

# Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier  
(Lehrstuhl Informatik 4)

## Übung 3



Wintersemester 2016/2017



## Inhalt

---

### Beliebte Fehler

- Sichtbarkeit & Lebensdauer
- Verwendung falscher Datentypen
- Typdefs & Enums
- volatile Verwendung

### Module

- Schnittstellenbeschreibung
- Ablauf vom Quellcode zum laufenden Programm
- Initialisierung eines Moduls

### Ein- & Ausgabe über Pins

- Active-high & Active-low
- Konfiguration der Pins

### Aufgabe 3: 7-Segment Modul

- Funktionsweise
- Implementierung
- Testen des Moduls



## Beliebte Fehler

Sichtbarkeit & Lebensdauer  
Verwendung falscher Datentypen  
Typdefs & Enums  
volatile Verwendung

## Module

## Ein- & Ausgabe über Pins

## Aufgabe 3: 7-Segment Modul



# Sichtbarkeit & Lebensdauer

| SB Sichtbarkeit<br>LD Lebensdauer |        | nicht static                                | static                        |
|-----------------------------------|--------|---------------------------------------------|-------------------------------|
| Variablen                         | lokal  | Block<br>SB<br>auto Variable<br>LD<br>Block | Block<br>SB<br>LD<br>Programm |
|                                   | global | Programm<br>SB<br>LD<br>Programm            | Modul<br>SB<br>LD<br>Programm |
| Funktionen                        |        | SB<br>Programm                              | SB<br>Modul                   |

- Lokale Variable, nicht static = auto Variable
  - automatisch allokiert & freigegeben
- Funktionen als static, wenn kein Export notwendig



```
1 static uint8_t state; // global static
2 uint8_t event_counter; // global
3
4 void main(void) {
5     ...
6 }
7
8 static void f(uint8_t a) {
9     static uint8_t call_counter = 0; // local static
10    uint8_t num_leds; // local (auto)
11    ...
12 }
```

- Sichtbarkeit/Gültigkeit möglichst weit **einschränken**
- Globale Variable  $\neq$  lokale Variable in `f()`
- Globale static Variablen: Sichtbarkeit auf Modul beschränken
- ⇒ static bei Funktionen und globalen Variablen verwenden, wo möglich



## Verwendung falscher Datentypen

- Die Größe von `int` ist nicht genau definiert (ATmega32: 16 bit)  
⇒ Gerade auf  $\mu$ C führt dies zu Fehlern und/oder langsameren Code
- Für die Übung:
  - Verwendung von `int` ist ein “Fehler”
  - Stattdessen: Verwendung der in der `stdint.h` definierten Typen: `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, etc.
- Wertebereich:
  - `limits.h`: `INT8_MAX`, `INT8_MIN`, ...
- Speicherplatz ist sehr teuer auf  $\mu$ C
- ⇒ Nur so viel Speicher verwenden, wie tatsächlich benötigt wird!



```
1 #define PB3 3
2 typedef enum { BUTTON0 = 4, BUTTON1 = 8
3 } BUTTON;
4 #define MAX_COUNTER 900
5 ...
6 void main(void) {
7     ...
8     PORTB |= (1 << PB3); // nicht (1 << 3)
9     ...
10    BUTTONEVENT old, new; // nicht uint8_t old, new;
11    ...
12    // Deklaration: BUTTONEVENT sb_button_getState(BUTTON btn);
13    old = sb_button_getState(BUTTON0); // nicht sb_button_getState(4)
14    ...
15 }
```

- Vordefinierte Typen verwenden
- Explizite Zahlenwerte nur verwenden, wenn notwendig



## volatile Verwendung

```
1 static volatile int event_counter;
2 ...
3 void main(void) {
4     int num_leds = 0;
5     ...
6 }
7
```

- volatile verwenden um Optimierungen des Compilers zu verhindern
- Nur bei nebenläufigen Ausführungsfäden notwendig
- Wert aus Register wird in den Speicher zurück geschrieben



Beliebte Fehler

Module

Schnittstellenbeschreibung

Ablauf vom Quellcode zum laufenden Programm

Initialisierung eines Moduls

Ein- & Ausgabe über Pins

Aufgabe 3: 7-Segment Modul



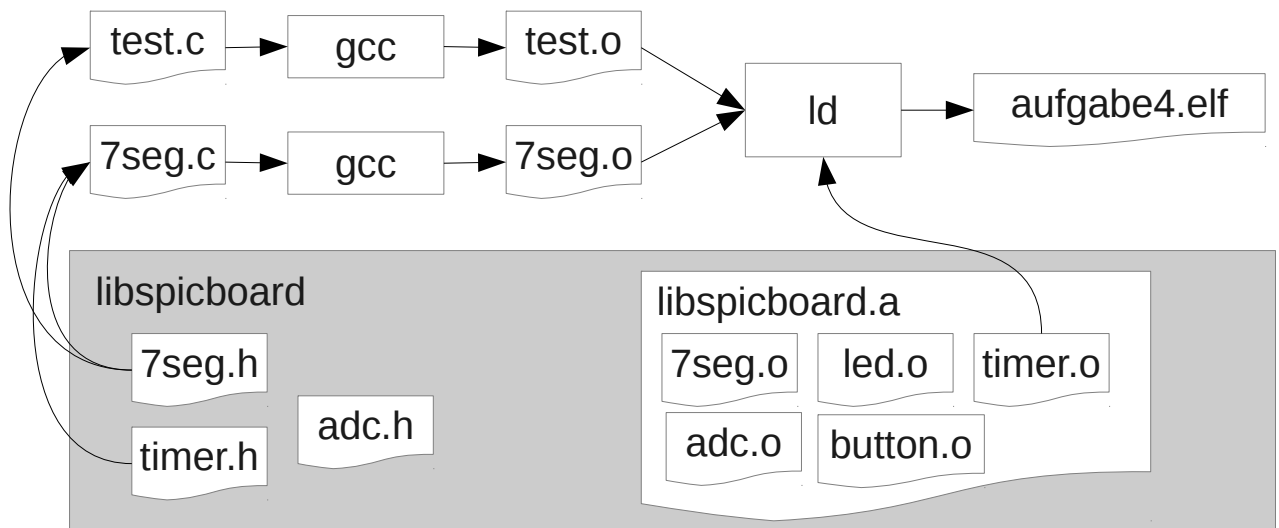
## Schnittstellenbeschreibung

- Erstellen einer .h-Datei (Konvention: gleicher Name wie .c-Datei)

```
1  #ifndef LED_H
2  #define LED_H
3  /* fixed-width Datentypen einbinden (im Header verwendet) */
4  #include <stdint.h>
5  /* LED-Typ */
6  typedef enum { RED0=0, YELLOW0=1, GREEN0=2, ... } LED;
7  /* Funktion zum Aktivieren einer bestimmten LED */
8  uint8_t sb_led_on(LED led);
9  ...
10 #endif
```

- Mehrfachinkludierung (evtl. Zyklen!) vermeiden ~> **Include-Guard**
  - durch Definition und Abfrage eines Präprozessormakros
  - Konvention: das Makro hat den Namen der .h-Datei, '.' ersetzt durch '\_'
  - Der Inhalt wird nur eingebunden, wenn das Makro noch nicht definiert ist
- **Vorsicht:** flacher Namensraum ~> Wahl möglichst eindeutiger Namen





1. Präprozessor
2. Compiler
3. Linker
4. Programmierer/Flasher



## Initialisierung eines Moduls

- Module müssen Initialisierung durchführen (z.B. Portkonfiguration)
  - z.B. in Java mit Klassenkonstruktoren möglich
  - C kennt kein solches Konzept
- Workaround: Modul muss bei erstem Aufruf einer seiner Funktionen ggf. die Initialisierung durchführen
  - muss sich merken, ob die Initialisierung schon erfolgt ist
  - Mehrfachinitialisierung vermeiden

```
1 static uint8_t initDone = 0;
2 // alternativ: lokale static Variable in init()
3
4 static void init(void) { ... }
5 void mod_func(void) {
6     if(initDone == 0) {
7         initDone = 1;
8         init();
9     }
10     ....
```

Initialisierung darf nicht mit anderen Modulen in Konflikt stehen!



Beliebte Fehler

Module

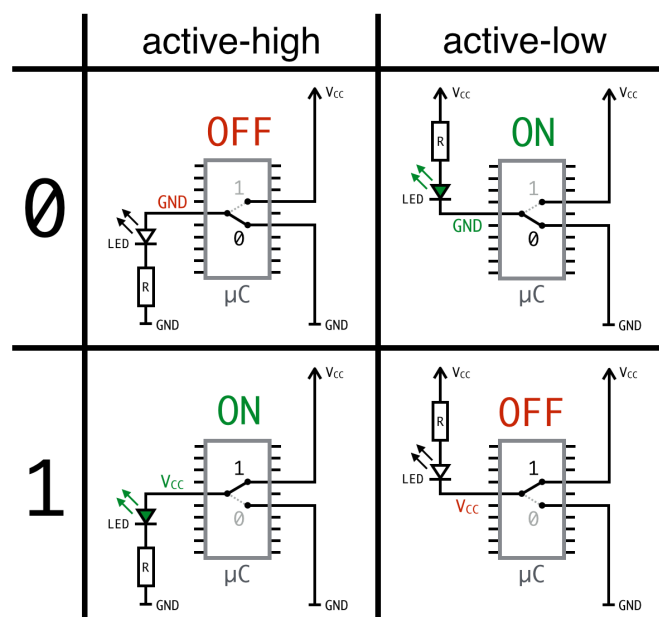
Ein- & Ausgabe über Pins  
Active-high & Active-low  
Konfiguration der Pins

Aufgabe 3: 7-Segment Modul



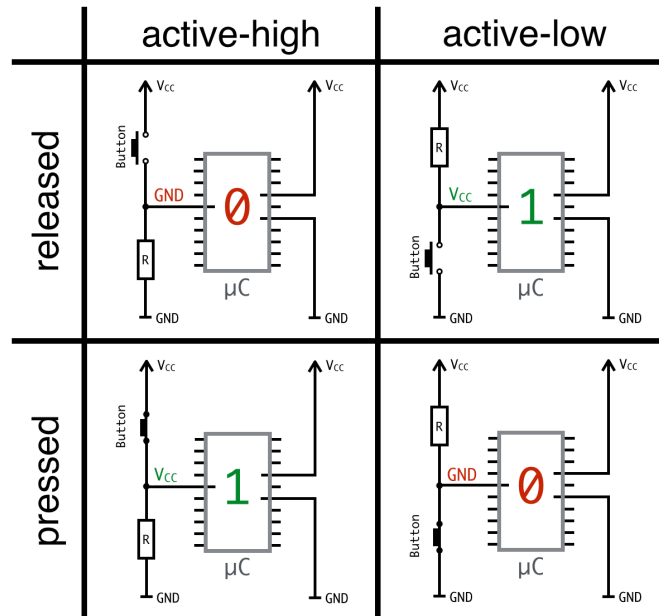
## Ausgang: active-high & active-low

- 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

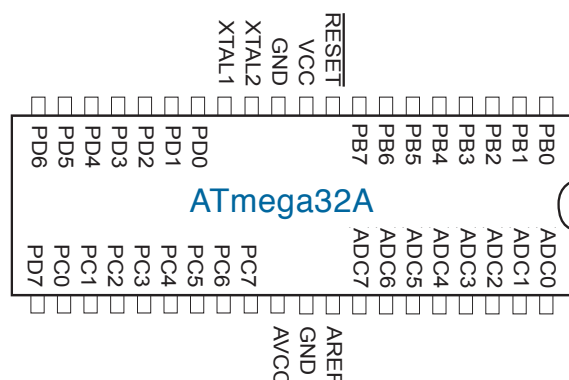


# Eingang: active-high & active-low

- 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)
- interner pull-up-Widerstand (im ATmega32) konfigurierbar



## Konfiguration der Pins



- Jeder I/O-Port des AVR-µC wird durch drei 8-bit Register gesteuert:
  - Datenrichtungsregister (DDRx = data direction register)
  - Datenregister (PORTx = port output register)
  - Port Eingabe Register (PINx = port input register, nur-lesbar)
- Jedem Anschluss-Pin ist ein Bit in jedem der 3 Register zugeordnet



- DDRx: hier konfiguriert man Pin i von Port x als Ein- oder Ausgang
  - Bit i = 1 → Pin i als Ausgang verwenden
  - Bit i = 0 → Pin i als Eingang verwenden
- PORTx: Auswirkung **abhängig von DDRx**:
  - ist Pin i **als Ausgang konfiguriert**, so steuert Bit i im PORTx Register ob am Pin i ein high- oder ein low-Pegel erzeugt werden soll
    - Bit i = 1 → high-Pegel an Pin i
    - Bit i = 0 → low-Pegel an Pin i
  - ist Pin i **als Eingang konfiguriert**, so kann man einen internen pull-up-Widerstand aktivieren
    - Bit i = 1 → pull-up-Widerstand an Pin i (Pegel wird auf high gezogen)
    - Bit i = 0 → Pin i als tri-state konfiguriert
- PINx: Bit i gibt aktuellen Wert des Pin i von Port x an (nur lesbar)



## Beispiel: Initialisierung eines Ports

- Pin 3 von Port B (PB3) als Ausgang konfigurieren und PB3 auf Vcc schalten:

```
1 DDRB |= (1 << PB3); /* =0x08; PB3 als Ausgang nutzen... */
2 PORTB |= (1 << PB3); /* ...und auf 1 (=high) setzen */
```

- Pin 2 von Port D (PD2) als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
1 DDRD &= ~(1 << PD2); /* PD2 als Eingang nutzen... */
2 PORTD |= (1 << PD2); /* pull-up-Widerstand aktivieren */
3 if ( (PIND & (1 << PD2)) == 0) { /* den Zustand auslesen */
4     /* ein low Pegel liegt an, der Taster ist gedrückt */
5 }
```

- Die Initialisierung der Hardware wird in der Regel einmalig zum Programmstart durchgeführt



Beliebte Fehler

Module

Ein- & Ausgabe über Pins

## Aufgabe 3: 7-Segment Modul

Funktionsweise

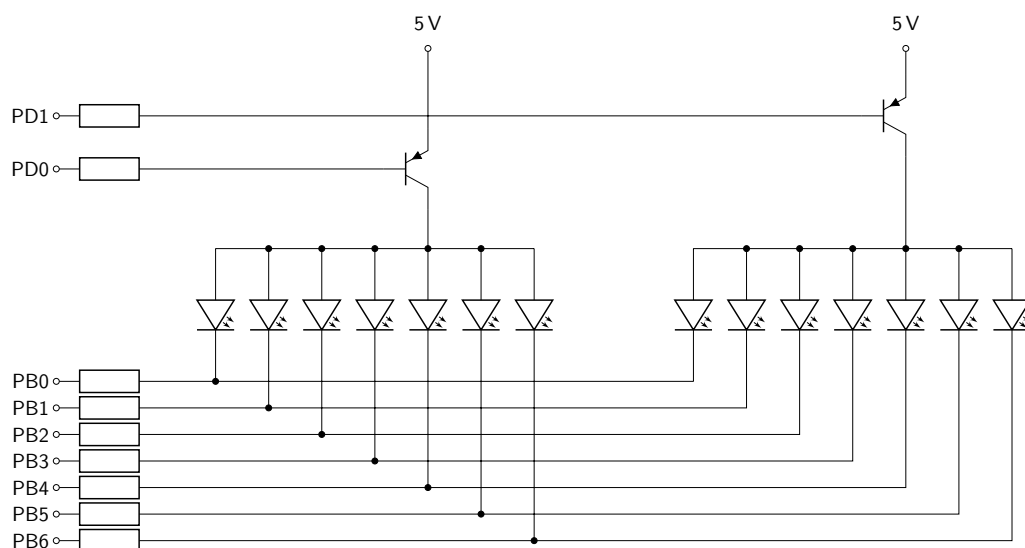
Implementierung

Testen des Moduls



## Funktionsweise der 7-Segment Anzeige

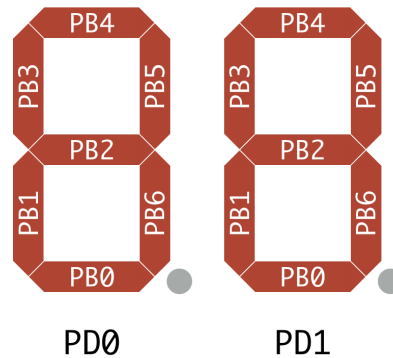
### ■ Schaltung der Siebensegmentanzeige



- PB7 geht zum ISP und kann als nicht angeschlossen betrachtet werden

- Durch alternierende Aktivierung der beiden Anzeigen erscheinen beide aktiv





- Gleiches Verhalten wie das Original
  - Ausnahme: `sb_7seg_showStr()` muss nicht implementiert werden
  - Beschreibung:  
[http://www4.cs.fau.de/Lehre/WS16/V\\_SPIC/Uebung/doc](http://www4.cs.fau.de/Lehre/WS16/V_SPIC/Uebung/doc)
- Timer-Modul ermöglicht es, dass zu bestimmten Zeitpunkten eine Funktion aufgerufen wird
  - Die Funktion muss dem Modul als Pointer übergeben werden ~ 13-19
  - Der Aufruf findet im Interrupt-Kontext statt
    - ⇒ Die aufgerufene Funktion sollte möglichst kurz sein



## 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 (~ gesperrte Interrupts)
- Zeiger & Funktionszeiger werden in der nächsten Übung behandelt

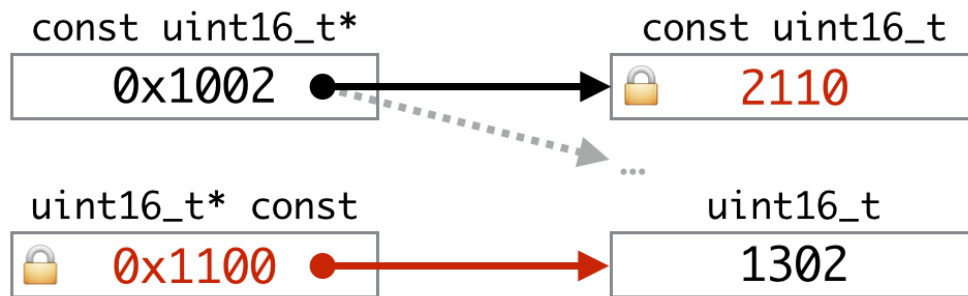
### ■ Alarme beenden

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

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



- `const uint8_t*`
  - ein Pointer auf einen `uint8_t`-**Wert**, der konstant ist
  - Wert nicht über den Pointer veränderbar
- `uint8_t* const`
  - ein **konstanter Pointer** auf einen (beliebigen) `uint8_t`-Wert
  - Pointer darf nicht mehr auf eine andere Speicheradresse zeigen



## Display Mapping

- Port und Pin Definitionen (in `avr/io.h`)

```
1 #define PORTD (* (volatile uint8_t*)0x2B)
2 ...
3 #define PD0    0
4 ...
```

- Adressoperator: `&`
- Dereferenzierungsoperator: `*`
- Display Mapping (Ziffer → LEDs):

```
1 static const uint8_t leds[] = {0x84, 0x9F, 0xC8, 0x8A,
2   0x93, 0xA2, 0xA0, 0x8F, 0x80, 0x82,
3   0x81, 0xB0, 0xE4, 0x98, 0xE0, 0xE1};
4
```



- Projekt wie gehabt anlegen
  - Initiale Quelldatei: `test.c`
  - Dann weitere Quelldatei `7seg.c` hinzufügen
- Wenn nun übersetzt wird, werden die Funktionen aus dem eigenen `7seg-Modul` verwendet
- Andere Teile der Bibliothek werden nach Bedarf hinzugebunden
- Temporäres deaktivieren zum Test der Originalfunktiononen:

```
1 #if 0
2 ....
3 #endif
```

- Sieht der Compiler diese "Kommentare"?



## Testen des Moduls

```
1 void main(void){
2     ...
3     // 1.) Testen bei korrekter Eingabe
4     int8_t result = sb_7seg_showNumber(42);
5     if(result != 0){
6         // Test fehlgeschlagen
7         // Ausgabe z.B. über LEDs
8     }
9
10    // 2.) Testen ungültiger Eingaben
11    ...
12 }
```

- Schnittstellenbeschreibung genau beachten (inkl. Rückgabewerte)
- Testen **aller möglichen Rückgabewerte**
- Fehler wenn Rückgabewert nicht der Spezifikation entspricht

