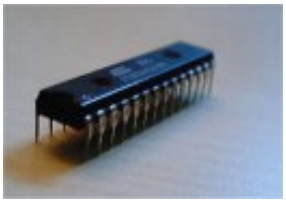
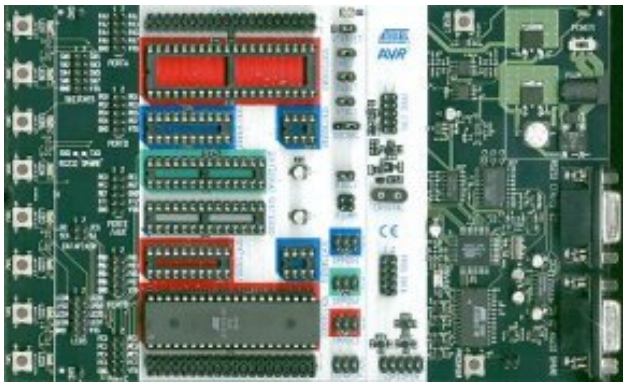
AVR-Tutorial - 1. Benötigte Ausrüstung

1. Hardware

Ein Mikrocontroller alleine ist noch zu nichts nützlich.

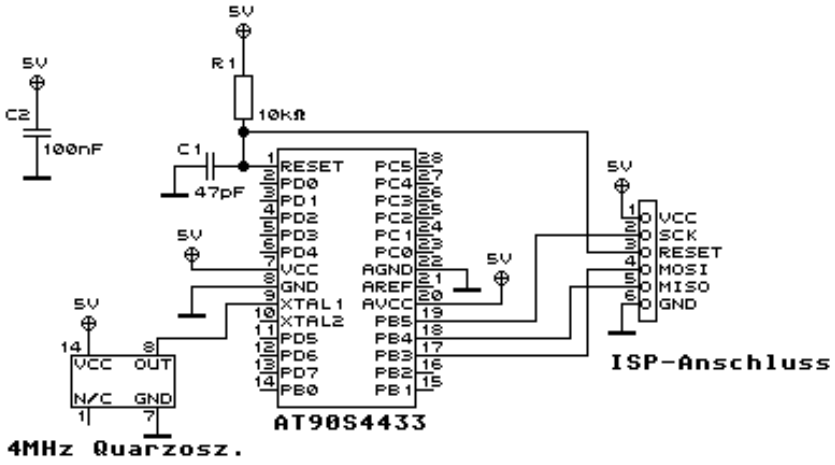




Damit man etwas damit anfangen kann, braucht man eine Schaltung in die der Controller eingesetzt wird. Dazu werden bei Elektronikhändlern Platinen angeboten, die alles nötige (Taster, LEDs, Steckverbinder...) enthalten, wie z.B. das STK500.

Das STK500 kostet ca. 100 Euro. Wer nicht so viel ausgeben möchte, der kann auch selber eine Schaltung aufbauen, die das Programmieren von AVR-Mikrocontrollern ermöglicht. So kompliziert wie das STK500 wird es nicht, es reichen eine Hand voll Bauteile.

Die folgende Schaltung baut man am besten auf einem **Breadboard** (Steckbrett) auf. Solche Breadboards gibt's z.B. bei ELV oder Conrad.



Über den Takteingang **XTAL1** ist der Mikrocontroller mit dem **Quarzoszillator** verbunden, der den benötigten Takt von 4MHz liefert. 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

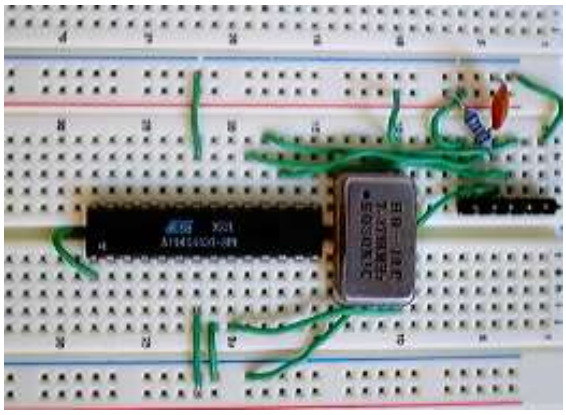
dieser Größe an den gleichen Positionen die Pins Nr. 1, 7, 8 und 14 wären).

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 AT90S4433/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.
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 kommt noch ein 100nF Keramik- oder Folienkondensator, um Störungen in der Versorgungsspannung zu unterdrücken.

Hier die Liste der benötigten Bauteile:

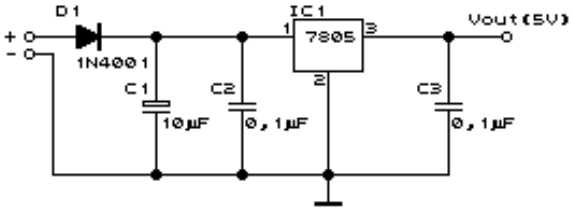
```
R1        Widerstand 10k
C1        Keramikkondensator 47p
C2        Keramik- oder Folienkondensator 100n

          Stiftleiste 6-polig

          Mikrocontroller ATmega8 oder AT90S4433
          (kann auf http://shop.mikrocontroller.net/ bestellt werden)

          Quarzoszillator 4MHz
```

Fertig aufgebaut könnte das etwa so aussehen:



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

Die Versorgungsspannung **Vcc** beträgt 5V und kann z.B. mit folgender Schaltung erzeugt werden:



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

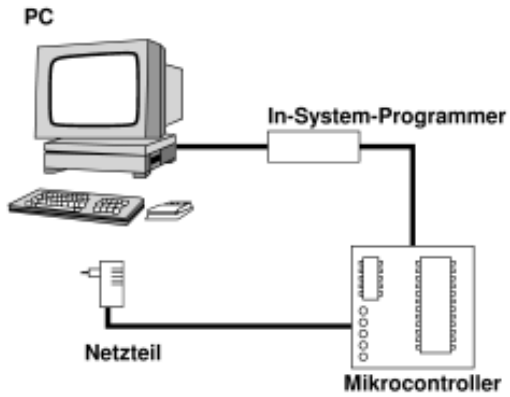
An den Eingang (+ und - im Schaltplan) wird ein Steckernetzteil mit einer Spannung von 9 - 12V.

Dann braucht man nur noch den **ISP-Programmieradapter**, über den man die Programme vom PC in den Controller übertragen kann. Eine Bauanleitung gibt es u.a. auf http://rumil.de/hardware/avrisp.html (allerdings sollte man statt dem im Schaltplan angegebenen 74HC244 einen 74HCT244 nehmen).

> Einen fertigen ISP-Programmer gibt es für 14,00 Euro auf http://shop.mikrocontroller.net/.

Den ISP-Adapter schließt man an den Parallelport an und verbindet ihn mit der Stiftleiste SV1 über ein 6-adriges Kabel (siehe Schaltplan).

So sieht die Anordnung also aus:



Für die anderen Teile des Tutorials sollte man sich noch die folgenden Bauteile besorgen:

```
Teil 2 (I/O-Grundlagen)
=======================
 5 LEDs 5mm
 5 Taster
 5 Widerstände 1k
 5 Widerstände 10k

Teil 4 (LCD-Display)
=======================
 1 Potentiometer 10k
 1 HD44780-kompatibles LCD, z.B. 4x20 oder 2x16 Zeichen

Teil 6 (Das UART)
=======================
 1 Pegelwandler MAX232 oder MAX202
 5 Elektrolytkondensatoren 22µF
 1 9-polige SUBD-Buchse (female)
 1 dazu passendes Modem(nicht Nullmodem!)-Kabel
```

2. Software

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.

Wer in C programmieren möchte, kann den kostenlosen GNU-C-Compiler AVR-GCC ("WinAVR") ausprobieren. In der Artikelsammlung gibt es ein umfangreiches Tutorial zu diesem Compiler; Fragen dazu stellt man am besten hier im GCC-Forum.

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

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

3. Literatur

Bevor man anfängt sollte man sich die folgenden PDF-Dateien runterladen:

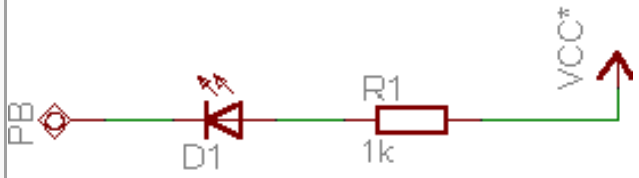
- Datenblatt des ATmega8
- Befehlssatz der AVRs (422kB)

Betrachten kann man PDF-Dateien mit dem Acrobat Reader von Adobe.

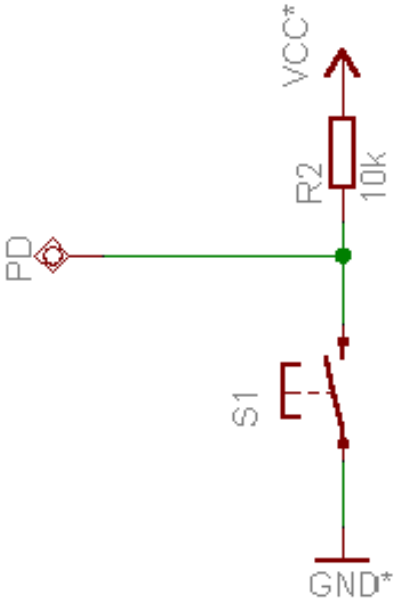
Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)
net

AVR-Tutorial - 2. I/O-Grundlagen

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 1kΩ gegen Vcc (5V) an:



An **PD0-PD5** kommen 4 Taster mit je einem 10kΩ Pullup-Widerstand:



Bevor es losgeht, hier noch ein paar Worte zu den verschiedenen Zahlensystemen.
Binärzahlen werden für den Assembler im Format **0b00111010** schreiben, 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

- Linksammlung
- weitere...
- [Letzte Änderungen]

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.

1. Ausgabe

Assembler-Sourcecode

Download leds.asm

```
.include "4433def.inc"       ;Definitionsdatei einbinden, ggf. durch
                             ;2333def.inc ersetzen

        ldi r16, 0xFF        ;0xFF ins Arbeitsregister r16 laden
        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

Aus diesem Programm möchten wir nun eine Hex-Datei erzeugen, die man in den Controller programmieren kann. Dazu füttern wir das Programm dem Assembler, bei wavrasm z.B. indem wir eine neues Fenster öffnen, den Programmtext hineinkopieren, speichern und auf "assemble" klicken. Wichtig ist, dass sich die Datei "4433def.inc" bzw. "m8def.inc" (wird beim Atmel-Assembler mitgeliefert) im gleichen Verzeichnis wie die Assembler-Datei befindet.

Die bei manchen Versionen des Atmel-Assemblers mitgelieferte 4433def.inc ist fehlerhaft! Eine korrigierte Version könnt ihr hier downloaden. Wer den AT90S2333 verwendet, bekommt die dazugehörige Include-Datei hier.

Beim **ATmega8** ist standardmäßig der interne 1 MHz-Oszillator aktiviert; weil dieser für viele Anwendungen (z. B. UART) 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.

Jetzt 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 4433def.inc eingebunden, welche die Bezeichnungen für die verschiedenen Register definiert. Wenn diese Datei fehlen würde wüsste der Assembler nicht was mit "PORTB", "DDRD" usw. gemeint ist.

In der 2. Zeile wird mit dem Befehl **ldi r16, 0xFF** der Wert 0xFF (entspricht 0b11111111) in das Register r16 geladen. 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.

Die Erklärungen nach dem Zeichen **;** sind Kommentare und werden vom Assembler nicht beachtet!

Der 3. Befehl 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.

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.

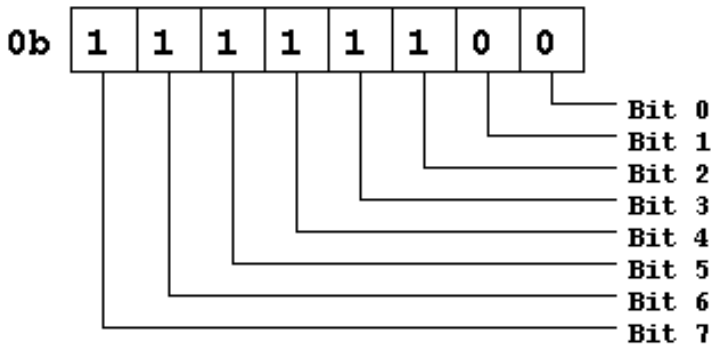
Bei Kopier- und Ladebefehlen (ldi, in, out...) wird immer der 2. Operand in den ersten kopiert:

Befehl	**Aktion**
ldi R16, 15	Die Konstante "15" wird in das Register "R16" geladen.
mov R16, R17	Das Register "R17" wird in das Register "R16" kopiert.
out PORTB, R16	Der Inhalt des Registers "R16" wird in das IO-Register "PORTB" kopiert.
in R16, PIND	Der Inhalt des IO-Registers "PIND" wird in das Register "R16" kopiert.

Wer mehr über die Befehle wissen möchte, sollte sich die PDF-Datei Instruction Set (422kB) 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 *letzten* 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). 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 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.

2. Eingabe

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

Download leds+buttons.asm

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

        ldi r16, 0xFF
        out DDRB, r16      ;Port B durch Ausgabe von 0xFF ins
                           ;Richtungsregister DDRB als Ausgang konfigurieren
        ldi r16, 0x00
        out DDRD, r16      ;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.

3. 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** überspringt den darauffolgenden Befehl, wenn das angegebene Bit 0 (low) ist; **sbis** 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!

Am besten verstehen kann man das natürlich an einem Beispiel:

Download bitaccess.asm

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

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

         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 eine 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.

AVR-Tutorial - 3. Der 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 gelegt Element auch zuerst wieder heruntergenommen wird. Es ist nicht möglich Elemente irgendwo in der Mitte des Stapels rauszuziehen oder reinzuschieben.

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 die momentane 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.

1. 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 beendet ist.

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 SPL, temp

         rcall sub1        ;sub1 aufrufen

loop:    rjmp loop


sub1:
                           ;hier könnten ein paar Befehle stehen
```

- Linksammlung
- weitere...
- [Letzte Änderungen]


```
        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" einsetzten 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 alleine. Deswegen gibt es bei diesen Controllern noch ein Register mit dem Namen **SPH**, in dem 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, LOW(RAMEND)           ; LOW-Byte der obersten RAM-Adresse
        out SPL, temp
        ldi temp, HIGH(RAMEND)          ; HIGH-Byte der obersten RAM-Adresse
        out SPH, 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 macht es keinen Sinn, 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 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 (ROM), 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 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 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.

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. 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.

A propos Program Counter: wer sehen will wie so ein Programm aussieht wenn es assembliert ist sollte mal die Datei mit der Endung ".lst" im Verzeichnis wo sich das Projekt befindet 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 eingerahmten Rest besteht aus Assemblerdirektiven, Labels (Sprungmarkierungen) und Kommentaren, die nicht direkt in OP-Code umgewandelt werden.

2. 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 Reichenfolge) 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 SPL, temp

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

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

        rcall sub                 ; Unterprogramm "sub" aufrufen

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

loop:   rjmp loop                 ; Endlosschleife


sub:
        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.

Weitere Informationen (von Lothar Müller):

- Der Stack - Funktion und Nutzen [pdf]
- Der Stack - Parameterübergabe an Unterprogramme [pdf]
- Der Stack - Unterprogramme mit variabler Parameteranzahl [pdf]

Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)
net

AVR-Tutorial - 4. 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, dass die häufigste Frage in Mikrocontroller-Foren ist: "Wie kann ich ein LCD anschließen?".

Diese Anleitung ist für ein 4x20-LCD geschrieben, sie sollte aber auch mit anderen Displaygrößen (z.B. 2x16) funktionieren.

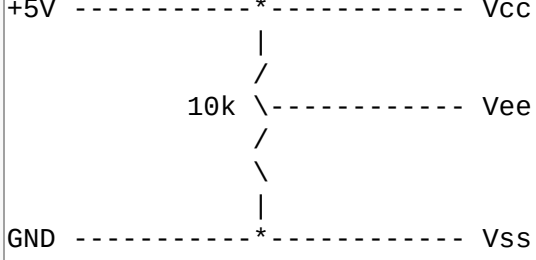
Die meisten Text-LCDs verwenden den Controller **HD44780** oder einen kompatiblen (z.B. KS0070) und haben 14 oder 16 Pins. Die Pinbelegung ist praktisch immer gleich:

1	Vss	GND
2	Vcc	5V
3	Vee	Kontrastspannung (0V bis 5V)
4	RS	Register Select (Befehle/Daten)
5	RW	Read/Write
6	E	Enable
7	DB0	Datenbit 0
8	DB1	Datenbit 1
9	DB2	Datenbit 2
10	DB3	Datenbit 3
11	DB4	Datenbit 4
12	DB5	Datenbit 5
13	DB6	Datenbit 6
14	DB7	Datenbit 7

Achtung: Unbedingt von der richtigen Seite zu zählen anfangen! Meistens ist neben Pin 1 eine kleine 1 auf der LCD-Platine, ansonsten im Datenblatt nachschauen.

Bei LCDs mit 16-poligem Anschluß sind die beiden letzten PINs für die Hintergrundbeleuchtung reserviert.

Vss wird ganz einfach an GND angeschlossen und Vcc an 5V. Vee kann man testweise auch an GND legen. Wenn das LCD dann zu dunkel sein sollte muss man ein 10k-Potentiometer zwischen GND und 5V schalten, mit dem Schleifer an Vee:



Es gibt zwei verschiedene Möglichkeiten zur Ansteuerung eines solchen Displays: den **8bit**- und den **4bit**-Modus. Für den **8bit-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 **4bit-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 legt man einfach auf GND.

Der 4bit-Modus hat den Vorteil, dass man 4 IO-Pins weniger benötigt als beim 8bit-Modus, weshalb ich mich hier für eine Ansteuerung mit 4bit entschieden habe.

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. Ist RS Low, dann wird das ankommende Byte als Befehl interpretiert, ist RS high, dann wird das Byte auf dem LCD angezeigt.

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** lesen, das anzeigt ob das LCD den vorhergehenden Befehl fertig verarbeitetet 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 vorsichtshalber ein paar Mikrosekunden warten sollte um dem LCD Zeit zum Ausführen des Befehls zu geben.

Anschluß an den Controller

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

1	Vss	GND
2	Vcc	5V
3	Vee	GND oder Poti (siehe oben)
4	RS	PD4 am AVR
5	RW	GND
6	E	PD5 am AVR
7	DB0	GND
8	DB1	GND
9	DB2	GND
10	DB3	GND
11	DB4	PD0 am AVR
12	DB5	PD1 am AVR
13	DB6	PD2 am AVR
14	DB7	PD3 am AVR

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?

Ansteuerung des LCDs im 4bit-Modus

Um ein Byte zu übertragen, muss man es erstmal in die beiden Nibble zerlegen, die getrennt übertragen werden. Da das obere Nibble (Bit4-Bit7) 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 PORTB, 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 lassen (=Befehl senden) oder auf 1 setzen (=Daten senden). Um ein Bit in einem normalen (nicht IO-!) Register zu setzen gibt es den Befehl **sbr** (**S**et **B**it in **R**egister). 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
```

An PD4 ist RS 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 geschalten 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(Enable)-Leitung (an PD5 angeschlossen) auf high und kurz darauf wieder auf low gesetzt:

```
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.

Routinen zur LCD-Ansteuerung

Die Routinen zur Kommunikation mit dem LCD sehen also so aus:

Download lcd-routines.asm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  LCD-Routinen                  ;;
;;                  ============                  ;;
```

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



 ;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
           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 ohne RS zu setzen
           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
lcd_enable:
           sbi PORTD, 5                 ;Enable high
           nop                          ;3 Taktzyklen warten
           nop
           nop
           cbi PORTD, 5                 ;Enable wieder low
           ret                          ;Und wieder
zurück

 ;Pause nach jeder Übertragung
delay50us:                              ;50us 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  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        ;noch was einstellen...
           rcall lcd_command
           ldi temp1, 0b00001100        ;...nochwas...
           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
```

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 4MHz 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 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. Wie das bequemer geht wird in Teil 7 des Tutorials erklärt werden.

Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)net

Home

AVR-Tutorial

- 1. Ausrüstung
- 2. IO-Grundlagen
- 3. Stack
- 4. LCD
- 5. Interrupts
- 5. UART
- 7. Speicher

Foren

- µC & Elektronik
- Programmierbare Logik
- DSP
- GCC
- Codesammlung
- Markt
- Platinen
- PC-Programmierung
- **Ausbildung & Beruf**
- Webseite
- Sonstiges/Offtopic

Chat

Shop

Artikel

- AVR-GCC-Tutorial
- AVR Checkliste
- AVR Assembler Befehlstabelle
- Operationsverstärker
- SMD löten
- Digitaler Funktionsgenerator
- Linksammlung
- weitere...
- [Letzte Änderungen]

AVR-Tutorial - 5. Interrupts

Bei bestimmten Ereignissen in Prozessoren wird ein sogenannter "Interrupt" ausgelöst. Dabei wird das Programm unterbrochen und ein Unterprogramm aufgerufen. Wenn dieses beendet ist, läuft das Hauptprogramm ganz normal weiter.

Bei Mikrocontrollern werden Interrupts z.B. ausgelöst wenn:

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

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

Wir wollen uns hier erst mal die beiden Interrupts **INT0** und **INT1** anschauen. INT0 wird ausgelöst, wenn sich der an PD2 (Pin 4) anliegende Wert ändert, INT1 reagiert auf Änderungen an PD3 (Pin 5).

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:

ISCx1	ISCx0	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 **GIMSK** auf 1 gesetzt werden.

> Die Register MCUCR und GIMSK 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 und im Datenblatt des Controllers). Somit bleiben zum Zugriff auf diese Register nur die Befehle **in** und **out** übrig.

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

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	Analogkomperator
18	0x011	TWI	Two-Wire Interface
19	0x012	SPM_RDY	Store Program Memory Ready

So, wir wissen jetzt dass der Controller zu 0x001 springt wenn INT0 auftritt. Aber dort ist ja nur Platz für einen Befehl, denn die nächste Adresse ist doch für INT1 reserviert? 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, an die dann gesprungen wird. Diese durch den Interrupt aufgerufene Routine nennt man **Interrupt-Handler**.

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.

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 die Assemblerdirektive **.org**. **.org 0x001** sagt dem Assembler, dass er die darauffolgenden Befehle ab Adresse 0x001 im Programmspeicher, der von INT0 angesprungenen Stelle, plazieren soll.

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 **INTOaddr** 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 eingeschalten wird) wird das Programm immerab 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. anlegen.

Wenn man mehrere Interrupts verwenden möchte kann man auch, anstatt jeden Interruptvektor

einzeln mit .org an die richtige Stelle zu rücken, die gesamte Sprungtabelle auszuschreiben:

Download 4333-intvectortable.asm

```
.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 Interrupt Handler
        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
```

Bei unbenutzten Interrupts ist es üblich, statt dem Sprungbefehl einfach den Befehl **reti** reinzuschreiben.

So könnte ein Minimal-Assemblerprogramm aussehen, das die Interrupts INT0 und INT1 verwendet:

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, 0b00001010  ;INT0 und INT1 konfigurieren
        out MCUCR, temp

        ldi temp, 0b11000000  ;INT0 und INT1 aktivieren
        out GIMSK, 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 abgeschalten. Das ist zwar nicht sonderlich spektakulär, aber das Prinzip sollte deutlich werden.

Meistens macht es keinen Sinn, Taster direkt an einen Interrupteingang anzuschließen. Häufiger werden Interrupts in Zusammenhang mit dem UART verwendet, um z.B. auf ein empfangenes Zeichen zu reagieren. Wie das funktioniert seht ihr im nächsten Kapitel.

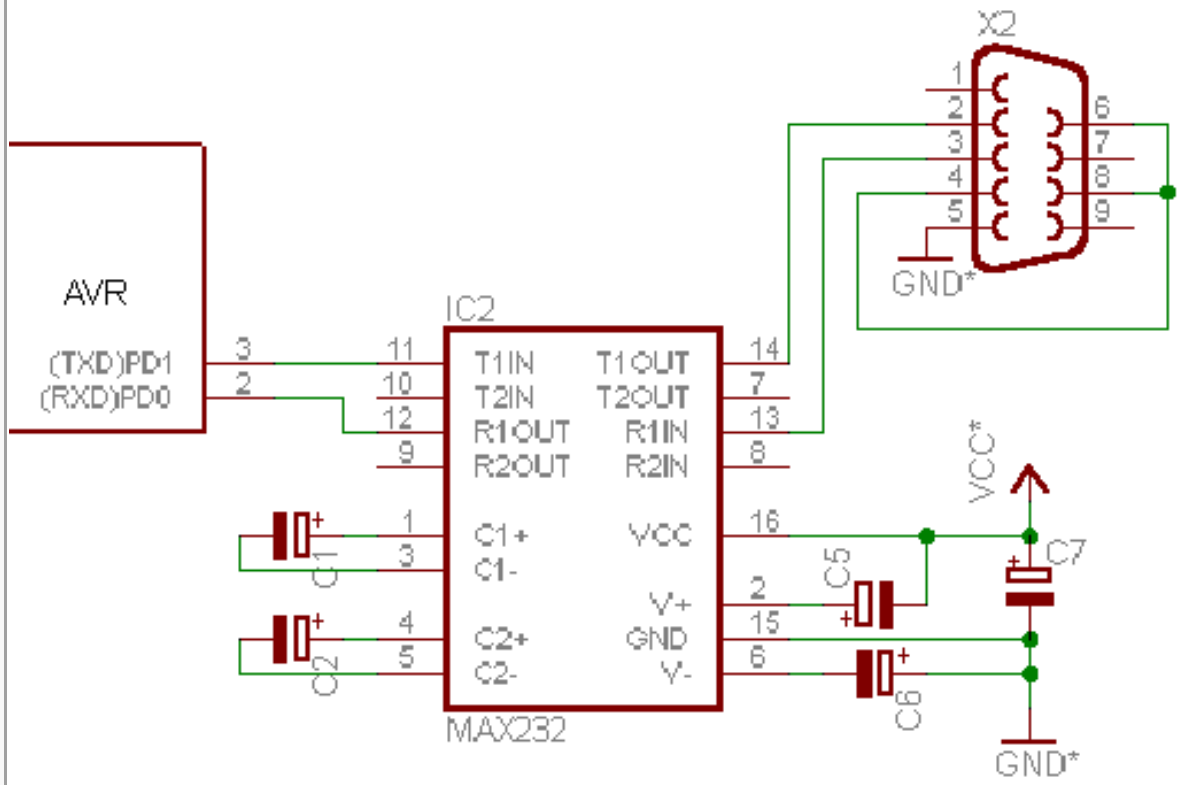
Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)net

AVR-Tutorial - 6. UART

Wie viele andere Controller besitzen die meisten AVRs ein UART ("Universal Asynchronous Receiver and Transmitter"). 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.

1. Hardware

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



Der MAX232 ist ein Pegelkonverter, der die -12V/+12V Signale an der seriellen Schnittstelle des PCs zu den

Funktionsgenerator
- Linksammlung
- weitere...
- [Letzte Änderungen]

5V/0V des AVRs kompatibel macht.
Die 5 Kondensatoren sind 22µF-Elkos. Auf die richtige Polung achten! Der exakte Wert ist hier relativ unkritisch, in der Praxis sollte alles von ca. 10µF bis 50µF funktionieren.
X2 ist ein 9-poliger Sub-D-Verbinder, female.

Auf dem Board von http://shop.mikrocontroller.net/ sind diese Bauteile bereits enthalten, man muss nur noch die Verbindungen zwischen MAX232 und AVR herstellen: Bild

Die Verbindung zwischen PC und Mikrocontroller erfolgt über ein 9-poliges Modem(nicht Nullmodem!)-Kabel, das an den seriellen Port des PCs angeschlossen wird.

2. Software

Senden

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

```
UBRR = Taktfrequenz / 16 * 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). Im Normalfall steht im UBRRH 0, da der berechnete Wert kleiner als 256 ist und somit in UBRRL alleine passt.

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.

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

Das Beispielprogramm überträgt die Zeichenkette "Test!" in einer Endlosschleife an den PC. Die folgende Version ist für den ATmega8 geschrieben; wer den AT90S4433 verwendet findet die passende Version hier.

Download uart-mega8.asm

```
.include "m8def.inc"

.def temp = r16
.equ CLOCK = 4000000
.equ BAUD = 9600
.equ UBRRVAL = CLOCK/(BAUD*16)-1

        ; Stackpointer initialisieren
        ldi temp, LOW(RAMEND)
        out SPL, temp
        ldi temp, HIGH(RAMEND)
        out SPH, temp

        ; Baudrate einstellen
        ldi temp, LOW(UBRRVAL)
        out UBRRL, temp
        ldi temp, HIGH(UBRRVAL)
        out UBRRH, temp

        ; Frame-Format: 8 Bit
        ldi temp, (1<<URSEL)|(3<<UCSZ0)
        out UCSRC, temp

        sbi UCSRB,TXEN                  ; TX aktivieren

loop:   ldi temp, 'T'
        rcall serout                    ; Unterprogramm aufrufen
        ldi temp, 'e'
        rcall serout                    ; Unterprogramm aufrufen
        ldi temp, 's'
        rcall serout                    ; ...
        ldi temp, 't'
        rcall serout
        ldi temp, '!'
        rcall serout
        ldi temp, 10
        rcall serout
```

```
        ldi temp, 13
        rcall serout
        rjmp loop

serout:
        sbis UCSRA,UDRE                 ; Warten bis UDR für das nächste
                                        ; Byte bereit ist
        rjmp serout
        out UDR, temp
        ret                             ; zurück zum Hauptprogramm
```

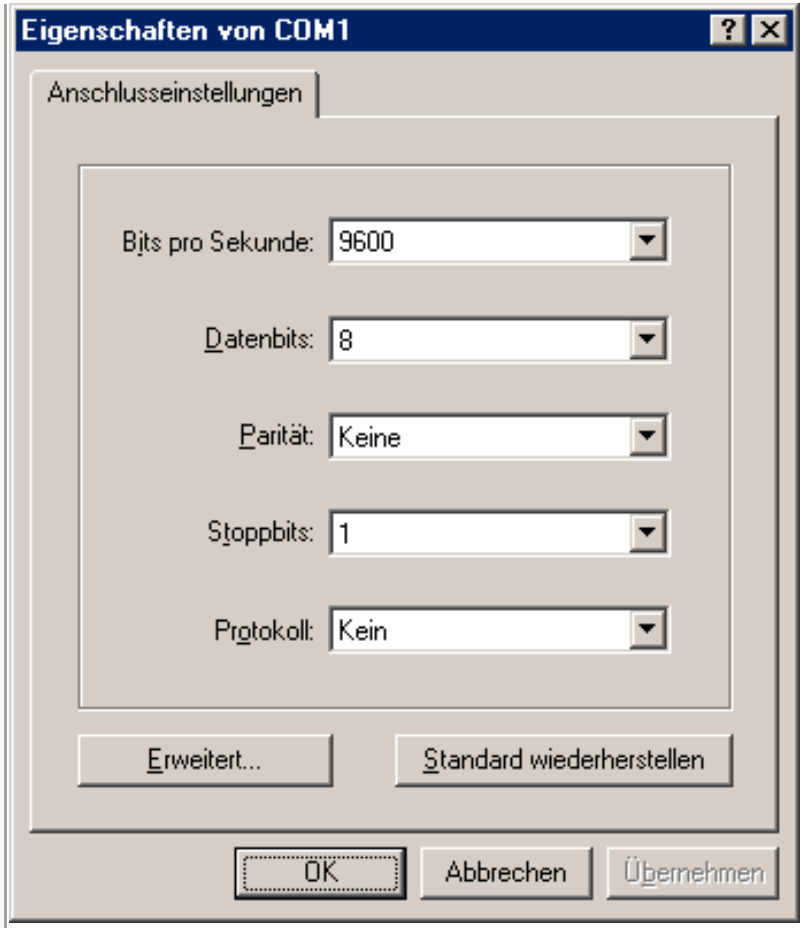
Der Befehl **rcall serout** ruft ein kleines Unterprogramm auf, das zuerst wartet bis das Datenregister UDR von der vorhergehenden Übertragung frei ist, und anschließend das in temp (=r16) gespeicherte Byte an UDR ausgibt.

Bevor **serout** aufgerufen wird, wird temp 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.

Eine bequemere Methode längere Zeichenketten zu übertragen, wird in Teil 7 des Tutorials behandelt werden.

Die Berechnung der Baudrate wird übrigens nicht im Controller 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.

Empfangen

Natürlich kann der AVR nicht nur Daten senden, sondern auch vom PC empfangen. Dazu muss man, nachdem die Baudrate wie oben beschrieben eingestellt wurde, das Bit RXEN setzen.

Sobald das UART dann ein Byte über die serielle Verbindung empfängt, wird das Bit RXC im Register UCSRA gesetzt, um anzuzeigen, dass das Byte im Register UDR es zur Weiterverarbeitung bereitsteht. Sobald es aus UDR gelesen wird, 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 ensprechende Bit im Register PORTD keine Funktion hat und damit auch nicht für die Datenanzeige verwendet werden kann.

Nachdem das UART konfiguriert ist wartet das Programm einfach in der Hauptschleife darauf, dass ein Byte über das 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.

Download uart-receive.asm

```
.include "m8def.inc"

.def temp = R16
.equ CLOCK = 4000000
.equ BAUD = 9600
.equ UBRRVAL = CLOCK/(BAUD*16)-1

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

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

        ; Baudrate einstellen
        ldi temp, LOW(UBRRVAL)
        out UBRRL, temp
        ldi temp, HIGH(UBRRVAL)
        out UBRRH, temp

        ; Frame-Format: 8 Bit
        ldi temp, (1<<URSEL)|(3<<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
```

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:

Download uart-receive-interrupt.asm

```
.include "m8def.inc"                    ; bzw. 2333def.inc

.def temp = R16
.equ CLOCK = 4000000
.equ BAUD = 9600
.equ UBRRVAL = CLOCK/(BAUD*16)-1

.org 0x00
        rjmp main

.org URXCaddr                           ; Interruptvektor für UART-Empfang
        rjmp int_rxc

; Hauptprogramm
main:
        ldi temp, LOW(RAMEND)
        out SPL, temp
        ldi temp, HIGH(RAMEND)
        out SPH, temp

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

        ; Baudrate einstellen
        ldi temp, LOW(UBRRVAL)
        out UBRRL, temp
        ldi temp, HIGH(UBRRVAL)
```

```
        out UBRRH, temp

        ; Frame-Format: 8 Bit
        ldi temp, (1<<URSEL)|(3<<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
        out PORTD, temp
        pop temp                        ; temp wiederherstellen
        reti                            ; Interrupt beenden
```

Diese Methode hat den großen Vorteil, dass das Hauptprogramm (hier nur eine leere Endlosschleife) völlig unbeeinflusst weiterläuft, während der Controller auf die Daten wartet. 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 ändern, müssen alle in der Interruptroutine geänderten Register am Anfang der Routine *gesichert* und am Ende *wiederhergestellt* werden. In dem diesem Zusammenhang wird der *Stack* wieder interessant: um die Register zu sichern kann man sie mit "push" einfach 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 im Interrupt 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 Vergleich positiv) und "brne label" (springe zu label, wenn Vergleich negativ) auswerten.

Download uart-led.asm

```
.include "m8def.inc"

.def temp = R16
.equ CLOCK = 4000000
.equ BAUD = 9600
.equ UBRRVAL = CLOCK/(BAUD*16)-1

.org 0x00
        rjmp main

.org URXCaddr
        rjmp int_rxc

; Hauptprogramm
main:
        ldi temp, LOW(RAMEND)
        out SPL, temp
        ldi temp, HIGH(RAMEND)
        out SPH, temp

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

        ; Baudrate einstellen
        ldi temp, LOW(UBRRVAL)
        out UBRRL, temp
        ldi temp, HIGH(UBRRVAL)
        out UBRRH, temp

        ; Frame-Format: 8 Bit
        ldi temp, (1<<URSEL)|(3<<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
        cpi temp, '1'                   ; empfangenes Byte mit '1' vergleichen
        brne int_rxc_1                  ; wenn nicht gleich, dann zu int_rcx_1
        cbi PORTB, 0                    ; LED einschalten
int_rxc_1:
        cpi temp, '0'                   ; empfangenes Byte mit '0' vergleichen
        brne int_rxc_2                  ; wenn nicht gleich, dann zu int_rcx_2
        sbi PORTB, 0                    ; LED ausschalten
int_rxc_2:
        pop temp                        ; temp wiederherstellen
        reti
```

Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)
net

AVR-Tutorial - 7. Flash, EEPROM, RAM

Die AVR-Mikrocontroller besitzen 3 verschiedene Arten von Speichern:

	Flash	EEPROM	RAM
Schreibzyklen	>10.000	>100.000	unbegrenzt
Lesezyklen	unbegrenzt	unbegrenzt	unbegrenzt
flüchtig	nein	nein	ja
Größe beim AT90S2333	2kB	128B	128B
Größe beim AT90S4433	4kB	256B	128B
Größe beim ATmega8	8kB	512B	1kB

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 Byte für Byte ausgelesen und ausgeführt, es lässt sich aber auch zur Speicherung von Daten (z.B. Texte für eine LCD-Anzeige) nutzen. Vom laufenden Programm aus kann man das ROM normalerweise nur lesen, nicht beschreiben.
Es kann beliebig oft ausgelesen werden, aber (theoretisch) nur ~10000 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, in dem z.B. bei Unterprogrammaufrufen (rcall) die Rücksprungadresse gespeichert wird (siehe Kapitel 3).

- Linksammlung
- weitere...
- [Letzte Änderungen]

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 R25 (Low-Byte) und R26 (High-Byte), daher kann man das Laden einer Konstante wie gewohnt mit dem Befehl "ldi" durchführen. Statt R25 und R26 kann man übrigens einfach "ZL" und "ZH" schreiben, da diese Synonyme bereits in der include-Datei 4433def.inc definiert sind.

Wenn die richtige Adresse erst mal 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 dass das durch den Z-Pointer addressierte Byte aus dem Programmspeicher in das Register R0 geladen wird, von wo aus man es weiterverarbeiten kann.

Jetzt muss man nur noch wissen, wie man dem Assembler überhaupt beibringt dass er die von uns festgelegte Daten im ROM plazieren 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
```

Dieser Ausschnitt sagt dem Assembler, dass er die angegebenen Bytes nacheinander im Speicher plazieren 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 2 sollte man sich erst mal keine Gedanken machen: "das ist einfach so"™.

Um zu zeigen wie das alles konkret funktioniert ist das folgende Beispiel nützlich:

Download lpm1.asm

```
.include "4433def.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 nach
R0 lesen
        mov R16, R0                     ; nach R16 kopieren
        out PORTB, R16                  ; 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.

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 das UART aus.

Download lpm-print.asm

```
.include "4433def.inc"

.def temp = R16

        ldi R16, RAMEND
        out SPL, R16                    ; Stackpointer initialisieren

        sbi UCSRB,TXEN                  ; UART TX aktivieren
        ldi temp,4000000/(9600*16)-1    ; Baudrate 9600 einstellen
        out UBRR,temp

start:
        ldi ZL, LOW(text*2)             ; Adresse des Strings in den
        ldi ZH, HIGH(text*2)            ; Z-Pointer laden
        rcall print                     ; Unterfunktion print aufrufen

        ldi R16, 10                     ; die Bytes 10 und 13 senden
        rcall sendbyte                  ; (Zeilenumbruch im Terminal)
        ldi R16, 13
        rcall sendbyte

        rjmp start                      ; das Ganze wiederholen


; 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

text:
.db "AVR-Assembler ist ganz einfach",0    ; Stringkonstante, durch eine 0
abgeschlossen
```

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 Buchtstaben 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.
- `tst [Register]`
  Prüft, ob der Inhalt eines Register 0 ist.
- `breq [Label]`
  Springt zu [Label], wenn der vorhergehende Vergleich wahr ist 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

Lesen

Zuerst wird die EEPROM-Adresse von der gelesen werden soll in das IO-Register "EEAR" (Eeprom Address

Register) geladen. 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 nun aus dem IO-Register "EEDR" (Eeprom Data Register) in ein normales Arbeitsregister kopiert und von dort weiterverarbeitet werden.

Doch um etwas aus dem EEPROM lesen zu können muss man natürlich erst mal Daten hineinbekommen.

Wie auch das Flash-ROM kann man das EEPROM direkt 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 Asembler 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.

Download eeprom1.asm

```
.include "4433def.inc"

        ldi R16, 0xFF
        out DDRB, R16                   ; Port B: Ausgang

        ldi r16, daten
        out EEAR, r16                   ; Adresse laden

        sbi EECR, EERE                  ; Lesevorgang aktivieren

        in r16, EEDR                    ; gelesenes Byte nach R16 kopieren

        out PORTB, r16                  ; ... und an PORTB ausgeben

loop:   rjmp loop

.eseg                                   ; EEPROM-Segment aktivieren
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.

Bei Controllern mit mehr als 256 Byte EEPROM, z.B. dem ATmega8, passt die EEPROM-Adresse nicht mehr in ein Byte alleine. Statt einem Adressregister (EEAR) gibt es bei diesen Controllern deshalb 2 Register, eines für das Low-Byte (EEARL) und eines für das High-Byte (EEARH). Für den Atmega8 muss das oben stehende Programm also folgendermaßen abgeändert werden:

Download eeprom1-m8.asm

```
.include "m8def.inc"

        ldi R16, 0xFF
        out DDRB, R16                   ; Port B: Ausgang

        ldi r16, HIGH(daten)            ; Adresse laden
        out EEARH, r16
        ldi r16, LOW(daten)
        out EEARL, r16

        sbi EECR, EERE                  ; Lesevorgang aktivieren

        in r16, EEDR

        out PORTB, r16

loop:   rjmp loop

.eseg
daten:
.db 0b10101010
```

Natürlich kann man auch aus dem EEPROM Strings lesen und an das UART senden:

Download eeprom-print.asm

```
.include "4433def.inc"

.def temp           = r16
.def address        = r17
.def data           = r18
```

```
        ldi temp, RAMEND
        out SPL, temp

        sbi UCSRB,TXEN                  ; UART TX aktivieren
        ldi temp,4000000/(9600*16)-1    ; Baudrate 9600 einstellen
        out UBRR,temp

        ldi address, text1              ; ersten String senden
        rcall eep_print

        ldi address, text2              ; zweiten String senden
        rcall eep_print

        ldi data, 10                    ; die Bytes 10 und 13 senden
        rcall sendbyte                  ; (Zeilenumbruch im Terminal)
        ldi data, 13
        rcall sendbyte


loop:   rjmp loop

eep_print:
        out EEAR, address               ; EEPROM-Adresse
        sbi EECR, EERE                  ; Lesevorgang starten
        in data, EEDR                   ; gelesenes Byte nach "data"
        tst data                        ; auf 0 (Stringende testen)
        breq eep_print_end              ; falls 0, Funktion beenden
        rcall sendbyte                  ; ansonsten Byte senden...
        inc address                     ; ... Adresse um 1 erhöhen...
        rjmp eep_print                  ; ... und zum Anfang der Funktion
eep_print_end:
        ret

; sendbyte: sendet das Byte aus "data" über das UART
sendbyte:
        sbis UCSRA, UDRE                ; warten bis das UART bereit ist
        rjmp sendbyte
        out UDR, data
        ret
```

```
.eseg
text1:
.db "Strings funktionieren auch ", 0
text2:
.db "im EEPROM", 0
```

Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)
net