Systemnahe Programmierung in C

4 Softwareschichten und Abstraktion

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

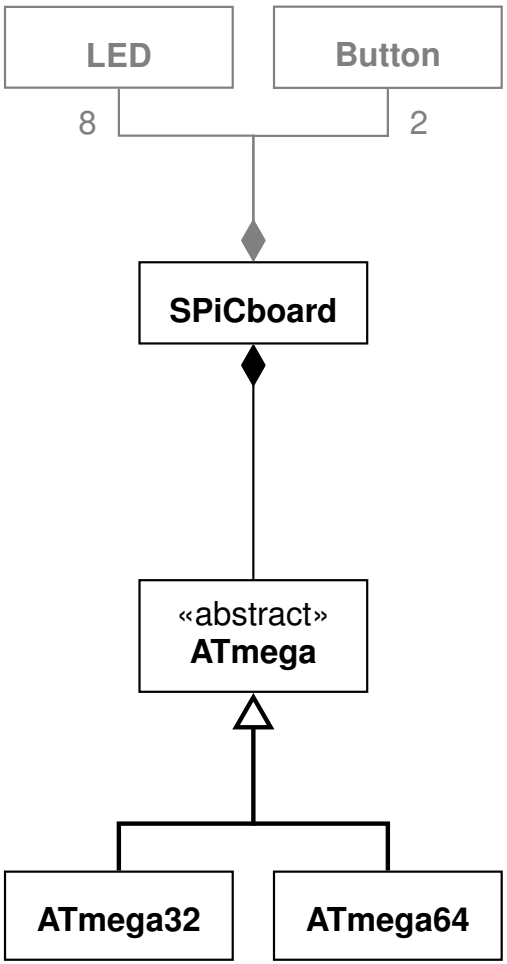
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2025

http://sys.cs.fau.de/lehre/ss25

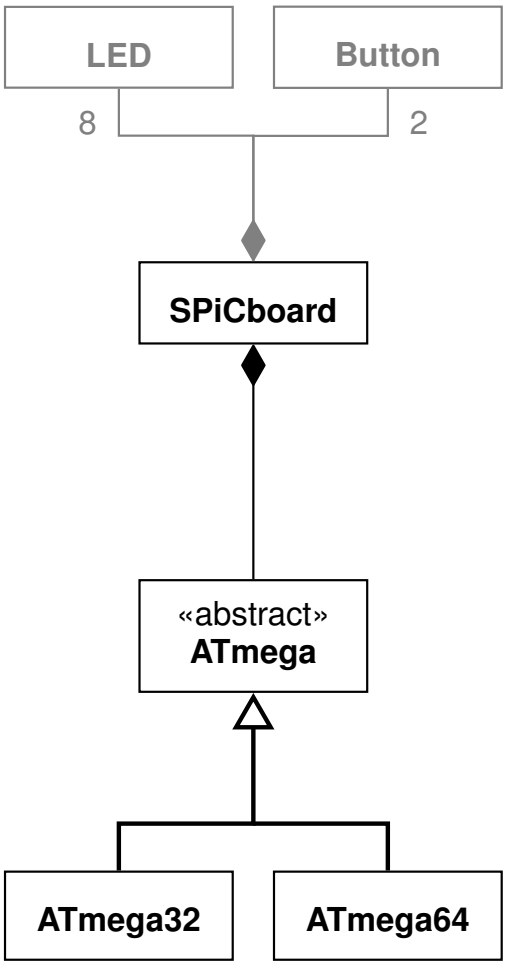
Abstraktion durch Softwareschichten: SPiCboard

Hardwaresicht

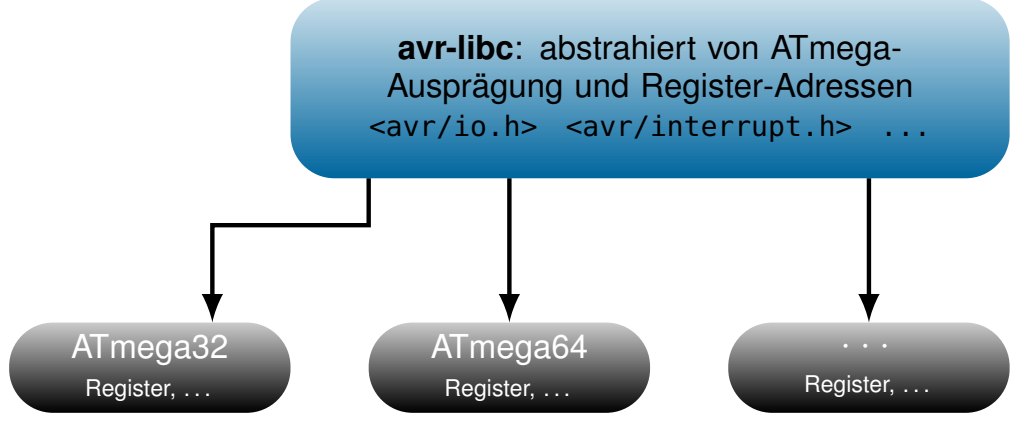


Abstraktion durch Softwareschichten: SPiCboard

Hardwaresicht

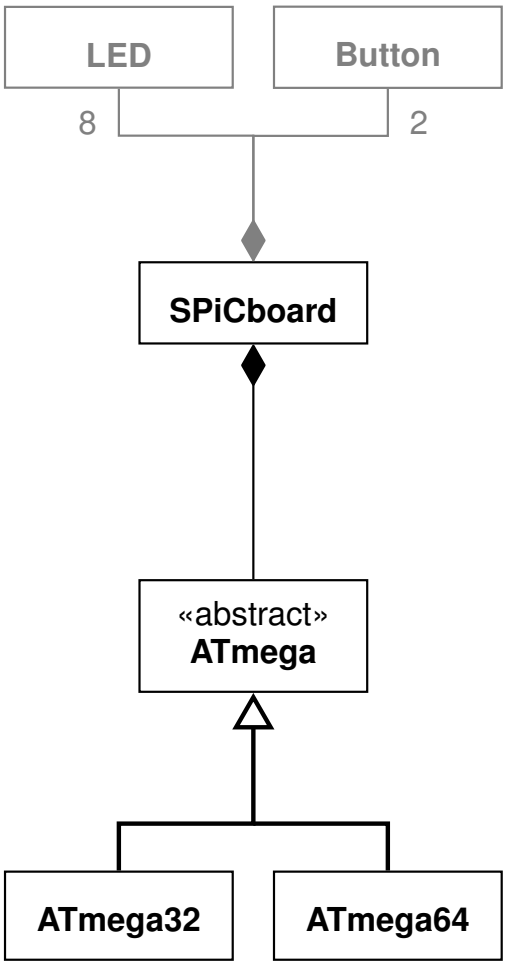


Softwareschichten

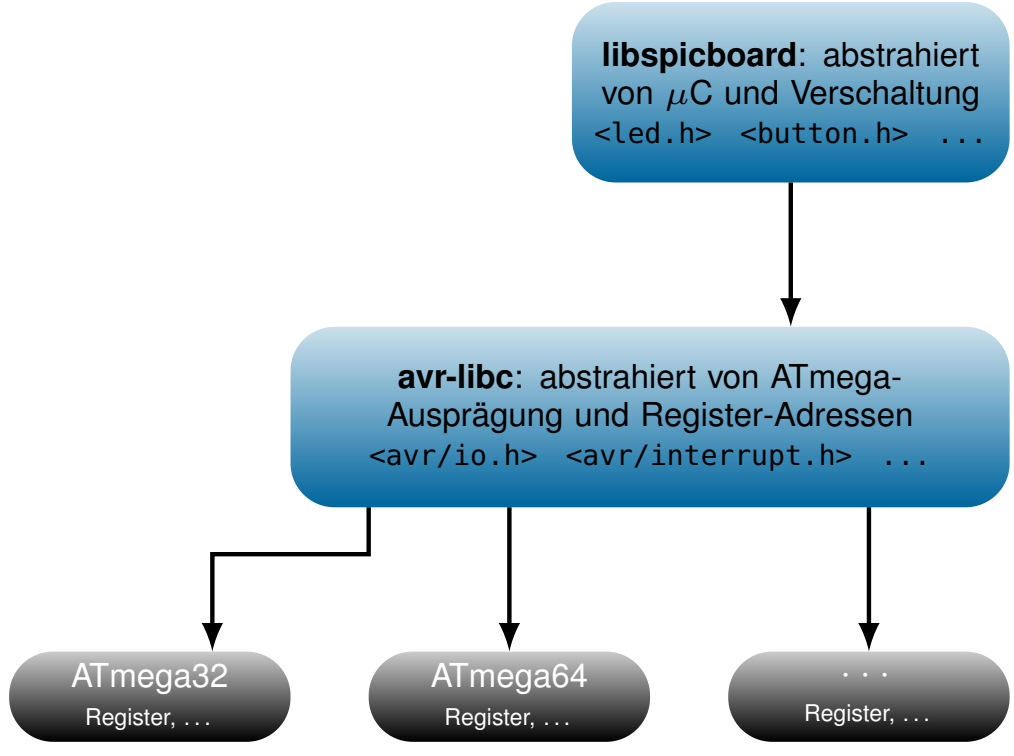


Abstraktion durch Softwareschichten: SPiCboard

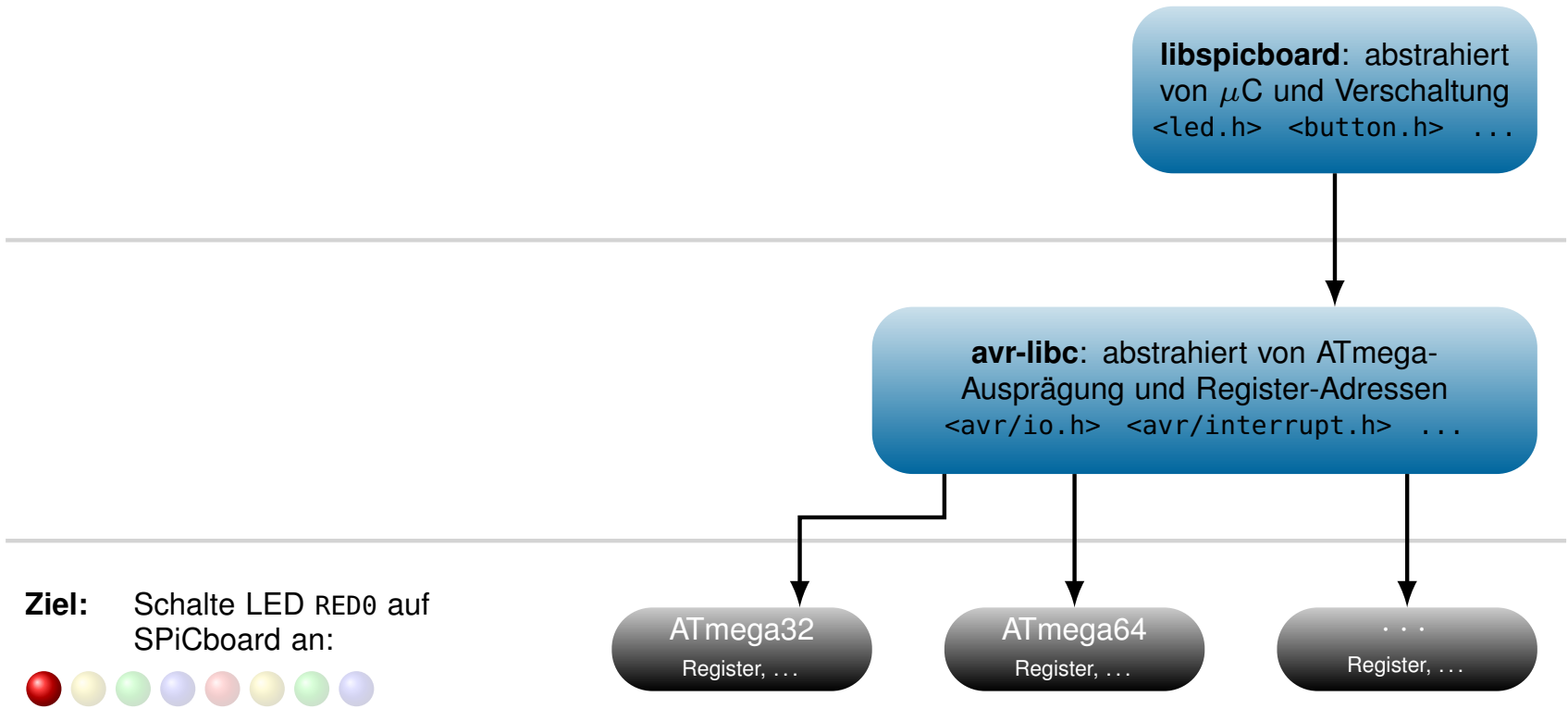
Hardwaresicht



Softwareschichten



Abstraktion durch Softwareschichten: *LED* → *on* im Vergleich



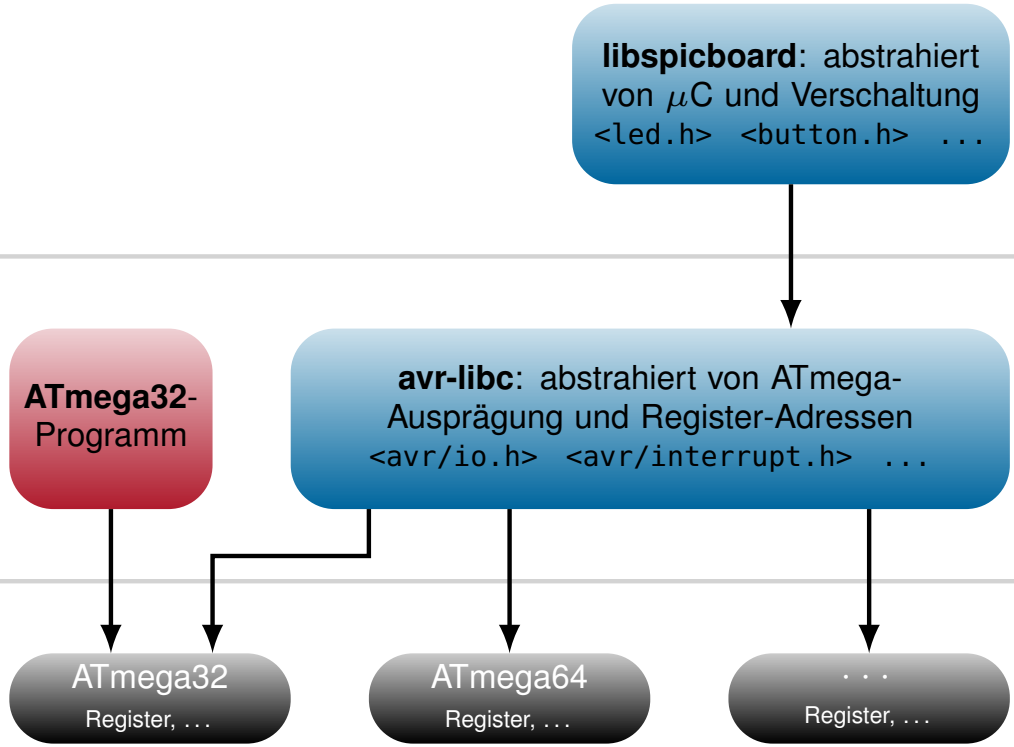
Ziel: Schalte LED `RED0` auf SPiCboard an:

Abstraktion durch Softwareschichten: *LED → on* im Vergleich

Programm läuft nur auf **ATmega32**. Es verwendet **ATmega32-spezifische** Registeradressen (wie `0x12`) und Merkmale:

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

Ziel: Schalte LED `RED0` auf SPiCboard an:



Abstraktion durch Softwareschichten: *LED → on* im Vergleich

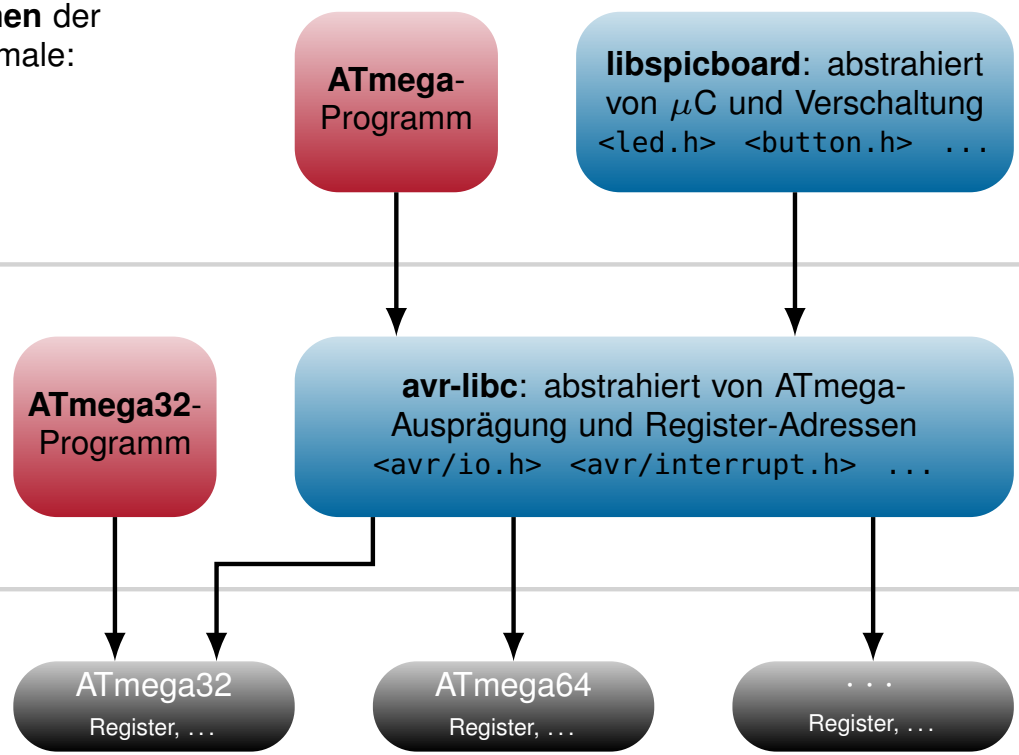
Programm läuft auf **jedem** μC der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

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

Programm läuft nur auf **ATmega32**. Es verwendet **ATmega32-spezifische** Registeradressen (wie `0x12`) und Merkmale:

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

Ziel: Schalte LED `RED0` auf SPiCboard an:



Abstraktion durch Softwareschichten: *LED → on* im Vergleich

Programm läuft nur auf dem **SPiCboard**. Es verwendet Funktionen (wie `sb_led_on()`) und Konstanten (wie `RED0`) der **libspicboard**, welche die konkrete Verschaltung von LEDs, Tastern, usw. mit dem μC repräsentieren:

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

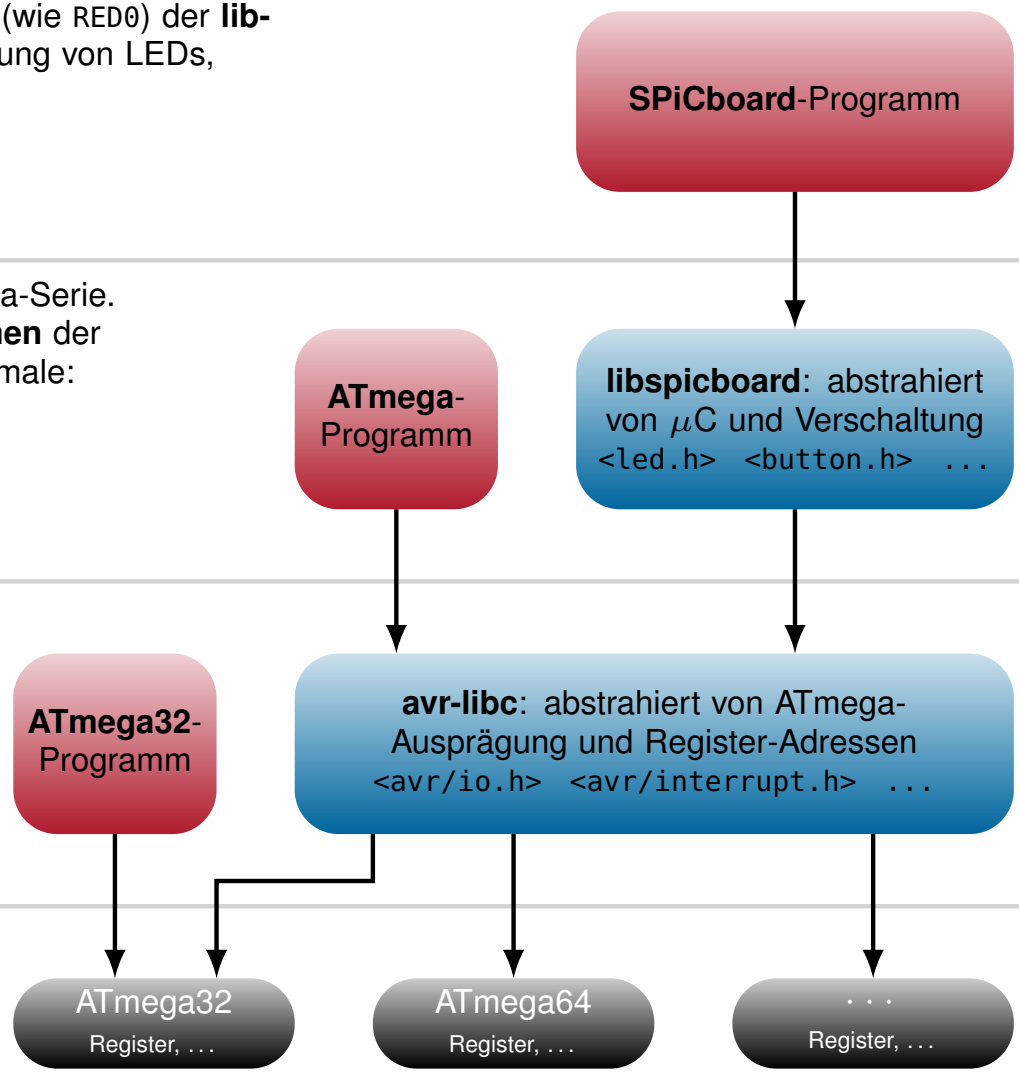
Programm läuft auf **jedem** μC der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

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

Programm läuft nur auf **ATmega32**. Es verwendet **ATmega32-spezifische** Registeradressen (wie `0x12`) und Merkmale:

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

Ziel: Schalte LED `RED0` auf SPiCboard an:



Abstraktion durch Softwareschichten: Vollständiges Beispiel

Bisher: Entwicklung mit 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) {
  }
}
```

(vgl. ↪ 3–11)

Nun: Entwicklung mit 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) {
  }
}
```

Abstraktion durch Softwareschichten: Vollständiges Beispiel

Bisher: Entwicklung mit 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) {
  }
}
```

(vgl. ↪ 3–11)

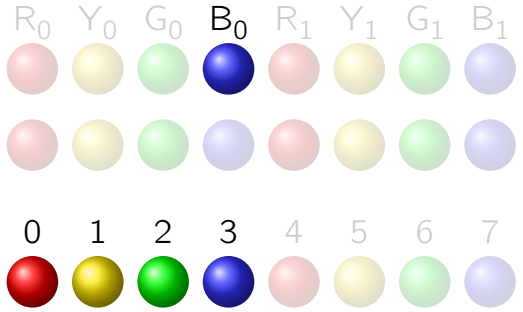
Nun: Entwicklung mit libspicboard

```
#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) {
  }
}
```

- Hardwareinitialisierung entfällt
- Programm ist einfacher und verständlicher durch **problemspezifische Abstraktionen**
  - Setze Bit 6 in `PORTD`
    ↦ `sb_led_on(RED0)`
  - Lese Bit 2 in `PORTD`
    ↦ `sb_button_getState(BUTTON0)`

Abstraktionen der libspicboard: Kurzüberblick

- Ausgabe-Abstraktionen (Auswahl)
  - LED-Modul (`#include <led.h>`)
    - LED einschalten: `sb_led_on(BLUE0)` ⇝
    - LED ausschalten: `sb_led_off(BLUE0)` ⇝
    - Alle LEDs ein-/ausschalten: `sb_led_setMask(0x0f)` ⇝

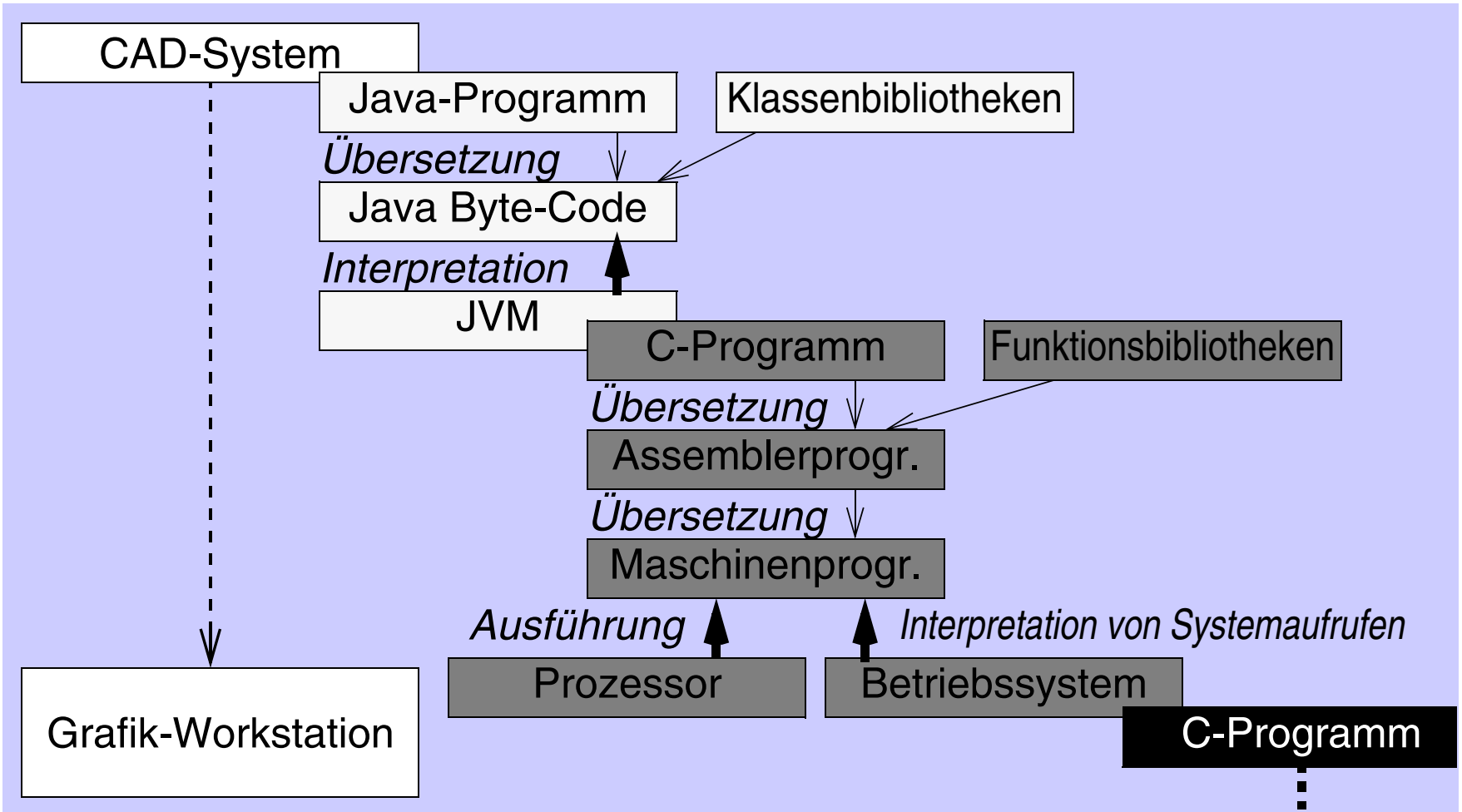


  - 7-Seg-Modul (`#include <7seg.h>`)
    - Ganzzahl $n \in \{-9 \ldots 99\}$ ausgeben: `sb_7seg_showNumber(47)` ⇝

- Eingabe-Abstraktionen (Auswahl)
  - Button-Modul (`#include <button.h>`)
    - Button-Zustand abfragen: `sb_button_getState(BUTTON0)` ↦ `BUTTONSTATE_{PRESSED,RELEASED}`
  - ADC-Modul (`#include <adc.h>`)
    - Potentiometer-Stellwert abfragen: `sb_adc_read(POTI)` ↦ $\{0 \ldots 1023\}$

Softwareschichten im Allgemeinen

Diskrepanz: Anwendungsproblem $\longleftrightarrow$ Abläufe auf der Hardware



Ziel: Ausführbarer Maschinencode

Problemnähe — Maschinennähe

Die Rolle des Betriebssystems

- **Anwendersicht:** Umgebung zum Starten, Kontrollieren und Kombinieren von Anwendungen
  - Shell, grafische Benutzeroberfläche
    - z. B. bash, Windows
  - Datenaustausch zwischen Anwendungen und Anwendern
    - z. B. über Dateien
- **Anwendungssicht:** Funktionsbibliothek mit Abstraktionen zur Vereinfachung der Softwareentwicklung
  - Generische Ein-/Ausgabe von Daten
    - z. B. auf Drucker, serielle Schnittstelle, in Datei
  - Permanentspeicherung und Übertragung von Daten
    - z. B. durch Dateisystem, über TCP/IP-Sockets
  - Verwaltung von Speicher und anderen Betriebsmitteln
    - z. B. CPU-Zeit

Die Rolle des Betriebssystems (Forts.)

- **Systemsicht:** Softwareschicht zum Multiplexen der Hardware ($\hookrightarrow$ Mehrbenutzerbetrieb)
  - Parallele Abarbeitung von Programminstanzen durch **Prozesskonzept**
    - Virtueller Speicher $\hookrightarrow$ eigener 32-/64-Bit-Adressraum
    - Virtueller Prozessor $\hookrightarrow$ wird transparent zugeteilt und entzogen
    - Virtuelle Ein-/Ausgabe-Geräte $\hookrightarrow$ umlenkbar in Datei, Socket, ...
  - Isolation von Programminstanzen durch **Prozesskonzept**
    - Automatische Speicherbereinigung bei Prozessende
    - Erkennung/Vermeidung von Speicherzugriffen auf fremde Prozesse
  - **Partieller Schutz** vor schwereren Programmierfehlern
    - Erkennung *einiger* ungültiger Speicherzugriffe (z. B. Zugriff auf Adresse `0`)
    - Erkennung *einiger* ungültiger Operationen (z. B. `div/0`)

μC-Programmierung ohne Betriebssystemplattform $\rightsquigarrow$ **kein Schutz**

- Ein Betriebssystem schützt **weit weniger** vor Programmierfehlern als z. B. Java.
- Selbst darauf müssen wir jedoch bei der μC-Programmierung i. a. **verzichten**.
- Bei 8/16-Bit-μC fehlt i. a. die für Schutz erforderliche **Hardware-Unterstützung**.

Beispiel: Fehlererkennung durch Betriebssystem

Linux: Division durch 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("Ergebnis: %d\n", b);
10
11   return 0;
12 }
```

Übersetzen und Ausführen ergibt:

```
gcc error-linux.c -o error-linux
./error-linux
Floating point exception
```

⇝ Programm wird **abgebrochen**.

SPiCboard: Division durch 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) {}
}
```

Ausführen ergibt:

⇝ Programm setzt Berechnung fort mit **falschen Daten**.