B2 Arbeiten mit Tabellen

In diesem Kapitel soll die direkte und die indirekte Adressierung des Datenspeichers bzw. des Programmspeichers wiederholt und vertieft werden.

Viele Programme müssen größerer Datenmengen verarbeiten. Da dann nicht ausreichend Arbeitsregister zur Verfügung um die Daten abzuspeichern, wird der Datenspeicher (SRAM) genutzt. Müssen nur einige wenige Bytes im Datenspeicher abgelegt werden, so nutzt man die **direkte Adressierung** des SRAM.

Soll mit mehr als 5-10 Bytes gearbeitet werden, so werden diese meist in Tabellen abgelegt. Um mit größeren Tabellen zu Arbeiten benötigt man die **indirekte Adressierung**.

Bei der indirekten Adressierung ist die Adresse des Operanden nicht direkt im Befehl enthalten sondern in einem der drei Registerpaare **X**, **Y** oder **Z**, welche als Adresszeiger (Indexregister, Pointer) dienen. Vor jeder indirekten Adressierung muss der Adresszeiger initialisiert werden!

Variablen im Datenspeicher

"Speichern" (*store direct to sram*, **sts**) und "Laden" (*load direct from sram*, **lds**) sind die beiden Befehle die der Verarbeitung von Variablen mittels direkter Adressierung im Datenspeicher dienen.

Am flexibelsten ist es den Speicher mit Hilfe von symbolischen Adressen (Labeln) zu organisieren. Dadurch ist man nicht an absolute Adressen gebunden und der Speicher kann optimal genutzt werden.

Mit der Assembleranweisung **.DSEG** teilt man dazu dem Assembler mit, dass die folgenden Zeilen sich auf den Datenspeicher (SRAM) beziehen. Dann reserviert man mit der Assembleranweisung **.BYTE** eine bestimmte Anzahl von Speicherzellen. Der vorangestellte Label dient als symbolische Adresse. Mit **.CSEG** wird dem Assembler mitgeteilt, dass der Rest im Programmspeicher landen soll.

Beispiel:

```
;---------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;---------------------------------------------------------------------------
.DSEG                           ;was ab hier folgt kommt in den SRAM-Speicher
VAR1:   .BYTE  1                ;1 Byte grosse Variable im Datensegment
VAR2:   .BYTE  4                ;4 Byte grosse Variable im Datensegment

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                            ;was ab hier folgt kommt in den FLASH-Speicher
```

Der Zugriff auf die Variablen erfolgt zum Beispiel mit:

```
        ldi     Tmp1,120
        sts     VAR1,Tmp1
```

```
        lds     r19,VAR2
        lds     r20,VAR2+1
        lds     r21,VAR2+2
        lds     r22,VAR2+3
```

Tabellen im Datenspeicher

Zur Verarbeitung von Tabellen im Datenspeicher dienen ebenfalls ein "Speicher"-Befehl (*store indirect*, **st**) und ein "Laden"-Befehl (*load indirect*, **ld**). Besonders praktisch sind Befehle zur indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers.

Auch hier ist es am flexibelsten den Speicher mit Hilfe von symbolischen Adressen (Labeln) zu organisieren. Dadurch ist man nicht an absolute Adressen gebunden und der Speicher kann optimal genutzt werden.

Im Kapitel zur Adressierung wurde ein Assemblerprogramm erstellt, dessen Aufgabe es war den SRAM-Speicher mit den Dezimalzahlen 0 bis 255 aufzufüllen. Dieses Programm soll jetzt ohne absolute Adresse programmiert werden. Eine mögliche Lösung könnte folgendermaßen aussehen:

```
;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;-----------------------------------------------------------------------------
.DSEG                           ;was ab hier folgt kommt in den SRAM-Speicher
TAB1:   .BYTE  256              ;256 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
RESET:  rjmp    INIT            ;springe nach INIT (ueberspringe ISR Vektoren)

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Cnt1 = r18              ;Register 18 dient als erster Zaehler
        clr     Cnt1            ;Zaehler (Zahl) mit 0 initialisieren
        ldi     XL,LOW(TAB1)    ;Adresszeiger mit Tabellenangfang initialisieren
        ldi     XH,HIGH(TAB1)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   st      X,Cnt1          ;Speichere den Inhalt des Zaehlers in
                                ;den Datenspeicher. Die Adresse befindet sich
                                ;im Doppelregister X
        adiw    XL,1            ;Inkrementiere den 16-Bit-Adresszeiger X
        inc     Cnt1            ;Inkrementiere den Zaehler (Zahl)
        cpi     Cnt1,0          ;Vergleiche mit Null (diese Zeile kann man auch weglassen!)
        brne    MAIN            ;Bleib solange in der Schleife bis 256
                                ;(Ueberlauf, Register wieder auf null)
        ;ab hier wird das Programm fuer die Aufgabe B200 erweitert
        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
END:    rjmp    END             ;Endlosschleife

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

Bemerkungen: Die beiden Programmzeilen **`st X,Cnt1`** und **`adiw XL,1`** können durch **`st X+,Cnt1`** ersetzt werden.

Damit auch wirklich 256 Werte(0-255) geschrieben werden, wird erst abgebrochen wenn das Register überläuft, also wieder Null wird! Die Zeile mit dem Vergleichsbefehl dient der Übersichtlichkeit. Man kann sie weglassen, da das Inkrementieren eine Rechenoperation ist und somit auch das Null-Flag im SREG-Register setzt.

✎ **B200** Das obige Programm soll erweitert werden.

a) Zusätzlich zum Erstellen der Zahlentabelle sollen die ersten 128 Byte der erstellten Tabelle in eine zweite Tabelle "**`Tab2`**" kopiert werden. Die Zahlen in der zweiten Tabelle sollen als **`ASCII`**-Zeichen interpretiert werden (siehe **`ASCII`**-Tabelle im Anhang). Die Kopie soll dann nur noch die Zeichen "**`A`**-**`Z`**" enthalten. Die restlichen Zeichen sollen mit dem Nullbyte (**`0x00`**) aufgefüllt werden.
Gib dem Programm den Namen "**`B200_sram_indirect_2.asm`**" und dokumentiere das Programm mit Hilfe eines Flussdiagramms.

b) Teste das Programm im Studio 4 mit dem Debugger. Lass das Programm durchlaufen (Autostep ("Alt+F5")) und beobachte das Resultat im Speicherfenster ("View/Memory" oder "Alt+4").

c) Für Fleißige: Ändere das Programm so um, dass zusätzlich "**`0`**-**`9`**", "**`a`**-**`z`**" gültig sind. Nenne das Programm "**`B200_sram_indirect_3.asm`**" und teste es ebenfalls.

Tipps zur Programmieraufgabe:

Beim Kopieren von Tabellen wird mit zwei verschiedenen Adresszeigern (z.B. **`X`** und **`Y`**) gearbeitet. Zum Filtern von **`ASCII`**-Zeichen kann man die Befehle "**`brsh`**" (Verzweige falls gleich oder größer) und "**`brlo`**" (Verzweige falls kleiner) verwenden.

`brsh k`

Bedingter relativer Sprung falls (vorzeichenlos) größer oder gleich (*branch if same or higher*).

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

Der Befehl wird meist gleich nach einem Vergleichsbefehl (**`cp`**, **`cpi`**) eingesetzt. **Der Sprung erfolgt falls kein Übertrag (Carry) aufgetreten ist** (**`C`**-Flag = 0). Beim Vergleich vorzeichenloser Zahlen erfolgt der Sprung wenn der Zieloperand (Rd) ⩾ dem Quelloperand (Rr, K). Der Befehl entspricht dem Befehl brcc.
Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

brlo k

Bedingter relativer Sprung falls (vorzeichenlos) kleiner (*branch if lower*). **(k = Adresskonstante)**

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

Der Befehl wird meist gleich nach einem Vergleichsbefehl (**cp**, **cpi**) eingesetzt. **Der Sprung erfolgt falls ein Übertrag (Carry) aufgetreten ist** (C-Flag = 1). Beim Vergleich vorzeichenloser Zahlen erfolgt der Sprung wenn der Zieloperand (Rd) < dem Quelloperand (Rr, K). Der Befehl entspricht dem Befehl brcs.

Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

Tabellen im Programmspeicher

Öfter benötigt man kleine Tabellen mit festen Werten. Da die Programmierung des EEPROM recht aufwändig ist, bietet es sich an diese Tabellen im Programmspeicher mit abzulegen.

"Lade Programmspeicher" (*load program memory*, **lpm**) ermöglicht es ein beliebiges Datenbyte aus dem Programmspeicher (Flash) in ein Arbeitsregister zu laden. Die indirekte Adresse muss sich dazu im **Z**-Pointer (Adresszeiger) befinden (siehe Kapitel "Befehle und Adressierung).

Mit der Direktive "**.ORG**" besteht die Möglichkeit den Beginn der Tabelle festzulegen. Ohne diese Anweisung wird die Tabelle gleich hinter dem Programm abgespeichert[4]. Am sichersten ist es die "**.ORG**" Direktive nicht einzusetzen sondern nur mit Labeln (symbolischen Adressen) zu arbeiten. Der Programmieraufwand ist dadurch eventuell größer, jedoch kann dann der Speicher optimal genutzt werden und man läuft nicht Gefahr den Speicher bei großen Projekten mit eingebundenen Bibliotheken mehrfach zu belegen.

Die Direktive "**.DB**" (define Byte) ermöglicht es die Tabelle (Liste mit Bytekonstanten) zu definieren. Da der Programmspeicher in Worten (2 Byte) organisiert ist, ist es nötig die Wortadresse mit dem Faktor zwei zu multiplizieren, da der Adresszeiger nur byteweise adressieren kann. Dies kann einfach im Assembler mit Multiplikationszeichen geschehen.

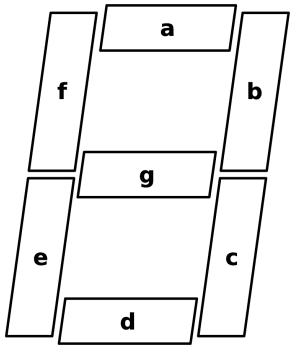
4 Eine Tabelle sollte sich immer zum Schluss der Assemblerdatei befinden, da sonst Teile des Programms oder eines Unterprogramms hinter der Tabelle landen und nicht mehr richtig angesprochen werden!

Ansteuerung eines Sieben-Segment-Displays

Ansteuerung einer Displaystelle

✎ **B201** Das Beispielprogramm "**A408_flash_indirect_textstring.asm**" kann als Vorlage dienen (Initialisierung des Stapels nicht vergessen, keine **.ORG** Anweisung vor dem Tabellenlabel).

a) Das untere Nibble (4 Bit) eines Registers (**r18**) soll in hexadezimaler Schreibweise auf einer Siebensegment_Anzeige ausgegeben werden. Die sieben Segmente (**a-f**) werden durch die unteren 7 Bit des Port D angesteuert. Das hochwertigste Bit aktiviert die Anzeige. Erstelle die Dekodiertabelle!



Bit:	7	6	5	4	3	2	1	0	
Seg:		g	f	e	d	c	b	a	Byte:
0	1	0	1	1	1	1	1	1	0xBF
1	1								
2	1								
3	1								
4	1								
5	1								
6	1								
7	1								
8	1								
9	1								
A	1								
b	1								
C	1								
d	1								
E	1								
F	1								

b) Die Dekodiertabelle soll sich hinter dem Programmcode befinden. Die Addition des aus-maskierten unteren Nibbles des Registers **r24** zur Anfangsadresse der Tabelle dient als Adresszeiger!
Achtung! Es muss eine 16-Bit-Addition durchgeführt werden!
Zeichne das Flussdiagramm. Schreibe den Assemblercode und speichere ihn als "**B201_flash_indirect_numdisplay_1.asm**".
Lade im Initialisierungsteil das Register **r24** mit unterschiedlichen Werten und teste so dein Programm.

c) Erweitere das Programm (und Flussdiagramm) so, dass im Sekunden rhythmus abwechselnd das untere und das obere Nibble des Registers auf der Siebensegmentanzeige dargestellt werden. Speichere das erweiterte Programm als "**B201_flash_indirect_numdisplay_2.asm**".

Tipps zur Programmieraufgabe:

Für eine 16-Bit Addition benötigt man die Befehle "**add**" und "**adc**". Im ersten Schritt werden mit "**add**" die beiden **niederwertigen Register** addiert. Der eventuell auftretende Übertrag (*carry*) wird mit dem "**adc**"-Befehl berücksichtigt. Mit diesem werden die **hochwertigen Register** dann addiert.
Wird ein 8-Bit-Register zu einem 16-Bit-Register addiert (wie in der obigen Aufgabe), so wird bei der Addition der hochwertigen Register Null addiert. Da der "**adc**" Befehl nicht als unmittelbarer Befehl zur Verfügung steht, wird ein gelöschtes Hilfsregister benötigt.

```
        clr     Tmp1            ;Hilfsregister = 0
        add     ZL, Counter     ;Addition der niederwertigen Register
        adc     ZH,Tmp1         ;eventuelles Carry berücksichtigen
```

adc Rd,Rr

Addition des Arbeitsregisters Rd mit dem Arbeitsregister Rr mit Berücksichtigung des Übertrags (*add with carry*).

0	0	0	1	1	1	r	d	d	d	d	d	r	r	r	r

Resultat in Rd. Carry-Flag wird zur Summe hinzuaddiert.
Beeinflusste Flags: H, S, V, N, Z, C Taktzyklen: 1

Mit dem "**swap**"-Befehl können die beiden Nibble (4 Bit-Gruppe, eine hexadezimale Stelle) eines Byte einfach vertauscht werden (Unterpunkt c))!

swap Rd

Niederwertiges und hochwertiges Nibble (4 Bit) eines Registers vertauschen (*swap nibbles*).

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

Beeinflusste Flags: keine Taktzyklen: 1

Ansteuerung von vier Displaystellen

Es besteht eine Unterprogramm-Bibliothek mit dem Namen "**SR_NUMDISPLAY.asm**". (http://www.weigu.lu/a/asm).

Sie erlaubt die Darstellung des Registerinhalts des Doppelregisters **W** während einer bestimmten Zeit auf einem 4-stelligen 7-Segment-Display. Dazu müssen nur im Hauptprogramm die beiden Unterprogramme **D7SINI** und **D7SHEX** aufgerufen werden. Das erste Unterprogramm initialisiert die Portpins (können im Zuweisungsteil der Bibliothek geändert werden), das zweite stellt den Inhalt des Doppelregisters **W** dar.

```
;*****************************************************************************
;*
;*      Titel:  Unterprogramme zur Darstellung von Daten auf einer 4-stelligen
;*              7-Segment-Anzeige (SR_NUMDISPLAY.asm)
;*
;*      Datum:  22/11/11        Version:        0.4 (02/01/13)
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32A        Quarzfrequenz:  16MHz
;*      Eingaenge:
;*      Ausgaenge:      Pin 0-7 (a-g, DP) am SegPort
;*                      4 Pin am DigPort
;*
;*      Informationen zur Funktionsweise:
;*
;*      Eine Zeitschleifenbibliothek (UP W1ms) muss im Hauptprogramm eingebunden
;*      sein. Bei den Zuweisungen wird festgelegt, welche Ports und Pins
;*      verwendet werden sollen (1 ganzes Port für die Segmente, 4 Bit eines
;*      Ports fuer die Stellen).
;*      Das Unterprogramm D7SINI muss im Hauptprogramm in der Initialisierungs-
;*      phase aufgerufen werden (Initialisiert Portpins als Ausgang).
;*
;*      Beim Aufruf von D7SHEX wird der Inhalt des Doppelregisters W (r25:r24)
;*      waehrend einer definierten Zeit in Millisekunden in Hexadezimal am
;*      Display ausgegeben.
;*
;*      Unterprogramme:
;*
;*      D7SINI          Muss im Hauptprogramm in der Initialisierungsphase
;*                      aufgerufen werden (Initialisiert Portpins als Ausgang)
;*      D7SHEX          Der Inhalt des Doppelregisters W (r25:r24) wird waehrend
;*                      einer definierten Zeit in Millisekunden in Hex am
;*                      Display ausgegeben. Die Dekodiertabelle befindet sich im
;*                      Programmspeicher. Die Hexstelle addiert zur Anfangs-
;*                      adresse dient als Adresszeiger.
;*
;*
;*      Copyright (c) 2011   Guy WEILER         weigu[at]weigu[dot]lu
;*
```

```
;*      Hiermit wird unentgeltlich, jeder Person, die eine Kopie der Software
;*      und der zugehoerigen Dokumentationen (die "Software") erhaelt, die
;*      Erlaubnis erteilt, uneingeschraenkt zu benutzen, inklusive und ohne
;*      Ausnahme, dem Recht, sie zu verwenden, kopieren, aendern, fusionieren,
;*      verlegen, verbreiten, unterlizenzieren und/oder zu verkaufen, und
;*      Personen, die diese Software erhalten, diese Rechte zu geben, unter den
;*      folgenden Bedingungen:
;*
;*      ;Der obige Urheberrechtsvermerk und dieser Erlaubnisvermerk sind in alle
;*      ;Kopien oder Teilkopien der Software beizulegen.
;*      ;
;*      ;DIE SOFTWARE WIRD OHNE JEDE AUSDRUECKLICHE ODER IMPLIZIERTE GARANTIE
;*      ;BEREITGESTELLT, EINSCHLIESSLICH DER GARANTIE ZUR BENUTZUNG FUER DEN
;*      ;VORGESEHENEN ODER EINEM BESTIMMTEN ZWECK SOWIE JEGLICHER RECHTS-
;*      ;VERLETZUNG, JEDOCH NICHT DARAUF BESCHRAENKT. IN KEINEM FALL SIND DIE
;*      ;AUTOREN ODER COPYRIGHTINHABER FUER JEGLICHEN SCHADEN ODER SONSTIGE
;*      ;ANSPRUECHE HAFTBAR ZU MACHEN, OB INFOLGE DER ERFUELLUNG EINES
;*      ;VERTRAGES, EINES DELIKTES ODER ANDERS IM ZUSAMMENHANG MIT DER SOFTWARE
;*      ;ODER SONSTIGER VERWENDUNG DER SOFTWARE ENTSTANDEN.
;*
;*****************************************************************************

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Zuweisungen
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EQU    SegDDR  = DDRC          ;Segment-Port definieren (8 Bit)
.EQU    SegPort = PORTC
.EQU    DigDDR  = DDRB          ;Digit-Port (Stellen) definieren
.EQU    DigPort = PORTB
.EQU    D1PinNr = 0
.EQU    D2PinNr = 1
.EQU    D3PinNr = 2
.EQU    D4PinNr = 3

.EQU    DispTime = 1000         ;Zeit der Darstellung in Millisekunden
.                               ;(min. 4ms; max. 262140 ms)

.EQU    D1_Dot = 0              ;hier kann der Dezimalpunkt (Bit 7) fuer jede
.EQU    D2_Dot = 0              ;Stelle freigeschaltet werden (1 = on, 0 = off)
.EQU    D3_Dot = 1
.EQU    D4_Dot = 0

;-----------------------------------------------------------------------------
;       7 Segmentanzeige initialisieren
;-----------------------------------------------------------------------------
D7SINI: push    r16
        ser     r16
        out     SegDDR,r16
        sbi     DigDDR,D1PinNr
        sbi     DigDDR,D2PinNr
        sbi     DigDDR,D3PinNr
        sbi     DigDDR,D4PinNr
        pop     r16
        ret

;-----------------------------------------------------------------------------
;       Unterprogramm zur Darstellung des Doppelreg. W auf dem 7-Seg.-Display
;-----------------------------------------------------------------------------
D7SHEX: push    r16             ;alle verwendeten Register (inkl. SREG) retten
        in      r16,SREG
        push    r16
        push    r17
        push    r18
        push    r19
        push    r24
        push    r25
        push    ZL
        push    ZH

        ;16 Bit Zeitzaehler r19:r18 initialisieren
        ldi     r18,LOW(DispTime/4)     ;Minimale Zeit = 4*1ms=4ms)
```

```
        ldi     r19,HIGH(DispTime/4)

D7SHE1: ;erste Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r24           ;Adresszeiger errechnen
        andi    r16,0x0F          ;Maskieren des untersten Nibble von r24
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
                                  ;Adresse im Z-Pointer steht in das Arbeits-
                                  ;register r16.
        .IF     D1_Dot == 1       ;Setze Dezimalpunkt Stelle 1 falls D1_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D1PinNr ;Erste Stelle einschalten
        out     SegPort,r16     ;Zeichen am SegPort ausgeben
        rcall   W1ms            ;Warte 1ms
        cbi     DigPort,D1PinNr ;Erste Stelle ausschalten

        ;zweite Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r24           ;Adresszeiger errechnen
        andi    r16,0xF0          ;Maskieren des obersten Nibble von r24
        swap    r16               ;Vertausche beide Nibble
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
                                  ;Adresse im Z-Pointer steht in das Arbeits-
                                  ;register r16.
        .IF     D2_Dot == 1       ;Setze Dezimalpunkt Stelle 2 falls D2_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D2PinNr ;Zweite Stelle einschalten
        out     SegPort,r16     ;Zeichen am SegPort ausgeben
        rcall   W1ms            ;Warte 1ms
        cbi     DigPort,D2PinNr ;Zweite Stelle ausschalten


        ;dritte Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r25           ;Adresszeiger errechnen
        andi    r16,0x0F          ;Maskieren des untersten Nibble von r25
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
                                  ;Adresse im Z-Pointer steht in das Arbeits-
                                  ;register r16.
        .IF     D3_Dot == 1       ;Setze Dezimalpunkt Stelle 3 falls D3_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D3PinNr ;Dritte Stelle einschalten
        out     SegPort,r16     ;Zeichen am SegPort ausgeben
        rcall   W1ms            ;Warte 1ms
        cbi     DigPort,D3PinNr ;Dritte Stelle ausschalten

        ;vierte Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r25           ;Adresszeiger errechnen
        andi    r16,0xF0          ;Maskieren des obersten Nibble von r25
        swap    r16               ;Vertausche beide Nibble
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
```

```
                                ;Adresse im Z-Pointer steht in das Arbeits-
                                ;register r16.
        .IF     D4_Dot == 1     ;Setze Dezimalpunkt Stelle 4 falls D4_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D4PinNr ;Vierte Stelle einschalten
        out     SegPort,r16     ;Zeichen am SegPort ausgeben
        rcall   W1ms            ;Warte 1ms
        cbi     DigPort,D4PinNr ;Vierte Stelle ausschalten

        ;Zeitzaehler ueberpruefen
        subi    r18,1           ;Zaehler dekrementieren und Zeit ueberpruefen
        sbci    r19,0
        brne    D7SHE1

        pop     ZH
        pop     ZL
        pop     r25
        pop     r24
        pop     r19
        pop     r18
        pop     r17
        pop     r16
        out     SREG,r16
        pop     r16
        ret                     ;Zurueck ins Hauptprogramm

;-----------------------------------------------------------------------------
;       Tabelle         (7-Segment-Dekodiertabelle (hoechstwertigstes Bit = 0))
;-----------------------------------------------------------------------------
NDISP2:
.DB     0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07     ;Dekodiertabelle fuer
.DB     0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71     ;alle Hex-Zahlen
```

✎ **B202** a) Analysiere die Bibliothek, und erstelle ein Flussdiagramm für das Unterprogramm **D7SHEX**.
b) Schreibe ein Assemblerprogramm, das im Sekundentakt alle möglichen Zustände des **W**-Registers durchläuft und teste es auf mit einem vierstelligen LED-Display (z.B. MICES2-Board, siehe Anhang). Das Programm soll als "**B202_numdisplay_4.asm**" abgespeichert werden.
c) Wie lange benötigt das Programm um alle Zustände zu durchlaufen?

Lauflicht aus einer Tabelle:

✎ **B203** Ein einfaches Lauflicht über 16 LEDs (2 Ports) soll mit Hilfe einer Flash-Tabelle realisiert werden.
Schreibe das Assemblerprogramm und nenne es "**B203_christmas_2.asm**".

Blinkmuster aus einer Tabelle:

Zur Festigung des Gelernten kann zusätzlich folgende Aufgabe gelöst werden:

✎ **B204** Ein Programm soll 32 verschiedene Blinkmuster ausgeben können. Die Blinkmuster befinden sich in einer Tabelle im Programmspeicher. Am Anfang des Programms soll die ganze Tabelle ins SRAM kopiert werden.
Über 5 Schalter soll das Bitmuster aus der SRAM-Tabelle geladen werden. Die Schalterstellung + Basisadresse soll hierbei gleich die Adresse des Bitmusters ergeben.
Ein Unterprogramm holt das Bitmuster aus der Tabelle ab, invertiert es, speichert es in die Tabelle zurück und gibt das Bitmuster an den LEDs aus.
Die Blinkzeit kann über weitere drei Schalter als Vielfaches von 100ms eingestellt werden.
a) Wieso ist dieses Umkopieren notwendig?
b) Erstelle das Flussdiagramm und die Bitmustertabelle!
c) Schreibe das Assemblerprogramm und speichere es als "**B204_christmas_3.asm**".

Beispiel für eine Bitmustertabelle mit 16 Blinkmustern:

Adresse	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Wert
Label+0x00	★		★		★		★		0xAA
Label+0x01	★	★			★	★			0xCC
Label+0x02	★	★	★	★					0xF0
Label+0x03	★			★	★			★	0x99
Label+0x04	★							★	0x81
Label+0x05	★	★					★	★	0xC3
Label+0x06	★	★	★			★	★	★	0xE7
Label+0x07	★	★	★	★	★	★	★	★	0xFF
Label+0x08				★				★	0x11
Label+0x09			★				★		0x22
Label+0x0A		★				★			0x44
Label+0x0B	★				★				0x88
Label+0x0C		★	★	★		★	★	★	0x77
Label+0x0D	★	★	★		★	★	★		0xEE
Label+0x0E		★	★			★	★		0x66
Label+0x0F	★		★	★	★	★		★	0xBD

Kleine Wiederholung zur Adressierung:

Programmspeicher (FLASH, ROM)

Tabellen zum Abspeichern von **Konstanten** (nur lesbar!). **Nur indirekte Adressierung und nur über den Adresszeiger Z!** Die Initialisierung der Tabelle erfolgt **am Ende des Programms** im Codesegment (Assembleranweisung **.CSEG**). Die Tabelle besteht aus dem Label (Word-Adresse), der Assembleranweisung **.DB** (*define Byte*)[5] und den Konstanten in beliebiger Schreibweise. Bei der Initialisierung des Adresszeigers **Z** muss mit zwei multipliziert werden, da der Programmspeicher 16 Bit (Word) breit ist!

- Initialisierungen (Beispiele):

```
        ldi     ZL,LOW(FTAB*2)
        ldi     ZH,HIGH(FTAB*2)

am Ende des Programms:
FTAB:   .DB     3,144, 0x32, 0b11101110, "Hall", 'o', 0
```

- Befehl zum Lesen eines Byte: **lpm** (*load programm memory*). Bsp.:

```
        lpm     Tmp1,Z+
```

Arbeitsspeicher (SRAM, RAM)

Arbeitsregister r0 bis r31

Alle Berechnungen und Datenmanipulationen werden über die Arbeitsregister ausgeführt. Auf sie entfallen die meisten Befehle. Die meisten Befehle die den Buchstaben **i** (*immediate*) enthalten verwenden die unmittelbare Adressierung (direktes Laden von Konstanten) und können nur auf die Register **r16**-**r31** angewendet werden. **r24**-**r31** können als Doppelregister (16 Bit, **W**, **X**, **Y** und **Z**) verwendet werden.

- Einige Befehle die auf Arbeitsregister angewendet werden:

```
r0-r31:  mov,mul,add, cp, swap, inc, sbrs, …
r16-r31: ldi, cpi, adiw, ori, andi, ….
```

5 Es besteht auch die Möglichkeit **.DW** (*define Word*) zu benutzen.

64 SF-Register

Um die 64 SF-Register anzusprechen existieren die beiden Befehle **in** und **out**. Die untersten 32 SF-Register können zusätzlich über **sbi**, **cbi**, **sbis** und **sbic** angesprochen werden!

- **Alle** Befehle die auf SF-Register angewendet werden:

```
alle 64 SF-Register:        in, out
unterste 32 SF-Register:    sbi, cbi, sbis, sbic
```

Datenbereich

Direkte Adressierung

Die direkte Adressierung wird verwendet um Variablen oder sehr kurze Tabellen im SRAM zu schreiben oder zu lesen. Es wird eine **feste Adresse, meist in Form eines Labels** übergeben. Die Reservierung der Variablen bzw. der Tabelle erfolgt ganz am Anfang des Programms im Datensegment (Assembleranweisung **.DSEG)**. Dies geschieht mit einem Label (Word-Adresse), der Assembleranweisung **.BYTE** gefolgt von der Anzahl der zu reservierenden Bytes.

- Reservierung des Speicherplatzes im Datensegment (Beispiele):

```
.DSEG                       ;Anfang des Programms vor dem Codesegment
VAR1:   .BYTE 1
VAR32B: .BYTE 4
```

- **lds** (*load direct from sram)* ist der Befehl zum Lesen eines Byte
  sts (*store direct to sram)* ist der Befehl zum Schreiben eines Byte. Bsp.:

```
        lds     Tmp1,VAR1
        sts     VAR32B+2,Tmp1   ;beschreibt drittes Byte der Variablen
```

Indirekte Adressierung

Zum Bearbeiten größerer Tabellen wird die indirekte Adressierung verwendet. Es können die Adresszeiger **X**, **Y** oder **Z** verwendet werden. Die Reservierung erfolgt wie bei der direkten Adressierung im Datensegment.

- Reservierung und Initialisierungen (Beispiele):

```
.DSEG                       ;Anfang des Programms vor dem Codesegment
STAB:   .BYTE 128

        ldi     XL,LOW(STAB)
        ldi     XH,HIGH(STAB)
```

- **ld** (*load indirect*) ist der Befehl zum Lesen eines Byte
  st (*store indirect*) ist der Befehl zum Schreiben eines Byte. Bsp.:

```
        ld      Tmp1,X
        st      Y+,Tmp1    ;+ für Autoinkrement des Adresszeigers
```

Stapelbereich

Der Stapelbereich funktioniert ebenfalls mit indirekter Adressierung. Der Adresszeiger heißt Stapelzeiger **SP** und besteht aus den beiden SF-Registern **SPL** und **SPH**. Der Stapelzeiger wird von der Hardware automatisch verwaltet. Er muss nur am Anfang des Programms initialisiert werden. Die Befehle für Unterprogramme und Interrupt-Routinen (**rcall**, **call**, **ret**, **reti**, ...) verwenden automatisch den Stapel.

- Initialisierung:

```
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1
```

- **push** ist der Befehl zum Schreiben eines Byte.
  pop ist der Befehl zum Lesen eines Byte. Bsp.:

```
        push    r16
        pop     r24
```