Ein erstes Assembler-Projekt

In diesem Kapitel wollen wir ein erstes einfaches Assembler-Programm für unsere Attiny-Platine schreiben. Worum soll es gehen? Wir wollen, dass der Attiny2313 an seinem Ausgang PortB.0 ein hochfrequentes Signal von 500 kHz abgibt. Da unser Attiny2313 selbst mit 4,0 MHz getaktet ist, stellt dies ein zeitkritisches Problem dar - genau das Richtige für unsere ersten Assembler-Versuche.

Öffnen wir also unseren Assembler AVR-Studio 4[1].Es erscheint der folgende Dialog zum Erstellen eines neuen Projektes:

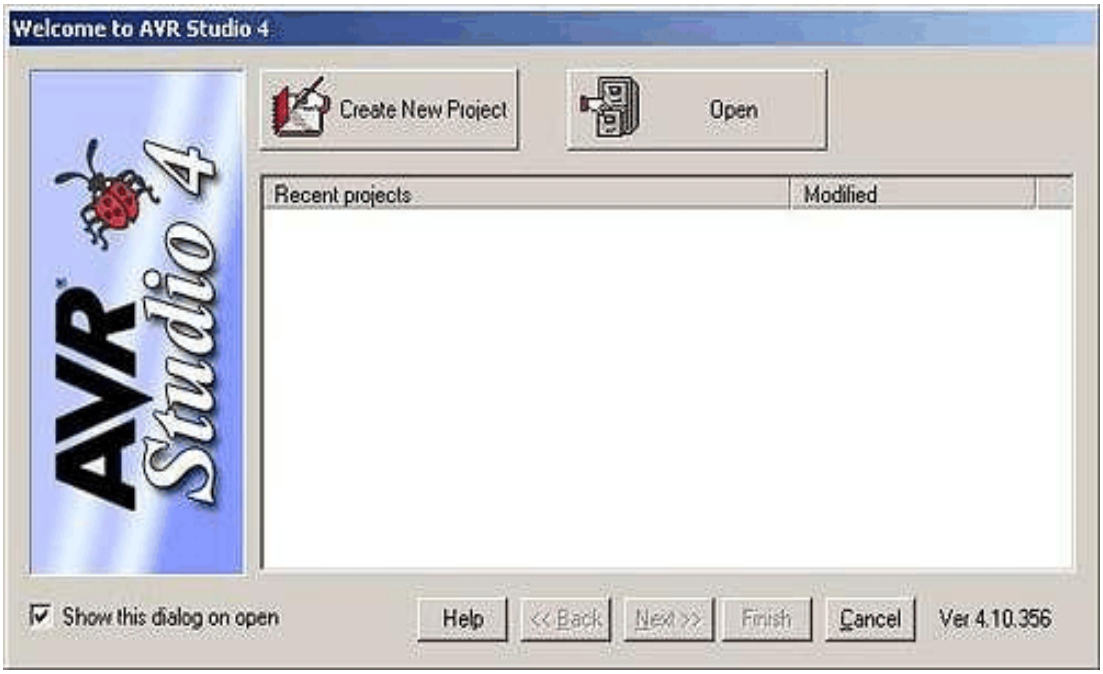

Abb. 1

Wir betätigen die Schaltfläche “Create New Projekt”. Ein weiterer Dialog wird gestartet (Abb. 2); hier geben wir in dem Textfeld rechts oben als Projektname “hf_sender” ein. Darunter lassen wir die beiden Häkchen stehen. Dadurch wird eine (leere) Startdatei “hf-sender.asm” erzeugt, in die wir gleich den Quelltext hineinschreiben werden. Außerdem wird ein eigener Ordner (Folder) für unser Projekt angelegt. Unter “Location” geben wir noch ein, wo dieser neue Ordner liegen soll. Anschließend beenden wir diesen Dialog mit der Schaltfläche “Finish”. (Nur wer später den eingebauten Simulator benutzen möchte, sollte an dieser Stelle mit “Next” die erforderlichen Einstellungen vornehmen.)

[1] Hinweise zur Installation finden Sie im Anhang.

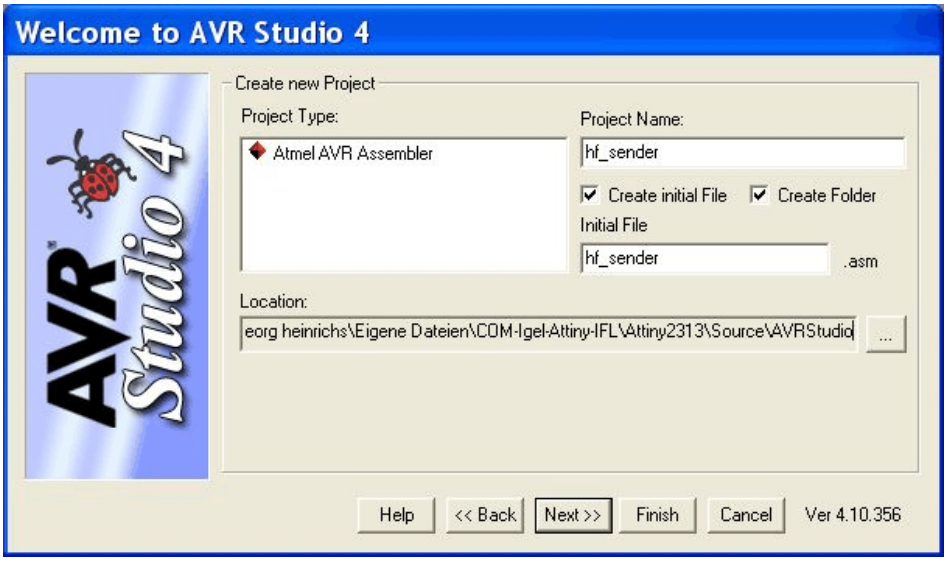

Abb. 2

Damit sind alle Vorbereitungen abgeschlossen und im Arbeitsbereich (Workspace) erkennt man, dass unser Projekt lediglich aus der Datei hf_sender.asm besteht (Abb. 3). Diese Datei wird in dem Fenster rechts daneben angezeigt. In diese Datei müssen wir nun die erforderlichen Befehle hineinschreiben. Zunächst wollen wir uns damit begnügen, durch das Programm lediglich PortB.0 auf High legen zu lassen. Das hat auch den Vorteil, dass wir das Ergebnis mit einer Leuchtdiode einfach überprüfen können.

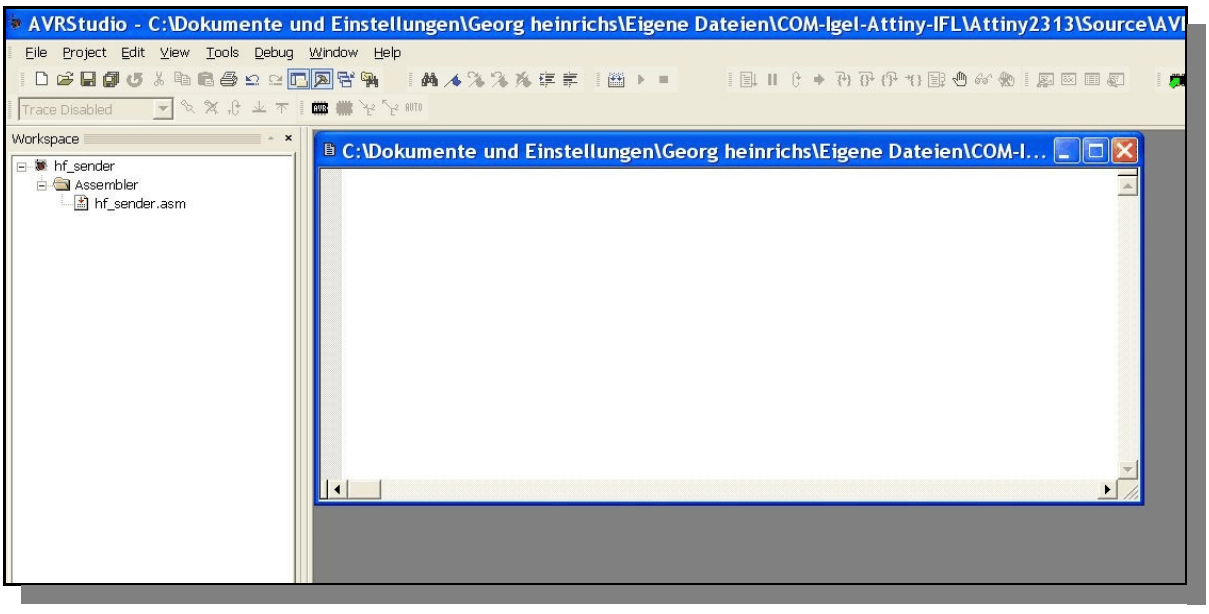

Abbildung 3

Geben wir also die benötigten Zeilen ein: Welche Bedeutung sie haben, darauf werden wir gleich noch ausführlich eingehen.

```
.include “tn2313def.inc”

rjmp init

init:
  sbi ddrb, 0
  sbi portb, 0

ende:
  rjmp ende
```

Nun speichern wir diese Datei mit File - Save (bzw. Diskettensymbol). Anschließend lassen wir den Quelltext assemblieren mit “Project - Build”. Im Ausgabebereich (Output) sehen wir, dass unser Programm im assemblierter Form lediglich 4 Wörter, also 8 Bytes beansprucht. Sollten hier irgendwo rote statt grüne Ampeln auftauchen, ist der Quelltext fehlerhaft. Klicken Sie in einem solchen Fall eine solche Zeile mit roter Ampel doppelt an, dann wird genau die Zeile im Quelltext angezeigt, welche diese Fehlermeldung hervorgerufen hat.

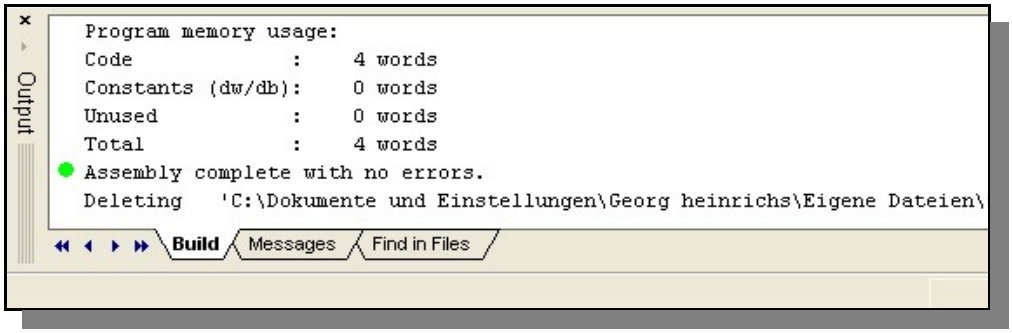

Abb. 4

Wir gehen jetzt davon aus, dass das Assemblieren erfolgreich war. In diesem Fall sind von unserem Assembler eine ganze Reihe von Dateien erzeugt und in unserem Projekt-Ordner abgelegt worden. Davon können wir uns leicht überzeugen: Wir öffnen mit einem Datei-Manager unseren Projekt-Ordner und finden folgende Dateien:

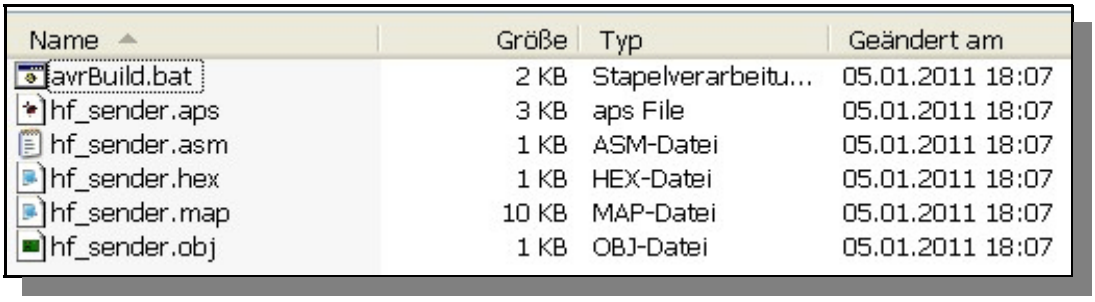

Abb. 5

Für uns ist jetzt nur die HEX-Datei hf_sender.hex wichtig. Diese gilt es den Attiny2313 hochzuladen. Dazu starten wir unseren Uploader und öffnen diese Datei mit der Schaltfläche “HEX-Datei öffnen”.

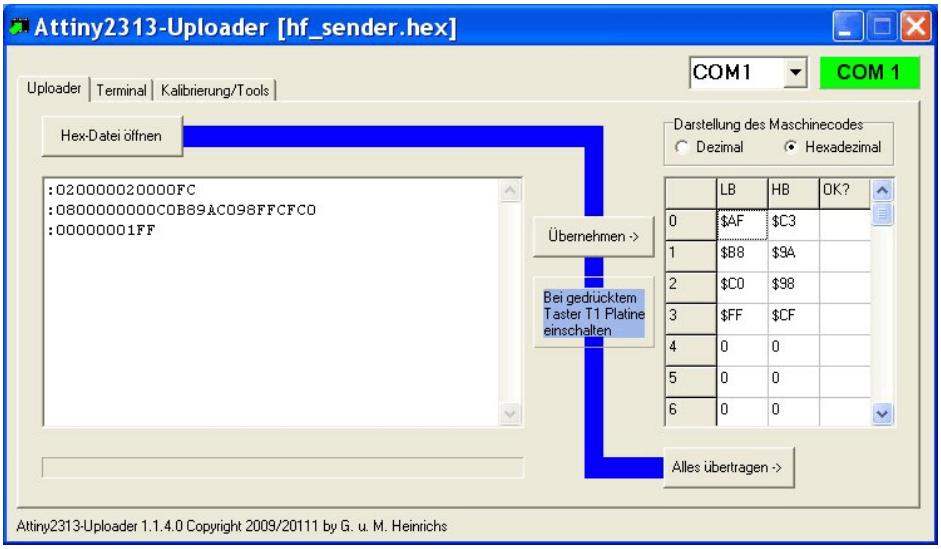

Abb. 6

In dem Textbereich sehen wir den Inhalt dieser HEX-Datei. Dass hier mehr als nur 8 Bytes zu sehen sind, hat folgenden Grund: Der Assembler benutzt - wie übrigens auch BASCOM und viele andere Compiler - das sogenannte INTEL-HEX-Format. Neben den eigentlichen Programmbytes werden dabei auch noch Kontrollbytes abgespeichert. Diese Kontrollbytes werden natürlich nicht auf den Attiny2313 übertragen. Das sehen wir, wenn wir auf die Schaltfläche “Übernehmen” klicken. Das Uploader-Programm entfernt dann diese Kontrollbytes. Was übrig bleibt, sehen wir in der Tabelle rechts daneben: die 4 Worte unseres Programms - jeweils aufgeteilt in ein Lowbyte (LB) und ein Highbyte (HB).

Die Übertragung auf die Attiny-Platine erfolgt wie üblich. Und wenn Sie nicht vergessen haben, eine LED bei PortB.0 anzuschließen, dann sollte diese jetzt auch leuchten!

Welche Bedeutung haben nun die einzelnen Zeilen in unserem Programm? Betrachten wir sie im Einzelnen. Wir beginnen mit

```
.include “tn2313def.inc”
```

Hierbei handelt es sich um eine so genannte Assemblerdirektive. Diese beginnen immer mit einem Punkt. Bei ihnen handelt es sich nicht um einzelne Befehl, die der Mikrocontroller ausführen soll; vielmehr stellen sie Anweisungen für den Assembler selbst dar. In diesem Fall sorgt die Anweisung dafür, dass der Assembler die Attiny2313-spezifischen Abkürzungen kennenlernt. Diese Abkürzungen sind in der Datei tn2313def.inc zusammengefasst.

Werfen wir einen Blick in diese Datei tn2313def.inc:

```
...
.equ    PORTA   = 0x1B
.equ    DDRA    = 0x1A
.equ    PINA    = 0x19
.equ    PORTB   = 0x18
.equ    DDRB    = 0x17
.equ    PINB    = 0x16
...
```

Als erstes erkennen wir die Abkürzungen für einige I/O-Register, so wie wir sie von BASCOM und auch aus dem AVR-Manual kennen. Diese Abkürzungen werden hier offensichtlich den Adressen dieser I/O-Register zugeordnet.

In weiteren Einträgen werden auch die Abkürzungen für die einzelnen Bits den zugehörigen Bitnummern zugeordnet, z. B.:

```
...
;***** UCSRA ******
.equ RXC      = 7
.equ TXC      = 6
.equ UDRE     = 5
...
```

Dadurch können wir auf die einzelnen I/O-Register und ihre Bits auch bei unserem Assembler in der gleichen Weise zugreifen, wie wir es schon bei BASCOM gemacht haben. Wir brauchen also auch beim Gebrauch des Assemblers nicht die Adressen der I/O-Register im Kopf haben, geschweige denn die Nummern aller Kontrollbits lernen.

`rjmp init` (rjmp = relative jump)

Mit diesem Befehl wird ein Sprung zur Marke init: durchgeführt. Der Assembler ersetzt diese Marke durch die Adresse desjenigen Befehls, der direkt hinter der Marke steht. Da die Marke direkt hinter dem Sprungbefehl steht, könnte dieser Sprung im Prinzip auch weggelassen dwerden. In diesem Fall darf er aber nicht entfallen, weil Der von uns benutzte Bootloader als Erstes immer einen Sprungbefehl erwartet. Im Kapitel über den Bootloader können Sie nachlesen, warum das so sein muss.

`sbi reg, x` (sbi = set bit in I/O-register)

Dieser Befehl setzt das Bit x des Registers reg auf 1. Reg steht dabei für die Abkürzung (oder Adresse) eines beliebigen I/O-Register und x für eine Zahl zwischen 0 und 7. Bei dem Befehl `sbi ddrb, 0` wird also das Bit 0 des Datenrichtungsregisters von Port B auf 1 gesetzt. Durch den anschließenden Befehl `sbi portb, 0` wird auch der Portausgang PortB.0 auf 1 gesetzt.

Um ein Bit auf 0 zu setzen, wird der Befehl `cbi` (clear bit in I/O-register) benutzt.

Durch

```
ende:
  rjmp ende
```

wird schließlich eine Endlosschleife erzeugt.

Kommen wir nun dazu, unseren HF-Sender zu programmieren. Was muss das Programm leisten? Der Ausgang von PortB.0 muss fortwährend seinen Zustand von 0 auf 1 umschalten und umgekehrt. Dazu benutzen wir eine Schleife, welche wir mit einem rjmp-Befehl bilden:

```
start:
  sbi portb, 0
  cbi portb, 0
rjmp start
```

Diese Schleife schaltet tatsächlich PortB.0 fortwährend ein und aus. Aber hat das Signal auch schon die richtige Frequenz? Um diese Frage beantworten zu können, muss man wissen, wie lange die Ausführung jedes einzelnen Befehls braucht. Bei unserem Attiny sind es in der Regel 2 Taktzyklen. Genauere Auskünfte darüber wird das nächste Kapitel geben.

Da unser Attiny mit 4 MHz getaktet wird, braucht er für jeden Befehl also genau eine halbe Mikrosekunde. Der Zustand PortB.0 = 1 dauert also 0,5 µs; der Zustand PortB.0 = 0 dauert aber doppelt so lange, also 1,0 µs, weil vor dem nächsten Umschalten noch der Sprungbefehl ausgeführt werden muss. Wenn wir nun eine Frequenz von 500 kHz am PortB.0 haben wollen, muss jede Phase genau 1 µs dauern. Um dies zu erreichen, müssen wir nach dem Befehl sbi portb,0 noch einen weiteren Befehl einfügen. Es ist ziemlich egal, was dieser Befehl macht, wichtig ist nur, dass er auch genau 0,5 µs braucht.

Für diesen Zweck gibt es den `nop`-Befehl (nop = no operation). Er macht nichts außer Zeit zu vergeuden. Allerdings benötigt dieser Befehl zu seiner Ausführung nur einen einzigen Taktzyklus, d. h. unser Attiny benötigt für ihn nur 0,25 µs. Für die geforderte Wartezeit von 0,5 µs sind demnach zwei nop-Befehle erforderlich.

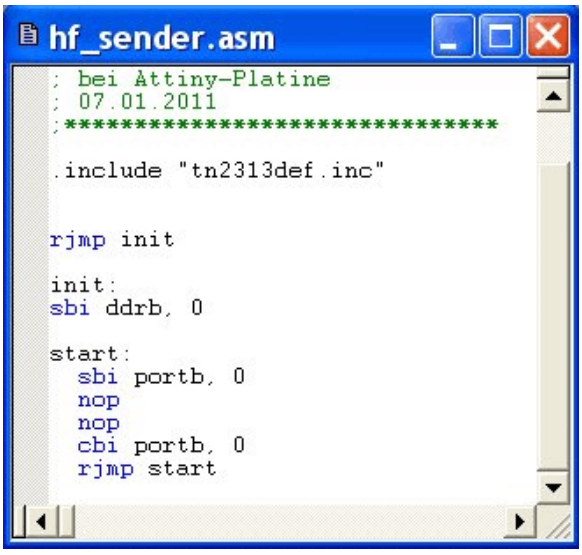

Abb. 7

Das gesamte Programm sieht dann so aus wie in Abb. 7. Zusätzlich merken wir uns: Kommentare werden durch ein Semikolon eingeleitet.

Nach dem Assemblieren übertragen wir das Programm auf den Attiny. Zur Kontrolle schließen wir PortB.0 an ein Oszilloskop an. Es zeigt auf dem Bildschirm eine Periode von 2 cm; das ent-

spricht bei einer x-Ablenkung von 1 cm pro 1 µs einer Periodendauer von 2 µs. Die Frequenz ist damit 500 kHz. Unser Ziel ist erreicht!

Würde man an PortB.0 eine kleine Antenne anschließen, könnte man das ausgestrahlte Signal mit einem Mittelwellenradio als Rauschen hören. Der Betrieb eines solchen Senders ist aber nicht erlaubt.

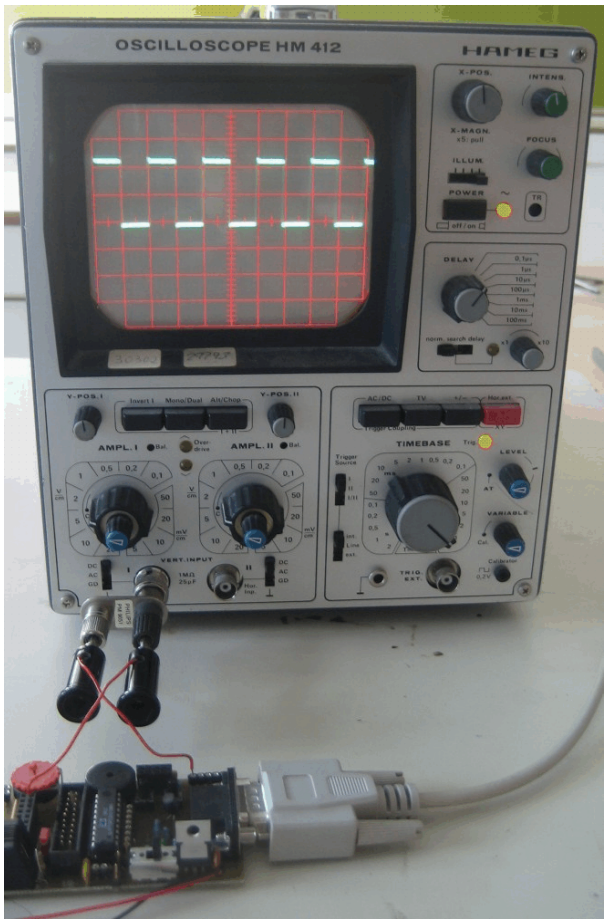

Abb. 8