Systemnahe Programmierung in C

3 Java versus C – Erste Beispiele

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

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

Das erste C-Programm

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

Das erste C-Programm

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

- Übersetzen und Ausführen (auf einem UNIX-System)

```
~> gcc -o hello hello.c
~> ./hello
Hello World!
~>
```

Gar nicht so schwer :-)

Das erste C-Programm – Vergleich mit Java

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

- Das berühmteste Programm der Welt in **Java**

```java
import java.lang.System;
class Hello {
  public static void main(String[] args) {
    /* greet user */
    System.out.println("Hello World!");
    return;
  }
}
```


Das erste C-Programm – Vergleich mit Java

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

C: Ein C-Programm startet in `main()`, einer **globalen Funktion** vom Typ `int`, die in genau einer **Datei** definiert ist.

- Das berühmteste Programm der Welt in **Java**

```java
import java.lang.System;
class Hello {
  public static void main(String[] args) {
    /* greet user */
    System.out.println("Hello World!");
    return;
  }
}
```

Java: Jedes Java-Programm startet in `main()`, einer **statischen Methode** vom Typ `void`, die in genau einer **Klasse** definiert ist.

Das erste C-Programm – Vergleich mit Java

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

C: <keine Entsprechung>

- Das berühmteste Programm der Welt in **Java**

```java
import java.lang.System;
class Hello {
  public static void main(String[] args) {
    /* greet user */
    System.out.println("Hello World!");
    return;
  }
}
```

Java: Jedes Java-Programm besteht aus mindestens einer **Klasse**.

Das erste C-Programm – Vergleich mit Java

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

C: Die Ausgabe einer Zeichenkette erfolgt mit der **Funktion** `printf()`. (`\n` ⇝ Zeilenumbruch)

- Das berühmteste Programm der Welt in **Java**

```java
import java.lang.System;
class Hello {
  public static void main(String[] args) {
    /* greet user */
    System.out.println("Hello World!");
    return;
  }
}
```

Java: Die Ausgabe einer Zeichenkette erfolgt mit der **Methode** `println()` aus der Klasse `out` aus dem Paket `System`.

Das erste C-Programm – Vergleich mit Java

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

C: Für die Benutzung von `printf()` wird die **Funktionsbibliothek** `stdio.h` mit der **Präprozessor-Anweisung** `#include` eingebunden.

- Das berühmteste Programm der Welt in **Java**

```java
import java.lang.System;
class Hello {
  public static void main(String[] args) {
    /* greet user */
    System.out.println("Hello World!");
    return;
  }
}
```

Java: Für die Benutzung der **Klasse** `out` wird das **Paket** `System` mit der `import`-Anweisung eingebunden.

Das erste C-Programm – Vergleich mit Java

- Das berühmteste Programm der Welt in **C**

```c
#include <stdio.h>

int main(int argc, char **argv) {
  /* greet user */
  printf("Hello World!\n");
  return 0;
}
```

C: Rückkehr zum Betriebssystem mit **Rückgabewert**. 0 bedeutet hier, dass kein Fehler aufgetreten ist.

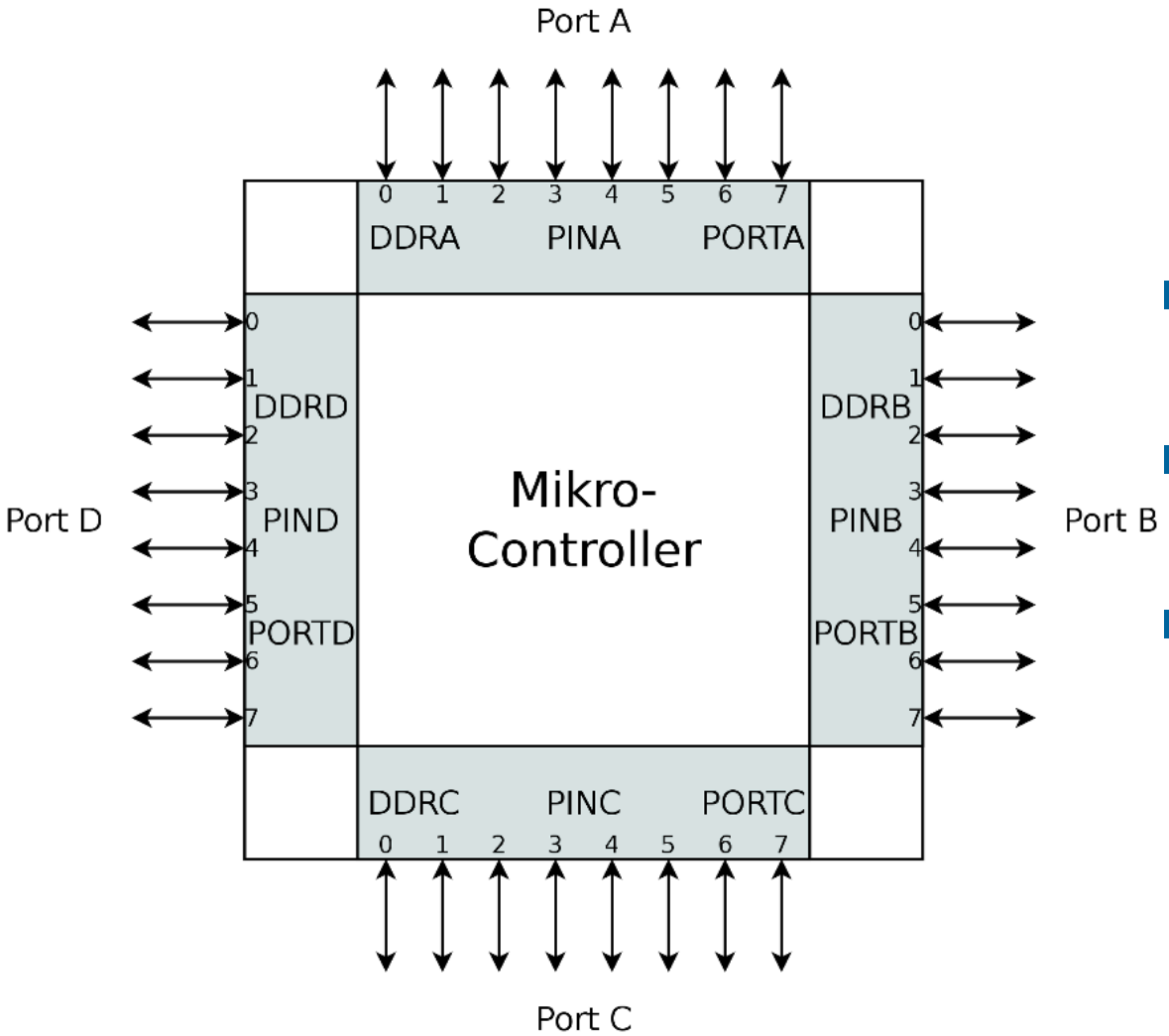
- Das berühmteste Programm der Welt in **Java**

```java
import java.lang.System;
class Hello {
  public static void main(String[] args) {
    /* greet user */
    System.out.println("Hello World!");
    return;
  }
}
```

Java: Rückkehr zum Betriebssystem.

Das erste C-Programm für einen μ-Controller

Vorbemerkung:

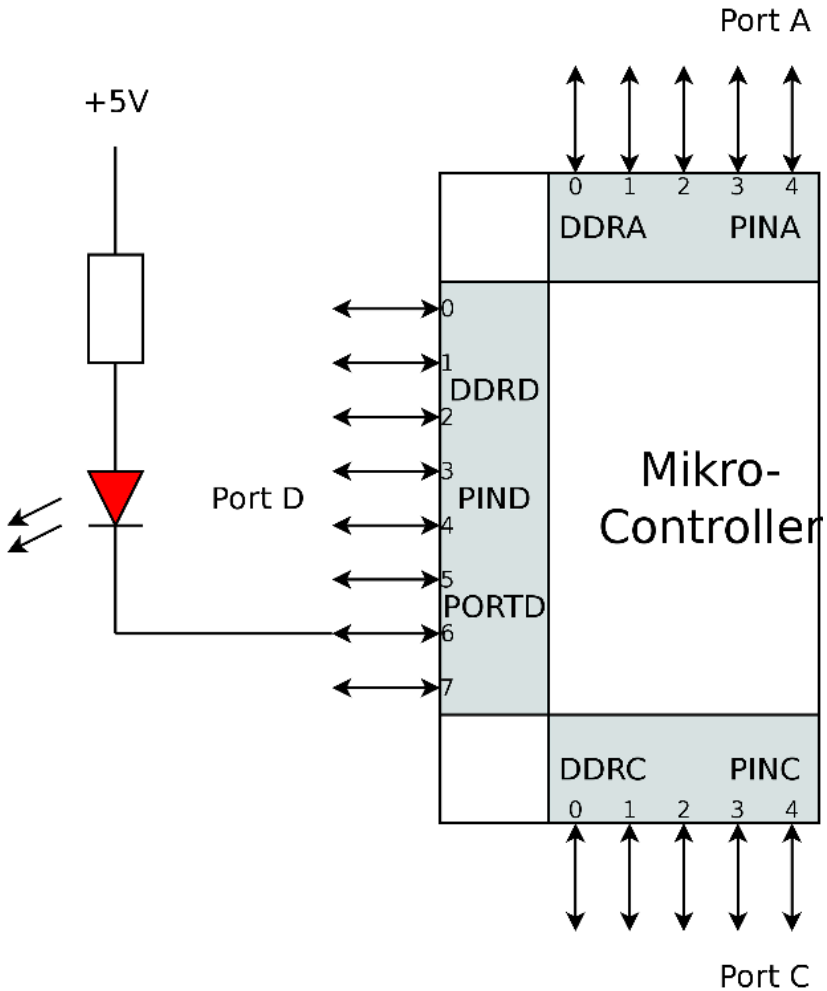


- DDRx: Data Direction Register
- PINx: Port Input Register
- PORTx: Port Ouput Register

(jeweils 8 Bit)

Das erste C-Programm für einen μ-Controller

Vorbemerkung:



- LED leuchtet nicht:
  - DDRD Bit 6: '1' (Output)
  - PORTD Bit 6: '1' (5V)
- LED leuchtet:
  - DDRD Bit 6: '1' (Output)
  - PORTD Bit 6: '0' (0V)

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (SPiCboard)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (SPiCboard)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

- Übersetzen und **Flashen** (mit SPiC-IDE) ⇝ Übung

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (SPiCboard)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

- Übersetzen und **Flashen** (mit SPiC-IDE) ⇝ Übung
- Ausführen (SPiCboard):  (rote LED leuchtet)

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (SPiCboard)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

μ-Controller-Programmierung ist „irgendwie anders“.

- Übersetzen und **Flashen** (mit SPiC-IDE) $\rightsquigarrow$ Übung
- Ausführen (SPiCboard):  (rote LED leuchtet)

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ 3–1)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

Die `main()`-Funktion hat **keinen Rückgabewert** (Typ `void`). Ein μ-Controller-Programm läuft **endlos** ⇝ `main()` terminiert nie.

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ 3–1)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

Es erfolgt **keine Rückkehr** zum Betriebssystem (wohin auch?). Die Endlosschleife stellt sicher, dass `main()` nicht terminiert.

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ 3–1)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

Zunächst wird die **Hardware** initialisiert (in einen definierten Zustand gebracht). Dazu müssen **einzelne Bits** in bestimmten **Hardware-Registern** manipuliert werden.

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ 3–1)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

Die Interaktion mit der Umwelt (hier: LED einschalten) erfolgt ebenfalls über die **Manipulation einzelner Bits** in Hardware-Registern.

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ 3–1)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

Für den Zugriff auf Hardware-Register (`DDRD`, `PORTD`, bereitgestellt als **globale Variablen**) wird die **Funktionsbibliothek** `avr/io.h` mit `#include` eingebunden.

Das zweite C-Programm – Eingabe unter Linux

- Benutzerinteraktion (Lesen eines Zeichens) unter Linux:

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Press key: ");
  char key = getchar();

  printf("You pressed %c\n", key);
  return 0;
}
```

Das zweite C-Programm – Eingabe unter Linux

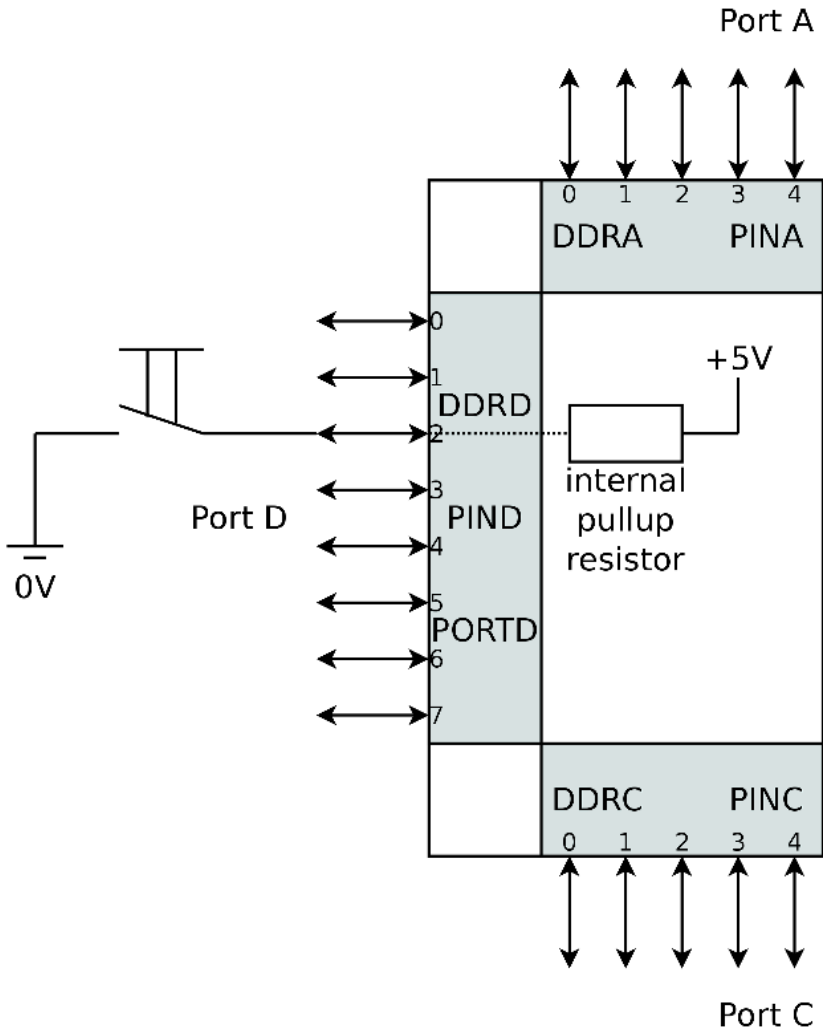
- Benutzerinteraktion (Lesen eines Zeichens) unter Linux:

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Press key: ");
  char key = getchar();

  printf("You pressed %c\n", key);
  return 0;
}
```

Die `getchar()`-Funktion liest ein Zeichen von der Standardeingabe (hier: Tastatur). Sie „wartet“ gegebenenfalls, bis ein Zeichen verfügbar ist.

Das zweite C-Programm für einen μ-Controller

Vorbemerkung:



- Initialisierung:
  - DDRD Bit 2: '0' (Input)
  - PORTD Bit 2: '1' (Pullup eingeschaltet)
- Erkennung:
  - PIND Bit 2: '1' => Taster nicht gedrückt
  - PIND Bit 2: '0' => Taster gedrückt

Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

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

void main(void) {
  /* initialize hardware: button on port D pin 2 */
  DDRD  &= ~(1 << 2);  /* PD2 is used as input */
  PORTD |=  (1 << 2);  /* activate pull-up: PD2: high */

  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1 << 6);  /* PD6 is used as output */
  PORTD |= (1 << 6);  /* PD6: high --> LED is off */

  /* wait until PD2 -> low (button is pressed) */
  while ((PIND >> 2) & 1) {
  }

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

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


Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

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

void main(void) {
  /* initialize hardware: button on port D pin 2 */
  DDRD  &= ~(1 << 2);  /* PD2 is used as input */
  PORTD |=  (1 << 2);  /* activate pull-up: PD2: high */

  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1 << 6);  /* PD6 is used as output */
  PORTD |= (1 << 6);  /* PD6: high --> LED is off */

  /* wait until PD2 -> low (button is pressed) */
  while ((PIND >> 2) & 1) {
  }

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

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

Wie die LED ist der Taster mit einem **digitalen IO-Pin** des μ-Controllers verbunden. Hier konfigurieren wir Pin 2 von Port D als **Eingang** durch **Löschen** des entsprechenden Bits im Register `DDRD`.

Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

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

void main(void) {
  /* initialize hardware: button on port D pin 2 */
  DDRD  &= ~(1 << 2);  /* PD2 is used as input */
  PORTD |=  (1 << 2);  /* activate pull-up: PD2: high */

  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1 << 6);  /* PD6 is used as output */
  PORTD |= (1 << 6);  /* PD6: high --> LED is off */

  /* wait until PD2 -> low (button is pressed) */
  while ((PIND >> 2) & 1) {
  }

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

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

Durch **Setzen** von Bit 2 im Register `PORTD` wird der interne Pull-Up-Widerstand (hochohmig) aktiviert, über den V_{CC} anliegt $\rightsquigarrow$ PD2 = *high*.

Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

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

void main(void) {
  /* initialize hardware: button on port D pin 2 */
  DDRD  &= ~(1 << 2);  /* PD2 is used as input */
  PORTD |=  (1 << 2);  /* activate pull-up: PD2: high */

  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1 << 6);  /* PD6 is used as output */
  PORTD |= (1 << 6);  /* PD6: high --> LED is off */

  /* wait until PD2 -> low (button is pressed) */
  while ((PIND >> 2) & 1) {
  }

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

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

Aktive Warteschleife: Wartet auf Tastendruck, d. h. solange PD2 (Bit 2 im Register `PIND`) *high* ist. Ein Tasterdruck zieht PD2 auf Masse $\rightsquigarrow$ Bit 2 im Register `PIND` wird *low* und die Schleife verlassen.

Zum Vergleich: Benutzerinteraktion als Java-Programm

```java
import java.lang.System;
import javax.swing.*;
import java.awt.event.*;

public class Input implements ActionListener {
  private JFrame frame;

  public static void main(String[] args) {
    // create input, frame and button objects
    Input input = new Input();
    input.frame = new JFrame("Java-Programm");
    JButton button = new JButton("Klick mich");

    // add button to frame
    input.frame.add(button);
    input.frame.setSize(400, 400);
    input.frame.setVisible(true);

    // register input as listener of button events
    button.addActionListener(input);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("Knopfdruck!");
    System.exit(0);
  }
}
```

Eingabe als „typisches“ Java-Programm (**objektorientiert**, **grafisch**)

Zum Vergleich: Benutzerinteraktion als Java-Programm

```
import java.lang.System;
import javax.swing.*;
import java.awt.event.*;

public class Input implements ActionListener {
  private JFrame frame;

  public static void main(String[] args) {
    // create input, frame and button objects
    Input input = new Input();
    input.frame = new JFrame("Java-Programm");
    JButton button = new JButton("Klick mich");

    // add button to frame
    input.frame.add(button);
    input.frame.setSize(400, 400);
    input.frame.setVisible(true);

    // register input as listener of button events
    button.addActionListener(input);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("Knopfdruck!");
    System.exit(0);
  }
}
```

Eingabe als „typisches“ Java-Programm (**objektorientiert**, **grafisch**)

Um Interaktionsereignisse zu empfangen, implementiert die Klasse `Input` ein entsprechendes **Interface**.

Zum Vergleich: Benutzerinteraktion als Java-Programm

```
import java.lang.System;
import javax.swing.*;
import java.awt.event.*;

public class Input implements ActionListener {
  private JFrame frame;

  public static void main(String[] args) {
    // create input, frame and button objects
    Input input = new Input();
    input.frame = new JFrame("Java-Programm");
    JButton button = new JButton("Klick mich");

    // add button to frame
    input.frame.add(button);
    input.frame.setSize(400, 400);
    input.frame.setVisible(true);

    // register input as listener of button events
    button.addActionListener(input);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("Knopfdruck!");
    System.exit(0);
  }
}
```

Eingabe als „typisches“ Java-Programm (**objektorientiert**, **grafisch**)

Das Programmverhalten ist implementiert durch eine Menge von **Objekten** (`frame`, `button`, `input`), die hier bei der Initialisierung erzeugt werden.

Zum Vergleich: Benutzerinteraktion als Java-Programm

```java
import java.lang.System;
import javax.swing.*;
import java.awt.event.*;

public class Input implements ActionListener {
  private JFrame frame;

  public static void main(String[] args) {
    // create input, frame and button objects
    Input input = new Input();
    input.frame = new JFrame("Java-Programm");
    JButton button = new JButton("Klick mich");

    // add button to frame
    input.frame.add(button);
    input.frame.setSize(400, 400);
    input.frame.setVisible(true);

    // register input as listener of button events
    button.addActionListener(input);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("Knopfdruck!");
    System.exit(0);
  }
}
```

Eingabe als „typisches“ Java-Programm (**objektorientiert**, **grafisch**)

Das erzeugte `button`-Objekt schickt nun seine Nachrichten an das `input`-Objekt.

Zum Vergleich: Benutzerinteraktion als Java-Programm

```
import java.lang.System;
import javax.swing.*;
import java.awt.event.*;

public class Input implements ActionListener {
  private JFrame frame;

  public static void main(String[] args) {
    // create input, frame and button objects
    Input input = new Input();
    input.frame = new JFrame("Java-Programm");
    JButton button = new JButton("Klick mich");

    // add button to frame
    input.frame.add(button);
    input.frame.setSize(400, 400);
    input.frame.setVisible(true);

    // register input as listener of button events
    button.addActionListener(input);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("Knopfdruck!");
    System.exit(0);
  }
}
```

Eingabe als „typisches“ Java-Programm (**objektorientiert**, **grafisch**)

Der Knopfdruck wird durch eine `actionPerformed()`-Nachricht (Methodenaufruf) signalisiert.

Ein erstes Fazit: Von Java → C (Syntax)

- **Syntaktisch** sind Java und C sich sehr ähnlich
  (Syntax: „Wie sehen gültige Programme der Sprache aus?“)
- C-Syntax war Vorbild bei der Entwicklung von Java
  ⇝ Viele Sprachelemente sind ähnlich oder identisch verwendbar
  - Blöcke, Schleifen, Bedingungen, Anweisungen, Literale
  - Werden in den folgenden Kapiteln noch im Detail behandelt
- Wesentliche Sprachelemente aus Java gibt es in C jedoch **nicht**
  - Klassen, Pakete, Objekte, Ausnahmen (Exceptions), ...

Ein erstes Fazit: Von Java → C (Idiomatik)

- **Idiomatisch** gibt es sehr große Unterschiede
  (Idiomatik: „Wie sehen übliche Programme der Sprache aus?“)
- **Java: Objektorientiertes Paradigma**
  - Zentrale Frage: Aus welchen Dingen besteht das Problem?
  - Gliederung der Problemlösung in **Klassen** und Objekte
  - Hierarchiebildung durch Vererbung und Aggregation
  - Programmablauf durch Interaktion zwischen Objekten
  - Wiederverwendung durch umfangreiche Klassenbibliothek
- **C: Imperatives Paradigma**
  - Zentrale Frage: Aus welchen Aktivitäten besteht das Problem?
  - Gliederung der Problemlösung in **Funktionen** und Variablen
  - Hierarchiebildung durch Untergliederung in Teilfunktionen
  - Programmablauf durch Aufrufe zwischen Funktionen
  - Wiederverwendung durch Funktionsbibliotheken

Ein erstes Fazit: Von Java → C (Philosophie)

- **Philosophisch** gibt es ebenfalls erhebliche Unterschiede (Philosophie: „Grundlegende Ideen und Konzepte der Sprache“)
- **Java:** Sicherheit und Portabilität durch **Maschinenferne**
  - Übersetzung für virtuelle Maschine (JVM)
  - Umfangreiche Überprüfung von Programmfehlern zur Laufzeit
    - Bereichsüberschreitungen, Division durch 0, ...
  - Problemnahes Speichermodell
    - Nur typsichere Speicherzugriffe, automatische Bereinigung zur Laufzeit
- **C:** Effizienz und Leichtgewichtigkeit durch **Maschinennähe**
  - Übersetzung für konkrete Hardwarearchitektur
  - Keine Überprüfung von Programmfehlern zur Laufzeit
    - Einige Fehler werden vom Betriebssystem abgefangen – falls vorhanden
  - Maschinennahes Speichermodell
    - Direkter Speicherzugriff durch **Zeiger**
    - Grobgranularer Zugriffsschutz und automatische Bereinigung (auf Prozessebene) durch das Betriebssystem – falls vorhanden

Ein erstes Fazit: μ-Controller-Programmierung

C $\mapsto$ Maschinennähe $\mapsto$ μC-Programmierung

Die Maschinennähe von C zeigt sich insbesondere auch bei der μ-Controller-Programmierung!

- Es läuft nur ein Programm
  - Wird bei RESET direkt aus dem Flash-Speicher gestartet
  - Muss zunächst die Hardware initialisieren
  - Darf nie terminieren (z. B. durch Endlosschleife in `main()`)
- Die Problemlösung ist maschinennah implementiert
  - Direkte Manipulation von einzelnen Bits in Hardwareregistern
  - Detailliertes Wissen über die elektrische Verschaltung erforderlich
  - Keine Unterstützung durch Betriebssystem (wie etwa Linux)
  - Allgemein geringes Abstraktionsniveau $\rightsquigarrow$ fehleranfällig, aufwändig

Ein erstes Fazit: μ-Controller-Programmierung

C $\mapsto$ Maschinennähe $\mapsto$ μC-Programmierung

Die Maschinennähe von C zeigt sich insbesondere auch bei der μ-Controller-Programmierung!

- Es läuft nur ein Programm
  - Wird bei RESET direkt aus dem Flash-Speicher gestartet
  - Muss zunächst die Hardware initialisieren
  - Darf nie terminieren (z. B. durch Endlosschleife in `main()`)
- Die Problemlösung ist maschinennah implementiert
  - Direkte Manipulation von einzelnen Bits in Hardwareregistern
  - Detailliertes Wissen über die elektrische Verschaltung erforderlich
  - Keine Unterstützung durch Betriebssystem (wie etwa Linux)
  - Allgemein geringes Abstraktionsniveau $\rightsquigarrow$ fehleranfällig, aufwändig

Ansatz: Mehr Abstraktion durch problemorientierte Bibliotheken