Grundlegende Programmiertechniken

Es gibt zwei Aspekte der Assemblerprogrammiertechnik, die als grundlegend angesehen werden können: Zum Einem der Umgang mit den verschiedenen Registertypen und zum Anderen die Realisierung von Verzweigungen und Schleifen. In diesem Kapitel wollen wir diese Aspekte anhand eines einfachen Beispiels studieren: Über die serielle Schnittstelle soll der Attiny Zahlen von einem Terminal entgegennehmen und über Port B als Bitmuster an LEDs ausgeben. Diese Übertragung soll erst dann abbrechen, wenn der Escape-Code “27“ gesendet wird.

Überlegen wir zuerst, welche Aktionen für den Empfang und die binäre Darstellung einer einzigen Zahl durchgeführt werden müssen. Dabei können wir uns an der Vorgehensweise bei der BASCOM-Programmierung des Attiny orientieren. Allerdings versuchen wir bei dem Umgang mit der seriellen Schnittstelle, Highlevel-Befehle auszublenden und stattdessen lieber unsere Kenntnisse darüber auszunutzen, wie man die serielle Schnittstelle über I/O-Register steuert.

Schritt	Aktion	Erläuterung
1	DDRB auf 255 setzen	Port B als Ausgang
2	UBRR auf 25 setzen	Baudrate auf 9600
3	RXEN-Bit von UCSRB auf 1 setzen	UART-Empfänger einschalten
4	Warten bis UCSRA.RXC = 1	UCSRA.RXC = 1, wenn ein Byte empfangen wurde; s. u.
5	Inhalt von UDR merken...	Im UDR-Register steht das empfangene Byte. Beim Auslesen dieses Registers wird UCSRA.RXC wieder auf 0 gesetzt.
6	... und auf Port B ausgeben	

Vorbereitung

Wie im vorletzten Kapitel dargestellt legen wir mit unserem Assembler zunächst ein neues Projekt “com2portb” an. In den Quellcode-Editor geben wir wieder

```
.include "tn2313def.inc"
```

zum Einbinden der Attiny-spezifischen Register-Abkürzungen und

```
rjmp init
init:
```

zur Programmierung des obligatorischen Sprungbefehls (vgl. Erstes Assemblerprojekt).

Schritte 1 und 2

Zunächst muss der Wert 255 im Register DDRB abgelegt werden. Dazu könnte man natürlich mit dem `sbi`-Befehl jedes einzelne Bit auf 1 setzen. Das wäre aber nicht nur umständlich, sondern würde auch eine Verschwendung von Programmspeicherplatz bedeuten. Sinnvoller ist es, sämtliche Bits von Port B mit einem Schlag auf 1 zu setzen. In BASCOM hatten wir dazu dem Register PortB den Wert 255 zugewiesen. Leider gibt es keinen Assembler-Befehl, der dieses leistet. Es ist nicht möglich, eine Zahl aus dem Programm direkt in ein I/O-Register zu laden.

An dieser Stelle kommt eine weitere Registersorte ins Spiel, das so genannte Rechenregister. Der Name gibt schon einen Teil seiner Fähigkeiten an: Mit den Inhalten von Rechenregistern kann der Attiny rechnen. Genauer: Es gibt eine Reihen von Maschinenbefehle, die den Attiny mit den Inhalten von Rechenregistern rechnen lassen. Darüber hinaus stehen aber auch zahlreiche Maschinenbefehle zur Verfügung, mit denen Daten aus dem Programmspeicher - und da steht ja unsere Zahl 255 - oder auch aus anderen Registern geholt werden können. Außerdem gibt es auch Befehle, durch die man Daten mit den I/O-Registern austauschen kann.

Von diesen Rechenregistern gibt es 32; sie werden mit r0, r1, ..., r31 bezeichnet. Die ersten 16 Register r0 bis r15 haben eine eingeschränkte Funktionalität; deswegen werden wir im Folgenden nur die Register r16 bis r31 benutzen.

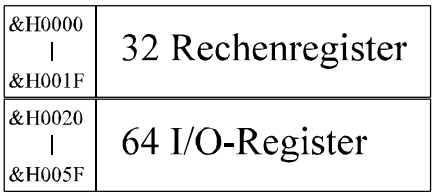

Abb. 1

Und hier wichtige Befehle für diese Rechenregister:

Mnemonic	**Abkürzung für**	**Bedeutung**
`ldi rx, k`	load immediate	lädt die Konstante k (aus dem Programmspeicher) in das Register rx
`mov rx, ry`	move	lädt den Inhalt des Rechenregisters ry in das Rechenregister rx
`in rx, IOReg`	in	lädt den Inhalt des I/O-Registers in das Rechenregister rx
`out IOReg, rx`	out	lädt den Inhalt des Rechenregisters rx in das I/O-Register

Beachten Sie, dass die Datenquelle (Woher?) immer nach dem Komma, und das Datenziel (Wohin?) immer vor dem Komma steht.

Damit können wir die Konstante 255 jetzt in zwei Schritten in das DDRB-Register befördern:

```
ldi r16, 255            ; PortB als Ausgang
out ddrb, r16
```

Und genauso verfahren wir mit dem Baudratenregister UBRR:

```
ldi r16, 25             ; Baudrate einstellen
out ubrr, r16
```

Es ist kein Versehen, dass wir hier wieder dasselbe Rechenregister benutzen. Der Wert von 255 wird ja nicht mehr für andere Zwecke benutzt: somit kann dieser Wert im Register `r16` überschrieben werden. Natürlich könnte man hier auch ein anderes Rechenregister benutzen; da es aber nur wenige davon gibt, sollte man von Anfang an lernen, mit ihnen sparsam umzugehen.

Bei längeren Programmen kann man auch leicht den Überblick darüber verlieren, welche Bedeutung die Inhalte der einzelnen Rechenregister haben. Aus diesem Grund bietet der Assembler die Möglichkeit, den Rechenregistern Namen zuzuweisen. Unser Rechenregister `r16` hat nur die Aufgabe, Zahlen kurzfristig zu speichern. Deswegen geben wir ihm den Namen `temp`. Die zugehörige Assemblerdirektive lautet:

```
.def temp = r16
```

Damit lauten die ersten Zeilen unseres Programmes:

```
.include "tn2313def.inc"

.def temp = r16

rjmp init

init:

  ldi temp, 255             ; PortB als Ausgang
  out ddrb, temp

  ldi temp, 25              ; Baudrate einstellen
  out ubrr, temp
```

Schritt 3

Zum Einschalten des UART-Empfängers muss nur ein einziges Bit des I/O-Registers UCSRB auf 1 gesetzt werden, nämlich das RXEN-Bit. Dazu benutzen wir natürlich den schon bekannten `sbi`-Befehl:

```
sbi ucsrb, rxen             ; UART einschalten
```

Schritt 4

Um den Empfang eines Bytes über die serielle Schnittstelle abzuwarten, benutzen wir die folgenden Befehle:

```
warte:
  sbis ucsra, rxc
  rjmp warte
```

Hier ist nur der `sbis`-Befehl neu:

Mnemonic	Abkürzung für	Bedeutung
`sbis IOReg, b`	skip if bit is set	überspringt den folgenden Befehl, wenn das Bit b des I/O-Registers (auf 1) gesetzt ist

In diesem Fall ist das RXC-Bit des UCSRA-Registers zunächst auf 0. Deswegen wird der nächste Befehl `rjmp warte` ausgeführt. Das Programm springt damit zur Marke `warte` und so wird der `sbis`-Befehl erneut ausgeführt... Dies geschieht solange, bis ein Byte über die serielle Schnittstelle empfangen worden ist. Dann wird nämlich das RXC-Bit durch die UART automatisch auf 1 gesetzt. Daher sorgt der `sbis`-Befehl nun dafür, dass der Befehl `rjmp warte` übersprungen wird. Der Attiny macht dann mit den darauf folgenden Befehlen weiter.

Auf diese Weise haben wir eine Warte-Schleife programmiert: Der Attiny durchläuft diese Schleife solange, bis das Kontrollbit gesetzt wird.

Will man den Attiny warten lassen, bis ein Kontrollbit zurückgesetzt, also auf 0 gesetzt wird, benutzt man den Befehl `sbic`:

Mnemonic	Abkürzung für	Bedeutung
`sbic IOReg, b`	skip if bit is clear	überspringt den folgenden Befehl, wenn das Bit b des I/O-Registers auf 0 gesetzt ist

Diese beiden skip-Befehle lassen sich aber nicht auf alle I/O-Register anwenden, sondern nur auf die ersten 32 (d.h. die I/O-Register mit den Adressen &H20 bis &H3F) anwenden (vgl. I/O-Register-Kapitel auf S. 211 des Manuals).

Schritte 5 und 6

Für diese Schritte benutzen wir den `in`- und den `out`-Befehl. Zum Zwischenspeichern benutzen wir das Register `r17`, welchem wir gemäß seiner Bedeutung den Namen “empfangen” geben:

```
.equ r17 = empfangen
...

  in empfangen, udr           ; empfangenes Byte merken
  out portb, empfangen        ; und auf Port B ausgeben
```

Wir bemerken noch, dass durch das Auslesen des UDR-Registers das RXC-Bit automatisch wieder gelöscht wird.

Damit haben wir alle Schritte für den Empfang und die Ausgabe eines einzigen Bytes programmiert. Um unsere Aufgabe vollständig zu erfüllen, müssen wir jetzt noch dafür sorgen, dass die Schritte 4 bis 6 immer wieder ausgeführt werden, bis das empfangene Byte “27" ist. Dazu muss der empfangene Wert mit dem konstanten Wert 27 verglichen werden. Wenn keine Übereinstimmung vorliegt, dann soll das Programm zur Marke `warte` verzweigen; ansonsten soll es einfach mit dem nächsten Befehl weitermachen. Dazu benutzen wir folgende Befehle:

Mnemonic	**Abkürzung für**	**Bedeutung**
`cpi rx, k`	compare immediate	vergleicht den Inhalt des Rechenregisters mit der Konstanten k. Bei Gleichheit wird das Z-Bit des Statusregisters SREG auf 0 gesetzt, sonst auf 1.
`brne adr`	branch if not equal	verzweigt zur Marke adr:, wenn das Z-Bit des Statusregisters auf 0 ist.

Durch die beiden Befehle

```
  cpi empfangen, 27       ; empfangen = 27?
  brne warte              ; nein -> nächstes Byte empfangen
```

wird immer wieder solange zur Marke `warte` gesprungen, bis der empfangene Wert gleich 27 ist.

Damit das Programm mit einer Endlosschleife abschließt, fügen wir wie üblich noch die beiden Zeilen

```
ende:
 rjmp ende
```

an. Damit lautet das vollständige Programm

```
.include "tn2313def.inc"

.def temp      = r16
.def empfangen = r17


rjmp init
```

```
init:
; initialisieren
  ldi temp, 255                 ; PortB als Ausgang
  out ddrb, temp

  ldi temp, 25                  ; Baudrate
  out ubrr, temp

  sbi ucsrb, rxen               ; UART-Empfänger einschalten


; warten bis Byte empfangen
warte:
  sbis ucsra, rxc               ; nächsten Befehl überspringen,
                                          wenn Byte empfangen
  rjmp warte                    ; weiter warten


  in empfangen, udr             ; empfangenes Byte merken
  out portb, empfangen          ; und auf Port B ausgeben

  cpi empfangen, 27             ; empfangen = 27?
  brne warte                    ; nein -> nächstes Byte empfangen

ende:                           ; Endlosschleife
  rjmp ende
```

Nach der Eingabe assemblieren wir das Programm und übertragen den Hex-Code auf den Attiny. Anschließend aktivieren wir den Terminal-Bereich des Uploader-Programms. Zum Test stecken wir 8 LEDs in die Pins von Port B, geben die Zahl 85 in die zweite Eingabezeile ein und betätigen die Schaltfläche “Sende Zahl”. Sofort erscheint das zugehörige Bitmuster an Port B: 01010101.
Übertragen Sie auch einmal einzelne Buchstaben an den Attiny. Benutzen Sie dazu die erste Zeile des Terminalprogramms. Die LEDs zeigen dann das Bitmuster des zugehörigen ASCII-Codes

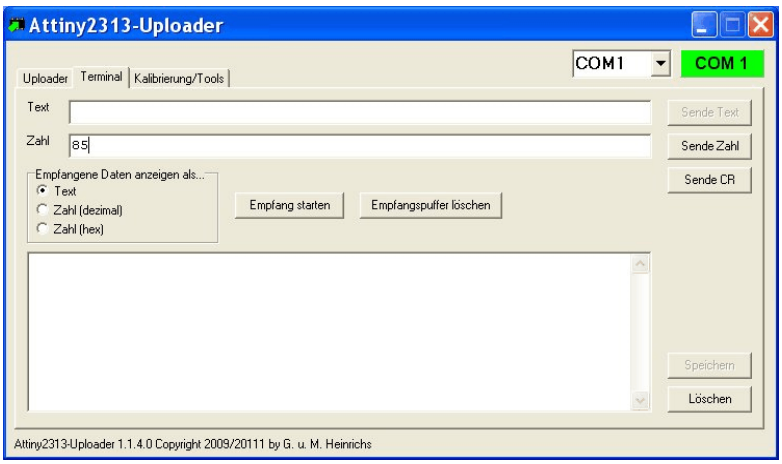

Abb. 2

an. Gibt man statt eines einzigen Buchstabens eine längere Zeichenkette ein, sieht man ein rasches Flackern der LEDs; der Attiny zeigt die Bitmuster der einzelnen Buchstaben so rasch hintereinander an, wie sie vom Terminalprogramm gesendet worden sind.

Fassen wir unsere Erkenntnisse über Schleifen noch einmal zusammen: Schleifen werden letztlich durch Sprünge realisiert. Für Endlosschleifen reichen hier `rjmp`-Befehle aus. Sollen die Schleifen dagegen abbrechen, haben wir zwei Fälle unterschieden:

1. Fall: Das Abbrechen der Schleife soll durch ein Bit eines I/O-Registers kontrolliert werden. Hier benutzt man die Kombination eines Skip-Befehls mit einem Sprung-Befehl.

2. Fall: Das Abbrechen der Schleife soll durch Vergleichen mit einer Konstanten kontrolliert werden. Hier benutzt man die Kombination eines Vergleichs-Befehls mit einem Branch-Befehl.

Der Attiny stellt noch eine große Zahl weiterer Schleifen- und Verzweigungsstrukturen zur Verfügung. Eine Übersicht bieten die Seiten 213 und 214 des Manuals. Einige der dort angegebenen Befehle sind in der folgenden Tabelle zusammen gefasst.

Mnemonic	**Abkürzung für**	**Erläuterung**
`sbic IOReg, b`	skip if bit in I/O-register is cleared	überspringt den folgenden Befehl, wenn das Bit b des I/O-Registers auf 0 gesetzt ist
`sbis IOReg, b`	skip if bit in I/O-register is set	überspringt den folgenden Befehl, wenn das Bit b des I/O-Registers auf 1 gesetzt ist
`sbrc rx, b`	skip if bit in register is cleared	überspringt den folgenden Befehl, wenn das Bit b des Rechenregisters auf 0 gesetzt ist
`sbrs rx, b`	skip if bit in register is set	überspringt den folgenden Befehl, wenn das Bit b des Rechenregisters auf 1 gesetzt ist
`cp rx, ry`	compare	vergleicht den Inhalt der Rechenregister. Das Vergleichsergebnis wird im Register SREG gespeichert und von den Branch-Befehlen ausgewertet.
`cpi rx, k`	compare immediate	vergleicht den Inhalt des Rechenregisters mit der Konstanten k. Bei Gleichheit wird das Z-Bit des Statusregisters SREG auf 0 gesetzt, sonst auf 1.
`breq adr`	branch if equal	verzweigt zur Marke `adr`, wenn das Z-Bit des Statusregisters auf 1ist

Mnemonic	**Abkürzung für**	**Erläuterung**
`brne adr`	branch if not equal	verzweigt zur Marke `adr`, wenn das Z-Bit des Statusregisters auf 0 ist
`brlo adr`	branch if lower (unsigned)	verzweigt zur Marke `adr`, wenn beim Vergleich der linke Wert kleiner als der rechte war
`brsh adr`	branch if same or higher (unsigned)	verzweigt zur Marke `adr`, wenn beim Vergleich der linke Wert nicht kleiner als der rechte war

Aufgaben

1. Der Attiny soll jedes empfangene Byte sofort als Echo zurück an das Terminal schicken. Schreiben Sie ein entsprechendes Programm und testen Sie es aus.

2. Schreiben Sie das Programm von S. 5f einmal mit BASCOM und vergleichen Sie die Länge der erzeugten Hex-Codes.

3. Schreiben Sie ein Assembler-Programm, welches 10 Bytes über die serielle Schnittstelle entgegennimmt und in den ersten 10 Speicherplätzen des EEPROMs speichert. Zum Testen des Programms können Sie die Inhalte des EEPROMs mit eeprom2com.bas anzeigen lassen.
   Achtung: Denken Sie daran, dass Sie die EEPROM-Zelle mit der Adresse 127 auf keinen Fall überschreiben dürfen!

4. Für einen Skip-Befehl soll das Bit eines I/O-Registers ausgewertet werden, welches eine Adresse über &H3F besitzt. Warum kann dazu nicht der `sbic`- bzw. `sbis`-Befehl benutzt werden? Überlegen Sie sich eine geeignete Strategie.