Übungen zu Systemnahe Programmierung in C (SPiC) – Wintersemester 2019/2020

Übung 2

Benedict Herzog
Bernhard Heinloth
Tim Rheinfels

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Bits & Bytes

Bitoperationen

- Übersicht:

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

∧	0	1
0	0	1
1	1	0

~	
0	1
1	0

Bitoperationen

- Übersicht:

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

∧	0	1
0	0	1
1	1	0

~	
0	1
1	0

- Beispiel:

	1100	1100	1100
~	&	|	∧
1001	1001	1001	1001
0110	1000	1101	0101

Shiftoperationen

- Beispiel:

`uint8_t x = 0x9d;`	1	0	0	1	1	1	0	1
`x = x << 2;`	0	1	1	1	0	1	**0**	**0**
`x = x >> 2;`	**0**	**0**	0	1	1	1	0	1

- Setzen von Bits:

`(1 << 0)`	0	0	0	0	0	0	0	**1**
`(1 << 3)`	0	0	0	0	**1**	0	0	0
`(1 << 3) | (1 << 0)`	0	0	0	0	**1**	0	0	**1**

- **Achtung:**
  Bei `signed`-Variablen ist das Verhalten des >>-Operators nicht 100% definiert. Im Normalfall(!) werden bei negativen Werten 1er geshiftet.

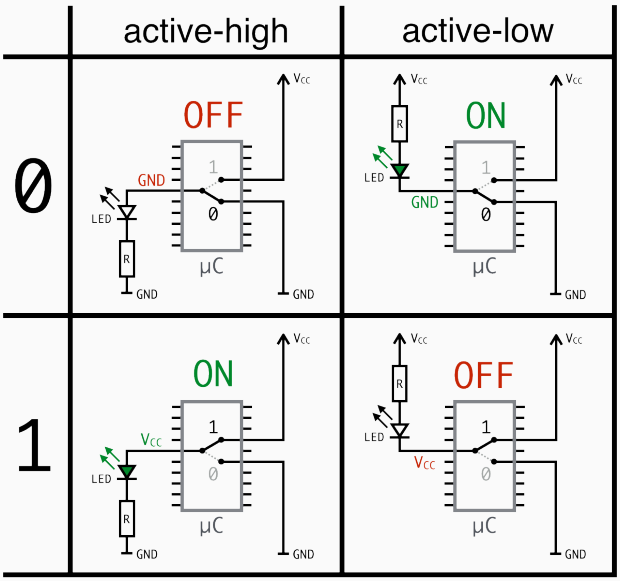
Ein- & Ausgabe über Pins

Ausgang: active-high & active-low

Ausgang je nach Beschaltung:

active-high high-Pegel (logisch 1; V_{cc} am Pin) → LED leuchtet

active-low low-Pegel (logisch 0; *GND* am Pin) → LED leuchtet

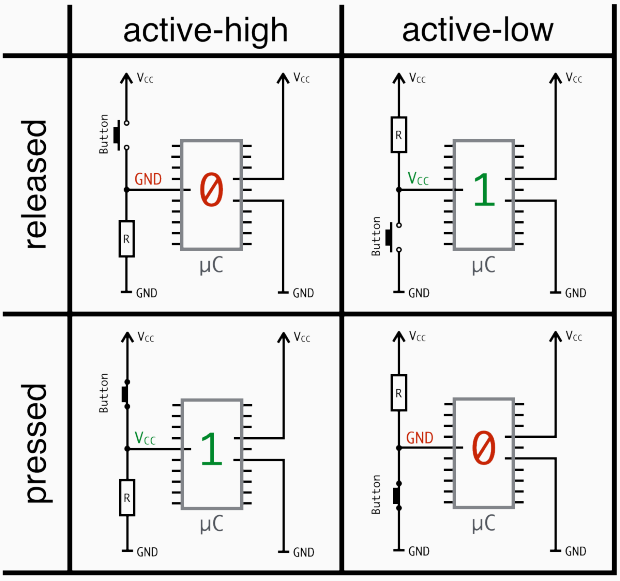


Eingang: active-high & active-low

Eingang je nach Beschaltung:

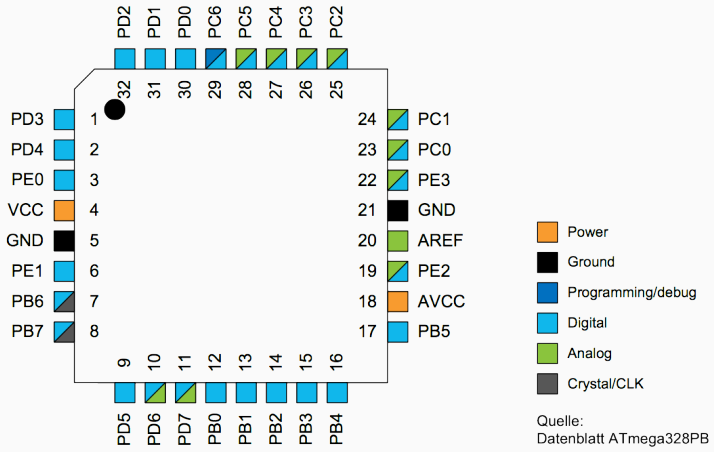
active-high Button gedrückt → high-Pegel (logisch 1; V_{cc} am Pin)

active-low Button gedrückt → low-Pegel (logisch 0; *GND* am Pin)



→ ATmega328PB besitzt einen internen (zuschaltbaren) Pull-Up Widerstand

Konfiguration der Pins



- jeweils acht Pins am AVR sind zu einem I/O Port zusammengefasst
- jeder I/O-Port des AVR wird durch drei 8-bit Register gesteuert:

  DDRx Datenrichtungsregister (Data Direction Register)
  PORTx Portausgaberegister (Port Output Register)
  PINx Porteingaberegister (Port Input Register)
- jedem Pin eines Ports ist jew. ein Bit in den drei Register zugeordnet

I/O-Port-Register (1)

DDRx: Data Direction Register konfiguriert Pin `i` als Ein- oder Ausgang

- `Bit i = 1` → Pin `i` als Ausgang verwenden
- `Bit i = 0` → Pin `i` als Eingang verwenden

Beispiel:

```
01 DDRC |= (1 << PC3);  // PC3 als Ausgang (Pin 3 an Port C)
02 DDRD &= ~(1 << PD2); // PD2 als Eingang (Pin 2 an Port D)
```

I/O-Port-Register (2)

PORTx: Port Output Register abhängig von DDRx Register

- wenn **Ausgang**: legt high- oder low-Pegel an Pin `i` an
  - `Bit i = 1` → high-Pegel an Pin `i`
  - `Bit i = 0` → low-Pegel an Pin `i`
- wenn **Eingang**: konfiguriert Eingangswiderstand an Pin `i`
  - `Bit i = 1` → aktiviert internen Pull-Up Widerstand für Pin `i`
  - `Bit i = 0` → konfiguriert Pin `i` als hochohmig (Tri-State)
    → Der Mikrocontroller selbst beeinflusst den Eingangspegel nicht.
    Der Eingangspegel muss deshalb durch externe Beschaltung stets auf einen definierten Pegel (high oder low) gezogen werden (z.B. durch einen externen Pull-Up Widerstand).

Beispiel:

```
01 PORTC |= (1 << PC3);  // zieht PC3 auf high (LED aus)
02 PORTC &= ~(1 << PC3); // zieht PC3 auf low (LED an)
03
04 PORTD |= (1 << PD2);  // aktiviert Pull-Up für PD2
05 PORTD &= ~(1 << PD2); // konfiguriert PD2 hochohmig
```


I/O-Port-Register (3)

PINx: Port Input Register (nur lesbar) aktuellen Wert von Pin `i`

- wenn **Eingang**: abrufen was von extern anliegt
- wenn **Ausgang**: abrufen ob high oder low ausgegeben wird

Beispiel:

```
01 if((PIND & (1 << PD2)) == 0) { // Testen ob Pin PD2 low ist
02   // low Pegel --> Button ist gedrückt
03   [...]
04 }
05
06 if((PIND & (1 << PD2)) != 0) { // Testen ob Pin PD2 high ist
07   // high Pegel --> Button ist nicht gedrückt
08   [...]
09 }
```

Interrupts

Interrupts

- Ablauf eines Interrupts (vgl. 15-7)
  0. Hardware setzt entsprechendes Flag
  1. Sind die Interrupts aktiviert und der Interrupt nicht maskiert, unterbricht der Interruptcontroller die aktuelle Ausführung
  2. weitere Interrupts werden deaktiviert
  3. aktuelle Position im Programm wird gesichert
  4. Adresse des Handlers wird aus Interrupt-Vektor gelesen und angesprungen
  5. Ausführung des Interrupt-Handlers
  6. am Ende des Handlers bewirkt ein Befehl “Return from Interrupt” die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts

Implementierung von Interruptbehandlungen

- Je Interrupt steht ein Bit zum Zwischenspeichern zur Verfügung
- Ursachen für den Verlust von weiteren Interrupts
  - Während einer Interruptbehandlung
  - Interruptsperren (zur Synchronisation von kritischen Abschnitten)
- Das Problem ist generell nicht zu verhindern

⇝ Risikominimierung: Interruptbehandlungen sollten möglichst kurz sein

- Schleifen und Funktionsaufrufe vermeiden
- Auf blockierende Funktionen verzichten (ADC/serielle Schnittstelle!)

Interrupts beim AVR

- Timer
- Serielle Schnittstelle
- ADC (Analog-Digital-Umsetzer)
- Externe Interrupts durch Pegel(änderung) an bestimmten I/O-Pins
  - Wahlweise pegel- oder flankengesteuert
  - Abhängig von der jeweiligen Interruptquelle

  ⇒ ATmega328PB: 2 Quellen an den Pins PD2 (`INT0`) und PD3 (`INT1`)

  ⇒ `BUTTON0` an PD2

  ⇒ `BUTTON1` an PD3
- Dokumentation im ATmega328PB-Datenblatt
  - Interruptbehandlung allgemein: S. 77-80
  - Externe Interrupts: S. 81-91

(De-)Aktivieren von Interrupts

- Interrupts können durch die spezielle Maschinenbefehle aktiviert bzw. deaktiviert werden.
- Die Bibliothek avr-libc bietet hierfür Makros an:
  `#include <avr/interrupt.h>`
  - `sei()` (Set Interrupt Flag): lässt Interrupts zu (1 Instruktion verzögert)
  - `cli()` (Clear Interrupt Flag): blockiert alle Interrupts (sofort)
- Beim Betreten eines Interrupt-Handlers werden automatisch alle Interrupts blockiert, beim Verlassen werden sie wieder deblockiert
- `sei()` sollte niemals in einer Interruptbehandlung ausgeführt werden
  - potentiell endlos geschachtelte Interruptbehandlung
  - Stackoverflow möglich
- Beim Start des µC sind die Interrupts abgeschaltet

Konfigurieren von Interrupts

- Interrupt Sense Control (ISC) Bits befinden sich beim ATmega328PB im External Interrupt Control Register A (EICRA)
- Position der ISC-Bits im Register durch Makros definiert

Interrupt 0		Interrupt bei	Interrupt 1	
ISC01	ISC00		ISC11	ISC10
0	0	low Pegel	0	0
0	1	beliebiger Flanke	0	1
1	0	fallender Flanke	1	0
1	1	steigender Flanke	1	1

- Beispiel: INT1 bei ATmega328PB für fallende Flanke konfigurieren

```
01 /* die ISC-Bits befinden sich im EICRA */
02 EICRA &= ~(1 << ISC10); // ISC10 löschen
03 EICRA |= (1 << ISC11);  // ISC11 setzen
```

(De-)Maskieren von Interrupts

- Einzelne Interrupts können separat aktiviert (=demaskiert) werden
  - ATmega328PB: External Interrupt Mask Register (`EIMSK`)
- Die Bitpositionen in diesem Register sind durch Makros `INTn` definiert
- Ein gesetztes Bit aktiviert den jeweiligen Interrupt
- Beispiel: Interrupt `1` aktivieren

```
01 EIMSK |= (1 << INT1); // demaskiere Interrupt 1
```


Interrupt-Handler

- Installieren eines Interrupt-Handlers wird durch C-Bibliothek unterstützt
- Makro `ISR` (Interrupt Service Routine) zur Definition einer Handler-Funktion (`#include <avr/interrupt.h>`)
- Parameter: gewünschten Vektor; z. B. `INT1_vect` für externen Interrupt `1`
  - verfügbare Vektoren: siehe avr-libc-Doku zu `avr/interrupt.h`
- Beispiel: Handler für Interrupt `1` implementieren

```
01 #include <avr/interrupt.h>
02 static uint16_t zaehler = 0;
03
04 ISR(INT1_vect) {
05   zaehler++;
06 }
```

Synchronisation

Schlüsselwort volatile

- Bei einem Interrupt wird `event = 1` gesetzt
- Aktive Warteschleife wartet, bis `event != 0`
- Der Compiler erkennt, 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
  - ⇒ Endlosschleife

```
01 static uint8_t event = 0;
02 ISR(INT0_vect) {
03     event = 1;
04 }
05
06 void main(void) {
07     while(1) {
08         while(event == 0) { /* warte auf Event */ }
09         // bearbeite Event [...]
10     }
11 }
```

Schlüsselwort volatile

- Bei einem Interrupt wird `event = 1` gesetzt
- Aktive Warteschleife wartet, bis `event != 0`
- Der Compiler erkennt, 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
  - ⇒ Endlosschleife
- `volatile` erzwingt das Laden bei jedem Lesezugriff

```
01 static volatile uint8_t event = 0;
02 ISR(INT0_vect) {
03     event = 1;
04 }
05
06 void main(void) {
07     while(1) {
08         while(event == 0) { /* warte auf Event */ }
09         // bearbeite Event [...]
10     }
11 }
```


Verwendung von volatile

- Fehlendes `volatile` kann zu unerwartetem Programmablauf führen
- Unnötige Verwendung von `volatile` unterbindet Optimierungen des Compilers
- Korrekte Verwendung von `volatile` ist Aufgabe des Programmierers!

⇝ Verwendung von `volatile` so selten wie möglich, aber so oft wie nötig

Lost Update

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

```
01 static volatile uint8_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07   while(1) {
08     if(counter > 0) {
09
10       counter--;
11
12       // verarbeite Tastendruck
13       // [...]
14     }
15   }
16 }
```


Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		

Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		
2	5	5	—

Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		
2	5	5	—
3	5	4	—

Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		
2	5	5	—
3	5	4	—
6	5	4	5

Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		
2	5	5	—
3	5	4	—
6	5	4	5
7	5	4	6

Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		
2	5	5	—
3	5	4	—
6	5	4	5
7	5	4	6
8	6	4	6

Lost Update

Hauptprogramm

```
01 ; C-Anweisung: counter--;
02 lds r24, counter
03 dec r24
04 sts counter, r24
```

Interruptbehandlung

```
05 ; C-Anweisung: counter++
06 lds r25, counter
07 inc r25
08 sts counter, r25
```

Zeile	counter	r24	r25
—	5		
2	5	5	—
3	5	4	—
6	5	4	5
7	5	4	6
8	6	4	6
4	4	4	—

16-Bit Zugriffe (Read-Write)

- Nebenläufige Nutzung von 16-Bit Werten (Read-Write)
  - Inkrementierung in der Unterbrechungsbehandlung
  - Auslesen im Hauptprogramm

```
01 static volatile uint16_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07   if(counter > 300) {
08     sb_led_on(YELLOW0);
09   } else {
10     sb_led_off(YELLOW0);
11   }
12
13   // [...]
14 }
```


16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01 ; C-Anweisung: if(counter > 300)
02 lds  r22, counter
03 lds  r23, counter+1
04 cpi  r22, 0x2D
05 sbci r23, 0x01
```

Interruptbehandlung

```
07 ; C-Anweisung: counter++;
08 lds r24, counter
09 lds r25, counter+1
10 adiw r24,1
11 sts counter+1, r25
12 sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—

16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01  ; C-Anweisung: if(counter > 300)
02  lds  r22, counter
03  lds  r23, counter+1
04  cpi  r22, 0x2D
05  sbci r23, 0x01
```

Interruptbehandlung

```
07  ; C-Anweisung: counter++;
08  lds r24, counter
09  lds r25, counter+1
10  adiw r24,1
11  sts counter+1, r25
12  sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—
2	0x00ff	0x??ff	—

16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01  ; C-Anweisung: if(counter > 300)
02  lds  r22, counter
03  lds  r23, counter+1
04  cpi  r22, 0x2D
05  sbci r23, 0x01
```

Interruptbehandlung

```
07  ; C-Anweisung: counter++;
08  lds r24, counter
09  lds r25, counter+1
10  adiw r24,1
11  sts counter+1, r25
12  sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—
2	0x00ff	0x??ff	—
8+9	0x00ff	0x??ff	0x00ff

16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01  ; C-Anweisung: if(counter > 300)
02  lds  r22, counter
03  lds  r23, counter+1
04  cpi  r22, 0x2D
05  sbci r23, 0x01
```

Interruptbehandlung

```
07  ; C-Anweisung: counter++;
08  lds r24, counter
09  lds r25, counter+1
10  adiw r24,1
11  sts counter+1, r25
12  sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—
2	0x00ff	0x??ff	—
8+9	0x00ff	0x??ff	0x00ff
10	0x00ff	0x??ff	0x0100

16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01  ; C-Anweisung: if(counter > 300)
02  lds  r22, counter
03  lds  r23, counter+1
04  cpi  r22, 0x2D
05  sbci r23, 0x01
```

Interruptbehandlung

```
07  ; C-Anweisung: counter++;
08  lds r24, counter
09  lds r25, counter+1
10  adiw r24,1
11  sts counter+1, r25
12  sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—
2	0x00ff	0x??ff	—
8+9	0x00ff	0x??ff	0x00ff
10	0x00ff	0x??ff	0x0100
11+12	0x0100	0x??ff	0x0100

16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01 ; C-Anweisung: if(counter > 300)
02 lds  r22, counter
03 lds  r23, counter+1
04 cpi  r22, 0x2D
05 sbci r23, 0x01
```

Interruptbehandlung

```
07 ; C-Anweisung: counter++;
08 lds r24, counter
09 lds r25, counter+1
10 adiw r24,1
11 sts counter+1, r25
12 sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—
2	0x00ff	0x??ff	—
8+9	0x00ff	0x??ff	0x00ff
10	0x00ff	0x??ff	0x0100
11+12	0x0100	0x??ff	0x0100
3	0x0100	0x01ff	—

16-Bit Zugriffe (Read-Write)

Hauptprogramm

```
01 ; C-Anweisung: if(counter > 300)
02 lds  r22, counter
03 lds  r23, counter+1
04 cpi  r22, 0x2D
05 sbci r23, 0x01
```

Interruptbehandlung

```
07 ; C-Anweisung: counter++;
08 lds r24, counter
09 lds r25, counter+1
10 adiw r24,1
11 sts counter+1, r25
12 sts counter, r24
```

Zeile	counter	r22 & r23	r24 & r25
—	0x00ff	—	—
2	0x00ff	0x??ff	—
8+9	0x00ff	0x??ff	0x00ff
10	0x00ff	0x??ff	0x0100
11+12	0x0100	0x??ff	0x0100
3	0x0100	0x01ff	—

⇒ Vergleich in Zeile 4+5 wird mit `0x01ff` (entspricht 511) statt korrekterweise mit `0x0100` (entspricht 256) durchgeführt. Der Vergleich ergibt also `true` und die LED wird angeschaltet.

Sperren der Unterbrechungsbehandlung beim AVR

- Viele weitere Nebenläufigkeitsprobleme möglich
  - nicht-atomare Modifikation von gemeinsamen Daten
  - Problemanalyse durch den Anwendungsprogrammierer
  - Auswahl geeigneter Synchronisationsprimitive
- Lösung hier: Einseitiger Ausschluss durch Sperren der Interrupts
  - Sperrung aller Interrupts: `cli()` und `sei()`
  - Maskieren einzelner Interrupts (`EIMSK`-Register)
- Problem: Interrupts während der Sperrung gehen evtl. verloren

⇒ Kritische Abschnitte müssen so kurz wie möglich sein

Lost Update

- Wie kann man das Lost Update verhindern?

```
01 static volatile uint8_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07   while(1) {
08     if(counter > 0) {
09
10       counter--;
11
12       // verarbeite Tastendruck
13       // [...]
14     }
15   }
16 }
```

Lost Update

- Wie kann man das Lost Update verhindern?

```
01 static volatile uint8_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07   while(1) {
08     if(counter > 0) {
09       cli();
10       counter--;
11       sei();
12       // verarbeite Tastendruck
13       // [...]
14     }
15   }
16 }
```


16-Bit Zugriffe (Read-Write)

- Wie kann man die Read-Write Anomalie verhindern?

```
01 static volatile uint16_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07
08
09
10   if(counter > 300) {
11
12     sb_led_on(YELLOW0);
13   } else {
14
15     sb_led_off(YELLOW0);
16   }
17
18   // [...]
19 }
```

16-Bit Zugriffe (Read-Write)

- Wie kann man die Read-Write Anomalie verhindern?

```
01 static volatile uint16_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07   cli();
08   uint16_t local_counter = counter;
09   sei();
10   if(local_counter > 300) {
11
12     sb_led_on(YELLOW0);
13   } else {
14
15     sb_led_off(YELLOW0);
16   }
17
18   // [...]
19 }
```


16-Bit Zugriffe (Read-Write)

- Wie kann man die Read-Write Anomalie verhindern?

```
01 static volatile uint16_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07
08
09   cli();
10   if(counter > 300) {
11
12     sb_led_on(YELLOW0);
13   } else {
14
15     sb_led_off(YELLOW0);
16   }
17   sei();
18   // [...]
19 }
```

16-Bit Zugriffe (Read-Write)

- Wie kann man die Read-Write Anomalie verhindern?

```
01 static volatile uint16_t counter = 0;
02 ISR(INT0_vect) {
03   counter++;
04 }
05
06 void main(void) {
07
08
09   cli();
10   if(counter > 300) {
11     sei();
12     sb_led_on(YELLOW0);
13   } else {
14     sei();
15     sb_led_off(YELLOW0);
16   }
17
18   // [...]
19 }
```


Stromsparmodi

Stromsparmodi von AVR-Prozessoren

- AVR-basierte Geräte oft batteriebetrieben (z.B. Fernbedienung)
- Energiesparen kann die Lebensdauer drastisch erhöhen
- AVR-Prozessoren unterstützen unterschiedliche Powersave-Modi
  - Deaktivierung funktionaler Einheiten
  - Unterschiede in der “Tiefe” des Schlafes
  - Nur aktive funktionale Einheiten können die CPU aufwecken
- Standard-Modus: Idle
  - CPU-Takt wird angehalten
  - Keine Zugriffe auf den Speicher
  - Hardware (Timer, externe Interrupts, ADC, etc.) sind weiter aktiv
- Dokumentation im ATmega328PB-Datenblatt, S. 58-66

Nutzung der Sleep-Modi

- Unterstützung aus der avr-libc: (`#include <avr/sleep.h>`)
  - `sleep_enable()` - aktiviert den Sleep-Modus
  - `sleep_cpu()` - setzt das Gerät in den Sleep-Modus
  - `sleep_disable()` - deaktiviert den Sleep-Modus
  - `set_sleep_mode(uint8_t mode)` - stellt den zu verwendenden Modus ein
- Dokumentation von `avr/sleep.h` in avr-libc-Dokumentation

```
01 #include <avr/sleep.h>
02
03 set_sleep_mode(SLEEP_MODE_IDLE); // Idle-Modus verwenden
04 sleep_enable();   // Sleep-Modus aktivieren
05 sleep_cpu();      // Sleep-Modus betreten
06 sleep_disable();  // Empfohlen: Sleep-Modus danach deaktivieren
```

Lost Wakeup

- Dornröschenschlaf
  - ⇒ **Problem:** Es kommt genau ein Interrupt

Hauptprogramm

```
01 sleep_enable();
02 event = 0;
03
04
05 while(!event) {
06
07     sleep_cpu();
08
09 }
10
11
12 sleep_disable();
```

Interruptbehandlung

```
01 ISR(TIMER1_COMPA_vect) {
02     event = 1;
03 }
```


Lost Wakeup

- Dornröschenschlaf
  - ⇒ **Problem:** Es kommt genau ein Interrupt

Hauptprogramm

```
01 sleep_enable();
02 event = 0;
03
04
05 while(!event) {
06     ↯ Interrupt ↯
07     sleep_cpu();
08
09 }
10
11
12 sleep_disable();
```

Interruptbehandlung

```
01 ISR(TIMER1_COMPA_vect) {
02     event = 1;
03 }
```

Lost Wakeup

- Dornröschenschlaf
  - ⇒ **Problem:** Es kommt genau ein Interrupt
  - ⇒ **Lösung:** Interrupts während des kritischen Abschnitts sperren

Hauptprogramm

```
01 sleep_enable();
02 event = 0;
03
04 cli();
05 while(!event) {
06     sei();
07     sleep_cpu();
08     cli();
09 }
10 sei();
11
12 sleep_disable();
```

Interruptbehandlung

```
01 ISR(TIMER1_COMPA_vect) {
02     event = 1;
03 }
```


Lost Wakeup

- Dornröschenschlaf
  - ⇒ **Problem:** Es kommt genau ein Interrupt
  - ⇒ **Lösung:** Interrupts während des kritischen Abschnitts sperren

Hauptprogramm

```
01 sleep_enable();
02 event = 0;
03
04 cli();
05 while(!event) {
06     sei(); ↯ Interrupt ↯
07     sleep_cpu();
08     cli();
09 }
10 sei();
11
12 sleep_disable();
```

Interruptbehandlung

```
01 ISR(TIMER1_COMPA_vect) {
02     event = 1;
03 }
```

⇒ Was ist wenn der Interrupt zwischen Zeile 6 und 7 auftritt?

Aufgabe: Interrupt Zähler

Aufgabe: Interrupt Zähler

- Zählen der Tastendrücke an Taster 0
- vorübergehendes Aktivieren der Anzeige durch Taster 1
  - Deaktivieren der Anzeige nach 1 - 10 Sekunden (einstellbar über das Potentiometer)
  - Anzeige über 7-Segment Anzeige und LEDs
  - bei Verlassen des anzeigbaren Wertebereichs Zähler zurücksetzen
  - aktive Anzeige bei Änderung des Zählerstandes aktualisieren
- Erkennung der Tastendrücke ohne Polling
  - Interrupts verwenden (fallende Flanke)
  - CPU in den Schlafmodus versetzen, wenn nichts zu tun ist
- Hinweise:
  - Erkennung der Tastendrücke **ohne** `libspicboard`
  - Interrupts nur kurzzeitig sperren; Interrupt Handler kurz halten
  - auf richtige Synchronisation achten

Hinweise zur Verwendung des Timers

```
01 static void alarm_handler(void) {
02     alarm_event = 1;
03     alarm = 0;
04 }
05
06 void main(void) {
07     sei();
08     alarm = sb_timer_setAlarm(alarm_handler, 1000, 0);
09
10     // [...]
11
12     cli();
13     if(alarm) {
14         sb_timer_cancelAlarm(alarm);
15     }
16     sei();
17 }
```

- Handler im Interrupt-Kontext ($\rightsquigarrow$ gesperrte Interrupts)
- Single-Shot Alarme (`cycle = 0`) dürfen nur abgebrochen werden, **bevor** sie ausgelöst haben (Nebenläufigkeit!)

Hands-on: Interrupts & Sleep

Hands-on: Einfacher Interruptzähler

- Zählen der Tastendrücke an `BUTTON0` (PD2)
- Erkennung der Tastendrücke mit Hilfe von Interrupts
- Ausgabe des aktuellen Zählerwerts über 7-Segment Anzeige
- CPU in den Schlafmodus versetzen, so lange **Zählerwert gerade**
- “Standby”-LED leuchtet während des Schlafens (`BLUE0`)
- Hinweise:
  - Erkennung der Tastendrücke **ohne** die `libspicboard`
  - PD2/`BUTTON0` ist der Eingang von `INT0`
  - Interrupt bei fallender Flanke:
    - `EICRA(ISC00) = 0`
    - `EICRA(ISC01) = 1`
  - 7-Segment Anzeige braucht regelmäßig Interrupts, um Werte anzeigen zu können