

# Exercises in System Level Programming (SLP) – Summer Term 2026

---

## Exercise 3

Arne Vogel

Maxim Ritter von Onciul

Eva Dengler

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



Lehrstuhl für Informatik 4  
Systemsoftware



Friedrich-Alexander-Universität  
Faculty of Engineering

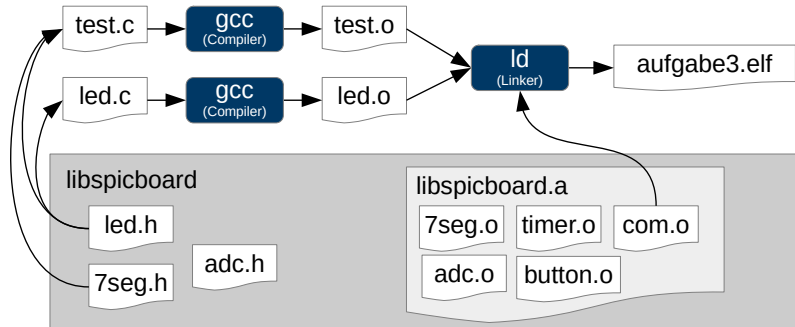
# Presentation Assignment 1

---

# Modules

---

# Overview: From the Source Code to a Program



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



- Header files contain the interface of a module
  - Function declarations
  - Preprocessor macros
  - Type definitions
- Header files can be included multiple times
  - `led.h` includes `avr/io.h`
  - `button.h` includes `avr/io.h`
  - ↪ Functions from `avr/io.h` declared multiple times
- Prevent Multiple inclusions/cycles ↪ **include-guards**
  - Definition and checking of a preprocessor macro
  - Convention: Macro has the same name as .h-file, '.' replaced by '\_'
  - e.g. for `button.h` ↪ `BUTTON_H`
  - File is only included if the macro has not already been defined
- **Attention:** Flat name space ↪ always use unique names



- Creating a .h-file (convention: same name as .c-file)

```
01 #ifndef COM_H
02 #define COM_H
03 /* Include fixed-width data types (used in the header) */
04 #include <stdint.h>
05
06 /* Data Types */
07 typedef enum {
08     ERROR_NO_STOP_BIT, ERROR_PARITY,
09     ERROR_BUFFER_FULL, ERROR_INVALID_POINTER
10 } COM_ERROR_STATUS;
11
12 /* Functions */
13 void sb_com_sendByte(uint8_t data);
14 [...]
15 #endif //COM_H
```



- Internal variables and auxiliary functions not part of the interface
  - C has a flat name space
  - Unexpected accesses can lead to wrong behaviour
- ⇒ Encapsulation: Visibility & life span should be restricted



| Visibility<br>and Life Span | <b>not static</b>                                     | <b>static</b>  |
|-----------------------------|---|--|
| <b>Locale variable</b>      | visibility <b>block</b><br>life span <b>block</b>     | visibility <b>block</b><br>life span <b>program</b>  |
| <b>Global variable</b>      | visibility <b>program</b><br>life span <b>program</b> | visibility <b>module</b><br>life span <b>program</b> |
| <b>Function</b>             | visibility <b>program</b>                             | visibility <b>module</b>                             |

- Local variables that are **not** declared as `static`:
  - ↪ auto variable (automatically allocated & freed)
- Global variables and functions declared as `static`, if no `export` is necessary



```
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 }
```

- Visibility & life span should be chosen as **restricted** as possible
- ↪ If possible: **static** for global variables and functions



- Modules have to perform an initialization
  - For example: Configuring ports
  - **Java:** Possible with class constructors
  - **C:** No such concepts
- *Workaround:* Modules have to initialize themselves upon the first function call
  - Remember completion of initialization
  - Prevent multiple initialization
- Creating an `initDone`-variable
  - Call of the `init` function in each function
  - `initDone`-variable initially set to 0
  - After initialization it is set to 1



- `initDone` is initially set to 0
  - Is set to 1 after initialization
- ~> Initialization only performed once

```
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 }
```

## **In- & Output via Pins**

---

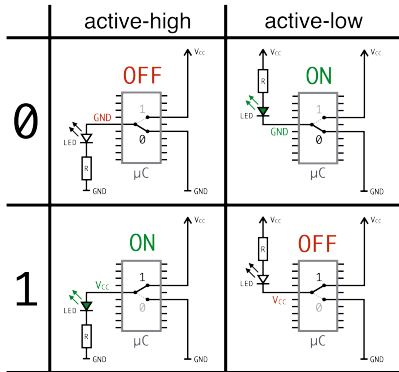


- Microcontroller interact with their environment
  - Besides some predefined protocols: Arbitrary (digital) signals
  - Many pins can be configured as an input or an output
- ↪ General Purpose Input/Output (GPIO)

**Output** dependent on wiring:

**active-high:** high-level (logically 1;  $V_{CC}$  at Pin) → LED is on

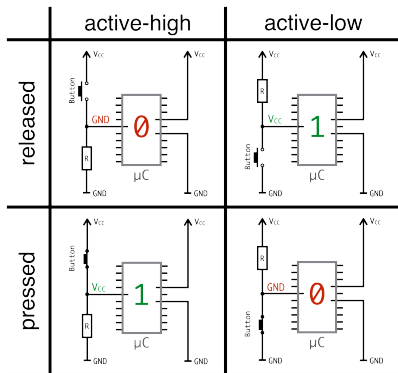
**active-low:** low-level (logically 0;  $GND$  at Pin) → LED is on



Input dependent on wiring:

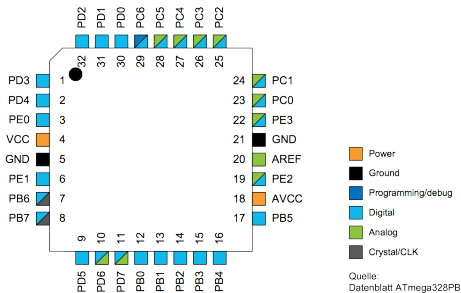
**active-high:** Button pressed  $\rightarrow$  high-level (logically 1;  $V_{CC}$  at Pin)

**active-low:** Button pressed  $\rightarrow$  low-level (logically 0;  $GND$  at Pin)



Inputs are of high impedance, a well defined level has to be present

$\rightarrow$  Use pull-down or (internal) pull-up resistors



- Eight pins are combined to an I/O port for the AVR
- Each I/O port of the AVR is controlled by three 8-bit registers
  - DDRx Data Direction Register
  - PORTx Port Output Register
  - PINx Port Input Register
- Every pin of a port has exactly one bit in each of the three register



**DDRx: Data Direction Register** configures pin  $i$  as an in- or output

- Bit  $i = 1 \rightarrow$  Pin  $i$  used as an output
- Bit  $i = 0 \rightarrow$  Pin  $i$  used as an input

## Example:

```
01 DDRC |= (1 << PC3); // PC3 as output (Pin 3 at Port C)
02 DDRD &= ~(1 << PD2); // PD2 as input (Pin 2 at Port D)
```



**PORTx: Port Output Register** depends on DDRx register

- If **output**: Sets level to high or low at pin  $i$ 
  - Bit  $i = 1 \rightarrow$  high-level at pin  $i$
  - Bit  $i = 0 \rightarrow$  low-level at pin  $i$
- If **input**: Sets the state of the internal pull-up resistor at pin  $i$ 
  - Bit  $i = 1 \rightarrow$  activates pull-up resistor for pin  $i$
  - Bit  $i = 0 \rightarrow$  deactivates pull-up resistor for pin  $i$

## Example:

```
01 PORTC |= (1 << PC3); // Pulls PC3 to high (LED off)
02 PORTC &= ~(1 << PC3); // Pulls PC3 to low (LED on)
03
04 PORTD |= (1 << PD2); // Activates internal pull up for PD2
05 PORTD &= ~(1 << PD2); // Deactivates internal pull up for PD2
```



**PINx: Port Input Register (read only)** current value of pin *i*

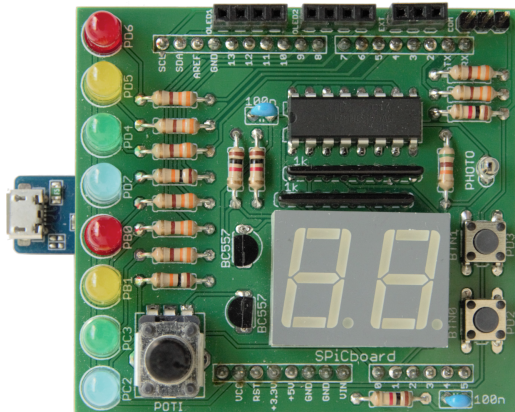
- If **input**: poll what level is set from outside
- If **output**: poll whether high or low is put out

**Example:**

```
01 if((PIND & (1 << PD2)) == 0) { // Testing whether Pin PD2 is low
02     // low-level --> button is pressed
03     [...]
04 }
05
06 if((PIND & (1 << PD2)) != 0) { // Testing whether Pin PD2 is high
07     // high-level --> button is not pressed
08     [...]
09 }
```

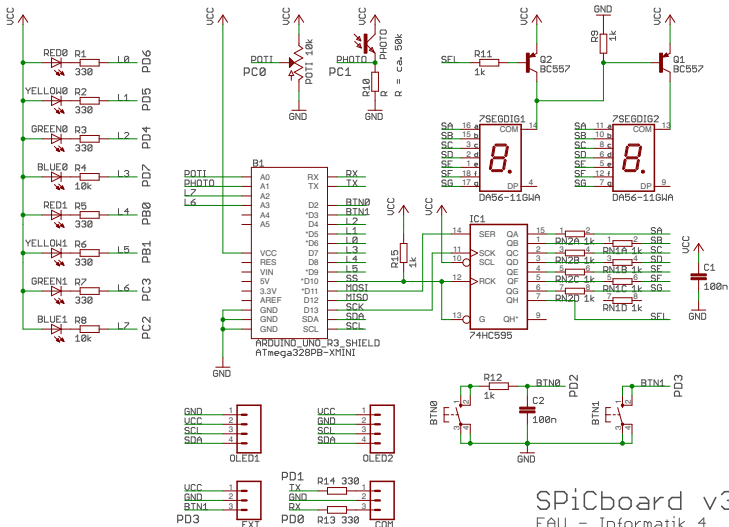
## **Task: LED Module**

---



- LED 0 (REDo) ⇒ PD6 ⇒ Port D, Pin 6 ⇒ Bit 6 at PORTD and DDRD
- ...
- LED 7 (BLUE1) ⇒ PC2 ⇒ Port C, Pin 2 ⇒ Bit 2 at PORTC and DDRC

# SPiCboard Block Circuit Diagram



SPiCboard v3  
FAU - Informatik 4  
2017-04-20



- Implement the LED module of the libspicboard
  - Same behaviour as the original
  - Description:  
[https://sys.cs.fau.de/lehre/ss26/spic/uebung/spicboard/libapi/extern/group\\_\\_LED.html](https://sys.cs.fau.de/lehre/ss26/spic/uebung/spicboard/libapi/extern/group__LED.html)
- Including led.h
  - Definition of LED-IDs
  - Declaration of function signatures

```
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);
```



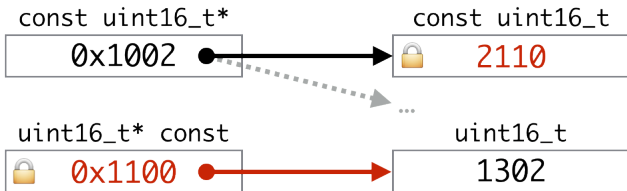
- Testen des Moduls
  - Eigenes Modul mit einem Testprogramm (`test-led.c`) linken
  - Andere Teile der Bibliothek können für den Test benutzt werden
- LEDs des SPiCboards
  - Anschlüsse und Namen der einzelnen LEDs können dem Übersichtsbildchen entnommen werden
  - Alle LEDs sind **active-low**, d.h. leuchten wenn ein low-Pegel auf dem Pin angelegt wird
  - PD6 = Port D, Pin 6



- Testing of the module
  - Link your own module with a test program (`test-led.c`)
  - Other parts of the library can be used for testing
- LEDs of the SPiCboard
  - Connections and names of the single LEDs can be extracted from the overview pictures
  - All LEDs are **active-low**  
i.e. they are switched on if a low-level is applied
  - PD6 = Port D, Pin 6



- `const uint8_t*`
  - Pointer to a **constant** `uint8_t`-value
  - **Value** cannot be modified via the pointer
- `uint8_t* const`
  - **Constant pointer** to an (arbitrary) `uint8_t`-value
  - **Pointer** is not allowed to point at a different memory address





- Address-of operator: &
- Indirection operator: \*
- Definitions for ports and pins (in avr/io.h)

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

- Macro replaces `PORTD` by `(* (volatile uint8_t *) 0x2B)`
  1. Takes the integer `0x2B` (address of `PORTD`)
  2. Casts it into a `(volatile uint8_t *)` pointer
  3. Dereferences pointer `*` ( $\Rightarrow$  `PORTD` is accessing the register contents)
  4. Brackets `(...)` enforce correct order of operations  
(Attention, macro!)



### ■ Port array:

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

- Reverses the dereferencing of the address operator  
⇒ Elements of `ports` are addresses in the form of `uint8_t` pointers

### ■ Pin array:

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

### ■ Access:

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



- Create project as usual
  - Initial source file: `test-led.c`
  - Then add second source file `led.c`
- When compiling, functions from your own module are used
- Additional parts of the library are included if required
- Code can be temporarily deactivated for testing the original functions:

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

- ⇒ Does the compiler see this “comment”?
- ⇒ How can we comment in the code again?



```
01 void main(void){
02     ...
03     // 1.) Testing with valid LED-ID
04     int8_t result = sb_led_on(RED0);
05     if(result != 0){
06         // Test failed
07         // Output e.g. with 7-Segment display
08     }
09     // wait some seconds
10
11     // 2.) Testing with invalid LED-ID
12     ...
13 }
```

- Pay close attention to the interface description (incl. return values)
- Testing of **all possible return values**
- Give an error if the returned value is different from the specification

# Hands-on: Statistics Module

Screenecast: <https://www.video.uni-erlangen.de/clip/id/16328>



- Statistics module and test program
- Functionality of the module (interface):

```
01 // Interface
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 // Internal auxiliary functions
07 uint16_t getMin(uint16_t a, uint16_t b);
08 uint16_t getMax(uint16_t a, uint16_t b);
```

- Return value:
  - 0: OK
  - 1: Error
- How to proceed:
  - Header file with module interface (and include guards)
  - Implementation of the module (consider visibility)
  - Testing of the module in the main program (incl. errors)