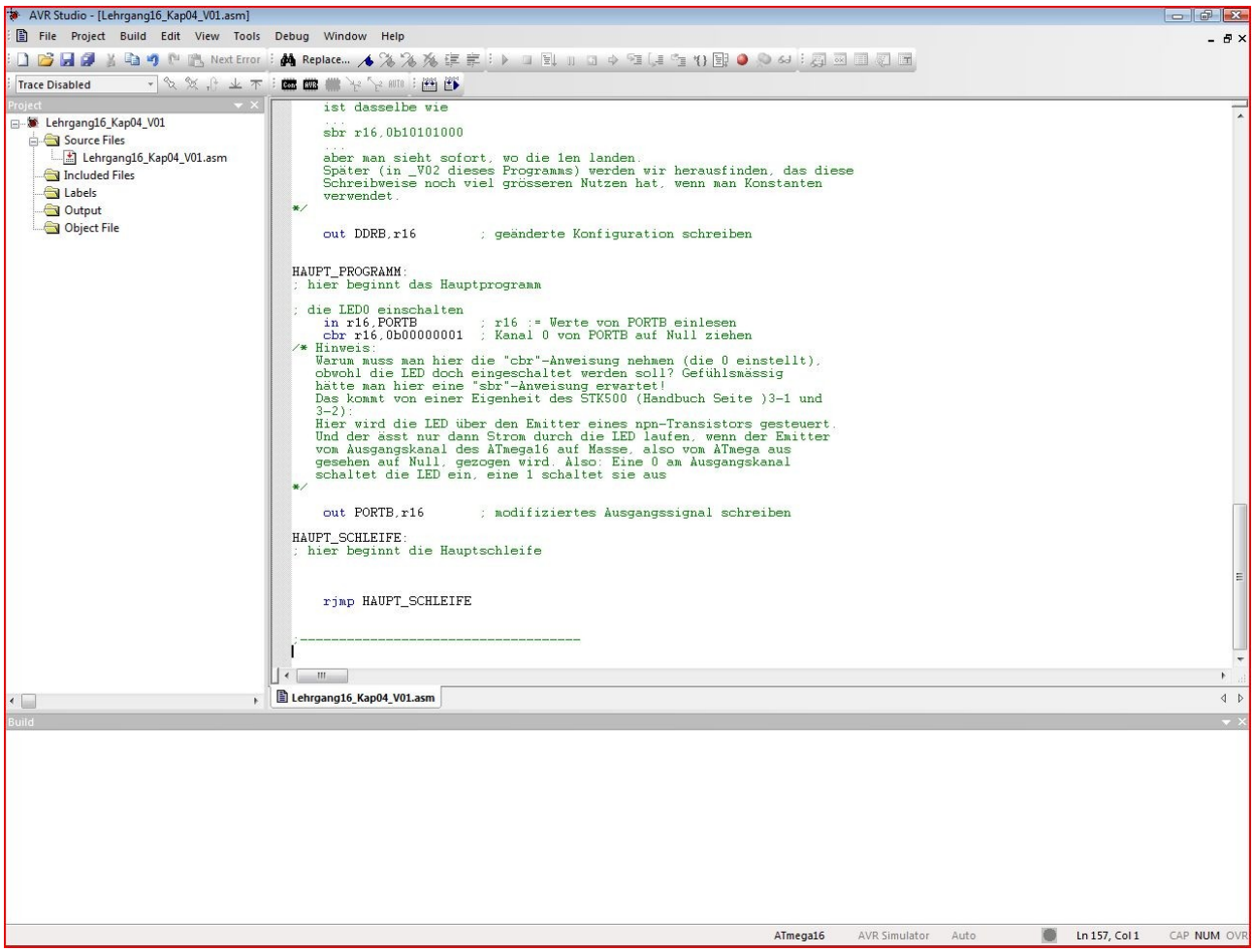
Kapitel 04: Das erste Projekt
Abschnitt 01: Den Quelltext des Programms einfügen

Um dem Assembler etwas zu Arbeiten zu geben, werden wir ihm jetzt unser erstes Programm zum Frass vorwerfen. Der Quelltext steht direkt unter diesem Absatz und beginnt mit einem Strichpunkt mit Stern, gefolgt von einer Reihe von Minuszeichen. Er endet mit der letzten Zeile, in der auf einen Strichpunkt eine Reihe Minuszeichen folgt. Diesen Quelltext gilt es jetzt in das Datei-Editorfenster des AVRStudios zu kopieren. Man kann's auch aus der .pdf-Datei abtippen - das hört sich zwar nach Vorsintflut an, ist aber gar nicht so schlecht, weil man dann viel genauer anschaut, was in dem Quelltext so alles drinsteht. Die mit „/* Hinweis: ... */" gekennzeichneten Kommentare kann man dabei ja weglassen (jeweils von „/*" bis inklusive dem nächsten „*/"). Ansonsten kann man auch auf die mitgelieferte Datei „Lehrgang16_Kap04_V01.txt" zurückgreifen und sie mit „copy" und „paste" in den Editor kopieren.

```
;*------------------------------------
; 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

/* Hinweis:
   Wenn man Fragen zu einer Anweisung hat, dann hilft einem AVRStudio: Man
   stellt den Cursor auf irgendeine Stelle in der Anweisung, oder auf die
   Position unmittelbar davor und drückt F1. Dann erscheint ein Fenster mit
   allen Details zu dieser Anweisung. Gleich mal an "sbr" ausprobieren!
*/
/* Hinweis:
   An dieser Stelle könnte man auf die Idee kommen eine "ldi"-Anweisung
   zu nehmen:
   ...
   ldi r16,0b00000001   ; Kanal 0 von PORTB als Ausgang einstellen
   ...
   Wenn man die F1-Erklärung zur "sbr"-Anweisung liest, sieht man, dass
   diese Anweisung im Register nur die Bits dort verändert, wo in dem
   Einstellbyte (in diesem Fall 0b00000001) eine 1 steht. An allen
   Stellen, an denen im Einstellbyte eine 0 steht, bleibt der alte Inhalt
   des Registers unverändert.
   Die "ldi"-Anweisung stellt aber alle Bits von r16 ein. Dort wo in
   0b00000001 eine Null steht, erscheint auch in r16 eine 0. Das bedeutet,
   dass alle Bits, die zuvor mit der "in"-Anweisung nach r16 eingelesen
   wurden, überschrieben werden. Dann hätte man sich die "in"-Anweisung
   auch gleich sparen können und diesen Abschnitt so programmieren können:
   ...
; Kanal0 von PORTB aus Ausgang einstellen
   ldi r16,0b00000001   ; Kanal 0 von PORTB als Ausgang einstellen
                        ; alle anderen Kanäle als Eingang einstellen

    Bei komplizierteren Programmen kommt es aber vor, dass an demselben
    PORT schon vorher einige Kanäle eingestellt wurden. Diese Einstellungen
    würden durch die "ldi"-Anweisung jetzt wieder überschrieben. Beim Testen
    des Programms wundert man sich dann, weil der Ausgangskanal einfach
    kein Signal von sich gibt.
    Diese Art von Fehlern ("PORT-Konfigurationsfehler") sind schwer zu
    finden. Deshalb ist es ratsam, sich zur Regel zu machen, DDRx Register
    grundsätzlich nur mit "sbr"- oder "cbr"-Anweisungen einzustellen.
*/
/* Hinweis:
    Das Einstellbyte ist schwer zu lesen; man muss immer nachzählen, ob
    die 1en an den richtigen Stellen stehen. Wenn man gewöhnt ist, mühelos
    mit Hexadezimalzahlen zu hantieren, würde man vielleicht die Schreibweise
    ...
```

```
    sbr r16,0x01        ; Kanal 0 von PORTB als Ausgang einstellen
    ...
    vorziehen.
    Wirklich gut lesbar und auch viel praktischer ist folgende Schreibweise
    ...
    sbr r16,(1<<0)      ; Kanal 0 von PORTB als Ausgang einstellen
    ...
    Das ist zunächst gewöhnungsbedürftig, weil es aus der Programmiersprache
    C stammt und wie vieles dort, schwer lesbar ist. Man muss es so lesen:
    "Schreibe eine 1 in Bit 0 des Bytes und schiebe sie um 0 Stellen nach
    links." Wenn man also eine 1 in das 5. Bit von r16 schreiben will, dann
    lautet die Anweisung
    ...
    sbr r16,(1<<5)
    ...
    Man kann also sofort sehen, an welcher Stelle des Bytes die 1
    erscheinen wird. Man kann auf diese Weise sogar gleich mehrer mehrere
    Bits auf einmal einstellen; z.B.
    ...
    sbr r16,(1<<5|1<<7|1<<3)
    ...
    ist dasselbe wie
    ...
    sbr r16,0b10101000
    ...
    aber man sieht sofort, wo die 1en landen.
    Später (in _V02 dieses Programms) werden wir herausfinden, das diese
    Schreibweise noch viel grösseren Nutzen hat, wenn man Konstanten
    verwendet.
*/

    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
/* Hinweis:
   Warum muss man hier die "cbr"-Anweisung nehmen (die 0 einstellt),
   obwohl die LED doch eingeschaltet werden soll? Gefühlsmässig
   hätte man hier eine "sbr"-Anweisung erwartet!
   Das kommt von einer Eigenheit des STK500 (Handbuch Seite )3-1 und
   3-2):
   Hier wird die LED über den Emitter eines npn-Transistors gesteuert.
   Und der ässt nur dann Strom durch die LED laufen, wenn der Emitter
   vom Ausgangskanal des ATmega16 auf Masse, also vom ATmega aus
   gesehen auf Null, gezogen wird. Also: Eine 0 am Ausgangskanal
   schaltet die LED ein, eine 1 schaltet sie aus
*/

    out PORTB,r16       ; modifiziertes Ausgangssignal schreiben

HAUPT_SCHLEIFE:
; hier beginnt die Hauptschleife


          rjmp HAUPT_SCHLEIFE


;-----------------------------------
```

(ProjectEdit_QuelleEingefuegt_V01.JPG)



So sieht's unmittelbar nach dem Einfügen des Quelltextes im Datei-Editorfenster aus. Jetzt versuchen wir das Ding einmal zu assemblieren. Dazu muss man nur auf den Knopf in der zweiten Toolbar-Zeile klicken, auf dem auf einem stilisierten, schwarzen IC mit weisser Schrift „AVR" steht (der Knopf befindet sich direkt unter dem Knopf „Replace"). Simsalabim (ProjectEdit_QuelleAssembliert_V01.JPG):

AVR Studio - [Lehrgang16_Kap04_V01.asm]

File Project Build Edit View Tools Debug Window Help

Trace Disabled

Project

- Lehrgang16_Kap04_V01
  - Source Files
    - Lehrgang16_Kap04_V01.asm
  - Included Files
  - Labels
  - Output
  - Object File

```
    ist dasselbe wie
    ...
    sbr r16,0b10101000
    ...
    aber man sieht sofort, wo die 1en landen.
    Später (in _V02 dieses Programms) werden wir herausfinden, das diese
    Schreibweise noch viel grösseren Nutzen hat, wenn man Konstanten
    verwendet.
*/

    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
/* Hinweis:
   Warum muss man hier die "cbr"-Anweisung nehmen (die 0 einstellt),
   obwohl die LED doch eingeschaltet werden soll? Gefühlsmässig
   hätte man hier eine "sbr"-Anweisung erwartet!
   Das kommt von einer Eigenheit des STK500 (Handbuch Seite )3-1 und
   3-2):
   Hier wird die LED über den Emitter eines npn-Transistors gesteuert.
   Und der ässt nur dann Strom durch die LED laufen, wenn der Emitter
   vom Ausgangskanal des ATmega16 auf Masse, also vom ATmega aus
   gesehen auf Null, gezogen wird. Also: Eine 0 am Ausgangskanal
   schaltet die LED ein, eine 1 schaltet sie aus
*/

    out PORTB,r16       ; modifiziertes Ausgangssignal schreiben

HAUPT_SCHLEIFE:
; hier beginnt die Hauptschleife


    rjmp HAUPT_SCHLEIFE


;------------------------------------
```

Lehrgang16_Kap04_V01.asm

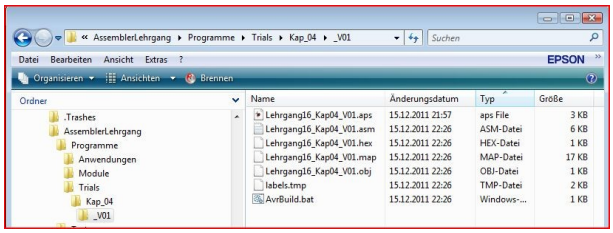
Build

```
ATmega16 memory use summary [bytes]:
Segment   Begin    End      Code   Data   Used    Size   Use%
----------------------------------------------------------------
[.cseg] 0x000000 0x00000e     14      0     14   16384   0.1%
[.dseg] 0x000060 0x000060      0      0      0    1024   0.0%
[.eseg] 0x000000 0x000000      0      0      0     512   0.0%

Assembly complete, 0 errors. 0 warnings
```

ATmega16 AVR Simulator Auto Ln 157, Col 1 CAP NUM OVR

Alles Entscheidende steht in dem Feld ganz unten mit der Überschrift „Build“: „Assembly complete, 0 errors, 0 warnings“. So einfach ist das.

Guckt man neugierig mit dem Datei-Explorer im Projektverzeichnis nach, dann stellt man fest, dass unsere Aktivität dort Spuren hinterlassen hat (es empfiehlt sich, die Ansicht „Details“ zu wählen, dann sieht man am meisten und kann die Dateien nach Typ ordnen). (Explorer_Project_Assembliert_V01.JPG)



Ganz oben ist wieder die Projektdatei mit der „.aps“-Endung. darauf folgt die „.asm“-Datei mit dem Quelltext des Programms. Als nächstes kommt das Resultat des Assembliervorgangs, die Datei mit der Endung „.hex“. Das ist die, die wir gleich nachher in den ATmega16 flashen werden.
Die nachfolgenden Dateien „.map“, „.obj“, „labels.tmp“ und „AvrBuild.bat“ erzeugt der Assembler bei jedem Durchgang neu. Für uns enthalten sie aber nichts wirklich Interessantes.