Maschinencode

Jedesmal, wenn wir den Uploader benutzen, wird uns vor Augen gehalten, dass es letztlich nur Zahlen sind, die unseren Mikrocontroller steuern. Diesen Code, der nur aus Zahlen besteht, nennt man Maschinencode. Üblicherweise wird dieser Maschinencode in Form von Hexadezimalzahlen angegeben, also Zahlen im Sechszehner-System. Im Gegensatz zum herkömmlichen Zehnersystem gibt es hier nicht nur die zehn Ziffern von 0 bis 9, sondern darüber hinaus 6 weitere Ziffern, die üblicherweise mit A bis F gekennzeichnet werden. Der Vorteil des Hexadezimalsystems besteht darin, dass es besser zu dem für die digitale Datenverarbeitung so bedeutsamen Zweiersystem (Dualsystem) passt. Die folgende Tabelle macht dies deutlich:

dezimal	dual	hexadezimal
0	&B 0000 0000	&H 00
1	&B 0000 0001	&H 01
2	&B 0000 0010	&H 02
3	&B 0000 0011	&H 03
...	...	...
9	&B 0000 1001	&H 09
10	&B 0000 1010	&H 0A
11	&B 0000 1011	&H 0B
...	...	...
15	&B 0000 1111	&H 0F
16	&B 0001 0000	&H 10
17	&B 0001 0001	&H 11
18	&B 0001 0010	&H 12
...	...	...
30	&B 0001 1110	&H 1E
31	&B 0001 1111	&H 1F
32	&B 0010 0000	&H 20
33	&B 0010 0001	&H 21
34	&B 0010 0010	&H 22
...	...	...
254	&B 1111 1110	&H FE
255	&B 1111 1111	&H FF

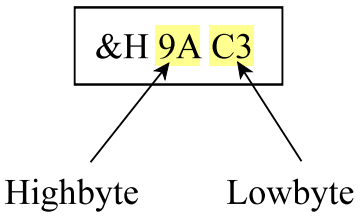
Der besseren Übersichtlichkeit halber wurden die einzelnen Bits eines Bytes in Vierergruppen

(Nibbles genannt) aufgeteilt. Diese Vierergruppen entsprechen exakt einer Ziffer im Hexadezimalsystem. Hexadezimalzahlen stellen eine der dualen Schreibweise gleichwertige Darstellung von Bytes dar.

Wie rechnet man nun eine Zahl aus dem Hexadezimalsystem in das Dezimalsystem um. Dazu muss man berücksichtigen, dass die einzelnen Stellen im Hexadezimalsystem (von rechts nach links schreitend) die Werte 1, $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, $16^4 = 65536$, ... besitzen. Für die Hexadezimalzahl &H 5B ergibt sich damit z. B.

$$\&H\ 5B = 5 \cdot 16 + 11 \cdot 1 = 91$$

Bei unserem Attiny besteht jeder Befehl (Maschinenbefehl genannt) aus zwei Bytes. Zwei Bytes bilden ein so genanntes Wort. Zum Beispiel wird durch das Wort &H 9A C3 der Pin 3 von PortB auf 1 gesetzt. Das linke Byte (&H 9A) wird Highbyte, das rechte (&H C3) wird Lowbyte genannt.



Jedes Programm unseres Attiny kann man demnach als Folge von Hexadezimalzahlen schreiben. Eine solche Datei mit Hexadezimalzahlen nennt man häufig kurz HEX-Datei. Solche HEX-Dateien sind es auch, die BASCOM (und andere Compiler) aus dem jeweiligen Quellcode herstellen. Und jedesmal wenn wir nach dem Kompilieren von BASCOM aus den Uploader aufrufen, dann wird die zugehörige HEX-Datei in den Uploader geladen (s. Abb. 1)

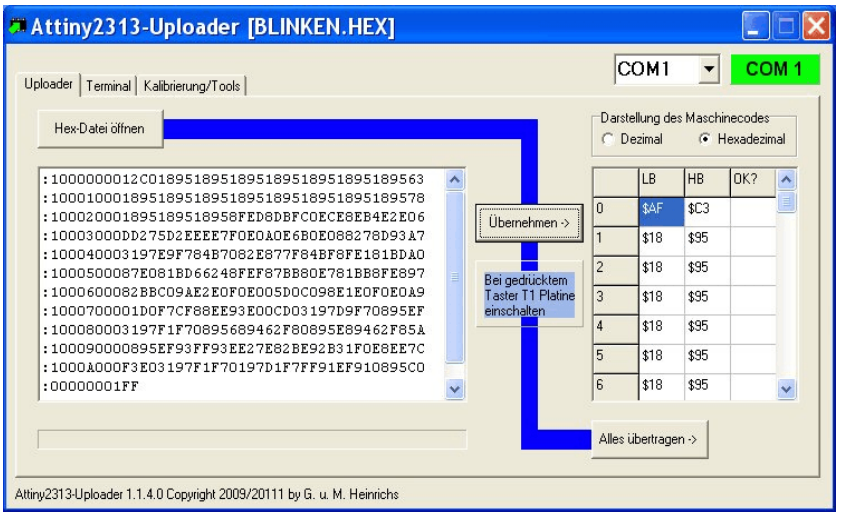


Abb. 1

Allerdings benutzen BASCOM und auch die meisten anderen Compiler hier das sogenannte INTEL-HEX-Format. Dabei wird das Lowbyte vor das Highbyte geschrieben. Außerdem werden noch Kontroll- und Prüfbytes eingefügt. Diese sind für den Attiny bedeutungslos und müssen vor dem Upload entfernt werden. Dies geschieht, wenn man die "Übernehmen"-Schaltfläche betätigt: In der Maschinencode-Tabelle stehen jetzt nur noch die Wörter, die tatsächlich den Attiny-Maschinencode darstellen. Der Vergleich zwischen dem INTEL-HEX-Code und der Maschinencode-Tabelle ist besonders einfach, wenn die einzelnen Bytes des Maschinencodes im Hexadezimalsystem angezeigt werden.

Allerdings fällt dabei auf, dass direkt in der ersten Zeile des Maschinencodes eine Abweichung vorliegt. Diese Veränderung wird vom Uploader vorgenommen, damit unser Attiny bei jedem Start immer erst in das Bootloader-Programm springt, um so gegebenfalls ein neues Programm entgegenzunehmen. Wie dieser Bootloader funktioniert, erfahren Sie unter dem Link https://www.g-heinrichs.de/attiny/Bootloader.pdf

Beim Upload werden die einzelnen Befehle, genauer gesagt die entsprechenden Doppelbytes bzw. Wörter nacheinander in die einzelnen Zellen des Programmspeichers vom Attiny geladen. Hierbei handelt es sich um einen Flash-Speicher; seine Inhalte gehen auch dann nicht verloren, wenn der Mikrocontroller von der elektrischen Quelle getrennt wird. Die einzelnen Zellen haben Adressen, durchnummeriert von 0 bis 1023. Allerdings stehen uns davon nur die ersten 943 zur Verfügung. Die restlichen Zellen werden vom Bootloader benutzt. Dieser unterstützt das Uploader-Programm beim Hochladen des Maschinencodes. Dabei gelangt das erste Wort in die Zelle mit der Adresse 0, das zweite in die Zelle mit der Adresse 1 usw.

Diese Befehle werden nach dem Upload oder auch nach einem erneuten Einschalten der Reihe nach abgearbeitet. Dazu benutzt der Attiny einen so genannten Programmzähler (PC = program counter); dieser weist immer auf die Programmzelle, deren Inhalt gerade ausgeführt wird. Nach dem Upload oder nach einem Neustart wird der PC zunächst auf 0 gesetzt und der erste Befehl ausgeführt. Danach wird der PC automatisch um 1 erhöht. Der PC weist jetzt auf den zweiten Befehl und nun wird dieser ausgeführt. Jedesmal wenn ein Befehl ausgeführt worden ist, wird der PC um 1 erhöht; dadurch wird gewährleistet, dass die Befehle der Reihe nach abgearbeitet werden.

Manchmal möchte man jedoch, dass einzelne Befehle übersprungen werden; das ist z. B. bei einer Verzweigung oder einer Schleife der Fall. Dann muss der PC durch spezielle Befehl entsprechend verändert werden. Auf welche Weise das geschieht, wird im übernächsten Kapitel noch aufgezeigt werden.

Wie geht eigentlich der Attiny mit den einzelnen Wörtern des Maschinencodes um? Nun wir wissen, dass diese Wörter nichts anderes darstellen als eine Reihe von 16 Nullen und Einsen. Alle Aktivitäten des Attiny werden durch ein so genanntes Steuerwerk getätigt. Dieses können wir ansehen als ein großes Regiepult mit vielen Schaltern und Kontrollleuchten. Mit diesen Schaltern können z. B. Signale an die einzelnen I/O-Ports gesendet werden. Stark vereinfacht können wir uns vorstellen, dass jede Stelle für eine spezielle Aktivität zuständig ist: Sie wird ausgelöst, wenn an ihr gerade eine 1 steht, sonst eben nicht. Wer hierzu mehr erfahren möchte, sollte sich https://www.g-heinrichs.de/wordpress/index.php/informatik/minipc/ anschauen.

Mnemonics

Auch für eingefleischte Programmierer ist es sehr schwierig, sich die Bedeutung der einzelnen Maschinencodes zu merken; und mindestens genau so schwierig ist es, aus dem HEX-Code seine Bedeutung zu erschließen: Welche Funktion hat z. B. der folgende Maschinencode?

&H C0 00
&H 9A B8
&H 9A C0
&H 00 00
&H 00 00
&H 98 C0
&H CF FB [1]

Als Ausweg greift man auf so genannte Mnemonics zurück. Das sind Abkürzungen für die einzelnen Befehle, die man sich leichter als ihre Codes merken kann. Ein solches Mnemonic ist z. B.

```
sbi ddrb, 0
```

Es steht für den HEX-Code &H 9A B8. Seine Bedeutung lässt sich leicht merken, wenn man weiß, dass "`sbi`" als Abkürzung für "set bit in I/O-register" steht: Hier wird offensichtlich das Bit 0 des Datenrichtungsregisters von Port B auf 1 gesetzt.

Will man ein Programm in Maschinencode schreiben, geht man üblicherweise wie folgt vor: Zunächst schreibt man das Programm mithilfe von Mnemonics. Dann übersetzt man diese Mnemonics in HEX-Code. Diesen Vorgang nennt man assemblieren. Das kann man mithilfe von Code-Büchern erledigen; und tatsächlich werden wir im übernächsten Kapitel an einem einfachen Beispiel zeigen, wie das geht.

Das Assemblieren von Hand ist aber recht umständlich und fehlerträchtig. Deswegen gibt es schon seit langem Programme, welche diese Übersetzungsarbeit zuverlässig und rasch übernehmen. Solche Programme werden Assembler genannt.

Auch Compiler erzeugen HEX-Code. Worin liegt nun der Unterschied zwischen einem Assembler und einem Compiler? Ein Compiler arbeitet mit einer so genannten Hochsprache für den Quellcode. Einem Befehl dieser Hochsprache entspricht in der Regel eine ganze Folge von Maschinenbefehlen. Bei Assemblern findet im Wesentlichen eine 1-zu-1-Übersetzung statt; d. h. einem einzigen Assemblerbefehl (Mnemonic) entspricht ein einziger Maschinenbefehl.

Mit Assemblern geschriebene Programme ergeben - wenn sie gut geschrieben sind - häufig einen kürzeren und effizienteren Maschinencode als solche, die mithilfe von Compilern erstellt

[1] Eine Lösung findet man im übernächsten Kapitel!

wurden. Warum das so ist, auch dieses werden wir in den folgenden Kapitel deutlich machen.

Assembler-Programmierung ist viel stärker mit den spezifischen Mikrocontroller-Strukturen verwoben als die Programmierung mit Compilern. Wer mit Assemblern programmiert, muss sich zwangsläufig intensiver mit Aufbau und Funktionsweise von Mikrocontrollern beschäftigen. Und genau das wird auch das Hauptziel der nächsten Kapitel sein: Weder werden wir hier einen vollständigen Assembler-Kurs anbieten, noch sollen hier umfangreiche Projekte mit einem Assembler realisiert werden. Vielmehr wird es darum gehen, einige typische Aspekte der Assembler-Programmierung exemplarisch zu beleuchten und dabei auch ein tieferes Verständnis für Mikrocontroller zu erlangen.

Wie immer soll es dabei nicht bei grauer Theorie bleiben. Deswegen wollen wir im kommenden Kapitel erst einmal einen Assembler kennen lernen, mit dem man Maschinencode für unseren Attiny herstellen kann: den Assembler AVR Studio 4.