Unterprogramme

Unterprogramme haben wir schon im Zusammenhang mit BASCOM kennen gelernt. Auch Assemblerprogramme können durch Unterprogramme strukturiert werden. Hier wie dort dienen sie dazu, Programme übersichtlicher zu gestalten.

Wie schon in den vorangehenden Kapiteln wollen wir das Thema anhand eines einfachen Beispiels studieren: Der Mikrocontroller soll zwei Zahlen von Terminal empfangen, sie addieren und anschließend die Summe zurück an das Terminal senden.

Welche Aufgaben muss der Mikrocontroller dazu erledigen? Um uns eine Übersicht zu verschaffen, halten wir die einzelnen Schritte in einer Tabelle zusammen:

Schritt	**Aktion**	**Erläuterung**
1	UBRR auf 25 setzen	Baudrate auf 9600
2a	RXEN-Bit von UCSRB auf 1 setzen	UART-Empfänger einschalten
2b	TXEN-Bit von UCSRB auf 1 setzen	UART-Sender einschalten
3	Warten bis UCSRA.RXC = 1	UCSRA.RXC = 1, wenn ein Byte empfangen wurde; s. u.
4	Inhalt von UDR in Rechenregister merken...	Im UDR-Register steht das empfangene Byte Beim Auslesen dieses Registers wird UCSRA.RXC wieder auf 0 gesetzt.
5	Warten bis UCSRA.RXC = 1	UCSRA.RXC = 1, wenn ein Byte empfangen wurde; s. u.
6	Inhalt von UDR in (weiterem) Rechenregister merken...	Im UDR-Register steht das zweite empfangene Byte. Beim Auslesen dieses Registers wird UCSRA.RXC wieder auf 0 gesetzt.
7	die Inhalte der beiden Rechenregister addieren	Die Summe wird automatisch im ersten Rechenregister abgelegt.

Schritt	Aktion	Erläuterung
8	die Summe ins UDR ausgeben	Das Ergebnis wird automatisch über die UART ausgegeben; eine Warteschleife ist hier nicht nötig, weil nur dieses Byte gesendet wird.
9	Endlosschleife	Beenden des Programms

Es fällt natürlich sofort auf, dass Schritte 5 und 6 im Wesentlichen eine Kopie der Schritte 3 und 4 sind. Das muss so sein, denn schließlich müssen hier jeweils dieselben Aufgaben erfüllt werden. Daher bietet es sich an, diese beiden Schritte zu einem einzigen Unterprogramm zusammen zu fassen und dieses Unterprogramm dann zweimal ausführen zu lassen. Zwar wird in diesem Fall der Maschinencode dabei kaum kürzer, aber das würde sich rasch ändern, wenn dieses Unterprogramm noch häufiger aufgerufen würde. Obendrein geht es uns hier - wie oben schon erwähnt - zunächst darum, das Prinzip zu verdeutlichen.

Das fertige Programm sieht so aus:

```
.include "tn2313def.inc"

.def temp      = r16
.def param     = r17
.def merke     = r18

rjmp init
init:
  ldi temp, 25           ; Baudrate
  out ubrr, temp

  sbi ucsrb, rxen        ; UART-Empfänger einschalten
  sbi ucsrb, txen        ; UART-Sender einschalten

  rcall empfangen        ; ersten Summanden empfangen
  mov merke, param       ; und speichern

  rcall empfangen        ; zweiten Summanden empfangen
  add param, merke       ; und zu merke addieren, Erg. in param

  out udr, param         ; Summe senden

ende:                    ; Endlosschleife
  rjmp ende

empfangen:
  warte:
     sbis ucsra, rxc     ; nächsten Befehl überspringen, wenn
```

```
                                                    Byte empfangen
    rjmp warte              ; weiter warten
  in param, udr             ; empfangenes Byte merken
ret                         ; zurück
```

Wie funktionieren nun die neuen Befehle `rcall` und `ret`? Wenn der Mikrocontroller auf den Befehl `rcall empfangen` stößt, führt er einen relativen Sprung zur Marke `empfangen` durch; darin entspricht er haargenau dem `rjmp`-Befehl: Der Programmzähler wird um einen entsprechenden Betrag k erhöht (vgl. Kapitel über das Assemblieren!):

PC ← PC + k + 1

Es gibt aber einen wesentlichen Unterschied zum `rjmp`-Befehl: Bevor nämlich der Sprung ausgeführt wird, merkt sich der Mikrocontroller beim `rcall`-Befehl die Adresse desjenigen Befehls, der auf den `rcall`-Befehl folgt. In unserem Fall ist dies die Adresse von `mov merke, param`. Warum dies erforderlich ist, werden wir gleich sehen.

Insgesamt werden durch den `rcall`-Befehl also zwei Aktionen durchgeführt:

1. Adresse des folgenden Befehls merken: MerkePC ← PC + 1
2. Zur Marke empfangen springen: PC ← PC + k + 1

Nach dem Unterprogramm-Aufruf `rcall empfangen` steht der Programmzeiger also auf dem ersten Befehl dieses Unterprogramms; der Mikrocontroller arbeitet dann alle folgenden Befehle ab; wie diese funktionieren, wurde im vorigen Kapitel schon dargelegt. Das empfangene Byte wird in dem Rechenregister `param` gespeichert. Schlussendlich stößt der Mikrocontroller auf den Befehl `ret`. Dieser Befehl ersetzt den Programmzähler durch den gemerkten Programmzählerwert:

PC ← MerkePC

Der Programmzähler weist damit auf den Befehl `mov merke, param`. Damit ist der Mikrocontroller an die richtige Stelle des Hauptprogramms zurückgekehrt. An dieses Zurückkehren erinnert auch die Abkürzung des Mnemonics `ret`: to return = Zurückkehren.

Wichtig ist: Egal von welcher Stelle aus das Unterprogramm aufgerufen wird, der Mikrocontroller kehrt immer zu demjenigen Programmschritt zurück, welcher hinter dem Aufruf selbst steht; dies wird durch das Zwischenspeichern des Programmzählers garantiert. Abb. 1 zeigt diesen Rücksprung in einem Speicherplatzdiagramm.

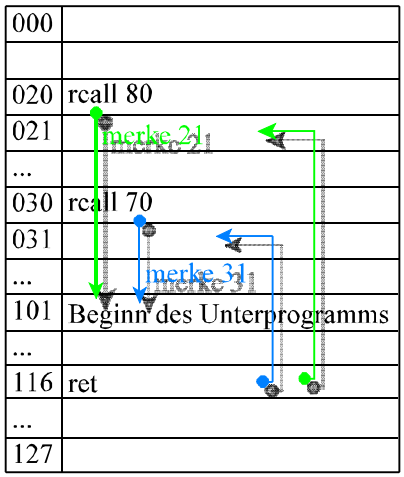

Abb. 1

Nun gehen wir noch einen Schritt weiter: Sämtliche Befehle, die unsere Summationsaufgabe bilden, sollen in einem einzigen Unterprogramm "`aufgabe`" zusammengefasst werden. Dabei wollen wir auf das nun schon vorhandene Unterprogramm `empfange` zurückgreifen. Die Vorteile dieser Vorgehensweise liegen auf der Hand: Einerseits wird das Programm übersichtlicher; andererseits wird aber auch der Programmieraufwand verkürzt, da wir nunmehr auf bereits getestete Unterprogramme benutzen können.

```
.include "tn2313def.inc"

.def temp      = r16
.def param          = r17
.def merke          = r18

rjmp init
init:
  ldi temp, 25           ; Baudrate
  out ubrr, temp
  sbi ucsrb, rxen        ; UART-Empfänger einschalten
  sbi ucsrb, txen        ; UART-Sender einschalten

schleife:
 rcall aufgabe           ; eine Aufgabe
 rjmp schleife

aufgabe:
  rcall empfangen        ; ersten Summanden empfangen
  mov merke, param       ; und speichern
  rcall empfangen        ; zweiten Summanden empfangen
  add param, merke       ; und zu merke addieren, Erg. in param
  out udr, param         ; Summe senden
ret

empfangen:
  warte:
    sbis ucsra, rxc      ; nächsten Befehl überspringen, wenn
                                          Byte empfangen
    rjmp warte           ; weiter warten
    in param, udr        ; empfangenes Byte merken
ret                      ; zurück
```

Neu ist hier, dass ein Unterprogramm von einem Unterprogramm aufgerufen wird; die Unterprogramme sind hier verschachtelt wie in Abb. 2. Es ist klar, dass jetzt für das Merken der Rücksprungadressen eine einzige Speicherzelle MerkePC (1 Wort = 2 Bytes) nicht mehr ausreicht: Stände nur eine einzige 1-Wort-Speicherzelle zur Verfügung, würde die erste Rücksprungadresse für den Aufruf von UPRG1 überschrieben werden, wenn das Unterprogramm UPRG3 aufgerufen wird. Das Programm könnte dann niemals mehr an den Befehl hinter `rcall UPRG1` zurückkehren.

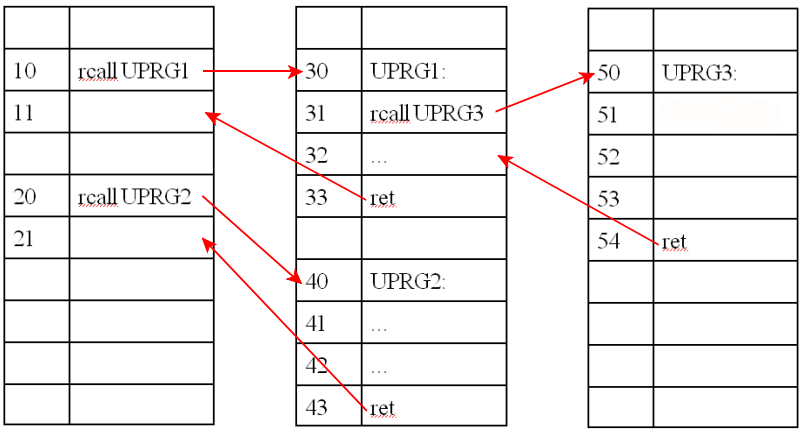

Abb. 2: Verschachtelte Unterprogramme

Für unser Problem braucht der Mikrocontroller also mindestens zwei verschiedene Adressspeicherzellen. Und das ist noch nicht alles: Wenn noch mehr Unterprogramme verschachtelt werden, muss sich der Mikrocontroller auch entsprechend viele Adressen merken. Obendrein muss der `ret`-Befehl immer auf den richtigen Adressspeicher zurückgreifen. Um dies zu erreichen, benutzt der Mikrocontroller einen so genannten Stapel oder stack.

Der Stapel

Mit dem Stapel wird folgende Vorstellung verbunden: Bei jedem Unterprogrammaufruf schreibt der Mikrocontroller die Rückkehradresse auf eine Karte und legt diese der Reihe nach auf einen Stapel. Und wenn ein `ret`-Befehl erfolgt, nimmt der Mikrocontroller die oberste Karte vom Stapel und springt zu der Adresse, die er auf der Karte vorgefunden hat.

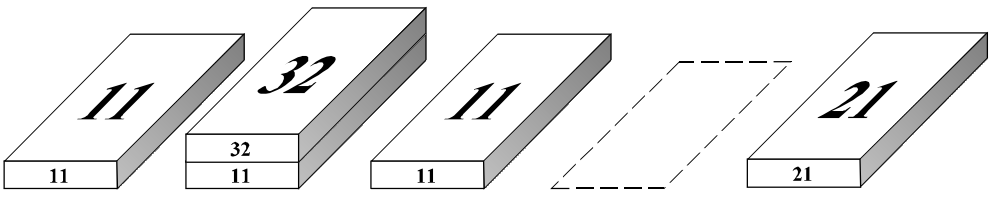

Abb. 3: Stapel mit Rücksprungadressen

Die Abb. 3 macht die Entwicklung des Stapels beim Ablauf des Programms von der vorigenSeite. Es wird deutlich, dass dieses Stapelmodell tatsächlich garantiert, dass immer auf die korrekte Rücksprungadresse zurückgegriffen wird. Bleibt nur noch zu klären, wie dieser Stapel konkret

im Mikrocontroller realisiert wird. Unsere Karten entsprechen natürlich Speicherzellen des Mikrocontrollers. Genauer gesagt handelt es sich um einen speziellen Teil des Datenspeichers.

Unser Attiny besitzt drei verschiedene Speicherbereiche:

- den Flash-Speicher, in dem die Programme abgelegt werden,
- das EEPROM, in dem Daten dauerhaft gespeichert werden können,
- den SRAM, in dem Daten kurzfristig gespeichert werden können.

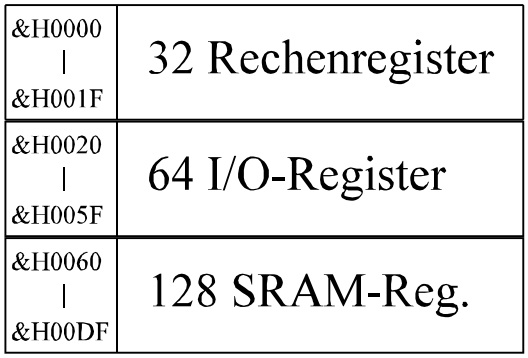

Abb. 4: Aufteilung des Datenspeichers

Der SRAM ist ein Teil des Datenspeichers. Zu dem Datenspeicher gehören auch die schon behandelten Rechenregister und die I/O-Register (Abb. 4). Während die Rechenregisater und die I/O-Register für ganz spezielle Zwecke reserviert sind, können die einzelnen Speicherplätze des SRAM zu ganz verschiedenen Zwecken eingesetzt werden. So legt z. B. BASCOM in diesem SRAM auch die Inhalte von Variablen ab. Dieses SRAM wird aber auch für unseren Stapel benutzt. Der Bereich von &H0000 bis &H001F ist für die 32 Rechenregister r0 bis r31 reserviert, der Bereich von &H0020 bis &H005F ist für die 64 I/O-Register. Die restlichen Zellen des Datenspeichers sind dem SRAM zugeordnet und können für den Stapel benutzt werden.

Der Stapel beginnt bei der Adresse &H00DF. Auf die Speicherzelle mit dieser Adresse weist ein so genannter Stapelzeiger (Stackpointer SP), wenn der Stapel leer ist. Der Stapelzeiger besteht beim Attiny2313 nur aus einem einzigen Byte. Das reicht aus, weil der Attiny weniger als 256 Speicherzellen im Datenspeicher besitzt. Andere Mikrocontroller, die mehr Datenspeicher besitzen, brauchen zwei Bytes dazu. Diesen Stapelzeiger bilden dann die I/O-Register `SPH` (StackPointerHigh) und `SPL` (StackPointerLow). Unser Attiny besitzt nur das I/O-Register `SPL`.

Wird eine Adresse auf den leeren Stapel gelegt, bedeutet dies: die beiden Bytes, welche die Adresse bilden, werden unter den Adressen &H00DF und &H00DE abgelegt. Der Stackpointer zeigt schließlich auf die nächste freie Zelle, also &H00DD. Der Stapel wächst also bezüglich des Adressbereichs nach unten. Die nächste Rücksprungadresse würde in den Zellen &H00DD und &H00DC abgelegt.

In Abb. 5 wurden zwei Rücksprungadressen, nämlich 11 und 32 auf den Stapel gelegt. `SPL` zeigt nun auf die Zelle mit der Adresse &H00DB. Erfolgt nun ein `ret`-Befehl, so werden die beiden Bytes aus den Adressen SPL+1 (hier &H00DC) und SPL+2 (hier &H00DD) geholt und in den Programmzähler geschrieben. Damit springt

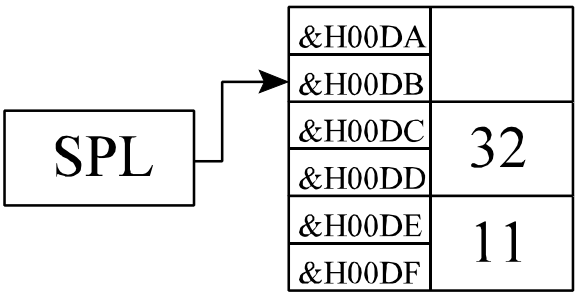

Abb. 5: Diese Situation entspricht dem zweiten Stapel aus Abb. 3

das Programm zu derjenigen Rücksprungadresse, welche als letzte auf den Stapel gelegt wurde (hier 32). Gleichzeitig wird der Stackpointer um 2 erhöht; er zeigt jetzt auf die Zelle &H00DD. Würde ein weiterer `ret`-Befehl folgen, so würde hier nun auf die Rücksprungadresse 21 zugegriffen werden. Würde jetzt hingegen erneut ein Unterprogrammaufruf erfolgen, so würde die zugehörige Rücksprungadresse in die Zellen &H00DD und &H00DC geschrieben; die alte Rücksprungadresse 32 würde dabei überschrieben.

Atmel schreibt für den Attiny2313 eine Initialisierung des Stacks vor. Im Manual des Attiny2313 findet man dazu die folgenden Zeilen:

```
ldi r16, &HDF
out SPL, r16
```

Allerdings wird bei jedem Reset und jedem Neustart des Attiny automatisch eine Initialisierung des Stackpointers vorgenommen. Insofern kann auch auf die explizite Initialisierung verzichtet werden. Manches Simulationsprogramm setzt jedoch für den Einsatz eines Stacks eine explizite Initialisierung voraus.

Kapseln von Unterprogrammen

Bei größeren Programmen kann es passieren, dass der Überblick über die benutzten Rechenregister verloren geht. So kann es geschehen, dass der Wert eines Rechenregisters aus dem Hauptprogramm ungewollt in einem Unterprogramm überschrieben wird:

```
...
ldi r16, 85
...
rcall test
...
out portb, Ldi      ; hier soll eigentlich 85 über Port B
                                    ausgegeben werden
...

test:
ldi r16, 25
out udr, r16        ; 25 über UART ausgeben
ret
```

Statt der gewünschten Zahl 85 wird hier die Zahl 25 über das Port B ausgegeben. Natürlich kann man diesen Fehler vermeiden, indem man im Unterprogramm ein anderes Rechenregister benutzt. Aber auch dies könnte gegebenenfalls störend auf das Hauptprogramm wirken.

Einen sicheren Ausweg bietet hier unser Stapel. Auf ihm können wir die Inhalte der der Rechenregister zwischenlagern. Dazu benutzt man die Befehle `push` und `pop`. In unserem Beispiel muss nur der Inhalt von `r16` zwischenzeitlich gerettet werden. Dazu schieben wir den Inhalt `r16` zu Beginn des Unterprogramms auf den Stapel:

```
push r16
```

Mit dem `push`-Befehl ändert sich natürlich auch der Stackpointer. Er weist jetzt auf die Zelle, in der die Zahl 85 zwischengelagert ist. Am Ende des Unterprogramms holen wir diesen Wert wieder vom Stapel und legen ihn im Register `r16` ab:

```
pop r16
```

Jetzt zeigt der Stackpointer wieder auf die Rücksprungadresse. Der Mikrocontroller kann so korrekt ins Hauptprogramm zurückkehren und steht obendrein auch der richtige Wert 85 im Register `r16`.

Das vollständige Unterprogramm lautet damit

```
test:
push r16            ; r16 auslagern
ldi r16, 25
out udr, r16        ; 25 über UART ausgeben
pop r16             ; r16 wiederherstellen
ret
```

Wenn man also in einem Unterprogramm alle benutzten Rechenregister zu Beginn mit `push`-Befehlen rettet und am Ende mit `pop`-Befehlen wieder herstellt, kann es keine ungewünschte Beeinflussung zwischen Haupt- und Unterprogramm geben. Wichtig ist dabei: Zu jedem `push` am Anfang des Unterprogramms muss es ein `pop` am Ende geben.

Aufgaben

1. Schreiben Sie für das Senden des Ergebnisbytes bei unserem Additionsprogramm ein Unterprogramm. Wie muss das hauptprogramm dann abgeändert werden?

2. Beim Kapseln eines Unterprogramms wurde ein `pop`-Befehl vergessen. Wie wirkt sich dies aus?

3. Wie verändert sich der Stackpointer bei einem `push`- bzw. `pop`-Befehl?

4. Bei einem Unterprogramm mussten drei Register gerettet werden. Dazu wurden die folgenden Befehle benutzt:

```
push a
push b
push c
...
pop a
pop b
```

```
pop c
ret
```

Was ist falsch? Warum? Verbessern Sie die falschen Befehle.