

U2 2. Übung

- Datentypen mit fester Größe
- (Statische) Programmbibliotheken
- SPiCboard-Bibliothek
- Wiederholung: **volatile**
- Wiederholung: Nebenläufigkeit / Synchronisation
- Aufgabe 1

U2-1 Datentypen fester Größe

U2-1 Datentypen fester Größe

- Die Größe der primitiven Datentypen in C ist architekturabhängig
- Beispiel für drei Architekturen (Angaben in Bits)

	8-bit AVR	Intel x86-32	Intel x86-64
char	8	8	8
short	16	16	16
int	16	32	32
long	32	32	64

- Probleme
 - ◆ Portabilität des Quellcodes eingeschränkt
 - ◆ Insbesondere in der hardwarenahen Programmierung braucht man oft Datentypen einer bekannten, festen Größe (I/O-Register)

U2-1 Datentypen fester Größe

- C99 definiert neue Datentypen mit definierter Größe
- Auszug der wichtigsten Typen:

<code>int8_t</code>	8-bit signed
<code>uint8_t</code>	8-bit unsigned
<code>int16_t</code>	16-bit signed
<code>uint16_t</code>	16-bit unsigned
<code>int32_t</code>	32-bit signed
<code>uint32_t</code>	32-bit unsigned

- verfügbar durch Einbinden von **stdint.h**
 - ☞ wird auch mit vielen nicht C99-konformen Compilern geliefert
 - ☞ kann ansonsten auch relativ einfach selbst erstellt werden
- die Größe von Zeigertypen ist immer architekturabhängig (Adressbusbreite)

U2-2 Statische Bibliotheken

- Zugriff auf Funktionen und Variablen über symbolische Namen
- Eine Übersetzungseinheit (C-Datei) **definiert** und **verwendet** Symbole
 - ◆ Hauptprogramm definiert das Symbol *main* für die main-Funktion
 - ◆ Programm ruft (verwendet) eine Funktion der C-Bibliothek (z.B. malloc)
 - ◆ C-Datei definiert eine globale (nicht-static) Variable
 - ◆ Programm verwendet eine globale Variable, die u.U. in einer anderen Übersetzungseinheit definiert wurden
- Der Namensraum ist flach
 - ◆ jeder Name muss eindeutig sein
 - ◆ es darf auch keine Funktion mit dem Namen einer globalen Variable geben
- Kompilierte Übersetzungseinheit (Objekt-Datei) enthält Symboltabelle
 - ◆ Liste mit Symbolen, die von der Einheit verwendet werden
 - ◆ Liste von Symbolen, die von der Einheit definiert werden

U2-2 Auflösung von Symbolreferenzen

- Statisches Binden: Binden der Übersetzungseinheiten zum Binärabbild
- Offene Symbolreferenzen werden aufgelöst
 - ◆ definiert in anderen Übersetzungseinheiten
 - ◆ Suche in Programmbibliotheken
- GCC sucht beim Binden implizit in der Standard C-Bibliothek (**libc.a**)
- weitere Bibliotheken können vom Entwickler angegeben werden

U2-2 Statische Bibliotheken

- **Archiv** (Dateiendung **.a**) von Objekt-Dateien (**.o**-Dateien)
 - ☞ Erstellen einer Bibliothek mit dem Kommando **ar**

```
avr-ar rcs libexample.a bar.o foo.o
```
- Jede Objekt-Datei enthält wiederum eine Symboltabelle
 - ☞ Anzeige mit dem Kommando **avr-nm libexample.a** bzw. **avr-nm bar.o**
- Die Bibliothek kann dem Linker als Symbolquelle angeboten werden
 - ◆ Parameter **-Lpfad**: Suche nach Bibliotheken (.a-Dateien) in *pfad*
 - ◆ Parameter **-llibname**: Binden mit der Bibliothek *libname*
 - ☞ diese wird in einer Datei *liblibname.a* in den Suchpfaden gesucht
- Der Linker bindet dann alle Objekt-Dateien, die **bis dahin** unaufgelöste Symbole definieren, zum Binärabbild dazu
- Die Reihenfolge von Objekt-Dateien und Bibliotheken ist wichtig

U2-2 Beispiel: Kompilieren mit Bibliotheken

```
#include <bar.h>
```

```
void main ( void ) {
    bar ( 42 );
}
```

main.c

```
#ifndef BAR_H
```

```
#define BAR_H
```

```
void bar ( int );
```

```
#endif
```

bar.h (Schnittstelle)

```
#include <bar.h>
```

```
void bar ( int param ) {
    /* do some magic */
}
```

bar.c (Implementierung)

- Module exportieren eine Schnittstelle (Header-Datei)
 - ◆ Funktionsdeklarationen und ggf. Deklarationen (*extern*) globaler Variablen
- Beim Übersetzen muss der Compiler den Typen eines Symbols kennen
 - ◆ Einbinden der Schnittstellenbeschreibung mit `#include <bar.h>`
 - ◆ Die Adresse der Funktion bzw. Variable ist an dieser Stelle nicht notwendig
- Parameter `-Ipfad`: Teilt Compiler zusätzlichen Suchpfad für Headerdateien mit

```
avr-gcc -c <...> -I/myincludes main.c
```

U2-2 Beispiel: Binden mit statischen Bibliotheken

```
#include <bar.h>
```

```
void main ( void ) {
    bar ( 42 );
}
```

main.c

```
#ifndef BAR_H
```

```
#define BAR_H
```

```
void bar ( int );
```

```
#endif
```

bar.h (Schnittstelle)

```
#include <bar.h>
```

```
void bar ( int param ) {
    /* do some magic */
}
```

bar.c (Implementierung)

- Modul *bar* **definiert** Symbol *bar* (Funktion `void bar(int);`)
- Hauptprogramm ruft die Funktion *bar* auf (**verwendet** Symbol *bar*)
- Annahme: Modul *bar.o* ist Teil der Bibliothek *libexample.a* in */libbase*
- Übersetzung mit Kommando:

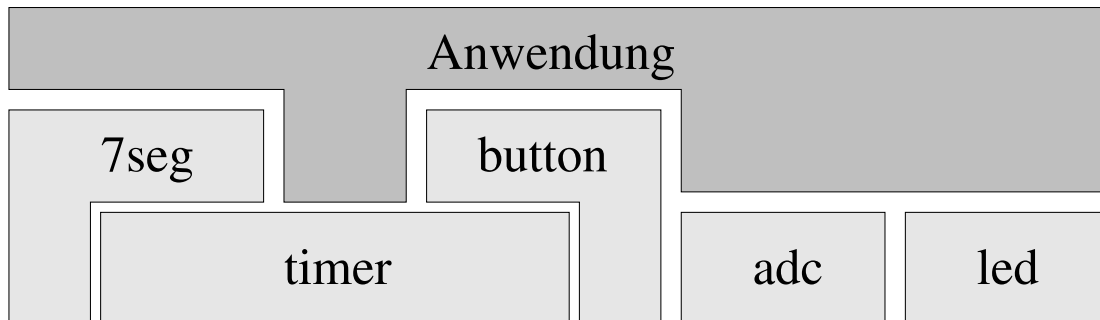
```
avr-gcc -L/libbase -o main.elf <...> main.o -lexample
```

- Falsch (warum?):

```
avr-gcc -L/libbase -o main.elf <...> -lexample main.o
```

U2-3 SPiCboard-Bibliothek

- modular
- einfacher Zugriff auf die Hardware ohne diese selbst zu programmieren
- die einzelnen Module eignen sich selbst gut als Übungsaufgabe - Schnittstellenbeschreibung im Doku-Bereich auf der Webseite (timer und button sind jedoch relativ schwierig)
- Header und Bibliothek *libspicboard.a* in */proj/i4spic/pub/i4* installiert



U2-3 Timer-Modul

- Erlaubt Registrierung sog. *Alarms* und ms-genaues Warten
- Ein Alarm ist beschrieben durch drei Parameter
 - ◆ Aktion (*Callback-Funktion*), die ausgeführt wird wenn der Alarm auslöst
 - ◆ Alarmzeit: Zeitspanne relativ zum aktuellen Zeitpunkt in ms, nach deren Verstreichen der Alarm auslösen soll
 - ◆ Zyklus: Zeitspanne in ms, in der der Alarm nach der Erstauslösung periodisch immer wieder auslöst. Ist die Zeitspanne 0, so löst der Alarm nur ein einziges Mal aus (nicht-zyklisch).

1 Delay-Funktion zur Realisierung von Wartezeiten

```
int8_t sb_timer_delay(uint16_t waittime);
```

- Wartet `waittime` Millisekunden
- Rückgabe: Im Erfolgsfall 0, sonst -1 (Wartezeit u.U. nicht verstrichen!)

2 Aktivieren eines Alarms

- Angabe einer Rückruf (Callback)-Funktion
 - ◆ diese wird, ähnlich einem Interrupthandler (ISR), bei Ablauf des Alarms asynchron ausgeführt
 - ◆ es gelten die selben Randbedingungen wie bei ISRs
- Eine Alarmcallback-Funktion hat den Typ


```
void callbackFunktion(void);
```
- Registrierung eines neuen Alarms mit


```
ALARM *sb_timer_setAlarm(alarmcallback_t cb,
                          uint16_t alarmtime, uint16_t cycletime);
```

 - ◆ die Funktion liefert einen Zeiger auf einen abstrakten ALARM-Datentyp
 - ◆ im Fehlerfall wird **NULL** zurückgeliefert, der ALARM wurde nicht aktiviert
 - ◆ diese Referenz kann nachher zum Abbruch des Alarms verwendet werden

3 Abbruch einer aktiven Verzögerung

- Funktion


```
void sb_timer_delay_abort();
```
- Kann nur auf ISR-Level ausgeführt werden (Programm gerade im Delay)
- Bricht eine aktive Verzögerung ab

4 Abbruch eines laufenden Alarms

- Funktion


```
uint8_t sb_timer_cancelAlarm(ALARM *alarm);
```
- Parameter ist die Referenz auf einen aktiven Alarm
 - ☞ diese Referenz hat man von `sb_timer_setAlarm` erhalten
- Nach dem Abbruch verliert die Referenz ihre Gültigkeit
- Die Funktion liefert 0 im Erfolgsfall, -1 im Fehlerfall (ungültige Referenz)

U2-3 7seg-Modul (7-Segment-Anzeigen)

- Ansteuerung der beiden 7-Segment-Anzeigen
- Verwendet Timer-Modul zur schnellen Umschaltung
☞ quasi-parallele Nutzung der beiden Anzeigen möglich
- Anzeige von Dezimalzahlen im Bereich [-9; 99]
`int8_t sb_7seg_showNumber(int8_t number);`
- Deaktivierung der 7-Segment-Anzeigen
`void sb_7seg_disable(void);`
- weitere Anzeigemöglichkeiten ☞ API-Dokumentation

U2-4 Button-Modul (Taster)

- Das Board verfügt über zwei Taster (Ort siehe Board-Übersicht)
C-Datentyp: `BUTTONEVENT`
 - ◆ `BUTTON0`: hardwareseitig entprellt
 - ◆ `BUTTON1`: softwareseitig durch SPiCboard-Bibliothek entprellt
 - ◆ für die Anwendung sind beide Taster gleichermaßen verwendbar
- Anwendung kann sich für Tasterereignisse an einem Taster registrieren
C-Datentyp: `BUTTONEVENT`
 - ◆ `BTNPRESSED`: Taster wurde gedrückt
 - ◆ `BTNRELEASED`: Taster wurde losgelassen
 - ◆ bitweise Veroderung ermöglicht Registrierung für beide Ereignisse

1 Rückruffunktion für Tasterereignisse

```
void buttoncallback(BUTTON btn, BUTTONEVENT eve);
```

- Parameter `btn`: ID des Buttons, der gedrückt wurde
- Parameter `eve`: Typ des eingetretenen Ereignisses
- Die gleiche Funktion kann bei verschiedenen Tastern für verschiedene Ereignisse registriert werden, pro Taster/Ereignis jedoch nur einmal
- Auch diese Rückruffunktion wird auf ISR-Ebene ausgeführt

2 Anmeldung für Benachrichtigung bei Tasterevents

```
uint8_t sb_button_registerListener(  
    BUTTON btn, BUTTONEVENT eve, buttoncallback_t callback);
```

- `btn`: für welchen Button sollen Events gemeldet werden
- `eve`: für welches Tasterereignis soll eine Benachrichtigung erfolgen (Drücken, Loslassen, oder beides)
- `callback`: die Funktion, die zur Benachrichtigung aufgerufen wird

3 Abmeldung, Polling

- siehe Online-Dokumentation

U2-5 Beispiel: Countdown von 99-0 mit Resettaste

```
#include <timer.h>
#include <7seg.h>
#include <button.h>
#include <avr/interrupt.h>
static int8_t cnt;
static void reset ( BUTTON btn, BUTTONEVENT eve ) { cnt = 99; }

void main( void ) {
    sb_button_registerListener(BUTTON0, BTNPRESSED, reset);
    sei(); /* Interrupts initial erlauben */

    while ( 1 ) {
        for(cnt=99; cnt >= 0; ) {
            sb_7seg_showNumber(cnt); /* akt. Zahl anzeigen */
            sb_timer_delay(100); /* 100 ms warten */
            cnt--; /* countdown */
        }

        sb_timer_delay(5000); /* 5s warten, dann von vorn beginnen */
    }
}
```

U2-6 Wiederholung: der volatile-Qualifier

- Manche Variablen werden "von außen" verändert
 - ◆ Hardware-Register
 - ◆ Variablen, auf die mehrere Programmabläufe nebenläufig zugreifen
 - Threads
 - Unterbrechungsbehandlungen
- Kann Probleme bei Compileroptimierungen verursachen

1 Beispiel

- Hauptprogramm wartet auf ein Ereignis, das durch einen Interrupt gemeldet wird (dieser setzt **event** auf 1)
- Aktive Warteschleife wartet, bis **event != 0**
- Der Compiler sieht, dass **event** innerhalb der Warteschleife nicht verändert wird
 - der Wert von **event** wird nur einmal **vor** der Warteschleife aus dem Speicher in ein Prozessorregister geladen
 - dann wird nur noch der Wert im Register betrachtet
 - Endlosschleife

```
static char event=0;
ISR (INT0_vect) { event=1; }

void main(void) {
    while(1) {
        while(event == 0) { /* warte auf Event */ }
        /* bearbeite Event */
    }
}
```

2 Beispiel (2)

- Lösung: Unterdrücken der Optimierung mit dem **volatile**-Schlüsselwort

```
static volatile char event=0;
ISR (INT0_vect) { event=1; }

void main(void) {
    while(1) {
        while(event == 0) { /* warte auf Event */ }
        /* bearbeite Event */
    }
}
```

- Teilt dem Compiler mit, dass der Wert extern verändert wird
- Wert wird bei jedem Lesezugriff erneut aus dem Speicher geladen
- **Sofort-Aufgabe:** Schreiben Sie gemeinsam ein Programm, das ähnlich obigem Beispiel die Bibliothek verwendet. Das Programm soll auf ein Event warten (wie oben), das von einem Taster-Eventhandler gesetzt wird. Danach aktiviert das Programm eine LED zur Bestätigung. Kompilieren und Testen Sie das Programm mit -O0 sowie -O3.

3 volatile in einem anderen Zusammenhang...

- Beispiel: Funktion zum aktiven Warten

```
void wait(unsigned int len) {
    while(len > 0) { len--; }
}
```

- Die Schleife wird terminieren
- Der Wert von `len` wird bei Verlassen der Funktion verworfen
 - Optimierender Compiler entfernt die komplette Schleife
 - Wartezeit wird stark verkürzt
- Mit **volatile** kann diese Optimierung verhindert werden
 - Sonderfall: `len` wird nicht extern modifiziert

4 Verwendung von volatile

- Fehlendes **volatile** kann zu unerwartetem Programmablauf führen
- Unnötige Verwendung von **volatile** unterbindet Optimierungen des Compilers und führt zu schlechterem Maschinencode
- Korrekte Verwendung von **volatile** ist Aufgabe des Programmierers!

Verwendung von **volatile** so selten wie möglich, aber so oft wie nötig

5 volatile im Beispielprogramm?

```
#include <timer.h>
#include <7seg.h>
#include <button.h>
#include <avr/interrupt.h>
static int8_t cnt;
static void reset ( BUTTON btn, BUTTONEVENT eve ) { cnt = 99; }

void main( void ) {
    sb_button_registerListener(BUTTON0, BTNPRESSED, reset);
    sei(); /* Interrupts initial erlauben */

    while ( 1 ) {
        for(cnt=99; cnt >= 0; ) {
            sb_7seg_showNumber(cnt); /* akt. Zahl anzeigen */
            sb_timer_delay(100); /* 100 ms warten */
            cnt--; /* countdown */
        }

        sb_timer_delay(5000); /* 5s warten, dann von vorn beginnen */
    }
}
```

5 volatile im Beispielprogramm?

```
#include <timer.h>
#include <7seg.h>
#include <button.h>
#include <avr/interrupt.h>
static volatile int8_t cnt;
static void reset ( BUTTON btn, BUTTONEVENT eve ) { cnt = 99; }

void main( void ) {
    sb_button_registerListener(BUTTON0, BTNPRESSED, reset);
    sei(); /* Interrupts initial erlauben */

    while ( 1 ) {
        for(cnt=99; cnt >= 0; ) {
            sb_7seg_showNumber(cnt); /* akt. Zahl anzeigen */
            sb_timer_delay(100); /* 100 ms warten */
            cnt--; /* countdown */
        }

        sb_timer_delay(5000); /* 5s warten, dann von vorn beginnen */
    }
}
```

U2-7 Synchronisation mit Interrupt-Handlern

- Unterbrechungsbehandlungen führen zu **Nebenläufigkeit**
- Nicht-atomare Modifikation von gemeinsamen Daten
- Kann zu Inkonsistenzen führen
- Einseitige Unterbrechung: Interrupt-Handler überlappt Hauptprogramm
- Lösung: Synchronisationsprimitiven
 - hier: zeitweilige Deaktivierung der Interruptbehandlung

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrucke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */

/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5		
2			
4			
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;
```

```
/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */
```

```
/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2			
4			
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrücke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;
```

```
/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */
```

```
/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4			
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrucke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;
```

```
/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */
```

```
/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5			
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrucke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;
```

```
/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */
```

```
/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5	5	4	6
6			
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrucke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */

/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5	5	4	6
6	6	4	6
3			

1 Beispiel 1: Lost Update

- Tastendruckzähler: Zählt noch zu bearbeitende Tastendrucke
 - Inkrementierung in der Unterbrechungsbehandlung
 - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

```
/* Hauptprogramm */
volatile unsigned char zaehler;

/* C-Anweisung: zaehler--; */
1 lds r24, zaehler
2 dec r24
3 sts zaehler, r24
```

```
/* Interrupt-Behandlung */

/* C-Anweisung: zaehler++ */
4 lds r24, zaehler
5 inc r24
6 sts zaehler, r24
```

Instruktion	zaehler	zaehler HP	zaehler INT
1	5	5	-
2	5	4	-
4	5	4	5
5	5	4	6
6	6	4	6
3	4	4	-

2 Beispiel 2: 16-bit-Zugriffe

■ Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile unsigned int zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r24, zaehler
2 lds r25, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	
3-7		
2		

2 Beispiel 2: 16-bit-Zugriffe

■ Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile unsigned int zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r24, zaehler
2 lds r25, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	0x??ff
3-7		
2		

2 Beispiel 2: 16-bit-Zugriffe

■ Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile unsigned int zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r24, zaehler
2 lds r25, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	0x??ff
3-7	0x0100	0x??ff
2		

2 Beispiel 2: 16-bit-Zugriffe

■ Nebenläufige Nutzung von 16-bit-Werten

```
/* Hauptprogramm */
volatile unsigned int zaehler;

/* C-Anweisung: z=zaehler; */
1 lds r24, zaehler
2 lds r25, zaehler+1
/* z verwenden... */
```

```
/* Interrupt-Behandlung */
/* C-Anweisung: zaehler++ */
3 lds r24, zaehler
4 lds r25, zaehler+1
5 adiw r24,1
6 sts zaehler+1, r25
7 sts zaehler, r24
```

Instruktion	zaehler	z HP
1	0x00ff	0x??ff
3-7	0x0100	0x??ff
2	0x0100	0x01ff

■ Weitere Problemszenarien?

■ Nebenläufige Zugriffe auf Werte >8-bit müssen i.d.R. geschützt werden!

3 Sperren der Unterbrechungsbehandlung beim AVR

- Viele weitere Nebenläufigkeitsprobleme möglich
 - ◆ Problemanalyse durch den Anwendungsprogrammierer
 - ◆ Vorsicht bei gemeinsamen Daten nebenläufiger Kontrollflüsse
 - ◆ Auswahl geeigneter Synchronisationsprimitive
- Lösung hier: Einseitiger Ausschluss durch Sperren der Interrupts
 - ◆ Sperrung aller Interrupts (`sei()`, `cli()`)
 - ◆ Maskieren einzelner Interrupts (GICR-Register)
- Problem: Interrupts während der Sperrung gehen evtl. verloren

4 Synchronisation im Beispiel?

```
#include <timer.h>
#include <7seg.h>
#include <button.h>
#include <avr/interrupt.h>
static int8_t cnt;
static void reset ( BUTTON btn, BUTTONEVENT eve ) { cnt = 99; }

void main( void ) {
    sb_button_registerListener(BUTTON0, BTNPRESSED, reset);
    sei(); /* Interrupts initial erlauben */

    while ( 1 ) {
        for(cnt=99; cnt >= 0; ) {
            sb_7seg_showNumber(cnt); /* akt. Zahl anzeigen */
            sb_timer_delay(100); /* 100 ms warten */
            cnt--; /* countdown */
        }

        sb_timer_delay(5000); /* 5s warten, dann von vorn beginnen */
    }
}
```

4 Synchronisation im Beispiel? Lost Update Problem

```
#include <timer.h>
#include <7seg.h>
#include <button.h>
#include <avr/interrupt.h>
static volatile int8_t cnt;
static void reset ( BUTTON btn, BUTTONEVENT eve ) { cnt = 99; }

void main( void ) {
    sb_button_registerListener(BUTTON0, BTNPRESSED, reset);
    sei(); /* Interrupts initial erlauben */

    while ( 1 ) {
        for(cnt=99; cnt >= 0; ) {
            sb_7seg_showNumber(cnt); /* akt. Zahl anzeigen */
            sb_timer_delay(100); /* 100 ms warten */
            cli(); cnt--; sei(); /* countdown, sync wozu? */
        }

        sb_timer_delay(5000); /* 5s warten, dann von vorn beginnen */
    }
}
```

U2-8 Aufgabe 1

- Messung der Dauer eines Tastendrucks in 100 ms Intervallen
- Anzeige der laufenden Messung auf den 7-Segment-Anzeigen
- Anzeige des Ergebnisses auf den 7-Segment-Anzeigen
- Verwendung der SPiCboard-Bibliothek zur Implementierung