

Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier
(Lehrstuhl Informatik 4)

Übung 4



Sommersemester 2017



Inhalt

Module

Schnittstellenbeschreibung
Ablauf vom Quellcode zum laufenden Programm
Initialisierung eines Moduls

Ein- & Ausgabe über Pins

Aufgabe 4: LED-Modul

Hands-on: Module, Felder & Zeiger



Inhalt

Module

Schnittstellenbeschreibung
Ablauf vom Quellcode zum laufenden Programm
Initialisierung eines Moduls

Ein- & Ausgabe über Pins

Active-high & Active-low
Konfiguration der Pins

Aufgabe 4: LED-Modul

Hinweise
Testen des Moduls

Hands-on: Module, Felder & Zeiger



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
```

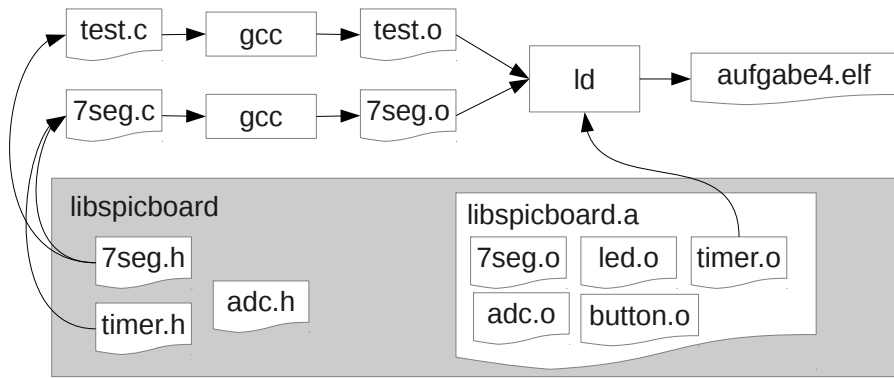
■ Mehrfachinkludierung (evtl. Zyklen!) vermeiden ~> **Include-Guard**

- 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



Ablauf vom Quellcode zum laufenden Programm



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



Inhalt

Module

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

Aufgabe 4: LED-Modul

Hands-on: Module, Felder & Zeiger



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

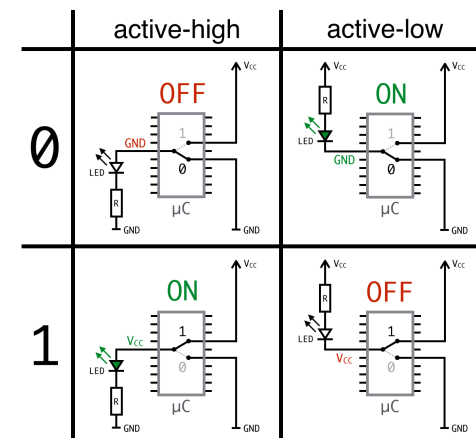
```
1 static uint8_t initDone = 0;
2 // alternativ: lokale static Variable in init()
3
4 static void init(void) { ... }
5 void mod_func(void) {
6     if(initDone == 0) {
7         initDone = 1;
8         init();
9     }
10     ....
```

- Initialisierung darf nicht mit anderen Modulen in Konflikt stehen!

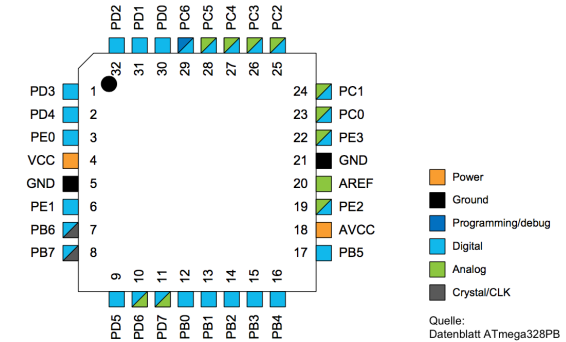
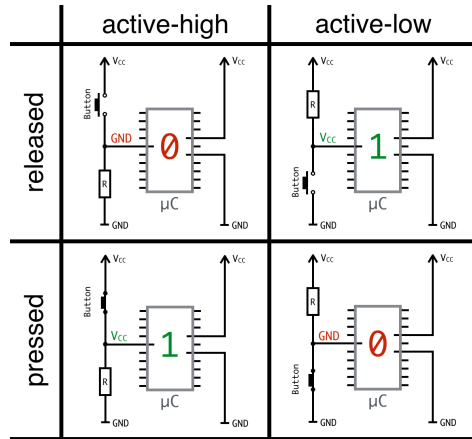


Ausgang: active-high & active-low

- 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 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 ATmega328PB) konfigurierbar



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

I/O-Port-Register

- $DDRx$: hier konfiguriert man 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 aktuellen Wert des Pin i von Port x an (nur lesbar)

Beispiel: Initialisierung eines Ports

- Pin 3 von Port C ($PC3$) als Ausgang konfigurieren und $PC3$ auf V_{CC} schalten:


```
1 DDRC |= (1 << PC3); /* =0x08; PC3 als Ausgang nutzen... */
2 PORTC |= (1 << PC3); /* ...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

Module

Ein- & Ausgabe über Pins

Aufgabe 4: LED-Modul

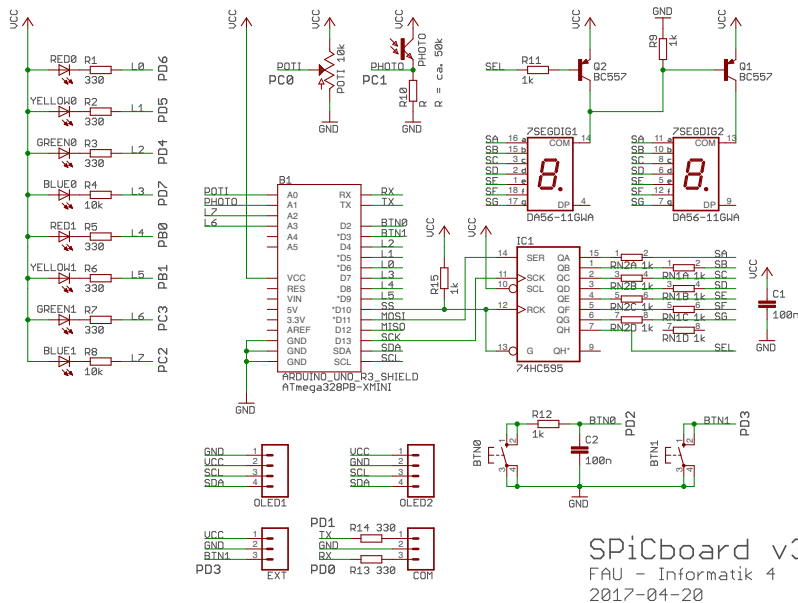
Hinweise

Testen des Moduls

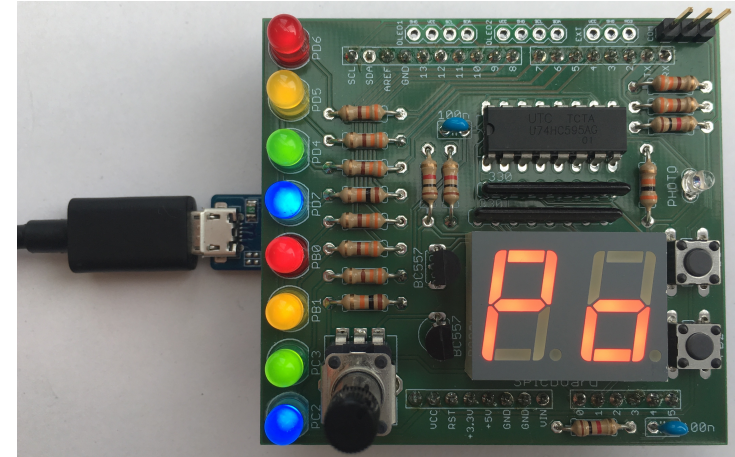
Hands-on: Module, Felder & Zeiger



SPiCboard Schaltplan



LED-Modul – Übersicht



- LED 0 (RED0) ⇒ PD6 ⇒ Port D, Pin 6 ⇒ Bit 6 in PORTD und DDRD
- ...
- LED 7 (BLUE1) ⇒ PC2 ⇒ Port C, Pin 2 ⇒ Bit 2 in PORTC und DDRC



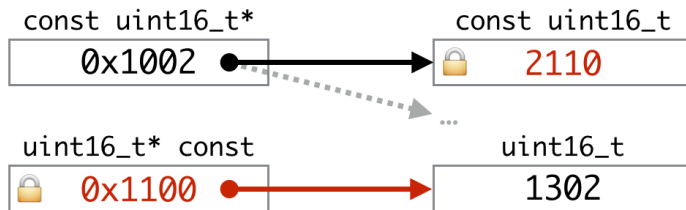
LED-Modul – Aufgabe

- LED-Modul der SPiCboard-Bibliothek selbst implementieren
 - Gleiches Verhalten wie das Original
 - Beschreibung: http://www4.cs.fau.de/Lehre/SS17/V_SPIC/Uebung/doc
- Testen des Moduls
 - Eigenes Modul mit einem Testprogramm (test.c) linken
 - Andere Teile der Bibliothek können für den Test benutzt werden
- LEDs des SPiCboards
 - 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
 - PD6 = Port D, Pin 6



Exkurs: const uint8_t* vs. uint8_t* const

- `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



Port- und Pin-Array

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

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

- Adressoperator: `&`
- Dereferenzierungsoperator: `*`
- Port Array:

```
1 static volatile uint8_t * const ports[] = { &PORTD,
2                                             ...,
3                                             &PORTC };
```

- Pin Array:

```
1 static uint8_t const pins[] = { PD6, ..., PC2 };
```



AVR-Studio Projekteinstellungen

- 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
```

- Sieht der Compiler diese "Kommentare"?



Testen des Moduls

```
1 void main(void){
2 ...
3 // 1.) Testen bei korrekter LED-ID
4 int8_t result = sb_led_on(RED0);
5 if(result != 0){
6     // Test fehlgeschlagen
7     // Ausgabe z.B. auf 7-Segment-Anzeige
8 }
9
10 // 2.) Testen bei ungueltiger LED-ID
11 ...
12 }
```

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



Module

Ein- & Ausgabe über Pins

Aufgabe 4: LED-Modul

Hands-on: Module, Felder & Zeiger



- Statistikmodul und Testprogramm
- Funktionalität des Moduls:
 - Mittelwertbildung

```
uint8_t avgArray(uint16_t *a, size_t s, uint16_t *avg);
```
 - Minimal- und Maximalwertermittlung

```
uint8_t minArray(uint16_t *a, size_t s, uint16_t *min);
uint8_t maxArray(uint16_t *a, size_t s, uint16_t *max);
```
 - Rückgabewert: 0: OK; 1: Fehler
 - Interne Hilfsfunktionen:

```
uint16_t getMin(uint16_t a, uint16_t b);
uint16_t getMax(uint16_t a, uint16_t b);
```
- Vorgehen:
 - Header-Datei mit Modulschnittstelle (und Include-Guards)
 - Implementierung des Moduls (Sichtbarkeit beachten)
 - Testen des Moduls im Hauptprogramm (inkl. Fehlerfälle)

