A2 Assemblerprogrammierung

In diesem Kapitel sollen einige Grundlagen zur Assemblerprogrammierung erläutert werden, und es soll eine Vorlage erstellt werden, die für alle zukünftige Programme verwendet werden kann.

Assembler ist eine Computersprache und gleichzeitig auch die Bezeichnung für ein Übersetzungsprogramm, das diese Sprache in den Maschinencode eines Controllers oder Prozessors überführt. Für jede Prozessor- oder Controller-Familie gibt es eigene Assembler-Übersetzungsprogramme, die meist vom Hersteller bereitgestellt werden.

Assembler ist keine Hochsprache wie Pascal, Fortran, Basic oder C sondern eine sehr maschinennahe Sprache. Sie kennt praktisch nur einzelne Maschinenbefehle, ermöglicht es aber mit Hilfe von Unterprogrammen, Makros, bedingter Assemblierung und Include-Dateien strukturierten und übersichtlichen Code zu gestalten. Assembler ist besonders gut geeignet um den Aufbau und die Funktionsweise eines Prozessors oder Controllers kennen zu lernen und zu verstehen.

Um den Kurs übersichtlich zu halten werden im folgenden allerdings weder Makros noch bedingte Assemblierung verwendet. Eine Ausnahme sind die fertigen Bibliotheken, die auf der Homepage (http://weigu.lu/a/asm) bereitstehen.

Zum besseren Verständnis der Programme werden Flussdiagramme zu den Programmen erstellt.

Dabei ist das Erstellen der Flussdiagramme immer der erste Schritt vor dem Schreiben des Programmcodes.

✎ **A200** Arbeite das Kapitel "**Das Flussdiagramm**" im Anhang durch.

Ein Assemblerprogramm ist eine reine Textdatei (Quelle). Diese kann mit jedem beliebigen Texteditor erstellt werden (Notepad, Word, Kate...). Natürlich kann man auch den im Studio 4 von ATMEL® enthaltenen Editor verwenden (Kapitel "Das erste Programm" im Anhang).

Der AVR-Assembler unterscheidet nicht zwischen Klein- und Großschrift.

```
        ldi     Tmp1,0xFF       ;DDRD = 11111111b
        out     DDRD,Tmp1       ;alle Bits im Datenrichtungsregister auf Eins
;-------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------
MAIN:   ser     Tmp1            ;PORTD = 11111111b
        out     PORTD,Tmp1      ;Alle PortD-Pins auf High setzen (LEDs ein)
```

Der Assembler überprüft die Syntax und übersetzt den Quellcode, wenn kein Fehler gefunden wurde, in die Maschinensprache (Hexcode). Zusätzlich können Mapdateien (**.map**), Listdateien (**.lst**) und Objectdateien (**.obj**) erstellt werden.

Die Listdatei und die Mapdatei sind Textdateien.

Die **Listdatei** listet alle Assembler-Befehle auf, wie sie auf den Controller geladen werden.

```
44: 00002A   EF0F  ldi     Tmp1,0xFF       ;DDRD = 11111111b
45: 00002B   BB01  out     DDRD,Tmp1       ;alle Bits im Datenrichtungsreg. auf 1
46: 00002C   EF0F  ser     Tmp1            ;PORTD = 11111111b
47: 00002D   BB02  out     PORTD,Tmp1      ;Alle PortD-Pins auf High setzen
```

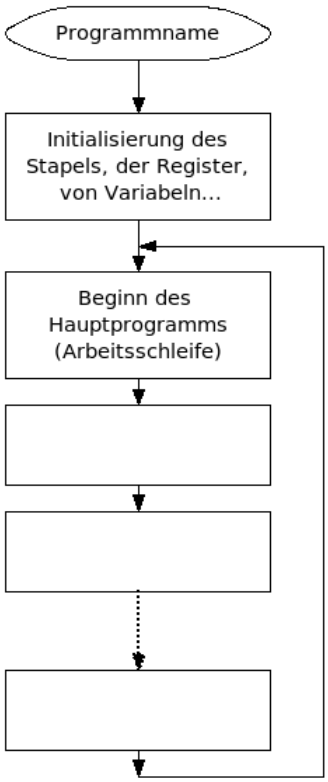
Links steht nach einer Zeilennummer die Adresse, der Maschinencode des Befehls und danach der Maschinencode in Assembler-Darstellung.

Eine **Mapdatei** gibt Auskunft darüber, an welchen Adressen Code und Objekte landen.

```
EQU  INT_VECTORS_SIZE 0000002a
CSEG RESET        00000000
CSEG INIT         0000002a
DEF  Tmp1         r16
CSEG MAIN         0000002c
CSEG END          0000002e
```

Die Datei die für den Programmer benötigt wird ist die **Hexdatei** (**.hex**), in der die Maschinensprache enthalten ist. Diese Datei wird durch den Programmer in den Speicher des ATmega-Controllers geladen.

```
:020000020000FC
:0200000029C015
:0A0054000FEF01BB0FEF02BBFFCF5F
:00000001FF
```



Das nebenstehende Flussdiagramm zeigt den prinzipiellen Ablauf eines Assemblerprogramms. Nach dem Einschalten der Versorgungsspannung beginnt das Programm mit dem ersten Befehl, der sich auf der Adresse **0x0000** (**0x** bedeutet, dass wir das hexadezimale Zahlensystem verwenden) im Programmspeicher (Flash) befindet. Als erstes müssen diverse Initialisierungen vorgenommen werden. Dann folgt die Arbeitsschleife (**MAIN**), welche natürlich auch Verzweigungen, Schleifen und Unterprogramme enthalten kann.

Die Assembler-Programmiervorlage

(A2_template.asm)

Um das Assembler-Grundgerüst nicht immer neu schreiben zu müssen wird hier eine Vorlage bereitgestellt, welche für alle eigenen Programme verwendet werden kann.

Schreibt man ein neues Programm, so wird zuerst die Vorlage geöffnet, und sofort wieder unter dem Namen des zu schreibenden Programms abgespeichert. Nicht benötigte Teile in der Vorlage werden einfach gelöscht.

Die Vorlage reduziert die Tipparbeit, und ermöglicht eine einheitliche Struktur aller Programme. Die vorgegebene Position für Programmteile vermeidet Fehler.

Anhand der Vorlage sollen einige der wichtigsten Assemblereigenheiten erklärt werden:

Eine Eingabezeile im Assembler besteht aus mehreren Feldern. Felder werden durch mindestens ein Leerzeichen getrennt. Es erhöht allerdings die Übersichtlichkeit, wenn statt Leerzeichen Tabulatoren verwendet werden.

```
[Label:]        Direktive oder Befehl [Operanden]      [;Kommentar]
```

Die in eckigen Klammern stehenden Teile können entfallen. Natürlich kann eine Zeile auch ausschließlich aus Kommentar oder einem Label bestehen.

- **Label** sind vom Anwender bestimmte **Namen für Adressen**. Diese **symbolischen Namen** erleichtern das Verständnis des Programms, da sie aussagekräftiger als reine Adressen sind. Der Assembler ersetzt beim Assemblieren die Labels durch die entsprechenden Adressen.

- **Direktiven** sind **Assembleranweisungen**, die dem Assembler mitteilen was dieser beim Assemblieren tun soll. Sie sind nicht zu verwechseln mit den Befehlen.

- **Befehle** mit ihren **Operanden** teilen dem Mikrocontroller mit was dieser tun soll. Der Befehlssatz eines Mikrocontrollers (siehe Anhang) zeigt welche Aktionen dieser Mikrocontroller beherrscht. Ein Befehl besteht aus dem Opcode und den Operanden. Es gibt Befehle ohne Operand, welche wo nur ein Operand benötigt wird und Befehle mit zwei Operanden. Bei zwei Operanden steht immer zuerst das Ziel (links) und dann die Quelle (rechts)!!

  Befehl Ziel,Quelle

  Beispiel: `ldi r16,0x0A ;lade 10 dez. ins Arbeitsregister 16`

- **Kommentare** sind wichtige Ergänzungen die zum Verständnis des Programms beitragen. Sie beginnen mit einem Strichpunkt!

Besonders die **Assembler-Direktiven** ermöglichen uns übersichtliche und verständliche Assemblerprogramme zu schreiben.

Direktiven werden nicht in Maschinencode übersetzt (Pseudo-Befehle!), sondern dienen nur der Steuerung des Übersetzungsvorgangs. Sie beginnen mit einem Punkt und stehen meist gleich am Anfang der Zeile.

Folgende Direktiven werden verwendet:

Direktive	Operand	Beschreibung
.LIST		Listing-Ausgabe einschalten (*default*). Der produzierte Code wird von Menschen lesbar in einer ***.LST**-Datei ausgegeben.
.NOLIST		Listing-Ausgabe ausschalten.
.INCLUDE	"**Textdatei**" oder <**Textdatei**>	Fügt eine externe Textdatei ein (als ob deren Inhalt an dieser Stelle stünde). Es kann sich hierbei z.B. um einen Programmteil in Assembler, ein Unterprogramm oder eine Definitionsdatei (Header-Datei) mit zusätzliche Direktiven sein. Beispiel: **.INCLUDE "m32def.inc"**
.DEVICE	Bausteintyp	Definiert den Bausteintyp. Wird nicht benötigt, da schon in der Definitionsdatei enthalten.
.DEF	Name = Register	Definiere (*define*) einen Namen für ein Arbeitsregister (Variable!, r0 bis r31). Beispiel: **.DEF Tmp1 = r16**
.EQU	Name = Ausdruck	Definiert eine Konstante mit Namen. Dieser Name ist dann nicht mehr veränderbar. Beispiel: **.EQU Clock = 1000**
.SET	Name = Ausdruck	Definiert eine Konstante mit Namen. Dieser Name ist innerhalb des Programms (Übersetzungsvorgangs) veränderbar. Beispiel: **.SET Value = 500 ;alter Wert** **.SET Value = 200 ;neuer Wert**
.ORG	Adresse	Legt eine Anfangsadresse fest ab der der folgende Code abgespeichert wird. Hiermit kann der Speicher organisiert werden. Beispiel: **.ORG 0xA00**
.EXIT		Ende des Quelltextes

Weiter Direktiven für die Organisation des Speicher-Bereichs:

Direktive	Operand	Beschreibung
.CSEG		Beginn des Codesegmentes. Alles Folgende wird als Code übersetzt und im Flash gespeichert. Mit den **.DB** und **.DW**-Direktiven können Variablen im **FLASH** abgelegt werden.
.DSEG		Beginn des Datensegmentes. Mit der **.BYTE**-Direktive und eventuell symbolischen Namen (Label) wird hier der **SRAM**-Speicher organisiert.
.ESEG		Beginn des EEPROM-Segments. Mit der **.DB** und **.DW**-Direktive werden Variablen im **EEPROM** abgelegt.
.DB	`Liste mit Bytekonstanten`	(engl.: „define Byte“) Fügt konstante Bytes ein. Dabei ist es die Bedeutung der Bytes egal (Zahl von 0..255, ASCII-Zeichen 'b', eine Zeichenkette "Hallo"; alle Bytes werden durch Kommas getrennt). Im Flash muss eine gerade Zahl von Bytes eingefügt werden (16 Bit-Worte), sonst hängt der Assembler ein Nullbyte an. Beispiel: **`.DB 5,0xA0,'H',0b11101110,"T3EC"`**
.DW	`Liste mit Wortkonstanten`	(engl.: „define Word“) Fügt konstantes binäres Wort (16 Bit) ein. Im EEPROM und Flash zuerst das niederwertige Byte, dann das höherwertige Byte.
.BYTE	`Anzahl`	Reserviert Speicherplatz (Bytes) im SRAM (Datensegment). Beispiel: **`.BYTE 5`**

Um die Übersichtlichkeit weiter zu erhöhen sollen hier einige **Abmachungen** getroffen werden, an die man sich für diesen Kurs weit möglichst halten soll, auch wenn Assembler nicht zwischen Klein- und Großschrift unterscheiden.

- Für Labels und Direktiven werden Großbuchstaben verwendet. Für die Labels verwenden wir englische Abkürzungen. Ein Label muss mit einem Buchstaben beginnen.
- Für Befehle werden nur Kleinbuchstaben verwendet.
- Variablen (Register) oder Konstanten werden öfter mit einem aussagekräftigem Namen versehen (mittels **.DEF**, **.EQU** oder **.SET**). Dieser Name soll mit einem Großbuchstaben beginnen. Er kann auch Kleinbuchstaben enthalten. Vorzugsweise sollen englische Bezeichner verwenden.
- Es sollen keine Sonderzeichen verwendet werden. Auch nicht in den Kommentaren, da öfter Probleme mit dem Zeichensatz auftreten, besonders wenn man mit unterschiedlichen Betriebssystemen arbeitet.

Zuerst der **Programmkopf** der Vorlage. Dieser besteht aus Kommentaren und dient der Dokumentation des Programms. Jede Kommentarzeile beginnt mit einem Semikolon (Strichpunkt).

Es ist im ureigenen Interesse des Programmierers Programme ausführlich zu dokumentieren. Der Mensch ist äußerst vergesslich und schon nach wenigen Wochen können Programmteile, welche heute sonnenklar erscheinen schon wieder unverständlich sein. Auch ermöglicht erst eine detaillierte Dokumentation anderen Programmierern den Code eines Programms ohne allzu viel Aufwand verstehen zu können.

```
;*****************************************************************************
;*
;*      Titel:  Programmiervorlage (A2_template.asm)
;*      Datum:  08/01/08        Version:        0.4
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor: ATmega32                     Quarzfrequenz:
;*      Eingaenge:
;*      Ausgaenge:
;*
;*      Informationen zur Funktionsweise:
;*
;*****************************************************************************
```

Als nächstes folgt das Einbinden einer **Definitionsdatei** zum verwendeten Baustein des Herstellers:

```
;-----------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten
```

Die **.NOLIST**-Direktive verhindert, dass die ***.LST**-Datei durch den langen Text der Definitionsdatei unübersichtlich wird. Diese Direktive ist natürlich nicht zwingend notwendig. Die **.LIST**-Direktive schaltet nach der den List-Output nach der **.INCLUDE** -Direktive wieder ein.

Mit **.INCLUDE** kann eine Textdatei eingebunden werden. Befindet sich die Datei nicht im gleichen Verzeichnis wie das Assemblerprogramm, so muss der Pfad mit angegeben werden.

Die hier verwendete Definitionsdatei "**m32def.inc**" wird von ATMEL geliefert (befindet sich im Verzeichnis C:\Program Files\Atmel\AVR Tools\AvrAssembler2\Appnotes) und erleichtert durch viele **.EQU**-Direktiven die Programmierung erheblich, da zum Beispiel für die SF-Register keine hexadezimale Adressen mehr benötigt werden, sondern die im Datenblatt verwendeten Abkürzungen (für den ATmega 8 wird die Datei "**m8def.inc**" benötigt).

Hier ein kleiner Auszug aus der verwendeten Datei:

```
; ***** SPECIFY DEVICE ************************************************
.device ATmega32
...
; ***** I/O REGISTER DEFINITIONS ****************************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ    SREG    = 0x3f
```

```
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
.equ    OCR0    = 0x3c
.equ    GICR    = 0x3b
.equ    GIFR    = 0x3a
.equ    TIMSK   = 0x39
...
```

✎ **A201** Schau die Datei "**m32def.inc**" mit einem Texteditor an.

Weiter mit der Vorlage:

```
;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;-----------------------------------------------------------------------------
;.DSEG                          ;was ab hier folgt kommt in den SRAM-Speicher
;TAB1:  .BYTE  100              ;100 Byte grosse Tabelle im Datensegment
```

Wird der SRAM (Datenspeicher) zum Abspeichern von Variablen oder Tabellen genutzt, so sollte dieser gleich am Anfang des Assemblerprogramms organisiert werden. Dies erhöht die Übersichtlichkeit des Programms (falls diese Zeilen benötigt werden, muss der Strichpunkt natürlich gelöscht werden).

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET1:  rjmp    INIT           ;springe nach INIT (ueberspringe ISR Vektoren)
```

Ab der **.CSEG**-Direktive erfolgt der Programmcode.

Beim Anlegen der Betriebsspannung an den Controller, oder nach Ablauf der "Watchdog"-Zeit wird ein **Reset** ausgelöst. Dabei wird der Programmzähler auf Null gesetzt. Ab dieser Adresse wird mit der Verarbeitung von Programmcode begonnen. Mit "**.ORG 0x0000**" wird die Startadresse festgelegt (Label "**RESET1:**").

Danach werden mehrere Adressen übersprungen, die für Interruptvektoren vorgesehen sind (Kapitel "Interrupt- und Systemsteuerung"). Dies geschieht mit dem Assemblerbefehl **rjmp**.[20]

rjmp k

Unbedingter (ohne Bedingung) relativer Sprung zu einer Adresse (*relative jump*).
(k steht für die Adresskonstante, 1 Wort-Befehl (2 Byte))

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

Relative Sprungbefehle können nur 2 KiWorte vor und rückwärts springen, was meist ausreicht. Sie sind schneller als nicht-relative Sprünge (Bsp.: jmp, 2 Worte, 3 Taktzyklen).
Beeinflusste Flags: keine Taktzyklen: 2

Der relative Sprungbefehl bewirkt, dass erst mit dem Code ab Label "**INIT**:" weitergearbeitet wird. Wird mit Interrupts gearbeitet, so werden in diesem Zwischenbereich die Sprungadressen zu den Interrupt-Behandlungsroutinen organisiert (siehe kommentierte Zeilen).

```
;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
```

20 Erscheint ein neuer Befehl im Text, so wird dieser mit einem Rahmen kurz vorgestellt. Es ist sinnvoll sich die englischen Bezeichnung der Befehle zu merken.

```
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
;.ORG   INT0addr                ;interner Vektor für INT0 (alt.: .ORG 0x0002)
;       rjmp    ISR_I0          ;Springe zur ISR von INT0

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
```

Da der reservierte Platz für Interruptvektoren für jeden Controller verschieden ist, wird hier der Name **INT_VECTORS_SIZE** (Definitionsdatei) benutzt um die reservierten Adressen zu überspringen. Dazu wird vor dem Label **INIT:** eine neue Anfangsadresse festgelegt (mit **.ORG**). Der Label **INIT:** entspricht dann dieser Adresse.

✎ **A202** Finde den Ausdruck **INT_VECTORS_SIZE** in der Definitionsdatei für den ATmega8 und den ATmega32. Wie lauten die beiden hexadezimalen Adressen bei denen das eigentliche Programm beginnen kann? Vergleiche mit der Speicherorganisation im Anhang.

```
;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 15 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    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister
.DEF    W = r24

        ;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
```

Im Initialisierungs- und Definitionsteil werden mehreren Registern Namen zugewiesen um die Übersichtlichkeit in den Programmen zu erhöhen.

Zum Rechnen wird öfter ein Register mit Null als Inhalt benötigt. **r15** wird dazu mit dem Namen **Zero** versehen und dann mit dem „**clr**“-Befehl auf Null gesetzt.

clr Rd

Lösche Arbeitsregister Rd (*clear register*).

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

Das Register wird mit sich selbst **ODER**-verknüpft und dadurch gelöscht (Rd = 0). Anders als beim Befehl **ldi Rd,0** werden hier die Flags beeinflusst!!
Beeinflusste Flags: S(0), V(0), N(0), Z(1)
Taktzyklen: 1

Zwei Arbeitsregister (**r16** und **r17**) werden als Zwischenspeicher (**Tmp1**, **Tmp2**) reserviert, da oft Zwischenspeicher im Programm benötigt werden, um zum Beispiel die SF-Register mit Konstanten zu laden. Da Zwischenspeicher meist unmittelbar (*immediate*) adressiert werden (Bsp.: **ldi Tmp1,0xAA**) verwendet man keines der ersten sechzehn Arbeitsregister, da diese nicht unmittelbar adressierbar sind (siehe später). Ein

Register (**r18**) wird für einen Zähler (**Cnt1**) reserviert. Register **r24** und **r25** können zusammen als universelles Doppelregister verwendet werden. Sie dienen aber auch zur Parameterübergabe bei Unterprogrammen.

Dann erfolgt noch die Initialisierung des Stapels, die immer benötigt wird, sobald man mit Unterprogrammen oder Interrupt-Routinen arbeitet.

Wir verwenden dazu die zwei Assembler-Funktionen (keine Befehle!) **LOW()** und **HIGH()**, welche das niederwertige und das höherwertige Byte der Adresse **RAMEND** liefert. **RAMEND** ist ein Name für die höchste Adresse des SRAM-Speichers. Auf diese wird der Stapelzeiger initialisiert (Kapitel "Stapelspeicher"). Da unterschiedliche Controller auch unterschiedliche Größen von SRAM-Speicherplatz besitzen befindet sich diese Adresse auch wieder in der Definitionsdatei.

ldi Rd,K

Lade unmittelbar eine Konstante K (8 Bit) in ein Arbeitsregister Rd (*load immediate*).

1	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K

Für die unmittelbare Adressierung können nur die **Arbeitsregister 16-31** verwendet werden!! (das d bei Rd steht für "*destination*" (Ziel)).
Beeinflusste Flags: keine **Taktzyklen: 1**

Mit dem Befehl zur unmittelbaren Adressierung "**ldi**" (*load immediate*) wird ein Byte der Adresse in den Zwischenspeicher geladen, und dann mit dem "**out**"-Befehl in den Stapelzeiger (2 SF-Register (**SPL**, **SPH**) im SRAM) geschrieben.

Der Transferbefehl "**ldi Tmp1,HIGH(RAMEND)**" lädt eine Konstante in das Arbeitsregister "**Tmp1**". Die **Quelle** bei Befehlen steht immer **rechts** und das **Ziel links**!

Der Befehl "**out SPH,Tmp1**" lädt den Inhalt des Arbeitsregisters in das SF-Register **SPH**. Dieses Ein/Ausgaberegister befindet sich auf der Adresse **0x005E** im SRAM, wird jedoch mit seiner eigenen Adresse **0x3E** angesprochen. In Assembler werden jedoch die Abkürzungen aus der Definitionsdatei verwendet, was das Programm wesentlich übersichtlicher gestaltet und die Programmierarbeit vereinfacht.

out P,Rr

Kopiere Inhalt (8 Bit) eines Arbeitsregisters Rr in ein SF-Register (*store register to SF-register*).

1	0	1	1	1	P	P	r	r	r	r	r	P	P	P	P

Die 64 SF-Register (manchmal als *Port* oder *I/O Space* bezeichnet) können nicht unmittelbar angesprochen werden. SF-Register können nur über den Umweg eines der 32 Arbeitsregister beschrieben werden (das r bei Rr steht für "*source*" (Quelle)).
Beeinflusste Flags: keine **Taktzyklen: 1**

✎ **A203** Finde die Ausdrücke **RAMEND**, **SPH** und **SPL** in der Definitionsdatei für den ATmega32. Beschreibe mit diesen Werten wie der Stapel initialisiert wird.

Der Schluss unserer Programmvorlage bietet Platz für das Hauptprogramm sowie das Einbinden von Unterprogrammen, Interrupt-Behandlungsroutinen und Tabellen.

```
;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:
        rjmp    MAIN    ;Endlosschleife

        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
;END:   rjmp    END             ;Endlosschleife

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

;-----------------------------------------------------------------------------
;       Tabellen im Programmspeicher (Flash)
;-----------------------------------------------------------------------------
;TAB:  .DB    "Hallo"

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

Die Hauptschleife ist üblicherweise eine Endlosschleife. Sollte dies nicht der Fall sein, so kann man noch eine **END**-Zeile an den Schluss setzen.

Werden einige der auskommentierten Teile der Vorlage benötigt, so wird der Strichpunkt vor dem Code gelöscht.

Nicht im Programm benötigt Teile werden vollständig gelöscht.

Hier nochmal die gesamte Programmvorlage:

```
;*****************************************************************************
;*
;*      Titel:  Programmiervorlage (A2_template.asm)
;*      Datum:  08/01/08        Version: 0.4 (27/12/12)
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor: ATmega32                     Quarzfrequenz:
;*      Eingaenge:
;*      Ausgaenge:
;*
;*      Informationen zur Funktionsweise:
;*
;*****************************************************************************

;-----------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;-----------------------------------------------------------------------------
;.DSEG                          ;was ab hier folgt kommt in den SRAM-Speicher
;TAB1:  .BYTE  100              ;100 Byte grosse Tabelle im Datensegment

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET1:  rjmp    INIT           ;springe nach INIT (ueberspringe ISR Vektoren)
```

```
;-------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
;.ORG   INT0addr                ;interner Vektor für INT0 (alt.: .ORG 0x0002)
;       rjmp    ISR_I0          ;Springe zur ISR von INT0

;-------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 15 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    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister
.DEF    W = r24

        ;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

;-------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------
MAIN:
        rjmp    MAIN    ;Endlosschleife

        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
;END:   rjmp    END             ;Endlosschleife

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

;-------------------------------------------------------------------------
;       Tabellen im Programmspeicher (Flash)
;-------------------------------------------------------------------------
;TAB:  .DB    "Hallo"

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

Bemerkung: Für Einführungskurse oder kleine Programme existiert eine abgespeckte Version der Vorlage mit dem Namen "**A2_template_light.asm**" (http://weigu.lu/a/asm).