1.8 Atmega-Programmierung in ASM/Stack und Unterprogramme

1.8.1 Aufgabe

Es soll ein Lauflicht programmiert werden. Problematisch ist es, dass nach jedem Weiterschalten eine Warteschleife programmiert werden muss:

- Man hat dreimal die gleiche Arbeit → Zeit und Kosten
- Man hat dreimal den gleichen Programmcode → Speicherplatz

Das erste Problem könnte man mit einem Präprozessor wie in C lösen, der dreimal den gleichen Textbaustein einfügt. Aber es gibt auch eine elegantere Lösung.

1.8.2 Was sind Unterprogramme?

In C und anderen Hochsprachen gibt es dafür Funktionen oder Prozeduren. In Assembler nennt man diese Programmstruktur *Unterprogramme*. Unterprogramme sind Programmbausteine. Man kann sie an verschiedenen Stellen im Programm benutzen. Nach der Benutzung des Unterprogramms geht das Programm an derselben Stelle weiter wie vorher (Abbildung 1).

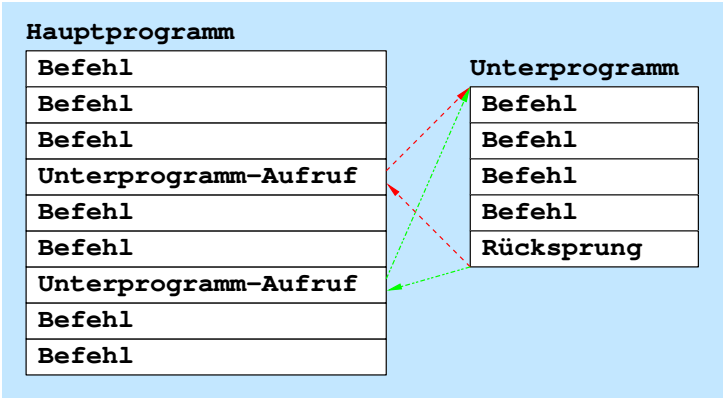

Abbildung 1: Prinzip eines Unterprogrammes

Ideal ist es, wenn man solche Unterprogramme mehrfach ineinander verschachteln kann (Abbildung 2).

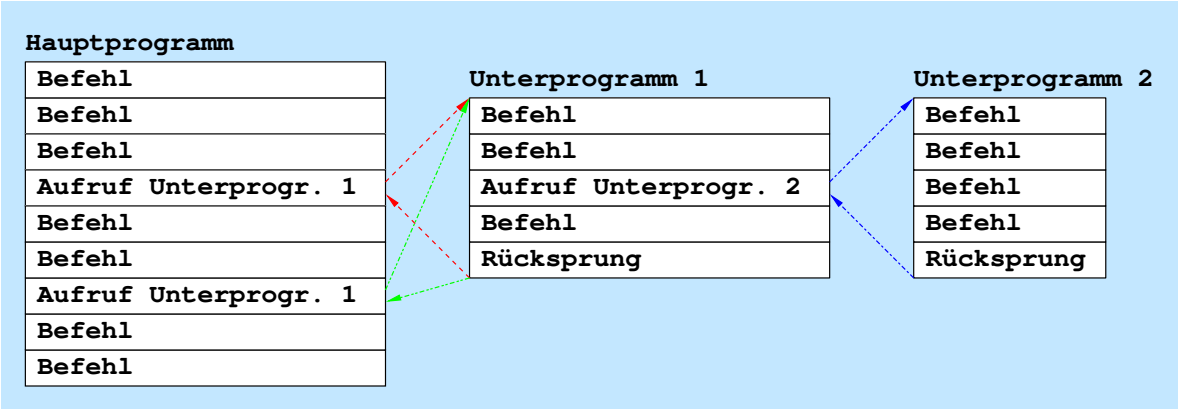

Abbildung 2: Verschachtelte Unterprogramme

1.8.3 Unterprogramm-Aufruf

Im Befehlssatz der AVR-Mikrocontroller gibt es dazu einen speziellen Sprungbefehl, bei dem der aktuelle Stand des Programmzählers gespeichert wird. Das folgende Programm zeigt ein Beispiel (`stackbsp2.asm`):

```
1  .include "/usr/share/avra/m32def.inc"
2
3  start:
4          ldi r16, 1<<PC7|1<<PC6|1<<PC5
5          out DDRC, r16
6          ldi r16, low(RAMEND)
7          out SPL,  r16
8          ldi r16, high(RAMEND)
9          out SPH, r16
10 schleife:
11         rcall allesEin
12         rcall allesAus
13         rjmp schleife
14 allesAus:
15         ldi r16, 1<<PC5|1<<PC6|1<<PC7
16         out PORTC, r16
17         ret
18 allesEin:
19         ldi r16, ~(1<<PC5|1<<PC6|1<<PC7)
20         out PORTC, r16
21         ret
```

Der genannte Befehl, der das Unterprogramm aufruft, heißt `rcall`[1]. Der Unterschied zu `rjmp` liegt darin, dass sich bei `rcall` der Controller die aktuelle Adresse in einem bestimmten Bereich abspeichert. Mit dem Befehl `ret` (für *return*) wird wieder zur abgespeicherten Adresse zurückgesprungen.

1.8.4 Stack und Stackpointer

Wozu dienen nun die Zeilen mit `SPL` und `SPH`? Es handelt sich dabei um ein spezielles Register. In diesem Register wird auf einen Bereich im SRAM gezeigt, in dem die aktuellen Rücksprungadressen liegen. Diesen Speicherbereich nennt man *Stack* (oder auf deutsch *Stapel*).

Das Register heißt dementsprechend *Stackpointer*. Da die Befehlsadressen bei den meisten AVR-Controllern größer als 255 sein können, muss der Stackpointer 16 Bit umfassen. Er ist in die beiden SFR-Register `SPL` (für das niedrigwertige Byte) und `SPH` (für das höerwertige Byte) aufgeteilt (Tabelle 1).

Beim Programmbeginn wird der Stackpointer zunächst so gesetzt, dass er auf das RAM-Ende zeigt (der genaue Wert ist in `m8def.inc` in der Konstanten `RAMEND` vermerkt). Der Stackpointer wird erstens benutzt durch den Befehl `rcall`:

a) Dort, wo der Stackpointer hinzeigt, wird die Rücksprungadresse (aktueller Wert des Befehlszählers+1, 16 Bit breit) hineingeschrieben

b) Der Stackpointer wird um 2 verringert, weil das RAM 8-bittig ist und daher zwei Byte zum Speichern der 16-Bit-Befehlsadresse gebraucht wurden und zeigt jetzt auf den nächsten freien Platz

Die umgekehrte Abfolge passiert beim Befehl `ret`:

[1] Das *r* steht für relativ; das Sprungziel wird relativ zur aktuellen Adresse angegeben. Mit diesem Befehl kann man maximal ±2047 Befehle vor- oder rückwärts springen. Die Adressberechung macht das Assembler-Programm.

Name									Adresse
	Bit	7	6	5	4	3	2	1	0	
SPH							SP10	SP9	SP8	SFR 62
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	Init.	0	0	0	0	0	0	0	0	
Name									Adresse
	Bit	7	6	5	4	3	2	1	0	
SPL		SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SFR 61
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	Init.	0	0	0	0	0	0	0	0	

Tabelle 1: Stackpointer-Register SPL und SPH

a) Der Stackpointer wird um 2 erhöht

b) Von dort, wo der Stackpointer jetzt hinzeigt, wird der Befehlszähler geladen

Der Stackpointer zeigt also immer auf die nächste freie Stelle im SRAM.

Somit ist es prinzipiell möglich, sehr viele Unterprogrammaufrufe zu verschachteln, und bei größeren Systemen und Hochsprachen wird davon ausgiebig Gebrauch gemacht. Der Stack wächst dann, wenn viele Ebenen von Unterprogrammen aufgerufen wurden und schrumpft beim Verlassen der Unterprogramme. Daher kommt auch sein Name *Stapel*: Wie bei einem Stapel von Gegenständen kann man immer nur das zuletzt aufgelegte Element wieder entnehmen (LIFO-Prinzip, *last in – first out*).

Anders als sein Name suggeriert, liegt der Stack aber nicht auf dem Boden des SRAM mit den niedrigsten Adressen, sondern er hängt quasi von der Decke des SRAM (durch die Konstante `RAMEND` ermittelt) und wächst mit zunehmender Benutzung von der Decke herab.

Wenn man zu viele Unterprogrammebenen verschachtelt, kann es sein, dass der Stack in Bereiche kommt, die für andere Zwecke reserviert sind (Variablen, Register). Das ist natürlich nicht erwünscht, da das Programm dann nicht mehr funktionieren kann.

1.8.5 Kochrezept

Was muss man also tun, um Unterprogramme zu ermöglichen?

a) Bei Programmstart: Setzen des Stackpointers

b) Unterprogrammaufruf: Befehl `rcall`

c) Im Unterprogramm: Ende des Unterprogramms mit `ret`

1.8.6 Sicherung der Registerinhalte auf dem Stack

Was ist, wenn im Unterprogramm Register (z.B. r16 oder das Statusregister SREG) geändert werden? Was kann man tun, damit Unterprogramme nicht den Lauf des Hauptprogramms verändern? Eine Lösung ist in `stackbsp3.asm` gegeben:

```
1 .include "/usr/share/avra/m8def.inc"
2
3 start:
4         ldi r16, 1<<PC7|1<<PC6|1<<PC5
5         out DDRC, r16
6         ldi r16, low(RAMEND)
7         out SPL,  r16
8         ldi r16, high(RAMEND)
9         out SPH, r16
```

```
10 schleife:
11         rcall allesEin
12         rcall allesAus
13         rjmp schleife
14 allesAus:
15         push r16
16 ;       in r16, SREG
17 ;       push r16
18         ldi r16, 1<<PC5|1<<PC6|1<<PC7
19         out PORTC, r16
20 ;       pop r16
21 ;       out SREG, r16
22         pop r16
23         ret
24 allesEin:
25         push r16
26 ;       in r16, SREG
27 ;       push r16
28         ldi r16, ~(1<<PDC|1<<PC6|1<<PC7)
29         out PORTC, r16
30 ;       pop r16
31 ;       out SREG, r16
32         pop r16
33         ret
```

- Der `push`-Befehl legt den Inhalt eines Registers (z.B. `r16`) auf dem Stack ab, anschließend wird der Stackpointer um eins verringert.
- Der `pop`-Befehl erhöht den Stackpointer um eins und sichert vom Stack das Register zurück.

Der Stack hat also mehrere Funktionen:

a) Unterprogramm-Rücksprung-Adressen sichern

b) Bei Bedarf Register in Unterprogrammen sichern

c) Lokale Unterprogramm-Variable anlegen (später)

d) Parameter per Stack an das Unterprogramm und zurück übergeben (später)

Eine Übersicht über die im Zusammenhang mit Unterprogrammen neuen Befehle zeigt Tabelle 2.

Befehl	a
`rcall a`	Sprungmarke
`ret`	—
`push a`	Univ.-Register
`pop a`	Univ.-Register

Tabelle 2: Befehle für Unterprogramme