
Assembler ab 0 und 1 8.10.2002

Gerhard Paulus
gp@gnomsoft.de

Dieser Text könnte Leuten nützlich sein, die sich für Mikrocontroller und Assembler-Programmierung interessieren und ganz von vorn anfangen (bei 0 und 1 sozusagen). Als konkrete Hardware dient dabei der Mikrocontroller AVR AT90S4433 von Atmel. Ansonsten wird als Assembler-Programm avrasm.exe von Atmel verwendet, das Teil der AVR-Studio-Software ist, die man sich bei `http://www.atmel.com` gratis besorgen kann.

Für Beispiel-Schaltungen und weiterführende Texte und Beispiele wird das Tutorial auf `http://www.mikrocontroller.net` und das Tutorial auf `http://www.avr-asm-tutorial.net` empfohlen.

1. Zahlen:

Im Mikrocontroller (MC fürderhin) gibt es Bereiche, die Zahlen speichern. Diese sogenannten Register sind direkt mit der Recheneinheit des MC verbunden, die diese Zahlen addieren, subtrahieren, vergleichen etc. kann.

Ein Register kann beim AVR 1 Byte fassen, also 8 Bit an Information. Ein Bit kann als Information eine 1 darstellen (eingeschaltet) oder eine 0 (ausgeschaltet), also eine durchaus überschaubare Informationsfülle. Bit ist übrigens ein Kunstwort, das sich aus “binary digit” ableitet.

Die Bits eines Bytes werden von hinten nach vorn gezählt: das letzte Bit (LSB, Least Significant Bit) ist Bit Nummer 0, das erste Bit (MSB, Most Significant Bit) ist Bit Nummer 7.

```
          ---------------------------------------------------------
1 Byte :  |      |      |      |      |      |      |      |      |
          ---------------------------------------------------------
          Bit 7   Bit 6  Bit 5  Bit 4  Bit 3  Bit 2  Bit 1  Bit 0
          (MSB)                                          (LSB)
```

In englischen Texten ist ein Bit “set” (gesetzt), wenn es 1 darstellt. Und ein Bit ist “cleared” (zurückgesetzt bzw. gelöscht), wenn es 0 darstellt.

Von besagten Registern gibt es recht viele: 32 davon sind als generell verfügbare Arbeitsregister konzipiert. Diese Arbeitsregister werden in Programmen angesprochen mit den Namen `R00` bis `R31` bzw. `r0` bis `r31` (der Assembler unterscheidet bei Namen nicht zwischen Groß- und Kleinschreibung). 64 Register sind ausgelegt als Register für Eingabe und Ausgabe; diese Input/Output-Register (bzw. I/O-Register) werden auch als “ports” bezeichnet (Anlegestellen, sozusagen). Und wie im reellen Leben gibt es da eine Arbeitsteilung, d.h. manche Register können mehr als andere.

Jetzt wollen wir doch mal schauen, was man mit den Bits in den Registern so alles machen kann.

Wenn alle Bits auf 0 gesetzt sind dann sieht das so aus

```
0 0 0 0  0 0 0 0
```

und damit wird die Zahl 0 dargestellt, klingt irgendwie logisch.

Wenn alle Bits auf 1 gesetzt sind dann sieht das so aus

```
1 1 1 1  1 1 1 1
```

und damit wird entweder die Zahl 255 oder die (negative) Zahl -1 dargestellt. Schauen wir uns mal die Zahl 255 näher an, die Sache mit den negativen Zahlen kommt später.

Das übliche dezimale Zahlensystem kennt 10 Ziffern (0,1,2,3,4,5,6,7,8,9) und hat als Basis die Zahl 10. Die Dezimal-Zahl 255 kann man damit folgendermaßen auseinandernehmen:

```
2 5 5
| | |_ 5 * 10 hoch 0  = 5 * 1   =   5
| |___ 5 * 10 hoch 1  = 5 * 10  =  50
|_____ 2 * 10 hoch 2  = 2 * 100 = 200
                                 -----
                                  255
```

OK, das wußten wir schon, aber wie ist das mit dem Byte und den lauter Einsen ?
Antwort: Das ist genauso, aber halt binär, das heißt als Zahlensystem mit Basis 2 und mit nur 2 Ziffern: 0 und 1.

```
1 1 1 1 1 1 1 1
| | | | | | | |_ 1 * 2 hoch 0  = 1 * 1   =   1
| | | | | | |___ 1 * 2 hoch 1  = 1 * 2   =   2
| | | | | |_____ 1 * 2 hoch 2  = 1 * 4   =   4
| | | | |_______ 1 * 2 hoch 3  = 1 * 8   =   8
| | | |_________ 1 * 2 hoch 4  = 1 * 16  =  16
| | |___________ 1 * 2 hoch 5  = 1 * 32  =  32
| |_____________ 1 * 2 hoch 6  = 1 * 64  =  64
|_______________ 1 * 2 hoch 7  = 1 * 128 = 128
                                         -----
                                         255
```

Jetzt könnte man natürlich fragen: Und wieso ist das nicht gleich 2 hoch 8, also 256 ?
Antwort: mit den 8 Bits eines Bytes kann man tatsächlich 256 Zahlen darstellen, aber 0 ist ja bereits eine Zahl. Daher ist die höchste mit 8 Bit darstellbare Zahl 256 - 1, also 255. Um die Zahl 256 binär zu speichern sind mindestens 9 Bit erforderlich.

Jetzt sind diese binären Zahlen beim Ausdrucken etwas unübersichtlich. Es hat sich in “Assembler-Kreisen” so entwickelt, daß die binären Daten in hexadezimaler Schreibweise ausgegeben werden. Also noch ein Zahlensystem, diesmal zur Basis 16. Im hexadezimalen System gibt es folgende Ziffern :

```
1, 2, 3, 4, 5, 6, 7, 8, 9,  A ,  B ,  C ,  D ,  E ,  F
                          (10) (11) (12) (13) (14) (15)
```

Die dezimale Zahl `255` wird hexadezimal als `FF` dargestellt:

```
F F
| |___ F * 16 hoch 0  = 15 * 1  =  15
|_____ F * 16 hoch 1  = 15 * 16 = 240
                                 -----
                                  255
```

Unbedingt klarer ist die Schreibweise damit nicht, aber mit 2 Druckzeichen pro Byte sehr kompakt.

```
dezimal: 14
binär:   00001110
hex:     0E
```

Da diese hexadezimale Schreibweise in Assemblerprogrammen sehr oft verwendet wird, gibt es durchaus Sinn, sich die entsprechenden Bitfolgen einzuprägen. Dann ist die hexadezimale Schreibweise quasi Steno für die binäre Schreibweise.

```
binär hex
----- ---
0001   1
0010   2
0011   3
0100   4
0101   5
0110   6
0111   7
1000   8
1001   9
1010   A (10)
1011   B (11)
1100   C (12)
1101   D (13)
1110   E (14)
1111   F (15)
```

Obige Zahlensysteme lassen sich übrigens direkt in Assembler-Programmen benutzen. Um die dezimale Zahl 14 in Register R16 zu speichern, sind die folgenden Anweisungen gleichwertig:

```
ldi R16, 14         ; dezimale Schreibweise ist Standard
ldi R16, 0b00001110 ; binäre Zahlen fangen mit 0b an
ldi R16, 0x0E       ; Hex-Zahlen fangen mit 0x an
ldi R16, $0E        ; alternative Schreibweise für Hex-Zahlen
```

Und wie funktioniert Addieren von binären Zahlen ?

```
 dezimal:                binär:

   1                     0000 0001
   1 +                   0000 0001 +
 -----                   -----------
   2                     0000 0010

   9                     0000 1001
   5 +                   0000 0101 +
 -----                   -----------
  14                     0000 1110
```


Im Prinzip wie bei dezimalen Zahlen. Man geht also von rechts nach links, zählt jeweils 2 Ziffern zusammen und wenn die Summe größer ist als die größtmögliche Ziffer wird eine 1 nach links übertragen. Bei binären Zahlen erfolgt dieses Übertragen immer wenn beide zu addierenden Ziffern 1 sind.

Als Regel für binäres Addieren:

```
0 + 0 = 0 kein Übertrag
1 + 0 = 1 kein Übertrag
0 + 1 = 1 kein Übertrag
1 + 1 = 0 mit Übertrag nach links benachbartem Bit
```

Und wie funktioniert Subtrahieren von binären Zahlen ?

```
dezimal:                binär:

   1                    0000 0001
   1 -                  0000 0001 -
 -----                  -----------
   0                    0000 0000

   9                    0000 1001
   5 -                  0000 0101 -
 -----                  -----------
   4                    0000 0100
```

Falls von 0 die 1 abgezogen wird, dann wird 1 vom links benachbarten Bit abgezogen bzw. geborgt.

Als Regel für binäres Subtrahieren:

```
0 - 0 = 0   ohne Übertrag
1 - 0 = 1         ohne Übertrag
0 - 1 = 1         mit “Borgen” vom links benachbarten Bit
1 - 1 = 0   ohne Übertrag
```

Obige Rechenoperationen würden mit Registern zB. folgendermaßen ablaufen:

```
; Addition:
ldi r16, 9   ; lade 9 in Register r16
ldi r17, 5   ; lade 5 in Register r17
add r16, r17 ; danach ist das Ergebnis (also 14) in Register 16

; Subtraktion:
ldi r16, 9   ; lade 9 in Register r16
ldi r17, 5   ; lade 5 in Register r17
sub r16, r17 ; danach ist das Ergebnis (also 4) in Register 16
```

Wenn in einem Byte maximal die Zahl 255 dargestellt werden kann und eine Zahl dazu addiert wird, dann kann es natürlich vorkommen, das das Ergebnis zu groß wird und in das Byte nicht mehr reinpaßt. Was passiert dann?

Im folgenden fängt das Register mit 0 an und dann wird immer wieder 1 addiert (das nennt man Inkrementieren und ist trotzdem eine anständige Sache).

```
Register:
---------
0000 0000                        am Anfang ist alles Null

             + 0000 0001
0000 0001
             + 0000 0001
0000 0010
             + 0000 0001
0000 0011
             + 0000 0001
... etc. ...

1111 1110
             + 0000 0001
1111 1111
             + 0000 0001
0000 0000                  und jetzt geht es wieder bei Null weiter !
             + 0000 0001
0000 0001
```

Wenn alle Bits gesetzt sind und es wird immer noch dazuaddiert dann bleibt dem MC nichts anderes übrig, als wieder bei Null (alle Bits auf 0) weiterzumachen.

Im folgenden fängt das Register mit 2 an und dann wird immer wieder 1 subtrahiert (das nennt man Dekrementieren und ist immer noch eine anständige Sache).

```
Register:
---------
0000 0010
             - 0000 0001
0000 0001
             - 0000 0001
0000 0000
             - 0000 0001
1111 1111
             - 0000 0001
1111 1110
```

Wenn beim Subtrahieren die niedrigste Zahl erreicht ist (also 0) und immer noch weiter abgezogen ist, dann springt der MC zur höchsten Zahl (alle Bits auf 1) und macht dort weiter.

Wenn beim Addieren bzw. Subtrahieren zwischen 0 und 255 umgesprungen wird (also von `0000 0000` zu/von `1111 1111`), dann weist der MC darauf hin, daß etwas außerordentliches passiert ist. Er setzt in dem Status-Register SREG das unterste Bit (Bit Nummer 0) auf 1. Mit anderen Worten: der Carry-Flag wird gesetzt (Übertrags-Fähnchen klingt etwas seltsam, wäre aber die Übersetzung ins Deutsche). In dem Statusregister gibt es noch weitere Bits, die wichtige Informationen beinhalten. Hier die komplette Flaggen-Parade:

```
Status-Register: Bit 7   I  Interrupt Enabled flag
(SREG)           Bit 6   T  Transfer flag
                 Bit 5   H  Half-carry flag
                 Bit 4   S  Sign flag
                 Bit 3   V  Overflow flag
                 Bit 2   N  Negative flag
                 Bit 1   Z  Zero flag
                 Bit 0   C  Carry flag
```


Im folgenden sind nur die beiden unterwertigsten Flags interessant. Der Carry-flag wird gesetzt, wenn der Wert eines Registers bei Addition durch's Dach geht bzw. bei Subtraktion durch den Boden geht; andernfalls wird er auf 0 gesetzt. Der Zero-Flag wird nach jeder Rechenoperation gesetzt (Bit hat Wert 1) wenn das Ergebnis der Berechnung eine Null ist; andernfalls wird der Zero-Flag rückgesetzt (Bit hat Wert 0).

Die meisten der anderen Flags sind relevant für Rechenoperationen für Zahlen mit Vorzeichen, die später behandelt werden. Bis jetzt wurden nur Rechenoperationen für Zahlen ohne Vorzeichen behandelt, wo negative Zahlen nicht vorkommen können.

Soweit mal zum Thema Zahlen, jetzt schauen wir uns doch mal die Sache mit dem Speicher an.

2. Speicher :

Der MC hat einen Hauptspeicher, der Daten (also Zahlen) speichern kann. Der Hauptspeicher fängt mit den Arbeitsregistern an, hinter denen sich die I/O-Register anschließen und dahinter fängt der SRAM an. Jedes Byte, das in diesem Speicher gehalten wird, kann durch seine Adresse angesprochen werden. Diese Adressen werden üblicherweise in hexadezimaler Schreibweise angegeben.

Beim AT90S4433 sieht das so aus (niedrigste Addresse oben, jedes Register belegt ein Byte):

```
Adresse
dez:  hex:
  0    00     Register 0   (R0)   +
  1    01     Register 1   (R1)   |
                                  |
               ...                |  32 Arbeitsregister a 1 Byte
                                  |
 30    1E     Register 30   (R30) |
 31    1F     Register 31   (R31) +

 32    20     ???                 +
                                  |
               ...                |  64 I/O Register a 1 Byte
                                  |
 95    5F     SREG                +

 96    60     erstes Byte              +
                                  |
               ...                |  128 Bytes SRAM
                                  |
223    DF     letztes Byte        +
```

Wird ein Byte in diesem Datenspeicher gespeichert, dann "wohnt" es sozusagen in einem Lagerregal. Man kann die Rechenheit via Programm anweisen, an einer bestimmten Addresse ein Byte zu holen, zu ändern und wieder an dieser Addresse zu speichern. Dazu gibt es bei avrasm die "`ld`", "`ldd`" bzw. "`lds`" Befehle.

```
ldi r16, 13   ; lade Wert dez 13 in Register 0
sts 0x60, r16 ; speichere den Wert von Register 16 an der untersten Adresse des SRAM
lds r0, 0x60  ; lese das Byte an der untersten Adresse des SRAM in das Register 0
```

In Assemblerprogrammen explizit mit Speicheraddressen zu arbeiten, wäre zu umständlich. Deshalb bekommen bestimmte Adressen im Hauptspeicher Namen, zB. werden die Arbeitsregister generell mit R0 bis R31 bezeichnet. Beim Assemblieren werden diese Namen automatisch in die richtige Adresse umgewandelt. Der Assembler nutzt dazu die Include-Datei (hier `4433def.inc`), die mit der Direktive .include in das Assembler-Programm eingebunden wird.
Das Status-Register (mit den diversen Flags) hat zum Beispiel im Hauptspeicher die Speicheradresse `hex 5F`, die im Programm mit dem Namen SREG spezifiziert werden kann.

Ein eigenständiger Speicher ist der Programmspeicher, in dem der Programmcode gespeichert wird, den der AVR zur Laufzeit der Anwendung abarbeitet. Dieser Flash-Speicher ist beim AT90S4433 4096 Bytes groß (4 KB). Der Programmcode ist das Ergebnis des Assemblierens, wobei jeder Programmbefehl in 2 Bytes gepackt wird (also insgesamt 16 Bit, auch "word" genannt).

Auch der EEPROM-Speicher ist ein eigenständige Speicher, dessen Adressbereich bei `0` anfängt. Es gibt keine expliziten Befehle, um in den EEPROM zu direkt schreiben bzw. von dort zu lesen. Für diese Operationen sind bestimmte IO-Register einzusetzen. In vereinfachter Form sieht das dann so aus:

```
ldi r16, 13   ; irgeneine Adresse kleiner 256
ldi r17, 3    ; irgendein Wert kleiner 256
out EEAR, r16 ; Adresse einstellen in "EE Address Register"
out EEDR, r17 ; Wert in EEPROM schreiben über "EE Data Register"
```

3. Programm :

Jetzt schauen wir uns doch mal an, wie die Assembler-Programme vom MC abgearbeitet werden. Dazu muß das assemblierte Programm in den Programmspeicher des MC geladen werden. Wenn der MC dann gestartet wird bzw. zurückgesetzt wird (reset), dann liest er automatisch den Befehl, der im Programmspeicher an der ersten Stelle steht (also in den ersten beiden Bytes) und führt ihn aus. Dann wird der Befehl ausgeführt, der an der darauffolgenden Stelle steht (also in den nächsten 2 Bytes). Der MC merkt sich dabei die laufende Nummer des aktuell ausgeführten Befehls im Register PC (program counter). Standardmäßig wird bei jedem nächsten Befehl der Inhalt des Registers PC um 1 erhöht.

Folgendes aufregende Programm

```
NOP ; erster Befehl:  no operation = tu nichts
NOP  ; zweiter Befehl: no operation = tu nichts
NOP ; dritter Befehl: no operation = tu nichts
```

würde im Programmspeicher folgendermaßen aussehen (in hexadezimaler Schreibweise mit Zeilenumbruch nach jeweils 2 Bytes):

```
0000
0000
0000
```

Obwohl es nicht so aussieht: obiges "Programm" tut tatsächlich etwas. Es beschäftigt den MC 3 Taktzyklen lang (bei 8MHz also 375 Nanosekunden). Das kann sinnvoll sein, wenn der MC in einem bestimmten Zusammenhang zu schnell ist und bei bestimmten Eingaben/Ausgaben Zeit gewonnen werden muß.

Die oben beschriebene strikt sukzessive Programm-Abarbeitung ist aber die Ausnahme. Normalerweise wird im Programm auch gesprungen, damit nicht der nächstfolgende Befehl ausgeführt wird sondern ein anderer, der im Programm entweder weiter unten oder oben steht.

In folgendem Programm sind 2 Sprünge eingebaut. Das Programm läßt LED's an Port B leuchten, wenn Taster an Port D gedrückt werden. Dazu muß Port B als "output" und Port D als "input" definiert werden.

Vorher eine kurze Bemerkung: um eine Zahl in die Arbeitsregister r16 bis r31 zu schreiben kann man den Befehl "ldi" benutzen, bei dem als Parameter die Addresse des Registers und die konkret zu speichernde Zahl mitgegeben werden muß. Bei I/O-Registern ist dazu der Befehl "out" notwendig mit Angabe der Addresse des jeweiligen I/O-Registers und der Addresse eines Arbeitsregisters, in dem die Zahl gespeichert ist. Der Befehl "ldi" funktioniert bei I/O-Registern nicht.

Das ganze sieht dann zB. so aus:

```
ldi r16, 0xFF
out DDRB, r16 ; PortB als output
```

Es wird also zuerst eine Zahl in ein Arbeitsregister geladen und von dort dann in des Register `DDRB` (Data Direction Register for Port B). Wenn in diesem I/O-Register wie in diesem Fall alle Bits auf 1 gesetzt sind, so gelten alle Pins dieses Ports als Ausgänge.

Mit dem Befehl `rjmp` (relative jump) kann der MC angewiesen werden, einen bestimmten Befehl als nächsten Befehl auszuführen. Dieser Befehl wird durch ein sog. "label" (Etikette, Warenauszeichnung oder so was) gekennzeichnet, das mit einem Doppelpunkt abschließt. Nach Ausführung des entsprechenden Befehls geht es im Programmablauf linear weiter. Das heißt, danach wird der Befehl ausgeführt, der nach dem angesprungenen Befehl steht. OK, wenn der angesprungene Befehl seinerseits ein `rjmp` Befehl ist, dann wird im Programm lustig weitergesprungen (ob das so furchtbar sinnvoll ist, sei dahingestellt).

Im Kapitel Maschinencode steht, wie diese Sprungadressen konkret vom Assembler verarbeitet werden. Dann wird auch klar, warum obige Sprünge "relativ" sind.

```
.include "4433def.inc"

      rjmp weiter  ; überspringe die nop's
      nop
      nop

weiter:
      ldi r16, 0xFF
      out DDRB, r16 ; PortB als output
      ldi r16, 0x00
      out DDRD, r16 ; PortD als input

loop:
      in r16, PIND ; lies Taster-Stellungen (lies alle Pins an Port D )
      out PORTB, r16 ; lasse entsprechend LED's leuchten (schreibe zu Pins an Port B)
      rjmp loop
```

Mit diesen Sprüngen ist auch eine Art strukturierte Programmierung möglich. Im folgenden wird eine if-Struktur programmiert, wie sie von Hochsprachen wie C oder Java bekannt sind:

```
      if ( ) {
            //
      } else {
            //
      }
```

Konkret schaltet das Programm zwei LED's an PortB alternierend an und aus und legt dann eine Kunstpause ein.

```
.include "4433def.inc"

      ldi r16, 50
      ldi r17, 0xFF
      out DDRB, r17 ; port B als Ausgabe (über LED's)

loop1:

      if1:
            cpi r16, 100 ; compare immediate
            brne else1   ; branch if not equal
            ldi r17, 0b11111110
```

```
        out PORTB, r17
        ldi r16, 50
        rjmp endif1
else1:
        ldi r17, 0b11111101
        out PORTB, r17
        ldi r16, 100
endif1:

ldi r17, 255
while1:
        subi r17, 1
        breq endwhile1 ; wenn Zero-Flag gesetzt
        nop ; Päuschen
        rjmp while1
endwhile1:
rjmp loop1
```

Der Knackpunkt in obigem Programm sind diese beiden Anweisungen:

```
        cpi r16, 100 ; compare immediate
        brne else1   ; branch if not equal
```

Es wird mit "cpi" der Inhalt von Register 16 verglichen mit der Zahl 100. Der MC macht das so, als wenn er von Register 16 die Zahl 100 abzieht und alle Flags des Status-Registers setzt, aber den Inhalt von Register 16 ansonsten so beläßt, wie er vor der Operation war. Es wird also nicht echt subtrahiert.

Die Anweisung "`brne`" prüft danach den Zero-Flag. Wenn der Inhalt des Registers 16 genauso groß ist wie die Zahl 100, dann steht nach der simulierten Subtraktion der Zero-Flag auf 1. "`brne`" wird in diesem Fall nicht zur Marke `else1` springen sondern der Programmablauf geht direkt bei der nächsten Anweisung weiter. Am Ende dieses Zweiges wird dann zur Marke `endif1` gesprungen.

Wenn der Inhalt des Registers 16 nicht identisch ist mit der Zahl 100, dann steht nach der simulierten Subtraktion der Zero-Flag auf 0. "brne" wird in diesem Fall direkt zur Marke else1 springen und von dort aus automatisch irgendwann die Marke `endif1` erreichen.

Die folgende Anweisung funktioniert ähnlich, allerdings wird hier echt subtrahiert:

```
        subi r17, 1
        breq endwhile1 ; wenn Zero-Flag gesetzt
```

Die Anweisung "`breq`" springt zur Anweisung "`endwhile1`" sobald das Ergebnis der Subtraktion 0 ist. Andernfalls arbeitet der MC den nächstfolgenden Befehl ab.

Da der MC recht schnell arebitet, ist obiges "Päuschen" so kurz, das das Auge den Unterschied an den LEDs garnicht richtig mitbekommt. Um die LEDs auch sichtbar alternierend blinken zu lassen, wäre eine ineinandergeschachtelte Schleifenkonstruktion erforderlich so ähnlich wie:

```
ldi r17, 0
ldi r18, 0
ldi r19, 30
delay:
        dec r17
        brne delay
        dec r18
        brne delay
        dec r19
        brne delay
```


Aus dieser Schleife kommt der MC erst wieder raus, wenn der Wert in Register 19 auf 0 dekrementiert wurde. Und das dauert eine Weile ...

4. Bit-Operationen:

Soweit zum Thema Programmieren. Jetzt manipulieren wir mal gezielt Bits in einem Register, wenn eine Zahl in dem Register schon geladen ist. Es werden also gezielt Bits gesetzt (Bit bekommt Wert 1) oder zurückgesetzt (Bit bekommt Wert 0). Das geht und hat recht nützliche Effekte (vorausgesetzt es funktioniert so, wie man sich das vorstellt):

Erst mal die logische Operation AND. Dabei werden 2 Bytes Bit für Bit verglichen und das jeweils resultierende Bit ergibt sich nach folgender Regel :

```
AND     1    1   ->  1
        1    0   ->  0
        0    1   ->  0
        0    0   ->  0
```

Das Ergebnis des logischen Vergleichs ist also nur dann `1` (wahr), wenn beide Bits den Wert `1` hatten. Die entsprechende Assembler-Anweisung lautet "`andi`" (AND immediate).

```
ldi  r16, 0b00001110  ; im Register:    0000 1110
andi r16, 0b00000001  ; im Register:    0000 0000
```

Das Ergebnis dieser Aktivitäten ist also Zahl `0` im Register `r16`. Das unterste Bit bleibt in diesem Fall so wie es war und alle anderen Bits werden zu `0` ("Isolieren" nennt man das ganze).

Bei der logischen Operation `OR` (inklusives `OR`) werden 2 Bytes Bit für Bit verglichen und das jeweils resultierende Bit ergibt sich nach folgender Regel:

```
OR      1    1   ->  1
        1    0   ->  1
        0    1   ->  1
        0    0   ->  0
```

Das Ergebnis des logischen Vergleichs ist also dann 1 (wahr), wenn irgendeins der beiden Bits den Wert 1 hatte. Die entsprechende Assembler-Anweisung lautet "`ori`" (OR immediate). Nur wenn beide Bits 0 sind dann ist das Ergebnis des Vergleichs auch 0.

```
ldi  r16, 0b00001110  ; im Register:    0000 1110
ori  r16, 0b00000001  ; im Register:    0000 1111
```

Das Ergebnis dieser Aktivitäten ist also Zahl 15 im Register `r16`. In diesem Fall wird das unterste Bit immer gesetzt (egal welchen Wert es vorher hatte), die restlichen Bits bleiben so wie vor der Bit-Operation.

Bei der logischen Operation `XOR` ("exklusives oder") werden 2 Bytes Bit für Bit verglichen und das jeweils resultierende Bit ergibt sich nach folgender Regel:

```
XOR     1    1   ->  0
        1    0   ->  1
        0    1   ->  1
        0    0   ->  0
```

Das Ergebnis des logischen Vergleichs ist also dann 1 (wahr), wenn nur eines der beiden Bits den Wert 1 hatte. Wenn beide Bits den Wert 1 hatten, dann ist das Ergebnis 0. Die entsprechende Assembler-Anweisung lautet `eor`.

```
ldi  r16, 0b00001110  ; im Register:    0000 1110
ldi  r17, 0b00001000  ; im Register:    0000 1000
eor  r16, r17         ; im Register:    0000 0110
```

Das Ergebnis dieser Aktivitäten ist also Zahl 6 im Register.

Mit diesen logischen Operationen lassen sich gezielt bestimmte Bits in einem Byte entweder setzen oder zurücksetzen.

Wie setze ich in einem Byte das Bit 3 auf 1 und lasse die restlichen Bits des Byte so wie sie momentan sind?

Dazu braucht man ein Byte, in dem die Bits folgendermaßen aufgebaut sind (das nennt man Bit-Maske) :

```
  0000 1000       nur Bit 3 hat Wert 1, alle anderen Bits haben Wert 0
      *
```

Und dann muß das Register-Byte und das Masken-Byte logisch mit `OR` verknüpft werden.
Der passende Befehl lautet:

```
ldi r16, 0b11000011  ; im Register:    1100 0011
ori r16, 0b00001000  ; im Register:    1100 1011
                       *
```

Danach ist im Byte von Register r16 das Bit Nummer 3 auf jeden Fall auf 1 gesetzt, egal welchen Wert das Bit vorher hatte. Die restlichen Bits bleiben so, wie sie sind.

Wie bekomme ich eine 0 in die Bits 3 und 4 in einem Register-Byte und lasse die restlichen Bits des Byte so wie sie momentan sind ?

Man nehme dazu ein Byte-Maske, in dem die Bits folgendermaßen aufgebaut sind

```
  1110 0111       nur Bit 3 und 4 haben Wert 0, alle anderen Bits haben Wert 1
    * *
```

Und dann muß das Register-Byte und das Masken-Byte logisch mit `AND` verknüpft werden.
Der passende Befehl lautet:

```
ldi  r16, 0b01010101  ; im Register:    0101 0101
andi r16, 0b11100111  ; im Register:    0100 0101
                          * *
```

Danach ist im Byte von Register `r16` das Bit Nummer 3 und 4 auf jeden Fall auf 0 gesetzt, egal welche Werte die Bits vorher hatten. Die restlichen Bits bleiben so, wie sie sind.

Praktisch sind diese Bit-Masken, weil damit alle Bits eine Byte mit einem einzigen Befehl gesetzt werden könen.

Für die unteren 32 I/O-Register (ports) gibt es einen eigenen Befehl, um gezielt ein Bit auf 0 oder 1 zu setzen. Mit folgendem Befehl wird in einem UART Control-Register das Bit `RXEN` auf 1 gesetzt.

```
sbi UCSRB, RXEN ; set bit in IO-register
```

Damit wird ein Empfang von Daten über die serielle Schnittstelle ermöglicht. Die Namen `UCSRB` und `RXEN` sind definiert in 4433def.inc, die in dem Assembler-Programm eingebunden sein muß (am besten hält man eine Kopie von 4433def.inc im gleichen Verzeichnis wo auch der Quellcode des Programms steht).


```
.equ  UCSRB = $0a
.equ  RXEN  = 4
```

`UCSRB` steht also für Adresse `Hex 0A` und `RXEN` steht für Bit Nummer 4.

Mit folgendem Befehl wird in einem UART Control-Register das Bit `RXEN` auf 0 zurückgesetzt und der MC liest keine Daten mehr von der Schnittstelle.

```
cbi UCSRB, RXEN ; clear bit IO-Register
```

OK, und wie kann ich alle Bits eine Byte umdrehen, so daß alle 1 zu 0 werden und umgekehrt?
Antwort: mit dem Einser-Komplement.

```
ldi r16, 0b11000011  ; im Register:    1100 0011
com r16              ; im Register:    0011 1100
```

So, und was kann man mit diesen Bits noch so alles machen? Man kann die Bits eines Register-Byte auch nach links und rechts schieben. Die Befehle lauten
`lsl` (logical shift left) und `lsr` (logical shift right).

```
ldi r16, 0b00001010 ; im Register: 00001010                        dezimal: 10
lsl r16             ; im Register: 00010100   <-- links geschoben;  dezimal: 20
lsr r16             ; im Register: 00001010   --> rechts geschoben; dezimal: 10
```

Wenn die Bits eine Stelle nach links geschoben werden, dann entspricht das einer Multiplikation mit 2. Von rechts wird dabei immer eine 0 eingeschoben. Das niederwertigste Bit hat also nach dem Schieben immer den Wert 0. Das nach links rausgeschobene Bit wird im Carry-Flag des Status-Registers gespeichert.

Wenn die Bits eine Stelle nach rechts geschoben werden, dann entspricht das einer Division durch 2. Von links wird 0 reingeschoben, das nach rechts rausgeschobene Bit wird zum Carry-Flag.

Dann kann man die Bits noch im Kreis rotieren lassen. Die Befehle sind `rol` (rotate left through carry) und `ror` (rotate right through carry). Dabei wird der Wert des Carry-Flags jeweils reingeschoben und das rausgeschobene Bit wird im Carry-Flag gespeichert.

```
                      Carry-Flag  Register   Carry-Flag
                      zu Beginn              am Ende
ldi r16, 0b01111110 ;     ?       0111 1110     0
rol r16             ;     0       1111 1100     0
rol r16             ;     0       1111 1000     1
rol r16             ;     1       1111 0001     1
```

5. negative Zahlen:

Jetzt zu den Zahlen mit Vorzeichen, also den negativen und positiven Zahlen (signed numbers).

In diesem Fall muß irgendwo in den 8 Bits eine Information stehen, ob die betreffende Zahl positiv ist (Vorzeichen +) oder negativ ist (Vorzeichen -). Dafür hat man sich das höchstwertige Bit ausgesucht, das Bit ganz links sozusagen. Wenn dieses Bit 0 ist, dann ist die Zahl positiv, wenn dieses Bit 1 ist, dann ist die Zahl negativ.

Jetzt gibt es aber ein kleines Problem: wie kann dann noch die Zahl 255 dargestellt werden, bei der ja das Bit Nummer 7 auch auf 1 gesetzt ist ?
Antwort: in einem Byte kann eine vorzeichenbehaftete +255 überhaupt nicht dargestellt werden. Die obere Grenze bei 8 Bits und Vorzeichen ist +127, die untere Grenze liegt dafür nicht mehr bei 0 sondern bei -128.

Allerdings ist -1 nicht dargestellt mit `1000 0001`, wie man ja durchaus vermuten könnte. Sondern die negative Zahl wird durch ein sogenanntes Zweier-Komplement dargestellt. Das ganze funktioniert dann so, als ob man von 0 die jeweilige (absolute) Zahl abzieht.

```
0000 0011    +3
0000 0010    +2
0000 0001    +1
0000 0000    +0
1111 1111    -1
1111 1110    -2
1111 1101    -3
```

Dazu gilt eine einfache Regel, um eine positive Zahl in eine negative Zahl zu verwandeln: man drehe alle Bits um (von 0 nach 1 und von 1 nach 0) und addiere am Ende noch `0000 0001` und schon hat man die richtige Bit-Folge für die negative Zahl. Nehmen wir als Beispiel die Zahl +3, die wir in -3 wandeln:

```
  dezimal:          binär:
    +3             0000 0011

                   1111 1100 + 1
    -3           = 1111 1101
```

Es gibt auch einen Befehl, um eine solche Negation zu erreichen (sinnigerweise `NEG`).

```
ldi r16, 3  ;  dezimal +3   Register: 0000 0011
neg r16     ;  dezimal -3   Register: 1111 1101
```

Dumme Frage: wie kann der MC eigentlich feststellen, ob die Zahl in einem Register jetzt positiv ist oder negativ ? Immerhin kann binär `1111 1111` die vorzeichenlose 255 darstellen oder auch -1. Der Programmierer weiß, um was es geht, aber wie kriegt der MC das raus? Antwort: der MC kann es überhaupt nicht feststellen. Der MC weiß nie, ob die Zahl in dem Register im Assembler-Programm jetzt als vorzeichenbehaftet interpretiert wird oder nicht.

Und da der MC nicht weiß was läuft, setzt er bei Rechenoperationen wie Addition und Division immer auch alle Flags, die für vorzeichenbehaftete Zahlen notwendig sind. Im ungünstigsten Fall macht der MC etwas, was vom Programmierer in dem jeweiligen Zusammenhang nicht gebraucht wird.

Für vorzeichenbehaftete Zahlen sind folgende Flags des Status-Registers relevant:

```
Status-Register: Bit 7
                 Bit 6
                 Bit 5
                 Bit 4   S  Sign flag
                 Bit 3   V  Overflow flag
                 Bit 2   N  Negative flag
                 Bit 1   Z  Zero flag
                 Bit 0   C  Carry flag
```

Der Carry-Flag wird gesetzt, wenn die Bit-Folge von `0000 0000` auf `1111 1111` dreht (bzw. umgekehrt).

Der Zero-Flag wird gesetzt, wenn das Ergebnis der Rechenoperation Null ist.

Der Negativ-Flag wird gesetzt, wenn das Ergebnis der Rechenoperation eine negative Zahl sein *könnte*. Der Negativ-Flag ist 1 bei allen Operationen, bei denen das Ergebnis zwischen folgenden Bit-Folgen liegen:

```
      binär:       dezimal:
      1111 1111      -1
      1111 1110      -2

  ...

      1000 0001      -127
      1000 0000      -128
```

Der Overflow-Flag wird gesetzt, wenn die Zahl im Register wechselt zwischen der höchsten positiven Zahl und der negativsten Zahl (das heißt +127 und minus 128).

```
      binär:       dezimal:
      0111 1111      +127     positivste Zahl
      1000 0000      -128     negativste Zahl
```

```
Beim Inkrementieren sieht das dann so aus :

dezimal:  Register:               Carry Negativ Overflow
  +126    0111 1110                 ?     ?        ?
                     + 0000 0001
  +127    0111 1111                 0     0        0
                     + 0000 0001
  -128    1000 0000                 0     1        1     !!!
                     + 0000 0001
  -127    1000 0001                 0     1        0
                     + 0000 0001
```

6. “große” Zahlen:

Hmmm ..., wenn in einem Register nur maximal eine Zahl bis 255 dargestellt werden kann, wie kann ich dann die Zahl 444 speichern?
Antwort: man nimmt halt 2 Register.

```
ldi r16, 0b10111100  ;  dezimal: 188 = 444 - 256
ldi r17, 0b00000001  ;  dezimal:   1
```

Huch, und wo kommt jetzt die 256 her, die von 444 abgezogen wird?
Antwort: wenn `LSB` (Bit 0) im dem oberen Byte (in `r17`) auf 1 steht und alle Bits im unteren Byte (in `r16`) auf 0 stehen, dann entspricht das bei zusammengefaßter Schreibweise dieser Bitfolge:

```
0000 0001 0000 0000

| r17    | r16     |
```

Und damit wird die Dezimalzahl 256 dargestellt. Das heißt mit anderen Worten: von der Zahl 444 ist anteilig 256 codiert in dem oberen Byte (in `r17`) und in dem unteren Byte (in `r16`) braucht dann nur noch der restliche Anteil codiert zu werden, besagte 188.

Wenn man jetzt beide Register `r17:16` bei Rechenoperationen zusammen behandelt, dann stellen sie zusammen die Zahl 444 dar. Dem MC ist das übrigens vollkommen egal, der kriegt das gar nicht mit. Daß die Register `r16` und `r17` logisch zusammengehören, das weiß nur der Programmierer.

```
dezimal:   binär:
  444      0000 0001 1011 1100
          |   R17   |   R16   |
```

Das ganze sieht zwar aus wie von hinten durch die Brust ins Auge, aber damit können jetzt fast beliebig große Zahlen dargestellt werden, solange wie die Register ausreichen.

Diese “Großzahlen” können auch addiert werden. Im folgenden Beispiel wird dezimal 444 addiert zu 444:

```
ldi r16, 0b10111100  ;  dezimal: 188
ldi r17, 0b00000001  ;  dezimal:   1
ldi r18, 0b10111100  ;  dezimal: 188
ldi r19, 0b00000001  ;  dezimal:   1

add r16, r18
adc r17, r19
```

`r16` und `r17` haben jetzt (zusammen betrachtet) das Resultat der Addition.

```
dezimal:   binär:
  888      0000 0011 0111 1000
          |   R17   |   R16   |
```

Die Addition erfolgt also hier 2-stufig :

1) mit `ADD` werden erst die niederwertigen Bytes addiert. Dabei wird vom MC automatisch der Carry-Flag gesetzt, wenn die Summe der beiden Bytes größer als 255 ist.

2) mit `ADC` (add with carry) werden dann die höherwertigen Bytes addiert. Der Trick besteht hier darin, den Übertrag von der ersten Addition auch noch dazuzuzählen, deswegen ADC.

Das funktioniert auch mit Subtraktion, nur lauten die Assembler-Befehle dann `sub` für Subtraktion und `sbc` für Subtraktion über Carry.

```
ldi r16, 0b10111100  ;  dezimal: 188
ldi r17, 0b00000001  ;  dezimal:   1
ldi r18, 0b10111100  ;  dezimal: 188
ldi r19, 0b00000001  ;  dezimal:   1

sub r16, r18
sbc r17, r19
```

Danach ist der Wert von Register `r16` und `r17` jeweils `0000 0000`.

7. Direktiven und Ausdrücke:

Wenn der Assembler Programmcode liest, dann interpretiert er den Programmtext und generiert ablauffähigen Maschinencode. Dabei kann er auch Ausdrücke sofort ausrechnen wie zum Beispiel in

```
ldi r16, 1 + 2 + 3   ;  Addition
```

Der Assembler interpretiert den Ausdruck “1 + 2 + 3” und kommt zum Ergebnis “6”. Er behandelt die Anweisung so, als wenn im Programm in dieser Zeile gestanden hätte:

```
ldi r16, 6
```


Der Assembler versteht dabei alle gängigen arithmetischen Ausdrücke, auch Bit-Operationen kann er gleich umsetzen:

```
ldi r16, (1<<7) + (1<<3)  ; Bits soundsoviel Stellen nach links schieben
```

Obige Anweisung wird also vom Assembler so behandelt, as wenn im Programm gleich gestanden hätte:

```
ldi r16, 0b10001000
```

Dann gibt es noch eine Reihe von Direktiven, mit denen die Generierung des Maschinencodes beinflußt werden kann. Diese Direktiven fangen mit einem Punkt an und sind bei avrasm erläutert in dem “AVR Assembler User Guide”. Mit “.def” kann man zB. Registern Namen zuordnen und mit “.equ” lassen sich Ausdrücke benennen. Die entsprechenden Namen können dann im weiteren Programm benutzt werden.

```
.def temp = r16       ; eigener Name für Register 16
.equ quartz = 3686400 ; Standard-Taktfrequenz beim STK500

ldi temp, quartz / (9600*16) - 1  ; damit kann dann die Baud-Rate des UART eingestellt werden
```

8. Stack und Interrupts:

Der Vollständigkeit sei erwähnt, daß sich “richtige” Assembler-Programme am Anfang gleich um Interrupts und den stack pointer (Stapelzeiger) kümmern sollten. Das könnte zB. so ähnlich aussehen, wenn der MC Daten von der seriellen Schnittstelle (UART) lesen soll:

```
.def temp = r16
.equ quartz = 3686400 ; für STK500
; .equ quartz = 4000000 ; für 4 MHz

rjmp main     ; Reset Handler : erste Stelle im Maschinencode, hier fängt der MC immer an
reti          ; IRQ0 Handler
reti          ; IRQ1 Handler
reti          ; Timer1 Capture Handler
reti          ; Timer1 compare Handler
reti          ; Timer1 Overflow Handler
reti          ; Timer0 Overflow Handler
reti          ; SPI Transfer Complete Handler
rjmp receive  ; UART RX Complete Handler : hierhin springt MC sofort, wenn am UART ein Byte an-
kam
reti          ; UDR Empty Handler
reti          ; UART TX Complete Handler
reti          ; ADC Conversion Complete Interrupt Handler
reti          ; EEPROM Ready Handler
reti          ; Analog Comparator Handler

main:         ; main program starts here
      ldi temp, RAMEND
      out SP, temp        ; setze stack pointer an Ende des Programmspeichers
      sbi UCSRB, RXEN     ; enable UART receive
      sbi UCSRB, RXCIE    ; enable UART receive interrupt
      sbi UCSRB, TXEN     ; enable UART transmit
      ldi temp, quartz / (9600*16) - 1
      out UBRR, temp      ; setze baud rate
      sei                 ; aktiviere interrupts

loop:
      rjmp loop           ; main loop
```

```
;---------------------------------------------------
; Unterprogramm als Interrupt-Handler
;---------------------------------------------------
receive:
       in temp, UDR        ; lies ein Byte
       rcall transmit      ; schicke es gleich wieder zurück
reti                       ;  wie ret, schaltet aber sofort globalen Interrupt-Flag wieder ein


;---------------------------------------------------
; Unterprogramm, das ein Byte über UART sendet
;---------------------------------------------------
transmit:
  sbis UCSRA,UDRE          ; Warten, bis UDR bereit ist
  rjmp transmit
  out UDR, temp
ret          ; Programm geht zurück zu "rcall" und macht beim nächstfolgenden Befehl weiter
```

Der "stack" (stapel bzw. Haufen) wird in dem obigen Programm gebraucht für den Aufruf des Unterprogramms "transmit".

```
       rcall transmit      ; schicke es gleich wieder zurück
```

Und zwar hat der Befehl `rcall` die nette Eigenschaft, zu einem Unterprogramm zu verzweigen, dasselbe auszuführen und dann wieder im Programmablauf automatisch zurückzuspringen und den Befehl nach dem `rcall` auszuführen. OK, ganz so automatisch ist es nicht, man muß in dem Unterprogramm mit dem Befehl `ret` (return) etwas nachhelfen.

Damit der MC an die richtige Stelle zurückspringt, merkt er sich diese Programm-Adresse, indem er sie im SRAM in 2 Bytes speichert. Und zwar speichert der MC die Bytes an der Adresse, zu der der stack pointer zeigt. Das ist ein spezielles Register namens SP, und dieses Register sollte man bei Programmstart initialisieren mit der höchstmöglichen SRAM-Adresse, da der MC diesen stack von oben nach unten "wachsen" läßt. Beim Rücksprung von dem Unterprogramm liest der MC in SP ein Byte, das er als SRAM-Adresse interpretiert.
Er liest dann die beiden Bytes ab besagter Adresse und damit kennt er die Adresse im Programmspeicher, von der er den nächsten auszuführenden Befehl lesen kann. Vorher läßt er aber den stack automatisch wieder "schrumpfen", der stack pointer wird also zweimal inkrementiert. Schematisch sieht das dann ungefähr so aus (`RAMEND` hat beim 4433 den Wert `$DF`).

```
                  vor Aufruf des    vor Rücksprung zum             nach Rücksprung zum
                  Unterprogramms:   Hauptprogramm:                 Hauptprogramm:

SP:                    RAMEND            RAMEND - 2                      RAMEND


SRAM:   RAMEND - 2     ???               ???                              ???
SRAM:   RAMEND - 1     ???               high byte der Rücksprungadresse  ???
SRAM:   RAMEND - 0     ???               low  byte der Rücksprungadresse  ???
```

In obigem Programm ist ein Interrupt aktiviert. Und zwar wird der MC automatisch aktiv, sobald an der seriellen Schnittstelle ein Byte angekommen ist. Er unterbricht (daher die bezeichnung "interrupt") dann den normalen Progammablauf, läßt alles stehen und liegen und verzweigt sofort im Programm-Speicher an die 9. Adresse. Daß es in diesem Fall immer die 9. Adresse ist, das ist im MC fest "verdrahtet" und läßt sich auch nicht ändern. Und an diese Programm-Adresse ist dann ein `rjmp` Befehl zu setzen, der zu einem Unterprogramm springt, das diesen "Unterbrechnungs-Fall" dann weiter bearbeitet. Wenn der MC dann auf den Befehl `reti` stößt, dann weiß er, daß er im normalen Programmablauf wieder weitermachen kann, so als wenn der "Unterbrechnungs-Fall" garnicht passiert wäre. Und damit er wieder in das normale Progamm zurückspringen kann, hat er sich

sinnvollerweise die entsprechende Programm-Adresse gemerkt. Und auch in diesem Fall speichert er die 2 Bytes der Programm-Adresse auf dem stack und liest sie vor dem Rücksprung von dort wieder zurück.

Wenn obiges Programm ausgeführt wird, dann fängt der MC bei der ersten Adresse im Programm-Speicher an. Dort steht der Befehl

```
       rjmp main     ; Reset Handler
```

und damit werden die Interrupt-Sprungadressen bei Einschalten des MC einfach übersprungen.

9. Maschinencode :

Zu guter letzt noch etwas für Neugierige, die es ganz genau wissen wollen. Wie sieht dieser Machinencode (auch opcode genannt) eigentlich aus, mit dem der MC gefüttert wird ?

```
.include "4433def.inc"

       rjmp weiter  ; überspringe die nop's
       nop
       nop

weiter:
       ldi r16, 0xFF
       out DDRB, r16 ; PortB als output
       ldi r16, 0x00
       out DDRD, r16 ; PortD als input

loop:
       in r16, PIND   ; lies Taster-Stellungen (lies alle Pins an Port D )
       out PORTB, r16 ; lasse entsprechende LED's leuchten (schreibe zu Pins an Port B)
       rjmp loop
```

Obiges Programm sieht im Programmspeicher des MC folgendermaßen aus. Dabei wird hexadezimale Schreibweise benutzt mit Zeilenumbruch nach jeweils 2 Byte. Für jeden Maschinenbefehl (opcode genannt) wird auch der Quellcode des Assemblerprogramms angegeben.

```
opcode:     Quellcode:
-------     -------------
c002         rjmp weiter
0000         nop
0000         nop
ef0f         ldi r16, 0xFF
bb07         out DDRB, r16
e000         ldi r16, 0x00
bb01         out DDRD, r16
b300         in r16, PIND
bb08         out PORTB, r16
cffd         rjmp loop
```

Der Assembler generiert also für jede Programmzeile einen Maschinencode, der in 2 Bytes reinpaßt. Leerzeilen/Kommentarzeilen werden ignoriert, das leuchtet ein. Aber für die Anweisung `loop:` gibt es ja gar keinen korrespondierenden Befehl im Programmspeicher. Hat der Assembler da was vergessen? Scheinbar nicht, denn das Prgramm funktioniert. Diese Anweisung mit dem abschließenden Doppelpunkt wird als "label" benutzt, also wie das Ding, das an Kaufhauswaren hängt und die Ware näher bezeichnet. Im Programm kann damit eine Addresse im Programmspeicher spezifiziert werden, ohne daß man sich als Programmierer diese Addresse mühsam ausrechnen muß. Diese Arbeit übernimmt der Assembler beim Übersetzen des Quellcodes.

Schaun wir uns doch mal die erste Anweisung an:

```
rjmp weiter
```

Entsprechend dem Befehlssatz (instruction set) des AVR wird diese Anweisung übersetzt in folgenden 16-bit opcode:

```
1100 kkkk kkkk kkkk
```

1100 ist dabei fest vorgegeben und ist die "Kennung" für rjmp (relative jump). Wenn der MC einen Befehl liest, der mit dieser Bitfolge anfängt, dann weiß er, daß im Programmspeicher an eine bestimmte Adresse zu springen hat und dort den nächsten Befehl holen soll. Mit den k's wird eine Zahl codiert, die angibt, wieviele Programmbefehle vor- bzw. zurückgesprungen werden soll. Mit k wird also keine absolute Adresse definiert sondern ein relativer Sprung spezifiziert. Die Syntax für diesen Befehl ist

```
PC <-- PC + k + 1
```

PC steht dabei für program counter und bezeichnet die Nummer des aktuell in Bearbeitung stehenden Befehls.

Konkret ist der Maschinenbefehl, der also im Programmspeicher steht, in diesem Fall hexadezimal

```
c002
```

Binär ausgedrückt ist das

```
1100 0000 0000 0010
```

und damit ist die relative Sprung-Adresse binär

```
 0000 0000 0010
```

Damit ist dezimal die Zahl 2 codiert, die den Sprung vorgibt. Konkret bedeutet dies, daß der MC vom aktuell bearbeiteten Befehl im Programmspeicher 2 Befehle weiterspringen soll und den darauffolgenden Befehl zu lesen und auszuführen hat. Und das ist die Programmanweisung `ef0f`, also ganz so wie geplant.

Die letzte Anweisung in dem Programm ist auch eine relative Sprung-Anweisung, aber diesmal wird im Programmablauf rückwärts gesprungen.

```
rjmp loop
```

Hex `cffd` ist binär

```
1100 1111 1111 1101
```

und damit ist die relative Sprung-Adresse binär `1111 1111 1101` bzw. dezimal `-3` (um das zu verstehen, muß man die Sache mit den Zahlen und den Vorzeichen lesen).

Für den MC ist der Programmcode `cffd` eine Anweisung, vom aktuell bearbeiteten Befehl im Programmspeicher 3 Befehle zurückspringen und den darauffolgenden Befehl zu lesen und auszuführen. Und das ist die Programmanweisung `b300`, also auch wieder so wie vorgesehen.

10. Befehlssatz

Und hier noch für die Leute, die ernsthaft zu Ergebnissen kommen wollen, der komplette Befehlssatz des AT90S4433 in alphabetischer Reihenfolge zum Auswendiglernen.

Bei Konstanten wird jeweils der größtmögliche Wert angegeben (255 bedeutet 0 bis 255).
`r0, r1` etc sind Arbeits-Register; `r25:24` ist Registerpaar `r25` (high byte) und `r24` (low byte).
`io31, io63` etc. sind Input-Output-Register (IO-Register), `SREG` bedeutet Status-Register

Wenn `r16` als Parameter angegeben ist, dann gilt der Befehl nur für die oberen 16 Register.
Wenn `io31` als Parameter angegeben ist, dann gilt der Befehl nur für die unteren 32 IO-Register.
Mit `label` ist die mit Doppelpunkt abgeschlossene Marke im Assembler-Programm gemeint (zB `loop:`).
X-Registerpaar ist `r27:26`, Y-Registerpaar ist `r29:28`, Z-Registerpaar ist `r31:30`

```
adc r1, r0          ; add with carry    (r1 + r0 + C, Ergebnis in r1)
add r1, r0          ; add (r1 + r0, Ergebnis in r1)
adiw r25:24, 63     ; add immediate to word (Registerpaar + Konstante, Ergebnis in Registerpaar)
adiw r27:26, 63     ; add immediate to word (Register-Paar X)
adiw r29:28, 63     ; add immediate to word (Register-Paar Y)
adiw r31:30, 63     ; add immediate to word (Register-Paar Z)
and r1, r0          ; logical AND (r1 AND r0, Ergebnis in r1)
andi r16, 255       ; logical AND with immediate (r1 AND Konstante, Ergebnis in r0)
asr r0              ; arithmetic shift right (Bits nach rechts schieben, MSB beibehalten)
bclr 7              ; bit clear in register SREG (Status-Register)
bld r0, 7           ; bit load from T flag (bit bekommt Wert von T flag)
brbc 7, label       ; branch if bit cleared (in SREG Register)
brbs 7, label       ; branch if bit set (in SREG Register)
brcc label          ; branch if carry cleared (C = 0)
brcs label          ; branch if carry set (C = 1)
breq label          ; branch if equal (Z = 1) nach Vergleich bzw. Subtraktion von Zahlen
brge label          ; branch if greater or equal (S = 0) nach Vergleich von Zahlen mit
Vorzeichen (signed)
brhc label          ; branch if half carry cleared (H = 0)
brhs label          ; branch if half carry set (H = 1)
brid label          ; branch if global interrupt disabled (I = 0)
brie label          ; branch if global interrupt enabled (I = 1)
brlo label          ; branch if lower (C = 1) nach Vergleich von Zahlen ohne Vorzeichen
brlt label          ; branch if less than (S = 1) nach Vergleich von Zahlen mit Vorzeichen
(signed)
brmi label          ; branch if minus (N = 1)
brne label          ; branch if not equal (Z = 0)
brpl label          ; branch if plus (N = 0)
brsh label          ; branch if same or higher (C = 0) nach Vergleich von Zahlen ohne Vorzeichen
brtc label          ; branch if transfer flag cleared (T = 0)
brts label          ; branch if transfer flag set (T = 1)
brvc label          ; branch if overflow flag cleared (V = 0)
brvs label          ; branch if overflow flag set (V = 1)
bset 7              ; bit set in SREG register
bst r0, 7           ; bit store in transfer flag (T flag bekommt Wert von Bit)
cbi io31, 7         ; clear bit in IO-Register (nur untere 32 IO-Adressen)
cbr r16, 255        ; clear bits in register (r16 AND 1-er Komplement von Konstante, Ergebnis in
r16)
clc                 ; clear carry flag (danach C = 0)
clh                 ; clear half carry flag (danach H = 0)
cli                 ; clear interrupt flag (danach I = 0)
cln                 ; clear negative flag (danach N = 0)
clr r0              ; clear bits in register (r0 ist danach 0000 0000)
cls                 ; clear signed flag (danach S = 0)
clt                 ; clear transfer flag (danach T = 0)
clv                 ; clear overflow flag (danach V = 0)
clz                 ; clear zero flag (danach Z = 0)
com r0              ; complement (1-er Komplement, alle Bits werden umgedreht)
cp  r1, r0          ; compare (wie r1 - r0, Register bleiben unverändert)
cpc r1, r0          ; compare with carry (wie r1 - r0 - C, Register bleiben unverändert)
cpi r16, 255        ; compare with immediate (wie r16 - Konstante, Register bleibt unverändert)
```

```
cpse r1, r0         ; compare and skip if equal (überspringe nächsten Befehl falls Z = 1)
dec r0              ; decrement (subtrahiere 1, Ergebnis in r0)
eor r1, r0          ; exclusive OR (r1 XOR r0, ergebnis in r1)
icall               ; indirect call to subroutine (Inhalt von Z-Paar als Sprung-Adresse)
ijmp                ; indirect jump (Inhalt von Z-Paar als Sprung-Adresse)
in r0, io63         ; input (lese Eingänge eines IO-Ports in Arbeitsregister)
inc r0              ; increment (addiere 1, Ergebnis in r0)
ld    r0,  X        ; load RAM data into register (RAM-Adresse ist von X-Registerpaar vorgege-
ben)
ld    r0,  X+       ; load RAM data into register (post-increment X-Registerpaar)
ld    r0, -X        ; load RAM data into register (pre-decrement X-Registerpaar)
ld    r0,  Y        ; load RAM data into register (RAM-Adresse ist von Y-Registerpaar vorgege-
ben)
ld    r0,  Y+       ; load RAM data into register (post-increment Y-Registerpaar)
ld    r0, -Y        ; load RAM data into register (pre-decrement Y-Registerpaar)
ldd   r0,  Y+63     ; load RAM data with displacement (RAM-Adresse ist Y plus Konstante 0 bis
63)
ld    r0,  Z        ; load RAM data into register (RAM-Adresse ist von Z-Registerpaar vorgege-
ben)
ld    r0,  Z+       ; load RAM data into register (post-increment Z-Registerpaar)
ld    r0, -Z        ; load RAM data into register (pre-decrement Z-Registerpaar)
ldd   r0,  Z+63     ; load RAM data with displacement (RAM-Adresse ist Z plus Konstante 0 bis
63)
ldi   r16, 255      ; load immediate  (Konstante in r16)
lds   r0, 65534     ; load RAM directly from data space  (Adresse ist 0 bis 65534, also 64 KB)
lpm                 ; load program memory to r0 (Adresse in Bits 1 bis 15 von Z, wenn LSB = 0
lese unteres Byte)
lsl   r0            ; logical shift left  (MSB wandert in carry flag, LSB = 0)
lsr   r0            ; logical shift right (LSB wandert in carry flag, MSB = 0)
mov   r1, r0        ; move (kopiert Inhalt von r0 zu r1, danach r1 = r0)
neg r0              ; negate (0 - r0)
nop                 ; no operation (ein Taktzyklus Leerlauf)
or r1, r0           ; logical OR (r1 OR r0, Ergebnis in r1)
ori r16, 255        ; logical OR with immediate (r16 OR Konstante, Ergebnis in r16)
out io63, r0        ; output (schreibe von Register zu Ausgängen eines IO-Ports)
pop  r0             ; pop from stack (Byte vom Stapel lesen und Zeiger um 1 erhöhen)
push r0             ; push to stack (Byte auf Stapel ablegen und Zeiger um 1 reduzieren)
rcall  label        ; relative call (Assembler kalkuliert relative Sprungadresse -2KB bis +2KB)
ret                 ; return from sub-routine
reti                ; return from interrupt routine (setzt Interrupt flag)
rjmp label          ; relative jump (Assembler kalkuliert relative Sprungadresse -2KB bis +2KB)
rol r0              ; rotate left through carry  (LSB von carry, MSB zu carry)
ror r0              ; rotate right through carry (MSB von carry, LSB zu carry)
sbc r1, r0          ; subtract with carry (r1 - r0 - C, Ergebnis in r1)
sbci r16, 255       ; subtract with carry immediate (r16 - Konstante - C, Ergebnis in r16)
sbi  io31, 7        ; set bit in IO-Register (unterste 32 Ports)
sbic io31, 7        ; skip if bit in IO-Register is cleared (nur für untere 32 Ports)
sbis io31, 7        ; skip if bit in IO-Register is set (nur für untere 32 Ports)
sbiw   r25:24, 63   ; subtract immediate from word (Registerpaar - Konstante, Ergebnis in Regis-
terpaar)
sbiw   r27:26, 63   ; subtract immediate from word (X-Registerpaar)
sbiw   r29:28, 63   ; subtract immediate from word (Y-Registerpaar)
sbiw   r31:30, 63   ; subtract immediate from word (Z-Registerpaar)
sbr r16, 255        ; set bits in register (r16 OR Konstante, Ergebnis in r16)
sbrc r0, 7          ; skip if bit in register is cleared
sbrs r0, 7          ; skip if bit in register is set
sec                 ; set carry (danach C = 1)
seh                 ; set half carry flag (danach H = 1)
sei                 ; set interrupt flag (danach I = 1)
sen                 ; set negative flag (danach N = 1)
ser    r0           ; set bits in register (r0 ist danach 1111 1111)
ses                 ; set signed flag (danach S = 1)
set                 ; set transfer flag (danach T = 1)
sev                 ; set overflow flag (danach V = 1)
sez                 ; set zero flag (danach Z = 1)
sleep               ; sleep
st   X , r0         ; store data in RAM from register (RAM-Adresse ist von X-Registerpaar
vorgegeben)
st   X+, r0         ; store data in RAM from register (post-increment X-Registerpaar)
st  -X , r0         ; store data in RAM from register (pre-decrement X-Registerpaar)
st   Y , r0         ; store data in RAM from register (RAM-Adresse ist von Y-Registerpaar
```



```
vorgegeben)
st   Y+, r0         ; store data in RAM from register (post-increment Y-Registerpaar)
st  -Y , r0         ; store data in RAM from register (pre-decrement Y-Registerpaar)
std  Y+63, r0       ; store data in RAM from register with displacement (post-increment Y-Regis-
terpaar)
st   Z , r0         ; store data in RAM from register (RAM-Adresse ist von Z-Registerpaar
vorgegeben)
st   Z+, r0         ; store data in RAM from register (post-increment Z-Registerpaar)
st  -Z , r0         ; store data in RAM from register (pre-decrement Z-Registerpaar)
std  Z+63, r0       ; store data in RAM from register with displacement (post-increment Z-Regis-
terpaar)
sts 65534, r0       ; store data directly from register in data space  (Adresse ist 0 bis 65534,
also 64 KB)
sub r1, r0          ; subtract (r1 - r0, Ergebnis in r1)
subi r16, 255       ; subtract immediate (r16 - Konstante,  Ergebnis in r16)
swap r0             ; swap nibbles (tausche unterste und höchste 4 Bits)
tst r0              ; test for zero or minus (logisches r0 AND r0, r0 unverändert)
wdr                 ; watch dog reset (Hund fängt wieder von vorne an)
```

Einige Regeln:

"Immediate"-Befehle (`andi, cpi, ldi, ori, subi`) gelten nur für obere 16 Arbeitsregister (ab `r16`). Mit Ausnahme von `in` und `out` gelten die IO-Register ansprechenden Befehle gelten nur für die unteren 32 IO-Register. Konkret sind das :

```
$1E  EEAR   EEPROM Address Register
$1D  EEDR   EEPROM Data Register
$1C  EECR   EEPROM Control Register
$18  PORTB  Data Register, Port B
$17  DDRB   Data Direction Register, Port B
$16  PINB   Input Pins, Port B
$15  PORTC  Data Register, Port C
$14  DDRC   Data Direction Register, Port C
$13  PINC   Input Pins, Port C
$12  PORTD  Data Register, Port D
$11  DDRD   Data Direction Register, Port D
$10  PIND   Input Pins, Port D
$0F  SPDR   SPI I/O Data Register
$0E  SPSR   SPI Status Register
$0D  SPCR   SPI Control Register
$0C  UDR    UART I/O Data Register
$0B  UCSRA  UART Control and Status Register A
$0A  UCSRB  UART Control and Status Register B
$09  UBRR   UART Baud Rate Register
$08  ACSR   Analog Comparator Control and Status Register
$07  ADMUX  ADC Multiplexer Select Register
$06  ADCSR  ADC Control and Status Register
$05  ADCH   ADC Data Register High
$04  ADCL   ADC Data Register Low
$03  UBRRHI UART Baud Rate Register High
```

Wenn die Bits der oberen IO-Register geändert werden sollen etc. dann muß das IO-Register zuerst mit “in” in ein Arbeitsregister gelesen werden, dort wird dann mit den Bits gewerkelt und danach wird mit “out” das Arbeitsregister zu dem IO-Register kopiert. Ausnahme ist das Status-Register, für das es separate Befehle zum Setzen und Zurücksetzen der einzelnen Bits gibt. Die oberen IO-Register sind :

```
$3F  SREG   Status REGister
$3D  SP     Stack Pointer
$3B  GIMSK  General Interrupt MaSK register
$3A  GIFR   General Interrupt Flag Register
$39  TIMSK  Timer/Counter Interrupt MaSK register
$38  TIFR   Timer/Counter Interrupt Flag register
$35  MCUCR  MCU general Control Register
$34  MCUSR  MCU general Status Register
$33  TCCR0  Timer/Counter0 Control Register
$32  TCNT0  Timer/Counter0 (8-bit)
$2F  TCCR1A Timer/Counter1 Control Register A
$2E  TCCR1B Timer/Counter1 Control Register B
$2D  TCNT1H Timer/Counter1 High Byte
$2C  TCNT1L Timer/Counter1 Low Byte
$2B  OCR1H  Timer/Counter1 Output Compare Register High Byte
$2A  OCR1L  Timer/Counter1 Output Compare Register Low Byte
$27  ICR1H  Timer/Counter1 Input Capture Register High Byte
$26  ICR1L  Timer/Counter 1 Input Capture Register Low Byte
$21  WDTCR  Watchdog Timer Control Register
```

Und hier noch einige kleine Sachen zur Erläuterung:

; isoliere Bit 0 in Register 1 (stelle Bits 1 bis 7 auf 0, aber ändere unterstes Bit nicht)

```
ldi r0, 0b00000001   ;  bzw. ldi r0, 1
and r1, r0  ; funktioniert bei allen Arbeitsregistern (0 - 31)
```

; setze Bits 0 und 1 in Register 16 auf 1, aber ändere die anderen Bits nicht :

```
sbr r16, 0b00000011   ;  bzw. sbr r16, 3   funktioniert nur bei oberen 16 Arbeitsregistern
```

; schreibe jeweils 0 in die untersten 2 Bits von Register 16, aber ändere die anderen Bits nicht)

```
cbr r16, 0b00000011   ;  bzw. cbr r16, 3   funktioniert nur bei oberen 16 Arbeitsregistern
```

; kopiere Bit 3 von Register 1 zu Bit 4 in Register 2 :

```
bst r1, 3    ; Bit zwischenspeichern im T flag
bld r2, 4    ; T flag lesen
```

; schiebe ein Register 4 mal nach links mit einem einzigen Befehl:

```
swap r0
```

; lade das Byte von Adresse 105 in Register 0 und schreibe es wieder zurück:

```
clr  r27       ; XH auf Null gestellt
ldi  r26, 105  ; XL hat Wert 105
ld   r0, X     ; im Register ist jetzt das Byte von SRAM Adresse 105
add  r0, 11    ; mach etwas mit dem Byte
st   X, r0     ; schreibe das Byte wieder zurück zum SRAM zu Adresse 105
```