = 0 (1 line display)

```
AAAAAAA = "00h" - "4Fh"
```

Wenn N = 1 (2 line display) ((1x16))

```
AAAAAAA = "00h" - "27h" Zeile 1. (0x80)
AAAAAAA = "40h" - "67h" Zeile 2. (0xC0)
```

Ausführungszeit: 40µs

Einschub: Code aufräumen

Es wird Zeit, sich einmal etwas kritisch mit den bisher geschriebenen Funktionen auseinander zu setzen.

Portnamen aus dem Code herausziehen

Wenn wir die LCD-Funktionen einmal genauer betrachten, dann fällt sofort auf, dass über die Funktionen verstreut immer wieder das **PORTD** sowie einzelne Zahlen für die Pins an diesem Port auftauchen. Wenn das LCD an einem anderen Port betrieben werden soll, oder sich die Pin-Belegung ändert, dann muss an all diesen Stellen eine Anpassung vorgenommen werden. Dabei darf keine einzige Stelle übersehen werden, ansonsten würden die LCD-Funktionen nicht oder nicht vollständig funktionieren.

Eine Möglichkeit, dem vorzubeugen, ist es, diese immer gleichbleibenden Dinge an den Anfang der LCD-Funktionen vorzuziehen. Anstelle von PORTD wird dann im Code ein anderer Name benutzt, den man frei vergeben kann. Dem Assembler wird nur noch mitgeteilt, das dieser Name für PORTD steht. Muss das LCD an einen anderen Port angeschlossen werden, so wird nur diese Zurodnung geändert und der Assembler passt dann im restlichen Code alle davon abhängigen Anweisungen an:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                 LCD-Routinen                ;;
;;                 ============                ;;
;;              (c)andreas-s@web.de            ;;
;;                                             ;;
;; 4bit-Interface                              ;;
;; DB4-DB7:       PD0-PD3                      ;;
;; RS:            PD4                          ;;
;; E:             PD5                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; .equ definiert ein Symbol und dessen Wert
.equ LCD_PORT = PORTD
.equ LCD_DDR  = DDRD
.equ PIN_E    = 5
.equ PIN_RS   = 4

 ;sendet ein Datenbyte an das LCD
lcd_data:
           mov temp2, temp1             ; "Sicherungskopie" für
                                        ; die Übertragung des 2.Nibbles
           swap temp1                   ; Vertauschen
           andi temp1, 0b00001111       ; oberes Nibble auf Null setzen
           sbr temp1, 1<<PIN_RS         ; entspricht 0b00010000
           out LCD_PORT, temp1          ; ausgeben
           rcall lcd_enable             ; Enable-Routine aufrufen
                                        ; 2. Nibble, kein swap da es schon
                                        ; an der richtigen stelle ist
           andi temp2, 0b00001111       ; obere Hälfte auf Null setzen
           sbr temp2, 1<<PIN_RS         ; entspricht 0b00010000
           out LCD_PORT, temp2          ; ausgeben
           rcall lcd_enable             ; Enable-Routine aufrufen
           rcall delay50us              ; Delay-Routine aufrufen
           ret                          ; zurück zum Hauptprogramm
```

```
 ; sendet einen Befehl an das LCD

lcd_command:                            ; wie lcd_data, nur RS=0
           mov temp2, temp1
           swap temp1
           andi temp1, 0b00001111
           out LCD_PORT, temp1
           rcall lcd_enable
           andi temp2, 0b00001111
           out LCD_PORT, temp2
           rcall lcd_enable
           rcall delay50us
           ret

 ; erzeugt den Enable-Puls
lcd_enable:
           sbi LCD_PORT, PIN_E          ; Enable high
           nop                          ; 3 Taktzyklen warten
           nop
           nop
           cbi LCD_PORT, PIN_E          ; Enable wieder low
           ret                          ; Und wieder zurück

 ; Pause nach jeder Übertragung
delay50us:                              ; 50µs Pause
           ldi  temp1, $42
delay50us_:dec  temp1
           brne delay50us_
           ret                          ; wieder zurück

 ; Längere Pause für manche Befehle
delay5ms:                               ; 5ms Pause
           ldi  temp1, $21
WGLOOP0:   ldi  temp2, $C9
WGLOOP1:   dec  temp2
           brne WGLOOP1
           dec  temp1
           brne WGLOOP0
           ret                          ; wieder zurück

 ; Initialisierung: muss ganz am Anfang des Programms aufgerufen werden
lcd_init:
           ldi   temp1, 0xFF            ; alle Pins am Ausgabeport auf Ausgang
           out   LCD_DDR, temp1

           ldi   temp3,6
powerupwait:
           rcall delay5ms
           dec   temp3
           brne  powerupwait
           ldi   temp1,    0b00000011   ; muss 3mal hintereinander gesendet
           out   LCD_PORT, temp1        ; werden zur Initialisierung
           rcall lcd_enable             ; 1
           rcall delay5ms
           rcall lcd_enable             ; 2
           rcall delay5ms
           rcall lcd_enable             ; und 3!
           rcall delay5ms
           ldi   temp1, 0b00000010      ; 4bit-Modus einstellen
           out   LCD_PORT, temp1
           rcall lcd_enable
           rcall delay5ms
           ldi   temp1, 0b00101000      ; 4 Bit, 2 Zeilen
```

```
            rcall lcd_command
            ldi   temp1, 0b00001100      ; Display on, Cursor off
            rcall lcd_command
            ldi   temp1, 0b00000100      ; endlich fertig
            rcall lcd_command
            ret

 ; Sendet den Befehl zur Löschung des Displays
lcd_clear:
            ldi   temp1, 0b00000001      ; Display löschen
            rcall lcd_command
            rcall delay5ms
            ret

 ; Sendet den Befehl: Cursor Home
lcd_home:
            ldi   temp1, 0b00000010      ; Cursor Home
            rcall lcd_command
            rcall delay5ms
            ret
```

Mittels **.equ** werden mit dem Assembler Textersetzungen vereinbart. Der Assembler ersetzt alle Vorkomnisse des Quelltextes durch den zu ersetzenden Text. Dadurch ist es z. B. möglich, alle Vorkommnisse von **PORTD** durch **LCD_PORT** auszutauschen. Wird das LCD an einen anderen Port, z. B. **PORTB** gelegt, dann genügt es, die Zeilen

```
.equ LCD_PORT = PORTD
.equ LCD_DDR  = DDRD
```

durch

```
.equ LCD_PORT = PORTB
.equ LCD_DDR  = DDRB
```

zu ersetzen. Der Assembler sorgt dann dafür, dass diese Portänderung an den relevanten Stellen im Code über die Textersetzungen einfließt. Selbiges natürlich mit der Pin-Zuordnung.

Registerbenutzung

Bei diesen Funktionen mussten einige Register des Prozessors benutzt werden, um darin Zwischenergebnisse zu speichern bzw. zu bearbeiten.

Beachtet werden muss dabei natürlich, dass es zu keinen Überschneidungen kommt. Solange nur jede Funktion jeweils für sich betrachtet wird, ist das kein Problem. In 20 oder 30 Code-Zeilen kann man gut verfolgen, welches Register wofür benutzt wird. Schwieriger wird es, wenn Funktionen wiederum andere Funktionen aufrufen, die ihrerseits wieder Funktionen aufrufen usw. Jede dieser Funktionen benutzt einige Register und mit zunehmender Programmgröße wird es immer schwieriger, zu verfolgen, welches Register zu welchem Zeitpunkt wofür benutzt wird.

Speziell bei Basisfunktionen wie diesen LCD-Funktionen, ist es daher oft ratsam, dafür zu sorgen, dass jede Funktion die Register wieder in dem Zustand hinterlässt, indem sie sie auch vorgefunden hat. Wir benötigen dazu wieder den Stack, auf dem die Registerinhalte bei Betreten einer Funktion zwischengespeichert werden und von dem die Register bei Verlassen einer Funktion

wiederhergestellt werden.

Nehmen wir die Funktion

```
 ; Sendet den Befehl zur Löschung des Displays
lcd_clear:
           ldi   temp1, 0b00000001      ; Display löschen
           rcall lcd_command
           rcall delay5ms
           ret
```

Diese Funktion verändert das Register temp1. Um das Register abzusichern, schreiben wir die Funktion um:

```
 ; Sendet den Befehl zur Löschung des Displays
lcd_clear:
           push  temp1                  ; temp1 auf dem Stack sichern
           ldi   temp1, 0b00000001      ; Display löschen
           rcall lcd_command
           rcall delay5ms
           pop   temp1                  ; temp1 vom Stack wiederherstellen
           ret
```

Am besten hält man sich an die Regel: Jede Funktion ist dafür zuständig, die Register zu sichern und wieder herzustellen, die sie auch selbst verändert. **lcd_clear** ruft die Funktionen **lcd_command** und **delay5ms** auf. Wenn diese Funktionen selbst wieder Register verändern (und das tun sie), so ist es die Aufgabe dieser Funktionen, sich um die Sicherung und das Wiederherstellen der entsprechenden Register zu kümmern. **lcd_clear** sollte sich nicht darum kümmern müssen. Auf diese Weise ist das Schlimmste, das einem passieren kann, dass ein paar Register unnütz gesichert und wiederhergestellt werden. Das kostet zwar etwas Rechenzeit und etwas Speicherplatz auf dem Stack, ist aber immer noch besser als das andere Extrem: Nach einem Funktionsaufruf haben einige Register nicht mehr den Wert, den sie haben sollten, und das Programm rechnet mit falschen Zahlen weiter.

Lass den Assembler rechnen

Betrachtet man den Code genauer, so fallen einige konstante Zahlenwerte auf (Das vorangestellte $ kennzeichnet die Zahl als Hexadezimalzahl):

```
delay50us:                              ; 50µs Pause
           ldi  temp1, $42
delay50us_:
           dec  temp1
           brne delay50us_
           ret                          ; wieder zurück
```

Der Code benötigt eine Warteschleife, die mindestens 50µs dauert. Die beiden Befehle innerhalb der Schleife benötigen 3 Takte: 1 Takt für den **dec** und der **brne** benötigt 2 Takte, wenn die Bedingung zutrifft, der Branch also genommen wird. Bei 4 Mhz werden also 4000000 / 3 * 50 / 1000000 = 66.6 Durchläufe durch die Schleife benötigt, um eine Verzögerungszeit von 50µs (0.000050 Sekunden) zu erreichen, hexadezimal ausgedrückt: $42.

Der springende Punkt ist: Bei anderen Taktfrequenzen müsste man nun jedesmal diese Berechnung machen und den entsprechenden Zahlenwert einsetzen. Das kann aber der Assembler genausogut erledigen. Am Anfang des Codes wird ein Eintrag definiert, der die Taktfrequenz festlegt. Traditionell heißt dieser Eintrag *XTAL*:

```
.equ XTAL  = 4000000

...

delay50us:                              ; 50µs Pause
           ldi  temp1, ( XTAL * 50 / 3 ) / 1000000
delay50us_:
           dec  temp1
           brne delay50us_
           ret                          ; wieder zurück
```

An einer anderen Codestelle gibt es weitere derartige magische Zahlen:

```
 ; Längere Pause für manche Befehle
delay5ms:                               ; 5ms Pause
           ldi  temp1, $21
WGLOOP0:   ldi  temp2, $C9
WGLOOP1:   dec  temp2
           brne WGLOOP1
           dec  temp1
           brne WGLOOP0
           ret                          ; wieder zurück
```

Was geht hier vor? Die innere Schleife benötigt wieder 3 Takte pro Durchlauf. Bei $C9 = 201 Durchläufen werden also 201 * 3 = 603 Takte verbraucht. In der äußeren Schleife werden pro Durchlauf also 603 + 1 + 2 = 606 Takte verbraucht und einmal 605 Takte (weil der brne nicht genommen wird). Da die äußere Schleife $21 = 33 mal wiederholt wird, werden 32 * 606 + 605 = 19997 Takte verbraucht. Noch 1 Takt mehr für den allerersten ldi und 4 Takte für den ret, macht 20002 Takte. Bei 4Mhz benötigt der Prozessor 20002 / 4000000 = 0.0050005 Sekunden, also rund 5 ms. Die 7. nachkommastelle kann man an dieser Stelle getrost ignorieren. Vor allen Dingen auch deshalb, weil auch der Quarz nicht exakt 4000000 Schwingungen in der Sekunde durchführen wird. Wird der Wiederholwert für die innere Schleife bei $C9 belassen, so werden 4000000 / 607 * 5 / 1000 Wiederholungen der äusseren Schleife benötigt. (Die Berechnung wurde hier etwas vereinfacht, die nicht berücksichtigten Takte fallen zeitmässig nicht weiter ins Gewicht bzw. wurden dadurch berücksichtigt, dass mit 607 anstelle von 606 gerechnet wird). Auch diese Berechnung kann wieder der Assembler übernehmen:

```
 ; Längere Pause für manche Befehle
delay5ms:                               ; 5ms Pause
           ldi  temp1, ( XTAL * 5 / 607 ) / 1000
WGLOOP0:   ldi  temp2, $C9
WGLOOP1:   dec  temp2
           brne WGLOOP1
           dec  temp1
           brne WGLOOP0
           ret                          ; wieder zurück
```

Ein kleines Problem kann bei der Verwendung dieses Verfahrens entstehen: Bei hohen

Taktfrequenzen und großen Wartezeiten kann der berechnete Wert größer als 255 werden und man bekommt die Fehlermeldung "Operand(s) out of range" beim Assemblieren. Dieser Fall tritt zum Beispiel für obige Konstruktion bei einer Taktfrequenz von 16 MHz ein (genauer gesagt ab 15,3 MHz), während darunter XTAL beliebig geändert werden kann. Als einfachste Lösung bietet es sich an, die Zahl der Takte pro Schleifendurchlauf durch das Einfügen von **nop** zu erhöhen und die Berechnungsvorschrift anzupassen.

Ausgabe eines konstanten Textes

Weiter oben wurde schon einmal ein Text ausgegeben. Dies geschah durch Ausgabe von einzelnen Zeichen. Das können wir auch anders machen. Wir können den Text im Speicher ablegen und eine Funktion schreiben, die die einzelnen Zeichen aus dem Speicher liest und aus gibt. Dabei stellt sich Frage: Woher 'weiß' die Funktion eigentlich, wie lang der Text ist? Die Antwort darauf lautet: Sie kann es nicht wissen. Wir müssen irgendwelche Vereinbarungen treffen, woran die Funktion erkennen kann, dass der Text zu Ende ist. Im Wesentlichen werden dazu 2 Methoden benutzt:

- Der Text enthält ein spezielles Zeichen, welches das Ende des Textes markiert
- Wir speichern nicht nur den Text selbst, sondern auch die Länge des Textes

Mit einer der beiden Methoden ist es der Textausgabefunktion dann ein Leichtes, den Text vollständig auszugeben.

Wir werden uns im Weiteren dafür entscheiden, ein spezielles Zeichen, eine 0 (den Wert 0, nicht das Zeichen '0'), dafür zu benutzen. Die Ausgabefunktionen werden dann etwas einfacher, als wenn bei der Ausgabe die Anzahl der bereits ausgegebenen Zeichen mitgezählt werden muss.

Den Text selbst speichern wir im Flash-Speicher, also dort, wo auch das Programm gespeichert ist:

```
 ; Einen konstanten Text aus dem Flash Speicher
 ; ausgeben. Der Text wird mit einer 0 beendet
lcd_flash_string:
           push  temp1
           push  ZH
           push  ZL

lcd_flash_string_1:
           lpm   temp1, Z+
           cpi   temp1, 0
           breq  lcd_flash_string_2
           rcall  lcd_data
           rjmp  lcd_flash_string_1

lcd_flash_string_2:
           pop   ZL
           pop   ZH
           pop   temp1
           ret
```

Diese Funktion benutzt den Befehl **lpm**, um das jeweils nächste Zeichen aus dem Flash Speicher in ein Register zur Weiterverarbeitung zu laden. Dazu wird der sog. **Z-Pointer** benutzt. So nennt man das Registerpaar **R30** und **R31**. Nach jedem Ladevorgang wird dabei durch den Befehl

```
            lpm   temp1, Z+
```

dieser Z-Pointer um 1 erhöht. Mittels **cpi** wird das in das Register **temp1** geladene Zeichen mit 0 verglichen. **cpi** vergleicht die beiden Zahlen und merkt sich das Ergebnis in einem speziellen Register in Form von Status Bits. **cpi** zieht dabei ganz einfach die beiden Zahlen voneinander ab. Sind sie gleich, so kommt da als Ergebnis 0 heraus und **cpi** setzt daher konsequenter Weise das Zero-Flag, das anzeigt, dass die vorhergegangene Operation eine 0 als Ergebnis hatte.**breq** wertet diese Status-Bits aus. Wenn die vorhergegangene Operation ein 0-Ergebnis hatte, das Zero-Flag also gesetzt ist, dann wird ein Sprung zum angegebenen Label durchgeführt. In Summe bewirkt also die Sequenz

```
            cpi   temp1, 0
            breq  lcd_flash_string_2
```

dass das gelesene Zeichen mit 0 verglichen wird und falls das gelesene Zeichen tatsächlich 0 war, an der Stelle lcd_flash_string_2 weiter gemacht wird. Im anderen Fall wird die bereits geschriebene Funktion **lcd_data** aufgerufen, welche das Zeichen ausgibt. **lcd_data** erwartet dabei das Zeichen im Register **temp1**, genau in dem Register, in welches wir vorher mittels **lpm** das Zeichen geladen hatten.

Das verwendende Programm sieht dann so aus:

```
.include "m8def.inc"

.def temp1 = r16
.def temp2 = r17
.def temp3 = r18


            ldi temp1, LOW(RAMEND)      ; LOW-Byte der obersten RAM-Adresse
            out SPL, temp1
            ldi temp1, HIGH(RAMEND)     ; HIGH-Byte der obersten RAM-Adresse
            out SPH, temp1

            rcall lcd_init              ; Display initialisieren
            rcall lcd_clear             ; Display löschen

            ldi ZL, LOW(text*2)         ; Adresse des Strings in den
            ldi ZH, HIGH(text*2)        ; Z-Pointer laden

            rcall lcd_flash_string      ; Unterprogramm gibt String aus der
                                        ; durch den Z-Pointer adressiert wird
loop:
            rjmp loop

text:
            .db "Test",0                ; Stringkonstante, durch eine 0
                                        ; abgeschlossen

.include "lcd-routines.asm"             ; LCD Funktionen
```

Genaueres über die Verwendung unterschiedlicher Speicher findet sich im Kapitel Speicher

Zahlen ausgeben

Um Zahlen, die beispielsweise in einem Register gespeichert sind, ausgeben zu können, ist es notwendig sich eine Textrepräsentierung der Zahl zu generieren. Die Zahl 123 wird also in den Text "123" umgewandelt welcher dann ausgegeben wird. Aus praktischen Gründen wird allerdings der Text nicht vollständig generiert (man müsste ihn ja irgendwo zwischenspeichern) sondern die einzelnen Buchstaben werden sofort ausgegeben, sobald sie bekannt sind.

Dezimal ausgeben

Das Prinzip der Umwandlung ist einfach. Um herauszufinden wieviele Hunderter in der Zahl 123 enthalten sind, genügt es in einer Schleife immer wieder 100 von der Zahl abzuziehen und mitzuzählen wie oft dies gelang, bevor das Ergebnis negativ wurde. In diesem Fall lautet die Antwort: 1 mal, denn 123 - 100 macht 23. Versucht man erneut 100 anzuziehen, so ergibt sich eine negative Zahl. Also muss eine '1' ausgeben werden. Die verbleibenden 23 werden weiter behandelt, indem festgestellt wird wieviele Zehner darin enthalten sind. Auch hier wiederum: In einer Schleife solange 10 abziehen, bis das Ergebnis negativ wurde. Konkret geht das 2 mal gut, also muss das nächste auszugebende Zeichen ein '2' sein. Damit verbleiben noch die Einer, welche direkt in das entsprechende Zeichen umgewandelt werden können. In Summe hat man also an das Display die Zeichen '1' '2' '3' ausgegeben.

```
;*********************************************************************
;
; Eine 8 Bit Zahl ohne Vorzeichen ausgeben
;
; Übergabe:            Zahl im Register temp1
; veränderte Register: keine
;
lcd_number:
           push  temp1            ; die Funktion verändert temp1 und temp2,
           push  temp2            ; also sichern wir den Inhalt, um ihn am Ende
                                  ; wieder herstellen zu können

           mov   temp2, temp1     ; das Register temp1 frei machen
                                  ; abzählen wieviele Hunderter
                                  ; in der Zahl enthalten sind
;** Hunderter **
           ldi   temp1, '0'-1     ; temp1 mit ASCII '0'-1 vorladen
lcd_number_1:
           inc   temp1            ; ASCII erhöhen (somit ist nach dem ersten
                                  ; Durchlauf eine '0' in temp1)
           subi  temp2, 100       ; 100 abziehen
           brcc  lcd_number_1     ; ist dadurch kein Unterlauf entstanden?
                                  ; nein, dann zurück zu lcd_number_1
           subi  temp2, -100      ; 100 wieder dazuzählen, da die
                                  ; vorherhgehende Schleife 100 zuviel
                                  ; abgezogen hat
           rcall lcd_data         ; die Hunderterstelle ausgeben

;** Zehner  **
           ldi   temp1, '0'-1     ; temp1 mit ASCII '0'-1 vorladen
lcd_number_2:
           inc   temp1            ; ASCII erhöhen (somit ist nach dem ersten
                                  ; Durchlauf eine '0' in temp1)
           subi  temp2, 10        ; 10 abziehen
           brcc  lcd_number_2     ; ist dadurch kein Unterlauf enstanden?
```

```
                                ; nein, dann zurück zu lcd_number_2
          subi  temp2, -10      ; 10 wieder dazuzählen, da die
                                ; vorherhgehende Schleife 10 zuviel
                                ; abgezogen hat
          rcall lcd_data        ; die Zehnerstelle ausgeben

;** Einer **
          ldi   temp1, '0'      ; die Zahl in temp2 ist jetzt im Bereich
          add   temp1, temp2    ; 0 bis 9. Einfach nur den ASCII Code für
          rcall lcd_data        ; '0' dazu addieren und wir erhalten dierekt
                                ; den ASCII Code für die Ziffer


          pop   temp2           ; den gesicherten Inhalt von temp2 und temp1
          pop   temp1           ; wieder herstellen
          ret                   ; und zurück
```

Beachte: Diese Funktion benutzt wiederrum die Funktion **lcd_data**. Anders als bei den bisherigen Aufrufen ist **lcd_number** aber darauf angewiesen, dass **lcd_data** das Register **temp2** unangetastet lässt. Falls sie es noch nicht getan haben, dann ist das jetzt die perfekte Gelegenheit, **lcd_data** mit den entsprechenden **push** und **pop** Befehlen zu versehen. Sie sollten dies unbedingt zur Übung selbst machen. Am Ende muß die Funktion dann wie diese hier aussehen:

```
 ;sendet ein Datenbyte an das LCD
lcd_data:
          push  temp2
          mov   temp2, temp1          ; "Sicherungskopie" für
                                      ; die Übertragung des 2.Nibbles
          swap  temp1                 ; Vertauschen
          andi  temp1, 0b00001111     ; oberes Nibble auf Null setzen
          sbr   temp1, 1<<PIN_RS      ; entspricht 0b00010000
          out   LCD_PORT, temp1       ; ausgeben
          rcall lcd_enable            ; Enable-Routine aufrufen
                                      ; 2. Nibble, kein swap da es schon
                                      ; an der richtigen stelle ist
          andi  temp2, 0b00001111     ; obere Hälfte auf Null setzen
          sbr   temp2, 1<<PIN_RS      ; entspricht 0b00010000
          out   LCD_PORT, temp2       ; ausgeben
          rcall lcd_enable            ; Enable-Routine aufrufen
          rcall delay50us             ; Delay-Routine aufrufen
          pop   temp2
          ret                         ; zurück zum Hauptprogramm

 ; sendet einen Befehl an das LCD
lcd_command:                          ; wie lcd_data, nur ohne RS zu setzen
          push  temp2
          mov   temp2, temp1
          swap  temp1
          andi  temp1, 0b00001111
          out   LCD_PORT, temp1
          rcall lcd_enable
          andi  temp2, 0b00001111
          out   LCD_PORT, temp2
          rcall lcd_enable
          rcall delay50us
          pop   temp2
          ret
```

Kurz zur Funktionsweise der Funktion **lcd_number**: Die Zahl in einem Register bewegt sich im Wertebereich 0 bis 255. Um herauszufinden, wie die Hunderterstelle lautet, zieht die Funktion

einfach in einer Schleife immer wieder 100 von der Schleife ab, bis bei der Subtraktion ein Unterlauf, angezeigt durch das Setzen des Carry-Bits bei der Subtraktion, entsteht. Die Anzahl wird im Register **temp1** mitgezählt. Da dieses Register mit dem ASCII Code von '0' initialisiert wurde, und dieser ASCII Code bei jedem Schleifendurchlauf um 1 erhöht wird, können wir das Register **temp1** direkt zur Ausgabe des Zeichens für die Hunderterstelle durch die Funktion **lcd_data** benutzen. Völlig analog funktioniert auch die Ausgabe der Zehnerstelle.

Unterdrückung von führenden Nullen

Diese Funktion gibt jede Zahl im Register **temp1** immer mit 3 Stellen aus. Führende Nullen werden nicht unterdrückt. Möchte man dies ändern, so ist das ganz leicht möglich: Vor Ausgabe der Hunderterstelle muss lediglich überprüft werden, ob die Entsprechende Ausgabe eine '0' wäre. Ist sie das, so wird die Ausgabe übersprungen. Ist es allerdings eine Zahl 1..9, so muss sie der Zehner Stelle signalisieren, daß eine Prüfung auf eine '0' nicht stattfinden darf. Und dazu wird das T-Flag im SREG genutzt. Lediglich in der Einerstelle wird jede Ziffer wie errechnet ausgegeben.

```
           ...
                                ; die Hunderterstelle ausgeben, wenn
                                ; sie nicht '0' ist
           clt                  ; T-Flag löschen
           cpi   temp1, '0'
           breq  lcd_number_1a
           rcall lcd_data       ; die Hunderterstelle ausgeben
           set                  ; T-Flag im SREG setzen da 100er Stelle eine
                                ; 1..9 war

lcd_number_1a:
           ...

           ...
           brts  lcd_number_2a  ; Test auf '0' überspringen, da 100er eine
                                ; 1..9 war (unbedingt anzeigen
                                ; auch wenn der Zehner eine '0' ist)
           cpi   temp1, '0'     ; ansonsten Test auf '0'
           breq  lcd_number_2b
lcd_number_2a:
           rcall lcd_data
lcd_number_2b:
           ...
```

Das Verfahren, die einzelnen Stellen durch Subtraktion zu bestimmen, ist bei kleinen Zahlen eine durchaus gängige Alternative. Vor allem dann, wenn keine hardwaremäßige Unterstützung für Multiplikation und Division zur Verfügung steht. Ansonsten könnte man die die einzelnen Ziffern auch durch Division bestimmen. Das Prinzip ist folgendes (beispielhaft an der Zahl 52783 gezeigt)

```
52783 / 10          -> 5278
52783 - 5278 * 10   ->          3

5278 / 10           -> 527
5278 - 527 * 10     ->          8

527 / 10            -> 52
527 - 52 * 10       ->          7
```

```
   52 / 10              -> 5
   52 - 5 * 10          ->          2

   5 / 10               -> 0
   5 - 0 * 10           ->          5
```

Das Prinzip ist also die Restbildung bei einer fortgesetzten Division durch 10, wobei die einzelnen Ziffern in umgekehrter Reihenfolge ihrer Wertigkeit entstehen. Dadurch hat man aber ein Problem: Damit die Zeichen in der richtigen Reihenfolge ausgegeben werden können, muß man sie meistens zwischenspeichern um sie in der richtigen Reihenfole ausgeben zu können. Wird die Zahl in einem Feld von immer gleicher Größe ausgegeben, dann kann man auch die Zahl von rechts nach links ausgeben (bei einem LCD ist das möglich).

Hexadezimal ausgeben

Zu guter letzt hier noch eine Funktion, die eine Zahl aus dem Register **temp1** in hexadezimaler Form ausgibt. Die Funktion weist keine Besonderheiten auf und sollte unmittelbar verständlich sein.

```
;*********************************************************************
;
; Eine 8 Bit Zahl ohne Vorzeichen hexadezimal ausgeben
;
; Übergabe:            Zahl im Register temp1
; veränderte Register: keine
;
lcd_number_hex:
           swap  temp1
           rcall lcd_number_hex_digit
           swap  temp1

lcd_number_hex_digit:
           push  temp1

           andi  temp1, $0F
           cpi   temp1, 10
           brlt  lcd_number_hex_digit_1
           subi  temp1, -( 'A' - '9' - 1 ) ; es wird subi mit negativer
                                           ; Konstante verwendet,
                                           ; weil es kein addi gibt
lcd_number_hex_digit_1:
           subi  temp1, -'0'               ; ditto
           rcall  lcd_data

           pop   temp1
           ret
```

Binär ausgeben

Um die Sache komplett zu machen; Hier eine Routine mit der man eine 8 Bit-Zahl binär auf das LC-Display ausgeben kann:

```
;*********************************************************************
;
; Eine 8 Bit Zahl ohne Vorzeichen binär ausgeben
;
; Übergabe:            Zahl im Register temp1
; veränderte Register: keine
```

```
; eine Zahl aus dem Register temp1 binär ausgeben
lcd_number_bit:
           push temp1             ; temp1 gesichert
           push temp2
           push temp3

           mov temp2, temp1;

           ldi temp3, 8;      ; 8 Bits werden ausgelesen
lcd_number_loop:
           dec temp3;
           rol temp2;         ; Datenbits ins Carry geschoben ...
           brcc lcd_number_bit_carryset_0;
           brcs lcd_number_bit_carryset_1;
           rjmp lcd_number_loop;

lcd_number_bit_carryset_0:
           ldi temp1, '0'     ; Bit low ausgeben
           rcall lcd_data
           tst temp3;
           breq lcd_number_ende;
           rjmp lcd_number_loop;

lcd_number_bit_carryset_1:
           ldi temp1, '1'     ; Bit high ausgeben
           rcall lcd_data
           tst temp3;
           breq lcd_number_ende;
           rjmp lcd_number_loop;

lcd_number_ende:
           pop temp3
           pop temp2
           pop temp1
           ret
```

Eine 16-Bit Zahl aus einem Registerpärchen ausgeben

Um eine 16 Bit Zahl auszugeben wird wieder das bewährte Schema benutzt die einzelnen Stellen durch Subtraktion abzuzählen. Da es sich hierbei allerdings um eine 16 Bit Zahl handelt, müssen die Subtraktionen als 16-Bit Arithmetik ausgeführt werden.

```
;*********************************************************************
;
; Eine 16 Bit Zahl ohne Vorzeichen ausgeben
;
; Übergabe:            Zahl im Register temp2 (low Byte) / temp3 (high Byte)
; veränderte Register: keine
;
lcd_number16:
           push  temp1
           push  temp2
           push  temp3

; ** Zehntausender **
           ldi   temp1, '0'-1
lcd_number1:
           inc   temp1
           subi  temp2, low(10000)
           sbci  temp3, high(10000)
```

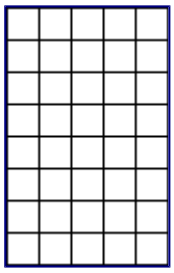
```
           brcc  lcd_number1
           subi  temp2, low(-10000)
           sbci  temp3, high(-10000)
           rcall lcd_data

; ** Tausender **
           ldi   temp1, '0'-1
lcd_number2:
           inc   temp1
           subi  temp2, low(1000)
           sbci  temp3, high(1000)
           brcc  lcd_number2
           subi  temp2, low(-1000)
           sbci  temp3, high(-1000)
           rcall lcd_data

; ** Hunderter **
           ldi   temp1, '0'-1
lcd_number3:
           inc   temp1
           subi  temp2, low(100)
           sbci  temp3, high(100)
           brcc  lcd_number3
           subi  temp2, -100             ; + 100 High-Byte nicht mehr
erforderlich
           rcall lcd_data

; ** Zehner **
           ldi   temp1, '0'-1
lcd_number4:
           inc   temp1
           subi  temp2, 10
           brcc  lcd_number4
           subi  temp2, -10
           rcall lcd_data

; ** Einer **
           ldi   temp1, '0'
           add   temp1, temp2
           rcall lcd_data

; ** Stack aufräumen **
           pop   temp3
           pop   temp2
           pop   temp1

           ret
```

Eine BCD Zahl ausgeben

```
;*********************************************************************
;
; Übergabe:            BCD Zahl in temp1
; veränderte Register: keine
;
lcd_bcd:
           push  temp2

           mov   temp2, temp1           ; temp1 sichern
           swap  temp1                  ; oberes mit unterem Nibble tauschen
           andi  temp1, 0b00001111      ; und "oberes" ausmaskieren
           subi  temp1, -0x30           ; in ASCII umrechnen
```

```
rcall lcd_data                ; und ausgeben
mov   temp1, temp2            ; ... danach unteres
andi  temp1, 0b00001111
subi  temp1, -0x30
rcall lcd_data
mov   temp1, temp2            ; temp1 rekonstruieren

pop   temp2
ret
```

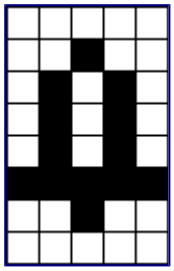
Benutzerdefinierte Zeichen



Zeichenraster für 1 Zeichen

Das LCD erlaubt für spezielle Zeichen, welche sich nicht im Zeichensatz finden, eigene Zeichen zu definieren. Dazu werden die ersten 8 ASCII Codes reserviert, auf denen sich laut ASCII Tabelle spezielle Steuerzeichen befinden, die normalerweise keine sichtbare Anzeige hervorrufen sondern zur Steuerung von angeschlossenen Geräten dienen. Da diese Zeichen auf einem LCD keine Rolle spielen, können diese Zeichen benutzt werden um sich selbst Sonderzeichen zu erzeugen, die für die jeweilige Anwendung massgeschneidert sind.

Das LCD stellt für jedes Zeichen eine 8*5 Matrix zur Verfügung. Um sich selbst massgeschneiderte Zeichen zu erstellen, ist es am einfachsten sich zunächst auf einem Stück karriertem Papier zu erstellen.



Zeichenraster für ein Glockensymbol

In diesem Raster markiert man sich dann diejenigen Pixel, die im fertigen Zeichen dunkel erscheinen sollen. Als Beispiel sei hier ein Glockensymbol gezeichnet, welches in einer Telefonapplikation zb als Kennzeichnung für einen Anruf dienen könnte.

Eine Zeile in diesem Zeichen repräsentiert ein an das LCD zu übergebendes Byte, wobei nur die Bits 0 bis 4 relevant sind. Gesetzte Pixel stellen ein 1 Bit dar, nicht gesetzte Pixel sind ein 0-Bit. Das niederwertigste Bit einer Zeile befindet sich rechts. Auf diese Art wird jede Zeile in eine Binärzahl übersetzt, und 8 Bytes repräsentieren ein komplettes Zeichen. Am Beispiel des Glockensymboles: Die 8 Bytes, welches das Symbol repräsentiern, lauten: 0x00, 0x04, 0x0A, 0x0A, 0x0A, 0x1F, 0x04, 0x00,

Dem LCD wird die neue Definition übertragen, indem man dem LCD die 'Schreibposition' mittels des Kommandos *Character RAM Address Set* in den Zeichensatzgenerator verschiebt. Danach werden die 8 Bytes ganz normal als Daten ausgegeben, die das LCD damit in seine Zeichensatztabelle schreibt.

Durch die Wahl der Speicheradresse definiert man, welches Zeichen (0 bis 7) man eigentlich durch eine eigene Definition ersetzen will.

ASCII Code	Zeichensatzadresse
0	0x00
1	0x08
2	0x10
3	0x18
4	0x20
5	0x28
6	0x30
7	0x38

Nach erfolgter Definition des Zeichens, muss die Schreibposition wieder explizit in den DDRAM-Bereich gesetzt werden. Danach kann ein entsprechendes Zeichen mit dem definierten ASCII Code ausgegeben werden, wobei das LCD die von uns definierte Pixelform zur Anzeige benutzt.

Zuerst müssen natürlich erstmal die Zeichen definiert werden. Dieses geschieht einmalig durch den Aufruf der Routine "lcd_load_user_chars" unmittelbar nach der Initialisierung des LCD-Displays.

```
.
.
rcall lcd_init                ; Display initialisieren
rcall lcd_load_user_chars     ; User Zeichen in das Display laden
rcall lcd_clear               ; Display löschen
.
.
```

Durch diesen Aufruf werden die im Flash definierten Zeichen in den GC-Ram übertragen. Diese Zeichen werden ab Adresse 0 im GC-Ram gespeichert und sind danach wie jedes andere Zeichen nutzbar.

```
.
.
ldi   temp1, 0                ; Ausgabe des User-Char "A"
rcall lcd_data
ldi   temp1, 6                ; Ausgabe des User-Char "G"
rcall lcd_data
ldi   temp1, 5                ; Ausgabe des User-Char "E"
rcall lcd_data
ldi   temp1, 4                ; Ausgabe des User-Char "M"
rcall lcd_data
ldi   temp1, 3                ; Ausgabe des User-Char "-"
rcall lcd_data
ldi   temp1, 2                ; Ausgabe des User-Char "R"
rcall lcd_data
```

```
            ldi   temp1, 1              ; Ausgabe des User-Char "V"
            rcall lcd_data
            ldi   temp1, 0              ; Ausgabe des User-Char "A"
            rcall lcd_data
            .
            .
```

Jetzt sollte der Schriftzug "AVR-MEGA" verkehrt herum (180 Grad gedreht) erscheinen.

Es fehlt natürlich noch die Laderoutine:

```
;*********************************************************************
;
; Lädt User Zeichen in den GC-Ram des LCD bis Tabellenende (0xFF)
; gelesen wird. (max. 8 Zeichen können geladen werden)
;
; Übergabe:            -
; veränderte Register: temp1, temp2, temp3, zh, zl
; Bemerkung:           ist einmalig nach lcd_init aufzurufen
;

lcd_load_user_chars:
        ldi    zl, LOW (ldc_user_char * 2) ; Adresse der Zeichentabelle
        ldi    zh, HIGH(ldc_user_char * 2) ; in den Z-Pointer laden
        clr    temp3                       ; aktuelles Zeichen = 0

lcd_load_user_chars_2:
        clr    temp2                       ; Linienzähler = 0

lcd_load_user_chars_1:
        ldi    temp1, 0b01000000           ; Kommando:    0b01aaalll
        add    temp1, temp3                ; + akt. Zeichen  (aaa)
        add    temp1, temp2                ; + akt. Linie       (lll)
        rcall  lcd_command                 ; Kommando schreiben

        lpm    temp1, Z+                   ; Zeichenline laden
        rcall  lcd_data                    ; ... und ausgeben

        ldi    temp1, 0b01001000           ; Kommando:    0b01aa1lll
        add    temp1, temp3                ; + akt. Zeichen  (aaa)
        add    temp1, temp2                ; + akt. Linie       (lll)
        rcall  lcd_command

        lpm    temp1, Z+                   ; Zeichenline laden
        rcall  lcd_data                    ; ... und ausgeben

        inc    temp2                       ; Linienzähler + 1
        cpi    temp2, 8                    ; 8 Linien fertig?
        brne   lcd_load_user_chars_1       ; nein, dann nächste Linie

        subi   temp3, -0x10                ; zwei Zeichen weiter (addi 0x10)
        lpm    temp1, Z                    ; nächste Linie laden
        cpi    temp1, 0xFF                 ; Tabellenende erreicht?
        brne   lcd_load_user_chars_2       ; nein, dann die nächsten
                                           ; zwei Zeichen
        ret
```

... und die Zeichendefinition:

```
ldc_user_char:
                              ;    Zeichen
```

```
                        ;   0       1
.db 0b10001, 0b00100    ; @   @ ,   @
.db 0b10001, 0b01010    ; @   @ ,  @ @
.db 0b11111, 0b10001    ; @@@@@ , @   @
.db 0b10001, 0b10001    ; @   @ , @   @
.db 0b10001, 0b10001    ; @   @ , @   @
.db 0b10001, 0b10001    ; @   @ , @   @
.db 0b01110, 0b10001    ;  @@@  , @   @
.db 0b00000, 0b00000    ;       ,

                        ;    Zeichen
                        ;   2       3
.db 0b10001, 0b00000    ; @   @ ,
.db 0b01001, 0b00000    ;  @  @ ,
.db 0b00101, 0b00000    ;   @ @ ,
.db 0b11111, 0b11111    ; @@@@@ , @@@@@
.db 0b10001, 0b00000    ; @   @ ,
.db 0b10001, 0b00000    ; @   @ ,
.db 0b01111, 0b00000    ;  @@@@ ,
.db 0b00000, 0b00000    ;       ,

                        ;    Zeichen
                        ;   4       5
.db 0b10001, 0b11111    ; @   @ , @@@@@
.db 0b10001, 0b00001    ; @   @ ,     @
.db 0b10001, 0b00001    ; @   @ ,     @
.db 0b10001, 0b01111    ; @   @ ,  @@@@
.db 0b10101, 0b00001    ; @ @ @ ,     @
.db 0b11011, 0b00001    ; @@ @@ ,     @
.db 0b10001, 0b11111    ; @   @ , @@@@@
.db 0b00000, 0b00000    ;       ,

                        ;    Zeichen
                        ;   6       7
.db 0b11110, 0b11111    ; @@@@  , @@@@@
.db 0b10001, 0b01010    ; @   @ ,  @ @
.db 0b10001, 0b00100    ; @   @ ,   @
.db 0b11101, 0b01110    ; @@@ @ ,  @@@
.db 0b00001, 0b00100    ;     @ ,   @
.db 0b10001, 0b01010    ; @   @ ,  @ @
.db 0b01110, 0b11111    ;  @@@  , @@@@@
.db 0b00000, 0b00000    ;       ,

; End of Tab
.db 0xFF, 0xFF
```

Der überarbeitete, komplette Code:

Hier also die komplett überarbeitete Version der LCD Funktionen.

Die für die Benutzung relevanten Funktionen

- **lcd_init**
- **lcd_clear**
- **lcd_home**
- **lcd_data**
- **lcd_command**
- **lcd_flash_string**

- **lcd_number**
- **lcd_number_hex**

sind so ausgeführt, dass sie kein Register (ausser dem Statusregister **SREG**) verändern. Die bei manchen Funktionen notwendige Argumente werden immer im Register **temp1** übergeben, wobei **temp1** vom Usercode definiert werden muss.

Download lcd-routines.asm

AVR-Tutorial: Interrupts

Inhaltsverzeichnis

- 1 Definition
- 2 Mögliche Auslöser
- 3 INT0, INT1 und die zugehörigen Register
- 4 Interrupts generell zulassen
- 5 Die Interruptvektoren
- 6 Beenden eines Interrupthandlers
- 7 Aufbau der Interruptvektortabelle
- 8 Beispiel
- 9 Besonderheiten des Interrupthandlers
- 10 Siehe auch

Definition

Bei bestimmten Ereignissen in Prozessoren wird ein sogenannter ***Interrupt*** ausgelöst. Interrupts machen es möglich, beim Eintreten eines Ereignisses sofort informiert zu werden, ohne permanent irgendeinen Status abzufragen, was teure Rechenzeit kosten würde. Dabei wird das Programm unterbrochen und ein Unterprogramm aufgerufen. Wenn dieses beendet ist, läuft das Hauptprogramm ganz normal weiter.

Mögliche Auslöser

Bei Mikrocontrollern werden Interrupts z. B. ausgelöst wenn:

- sich der an einem bestimmten Eingangs-Pin anliegende Wert von High auf Low ändert (oder umgekehrt)
- eine vorher festgelegte Zeitspanne abgelaufen ist (Timer)
- eine serielle Übertragung abgeschlossen ist (UART)
- ...

Der ATmega8 besitzt 18 verschiedene Interruptquellen. Standardmäßig sind diese alle deaktiviert und müssen über verschiedene IO-Register einzeln eingeschaltet werden.

INT0, INT1 und die zugehörigen Register

Wir wollen uns hier erst mal die beiden Interrupts **INT0** und **INT1** anschauen. INT0 wird ausgelöst, wenn sich der an PD2 anliegende Wert ändert, INT1 reagiert auf Änderungen an PD3.

Als erstes müssen wir die beiden Interrupts konfigurieren. Im Register **MCUCR** wird eingestellt, ob die Interrupts bei einer steigenden Flanke (low nach high) oder bei einer fallenden Flanke (high nach low) ausgelöst werden. Dafür gibt es in diesem Register die Bits **ISC00**, **ISC01** (betreffen INT0) und **ISC10** und **ISC11** (betreffen INT1).

Hier eine Übersicht über die möglichen Einstellungen und was sie bewirken:

ISC11 o. ISC01	ISC10 o. ISC00	Beschreibung
0	0	Low-Level am Pin löst den Interrupt aus
0	1	Jede Änderung am Pin löst den Interrupt aus
1	0	Eine fallende Flanke löst den Interrupt aus
1	1	Eine steigende Flanke löst den Interrupt aus

Danach müssen diese beiden Interrupts aktiviert werden, indem die Bits **INT0** und **INT1** im Register **GICR** auf **1** gesetzt werden.

Die Register **MCUCR** und **GICR** gehören zwar zu den IO-Registern, können aber nicht wie andere mit den Befehlen **cbi** und **sbi** verwendet werden. Diese Befehle wirken nur auf die IO-Register bis zur Adresse 0x1F (welches Register sich an welcher IO-Adresse befindet, steht in der Include-Datei, hier "m8def.inc", und im Datenblatt des Controllers). Somit bleiben zum Zugriff auf diese Register nur die Befehle **in** und **out** übrig.

Interrupts generell zulassen

Schließlich muss man noch das Ausführen von Interrupts allgemein aktivieren, was man durch einfaches Aufrufen des Assemblerbefehls **sei** bewerkstelligt.

Die Interruptvektoren

Woher weiß der Controller jetzt, welche Routine aufgerufen werden muss wenn ein Interrupt ausgelöst wird?

Wenn ein Interrupt auftritt, dann springt die Programmausführung an eine bestimmte Stelle im Programmspeicher. Diese Stellen sind festgelegt und können nicht geändert werden:

Nr.	Adresse	Interruptname	Beschreibung
1	0x000	RESET	Reset bzw. Einschalten der Stromversorgung
2	0x001	INT0	Externer Interrupt 0
3	0x002	INT1	Externer Interrupt 1
4	0x003	TIMER2_COMP	Timer/Counter2 Compare Match
5	0x004	TIMER2_OVF	Timer/Counter2 Overflow
6	0x005	TIMER1_CAPT	Timer/Counter1 Capture Event

7	0x006	TIMER1_COMPA	Timer/Counter1 Compare Match A
8	0x007	TIMER1_COMPB	Timer/Counter1 Compare Match B
9	0x008	TIMER1_OVF	Timer/Counter1 Overflow
10	0x009	TIMER0_OVF	Timer/Counter0 Overflow
11	0x00A	SPI_STC	SPI-Übertragung abgeschlossen
12	0x00B	USART_RX	USART-Empfang abgeschlossen
13	0x00C	USART_UDRE	USART-Datenregister leer
14	0x00D	USART_TX	USART-Sendung abgeschlossen
15	0x00E	ADC	AD-Wandlung abgeschlossen
16	0x00F	EE_RDY	EEPROM bereit
17	0x010	ANA_COMP	Analogkomparator
18	0x011	TWI	Two-Wire Interface
19	0x012	SPM_RDY	Store Program Memory Ready

So, wir wissen jetzt, dass der Controller zu Adresse 0x001 springt, wenn **INT0** auftritt. Aber dort ist ja nur Platz für einen Befehl, denn die nächste Adresse ist doch für INT1 reserviert. Wie geht das? Ganz einfach: Dort kommt ein Sprungbefehl rein, z. B. **rjmp interrupt0**. Irgendwo anders im Programm muss in diesem Fall eine Stelle mit *interrupt0:* gekennzeichnet sein, zu der dann gesprungen wird. Diese durch den Interrupt aufgerufene Routine nennt man ***Interrupthandler*** (engl. **I**nterrupt **S**ervice **R**outine, **ISR**).

Beenden eines Interrupthandlers

Und wie wird die Interruptroutine wieder beendet? Durch den Befehl **reti**. Wird dieser aufgerufen, dann wird das Programm ganz normal dort fortgesetzt, wo es durch den Interrupt unterbrochen wurde. Es ist dabei wichtig, daß hier der Befehl **reti** und nicht ein normaler **ret** benutzt wird. Wird ein Interrupt Handler betreten, so sperrt der Mikrocontroller automatisch alle weiteren Interrupts. Im Unterschied zu **ret**, hebt ein **reti** diese Sperre wieder auf.

Aufbau der Interruptvektortabelle

Jetzt müssen wir dem Assembler nur noch klarmachen, dass er unser **rjmp interrupt0** an die richtige Stelle im Programmspeicher schreibt, nämlich an den Interruptvektor für **INT0**. Dazu gibt es eine Assemblerdirektive. Durch **.org 0x001** sagt man dem Assembler, dass er die darauffolgenden Befehle ab Adresse 0x001 im Programmspeicher platzieren soll. Diese Stelle wird von **INT0** angesprungen.

Damit man nicht alle Interruptvektoren immer nachschlagen muss, sind in der Definitionsdatei m8def.inc einfach zu merkende Namen für die Adressen definiert. Statt 0x001 kann man z. B. einfach ***INT0addr*** schreiben. Das hat außerdem den Vorteil, dass man bei Portierung des Programms auf einen anderen AVR-Mikrocontroller nur die passende Definitionsdatei einbinden muss, und sich über evtl. geänderte Adressen für die Interruptvektoren keine Gedanken zu machen braucht.

Nun gibt es nur noch ein Problem: Beim Reset (bzw. wenn die Spannung eingeschaltet wird) wird das Programm immer ab der Adresse 0x000 gestartet. Deswegen muss an diese Stelle ein Sprungbefehl zum Hauptprogramm erfolgen, z. B. **rjmp RESET** um an die mit ***RESET:*** markierte Stelle zu springen.

Wenn man mehrere Interrupts verwenden möchte, kann man auch, anstatt jeden Interruptvektor einzeln mit .org an die richtige Stelle zu rücken, die gesamte Sprungtabelle ausschreiben:

```
.include "m8def.inc"

.org 0x000                      ; kommt ganz an den Anfang des Speichers
         rjmp RESET             ; Interruptvektoren überspringen
                                ; und zum Hauptprogramm
         rjmp EXT_INT0          ; IRQ0 Handler
         rjmp EXT_INT1          ; IRQ1 Handler
         rjmp TIM2_COMP
         rjmp TIM2_OVF
         rjmp TIM1_CAPT         ; Timer1 Capture Handler
         rjmp TIM1_COMPA        ; Timer1 CompareA Handler
         rjmp TIM1_COMPB        ; Timer1 CompareB Handler
         rjmp TIM1_OVF          ; Timer1 Overflow Handler
         rjmp TIM0_OVF          ; Timer0 Overflow Handler
         rjmp SPI_STC           ; SPI Transfer Complete Handler
         rjmp USART_RXC         ; USART RX Complete Handler
         rjmp USART_DRE         ; UDR Empty Handler
         rjmp USART_TXC         ; USART TX Complete Handler
         rjmp ADC               ; ADC Conversion Complete Interrupthandler
         rjmp EE_RDY            ; EEPROM Ready Handler
         rjmp ANA_COMP          ; Analog Comparator Handler
         rjmp TWSI              ; Two-wire Serial Interface Handler
         rjmp SPM_RDY           ; Store Program Memory Ready Handler

RESET:                          ; hier beginnt das Hauptprogramm
```

Hier ist es unbedingt nötig, bei unbenutzten Interruptvektoren statt des Sprungbefehls den Befehl **reti** (bzw. **reti nop**, wenn jmp 4 Byte lang ist) reinzuschreiben. Wenn man einen Vektor einfach weglässt stehen die nachfolgenden Sprungbefehle sonst alle an der falschen Adresse im Speicher.

Wer auf Nummer sicher gehen möchte kann aber auch alle Vektoren einzeln mit .org adressieren:

```
.include "m8def.inc"

.org 0x000
       rjmp RESET
.org INT0addr                   ; External Interrupt0 Vector Address
       reti
.org INT1addr                   ; External Interrupt1 Vector Address
       reti
.org OC2addr                    ; Output Compare2 Interrupt Vector Address
       reti
.org OVF2addr                   ; Overflow2 Interrupt Vector Address
       reti
.org ICP1addr                   ; Input Capture1 Interrupt Vector Address
       reti
.org OC1Aaddr                   ; Output Compare1A Interrupt Vector Address
       reti
.org OC1Baddr                   ; Output Compare1B Interrupt Vector Address
       reti
```

```
.org OVF1addr                   ; Overflow1 Interrupt Vector Address
       reti
.org OVF0addr                   ; Overflow0 Interrupt Vector Address
       reti
.org SPIaddr                    ; SPI Interrupt Vector Address
       reti
.org URXCaddr                   ; USART Receive Complete Interrupt Vector Address
       reti
.org UDREaddr                   ; USART Data Register Empty Interrupt Vector
Address
       reti
.org UTXCaddr                   ; USART Transmit Complete Interrupt Vector Address
       reti
.org ADCCaddr                   ; ADC Interrupt Vector Address
       reti
.org ERDYaddr                   ; EEPROM Interrupt Vector Address
       reti
.org ACIaddr                    ; Analog Comparator Interrupt Vector Address
       reti
.org TWIaddr                    ; Irq. vector address for Two-Wire Interface
       reti
.org SPMRaddr                    ; SPM complete Interrupt Vector Address
       reti

.org INT_VECTORS_SIZE
RESET:                          ; hier beginnt das Hauptprogramm
```

Statt die unbenutzten Interruptvektoren mit **reti** zu füllen könnte man sie hier auch einfach weglassen, da die **.org**-Direktive dafür sorgt dass jeder Vektor in jedem Fall am richtigen Ort im Speicher landet.

Beispiel

So könnte ein Minimal-Assemblerprogramm aussehen, das die Interrupts INT0 und INT1 verwendet. An die Interrupt Pins können zb Taster nach bewährter Manier angeschlossen werden. Die Interrupts werden auf fallende Flanke konfiguriert, da ja die Taster so angeschlossen sind, dass sie im Ruhezustand eine 1 liefern und bei einem Tastendruck nach 0 wechseln.

Download extinttest.asm

```
.include "m8def.inc"

.def temp = r16

.org 0x000
         rjmp main             ; Reset Handler
.org INT0addr
         rjmp int0_handler     ; IRQ0 Handler
.org INT1addr
         rjmp int1_handler     ; IRQ1 Handler


main:                          ; hier beginnt das Hauptprogramm

         ldi temp, LOW(RAMEND)
         out SPL, temp
         ldi temp, HIGH(RAMEND)
         out SPH, temp
```

```
        ldi temp, 0x00
        out DDRD, temp

        ldi temp, 0xFF
        out DDRB, temp

        ldi temp, (1<<ISC01) | (1<<ISC11) ; INT0 und INT1 auf fallende Flanke
konfigurieren
        out MCUCR, temp

        ldi temp, (1<<INT0) | (1<<INT1) ; INT0 und INT1 aktivieren
        out GICR, temp

        sei                   ; Interrupts allgemein aktivieren

loop:   rjmp loop             ; eine leere Endlosschleife

int0_handler:
        sbi PORTB, 0
        reti

int1_handler:
        cbi PORTB, 0
        reti
```

Für dieses Programm braucht man nichts weiter als eine LED an PB0 und je einen Taster an PD2 (INT0) und PD3 (INT1). Wie diese angeschlossen werden, steht in Teil 2 des Tutorials.

Die Funktion ist auch nicht schwer zu verstehen: Drückt man eine Taste, wird der dazugehörige Interrupt aufgerufen und die LED an- oder abgeschaltet. Das ist zwar nicht sonderlich spektakulär, aber das Prinzip sollte deutlich werden.

Meistens macht es keinen Sinn, Taster direkt an einen Interrupteingang anzuschließen. Das kann bisweilen sogar sehr schlecht sein, siehe Entprellung. Häufiger werden Interrupts in Zusammenhang mit dem UART verwendet, um z. B. auf ein empfangenes Zeichen zu reagieren. Wie das funktioniert, wird im Kapitel über den UART beschrieben.

Besonderheiten des Interrupthandlers

Der Interrupthandler kann ja mehr oder weniger zu jedem beliebigen Zeitpunkt unabhängig vom restlichen Programm aufgerufen werden. Dabei soll das restliche Programm auf keinen Fall durch den Interrupthandler negativ beeinflusst werden, das heißt das Hauptprogramm soll nach dem Beenden des Handlers weiterlaufen als wäre nichts passiert. Insbesondere muss deshalb darauf geachtet werden, dass im Interrupthandler Register, die vom Programmierer nicht ausschließlich nur für den Interrupthandler reserviert wurden, auf dem Stack gesichert und zum Schluss wieder hergestellt werden müssen.

Ein Register, das gerne übersehen wird, ist das ***Status Register***. In ihm merkt sich der Prozessor bestimmte Zustände von Berechnungen, z. B. ob ein arithmetischer Überlauf stattgefunden hat, ob das letzte Rechenergebnis 0 war, etc. Sobald ein Interrupthandler etwas komplizierter wird als im obigen Beispiel, tut man gut daran, das **SREG** Register auf jeden Fall zu sichern. Ansonsten kann das Hinzufügen von weiterem Code zum Interrupthandler schnell zum Boomerang werden: Die dann möglicherweise notwendige Sicherung des **SREG** Registers wird vergessen. Überhaupt

empfiehlt es sich, in diesen Dingen bei der Programmierung eines Interrupthandlers eher vorausschauend, übervorsichtig und konservativ zu programmieren. Wird dies getan, so vergeudet man höchstens ein bischen Rechenzeit. Im anderen Fall handelt man sich allerdings einen Super-GAU ein: Man steht dann vor einem Programm, das sporadisch nicht funktioniert und keiner weiss warum. Solche Fehler sind nur sehr schwer und oft nur mit einem Quäntchen Glück zu finden.

Im Beispiel wäre zwar das Sichern und Wiederherstellen der Register **temp** und **SREG** nicht wirklich notwendig, aber hier soll die grundsätzliche Vorgehensweise gezeigt werden:

```
.include "m8def.inc"

.def temp = r16

.org 0x000
         rjmp main            ; Reset Handler
.org INT0addr
         rjmp int0_handler    ; IRQ0 Handler
.org INT1addr
         rjmp int1_handler    ; IRQ1 Handler


main:                         ; hier beginnt das Hauptprogramm

         ldi temp, LOW(RAMEND)
         out SPL, temp
         ldi temp, HIGH(RAMEND)
         out SPH, temp

         ldi temp, 0x00
         out DDRD, temp

         ldi temp, 0xFF
         out DDRB, temp

         ldi temp, (1<<ISC01) | (1<<ISC11) ; INT0 und INT1 auf fallende Flanke
konfigurieren
         out MCUCR, temp

         ldi temp, (1<<INT0) | (1<<INT1) ; INT0 und INT1 aktivieren
         out GICR, temp

         sei                   ; Interrupts allgemein aktivieren

loop:    rjmp loop             ; eine leere Endlosschleife

int0_handler:
         push temp             ; Das SREG in temp sichern. Vorher
         in   temp, SREG       ; muss natürlich temp gesichert werden

         sbi PORTB, 0

         out SREG, temp        ; Die Register SREG und temp wieder
         pop temp              ; herstellen
         reti

int1_handler:
         push temp             ; Das SREG in temp sichern. Vorher
         in   temp, SREG       ; muss natürlich temp gesichert werden

         cbi PORTB, 0
```

```
out SREG, temp        ; Die Register SREG und temp wieder
pop temp              ; herstellen
reti
```

Siehe auch

- Interrupt: Interrupts in C
- Forumsbeitrag: Flexible Verwaltung von Interruptroutinen ohne zentrale Interrupttabelle

AVR-Tutorial: Vergleiche

Vergleiche und Entscheidungen sind in jeder Programmiersprache ein zentrales Mittel um den Programmfluss abhängig von Bedingungen zu kontrollieren. In einem AVR spielen dazu 4 Komponenten zusammen:

- Vergleichsbefehle
- die Flags im Statusregister
- bedingte Sprungbefehle
- andere Befehle, die die Flags im Statusregister beeinflussen, wie zb die meisten arithmetischen Funktionen

Der Zusammenhang ist dabei folgender: Die Vergleichsbefehle führen einen Vergleich durch, zum Beispiel zwischen zwei Registern oder zwischen einem Register und einer Konstante. Das Ergebnis des Vergleiches wird in den Flags abgelegt. Die bedingten Sprungbefehle werten die Flags aus und führen bei einem positiven Ergebnis den Sprung aus. Besonders der erste Satzteil ist wichtig! Den bedingten Sprungbefehlen ist es nämlich völlig egal, ob die Flags über Vergleichsbefehle oder über sonstige Befehle gesetzt wurden. Die Sprungbefehle werten einfach nur die Flags aus, wie auch immer diese zu ihrem Zustand kommen.

Inhaltsverzeichnis

- 1 Flags
  - 1.1 Carry (C)
  - 1.2 Zero (Z)
  - 1.3 Negative (N)
  - 1.4 Overflow (V)
  - 1.5 Signed (S)
  - 1.6 Half Carry (H)
  - 1.7 Transfer (T)
  - 1.8 Interrupt (I)
- 2 Vergleiche
  - 2.1 CP - Compare
  - 2.2 CPC - Compare with Carry
  - 2.3 CPI - Compare Immediate
- 3 Bedingte Sprünge
  - 3.1 Bedingte Sprünge für vorzeichenlose Zahlen
  - 3.2 Bedingte Sprünge für vorzeichenbehaftete Zahlen

    - 3.3 Sonstige bedingte Sprünge
    - 3.4 Selten verwendete bedingte Sprünge
- 4 Beispiele
    - 4.1 Entscheidungen
    - 4.2 Schleifenkonstrukte
- 5 Literatur

Flags

Die Flags sind Bits im Statusregister **SREG**. Ihre Aufgabe ist es, das Auftreten bestimmter Ereignisse, die während Berechnungen eintreten können, festzuhalten. Speicherbefehle (LD, LDI, ST, MOV, ...) haben auf dem AVR grundsätzlich keinen Einfluss auf das Statusregister. Will man den Inhalt eines Registers explizit testen (z. B. nach dem Laden aus dem SRAM), so kann man hierfür den TST-Befehl verwenden.

Bits im SREG

I	T	H	S	V	N	Z	C

Carry (C)

Das Carry Flag hält fest, ob es bei der letzten Berechnung einen Über- oder Unterlauf gab. Aber Achtung: Nicht alle arithmetischen Befehle verändern tatsächlich das Carry Flag. So haben zb die Inkrementier und Dekrementierbefehle keine Auswirkung auf dieses Flag.

Zero (Z)

Das Zero Flag hält fest, ob das Ergebnis der letzten 8-Bit Berechnung 0 war oder nicht.

Negative (N)

Spiegelt den Zustand des höchstwertigen Bits (Bit 7) der letzten 8-Bit-Berechnung wieder. In 2-Komplement Arithmetik bedeutet ein gesetztes Bit 7 eine negative Zahl, das Bit kann also dazu genutzt werden um festzustellen ob das Ergebnis einer Berechnung im Sinne einer 2-Komplement Arithmetik positiv oder negativ ist.

Overflow (V)

Dieses Bit wird gesetzt, wenn bei einer Berechnung mit 2-Komplement Arithmetik ein Überlauf (Unterlauf) stattgefunden hat. Dies entspricht einem Überlauf von Bit 6 ins Bit 7.

Der Übertrag, der bei der Addition/Subtraktion von Bit 6 auf Bit 7 auftritt, zeigt daher – wenn er vorhanden ist – an, dass es sich hier um einen Überlauf (Overflow) des Zahlenbereichs handelt und das Ergebnis falsch ist. Das ist allerdings nicht der Fall, wenn auch der Übertrag von Bit 7 nach Bit 8 (Carry) aufgetreten ist. Daher ist das Overflow-Flag die XOR-Verknüpfung aus den Übertrag von bit 6 nach Bit 7 und dem Carry.

Beispiele für die Anwendung des V-Flags finden sich in saturierter Arithmetik.

Signed (S)

Das Signed-Bit ergibt sich aus der Antivalenz der Flags N und V, also S = N XOR V. Mit Hilfe des Signed-Flags können vorzeichenbehaftete Werte miteinander verglichen werden. Ist nach einem Vergleich zweier Register S=1, so ist der Wert des ersten Registers kleiner dem zweiten (in der Signed-Darstellung). Damit entspricht das Signed-Flag gewissermaßen dem Carry-Flag für Signed-Werte. Es wird hauptsächlich für 'Signed' Tests benötigt. Daher auch der Name.

Half Carry (H)

Das Half Carry Flag hat die gleiche Aufgabe wie das Carry Flag, nur beschäftigt es sich mit einem Überlauf von Bit 3 nach Bit 4, also dem Übertrag zwischen dem oberen und unteren Nibble. Wie beim Carry-Flag gilt, dass das Flag nicht durch Inkrementieren bzw. Dekrementieren ausgelöst werden kann. Das Haupteinsatzgebiet ist der Bereich der BCD Arithmetik, bei der jeweils 4 Bits eine Stelle einer Dezimalzahl repräsentieren.

Transfer (T)

Das T-Flag ist kein Statusbit im eigentlichen Sinne. Es steht dem Programmierer als 1-Bit-Speicher zur Verfügung. Der Zugriff erfolgt über die Befehle Bit Load (BLD), Bit Store (BST), Set (SET) und Clear (CLT) und wird sonst von keinen anderen Befehlen beeinflusst. Damit können Bits von einer Stelle schnell an eine andere kopiert oder getestet werden.

Interrupt (I)

Das Interrupt Flag fällt hier etwas aus dem Rahmen; es hat nichts mit Berechnungen zu tun, sondern steuert ob Interrupts im Controller zugelassen sind (siehe AVR-Tutorial: Interrupts).

Vergleiche

Um einen Vergleich durchzuführen, wird intern eine Subtraktion der beiden Operanden durchgeführt. Das eigentliche Ergebnis der Subtraktion wird allerdings verworfen, es bleibt nur die neue Belegung der Flags übrig, die in weiterer Folge ausgewertet werden kann

CP - Compare

Vergleicht den Inhalt zweier Register miteinander. Prozessorintern wird dabei eine Subtraktion der beiden Register durchgeführt. Das eigentliche Subtraktionsergebnis wird allerdings verworfen, das Subtraktionsergebnis beeinflusst lediglich die Flags.

CPC - Compare with Carry

Vergleicht den Inhalt zweier Register, wobei das Carry Flag in den Vergleich mit einbezogen wird. Dieser Befehl wird für Arithmetik mit großen Variablen (16/32 Bit) benötigt. Siehe AVR-Tutorial: Arithmetik.

CPI - Compare Immediate

Vergleicht den Inhalt eines Registers mit einer direkt angegebenen Konstanten. Der Befehl ist nur auf die Register r16..r31 anwendbar.

Bedingte Sprünge

Die bedingten Sprünge werten immer bestimmte Flags im Statusregister **SREG** aus. Es spielt dabei keine Rolle, ob dies nach einem Vergleichsbefehl oder einem sonstigen Befehl gemacht wird. Entscheidend ist einzig und alleine der Zustand des abgefragten Flags. Die Namen der Sprungbefehle wurden allerdings so gewählt, daß sich im Befehlsnamen die Beziehung der Operanden direkt nach einem Compare Befehl widerspiegelt. Zu beachten ist auch, daß die Flags nicht nur durch Vergleichsbefehle verändert werden, sondern auch durch arithmetische Operationen, Schiebebefehle und logische Verknüpfungen. Da dieses Information wichtig ist, ist auch in der bei Atmel erhältlichen Übersicht über alle Assemblerbefehle bei jedem Befehl angegeben, ob und wie er Flags beeinflusst. Ebenso ist dort eine kompakte Übersicht aller bedingten Sprünge zu finden. Beachten muss man jedoch, dass die bedingten Sprünge maximal 64 Worte weit springen können.

Bedingte Sprünge für vorzeichenlose Zahlen

BRSH - Branch if Same or Higher

Der Sprung wird durchgeführt, wenn das Carry-Flag (C) nicht gesetzt ist. Wird dieser Branch direkt nach einer CP, CPI, SUB oder SUBI-Operation eingesetzt, so findet der Sprung dann statt, wenn der erste Operand größer oder gleich dem zweiten Operanden ist. BRSH ist identisch mit BRCC (Branch if Carry Cleared).

BRLO - Branch if Lower

Der Sprung wird durchgeführt, wenn das Carry-Flag (C) gesetzt ist. Wird dieser Branch direkt nach einer CP, CPI, SUB oder SUBI Operation eingesetzt, so findet der Sprung dann statt, wenn der erste Operand kleiner dem zweiten Operanden ist. BRLO ist identisch mit BRCS (Branch if Carry Set).

Bedingte Sprünge für vorzeichenbehaftete Zahlen

BRGE - Branch if Greater or Equal

Der Sprung wird durchgeführt, wenn das Signed-Flag (S) nicht gesetzt ist. Wird dieser Branch direkt nach einer CP, CPI, SUB oder SUBI eingesetzt, so findet der Sprung dann und nur dann statt, wenn der erste Operand größer oder gleich dem zweiten Operanden ist.

BRLT - Branch if Less Than

Der Sprung wird durchgeführt, wenn das Signed-Flag (S) gesetzt ist. Wird dieser Branch direkt nach einer CP, CPI, SUB oder SUBI Operation eingesetzt, so findet der Sprung dann und nur dann statt, wenn der erste Operand kleiner als der zweite Operand ist.

BRMI - Branch if Minus

Der Sprung wird durchgeführt, wenn das Negativ-Flag (N) gesetzt ist, das Ergebnis der letzten Operation also negativ war.

BRPL - Branch if Plus

Der Sprung wird durchgeführt, wenn das Negativ Flag (N) nicht gesetzt ist, das Ergebnis der letzten Operation also positiv war (einschießlich Null).

Sonstige bedingte Sprünge

BREQ - Branch if Equal

Der Sprung wird durchgeführt, wenn das Zero-Flag (Z) gesetzt ist. Ist nach einem Vergleich das Zero Flag gesetzt, lieferte die interne Subtraktion also 0, so waren beide Operanden gleich.

BRNE - Branch if Not Equal
Der Sprung wird durchgeführt, wenn das Zero-Flag (Z) nicht gesetzt ist. Ist nach einem Vergleich das Zero Flag nicht gesetzt, lieferte die interne Subtraktion also nicht 0, so waren beide Operanden verschieden.

BRCC - Branch if Carry Flag is Cleared
Der Sprung wird durchgeführt, wenn das Carry-Flag (C) nicht gesetzt ist. Dieser Befehl wird oft für Arithmetik mit grossen Variablen (16/32 Bit) bzw. im Zusammenhang mit Schiebeoperationen verwendet. BRCC ≡ BRSH

BRCS - Branch if Carry Flag is Set
Der Sprung wird durchgeführt, wenn das Carry-Flag (C) gesetzt ist. Die Verwendung ist sehr ähnlich zu BRCC. BRCS ≡ BRLO

Selten verwendete bedingte Sprünge

BRHC - Branch if Half Carry Flag is Cleared
Der Sprung wird durchgeführt, wenn das Half-Carry Flag (H) nicht gesetzt ist.

BRHS - Branch if Half Carry Flag is Set
Der Sprung wird durchgeführt, wenn das Half-Carry Flag (H) gesetzt ist.

BRID - Branch if Global Interrupt is Disabled
Der Sprung wird durchgeführt, wenn das Interrupt-Flag (I) nicht gesetzt ist.

BRIE - Branch if Global Interrupt is Enabled
Der Sprung wird durchgeführt, wenn das Interrupt-Flag (I) gesetzt ist.

BRTC - Branch if T Flag is Cleared
Der Sprung wird durchgeführt, wenn das T-Flag nicht gesetzt ist.

BRTS - Branch if T Flag is Set
Der Sprung wird durchgeführt, wenn das T-Flag gesetzt ist.

BRVC - Branch if Overflow Cleared
Der Sprung wird durchgeführt, wenn das Overflow-Flag (V) nicht gesetzt ist.

BRVS - Branch if Overflow Set
Der Sprung wird durchgeführt, wenn das Overflow-Flag (V) gesetzt ist.

Beispiele

Entscheidungen

In jedem Programm kommt früher oder später das Problem, die Ausführung von Codeteilen von irgendwelchen Zahlenwerten, die sich in anderen Registern befinden abhängig zu machen. Sieht beispielweise die Aufgabe vor, daß Register r18 auf 0 gesetzt werden soll, wenn im Register r17 der

Zahlenwert 25 enthalten ist und in allen anderen Fällen soll r18 auf 123 gesetzt werden, dann lautet der Code

```
    cpi     r17, 25         ; vergleiche r17 mit der Konstante 25
    brne    nicht_gleich    ; wenn nicht gleich, dann mach bei nicht_gleich
weiter
    ldi     r18, 0          ; hier stehen nun Anweisungen für den Fall
                            ; dass R17 gleich 25 ist
    rjmp    weiter          ; meist will man den anderen Zweig nicht
durchlaufen, darum der Sprung
nicht_gleich:
    ldi     r18,123         ; hier stehen nun Anweisungen für den Fall
                            ; dass R17 ungleich 25 ist
weiter:                     ; hier geht das Programm weiter
```

In ähnlicher Weise können die anderen bedingten Sprungbefehle eingesetzt werden, um die üblicherweise vorkommenden Vergleiche auf Gleichheit, Ungleichheit, Größer, Kleiner zu realisieren.

Schleifenkonstrukte

Ein immer wiederkehrendes Muster in der Programmierung ist eine **Schleife**. Die einfachste Form einer Schleife ist die **Zählschleife**. Dabei wird ein Register von einem Startwert ausgehend eine gewisse Anzahl erhöht, bis ein Endwert erreicht wird.

```
    ldi     r17, 10         ; der Startwert sei in diesem Beispiel 10
loop:
                            ; an dieser Stelle stehen die Befehle, welche
innerhalb der Schleife
                            ; mehrfach ausgeführt werden sollen

    inc     r17             ; erhöhe das Zaehlregister
    cpi     r17, 134        ; mit dem Endwert vergleichen
    brne    loop            ; und wenn der Endwert noch nicht erricht ist
                            ; wird bei der Marke loop ein weiterer
Schleifendurchlauf ausgeführt
```

Sehr oft ist es auch möglich das Konstrukt umzudrehen. Anstatt von einem Startwert aus zu inkrementieren genügt es die Anzahl der gewünschten Schleifendurchläufe in ein Register zu laden und dieses Register zu dekrementieren. Dabei kann man von der Eigenschaft der Dekrementieranweisung gebrauch machen, das **Zero Flag (Z)** zu beeinflussen. Ist das Ergebnis des Dekrements 0, so wird das **Zero Flag (Z)** gesetzt, welches wiederum in der nachfolgenden **BRNE** Anweisung für einen bedingen Sprung benutzt werden kann. Das vereinfacht die Schleife und spart eine Anweisung sowie einen Takt Ausführungzeit.

```
    ldi     r17, 124        ; Die Anzahl der Wiederholungen in ein Register
laden
loop:
                            ; an dieser Stelle stehen die Befehle, welche
innerhalb der Schleife
                            ; mehrfach ausgeführt werden sollen

    dec     r17             ; Schleifenzähler um 1 verringern, dabei wird das
Zero Flag beeinflusst
```

```
    brne    loop            ; wenn r17 noch nicht 0 geworden ist -> Schleife
wiederholen
```

Literatur

AVR Instruction Set

AVR-Tutorial: Mehrfachverzweigung

Inhaltsverzeichnis

- 1 Einleitung
- 2 Einfacher Ansatz
- 3 Sprungtabelle
- 4 Lange Sprungtabelle
- 5 Z-Pointer leicht verständlich

Einleitung

Oft ist es in einem Programm notwendig, eine Variable auf mehrere Werte zu prüfen und abhängig vom Ergebnis verschiedene Aktionen auszulösen. Diese Konstruktion nennt man Mehrfachverzweigung. In einem Struktogramm sieht das so aus.

Prüfe X					
X=1	X=17	X=33	X=9	X=22	sonst

In C gibt es direkt dafür eine Konstruktion namens **switch**.

```
switch (variable) {
  case 1:       // Anweisungen für diesen Zweig, wenn variable == 1
    break;
  case 17:      // Anweisungen für diesen Zweig, wenn variable == 17
    break;
  case 33:      // Anweisungen für diesen Zweig, wenn variable == 33
    break;
  case 9:       // Anweisungen für diesen Zweig, wenn variable == 9
    break;
  case 22:      // Anweisungen für diesen Zweig, wenn variable == 22
    break;
  default:      // Anweisungen wenn keine der oben definierten Bedingungen
erfüllt ist
    break;
}
```

In Assembler muss man so etwas "zu Fuß" programmieren. Die verschiedene Lösungen sollen hier betrachtet werden.

Einfacher Ansatz

Im einfachsten Fall verwendet man eine lange Kette von **cpi** und **breq** Befehlen. Für jeden Zweig benötigt man zwei Befehle.

```
; Mehrfachverzeigung Version A

; Einfacher Ansatz, mit vielen CPI

start_vergleich:

    cpi     r16,1
    brne    zweig_0

; hier stehen jetzt alle Anweisungen für diesen Zweig r16=1

    rjmp    ende_vergleich
zweig_0:
    cpi     r16,17
    brne    zweig_1

; hier stehen jetzt alle Anweisungen für diesen Zweig r16=17

    rjmp    ende_vergleich
zweig_1:
    cpi     r16,33
    brne    zweig_2

; hier stehen jetzt alle Anweisungen für diesen Zweig r16=33

    rjmp    ende_vergleich
zweig_2:
    cpi     r16,9
    brne    zweig_3

; hier stehen jetzt alle Anweisungen für diesen Zweig r16=9

    rjmp    ende_vergleich
zweig_3:
    cpi     r16,22
    brne    kein_Treffer

; hier stehen jetzt alle Anweisungen für diesen Zweig r16=22

    rjmp    ende_vergleich
kein_Treffer:

; hier stehen jetzt alle Anweisungen für den Fall, dass keiner der Vergleiche
erfolgreich war

ende_vergleich:

    rjmp    ende_vergleich          ; nur für Simulationszwecke! ENTFERNEN!
```

Eigenschaften

- Programmspeicherbedarf: 6*N Bytes (N = Anzahl der Zweige)
- Laufzeit: n*3-1, Nicht gefunden: N*3 (N = Anzahl der Zweige, n = Ausgewählter Zweig)

Vorteile

- leicht verständlich
- Es können beliebige Vergleichswerte geprüft werden

Nachteile

- relativ hoher Programmspeicherbedarf
- die Größe der Zweige ist stark begrenzt, weil der Befehl **breq** maximal 63 Worte weit springen kann!
- die einzelnen Zweige haben unterschiedliche Durchlaufzeiten, der letzte Zweig ist am langsamsten
- nur bedingt übersichtlicher Quellcode

Sprungtabelle

Oft liegen die einzelnen Vergleichswerte nebeneinander (z. B. 7..15), z. B. bei der Übergabe von Parametern, Zustandsautomaten, Menueinträgen etc. . In so einem Fall kann man mittels einer **Sprungtabelle** das Programm verkürzen, beschleunigen und übersichtlicher gestalten.

```
.include "m8def.inc"

; Mehrfachverzweigung Version B

; Clevere Version mit Sprungtabelle
; minimum und maximum sind auf 0..255 begrenzt!

.equ minimum = 3
.equ maximum = 7

start_vergleich:

    subi    r16,minimum                 ; Nullpunkt verschieben
    cpi     r16,(maximum-minimum+1)     ; Index auf Maximum prüfen
    brsh    kein_Treffer                ; Index zu gross -> Fehler
    ldi     ZL,low(Sprungtabelle)       ; Tabellenzeiger laden, 16 Bit
    ldi     ZH,high(Sprungtabelle)
    add     ZL,r16                      ; Index addieren, 16 Bit
    ldi     r16,0
    adc     ZH,r16
    ijmp                                ; indirekter Sprung in Sprungtabelle

kein_treffer:

; hier stehen jetzt alle Anweisungen für den Fall, dass keiner der Vergleiche
erfolgreich war

    rjmp    ende_vergleich

Sprungtabelle:
    rjmp    zweig_0
    rjmp    zweig_1
    rjmp    zweig_2
    rjmp    zweig_3
    rjmp    zweig_4

zweig_0:

; hier stehen jetzt alle Anweisungen für diesen Zweig
```

```
    rjmp    ende_vergleich
zweig_1:
; hier stehen jetzt alle Awneisungen für diesen Zweig
    rjmp    ende_vergleich
zweig_2:
; hier stehen jetzt alle Anweisungen für diesen Zweig
    rjmp    ende_vergleich
zweig_3:
; hier stehen jetzt alle Anweisungen für diesen Zweig
    rjmp    ende_vergleich
zweig_4:
; hier stehen jetzt alle Anweisungen für diesen Zweig
    rjmp    ende_vergleich
ende_vergleich:
; hier geht das Programm weiter
    rjmp    ende_vergleich          ; nur für Simulationszwecke! ENTFERNEN!
```

Programmbeschreibung

Wie ist dieses Programm nun zu verstehen? Das Prinzip beruht darauf, daß in einer gleichmässigen Tabelle Sprungbefehle auf einzelne Programmzweige abgelegt werden. Das ist praktisch genauso wie der AVR Interrupts verarbeitet. Über einen Index (0...N) wird ein Sprungbefehl ausgewählt und ausgeführt.Der entscheidende Befehl dazu ist **ijmp**.

Zunächst muss der Wertebereich, auf welchen die Variable geprüft werden soll (minimum bis maximum), normiert werden (0 bis (Maximum-Minimum)). Dazu wird einfach das Minimum subtrahiert.

```
    subi    r16,minimum                 ; Nullpunkt verschieben
```

Danach muss geprüft werden, ob der maximale Index nicht überschritten wird. Denn ein Sprung auf nichtexistierende Einträge oberhalb der Sprungtabelle wäre fatal!

```
    cpi     r16,(maximum-minimum+1)     ; Index auf Maximum prüfen
    brsh    kein_Treffer                ; Index zu gross -> Fehler
```

Danach muss der indirekte Sprung vorbereitet werden. Dazu wird die Adresse der Sprungtabelle in das Z-Register geladen, welches ein 16 Bit Register ist und gleichbedeutend mit r30 und r31.

```
    ldi     ZL,low(Sprungtabelle)       ; Tabellenzeiger laden, 16 Bit
    ldi     ZH,high(Sprungtabelle)
```

Danach muss der Index addiert werden, dies ist eine 16-Bit Addition.

```
    add     ZL,r16                      ; Index addieren, 16 Bit
    ldi     r16,0
    adc     ZH,r16
```

Zu guter Letzt wird der indirekte Sprung in die Sprungtabelle ausgeführt.

```
    ijmp                                ; indirekter Sprung in Sprungtabelle
```

In der Sprungtabelle wird dann zum jeweiligen Zweig verzweigt.

```
Sprungtabelle:
    rjmp    zweig_0
    rjmp    zweig_1
    rjmp    zweig_2
    rjmp    zweig_3
    rjmp    zweig_4
```

Der Zweig für einen ungültigen Index folgt direkt nach dem **ijmp**, weil der Befehl **brsh** nur maximal 63 Worte weit springen kann.

Eigenschaften

- Programmspeicherbedarf: 4*N +18 Bytes (N = Anzahl der Zweige)
- Laufzeit: 12, Nicht gefunden: 4
- maximale Gesamtgröße der Zweige wird durch den Befehl rjmp begrenzt (+/-4kB). Das sollte aber nur in sehr wenigen Fällen ein Problem sein (Man wird kaum einen AVR mit 8 kB FLASH mit einer einzigen Mehrfachverzweigung füllen!)

Vorteile

- relativ niedriger Programmspeicherbedarf
- die einzelnen Zweige haben unabhängig von der Grösse der Sprungtabelle eine konstante und kurze Durchlaufzeit von 12 Takten.
- übersichtlicher Quellcode

Nachteile

- Die Vergleichswerte müssen lückenlos aufeinander folgen

Lange Sprungtabelle

Wenn man doch mal eine GIGA-Mehrfachverzweigung braucht, dann hilft die Version C.

```
.include "m16def.inc"

; Mehrfachverzweigung Version C

; Clevere Version mit langer Sprungtabelle
; funktioniert nur mit AVRs mit mehr als 8KB FLASH
; minimum und maximum sind auf 0..127 begrenzt!
```

```
.equ minimum = 3
.equ maximum = 7

start_vergleich:

    subi    r16,minimum                 ; Nullpunkt verschieben
    cpi     r16,(maximum-minimum+1)     ; Index auf Maximum prüfen
    brsh    kein_Treffer                ; Index zu gross -> Fehler
    ldi     ZL,low(Sprungtabelle*2)     ; Tabellenzeiger laden, 16 Bit
    ldi     ZH,high(Sprungtabelle*2)
    lsl     r16                         ; Index mit 2 multiplizieren
    add     zl,r16                      ; Index addieren, 16 Bit
    ldi     r16,0
    adc     zh,r16
    lpm     r16,Z+                      ; Low Byte laden und Pointer erhöhen
    lpm     ZH,Z                        ; zweites Byte laden
    mov     ZL,r16                      ; erstes Byte in Z-Pointer kopieren
    ijmp                                ; indirekter Sprung

kein_treffer:

; hier stehen jetzt alle Anweisungen für den Fall, dass keiner der Vergleiche
erfolgreich war

    jmp     ende_vergleich

Sprungtabelle:
.dw zweig_0
.dw zweig_1
.dw zweig_2
.dw zweig_3
.dw zweig_4

zweig_0:

; hier stehen jetzt alle Anweisungen für diesen Zweig

    jmp    ende_vergleich

zweig_1:

; hier stehen jetzt alle Awneisungen für diesen Zweig

    jmp    ende_vergleich

zweig_2:

; hier stehen jetzt alle Anweisungen für diesen Zweig

    jmp    ende_vergleich

zweig_3:

; hier stehen jetzt alle Anweisungen für diesen Zweig

    jmp    ende_vergleich

zweig_4:

; hier stehen jetzt alle Anweisungen für diesen Zweig

ende_vergleich:
```

```
; hier geht das Programm weiter

    jmp    ende_vergleich          ; nur für Simulationszwecke! ENTFERNEN!
```

Programmbeschreibung

Diese Version ist der Version B sehr ähnlich. Der Unterschied besteht darin, daß in Version B die Sprungtabelle mit Sprungbefehlen gefüllt ist (rjmp) während in Version C die Startadressen der Funktionen ablegt sind. D.H. man kann nicht in die Sprungtabelle springen, sondern muss sich mit Hilfe des Index die richtige Adresse aus der Sprungtabelle lesen und mit **ijmp** anspringen. Klingt sehr ähnlich, ist aber dennoch verschieden.

Die ersten drei Befehle sind identisch, es wird der Index normiert und auf das Maximum geprüft.

```
    subi    r16,minimum                 ; Nullpunkt verschieben
    cpi     r16,(maximum-minimum+1)     ; Index auf Maximum prüfen
    brsh    kein_Treffer                ; Index zu gross -> Fehler
```

Die nächsten zwei Befehle laden wieder die Anfangsadresse der Sprungtabelle. Doch halt, hier wird die Adresse der Sprungtabelle mit zwei multipliziert. Des Rätsels Lösung gibt es weiter unten.

```
    ldi     ZL,low(Sprungtabelle*2)     ; Tabellenzeiger laden, 16 Bit
    ldi     ZH,high(Sprungtabelle*2)
```

Der Index wird ebenfalls mit zwei multipliziert.

```
    lsl     r16                         ; Index mit 2 multiplizieren
```

Danach erfolgt eine 16-Bit Addition.

```
    add     zl,r16                      ; Index addieren, 16 Bit
    ldi     r16,0
    adc     zh,r16
```

Nun zeigt unser Z-Zeiger auf den richtigen Tabelleneintrag. Jetzt müssen zwei Bytes aus dem FLASH geladen werden. Das geschieht mit Hilfe des **lpm**-Befehls (**L**oad **P**rogram **M**emory). Hier wird die erweiterte Version des lpm-Befehls verwendet, wie sie nur auf grösseren AVRs verfügbar ist. Dabei wird ein Byte in Register r16 geladen und gleichzeitig der Z-Pointer um eins erhöht. Damit zeigt er wunderbar auf das nächste Byte, welches auch geladen werden muss.

```
    lpm     r16,Z+                      ; Low Byte laden und Zeiger erhöhen
```

Der zweite lpm-Befehl ist etwas ungewöhnlich, denn er überschreibt einen Teil des Z-Pointers! In den meisten Programmen wäre das ein Schuss ins Knie (Programmierfehler!), da wir aber den Z-Pointer danach sowieso mit neuen Daten laden ist das OK.

```
    lpm     ZH,Z                        ; zweites Byte laden
```

Das zuerst gelesene Byte wird in den Z-Pointer kopiert. Nun steht die Startadresse des gewählten Zweigs im Z-Pointer.

```
    mov     ZL,r16                      ; erstes Byte in Z-zeiger kopieren
```

Zu guter Letzt wird der indirekte Sprung ausgeführt und bringt uns direkt in den Programmzweig.

```
    ijmp                                ; indirekter Sprung direkt in den
Programmzweig
```

Der Zweig für einen ungültigen Index folgt direkt nach dem **ijmp**, weil der Befehl **brsh** nur maximal 63 Worte weit springen kann.

Eigenschaften

- Programmspeicherbedarf: 6*N +26 Bytes (N = Anzahl der Zweige)
- unbegrenzte Sprungweite

Vorteile

- relativ niedriger Programmspeicherbedarf
- die einzelnen Zweige haben unabhängig von der Grösse der Sprungtabelle eine konstante und kurze Durchlaufzeit von 18 Takten
- übersichtlicher Quellcode

Nachteile

- Die Vergleichswerte müssen lückenlos aufeinander folgen
- geringfügig höherer Programmspeicherbedarf (8 Byte mehr) und grössere Durchlaufzeit (6 Takte mehr)als Version B

Z-Pointer leicht verständlich

Auf den ersten Blick scheint es sonderbar, daß Version B die Adresse der Sprungtabelle direkt lädt, während Version C sowohl Anfangsadresse als auch Index mit zwei multipliziert. Warum ist das so?

Version B verwendet nur den Befehl **ijmp**. Dieser erwartet im Z-Register eine Adresse zur Programmausführung, eine **Wort-Adresse**. Da der Programmspeicher des AVR 16 Bit breit ist (=1 Wort = 2 Bytes), werden nur Worte adressiert, nicht jedoch Bytes! Genauso arbeitet der Assembler. Jedes Label entspricht einer Wort-Adresse. Damit kann man mit einer 12 Bit-Adresse 4096 Worte adressieren (=8192 Bytes). Wenn man sich die Befehle der einzelnen AVRs anschaut wird klar, daß alle AVRs mit 8KB und weniger FLASH nur die Befehle rjmp und rcall besitzen. Denn sie brauchen nicht mehr! (Hinweis: Der Atmega8 besitzt die Befehle ijmp und icall).

Mit 12 Adressbits, welche direkt in einem Wort im Befehl rjmp bzw. rcall kodiert sind, kann der gesamte Programmspeicher erreicht werden. Größere AVRs besitzen call und jmp, dort ist die Adresse als 22 bzw. 16 Bit Zahl kodiert, deshalb brauchen diese Befehle auch 2 Worte Programmspeicher.

Der Befehl **lpm** dient zum Laden einzelner Bytes aus dem Programmspeicher. Das ist vor allem für Tabellen mit konstanten Werten sehr nützlich (7-Segmentdekoder, Zeichensätze, Kennlinien, Parameter Texte, etc.) Doch wie kommt man nun in dem wortweise adressierten Programmspeicher

an einzelne Bytes? Ganz einfach. Der AVR "mogelt" hier und erwartet im Z-Register eine **Byte-Adresse**. Von dieser Adresse bilden die Bits 15..1 die Wortadresse, welche zur Adressierung des Programmspeichers verwendet wird. Bit 0 entscheidet dann, ob das hoch- oder niederwertige Byte in das Zielregister kopiert werden soll (0=niederwertiges Byte; 1=höherwertiges Byte).

Darum muss bei Verwendung des Befehls lpm die Anfangsadresse immer mit zwei multipliziert werden.

```
ldi     ZL,low(Sprungtabelle*2)     ; Tabellenzeiger laden, 16 Bit
ldi     ZH,high(Sprungtabelle*2)
```

In Version C muss zusätzlich der Index mit zwei multipliziert werden, weil jeder Tabelleneintrag (Adresse des Programmzweigs) ein Wort breit ist. Damit wird aus einem Index von 0,1,2,3,4 ein Offset von 0,2,4,6,8.

VR-Tutorial: UART

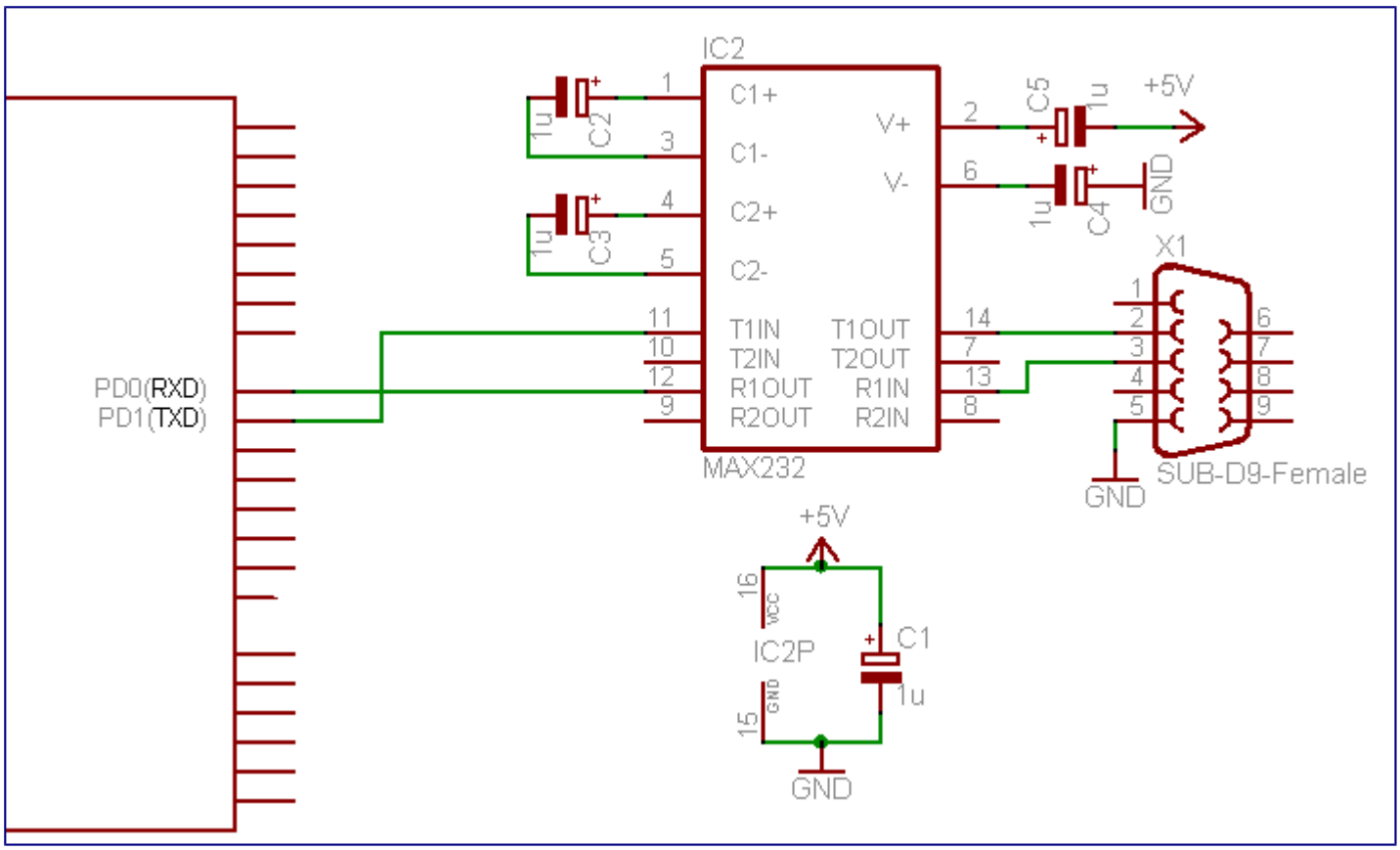
Wie viele andere Controller besitzen die meisten AVRs einen **UART** (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter). Das ist eine serielle Schnittstelle, die meistens zur Datenübertragung zwischen Mikrocontroller und PC genutzt wird. Zur Übertragung werden zwei Pins am Controller benötigt: **TXD** und **RXD**. Über **TXD** ("Transmit Data") werden Daten gesendet, **RXD** ("Receive Data") dient zum Empfang von Daten.

Inhaltsverzeichnis

- 1 Hardware
- 2 Software
  - 2.1 UART konfigurieren
  - 2.2 Senden von Zeichen
  - 2.3 Senden von Zeichenketten
  - 2.4 Empfangen von Zeichen per Polling
  - 2.5 Empfangen von Zeichen per Interrupt
- 3 Handshake
  - 3.1 Hardware Handshake
  - 3.2 Software Handshake
- 4 Weblinks

Hardware

Um den UART des Mikrocontrollers zu verwenden, muss der Versuchsaufbau um folgende Bauteile erweitert werden:

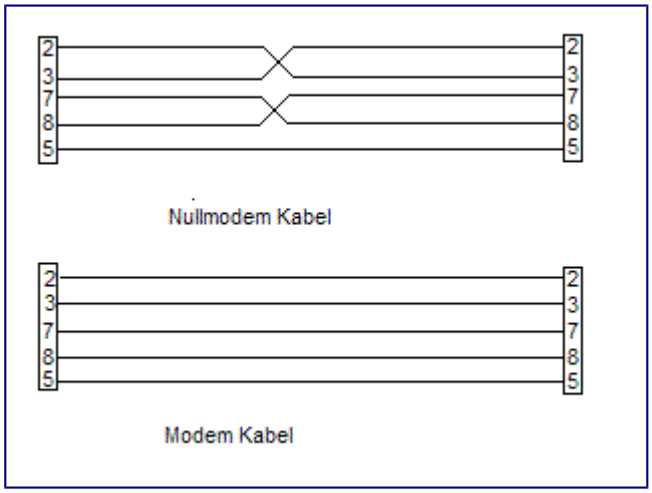


UART/MAX232 Standardbeschaltung

Auf dem Board vom Shop sind diese Bauteile bereits enthalten, man muss nur noch die Verbindungen zwischen MAX232 (IC2) und AVR herstellen wie im Bild zu sehen.

- Der MAX232 ist ein Pegelwandler, der die -12V/+12V Signale an der seriellen Schnittstelle des PCs zu den 5V/0V des AVRs kompatibel macht.
- C1 ist ein kleiner Elektrolyt-, Tantal- oder Keramikkondensator, wie er immer wieder zur Entkopplung der Versorgungsspannungen an digitalen ICs verwendet wird.
- Die vier Kondensatoren C2..C5 sind Elektrolyt-, Tantal- oder Keramikkondensatoren. Falls Elkos oder Tantals verwendet werden, auf die richtige Polung achten! Der exakte Wert ist hier relativ unkritisch, in der Praxis sollte alles von ca. 1µF bis 47µF mit einer Spannungsfestigkeit von 16V und höher funktionieren.
- X1 ist ein weiblicher 9-poliger SUB-D-Verbinder.
- Die Verbindung zwischen PC und Mikrocontroller erfolgt über ein 9-poliges Modem-Kabel (also ein *Verlängerungskabel*, **kein Nullmodem-Kabel!)**, das an den seriellen Port des PCs angeschlossen wird. Bei einem Modem-Kabel sind die Pins 2 und 3 des einen Kabelendes mit den Pins 2 und 3 des anderen Kabelendes durchverbunden. Bei einem Nullmodem-Kabel sind die Leitungen gekreuzt, sodass Pin 2 von der einen Seite mit Pin 3 auf der anderen Seite verbunden ist und umgekehrt.
- Als Faustregel kann man annehmen: Befinden sich an den beiden Enden des Kabels die gleiche Art von Anschlüssen (Männchen = Stecker; Weibchen = Buchse), dann benötigt man ein gekreuztes, also ein Nullmodem-Kabel. Am PC-Anschluss selbst befindet sich ein Stecker, also ein Männchen, sodaß am Kabel auf dieser Seite eine Buchse (also ein Weibchen) sitzen muss. Da am AVR laut obigem Schaltbild eine Buchse verbaut wird, muss

daher an diesem Ende des Kabels ein Stecker sitzen. Das Kabel hat daher an einem Ende einen Stecker und am anderen Ende eine Buchse und ist daher ein normales Modem-Kabel (= nicht gekreuzt).



Kabelbeschaltungen

Software

UART konfigurieren

Als erstes muss die gewünschte Baudrate im Register **UBRR** festgelegt werden. Der in dieses Register zu schreibende Wert errechnet sich nach der folgenden Formel:

$$\mathrm{UBRR} = \frac{\text{Taktfrequenz (in Hz)}}{16 \cdot \text{Baudrate}} - 1$$

Beim AT90S4433 kann man den Wert direkt in das Register **UBRR** laden, beim ATmega8 gibt es für **UBRR** zwei Register: **UBRRL** (Low-Byte) und **UBRRH** (High-Byte). Bei Baudraten über etwa 3900 Bit/s (gilt nur bei Verwendung eines Takts von 16 MHz) steht in **UBRRH** 0, da der berechnete Wert kleiner als 256 ist und somit in **UBRRL** alleine passt. Beachtet werden muss, dass das Register **UBRRH** vor dem Register **UBRRL** beschrieben werden muss. Der Schreibzugriff auf **UBRRL** löst das Neusetzen des internen Taktteilers aus.

WICHTIGER HINWEIS 1

Es empfiehlt sich statt der oben genannten Formel, die Formel der Codebeispiele zu verwenden:

$$\mathrm{UBRR} = \frac{\text{Taktfrequenz (in Hz)} + (\text{Baudrate} \cdot 8)}{(\text{Baudrate} \cdot 16)} - 1 \quad \left(= \frac{\text{Taktfrequenz (in Hz)}}{16 \cdot \text{Baudrate}} - 0. \right.$$

Beispiel: Bei einem ATMega mit 16MHz und 115200 Baud ist der Wert laut Datenblatt UBBRL=8. Rechnet man mit der Formel UBRRL=(F_CPU / (UART_BAUDRATE* 16L) - 1) ergibt sich ein Wert von 7,680555 und im UBRRL Register steht somit eine **7 statt einer 8**. Die Verwendung der Formel aus dem Codebeispiel ergibt 8.180555 und im UBRRL Register steht somit der richtige Wert - nämlich 8

WICHTIGER HINWEIS 2

Auf Grund permanent wiederkehrender Nachfrage sei hier **AUSDRÜCKLICH** darauf hingewiesen, dass bei Verwendung des UART im asynchronen Modus dringend ein Quarz oder Quarzoszillator verwendet werden sollte! Der interne RC-Oszillator der AVRs ist recht ungenau! Damit kann es in Ausnahmefällen funktionieren, muss es aber nicht! Auch ist der interne Oszillator temperaturempfindlich. Damit hat man dann den schönen Effekt, dass eine UART-Schaltung die im Winter noch funktionierte, im Sommer den Dienst verweigert.

Außerdem muss bei der Berechnung von **UBRR** geprüft werden, ob mit der verwendeten Taktfrequenz die gewünschte Baudrate mit einem Fehler von <1% generiert werden kann. Das Datenblatt bietet hier sowohl die Formel als auch Tabellen unter der Überschrift des U(S)ART an.

$$\text{Fehler}_{\text{Baudrate}}[\%] = \left(\frac{\text{UBRR}_{\text{gerundet}} + 1}{\text{UBRR}_{\text{genau}} + 1} - 1\right) \cdot 100$$

Siehe auch Baudratenquarz

Wer es ganz einfach haben will, nimmt die folgenden Macros. Die rechnen sogar den Fehler aus und brechen die Assemblierung ggf. ab. Das ist dann praktisch idiotensicher.

```
.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif
```

Wer dennoch den internen RC-Oszillator verwenden will, muss diesen kalibrieren. Näheres findet man dazu im Datenblatt, Stichwort Register OSCCAL.

Um den Sendekanal des UART zu aktivieren, muss das Bit **TXEN** im UART Control Register **UCSRB** auf 1 gesetzt werden.

Danach kann das zu sendende Byte in das Register **UDR** eingeschrieben werden - vorher muss jedoch sichergestellt werden, dass das Register leer ist, die vorhergehende Übertragung also schon abgeschlossen wurde. Dazu wird getestet, ob das Bit **UDRE** ("UART Data Register Empty") im Register **UCSRA** auf 1 ist.

Genaueres über die UART-Register findet man im Datenblatt des Controllers.

An dieser Stelle sei noch folgendes angemerkt: Das **UDRE**-Bit sagt nichts darüber aus, ob der Controller immer noch damit beschäftigt ist, Daten zu senden. Da das Senderegister mehrfach

gepuffert ist, wird **UDRE** bereits gesetzt, obwohl das letzte Zeichen den AVR noch nicht komplett verlassen hat. Dies kann insbesondere bei der Verwendung von Sleep-Modes ein Problem werden, wenn der Controller schlafen gelegt wird, bevor das letzte Zeichen versendet wurde, da dies gezwungenermassen zu einem Frame-Error beim Empfänger führen wird. Um sicher zu gehen, dass der UART nicht mehr beschäftigt ist, kann das Bit **TXC** ("UART Transmit complete") getestet werden. Dieses wird jedoch wirklich erst nach dem Senden eines Zeichens gesetzt, beinhaltet also auch nach dem Systemstart eine 0, obwohl der Controller nichts sendet.

Der ATmega8 bietet noch viele weitere Optionen zur Konfiguration des UARTs, aber für die Datenübertragung zum PC sind im Normalfall keine anderen Einstellungen notwendig.

Senden von Zeichen

Das Beispielprogramm überträgt die Zeichenkette "Test!" in einer Endlosschleife an den PC.

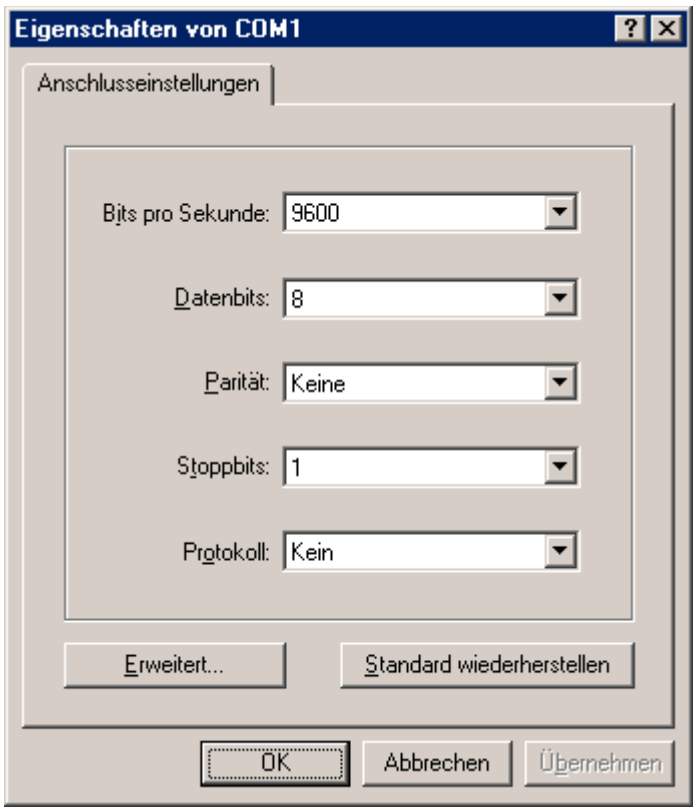
Hinweis

Wenn man das nachfolgende Programm laufen lässt und Hyperterminal startet, scheint es problemlos zu funktionieren. Wenn man aber das RS232 Kabel zwischenzeitlich abzieht und wieder ansteckt wird es oft passieren, dass nur noch wirre Zeichen auf dem PC erscheinen. Das liegt daran, dass der PC aus einem ununterbrochen Zeichenstrom nicht den Anfang eines Zeichens erkennen kann. Darum muss in solchen Fällen periodisch eine kleine Pause von der Länge mindestens eines Zeichens eingelegt werden, damit der PC sich wieder synchronisieren kann.

Die folgenden Beispiele sind für den ATmega8 geschrieben.

```
.include "m8def.inc"

.def temp    = r16                                ; Register für kleinere Arbeiten
.def zeichen = r17                                ; in diesem Register wird das
Zeichen an die
                                                  ; Ausgabefunktion übergeben

.equ F_CPU = 4000000                              ; Systemtakt in Hz
.equ BAUD  = 9600                                 ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))      ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

    ; Stackpointer initialisieren

    ldi     temp, HIGH(RAMEND)
    out     SPH, temp
    ldi     temp, LOW(RAMEND)
    out     SPL, temp

    ; Baudrate einstellen
```

```
    ldi     temp, HIGH(UBRR_VAL)
    out     UBRRH, temp
    ldi     temp, LOW(UBRR_VAL)
    out     UBRRL, temp

    ; Frame-Format: 8 Bit

    ldi     temp, (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)
    out     UCSRC, temp

    sbi     UCSRB,TXEN                  ; TX aktivieren

loop:
    ldi     zeichen, 'T'
    rcall   serout                      ; Unterprogramm aufrufen
    ldi     zeichen, 'e'
    rcall   serout                      ; Unterprogramm aufrufen
    ldi     zeichen, 's'
    rcall   serout                      ; ...
    ldi     zeichen, 't'
    rcall   serout
    ldi     zeichen, '!'
    rcall   serout
    ldi     zeichen, 10
    rcall   serout
    ldi     zeichen, 13
    rcall   serout
    rcall   sync
    rjmp    loop

serout:
    sbis    UCSRA,UDRE                  ; Warten bis UDR für das nächste
                                        ; Byte bereit ist
    rjmp    serout
    out     UDR, zeichen
    ret                                 ; zurück zum Hauptprogramm

; kleine Pause zum Synchronisieren des Empfängers, falls zwischenzeitlich
; das Kabel getrennt wurde

sync:
    ldi     r16,0
sync_1:
    ldi     r17,0
sync_loop:
    dec     r17
    brne    sync_loop
    dec     r16
    brne    sync_1
    ret
```



Der Befehl **rcall serout** ruft ein kleines Unterprogramm auf, das zuerst wartet bis das Datenregister **UDR** von der vorhergehenden Übertragung frei ist, und anschließend das in zeichen (=r17) gespeicherte Byte an **UDR** ausgibt.

Bevor *serout* aufgerufen wird, wird zeichen jedesmal mit dem ASCII-Code des zu übertragenden Zeichens geladen (so wie in Teil 4 bei der LCD-Ansteuerung). Der Assembler wandelt Zeichen in einfachen Anführungsstrichen automatisch in den entsprechenden ASCII-Wert um. Nach dem Wort "Test!" werden noch die Codes 10 (New Line) und 13 (Carriage Return) gesendet, um dem Terminalprogramm mitzuteilen, dass eine neue Zeile beginnt.

Eine Übersicht aller ASCII-Codes gibt es auf www.asciitable.com.

Die Berechnung der Baudrate wird übrigens nicht im Controller während der Programmausführung durchgeführt, sondern schon beim Assemblieren, wie man beim Betrachten der Listingdatei feststellen kann.

Zum Empfang muss auf dem PC ein Terminal-Programm wie z. B. HyperTerminal gestartet werden. Der folgende Screenshot zeigt, welche Einstellungen im Programm vorgenommen werden müssen:

Linux-Benutzer können das entsprechende Device (z. B. /dev/ttyS0) mit stty konfigurieren und mit cat die empfangenen Daten anzeigen oder ein Terminalprogramm wie minicom nutzen.

Alternativ kann unter Windows und Linux HTerm genutzt werden. (Freeware)

Senden von Zeichenketten

Eine bequemere Methode um längere Zeichenketten (Strings) zu übertragen ist hier zu sehen. Dabei

werden die Zeichenketten im Flash gespeichert. Als Abschluss des Strings wird der Wert 0x00 genutzt, so wie auch in der Programmiersprache C.

```
.include "m8def.inc"

.def temp    = r16                              ; Register für kleinere Arbeiten
.def zeichen = r17                              ; in diesem Register wird das
Zeichen an die
                                                ; Ausgabefunktion übergeben

.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

; hier geht unser Programm los

    ; Stackpointer initialisieren

    ldi     temp, HIGH(RAMEND)
    out     SPH, temp
    ldi     temp, LOW(RAMEND)
    out     SPL, temp

    ; Baudrate einstellen

    ldi     temp, HIGH(UBRR_VAL)
    out     UBRRH, temp
    ldi     temp, LOW(UBRR_VAL)
    out     UBRRL, temp

    ; Frame-Format: 8 Bit

    ldi     temp, (1<<URSEL)|(3<<UCSZ0)
    out     UCSRC, temp

    sbi     UCSRB,TXEN                      ; TX aktivieren

loop:
    ldi     zl,low(my_string*2);            ; Z Pointer laden
    ldi     zh,high(my_string*2);
    rcall   serout_string
    rjmp    loop

; Ausgabe eines Strings aus dem Flash

serout_string:
    lpm                             ; nächstes Byte aus dem Flash laden
    and     r0,r0                   ; = Null?
    breq    serout_string_ende      ; wenn ja, -> Ende
serout_string_wait:
    sbis    UCSRA,UDRE              ; Warten bis UDR für das nächste
                                    ; Byte bereit ist
    rjmp    serout_string_wait
```

```
    out     UDR, r0
    adiw    zl:zh,1                 ; Zeiger erhöhen
    rjmp    serout_string           ; nächstes Zeichen bearbeiten
serout_string_ende:
    ret                             ; zurück zum Hauptprogramm

; Hier wird jetzt der String definiert und im Flash gespeichert

my_string:  .db "Test!",10,13,0
```

Empfangen von Zeichen per Polling

Der AVR kann nicht nur Daten seriell senden, sondern auch empfangen. Dazu muss man, nachdem die Baudrate wie oben beschrieben eingestellt wurde, das Bit **RXEN** setzen.

Sobald der UART ein Byte über die serielle Verbindung empfangen hat, wird das Bit **RXC** im Register **UCSRA** gesetzt, um anzuzeigen, dass ein Byte im Register **UDR** zur Weiterverarbeitung bereitsteht. Sobald es aus **UDR** gelesen wurde, wird **RXC** automatisch wieder gelöscht, bis das nächste Byte angekommen ist.

Das erste einfache Testprogramm soll das empfangene Byte auf den an Port D angeschlossenen LEDs ausgeben. Dabei sollte man daran denken, dass PD0 (RXD) bereits für die Datenübertragung zuständig ist, so dass das entsprechende Bit im Register PORTD keine Funktion hat und damit auch nicht für die Datenanzeige verwendet werden kann.

Nachdem der UART konfiguriert ist, wartet das Programm einfach in der Hauptschleife darauf, dass ein Byte über den UART ankommt (z. B. indem man im Terminalprogramm ein Zeichen eingibt), also **RXC** gesetzt wird. Sobald das passiert, wird das Register **UDR**, in dem die empfangenen Daten stehen, nach *temp* eingelesen und an den Port D ausgegeben.

```
.include "m8def.inc"

.def temp = R16

.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

    ; Stackpointer initialisieren

    ldi     temp, HIGH(RAMEND)
    out     SPH, temp
    ldi     temp, LOW(RAMEND)
    out     SPL, temp

    ; Port D = Ausgang

    ldi     temp, 0xFF
```

```
    out     DDRD, temp

    ; Baudrate einstellen

    ldi     temp, HIGH(UBRR_VAL)
    out     UBRRH, temp
    ldi     temp, LOW(UBRR_VAL)
    out     UBRRL, temp

    ; Frame-Format: 8 Bit

    ldi     temp, (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)
    out     UCSRC, temp

    sbi     UCSRB, RXEN                     ; RX (Empfang) aktivieren

receive_loop:
   sbis     UCSRA, RXC                      ; warten bis ein Byte angekommen ist
   rjmp     receive_loop
   in       temp, UDR                       ; empfangenes Byte nach temp
kopieren
   out      PORTD, temp                     ; und an Port D ausgeben.
   rjmp     receive_loop                    ; zurück zum Hauptprogramm
```

Empfangen von Zeichen per Interrupt

Dieses Programm lässt sich allerdings noch verfeinern. Statt in der Hauptschleife auf die Daten zu warten, kann man auch veranlassen dass ein Interrupt ausgelöst wird, sobald ein Byte angekommen ist. Das sieht in der einfachsten Form so aus:

```
.include "m8def.inc"

.def temp = R16

.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

.org 0x00
        rjmp main

.org URXCaddr                                   ; Interruptvektor für UART-
Empfang
        rjmp int_rxc

; Hauptprogramm

main:

    ; Stackpointer initialisieren

    ldi     temp, HIGH(RAMEND)
```

```
    out     SPH, temp
    ldi     temp, LOW(RAMEND)
    out     SPL, temp

    ; Port D = Ausgang

    ldi     temp, 0xFF
    out     DDRD, temp

    ; Baudrate einstellen

    ldi     temp, HIGH(UBRR_VAL)
    out     UBRRH, temp
    ldi     temp, LOW(UBRR_VAL)
    out     UBRRL, temp

    ; Frame-Format: 8 Bit

    ldi     temp, (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)
    out     UCSRC, temp

    sbi     UCSRB, RXCIE                    ; Interrupt bei Empfang
    sbi     UCSRB, RXEN                     ; RX (Empfang) aktivieren

    sei                                     ; Interrupts global aktivieren

loop:
    rjmp loop                               ; Endlosschleife

; Interruptroutine: wird ausgeführt sobald ein Byte über das UART empfangen
wurde

int_rxc:
    push    temp                            ; temp auf dem Stack sichern
    in      temp, UDR                       ; empfangenes Byte lesen,
                                            ; dadurch wird auch der Interrupt
gelöscht
    out     PORTD, temp                     ; Daten ausgeben
    pop     temp                            ; temp wiederherstellen
    reti                                    ; Interrupt beenden
```

Diese Methode hat den großen Vorteil, dass das Hauptprogramm (hier nur eine leere Endlosschleife) andere Dinge erledigen kann, während der Controller Daten empfängt. Auf diese Weise kann man mehrere Aktionen quasi gleichzeitig ausführen, da das Hauptprogramm nur kurz unterbrochen wird, um die empfangenen Daten zu verarbeiten.

Probleme können allerdings auftreten, wenn in der Interruptroutine die gleichen Register verwendet werden wie im Hauptprogramm, da dieses ja an beliebigen Stellen durch den Interrupt unterbrochen werden kann. Damit sich aus der Sicht der Hauptschleife durch den Interruptaufruf nichts ändert, müssen alle in der Interruptroutine geänderten Register am Anfang der Routine gesichert und am Ende wiederhergestellt werden. Das gilt vor allem für das CPU-Statusregister (**SREG**)! Sobald ein einziger Befehl im Interrupt ein einziges Bit im SREG beeinflusst, muss das SREG gesichert werden. Das ist praktisch fast immer der Fall, nur in dem ganz einfachen Beispiel oben ist es überflüssig, weil die verwendeten Befehle das SREG nicht beeinflussen. In diesem Zusammenhang wird der Stack wieder interessant. Um die Register zu sichern, kann man sie mit **push** oben auf den Stapel legen und am Ende wieder in der umgekehrten Reihenfolge(!) mit **pop** vom Stapel herunternehmen.

Im folgenden Beispielprogramm werden die empfangenen Daten nun nicht mehr komplett angezeigt. Stattdessen kann man durch Eingabe einer 1 oder einer 0 im Terminalprogramm eine LED (an PB0) an- oder ausschalten. Dazu wird das empfangene Byte in der Interruptroutine mit den entsprechenden ASCII-Codes der Zeichen 1 und 0 (siehe www.asciitable.com) verglichen.

Für den Vergleich eines Registers mit einer Konstanten gibt es den Befehl **cpi register, konstante**. Das Ergebnis dieses Vergleichs kann man mit den Befehlen **breq label** (springe zu label, wenn gleich) und **brne label** (springe zu label, wenn ungleich) auswerten.

```
.include "m8def.inc"

.def temp = R16

.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

.org 0x00
        rjmp main

.org URXCaddr
        rjmp int_rxc

; Hauptprogramm
main:

    ; Stackpointer initialisieren

    ldi     temp, HIGH(RAMEND)
    out     SPH, temp
    ldi     temp, LOW(RAMEND)
    out     SPL, temp

    ; Port B = Ausgang

    ldi     temp, 0xFF
    out     DDRB, temp

    ; Baudrate einstellen

    ldi     temp, HIGH(UBRR_VAL)
    out     UBRRH, temp
    ldi     temp, LOW(UBRR_VAL)
    out     UBRRL, temp

    ; Frame-Format: 8 Bit

    ldi     temp, (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0)
    out     UCSRC, temp

    sbi     UCSRB, RXCIE                ; Interrupt bei Empfang
```

```
    sbi     UCSRB, RXEN                 ; RX (Empfang) aktivieren

    sei                                 ; Interrupts global aktivieren

loop:
    rjmp loop                           ; Endlosschleife

; Interruptroutine: wird ausgeführt sobald ein Byte über das UART empfangen
wurde

int_rxc:
    push    temp                        ; temp auf dem Stack sichern
    in      temp, sreg                  ; SREG sichern
    push    temp

    in      temp, UDR                   ; UART Daten lesen
    cpi     temp, '1'                   ; empfangenes Byte mit '1' vergleichen
    brne    int_rxc_1                   ; wenn nicht gleich, dann zu int_rxc_1
    cbi     PORTB, 0                    ; LED einschalten, low aktiv
    rjmp    int_rxc_2                   ; Zu int_rxc_2 springen
int_rxc_1:
    cpi     temp, '0'                   ; empfangenes Byte mit '0' vergleichen
    brne    int_rxc_2                   ; wenn nicht gleich, dann zu int_rxc_2
    sbi     PORTB, 0                    ; LED ausschalten, low aktiv
int_rxc_2:

    pop     temp
    out     sreg, temp                  ; SREG wiederherstellen
    pop     temp                        ; temp wiederherstellen
    reti
```

Handshake

Werden Daten schnell über eine serielle Leitung an ein langsames Gerät übertragen, dann kann es passieren, dass die Situation eintritt, dass das empfangende Gerät nicht mehr mitkommt. Das kann z.B. dadurch passieren, dass das empfangende Gerät selbst etwas Zeit für die Bearbeitung der Daten benötigt. Man denke z.B. an die Situation, dass an ein Modem Daten übertragen werden. Das Modem muss diese Daten bearbeiten und unter Umständen über eine langsame Telefonleitung absetzen. Überträgt der AVR seine Daten mit voller Geschwindigkeit an das Modem, so wird auch dem besten Modem irgendwann der interne Speicher ausgehen, in dem es die Daten zwischenspeichern kann.

Was benötigt wird, ist also eine Möglichkeit, wie die Gegenstelle dem Sender signalisieren kann: "Bitte jetzt nichts senden, ich bin beschäftigt!". Die einfachste Form eines derartigen Protokolls, nennt sich Handshake. Es gibt bei RS232 2 Arten, wie dieses Handshake implementiert werden kann: **Software-Handshake** und **Hardware-Handshake**.

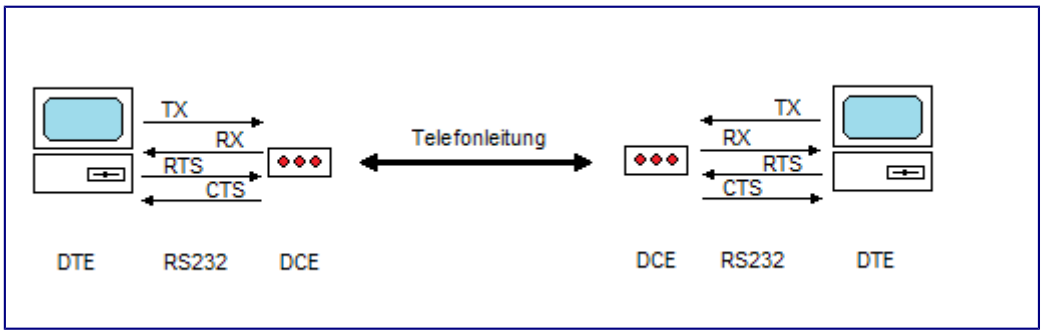
Hardware Handshake

Hardware Handshake benutzt die beiden Steuerleitungen **RTS - Request to Send** und **CTS - Clear to Send** um die Flusskontrolle durchzuführen.

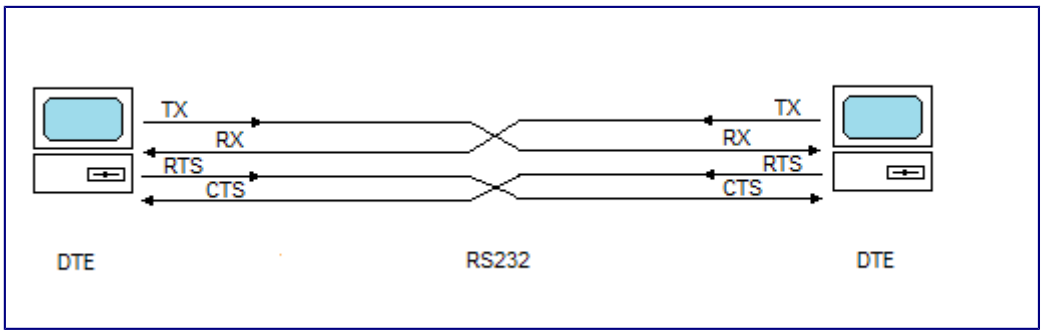
Die etwas seltsam anmutenden Namen haben historische Ursache. Ursprünglich war RS232 dazu gedacht ein Modem (ein sog. Data Carrier Equipment oder DCE) an einen Endpunkt (DTE oder Data Terminal Equipment) anzuschliessen. Wenn das DTE Daten senden wollte, aktivierte es die

Leitung RTS, es fragte praktisch beim DCE an: "Darf ich senden? (engl. Request sending)". Wenn das DCE bereit war, dann aktivierte es seinerseits die CTS Leitung und signalisierte damit "Alles ok. Daten marsch! (engl. Clear to send)". Solange das DCE nicht bereit war, Daten entgegenzunehmen, musste das DTE warten, bis es vom DCE die Freigabe zum Senden bekam.

- Für das DTE gilt: RTS ist ein Ausgang, CTS ist ein Eingang.
- Für das DCE gilt: RTS ist ein Eingang, CTS ist ein Ausgang.

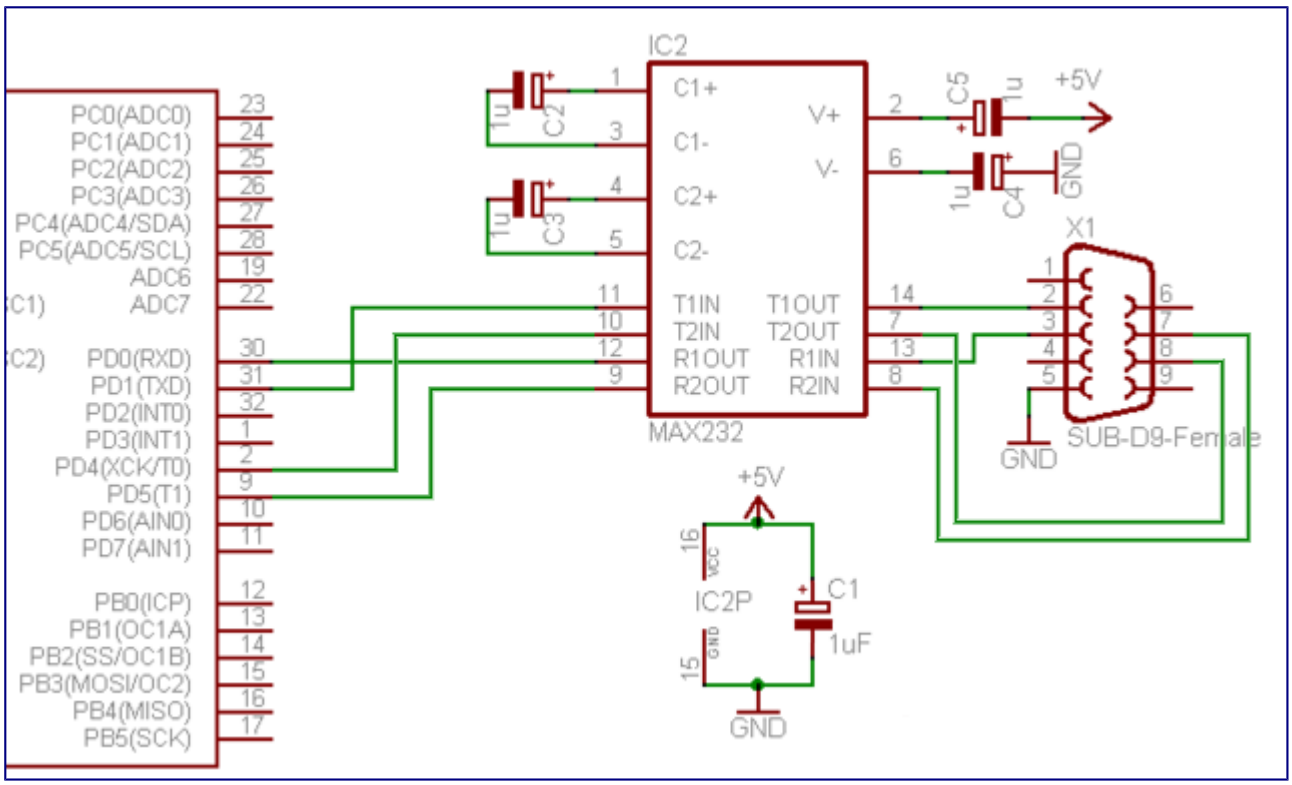


Das war die ursprüngliche Idee. Heutzutage ist es aber normal, dass 2 DTE miteinander über eine RS232 Verbindung gekoppelt werden. Wird in so einem Fall Hardware Handshake benutzt, so muss jedes DTE seiner Gegenstelle eine korrekte Bedienung der RTS/CTS Leitung vortäuschen.



Der Teil, dass CTS nur dann bedient wird, wenn über RTS die Anfrage nach der Sendefreigabe erfolgt entfällt dabei. Jeder Gesprächspartner überprüft ganz einfach vor dem Sendevorgang den Zustand der CTS Leitung der Gegenstelle, während der eigene RTS Ausgang zur Signalisierung der Empfangsbereitschaft für die Gegenstelle dient. Dies ist auch der Grund warum bei einem Null-Modem-Kabel nicht nur die RX/TX Leitungen, sondern auch die RTS/CTS Leitungen gekreuzt werden müssen.

Möchte man obige Schaltung um eine Hardware-Flusskontrolle erweitern, so bietet es sich an, die beiden noch freien Kanäle des MAX232 dafür zu verwenden. Die Schaltung sieht dann wie folgt aus:



UART/MAX232 Beschaltung für RTS/CTS am Beispiel eines Mega16. Achtung: Pinbelegung an den Mega8 anpassen

Am Mega8 stehen dann die Signale RTS bzw. CTS an den Pins PD4 bzw. PD5 zur Verfügung. An PD5 kann abgefragt werden, ob die Gegenstelle zum Empfang von Daten bereit ist, während der Mega8 über PD4 signalisieren kann, dass er im Moment keine Daten über die serielle Schnittstelle empfangen kann.

Bedenken sollte man dabei allerdings, dass nach der Rücknahme der Empfangsbereitschaft es je nach Gegenstelle dazu kommen kann, dass noch ein paar Zeichen über die UART eintreffen. Wird zb der Zustand der RTS Leitung vom Füllgrad eines Puffers abhängig gemacht, dann sollte man RTS nicht erst dann abschalten, wenn der Buffer komplett gefüllt ist, sondern schon ein paar Zeichen früher. Ursache könnte zb sein, dass die Gegenstelle über eine Hardware UART verfügt, die einen internen Buffer verfügt. Hat die Gegenstelle erst mal diesen Hardware Buffer gefüllt, dann gibt es oft keine Möglichkeit mehr für das dortige Programm diese UART-Hardware zu stoppen - die Zeichen die in die UART übertragen wurden, werden auf jeden Fall von der Hardware ausgegeben.

Software Handshake

Software Handshake benutzt die Datenleitung selbst, um die Flußkontrolle von Sender/Empfänger zu erreichen. Dazu wurden im ASCII Code 2 spezielle 'Zeichen' vorgesehen: XON (mit dem Code 0x11) und XOFF (mit dem Code 0x13).

Bemerkt ein Empfänger, dass er in Kürze keine Daten mehr vom Sender aufnehmen kann, dann sendet er seinerseits ein XOFF, woraufhin der Sender das Senden der Daten unterbricht. Ist der Empfänger wieder aufnahmebereit, so gibt er die Übertragung durch das Senden eines XON wieder frei.

Der Nachteil des Software-Handshaking besteht also in mehreren Punkten

- zum einen können nicht mehr alle Datenbytes übertragen werden, da ja die Bytes 0x11 und 0x13 eine spezielle Bedeutung haben. Möchte man Bytes binär übertragen, muss man daher spezielle Vorkehrungen treffen, damit diese Datenbytes nicht durch das Software-Handshaking fehlinterpretiert werden.
- zum anderen muss jeder Sender während er sendet auch gleichzeitig einen möglichen Empfang von Daten überwachen. Die Gegenstelle könnte ja mittels XOFF eine kurzfristige Unterbrechung der Sendung anfordern. Auch muss jeder Sender exakt darüber Buch führen, ob die Leitung zur Zeit im Status XOFF liegt und ob daher Übertragungen überhaupt möglich sind.
- das Senden von XOFF muss rechtzeitig erfolgen. Denn meistens benötigt die Gegenstelle etwas Zeit um das Senden einzustellen. Es kann durchaus sein, dass nach einem XOFF noch ein paar Zeichen von der Gegenstelle eintreffen
- es besteht die Gefahr eines Deadlocks, indem sich beide Seiten gegenseitig mit einem XOFF blockieren, aus dem sie nicht mehr herauskommen.

Weblinks

- WormFood's AVR Baud Rate Calculator online.
- Online Baudraten-Rechner für ATmega AVRs (JavaScript)

AVR-Tutorial: Speicher

Inhaltsverzeichnis

- 1 Speichertypen
  - 1.1 Flash-ROM
  - 1.2 EEPROM
  - 1.3 RAM
- 2 Anwendung
  - 2.1 Flash-ROM
  - 2.2 EEPROM
  - 2.3 SRAM
- 3 Siehe auch

Speichertypen

Die AVR-Mikrocontroller besitzen 3 verschiedene Arten von Speicher:

has	Flash	EEPROM	RAM
Schreibzyklen	>10.000	>100.000	unbegrenzt
Lesezyklen	unbegrenzt	unbegrenzt	unbegrenzt
flüchtig	nein	nein	ja
Größe beim ATtiny2313	2 KB	128 Byte	128 Byte
Größe beim ATmega8	8 KB	512 Byte	1 KB
Größe beim ATmega32	32 KB	1 KB	2 KB

Flash-ROM

Das **Flash-ROM** der AVRs dient als Programmspeicher. Über den Programmieradapter werden die kompilierten Programme vom PC an den Controller übertragen und im Flash-ROM abgelegt. Bei der Programmausführung wird das ROM Wort für Wort ausgelesen und ausgeführt. Es lässt sich aber auch zur Speicherung von Daten nutzen (z. B. Texte für ein LCD). Vom laufenden Programm aus kann man das ROM normalerweise nur lesen, nicht beschreiben. Es kann beliebig oft ausgelesen werden, aber theoretisch nur ~10.000 mal beschrieben werden.

EEPROM

Das **EEPROM** ist wie das Flash ein nichtflüchtiger Speicher, die Daten bleiben also auch nach dem Ausschalten der Betriebsspannung erhalten. Es kann beliebig oft gelesen und mindestens 100.000

mal beschrieben werden. Bei den AVRs kann man es z. B. als Speicher für Messwerte oder Einstellungen benutzen.

RAM

Das **RAM** ist ein flüchtiger Speicher, d.h. die Daten gehen nach dem Ausschalten verloren. Es kann beliebig oft gelesen und beschrieben werden, weshalb es sich zur Speicherung von Variablen eignet für die die Register R0-R31 nicht ausreichen. Daneben dient es als Speicherort für den Stack, auf dem z. B. bei Unterprogrammaufrufen (rcall) die Rücksprungadresse gespeichert wird (siehe AVR-Tutorial: SRAM).

Anwendung

Flash-ROM

Die erste und wichtigste Anwendung des Flash-ROMs kennen wir bereits: Das Speichern von Programmen, die wir nach dem Assemblieren dort hineingeladen haben. Nun sollen aber auch vom laufenden Programm aus, Daten ausgelesen werden.

Um die Daten wieder auszulesen, muss man die Adresse, auf die zugegriffen werden soll, in den **Z-Pointer** laden. Der Z-Pointer besteht aus den Registern **R30** (Low-Byte) und **R31** (High-Byte), daher kann man das Laden einer Konstante wie gewohnt mit dem Befehl **ldi** durchführen. Statt R30 und R31 kann man übrigens einfach **ZL** und **ZH** schreiben, da diese Synonyme bereits in der include-Datei m8def.inc definiert sind.

Wenn die richtige Adresse erstmal im Z-Pointer steht, geht das eigentliche Laden der Daten ganz einfach mit dem Befehl **lpm**. Dieser Befehl, der im Gegensatz zu out, ldi usw. keine Operanden hat, veranlasst das Laden des durch den Z-Pointer adressierten Bytes aus dem Programmspeicher in das Register **R0**, von wo aus man es weiterverarbeiten kann.

Jetzt muss man nur noch wissen, wie man dem Assembler überhaupt beibringt, dass er die von uns festgelegten Daten im ROM platzieren soll und wie man dann an die Adresse kommt an der sich diese Daten befinden. Um den Programmspeicher mit Daten zu füllen, gibt es die Direktiven .db und .dw. In der Regel benötigt man nur .db, was folgendermaßen funktioniert:

```
daten:
    .db 12, 20, 255, 0xFF, 0b10010000
```

Direktiven wie .db sind Anweisungen an den Assembler, keine Prozessorbefehle. Von denen kann man sie durch den vorangestellten Punkt unterscheiden. In diesem Fall sagen wir dem Assembler, dass er die angegebenen Bytes nacheinander im Speicher platzieren soll; wenn man die Zeile also assembliert, erhält man eine Hex-Datei, die nur diese Daten enthält.

Aber was soll das **daten:** am Anfang der Zeile? Bis jetzt haben wir Labels nur als Sprungmarken verwendet, um den Befehlen **rcall** und **rjmp** zu sagen, an welche Stelle im Programm gesprungen werden soll. Würden wir in diesem Fall **rjmp daten** im Programm stehen haben, dann würde die Programmausführung zur Stelle **daten:** springen, und versuchen die sinnlosen Daten als Befehle zu interpretieren - was mit Sicherheit dazu führt, dass der Controller Amok läuft.

Statt nach **daten:** zu springen, sollten wir die Adresse besser in den Z-Pointer laden. Da der Z-Pointer aus zwei Bytes besteht, brauchen wir dazu zweimal den Befehl **ldi**:

```
    ldi ZL, LOW(daten*2)    ; Low-Byte der Adresse in Z-Pointer
    ldi ZH, HIGH(daten*2)   ; High-Byte der Adresse in Z-Pointer
```

Wie man sieht, ist das Ganze sehr einfach: Man kann die Labels im Assembler direkt wie Konstanten verwenden. Über die Multiplikation der Adresse mit zwei sollte man sich erst mal keine Gedanken machen: "Das ist einfach so." Wer es genauer wissen will schaut hier nach.

Um zu zeigen wie das alles konkret funktioniert, ist das folgende Beispiel nützlich:

```
.include "m8def.inc"

    ldi     R16, 0xFF
    out     DDRB, R16               ; Port B: Ausgang

    ldi     ZL, LOW(daten*2)        ; Low-Byte der Adresse in Z-Pointer
    ldi     ZH, HIGH(daten*2)       ; High-Byte der Adresse in Z-Pointer

    lpm                             ; durch Z-Pointer adressiertes Byte
                                    ; in R0 laden
    out     PORTB, R0               ; an PORTB ausgeben

ende:
    rjmp ende                       ; Endlosschleife

daten:
    .db 0b10101010
```

Wenn man dieses Programm assembliert und in den Controller überträgt, dann kann man auf den an Port B angeschlossenen LEDs das mit **.db 0b10101010** im Programmspeicher abgelegte Bitmuster sehen.

Fallbeispiel: Zeichenketten

Eine häufige Anwendung von **lpm** ist das Auslesen von Zeichenketten ("Strings") aus dem Flash-ROM und die Ausgabe an den seriellen Port oder ein LCD. Das folgende Programm gibt in einer Endlosschleife den Text "AVR-Assembler ist ganz einfach", gefolgt von einem Zeilenumbruch, an den UART aus.

```
.include "m8def.inc"

.def temp = r16
.def temp1 = r17

.equ CLOCK = 4000000                ; Frequenz des Quarzes
.equ BAUD = 9600                    ; Baudrate
.equ UBRRVAL = CLOCK/(BAUD*16)-1    ; Baudratenteiler

; hier geht das Programmsegment los

.CSEG
.org 0
    ldi     r16, low(RAMEND)            ; Stackpointer initialisieren
    out     SPL, r16
```

```
    ldi     r16, high(RAMEND)
    out     SPH, r16

    ldi     temp, LOW(UBRRVAL)          ; Baudrate einstellen
    out     UBRRL, temp
    ldi     temp, HIGH(UBRRVAL)
    out     UBRRH, temp

    ldi     temp, (1<<URSEL)|(3<<UCSZ0) ; Frame-Format: 8 Bit
    out     UCSRC, temp
    sbi     UCSRB, TXEN                 ; TX (Senden) aktivieren

loop:
    ldi     ZL, LOW(text*2)             ; Adresse des Strings in den
    ldi     ZH, HIGH(text*2)            ; Z-Pointer laden
    rcall   print                       ; Funktion print aufrufen
    rcall   wait                        ; kleine Pause
    rjmp    loop                        ; das Ganze wiederholen

; kleine Pause
wait:
    ldi     temp,0
wait_1:
    ldi     temp1,0
wait_2:
    dec     temp1
    brne    wait_2
    dec     temp
    brne    wait_1
    ret

; print: sendet die durch den Z-Pointer adressierte Zeichenkette

print:
    lpm                                 ; Erstes Byte des Strings nach R0 lesen
    tst     R0                          ; R0 auf 0 testen
    breq    print_end                   ; wenn 0, dann zu print_end
    mov     r16, r0                     ; Inhalt von R0 nach R16 kopieren
    rcall   sendbyte                    ; UART-Sendefunktion aufrufen
    adiw    ZL, 1                       ; Adresse des Z-Pointers um 1 erhöhen
    rjmp    print                       ; wieder zum Anfang springen
print_end:
    ret

; sendbyte: sendet das Byte aus R16 über das UART

sendbyte:
    sbis    UCSRA, UDRE                 ; warten bis das UART bereit ist
    rjmp    sendbyte
    out     UDR, r16
    ret

; Konstanten werden hier im Flash abgelegt

text:
    .db "AVR-Assembler ist ganz einfach",10,13,0
    ; Stringkonstante, durch eine 0 abgeschlossen
    ; die 10 bzw. 13 sind Steuerzeichen für Wagenrücklauf und neue Zeile
```

Neuere AVR-Controller besitzen einen erweiterten Befehlssatz. Darunter befindet sich auch der folgende Befehl:

```
    lpm     r16, Z+
```

Dieser Befehl liest ein Byte aus dem Flash und speichert es in einem beliebigen Register, hier r16. Danach wird der Zeiger Z um eins erhöht. Für die neuen Controller, wie ATmegas kann das Codebeispiel also so abgeändert werden:

```
; print: sendet die durch den Z-Pointer adressierte Zeichenkette
print:
    lpm     r16, Z+         ; Erstes Byte des Strings nach r16 lesen
    tst     r16             ; r16 auf 0 testen
    breq    print_end       ; wenn 0, dann zu print_end
    rcall   sendbyte        ; UART-Sendefunktion aufrufen
    rjmp    print           ; wieder zum Anfang springen
print_end:
    ret
```

Wenn man bei .db einen Text in doppelten Anführungszeichen angibt, werden die Zeichen automatisch in die entsprechenden ASCII-Codes umgerechnet:

```
    .db     "Test", 0
    ; ist äquivalent zu
    .db     84, 101, 115, 116, 0
```

Damit das Programm das Ende der Zeichenkette erkennen kann, wird eine 0 an den Text angehängt.

Das ist doch schonmal sehr viel praktischer, als jeden Buchstaben einzeln in ein Register zu laden und abzuschicken. Und wenn man statt **sendbyte** einfach die Routine **lcd_data** aus dem 4. Teil des Tutorials aufruft, dann funktioniert das gleiche sogar mit dem LCD!

Neue Assemblerbefehle

```
    lpm                             ; Liest das durch den Z-Pointer
                                    ; addressierte Byte aus dem Flash-ROM
                                    ; in das Register R0 ein.

    lpm     [Register], Z           ; Macht das gleiche wie lpm, jedoch in
                                    ; ein beliebiges Register

    lpm     [Register], Z+          ; Erhöht zusätzlich den Z-Zeiger

    tst     [Register]              ; Prüft, ob Inhalt eines Registers
                                    ; gleich 0 ist.

    breq    [Label]                 ; Springt zu [Label], wenn der
                                    ; vorhergehende Vergleich wahr ist.

    adiw    [Register], [Konstante] ; Addiert eine Konstante zu einem
                                    ; Registerpaar. [Register] bezeichnet das
                                    ; untere der beiden Register.
                                    ; Kann nur auf die Registerpaare
                                    ; R25:R24, R27:R26, R29:R28 und R31:R30
                                    ; angewendet werden.
```

EEPROM

Den Flash Speicher kann man also benutzen, um dort Daten abzulegen, die sich während eines

Programmlaufs nicht verändern. Irgendwelche Tabellen oder konstante Texte. Aber des öfteren möchte man auch die Möglichkeit haben, Daten zu speichern und wieder zu lesen, die sich während des Programmlaufs ändern. Ganz besonders möchte man eine Speicherfläche zur Verfügung haben, die ihren Inhalt auch dann behält, wenn dem µC die Versorgungsspannung abgedreht wird. Man denke z.B. an irgendwelche Konfigurationsdaten oder an Informationen, wie weit der µC in der Bearbeitung von Daten vorangekommen ist oder irgendwelche Statistikdaten, die auch nach einem Stromausfall noch verfügbar sein sollen. In solchen Fällen kommt das EEPROM zum Einsatz

Lesen

Als erstes muss geprüft werden, ob ein vorheriger Schreibzugriff schon abgeschlossen ist. Danach wird die EEPROM-Adresse von der gelesen werden soll in das IO-Registerpaar **EEARH/EEARL** (EEPROM Address Register) geladen. Da der ATmega8 mehr als 256 Byte EEPROM hat, passt die Adresse nicht in ein einziges 8-Bit-Register, sondern muss in zwei Register aufgeteilt werden: EEARH bekommt das obere Byte der Adresse, EEARL das untere Byte. Dann löst man den Lesevorgang durch das Setzen des Bits **EERE** (EEPROM Read Enable) im IO-Register **EECR** (EEPROM Control Register) aus. Das gelesene Byte kann sofort aus dem IO-Register **EEDR** (EEPROM Data Register) in ein normales CPU-Register kopiert und dort weiterverarbeitet werden.

Wie auch das Flash-ROM kann man das EEPROM über den ISP-Programmer programmieren. Die Daten, die im EEPROM abgelegt werden sollen, werden wie gewohnt mit .db angegeben; allerdings muss man dem Assembler natürlich sagen, dass es sich hier um Daten für das EEPROM handelt. Das macht man durch die Direktive **.eseg**, woran der Assembler erkennt, dass alle nun folgenden Daten für das EEPROM bestimmt sind.

Damit man die Bytes nicht von Hand abzählen muss um die Adresse herauszufinden, kann man auch im EEPROM-Segment wieder Labels einsetzen und diese im Assemblerprogramm wie Konstanten verwenden.

```
.include "m8def.inc"

; hier geht die Programmsektion los
.cseg

    ldi     r16, low(RAMEND)            ; Stackpointer initialisieren
    out     SPL, r16
    ldi     r16, high(RAMEND)
    out     SPH, r16

    ldi     r16, 0xFF
    out     DDRB, r16                   ; Port B Ausgang

    ldi     ZL,low(daten)               ; Z-Zeiger laden
    ldi     ZH,high(daten)
    rcall   EEPROM_read                 ; Daten aus EEPROM lesen
    out     PORTB, r16

loop:
    rjmp loop

EEPROM_read:
    sbic    EECR,EEWE                   ; prüfe ob der vorherige Schreibzugriff
                                        ; beendet ist
```

```
    rjmp    EEPROM_read                 ; nein, nochmal prüfen

    out     EEARH, ZH                   ; Adresse laden
    out     EEARL, ZL
    sbi     EECR, EERE                  ; Lesevorgang aktivieren
    in      r16, EEDR                   ; Daten in CPU Register kopieren
    ret

; Daten im EEPROM definieren
.eseg
daten:
    .db     0b10101010
```

Wenn man dieses Programm assembliert, erhält man außer der .hex-Datei noch eine Datei mit der Endung **.eep**. Diese Datei enthält die Daten aus dem EEPROM-Segment (.eseg), und muss zusätzlich zu der hex-Datei in den Controller programmiert werden.

Das Programm gibt die Binärzahl 0b10101010 an den Port B aus, das heißt jetzt sollte jede zweite LED leuchten.

Natürlich kann man auch aus dem EEPROM Strings lesen und an den UART senden:

```
.include "m8def.inc"

.def temp = r16

.equ CLOCK = 4000000                ; Frequenz des Quarzes

.equ BAUD = 9600                    ; Baudrate
.equ UBRRVAL = CLOCK/(BAUD*16)-1    ; Baudratenteiler

; hier geht das Programmsegment los

.CSEG

; Hauptprogramm
main:
    ldi     temp, LOW(RAMEND)           ; Stackpointer initialisieren
    out     SPL, temp
    ldi     temp, HIGH(RAMEND)
    out     SPH, temp

    ldi     temp, LOW(UBRRVAL)          ; Baudrate einstellen
    out     UBRRL, temp
    ldi     temp, HIGH(UBRRVAL)
    out     UBRRH, temp

    ldi     temp, (1<<URSEL)|(3<<UCSZ0) ; Frame-Format: 8 Bit
    out     UCSRC, temp
    sbi     UCSRB, TXEN                 ; TX (Senden) aktivieren

    ldi     ZL, low(text1)              ; ersten String senden
    ldi     ZH, high(text1)             ; Z-Pointer laden
    rcall   EEPROM_print

    ldi     ZL, low(text2)              ; zweiten String senden
    ldi     ZH, high(text2)             ; Z-Pointer laden
    rcall   EEPROM_print

loop:
    rjmp    loop                        ; Endlosschleife
```

```
; EEPROM Lesezugriff auf Strings + UART Ausgabe

EEPROM_print:
    sbic    EECR,EEWE           ; prüf ob der vorherige Schreibzugriff
                                ; beendet ist
    rjmp    EEPROM_print        ; nein, nochmal prüfen

    out     EEARH, ZH           ; Adresse laden
    out     EEARL, ZL

    sbi     EECR, EERE          ; Lesevorgang aktivieren
    in      temp, EEDR          ; Daten in CPU Register kopieren
    tst     temp                ; auf 0 testen (=Stringende)
    breq    eep_print_end       ; falls 0, Funktion beenden
    rcall   sendbyte            ; ansonsten Byte senden...
    adiw    ZL,1                ; Adresse um 1 erhöhen...
    rjmp    EEPROM_print        ; und zum Anfang der Funktion
eep_print_end:
    ret

; sendbyte: sendet das Byte aus "data" über den UART

sendbyte:
    sbis    UCSRA, UDRE         ; warten bis das UART bereit ist
    rjmp    sendbyte
    out     UDR, temp
    ret

; hier wird der EEPROM-Inhalt definiert

.ESEG

text1:
    .db     "Strings funktionieren auch ", 0
text2:
    .db     "im EEPROM",10,13, 0
```

Schreiben

Als erstes muss geprüft werden, ob ein vorheriger Schreibzugriff schon abgeschlossen ist. Danach wird die EEPROM-Adresse, auf die geschrieben wird, in das IO-Register **EEAR** (**EE**PROM **A**ddress **R**egister) geladen. Dann schreibt man die Daten, welche man auf der im Adressregister abgespeicherten Position ablegen will ins Register **EEDR** (**EE**PROM **D**ata **R**egister). Als nächstes setzt man das **EEMWE** Bit im EEPROM-Kontrollregister **EECR** (**EE**PROM **C**ontrol **R**egister) um den Schreibvorgang vorzubereiten. Nun wird es zeitkritisch - es darf nun keinesfalls ein Interrupt dazwischenfahren - denn man muss innerhalb von 4 Taktzyklen das **EEWE** Bit setzen um den Schreibvorgang auszulösen. Um das unter allen Bedingungen sicherzustellen werden die Interrupts kurz gesperrt. Danach startet der Schreibvorgang und läuft automatisch ab. Wenn er beendet ist, wird von der Hardware das **EEWE** Bit im Register **EECR** wieder gelöscht.

In diesem Beispiel werden Zeichen per UART und Interrupt empfangen und nacheinander im EEPROM gespeichert. Per Terminalprogramm kann man nun bis zu 512 Zeichen in den EEPROM schreiben. Per Programmieradapter kann man denn EEPROM wieder auslesen und seine gespeicherten Daten anschauen.

```
.include "m8def.inc"

.def temp      = r16
.def sreg_save = r17

.equ CLOCK = 4000000

.equ BAUD = 9600
.equ UBRRVAL = CLOCK/(BAUD*16)-1

; hier geht das Programmsegment los

.CSEG
.org 0x00
    rjmp    main

.org URXCaddr
    rjmp    int_rxc

; Hauptprogramm
main:
    ldi     temp, LOW(RAMEND)           ; Stackpointer initialisieren
    out     SPL, temp
    ldi     temp, HIGH(RAMEND)
    out     SPH, temp

    ldi     temp, LOW(UBRRVAL)          ; Baudrate einstellen
    out     UBRRL, temp
    ldi     temp, HIGH(UBRRVAL)
    out     UBRRH, temp


    ldi     temp, (1<<URSEL)|(3<<UCSZ0) ; Frame-Format: 8 Bit
    out     UCSRC, temp

    sbi     UCSRB, RXCIE                ; Interrupt bei Empfang
    sbi     UCSRB, RXEN                 ; RX (Empfang) aktivieren

    ldi     ZL,low(daten)               ; der Z-Zeiger wird hier exclusiv
    ldi     ZH,high(daten)              ; für die Datenadressierung verwendet

    sei                                 ; Interrupts global aktivieren

loop:
    rjmp    loop                        ; Endlosschleife (ABER Interrupts!)

; Interruptroutine wird ausgeführt,
; sobald ein Byte über den UART empfangen wurde

int_rxc:
    push    temp                        ; temp auf dem Stack sichern
    in      temp,sreg                   ; SREG sicher, muss praktisch in jeder
                                        ; Interruptroutine gemacht werden
    push    temp

    in      temp, UDR                   ; empfangenes Byte lesen
    rcall   EEPROM_write                ; Byte im EEPROM speichern
    adiw    ZL,1                        ; Zeiger erhöhen
    cpi     ZL,low(EEPROMEND+1)         ; Vergleiche den Z Zeiger
    ldi     temp,high(EEPROMEND+1)      ; mit der maximalen EEPROM Adresse +1
    cpc     ZH,temp
    brne    int_rxc_1                   ; wenn ungleich, springen
    ldi     ZL,low(Daten)               ; wenn gleich, Zeiger zurücksetzen
    ldi     ZH,high(Daten)
```

```
int_rxc_1:

    pop     temp
    out     sreg,temp
    pop     temp                        ; temp wiederherstellen
    reti

; der eigentliche EEPROM Schreibzugriff
; Adresse in ZL/ZH
; Daten in temp

EEPROM_write:
    sbic    EECR, EEWE                  ; prüfe ob der letzte Schreibvorgang
beendet ist
    rjmp    EEPROM_write                ; wenn nein, nochmal prüfen

    out     EEARH, ZH                   ; Adresse schreiben
    out     EEARL, ZL                   ;
    out     EEDR,temp                   ; Daten  schreiben
    in      sreg_save,sreg              ; SREG sichern
    cli                                 ; Interrupts sperren, die nächsten
                                        ; zwei Befehle dürfen NICHT
                                        ; unterbrochen werden
    sbi     EECR,EEMWE                  ; Schreiben vorbereiten
    sbi     EECR,EEWE                   ; Und los !
    out     sreg, sreg_save             ; SREG wieder herstellen
    ret

; hier wird der EEPROM-Inhalt definiert
.ESEG

Daten:
    .db     0
```

SRAM

Die Verwendung des SRAM wird in einem anderen Kapitel erklärt: AVR-Tutorial: SRAM

Siehe auch

- Adressierung

AVR-Tutorial: Timer

Timer sind eines der Hauptarbeitspferde in unserem Mikrocontroller. Mit ihrer Hilfe ist es möglich, in regelmäßigen Zeitabständen Aktionen zu veranlassen. Aber Timer können noch mehr!

- Timer können mit einem externen Pin hochgezählt werden
- Es gibt Möglichkeiten, bei bestimmten Zählerständen einen Interrupt auslösen zu lassen
- Timer können aber auch völlig selbstständig Signale an einem Ausgabepin erzeugen
- ...

Aber der Reihe nach und kurz zur Erinnerung: Die Beispiele passen zu einem ATmega8 und ggf. einer Reihe anderer AVR, können bei einigen Typen aber abweichen.

Inhaltsverzeichnis

- 1 Was ist ein Timer?
- 2 Der Vorteiler (Prescaler)
- 3 Erste Tests
- 4 Simulation im AVR-Studio
- 5 Wie schnell schaltet der Port?
- 6 Timer 0
  - 6.1 TCCR0
  - 6.2 TIMSK
- 7 Timer 1
  - 7.1 TCCR1B
  - 7.2 TCCR1A
  - 7.3 OCR1A
  - 7.4 OCR1B
  - 7.5 ICR1
  - 7.6 TIMSK1
- 8 Timer 2
  - 8.1 TCCR2
  - 8.2 OCR2
  - 8.3 TIMSK2
- 9 Was geht noch mit einem Timer?
- 10 Weblinks

Was ist ein Timer?

Ein Timer ist im Grunde nichts anderes als ein bestimmtes Register im Mikrocontroller, das hardwaregesteuert fortlaufend um 1 erhöht (oder verringert) wird (statt *um 1 erhöhen* sagt man auch **inkrementieren**, und das Gegenstück, **dekrementieren**, bedeutet *um 1 verringern*). Anstatt also Befehle im Programm vorzusehen, die regelmäßig ausgeführt werden und ein Register inkrementieren, erledigt dies der Mikrocontroller ganz von alleine. Dazu ist es möglich, den Timer mit dem Systemtakt zu verbinden und so die Genauigkeit des Quarzes auszunutzen, um ein Register regelmäßig und vor allen Dingen unabhängig vom restlichen Programmfluss (!) hochzählen zu lassen.

Davon alleine hätte man aber noch keinen großen Gewinn. Nützlich wird das Ganze erst dann, wenn man bei bestimmten Zählerständen eine Aktion ausführen lassen kann. Einer der 'bestimmten Zählerstände' ist zum Beispiel der **Overflow**. Das Zählregister eines Timers kann natürlich nicht beliebig lange inkrementiert werden – z. B. ist der höchste Zählerstand, den ein 8-Bit-Timer erreichen kann, $2^8 – 1 = 255$. Beim nächsten Inkrementierschritt tritt ein Überlauf (engl. Overflow) auf, der den Timerstand wieder zu 0 werden lässt. Und hier liegt der springende Punkt. Wir können uns nämlich an diesen Overflow "anhängen" und den Controller so konfigurieren, dass beim Auftreten des Timer-Overflows ein Interrupt ausgelöst wird.

Die kann auch alles nochmal im Datenblatt von Atmel zu einigen ATMega Datenblättern nachgelesen werden: http://www.atmel.com/Images/doc8161.pdf

Der Vorteiler (Prescaler)

Wenn also der Quarzoszillator mit 4 MHz schwingt, dann würde auch der Timer 4 Millionen mal in der Sekunde erhöht werden. Da der Timer jedes Mal von 0 bis 255 zählt, bevor ein Overflow auftritt, heißt das auch, dass in einer Sekunde 4000000 / 256 = 15625 Overflows vorkommen. Ganz schön schnell! Nur: Oft ist das nicht sinnvoll. Um diese Raten zu verzögern, gibt es den Vorteiler, oder auf Englisch, Prescaler. Er kann z.B. auf die Werte 1, 8, 64, 256 oder 1024 eingestellt werden, je nach Timer (Bitte Datenblatt konsultieren!). Seine Aufgabe ist es, den Systemtakt um den angegebenen Faktor zu teilen. Steht der Vorteiler also auf 1024, so wird nur bei jedem 1024-ten Impuls vom Systemtakt das Timerregister um 1 erhöht. Entsprechend weniger häufig kommen dann natürlich die Overflows. Der Systemtakt sei wieder 4000000. Dann wird der Timer in 1 Sekunde 4000000 / 1024 = 3906,25 mal erhöht. Da der Timer wieder jedesmal bis 255 zählen muss bis ein Overflow auftritt, bedeutet dies, dass in 1 Sekunde 3906,25 / 256 = 15,25 Overflows auftreten.

Systemtakt: 4Mhz

Vorteiler	Overflows/Sekunde	Zeit zwischen 2 Overflows [s]	
1	15625	0.000064	= 64 µs
8	1953.125	0.000512	= 512 µs
64	244.1406	0.004096	≈ 4,1 ms
256	61.0351	0.016384	≈ 16,4 ms
1024	15.2587	0.065536	≈ 65,5 ms

Die Zeit (in Sekunden) zwischen 2 Overflows lässt sich sehr leicht berechnen:

$$t = \frac{2^{\text{Bit des Timers}} \cdot \text{Vorteiler}}{\text{Systemtakt in Hz}}.$$

Erste Tests

Ein Programm, das einen Timer Overflow in Aktion zeigt, könnte z. B. so aussehen:

```
.include "m8def.inc"

.def temp = r16
.def leds = r17

.org 0x0000
        rjmp    main                    ; Reset Handler
.org OVF0addr
        rjmp    timer0_overflow         ; Timer Overflow Handler

main:
        ; Stackpointer initialisieren
        ldi     temp, HIGH(RAMEND)
        out     SPH, temp
        ldi     temp, LOW(RAMEND)
        out     SPL, temp

        ldi     temp, 0xFF              ; Port B auf Ausgang
        out     DDRB, temp

        ldi     leds, 0xFF

        ldi     temp, (1<<CS00)         ; CS00 setzen: Teiler 1
        out     TCCR0, temp

        ldi     temp, (1<<TOIE0)        ; TOIE0: Interrupt bei Timer Overflow
        out     TIMSK, temp

        sei

loop:   rjmp    loop

timer0_overflow:                        ; Timer 0 Overflow Handler
        out     PORTB, leds
        com     leds
        reti
```

Das Programm beginnt mit der Interrupt-Vektoren-Tabelle. Dort ist an der Adresse *OVF0Addr* ein Sprung zur Marke *timer0_overflow* eingetragen. Wenn also ein Overflow Interrupt vom Timer 0 auftritt, so wird dieser Interrupt durch den **rjmp** weitergeleitet an die Stelle *timer0_overflow*.

Das Hauptprogramm beginnt ganz normal mit der Belegung des Stackpointers. Danach wird der Port B auf Ausgang geschaltet, wir wollen hier wieder die LED anschliessen.

Durch Beschreiben von TCCR0 mit dem Bitmuster 0b00000001, hier ausgedrückt durch (1<<CS00), wird der Vorteiler auf 1 gesetzt. Für die ersten Versuche empfiehlt es sich, das Programm mit dem AVR-Studio zunächst zu simulieren. Würden wir einen größeren Vorteiler benutzen, so müsste man ziemlich oft mittels F11 einen simulierten Schritt ausführen, um eine

Änderung im Timerregister zu erreichen.

Die nächsten Anweisungen setzen im TIMSK Register das TOIE0 Bit. Sinn der Sache ist es, dem Timer zu erlauben, bei Erreichen eines Overflow einen Interrupt auszulösen.

Zum Schluss noch die Interrupts generell mittels **sei** freigeben. Dieser Schritt ist obligatorisch. Im Mikrocontroller können viele Quellen einen Interrupt auslösen. Daraus folgt: Für jede mögliche Quelle muss festgelegt werden, ob sie einen Interrupt erzeugen darf oder nicht. Die Oberhoheit hat aber das globale Interrupt Flag. Mit ihm können alle Interrupts, egal von welcher Quelle sie kommen, unterdrückt werden.

Damit ist die Initialisierung beendet und das Hauptprogramm kann sich schlafen legen. Die **loop: rjmp loop** Schleife macht genau dieses.

Tritt nun ein Overflow am Timer auf, so wird über den Umweg über die Interrupt Vektor Tabelle der Programmteil *timer0_overflow* angesprungen. Dieser gibt einfach nur den Inhalt des Registers leds am Port B aus. Danach wird das leds Register mit einer **com** Operation negiert, so dass aus allen 0 Bits eine 1 wird und umgekehrt. Die Overflow Behandlung ist damit beendet und mittels **reti** wird der Interrupt Handler wieder verlassen.

Simulation im AVR-Studio

Es lohnt sich, den ganzen Vorgang im AVR-Studio simulieren zu lassen. Dazu am besten in der linken I/O View Ansicht die Einträge für PORTB und TIMER_COUNTER_0 öffnen. Wird mittels F11 durch das Programm gegangen, so sieht man, dass ab dem Moment, ab dem der Vorteiler auf 1 gesetzt wird, der Timer 0 im TCNT0 Register zu zählen anfängt. Mit jedem Druck auf F11 erhöht sich der Zählerstand. Irgendwann ist dann die Endlosschleife loop erreicht. Drücken Sie weiterhin F11 und beobachten sie, wie TCNT0 immer höher zählt, bis der Overflow erreicht wird. In dem Moment, in dem der Overflow erreicht wird, wird der Interrupt ausgelöst (hierfür muss für das Debuggen im AVR-Studio "Mask interrupts while stepping" deaktiviert sein, zu finden in Tools -> Options). Mit dem nächsten F11 landen sie in der Interrupt Vektor Tabelle und von dort geht es weiter zu timer_0_overflow. Weitere Tastendrücke von F11 erledigen dann die Ausgabe auf den Port B, das Invertieren des Registers r17 und der Interrupt ist damit behandelt. Nach dem **reti** macht der Microcontroller genau an der Stelle weiter, an der er vom Interrupt unterbrochen wurde. Und der Timer 0 hat in der Zwischenzeit weitergezählt! Nach exakt weiteren 256 Schritten, vom Auftreten des ersten Overflows an gerechnet, wird der nächste Overflow ausgelöst.

Wie schnell schaltet der Port?

Eine berechtigte Frage. Dazu müssen wir etwas rechnen. Keine Angst, es ist nicht schwer, und wer das Prinzip bisher verstanden hat, der sollte keine Schwierigkeiten haben, die Berechnung nachzuvollziehen.

Der Quarzoszillator schwingt mit 4 MHz. Das heißt, in 1 Sekunde werden 4000000 Taktzyklen generiert. Durch die Wahl des Vorteilers von 1 bedeutet das auch, dass der Timer 4000000 mal in der Sekunde erhöht wird. Von einem Overflow zum nächsten muss der Timer 256 Zählvorgänge ausführen. Also werden in 1 Sekunde 4000000 / 256 = 15625 Overflows generiert. Bei jedem

Overflow schalten wir die LEDs jeweils in den anderen Zustand. D.h die LEDs blinken mit einer Frequenz von 7812.5 Hz. Das ist zuviel als dass wir es noch sehen könnten.

Was können wir also tun, um diese Blinkfrequenz zu verringern? Im Moment ist unsere einzige Einflussgröße der Vorteiler. Wie sieht die Rechnung aus, wenn wir einen Vorteiler von 1024 wählen?

Wiederum: Der Systemtakt sei 4 MHz. Durch den Vorteiler von 1024 werden daraus 4000000 / 1024 = 3906.25 Pulse pro Sekunde für den Timer. Der zählt wiederum 256 Zustände von einem Overflow zum nächsten. 3906.25 / 256 = 15.2587. Und wiederum: Im Overflow werden die LEDs ja abwechselnd ein und ausgeschaltet, also dividieren wir noch durch 2: 15.2587 / 2 = 7.629. Also knapp 7 Hz. Diese Frequenz müsste man schon mit freiem Auge sehen. Die LEDs werden ziemlich schnell vor sich hin blinken.

Reicht diese Verzögerung noch immer nicht, dann haben wir 2 Möglichkeiten:

- Entweder wir benutzen einen anderen Timer. Timer 1 beispielsweise ist ein 16 Bit Timer. Der Timer zählt also nicht von 0 bis 255 sondern von 0 bis 65535. Bei entsprechender Umarbeitung des Programms und einem Vorteiler von 1024 bedeutet das, dass die LEDs einen Ein/Aus Zyklus in 33 Sekunden absolvieren.
- Oder wir schalten die LEDs nicht bei jedem Timer Overflow um. Man könnte zum Beispiel in einem Register bis 7 zählen und nur dann, wenn dieses Register 7 erreicht hat, wird
    - das Register wieder auf 0 gesetzt und
    - die LEDs umgeschaltet.

Timer 0

Timer 0 ist ein 8 Bit Timer.

- Overflow Interrupt

TCCR0

TCCR - Timer/Counter Control Register

TCCR0

	CS02	CS01	CS00

CS02/CS00 - Clock Select

CS02	CS01	CS00	Bedeutung
0	0	0	keine (Der Timer ist angehalten)
0	0	1	Vorteiler: 1
0	1	0	Vorteiler: 8
0	1	1	Vorteiler: 64
1	0	0	Vorteiler: 256

1	0	1	Vorteiler: 1024
1	1	0	Vorteiler: Externer Takt vom Pin T0, fallende Flanke
1	1	1	Vorteiler: Externer Takt vom Pin T0, steigende Flanke

TIMSK

TIMSK - Timer/Counter Interrupt Mask Register

TIMSK

					OCIE0	**TOIE0**

TOIE0 - Timer 0 Overflow Interrupt Enable

Ist dieses Bit gesetzt, so wird beim Auftreten eines Overflows am Timer ein Interrupt ausgelöst.

Anstatt der Schreibweise

```
        ldi     temp, 0b00000001      ; TOIE0: Interrupt bei Timer Overflow
        out     TIMSK, temp
```

ist es besser, die Schreibweise

```
        ldi     temp, 1 << TOIE0
        out     TIMSK, temp
```

zu wählen, da hier unmittelbar aus dem Ladekommando hervorgeht, welche Bedeutung das gesetzte Bit hat. Die vorher inkludierte m8def.inc definiert dazu alles Notwendige.

Timer 1

Timer 1 ist ein 16 Bit Timer

- Overflow Interrupt
- Clear Timer on Compare Match
- Input Capture
- 2 Compare Einheiten
- div. PWM Modi

TCCR1B

TCCR1B

ICNC1	**ICES1**	**WGM13**	**WGM12**	**CS12**	**CS11**	**CS10**

CS12/CS10 - Clock Select

CS12	CS11	CS10	Bedeutung
0	0	0	keine (Der Timer ist angehalten)

0	0	1	Vorteiler: 1
0	1	0	Vorteiler: 8
0	1	1	Vorteiler: 64
1	0	0	Vorteiler: 256
1	0	1	Vorteiler: 1024
1	1	0	Vorteiler: Externer Takt vom Pin T1, fallende Flanke
1	1	1	Vorteiler: Externer Takt vom Pin T1, steigende Flanke

ICES1 - Input Capture Edge Select

ICES1 = 0 , falling edge ICES1 = 1 , rising edge

ICNC1 - Input Capture Noise Canceler

TCCR1A

TCCR1A

COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10

OCR1A

OCR1B

ICR1

TIMSK1

TIMSK

TICIE1	OCIE1A	OCIE1B	TOIE1

TICIE1 - Timer 1 Input Capture Interrupt Enable

OCIE1A - Timer 1 Output Compare A Match Interrupt Enable

OCIE1B - Timer 1 Output Compare B Match Interrupt Enable

TOIE1 - Timer 1 Overflow Interrupt Enable

Timer 2

Timer 2 ist ein 8 Bit Timer.

- Overflow Interrupt
- Compare Match Interrupt
- Clear Timer on Compare Match
- Phasen korrekte PWM

TCCR2

TCCR2							
FOC2	**WGM20**	**COM21**	**COM20**	**WGM21**	**CS22**	**CS21**	**CS20**

WGM21 - Waveform Generator Mode

Ist diese Option Aktiviert dann wird das Zählregister des Timers (TCNT2) wieder auf 0 gesetzt, sobal das Register as Output Compare Register (OCR2) und TCNT2 übereinstimmen. Zu beachten ist dabei das wenn das Register kleiner als 255 gestellt ist, wird der Timer nicht mehr überschritten und der Interrupt für den Overflow (TIMER2_OVF_vect) nicht mehr ausgelöst. Stattdessen gibt es den Interrupt für Compare Match (TIMER2_COMP_vect) Die Frequenz für die Interruptauslösung lässt sich dann wie folgt berechnen: ISR_Freq = F_CPU : (Prescaler * ORC2)

CS22/CS20 - Clock Select

CS22	CS21	CS20	Bedeutung
0	0	0	keine (Der Timer ist angehalten)
0	0	1	Vorteiler: 1
0	1	0	Vorteiler: 8
0	1	1	Vorteiler: 32
1	0	0	Vorteiler: 64
1	0	1	Vorteiler: 128
1	1	0	Vorteiler: 256
1	1	1	Vorteiler: 1024

OCR2

TIMSK2

TIMSK							
OCIE2	TOIE2						

OCIE2 - Timer 2 Output Compare Interrupt Enable

TOIE2 Timer 2 Overflow Interrupt Enable

Was geht noch mit einem Timer?

Timer sind sehr universelle Microcontroller-Bestandteile. Für weitergehende Studien ist es daher unerlässlich, das entsprechende Datenblatt des Microcontrollers zu studieren. Oft ist es z. B. möglich, dass der Timer bei Erreichen von bestimmten Zählerständen einen Ausgabe-Pin von sich aus ein-/aus-/umschaltet. Er erledigt dann das, was wir oben noch mit einem Interrupt gemacht haben, eigenständig komplett in Hardware. Bei einigen Timern ist es möglich, damit eine PWM (Pulsweiten-Modulation) aufzubauen.

Ein paar der Timermodule lassen sich auch als Counter verwenden. Damit kann man z. B. die Anzahl externer Ereignisse wie Schaltvorgänge eines Inkrementalgebers bestimmen.

Andere bieten die Möglichkeit, über einen externen Uhrenquarz getaktet zu werden (Anwendung z. B. eine "Echtzeituhr" oder als "Weckfunktion" aus einem Standby/Powerdownmodus).

Durch geschickte Umprogrammierung in Echtzeit lässt sich mit einem Timer eine PLL aufbauen, die sich fortwährend auf einen Eingangstakt synchronisiert. Damit wird vermieden, dass der Controller ein Signal ständig abfragen ("pollen") muss, sondern das Signal wird per Timer-Interrupt verarbeitet. Maßgeblich ist die Messung der aktuellen und Schätzung der kommenden Flanke mithilfe eines einstellbaren Taktteiler-Verhältnisses.

Weblinks

- Timer/Counter und PWM beim ATMega16 Mikrocontroller Proseminar von Marcel Jakobs, September 2006 (PDF)
- AVR Timer Calculator

AVR-Tutorial: Uhr

Eine beliebte Übung für jeden Programmierer ist die Implementierung einer Uhr. Die meisten Uhren bestehen aus einem Taktgeber und einer Auswerte- und Anzeigevorrichtung. Wir wollen hier beides mittels eines Programmes in einem Mikrocontroller realisieren. Voraussetzung für diese Fallstudie ist das Verständnis der Kapitel über

- Ansteuerung eines LC-Displays
- Timer

Inhaltsverzeichnis

- 1 Aufbau und Funktion
- 2 Das erste Programm
- 3 Ganggenauigkeit
- 4 Der CTC-Modus des Timers

Aufbau und Funktion

Die Aufgabe des Taktgebers, der uns einen möglichst konstanten und genauen Takt liefert, übernimmt ein Timer. Der Timer ermöglicht einen einfachen Zugang zum Takt, den der AVR vom Quarz abgreift. Wie schon im Einführungskapitel über den Timer beschrieben, wird dazu ein Timer Overflow Interrupt installiert, und in diesem Interrupt wird die eigentliche Uhr hochgezählt. Die Uhr besteht aus vier Registern. Drei davon repräsentieren die Sekunden, Minuten und Stunden unserer Uhr. Nach jeweils einer Sekunde wird das Sekundenregister um eins erhöht. Sind 60 Sekunden vergangen, wird das Sekundenregister wieder auf Null gesetzt und gleichzeitig das Minutenregister um eins erhöht. Dies ist ein Überlauf. Nach 60 Minuten werden die Minuten wieder auf Null gesetzt und für diese vergangenen 60 Minuten eine Stunde aufaddiert. Nach 24 Stunden schliesslich werden die Stunden wieder auf Null gesetzt, ein ganzer Tag ist vergangen.

Aber wozu das vierte Register?

Der Mikrocontroller wird mit 4 MHz betrieben. Bei einem Teiler von 1024 zählt der Timer also mit 4000000 / 1024 = 3906,25 Pulsen pro Sekunde. Der Timer muss einmal bis 256 zählen, bis er einen Überlauf auslöst. Es ereignen sich also 3906,25 / 256 = 15,2587 Überläufe pro Sekunde. Die Aufgabe des vierten Registers ist es nun, diese 15 Überläufe zu zählen. Bei Auftreten des 15. ist eine Sekunde vergangen. Dies stimmt jedoch nicht exakt, denn die Division weist ja auch Nachkommastellen auf, hat einen Rest, der hier im Moment der Einfachheit halber ignoriert wird. In einem späteren Abschnitt wird darauf noch eingegangen werden.

Im Überlauf-Interrupt wird also diese Kette von Zählvorgängen auf den Sekunden, Minuten und

Stunden durchgeführt und anschliessend zur Anzeige gebracht. Dazu werden die in einem vorhergehenden Kapitel entwickelten LCD Funktionen benutzt.

Das erste Programm

```
.include "m8def.inc"

.def temp1 = r16
.def temp2 = r17
.def temp3 = r18
.def flag  = r19

.def SubCount = r21
.def Sekunden = r22
.def Minuten  = r23
.def Stunden  = r24

.org 0x0000
        rjmp    main                ; Reset Handler
.org OVF0addr
        rjmp    timer0_overflow     ; Timer Overflow Handler

.include "lcd-routines.asm"

main:
        ldi     temp1, HIGH(RAMEND)
        out     SPH, temp1
        ldi     temp1, LOW(RAMEND)  ; Stackpointer initialisieren
        out     SPL, temp1

        ldi temp1, 0xFF    ; Port D = Ausgang
        out DDRD, temp1

        rcall   lcd_init
        rcall   lcd_clear


        ldi     temp1, (1<<CS02) | (1<<CS00)   ; Teiler 1024
        out     TCCR0, temp1

        ldi     temp1, 1<<TOIE0     ; TOIE0: Interrupt bei Timer Overflow
        out     TIMSK, temp1

        clr     Minuten             ; Die Uhr auf 0 setzen
        clr     Sekunden
        clr     Stunden
        clr     SubCount
        clr     Flag                ; Merker löschen

        sei

loop:
        cpi     flag,0
        breq    loop                ; Flag im Interrupt gesetzt?
        ldi     flag,0              ; flag löschen

        rcall   lcd_clear           ; das LCD löschen
        mov     temp1, Stunden      ; und die Stunden ausgeben
        rcall   lcd_number
        ldi     temp1, ':'          ; zwischen Stunden und Minuten einen ':'
        rcall   lcd_data
```

```
        mov     temp1, Minuten      ; dann die Minuten ausgeben
        rcall   lcd_number
        ldi     temp1, ':'          ; und noch ein ':'
        rcall   lcd_data
        mov     temp1, Sekunden     ; und die Sekunden
        rcall   lcd_number

        rjmp    loop

timer0_overflow:                    ; Timer 0 Overflow Handler

        push    temp1               ; temp 1 sichern
        in      temp1,sreg          ; SREG sichern
        push    temp1

        inc     SubCount            ; Wenn dies nicht der 15. Interrupt
        cpi     SubCount, 15        ; ist, dann passiert gar nichts
        brne    end_isr

                                    ; Überlauf
        clr     SubCount            ; SubCount rücksetzen
        inc     Sekunden            ; plus 1 Sekunde
        cpi     Sekunden, 60        ; sind 60 Sekunden vergangen?
        brne    Ausgabe             ; wenn nicht kann die Ausgabe schon
                                    ; gemacht werden

                                    ; Überlauf
        clr     Sekunden            ; Sekunden wieder auf 0 und dafür
        inc     Minuten             ; plus 1 Minute
        cpi     Minuten, 60         ; sind 60 Minuten vergangen ?
        brne    Ausgabe             ; wenn nicht, -> Ausgabe

                                    ; Überlauf
        clr     Minuten             ; Minuten zurücksetzen und dafür
        inc     Stunden             ; plus 1 Stunde
        cpi     Stunden, 24         ; nach 24 Stunden, die Stundenanzeige
        brne    Ausgabe             ; wieder zurücksetzen

                                    ; Überlauf
        clr     Stunden             ; Stunden rücksetzen

Ausgabe:
        ldi     flag,1              ; Flag setzen, LCD updaten

end_isr:

        pop     temp1
        out     sreg,temp1          ; sreg wieder herstellen
        pop     temp1
        reti                        ; das wars. Interrupt ist fertig

; Eine Zahl aus dem Register temp1 ausgeben

lcd_number:
        push    temp2               ; register sichern,
                                    ; wird für Zwsichenergebnisse gebraucht
        ldi     temp2, '0'
lcd_number_10:
        subi    temp1, 10           ; abzählen wieviele Zehner in
        brcs    lcd_number_1        ; der Zahl enthalten sind
        inc     temp2
        rjmp    lcd_number_10
lcd_number_1:
        push    temp1               ; den Rest sichern
```

```
(http://www.mikrocontroller.net/topic/172026)
        mov     temp1,temp2         ;
        rcall   lcd_data            ; die Zehnerstelle ausgeben
        pop     temp1               ; den Rest wiederherstellen
        subi    temp1, -10          ; 10 wieder dazuzählen, da die
                                    ; vorhergehende Schleife 10 zuviel
                                    ; abgezogen hat
                                    ; das Subtrahieren von -10
                                    ; = Addition von +10 ist ein Trick
                                    ; da kein addi Befehl existiert
        ldi     temp2, '0'          ; die übrig gebliebenen Einer
        add     temp1, temp2        ; noch ausgeben
        rcall   lcd_data

        pop     temp2               ; Register wieder herstellen
        ret
```

In der ISR wird nur die Zeit in den Registern neu berechnet, die Ausgabe auf das LCD erfolgt in der Hauptschleife. Das ist notwendig, da die LCD-Ausgabe bisweilen sehr lange dauern kann. Wenn sie länger als ~2/15 Sekunden dauert werden Timerinterrupts "verschluckt" und unsere Uhr geht noch mehr falsch. Dadurch, dass aber die Ausgabe in der Hauptschleife durchgeführt wird, welche jederzeit durch einen Timerinterrupt unterbrochen werden kann, werden keine Timerinterrupts verschluckt. Das ist vor allem wichtig, wenn mit höheren Interruptfrequenzen gearbeitet wird, z. B. 1/100s im Beispiel weiter unten. Auch wenn in diesem einfachen Beispiel die Ausgabe bei weitem nicht 2/15 Sekunden dauert, sollte man sich diesen Programmierstil allgemein angewöhnen. Siehe auch Interrupt.

Eine weitere Besonderheit ist das Register **flag** (=r19). Dieses Register fungiert als Anzeiger, wie eine Flagge, daher auch der Name. In der ISR wird dieses Register auf 1 gesetzt. Die Hauptschleife, also alles zwischen *loop* und *rjmp loop*, prüft dieses Flag und nur dann, wenn das Flag auf 1 steht, wird die LCD Ausgabe gemacht und das Flag wieder auf 0 zurückgesetzt. Auf diese Art wird nur dann Rechenzeit für die LCD Ausgabe verbraucht, wenn dies tatsächlich notwendig ist. Die ISR teilt mit dem Flag der Hauptschleife mit, dass eine bestimmte Aufgabe, nämlich der Update der Anzeige gemacht werden muss und die Hauptschleife reagiert darauf bei nächster Gelegenheit, indem sie diese Aufgabe ausführt und setzt das Flag zurück. Solche Flags werden daher auch **Job-Flags** genannt, weil durch ihr setzten das Abarbeiten eines Jobs (einer Aufgabe) angestoßen wird. Auch hier gilt wieder: Im Grunde würde man in diesem speziellen Beispiel kein Job-Flag benötigen, weil es in der Hauptschleife nur einen einzigen möglichen Job, die Neuausgabe der Uhrzeit, gibt. Sobald aber Programme komplizierter werden und mehrere Jobs möglich sind, sind Job-Flags eine gute Möglichkeit, ein Programm so zu organsieren, dass bestimmte Dinge nur dann gemacht werden, wenn sie notwendig sind.

Im Moment gibt es keine Möglichkeit, die Uhr auf eine bestimmte Uhrzeit einzustellen. Um dies tun zu können, müssten noch zusätzlich Taster an den Mikrocontroller angeschlossen werden, mit deren Hilfe die Sekunden, Minuten und Stunden händisch vergrößert bzw. verkleinert werden können. Studieren Sie mal die Bedienungsanleitung einer käuflichen Digitaluhr und versuchen sie zu beschreiben, wie dieser Stellvorgang bei dieser Uhr vor sich geht. Sicherlich werden Sie daraus eine Idee entwickeln können, wie ein derartiges Stellen mit der hier vorgestellten Digitaluhr funktionieren könnte. Als Zwischenlösung kann man im Programm die Uhr beim Start anstelle von 00:00:00 z. B. auch auf 20:00:00 stellen und exakt mit dem Start der Tagesschau starten. Wobei der

Start der Tagesschau verzögert bei uns ankommt, je nach Übertragung können das mehrere Sekunden sein.

Ganggenauigkeit

Wird die Uhr mit einer gekauften Uhr verglichen, so stellt man schnell fest, dass sie ganz schön ungenau geht. Sie geht vor! Woran liegt das? Die Berechnung sieht so aus:

- Frequenz des Quarzes: 4.0 MHz
- Vorteiler des Timers: 1024
- Überlauf alle 256 Timertakte

Daraus errechnet sich, dass in einer Sekunde 4000000 / 1024 / 256 = 15.258789 Overflow Interrupts auftreten. Im Programm wird aber bereits nach 15 Overflows eine Sekunde weitergeschaltet, daher geht die Uhr vor. Rechnen wir etwas:

$$F_r = (\frac{15}{15,258789} - 1) \cdot 100\% = -1,69\%$$

So wie bisher läuft die Uhr also rund 1.7 % zu schnell. In einer Minute ist das immerhin etwas mehr als eine Sekunde. Im Grunde ist das ein ähnliches Problem wie mit unserer Jahreslänge. Ein Jahr dauert nicht exakt 365 Tage, sondern in etwa einen viertel Tag länger. Die Lösung, die im Kalender dafür gemacht wurde - der Schalttag -, könnte man fast direkt übernehmen. Nach 3 Stück 15er Overflow Sekunden folgt eine Sekunde für die 16 Overflows ablaufen müssen. Wie sieht die Rechnung bei einem 15, 15, 15, 16 Schema aus? Für 4 Sekunden werden exakt 15.258789 * 4 = 61,035156 Overflow Interrupts benötigt. Mit einem 15, 15, 15, 16 Schema werden in 4 Sekunden genau 61 Overflow Interrupts durchgeführt. Der relative Fehler beträgt dann

$$F_r = (\frac{61}{61,035156} - 1) \cdot 100\% = -0,0575\%$$

Mit diesem Schema ist der Fehler beträchtlich gesunken. Nur noch 0.06%. Bei dieser Rate muss die Uhr immerhin etwas länger als 0,5 Stunden laufen, bis der Fehler auf eine Sekunde angewachsen ist. Das sind aber immer noch 48 Sekunden pro Tag bzw. 1488 Sekunden (=24,8 Minuten) pro Monat. So schlecht sind nicht mal billige mechanische Uhren!

Jetzt könnte man natürlich noch weiter gehen und immer kompliziertere "Schalt-Overflow"-Schemata austüfteln und damit die Genauigkeit näher an 100% bringen. Aber gibt es noch andere Möglichkeiten?

Im ersten Ansatz wurde ein Vorteiler von 1024 eingesetzt. Was passiert bei einem anderen Vorteiler? Nehmen wir mal einen Vorteiler von 64. Das heißt, es müssen (4000000 / 64) / 256 = 244.140625 Overflows auflaufen, bis 1 Sekunde vergangen ist. Wenn also 244 Overflows gezählt werden, dann beläuft sich der Fehler auf

$$F_r = (\frac{244}{244,140625} - 1) \cdot 100\% = -0,0576\%$$

Nicht schlecht. Nur durch Verändern von 2 Zahlenwerten im Programm (Teilerfaktor und Anzahl der Overflow Interrupts bis zu einer Sekunden) kann die Genauigkeit gegenüber dem ursprünglichen Overflow-Schema beträchtlich gesteigert werden. Aber geht das noch besser? Ja das geht. Allerdings nicht mit dem Overflow Interrupt.

Der CTC-Modus des Timers

Worin liegt denn das eigentliche Problem, mit dem die Uhr zu kämpfen hat? Es liegt darin, dass jedesmal ein kompletter Timerzyklus bis zum Overflow abgewartet werden muss, um darauf zu reagieren. Da aber nur jeweils ganzzahlige Overflowzyklen abgezählt werden können, heißt das auch, dass im ersten Fall nur in Vielfachen von 1024 * 256 = 262.144 Takten operiert werden kann, während im letzten Fall immerhin schon eine Granulierung von 64 * 256 = 16.384 Takten erreicht wird. Aber offensichtlich ist das nicht genau genug. Bei 4 MHz entsprechen 262.144 Takte bereits einem Zeitraum von 65,5 ms, während 16384 Takte einem Zeitbedarf von 4,096 ms entsprechen. Beide Zahlen teilen aber 1.000 ms nicht ganzzahlig, Nachkommareste fallen unter den Tisch und daraus summiert sich der Fehler auf. Angestrebt wird ein Timer, der seinen *Overflow* so erreicht, dass sich ein ganzzahliger Teiler von einer Sekunde einstellt. Dann gibt es keinen Rest. Dazu müsste man dem Timer aber vorschreiben können, bei welchem Zählerstand der *Overflow* erfolgen soll. Und genau dies ist im **CTC**-Modus möglich, allerdings nur beim Timer 1. **CTC** bedeutet "**C**lear **T**imer on **C**ompare match".

Timer 1, ein 16-Bit-Timer, wird mit einem Vorteiler von 1 betrieben. Dadurch wird erreicht, dass der Timer mit höchster Zeitauflösung arbeiten kann. Bei jedem Ticken des Systemtaktes von 4 MHz wird auch der Timer um 1 erhöht. Zusätzlich wird noch das WGM12-Bit bei der Konfiguration gesetzt. Dadurch wird der Timer in den **CTC**-Modus gesetzt. Dabei wird der Inhalt des Timers hardwaremäßig mit dem Inhalt des **OCR1A**-Registers verglichen. Stimmen beide überein, so wird der Timer auf 0 zurückgesetzt und im nächsten Taktzyklus ein **OCIE1A**-Interrupt ausgelöst. Dadurch ist es möglich, exakt die Anzahl an Taktzyklen festzulegen, die von einem Interrupt zum nächsten vergehen sollen. Das Compare Register **OCR1A** wird mit dem Wert 39.999 vorbelegt. Dadurch vergehen exakt 40.000 Taktzyklen von einem Compare-Interrupt zum nächsten. "Zufällig" ist dieser Wert so gewählt, dass bei einem Systemtakt von 4 MHz von einem Interrupt zum nächsten genau 1/100 Sekunde vergeht, denn 40.000 / 4.000.000 = 0,01. Bei einem möglichen Umbau der Uhr zu einer Stoppuhr könnte sich die Hundertstelsekunde als nützlich erweisen. Im Interrupt wird das Hilfsregister SubCount bis 100 hochgezählt, und nach 100 Interrupts kommt wieder die Sekundenweiterschaltung wie oben in Gang.

```
.include "m8def.inc"

.def temp1 = r16
.def temp2 = r17
.def temp3 = r18
.def Flag  = r19

.def SubCount = r21
.def Sekunden = r22
.def Minuten  = r23
.def Stunden  = r24
```

```
.org 0x0000
           rjmp    main             ; Reset Handler
.org OC1Aaddr
           rjmp    timer1_compare   ; Timer Compare Handler

.include "lcd-routines.asm"

main:
        ldi     temp1, HIGH(RAMEND)
        out     SPH, temp1
        ldi     temp1, LOW(RAMEND)  ; Stackpointer initialisieren
        out     SPL, temp1

        ldi temp1, 0xFF    ; Port D = Ausgang
        out DDRD, temp1

        rcall   lcd_init
        rcall   lcd_clear

                                    ; Vergleichswert
        ldi     temp1, high( 40000 - 1 )
        out     OCR1AH, temp1
        ldi     temp1, low( 40000 - 1 )
        out     OCR1AL, temp1
                                    ; CTC Modus einschalten
                                    ; Vorteiler auf 1
        ldi     temp1, ( 1 << WGM12 ) | ( 1 << CS10 )
        out     TCCR1B, temp1

        ldi     temp1, 1 << OCIE1A  ; OCIE1A: Interrupt bei Timer Compare
        out     TIMSK, temp1

        clr     Minuten             ; Die Uhr auf 0 setzen
        clr     Sekunden
        clr     Stunden
        clr     SubCount
        clr     Flag                ; Flag löschen

        sei
loop:
        cpi     flag,0
        breq    loop                ; Flag im Interrupt gesetzt?
        ldi     flag,0              ; Flag löschen

        rcall   lcd_clear           ; das LCD löschen
        mov     temp1, Stunden      ; und die Stunden ausgeben
        rcall   lcd_number
        ldi     temp1, ':'          ; zwischen Stunden und Minuten einen ':'
        rcall   lcd_data
        mov     temp1, Minuten      ; dann die Minuten ausgeben
        rcall   lcd_number
        ldi     temp1, ':'          ; und noch ein ':'
        rcall   lcd_data
        mov     temp1, Sekunden     ; und die Sekunden
        rcall   lcd_number

        rjmp    loop

timer1_compare:                     ; Timer 1 Output Compare Handler

        push    temp1               ; temp 1 sichern
        in      temp1,sreg          ; SREG sichern

        inc     SubCount            ; Wenn dies nicht der 100. Interrupt
```

```
        cpi     SubCount, 100       ; ist, dann passiert gar nichts
        brne    end_isr

                                    ; Überlauf
        clr     SubCount            ; SubCount rücksetzen
        inc     Sekunden            ; plus 1 Sekunde
        cpi     Sekunden, 60        ; sind 60 Sekunden vergangen?
        brne    Ausgabe             ; wenn nicht kann die Ausgabe schon
                                    ; gemacht werden

                                    ; Überlauf
        clr     Sekunden            ; Sekunden wieder auf 0 und dafür
        inc     Minuten             ; plus 1 Minute
        cpi     Minuten, 60         ; sind 60 Minuten vergangen ?
        brne    Ausgabe             ; wenn nicht, -> Ausgabe

                                    ; Überlauf
        clr     Minuten             ; Minuten zurücksetzen und dafür
        inc     Stunden             ; plus 1 Stunde
        cpi     Stunden, 24         ; nach 24 Stunden, die Stundenanzeige
        brne    Ausgabe             ; wieder zurücksetzen

                                    ; Überlauf
        clr     Stunden             ; Stunden rücksetzen

Ausgabe:
        ldi     flag,1              ; Flag setzen, LCD updaten

end_isr:

        out     sreg,temp1          ; sreg wieder herstellen
        pop     temp1
        reti                        ; das wars. Interrupt ist fertig

; Eine Zahl aus dem Register temp1 ausgeben

lcd_number:
        push    temp2               ; register sichern,
                                    ; wird für Zwsichenergebnisse gebraucht
        ldi     temp2, '0'
lcd_number_10:
        subi    temp1, 10           ; abzählen wieviele Zehner in
        brcs    lcd_number_1        ; der Zahl enthalten sind
        inc     temp2
        rjmp    lcd_number_10
lcd_number_1:
        push    temp1               ; den Rest sichern
(http://www.mikrocontroller.net/topic/172026)
        mov     temp1,temp2         ;
        rcall   lcd_data            ; die Zehnerstelle ausgeben
        pop     temp1               ; den Rest wieder holen
        subi    temp1, -10          ; 10 wieder dazuzählen, da die
                                    ; vorhergehende Schleife 10 zuviel
                                    ; abgezogen hat
                                    ; das Subtrahieren von -10
                                    ; = Addition von +10 ist ein Trick
                                    ; da kein addi Befehl existiert
        ldi     temp2, '0'          ; die übrig gebliebenen Einer
        add     temp1, temp2        ; noch ausgeben
        rcall   lcd_data

        pop     temp2               ; Register wieder herstellen
        ret
```

In der Interrupt-Routine werden wieder, genauso wie vorher, die Anzahl der Interrupt-Aufrufe gezählt. Beim 100. Aufruf sind daher 40.000 * 100 = 4.000.000 Takte vergangen und da der Quarz mit 4.000.000 Schwingungen in der Sekunde arbeitet, ist daher eine Sekunde vergangen. Sie wird genauso wie vorher registriert und die Uhr entsprechend hochgezählt. Wird jetzt die Uhr mit einer kommerziellen verglichen, dann fällt nach einiger Zeit auf ... Sie geht immer noch falsch! Was ist jetzt die Ursache? Die Ursache liegt in einem Problem, das nicht direkt behebbar ist. Am Quarz! Auch wenn auf dem Quarz drauf steht, dass er eine Frequenz von 4MHz hat, so stimmt das nicht exakt. Auch Quarze haben Fertigungstoleranzen und verändern ihre Frequenz mit der Temperatur. Typisch liegt die Fertigungstoleranz bei +/- 100ppm = 0,01% (**p**arts **p**er **m**illion, Millionstel Teile), die Temperaturdrift zwischen -40 Grad und 85 Grad liegt je nach Typ in der selben Größenordnung. Das bedeutet, dass die Uhr pro Monat um bis zu 268 Sekunden (~4 1/2 Minuten) falsch gehen kann. Diese Einflüsse auf die Quarzfrequenz sind aber messbar und per Hardware oder Software behebbar. In Uhren kommen normalerweise genauer gefertigte Uhrenquarze zum Einsatz, die vom Uhrmacher auch noch auf die exakte Frequenz abgeglichen werden (mittels Kondensatoren und Frequenzzähler). Ein Profi verwendet einen sehr genauen Frequenzzähler, womit er innerhalb weniger Sekunden die Frequenz sehr genau messen kann. Als Hobbybastler kann man die Uhr eine zeitlang (Tage, Wochen) laufen lassen und die Abweichung feststellen (z. B. exakt 20:00 Uhr zum Start der Tagsschau). Aus dieser Abweichung lässt sich dann errechnen, wie schnell der Quarz wirklich schwingt. Und da dank CTC die Messperiode taktgenau eingestellt werden kann, ist es möglich, diesen Frequenzfehler auszugleichen. Der genaue Vorgang ist in dem Wikiartikel AVR - Die genaue Sekunde / RTC beschrieben.

AVR-Tutorial: ADC

Inhaltsverzeichnis

- 1 Was macht der ADC?
- 2 Elektronische Grundlagen
  - 2.1 Beschaltung des ADC-Eingangs
  - 2.2 Referenzspannung AREF
- 3 Ein paar ADC-Grundlagen
- 4 Umrechnung des ADC Wertes in eine Spannung
- 5 Kalibrierung
- 6 Die Steuerregister des ADC
  - 6.1 ADMUX
  - 6.2 ADCSRA
- 7 Die Ergebnisregister ADCL und ADCH
- 8 Beispiele
  - 8.1 Ausgabe als ADC-Wert
  - 8.2 Ausgabe als Spannungswert

Was macht der ADC?

Wenn es darum geht, Spannungen zu messen, wird der Analog-/Digital-Wandler (kurz: A/D-Wandler) oder englisch **A**nalog **D**igital **C**onverter (ADC) benutzt. Er konvertiert eine elektrische Spannung in eine Digitalzahl. Prinzipiell wird dabei die Messspannung mit einer Referenzspannung verglichen. Die Zahl drückt daher das Verhältnis der Messspannung zu dieser Referenzspannung aus. Sie kann in gewohnter Weise von einem Mikrocontroller weiterverarbeitet werden.

Elektronische Grundlagen

Die ADC-Versorgungsspannung (AVCC) darf maximal um +/-0,3V von der Versorgung des Digitalteils (VCC) abweichen, jedoch nicht 5,5V überschreiten. Die externe Referenzspannung VREF darf nicht kleiner als die im Datenblatt unter ADC Characteristics als VREFmin angegebene Spannung (z. B. ATmega8: 2V, ATmega644P: 1V) und nicht größer als AVCC sein. Die Spannungen an den Wandlereingängen müssen im Intervall GND ≤ VIN ≤ VREF liegen.

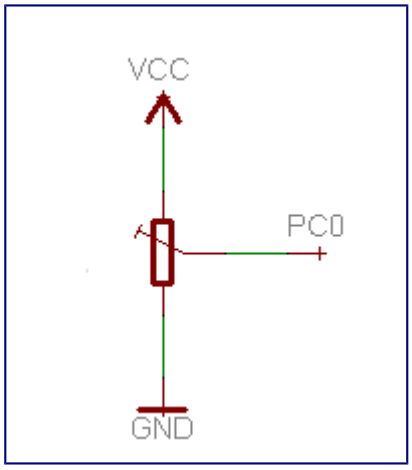
Im Extremfall bedeutet dies: Sei VCC = 5,5V, folgt AVCC_max = VREF_max = VIN_max = 5,5V.

Der Eingangswiderstand des ADC liegt in der Größenordnung von einigen Megaohm, so dass der

ADC die Signalquelle praktisch nicht belastet. Desweiteren enthält der Mikrocontroller eine sog. **Sample&Hold** Schaltung. Dies ist wichtig, wenn sich während des Wandlungsvorgangs die Eingangsspannung verändert, da die AD-Wandlung eine bestimmte Zeit dauert. Die Sample&Hold-Stufe speichert zum Beginn der Wandlung die anliegende Spannung und hält sie während des Wandlungsvorgangs konstant.

Beschaltung des ADC-Eingangs

Um den ADC im Folgenden zu testen wird eine einfache Schaltung an den PC0-Pin des ATmega8 angeschlossen. Dies ist der ADC-Kanal 0. Bei anderen AVR-Typen liegt der entsprechende Eingang auf einem andern Pin, hier ist ein Blick ins Datenblatt angesagt.



Testschaltung

Der Wert des Potentiometers ist Dank des hohen Eingangswiderstandes des ADC ziemlich unkritisch. Es kann jedes Potentiometer von 1kΩ bis 1MΩ benutzt werden.

Wenn andere Messgrößen gemessen werden sollen, so bedient man sich oft und gern des Prinzips des Spannungsteilers. Der Sensor ist ein veränderlicher Widerstand. Zusammen mit einem zweiten, konstanten Widerstand bekannter Größe wird ein Spannungsteiler aufgebaut. Aus der Variation der durch den variablen Spannungsteiler entstehenden Spannung kann auf den Messwert zurückgerechnet werden.

```
Vcc ----------+                Vcc ---------+
              |                             |
             ---                        Sensor,
             | |                   der seinen Widerstand
             | |                   in Abhängigkeit der
             ---                   Messgröße ändert
              |                            |
              +------- PC0                 +-------- PC0
              |                            |
          Sensor,                         ---
     der seinen Widerstand                | |
     in Abhängigkeit der                  | |
     Messgröße ändert                     ---
              |                            |
 GND ---------+                 GND --------+
```

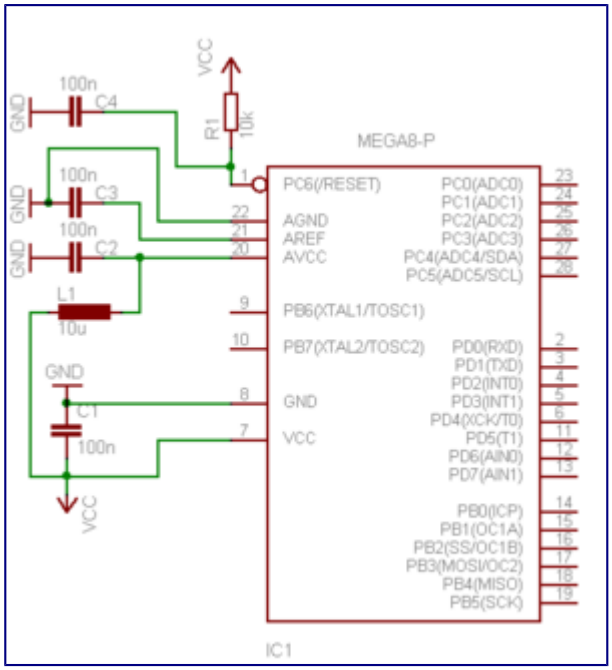
Die Größe des zweiten Widerstandes im Spannungsteiler richtet sich nach dem Wertebereich, in welchem der Sensor seinen Wert ändert. Als Daumenregel kann man sagen, dass der Widerstand so

gross sein sollte wie der Widerstand des Sensors in der Mitte des Messbereichs.

Beispiel: Wenn ein Temperatursensor seinen Widerstand von 0..100 Grad von 2kΩ auf 5kΩ ändert, sollte der zweite Widerstand eine Grösse von etwa (2+5)/2 = 3,5kΩ haben.

Aber egal wie immer man das auch macht, der entscheidende Punkt besteht darin, dass man seine Messgröße in eine veränderliche Spannung 'übersetzt' und mit dem ADC des Mega8 die Höhe dieser Spannung misst. Aus der Höhe der Spannung kann dann wieder in der Umkehrung auf die Messgröße zurückgerechnet werden.

Referenzspannung AREF



Beschaltung von A_{REF}

Der ADC benötigt für seine Arbeit eine Referenzspannung. Dabei gibt es 2 Möglichkeiten:

- interne Referenzspannung
- externe Referenzspannung

Bei der Umstellung der Referenzspannung sind Wartezeiten zu beachten, bis die ADC-Hardware einsatzfähig ist (Datenblatt und [1]).

Interne Referenzspannung

Mittels Konfigurationsregister können beim ATmega8 verschiedene Referenzspannungen eingestellt werden. Dies umfasst die Versorgungsspannung AVcc sowie eine vom AVR bereitgestellte Spannung von 2,56V (bzw. bei den neueren AVRs 1,1V, wie z. B. beim ATtiny13, ATmega48, 88, 168, ...). In beiden Fällen wird an den AREF-Pin des Prozessors ein Kondensator von 100nF als Minimalbeschaltung nach Masse angeschlossen, um die Spannung zu puffern/glätten. Es ist jedoch zu beachten, dass die interne Referenzspannung ca. +/-10% vom Nominalwert abweichen kann, vgl. dazu das Datenblatt Abschnitt ADC Characteristics VINT (z. B. ATmega8: 2,3-2,9V, ATmega324P: 2,33-2,79V bzw. 1,0-1,2V "Values are guidelines only."). Die typische Abweichung der internen Referenzspannung vom Sollwert bei einigen AVR-Controllern wird in dieser Testschaltung

exemplarisch untersucht.

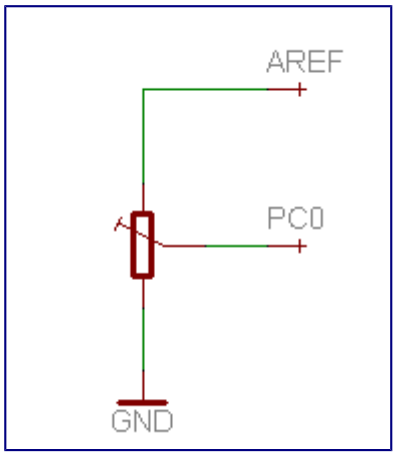
Externe Referenzspannung

Wird eine externe Referenz verwendet, so wird diese an AREF angeschlossen. Aber aufgepasst! Wenn eine Referenz in Höhe der Versorgungsspannung benutzt werden soll, so ist es besser, dies über die interne Referenz zu tun. Außer bei anderen Spannungen als 5V bzw. 2,56V gibt es eigentlich keinen Grund, an AREF eine Spannungsquelle anzuschließen. In Standardanwendungen fährt man immer besser, wenn die interne Referenzspannung mit einem Kondensator an AREF benutzt wird. Die 10µH-Spule L1 kann man meist auch durch einen 47Ω-Widerstand ersetzen.

Ein paar ADC-Grundlagen

Der ADC ist ein 10-Bit ADC, d.h. er liefert Messwerte im Bereich 0 bis 1023. Liegt am Eingangskanal 0V an, so liefert der ADC einen Wert von 0. Hat die Spannung am Eingangskanal die Referenzspannung erreicht (stimmt nicht ganz), so liefert der ADC einen Wert von 1023. Unterschreitet oder überschreitet die zu messende Spannung diese Grenzen, so liefert der ADC 0 bzw. 1023. Wird die Auflösung von 10 Bit nicht benötigt, so ist es möglich die Ausgabe durch ein Konfigurationsregister so einzuschränken, dass ein leichter Zugriff auf die 8 höchstwertigen Bits möglich ist.

Wie bei vielen analogen Schaltungen, unterliegt auch der ADC einem Rauschen. Das bedeutet, dass man nicht davon ausgehen sollte, dass der ADC bei konstanter Eingangsspannung auch immer denselben konstanten Wert ausgibt. Ein "Zittern" der niederwertigsten 2 Bits ist durchaus nicht ungewöhnlich. Besonders hervorgehoben werden soll an dieser Stelle nochmals die Qualität der Referenzspannung. Diese Qualität geht in erheblichem Maße in die Qualität der Wandlergebnisse ein. Die Beschaltung von AREF mit einem Kondensator ist die absolut notwendige Mindestbeschaltung, um eine einigermaßen akzeptable Referenzspannung zu erhalten. Reicht dies nicht aus, so kann die Qualität einer Messung durch *Oversampling* erhöht werden. Dazu werden mehrere Messungen gemacht und deren Mittelwert gebildet.



Oft interessiert auch der absolute Spannungspegel nicht. Im Beschaltungsbeispiel oben ist man normalerweise nicht direkt an der am Poti entstehenden Spannung interessiert. Viel mehr ist diese Spannung nur ein notwendiges Übel, um die Stellung des Potis zu bestimmen. In solchen Fällen kann die Poti-Beschaltung wie folgt abgewandelt werden:

Hier wird AREF (bei interner Referenz) als vom µC gelieferte Spannung benutzt und vom Spannungsteiler bearbeitet wieder an den µC zur Messung zurückgegeben. Dies hat den Vorteil, dass der Spannungsteiler automatisch Spannungen bis zur Höhe der Referenzspannung ausgibt, ohne dass eine externe Spannung mit AREF abgeglichen werden müsste. Selbst Schwankungen in AREF wirken sich hier nicht mehr aus, da ja das Verhältnis der Spannungsteilerspannung zu AREF immer konstant bleibt (ratiometrische Messung). Und im Grunde bestimmt der ADC ja nur dieses Verhältnis. Wird diese Variante gewählt, so muss berücksichtigt werden, dass die Ausgangsspannung an AREF nicht allzusehr belastet wird. Der Spannungsteiler muss einen Gesamtwiderstand von deutlich über 10kΩ besitzen. Werte von 100kΩ oder höher sind anzustreben. Verwendet man anstatt AREF AVCC und schaltet auch die Referenzspannung auf AVCC um, ist die Belastung durch den Poti unkritisch, weil hier die Stromversorgung direkt zur Speisung verwendet wird.

Ist hingegen die absolute Spannung von Interesse, so muss man darauf achten, dass ein ADC in digitalen Bereichen arbeitet (Quantisierung). An einem einfacheren Beispiel soll demonstriert werden was damit gemeint ist.

Angenommen der ADC würde nur 5 Stufen auflösen können und AREF sei 5V:

```
Volt    Wert vom ADC

 0 -+
    |         0
 1 -+
    |         1
 2 -+
    |         2
 3 -+
    |         3
 4 -+
    |         4
 5 -+
```

Ein ADC Wert von 0 bedeutet also keineswegs, dass die zu messende Spannung exakt den Wert 0 hat. Es bedeutet lediglich, dass die Messspannung irgendwo im Bereich von 0V bis 1V liegt. Sinngemäß bedeutet daher auch das Auftreten des Maximalwertes nicht, dass die Spannung exakt AREF beträgt, sondern lediglich, dass die Messspannung sich irgendwo im Bereich der letzten Stufe (also von 4V bis 5V) bewegt.

Umrechnung des ADC Wertes in eine Spannung

Die Größe eines "Bereiches" bestimmt sich also zu

$$Bereichsbreite = \frac{Referenzspannung}{Maximalwert + 1}$$

Der Messwert vom ADC rechnet sich dann wie folgt in eine Spannung um:

$$Spannung = ADCwert \cdot \frac{Referenzspannung}{Maximalwert}$$

Wird der ADC also mit 10 Bit an 5 V betrieben, so lauten die Umrechnungen:

$$Bereichsbreite = \frac{5\ \mathrm{V}}{1024} = 0,004883\ \mathrm{V} = 4,883\ \mathrm{mV}$$

$$Spannung = ADCwert \cdot 4,883\ \mathrm{mV}$$

Wenn man genau hinsieht stellt man fest, dass sowohl die Referenzspannung als auch der Maximalwert Konstanten sind. D.h. der Quotient aus Referenzspannung und Maximalwert ist konstant. Somit muss nicht immer eine Addition und Division ausgeführt werden, sondern nur eine Multiplikation! Das spart viel Aufwand und Rechenzeit! Dabei kann sinnvollerweise Festkommaarithmetik zum Einsatz kommen.

Kalibrierung

Hat man eine externe, genaue Referenzspannung zur Hand, dann kann ein Korrekturfaktor berechnet werden, mit dem die Werte des ADCs im Nachhinein korrigiert werden können. Dies geschieht normalerweise über eine sogenannte gain offset Korrektur an einer Geraden oder einer Parabel. In erster Näherung kann man auch die interne Referenzspannung um das Inverse des ermittelten Korrekturwertes verstellen, um einen genaueren bereits digitalisierten Wert zu bekommen.

Die Steuerregister des ADC

ADMUX

ADMUX						
REFS1	**REFS0**	**ADLAR**	**MUX3**	**MUX2**	**MUX1**	**MUX0**

- Referenzspannung *REFS1, REFS0*

REFS1	REFS0	Referenz
0	0	externe Referenz
0	1	interne Referenz: Avcc
1	0	wird beim Mega8 nicht benutzt
1	1	interne Referenz: 2.56 Volt

- Ausrichtung *ADLAR*

ADLAR

0	Das Ergebnis wird in den Registern ADCH/ADCL rechtsbündig ausgerichtet. Die 8 niederwertigsten Bits des Ergebnisses werden in ADCL abgelegt. Die verbleibenden 2 Bits des Ergebnisses werden im Register ADCH in den Bits 0 und 1 abgelegt.
1	Das Ergebnis wird in den Registern ADCH/ADCL linksbündig ausgerichtet. Die 8 höchstwertigen Bits des Ergebnisses werden in ADCH abgelegt. Die verbleibenden 2 niederwertigen Bits werden im Register ADCL in den Bits 6 und 7 abgelegt.

- Kanalwahl *MUX3, MUX2, MUX1, MUX0*

MUX3	MUX2	MUX1	MUX0	Kanal
0	0	0	0	Kanal 0, Pin PC0
0	0	0	1	Kanal 1, Pin PC1
0	0	1	0	Kanal 2, Pin PC2
0	0	1	1	Kanal 3, Pin PC3
0	1	0	0	Kanal 4, Pin PC4
0	1	0	1	Kanal 5, Pin PC5
0	1	1	0	Kanal 6 (*)
0	1	1	1	Kanal 7 (*)
1	1	1	0	1.23V, Vbg
1	1	1	1	0V, GND

(*) Bei Atmega8 nur in der Gehäusebauform TQFP und MLF verfügbar, nicht in PDIP

ADCSRA

ADCSRA

ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADEN
: "ADC Enable": Mittels ADEN wird der ADC ein und ausgeschaltet. Eine 1 an dieser Bitposition schaltet den ADC ein.

ADSC
: "ADC Start Conversion": Wird eine 1 an diese Bitposition geschrieben, so beginnt der ADC mit der Wandlung. Das Bit bleibt auf 1, solange die Wandlung im Gange ist. Wenn die Wandlung beendet ist, wird dieses Bit von der ADC Hardware wieder auf 0 gesetzt.

ADFR
: "ADC Free Running": Wird eine 1 an ADFR geschrieben, so wird der ADC im Free Running Modus betrieben. Dabei startet der ADC nach dem Abschluss einer Messung automatisch die nächste Messung. Die erste Messung wird ganz normal über das Setzen des ADSC-Bits gestartet.

ADIF
: "ADC Interrupt Flag": Wenn eine Messung abgeschlossen ist, wird das ADIF Bit gesetzt. Ist zusätzlich noch das *ADIE* Bit gesetzt, so wird ein Interrupt ausgelöst und der entsprechende Interrupt Handler angesprungen.

ADIE
: "ADC Interrupt Enable": Wird eine 1 an ADIE geschrieben, so löst der ADC nach Beendigung einer Messung einen Interrupt aus.

ADPS2, ADPS1, ADPS0
: "ADC Prescaler": Mit dem Prescaler kann die ADC-Frequenz gewählt werden. Laut Datenblatt sollte diese für die optimale Auflösung zwischen 50kHz und 200kHz liegen. Ist die Wandlerfrequenz langsamer eingestellt, kann es passieren dass die eingebaute Sample & Hold Schaltung die Eingangsspannung nicht lange genug konstant halten kann. Ist die Frequenz aber zu schnell eingestellt, dann kann es passieren dass sich die Sample & Hold Schaltung nicht schnell genug an die Eingangsspannung anpassen kann.

ADPS2	ADPS1	ADPS0	Vorteiler

0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Beispiel

8 MHz Prozessortakt: 8.000.000Hz / 200.000Hz = 40

Da mit 200kHz gerechnet wurde(Maximale Frequenz), nimmt man den nächst höheren Wert, also 64.

8.000.000 Hz / 64 = 125.000Hz = 125kHz So erhält man bei 8 MHz einen Prescaler von 64 und eine Frequenz von 125kHz.

Die Ergebnisregister ADCL und ADCH

Da das Ergebnis des ADC ein 10 Bit Wert ist, passt dieser Wert naturgemäß nicht in ein einzelnes Register, das ja bekanntlich nur 8 Bit breit ist. Daher wird das Ergebnis in 2 Register **ADCL** und **ADCH** abgelegt. Standardmäßig (d.h. **ADLAR** = 0) werden von den 10 Ergebnisbits die niederwertigsten 8 im Register **ADCL** abgelegt und die noch fehlenden 2 Bits im Register **ADCH** an den niederwertigsten Bitpositionen gespeichert.

```
            ADCH                                ADCL
+---+---+---+---+---+---+---+---+   +---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+   +---+---+---+---+---+---+---+---+
                            9   8       7   6   5   4   3   2   1   0
```

Ist keine 10-bit Genauigkeit gefragt, kann diese Zuordnung aber auch geändert werden: Durch Setzen des **ADLAR** Bits im **ADMUX** Register wird die Ausgabe geändert zu:

```
            ADCH                                ADCL
+---+---+---+---+---+---+---+---+   +---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+   +---+---+---+---+---+---+---+---+
  9   8   7   6   5   4   3   2       1   0
```

Auf diese Weise kann das ADC Ergebnis direkt als 8 Bit Zahl weiterverarbeitet werden: Die 8 höchstwertigen Bits stehen bereits verarbeitungsfertig im Register **ADCH** zur Verfügung.

Beim Auslesen der ADC-Register ist zu beachten: Immer zuerst **ADCL** und erst dann **ADCH** auslesen. Beim Zugriff auf **ADCL** wird das **ADCH** Register gegenüber Veränderungen vom **ADC** gesperrt. Erst beim nächsten Auslesen des **ADCH**-Registers wird diese Sperre wieder aufgehoben. Dadurch ist sichergestellt, daß die Inhalte von **ADCL** und **ADCH** immer aus demselben Wandlungsergebnis stammen, selbst wenn der **ADC** im Hintergrund selbsttätig weiterwandelt. Das **ADCH** Register **muss** ausgelesen werden!

Beispiele

Ausgabe als ADC-Wert

Das folgende Programm liest in einer Schleife ständig den ADC aus und verschickt das Ergebnis im Klartext (ASCII) über die UART. Zur Verringerung des unvermeidlichen Rauschens werden 256 Messwerte herangezogen und deren Mittelwert als endgültiges Messergebnis gewertet. Dazu werden die einzelnen Messungen in den Registern temp2, temp3, temp4 als 24 Bit Zahl aufaddiert. Die Division durch 256 erfolgt dann ganz einfach dadurch, dass das Register temp2 verworfen wird und die Register temp3 und temp4 als 16 Bit Zahl aufgefasst werden. Eine Besonderheit ist noch, dass je nach dem Wert in temp2 die 16 Bit Zahl in temp3 und temp4 noch aufgerundet wird: Enthält temp2 einen Wert größer als 128, dann wird zur 16 Bit Zahl in temp3/temp4 noch 1 dazu addiert.

In diesem Programm findet man oft die Konstruktion

```
    subi    temp3, low(-1)      ; addieren von 1
    sbci    temp4, high(-1)     ; addieren des Carry
```

Dabei handelt es sich um einen kleinen Trick. Um eine Konstante zu einem Register direkt addieren zu können bräuchte man einen Befehl ala addi (Add Immediate, Addiere Konstante), den der AVR aber nicht hat. Ebenso gibt es kein adci (Add with carry Immediate, Addiere Konstante mit Carry Flag). Man müsste also erst eine Konstante in ein Register laden und addieren. Das kostet aber Programmspeicher, Rechenzeit und man muss ein Register zusätzlich frei haben.

```
; 16 Bit Addition mit Konstante, ohne Cleverness
    ldi     temp5, low(1)
    add     temp3, temp5        ; addieren von 1
    ldi     temp5, high(1)
    adc     temp3, temp5        ; addieren des Carry
```

Hier greift man einfach zu dem Trick, dass eine Addition gleich der Subtraktion der negativen Werts ist. Also "addiere +1" ist gleich "subtrahiere -1". Dafür hat der AVR zwei Befehle, subi (Substract Immediate, Subtrahiere Konstante) und sbci (Substract Immediate with carry, Subtrahiere Konstante mit Carry Flag).

Das folgende Programm ist für den **ATmega8** geschrieben. Für moderne Nachfolgetypen wie den ATmega88 muss der Code angepasst werden ([2], AVR094: Replacing ATmega8 by ATmega88 (PDF)).

```
.include "m8def.inc"

.def temp1     = r16         ; allgemeines temp Register, zur kurzfristigen
Verwendung
.def temp2     = r17         ; Register für 24 Bit Addition, Lowest Byte
.def temp3     = r18         ; Register für 24 Bit Addition, Middle Byte
.def temp4     = r19         ; Register für 24 Bit Addition, Highest Byte
.def adlow     = r20         ; Ergebnis vom ADC / Mittelwert der 256 Messungen
.def adhigh    = r21         ; Ergebnis vom ADC / Mittelwert der 256 Messungen
.def messungen = r22         ; Schleifenzähler für die Messungen
.def ztausend  = r23         ; Zehntausenderstelle des ADC Wertes
.def tausend   = r24         ; Tausenderstelle des ADC Wertes
.def hundert   = r25         ; Hunderterstelle des ADC Wertes
```

```
.def zehner    = r26         ; Zehnerstelle des ADC Wertes
.def zeichen   = r27         ; Zeichen zur Ausgabe auf den UART

.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

; hier geht das Programm los

    ldi     temp1, LOW(RAMEND)                  ; Stackpointer initialisieren
    out     SPL, temp1
    ldi     temp1, HIGH(RAMEND)
    out     SPH, temp1

;UART Initalisierung

    ldi     temp1, LOW(UBRR_VAL)                    ; Baudrate einstellen
    out     UBRRL, temp1
    ldi     temp1, HIGH(UBRR_VAL)
    out     UBRRH, temp1

    sbi     UCSRB, TXEN                         ; TX einschalten

; ADC initialisieren: ADC0, Vcc als Referenz, Single Conversion, Vorteiler 128

    ldi     temp1, (1<<REFS0)                   ; Kanal 0, interne
Referenzspannung 5V
    out     ADMUX, temp1
    ldi     temp1, (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out     ADCSRA, temp1

Main:
    clr     temp1
    clr     temp2
    clr     temp3
    clr     temp4

    ldi     messungen, 0        ; 256 Schleifendurchläufe

; neuen ADC-Wert lesen  (Schleife - 256 mal)

sample_adc:
    sbi     ADCSRA, ADSC        ; den ADC starten

wait_adc:
    sbic    ADCSRA, ADSC        ; wenn der ADC fertig ist, wird dieses Bit
gelöscht
    rjmp    wait_adc

; ADC einlesen:

    in      adlow, ADCL         ; immer zuerst LOW Byte lesen
    in      adhigh, ADCH        ; danach das mittlerweile gesperrte High Byte

; alle 256 ADC-Werte addieren
```

```
; dazu wird mit den Registern temp4, temp3 und temp2 ein
; 24-Bit breites Akkumulationsregister gebildet, in dem
; die 10 Bit Werte aus adhigh, adlow aufsummiert werden

    add     temp2, adlow        ; addieren
    adc     temp3, adhigh       ; addieren über Carry
    adc     temp4, temp1        ; addieren über Carry, temp1 enthält 0
    dec     messungen           ; Schleifenzähler MINUS 1
    brne    sample_adc          ; wenn noch keine 256 ADC Werte -> nächsten Wert
einlesen

; Aus den 256 Werten den Mittelwert berechnen
; Mathematisch eine Division durch 256
; Da aber 2^8 = 256 ist ist da einfach durch das weglassen des niederwertigsten
Bytes
; erreicht werden
;
; allerdings wird der Wert noch gerundet

    cpi     temp2,128           ; "Kommastelle" kleiner als 128 ?
    brlo    no_round            ; ist kleiner ==> Sprung

; Aufrunden
    subi    temp3, low(-1)      ; addieren von 1
    sbci    temp4, high(-1)     ; addieren des Carry

no_round:

;   Ergebnis nach adlow und adhigh kopieren
;   damit die temp Register frei werden

    mov     adlow, temp3
    mov     adhigh, temp4

;in ASCII umwandeln
; Division durch mehrfache Subtraktion

    ldi     ztausend, '0'-1     ; Ziffernzähler direkt als ASCII Code
    ; bzgl. '0'-1 siehe http://www.mikrocontroller.net/topic/198681
Z_ztausend:
    inc     ztausend
    subi    adlow, low(10000)   ; -10,000
    sbci    adhigh, high(10000) ; 16 Bit
    brcc    Z_ztausend

    subi    adlow, low(-10000)  ; nach Unterlauf wieder einmal addieren
    sbci    adhigh, high(-10000); +10,000

    ldi     tausend, '0'-1      ; Ziffernzähler direkt als ASCII Code
Z_tausend:
    inc     tausend
    subi    adlow, low(1000)    ; -1,000
    sbci    adhigh, high(1000)  ; 16 Bit
    brcc    Z_tausend

    subi    adlow, low(-1000)   ; nach Unterlauf wieder einmal addieren
    sbci    adhigh, high(-1000) ; +1,000

    ldi     hundert, '0'-1      ; Ziffernzähler direkt als ASCII Code
Z_hundert:
    inc     hundert
    subi    adlow, low(100)     ; -100
    sbci    adhigh, high(100)   ; 16 Bit
    brcc    Z_hundert
```

```
    subi    adlow, low(-100)    ; nach Unterlauf wieder einmal addieren
    sbci    adhigh, high(-100)  ; +100

    ldi     zehner, '0'-1       ; Ziffernzähler direkt als ASCII Code
Z_zehner:
    inc     zehner
    subi    adlow, low(10)      ; -10
    sbci    adhigh, high(10)    ; 16 Bit
    brcc    Z_zehner

    subi    adlow, low(-10)     ; nach Unterlauf wieder einmal addieren
    sbci    adhigh, high(-10)   ; +10

    subi    adlow, -'0'         ; adlow enthält die Einer, Umwandlung in ASCII

;an UART Senden

    mov     zeichen, ztausend   ; Zehntausender Stelle
    rcall   transmit
    mov     zeichen, tausend    ; Tausender Stelle ausgeben
    rcall   transmit
    mov     zeichen, hundert    ; Hunderter Stelle ausgeben
    rcall   transmit
    mov     zeichen, zehner     ; Zehner Stelle ausgeben
    rcall   transmit
    mov     zeichen, adlow      ; Einer Stelle ausgeben
    rcall   transmit
    ldi     zeichen, 13         ; CR, Carrige Return (Wagenrücklauf)
    rcall   transmit
    ldi     zeichen, 10         ; LF, Line Feed (Neue Zeile)
    rcall   transmit

    rjmp    Main

transmit:
    sbis    UCSRA,UDRE          ; Warten, bis UDR bereit ist ...
    rjmp    transmit
    out     UDR, zeichen        ; und Zeichen ausgeben
    ret
```

Ausgabe als Spannungswert

Das zweite Beispiel ist schon um einiges größer. Hier wird der gemittelte ADC-Wert in eine Spannung umgerechnet. Dazu wird Festkommaarithmetik verwendet. Die Daten sind in diesem Fall

- Referenzspannung : 5V
- alte Auflösung : 5V / 1024 = 4,8828125mV
- neue Auflösung : 1mV

-> Faktor = 4,8828125mV / 1mV = 4,8828125

Der Faktor wird dreimal mit 10 multipliziert und das Ergebnis auf 4883 gerundet. Die neue Auflösung wird dreimal durch 10 dividiert und beträgt 1µV. Der relative Fehler beträgt

$$F_r = \frac{4883}{4882,8125} - 1 = 0,00384\% = \frac{1}{26042}$$

Dieser Fehler ist absolut vernachlässigbar. Nach der Multiplikation des ADC-Wertes mit 4883 liegt

die gemessene Spannung in der Einheit µV vor. Vorsicht! Das ist **nicht** die reale Auflösung und Genauigkeit, nur rein mathematisch bedingt. Für maximale Genauigkeit sollte man die Versorgungsspannung AVCC, welche hier gleichzeitig als Referenzspannung dient, exakt messen, die Rechnung nachvollziehen und den Wert im Quelltext eintragen. Damit führt man eine einfach Einpunktkalibrierung durch.

Da das Programm schon um einiges größer und komplexer ist, wurde es im Vergleich zur Vorgängerversion geändert. Die Multiplikation sowie die Umwandung der Zahl in einen ASCII-String sind als Unterprogramme geschrieben, dadurch erhält man wesentlich mehr Überblick im Hauptprogramm und die Wiederverwendung in anderen Programmen vereinfacht sich. Ausserdem wird der String im RAM gespeichert und nicht mehr in CPU-Registern. Die Berechung der einzelnen Ziffern erfolgt über ein Schleife, das ist kompakter und übersichtlicher.

```
.include "m8def.inc"

.def z0        = r1          ; Zahl für Integer -> ASCII Umwandlung
.def z1        = r2
.def z2        = r3
.def z3        = r4
.def temp1     = r16         ; allgemeines Register, zur kurzfristigen
Verwendung
.def temp2     = r17         ; Register für 24 Bit Addition, niederwertigstes
Byte (LSB)
.def temp3     = r18         ; Register für 24 Bit Addition, mittlerers Byte
.def temp4     = r19         ; Register für 24 Bit Addition, höchstwertigstes
Byte (MSB)
.def adlow     = r20         ; Ergebnis vom ADC-Mittelwert der 256 Messungen
.def adhigh    = r21         ; Ergebnis vom ADC-Mittelwert der 256 Messungen
.def messungen = r22         ; Schleifenzähler für die Messungen
.def zeichen   = r23         ; Zeichen zur Ausgabe auf den UART
.def temp5     = r24
.def temp6     = r25

; Faktor für Umrechung des ADC-Wertes in Spannung
; = (Referenzspannung / 1024 ) * 100000
; = 5V / 1024 * 1.000.000
.equ Faktor = 4883

.equ F_CPU = 4000000                            ; Systemtakt in Hz
.equ BAUD  = 9600                               ; Baudrate

; Berechnungen
.equ UBRR_VAL   = ((F_CPU+BAUD*8)/(BAUD*16)-1)  ; clever runden
.equ BAUD_REAL  = (F_CPU/(16*(UBRR_VAL+1)))     ; Reale Baudrate
.equ BAUD_ERROR = ((BAUD_REAL*1000)/BAUD-1000)  ; Fehler in Promille

.if ((BAUD_ERROR>10) || (BAUD_ERROR<-10))       ; max. +/-10 Promille Fehler
  .error "Systematischer Fehler der Baudrate grösser 1 Prozent und damit zu
hoch!"
.endif

; RAM
.dseg
.org 0x60
Puffer: .byte 10

; hier geht das Programm los
.cseg
```

```
.org 0

    ldi     temp1, LOW(RAMEND)                  ; Stackpointer initialisieren
    out     SPL, temp1
    ldi     temp1, HIGH(RAMEND)
    out     SPH, temp1

;UART Initalisierung

    ldi     temp1, LOW(UBRR_VAL)                ; Baudrate einstellen
    out     UBRRL, temp1
    ldi     temp1, HIGH(UBRR_VAL)
    out     UBRRH, temp1

    sbi     UCSRB, TXEN                         ; TX einschalten

; ADC initialisieren: Single Conversion, Vorteiler 128
; Kanal 0, interne Referenzspannung AVCC

    ldi     temp1, (1<<REFS0)
    out     ADMUX, temp1
    ldi     temp1, (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)
    out     ADCSRA, temp1

Hauptschleife:
    clr     temp1
    clr     temp2
    clr     temp3
    clr     temp4

    ldi     messungen, 0        ; 256 Schleifendurchläufe

; neuen ADC-Wert lesen  (Schleife - 256 mal)

adc_messung:
    sbi     ADCSRA, ADSC        ; den ADC starten

adc_warten:
    sbic    ADCSRA, ADSC        ; wenn der ADC fertig ist, wird dieses Bit
gelöscht
    rjmp    adc_warten

; ADC einlesen:

    in      adlow, ADCL         ; immer zuerst LOW Byte lesen
    in      adhigh, ADCH        ; danach das mittlerweile gesperrte High Byte

; alle 256 ADC-Werte addieren
; dazu wird mit den Registern temp4, temp3 und temp2 ein
; 24-Bit breites Akkumulationsregister gebildet, in dem
; die 10 Bit Werte aus adhigh, adlow aufsummiert werden

    add     temp2, adlow        ; addieren
    adc     temp3, adhigh       ; addieren über Carry
    adc     temp4, temp1        ; addieren über Carry, temp1 enthält 0
    dec     messungen           ; Schleifenzähler MINUS 1
    brne    adc_messung         ; wenn noch keine 256 ADC Werte -> nächsten Wert
einlesen

; Aus den 256 Werten den Mittelwert berechnen
; Bei 256 Werten ist das ganz einfach: Das niederwertigste Byte
; (im Register temp2) fällt einfach weg
;
; allerdings wird der Wert noch gerundet
```

```
    cpi     temp2,128           ; "Kommastelle" kleiner als 128 ?
    brlo    nicht_runden        ; ist kleiner ==> Sprung

; Aufrunden
    subi    temp3, low(-1)      ; addieren von 1
    sbci    temp4, high(-1)     ; addieren des Carry

nicht_runden:

;   Ergebnis nach adlow und adhigh kopieren
;   damit die temp Register frei werden

    mov     adlow, temp3
    mov     adhigh, temp4

; in Spannung umrechnen

    ldi     temp5,low(Faktor)
    ldi     temp6,high(Faktor)
    rcall   mul_16x16

; in ASCII umwandeln

    ldi     XL, low(Puffer)
    ldi     XH, high(Puffer)
    rcall   Int_to_ASCII

;an UART Senden

    ldi     ZL, low(Puffer+3)
    ldi     ZH, high(Puffer+3)
    ldi     temp1, 1
    rcall   sende_zeichen       ; eine Vorkommastelle ausgeben

    ldi     zeichen, ','        ; Komma ausgeben
    rcall   sende_einzelzeichen

    ldi     temp1, 3            ; Drei Nachkommastellen ausgeben
    rcall   sende_zeichen

    ldi     zeichen, 'V'        ; Volt Zeichen ausgeben
    rcall   sende_einzelzeichen

    ldi     zeichen, 10         ; New Line Steuerzeichen
    rcall   sende_einzelzeichen

    ldi     zeichen, 13         ; Carrige Return Steuerzeichen
    rcall   sende_einzelzeichen

    rjmp    Hauptschleife

; Ende des Hauptprogramms

; Unterprogramme

 ; ein Zeichen per UART senden

sende_einzelzeichen:
    sbis    UCSRA,UDRE          ; Warten, bis UDR bereit ist ...
    rjmp    sende_einzelzeichen
    out     UDR, zeichen        ; und Zeichen ausgeben
    ret
```

```
; mehrere Zeichen ausgeben, welche durch Z adressiert werden
; Anzahl in temp1

sende_zeichen:
    sbis    UCSRA,UDRE          ; Warten, bis UDR bereit ist ...
    rjmp    sende_zeichen
    ld      zeichen, Z+         ; Zeichen laden
    out     UDR, zeichen        ; und Zeichen ausgeben
    dec     temp1
    brne    sende_zeichen
    ret

; 32 Bit Zahl in ASCII umwandeln
; Zahl liegt in temp1..4
; Ergebnis ist ein 10stelliger ASCII String, welcher im SRAM abgelegt wird
; Adressierung über X Pointer
; mehrfache Subtraktion wird als Ersatz für eine Division durchgeführt.

Int_to_ASCII:

    push    ZL                      ; Register sichern
    push    ZH
    push    temp5
    push    temp6

    ldi     ZL,low(Tabelle*2)       ; Zeiger auf Tabelle
    ldi     ZH,high(Tabelle*2)
    ldi     temp5, 10               ; Schleifenzähler

Int_to_ASCII_schleife:
    ldi     temp6, -1+'0'           ; Ziffernzähler zählt direkt im ASCII Code
    lpm     z0,Z+                   ; Nächste Zahl laden
    lpm     z1,Z+
    lpm     z2,Z+
    lpm     z3,Z+

Int_to_ASCII_ziffer:
    inc     temp6                   ; Ziffer erhöhen
    sub     temp1, z0               ; Zahl subrahieren
    sbc     temp2, z1               ; 32 Bit
    sbc     temp3, z2
    sbc     temp4, z3
    brge    Int_to_ASCII_ziffer     ; noch kein Unterlauf, nochmal

    add     temp1, z0               ; Unterlauf, eimal wieder addieren
    adc     temp2, z1               ; 32 Bit
    adc     temp3, z2
    adc     temp4, z3
    st      X+,temp6                ; Ziffer speichern
    dec     temp5
    brne    Int_to_ASCII_schleife   ; noch eine Ziffer?

    pop     temp6
    pop     temp5
    pop     ZH
    pop     ZL                      ; Register wieder herstellen
    ret

; Tabelle mit Zahlen für die Berechung der Ziffern
; 1 Milliarde bis 1
Tabelle:
.dd 1000000000, 100000000, 10000000, 1000000, 100000, 10000, 1000, 100, 10, 1

; 16 Bit Wert in Spannung umrechnen
```

```
;
; = 16Bitx16Bit=32 Bit Multiplikation
; = vier 8x8 Bit Multiplikationen
;
; adlow/adhigh * temp5/temp6

mul_16x16:
    push    zeichen
    clr     temp1                   ; 32 Bit Akku löschen
    clr     temp2
    clr     temp3
    clr     temp4
    clr     zeichen                 ; Null, für Carry-Addition

    mul     adlow, temp5            ; erste Multiplikation
    add     temp1, r0               ; und akkumulieren
    adc     temp2, r1

    mul     adhigh, temp5           ; zweite Multiplikation
    add     temp2, r0               ; und gewichtet akkumlieren
    adc     temp3, r1

    mul     adlow, temp6            ; dritte Multiplikation
    add     temp2, r0               ; und gewichtet akkumlieren
    adc     temp3, r1
    adc     temp4, zeichen          ; carry addieren

    mul     adhigh, temp6           ; vierte Multiplikation
    add     temp3, r0               ; und gewichtet akkumlieren
    adc     temp4, r1

    pop     zeichen
    ret
```

Für alle, die es besonders eilig haben gibt es hier eine geschwindigkeitsoptimierte Version der Integer in ASCII Umwandlung. Zunächst wird keine Schleife verwendet sondern alle Stufen der Schleife direkt hingeschrieben. Das braucht zwar mehr Programmspeicher, ist aber schneller. Ausserdem wird abwechselnd subtrahiert und addiert, dadurch entfällt das immer wieder notwendige addieren nach dem Unterlauf. Zu guter Letzt werden die Berechnungen nur mit der minimal notwenigen Wortbreite durchgeführt. Am Anfang mit 32 Bit, dann nur noch mit 16 bzw. 8 Bit.

```
; 32 Bit Zahl in ASCII umwandeln
; geschwindigkeitsoptimierte Version
; Zahl liegt in temp1..4
; Ergebnis ist ein 10stelliger ASCII String, welcher im SRAM abgelegt wird
; Adressierung über X Pointer

Int_to_ASCII:
    ldi     temp5, -1 + '0'
_a1ser:
    inc     temp5
    subi    temp1,BYTE1(1000000000) ; - 1.000.000.000
    sbci    temp2,BYTE2(1000000000)
    sbci    temp3,BYTE3(1000000000)
    sbci    temp4,BYTE4(1000000000)
    brcc    _a1ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, 10 + '0'
```

```
_a2ser:
    dec     temp5
    subi    temp1,BYTE1(-100000000) ; + 100.000.000
    sbci    temp2,BYTE2(-100000000)
    sbci    temp3,BYTE3(-100000000)
    sbci    temp4,BYTE4(-100000000)
    brcs    _a2ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, -1 + '0'
_a3ser:
    inc     temp5
    subi    temp1,low(10000000)     ; - 10.000.000
    sbci    temp2,high(10000000)
    sbci    temp3,BYTE3(10000000)
    sbci    temp4,BYTE4(10000000)
    brcc    _a3ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, 10 + '0'
_a4ser:
    dec     temp5
    subi    temp1,low(-1000000)     ; + 1.000.000
    sbci    temp2,high(-1000000)
    sbci    temp3,BYTE3(-1000000)
    sbci    temp4,BYTE4(-1000000)
    brcs    _a4ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, -1 + '0'
_a5ser:
    inc     temp5
    subi    temp1,low(100000)       ; -100.000
    sbci    temp2,high(100000)
    sbci    temp3,BYTE3(100000)
    brcc    _a5ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, 10 + '0'
_a6ser:
    dec     temp5
    subi    temp1,low(-10000)       ; +10,000
    sbci    temp2,high(-10000)
    sbci    temp3,BYTE3(-10000)
    brcs    _a6ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, -1 + '0'
_a7ser:
    inc     temp5
    subi    temp1,low(1000)         ; -1000
    sbci    temp2,high(1000)
    brcc    _a7ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, 10 + '0'
_a8ser:
    dec     temp5
    subi    temp1,low(-100)         ; +100
    sbci    temp2,high(-100)
    brcs    _a8ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, -1 + '0'
```

```
_a9ser:
    inc     temp5
    subi    temp1, 10               ; -10
    brcc    _a9ser

    st      X+,temp5                ; im Puffer speichern
    ldi     temp5, 10 + '0'
_a10ser:
    dec     temp5
    subi    temp1, -1               ; +1
    brcs    _a10ser

    st      X+,temp5                ; im Puffer speichern
    ret
```

AVR-Tutorial: Tasten

Bisher beschränkten sich die meisten Programme auf reine Ausgabe an einem Port. Möchte man Eingaben machen, so ist der Anschluss von Tasten an einen Port unumgänglich. Dabei erheben sich aber 2 Probleme

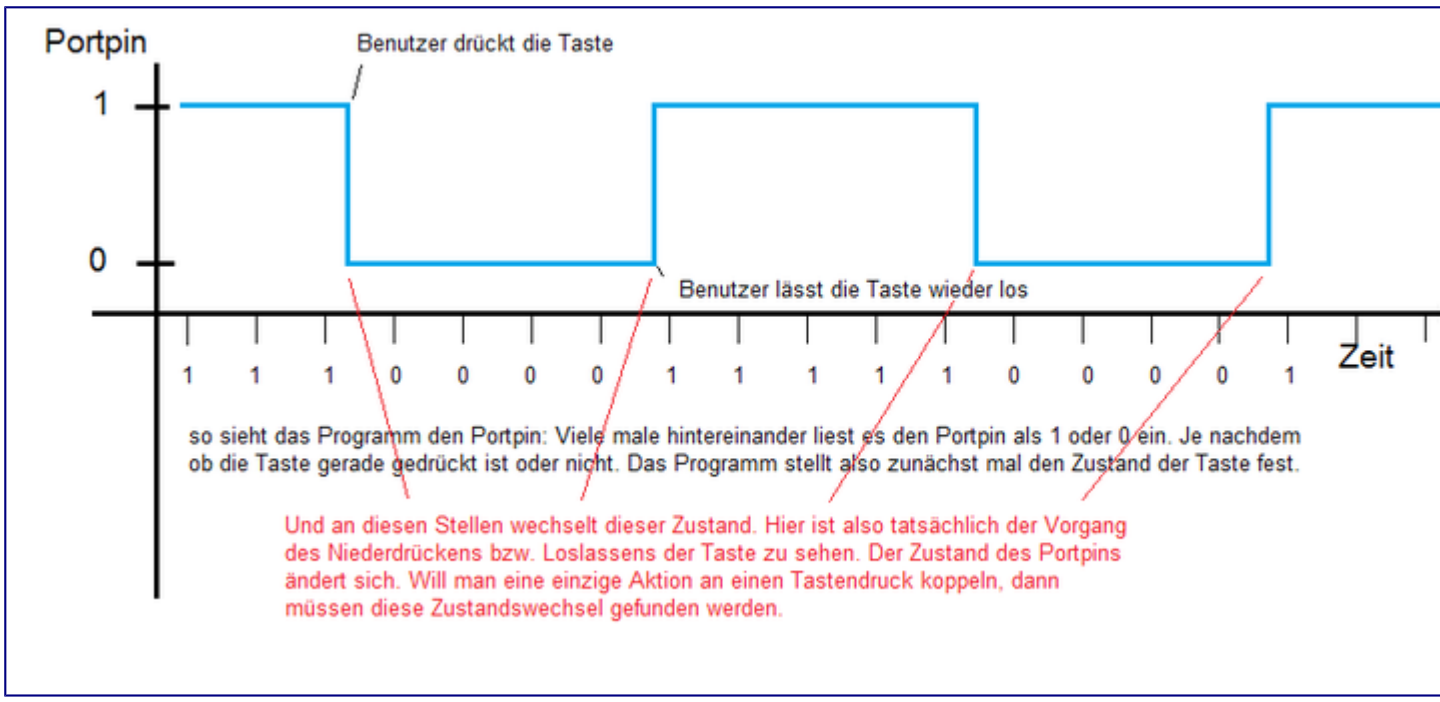
- Wie kann man erreichen, dass ein Tastendruck nur einmal ausgewertet wird?
- Tasten müssen entprellt werden

Inhaltsverzeichnis

- 1 Erkennung von Flanken am Tasteneingang
- 2 Prellen
- 3 Entprellung
- 4 Kombinierte Entprellung und Flankenerkennung
    - 4.1 Einfache Tastenentprellung und Abfrage
    - 4.2 Tastenentprellung, Abfrage und Autorepeat
- 5 Fallbeispiel
- 6 Weblinks

Erkennung von Flanken am Tasteneingang

Möchte man eine Taste auswerten, bei der eine Aktion nicht ausgeführt werden soll, *solange* die Taste gedrückt ist, sondern nur einmal *beim Drücken* einer Taste, dann ist eine Erkennung der Schaltflanke der Weg zum Ziel. Anstatt eine gedrückte Taste zu erkennen, wird bei einer Flankenerkennung der Wechsel des Zustands des Eingangspins detektiert.



Dazu vergleicht man in regelmäßigen Zeitabständen den momentanen Zustand des Eingangs mit dem Zustand zum vorhergehenden Zeitpunkt. Unterscheiden sich die beiden, so hat man eine Schaltflanke erkannt und kann darauf reagieren. Solange sich der Tastenzustand nicht ändert, egal ob die Taste gedrückt oder losgelassen ist, unternimmt man nichts.

Die Erkennung des Zustandswechsels kann am einfachsten mit einer XOR (Exklusiv Oder) Verknüpfung durchgeführt werden.

Wahrheitstabelle XOR

A	B	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	0

Nur dann, wenn sich der Zustand A vom Zustand B unterscheidet, taucht im Ergebnis eine 1 auf. Sind A und B gleich, so ist das Ergebnis 0.

A ist bei uns der vorhergehende Zustand eines Tasters, B ist der jetzige Zustand so wie er vom Port Pin eingelesen wurde. Verknüpft man die beiden mit einem XOR, so bleiben im Ergebnis genau an jenen Bitpositionen 1en übrig, an denen sich der jetzige Zustand vom vorhergehenden unterscheidet.

Nun ist bei Tastern aber nicht nur der erkannte Flankenwechsel interessant, sondern auch in welchen Zustand die Taste gewechselt hat:

- Ist dieser 0, so wurde die Taste gedrückt.
- Ist dieser 1, so wurde die Taste losgelassen.

Eine einfache UND Verknüpfung der Tastenflags mit dem XOR Ergebnis liefert diese Information

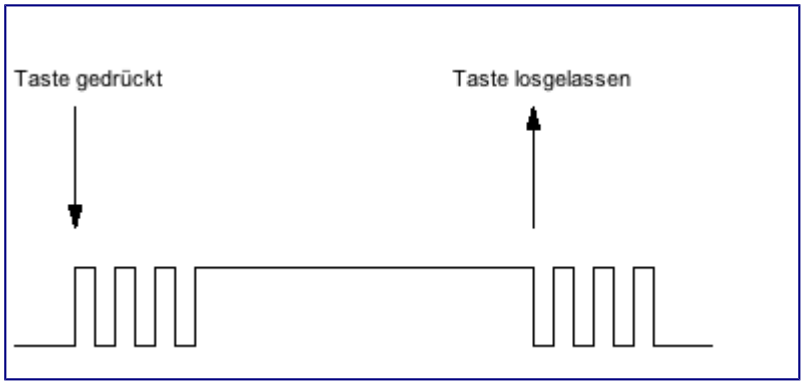
Das folgende Programm soll bei jedem Tastendruck eines Tasters am Port D (egal welcher Pin) eine LED am Port B0 in den jeweils anderen Zustand umschalten:

```
.include "m8def.inc"

.def key_old   = r3
.def key_now   = r4

.def temp1     = r17
.def temp2     = r18

.equ key_pin   = PIND
.equ key_port  = PORTD
.equ key_ddr   = DDRD

.equ led_port  = PORTB
.equ led_ddr   = DDRB
.equ LED       = 0


      ldi  temp1, 1<<LED
      out  led_ddr, temp1         ; den LED Port auf Ausgang

      ldi  temp1, $00             ; den Key Port auf Eingang schalten
      out  key_ddr, temp1
      ldi  temp1, $FF             ; die Pullup Widerstände aktivieren
      out  key_port, temp1

      mov  key_old, temp1         ; bisher war kein Taster gedrückt

loop:
      in   key_now, key_pin       ; den jetzigen Zustand der Taster holen
      mov  temp1, key_now         ; und in temp1 sichern
      eor  key_now, key_old       ; mit dem vorhergehenden Zustand XOR
      mov  key_old, temp1         ; und den jetzigen Zustand für den nächsten
                                  ; Schleifendurchlauf als alten Zustand merken

      breq loop                   ; Das Ergebnis des XOR auswerten:
                                  ; wenn keine Taste gedrückt war -> neuer
Schleifendurchlauf

      and  temp1, key_now         ; War das ein 1->0 Übergang, wurde der Taster
also
                                  ; gedrückt (in key_now steht das Ergebnis vom
XOR)
      brne loop                   ;


      in   temp1, led_port        ; den Zustand der LED umdrehen
      com  temp1
      out  led_port, temp1

      rjmp  loop
```

Probiert man diese Implementierung aus, so stellt man fest: Sie funktioniert nicht besonders gut. Es kann vorkommen, dass bei einem Tastendruck die LED zwar kurzzeitig umschaltet aber gleich darauf wieder ausgeht. Genauso gut kann es passieren, dass die LED beim Loslassen einer Taste ebenfalls wieder den Zustand wechselt. Die Ursache dafür ist: Taster prellen.

Prellen

Das Prellen entsteht in der Mechanik der Tasten: Eine Kontaktfeder wird durch das Drücken des Tastelements auf einen anderen Kontakt gedrückt. Wenn die Kontaktfeder das Kontaktfeld berührt, federt sie jedoch nach. Dies kann soweit gehen, dass die Feder wieder vom Feld abhebt und den elektrischen Kontakt kurzzeitig wieder unterbricht. Auch wenn diese Effekte sehr kurz sind, sind sie für einen Mikrocontroller viel zu lang. Für ihn sieht die Situation so aus, dass beim Drücken der Taste eine Folge von: *Taste geschlossen*, *Taste offen*, *Taste geschlossen*, *Taste offen* Ereignissen am Port sichtbar sind, die sich dann nach einiger Zeit auf den Zustand *Taste geschlossen* einpendelt. Beim Loslassen der Taste dann dasselbe Spielchen in der umgekehrten Richtung.



Signal eines prellenden Tasters

Nun kann es natürlich sein, dass ein neuer Taster zunächst überhaupt nicht prellt. Ist der Taster vom Hersteller nicht explizit als 'prellfreier Taster' verkauft worden, besteht aber kein Grund zur Freude. Auch wenn der Taster heute noch nicht prellt, irgendwann wird er es tun. Dann nämlich, wenn die Kontaktfeder ein wenig ihrer Spannung verliert und ausleiert, bzw. wenn sich die Kontaktflächen durch häufige Benutzung abgewetzt haben.

Entprellung

Aus diesem Grund müssen Tasten entprellt werden. Im Prinzip kann eine Entprellung sehr einfach durchgeführt werden. Ein 'Tastendruck' wird nicht bei der Erkennung der ersten Flanke akzeptiert, sondern es wird noch eine zeitlang gewartet. Ist nach Ablauf dieser Zeitdauer die Taste immer noch gedrückt, dann wird diese Flanke als Tastendruck akzeptiert und ausgewertet.

```
.include "m8def.inc"

.def key_old   = r3
.def key_now   = r4

.def temp1     = r17
.def temp2     = r18

.equ key_pin   = PIND
.equ key_port  = PORTD
.equ key_ddr   = DDRD

.equ led_port  = PORTB
.equ led_ddr   = DDRB
.equ LED       = 0
```

```
      ldi  temp1, 1<<LED
      out  led_ddr, temp1         ; den Led Port auf Ausgang

      ldi  temp1, $00             ; den Key Port auf Eingang schalten
      out  key_ddr, temp1
      ldi  temp1, $FF             ; die Pullup Widerstände aktivieren
      out  key_port, temp1

      mov  key_old, temp1         ; bisher war kein Taster gedrückt

loop:
      in   key_now, key_pin       ; den jetzigen Zustand der Taster holen
      mov  temp1, key_now         ; und in temp1 sichern
      eor  key_now, key_old       ; mit dem vorhergehenden Zustand XOR
      mov  key_old, temp1         ; und den jetzigen Zustand für den nächsten
                                  ; Schleifendurchlauf als alten Zustand merken

      breq loop                   ; Das Ergebnis des XOR auswerten:
                                  ; wenn keine Taste gedrückt war -> neuer
Schleifendurchlauf

      and  temp1, key_now         ; War das ein 1->0 Übergang, wurde der Taster
also
                                  ; gedrückt (in key_now steht das Ergebnis vom
XOR)
      brne loop                   ;

      ldi  temp1, $FF             ; ein bisschen warten ...
wait1:
      ldi  temp2, $FF
wait2:
      dec  temp2
      brne wait2
      dec  temp1
      brne wait1
                                  ; ... und nachsehen, ob die Taste immer noch
gedrückt ist
      in   temp1, key_pin
      and  temp1, key_now
      brne loop

      in   temp1, led_port        ; den Zustand der LED umdrehen
      com  temp1
      out  led_port, temp1


      rjmp  loop
```

Wie lange gewartet werden muss, hängt im wesentlichen von der mechanischen Qualität und dem Zustand des Tasters ab. Neue und qualitativ hochwertige Taster prellen wenig, ältere Taster prellen mehr. Grundsätzlich prellen aber alle mechanischen Taster irgendwann. Man sollte nicht dem Trugschluss verfallen, daß ein Taster nur weil er heute nicht erkennbar prellt, dieses auch in einem halben Jahr nicht tut.

Kombinierte Entprellung und Flankenerkennung

Von Herrn Peter Dannegger stammt eine clevere Routine, die mit wenig Aufwand an einem Port gleichzeitig bis zu 8 Tasten erkennen und zuverlässig entprellen kann. Dazu wird ein Timer benutzt,

der mittels Overflow-Interrupt einen Basistakt erzeugt. Die Zeitdauer von einem Interrupt zum nächsten ist dabei ziemlich unkritisch. Sie sollte sich im Bereich von 5 bis 50 Millisekunden bewegen.

In jedem Overflow Interrupt wird der jetzt am Port anliegende Tastenzustand mit dem Zustand im letzten Timer Interrupt verglichen. Nur dann wenn an einem Pin eine Änderung festgestellt werden kann (Flankenerkennung) wird dieser Tastendruck zunächst registriert. Ein clever aufgebauter Zähler zählt danach die Anzahl der Timer Overflows mit, die die Taste nach Erkennung der Flanke im gedrückten Zustand verharrte. Wurde die Taste nach Erkennung der Flanke 4 mal hintereinander als gedrückt identifiziert, so wird der Tastendruck weitergemeldet. Die 4 mal sind relativ willkürlich und so gewählt, dass man einen Zähler leicht aufbauen kann. Wird die Interrupt Routine also alle 5 Millisekunden aufgerufen, dann muss die Taste bei 4 Stichproben hintereinander durchgehend gedrückt worden sein. Prellt die Taste in dieser Zeit, dann wird der Zähler einfach auf 0 zurückgesetzt und die Wartezeit beginnt erneut zu laufen. Spätestens 20 Millisekunden nach dem letzten Tastenpreller vermeldet daher diese Routine einen Tastendruck, der dann ausgewertet werden kann.

Einfache Tastenentprellung und Abfrage

```
.include "m8def.inc"

.def iwr0      = r1
.def iwr1      = r2

.def key_old   = r3
.def key_state = r4
.def key_press = r5

.def temp1     = r17

.equ key_pin   = PIND
.equ key_port  = PORTD
.equ key_ddr   = DDRD

.def leds      = r16
.equ led_port  = PORTB
.equ led_ddr   = DDRB

.org 0x0000
    rjmp    init

.org OVF0addr
    rjmp    timer_overflow0

timer_overflow0:                ; Timer Overflow Interrupt

    push    r0                  ; temporäre Register sichern
    in      r0, SREG
    push    r0
    push    iwr0
    push    iwr1

get8key:                        ;/old      state     iwr1      iwr0
    mov     iwr0, key_old       ;00110011  10101010            00110011
    in      key_old, key_pin    ;11110000
    eor     iwr0, key_old       ;                                11000011
```

```
    com     key_old             ;00001111
    mov     iwr1, key_state     ;                    10101010
    or      key_state, iwr0     ;          11101011
    and     iwr0, key_old       ;                              00000011
    eor     key_state, iwr0     ;          11101000
    and     iwr1, iwr0          ;                    00000010
    or      key_press, iwr1     ; gedrückte Taste merken
;
;
    pop     iwr1                ; Register wiederherstellen
    pop     iwr0
    pop     r0
    out     SREG, r0
    pop     r0
    reti


init:
    ldi      temp1, HIGH(RAMEND)
    out      SPH, temp1
    ldi      temp1, LOW(RAMEND)     ; Stackpointer initialisieren
    out      SPL, temp1

    ldi      temp1, 0xFF
    out      led_ddr, temp1

    ldi      temp1, 0xFF            ; Tasten sind auf Eingang
    out      key_port, temp1        ; Pullup Widerstände ein

    ldi      temp1, 1<<CS02 | 1<<CS00   ; Timer mit Vorteiler 1024
    out      TCCR0, temp1
    ldi      temp1, 1<<TOIE0            ; Timer Overflow Interrupt einrichten
    out      TIMSK, temp1

    clr      key_old                ; die Register für die Tastenauswertung im
    clr      key_state              ; Timer Interrupt initialisieren
    clr      key_press

    sei                             ; und los gehts: Timer frei

    ldi      leds, 0xFF
    out      led_port, leds
main:
    cli                             ;
    mov      temp1, key_press       ; Einen ev. Tastendruck merken und ...
    clr      key_press              ; Tastendruck zurücksetzen
    sei

    cpi      temp1, 0               ; Tastendruck auswerten. Wenn eine von 8
Tasten
    breq     main                   ; gedrückt worden wäre, wäre ein
entsprechendes
                                    ; Bit in key_press gesetzt gewesen

    eor      leds, temp1            ; Die zur Taste gehörende Led umschalten
    out      led_port, leds
    rjmp     main
```

Tastenentprellung, Abfrage und Autorepeat

Gerade bei Zahlenreihen ist oft eine **Autorepeat** Funktion eine nützliche Einrichtung: Drückt der Benutzer eine Taste wird eine Funktion ausgelöst. Drückt er eine Taste und hält sie gedrückt, so

setzt nach kurzer Zeit der **Autorepeat** ein. Das System verhält sich so, als ob die Taste in schneller Folge immer wieder gedrückt und wieder losgelassen würde.

Leider muss hier für die Wartezeit ein Register im oberen Bereich benutzt werden. Der **ldi** Befehl macht dies notwendig. Alternativ könnte man die Wartezeiten beim Init in eines der unteren Register laden und von dort das *Repeat Timer* Register **key_rep** jeweils nachladen.

Alternativ wurde in diesem Code auch die Rolle des Registers **key_state** umgedreht. Ein gesetztes 1 Bit bedeutet hier, dass die zugehörige Taste zur Zeit gedrückt ist.

Insgesamt ist dieser Code eine direkte Umsetzung des von Herrn Dannegger vorgestellten C-Codes. Durch die Möglichkeit eines Autorepeats bei gedrückter Taste erhöhen sich die Möglichkeiten im Aufbau von Benutzereingaben enorm. Das bischen Mehraufwand im Vergleich zum vorher vorgestellten Code, rechtfertigt dies auf jeden Fall.

```
.include "m8def.inc"

.def iwr0          = r1
.def iwr1          = r2

.def key_state     = r4
.def key_press     = r5
.def key_rep_press = r6
.def key_rep       = r16

.def temp1         = r17

.equ KEY_PIN       = PIND
.equ KEY_PORT      = PORTD
.equ KEY_DDR       = DDRD

.equ KEY_REPEAT_START = 50
.equ KEY_REPEAT_NEXT  = 15

.def leds      = r20
.equ led_port  = PORTB
.equ led_ddr   = DDRB

.equ XTAL  = 4000000

    rjmp    init

.org OVF0addr
    rjmp    timer_overflow0

timer_overflow0:               ; Timer Overflow Interrupt

    push    r0                 ; temporäre Register sichern
    in      r0, SREG
    push    r0

    push    r16
    ; TCNT0 so vorladen, dass der nächste Overflow nach 10 ms auftritt.
    ldi     r16, -( XTAL / 1024 * 10 / 1000)
    ;                ^      ^     ^^^^^^^^^
    ;                |      |      = 10 ms
    ;                |      Vorteiler
    ;                Quarz-Takt
    ;
```

```
    out     TCNT0, r16
    pop     r16

get8key:
    in      r0, KEY_PIN         ; Tasten einlesen
    com     r0                  ; gedrückte Taste werden zu 1
    eor     r0, key_state       ; nur Änderunden berücksichtigen
    and     iwr0, r0            ; in iwr0 und iwr1 zählen
    com     iwr0
    and     iwr1, r0
    eor     iwr1, iwr0
    and     r0, iwr0
    and     r0, iwr1
    eor     key_state, r0       ;
    and     r0, key_state
    or      key_press, r0       ; gedrückte Taste merken
    tst     key_state           ; irgendeine Taste gedrückt ?
    breq    get8key_rep         ; Nein, Zeitdauer zurücksetzen
    dec     key_rep
    brne    get8key_finish;     ; Zeit abgelaufen?
    mov     key_rep_press, key_state
    ldi     key_rep, KEY_REPEAT_NEXT
    rjmp    get8key_finish

get8key_rep:
    ldi     key_rep, KEY_REPEAT_START

get8key_finish:
    pop     r0                  ; Register wiederherstellen
    out     SREG, r0
    pop     r0
    reti
;
;

init:
    ldi      temp1, HIGH(RAMEND)
    out      SPH, temp1
    ldi      temp1, LOW(RAMEND)     ; Stackpointer initialisieren
    out      SPL, temp1

    ldi      temp1, 0xFF
    out      led_ddr, temp1

    ldi      temp1, 0xFF            ; Tasten sind auf Eingang
    out      KEY_PORT, temp1        ; Pullup Widerstände ein

    ldi      temp1, 1<<CS00 | 1<<CS02
    out      TCCR0, temp1
    ldi      temp1, 1<<TOIE0            ; Timer mit Vorteiler 1024
    out      TIMSK, temp1

    clr      key_state
    clr      key_press
    clr      key_rep_press
    clr      key_rep

    ldi      leds, 0xFF
    out      led_port, leds

main:
                                    ; einen einzelnen Tastendruck auswerten
    cli
    mov      temp1, key_press
```

```
    clr      key_press
    sei

    cpi      temp1, 0x01            ; Nur dann wenn Taste 0 gedrückt wurde
    breq     toggle

                                    ; Tasten Autorepeat auswerten
    cli
    mov      temp1, key_rep_press
    clr      key_rep_press
    sei
                                    ; Nur dann wenn Taste 0 gehalten wurde
    cpi      temp1, 0x01
    breq     toggle

    rjmp     main                   ; Hauptschleife abgeschlossen

toggle:
    eor      leds, temp1            ; Die zur Taste gehörende Led umschalten
    out      led_port, leds
    rjmp     main
```

Fallbeispiel

Das folgende Programm hat durchaus praktischen Wert. Es zeigt auf dem LCD den ASCII Code dezimal und in hexadezimal an, sowie das zugehörige LCD-Zeichen. An den **PORTD** werden an den Pins 0 und 1 jeweils 1 Taster angeschlossen. Mit dem einen Taster kann der ASCII Code erhöht werden, mit dem anderen Taster wird der ASCII Code erniedrigt. Auf beiden Tastern liegt jeweils ein Autorepeat, sodass jeder beliebige Code einfach angesteuert werden kann. Insbesondere die ASCII Codes größer als 128 sind interessant :-)

```
.include "m8def.inc"

.def iwr0          = r1
.def iwr1          = r2

.def key_state     = r4
.def key_press     = r5
.def key_rep_press = r6
.def key_rep       = r16

.def temp1         = r17

.equ KEY_PIN       = PIND
.equ KEY_PORT      = PORTD
.equ KEY_DDR       = DDRD

.equ KEY_REPEAT_START = 40
.equ KEY_REPEAT_NEXT  = 15

.def code          = r20

.equ XTAL = 4000000

    rjmp    init

.org OVF0addr
    rjmp    timer_overflow0
```

```
timer_overflow0:                ; Timer Overflow Interrupt

    push    r0                  ; temporäre Register sichern
    in      r0, SREG
    push    r0

    push    r16
    ldi     r16, -( XTAL / 1024 * 10 / 1000 + 1 )
    out     TCNT0, r16
    pop     r16

get8key:
    in      r0, KEY_PIN         ; Tasten einlesen
    com     r0                  ; gedrückte Taste werden zu 1
    eor     r0, key_state       ; nur Änderunden berücksichtigen
    and     iwr0, r0            ; in iwr0 und iwr1 zählen
    com     iwr0
    and     iwr1, r0
    eor     iwr1, iwr0
    and     r0, iwr0
    and     r0, iwr1
    eor     key_state, r0       ;
    and     r0, key_state
    or      key_press, r0       ; gedrückte Taste merken
    tst     key_state           ; irgendeine Taste gedrückt ?
    breq    get8key_rep         ; Nein, Zeitdauer zurücksetzen
    dec     key_rep
    brne    get8key_finish;     ; Zeit abgelaufen?
    mov     key_rep_press, key_state
    ldi     key_rep, KEY_REPEAT_NEXT
    rjmp    get8key_finish

get8key_rep:
    ldi     key_rep, KEY_REPEAT_START

get8key_finish:
    pop     r0                  ; Register wiederherstellen
    out     SREG, r0
    pop     r0
    reti
;
;

init:
    ldi      temp1, HIGH(RAMEND)
    out      SPH, temp1
    ldi      temp1, LOW(RAMEND)     ; Stackpointer initialisieren
    out      SPL, temp1

    ldi      temp1, 0xFF            ; Tasten sind auf Eingang
    out      KEY_PORT, temp1        ; Pullup Widerstände ein

    rcall     lcd_init
    rcall     lcd_clear

    ldi      temp1, 1<<CS00 | 1<<CS02
    out      TCCR0, temp1
    ldi      temp1, 1<<TOIE0            ; Timer mit Vorteiler 1024
    out      TIMSK, temp1

    clr      key_state
    clr      key_press
    clr      key_rep_press
    clr      key_rep
```

```
    ldi     code, 0x30
    rjmp    update

main:
    cli                         ; normaler Tastendruck
    mov     temp1, key_press
    clr     key_press
    sei
    cpi     temp1, 0x01         ; Increment
    breq    increment
    cpi     temp1, 0x02         ; Decrement
    breq    decrement

    cli                         ; gedrückt und halten -> repeat
    mov     temp1, key_rep_press
    clr     key_rep_press
    sei
    cpi     temp1, 0x01         ; Increment
    breq    increment
    cpi     temp1, 0x02         ; Decrement
    breq    decrement

    rjmp    main

increment:
    inc     code
    rjmp    update

decrement:
    dec     code

update:
    rcall   lcd_home
    mov     temp1, code
    rcall   lcd_number
    ldi     temp1, ' '
    rcall   lcd_data
    mov     temp1, code
    rcall   lcd_number_hex
    ldi     temp1, ' '
    rcall   lcd_data
    mov     temp1, code
    rcall   lcd_data

    rjmp    main

.include "lcd-routines.asm"
```

Weblinks

- 10 Keys on One Port Pin? - Eine ganz andere Art der Tastenerkennung über einen ADC. Damit lässt sich auch eine von vielen Tasten an nur einem ADC-Eingangspin abfragen.
- Input Matrix Scanning von Open Music Labs. Tutorials und Kostenbetrachtung für 12 Verfahren, um eine Eingabematrix (1-128 Schalter an 1-20 Pins) abzufragen.

AVR-Tutorial: PWM

PWM - Dieses Kürzel steht für **P**uls **W**eiten **M**odulation.

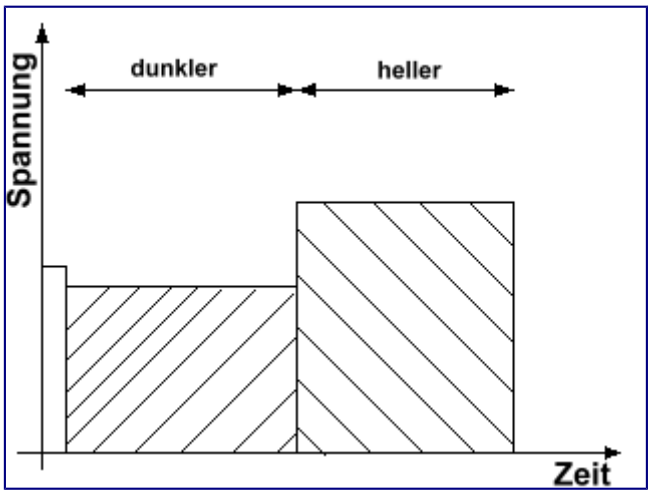
Inhaltsverzeichnis

- 1 Was bedeutet PWM?
- 2 PWM und der Timer
  - 2.1 Fast PWM
- 3 PWM in Software
  - 3.1 Prinzip
  - 3.2 Programm
- 4 Siehe auch

Was bedeutet PWM?

Viele elektrische Verbraucher können in ihrer Leistung reguliert werden, indem die Versorgungsspannung in weiten Bereichen verändert wird. Ein normaler Gleichstrommotor wird z. B. langsamer laufen, wenn er mit einer geringeren Spannung versorgt wird, bzw. schneller laufen, wenn er mit einer höheren Spannung versorgt wird. LEDs werden zwar nicht mit einer Spannung gedimmt, sondern mit dem Versorgungsstrom. Da dieser Stromfluss aber im Normalfall mit einem Vorwiderstand eingestellt wird, ist durch das Ohmsche Gesetz dieser Stromfluss bei konstantem Widerstand wieder direkt proportional zur Höhe der Versorgungsspannung.

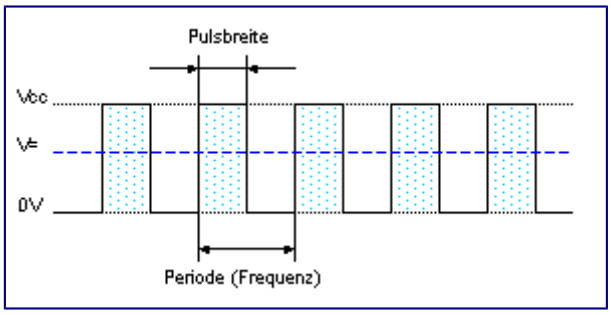
Im wesentlichen geht es also immer um diese Kennlinie, trägt man die Versorgungsspannung entlang der Zeitachse auf:



Die Fläche unter der Kurve ist dabei ein direktes Maß für die Energie die dem System zugeführt

wird. Bei geringerer Energie ist die Helligkeit geringer, bei höherer Energie entsprechend heller.

Jedoch gibt es noch einen zweiten Weg, die dem System zugeführte Energie zu verringern. Anstatt die Spannung abzusenken, ist es auch möglich die volle Versorgungsspannung über einen geringeren Zeitraum anzulegen. Man muß nur dafür Sorge tragen, dass im Endeffekt die einzelnen Pulse nicht mehr wahrnehmbar sind.



Die Fläche unter den Rechtecken hat in diesem Fall dieselbe Größe wie die Fläche unter der Spannung V=, glättet man die Spannung also mit einem Kondensator, ergibt sich eine niedrigere konstante Spannung. Die Rechtecke sind zwar höher, aber dafür schmaler. Die Flächen sind aber dieselben. Diese Lösung hat den Vorteil, dass keine Spannung geregelt werden muss, sondern der Verbraucher immer mit derselben Spannung versorgt wird.

Und genau das ist das Prinzip einer PWM. Durch die Abgabe von Pulsen wird die abgegebene Energiemenge gesteuert. Es ist auf einem µC wesentlich einfacher Pulse mit einem definiertem Puls/Pausen Verhältnis zu erzeugen als eine Spannung zu variieren.

PWM und der Timer

Der Timer1 des Mega8 unterstützt direkt das Erzeugen von PWM. Beginnt der Timer beispielsweise bei 0 zu zählen, so schaltet er gleichzeitig einen Ausgangspin ein. Erreicht der Zähler einen bestimmten Wert X, so schaltet er den Ausgangspin wieder aus und zählt weiter bis zu seiner Obergrenze. Danach wiederholt sich das Spielchen. Der Timer beginnt wieder bei 0 und schaltet gleichzeitig den Ausgangspin ein etc. Durch Verändern von X kann man daher steuern, wie lange der Ausgangspin im Verhältnis zur kompletten Zeit, die der Timer benötigt, um seine Obergrenze zu erreichen, eingeschaltet ist.

Dabei gibt es aber verwirrenderweise verschiedene Arten der PWM:

- Fast PWM
- Phasen-korrekte PWM
- Phasen- und frequenzkorrekte PWM

Für die Details zu jedem PWM-Modus sei auf das Datenblatt verwiesen.

Fast PWM

Die Fast PWM gibt es beim Mega8 mit mehreren unterschiedlichen Bit-Zahlen. Bei den Bit-Zahlen geht es immer darum, wie weit der Timer zählt, bevor ein Rücksetzen des Timers auf 0 erfolgt

- Modus 5: 8 Bit Fast PWM - Der Timer zählt bis 255
- Modus 6: 9 Bit Fast PWM - Der Timer zählt bis 511
- Modus 7: 10 Bit Fast PWM - Der Timer zählt bis 1023
- Modus 14: Fast PWM mit beliebiger Schrittzahl (festgelegt durch **ICR1**)
- Modus 15: Fast PWM mit beliebiger Schrittzahl (festgelegt durch **OCR1A**)

Grundsätzlich funktioniert der Fast-PWM Modus so, dass der Timer bei 0 anfängt zu zählen, wobei natürlich der eingestellte Vorteiler des Timers berücksichtigt wird. Erreicht der Timer einen bestimmten Zählerstand (festgelegt durch die Register **OCR1A** und **OCR1B**) wird eine Aktion ausgelöst. Je nach Festlegung kann der entsprechende µC Pin (OC1A und OC1B) entweder

- umgeschaltet
- auf 1 gesetzt
- auf 0 gesetzt

werden. Wird der OC1A/OC1B Pin so konfiguriert, dass er auf 1 oder 0 gesetzt wird, so wird automatisch der entsprechende Pin beim Timerstand 0 auf den jeweils gegenteiligen Wert gesetzt.

Der OC1A Pin befindet sich beim Mega8 am Port B, konkret am Pin **PB1**. Dieser Pin muss über das zugehörige Datenrichtungsregister **DDRB** auf Ausgang gestellt werden. Anders als beim UART geschieht dies nicht automatisch.

Das Beispiel zeigt den Modus 14. Dabei wird der Timer-Endstand durch das Register **ICR1** festgelegt. Des Weiteren wird die Funktion des OC1A Pins so festgelegt, dass der Pin bei einem Timer Wert von 0 auf 1 gesetzt wird und bei Erreichen des im **OCR1A** Registers festgelegten Wertes auf 0 gesetzt wird. Der Vorteiler des Timers, bzw. der ICR-Wert wird zunächst so eingestellt, dass eine an **PB1** angeschlossene LED noch blinkt, damit die Auswirkungen unterschiedlicher Register Werte gut beobachtet werden können. Den Vorteiler zu verringern ist kein Problem, hier geht es aber darum, zu demonstrieren wie PWM funktioniert.

Hinweis: Wie überall im ATMega8 ist darauf zu achten, dass beim Beschreiben eines 16-Bit Registers zuerst das High-Byte und dann das Low-Byte geschrieben wird.

```
.include "m8def.inc"

.def temp1         = r17

.equ XTAL = 4000000

    rjmp    init

;.include "keys.asm"
;
;

init:
    ldi      temp1, HIGH(RAMEND)     ; Stackpointer initialisieren
    out      SPH, temp1
    ldi      temp1, LOW(RAMEND)
    out      SPL, temp1

    ;
    ; Timer 1 einstellen
```

```
    ;
    ; Modus 14:
    ;    Fast PWM, Top von ICR1
    ;
    ;     WGM13    WGM12   WGM11    WGM10
    ;      1        1       1        0
    ;
    ;    Timer Vorteiler: 256
    ;     CS12     CS11    CS10
    ;      1        0       0
    ;
    ; Steuerung des Ausgangsport: Set at BOTTOM, Clear at match
    ;     COM1A1   COM1A0
    ;      1        0
    ;

    ldi      temp1, 1<<COM1A1 | 1<<WGM11
    out      TCCR1A, temp1

    ldi      temp1, 1<<WGM13 | 1<<WGM12 | 1<<CS12
    out      TCCR1B, temp1

    ;
    ; den Endwert (TOP) für den Zähler setzen
    ; der Zähler zählt bis zu diesem Wert
    ;
    ldi      temp1, 0x6F
    out      ICR1H, temp1
    ldi      temp1, 0xFF
    out      ICR1L, temp1

    ;
    ; der Compare Wert
    ; Wenn der Zähler diesen Wert erreicht, wird mit
    ; obiger Konfiguration der OC1A Ausgang abgeschaltet
    ; Sobald der Zähler wieder bei 0 startet, wird der
    ; Ausgang wieder auf 1 gesetzt
    ;
    ldi      temp1, 0x3F
    out      OCR1AH, temp1
    ldi      temp1, 0xFF
    out      OCR1AL, temp1

    ; Den Pin OC1A zu guter letzt noch auf Ausgang schalten
    ldi      temp1, 0x02
    out      DDRB, temp1

main:
    rjmp     main
```

Wird dieses Programm laufen gelassen, dann ergibt sich eine blinkende LED. Die LED ist die Hälfte der Blinkzeit an und in der anderen Hälfte des Blinkzyklus aus. Wird der Compare Wert in **OCR1A** verändert, so lässt sich das Verhältnis von LED Einzeit zu Auszeit verändern. Ist die LED wie im I/O Kapitel angeschlossen, so führen höhere **OCR1A** Werte dazu, dass die LED nur kurz aufblitzt und in der restlichen Zeit dunkel bleibt.

```
    ldi      temp1, 0x6D
    out      OCR1AH, temp1
    ldi      temp1, 0xFF
    out      OCR1AL, temp1
```

Sinngemäß führen kleinere **OCR1A** Werte dazu, dass die LED länger leuchtet und die Dunkelphasen kürzer werden.

```
ldi      temp1, 0x10
out      OCR1AH, temp1
ldi      temp1, 0xFF
out      OCR1AL, temp1
```

Nachdem die Funktion und das Zusammenspiel der einzelnen Register jetzt klar ist, ist es Zeit aus dem Blinken ein echtes Dimmen zu machen. Dazu genügt es den Vorteiler des Timers auf 1 zu setzen:

```
ldi      temp1, 1<<WGM13 | 1<<WGM12 | 1<<CS10
out      TCCR1B, temp1
```

Werden wieder die beiden **OCR1A** Werte 0x6DFF und 0x10FF ausprobiert, so ist deutlich zu sehen, dass die LED scheinbar unterschiedlich hell leuchtet. Dies ist allerdings eine optische Täuschung. Die LED blinkt nach wie vor, nur blinkt sie so schnell, dass dies für uns nicht mehr wahrnehmbar ist. Durch Variation der Einschalt- zu Ausschaltzeit kann die LED auf viele verschiedene Helligkeitswerte eingestellt werden.

Theoretisch wäre es möglich die LED auf 0x6FFF verschiedene Helligkeitswerte einzustellen. Dies deshalb, weil in **ICR1** genau dieser Wert als Endwert für den Timer festgelegt worden ist. Dieser Wert könnte genauso gut kleiner oder größer eingestellt werden. Um eine LED zu dimmen ist der Maximalwert aber hoffnungslos zu hoch. Für diese Aufgabe reicht eine Abstufung von 256 oder 512 Stufen normalerweise völlig aus. Genau für diese Fälle gibt es die anderen Modi. Anstatt den Timer Endstand mittels **ICR1** festzulegen, genügt es den Timer einfach nur in den 8, 9 oder 10 Bit Modus zu konfigurieren und damit eine PWM mit 256 (8 Bit), 512 (9 Bit) oder 1024 (10 Bit) Stufen zu erzeugen.

PWM in Software

Die Realisierung einer PWM mit einem Timer, wobei der Timer die ganze Arbeit macht, ist zwar einfach, hat aber einen Nachteil. Für jede einzelne PWM ist ein eigener Timer notwendig (Ausnahme: Der Timer 1 besitzt 2 Compare Register und kann damit 2 PWM Stufen erzeugen). Und davon gibt es in einem Mega8 nicht all zu viele.

Es geht auch anders: Es ist durchaus möglich viele PWM Stufen mit nur einem Timer zu realisieren. Der Timer wird nur noch dazu benötigt, eine stabile und konstante Zeitbasis zu erhalten. Von dieser Zeitbasis wird alles weitere abgeleitet.

Prinzip

Das Grundprinzip ist dabei sehr einfach: Eine PWM ist ja im Grunde nichts anderes als eine Blinkschleife, bei der das Verhältnis von Ein- zu Auszeit variabel eingestellt werden kann. Die Blinkfrequenz selbst ist konstant und ist so schnell, dass das eigentliche Blinken nicht mehr wahrgenommen werden kann. Das lässt sich aber auch alles in einer ISR realisieren:

- Ein Timer (Timer0) wird so aufgesetzt, dass er eine Overflow-Interruptfunktion (ISR) mit dem 256-fachen der gewünschten Blinkfrequenz aufruft.
- In der ISR wird ein weiterer Zähler betrieben (*PWMCounter*), der ständig von 0 bis 255 zählt.
- Für jede zu realisierende PWM Stufe gibt es einen Grenzwert. Liegt der Wert des PWMCounters unter diesem Wert, so wird der entsprechende Port Pin eingeschaltet. Liegt er darüber, so wird der entsprechende Port Pin ausgeschaltet

Damit wird im Grunde nichts anderes gemacht, als die Funktionalität der Fast-PWM in Software nachzubilden. Da man dabei aber nicht auf ein einziges OCR Register angewiesen ist, sondern in gewissen Umfang beliebig viele davon implementieren kann, kann man auch beliebig viele PWM Stufen erzeugen.

Programm

Am **Port B** werden an den Pins **PB0** bis **PB5** insgesamt 6 LEDs gemäß der Verschaltung aus dem I/O Artikel angeschlossen. Jede einzelne LED kann durch Setzen eines Wertes von 0 bis 127 in die zugehörigen Register *ocr_1* bis *ocr_6* auf einen anderen Helligkeitswert eingestellt werden. Die PWM-Frequenz (Blinkfrequenz) jeder LED beträgt: (4000000 / 256) / 127 = 123Hz. Dies reicht aus, um das Blinken unter die Wahrnehmungsschwelle zu drücken und die LEDs gleichmässig erleuchtet erscheinen zu lassen.

```
.include "m8def.inc"

.def temp  = r16

.def PWMCount = r17

.def ocr_1 = r18                        ; Helligkeitswert Led1: 0 .. 127
.def ocr_2 = r19                        ; Helligkeitswert Led2: 0 .. 127
.def ocr_3 = r20                        ; Helligkeitswert Led3: 0 .. 127
.def ocr_4 = r21                        ; Helligkeitswert Led4: 0 .. 127
.def ocr_5 = r22                        ; Helligkeitswert Led5: 0 .. 127
.def ocr_6 = r23                        ; Helligkeitswert Led6: 0 .. 127

.org 0x0000
        rjmp    main                    ; Reset Handler
.org OVF0addr
        rjmp    timer0_overflow         ; Timer Overflow Handler

main:
        ldi     temp, LOW(RAMEND)       ; Stackpointer initialisieren
        out     SPL, temp
        ldi     temp, HIGH(RAMEND)
        out     SPH, temp

        ldi     temp, 0xFF              ; Port B auf Ausgang
        out     DDRB, temp

        ldi     ocr_1, 0
        ldi     ocr_2, 1
        ldi     ocr_3, 10
        ldi     ocr_4, 20
        ldi     ocr_5, 80
        ldi     ocr_6, 127
```

```
        ldi     temp, 1<<CS00           ; CS00 setzen: Teiler 1
        out     TCCR0, temp

        ldi     temp, 1<<TOIE0          ; TOIE0: Interrupt bei Timer Overflow
        out     TIMSK, temp

        sei

loop:   rjmp    loop

timer0_overflow:                        ; Timer 0 Overflow Handler
        inc     PWMCount                ; den PWM Zähler von 0 bis
        cpi     PWMCount, 128           ; 127 zählen lassen
        brne    WorkPWM
        clr     PWMCount

WorkPWM:
        ldi     temp, 0b11000000        ; 0 .. Led an, 1 .. Led aus

        cp      PWMCount, ocr_1         ; Ist der Grenzwert für Led 1 erreicht
        brlo    OneOn
        ori     temp, $01

OneOn:  cp      PWMCount, ocr_2         ; Ist der Grenzwert für Led 2 erreicht
        brlo    TwoOn
        ori     temp, $02

TwoOn:  cp      PWMCount, ocr_3         ; Ist der Grenzwert für Led 3 erreicht
        brlo    ThreeOn
        ori     temp, $04

ThreeOn:cp      PWMCount, ocr_4         ; Ist der Grenzwert für Led 4 erreicht
        brlo    FourOn
        ori     temp, $08

FourOn: cp      PWMCount, ocr_5         ; Ist der Grenzwert für Led 5 erreicht
        brlo    FiveOn
        ori     temp, $10

FiveOn: cp      PWMCount, ocr_6         ; Ist der Grenzwert für Led 6 erreicht
        brlo    SetBits
        ori     temp, $20

SetBits:                                ; Die neue Bitbelegung am Port ausgeben
        out     PORTB, temp

        reti
```

Würde man die LEDs anstatt direkt an einen Port anzuschliessen, über ein oder mehrere Schieberegister anschließen, so kann auf diese Art eine relativ große Anzahl an LEDs gedimmt werden. Natürlich müsste man die softwareseitige LED Ansteuerung gegenüber der hier gezeigten verändern, aber das PWM Prinzip könnte so übernommen werden.

Siehe auch

- PWM
- AVR_PWM
- Soft-PWM - optimierte Software-PWM in C

- LED-Fading - LED dimmen mit PWM

AVR-Tutorial: Schieberegister

Ab und an stellt sich folgendes Problem: Man würde wesentlich mehr Ausgangspins oder Eingangspins benötigen als der Mikrocontroller zur Verfügung stellt. Ein möglicher Ausweg ist eine Porterweiterung mit einem Schieberegister. Zwei beliebte Schieberegister sind beispielsweise der 74xx595 bzw. der 74xx165.

Inhaltsverzeichnis

- 1 Porterweiterung für Ausgänge
  - 1.1 Aufbau 74xx595
  - 1.2 Pinbelegung 74xx595
  - 1.3 Funktionsweise
  - 1.4 Ansteuerung per Software
  - 1.5 Ansteuerung per SPI-Modul
  - 1.6 Kaskadieren von Schieberegistern
  - 1.7 Acht LEDs mit je 20mA pro Schieberegister
  - 1.8 Achtung beim Startup!
- 2 Porterweiterung für Eingänge
  - 2.1 Aufbau
  - 2.2 Funktionsweise
  - 2.3 Schaltung
  - 2.4 Ansteuerung per Software
  - 2.5 Ansteuerung per SPI-Modul
- 3 Beides in *einer* Kette
- 4 Bekannte Probleme
- 5 Weblinks

Porterweiterung für Ausgänge

Um neue Ausgangspins zu gewinnen kann der 74xx595 verwendet werden. Dabei handelt es sich um ein *8-Bit 3-state Serial-in/Serial-out or Parallel-Out Schieberegister mit einem Ausgangsregister und einem asynchronen Reset*.

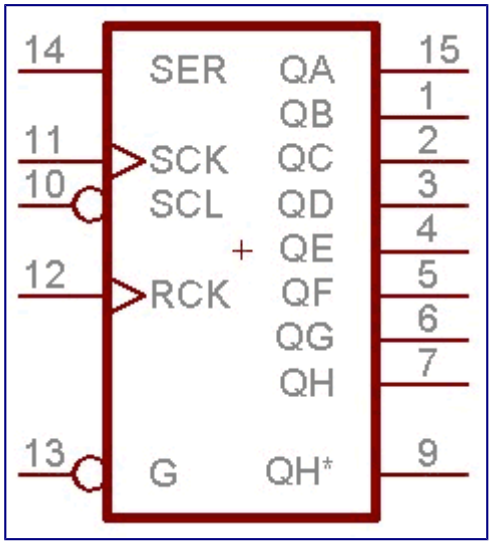
Hinter dieser kompliziert anmutenden Beschreibung verbirgt sich eine einfache Funktionalität: Das Schieberegister besteht aus zwei Funktionseinheiten: Dem eigentlichen Schieberegister und dem Ausgangsregister. In das Schieberegister können die Daten seriell hineingetaktet werden und durch ein bestimmtes Signal werden die Daten des Schieberegisters in das Ausgangsregister übernommen

und können von dort auf die Ausgangspins geschaltet werden.

Im Einzelnen bedeuten die Begriffe:

Begriff	Erklärung
8-Bit	Acht Ausgangsbits
3-state	Die acht Registerausgänge können drei Zustände, Low, High und High-Impedanz annehmen. Siehe Ausgangsstufen Logik-ICs
Serial-in	Serieller Eingang des Schieberegisters
Serial-out	Serieller Ausgang des Schieberegisters
Parallel-Out	Parallele Ausgänge des Ausgangsregisters
Schieberegister	Serielle Daten werden durch den Baustein durchgeschoben
Ausgangsregister	Ein Speicher, welcher die Daten des Schieberegisters zwischenspeichern kann. Dieses besteht aus acht FlipFlops.
Asynchroner Reset	Die Daten im Schieberegister können asynchron zurückgesetzt werden.

Aufbau 74xx595



Pinbelegung eines 595

Hinweis: Die Benennung der Pins in den Datenblättern verschiedener Hersteller unterscheidet sich zum Teil. Die Funktionen der Pins sind jedoch gleich.

Achtung: Es gibt auch noch einen IC von TI mit eingebauten Treibern 50 V 150 mA, den TPIC6B595, der hat 20 Pins und eine abweichende Pinbelegung
http://www.ti.com/product/tpic6b595

HC oder HCT?

Mal gibt es 74**HC**595, mal 74**HCT**595. Diese beiden Typen unterscheiden sich nur dadurch, wie sie Eingangs-Signale erkennen:

- HC: High-Signal muss mindestens ca. 75% der Betriebsspannung haben (siehe V_{IH} im

Datenblatt);
- HC**T**: TTL-kompatibel: High-Signal muss mindestens 2 V haben (Betriebsspannung ist immer 5 V).

Es gibt auch 74LS595. In der modernen CMOS-Welt sollte man sich Low-Power-Schottky (= bipolar = Ruhestrom fressend) nicht mehr antun.

Im allgemeinen kann man alle Typen gleichermaßen verwenden und nimmt einfach den billigsten oder verfügbarsten. Nur beim Übergang zu echtem TTL oder verschiedenen Speisespannungen für Mikrocontroller und Portexpander wird es interessant.

Ein häufiges Szenario ist ein Mikrocontroller mit 3-V-Speisung (etwa ein ARM7 oder MSP430). Dann kann man mit einem 74**HCT**595, an 5 V betrieben, echte 5-V-Ausgänge und die Pegelkonvertierung dazu haben. 74**HC**595 funktionieren hier nur mit Glück, und bei noch geringerer Speisespannung des Controllers — etwa 2,5 V — gar nicht.

Pinbelegung 74xx595

DIL Pin-Nummer	Funktion	Dieses Tutorial	Motorola / ON Semi	Philips / NXP	Fairchild	SGS	Texas Instruments
1	Ausgang B	QB	Q_B	Q_1	Q_B	QB	Q_B
2	Ausgang C	QC	Q_C	Q_2	Q_C	QC	Q_C
3	Ausgang D	QD	Q_D	Q_3	Q_D	QD	Q_D
4	Ausgang E	QE	Q_E	Q_4	Q_E	QE	Q_E
5	Ausgang F	QF	Q_F	Q_5	Q_F	QF	Q_F
6	Ausgang G	QG	Q_G	Q_6	Q_G	QG	Q_G
7	Ausgang H	QH	Q_H	Q_7	Q_H	QH	Q_H
8	Masse, 0 V	[nicht dargestellt]	GND	GND	GND	GND	GND
9	Serieller Ausgang	QH*	SQ_H	Q_7´	Q'_H	QH´	$Q_{H'}$
10	Reset für Schieberegister	SCL	RESET	/MR	/SCLR	/SCLR	/SRCLR
11	Schiebetakt	SCK	SHIFT CLOCK	SH_{CP}	SCK	SCK	SRCLK
12	Speichertakt	RCK	LATCH CLOCK	ST_{CP}	RCK	RCK	RCLK
13	Ausgangssteuerung	G	OUTPUT ENABLE	/OE	/G	/G	/OE
14	Serieller Dateneingang	SER	A	D_S	SER	SI	SER

15	Ausgang A	QA	Q_A	Q_0	Q_A	QA	Q_A
16	Betriebsspannung	[nicht dargestellt]	V_{CC}	V_{CC}	V_{CC}	V_{CC}	V_{CC}

Der Baustein besteht aus zwei Einheiten:

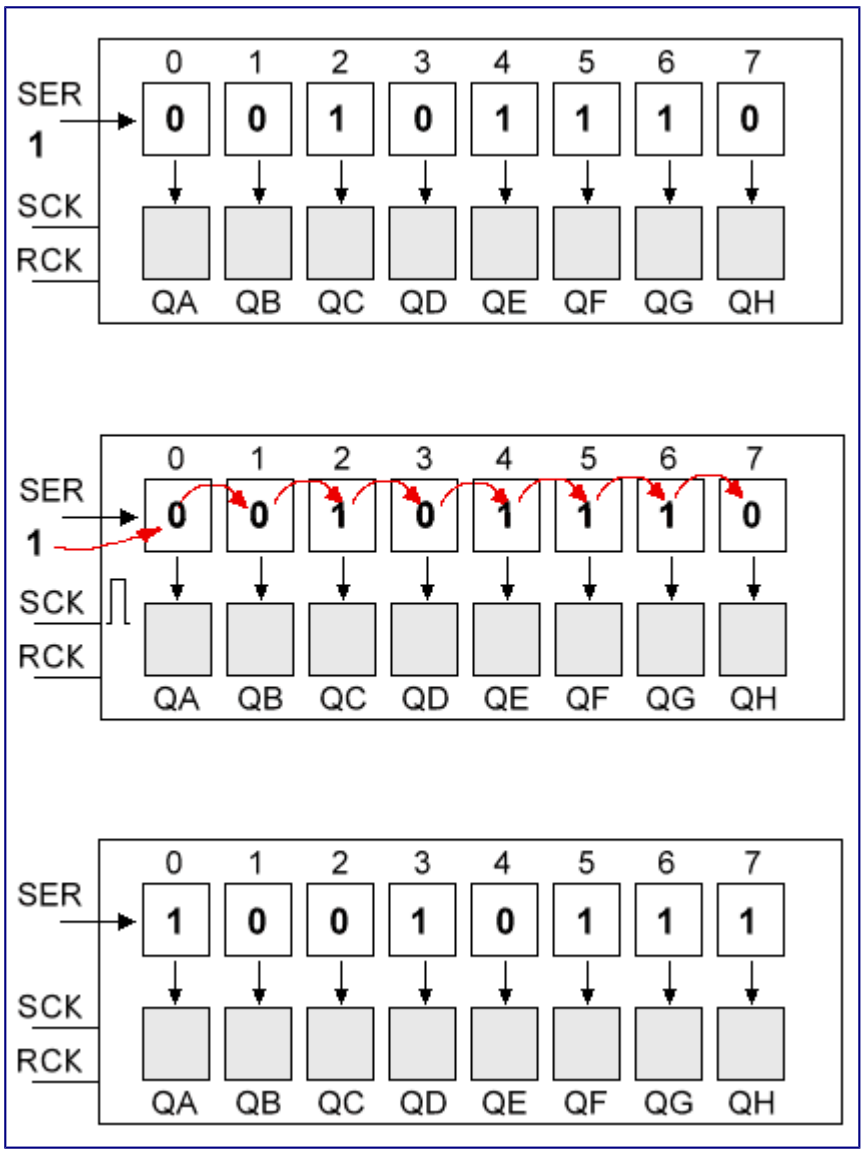
- dem Schieberegister
- dem Ausgangsregister

Im Schieberegister werden die einzelnen Bits durchgeschoben. Mit jeder positiven Taktflanke(LOW -> HIGH) an **SCK** wird eine Schiebeoperation durchgeführt.

Das Ausgangsregister hat die Aufgabe die Ausgangspins des Bausteins anzusteuern. Durch dieses Ausgangsregister ist es möglich, die Schiebeoperationen im Hintergrund durchzuführen, ohne dass IC Pins ihren Wert ändern. Erst wenn die Schiebeoperation abgeschlossen ist, wird der aktuelle Zustand der Schieberegisterkette durch einen Puls an **RCK** in das Ausgangsregister übernommen.

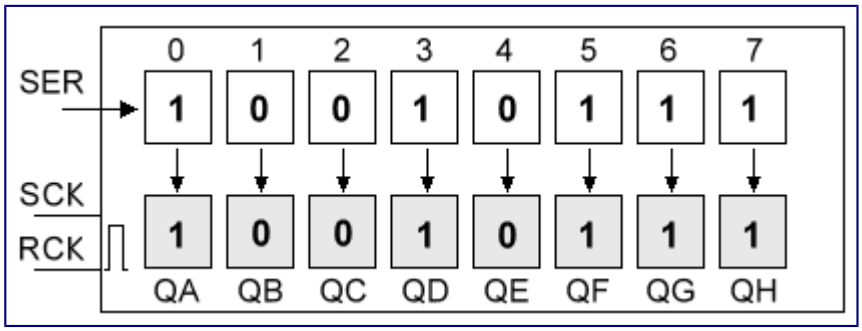
Funktionsweise

Am Eingang **SER** (Pin 14) wird das gewünschte nächste Datum (0 oder 1) angelegt. Durch einen positiven Puls an **SCK** (Pin 11) wird der momentan an **SER** anliegende Wert als neuer Wert für Bit 0, das unterste Bit des Schieberegisters, übernommen. Gleichzeitig werden alle anderen Bits im Schieberegister um eine Stelle verschoben: Das Bit 6 wird ins Bit 7 übernommen, Bit 5 ins Bit 6, Bit 4 ins Bit 5, etc. sodass das Bit 0 zur Aufnahme des **SER** Bits frei wird.



Eine Sonderstellung nimmt das ursprüngliche Bit 7 ein. Dieses Bit steht direkt auch am Ausgang **QH*** (Pin 9) zur Verfügung. Dadurch ist es möglich an ein Schieberegister einen weiteren Baustein 74xxx595 anzuschließen und so beliebig viele Schieberegister hintereinander zu schalten (kaskadieren). Auf diese Art lassen sich Schieberegister mit beliebig vielen Stufen aufbauen.

Wurde das Schieberegister mit den Daten gefüllt, so wird mit einem LOW-HIGH Puls am Pin 12, **RCK** der Inhalt des Schieberegisters in das Ausgangsregister übernommen.



Mit dem Eingang **G** (Pin 13) kann das Ausgangsregister freigegeben werden. Liegt **G** auf 0, so führen die Ausgänge **QA** bis **QH** entsprechende Pegel. Liegt **G** auf 1, so schalten die Ausgänge **QA**

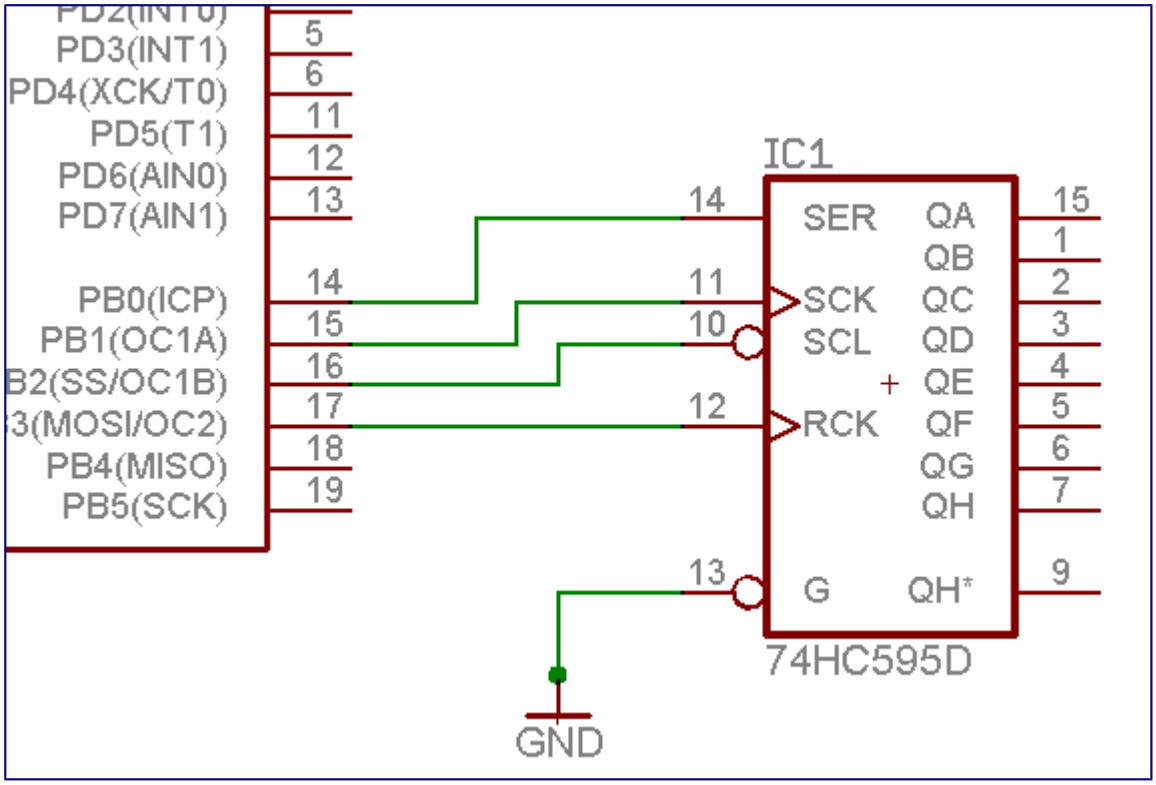
bis **QH** auf Tristate. D.h. sie treiben aktiv weder LOW oder HIGH, sondern sind hochohmig wie ein Eingang und nehmen jeden Pegel an, der ihnen von außen aufgezwungen wird.

Bleibt nur noch der Eingang **SCL**(Pin 10). Mit ihm kann das Schieberegister im Baustein gelöscht, also auf eine definierte 0, gesetzt werden.

Die Ansteuerung eines 74xxx595 Schieberegisters gestaltet sich sehr einfach. Im Grunde gibt es 2 Möglichkeiten:

- Mittels SPI kann der AVR das Schieberegister direkt und autark ansteuern. Das ist sehr schnell und verbraucht nur wenig CPU-Leistung
- Sind die entsprechenden SPI-Pins am AVR nicht frei, so ist auch eine softwaremäßige Ansteuerung des Schieberegisters mit einfachen Mitteln durchführbar.

Ansteuerung per Software



Anschluss eines 595

Für eine komplette Softwarelösung kann das Schieberegister an jede beliebige Port-Pin Kombination angeschlossen werden. Wir wählen die Pins **PB0**, **PB1**, **PB2** und **PB3** um dort die Schieberegisteranschlüsse **SER**, **SCK**, **SCL** und **RCK** anzuschließen. Des Weiteren muss das Schieberegister mit Spannung versorgt werden, auch wenn das rechts im Bild nicht eingezeichnet ist. Pin 16 muss auf +5V und Pin 8 auf GND geschaltet werden.

Die Programmierung gestaltet sich dann nach folgendem Schema: Die 8 Bits eines Bytes werden nacheinander an den Ausgang **PB0** (**SER**) ausgegeben. Durch Generierung eines Pulses 0-1-0 an Pin **PB1** (**SCK**) übernimmt das Schieberegister nacheinander die einzelnen Bits. Dabei ist zu beachten, dass die Ausgabe mit dem höherwertigen Bit beginnen muss, denn dieses Bit wandert ja

am weitesten zur Stelle **QH**. Sind alle 8 Bits ausgegeben, so wird durch einen weiteren 0-1-0 Impuls am Pin **PB3** (**RCK**) der Inhalt der Schieberegisterbits 0 bis 7 in die Ausgaberegister **QA** bis **QH** übernommen. Dadurch, dass am Schieberegister der Eingang **G** konstant auf 0-Pegel gehalten wird, erscheint dann auch die Ausgabe sofort an den entsprechenden Pins und kann zb. mit LEDs (low-current LEDs + Vorwiderstand verwenden) sichtbar gemacht werden.

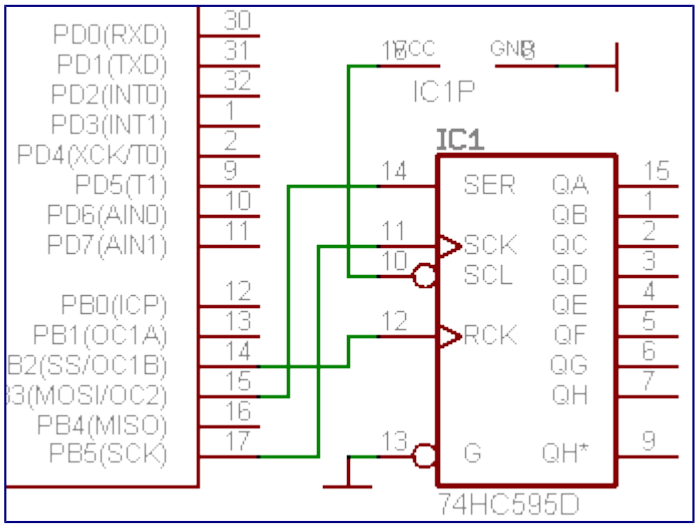
Der Schieberegistereingang **SCL** wird auf einer 1 gehalten. Würde er auf 0 gehen, so würde die Schieberegisterkette gelöscht. Möchte man einen weiteren Prozessorpin einsparen, so kann man diesen Pin auch generell auf Vcc legen. Das Schieberegister könnte man in so einem Fall durch Einschreiben von 0x00 immer noch löschen.

```
.include "m8def.inc"

.def temp1 = r16
.def temp2 = r17

.equ SCHIEBE_DDR  = DDRB
.equ SCHIEBE_PORT = PORTB
.equ RCK          = 3
.equ SCK          = 1
.equ SCL          = 2
.equ SIN          = 0

    ldi   temp1, LOW(RAMEND)     ; Stackpointer initialisieren
    out   SPL, temp1
    ldi   temp1, HIGH(RAMEND)
    out   SPH, temp1

;
; Die Port Pins auf Ausgang konfigurieren
;
    ldi   temp1, (1<<RCK) | (1<<SCK) | (1<<SCL) | (1<<SIN) ; Anm.1
    out   SCHIEBE_DDR, temp1

;
; die Clear Leitung am Schieberegister auf 1 stellen
;
    sbi   SCHIEBE_PORT, SCL

;
; Ein Datenbyte ausgeben
;
    ldi   temp1, 0b10101010
    rcall Schiebe
    rcall SchiebeOut

loop:
    rjmp  loop

;-------------------------------------------------------------------------------
;
; Die Ausgabe im Schieberegister in das Ausgaberegister übernehmen
;
; Dazu am RCK Eingang am Schieberegister einen 0-1-0 Puls erzeugen
;
SchiebeOut:
    sbi   SCHIEBE_PORT, RCK
    cbi   SCHIEBE_PORT, RCK
    ret
```

```
;-----------------------------------------------------------------------------
;
; 8 Bits aus temp1 an das Schieberegister ausgeben
Schiebe:
    push  temp2
    ldi   temp2, 8              ; 8 Bits müssen ausgegeben werden

Schiebe_1:
     ;
     ; jeweils das höchstwertige Bit aus temp1 ins Carry-Flag schieben
     ; Je nach Zustand des Carry-Flags wird die Datenleitung entsprechend
     ; gesetzt oder gelöscht
     ;
    rol  temp1                  ; MSB -> Carry
    brcs Schiebe_One            ; Carry gesetzt? -> weiter bei Schiebe_One
    cbi  SCHIEBE_PORT, SIN      ; Eine 0 ausgeben
    rjmp Schiebe_Clock          ; und Sprung zur Clock Puls Generierung
Schiebe_One:
    sbi  SCHIEBE_PORT, SIN      ; Eine 1 ausgeben

     ;
     ; einen Impuls an SCK zur Übernahme des Bits nachschieben
     ;
Schiebe_Clock:
    sbi   SCHIEBE_PORT, SCK     ; Clock-Ausgang auf 1 ...
    cbi   SCHIEBE_PORT, SCK     ; und wieder zurück auf 0

    dec   temp2                 ; Anzahl der ausgegebenen Bits runterzählen
    brne  Schiebe_1             ; Wenn noch keine 8 Bits ausgegeben -> Schleife
bilden

    pop   temp2
    ret
```

Anm.1: Siehe Bitmanipulation

Ansteuerung per SPI-Modul



Anschluss eines 595 per SPI

Noch schneller geht die Ansteuerung des Schieberegisters mittels SPI-Modul, welches in fast allen AVRs vorhanden ist. Hier wird der Pin **SCL** nicht benutzt, da das praktisch keinen Sinn hat. Er muss also fest auf VCC gelegt werden. (Oder mit den Reset-Pin des AVRs, das mit einer RC

Schaltung versehen ist, verbunden werden. Damit erreicht man einen definierten Anfangszustand des Schieberegisters.) Die Pins für **SCK** und **SIN** sind durch den jeweiligen AVR fest vorgegeben. **SCK** vom 74xxx595 wird mit **SCK** vom AVR verbunden sowie **SIN** mit **MOSI** (**M**aster **O**ut, **S**lave **I**n). **MISO** (**M**aster **I**n, **S**lave **O**ut) ist hier ungenutzt. Es kann NICHT für **RCK** verwendet werden, da es im SPI-Master Modus immer ein Eingang ist! Es kann aber als allgemeiner Eingang oder für 74HC165 (siehe unten) verwendet werden.

Der AVR-Pin **SS** wird sinnvollerweise als **RCK** benutzt, da er sowieso als Ausgang geschaltet werden **muss**, sonst gibt es böse Überaschungen (siehe Datenblatt „SS Pin Functionality“). Dieser sollte mit einem Widerstand von 10 kΩ nach Masse, während der Start- und Initialisierungsphase, auf L-Potential gehalten werden. `(**SS** ist während dieser Zeit noch im Tri-State und es könnte passieren, dass die zufälligen Daten des Schieberegisters in das Ausgangslatch übernommen werden.)

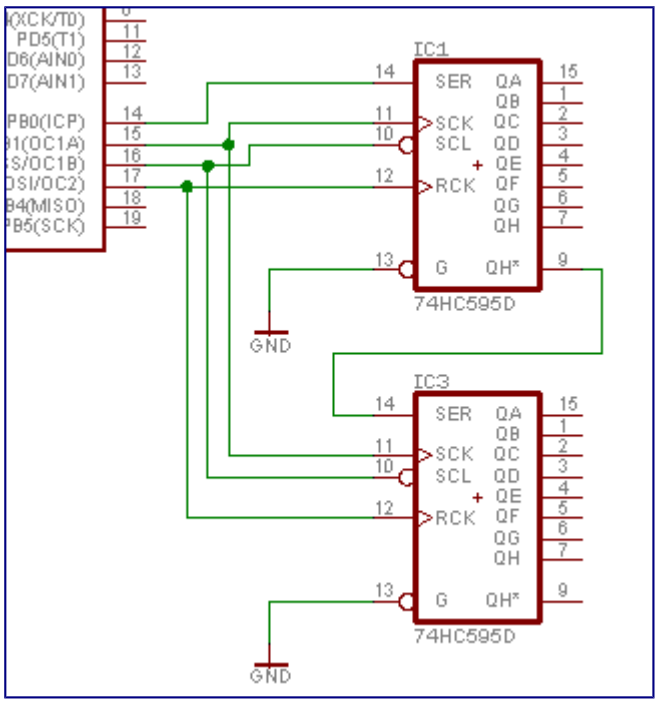
Bei den kleineren ATtinys mit **USI** (Universal Serial Interface) darf man sich von den Pin-Bezeichnungen MOSI und MISO nicht ins Bockshorn jagen lassen: Hier ist MISO der Ausgang(!) **DO** und MOSI der Eingang(!) **DI**. Die Pinbezeichnungen MOSI und MISO sind nur zum Programmieren und irreführend, weil ohnehin nur für den Slave-Betrieb.

Je nach Bedarf kann man die Taktrate des SPI-Moduls zwischen 1/2 ... 1/128 des CPU-Taktes wählen. Es spricht kaum etwas dagegen mit maximaler Geschwindigkeit zu arbeiten. Die AVRs können zur Zeit mit maximal 20 MHz getaktet werden, d.h. es sind maximal 10 MHz SPI-Takt möglich. Das ist für ein 74xxx595 kein Problem. Die Übertragung von 8 Bit dauert dann gerade mal 800 ns!

```
.include "m8def.inc"

.def temp1 = r16

; Die Definitionen müssen an den jeweiligen AVR angepasst werden

.equ SCHIEBE_DDR  = DDRB
.equ SCHIEBE_PORT = PORTB
.equ RCK          = PB2     ; SS
.equ SCK          = PB5     ; SCK
.equ SIN          = PB3     ; MOSI

    ldi   temp1, LOW(RAMEND)     ; Stackpointer initialisieren
    out   SPL, temp1
    ldi   temp1, HIGH(RAMEND)
    out   SPH, temp1
;
; SCK, MOSI, SS als Ausgänge schalten
;
    in    temp1, SCHIEBE_DDR
    ori   temp1, (1<<SIN) | (1<<SCK) | (1<<RCK)
    out   SCHIEBE_DDR,temp1
;
; SPI Modul konfigurieren
;
    ldi   temp1, (1<<SPE) | (1<<MSTR)
    out   SPCR, temp1           ; keine Interrupts, MSB first, Master
                                ; CPOL = 0, CPHA =0
                                ; SCK Takt = 1/2 XTAL
    ldi   temp1, (1<<SPI2X)
```

```
    out   SPSR, temp1           ; double speed aktivieren
    out   SPDR, temp1           ; Dummy Daten, um SPIF zu setzen
;
; Ein Datenbyte ausgeben
;
    ldi   temp1, 0b10101010
    rcall Schiebe               ; Daten schieben
    rcall SchiebeOut            ; Daten in Ausgangsregister übernehmen

loop:
    rjmp  loop

;-------------------------------------------------------------------------------
;
; Die Daten im Schieberegister in das Ausgaberegister übernehmen
;
; Dazu am RCK Eingang am Schieberegister einen 0-1-0 Puls erzeugen
;
SchiebeOut:
    sbis  SPSR, SPIF            ; prüfe ob eine alte Übertragung beendet ist
    rjmp  SchiebeOut
    sbi   SCHIEBE_PORT, RCK
    cbi   SCHIEBE_PORT, RCK
    ret

;-------------------------------------------------------------------------------
;
; 8 Bits aus temp1 an das Schieberegister ausgeben
;
Schiebe:
    sbis    SPSR, SPIF      ; prüfe ob eine alte Übertragung beendet ist
    rjmp    Schiebe
    out     SPDR, temp1     ; Daten ins SPI Modul schreiben, Übertragung beginnt
automatisch
    ret
```

Kaskadieren von Schieberegistern



Kaskadieren mehrerer 595

Um ein Schieberegister anzuschließen genügen also im einfachsten Fall 4 freie Prozessorpins (3 wenn **SCL** nicht benutzt wird) um weitere 8 Ausgangsleitungen zu bekommen. Genügen diese 8 Leitungen nicht, so kann ohne Probleme ein weiteres Schieberegister an das bereits vorhandene angeschlossen werden.

Das nächste Schieberegister wird mit seinem Dateneingang **SER** einfach an den dafür vorgesehenen Ausgang **QH*** des vorhergehenden Schieberegisters angeschlossen. Die Steuerleitungen **SCK**, **RCK** und **SCL** werden parallel zu den bereits vorhandenen geschaltet. Konzeptionell erhält man dadurch ein Schieberegister mit einer Breite von 16 Bit. Werden weiter Bausteine in derselben Manier angeschlossen, so erhöht sich die Anzahl der zur Verfügung stehenden Ausgabeleitungen mit jedem Baustein um 8 ohne dass sich die Anzahl der am Prozessor notwendigen Ausgabepins erhöhen würde. Um diese weiteren Register zu nutzen, muss man in der reinen Softwarelösung nur mehrfach die Funktion **Schiebe** aufrufen, um alle Daten auszugeben. Am Ende werden dann mit **SchiebeOut** die Daten in die Ausgangsregister übernommen.

Bei der SPI Lösung werden ebenfalls ganz einfach mehrere Bytes über SPI ausgegeben, ehe dann mittels **RCK** die in die Schieberegisterkette eingetakteten Bits in das Ausgangsregister übernommen werden. Um das Ganze ein wenig zu vereinfachen, soll hier eine Funktion zur Ansteuerung mehrerer kaskadierter Schieberegister über das SPI-Modul gezeigt werden. Dabei wird die Ausgabe mehrerer Bytes über eine Schleife realisiert, mehrfache Aufrufe der Funktion sind damit nicht nötig. Statt dessen übergibt man einen Zeiger auf einen Datenblock im RAM sowie die Anzahl der zu übertragenden Bytes. Außerdem wird die Datenübernahme durch **RCK** standardkonform integriert. Denn bei nahezu allen ICs mit SPI wird ein sog. CS-Pin verwendet (**C**hip **S**elect) Dieser Pin ist meist LOW aktiv, d.h. wenn er HIGH ist, ignoriert der IC alle Signale an **SCK** und **MOSI** und gibt keine Daten an MISO aus. Ist er LOW, dann ist der IC aktiv und funktioniert normal. Bei der steigenden Flanke an **CS** werden die Daten ins Ausgangsregister übernommen. Die Funktion ist sehr schnell, da die Zeit während der die Übertragung eines Bytes läuft, dazu genutzt wird, den Schleifenzähler zu verringern und zu prüfen sowie neue Sendedaten zu laden. Zwischen den einzelnen Bytes gibt es somit nur eine Pause von max. 6 Systemtakten.

```
.include "m8def.inc"
.def temp1 = r16

; Die Definitionen müssen an den jeweiligen AVR angepasst werden

.equ SCHIEBE_DDR  = DDRB
.equ SCHIEBE_PORT = PORTB
.equ RCK          = PB2     ; SS
.equ SCK          = PB5     ; SCK
.equ SIN          = PB3     ; MOSI

;-------------------------------------------------------------------------------
;
; Datensegment im RAM
;
;-------------------------------------------------------------------------------

.dseg
.org $60
Schiebedaten:       .byte 2
```

```
;-----------------------------------------------------------------------------
;
; Programmsegment im FLASH
;
;-----------------------------------------------------------------------------

.cseg
    ldi     temp1, LOW(RAMEND)     ; Stackpointer initialisieren
    out     SPL, temp1
    ldi     temp1, HIGH(RAMEND)
    out     SPH, temp1
;
; SCK, MOSI, SS als Ausgänge schalten
;
    in      temp1,SCHIEBE_DDR
    ori     temp1,(1<<SIN) | (1<<SCK) | (1<<RCK)
    out     SCHIEBE_DDR,temp1

    sbi     SCHIEBE_PORT, RCK   ; Slave select inaktiv
;
; SPI Modul konfigurieren
;
    ldi     temp1, 0b01010000
    out     SPCR, temp1         ; keine Interrupts, MSB first, Master
                                ; CPOL = 0, CPHA =0
                                ; SCK Takt = 1/2 XTAL
    ldi     r16,1
    out     SPSR,r16            ; Double Speed
    out     SPDR,temp1          ; Dummy Daten, um SPIF zu setzen

; den Datenblock mit Daten füllen

    ldi     temp1,$F0
    sts     Schiebedaten,temp1
    ldi     temp1,$55
    sts     Schiebedaten+1,temp1

loop:
; den Datenblock ausgeben

    ldi     r16,2
    ldi     zl,low(Schiebedaten)
    ldi     zh, high(Schiebedaten)
    rcall   Schiebe_alle                    ; Daten ausgeben

    rjmp  loop                              ; nur zur Simulation

;-----------------------------------------------------------------------------
;
; N Bytes an das Schieberegister ausgeben und in das Ausgaberegister übernehmen
;
; r16: Anzahl der Datenbytes
; Z: Zeiger auf Datenblock im RAM
;
;-----------------------------------------------------------------------------
Schiebe_alle:
    cbi     SCHIEBE_PORT, RCK   ; RCK LOW, SPI Standardverfahren
    push    r17

Schiebe_alle_2:
    ld      r17,Z+
Schiebe_alle_3:
    sbis    SPSR,SPIF           ; prüfe ob eine alte Übertragung beendet ist
    rjmp    Schiebe_alle_3
```

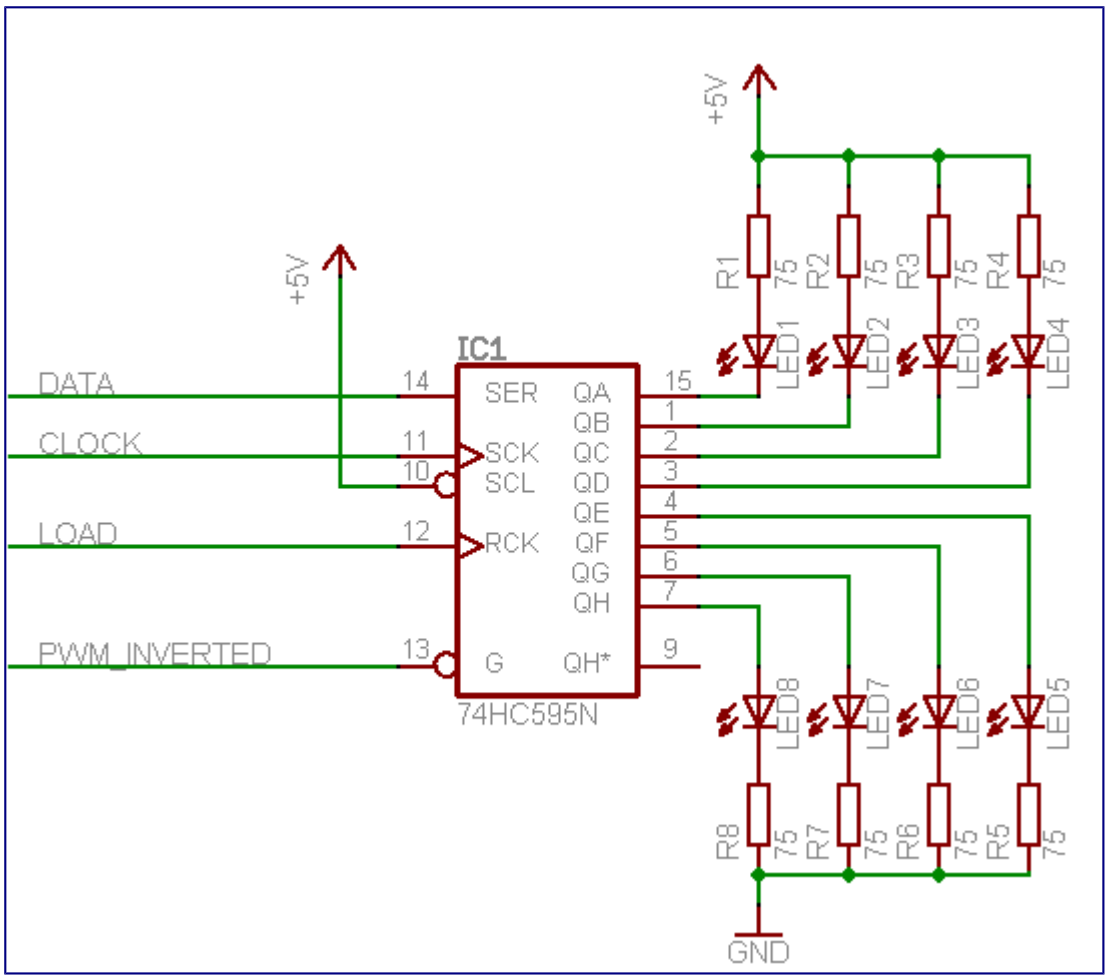
```
    out     SPDR,r17            ; Daten ins SPI Modul schreiben, Übertragung
beginnt automatisch
    dec     r16
    brne    Schiebe_alle_2

Schiebe_alle_4:
    sbis    SPSR,SPIF           ; prüfe ob die letzte Übertragung beendet ist
    rjmp    Schiebe_alle_4

    pop     r17
    sbi     SCHIEBE_PORT, RCK   ; RCK inaktiv, Datenübernahme
    ret
```

Der Nachteil von Schieberegistern ist allerdings, dass sich die Zeit zum Setzen aller Ausgabeleitungen mit jedem weiteren Baustein immer weiter erhöht. Dies deshalb, da ja die einzelnen Bits im Gänsemarsch durch alle Bausteine geschleust werden müssen und für jeden einzelnen Schiebevorgang etwas Zeit notwendig ist. Ein Ausweg ist die Verwendung des SPI-Moduls, welches schneller arbeitet als die reine Softwarelösung. Ist noch mehr Geschwindigkeit gefragt, so sind mehr Port-Pins nötig. Kann ein kompletter Port mit 8 Pins für die Daten genutzt werden, sowie ein paar weitere Steuerleitungen, so können ein oder mehrere 74xxx573 eine Alternative sein, um jeweils ein vollständiges Byte auszugeben. Natürlich kann der 74xxx573 (oder ein ähnliches Schieberegister) auch mit dem 74xxx595 zusammen eingesetzt werden, beispielsweise in dem über das Schieberegister verschiedene 74xxx595 nacheinander aktiviert werden. Weitere Tips und Tricks dazu gibt es vielleicht in einem weiteren Tutorial...

Acht LEDs mit je 20mA pro Schieberegister

Will man nun acht LEDs mit dem Schieberegister ansteuern, kann man diese direkt über Vorwiderstände anschließen. Doch ein genauer Blick ins Datenblatt verrät, dass der 74xx595 nur maximal 70mA über VCC bzw. GND ableiten kann. Und wenn man den IC nicht gnadenlos quälen, und damit die Lebensdauer und Zuverlässigkeit drastisch reduzieren will, gibt es nur zwei Auswege.

- Den Strom pro LED auf 70/8 = 8,75mA begrenzen; Das reicht meistens aus um die LEDs schön leuchten zu lassen, vor allem bei low-current und ultrahellen LEDs
- Wenn doch 20 mA pro LED gebraucht werden, kann man die folgende Trickschaltung anwenden.



Mehrere LED anschließen

Der Trick besteht darin, dass 4 LEDs ihren Strom über das Schieberegister von VCC beziehen (HIGH aktiv) während die anderen vier ihren Strom über GND leiten (LOW aktiv). Damit bleiben ganz offiziell für jede LED 70/4 = 17,5mA. Um die Handhabung in der Software zu vereinfachen muss nur vor der Ausgabe der Daten das jeweilige Byte mit 0x0F XOR verknüpft werden, bevor es in das Schieberegister getaktet wird. Dadurch werden die LOW-aktiven LEDs richtig angesteuert und die Datenhandhabung in der Software muss nur mit HIGH-aktiven rechnen. Außerdem wird der G Eingang verwendet, um die Helligkeit aller LEDs per PWM zu steuern. Beachtet werden muss, dass die PWM im invertierten Modus generiert werden muss, da der Eingang G LOW aktiv ist.

Achtung! Die Widerstände sind auf blaue LEDs mit 3,3V Flussspannung ausgelegt. Bei roten, gelben und grünen LEDs ist die Flussspannung geringer und dementsprechend muss der Vorwiderstand größer sein.

Wenn 20mA immer noch nicht reichen sollten oder z.B. RGB-LEDs mit gemeinsamer Anode angesteuert werden müssen, dann hilft nur ein stärkerer IC. Der Klassiker ist der TPIC6A595 von TI, er kombiniert ein Schieberegister mit MOSFETs, sodass hier 250mA pro Kanal zur Verfügung stehen.

Achtung beim Startup!

Die Ausgänge des '595 führen beim Einschalten undefinierte, also zufällige Pegel! Und der Eingang ¬SCL löscht nur die Schieberegister (= unnütz, um mit ¬RESET zu verbinden). Sind deshalb

definierte Pegel beim Einschalten unerlässlich, muss man ¬OE beschalten:

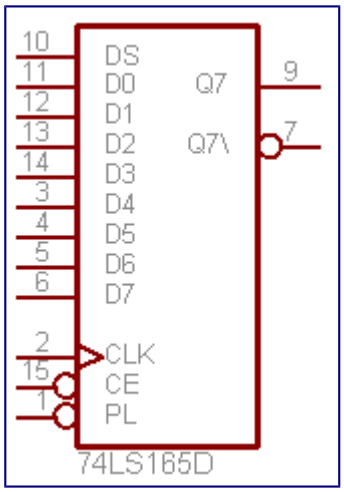
- Entweder mit einem weiteren Mikrocontroller-Ausgang (der mit einem externen Pull-Up-Widerstand zu beschalten ist und nach dem ersten Durchschieben dauerhaft auf LOW gelegt wird)
- Oder mit einer RC(D)-Kombination, die die Ausgänge für eine gewisse Zeit auf Tristate hält. Spart einen Mikrocontroller-Pin.

Obendrein sind, wie sonst am Mikrocontroller, die betreffenden Ausgänge mit externen Pull-Up- oder Pull-Down-Widerständen zu beschalten.

Porterweiterung für Eingänge

Ein naher Verwandter des 74xx595 ist der 74xx165, er ist quasi das Gegenstück. Hierbei handet es sich um ein *8-bit parallel-in/serial-out shift register*. Auf Deutsch: Ein 8 Bit Schieberegister mit parallelem Eingang und seriellem Ausgang. Damit kann man eine große Anzahl Eingänge sehr einfach und preiswert zu seinem Mikrocontroller hinzufügen.

Aufbau



Pinbelegung eines 165

Der Aufbau ist sehr ähnlich zum 74xx595. Allerdings gibt es kein Register zum Zwischenspeichern. Das ist auch gar nicht nötig, da der IC ja einen parallelen Eingang hat. Der muss nicht zwischengespeichert werden. Es gibt hier also wirklich nur das Schieberegister. Dieses wird über den Eingang PL mit den parallelen Daten geladen. Dann können die Daten seriell mit Takten an CLK aus dem Ausgang Q7 geschoben werden.

Funktionsweise

DS ist der serielle Dateneingang, welcher im Falle von kaskadierten Schieberegistern mit dem Ausgang des vorhergehenden ICs verbunden wird.

D0..D7 sind die parallelen Dateneingänge.

Mittels des Eingangs PL (**P**arallel **L**oad) werden die Daten vom parallelen Eingang in das Schieberegister übernommen, wenn dieses Signal LOW ist. Hier muss man aber ein klein wenig

aufpassen. Auf Grund der Schaltungsstruktur ist der Eingang PL mit dem Takt CLK verknüpft (obwohl es dafür keinen logischen Grund gibt :-0). Damit es nicht zu unerwünschten Fehlschaltungen kommt, muss der Takt CLK während des Ladens auf HIGH liegen. Wird PL wieder auf HIGH gesetzt, sind die Daten geladen. Das erste Bit liegt direkt am Ausgang Q7 an. Die restlichen Bits können nach und nach durch das Register geschoben werden.

Der Eingang CE (**C**lock **E**nable) steuert, ob das Schieberegister auf den Takt CLK reagieren soll oder nicht. Ist CE gleich HIGH werden alle Takte an CLK ignoriert. Bei LOW werden mit jeder positiven Flanke die Daten um eine Stufe weiter geschoben.

Wird am Eingang CLK eine LOW-HIGH Flanke angelegt und ist dabei CE auf LOW, dann werden die Daten im Schieberegister um eine Position weiter geschoben: DS->Q0, Q0->Q1, Q1->Q2, Q2->Q3, Q3->Q4, Q4->Q5, Q5->Q6, Q6->Q7. Q0..Q6 sind interne Signale, siehe Datenblatt.

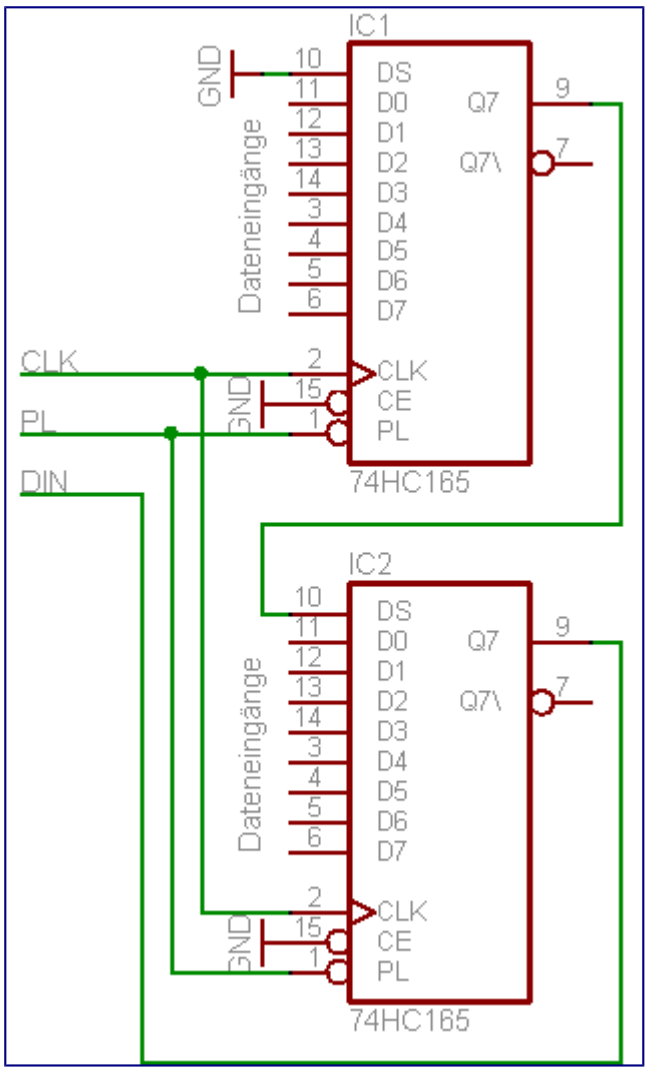
Q7 ist der serielle Ausgang des Schieberegisters. Dort Werden Takt für Takt die Daten ausgegeben. Hier wird normalerweise der Eingang des Mikrocontrollers oder der Eingang des nächsten Schieberegisters angeschlossen.

Q7\ ist der invertierte Ausgang des Schieberegisters. Er wird meist nicht verwendet.

Schaltung

Um nun beispielsweise zwei Schieberegister zu kaskadieren um 16 Eingangspins zu erhalten sollte man folgende Verschaltung vornehmen. Beachten sollte man dabei, dass

- der serielle Eingang DS des ersten Schieberegisters (hier IC1) auf einen festen Pegel gelegt wird (LOW oder HIGH).
- der serielle Datenausgang bei der Benutzung des SPI-Moduls an MISO und nicht an MOSI angeschlossen wird.



Anschluss eines 165

Nachfolgend werden zwei Beispiele gezeigt, welche die Ansteuerung nach bekanntem Muster übernehmen. Nur dass hier eben Daten gelesen anstatt geschrieben werden. Zu beachten ist, dass hier ein anderer Modus der SPI-Ansteuerung verwendet werden muss, weil der Baustein das nötig macht. Das muss beachtet werden, wenn auch Schieberegister für Ausgänge verwendet werden. Dabei muss jeweils vor dem Zugriff auf die Ein- oder Ausgangsregister der Modus des Taktes (CPOL) umgeschaltet werden.

Ansteuerung per Software

```
; Porterweiterung für Eingänge mit Schieberegister 74xx165
; Ansteuerung per Software

.include "m8def.inc"

.def temp1 = r16
.def temp2 = r17
.def temp3 = r18

; Pins anpassen, frei wählbar

.equ SCHIEBE_DDR  = DDRB
.equ SCHIEBE_PORT = PORTB
```

```
.equ SCHIEBE_PIN  = PINB
.equ CLK          = PB3
.equ PL           = PB1
.equ DIN          = PB2

;-------------------------------------------------------------------------------
;
; Datensegment im RAM
;
;-------------------------------------------------------------------------------

.dseg
.org 0x60
Daten:      .byte 2             ; Speicherplatz für Eingangsdaten

;-------------------------------------------------------------------------------
;
; Programmsegment im FLASH
;
;-------------------------------------------------------------------------------
.cseg
    ldi     temp1, LOW(RAMEND)  ; Stackpointer initialisieren
    out     SPL, temp1
    ldi     temp1, HIGH(RAMEND)
    out     SPH, temp1

; CLK und PL als Ausgänge schalten

    ldi     temp1, (1<<clk) | (1<<pl)
    out     SCHIEBE_DDR, temp1

    sbi     schiebe_port, clk   ; Takt im Ruhezustand immer auf 1
                                ; komische Schaltung im 74xx165

; Zwei Bytes einlesen

    ldi     ZL,low(Daten)
    ldi     ZH,high(Daten)
    ldi     temp1,2
    rcall   schiebe_eingang

loop:
    rjmp    loop

;-------------------------------------------------------------------------------
;
; N Bytes seriell einlesen
;
; temp1 : N, Anzahl der Bytes
; Z     : Zeiger auf einen Datenbereich im SRAM
;-------------------------------------------------------------------------------

schiebe_eingang:
    push    temp2               ; Register sichern
    push    temp3

    cbi     schiebe_port, pl    ; Daten parallel laden
    sbi     schiebe_port, pl

schiebe_eingang_byte_schleife:

    ldi     temp3, 8            ; Bitzähler
schiebe_eingang_bit_schleife:
    lsl     temp2               ; Daten weiterschieben
```

```
; das IO Bit Din in das niederwerigste Bit von temp2 kopieren

    sbic    schiebe_pin, din    ; wenn Null, nächsten Befehl überspringen
    ori     temp2,1             ; nein, Bit setzen

    cbi     SCHIEBE_PORT, CLK   ; Taktausgang auf 0
    sbi     SCHIEBE_PORT, CLK   ; und wieder zurück auf 1, dabei Daten schieben

    dec     temp3               ; Bitzähler um eins verringern
    brne    schiebe_eingang_bit_schleife ;wenn noch keine 8 Bits ausgegeben,
nochmal

    st      z+,temp2            ; Datenbyte speichern
    dec     temp1               ; Anzahl Bytes um eins verringern
    brne    schiebe_eingang_byte_schleife   ; wenn noch mehr Bytes zu lesen sind

    pop     temp3
    pop     temp2
    ret
```

Ansteuerung per SPI-Modul

```
; Porterweiterung für Eingänge mit Schieberegister 74xx165
; Ansteuerung per SPI-Modul

.include "m8def.inc"

.def temp1 = r16
.def temp2 = r17
.def temp3 = r18

; Pins anpassen
; diese müssen mit den SPI-Pins des AVR Typs übereinstimmen!

.equ SCHIEBE_DDR  = DDRB
.equ SCHIEBE_PORT = PORTB
.equ PL           = PB2         ; SS
.equ CLK          = PB5         ; SCK
.equ DIN          = PB4         ; MISO

;-------------------------------------------------------------------------------
;
; Datensegment im RAM
;
;-------------------------------------------------------------------------------

.dseg
.org 0x60
Daten:      .byte 2             ; Speicherplatz für Eingangsdaten

;-------------------------------------------------------------------------------
;
; Programmsegment im FLASH
;
;-------------------------------------------------------------------------------
.cseg
    ldi     temp1, LOW(RAMEND)  ; Stackpointer initialisieren
    out     SPL, temp1
    ldi     temp1, HIGH(RAMEND)
    out     SPH, temp1
```

```
; CLK und PL als Ausgänge schalten

    ldi     temp1,(1<<CLK) | (1<<PL)
    out     SCHIEBE_DDR,temp1
;
; SPI Modul konfigurieren
;
    ldi     temp1, (1<<SPE) | (1<<MSTR) | (1<<CPOL)
    out     SPCR, temp1         ; keine Interrupts, MSB first, Master
                                ; CPOL = 1, CPHA =0
                                ; SCK Takt = 1/2 XTAL
    ldi     temp1, (1<<SPI2X)
    out     SPSR,temp1          ; double speed aktivieren
    out     SPDR,temp1          ; Dummy Daten, um SPIF zu setzen

; Zwei Bytes einlesen

    ldi     ZL,low(Daten)
    ldi     ZH,high(Daten)
    ldi     temp1,2
    rcall   schiebe_eingang

loop:
    rjmp    loop

;-------------------------------------------------------------------------------
;
; N Bytes seriell einlesen
;
; temp1 : N, Anzahl der Bytes
; Z     : Zeiger auf einen Datenbereich im SRAM
;-------------------------------------------------------------------------------
schiebe_eingang:
    push    temp2               ; Register sichern

    ; CLK ist im Ruhezustand schon auf HIGH, CPOL=1

dummyende:
    sbis    SPSR,7              ; prüfe ob Dummy Übertragung beendet ist, sonst
    rjmp    dummyende           ; kann es bei langsameren Übertragungsraten zu
                                ; Überschneidungen kommen.


    cbi     schiebe_port, pl    ; Daten parallel laden
    sbi     schiebe_port, pl

schiebe_eingang_1:
    sbis    SPSR,7              ; prüfe ob eine alte Übertragung beendet ist
    rjmp    schiebe_eingang_1

schiebe_eingang_byte_schleife:
    out     SPDR,temp1          ; beliebige Daten ins SPI Modul schreiben
                                ; um die Übertragung zu starten
schiebe_eingang_2:
    sbis    SPSR,7              ; auf das Ende der Übertragung warten
    rjmp    schiebe_eingang_2

    in      temp2, spdr         ; Daten lesen
    st      z+,temp2            ; Datenbyte speichern
    dec     temp1               ; Anzahl Bytes um eins verringern
    brne    schiebe_eingang_byte_schleife   ; wenn noch mehr Bytes zu lesen sind

    pop     temp2
    ret
```

Beides in *einer* Kette

Kann man die '595 und die '165 an *einem* SPI-Anschluss betreiben?

Ja, einfach verketten. Das geht auch „durcheinander“. Dafür gibt es zwei Möglichkeiten:

- Man verwendet *getrennte* Anschlüsse am Mikrocontroller für die Parallelübernahme der '595 (Ausgänge) und '165 (Eingänge). Dann sollten die Ausgänge am Ketten-Anfang (MOSI) und die Eingänge am Ketten-Ende (MISO) liegen. Beispielsweise werden für 3 Stück '595 und 2 Stück '165 24 Schiebetakte zum Ausgeben (mit anschließendem ↑RCLK) und 16 Schiebetakte (mit vorausgehendem ¬PL) zum Einlesen benötigt.
- Man verwendet einen *gemeinsamen* Anschluss **¬L** für ↑RCLK und ¬PL, ein Pin sparend. Die o.g. Reihenfolge der '595 und '165 ist weiterhin vorteilhaft. Nach dem o.g. Beispiel gilt folgendes Regime zum Ausgeben und Einlesen in einem Rutsch:
    - Low-Nadel an ¬L (zum Laden der '165-Flipflops; die '595-Ausgänge verändern sich nicht)
    - 8 x 3 = 24 Schiebetakte (auch zum Nur-Einlesen müssen die Ausgabedaten wiederholt ausgegeben werden)
    - Low-Nadel an ¬L (zum Aktualisieren der '595-Ausgänge; das erneute Einlesen in den '165 ist ohne Belang)

Kann man per Software messen, wie lang die Kette ist?

Für bestimmte PnP-artige Lösungen ist es zweckmäßig, die Länge der Kette zu kennen: Man schiebt einfach eine einzelne 1 (oder eine einzelne 0) durch die Kette, ohne Parallelübernahmen. Vorher muss man die Kette in ihrer zu erwartenden Maximallänge initialisieren. Benutzt man byteweise arbeitende Hardware-SPI, wird man auf ganze Byte-Längen vertrauen und kann für mehr Sicherheit ein bestimmtes Kennbyte durchschicken.

Kann man per Software messen, *wo* die '595 und die '165 liegen?

Nur wenn man davon ausgeht, dass sich die Eingänge nicht allzu schnell ändern. Das ist bei Tastern klar der Fall. Eine 100-prozentige Sicherheit gibt es nicht. Man schiebt ein bestimmtes Bitmuster (Byte) durch und nimmt '595 dort an, wo das Bitmuster auch zurückkommt. Man verwende mindestens 2 verschiedene Bitmuster, um eine derartige Eingangskombination auszuschließen. Wer ziemlich sicher gehen will, schiebt in einer Schleife alle 256 verschiedenen Kombinationen durch. Wenn (bei gemeinsamen ¬L) die Ausgänge unzulässig zappeln, muss man ¬OE beschalten, oder genau jene Bitkombinationen vermeiden.

Kann man andere SPI-Hardware in die Kette einfügen?

Genau dafür ist SPI gemacht! Aber nicht jeder (bspw.) A/D-Wandler ist dafür geeignet: Es muss möglich sein, Daten *unverändert* durchzuschieben. Oder wenigstens herumzuleiten. Wenn nicht, ist die einzig mögliche Anordnung *zwischen* den '595 (am Kettenanfang) und den '165 (am Kettenende). In der Regel wird eine gesonderte Chipselect-Leitung vom Mikrocontroller benötigt; nur in Ausnahmefällen ist o.g. **¬L** dafür geeignet (Datenblatt studieren!). Das Chipselect aus einem '595 zu generieren geht auch.

Was für Vorteile bringen '595 und '165 gegenüber einem größeren Mikrocontroller?

- Preis! Controller mit mehr Beinchen kosten oftmals deutlich mehr
- Günstigeres Routing zu weiter entfernten Schaltungsteilen
- Kleinere Steckverbinder zu weiteren Platinen, etwa Anzeige- und Bedien-Frontplatten
- Beliebige Verlängerbarkeit
- ESD-Schutz und Überlastschutz des Controllers
- Verbleib beim „gewohnten“ Mikrocontroller möglich

Und welche Nachteile handelt man sich ein?

- Fehlende Sonderfunktionen (etwa Hardware-PWM, Interrupts)
- Fehlende Bidirektionalität
- Geringere Geschwindigkeit sowie zusätzliche Verzögerung beim Startup
- Fehlende Weak-Pullups (bei den Eingängen, es sei denn, man verwendet veraltete 74LS165)
- Höherer Bestückungsaufwand durch das Mehr an Bauteilen

Bekannte Probleme

AVR Studio 4.12 (Build 498) hat Probleme bei der korrekten Simulation des SPI-Moduls.

- Der Double-Speed Modus funktioniert nicht.
- Das Bit SPIF im Register SPSR, welches laut Dokumentation nur lesbar ist, ist im Simulator auch schreibbar! Das kann zu Verwirrung und Fehlern in der Simulation führen.

Hardwareprobleme

- Wenn das SPI-Modul aktiviert wird, wird **NICHT** automatisch SPIF gesetzt, es bleibt auf Null. Damit würde die erste Abfrage in *Schiebe_alles* in einer Endlosschleife hängen bleiben. Deshalb muss nach der Initialisierung des SPI-Moduls ein Dummy-Byte gesendet werden, damit am Ende der Übertragung SPIF gesetzt wird
- Da das SPI-Modul in Senderichtung nur einfach gepuffert ist, ist es nicht möglich absolut lückenlos Daten zu senden, auch wenn man mit **nop** eine feste minimale Zeit zwischen zwei Bytes warten würde. Zwischen zwei Bytes muss immer eine Pause von mind. 2 Systemtakten eingehalten werden.

Weblinks

- AVR151: Setup And Use of The SPI Atmel Application Note (PDF)
- datasheetcatalog.com: 74HC595
- Roboternetz: Portexpander am AVR
- Interactive 595 Shift Register Simulator
- 48 I/O Porterweiterung mit Schieberegister 74HC595 und 74HC165

AVR-Tutorial: SRAM

Inhaltsverzeichnis

- 1 SRAM - Der Speicher des Controllers
- 2 Das .DSEG und .BYTE
- 3 spezielle Befehle
  - 3.1 LDS
  - 3.2 STS
  - 3.3 Beispiel
- 4 Spezielle Register
  - 4.1 Der Z-Pointer (R30 und R31)
  - 4.2 LD
  - 4.3 LDD
  - 4.4 ST
  - 4.5 STD
  - 4.6 Beispiel
  - 4.7 X-Pointer, Y-Pointer
- 5 Siehe auch

SRAM - Der Speicher des Controllers

Nachdem in einem der vorangegangenen Kapitel eine Software-PWM vorgestellt und in einem weiteren Kapitel darüber gesprochen wurde, wie man mit Schieberegistern die Anzahl an I/O-Pins erhöhen kann, wäre es naheliegend, beides zu kombinieren und den ATmega8 mal 20 oder 30 LEDs ansteuern zu lassen. Wenn es da nicht ein Problem gäbe: die Software-PWM hält ihre Daten in Registern, so wie das praktisch alle Programme bisher machten. Während allerdings 6 PWM-Kanäle noch problemlos in den Registern untergebracht werden konnten, ist dies mit 30 oder noch mehr PWM-Kanälen nicht mehr möglich. Es gibt schlicht und ergreifend nicht genug Register.

Es gibt aber einen Ausweg. Der ATmega8 verfügt über 1kByte **SRAM** (statisches RAM). Dieses RAM wurde bereits indirekt durch den **Stack** benutzt. Bei jedem Aufruf eines Unterprogrammes, sei es über einen expliziten **CALL** (bzw. **RCALL**) oder einen Interrupt, wird die Rücksprungadresse irgendwo gespeichert. Dies geschieht genau in diesem SRAM. Auch **PUSH** und **POP** operieren in diesem Speicher.

Ein Programm darf Speicherzellen im **SRAM** direkt benutzen und dort Werte speichern bzw. von

dort Werte einlesen. Es muss nur darauf geachtet werden, dass es zu keiner Kollision mit dem Stack kommt, in dem z. B. die erwähnten Rücksprungadressen für Unterprogramme gespeichert werden. Da viele Programme aber lediglich ein paar Byte **SRAM** brauchen, der Rücksprungstack von der oberen Grenze des **SRAM** nach unten wächst und der ATmega8 immerhin über **1kByte SRAM** verfügt, ist dies in der Praxis kein all zu großes Problem.

Das .DSEG und .BYTE

Um dem Assembler mitzuteilen, dass sich der folgende Abschnitt auf das SRAM bezieht, gibt es die Direktive **.DSEG** (Data Segment). Alle nach einer **.DSEG** Direktive folgenden Speicherreservierungen werden vom Assembler im SRAM durchgeführt.

Die Direktive **.BYTE** stellt dabei eine derartige Speicherreservierung dar. Es ermöglicht, der Speicherreservierung einen Namen zu geben und es erlaubt auch, nicht nur 1 Byte sondern eine ganze Reihe von Bytes unter einem Namen zu reservieren.

```
            .DSEG                ; Umschalten auf das SRAM Datensegment
Counter:    .BYTE  1             ; 1 Byte unter dem Namen 'Counter' reservieren
Test:       .BYTE  20            ; 20 Byte unter dem Namen 'Test' reservieren
```

spezielle Befehle

Für den Zugriff auf den **SRAM**-Speicher gibt es spezielle Befehle. Diese holen entweder den momentanen Inhalt einer Speicherzelle und legen ihn in einem Register ab oder legen den Inhalt eines Registers in einer **SRAM**-Speicherzelle ab.

LDS

Liest die angegebene **SRAM**-Speicherzelle und legt den gelesenen Wert in einem Register ab.

```
        LDS  r17, Counter      ; liest die Speicherzelle mit dem Namen 'Counter'
                               ; und legt den gelesenen Wert im Register r17 ab
```

STS

Legt den in einem Register gespeicherten Wert in einer **SRAM**-Speicherzelle ab.

```
        STS  Counter, r17      ; Speichert den Inhalt von r17 in der
                               ; Speicherzelle 'Counter'
```

Beispiel

Eine mögliche Implementierung der Software-PWM, die den PWM-Zähler sowie die einzelnen OCR-Grenzwerte im **SRAM** anstelle von Registern speichert, könnte z. B. so aussehen:

```
.include "m8def.inc"

.def temp  = r16
.def temp1 = r17
```

```
.def temp2 = r18

.org 0x0000
        rjmp    main                    ; Reset Handler
.org OVF0addr
        rjmp    timer0_overflow         ; Timer Overflow Handler

main:
        ldi     temp, HIGH(RAMEND)
        out     SPH, temp
        ldi     temp, LOW(RAMEND)       ; Stackpointer initialisieren
        out     SPL, temp

        ldi     temp, 0xFF              ; Port B auf Ausgang
        out     DDRB, temp

        ldi     temp2, 0
        sts     OCR_1, temp2
        ldi     temp2, 1
        sts     OCR_2, temp2
        ldi     temp2, 10
        sts     OCR_3, temp2
        ldi     temp2, 20
        sts     OCR_4, temp2
        ldi     temp2, 80
        sts     OCR_5, temp2
        ldi     temp2, 127
        sts     OCR_6, temp2

        ldi     temp, (1<<CS00)         ; CS00 setzen: Teiler 1
        out     TCCR0, temp

        ldi     temp, (1<<TOIE0)        ; TOIE0: Interrupt bei Timer Overflow
        out     TIMSK, temp

        sei

loop:   rjmp    loop

; ***************************************************************************
; Behandlung des Timer Overflows
;
; realisiert die PWM auf 6 Kanälen
;
; veränderte Register: keine
;
timer0_overflow:                    ; Timer 0 Overflow Handler
        push    temp                ; Alle verwendeten Register sichern
        push    temp1
        push    temp2
        in      temp, SREG
        push    temp

        lds     temp1, PWMCount     ; den PWM-Zaehler aus dem Speicher holen
        inc     temp1               ; Zaehler erhoehen
        cpi     temp1, 128          ; wurde 128 erreicht ?
        brne    WorkPWM             ; Nein
        clr     temp1               ; Ja: PWM-Zaehler wieder auf 0

WorkPWM:
        sts     PWMCount, temp1     ; den PWM-Zaehler wieder speichern
        ldi     temp, 0b11000000    ; 0 .. LED an, 1 .. LED aus

        lds     temp2, OCR_1
```

```
        cp      temp1, temp2        ; Ist der Grenzwert für LED 1 erreicht
        brlt    OneOn
        ori     temp, $01

OneOn:  lds     temp2, OCR_2
        cp      temp1, temp2        ; Ist der Grenzwert für LED 2 erreicht
        brlt    TwoOn
        ori     temp, $02

TwoOn:  lds     temp2, OCR_3
        cp      temp1, temp2        ; Ist der Grenzwert für LED 3 erreicht
        brlt    ThreeOn
        ori     temp, $04

ThreeOn:lds     temp2, OCR_4
        cp      temp1, temp2        ; Ist der Grenzwert für LED 4 erreicht
        brlt    FourOn
        ori     temp, $08

FourOn: lds     temp2, OCR_5
        cp      temp1, temp2        ; Ist der Grenzwert für LED 5 erreicht
        brlt    FiveOn
        ori     temp, $10

FiveOn: lds     temp2, OCR_6
        cp      temp1, temp2        ; Ist der Grenzwert für LED 6 erreicht
        brlt    SetBits
        ori     temp, $20

SetBits:                            ; Die neue Bitbelegung am Port ausgeben
        out     PORTB, temp

        pop     temp                ; die gesicherten Register wiederherstellen
        out     SREG, temp
        pop     temp2
        pop     temp1
        pop     temp

        reti
;
; ***********************************************
;
          .DSEG                         ; das Folgende kommt ins SRAM

PWMCount: .BYTE   1                     ; Der PWM-Counter (0 bis 127)
OCR_1:    .BYTE   1                     ; 6 Bytes für die OCR-Register
OCR_2:    .BYTE   1
OCR_3:    .BYTE   1
OCR_4:    .BYTE   1
OCR_5:    .BYTE   1
OCR_6:    .BYTE   1
```

Die ISR sichert alle verwendeten Register und stellt sie am Ende der ISR wieder her. Dies ist zwar streng genommen in diesem Beispiel nicht notwendig, da das eigentliche Programm in der Hauptschleife ja nichts tut, aber ausser ein bischen Zeit kostet das nichts und es erspart das Kopfkratzen, wenn dann irgendwann in der Hauptschleife Ergänzungen und anderer Code dazu kommen. Man kann natürlich einige Register speziell für die Verwendung ausschliesslich in der ISR reservieren und sich so das Sichern/Wiederherstellen ersparen, aber das SREG muss im Normalfall innerhalb einer ISR auf jeden Fall gesichert und wiederhergestellt werden.

Spezielle Register

Der Z-Pointer (R30 und R31)

Das Registerpärchen **R30** und **R31** kann zu einem einzigen logischen Register zusammengefasst werden und heisst dann **Z-Pointer**. Diesem kann eine spezielle Aufgabe zukommen, indem er als Adressangabe fungieren kann, von welcher Speicherzelle im SRAM ein Ladevorgang (bzw. Speichervorgang) durchgeführt werden soll. Anstatt die Speicheradresse wie beim **LDS** bzw. **STS** direkt im Programmcode anzugeben, kann diese Speicheradresse zunächst in den **Z-Pointer** geladen werden und der Lesevorgang (Schreibvorgang) über diesen **Z-Pointer** abgewickelt werden. Dadurch wird aber die **SRAM**-Speicheradresse berechenbar, denn natürlich kann mit den Registern **R30** und **R31**, wie mit den anderen Registern auch, Arithmetik betrieben werden. Besonders komfortabel ist dies, da im Ladebefehl noch zusätzliche Manipulationen angegeben werden können, die oft benötigte arithmetische Operationen implementieren.

LD

- **LD rxx, Z**
- **LD rxx, Z+**
- **LD rxx, -Z**

Lädt das Register **rxx** mit dem Inhalt der Speicherzelle, deren Adresse im **Z-Pointer** angegeben ist. Bei den Varianten mit **Z+** bzw. **-Z** wird zusätzlich der **Z-Pointer** ***nach*** der Operation um 1 erhöht bzw. ***vor*** der Operation um 1 vermindert.

LDD

- **LDD rxx, Z+q**

Hier erfolgt der Zugriff wieder über den **Z-Pointer** wobei vor dem Zugriff zur Adressangabe im **Z-Pointer** noch das Displacement **q** addiert wird.

Enthält also der **Z-Pointer** die Adresse $1000 und sei **q** der Wert $28, so wird mit einer Ladeanweisung

```
LDD r18, Z + $28
```

der Inhalt der Speicherzellen $1000 + $28 = $1028 in das Register r18 geladen.

Der Wertebereich für **q** erstreckt sich von 0 bis 63.

ST

- **ST Z, rxx**
- **ST Z+, rxx**
- **ST -Z, rxx**

Speichert den Inhalt des Register **rxx** in der Speicherzelle, deren Adresse im **Z-Pointer** angegeben ist. Bei den Varianten mit **Z+** bzw. **-Z** wird zusätzlich der **Z-Pointer** ***nach*** der Operation um 1

erhöht bzw. ***vor*** der Operation um 1 vermindert.

STD

- **STD Z+q, rxx**

Hier erfolgt der Zugriff wieder über den **Z-Pointer** wobei vor dem Zugriff zur Adressangabe im **Z-Pointer** noch das Displacement **q** addiert wird.

Enthält also der **Z-Pointer** die Adresse $1000 und sei **q** der Wert $28, so wird mit einer Speicheranweisung

```
        STD Z + $28, r18
```

der Inhalt des Registers r18 in der Speicherzellen $1000 + $28 = $1028 gespeichert.

Der Wertebereich für **q** erstreckt sich von 0 bis 63.

Beispiel

Durch Verwendung des **Z-Pointers** ist es möglich die Interrupt Funktion wesentlich kürzer und vor allem ohne ständige Wiederholung von im Prinzip immer gleichem Code zu formulieren. Man stelle sich nur mal vor wie dieser Code aussehen würde, wenn anstelle von 6 PWM Stufen, deren 40 gebraucht würden. Mit dem **Z-Pointer** ist es möglich diesen auf das erste der OCR Bytes zu setzen und dann in einer Schleife eines nach dem anderen abzuarbeiten. Nach dem Laden des jeweiligen OCR Wertes, wird der **Z-Pointer** automatisch durch den **LD**-Befehl auf das nächste zu verarbeitende OCR Byte weitergezählt.

```
.include "m8def.inc"

.def temp  = r16
.def temp1 = r17
.def temp2 = r18
.def temp3 = r19

.org 0x0000
        rjmp    main                    ; Reset Handler
.org OVF0addr
        rjmp    timer0_overflow         ; Timer Overflow Handler

main:
        ldi     temp, HIGH(RAMEND)
        out     SPH, temp
        ldi     temp, LOW(RAMEND)       ; Stackpointer initialisieren
        out     SPL, temp

        ldi     temp, 0xFF              ; Port B auf Ausgang
        out     DDRB, temp

        ldi     r30,LOW(OCR)            ; den Z-Pointer mit dem Start der OCR
Bytes laden
        ldi     r31,HIGH(OCR)

        ldi     temp2, 0
        st      Z+, temp2
        ldi     temp2, 1
```

```
        st      Z+, temp2
        ldi     temp2, 10
        st      Z+, temp2
        ldi     temp2, 20
        st      Z+, temp2
        ldi     temp2, 80
        st      Z+, temp2
        ldi     temp2, 127
        st      Z+, temp2

        ldi     temp2, 0                ; den PWM Counter auf 0 setzen
        sts     PWMCount, temp2

        ldi     temp, (1<<CS00)         ; CS00 setzen: Teiler 1
        out     TCCR0, temp

        ldi     temp, (1<<TOIE0)        ; TOIE0: Interrupt bei Timer Overflow
        out     TIMSK, temp

        sei

loop:   rjmp    loop

; ************************************************************************
; Behandlung des Timer Overflows
;
; realisiert die PWM auf 6 Kanälen
;
; veränderte Register: keine
;
timer0_overflow:                        ; Timer 0 Overflow Handler
        push    temp                    ; Alle verwendeten Register sichern
        push    temp1
        push    temp2
        push    R30
        push    R31
        in      temp, SREG
        push    temp

        lds     temp1, PWMCount         ; den PWM ZAehler aus dem Speicher holen
        inc     temp1                   ; Zaehler erhoehen
        cpi     temp1, 128              ; wurde 128 erreicht ?
        brne    WorkPWM                 ; Nein
        clr     temp1                   ; Ja: PWM Zaehler auf 0 setzen

WorkPWM:
        sts     PWMCount, temp1         ; den PWM Zaehler wieder speichern

        ldi     r30,LOW(OCR)            ; den Z-Pointer mit dem Start der OCR
Bytes laden
        ldi     r31,HIGH(OCR)
        ldi     temp3, $01              ; das Bitmuster für PWM Nr. i
        ldi     temp, 0b11000000        ; 0 .. Led an, 1 .. Led aus

pwmloop:
        ld      temp2, Z+               ; den OCR Wert für PWM Nr. i holen und Z-
Pointer erhöhen
        cp      temp1, temp2            ; ist der Grenzwert für PWM Nr. i
erreicht?
        brlo    LedOn
        or      temp, temp3
LedOn:
        lsl     temp3                   ; das Bitmuster schieben
        cpi     temp3, $40              ; alle Bits behandelt ?
```

```
        brne    pwmloop                 ; nächster Schleifendurchlauf

        out     PORTB, temp             ; Die neue Bitbelegung am Port ausgeben

        pop     temp                    ; die gesicherten Register
wiederherstellen
        out     SREG, temp
        pop     R31
        pop     R30
        pop     temp2
        pop     temp1
        pop     temp

        reti
;
; ********************************************************************
;
        .DSEG                           ; das Folgende kommt ins SRAM

PWMCount: .BYTE  1                      ; der PWM Zaehler (0 bis 127)
OCR:      .BYTE  6                      ; 6 Bytes für die OCR Register
```

X-Pointer, Y-Pointer

Neben dem **Z-Pointer** gibt es noch den **X-Pointer** bzw. **Y-Pointer**. Sie werden gebildet von den Registerpärchen

- **X-Pointer**: r26, r27
- **Y-Pointer**: r28, r29
- **Z-Pointer**: r30, r31

Alles über den **Z-Pointer** gesagte gilt sinngemäß auch für den **X-Pointer** bzw. **Y-Pointer** mit einer Ausnahme: Mit dem **X-Pointer** ist kein Zugriff **LDD**/**STD** mit einem Displacement möglich.

Siehe auch

- Adressierung

AVR-Tutorial: 7-Segment-Anzeige

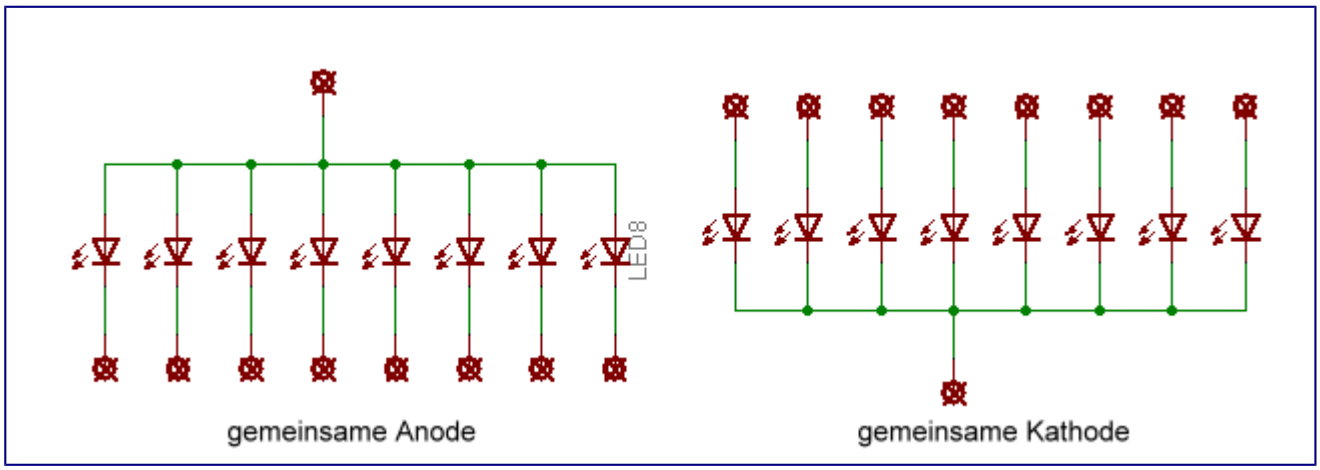
Die Ausgabe von Zahlenwerten auf ein Text-LCD ist sicherlich das Nonplusultra, aber manchmal liegen die Dinge sehr viel einfacher. Um beispielsweise eine Temperatur anzuzeigen ist ein LCD etwas Overkill. In solchen Fällen kann die Ausgabe auf ein paar 7-Segmentanzeigen gemacht werden. Ausserdem haben 7-Segmentanzeigen einen ganz besonderen Charme :-)

Inhaltsverzeichnis

- 1 Typen von 7-Segment Anzeigen
- 2 Eine einzelne 7-Segment Anzeige
  - 2.1 Schaltung
  - 2.2 Codetabelle
  - 2.3 Programm
- 3 Mehrere 7-Segment Anzeigen (Multiplexen)
  - 3.1 Programm
- 4 Forenbeiträge

Typen von 7-Segment Anzeigen

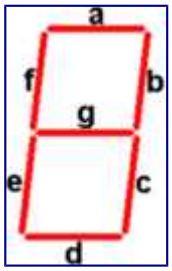
Eine einzelne 7-Segmentanzeige besteht aus sieben (mit Dezimalpunkt acht) einzelnen LEDs in einem gemeinsamen Gehäuse. Aus praktischen Gründen wird einer der beiden Anschlüsse jeder LED mit den gleichen Anschlüssen der anderen LED verbunden und gemeinsam aus dem Gehäuse herausgeführt. Das spart Pins am Gehäuse und später bei der Ansteuerung. Dementsprechend spricht man von Anzeigen mit *gemeinsamer Anode* (engl. common anode) bzw. *gemeinsamer Kathode* (engl. common cathode).



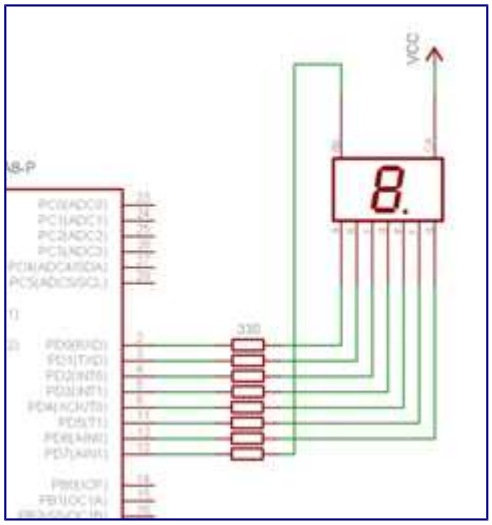
Interne Verschaltung der 7-Segmentanzeigen

Eine einzelne 7-Segment Anzeige

Schaltung



Pin-
belegung einer 7-Segment-
anzeige



Ansteuerung einer einzelnen 7-Segmentanzeige

Eine einzelne 7-Segmentanzeige wird nach dem folgenden Schema am **Port D** des Mega8 angeschlossen. Port D wurde deshalb gewählt, da er am Mega8 als einziger Port aus den vollen 8 Bit besteht. Die 7-Segmentanzeige hat neben den Segmenten **a** bis **g** eine gemeinsame Anode **CA** sowie einen Dezimalpunkt **dp** (siehe Bild).

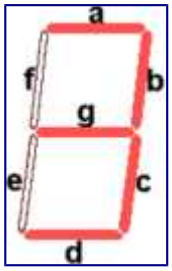
Welcher Pin an der Anzeige welchem Segment (a-g) bzw. dem Dezimalpunkt entspricht wird am besten dem Datenblatt zur Anzeige entnommen. Hat man kein Datenblatt dann kann man auch empirisch die Pinbelegung feststellen, indem man mit einer 5V Quelle und einem 1k Widerstand einfach probweise alle Pins 'abklappert'. 1k deswegen, damit man im Zweifellsfall Low-Current LED nicht überfordert. Eine 7-Segment Anzeige, die auf die üblichen 10 bis 20mA ausgelegt ist, wird damit immer noch leuchten, wenn auch schwach. Aber mehr braucht es ja auch nicht um die Pinbelegung feststellen zu können. Im Folgenden wird von dieser Segmentbelegung ausgegangen:

Wird eine andere Belegung genutzt dann ist das prinzipiell möglich, jedoch müsste das in der Programmierung berücksichtigt werden.

Da eine 7-Segmentanzeige konzeptionell sieben einzelnen LEDs entspricht, ergibt sich im Prinzip

keine Änderung in der Ansteuerung einer derartigen Anzeige im Vergleich zur LED Ansteuerung wie sie im AVR-Tutorial: IO-Grundlagen gezeigt wird. Genau wie bei den einzelnen LEDs wird eine davon eingeschaltet, indem der zugehörige Port Pin auf 0 gesetzt wird. Aber anders als bei einzelnen LED möchte man mit einer derartigen Anzeige eine Ziffernanzeige erhalten. Dazu ist es lediglich notwendig, für eine bestimmte Ziffer die richtigen LEDs einzuschalten.

Codetabelle



Darstellung der Ziffer "3"

Die Umkodierung von einzelnen Ziffern in ein bestimmtes Ausgabemuster kann über eine sog. Codetabelle geschehen: Die auszugebende Ziffer wird als Offset zum Anfang dieser Tabelle aufgefasst und aus der Tabelle erhält man ein Byte (Code), welches direkt auf den Port ausgegeben werden kann und das entsprechende Bitmuster enthält, sodass die für diese Ziffer notwendigen LED ein- bzw. ausgeschaltet sind.

Beispiel

> Um die Ziffer **3** anzuzeigen, müssen auf der Anzeige die Segmente **a**, **b**, **c**, **d** und **g** aufleuchten. Alle anderen Segmente sollen dunkel sein.

Aus dem Anschlußschema ergibt sich, dass die dazu notwendige Ausgabe am Port binär **10110000** lauten muss. Untersucht man dies für alle Ziffern, so ergibt sich folgende Tabelle:

```
.db  0b11000000     ; 0: a, b, c, d, e, f
.db  0b11111001     ; 1: b, c
.db  0b10100100     ; 2: a, b, d, e, g
.db  0b10110000     ; 3: a, b, c, d, g
.db  0b10011001     ; 4: b, c, f, g
.db  0b10010010     ; 5: a, c, d, f, g
.db  0b10000010     ; 6: a, c, d, e, f, g
.db  0b11111000     ; 7: a, b, c
.db  0b10000000     ; 8: a, b, c, d, e, f, g
.db  0b10010000     ; 9: a, b, c, d, f, g
```

Programm

Das Testprogramm stellt nacheinander die Ziffern 0 bis 9 auf der 7-Segmentanzeige dar. Die jeweils auszugebende Zahl steht im Register **count** und wird innerhalb der Schleife um jeweils 1 erhöht. Hat das Register den Wert 10 erreicht, so wird es wieder auf 0 zurückgesetzt. Nach der Erhöhung folgt eine Warteschleife, welche dafür sorgt, dass bis zur nächsten Ausgabe eine gewisse Zeit vergeht. Normalerweise macht man keine derartigen langen Warteschleifen, aber hier geht es ja nicht ums Warten sondern um die Ansteuerung einer 7-Segmentanzeige. Einen Timer dafür zu

benutzen wäre zunächst zuviel Aufwand.

Die eigentliche Ausgabe und damit der in diesem Artikel interessante Teil findet jedoch direkt nach dem Label *loop* statt. Die bereits bekannte Codetabelle wird mittels **.db** Direktive (**d**efine **b**yte) in den Flash-Speicher gelegt. Der Zugriff darauf erfolgt über den Z-Pointer und dem Befehl **lpm**. Zusätzlich wird vor dem Zugriff noch der Wert des Registers **count** und damit der aktuelle Zählerwert zum Z-Pointer addiert.

Beachtet werden muss nur, dass der Zählerwert verdoppelt werden muss. Dies hat folgenden Grund: Wird die Tabelle so wie hier gezeigt mittels einzelnen **.db** Anweisungen aufgebaut, so fügt der Assembler sog. **Padding Bytes** zwischen die einzelnen Bytes ein, damit jede **.db** Anweisung auf einer geraden Speicheradresse liegt. Dies ist eine direkte Folge der Tatsache, dass der Flash-Speicher **wortweise** (16 Bit) und nicht **byteweise** (8 Bit) organisiert ist. Da aber von einem .db in der Tabelle zum nächsten .db eine Differenz von 2 Bytes vorliegt, muss dies in der Berechnung berücksichtigt werden. Im zweiten Beispiel auf dieser Seite wird dies anders gemacht. Dort wird gezeigt wie man durch eine andere Schreibweise der Tabelle das Erzeugen der Padding Bytes durch den Assembler verhindern kann.

Aus dem gleichen Grund wird auch der Z-Pointer mit dem 2-fachen der Startadresse der Tabelle geladen. Die Startadresse wird vom Assembler in wortweiser Adressierung eingesetzt, **lpm** möchte die Zugriffsadresse als Byteadresse angegeben haben.

Interessant ist auch, dass in der Berechnung ein Register benötigt wird, welches den Wert 0 enthält. Dies deshalb, da es im AVR keinen Befehl gibt der eine Konstante mit gleichzeitiger Berücksichtigung des Carry-Bits addieren kann. Daher muss diese Konstante zunächst in ein Register geladen werden und erst dann kann die Addition mithilfe dieses Registers vorgenommen werden. Das Interessante daran ist nun, dass dieser Umstand in sehr vielen Programmen vorkommt und es sich bei der Konstanten in der überwiegenden Mehrzahl der Fälle um die Konstante 0 handelt. Viele Programmierer reservieren daher von vorne herein ein Register für diesen Zweck und nennen es das Zero-Register. Sinnvollerweise legt man dieses Register in den Bereich r0..r15, da diese Register etwas zweitklassig sind (ldi, cpi etc. funktionieren nicht damit).

```
.include "m8def.inc"

.def zero  = r1
.def count = r16
.def temp1 = r17

.org 0x0000
           rjmp    main                ; Reset Handler
;
main:
           ldi     temp1, HIGH(RAMEND)
           out     SPH, temp1
           ldi     temp1, LOW(RAMEND)  ; Stackpointer initialisieren
           out     SPL, temp1

;
           ldi     temp1, $FF         ; die Anzeige hängt am Port D
           out     DDRD, temp1        ; alle Pins auf Ausgang
;
           ldi     count, 0           ; und den Zähler initialisieren
```

```
            mov     zero, count
;
loop:
            ldi     ZL, LOW(Codes*2)  ; die Startadresse der Tabelle in den
            ldi     ZH, HIGH(Codes*2) ; Z-Pointer laden

            mov     temp1, count      ; die wortweise Adressierung der Tabelle
            add     temp1, count      ; berücksichtigen

            add     ZL, temp1         ; und ausgehend vom Tabellenanfang
            adc     ZH, zero          ; die Adresse des Code Bytes berechnen

            lpm                       ; dieses Code Byte in das Register r0 laden

            out     PORTD, r0         ; und an die Anzeige ausgeben
;
            inc     count             ; den Zähler erhöhen, wobei der Zähler
            cpi     count, 10         ; immer nur von 0 bis 9 zählen soll
            brne    wait
            ldi     count, 0
;
wait:       ldi     r17, 10           ; und etwas warten, damit die Ziffer auf
wait0:      ldi     r18, 0            ; der Anzeige auch lesbar ist, bevor die
wait1:      ldi     r19, 0            ; nächste Ziffer gezeigt wird
wait2:      dec     r19
            brne    wait2
            dec     r18
            brne    wait1
            dec     r17
            brne    wait0
;
            rjmp    loop              ; auf zur nächsten Ausgabe
;
Codes:                                ; Die Codetabelle für die Ziffern 0 bis 9
                                      ; sie regelt, welche Segmente für eine
bestimmte
                                      ; Ziffer eingeschaltet werden müssen
                                      ;
            .db     0b11000000        ; 0: a, b, c, d, e, f
            .db     0b11111001        ; 1: b, c
            .db     0b10100100        ; 2: a, b, d, e, g
            .db     0b10110000        ; 3: a, b, c, d, g
            .db     0b10011001        ; 4: b, c, f, g
            .db     0b10010010        ; 5: a, c, d, f, g
            .db     0b10000010        ; 6: a, c, d, e, f, g
            .db     0b11111000        ; 7: a, b, c
            .db     0b10000000        ; 8: a, b, c, d, e, f, g
            .db     0b10010000        ; 9: a, b, c, d, f, g
```

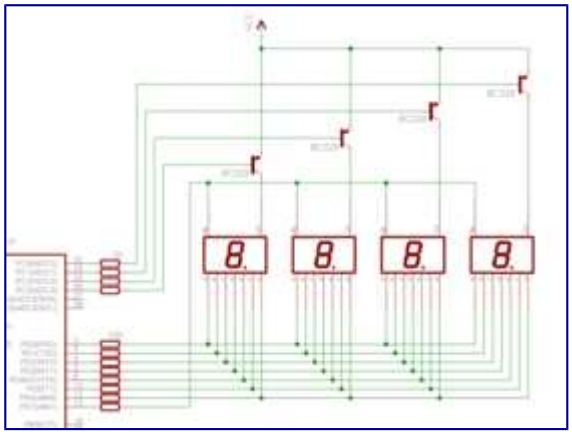
Mehrere 7-Segment Anzeigen (Multiplexen)

Mit dem bisherigen Vorwissen könnte man sich jetzt daran machen, auch einmal drei oder vier Anzeigen mit dem Mega8 anzusteuern. Leider gibt es da ein Problem, denn für eine Anzeige sind acht Portpins notwendig - vier Anzeigen würden demnach 32 Portpins benötigen. Die hat der Mega8 aber nicht. Dafür gibt es aber mehrere Auswege. Schieberegister sind bereits in einem anderen Tutorial beschrieben. Damit könnte man sich ganz leicht die benötigten 32 Ausgangsleitungen mit nur 3 Portpins erzeugen. Das Prinzip der Ansteuerung unterscheidet sich in nichts von der Ansteurung einer einzelnen 7-Segment Anzeige, lediglich die Art und Weise, wie die 'Ausgangspins' zu ihren Werten kommen ist anders und durch die Verwendung von Schieberegistern

vorgegeben. An dieser Stelle soll aber eine andere Variante der Ansteuerung gezeigt werden. Im Folgenden werden wir uns daher das Multiplexen einmal näher ansehen.

Multiplexen bedeutet, dass nicht alle vier Anzeigen gleichzeitig eingeschaltet sind, sondern immer nur Eine für eine kurze Zeit. Geschieht der Wechsel zwischen den Anzeigen schneller als wir Menschen das wahrnehmen können, so erscheinen uns alle vier Anzeigen gleichzeitig in Betrieb zu sein obwohl immer nur Eine für eine kurze Zeit aufleuchtet. Dabei handelt es sich praktisch um einen Sonderfall einer LED-Matrix mit nur einer Zeile. Die vier Anzeigen können sich dadurch die einzelnen Segmentleitungen teilen und alles was benötigt wird sind 4 zusätzliche Steuerleitungen für die 4 Anzeigen, mit denen jeweils eine Anzeige eingeschaltet wird. Dieses Ein/Ausschalten wird mit einem pnp-Transistor in der Versorgungsspannung jeder Anzeige realisiert, die vom Mega8 am **PortC** angesteuert werden.

Ein Aspekt dieser Ansteuerungsart ist die Multiplexfrequenz, also ein kompletter Zyklus das Weiterschaltens von einer Anzeige zur nächsten . Sie muss hoch genug sein, um ein Flimmern der Anzeige zu vermeiden. Das menschliche Auge ist träge, im Kino reichen 24 Bilder pro Sekunde, beim Fernseher sind es 50. Um auf der sicheren Seite zu sein, dass auch Standbilder ruhig wirken, sollen jedes Segment mit mindestens 100 Hz angesteuert werden, es also mindestens alle 10ms angeschaltet ist. In Ausnahmefällen können aber selbst 100 Hz noch flimmern, z. B. wenn die Anzeige schnell bewegt wird oder wenn es zu Interferenzerscheinungen mit künstlichen Lichtquellen kommt, die mit Wechselstrom betrieben werden.



Ansteuerung von vier 7-Segmentanzeigen per Zeit-Multiplex

Bei genauerer Betrachtung fällt auch auf, dass die vier Anzeigen nicht mehr ganz so hell leuchten wie die eine einzelne Anzeige ohne Multiplexen. Bei wenigen Anzeigen ist dies praktisch kaum sichtbar, erst bei mehreren Anzeigen wird es deutlich. Um dem entgegen zu wirken lässt man pro Segment einfach mehr Strom fließen, bei LEDs dürfen dann 20mA überschritten werden. Als Faustregel gilt, dass der n-fache Strom für die (1/n)-fache Zeit fließen darf. Details finden sich im Datenblatt unter dem Punkt **Peak-Current** (Spitzenstrom) und **Duty-Cycle** (Tastverhältnis).

Allerdings gibt es noch ein anderes Problem wenn insgesamt zu viele Anzeigen gemultiplext werden. Die Pulsströme durch die LEDs werden einfach zu hoch. Die meisten LEDs kann man bis 8:1 multiplexen, manchmal auch bis 16:1. Hier fliesst aber schon ein Pulsstrom von 320mA (16 x 20mA), was nicht mehr ganz ungefährlich ist. **Strom lässt sich durch Multiplexen nicht sparen**, denn die verbrauchte Leistung ändert sich beim n-fachen Strom für 1/n der Zeit nicht. Kritisch wird es aber, wenn das Multiplexen deaktiviert wird (Ausfall der Ansteuerung durch Hardware- oder

Softwarefehler) und der n-fache Strom dauerhaft durch eine Segment-LED fließt. Bei 320mA werden die meisten LEDs innerhalb von Sekunden zerstört. Hier muss sichergestellt werden, dass sowohl Programm (Breakpoint im Debugger) als auch Schaltung (Reset, Power-On, [1]) diesen Fall verhindern. Prinzipiell sollte man immer den Pulsstrom und die Multiplexfrequenz einmal überschlagen, bevor der Lötkolben angeworfen wird.

Sollten die Anzeigen zu schwach leuchten, wie bereits beschrieben, können die Ströme durch die Anzeigen erhöht werden. Dazu werden die 330Ω Widerstände kleiner gemacht. Da hier 4 Anzeigen gemultiplext werden, würden sich Widerstände in der Größenordnung von 100Ω anbieten. Auch muss dann der Basiswiderstand der Transistoren verkleinert werden. Auch muss berücksichtigt werden, dass der Mega8 in Summe an seinen Portpins und an den Versorgungsleitungen nicht beliebig viel Strom liefern oder abführen kann. Auch hier ist daher wieder ein Blick ins Datenblatt angebracht und gegebenenfalls muss wieder ein Transistor als Verstärker eingesetzt werden (oder eben fertige Treiberstufen in IC-Form).

Programm

Das folgende Programm zeigt eine Möglichkeit zum Multiplexen. Dazu wird ein Timer benutzt, der in regelmässigen Zeitabständen einen Overflow Interrupt auslöst. Innerhalb der Overflow Interrupt Routine wird

- die momentan erleuchtete Anzeige abgeschaltet
- das Muster für die nächste Anzeige am Port D ausgegeben
- die nächste Anzeige durch eine entsprechende Ausgabe am Port C eingeschaltet

Da Interruptfunktionen kurz sein sollten, holt die Interrupt Routine das auszugebende Muster für jede Stelle direkt aus dem SRAM, wo sie die Ausgabefunktion hinterlassen hat. Dies hat 2 Vorteile:

- Zum einen braucht die Interrupt Routine die Umrechnung einer Ziffer in das entsprechende Bitmuster nicht selbst machen
- Zum anderen ist die Anzeigefunktion dadurch unabhängig von dem was angezeigt wird. Die Interrupt Routine gibt das Bitmuster so aus, wie sie es aus dem SRAM liest. Werden die SRAM Zellen mit geeigneten Bitmustern gefüllt, können so auch einige Buchstaben oder Balkengrafik oder auch kleine Balken-Animationen abgespielt werden. Insbesondere letzteres sieht man manchmal bei Consumer-Geräten kurz nach dem Einschalten des Gerätes um eine Art Defektkontrolle zu ermöglichen oder einfach nur als optischer Aufputz.

Die Funktion out_number ist in einer ähnlichen Form auch schon an anderer Stelle vorgekommen: Sie verwendet die Technik der fortgesetzten Subtraktionen um eine Zahl in einzelne Ziffern zu zerlegen. Sobald jede Stelle feststeht, wird über die Codetabelle das Bitmuster aufgesucht, welches für die Interrupt Funktion an der entsprechenden Stelle im SRAM abgelegt wird.

Achtung: Anders als bei der weiter oben gezeigten Variante wurde die Codetabelle ohne Padding Bytes angelegt. Dadurch ist es auch nicht notwendig derartige Padding Bytes in der Programmierung zu berücksichtigen.

Der Rest ist wieder die übliche Portinitialisierung, Timerinitialisierung und eine einfache Anwendung, indem ein 16 Bit Zähler laufend erhöht und über die Funktion out_number ausgegeben

wird. Wie schon im ersten Beispiel, wurde auch hier kein Aufwand getrieben: Zähler um 1 erhöhen und mit Warteschleifen eine gewisse Verzögerungszeit einhalten. In einer realen Applikation wird man das natürlich nicht so machen, sondern ebenfalls einen Timer für diesen Teilaspekt der Aufgabenstellung einsetzen.

Weiterhin ist auch noch interessant. Die Overflow Interrupt Funktion ist wieder so ausgelegt, dass sie völlig transparent zum restlichen Programm ablaufen kann. Dies bedeutet, dass alle verwendeten Register beim Aufruf der Interrupt Funktion gesichert und beim Verlassen wiederhergestellt werden. Dadurch ist man auf der absolut sicheren Seite, hat aber den Nachteil etwas Rechenzeit für manchmal unnötige Sicherungs- und Aufräumarbeiten zu 'verschwenden'. Stehen genug freie Register zur Verfügung, dann wird man natürlich diesen Aufwand nicht treiben, sondern ein paar Register ausschließlich für die Zwecke der Behandlung der 7-Segment Anzeige abstellen und sich damit den Aufwand der Registersicherung sparen (mit Ausnahme von **SREG** natürlich!).

```
.include "m8def.inc"

.def temp  = r16
.def temp1 = r17
.def temp2 = r18

.org 0x0000
           rjmp    main                ; Reset Handler
.org OVF0addr
           rjmp    multiplex

;
;*******************************************************************
; Die Multiplexfunktion
;
; Aufgabe dieser Funktion ist es, bei jedem Durchlauf eine andere Stelle
; der 7-Segmentanzeige zu aktivieren und das dort vorgesehene Muster
; auszugeben
; Die Funktion wird regelmässig in einem Timer Interrupt aufgerufen
;
; verwendet werden 6 Bytes im SRAM (siehe .DSEG weiter unten im Programm)
;    NextDigit    Bitmuster für die Aktivierung des nächsten Segments
;    NextSegment  Nummer des nächsten aktiven Segments
;    Segment0     Ausgabemuster für Segment 0
;    Segment1     Ausgabemuster für Segment 1
;    Segment2     Ausgabemuster für Segment 2
;    Segment3     Ausgabemuster für Segment 3
;
; NextSegment ist einfach nur ein Zähler, der bei jedem Aufruf der Funktion
;   um 1 weitergezählt wird und bei 4 wieder auf 0 zurückgestellt wird.
;   er wird benutzt, um ausgehend von der Adresse von Segment0, auf das
;   jeweils als nächstes auszugebende Muster aus Segement0, Segment1,
;   Segment2 oder Segment3 zuzugreifen. Die Adresse von Segment0 wird
;   in den Z-Pointer geladen und NextSegment addiert.
;
; NextDigit enthält das Bitmuster, welches direkt an den Port C ausgegeben
;   wird und den jeweils nächsten Transistor durchschaltet.
;   Dazu enthält NextDigit am Anfang das Bitmuster 0b11111110, welches
;   bei jedem Aufruf um 1 Stelle nach links verschoben wird. Beim nächsten
;   Aufruf findet sich dann 0b11111101 in NextDigit, dann 0b11111011 und
;   zu guter letzt 0b11110111. Wird beim nächsten Schiebevorgang 0b11101111
;   erkannt, dann wird NextDigit auf 0b11111110 zurückgesetzt (und NextSegment
;   auf 0) und das ganze Spiel beginnt beim nächsten Funktionsaufruf wieder
;   von vorne.
```

```
;
; Segment0 .. 3  enthalten die auszugebenden Bitmuster für die Einzelleds
;   der jeweiligen 7-Segment Anzeigen. Diese Muster werden so wie sie sind
;   einfach ausgegeben. Soll eine der Anzeigen etwas bestimmtes anzeigen
;   (zb eine Ziffer), so obliegt es dem Code, der Werte in diese SRAM
;   Zellen schreibt, das dafür zuständige Bitmuster dort zu hinterlassen.
;   Die Multiplexroutine kümmert sich nicht darum, dass diese Bitmuster
;   in irgendeiner Art und Weise sinnvoll (oder was man dafür halten könnte)
;   sind.
;
; veränderte CPU-Register: keine
;
multiplex:
           push    temp                ; Alle verwendeten Register sichern
           push    temp1
           in      temp, SREG
           push    temp
           push    ZL
           push    ZH

           ldi     temp1, 0            ; Die 7 Segment ausschalten
           out     PORTC, temp1

                                       ; Das Muster für die nächste Stelle
ausgeben
                                       ; Dazu zunächst mal berechnen, welches
Segment als
                                       ; nächstest ausgegeben werden muss
           ldi     ZL, LOW( Segment0 )
           ldi     ZH, HIGH( Segment0 )
           lds     temp, NextSegment
           add     ZL, temp
           adc     ZH, temp1

           ld      temp, Z             ; das entsprechende Muster holen und
ausgeben
           out     PORTD, temp

           lds     temp1, NextDigit    ; Und die betreffende Stelle einschalten
           out     PORTC, temp1

           lds     temp, NextSegment
           inc     temp
           sec
           rol     temp1               ; beim nächsten Interrupt kommt reihum
die
           cpi     temp1, 0b11101111   ; nächste Stelle dran.
           brne    multi1
           ldi     temp, 0
           ldi     temp1, 0b11111110

multi1:
           sts     NextSegment, temp
           sts     NextDigit, temp1

           pop     ZH                  ; die gesicherten Register
wiederherstellen
           pop     ZL
           pop     temp
           out     SREG, temp
           pop     temp1
           pop     temp
           reti
;
```

```
;*************************************************************************
; 16 Bit-Zahl aus dem Registerpaar temp (=low), temp1 (=high) ausgeben
; die Zahl muss kleiner als 10000 sein, da die Zehntausenderstelle
; nicht berücksichtigt wird.
; Werden mehr als 4 7-Segmentanzeigen eingesetzt, dann muss dies
; natürlich auch hier berücksichtigt werden
;
out_number:
           push    temp
           push    temp1

           ldi     temp2, -1            ; Die Tausenderstelle bestimmen
_out_tausend:
           inc     temp2
           subi    temp, low(1000)      ; -1000
           sbci    temp1, high(1000)
           brcc    _out_tausend

           ldi     ZL, low(2*Codes)     ; fuer diese Ziffer das Codemuster fuer
           ldi     ZH, high(2*Codes)    ; die Anzeige in der Codetabelle
nachschlagen
           add     ZL, temp2

           lpm
           sts     Segment3, r0         ; und dieses Muster im SRAM ablegen
                                        ; die OvI Routine sorgt dann duer die
Anzeige
           ldi     temp2, 10

_out_hundert:                           ; die Hunderterstelle bestimmen
           dec     temp2
           subi    temp, low(-100)      ; +100
           sbci    temp1, high(-100)
           brcs    _out_hundert

           ldi     ZL, low(2*Codes)     ; wieder in der Codetabelle das
entsprechende
           ldi     ZH, high(2*Codes)    ; Muster nachschlagen
           add     ZL, temp2

           lpm
           sts     Segment2, r0         ; und im SRAM hinterlassen

           ldi     temp2, -1
_out_zehn:                              ; die Zehnerstelle bestimmen
           inc     temp2
           subi    temp, low(10)        ; -10
           sbci    temp1, high(10)
           brcc    _out_zehn

           ldi     ZL, low(2*Codes)     ; wie gehabt: Die Ziffer in der
Codetabelle
           ldi     ZH, high(2*Codes)    ; aufsuchen
           add     ZL, temp2

           lpm
           sts     Segment1, r0         ; und entsprechend im SRAM ablegen

_out_einer:                             ; bleiben noch die Einer
           subi    temp, low(-10)       ; +10
           sbci    temp1, high(-10)

           ldi     ZL, low(2*Codes)     ; ... Codetabelle
           ldi     ZH, high(2*Codes)
```

```
            add     ZL, temp

            lpm
            sts     Segment0, r0         ; und ans SRAm ausgeben

            pop     temp1
            pop     temp

            ret
;
;*************************************************************************
;
main:
            ldi     temp, HIGH(RAMEND)
            out     SPH, temp
            ldi     temp, LOW(RAMEND)  ; Stackpointer initialisieren
            out     SPL, temp
;                                      die Segmenttreiber initialisieren
            ldi     temp, $FF
            out     DDRD, temp
;                                      die Treiber für die einzelnen Stellen
            ldi     temp, $0F
            out     DDRC, temp
;                                      initialisieren der Steuerung für die
;                                      Interrupt Routine
            ldi     temp, 0b11111110
            sts     NextDigit, temp

            ldi     temp, 0
            sts     NextSegment, temp

            ldi     temp, ( 1 << CS01 ) | ( 1 << CS00 )
            out     TCCR0, temp

            ldi     temp, 1 << TOIE0
            out     TIMSK, temp

            sei

            ldi     temp, 0
            ldi     temp1, 0

loop:
            inc     temp
            brne    _loop
            inc     temp1
_loop:
            rcall    out_number

            cpi     temp, low( 4000 )
            brne    wait
            cpi     temp1, high( 4000 )
            brne    wait

            ldi     temp, 0
            ldi     temp1, 0

wait:       ldi     r21, 1
wait0:      ldi     r22, 0
wait1:      ldi     r23, 0
wait2:      dec     r23
            brne    wait2
            dec     r22
            brne    wait1
```

```
            dec     r21
            brne    wait0

            rjmp    loop

Codes:
    .db  0b11000000, 0b11111001     ; 0: a, b, c, d, e, f
                                    ; 1: b, c
    .db  0b10100100, 0b10110000     ; 2: a, b, d, e, g
                                    ; 3: a, b, c, d, g
    .db  0b10011001, 0b10010010     ; 4: b, c, f, g
                                    ; 5: a, c, d, f, g
    .db  0b10000010, 0b11111000     ; 6: a, c, d, e, f, g
                                    ; 7: a, b, c
    .db  0b10000000, 0b10010000     ; 8: a, b, c, d, e, f, g
                                    ; 9: a, b, c, d, f, g

            .DSEG
NextDigit:   .byte 1         ; Bitmuster für die Aktivierung des nächsten
Segments
NextSegment: .byte 1         ; Nummer des nächsten aktiven Segments
Segment0:    .byte 1         ; Ausgabemuster für Segment 0
Segment1:    .byte 1         ; Ausgabemuster für Segment 1
Segment2:    .byte 1         ; Ausgabemuster für Segment 2
Segment3:    .byte 1         ; Ausgabemuster für Segment 3
```

Forenbeiträge

- Geisterleuchten bei 7-Segment Anzeige
- Re: Zehn 7Segmente im Multiplexbetrieb : Hilfe ! - Ausführliche Erklärung wie und warum man entweder die **Stellen** oder die **Segmente** multiplext.

AVR-Tutorial: Servo

Die Seite ist noch im Entstehen und, bis sie einigermaßen vollständig ist, noch kein Teil des Tutorials

Inhaltsverzeichnis

- 1 Vorbemerkung
- 2 Allgemeines über Servos
- 3 Stromversorgung
- 4 Das Servo-Impulstelegram
- 5 Programmierung
  - 5.1 einfache Servoansteuerung mittels Warteschleifen
  - 5.2 einfache Servoansteuerung mittels Timer
  - 5.3 Ansteuerung mehrerer Servos mittels Timer

Vorbemerkung

Dieses Tutorial überschneidet sich inhaltlich mit dem Artikel Modellbauservo_Ansteuerung. Wir beschreiben hier die grundsätzliche Vorgehensweise zur Ansteuerung von Servos und geben ein Beispiel in Assembler, während der genannte Artikel ein in C formuliertes Beispiel zur Ansteuerung mehrerer Servos liefert.

Allgemeines über Servos

Stromversorgung

Werden Servos an einem µC betrieben, so ist es am Besten, sie aus einer eigenen Stromquelle (Akku) zu betreiben. Manche Servos erzeugen kleine Störungen auf der Versorgungsspannung, die einen µC durchaus zum Abstürzen bringen können. Muss man Servos gemeinsam mit einem µC von derselben Stromquelle betreiben, so sollte man sich gleich darauf einrichten, diesen Störimpulsen mit Kondensatoren zu Leibe rücken zu müssen. Unter Umständen ist hier auch eine Mischung aus kleinen, schnellen Kondensatoren (100nF) und etwas größeren, aber dafür auch langsameren Kondensatoren (einige µF) notwendig.

Die eindeutig beste Option ist es aber, die Servos strommäßig vom µC zu entkoppeln und ihnen ihre eigene Stromquelle zu geben. Servos sind nicht besonders heikel. Auch im Modellbau müssen sie mit unterschiedlichen Spannungen zurechtkommen, bedingt durch die dort übliche Versorgung aus

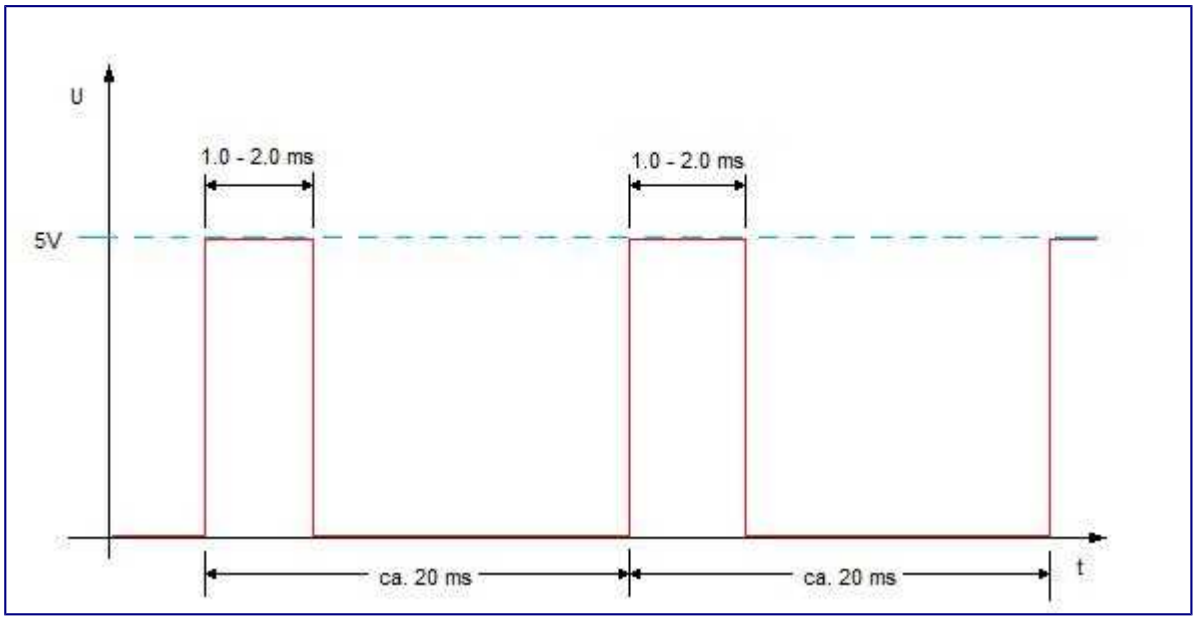
Akkus, die im Laufe der Betriebszeit des Modells natürlich durch die Entladung ihre Voltzahl immer weiter reduzieren. Im Modellbau werden Akkus mit 4 oder 5 Zellen verwendet, sodass Servos mit Spannungen von ca. 4V bis hinauf zu ca. 6V zurecht kommen müssen, wobei randvolle Akkus diese 6V schon auch mal überschreiten können. Bei sinkenden Spannungslage verlieren Servos naturgemäß etwas an Kraft bzw. werden in ihrer Stellgeschwindigkeit unter Umständen langsamer.

Die Servos werden dann nur mit ihrer Masseleitung und natürlich mit ihrer Impulsleitung mit dem µC verbunden.

Das Servo-Impulstelegram

Das Signal, das an den Servo geschickt wird, hat eine Länge von ungefähr 20ms. Diese 20ms sind nicht besonders kritisch und sind ein Überbleibsel von der Technik mit der mehrere Kanäle über die Funkstrecke einer Fernsteuerung übertragen werden. Für das Servo wichtig ist die Impulsdauer in der ersten Phase eines Servosignals. Nominell ist dieser Impuls zwischen 1ms und 2ms lang. Wobei das jeweils die Endstellungen des Servos sind, an denen es noch nicht mechanisch begrenzt wird. Eine Pulslänge von 1.5ms wäre dann Servomittelstellung. Für die Positionsauswertung des Servos haben die 20ms Wiederholdauer keine besondere Bedeutung, sieht man einmal davon ab, dass ein Servo bei kürzeren Zeiten entsprechend öfter Positionsimpulse bekommt und daher auch öfter die Position gegebenenfalls korrigiert, was möglicherweise in einem etwas höheren Stromverbrauch resultiert.

Umgekehrt lässt sich definitiv Strom sparen, indem die Pulse ganz ausgesetzt werden: Der Servo bleibt in der Position, in der er sich gerade befindet - korrigiert sich aber auch nicht mehr. Kommen die Impulse selten, also z.B. alle 50ms, läuft der Servo langsamer in seine Zielposition (praktische Erfahrungen, vermutlich nirgends spezifiziert). Dieses Verhalten lässt sich nutzen, um die manchmal unerwünschten ruckartigen Bewegungen eines Servos abzumildern.



Servo Impulsdiagramm

Den meisten Servos macht es nichts aus, wenn die Länge des Servoprotokolls anstelle von 20ms auf

zb 10ms verkürzt wird. Bei der Generierung des Servosignals muss man daher den 20ms keine besondere Beachtung schenken. Eine kleine Pause nach dem eigentlichen Positionssignal reicht in den meisten Fällen aus und es spielt keine allzugroße Rolle, wie lange diese Pause tatsächlich ist. Generiert man das Imulsdiagramm zb. mit einem Timer, so orientiert man sich daher daran, dass man den 1.0 - 2.0ms Puls gut generieren kann und nicht an den 20ms.

Reale Servos haben allerdings in den Endstellungen noch Reserven, so dass man bei vielen Servos auch Pulslängen von 0.9 bis 2.1 oder sogar noch kleinere/größere Werte benutzen kann. Allerdings sollte man hier etwas Vorsicht walten lassen. Wenn das Servo unbelastet in einer der Endstellungen deutlich zu 'knurren' anfängt, dann hat man es übertrieben. Das Servo ist an seinen mechanischen Endanschlag gefahren worden und auf Dauer wird das der Motor bzw. das Getriebe nicht aushalten.

Programmierung

einfache Servoansteuerung mittels Warteschleifen

Im folgenden Programm wurden einfache Warteschleifen auf die im Tutorial übliche Taktfrequen von 4Mhz angepasst, so dass sich die typischen Servo-Pulsdauern ergeben. Ein am Port D, beliebiger Pin angeschlossenes Servo dreht damit ständig vor und zurück. Die Servoposition kann durch laden eines Wertes im Bereich 1 bis ca 160 in das Register r18 und anschliessendem Aufruf von servoPuls in einen Puls für ein Servo umgewandelt werden.

```
.include "m8def.inc"

.equ XTAL = 4000000

         rjmp    init

init:
          ldi      r16, HIGH(RAMEND)     ; Stackpointer initialisieren
          out      SPH, r16
          ldi      r16, LOW(RAMEND)
          out      SPL, r16

          ldi r16, 0xFF
          out DDRD, r16

loop:     ldi  r18, 0

loop1:    inc r18
          cpi r18, 160
          breq loop2

          rcall servoPuls

          rjmp loop1

loop2:    dec r18
          cpi r18, 0
          breq loop1

          rcall servoPuls

          rjmp loop2
```

```
servoPuls:
          push r18
          ldi r16, 0xFF         ; Ausgabepin auf 1
          out PORTD, r16

          rcall wait_puls        ; die Wartezeit abwarten

          ldi r16, 0x00         ; Ausgabepin wieder auf 0
          out PORTD, r16

          rcall wait_pause       ; und die Pause hinten nach abwarten
          pop r18
          ret
;
wait_pause:
          ldi r19, 15
w_paus_1: rcall wait_1ms
          dec r19
          brne w_paus_1
          ret
;
wait_1ms: ldi r18, 10           ; 1 Millisekunde warten
w_loop2:  ldi r17, 132          ; Es muessen bei 4 Mhz 4000 Zyklen verbraten
werden
w_loop1:  dec r17               ; die innerste Schleife umfasst 3 Takte und wird
132
          brne w_loop1          ; mal abgearbeitet: 132 * 3 = 396 Takte
          dec r18               ; dazu noch 4 Takte für die äussere Schleife =
400
          brne w_loop2          ; 10 Wiederholungen: 4000 Takte
          ret                   ; der ret ist nicht eingerechnet
;
; r18 muss mit der Anzahl der Widerholungen belegt werden
; vernünftige Werte laufen von 1 bis ca 160
wait_puls:
w_loop4:  ldi r17, 10           ; die variable Zeit abwarten
w_loop3:  dec r17
          brne w_loop3
          dec r18
          brne w_loop4

          rcall wait_1ms         ; und noch 1 Millisekunde drauflegen
          ret
```

Wie meistens gilt auch hier: Warteschleifen sind in der Programmierung nicht erwünscht. Der Prozessor kann in diesen Warteschleifen nichts anderes machen. Etwas ausgeklügeltere Programme, bei denen mehrere Dinge gleichzeitig gemacht werden sollen, sind damit nicht vernünftig realisierbar. Daher sollte die Methode mittels Warteschleifen nur dann benutzt werden, wenn dies nicht benötigt wird, wie zb einem simplen Servotester, bei dem man die Servoposition zb durch Auslesen eines Potis mit dem ADC festlegt.

Ausserdem ist die Berechnung der Warteschleifen auf eine bestimmte Taktfrequenz unangenehm und fehleranfällig :-)

einfache Servoansteuerung mittels Timer

Ansteuerung mit dem 16-Bit Timer 1

Im Prinzip programmiert man sich hier eine Software-PWM. Beginnt der Timer bei 0 zu zählen (nach Overflow Interrupt oder Compare Match Interrupt im CTC-Modus (Clear Timer on Compare)), so setzt man den gewünschten Ausgangspin auf 1. Ein Compare Match Register wird so mit einem berechneten Wert versorgt, daß es nach der gewünschten Pulszeit einen Interrupt auslöst. In der zugehörigen Interrupt Routine wird der Pin dann wieder auf 0 gesetzt.

Auch hier wieder: Der Vorteiler des Timers wird so eingestellt, dass man die Pulszeit gut mit dem Compare Match erreichen kann, die nachfolgende Pause, bis der Timer dann seinen Overflow hat (oder den CTC Clear macht) ist von untergeordneter Bedeutung. Man nimmt was vom Zählbereich des Timers übrig bleibt. Beispiel:

- F_CPU = 16 MHz
- Vorteiler = 8
- 16 Bit Registerbreite = Overflow nach 2^16 Zyklen
- CTC-Modus *aus*, steigende Flanke durch Overflow-Interrupt erzeugen
- Overflow-Interrupt: 16 MHz / 2^16 / 8 = 30,52 Hz, 32,8 ms, Ausgangspin auf 1 setzen
- Servostellung links .. rechts = Pulsbreite 1 .. 2 ms = 1 s / 16000000 * 8 * OCR1; OCR1 = 2000 .. 4000
- Output-Compare-Match-Interrupt: Ausgangspin auf 0 setzen

Alternativ zum Overflow-Interrupt könnte, um eine Gesamtpulsbreite von exakt 20 ms zu erreichen, z.B. im Output-Compare-Match-Interrupt (sofern nur einer pro Timer verfügbar ist) im Wechsel

- der Ausgangspin auf 1 und OCR1 auf 1000 .. 2000 gesetzt sowie das CTC-Bit gelöscht werden
- der Ausgangspin auf 0 und OCR1 auf 40000 gesetzt sowie das CTC-Bit gesetzt werden

Neuere ATmegas verfügen über zwei Output-Compare-Register (Suffix *A* und *B*) ebenso wie zwei OCR-Interrupt-Vektoren pro Timer, wobei das CTC-Bit nur bei einem Match mit dem Output-Compare-Register *A* wirkt. Damit können für die Pulserzeugung zwei getrennte Interrupt-Handler angelegt werden, und ein Umbelegen von OCR und CTC-Bit entfällt.

Ansteuerung mit dem 8-Bit Timer 2

Mit einem 8 Bit Timer ist es gar nicht so einfach, sowohl die Zeiten für den Servopuls als auch die für die Pause danach unter einen Hut zu bringen. Abhilfe schafft ein Trick.

Der Timer wird so eingestellt, dass sich der Servopuls gut erzeugen lässt. Dazu wird der Timer in den CTC Modus gestellt und das zugehörige Vergleichsregister so eingestellt, dass sich die entsprechenden Interrupts zeitlich so ergeben, wie es für einen Puls benötigt wird. In einem Aufruf des Interrupts wird der Ausgangspin für das Servo auf 1 gestellt, im nächsten wird er wieder auf 0 gestellt. Die kleine Pause bis zum nächsten Servoimpuls wird so erzeugt, dass eine gewisse Anzahl an Interrupt Aufrufen einfach nichts gemacht wird. Ähnlich wie bei einer PWM wird also auch hier wieder ein Zähler installiert, der die Anzahl der Interrupt Aufrufe mitzählt und immer wieder auf 0

zurückgestellt wird.

Die eigentliche Servoposition steht im Register OCR2. Rein rechnerisch beträgt ihr Wertebereich:

1ms 4000000 / 64 / 1000 OCR2 = 62.5

2ms 4000000 / 64 / 500 OCR2 = 125

mit einer Mittelstellung von (62.5 + 125) / 2 = 93.75

```
.include "m16def.inc"

.equ XTAL = 4000000

          rjmp    init

.org OC2addr
          rjmp    Compare_vect

init:
          ldi      r16, HIGH(RAMEND)     ; Stackpointer initialisieren
          out      SPH, r16
          ldi      r16, LOW(RAMEND)
          out      SPL, r16

          ldi  r16, 0x80
          out  DDRB, r16                ; Servo Ausgangspin -> Output

          ldi  r17, 0                   ; Software-Zähler

          ldi  r16, 120
          out  OCR2, r16                ; OCR2 ist der Servowert

          ldi  r16, 1<<OCIE2
          out  TIMSK, r16

          ldi  r16, (1<<WGM21) | (1<<CS22) ; CTC, Prescaler: 64
          out  TCCR2, r16

          sei

main:
          rjmp main

Compare_vect:
          in   r18, SREG
          inc  r17
          cpi  r17, 1
          breq PulsOn
          cpi  r17, 2
          breq PulsOff
          cpi  r17, 10
          brne return
          ldi  r17, 0
return:   out  SREG, r18
          reti

PulsOn:   sbi  PORTB, 0
          rjmp return

PulsOff:  cbi  PORTB, 0
          rjmp return
```

AVR-Tutorial: Watchdog

Dieser Artikel ist im Entstehen: Die Diskussion wird in [1] geführt.

Der Watchdog im AVR (WDT) ist ein spezieller Timer, der nach Ablauf (typisch ein paar ms) automatisch einen RESET auslöst. Im Normalbetrieb wird der Watchdog in der Hauptschleife des Programms regelmäßig zurückgesetzt. Wenn durch einen Fehler dieses Zurücksetzen nicht mehr stattfindet, läuft der Watchdog-Timer ab und löst einen RESET aus, um den Mikrocontroller und damit das Programm neu zu starten.

Inhaltsverzeichnis

- 1 Anwendung
- 2 Steuerung
- 3 Beispiel
  - 3.1 WDT durch WDTON-Fuse aktivieren
  - 3.2 WDT durch Software aktivieren/deaktivieren
- 4 Tipps & Tricks
  - 4.1 WDT nach einem Reset
  - 4.2 WDT gezielt zum Reset verwenden
  - 4.3 Resetquelle auswerten
  - 4.4 Aufwecken aus einem Sleep Mode
  - 4.5 WDTON Fuse zurücksetzen
- 5 Weblinks

Anwendung

(nach Ganssle-03)

Der Watchdog ist im Prinzip zur Stelle, wenn kein Anwender da ist, um den Resetknopf zu drücken.

Der Watchdog bringt dabei das System aus einem unvorhergesehenen Fehlerzustand wieder in einen betriebsbereiten Zustand.

Dieser Zustand nach einem WDT Reset kann je nach Implementierung im Programm sein:

- Debugzustand
- Sicherheitszustand
- Betriebszustand

Den Debugzustand kann man während der Entwicklung nutzen, um unvorhergesehene Ereignisse herauszufinden. Im fertigen System sollten diese durch das Debuggen bekannten Ereignisse korrekt,

d.h. nicht über WDT behandelt werden.

Den Sicherheitszustand kann man verwenden, wenn das System aufgrund von Hardwareproblemen den WDT ausgelöst hat. Nach dem Reset durch den WDT wird die Resetquelle (normaler Reset oder WDT Reset?) ausgewertet und das System/die Hardware geprüft und ggf. in eine sichere Konfiguration statt in den normalen Betrieb gebracht.

Der normale Betriebszustand ist im Prinzip ein Sonderfall des Sicherheitszustands. Es ist zwar ein unerwartetes Ereignis eingetreten (z. B. einzelner zufälliger Speicherlesefehler), aber ein Neustart des Programms scheint nach einer Neuinitialisierung möglich. Ein Sonderfall ist die Anwendung des WDT zum bewussten Reset (s. Tipps & Tricks).

Steuerung

Der WDT wird durch das Watchdog Timer Control Register WDTCR gesteuert.

WDTCR

—	—	—	WDCE[1]	WDE[2]	WDP2[3]	WDP1[3]	WDP0[3]
7	6	5	4	3	2	1	0

1.

- Watch Dog Change Enable; heißt bei ATmega16/32: WDTOE – Watchdog Turn Off Enable
- Watch Dog Enable

3. Watch Dog Timer Prescaler, bit x

WDP2	WDP1	WDP0	Time-out nach [ms]
0	0	0	16,3
0	0	1	32,5
0	1	0	65
0	1	1	130
1	0	0	260
1	0	1	520
1	1	0	1100
1	1	1	2100

Achtung

Die Zeiten sind abhängig von der Betriebspannung und der Taktfrequenz des WDT-Oszillators. Sie sollten daher aus dem jeweiligen Datenblatt des µCs entnommen werden.

Beispiel

WDT durch WDTON-Fuse aktivieren

Am Einfachsten läßt es sich durch ein kleines Programmbeispiel demonstrieren.

Ein ATmega8 wird mit 4 MHz des internen Taktgenerators mit einer Startup-Zeit von 64 ms getaktet. Die WDTON-Fuse ist gesetzt (WDT aktiviert). An Port B ist eine LED angeschlossen (Pin egal).

```
.include "m8def.inc"

; ATMega8L mit internen 4 MHz getaktet + 64 ms Startuptime
; WDTON aktiviert!

.def Temp1 = R16
.def SubCount = R17

.org 0x0000
        rjmp    Reset               ; Reset Handler
.org OC1Aaddr
        rjmp    timer1_compare      ; Timer Compare Handler

Reset:
        ldi     Temp1, HIGH(RAMEND)
        out     SPH, Temp1
        ldi     Temp1, LOW(RAMEND)  ; Stackpointer initialisieren
        out     SPL, Temp1

        ldi     Temp1, 0xFF         ; Port B auf Ausgang
        out     DDRB, Temp1

        ldi     Temp1, high(40000 - 1)
        out     OCR1AH, temp1
        ldi     Temp1, low(40000 - 1)
        out     OCR1AL, temp1
                                    ; CTC Modus einschalten
                                    ; Vorteiler auf 1
        ldi     Temp1, ( 1 << WGM12 ) | ( 1 << CS10 )
        out     TCCR1B, temp1

        ldi     Temp1, 1 << OCIE1A  ; OCIE1A: Interrupt bei Timer Compare
        out     TIMSK, temp1

        ; kann auch weggelasen werden, da nach einem Reset das Register
        ; auf 0x00 steht, WDT Reset nach 16ms
        ldi Temp1, (0<<WDCE)|(0<<WDE)|(0<<WDP2)|(0<<WDP1)|(0<<WDP0)
        out WDTCR, Temp1

        sei

Mainloop:
        rjmp    Mainloop

timer1_compare:                     ; Timer 1 Output Compare Handler

        ;** findet 100 x pro  Sekunde statt (10 ms)
        wdr                         ; Watch-Dog-Reset

        inc     SubCount            ; Wenn dies nicht der 50. Interrupt
        cpi     SubCount, 50        ; ist, dann passiert gar nichts
        brne    exit_isr

        ;** findet 2 x pro  Sekunde statt (500ms)
        clr     SubCount

        ;** Port B negieren
        in      Temp1, PinB
```

```
        com     Temp1
        out     PortB, Temp1

exit_isr:
        reti                        ; das wars. Interrupt ist fertig
```

Der Timer 1 läuft im CTC-Modus mit einer Frequenz von 100 Hz (10 ms). Durch den Soft-Subcounter wird die Frequenz auf 2 Hz geteilt und jeweils nach 500 ms das Port B negiert.

Da die LED in diesem Beispiel nach 500 ms jeweils ein- und ausgeschaltet wird, blinkt sie mit einer Frequenz von 1 Hz. Der WDT wird nach 10 ms zurückgesetzt, so dass er keinen RESET auslösen kann.

Wird jetzt der Befehl WDR auskommentiert, führt der WDT nach 16 ms einen RESET aus. Die LED blinkt nun, bedingt durch die Startup-Zeit von 64 ms und einem Time-out von 16ms, mit rund 6 Hz. 1/(64ms + 16ms) ~ 12 Hz (halbe Periode)

WDT durch Software aktivieren/deaktivieren

Der WDT läßt sich auch softwaremäßig durch Setzen des WDE-Bits im WDTCR Register aktivieren.

```
WDT_on:

  in Temp1, WDTCR      ; Write logical one to WDE
  ori Temp1, (1<<WDE)
  out WDTCR, Temp1
  ret
```

Dieses hat den Vorteil, dass man den WDT auch softwaremäßig wieder deaktivieren kann.

Ein Deaktivieren des WDTs ist nicht möglich, wenn die WDTON - Fuse gesetzt ist!

Das softwaremäßige Deaktivieren verlangt allerdings eine besondere Deaktivierungssequenz. (in drei Phasen)

```
WDT_off:

  ;** 1. Phase
  wdr                   ; reset WDT

  ;** 2. Phase
  in  Temp1, WDTCR      ; Write logical one to WDCE and WDE
  ori Temp1, (1<<WDCE)|(1<<WDE)

  in  Temp2, SREG       ; save I Flag
  cli                   ; we have only 5 cycles to reset WDE

  out WDTCR, Temp1

  ;** 3. Phase
  ldi Temp1, (0<<WDE)   ; Turn off WDT
  out WDTCR, Temp1

  out SREG, Temp2       ; restore I Flag
  ret
```

Wenn WDCE und WDE nicht in einem Zug vor dem Deaktivieren auf 1 gesetzt werden, hat das Rücksetzen des WDE-Bits keine Wirkung und der WDT läuft munter weiter!

Dazu hat man maximal 5 Takte Zeit. Diese Sequenz darf auch nicht durch einen Interrupt unterbrochen werden.

Tipps & Tricks

Wem die Timer im AVR nicht reichen kann den WD als einen zusätzlichen Timer verwenden. In den neueren AVRs hat der WDT einen eigenen Interruptvector. Dazu den WDT im Interruptmode einstellen. Der AVR springt dann bei abgelaufenem WD-Timer durch den Vector die ISR an. Dort kann man entscheiden ob ein RESET durchgeführt wird oder ob man nur irgendwelche anderen Operationen ausführt. Besonders wenn lange Timerzeiten benötigt werden (ATtiny2313 eine WDT-Periode bis zu 8sek.) ist der WDT interessant. Insbesondere wird das ganze System nicht weiter belastet, da der WDT von einem eigenen Oszillator getaktet wird.

Nötige Einstellungen:

WDE=0 WDIE=1 WDTON=1

WDT nach einem Reset

Der WDT bleibt bei manchen AVRs nach einem Reset (ob durch den Watchdog, extern oder aus sonstigen Gründen, also auch über das Flashen einer neuen Software hinweg!) aktiv, wenn er einmal an war. Er läuft danach mit der kürzesten Zeit weiter, da die Prescaler Bits beim Reset gelöscht werden und somit die Watchdog Zeit auf die kürzeste Zeit gesetzt wird. Das kann zu unerwarteten Problemen führen ([2])

Dies steht nicht explizit im Datenblatt, sondern man kann es nur anhand der Defaultwerte der Bits entnehmen, was viele übersehen. Dies ist vor allem beim Programmieren in einer Hochsprache wie C wichtig, denn da verwendet man meist Makros für den Watchdog und kommt somit nicht direkt mit den Registern aus dem Datenblatt in Berührung. Weiterhin dauert die Initialisierung der globalen Variablen vor dem Start der main Funktion oft länger als die Watchdog Periode, was dazu führt, dass die main Funktion nie erreicht wird. Der Watchdog Timer muss daher vorher abgeschaltet werden, was beim gcc über Code in einer speziellen Section geschieht, die unmittelbar nach dem Reset ausgeführt wird. Ein Beispiel findet sich dazu in folgendem Thread: [3]

WDT gezielt zum Reset verwenden

Man kann den WDT auch verwenden, um gezielt per Software einen Reset des AVR auszulösen. Das wird z. B. im AVR Bootloader FastBoot von Peter Dannegger gemacht.

Variante1: Ohne Unterbrechung des Programmablaufs über eine Statusvariable oder Statusflag, z.B:

```
tst rFlag
brne PC+2
wdr
```

Um ein WDT-Reset auszulösen wird das Register rFlag an beliebiger Stelle auf ein Wert ungleich Null gesetzt.

Variante 2: Programm anhalten und auf WDT-Reset warten

```
rjmp PC
```

Resetquelle auswerten

Nach einem WDT-Reset wird die gleiche Adresse (0x0000) angesprungen, wie nach einem normalen Reset. Zur Unterscheidung der Reset-Quelle ist eine Auswertung des WDRF-Flags im MCU Control -und Statusregister erforderlich.

```
in Temp1,MCUCSR                 ;MCU Control- und Statusregister lesen
sbrs Temp1,WDRF                 ;Prüfen ob Watchdog-Restart erfolgte
rjmp normalReset                ;nein: Normalstart
cbr Temp1,1<<WDRF               ;Watchdog-Resetflag zurücksetzen
out MCUCSR, Temp1
rjmp watchDogReset              ;WDRStart
```

Aufwecken aus einem Sleep Mode

Der WDT kann auch verwendet werden, um einen AVR im Rahmen der möglichen WDT-Zeiten zeitgesteuert aus einem Sleep Mode aufzuwecken. Allerdings verbraucht der eingeschaltete WDT einen gewissen Strom [4]. Beispiel in C: Pollin Funk-AVR-Evaluationsboard: Pennen bis der Hund bellt.

WDTON Fuse zurücksetzen

Die Änderung der WDTON Fuse wird erst nach einer Spannungsunterbrechung wirksam. Ein Reset ist nicht ausreichend. [5]

Weblinks

- Li'l Bow Wow By Jack Ganssle, Embedded Systems Design (engl.)
- Watchdog Reset Flag auswerten Erklärung, wie das WDRF unter C abgefragt wird

AVR-Tutorial: Power Management

ACHTUNG! Dieser Artikel befindet sich noch im Aufbau!

Vorallem in batteriebetriebenen Systemen spielt die Leistungsaufnahme eine wichtige Rolle, d.h. sie soll so niedrig wie möglich gehalten werden um eine lange Laufzeit zu erreichen. Den sparsamen Umgang mit der verfügbaren el. Ladung nennt man **Power Management** (dt. Energiesparen).

Im Rahmen des Power Managements stehen uns beispielsweise die Sleep-Modi zur Verfügung, mit denen wir bestimmte Module zeitweise deaktivieren können. Andere garnicht genutzte Module können wir durch entsprechende Konfiguration (z.B. in den Fuses) auch komplett deaktivieren.

Inhaltsverzeichnis

- 1 Theorie
  - 1.1 Sleep Modi
  - 1.2 Manuelles Deaktivieren
- 2 Praxis
  - 2.1 Assembler
  - 2.2 C
- 3 Quellen

Theorie

Der Stromverbrauch einer CMOS-Schaltung (was auch Mikrocontroller sind) hängt letztlich nur davon ab, wie viele Gates bei welcher Spannung umgeladen werden. Zur Reduktion steht demnach zur Verfügung:

- Reduktion der Umladevorgänge pro Sekunde, also Reduktion der Taktfrequenz
- Reduktion der beteiligten Kapazitäten, also Abschalten nicht benutzter Funktionsblöcke
- Reduktion der Speisespannung (bei manchen Controllern ist das die *Kernspannung*)

Den statischen Reststrom kann man heutzutage vernachlässigen, auch wenn an dieser Stelle die Anmerkung fallen muss, dass AVRs da eher schlecht abschneiden. Hingegen muss Strom durch ohmsche Lasten, etwa Pull-Up-Widerstände oder LEDs, unbedingt beachtet werden! Außerdem dürfen keine offenen (CMOS!-)Eingänge „herumfliegen“ sondern müssen allesamt an definiertem Potenzial liegen. In der Regel können Mikrocontroller unbenutzte Eingänge im Schlafmodus oder Analogeingangsbetrieb automatisch abtrennen.

Grundsätzlich sollte für Stromsparbetrieb die niedrigstmögliche Taktfrequenz gewählt werden, die möglich ist. Übliche Mikrocontroller können direkt von einem Uhrenquarz mit 32,768 kHz gespeist

werden. Macht natürlich auch nur maximal 32768 Befehle pro Sekunde. Die Stromaufnahme ist dabei sehr gering und im Gegensatz zu Sleep-Mode-Tricksereien ohne hohe Spitzen.

Da das Antwortverhalten bei derart geringen Taktfrequenzen leidet, wird man eine höhere Frequenz wählen und den Prozessor bei Nicht-Bedarf schlafen schicken. Die Strom-Spitzen sind dementsprechend höher, die Gesamtstromaufnahme bleibt jedoch gering.

Viele Mikrocontroller können einzelne Funktionsblöcke abschalten. Bei den AVRs schlägt da vor allem der 16-Bit-Timer 1 zu Buche. Aber auch die Referenzspannungsquelle für den A/D-Wandler, Analogvergleicher oder den Brown-Out-Detektor kann man abschalten.

Für die Kontrolle der Stromaufnahme aus einer Batterie ist ein Digitalmultimeter mit vielen Strommessbereichen bis 20 µA (Auflösung 10 nA) vorteilhaft. Außerdem ein Plättchen einer doppelseitigen Leiterplatte mit dünnen Drähtchen, das man unter die Batteriekontaktierung schieben kann. Weicht die Stromaufnahme vom Datenblatt ab, hat man wohl noch irgendetwas vergessen.

Sleep Modi

Welche Sleep-Modi es gibt, hängt vom verwendeten µC ab, dieser Artikel nimmt jedoch Bezug auf den ATmega32. Um einen der verfügbaren Sleep-Modi des ATmega32 zu betreten müssen folgende Schritte ausgeführt werden

1. Das SE-Bit im MCUCR-Register wird auf 1 gesetzt
2. Die SMx-Bits im MCUCR-Register je nach gewünschtem Modus setzen
3. Der SLEEP-Befehl wird ausgeführt

Der Mikrocontroller geht dann sofort in den SLEEP-Modus, d.h. noch vor eventuell anstehenden Interrupts, und wacht erst wieder auf wenn ein Signal eines geeigneten Moduls (je nach Modus) ihn aufweckt.

Die Arbeit wird dann mit der ersten Anweisung hinter dem SLEEP-Befehl wieder aufgenommen.

MCUCR - MCU Control Register

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

Bit 7 - SE
Sleep Enable
Mit diesem Bit wird bestimmt ob der Sleep-Befehl ausgeführt wird (1) oder nicht (0).

Bit 6..4 - SM2..0
Sleep Mode Select
Mit diesen drei Bits wird der gewünschte Sleep-Modus gewählt

SM2	**SM1**	**SM0**	**Sleep Modus**
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved

1	0	1	Reserved
1	1	0	Standby[1]
1	1	1	Extended Standby[1]

[1] Nur verfügbar mit externem Taktgeber

Modi Übersicht

Generell ist der Modus zu wählen, der die meisten nicht benötigten Module abschaltet.

	Aktive Takte					Aktive Oszillatoren					Weckquell
Sleep Modus	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Haupttaktgeber	Timer Oszillator	INT2 INT1 INT0	TWI Address Match	Timer2	SPM/ ROM Read
Idle			x	x	x	x	x[2]	x	x	x	x
ADC Noise Reduction				x	x	x	x[2]	x[3]	x	x	x
Power-down								x[3]	x		
Power-save					x[2]		x[2]	x[3]	x	x[2]	
Standby[1]						x		x[3]	x		
Extended Standby[1]					x[2]	x	x[2]	x[3]	x	x[2]	

[1] Nur verfügbar bei externer Taktquelle

[2] Wenn AS2-Bit in ASSR-Register gesetzt

[3] Nur INT2 oder Level Interrupt INT1 und INT0

Manuelles Deaktivieren

Einzelne Module können auch manuell deaktviert werden um Strom zu sparen, das bietet sich vorallem an wenn bestimmte Module im gegebenen Projekt generell nicht benötigt werden und damit deaktiviert werden können.

Analog to Digital Converter

Der ADC ist standardmäßig aktiviert. Um ihn zu deaktivieren, muss man ADEN (Bit 7) im Register ADCSRA auf Null setzen.

Analog Comparator

Der Analogkomparator ist standardmäßig aktiviert. Um ihn zu deaktivieren, muss man ACD (Bit 7) im Register ACSR setzen.

Brown-Out Detector

Der Brown-Out Detector lässt sich entweder durch das BODEN-Bit in den Fuses oder mit entsprechenden Befehlen aktivieren oder deaktivieren. Das Fuse-Bit ist standardmäßig gesetzt (Achtung: Umgekehrte Logik!) und der BOD damit deaktiviert.

Watchdog

Auch der Watchdog-Timer lässt sich in den Fuses standardmäßig aktivieren/deaktivieren, hier über das WDTON-Bit. Natürlich geht auch das softwareseitig [1]

Praxis

Assembler

Hier mal etwas Code der nicht aufgeräumt ist aber sich gut eignet für Versuche und funktioniert. Verbesserungen folgen.

```
;**********
;* Dateibeschrieb:
;* Interrupttest_Eingängexx.ASM
;* Autor Martin Kuch
;* V02 23.10.2014 mit Sleep und LED Blink in Go raus aus ISR ST:OK!
;* V03 23.10.2014 cli und sei in Go - nur 1x Flanke erkennen nach Schlußdelay
ST: OK!
;* V04 23.10.2014 sbic warten bis alle Taster wieder frei sind ST:OK!
;**********
;* Funtionen:
;* 4 Taster an PB0 ... PB3 zur Eingabesteuerung
;* Interrupt 0B (PinChangeInt)
;* PB4 LED
;**********
;* Hardware Zielschaltung:
;* Test-Board mit ATTiny2313 PU
;* Steckernetzgerät Sek: 6VAC 350mA
;* Netzteil eingenbau: Brückengleichrichter, 330uF, 7805CT, 220uF, 6,8nF 2x, LED
+ 261R
;**********
;* Anschlussschema:
; ATTiny2313
; Port Pin - Pin Funktion
; PB1 12 - Eingang Schalter 1 mit Pullup(+LED) gegen Masse (Set)
; PB2 13 - Eingang Schalter 2 mit Pullup(+LED) gegen Masse (Back)
; PB3 14 - Eingang Schalter 3 " (vor)
; PB4 15 - Eingang Schalter 4 " (zurück)
; PB0 16 - Ausgang Lüfter Ein, LED mit Widerstand gegen +5VDC (Kathode gegen
Ausgang)

;* Controllertyp
.device ATtiny2313   ;für Gasm   .include Headerdatei für at2313.asm bei Gasm
nicht nötig
;* Definitionen
.def temp1 = r16 ;Allzweckregister und für LCD_Routines3
.def temp2 = r17 ;für LCD_Routines
.def temp3 = r18 ;für LCD_Routines
.def TM = r19 ;Allzweckregister Programm
.def TM1 = r20 ;Allzweckregister 2
.def TMz = r21 ;Zähler allg.
.def Ct0 = r24 ;Couter Delay Vorgabe, Übergabewert für Zeitschleife
.def Ct1 = r25 ;Conter Delay intern

;* Flash Programm
.cseg ;Flash Segment
; Interrupt-Handler
.org 0x0000
rjmp Main       ;0x00 Reset
reti;rjmp SInt  ;0x01 INT0 PD2 (6)
```

```
reti;rjmp SInt  ;0x02 INT1 PD3 (7)
reti;rjmp SInt  ;0x03 Tim1_Capt
reti;rjmp SInt  ;0x04 Tim1_CompA
reti;rjmp SInt  ;0x05 Tim1_OVF
reti;rjmp SInt  ;0x06 Tim0_OVF
reti;rjmp SInt  ;0x07 Uart0_rxc
reti;rjmp SInt  ;0x08 Uart0_dre
reti;rjmp SInt  ;0x09 Uart0_txc
reti;rjmp SInt  ;0x0A ANA_Comp
rjmp IntPC      ;0x0B PinChangeInt
reti;rjmp SInt  ;0x0C Tim1_CompB
reti;rjmp SInt  ;0x0D Tim0_CompA
reti;rjmp SInt  ;0x0E Tim0_CompB
reti;rjmp SInt  ;0x0F USI_Start
reti;rjmp SInt  ;0x10 USI_OVERFLOW
reti;rjmp SInt  ;0x11 EE_READ
reti;rjmp SInt  ;0x12 WDT_OVERFLOW
nop

;* Main Programm Endlos Steuerung Taster
Main:
;Stack init
         ldi TM, RAMEND
         out SPL, TM
;*Init
                ldi TM, 0b00010000           ;Port B Bit0...3 => Eingang B4 =>
Ausgang
                out DDRB, TM
                nop
;*Hallo blinken testen, LED-Test und schauen ob uC läuft
        cbi PortB,4     ;LED an
        ldi Ct0, 100
        rcall delayms
        sbi PortB,4     ;LED aus
        rcall delayms
        ldi Ct0, 255

;*Init Interrupt PCINT
                ldi TM, 0b00100000      ;=32 dez.
                out MCUCR, TM           ;Bit5 = EN enable Sleep
                out GIMSK, TM           ;PCIE PC.Int.Enable
                ldi TM, 0b00001111
                out PCMSK, TM           ;Maske Int. Pin zulassen
                sei                                     ;Interrupts zulassen
Go:
        sleep           ;Schlafen legen = Energiesparen
        nop
        cli
        cbi PortB,4     ;LED an
        rcall delayms

Go_1:
        sbis PINB,0     ;Taste noch gedrückt? = 0 => warten
rjmp Go_1
        sbis PINB,1     ;Taste noch gedrückt? = 0 => warten
rjmp Go_1
        sbis PINB,2     ;Taste noch gedrückt? = 0 => warten
rjmp Go_1
        sbis PINB,3     ;Taste noch gedrückt? = 0 => warten
rjmp Go_1

        sbi PortB,4     ;LED aus
;       rcall delayms
        sei             ;Interrupts einschalten
```

```
rjmp Go

;*ISR für PinChangeInt. (0x0B)
IntPC:
;       push TM
;       in TM, SREG
        nop
;       out SREG, TM
;       pop TM
reti

; Pause Count * 5ms
delayms:
         mov Ct1, Ct0
delayms_:
         rcall delay5ms
         dec Ct1
         brne delayms_
         ret

 ; Längere Pause für manche Befehle
delay5ms:                                ; 5ms Pause
           Push temp2
           ldi  temp1, ( XTAL * 5 / 607 ) / 1000
WGLOOP0:   ldi  temp2, $C9
WGLOOP1:   dec  temp2
           brne WGLOOP1
           dec  temp1
           brne WGLOOP0
           pop temp2
           ret                           ; wieder zurück
```

C

Ein simples Testprogramm, um mit sleep modi im AVR zu spielen. Es funktioniert sehr gut mit dem ATmega8 und ist auch auf andere AVRs portierbar. Teilweise sind weitere Modi verfügbar.

Wichtig ist, daß die Interruptroutine für den "Weckruf" vorhanden sein muss. Es müssen nicht zwingend Aktionen in ihr durchgeführt werden.

```
/* ATmega8 with internal 4Mhz clock (6cycle + 64ms) */

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

int main(void)
{
   DDRC |= (1 << PC2) | (1 << PC1); // leds for testing

   DDRD &= ~(1 << PD2); // INT0: input...
   PORTD |= (1 << PD2); // ...with pullup.

   // level interrupt INT0 (low level)
   MCUCR &= ~((1 << ISC01) | (1 << ISC00));

   // infinite main loop
   while (1)
   {
      // trigger leds for testing
      PORTC ^= (1 << PC1);
```

```
      _delay_ms(500);
      PORTC ^= (1 << PC1);

      // enable external interrupt
      GICR |= (1 << INT0);

      // set sleep mode
      set_sleep_mode(SLEEP_MODE_PWR_DOWN);

      // sleep_mode() has a possible race condition
      sleep_enable();
      sei();
      sleep_cpu();
      sleep_disable();

      // waking up...
      // disable external interrupt here, in case the external low pulse is too
long
      GICR &= ~(1 << INT0);

      // disable all interrupts
      cli();
   }
}


ISR(INT0_vect)
{
   // ISR might be empty, but is necessary nonetheless
   PORTC ^= (1 << PC2); // debugging
}
```

Quellen

ATMEL AVR ATmega32 Datenblatt