

# Übungen zu Systemnahe Programmierung in C (SPiC)

Peter Wägemann, Heiko Janker, Moritz Strübe, Rainer Müller  
(Lehrstuhl Informatik 4)



Wintersemester 2014/2015



## Beliebte Fehler

- Globale Variablen

- volatile Verwendung

- Typdefs und Enums

## Bit- und Shiftoperationen

- Bitoperationen

- Shiftoperationen

## LED Modul

- Module

- Konfiguration der Pins

- Hinweise zur Aufgabe



## Beliebte Fehler

- Globale Variablen

- volatile Verwendung

- Typdefs und Enums

Bit- und Shiftoperationen

LED Modul



```
1 int event_counter;  
2 int num_leds;  
3 ...  
4 void main(void) {  
5     ...  
6 }
```

- Sichtbarkeit/Gültigkeit einschränken soweit wie möglich
- Globale Variable  $\neq$  lokale Variable in der main()
- Globale static Variablen: Sichtbarkeit auf Modul beschränken



```
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



# Typdefs und Enums

```
1 #define PD3 3
2 void f();
3 typedef enum { BUTTON0 = 4, BUTTON1 = 8
4 } BUTTON;
5 #define MAX_COUNTER 900
6 ...
7 void main(void) {
8     ...
9     PORTB |= (1 << PB3); // nicht (1 << 3)
10    ...
11    BUTTONEVENT old, new; // nicht uint8_t old, new;
12    ...
13    // Deklaration: BUTTONEVENT sb_button_getState(BUTTON btn);
14    old = sb_button_getState(BUTTON0); // nicht sb_button_getState(4)
15    ...
16    alarmcallback_t my_callback = f;
17    ...
18 }
```

- Vordefinierte Typen verwenden
- Explizite Zahlenwerte nur verwenden wenn es notwendig ist



Beliebte Fehler

Bit- und Shiftoperationen

Bitoperationen

Shiftoperationen

LED Modul



## ■ 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



- Beispiel:

```
uint8_t x = 0x9d;  

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

  
x <<= 2;           

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

  
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 nicht 100% definiert. Im Normalfall(!) werden bei negativen Werten 1er geschiftet.

Beliebte Fehler

Bit- und Shiftoperationen

LED Modul

- Module

- Konfiguration der Pins

- Hinweise zur Aufgabe



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

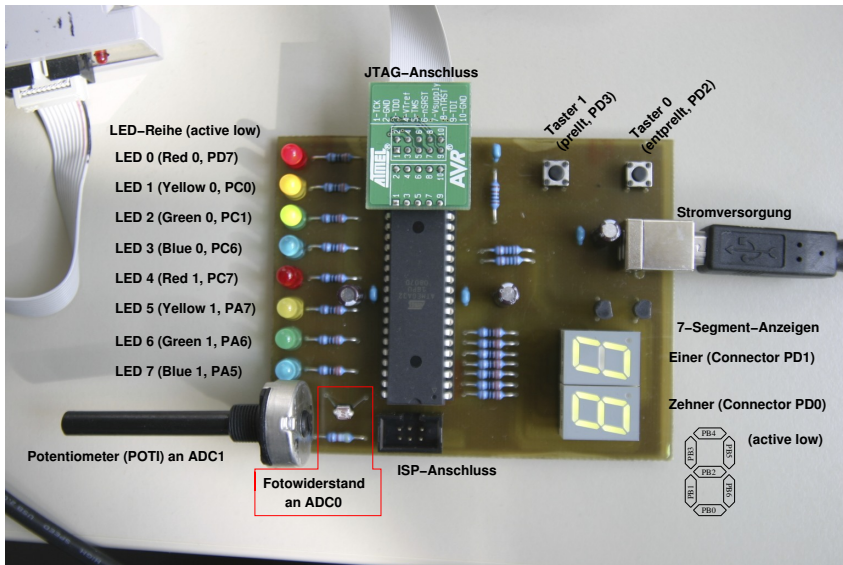
```
1 #ifndef LED_H
2 #define LED_H
3 /* fixed-width Datentypen inkludieren (im Header verwendet) */
4 #include <stdint.h>
5 /* LED-Typ */
6 typedef enum { RED0=0, YELLOW0=1, GREEN0=2, ... } LED;
7 /* Funktion zum Aktivieren einer bestimmten LED */
8 uint8_t sb_led_on(LED led);
9 /* Irgendeine Variable */
10 extern uint8_t einevariable;
11 ...
12 #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



# LED Modul



Das LED-Modul der SPiCboard-Bibliothek selbst implementieren

- Gleiches Verhalten wie das Original
  - Beschreibung:  
[http://www4.cs.fau.de/Lehre/WS14/V\\_SPIC/Uebung/doc](http://www4.cs.fau.de/Lehre/WS14/V_SPIC/Uebung/doc)
- Das eigene Modul dann mit einem Testprogramm linken
- Andere Teile der Bibliothek können für den Test benutzt werden
- LEDs des SPiCboard
- Die Anschlüsse und Namen der einzelnen LEDs können dem Übersichtsbildchen entnommen werden
- Alle LEDs sind **active low**, d.h. leuchten wenn ein Low-Pegel auf dem Pin angelegt wird.
- PD7 = Port D, Pin 7



# Konfiguration der Pins

---

- Jeder I/O-Port des AVR- $\mu$ C 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 einen Pin  $i$  von Port  $x$  als Ein- oder Ausgang
  - Bit  $i = 1 \rightarrow$  Pin  $i$  als Ausgang verwenden
  - Bit  $i = 0 \rightarrow$  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  $i = 1 \rightarrow$  high-Pegel an Pin  $i$
    - Bit  $i = 0 \rightarrow$  low-Pegel an Pin  $i$
  - ist Pin  $i$  als Eingang konfiguriert, so kann man einen internen pull-up-Widerstand aktivieren
    - Bit  $i = 1 \rightarrow$  pull-up-Widerstand an Pin  $i$  (Pegel wird auf high gezogen)
    - Bit  $i = 0 \rightarrow$  Pin  $i$  als tri-state konfiguriert
- **PINx:** Bit  $i$  gibt den 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 auf Vcc schalten:

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

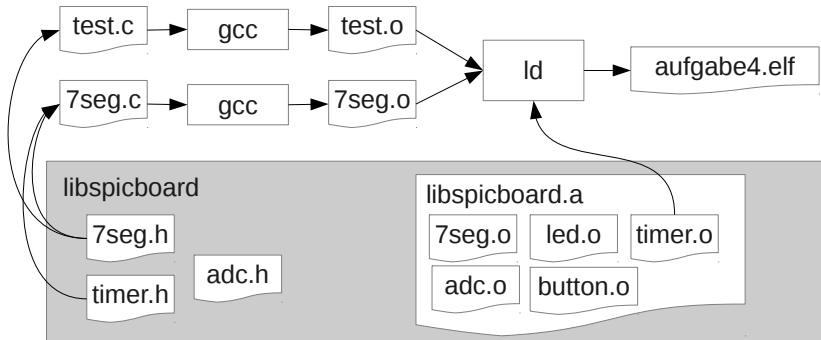
- Pin 2 von Port D (PD2) als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
1 DDRD &= ~(1 << 2); /* PD2 als Eingang nutzen... */
2 PORTD |= (1 << 2); /* pull-up-Widerstand aktivieren */
3 if ( (PIND & (1 << 2)) == 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



# Funktionsweise des Linkers



Aus der elf-Datei wird automatisch die flashbare hex Datei generiert.

# Initialisierung eines Moduls

- Module sind oft zu initialisieren (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
  - Reihenfolge der Initialisierung beachten
  - Mehrfachinitialisierung vermeiden

```
1 static uint8_t initDone = 0;
2 static void init(void) { ... }
3 void mod_func(void) {
4     if(initDone == 0) {
5         initDone = 1;
6         init();
7     }
8     ....
```



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

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

