Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier
(Lehrstuhl Informatik 4)

Übung 5

Sommersemester 2016

Inhalt

Inhalt

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

⇒ ATmega32: 3 Quellen an den Pins `PD2`, `PD3` und `PB2`

  - Pegel- oder flankengesteuert
  - Abhängig von der jeweiligen Interruptquelle
  - Konfiguration über Bits
  - Beispiel: Externer Interrupt 2 (`INT2`)

`ISC2`	IRQ bei:
0	fallender Flanke
1	steigender Flanke

- Dokumentation im ATmega32-Datenblatt
  - Interruptbehandlung allgemein: S. 45-49
  - Externe Interrupts: S. 69-72

(De-)Aktivieren von Interrupts beim AVR

- 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 ab dem nächsten Takt Interrupts zu
  - `cli()` (Clear Interrupt Flag) - blockiert (sofort) alle Interrupts
- 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 (Vorlesung, voraussichtlich Kapitel 17)
- Beim Start des µC sind die Interrupts abgeschaltet

Konfigurieren von Interrupts

- Beim ATmega32 verteilen sich die Interrupt Sense Control (ISC)-Bits zur Konfiguration der externen Interrupts auf zwei Register:
  - INT0, INT1: MCU Control Register (MCUCR)
  - INT2: MCU Control and Status Register (MCUCSR)
- Position der ISC-Bits in den Registern durch Makros definiert ISCn0 und ISCn1 (INT0 und INT1) oder ISC2 (INT2)
- Beispiel: INT2 bei ATmega32 für fallende Flanke konfigurieren

```
1 /* die ISCs für INT2 befinden sich im MCUCSR */
2 MCUCSR &= ~(1<<ISC2); /* ISC2 löschen */
```

(De-)Maskieren von Interrupts

- Einzelne Interrupts können separat aktiviert (=demaskiert) werden
  - ATmega32: General Interrupt Control Register (`GICR`)
- Die Bitpositionen in diesem Register sind durch Makros `INTn` definiert
- Ein gesetztes Bit aktiviert den jeweiligen Interrupt
- Beispiel: Interrupt 2 aktivieren

```
1 GICR |= (1<<INT2); /* demaskiere Interrupt 2 */
```

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. `INT2_vect` für externen Interrupt 2
  - verfügbare Vektoren: siehe avr-libc-Doku zu `avr/interrupt.h`
    ⇒ verlinkt im Doku-Bereich auf der SPiC-Webseite
- Beispiel: Handler für Interrupt 2 implementieren

```
1 #include <avr/interrupt.h>
2 static uint16_t zaehler = 0;
3
4 ISR (INT2_vect){
5         zaehler++;
6 }
```

Inhalt

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

```
1  static uint8_t event = 0;
2  ISR (INT0_vect) { event = 1; }
3
4  void main(void) {
5      while(1) {
6          while(event == 0) { /* warte auf Event */ }
7          /* bearbeite Event */
```

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

```
1  volatile static uint8_t event = 0;
2  ISR (INT0_vect) { event = 1; }
3
4  void main(void) {
5      while(1) {
6          while(event == 0) { /* warte auf Event */ }
7          /* bearbeite Event */
```

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		

Lost Update

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3	5	5	-

Lost Update

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3	5	5	-
4	5	4	-

Lost Update

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3	5	5	-
4	5	4	-
8	5	4	5

Lost Update

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3	5	5	-
4	5	4	-
8	5	4	5
9	5	4	6

Lost Update

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3	5	5	-
4	5	4	-
8	5	4	5
9	5	4	6
10	6	4	6

Lost Update

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

Hauptprogramm

```
1  ; volatile uint8_t zaehler;
2  ; C-Anweisung: zaehler--;
3  lds r24, zaehler
4  dec r24
5  sts zaehler, r24
```

Interruptbehandlung

```
7  ; C-Anweisung: zaehler++
8  lds r25, zaehler
9  inc r25
10 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3	5	5	-
4	5	4	-
8	5	4	5
9	5	4	6
10	6	4	6
5	4	4	-

16-Bit-Zugriffe (Read-Write)

- Nebenläufige Nutzung von 16-Bit-Werten (Read-Write)

Hauptprogramm

```
1  volatile uint16_t zaehler;
2
3  ; C-Anweisung: z=zaehler;
4  lds r22, zaehler
5  lds r23, zaehler+1
6  ; Verwendung von z
```

Interruptbehandlung

```
8  ; C-Anweisung: zaehler++
9  lds r24, zaehler
10 lds r25, zaehler+1
11 adiw r24,1
12 sts zaehler+1, r25
13 sts zaehler, r24
```

Zeile	zaehler	zaehler (in r22 & r23)
-	0x00ff	

16-Bit-Zugriffe (Read-Write)

- Nebenläufige Nutzung von 16-Bit-Werten (Read-Write)

Hauptprogramm

```
1  volatile uint16_t zaehler;
2
3  ; C-Anweisung: z=zaehler;
4  lds r22, zaehler
5  lds r23, zaehler+1
6  ; Verwendung von z
```

Interruptbehandlung

```
8  ; C-Anweisung: zaehler++
9  lds r24, zaehler
10 lds r25, zaehler+1
11 adiw r24,1
12 sts zaehler+1, r25
13 sts zaehler, r24
```

Zeile	zaehler	zaehler (in r22 & r23)
-	0x00ff	
4	0x00ff	0x??ff

16-Bit-Zugriffe (Read-Write)

- Nebenläufige Nutzung von 16-Bit-Werten (Read-Write)

Hauptprogramm

```
1  volatile uint16_t zaehler;
2
3  ; C-Anweisung: z=zaehler;
4  lds r22, zaehler
5  lds r23, zaehler+1
6  ; Verwendung von z
```

Interruptbehandlung

```
8  ; C-Anweisung: zaehler++
9  lds r24, zaehler
10 lds r25, zaehler+1
11 adiw r24,1
12 sts zaehler+1, r25
13 sts zaehler, r24
```

Zeile	zaehler	zaehler (in r22 & r23)
-	0x00ff	
4	0x00ff	0x??ff
9 - 13	0x0100	0x??ff

16-Bit-Zugriffe (Read-Write)

- Nebenläufige Nutzung von 16-Bit-Werten (Read-Write)

Hauptprogramm

```
1  volatile uint16_t zaehler;
2
3  ; C-Anweisung: z=zaehler;
4  lds r22, zaehler
5  lds r23, zaehler+1
6  ; Verwendung von z
```

Interruptbehandlung

```
8  ; C-Anweisung: zaehler++
9  lds r24, zaehler
10 lds r25, zaehler+1
11 adiw r24,1
12 sts zaehler+1, r25
13 sts zaehler, r24
```

Zeile	zaehler	zaehler (in r22 & r23)
-	0x00ff	
4	0x00ff	0x??ff
9 - 13	0x0100	0x??ff
5 - 6	0x0100	0x01ff

⇒ Abweichung um 255 !

Sperren der Unterbrechungsbehandlung beim AVR

- Viele weitere Nebenläufigkeitsprobleme möglich
  - Nicht-atomare Modifikation von gemeinsamen Daten kann zu Inkonsistenzen führen
  - Problemanalyse durch den Anwendungsprogrammierer
  - Auswahl geeigneter Synchronisationsprimitive
- Lösung hier: Einseitiger Ausschluss durch Sperren der Interrupts
  - Sperrung aller Interrupts (`cli()`, `sei()`)
  - Maskieren einzelner Interrupts (`GICR`-Register)
- Problem: Interrupts während der Sperrung gehen evtl. verloren
  - Kritische Abschnitte sollten so kurz wie möglich gehalten werden

Inhalt

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 ATmega32-Datenblatt, S. 33-37

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
  - verlinkt im Doku-Bereich auf der SPiC-Webseite
- Beispiel

```
1 #include <avr/sleep.h>
2 set_sleep_mode(SLEEP_MODE_IDLE); /* Idle-Modus verwenden */
3 sleep_enable();  /* Sleep-Modus aktivieren */
4 sleep_cpu();     /* Sleep-Modus betreten */
5 sleep_disable(); /* Empfohlen: Sleep-Modus danach deaktivieren */
```

Lost Wakeup

- Dornröschenschlaf

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

Hauptprogramm

```
1  sleep_enable();
2  event = 0;
3
4  while( !event ) {
5
6      sleep_cpu();
7
8  }
9
10 sleep_disable();
```

Interruptbehandlung

```
11 ISR(TIMER1_COMPA_vect) {
12     event = 1;
13 }
```

Lost Wakeup

- Dornröschenschlaf

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

Hauptprogramm

```
1  sleep_enable();
2  event = 0;
3
4  while( !event ) {
5          ↯ Interrupt ↯
6      sleep_cpu();
7
8  }
9
10 sleep_disable();
```

Interruptbehandlung

```
11 ISR(TIMER1_COMPA_vect) {
12     event = 1;
13 }
```

Lost Wakeup

- Dornröschenschlaf

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

⇒ **Lösung:** Interrupts während des kritischen Abschnitts sperren

Hauptprogramm

```
1  sleep_enable();
2  event = 0;
3  cli();
4  while( !event ) {
5      sei();
6      sleep_cpu();
7      cli();
8  }
9  sei();
10 sleep_disable();
```

Interruptbehandlung

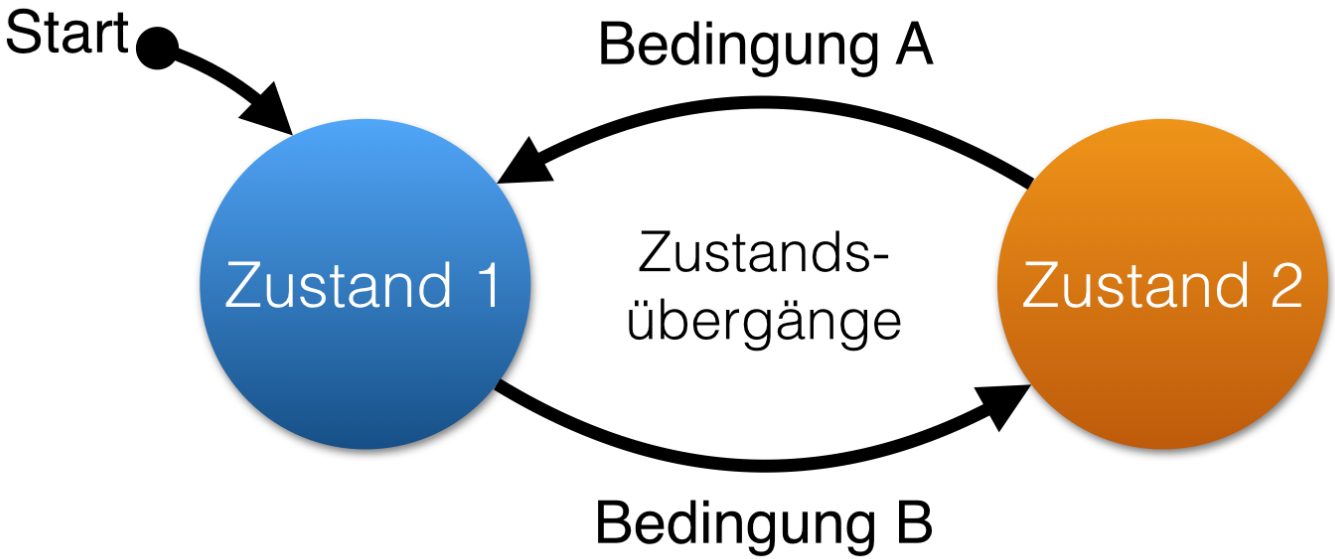
```
11 ISR(TIMER1_COMPA_vect) {
12     event = 1;
13 }
```

Inhalt

Aufgabe 5: Ampel

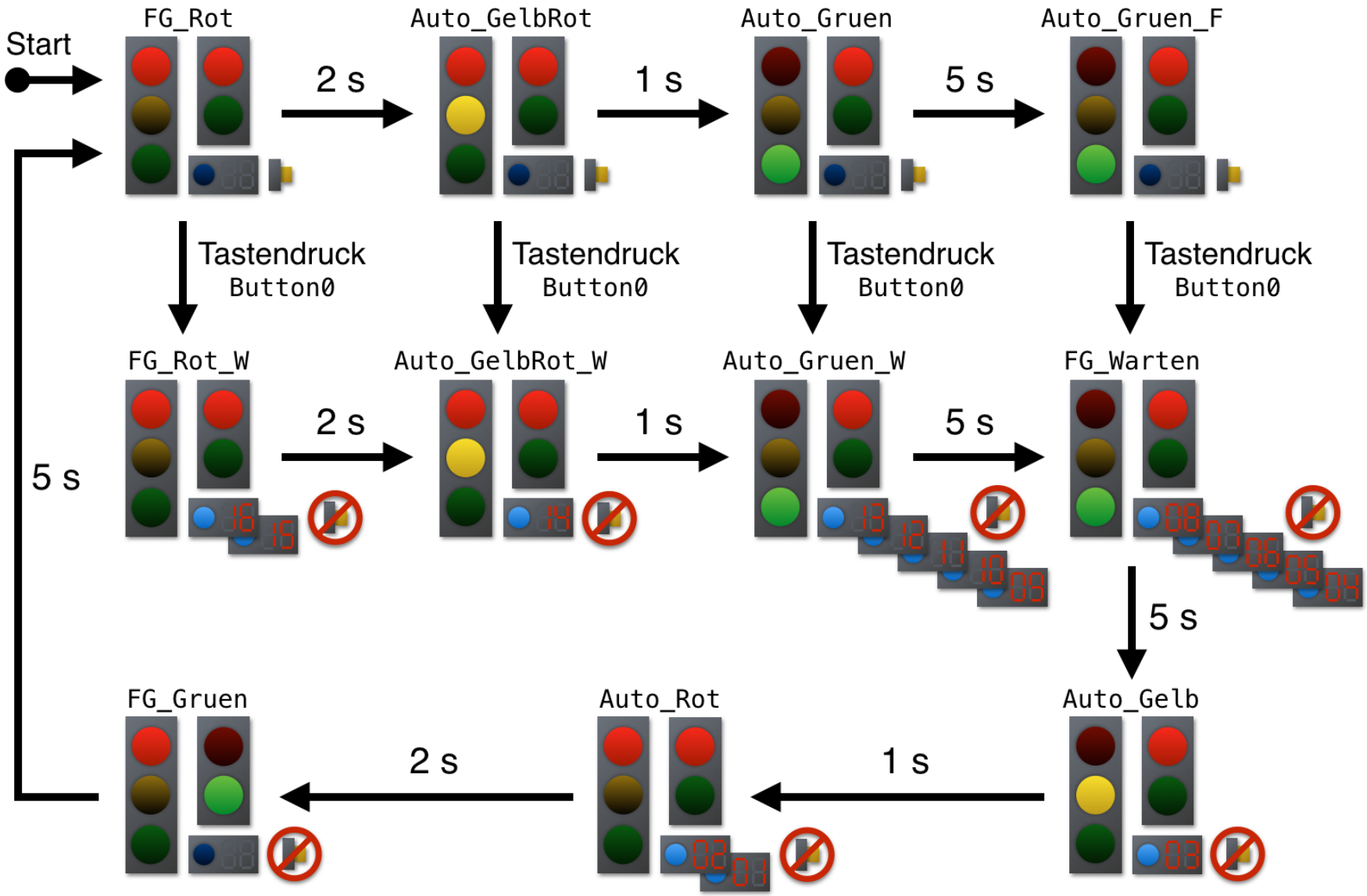
- Implementierung einer (Fußgänger-)Ampel mit Wartezeitanzeige
- Ablauf (exakt) nach Aufgabenbeschreibung (Musterlösung verfügbar)
- Hinweise
  - Tastendrücke und Alarme als Events (kein aktives Warten)
  - In Sleep-Modus wechseln, wenn keine Events zu bearbeiten sind
  - nur eine Stelle zum Warten auf Events (`sleep-Loop`)
  - Deaktiveren des Tasters durch Ignorieren des Events
    (Änderung der Interrupt-Konfiguration ist nicht notwendig)
  - Abbildung auf Zustandsmaschine sinnvoll
  - Verwendung von `volatile` begründen

Zustandsmaschinen



- **Zustände** mit bestimmten Eigenschaften; definierter Initialzustand
- **Zustandswechsel** in Abhängigkeit von definierten Bedingungen

Ampel als Zustandsmaschine



Festlegen von Zuständen: enum-Typen

- Festlegung durch Zahlen ist fehleranfällig
  - Schwer zu merken
  - Wertebereich nur bedingt einschränkbar
- Besser `enum`:

```
1 enum state { STATE_RED, STATE_YELLOW, STATE_GREEN };
2
3 enum state my_state = STATE_RED;
```

- Mit `typedef` noch lesbarer:

```
1 typedef enum { STATE_RED, STATE_YELLOW, STATE_GREEN } state;
2
3 state my_state = STATE_RED;
```

Zustandsabfragen: switch-case-Anweisung

```
1  switch ( my_state ) {
2  case STATE_RED:
3    ...
4    break;
5  case STATE_YELLOW:
6    ...
7    break;
8  case STATE_GREEN:
9    ...
10   break;
11 default:
12   // maybe invalid state
13   ...
14 }
```

- Vermeidung von `if-else`-Kaskaden
- `switch`-Ausdruck muss eine Zahl sein (besser ein `enum`-Typ)
- `break`-Anweisung nicht vergessen!
- Ideal für die Abarbeitung von Systemen mit verschiedenen Zuständen

⇒ Implementierung von Zustandsmaschinen

Hinweise zur Verwendung des Timers

Alarme registrieren

```
1  typedef void(* alarmcallback_t )(void);
2
3  ALARM * sb_timer_setAlarm (alarmcallback_t callback,
4                             uint16_t alarmtime, uint16_t cycle);
```

- Es können "beliebig" viele Alarme registriert werden
- Handler wird im Interrupt-Kontext ausgeführt ($\rightsquigarrow$ gesperrte Interrupts)
- Zeiger & Funktionszeiger werden in der nächsten Übung behandelt

Alarme beenden

```
1  int8_t sb_timer_cancelAlarm (ALARM *alrm);
```

- Single-Shot Alarme (`cycle = 0`) dürfen nur abgebrochen werden, **bevor** sie ausgelöst haben (Nebenläufigkeit!)

Inhalt

Hands-on A: Einfacher Interrupt-Zähler

- Zählen der Tastendrücke an Taster 0 (`PD2`)
- Erkennung der Tastendrücke mit Hilfe von Interrupts
- Ausgabe der Anzahl über 7-Segment Anzeige
- CPU in den Schlafmodus versetzen, wenn **Anzahl geradzahlig**
- "Standby"-LED leuchtet während dem Schlafen
- Hinweise:
  - Erkennung der Tastendrücke **ohne** Verwendung der `libspicboard`
  - `PD2` ist der Eingang von `INT0`
  - `PD7` ist der Ausgang für die "Standby"-LED
  - Interrupt bei fallender Flanke:
    - `MCUCR(ISC00) = 0`
    - `MCUCR(ISC01) = 1`

Hands-on B: Erweiterter Interrupt-Zähler

- Zählen der Tastendrücke an Taster 0
- vorübergehendes Aktivieren der Anzeige durch Drücken von Taster 1
  - Deaktivieren der Anzeige nach 1 - 10 Sekunden (einstellbar über Potentiometer)
  - Ausgabe über 7-Segment Anzeige und LEDs (Hunderterstelle)
  - bei Verlassen des anzeigbaren Wertebereichs Zähler zurücksetzen
- Erkennung der Tastendrücke ohne Polling
  - Interrupts verwenden (fallende Flanke)
  - CPU in den Schlafmodus versetzen, wenn nichts zu tun ist
- Hinweise:
  - Erkennung der Tastendrücke **ohne** Verwendung der `libspicboard`
  - Interrupts nur kurzzeitig sperren; Interrupt Handler kurz halten
  - auf richtige Synchronisation achten
  - Informationen zu Atmega32 und relevante Register siehe Datenblatt