

# System-Level Programming

## 4 Software Layers and Abstraction

**J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann**

Lehrstuhl für Informatik 4  
Systemsoftware

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Summer Term 2024

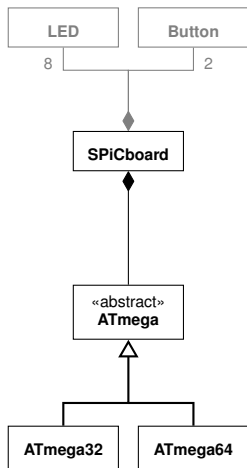
<http://sys.cs.fau.de/lehre/ss24>



# Abstraction by Software Layers: SPiCboard

close to problem ↑  
↓ close to machine

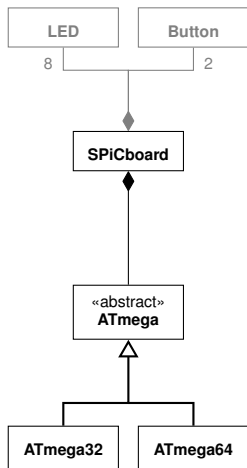
## Hardware view



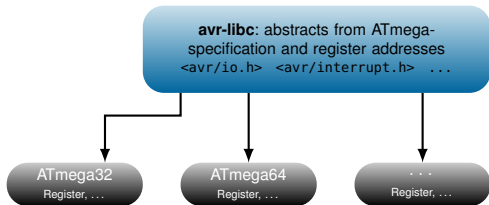
# Abstraction by Software Layers: SPiCboard

close to problem ↑  
↓ close to machine

## Hardware view



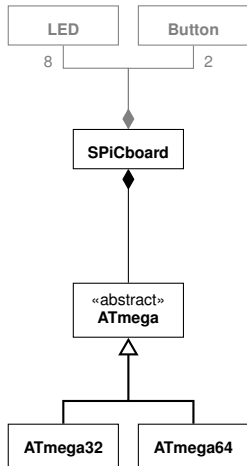
## Software layers



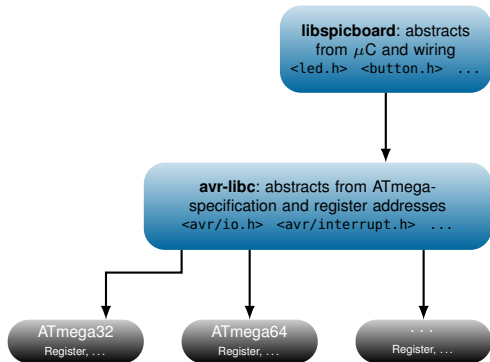
# Abstraction by Software Layers: SPiCboard

close to problem ↑  
↓ close to machine

## Hardware view



## Software layers

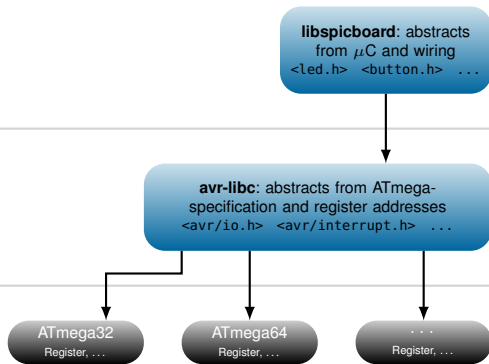


# Abstraction by Software Layers: *LED* → *on* Compared

close to problem

close to machine

**Goal:** Switch on LED RED0 on the SPiCboard:



04-Abstraktion\_en



# Abstraction by Software Layers: *LED* → *on* Compared

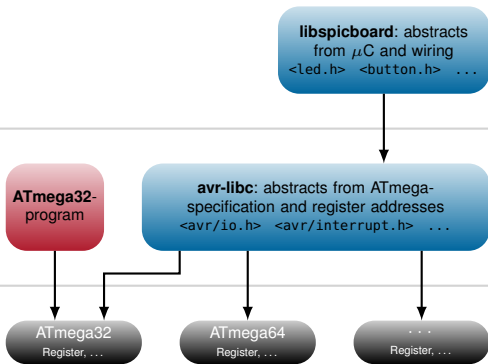
close to problem ↑

↓ close to machine

Program only runs on **ATmega32**. It uses register addresses specific to the **ATmega32** (like 0x12) and characteristics:

```
...  
(* (unsigned char*) (0x11)) |= (1<<7);  
(* (unsigned char*) (0x12)) &= ~(1<<7);
```

**Goal:** Switch on LED RED0 on the SPiCboard:



# Abstraction by Software Layers: *LED* → *on* Compared

close to problem ↑

↓ close to machine

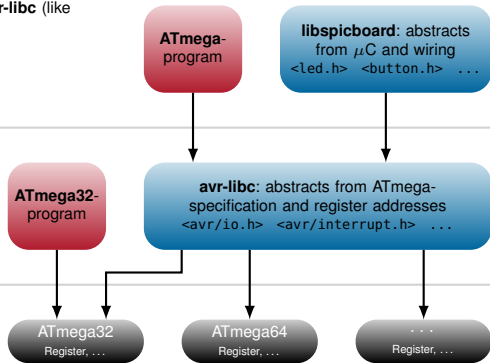
Program runs on **each**  $\mu$ C of the ATmega-series. It uses **symbolic register names** of the **avr-libc** (like PORTD) and general characteristics:

```
#include <avr/io.h>
...
DDRD |= (1<<7);
PORTD &= ~(1<<7);
```

Program only runs on **ATmega32**. It uses register addresses specific to the **ATmega32** (like 0x12) and characteristics:

```
...
(*(unsigned char*)(0x11)) |= (1<<7);
(*(unsigned char*)(0x12)) &= ~(1<<7);
```

**Goal:** Switch on LED RED0 on the SPiCboard:



04-Abstraktion\_en



# Abstraction by Software Layers: *LED* → *on* Compared

close to problem ↑

Program only runs on the **SPiCboard**. It uses functions (like `sb_led_on()`) and constants (like `RED0`) of the **libspicboard** that represent concrete wiring of LEDs, buttons, etc. with the  $\mu$ C:

```
#include <led.h>
...
sb_led_on(RED0);
```

Program runs on **each**  $\mu$ C of the ATmega-series. It uses **symbolic register names** of the **avr-libc** (like `PORTD`) and general characteristics:

```
#include <avr/io.h>
...
DDRD |= (1<<7);
PORTD &= ~(1<<7);
```

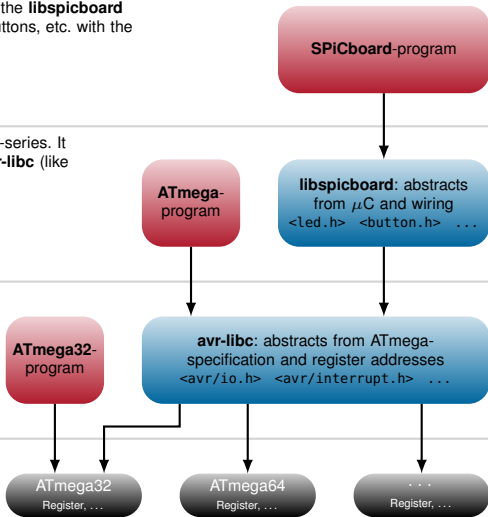
Program only runs on **ATmega32**. It uses register addresses specific to the **ATmega32** (like `0x12`) and characteristics:

```
...
(*(unsigned char*)(0x11)) |= (1<<7);
(*(unsigned char*)(0x12)) &= ~(1<<7);
```

**Goal:** Switch on LED `RED0` on the SPiCboard:



↓ close to machine





# Abstraction by Software Layers: Complete Example

**Until now:** development with avr-libc

```
#include <avr/io.h>

void main(void) {
    // initialize hardware

    // button0 on PD2
    DDRD  &= ~(1 << 2);
    PORTD |= (1 << 2);
    // LED on PD6
    DDRD  |= (1 << 6);
    PORTD |= (1 << 6);

    // wait until PD2: low --> (button0 pressed)
    while ((PIND >> 2) & 1) {
    }

    // greet user (red LED)
    PORTD &= ~(1 << 6); // PD6: low --> LED is on

    // wait forever
    while (1) {
    }
}
```

(ref. [↩ 3-12](#))

**Now:** development with  
libspicboard

```
#include <led.h>
#include <button.h>

void main(void) {

    // wait until Button0 is pressed
    while (sb_button_getState(BUTTON0)
           != PRESSED) {
    }

    // greet user
    sb_led_on(RED0);

    // wait forever
    while (1) {
    }
}
```



# Abstraction by Software Layers: Complete Example

**Until now:** development with avr-libc

```
#include <avr/io.h>

void main(void) {
    // initialize hardware

    // button0 on PD2
    DDRD  &= ~(1 << 2);
    PORTD |= (1 << 2);
    // LED on PD6
    DDRD  |= (1 << 6);
    PORTD |= (1 << 6);

    // wait until PD2: low --> (button0 pressed)
    while ((PIND >> 2) & 1) {
    }

    // greet user (red LED)
    PORTD &= ~(1 << 6); // PD6: low --> LED is on

    // wait forever
    while (1) {
    }
}
```

(ref. [↔](#) 3-12)

**Now:** development with libspicboard

```
#include <led.h>
#include <button.h>

void main(void) {

    // wait until Button0 is pressed
    while (sb_button_getState(BUTTON0)
           != PRESSED) {
    }

    // greet user
    sb_led_on(RED0);

    // wait forever
    while (1) {
    }
}
```

- Hardware initialisation not needed anymore
- Program simpler to understand due to **problem-specific abstraction**
  - setting bit 6 in PORTD  
↳ `sb_led_on(RED0)`
  - reading bit 2 in PORTD  
↳ `sb_button_getState(BUTTON0)`

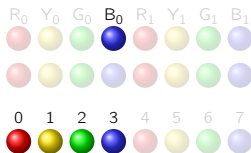


# Abstraction of the libspicboard: Short Overview

## ■ Output abstractions (selection)

### ■ LED module (`#include <led.h>`)

- switch LED on: `sb_led_on(BLUE0)`  $\rightsquigarrow$
- switch LED off: `sb_led_off(BLUE0)`  $\rightsquigarrow$
- switching all LEDs on or off:  
`sb_led_setMask(0x0f)`  $\rightsquigarrow$



### ■ 7 segment module (`#include <7seg.h>`)

- showing an integer  $n \in \{-9 \dots 99\}$ :  
`sb_7seg_showNumber(47)`  $\rightsquigarrow$



## ■ Input abstractions (selection)

### ■ Button module (`#include <button.h>`)

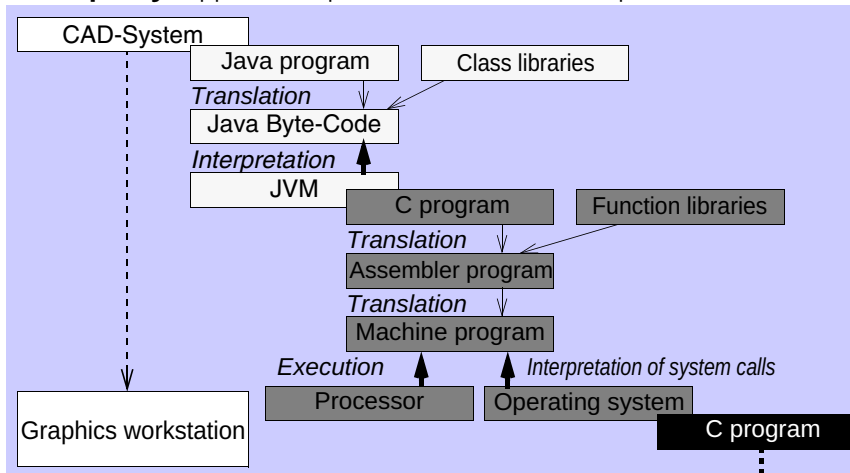
- reading the button state:  
`sb_button_getState(BUTTON0)`  $\mapsto$  `BUTTONSTATE_{PRESSED,RELEASED}`

### ■ ADC module (`#include <adc.h>`)

- reading the value of the potentiometer:  
`sb_adc_read(POTI)`  $\mapsto$  `{0...1023}`

# Software Layers in General

**Discrepancy:** application problem  $\longleftrightarrow$  hardware processes



**Goal:** executable machine code



# The Role of the Operating System

- **User view:** Environment for starting, controlling and combining of applications
  - Shell, graphical user interface
    - e. g., bash, Windows
  - Communication between applications and users
    - e. g., with files
- **Application view:** Function libraries with abstraction for easier software development
  - Generic in-/output of data
    - e. g., on printers, serial interfaces, in files
  - Permanent storage and transfer of data
    - e. g., by the file system, over TCP/IP sockets
  - Management of memory and other resources
    - e. g., CPU time



- **System view:** Software layers for multiplexing of the hardware ( $\leftrightarrow$  multi-user mode)
  - Parallel handling of program instances with **process concepts**
    - virtual memory  $\leftrightarrow$  own 32-/64-bit address space
    - virtual processor  $\leftrightarrow$  scheduled/preempted transparently
    - virtual in/output devices  $\leftrightarrow$  can be piped in files, sockets, ...
  - Isolation of program instances with **process concepts**
    - automatic garbage collection at the end of process life
    - detection/prevention of memory access to other processes
  - **Partial protection** from critical programming errors
    - detection of *some* invalid memory accesses (e. g., access to address 0)
    - detection of *some* invalid operations (e. g.,  $\text{div}/0$ )

$\mu\text{C}$  programming without operating system platform  $\rightsquigarrow$  **no protection**

- Operating system **protects programmer less** from bugs compared to e. g., Java.
- For the  $\mu\text{C}$  programming, we even have to **give up this protection**.
- 8/16-bit  $\mu\text{C}$  often have **no hardware support** for protection.



# Example: Error Detection by the Operating System

## Linux: Division by 0

```
1 #include <stdio.h>
2
3
4 int main(int argc, char **argv) {
5     int a = 23;
6     int b;
7
8     b = 4711 / (a - 23);
9     printf("Result: %d\n", b);
10
11     return 0;
12 }
```

Compilation and execution yields:

```
gcc error-linux.c -o error-linux
./error-linux
```

Floating point exception

~> program gets **terminated**.

## SPiCboard: Division by 0

```
#include <7seg.h>
#include <avr/interrupt.h>

void main(void) {
    int a = 23;
    int b;
    sei();
    b = 4711 / (a - 23);
    sb_7seg_showNumber(b);

    while (1) {}
}
```

Execution yields:



~> Program continues  
computation  
with **wrong data**.

