A6 Unterprogramme

Die letzte Aufgabe hat gezeigt, dass es nicht sehr praktisch ist gewisse Programmteile im Code mehrfach zu wiederholen. Diese Programmteile lassen sich sehr günstig in einem Unterprogramm[25] (*subroutine*) zusammenfassen.

Unterprogramme sind Programmteile, die von einem Hauptprogramm aus (oder aus einem anderen Unterprogramm) aufgerufen werden und nach der Ausführung wieder an dieselbe Stelle des Hauptprogramms zurückspringen.

Häufig stellt man bei der Analyse einer neuen, umfangreichen Programmieraufgabe fest, dass bestimmte Schrittfolgen mehrfach in derselben Weise vorkommen. Hier setzt die Unterprogramm-Technik ein.

Unterprogramme sind also Hilfsprogramme, die für Sonderaufgaben eingesetzt werden.

Eigenschaften von Unterprogrammen:

- Unterprogramme unterteilen die Aufgabe in Teilprobleme, die sich einzeln besser programmieren und testen lassen. Oft ist es von Vorteil, zuerst die Unterprogramme zu entwerfen und zu testen, bevor mit der Programmierung des Hauptprogramms begonnen wird.
- Unterprogramme liegen oft bereits als fertige Lösungen in einer Bibliothek vor oder können der Literatur entnommen werden.
- Unterprogramme verkürzen das Hauptprogramm da gleiche Programmteile nicht mehrmals wiederholt werden müssen.
- Unterprogramme sind vergleichbar mit Funktionen und Prozeduren in Programmier-Hochsprachen.

Unterprogramme werden mit dem "**rcall**"-Befehl aus einem Programm heraus aufgerufen! Im Unterprogramm selbst ist der einzige spezifisch Befehl der "**ret**"-Befehl (Rücksprung), der bewirkt, dass der Programmablauf an der Stelle gleich hinter den Aufruf (**rcall**) weitergeführt wird. Dazu muss eine Rücksprungadresse auf dem Stapel (*stack*), einem reservierter Speicherbereich im SRAM, abgelegt werden. In einem späteren Kapitel wird im Modul B genauer auf diesen Stapel eingegangen.

rcall k

Relativer Aufruf eines Unterprogramms
(Adresskonstante k) (*relative call to subroutine*).

1	1	0	1	k	k	k	k	k	k	k	k	k	k	k	k

Relative Aufruf (max. 2 KiWorte vor und rückwärts). Die Rücksprungadresse wird auf dem Stapel abgelegt.
Beeinflusste Flags: keine Taktzyklen: 3

25 Ein Synonym für Unterprogramme ist die Bezeichnung **Routine**.

Neben dem relativen "**rcall**"-Befehl existiert wie bei den Sprüngen auch der direkte "**call**"-Befehl. Beim relativen "**rcall**"-Befehl kann maximal über 2Ki Worte vorwärts und rückwärts gesprungen werden. Der Assembler überwacht beim "**rcall**"-Befehl ob die Grenze überschritten wird und meldet in diesem Fall einen Fehler. Soll im Programm weiter gesprungen werden, so wird der direkte "**call**"-Befehl verwendet.

ret

Rücksprung aus einem Unterprogramm (*return from subroutine*).

1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0

Die Rücksprungadresse wird vom Stapel genommen.
Beeinflusste Flags: keine Taktzyklen: 4

Die relative Befehl ist schneller und Platz sparender als der direkte Befehl. Man soll also, falls möglich, den relativen Befehl verwenden.

Der Vollständigkeit halber soll auch noch der indirekte "**icall**"-Befehl erwähnt werden, mit dem ein Unterprogramm indirekt aufgerufen werden kann (Adresse in **Z**).

Damit Unterprogramm-Aufrufe funktionieren können, muss der Stapel wie in der Assemblervorlage (Kapitel A2) initialisiert werden!

```
;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
out     SPH,Tmp1                ;datei festgelegt
ldi     Tmp1,LOW(RAMEND)
out     SPL,Tmp1
```

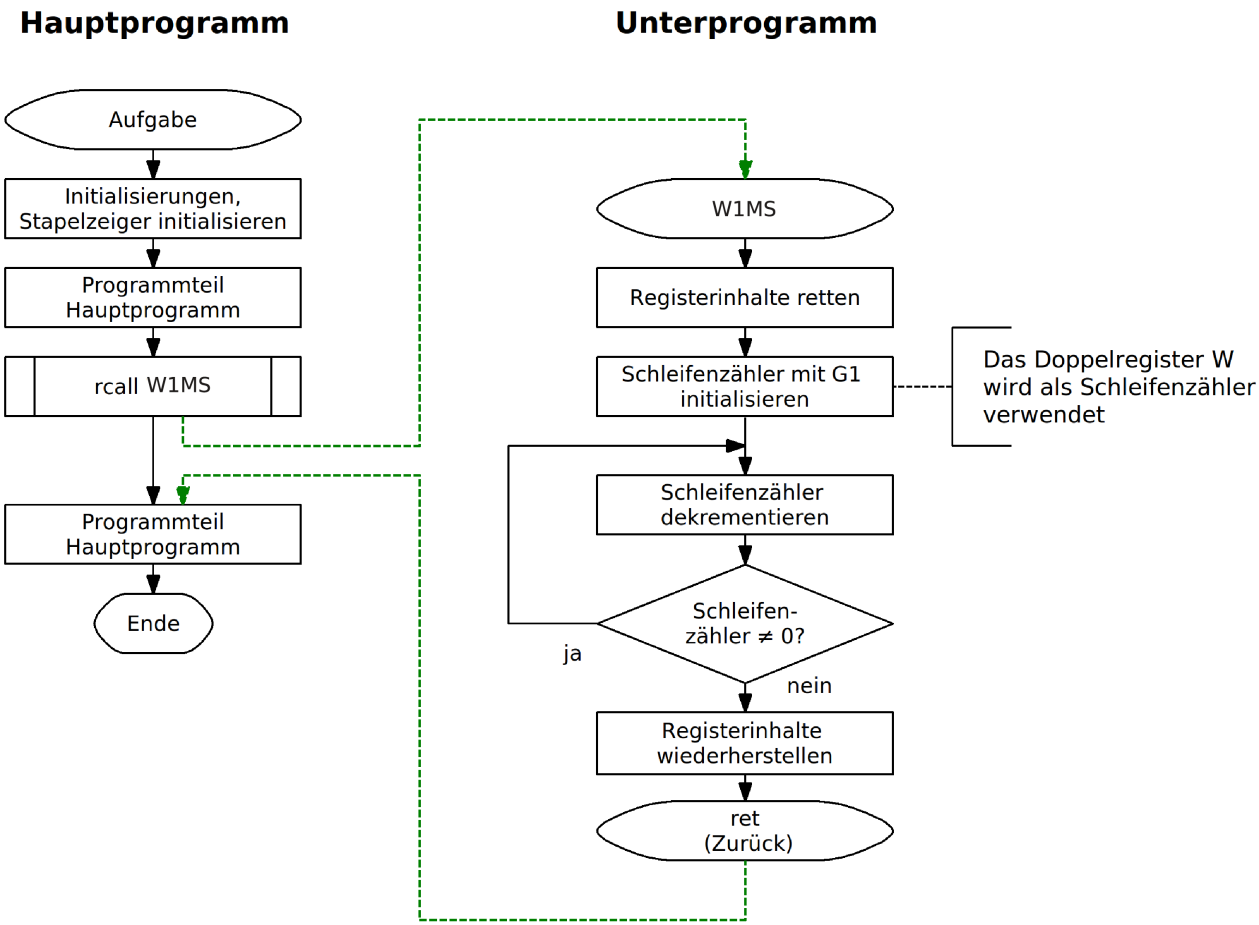
!Achtung!: Unterprogramme können nur verwendet werden, wenn der Stapel vorher initialisiert wurde!

Das Unterprogramm wird mit seinem Namen (symbolische Adresse bzw. Label) aufgerufen. Der Label soll zur besseren Unterscheidung zu den Befehlen mit großen Buchstaben geschrieben werden und eine sinnvoll sein, also einen Hinweis auf die Funktion des Unterprogramms liefern.

Beispiel:

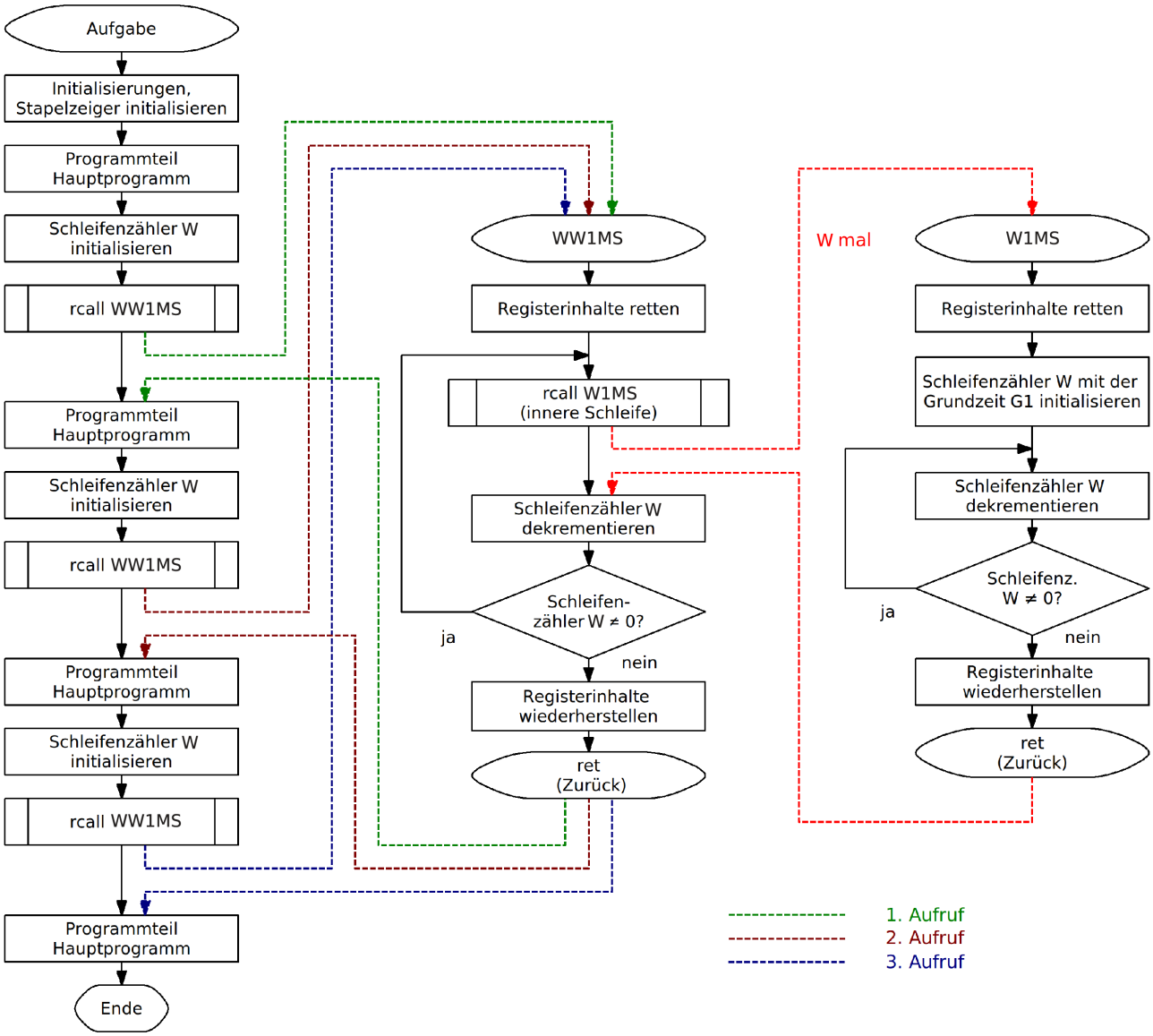
```
rcall   W1MS    ;rufe ein Zeitschleifenunterprogramm, das 1 MilliSekunde lang wartet
```

Das Verhalten eines Programms beim Aufruf von Unterprogrammen wird mit Hilfe folgender Flussdiagramme dargestellt (Ausnahmsweise wurden zum besseren Verständnis Befehle (**rcall** und **ret**) im Flussdiagramm eingetragen). Aus einem Hauptprogramm heraus wird eine Zeitschleife als Unterprogramm aufgerufen. Die gestrichelten grüne Linien deuten den Aufruf des Unterprogramms Nach der Abarbeitung des Unterprogramms bewirkt der "**ret**"-Befehl, dass das Programm im Hauptprogramm gleich hinter dem "**rcall**"-Befehl weiterarbeitet.



Unterprogramme können auch aus Unterprogrammen heraus aufgerufen werden. Als Beispiel soll eine verschachtelte Zeitschleife dreimal aus dem Hauptprogramm heraus aufgerufen werden. Die innere Schleife der verschachtelten Zeitschleife wurde als eigenes Unterprogramm ausgekoppelt.

Bemerkung: Mit Sprungbefehlen wäre ein dreimaliger Aufruf eines Programmteils nicht zu bewerkstelligen, da man sonst mit einem Sprungbefehl drei unterschiedliche Labels anspringen müsste.



Globale Variablen

Zur Speicherung von Variablen werden meist die Arbeitsregister verwendet. Die Variablen sind dadurch im ganzen Programm, also auch in den Unterprogrammen sichtbar. Es kann von überall auf diese Variablen zugegriffen werden. Es ist also nicht unbedingt nötig den Unterprogrammen oder Interrupt-Routinen extra Parameter[26] zu übergeben. Das Unterprogramm muss nur wissen in welchem Arbeitsregister sich die Information befindet.

26 Parameter sind die Informationen (Daten) die zwischen Hauptprogramm und Unterprogramm ausgetauscht werden. Die Parameterübergabe kann natürlich auch statt über Register über den Speicher (Stapel, SRAM) geschehen, und das sogar auch indirekt indem nur die Adresse übermittelt wird.

Lokale Variablen

Unterprogramme benötigen zur Bewältigung ihrer Aufgabe oft interne Variablen (Register) die nur im Unterprogramm benötigt werden. Diese Variablen werden als lokale Variablen bezeichnet.

Da Unterprogramme flexibel eingesetzt werden, ist allerdings nicht immer bekannt, welche Register vom Hauptprogramm schon besetzt sind. Es ist deshalb wichtig die den Inhalt der Register, die zur Speicherung von lokale Variablen benutzt werden sollen, zuerst in eine Speicherzelle zu retten und nach dem Abarbeiten des Unterprogramms wiederherzustellen. Es werden dazu Speicherzellen des Stapels verwendet. Das Retten und Wiederherstellen der Registerinhalte geschieht mit den beiden Befehle "**push**" und "**pop**".

Retten und Wiederherstellen mit "push" und "pop"

Mit dem Befehle "**push r16**" wird der Inhalt vom Arbeitsregister **r16** auf dem Stapel abgelegt. Mit "**pop r16**" wird der Inhalt des Registers wiederhergestellt. Der Stapel arbeitet nach dem **LIFO-Prinzip** (**Last In First Out**), also wie ein richtiger Papierstapel. Das letzte Blatt das abgelegt wurde wird auch wieder als erstes entnommen.

push Rr

Kopiere Inhalt von Register Rr auf den Stapel (*push register on stack*).

1	0	0	1	0	0	1	r	r	r	r	r	1	1	1	1

Nach dem Kopieren wird der Stapelzeiger um 1 dekrementiert.
Beeinflusste Flags: keine Taktzyklen: 2

pop Rd

Kopiere Inhalt vom Stapel ins Register Rd (*pop register from stack*).

1	0	0	1	0	0	0	d	d	d	d	d	1	1	1	1

Vor dem Kopieren wird der Stapelzeiger um 1 inkrementiert.
Beeinflusste Flags: keine Taktzyklen: 2

Der Code für ein typisches Unterprogramm, das die Register **r16** bis **r18** als interne Variablen benutzt, könnte folgendermaßen aussehen:

```
;-------------------------------------------------------------------------
;       Unterprogramm mit drei internen Variablen
;-------------------------------------------------------------------------

SR_3WR: push    r16         ;Alle drei Variablen auf den Stapel retten
        push    r17
        push    r18
        ;
        ;Code des Unterprogramms
        ;
```

```
        pop     r18         ;Alle drei Variablen wiederherstellen
        pop     r17         ;Achtung !! Umgekehrte Reihenfolge !!
        pop     r16
        ret                 ;Ruecksprung ins Hauptprogramm
```

Bemerkungen: Innerhalb eines Unterprogramms sollen keine Definitionen (**.DEF**) oder Zuweisungen (**.EQU**) verwendet werden, da nicht bekannt ist welche Definitionen bzw. Zuweisungen das Hauptprogramm schon benutzt. Für interne Labels soll eine Erweiterung des Unterprogrammlabels zur Anwendung kommen um so doppelte Belegung von Labeln zu vermeiden (Bsp: Unterprogrammlabel: W1ms; interne Labels: W1ms_1, W1ms_2 ...).

Soll ein Unterprogramm zu jedem beliebigen Zeitpunkt aufgerufen werden können, so soll man auch das Statusregister **SREG** auf den Stapel retten. Dadurch werden im Hauptprogramm, durch den Aufruf des Unterprogramms, die Statusbits nicht verändert. Das Retten des SF-Register SREG muss über ein Arbeitsregister erfolgen!
Beispiel:

```
;-------------------------------------------------------------------------
;       Unterprogramm mit zwei internen Variablen, SREG wird auch gerettet
;-------------------------------------------------------------------------

SR_3WS: push    r16         ;r16 auf den Stapel retten
        in      r16,SREG    ;SREG in einen Zwischenspeicher laden
        push    r16         ;SREG auf den Stapel retten
        push    r17         ;r17 auf den Stapel retten
        ;
        ;Code des Unterprogramms
        ;
        pop     r17         ;r17 wiederherstellen ! Umgekehrte Reihenfolge !
        pop     r16         ;SREG wiederherstellen
        out     SREG,r16    ;
        pop     r16         ;r16 wiederherstellen
        ret                 ;Ruecksprung ins Hauptprogramm
```

Externe Unterprogramme Einbinden mit ".INCLUDE"

In Assembler gibt es keine so schöne kompakte Befehle wie in den Hochsprachen. Trotzdem wird Assembler-Programmieren fast genauso bequem, wenn man sich die entsprechenden Befehle mit Unterprogrammen nachbildet und diese in einer Bibliothek sammelt (oder bestehende Bibliotheken benutzt).

Zeitschleifen werden immer wieder benötigt, und im Folgen soll eine kleine Bibliothek mit unterschiedlichen Zeitschleifen erstellt werden. Die entsprechende Datei soll mit dem Namen "**SR_TIME_16M.asm**" versehen werden und einige Zeitschleifen-Unterprogramme enthalten die mit einem externen Quarz von 16 MHz funktionieren.

Um den Sachverhalt zu verdeutlichen soll im folgenden der Code für die Unterprogramme aus dem obigen Flussdiagramm dargestellt werden.

Zuerst die 16-Bit-Zeitschleife, hier mit einer Grundzeit von 1ms:

```
;***************************************************************************
;*
;*      Titel:  Bibliothek mit Zeitunterprogrammen (SR_TIME_16M.asm)
;*      Datum:  10/05/07        Version:        0.1
;*      Autor:
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32A       Quarzfrequenz:   extern 16MHz
;*
;***************************************************************************

;---------------------------------------------------------------------------
;       16-Bit-Zeitschleife wartet 1ms (Wait 1ms)
;---------------------------------------------------------------------------
W1MS:   push    r24     ;rette verwendete Registerinhalte
        push    r25
        ldi     r24,0x9C ;Initialisiere den Schleifenzaehler W
        ldi     r25,0x0F ;G = t-tT/4tT = 1ms/4*62,5ns-1/4 = 4000-1/4
                         ;Korrektur: -15tT fuer rcall, 2 push, 2 pop, ret
                         ;1 Durchlauf der Schleife benoetigt 4tT
                         ;GKorrektur = G-1/4-15/4 = 4000-4 = 3996 = 0x0F9C
                         ;(Korrektur koennte man vernachlaessigen,Fehler 0,1%)
W1MSL:  sbiw    r24,1   ;Dekrementiere den Schleifenzaehler W
        brne    W1MSL   ;Verlasse die Schleife bei Schleifenzaehler W = 0
        pop     r25     ;Wiederherstellung der verwendeten Registerinhalte
        pop     r24
        ret             ;Ruecksprung ins Hauptprogramm
```

Dann die verschachtelte 16-Bit-Zeitschleife, bei der der Startwert des Schleifenzählers über das Doppelregister **X** übergeben wird:

```
;---------------------------------------------------------------------------
;       Verschachtelte 16-Bit-Zeitschleife wartet W mal 1ms (Wait W*1ms)
;       Der Anfangswert von W (r25:r24) muss im Hauptprogramm gesetzt werden!
;       Der Fehler durch die zusaetzlichen Befehle kann hier vernachlässigt
;       werden (max Fehler bei einmaligem Aufruf < 0,12%).
;---------------------------------------------------------------------------
WW1MS:  push    r24     ;rette verwendete Registerinhalte
        push    r25
WW1MSL: rcall   W1MS    ;Rufe innere Schleife (Unterprogramm W1ms)
        sbiw    r24,1   ;Dekrementiere den Schleifenzaehler W
        brne    WW1MSL  ;Verlasse die Schleife bei Schleifenzaehler W = 0
        pop     r25     ;Wiederherstellung der verwendeten Registerinhalte
        pop     r24
        ret             ;Ruecksprung ins Hauptprogramm
```

Diese Programmteile werden in der Datei "**SR_TIME_16M.asm**" abgespeichert. Um das ganze übersichtlicher zu gestalten soll unser Arbeitsverzeichnis ein Unterverzeichnis mit der Bezeichnung **lib** erhalten, in dem sich die erstellten Assemblerbibliotheken befinden. Eingebunden wird die Datei mit der "**.INCLUDE**"-Direktive (siehe Kapitel A2).

Ein einfaches Hauptprogramm bei dem eine LED im Sekundentakt aufleuchtet könnte dann folgendermaßen aussehen:

```
;---------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;---------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
```

```
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    Mask = r19              ;Register 19 dient zur Maskierung
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe
.DEF    W = r24

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,LOW(RAMEND)        ;RAMEND (SRAM) ist in der Definitions-
        out     SPL,Tmp1                ;datei festgelegt
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1

        sbi     DDRD,0          ;nur PD0 als Ausgang initialisieren
        ldi     Mask,0x01       ;Setze Maske zur Invertierung (PD0)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   ldi     WL,LOW(500)     ;16-Bit-Schleifenzaehler mit
        ldi     WH,HIGH(500)    ;G2 = 500 t = 0,5s initialisieren
        rcall   WW1MS           ;Zeitschleife aufrufen
        ;Toggeln und Ausgabe
        in      Tmp1,PIND
        eor     Tmp1,Mask       ;Invertiere mit EXOR
        out     PORTD,Tmp1      ;LED (Pin PD0) wechselt Zustand
        rjmp    MAIN            ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
.INCLUDE "lib/SR_TIME_16M.asm"

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

1. Bemerkung: Ruft man die Zeitschleife **WW1MS** mit einem leeren **W**-Register (**W** = 0) auf, so wird diese nicht Null-Mal sondern **65536**-mal aufgerufen, da ja vor der Kontrolle dekrementiert wird. Dies kann unerwünschten Effekten führen, wenn zum Beispiel eine Midi-Datei (Musikdatei) abgespielt wird, wo die Notendauer auch Null sein kann! Es ist also sinnvoll das Unterprogramm um einige Zeilen zu erweitern:

```
WW1MS:  tst     r24      ;Falls W = 0 schnellstmoeglich zurueck
        brne    WW1MSP
        tst     r25
        brne    WW1MSP
        ret
WW1MSP: push    r24      ;rette verwendete Registerinhalte
        push    r25
WW1MSL: rcall   W1MS     ;Rufe innere Schleife (Unterprogramm W1MS)
        sbiw    r24,1    ;Dekrementiere den Schleifenzaehler W
        brne    WW1MSL   ;Verlasse die Schleife bei Schleifenzaehler W = 0
        pop     r25      ;Wiederherstellung der verwendeten Registerinhalte
        pop     r24
        ret              ;Ruecksprung ins Hauptprogramm
```

tst Rd

Teste ob das Register Rd Null oder Negativ ist (*test for zero or minus*).

0	0	1	0	0	0	d	d	d	d	d	d	d	d	d	d

Entspricht einer logischen Und-Verknüpfung des Registers mit sich selbst (AND Rd,Rd) .

Beeinflusste Flags: S, V (0), N, Z Taktzyklen: 1

2. Bemerkung: Es besteht auch die Möglichkeit eine vom Quarz unabhängige Zeitschleifenbibliothek zu programmieren. Dabei lässt man den Assembler den Anfangswert der Zeitschleifen beim Übersetzen errechnen. Im Hauptprogramm definiert man zum Beispiel die Konstante **Freq** und weist ihr die Quarz- oder Oszillatorfrequenz zu (Bsp.: **.EQU Freq = 8000000**) mit der dann in der Bibliothek umgerechnet wird. Zum Beispiel für die Zeitschleife mit einer Millisekunde:

```
        ldi     XL,LOW(Freq/4000-4)
        ldi     XH,HIGH(Freq/4000-4)
```

Leider sind die Rechenmöglichkeiten des Assembler recht eingeschränkt, so dass die Formel ein wenig umgestellt werden musste (Division vor Subtraktion). Ausgehend von der präzisen Formel und einer Korrektur von 15 Taktzyklen (**push**, **call**, ...) ergibt sich:

$$G = \frac{t - t_T}{4t_T} = \frac{t}{4t_T} - \frac{1}{4} \qquad \text{-Korrektur} \frac{15t_T}{4t_T \text{ pro Durchlauf}}:$$

$$G = \frac{t}{4t_T} - \frac{16}{4} = \frac{t}{4t_T} - 4 = \frac{t}{4\left(\frac{1}{Freq}\right)} = \frac{Freq \cdot t}{4} - 4$$

$$G = \frac{Freq \cdot 1\text{ms}}{4} - 4 = \frac{Freq \cdot \frac{1}{1000\text{s}}}{4} - 4 = \frac{Freq}{4000} - 4 \qquad \text{(Freq in Hertz!)}$$

✎ **A600**

a) Ergänze die Zeitschleifen-Bibliothek "**SR_TIME_16M.asm**" um eine präzise Zeitschleife mit 1 Mikrosekunde (**W1US**; nutze dazu den "**NOP**"-Befehl).

b) Ergänze die Zeitschleifen-Bibliothek um folgende Zeitschleifen mit fester Zeit: **W10US**, **W100US**, **W10MS**, **W100MS**, **W500MS**, **W1S**, **W5S**, **W10S**, **W1MIN**, **W10MIN**.

c) Ergänze die Zeitschleifen-Bibliothek zusätzlich um folgende Zeitschleifen mit veränderbarer Zeit über das W-Register: **WW100US**, **WW10MS**, **WW100MS**, **WW1S**, **WW1MIN**.

d) Teste alle Zeitschleifen. Gib dazu ein Rechtecksignal an **PA0** aus und kontrolliere die Frequenz mit einem Zähler. Nenne das neue Programm "**A600_test_time_lib.asm**".

✎ **A601** Ändere das Programm "**A508_pulse_delay_1.asm**" aus dem vorigen Kapitel so um, dass es mit der erstellten Bibliothek arbeitet. Nenne das neue Programm "**A601_pulse_delay_2.asm**".

✎ **A602** Zeichne ein Flussdiagramm und schreibe ein Programm für einen 8-Bit-Vorwärtszähler, dessen Schrittweite an vier Schaltern (Port C, **PC0** bis **PC3**) während des Zählens eingestellt werden kann und dessen Stand auf acht Leuchtdioden (Port D) ausgegeben wird. Der Zähler soll mit einer Geschwindigkeit von einem Schritt pro Sekunde zählen. Der Anfangswert des Zählers beträgt **0**. Die Schalterabfrage und die Ausgabe an die LEDs soll als Unterprogramm realisiert werden.
Nenne das Programm "**A602_counter_1.asm**".

✎ **A603** Zeichne ein Flussdiagramm und schreibe ein Programm das die Quersumme einer über acht Schalter (Port D) eingelesene Hex-Zahl bildet. Vor dem Rücksprung zum erneuten Einlesen einer Hex-Zahl, wird die Quersumme für drei Sekunden auf fünf Leuchtdioden (Port C, **PC0**-**PC4**) ausgegeben.

Beispiel: eingelesene Zahl: **0x87** = **1000 0111**
Quersumme: **0x08** + **0x07** = **0x0F**

Die Bildung der Quersumme geschieht in einem Unterprogramm. Die Arbeitsregister **r24** und **r25** dienen der Parameterübergabe. Das Unterprogramm befindet sich in der gleichen Datei wie das Hauptprogramm. Nenne das Programm "**A603_checksum.asm**".

✎ **A604** Ergänze das Programm "**A602_counter_1.asm**" um die Fähigkeit vor- und rückwärts zu zählen. Die Umschaltung erfolgt durch einen zusätzlichen Schalter (**PC4**). Rückwärts wird bei betätigtem Schalter gezählt. Zusätzlich soll dieses neue Programm nur gerade Zahlen zum Zählen benutzen. Nenne das Programm "**A604_counter_2.asm**".