Kapitel 04: Das erste Projekt
Abschnitt 04: Überblick über das Programm

Beim Durchgehen des Programms will ich die verschiedenen Abschnitte zeigen, aus denen jedes ATmega-Assemblerprogramm besteht. Da ist es besser, die „Hinweise" im Quelltext der besseren Übersicht halber, wegzulassen:

```
-----  Beginn Quelltext  -----

;*-----------------------------------
; PROGRAMM Lehrgang16_Kap04_V01.asm
; getestet, läuft
;*-----------------------------------

/*

Ergebnisse mit LehrgangTrial04_V01.asm:
 1. Programm funktioniert wie gewünscht

*/
/*
BESCHREIBUNG

Im Versuchsprogramm "LehrgangTrial04_V01.asm" soll LED0
des STK500 eingeschaltet werden.

*/

;-----------------------------------
; Hardware-Definitionen und Macros einbinden
;
.include  "m16def.inc"


;*-----------------------------------
.CSEG
.ORG $0000
;*-----------------------------------
;

INIT:

;-----------------------------------
; MC-Hardware einrichten
;


;-----------------------------------
; Umgebungsfunktionen einrichten
;


; Die LED0 des STK500 sollen über PORTB angesteuert werden
; LED0 soll eingeschaltet werden
;

; Kanal0 von PORTB aus Ausgang einstellen
    in r16,DDRB         ; r16 := Konfigurationsregister für PORTB einlesen
    sbr r16,0b00000001  ; Kanal 0 von PORTB als Ausgang einstellen
    out DDRB,r16        ; geänderte Konfiguration schreiben


HAUPT_PROGRAMM:
; hier beginnt das Hauptprogramm

; die LED0 einschalten
    in r16,PORTB        ; r16 := Werte von PORTB einlesen
    cbr r16,0b00000001  ; Kanal 0 von PORTB auf Null ziehen

    out PORTB,r16       ; modifiziertes Ausgangssignal schreiben

HAUPT_SCHLEIFE:
; hier beginnt die Hauptschleife

    rjmp HAUPT_SCHLEIFE


;-----------------------------------

----  Ende des Quelltextes  -----
```

Es ist eine gute Idee, beim Lesen dieses Abschnitts, das AVRStudio mit dem geladenen Programm geöffnet zu haben. Dann kann man leicht in den AVRStudio-Editor umschalten und dies und das ausprobieren.

Das Programm enthält jede Menge Kommentare, um die Funktion des Programm zu erklären. Das ist auch gut so, denn wenn man später noch einmal verstehen will, was man sich beim Schreiben des Programms überlegt hatte, dann braucht man dazu ohne diese Kommentare sehr viel Zeit. Der AVRStudio-Assembler erlaubt es, Kommentare auf verschiedene Weisen zu kennzeichnen:

1. mit einem vorangestellten Semikolon („;"). Alles, was in derselben Zeile nach dem Semikolon steht, wird beim Assemblieren nicht bearbeitet. Diese Art der Kennzeichnung ist praktisch, um direkt nach einer Anweisung in derselben Zeile ausfürhlichere Erläuterungen zu der Anweisung zu geben; z.B. zu beschreiben, welchem Zweck die Anweisung im Rahmen des Programms erfüllt.
2. ein vorangestellter doppelter Schrägstrich („//") hat dieselbe Wirkung wie ein Semikolon. Statt
   „...
   ; die LED0 einschalten
   in r16,PORTB   ; r16 := Werte von PORTB einlesen
   ..."
   Kann man genausogut schreiben
   „...
   // die LED0 einschalten
   in r16,PORTB   // r16 := Werte von PORTB einlesen
   ..."
   Welche dieser beiden Schreibweisen man verwendet, ist Geschmacksache. Vielleicht hat das „//" den Vorteil, dass man es beim schnellen Überfliegen des Quelltextes stärker ins Auge springt.
3. Sowohl das „;" als auch das „//" haben den Nachteil, dass sie in jeder neuen Zeile wiederholt werden müssen, z.B. hier:

„;*-----------------------------------
 ; PROGRAMM Lehrgang16_Kap04_V01.asm
 ; getestet, läuft
 ;*-----------------------------------
...“
Das ist bei längeren Kommentaren, die sich über mehrere Zeilen erstrecken, sehr umständlich (man muss die Zeilen manuell umbrechen und die Markierungen jedesmal löschen und neu einfügen, wenn sich die Zeilenumbrüche ändern). Deshalb erlaubt AVRStudio Kommentarblöcke über mehrere Zeilen am Anfang mit „/*“ und am Ende mit „*/“ zu kennzeichnen. Beispiel:
„...
 /*

 Ergebnisse mit LehrgangTrial04_V01.asm:
 1. Programm funktioniert wie gewünscht

 */
...“
Zwischen den beiden Markierungen kann man schreiben, was man will - der Assembler nimmt davon keine Notiz.

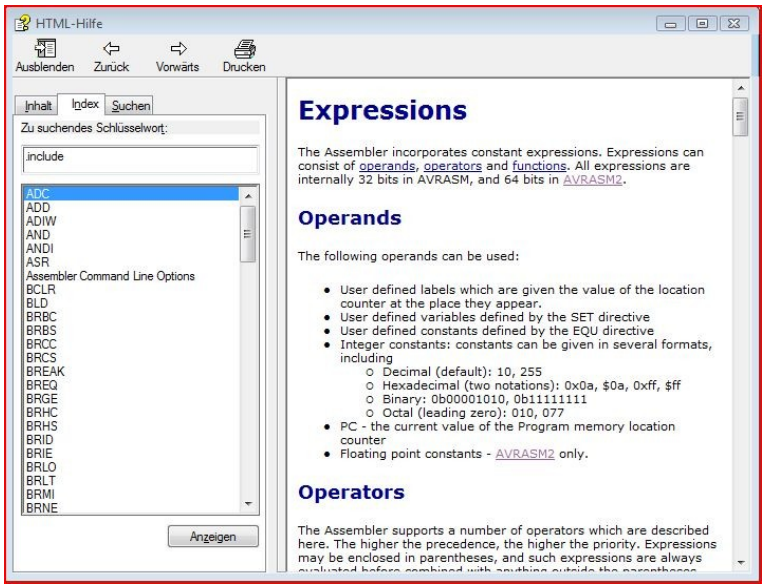
Liesse man in dem Quelltext alle Kommentare weg, dann sähe er so aus:

```
-----  Beginn Quelltext  -----

.include  "m16def.inc"
.CSEG
.ORG $0000

INIT:
    in r16,DDRB
    sbr r16,0b00000001
    out DDRB,r16
    in r16,PORTB
    cbr r16,0b00000001
    out PORTB,r16
HAUPT_SCHLEIFE:
    rjmp HAUPT_SCHLEIFE

----  Ende des Quelltextes  -----
```

Dass so etwas bei längeren Programmen mindestens schwer, wenn nicht völlig unverständlich ist, leuchtet sofort ein. Ohne Kommentare kann man Assembler-Quelltexte nicht verstehen.

Einige Zeilen enthalten Quelltext, der mit einem Punkt („.“) beginnt, z.B.
„...
 .include "m16def.inc"
...“
Anweisungen, die mit einem Punkt beginnen heisssen „Assembler Direktive“, „Pseudoanweisungen“ oder einfach „Direktive“ (Englisch: „assembler directives“). Die dienen der Steuerung des Assemblers und erzeugen selber keine Maschinenanweisungen.

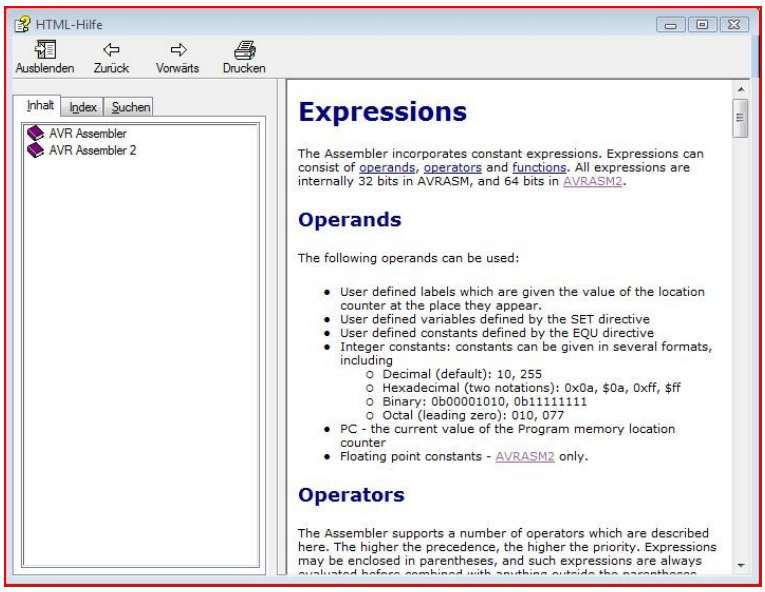
Das ist eine gute Gelegenheit, die Hilfefunktion des AVRStudio auszuprobieren. Wenn man die Cursor an den Anfang oder in das Wort „include“ stellt und dann die Taste „F1“ der Tastatur drückt, geht folgendes Fenster auf (Studio_Hilfe001_V01.jpg):



Das sieht erstmal nicht vielversprechend aus. Zu dem Schlüsselwort „.include“ gibt es im Index offenbar keinen passenden Eintrag.

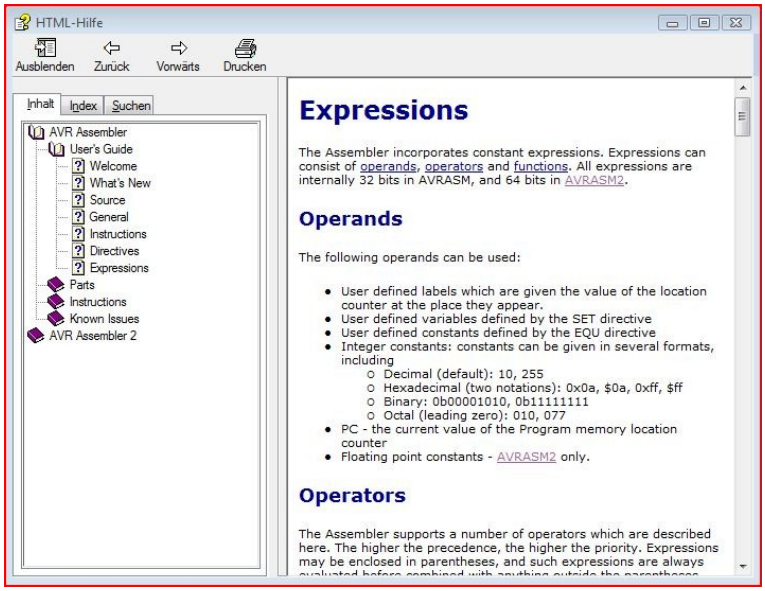
Dafür sieht man aber schon, dass hier Hilfe für jede einzelne Assembler-Anweisung verfügbar ist. Das werden wir später noch ausprobieren.

Um die Hilfe zu der Direktive „.include“ zu finden, muss man im linken Feld auf den Reiter „Inhalt“ klicken. Das führt zu dem Fenster (Studio_DirektiveHilfe02_V01.JPG):



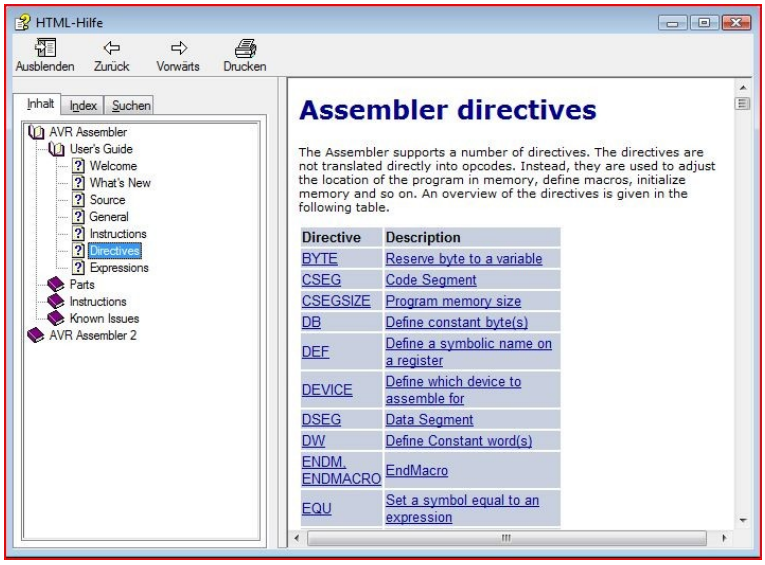
Und jetzt noch einmal im linken Feld auf die Zeile „AVR Assembler" doppelklicken und auf die dann unter „AVR Assembler" erscheinende Zeile „User's Guide" klicken.

Dann erscheint (Studio_DirektiveHilfe03_V01.JPG):



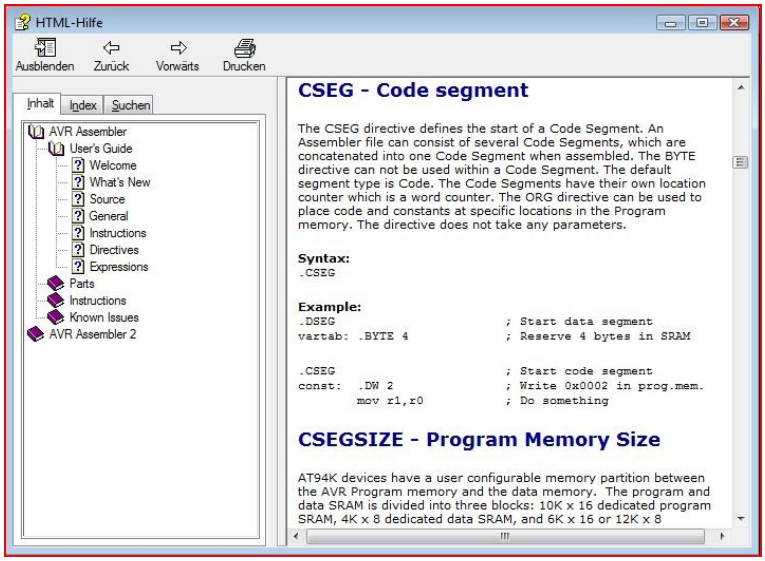
Und richtig, da findet sich in der vorletzen Unterzeile zu „User"s Guide" die Hilfe zu „Direktives".

Doppelklicken auf „Directives" führt zu:



Jetzt sieht man im rechten Feld die Liste aller verfügbaren Direktiven, die der Assembler versteht. Von einigen davon, z.B. „.EQU" werden wir noch sehr ausgiebig Gebrauch machen.

Jetzt noch einmal im rechten Feld auf das unterstrichene Wort „CSEG" klicken bringt (Studio_CSEGHilfe01_V01.JPG):



Hier kann man jetzt sehr ausführlich lesen, wozu die Direktive nützlich ist. Zu den meisten Erklärungen gibt es, wie auch hier, einen kurzen Quelltext als Beispiel.

So, die Suche nach der Hilfe für die Direktiven war vielleicht nicht ganz so geradlinig, wie man sich das vorstellen könnte - aber wenn man's weiss, ist's ganz einfach.

Die „.cseg"-Direktive zeigt dem Assembler also die Stelle im Quelltext an, an der der assemblierbare Programmtext anfängt. Es dient dazu, diesen Bereich von denen zu unterscheiden, die Daten enthalten. Diese Bereiche werden mit der Direktiven „.dseg" gekennzeichnet. Beim Assemblieren werden alle Daten- und Programmtext zu zusammenhängenden Blöcken zusammengefasst. Der Programmtext wird assembliert und ins Flash-RAM geschrieben, der Datentext landet im RAM. Man kann mit Hilfe dieser beiden Anweisungen also Daten und Programmtext mischen, und trotzdem genau steuern, ob sie im RAM oder im Flash-RAM abgelegt werden.

In unserem Programm zeigt die Anweisung nur den Beginn des Programmtextes an. Es übrigens egal, ob die Direktiven mit Gross- („.CSEG") oder Kleinbuchstaben („.cseg") geschrieben werden; sie werden in jedem Fall richtig verarbeiet. Ich ziehe Kleinbuchstaben vor, weil das beim Tippen einfacher ist. An dieser Stelle haben sich die Grossbuchstaben erhalten, weil ich den Anfangsteil meiner Programme normalerweise mit „copy-paste" von einem ins andere übertrage.

Auch die Erklärung zu der „.ORG" findet sich in der Hilfe. Die „.org"-Direktive legt die Speicheradresse fest, an der der folgende Quelltext, egal ob Daten oder Programm, im zugeordneten Speichertyp beginnt. In unserm Fall ging die „.cseg"-Direktive voraus; deshalb bezieht sich die „.org"-Direktive auf den Programmspeicher, das Flash-RAM. Wenn nach dem „.org" nichts folgt, wird als Adresse die erste verfügbare Adresse in dem Speicher genommen. Im Fall des Programmspeichers beim ATMega ist das die 0x0000. Die beiden Direktiven zusammengenommen bedeuten also nicht mehr, als dass hier der Programmtext anfängt und dass er nach dem Assemblieren beginnend mit von der Adresse 0x0000 an im Flash-RAM gespeichert werden soll.

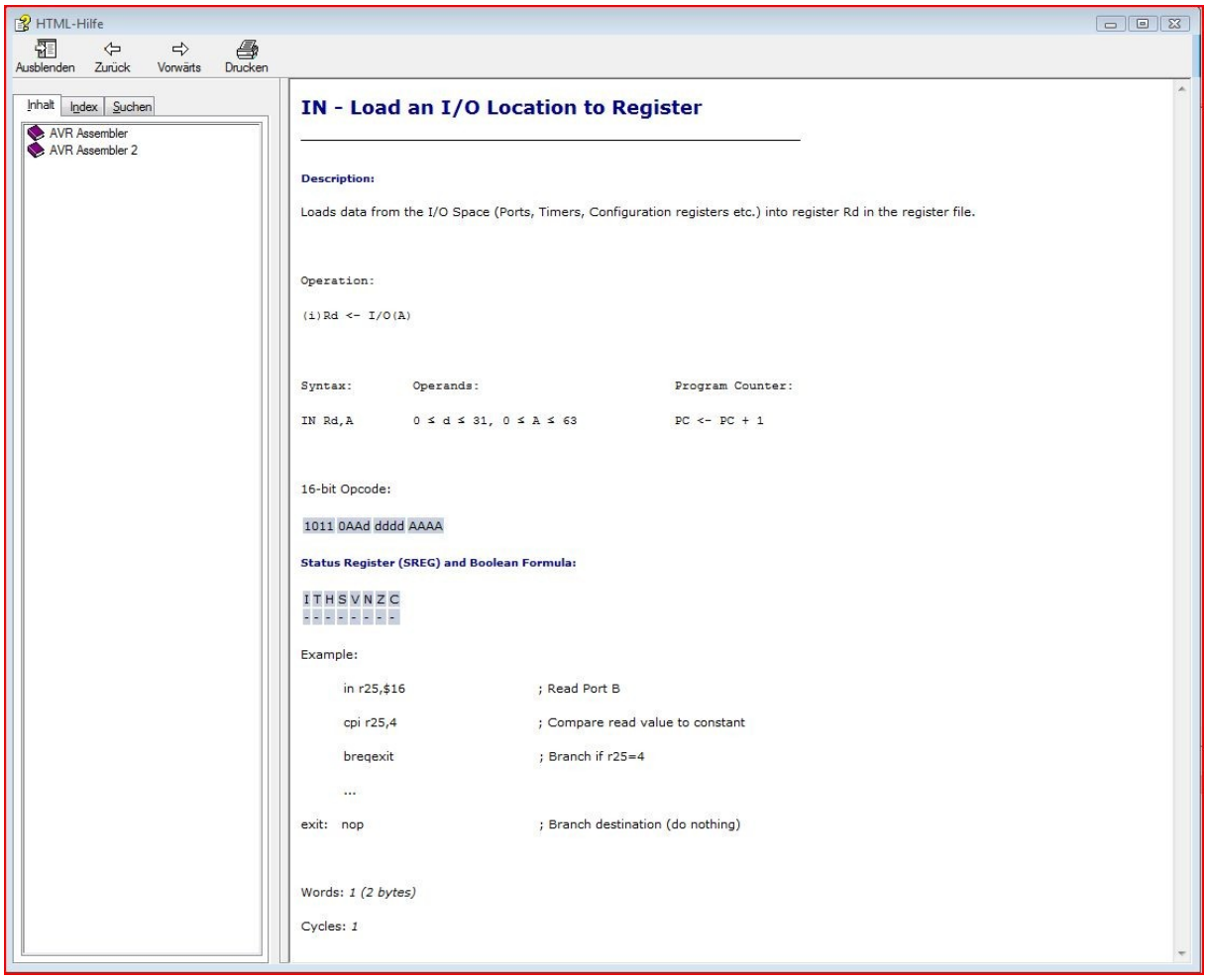
Obwohl die Direktiven „.org", „.cseg" und „.dseg" mehrfach im Quelltext vorkommen dürfen, verwende ich sie nur an dieser Stelle. Wenn man erzwingt, dass bestimmte Daten oder Programmteile an festgelegten Speicheradressen stehen, dann führt das früher oder später zu Konflikten. Das trifft besonders für Programme zu, die aus mehreren Modulen zusammengesetzt sind - wie wir das ja auch machen wollen. Bei modularen Konzepten sind relative Adressierungen sehr viel praktischer. Da verschieben sich die Adressen mit den Modulen. Und für das Rervieren von Speicherplatz im RAM verwende ich eine andere Methode, die ohne die „.dseg"-Direktive auskommt und eine übersichtlichere RAM-Aufteilung zulässt.

Im Quelltext tauchen auch noch Worte auf, denen in derselben Zeile direkt ein Doppelpunkt folgt, z.B. „INIT:" und „HAUPTSCHLEIFE:". Das sind Sprungmarken. Solche Sprungmarken schreibe ich immer nicht eingerückt, dann fallen sie besser ins Auge und erleichtern einem das Verfolgen des Programmablaufs.

Jetzt wollen wir noch ausprobieren, wie das AVRStudio uns bei den normalen Assembler-Anweisungen unterstützt. Nehmen wir doch die Zeile

```
„...
 in r16,DDRB         ; r16 := Konfigurationsregister für PORTB einlesen
..."
```

und stellen den Cursor auf oder unmittelbar vor das Wort „in"; es geht auch, wenn man das Wort durch Doppelklick hervorhebt. Anschliessend wird die Taste „F1" auf der Tastatur gedrückt. Hervor kommt



HTML-Hilfe

Ausblenden Zurück Vorwärts Drucken

Inhalt Index Suchen

AVR Assembler
AVR Assembler 2

IN - Load an I/O Location to Register

Description:

Loads data from the I/O Space (Ports, Timers, Configuration registers etc.) into register Rd in the register file.

Operation:

(i)Rd <- I/O(A)

Syntax:	Operands:	Program Counter:
IN Rd,A	0 ≤ d ≤ 31, 0 ≤ A ≤ 63	PC <- PC + 1

16-bit Opcode:

1011 0AAd dddd AAAA

Status Register (SREG) and Boolean Formula:

I T H S V N Z C
- - - - - - - -

Example:

```
        in r25,$16          ; Read Port B
        cpi r25,4           ; Compare read value to constant
        breqexit            ; Branch if r25=4
        ...
exit:   nop                 ; Branch destination (do nothing)
```

Words: *1 (2 bytes)*

Cycles: *1*

Das ist doch ziemlich ausführlich! Die Überschrift gibt so etwas wie die Langform des Anweisungstextes: „IN - lädt [den Inhalt] einer I/O-Adresse in ein Register" lautet die Übersetzung [mit Ergänzungen].

Unter „description" kommen dann ausführlichere Erklärungen: „Lädt Daten aus dem I/O-Bereich [des RAM] (Ports, Timer, Konfigurationsregister [z.B. von UART, TWI usw.] in das Register Rd des Register-Blocks". „Rd" ist die Kurzform von „Zielregister" („register destination") und kann für verschiedene Registernamen stehen (wird weiter unten genauer erklärt).

„Operation" zeigt kurz und bündig an, was passiert. Das „(i)" ist nur als „1.", also als Nummerierung zu lesen; mit der Anweisung selbst hat das nichts zu tun. Bei anderen Anweisungen, z.B. „ldd" gibt es mehrere Varianten der Anweisung. Die sind dann mit „(i)", „(ii)","(iii)" usw. durchnummeriert.

Entscheidend ist, was danach kommt: „Rd" steht für das Register. Der Pfeil „<-" zeigt, das Daten in dieses Register befördert werden. „I/O (A)" gibt an, wo die Daten herkommen sollen, nämlich aus dem I/O-Bereich des RAMs. Und zwar sollen die Daten (in diesem Fall ein Byte) aus I/O-RAM-Adresse „A" gelesen werden.

Unter „Syntax" (kann man grob mit „Schreibweise" übersetzen) steht, wie diese Anweisung im Quelltext eines Programmes aussehen soll, damit der Assembler sie erkennt und richtig übersetzt: Zuerst müssen die beiden Buchstaben „IN" kommen, dann mindestens ein Zwischenraum, dann die Bezeichnung des Registers (hier wieder symbolisiert durch „Rd"), dann ein Komma, mit und ohne zusätzliche Zwischenräume links und rechts davon, und schliesslich die gewünschte IO-RAM-Adresse (hier symbolisiert durch „A").

Unter „Operands" (auf Deutsch eigentlich „Operanden" besser aber mit „Parameter" zu übersetzen) ist genauer festgelegt, welche Werte für die Registerbezeichnungen und die IO-RAM-Adressen in Verbindung mit dieser Anweisung zulässig sind: Die Nummer hinter dem „R" der Registerbezeichnung dürfen die Zahlen zwischen „0" und „31" genommen werden; d.h. alle Register sind möglich. Als IO-RAM-Adresse sind nur Werte zwischen „0" und „63" erlaubt; in Hex-Schreibweise 0x0000 bis 0x005F. Ab 0x0060 fängt schon der allgemein zugängliche Bereich des RAM an. Aber keine Angst: Man kann bei der Auswahl der Adresse fast nichts falsch machen, weil der Assembler einen Fehler bei der Adressangabe anzeigt.

Unter der Überschrift „Programm counter" („Programmzähler" oder „Programmzeiger") wird angegeben, wie bei der Ausführung dieser Anweisung die Flash-RAM-Adresse ausgewählt wird, von der die nächste Anweisung gelesen wird. Der Programmzähler ist nichts anderes als ein 16bit langes Register, das immer die Adresse der Flash-RAM-Zelle enthält, von beim Ausführen des Programms die nächste auszuführende Maschinenanweisung gelesen wird. Der Inhalt dieses Registers ist mit dem „PC" gemeint. Der Pfeil „<-" zeigt an, dass der Inhalt des Programmzeiger-Registers ersetzt wird; und zwar wird zum aktuellen Inhalt einfach eine Eins dazugezählt („PC+1"). D.h. der Programmzeiger zeigt, nachdem die IN-Anweisung ausgeführt ist, einfach auf die Anweisung in der nächsten Flash-RAM-Adresse.

Der „16-Bit OpCode" zeigt die Maschinenanweisung, in die eine „IN Rd,A"-Anweisung durch den Assembler übersetzt und im Flash-RAM gespeichert wird; das braucht uns aber nicht weiter zu interessieren. Das brauchen die nur die echten Hardcore-Programmier-Junkies, die entweder Maschinenanweisungen selbst schreiben (nicht lachen: Das habe ich ganz am Anfang, als die Assembler für den Z80 noch richtig viele Deutsche Mark kosteten, auch gemacht!) oder fertig assemblierte Programme selbst in Quelltext zurückverwandeln. Soweit soll's aber nicht kommen.

Viel wichtiger ist, was unter „Status Register (SREG) and Boolean Formula" (auf Deutsch etwa: „[Inhalt des] Zustandsregister und Formel der logischen Verknüpfung [die diesen Inhalt erzeugt]"). Das SREG enthält ja die Flags für Übertrag („C"), Vorzeichen („S"), Nullbetrag („Z"), Überlauf („V") usw., die bei vielen Anweisungen Aufschluss über den Ablauf oder das Ergebnis der Anweisung geben. Bei der „IN"-Anweisung ist das ganz einfach: Der Bindestich „-" unter den Bezeichnungen der Flags „I" bis „C" bedeutet, dass kein Flag verändert wird. Das ist wichtig zu wissen, denn es kann ja sein, dass man direkt vor der „IN"-Anweisung den Inhalt eines anderen

Registers mit der „TST"-Anweisung geprüft hat; dabei werden die Flags in SREG entsprechend dem Inhalt des geprüften Registers verändert. Dann können wir jetzt sicher sein, dass die „IN"-Anwesiung diese Flags nicht verändert hat. Die Auswertung der Prüfergebnisse darf also getrost erst nach der „IN"-Anweisung erfolgen. Wenn es statt der „IN"-Anweisung aber eine „DEC"-Anweisung gewesen wäre, dann hätte sie die Flags noch einmal verändert und dabei die Prüfungsergebnisse von vorher verändert.

Nein, wir gehen nicht alle Assembler-Anweisungen in dieser Ausführlichkeit durch. - Aber es ist eine gute Übung, sich selber einmal die anderen Anweisungen des Programms, so wie oben beschrieben, von der Hilfefunktion erklären zu lassen. -