1.7 Atmega-Programmierung in ASM/Verschachtelte Schleifen

1.7.1 Aufgabe

Die beiden LEDs sollen abwechselnd blinken. Mit der bisherigen Lösung flackern sie nur (Beispiel: `blink0.asm`):

```
1  .include "/usr/share/avra/m32def.inc"
2
3          sbi DDRC, PC1    ; LED1 als Ausgang
4          sbi DDRC, PC2    ; LED2 als Ausgang
5  start:
6          ldi r16, 255
7  weiter1:
8          dec r16
9          brne weiter1
10         cbi PORTC, PC1
11         sbi PORTC, PC2
12
13         ldi r16, 255
14 weiter2:
15         dec r16
16         brne weiter2
17         sbi PORTC, PC1
18         cbi PORTC, PC2
19
20         rjmp start
```

1.7.2 Überlegung

Dazu brauchen wir eine längere Wartezeit. Die bisherigen Schleifen konnten maximal 256 Durchläufe bereitstellen:

```
1          ldi r16, 255     ; 1 Takt
2  nochmal:
3          dec r16          ; 1 Takt
4          brne nochmal     ; 2 Takte bei Sprung, sonst 1
```

Die Dauer eines Schleifendurchlaufs ist:

$$T_d = T_{CPU} \cdot (1+2) = T_{CPU} \cdot 3 = \frac{1}{16\,\mathrm{MHz}} \cdot 3 = 0{,}1875\,\mu\mathrm{s}$$

Nur der letzte Durchlauf ist einen Takt kürzer. Die Gesamtzeit ist dann

$$T_g = T_{CPU} \cdot (1 + 255 \cdot (1+2) - 1) = T_{CPU} \cdot 765 = \frac{1}{16\,\mathrm{MHz}} \cdot 765 = 47{,}8\,\mu\mathrm{s}$$

Bei zwei dieser Schleifen erhält man

$$T = T_g, ein + T_g, aus = 95{,}6\,\mu\mathrm{s}$$

und

$$f = \frac{1}{T} = 10{,}46\,\mathrm{kHz}$$

(bzw. $f = \frac{1}{T} = 654\,\mathrm{Hz}$ mit dem internen 1-MHz-Oszillator).

In so einem Fall kann man

- zwei Schleifen nacheinander setzen – die Zahl der Durchläufe addiert sich:

$$N_{ges} = N_1 + N_2$$

- zwei Schleifen ineinander verschachteln – die Zahl der Durchläufe multipliziert sich:

$$N_{ges} = N_1 \cdot N_2$$

1.7.3 Konstruktion

Wenn man von der C-Programmierung Struktogramme gewohnt ist, kann man von dort ausgehen (Abbildung 1). Dort werden – wie im letzten Beispiel – Fußgesteuerte Schleifen benutzt, die jeweils

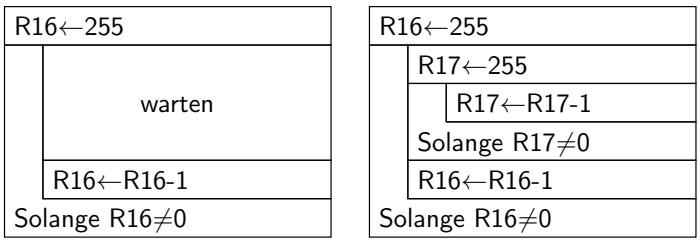

Abbildung 1: Verschachtelte Schleifen als Struktogramm

eine Zählschleife bilden. Erstmals wird hier das Register `r17` benutzt. Man kann es genauso wie `r16` benutzen. Dazu gibt es noch die Register `r18`–`r31`, die sich ebenso verwenden lassen[1].

Als nächsten Schritt kann man ein Flussdiagramm aufbauen (Abbildung 2). Daraus kann man

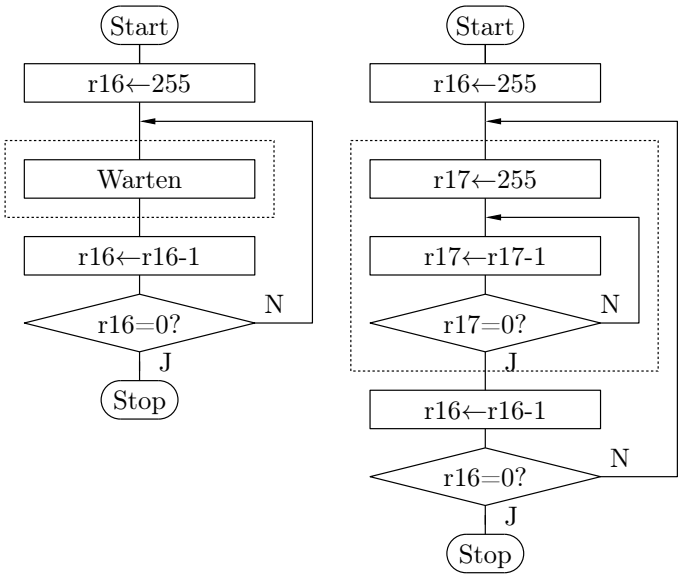

Abbildung 2: Verschachtelte Schleifen als Flussdiagramm

dann ein verbessertes Programm erstellen (`blink1.asm`):

```
1 .include "/usr/share/avra/m32def.inc"
2
3         sbi DDRC, PC1    ; LED1 als Ausgang
```

[1] dazu noch die Register `r0`–`r15`, bei denen aber einige Einschränkungen existieren.

```
4           sbi DDRC, PC2    ; LED2 als Ausgang
5   start:
6   ;================== Start aeussere Schleife
7           ldi r16, 255
8   weiter1:
9   ;------------------ Start innere Schleife
10          ldi r17, 255
11  weiter1a:
12          nop
13          nop
14          dec r17
15          brne weiter1a
16  ;------------------ Ende innere Schleife
17          dec r16
18          brne weiter1
19  ;================== Ende aeussere Schleife
20          cbi PORTC, PC1
21          sbi PORTC, PC2
22  ;==================
23          ldi r16, 255
24  weiter2:
25  ;------------------
26          ldi r17, 255
27  weiter2a:
28          nop
29          nop
30          dec r17
31          brne weiter2a
32  ;------------------
33          dec r16
34          brne weiter2
35  ;==================
36          sbi PORTC, PC1
37          cbi PORTC, PC2
38          rjmp start
```

Die Blinkfrequenz beträgt hier noch 40 Hz bei einer CPU-Taktfrequenz von 16 MHz. Das ist noch zu hoch. Wenn man den internen CPU-Takt von 1 MHz nutzt, kann man das Blinken dagegen schon deutlich erkennen ($f = 2{,}5\,\text{Hz}$). Also braucht man noch eine dritte Schleife, um ein langsames Blinken zu erreichen.

1.7.4 16-Bit-Register und Warteschleifen

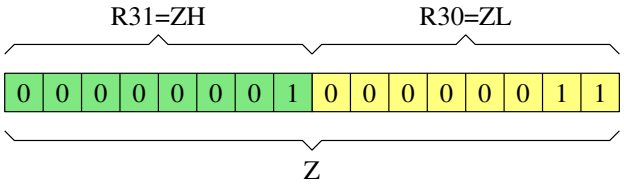

Abbildung 3: 16-Bit-Register Z

Mit drei verschachtelten Schleifen wird so ein kleines Blinkprogramm schon sehr aufwändig. Die AVR-Mikrocontroller bieten aber einen Weg an, bei dem man mit weniger Schleifen auskommt: Die zwei 8-Bit-Universal-Register R30 und R31 lassen sich als ein 16-Bit-Register Z auffassen[2] (Abbildung 3). Das niederwertige Byte (*low byte*) wird in R30 gespeichert und das höherwertige Byte (*high byte*) in R31. Damit der Programmierer es leichter hat, kann er R30 unter dem Namen ZL ansprechen und R31 unter dem Namen ZH. In diesem Beispiel enthält R31=ZH den Wert 1. R30=ZL enthält den Wert 3. Damit enthält Z den Wert:

$$Z = 256 \cdot ZH + 1 \cdot ZL = 256 \cdot 1 + 1 \cdot 3 = 259$$

Umgekehrt kann man aus dem Wert von Z den Inhalt von ZH und ZL ermitteln[3]:

$$ZH = Z/256 = 259/256 = 1$$

$$ZL = Z\%256 = 259\%256 = 3$$

Es gibt noch weitere 16-Bit-Register wie Y und X (siehe Tabelle). Für die 16-Bit-Register gibt es

high byte	*low byte*	16-Bit-Register
R31	R30	Z
R29	R28	Y
R27	R26	X
R25	R24	namenlos

Tabelle 1: 16-Bit-Register bei ATmega-Controllern

eigene Befehle, z. B.:

- adiw zl, zahl – addiert zu einem der in Tabelle genannten 16-Bit-Register (hier Z) eine Zahl, die zwischen 0 und 63 (hier 50) liegen darf; Achtung: Es wird das Low-Byte-Register (hier Z) als Register genannt, nicht das 16-Bit-Register (hier Z).
- sbiw zl, zahl – subtrahiert von Z eine Zahl, die zwischen 0 und 63 liegen darf

Eine Schleife mit 259 Durchläufen kann man jetzt ganz einfach erstellen:

```
1         ldi zh, high(259); der Assembler macht daraus: ldi r31, 1
2         ldi zl, low(259) ; der Assembler macht daraus: ldi r30, 3
3 schleife:
4         sbiw zl, 1       ; 2 Takte
5         brne schleife    ; 2 Takte bei Sprung, 1 Takt sonst
```

Damit die Schleife funktioniert, setzt der sbiw-Befehl die Bits im Statusregister SREG richtigerweise passend zur 16-Bit-Bedeutung des Befehls: Nur, wenn ZH und ZL beide gleich null sind, wird das Z-Flag gesetzt. Ebenso ist es mit den anderen Bits in SREG.

[2] Das Z-Register hat nichts zu tun mit dem gleichnamigen Z-Flag des Statusregisters SREG!

[3] Das Zeichen / steht für eine Ganzzahldivision ohne Rest, das Zeichen % steht hier wie in der Programmiersprache C für eine Modulo-Operation.

1.7.5 Kopfgesteuerte Schleifen und Vergleichsbefehle

In manchen Fällen braucht man in einem Programm eine kopfgesteuerte Schleife (Beispiel in Abbildung 4). Bei einer kopfgesteuerten Schleife wird die Bedingung schon am Anfang abgefragt.

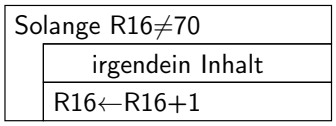

Abbildung 4: Kopfgesteuerte Schleife, Beispiel

Damit kann es passieren, dass die Schleife kein einzige Mal durchlaufen wird. Wie kann man das in Assembler programmieren, wenn doch erst ein Additions- oder Subtraktionsbefehl das Bit im Statusregister SREG setzt, das im bedingten Sprungbefehl abgefragt wird?

Man braucht dazu einen Vergleichsbefehl, der unabhängig von einer Rechenoperation einfach zwei Inhalte auf größer/kleiner bzw. gleich/ungleich überprüft (Abbildung 5).

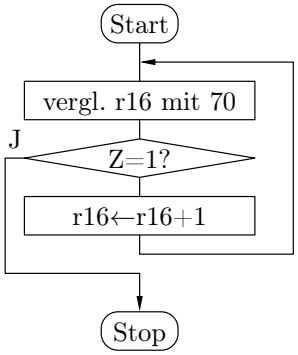

Abbildung 5: Kopfgesteuerte Schleife in Assembler, Beispiel

Dazu gibt es die Befehle cpi (Vergleiche Register mit Zahl) und cp (Vergleiche Register mit Register):

```
1 schleife:
2         cpi r16, 70       ; Vergleich
3         breq nachschleife ; bei Gleichheit Sprung hinter die Schleife
4         inc r16           ; Schleifenrumpf
5         rjmp schleife     ; Sprung zum Anfang
6 nachschleife:
7         ; hier geht es weiter ...
```

Der Befehl in Zeile 2 zieht von R16 den Wert 70 ab. Das Ergebnis wird nicht gespeichert, aber das Statusregister SREG wird gesetzt. Falls sich bei der Subtraktion der Wert null ergab, wird das Z-Flag auf eins gesetzt. Ebenso werden auch alle anderen Status-Bits genauso gesetzt wie bei einem „richtigen“ Subtraktions-Befehl, nur das Ergebnis wird eben *nicht* nach R16 geschrieben.