# Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2025

## Übung 3

Maxim Ritter von Onciul
Eva Dengler

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

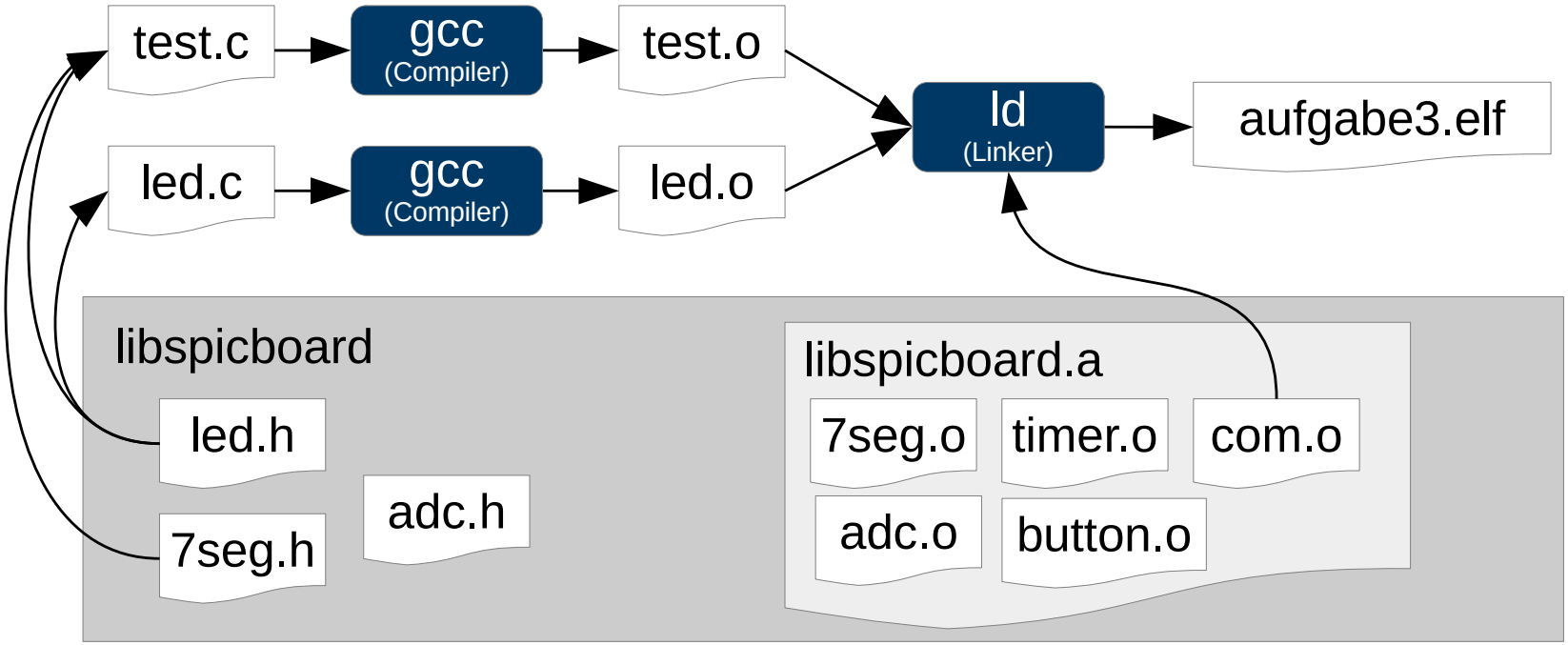
Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

# Vorstellung Aufgabe 1

# Module

# Ablauf vom Quellcode zum laufenden Programm




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

# Schnittstellenbeschreibung (1)

- Header Dateien enthalten die Schnittstelle eines Moduls
  - Funktionsdeklarationen
  - Präprozessormakros
  - Typdefinitionen
- Header Dateien können mehrmals eingebunden werden
  - `led.h` bindet `avr/io.h` ein
  - `button.h` bindet `avr/io.h` ein

  ⇝ Funktionen aus `avr/io.h` mehrmals deklariert
- Mehrfachinkludierung/Zyklen vermeiden ⇝ **Include-Guards**
  - Definition und Abfrage eines Präprozessormakros
  - Konvention: Makro hat den Namen der .h-Datei, '.' ersetzt durch '_'
  - z.B. für `button.h` ⇝ `BUTTON_H`
  - Inhalt nur einbinden, wenn das Makro noch nicht definiert ist
- Vorsicht: Flacher Namensraum ⇝ möglichst eindeutige Namen

# Schnittstellenbeschreibung (2)

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

```
01 #ifndef COM_H
02 #define COM_H
03 /* Fixed-width Datentypen einbinden (im Header verwendet) */
04 #include <stdint.h>
05
06 /* Datentypen */
07 typedef enum {
08     ERROR_NO_STOP_BIT, ERROR_PARITY,
09     ERROR_BUFFER_FULL, ERROR_INVALID_POINTER
10 } COM_ERROR_STATUS;
11
12 /* Funktionen */
13 void sb_com_sendByte(uint8_t data);
14 [...]
15 #endif //COM_H
```

# Implementierung: Kapselung

- Interne Variablen und Hilfsfunktionen nicht Teil der Schnittstelle
- C besitzt einen flachen Namensraum
- Unvorhergesehen Zugriffe können Fehlverhalten auslösen

$\Rightarrow$ Kapselung: Sichtbarkeit & Lebensdauer einschränken

# Implementierung: Sichtbarkeit & Lebensdauer (1)

| Sichtbarkeit und Lebensdauer | **nicht `static`** | **`static`** |
|---|---|---|
| **lokale Variable** | Sichtbarkeit **Block**<br>Lebensdauer **Block** | Sichtbarkeit **Block**<br>Lebensdauer **Programm** |
| **globale Variable** | Sichtbarkeit **Programm**<br>Lebensdauer **Programm** | Sichtbarkeit **Modul**<br>Lebensdauer **Programm** |
| **Funktion** | Sichtbarkeit **Programm** | Sichtbarkeit **Modul** |

- Lokale Variablen, die **nicht** `static` deklariert werden:

  ⇝ `auto` Variable (automatisch allokiert & freigegeben)
- Globale Variablen und Funktionen als `static`, wenn kein Export notwendig

# Implementierung: Sichtbarkeit & Lebensdauer (2)

```
01 static uint8_t state; // global static
02 uint8_t event_counter; // global
03
04 static void f(uint8_t a) {
05   static uint8_t call_counter = 0; // local static
06   uint8_t num_leds; // local (auto)
07   /* ... */
08 }
09
10 void main(void) {
11   /* ... */
12 }
```

- Sichtbarkeit & Lebensdauer möglichst weit **einschränken**

⇝ Wo möglich: `static` für globale Variablen und Funktionen

# Implementierung: Initialisierung eines Moduls (1)

- Module müssen Initialisierung durchführen
  - Zum Beispiel Portkonfiguration
  - **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
- Anlegen einer Init-Variable
  - Aufruf der Init-Funktion bei jedem Funktionsaufruf
  - Init-Variable anfangs 0
  - Nach der Initialisierung auf 1 setzen

# Implementierung: Initialisierung eines Moduls (2)

- `initDone` ist initial `0`
- Wird nach der Initialisierung auf 1 gesetzt

⇝ Initialisierung wird nur einmal durchgeführt

```
01 static void init(void) {
02     static uint8_t initDone = 0;
03     if (initDone == 0) {
04       initDone = 1;
05       ...
06     }
07 }
08
09 void mod_func(void) {
10     init();
11     ...
12 }
```

# General Purpose Input/Output (GPIO)

- Mikrocontroller interagieren mit der Außenwelt
- Neben definierten Protokollen auch beliebige (digitale) Signale
- Viele Pins können sowohl als Eingang als auch als Ausgang konfiguriert werden

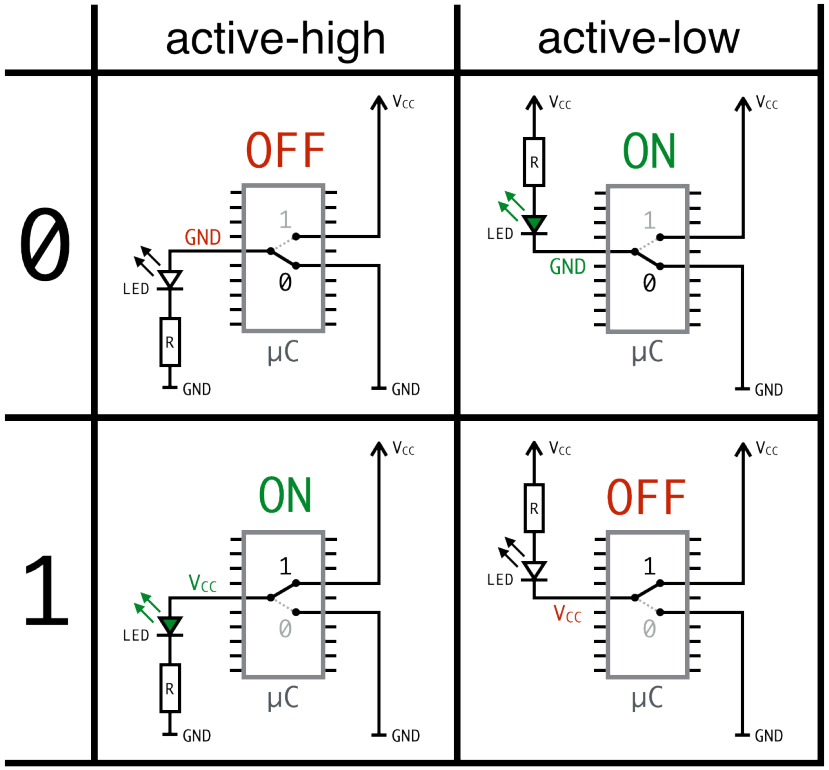
⇝ General Purpose Input/Output (GPIO)

# Ausgang: active-high & active-low

**Ausgang** je nach Beschaltung:

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

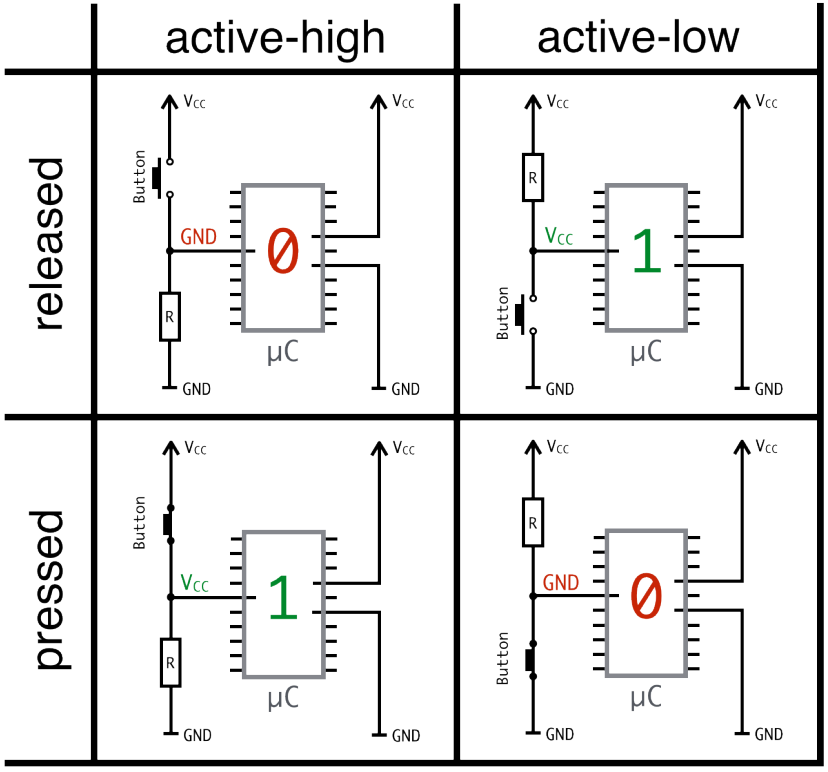
**active-low:** low-Pegel (logisch 0; *GND* am Pin) → LED leuchtet



# Eingang: active-high & active-low

**Eingang** je nach Beschaltung:

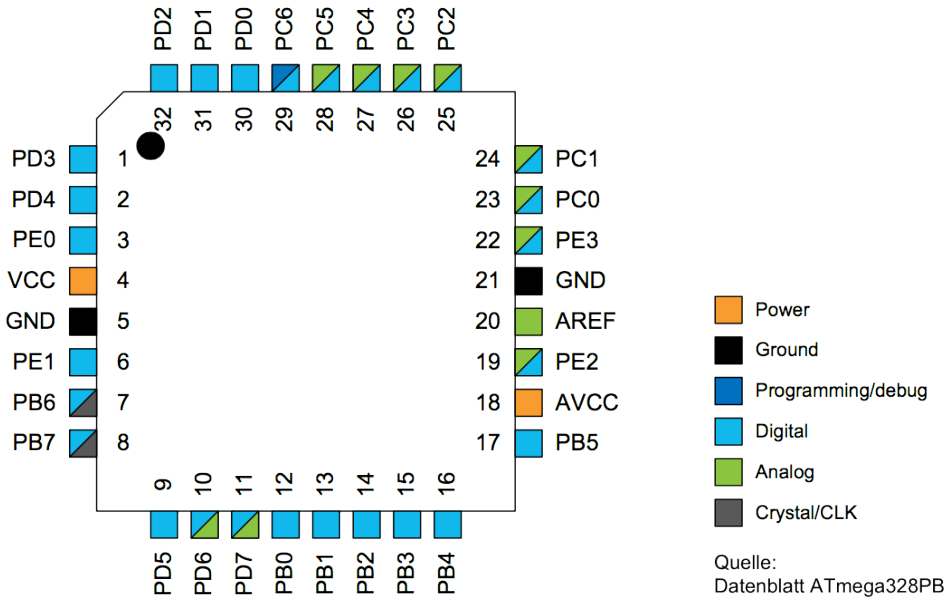
**active-high:** Button gedrückt → high-Pegel (logisch 1; $V_{CC}$ am Pin)
**active-low:** Button gedrückt → low-Pegel (logisch 0; *GND* am Pin)




Eingänge sind hochohming, es muss ein definierter Pegel anliegen

⇝ Pull-down oder (interne) Pull-up Widerstände verwenden

# Konfiguration der Pins




Quelle:
Datenblatt ATmega328PB

- 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 jeweils 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 internen Pull-Up Widerstand an Pin `i`
  - `Bit i = 1` → aktiviert Pull-Up Widerstand für Pin `i`
  - `Bit i = 0` → deaktiviert Pull-Up Widerstand für Pin `i`

**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 internen Pull-Up für PD2
05 PORTD &= ~(1 << PD2); // Deaktiviert internen Pull-Up für PD2
```

# I/O-Port-Register (3)

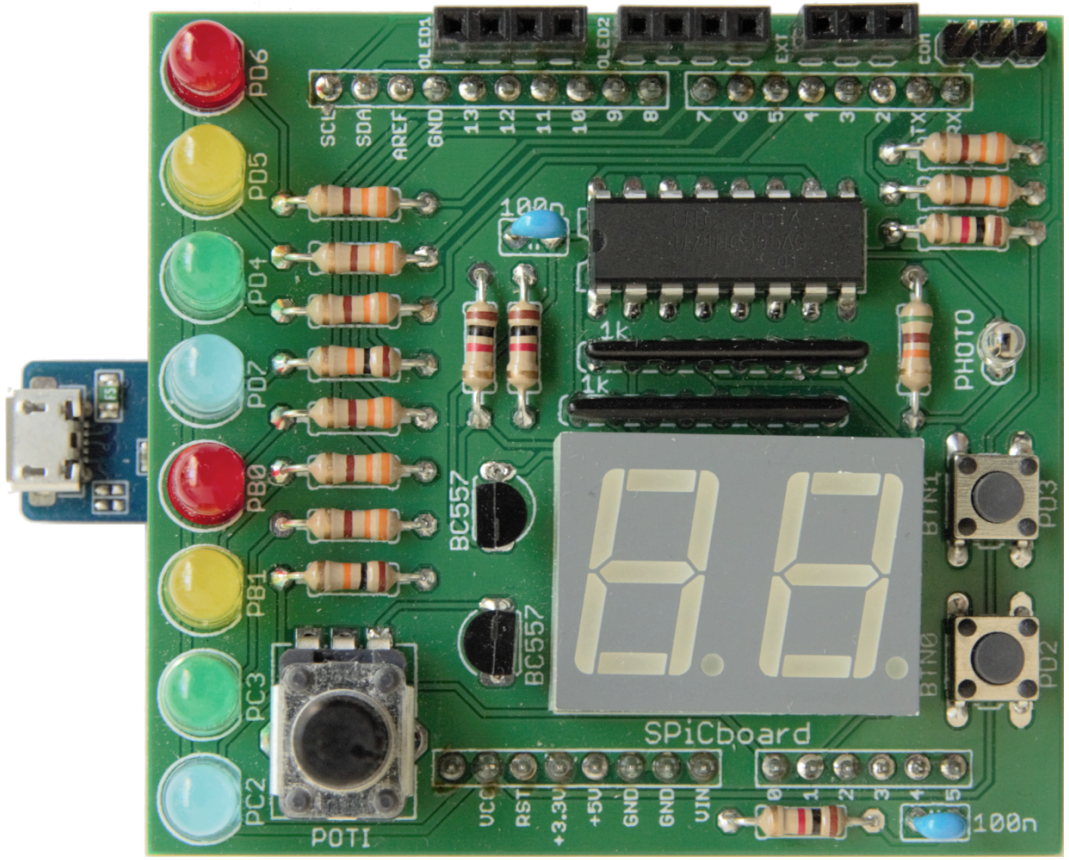
**PINx: Port Input Register (nur lesbar)** aktuellen Wert von Pin `i`

- Wenn **Eingang**: Abrufen was von extern anliegt
- Wenn **Ausgang**: Abrufen ob high oder low ausgegeben wird

**Beispiel:**

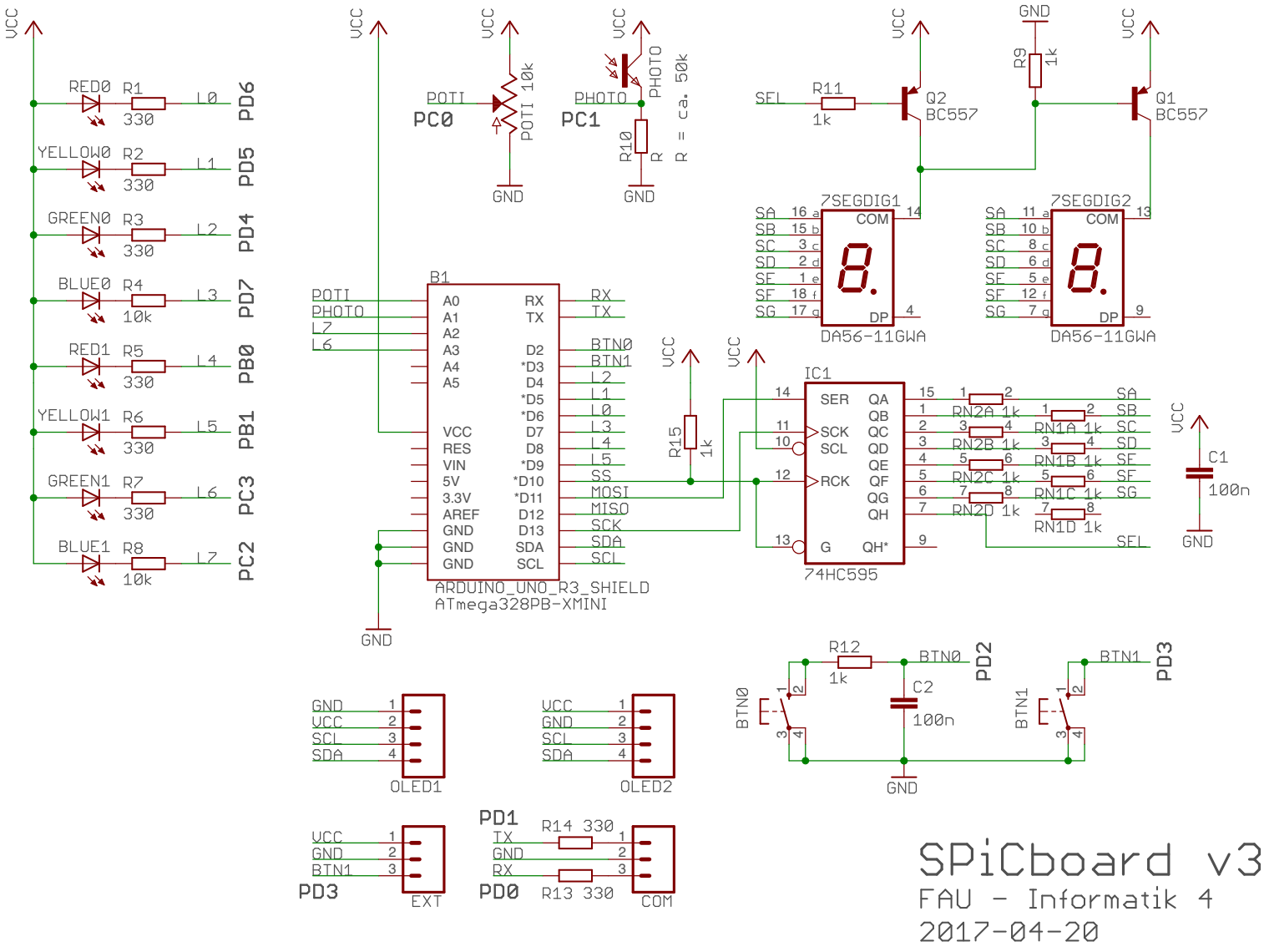
```
01 if((PIND & (1 << PD2)) == 0) { // Testen ob Pin PD2 low ist
02   // low-Pegel --> Button ist gedrückt
03   [...]
04 }
05
06 if((PIND & (1 << PD2)) != 0) { // Testen ob Pin PD2 high ist
07   // high-Pegel --> Button ist nicht gedrückt
08   [...]
09 }
```

# LED-Modul – Übersicht




- LED 0 (REDo) ⇒ 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

# SPiCboard Schaltplan



# LED-Modul – Aufgabe (1)

- LED-Modul der libspicboard selbst implementieren
  - Gleiches Verhalten wie das Original
  - Beschreibung: `https://sys.cs.fau.de/lehre/ss25/spic/uebung/spicboard/libapi/extern/group__LED.html`
- Einbetten von `led.h`
  - Definition der LED-IDs
  - Deklaration der Funktionssignaturen

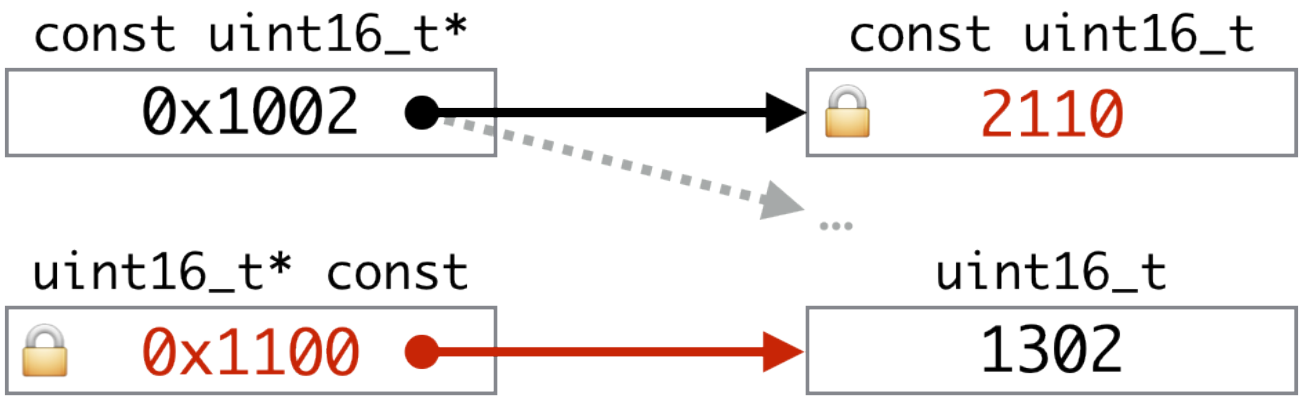
```
01 typedef enum {
02     RED0    = 0, /**< Upper red led    **/
03 ...
04     BLUE1   = 7  /**< Lower blue led   **/
05 } LED;
06
07 int8_t sb_led_on(LED led);
08 ...
09 int8_t sb_led_off(LED led);
```

# LED-Modul – Aufgabe (2)

- Testen des Moduls
  - Eigenes Modul mit einem Testprogramm (`test-led.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 Zeiger auf einen **konstanten** `uint8_t`-**Wert**
  - **Wert** nicht über den Zeiger veränderbar
- `uint8_t* const`
  - ein **konstanter Zeiger** auf einen (beliebigen) `uint8_t`-Wert
  - **Zeiger** darf nicht mehr auf eine andere Speicheradresse zeigen



# Port- und Pin-Array (1)

- Adressoperator: `&`
- Verweisoperator: `*`
- Port und Pin Definitionen (in `avr/io.h`)

```
01 #define PORTD (* (volatile uint8_t *) 0x2B)
02 ...
03 #define PD0     0
04 ...
```

- Makro ersetzt `PORTD` durch `(* (volatile uint8_t *) 0x2B)`
  1. Nimmt die Zahl `0x2B` (Adresse von PORTD)
  2. Castet sie in einen `(volatile uint8_t *)` Zeiger
  3. Dereferenziert Zeiger `*` (⇒PORTD greift auf Registerinhalt zu)
  4. Klammern `(...)` erzwingen richtige Operatorreihenfolge (Vorsicht, Makro!)

# Port- und Pin-Array (2)

- Port Array:

```
01 static volatile uint8_t * const ports[8] = { &PORTD,
02                                              ...,
03                                              &PORTC };
```

  - Macht Dereferenzierung durch Adressoperator wieder rückgängig

    ⇒ In `ports` stehen Adressen als `uint8_t` Zeiger

- Pin Array:

```
01 static uint8_t const pins[8] = { PD6, ..., PC2 };
```

- Zugriff:

```
01 * (ports[0]) &= ~(1 << pins[0]);
```

# Übersetzereinstellungen

- Projekt wie gehabt anlegen
  - Initiale Quelldatei: `test-led.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:

```
01 #if 0
02   ....
03 #endif
```

  ⇒ Sieht der Compiler diese “Kommentare”?

  ⇒ Wie kann der Code wieder einkommentiert werden?

# Testen des Moduls

```
01 void main(void){
02   ...
03   // 1.) Testen bei korrekter LED-ID
04   int8_t result = sb_led_on(RED0);
05   if(result != 0){
06     // Test fehlgeschlagen
07     // Ausgabe z.B. auf 7-Segment-Anzeige
08   }
09   // Einige Sekunden warten
10
11   // 2.) Testen bei ungueltiger LED-ID
12   ...
13 }
```

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

# Hands-on: Statistikmodul

**Screencast:** https://www.video.uni-erlangen.de/clip/id/16328

# Hands-on: Statistikmodul

- Statistikmodul und Testprogramm
- Funktionalität des Moduls (Schnittstelle):

```
01 // Schnittstelle
02 uint8_t avgArray(uint16_t *a, size_t s, uint16_t *avg);
03 uint8_t minArray(uint16_t *a, size_t s, uint16_t *min);
04 uint8_t maxArray(uint16_t *a, size_t s, uint16_t *max);
05
06 // interne Hilfsfunktionen
07 uint16_t getMin(uint16_t a, uint16_t b);
08 uint16_t getMax(uint16_t a, uint16_t b);
```

- Rückgabewert:
  - 0: OK
  - 1: Fehler
- Vorgehen:
  - Header-Datei mit Modulschnittstelle (und Include-Guards)
  - Implementierung des Moduls (Sichtbarkeit beachten)
  - Testen des Moduls im Hauptprogramm (inkl. Fehlerfälle)