Systemnahe Programmierung in C (SPiC)

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

http://www4.cs.fau.de/Lehre/SS16/V_SPIC

Referenzen

[1] *ATmega32 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. 8155-AVR-07/09. Atmel Corporation. July 2009.

[GDI] Frank Bauer. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2015 (jährlich). URL: `https://gdi.cs.fau.de/w15/material`.

[2] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop, et al. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. (Als E-Book aus dem Uninetz verfügbar; PDF-Version unter `/proj/i4gspic/pub`). Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: `http://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1`.

[3] Brian W. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1978.

[4] Brian W. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1988. ISBN: 978-8120305960.

[5] Dennis MacAlistair Ritchie and Ken Thompson. "The Unix Time-Sharing System". In: *Communications of the ACM* 17.7 (July 1974), pp. 365–370. DOI: `10.1145/361011.361061`.

Referenzen (Forts.)

[6] David Tennenhouse. “Proactive Computing”. In: *Communications of the ACM* (May 2000), pp. 43–45.

[7] Jim Turley. “The Two Percent Solution”. In: *embedded.com* (Dec. 2002). `http://www.embedded.com/story/OEG20021217S0039`, visited 2011-04-08.

Veranstaltungsüberblick

Systemnahe Programmierung in C (SPiC)

Teil A Konzept und Organisation

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

http://www4.cs.fau.de/Lehre/SS16/V_SPIC

Überblick: Teil A Konzept und Organisation

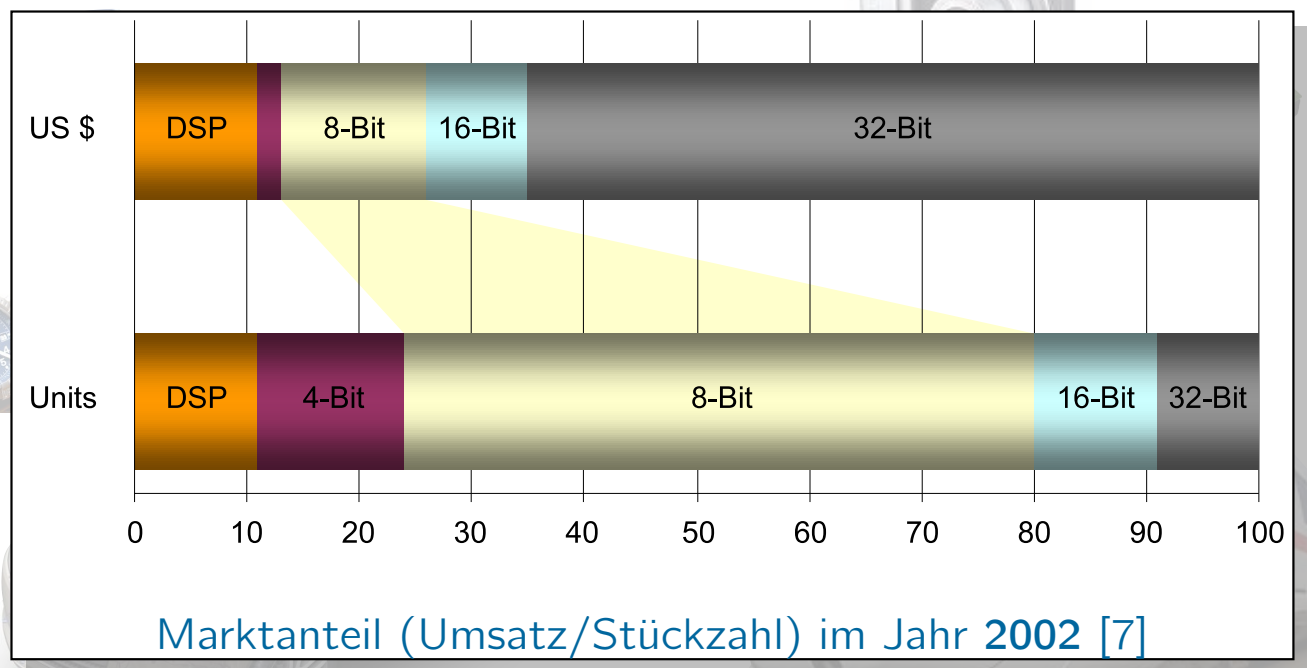
Lernziele

- **Vertiefen** des Wissens über Konzepte und Techniken der Informatik für die Softwareentwicklung
  - Ausgangspunkt: Grundlagen der Informatik (GdI)
  - Schwerpunkt: Systemnahe Softwareentwicklung in C
- **Entwickeln** von Software in C für einen μ-Controller (μC) und eine Betriebssystem-Plattform (Linux)
  - SPiCboard-Lehrentwicklungsplattform mit ATmega-μC
  - Praktische Erfahrungen in hardware- und systemnaher Softwareentwicklung machen
- **Verstehen** der technologischen Sprach- und Hardwaregrundlagen für die Entwicklung systemnaher Software
  - Die Sprache C verstehen und einschätzen können
  - Umgang mit Nebenläufigkeit und Hardwarenähe
  - Umgang mit den Abstraktionen eines Betriebssystems (Dateien, Prozesse, ...)

V_SPIC_handout

Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren wurden im Jahr 2000 in einem **eingebetteten System** verbaut [6]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und μ-Controller, **8-Bit oder kleiner** [6, 7]



Marktanteil (Umsatz/Stückzahl) im Jahr **2002** [7]

Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren wurden im Jahr 2000 in einem **eingebetteten System** verbaut [6]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und μ-Controller, **8-Bit oder kleiner** [6, 7]
- **Relevant:** **25 Prozent** der Stellenanzeigen für EE-Ingenieure enthalten die Stichworte *embedded* oder *automotive* (`http://stepstone.com`, 4. April 2011)

Bei den oberen Zahlen ist gesunde Skepsis geboten

- Die Veröffentlichungen [6, 7] sind mehr als 10 Jahre alt!
- Man kann dennoch davon ausgehen, dass die relativen Größenordnungen nach wie vor stimmen
  - 2016 liegt der Anteil an 8-Bittern (vermutlich) noch bei 40 Prozent
  - 4-Bitter dürften inzwischen jedoch weitgehend ausgestorben sein

Motivation: Die ATmega-μC-Familie (8-Bit)

Type	Flash	SRAM	IO	Timer 8/16	UART	SPI	ADC	PWM	EUR
ATTINY13	1 KiB	64 B	6	1/-	-	-	1*4	-	**0,86**
ATTINY2313	2 KiB	128 B	18	1/1	-	1	-	-	**0,99**
ATMEGA48	4 KiB	512 B	23	2/1	1	1	8*10	6	**1,40**
ATMEGA16	16 KiB	1024 B	32	2/1	1	1	8*10	4	**2,05**
ATMEGA32	32 KiB	2048 B	32	2/1	1	1	8*10	4	**3,65**
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	8	**5,70**
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	8	**7,35**
ATMEGA256	256 KiB	8192 B	86	2/2	4	1	16*10	16	**8,99**

ATmega-Varianten (Auswahl) und Handelspreise (Reichelt Elektronik, April 2015)

- Sichtbar wird: **Ressourcenknappheit**
  - **Flash** (Speicher für Programmcode und konstante Daten) ist **knapp**
  - **RAM** (Speicher für Laufzeit-Variablen) ist **extrem knapp**
  - Wenige Bytes „Verschwendung” $\rightsquigarrow$ signifikant höhere Stückzahlkosten

Motivation: Die Sprache C

- Systemnahe Softwareentwicklung erfolgt überwiegend in **C**
  - **Warum C?** (und nicht Java/Cobol/Scala/<*Lieblingssprache*>)
- C steht für eine Reihe hier wichtiger Eigenschaften
  - Laufzeiteffizienz (CPU)
    - Übersetzter C-Code läuft direkt auf dem Prozessor
    - Keine Prüfungen auf Programmierfehler zur Laufzeit
  - Platzeffizienz (Speicher)
    - Code und Daten lassen sich sehr kompakt ablegen
    - Keine Prüfung der Datenzugriffe zur Laufzeit
  - Direktheit (Maschinennähe)
    - C erlaubt den direkten Zugriff auf Speicher und Register
  - Portabilität
    - Es gibt für **jede** Plattform einen C-Compiler
    - C wurde „erfunden" (1973), um das Betriebssystem UNIX portabel zu implementieren [3, 5]

↝ **C** ist die **lingua franca** der systemnahen Softwareentwicklung!

Motivation: SPiC – Stoffauswahl und Konzept

- **Lehrziel:** Systemnahe Softwareentwicklung in C
  - Das ist ein sehr umfangreiches Feld: Hardware-Programmierung, Betriebssysteme, Middleware, Datenbanken, Verteilte Systeme, Übersetzerbau, ...
  - Dazu kommt dann noch das Erlernen der Sprache C selber
- **Ansatz**
  - Konzentration auf zwei Domänen
    - μ-Controller-Programmierung
    - Softwareentwicklung für die Linux-Systemschnittstelle
  - Gegensatz μC-Umgebung $\leftrightarrow$ Betriebssystemplattform erfahren
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (ME)

Vorlesungsskript

- Das Handout der Vorlesungsfolien wird online und als 4×1-Ausdruck auf Papier zur Verfügung gestellt
  - Ausdrucke werden vor der Vorlesung verteilt
  - Online-Version wird vor der Vorlesung aktualisiert
  - Handout enthält (in geringem Umfang) zusätzliche Informationen

- **Das Handout kann eine eigene Mitschrift nicht ersetzen!**

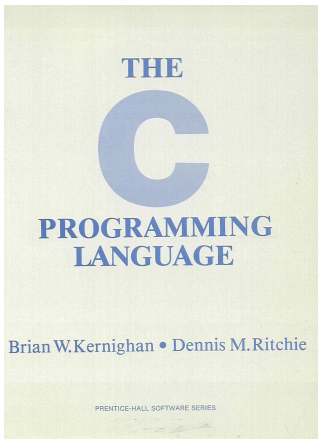
Literaturempfehlungen

[2] Für den Einstieg empfohlen:

Manfred Dausmann, Ulrich Bröckl, Dominic Schoop, et al. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. (Als E-Book aus dem Uninetz verfügbar; PDF-Version unter `/proj/i4gspic/pub`). Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: `http://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1`

[4] Der „Klassiker” (eher als Referenz geeignet):

Brian W. Kernighan and Dennis MacAlistair Ritchie. *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1988. ISBN: 978-8120305960



Vorlesung

- Inhalt und Themen
  - Grundlegende Konzepte der systemnahen Programmierung
  - Einführung in die Programmiersprache C
    - Unterschiede zu Java
    - Modulkonzept
    - Zeiger und Zeigerarithmetik
  - Softwareentwicklung auf „der nackten Hardware“ (ATmega-μC)
    - Abbildung Speicher $\leftrightarrow$ Sprachkonstrukte
    - Unterbrechungen (*interrupts*) und Nebenläufigkeit
  - Softwareentwicklung auf „einem Betriebssystem“ (Linux)
    - Betriebssystem als Ausführungsumgebung für Programme
    - Abstraktionen und Dienste eines Betriebssystems
- Termin: Do 16:15–17:45, H11
  - Einzeltermin am 13. April (Mi), 16:00–17:30, H8
  - insgesamt 13 Vorlesungstermine

↪ 2–7

Übungen

- Tafelübung und Rechnerübung
  - Tafelübungen
    - Ausgabe und Erläuterung der Programmieraufgaben
    - Gemeinsame Entwicklung einer Lösungsskizze
    - Besprechung der Lösungen
  - Rechnerübungen
    - selbstständige Programmierung
    - Umgang mit Entwicklungswerkzeug (Atmel Studio)
    - Betreuung durch Übungsbetreuer
- Termin: Initial 9 Gruppen zur Auswahl
  - Anmeldung über Waffel (siehe Webseite): Heute, 18:00 – So, 18:00
  - Bei zu wenigen Teilnehmern behalten wir uns eine Verteilung auf andere Gruppen vor. Ihr werdet in diesem Fall per E-Mail angeschrieben.

Zur Übungsteilnahme wird ein gültiges Login in Linux-CIP gebraucht!

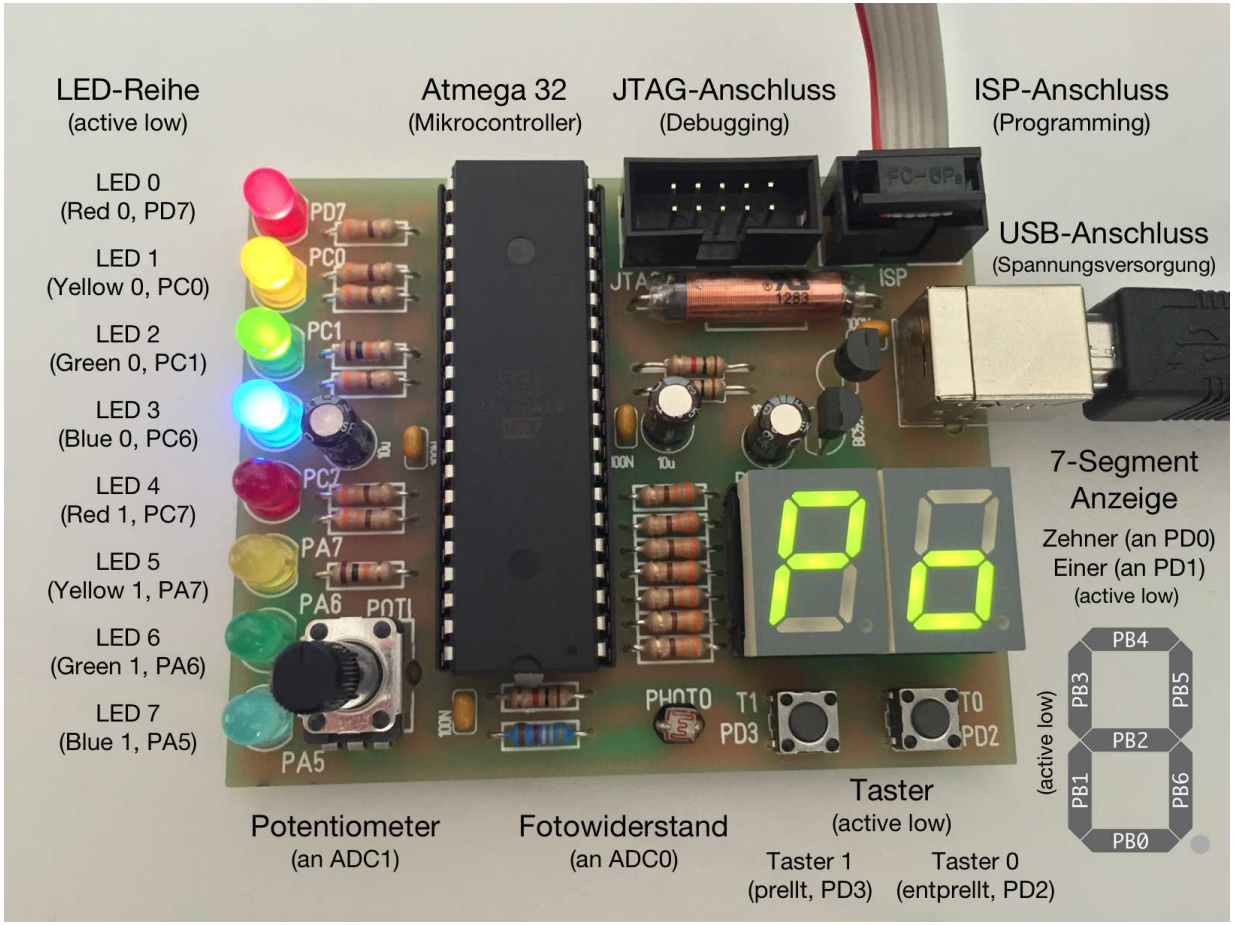
Programmieraufgaben

- Praktische Umsetzung des Vorlesungsstoffs
  - Acht Programmieraufgaben ↪ 2–7
  - Bearbeitung teilweise alleine / mit Übungspartner
- Lösungen mit Abgabeskript am Rechner abgeben
  - Lösung wird durch Skripte überprüft
  - Wir korrigieren und bepunkten die Abgaben und geben sie zurück
- ★ Abgabe der Übungsaufgaben ist freiwillig; ↪ 2–6
  es können jedoch bis zu **10% Bonuspunkte**
  für die Prüfungsklausur erarbeitet werden!

Unabhängig davon ist die Teilnahme an den Übungen **dringend empfohlen!**

Übungsplattform: Das SPiCboard

- ATmega32-μC
- JTAG-Anschluss
- 8 LEDs
- 2 7-Seg-Elemente
- 2 Taster
- 1 Potentiometer
- 1 Fotosensor



- Ausleihe zur Übungsbearbeitung möglich
- Oder noch besser ↪ selber Löten

SPiCboard-Lötabend

- Die Fachschaften (EEI / ME) bieten einen „Lötabend“ an
  - Teilnahme ist freiwillig
  - (Erste) Löterfahrung sammeln beim Löten eines eigenen SPiCboards

- **Termine:** Mo. 18. – Do. 21. April, 18:30 – 21:30
  Fr. 22. April, 16:00 – 19:00
- **Anmeldung:** über Waffel (Heute, 18:00 – So, 18:00, siehe Webseite)
- **Kostenbeitrag:** SPiCBoard: kostenlos
  Progger: 30 EUR (optional!)

Die Progger (ISPs) können ggfs. auch gebraucht erworben werden.

Thread im EEI-Forum: `http://eei.fsi.fau.de/forum/post/3208`

Prüfung und Modulnote

- Prüfung (Klausur)
  - Termin: voraussichtlich Ende Juli / Anfang August
  - Dauer: 90 min
  - Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe
- Klausurnote $\mapsto$ Modulnote
  - Bestehensgrenze (in der Regel): 50% der möglichen Klausurpunkte (KP)
  - Falls bestanden ist eine Notenverbesserung möglich durch Bonuspunkte aus den Programmieraufgaben
    - Basis (Minimum): 50% der möglichen Übungspunkte (ÜP)
    - Jede weiteren 5% der möglichen ÜP $\mapsto$ +1% der möglichen KP

  $\rightsquigarrow$ 100% der möglichen ÜP $\mapsto$ +10% der möglichen KP

Semesterplanung

KW	Mo	Di	Mi	Do	Fr	Themen	Kapitel im Skript
15	11.04.	12.04.	13.04.	14.04.	15.04.	*Einführung, Organisation, Java nach C, Abstraktion, Sprachüberblick, Datentypen*	*VL 1: 1.1 – 3.15, VL 2: 4.1 – 6.14*
			VL 1	VL 2			
16	18.04.	19.04.	20.04.	21.04.	22.04.	*Ausdrücke, Kontrollstrukturen, Funktionen, Variablen*	*7.1 – 10.2*
	A1 (Blink)			VL 3			
17	25.04.	26.04.	27.04.	28.04.	29.04.	*Präprozessor, Programmstruktur, Module, Zeiger, Felder*	*11.1 – 13.11*
	A2 (Snake)			VL 4			
18	02.05.	03.05.	04.05.	05.05.	06.05.		
	A3 (Spiel)			*Himmelf.*			
19	09.05.	10.05.	11.05.	12.05.	13.05.	*Zeigerarithmetik, Mikrocontroller-Systemarchitektur, volatile, Verbundtypen (struct, union)*	*13.12 – 14.22*
	A4 (LED-Modul)			VL 5			
20	16.05.	17.05.	18.05.	19.05.	20.05.	*Interrupts, Nebenläufigkeit*	*15.1 – 15.23*
	Pfingsten/Berg			VL 6			
21	23.05.	24.05.	25.05.	26.05.	27.05.		
	A5 (Ampel)			*Fronleich.*			
22	30.05.	31.06.	01.06.	02.06.	03.06.	*Ergänzungen zur Einführung in C, Betriebssysteme*	*16.1 – 17.7*
				VL 7			
23	06.06.	07.06.	08.06.	09.06.	10.06.	*Dateisysteme*	*18.1 – 18.29*
	A6 (Hello World)			VL 8			
24	13.06.	14.06.	15.06.	16.06.	17.06.	*Programme und Prozesse*	*19.1 – 19.25*
	A7 (Printdir)			VL 9			
25	20.06.	21.06.	22.06.	23.06.	24.06.	*Speicherorganisation*	*20.1 – 20.12*
				VL 10			
26	27.06.	28.06.	29.06.	30.06.	01.07.	*Nebenläufige Prozesse*	*21.1 – 21.7*
	A8 (Lish)			VL 11			
27	04.07.	05.07.	06.07.	07.07.	08.07.	*Synchronisation*	*21.8 – 21.34*
				VL 12			
28	11.07.	12.07.	13.07.	14.07.	15.07.	*Fragestunde*	
				VL 13			

Beteiligte Personen, LS Informatik 4

Dozenten Vorlesung

Volkmar Sieh

Daniel Lohmann

Jürgen Kleinöder

Organisatoren des Übungsbetriebs

Rainer Müller

Sebastian Maier

Heiko Janker

Beteiligte Personen, LS Informatik 4 (Forts.)

Techniker (Ausleihe SPiCboard und Debugger)

Harald Junggunst

Christian Preller

Daniel Christiani

Beteiligte Personen, LS Informatik 4 (Forts.)

Übungsleiter

Carsten Braun

Sabrina Bruckmeier

Benjamin Brunner

Max Heidbrink

Ivo Ihlemann

Lukas Neckermann

Hannes Wagner

Manfred Wich

Bei Fragen oder Problemen

- Vorlesungs- und Übungsfolien konsultieren
- Häufig gestellte Fragen (FAQ) und Antworten siehe Webseite
  → `http://www4.cs.fau.de/Lehre/SS16/V_SPIC`
  → Übungen
  → FAQ
- Allgemeine Fragen zu Übungsaufgaben etc. im EEI-Forum posten
  → `https://eei.fsi.uni-erlangen.de/forum/forum/16`
- Bei speziellen Fragen Mail an Mailingliste (alle Übungsleiter)
  → i4spic@cs.fau.de

Systemnahe Programmierung in C (SPiC)

Teil B Einführung in C

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

http://www4.cs.fau.de/Lehre/SS16/V_SPIC

Überblick: Teil B Einführung in C

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 – Erläuterungen [Handout]

- **C**-Version zeilenweise erläutert
  1. Für die Benutzung von `printf()` wird die **Funktionsbibliothek** `stdio.h` mit der **Präprozessor-Anweisung** `#include` eingebunden.
  3. Ein C-Programm startet in `main()`, einer **globalen Funktion** vom Typ `int`, die in genau einer **Datei** definiert ist.
  5. Die Ausgabe einer Zeichenkette erfolgt mit der **Funktion** `printf()`. (`\n` ⇝ Zeilenumbruch)
  6. Rückkehr zum Betriebssystem mit **Rückgabewert**. 0 bedeutet hier, dass kein Fehler aufgetreten ist.

- **Java**-Version zeilenweise erläutert
  1. Für die Benutzung der **Klasse** `out` wird das **Paket** `System` mit der `import`-Anweisung eingebunden.
  2. Jedes Java-Programm besteht aus mindestens einer **Klasse**.
  3. Jedes Java-Programm startet in `main()`, einer **statischen Methode** vom Typ `void`, die in genau einer **Klasse** definiert ist.
  5. Die Ausgabe einer Zeichenkette erfolgt mit der **Methode** `println()` aus der Klasse `out` aus dem Paket `System`. [↪ GDI, 01-10]
  6. Rückkehr zum Betriebssystem.

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR-ATmega (SPiCboard)

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

void main() {
  // initialize hardware: LED on port D pin 7, active low
  DDRD  |= (1<<7);  // PD7 is used as output
  PORTD |= (1<<7);  // PD7: high --> LED is off

  // greet user
  PORTD &= ~(1<<7); // PD7: low --> LED is on

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

μ-Controller-Programmierung ist „irgendwie anders“.

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

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ ??)

```
1  #include <avr/io.h>
2
3  void main() {
4    // initialize hardware: LED on port D pin 7, active low
5    DDRD  |= (1<<7);  // PD7 is used as output
6    PORTD |= (1<<7);  // PD7: high --> LED is off
7
8    // greet user
9    PORTD &= ~(1<<7); // PD7: low --> LED is on
10
11   // wait forever
12   while(1){
13   }
14 }
```


μ-Controller-Programm – Erläuterungen [Handout]

- μ-Controller-Programm zeilenweise erläutert
  (Beachte Unterschiede zur Linux-Version ↪ 3–2)

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

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

  5-6 Zunächst wird die **Hardware** initialisiert (in einen definierten Zustand gebracht). Dazu müssen **einzelne Bits** in bestimmten **Hardware-Registern** manipuliert werden.

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

  12-13 Es erfolgt **keine Rückkehr** zum Betriebssystem (wohin auch?). Die Endlosschleife stellt sicher, dass `main()` nicht terminiert.

Das zweite C-Programm – Eingabe unter Linux

- Benutzerinteraktion (Lesen eines Zeichens) unter Linux:

```
#include <stdio.h>

int main(int argc, char** argv){

  printf("Press key: ");
  int 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. In dieser Zeit entzieht das Betriebssystem den Prozessor.

Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

```
1  #include <avr/io.h>
2
3  void main() {
4    // initialize hardware: button on port D pin 2
5    DDRD  &= ~(1<<2);  // PD2 is used as input
6    PORTD |=  (1<<2);  // activate pull-up: PD2: high
7
8    // initialize hardware: LED on port D pin 7, active low
9    DDRD  |= (1<<7);  // PD7 is used as output
10   PORTD |= (1<<7);  // PD7: high --> LED is off
11
12   // wait until PD2 -> low (button is pressed)
13   while(PIND & (1<<2))
14     ;
15
16   // greet user
17   PORTD &= ~(1<<7); // PD7: low --> LED is on
18
19   // wait forever
20   while(1)
21     ;
22 }
```

Warten auf Tasterdruck – Erläuterungen [Handout]

- Benutzerinteraktion mit SPiCboard zeilenweise erläutert

  5 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`.

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

  13-14 **Aktive Warteschleife:** Wartet auf Tastendruck, d. h. solange PD2 (Bit 2 im Register `PIND`) *high* ist. Ein Tasterdruck zieht PD2 auf Masse $\leadsto$ Bit 2 im Register `PIND` wird *low* und die Schleife verlassen.

Zum Vergleich: Benutzerinteraktion als Java-Programm

```
1  import java.lang.System;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  public class Input implements ActionListener {
6    private JFrame frame;
7
8    public static void main(String[] args) {
9      // create input, frame and button objects
10     Input input = new Input();
11     input.frame = new JFrame("Java-Programm");
12     JButton button = new JButton("Klick mich");
13
14     // add button to frame
15     input.frame.add(button);
16     input.frame.setSize(400, 400);
17     input.frame.setVisible(true);
18
19     // register input as listener of button events
20     button.addActionListener(input);
21   }
22
23   public void actionPerformed(ActionEvent e) {
24     System.out.println("Knopfdruck!");
25     System.exit(0);
26   }
27 }
```

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

- Das Programm ist mit der C-Variante nicht unmittelbar vergleichbar
  - Es verwendet das in Java übliche (und Ihnen bekannte) **objektorientierte Paradigma**.
  - Dieser Unterschied soll hier verdeutlicht werden.

- Benutzerinteraktion in Java zeilenweise erläutert

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

  10-12 Das Programmverhalten ist implementiert durch eine Menge von **Objekten** (`frame`, `button`, `input`), die hier bei der Initialisierung erzeugt werden.

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

  23-26 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

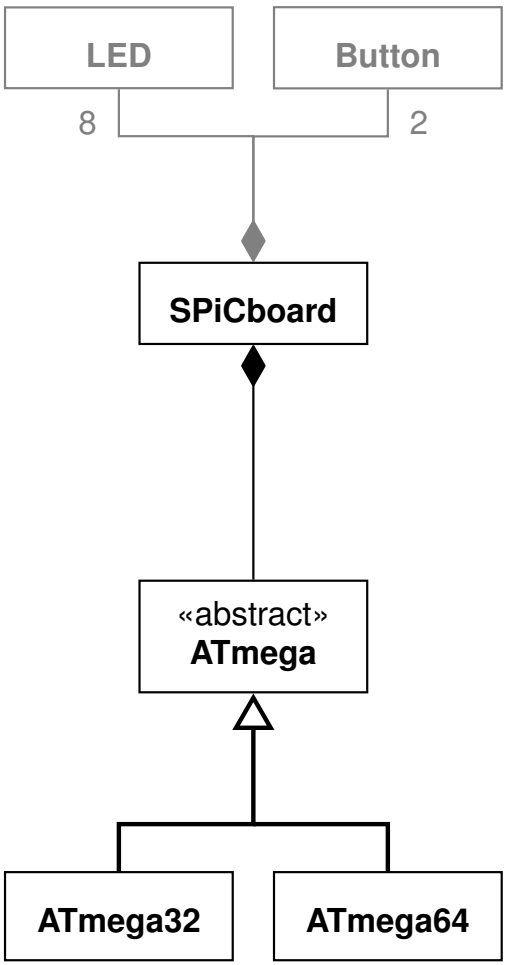
Die Maschinennähe von C zeigt sich insbesondere auch bei der μ-Controller-Programmierung!

- 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

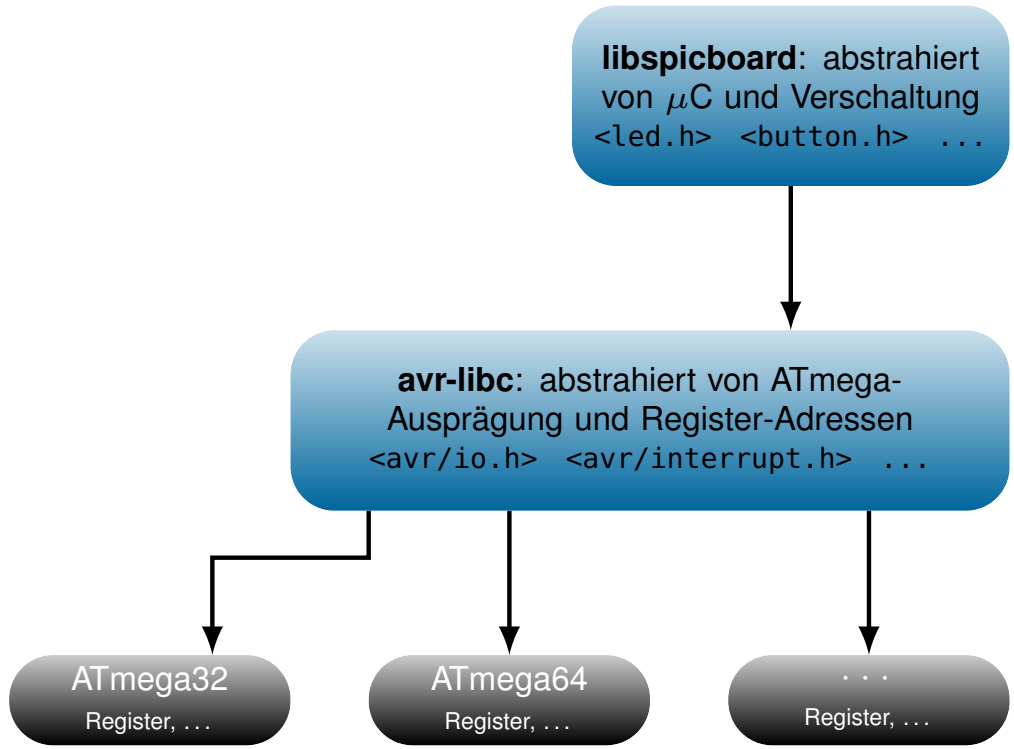
Ansatz: Mehr Abstraktion durch problemorientierte Bibliotheken

Abstraktion durch Softwareschichten: SPiCboard

Hardwaresicht



Softwareschichten



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);
```

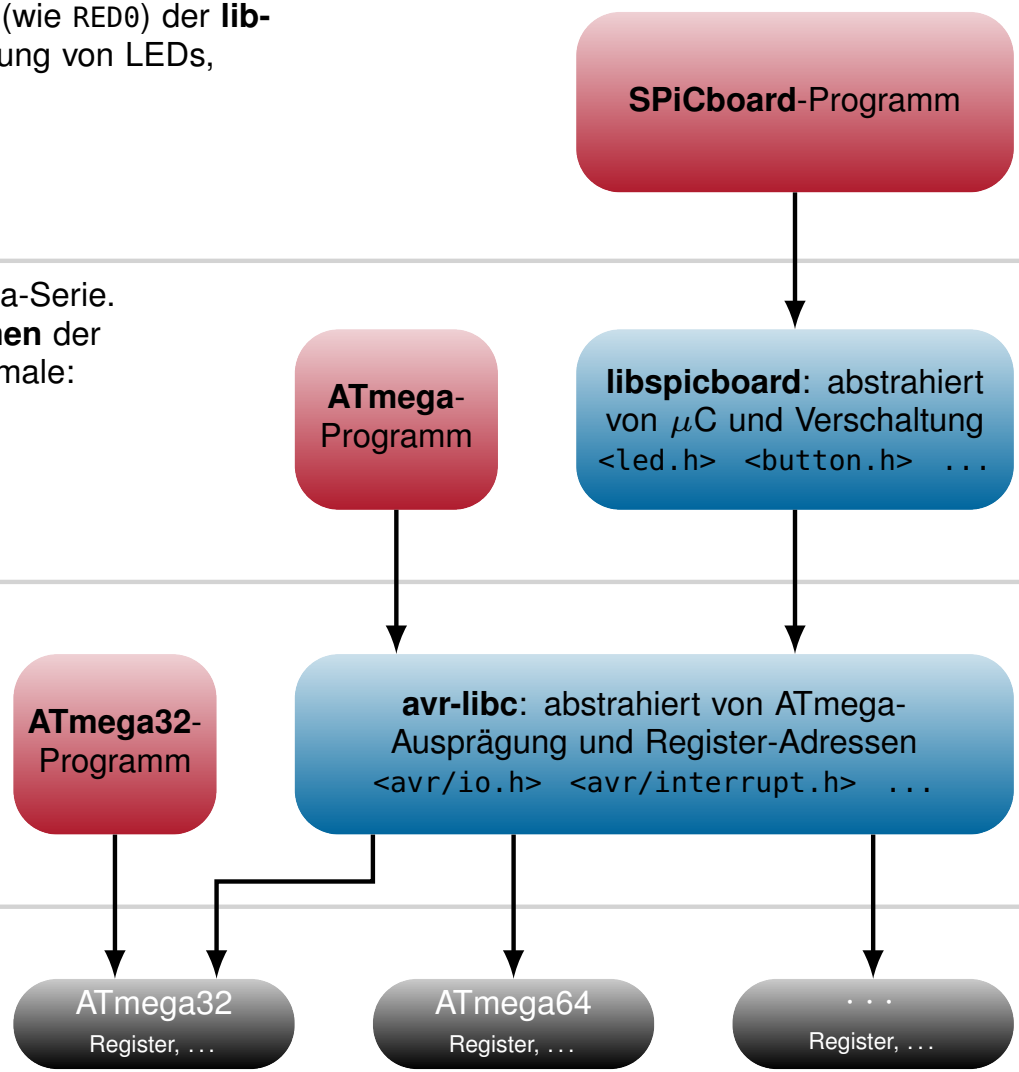
Programm läuft auf **jedem** μC der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

```
#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() {
  // initialize hardware

  // button0 on PD2
  DDRD  &= ~(1<<2);
  PORTD |=  (1<<2);
  // LED on PD7
  DDRD  |= (1<<7);
  PORTD |= (1<<7);

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

  // greet user (red LED)
  PORTD &= ~(1<<7); // PD7: low --> LED is on

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

(vgl. ↪ 3–7)

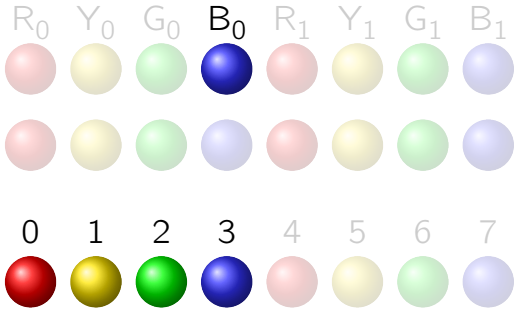
Nun: Entwicklung mit libspicboard

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

void main() {

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

  // greet user
  sb_led_on(RED0);

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

- Hardwareinitialisierung entfällt
- Programm ist einfacher und verständlicher durch **problemspezifische Abstraktionen**
  - Setze Bit 7 in `PORTD`
    ↦ `sb_set_led(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_set_all_leds(0x0f)` ⇝

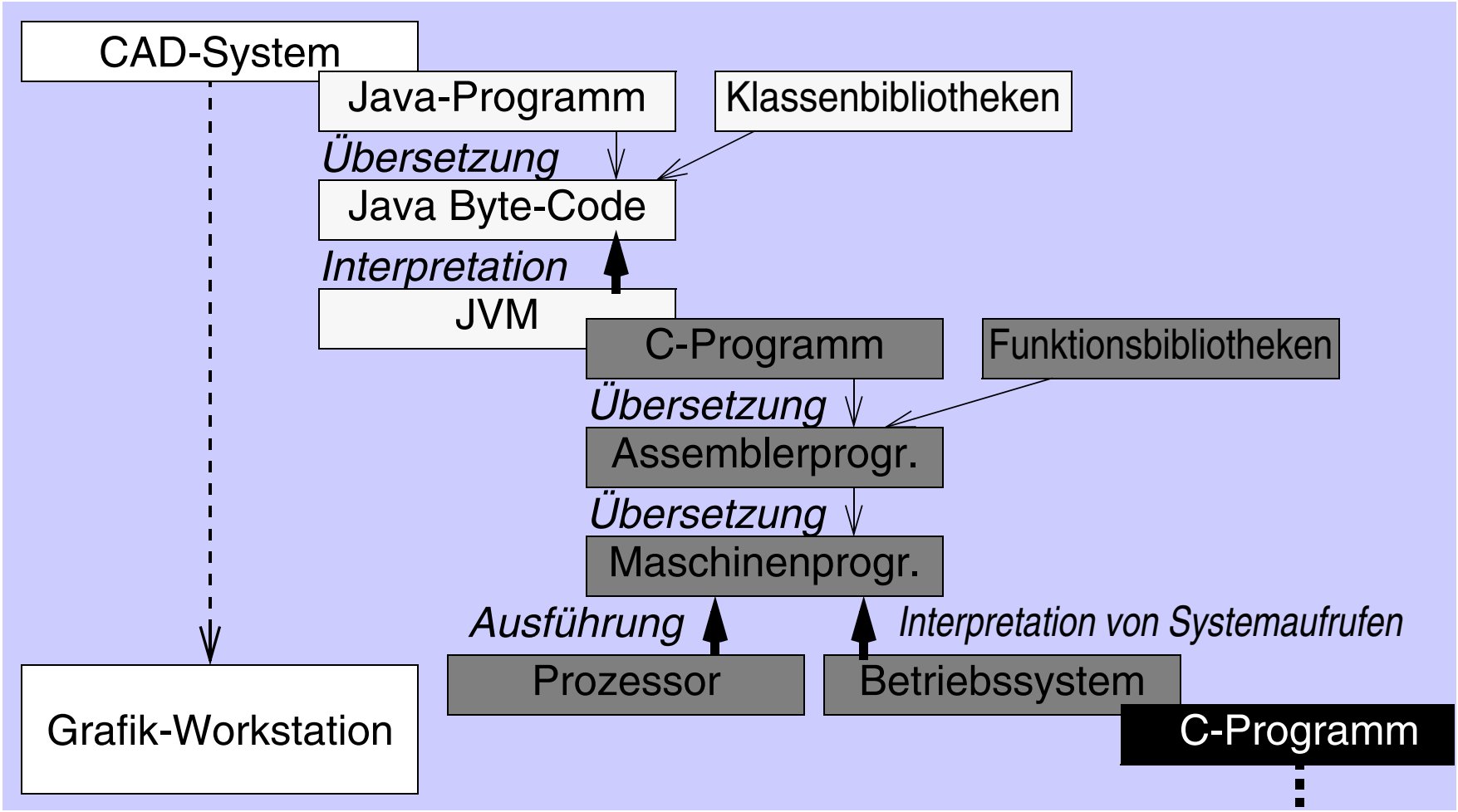


  - 7-Seg-Modul (`#include <7seg.h>`)
    - Ganzzahl $n \in \{-9 \ldots 99\}$ ausgeben: `sb_7seg_showNumber(47)` ⇝

- Eingabe-Abstraktionen (Auswahl)
  - Button-Modul (`#include <button.h>`)
    - Button-Zustand abfragen: `sb_button_getState(BUTTON0)` ↦ {`BTNPRESSED`, `BTNRELEASED`}
  - 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

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 $\leadsto$ **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 = 0;
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
```

$\rightsquigarrow$ Programm wird **abgebrochen**.

SPiCboard: Division durch 0

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

void main() {
  int a = 23;
  int b = 0;
  sei();
  b = 4711 / (a-23);
  sb_7seg_showNumber(b);

  while(1){}
}
```

Ausführen ergibt:

$\rightsquigarrow$ Programm setzt Berechnung fort mit **falschen Daten**.

Überblick: Teil B Einführung in C

Struktur eines C-Programms – allgemein

```
1  // include files
2  #include ...
3
4  // global variables
5  ... variable1 = ...
6
7  // subfunction 1
8  ... subfunction_1(...) {
9    // local variables
10   ... variable1 = ...
11   // statements
12   ...
13 }
```

```
14 // subfunction n
15 ... subfunction_n(...) {
16
17   ...
18
19 }
20
21 // main function
22 ... main(...) {
23
24   ...
25
26 }
```

- Ein C-Programm besteht (üblicherweise) aus
  - Menge von globalen Variablen
  - Menge von (Sub-)Funktionen
    - Menge von lokalen Variablen
    - Menge von Anweisungen
  - Der Funktion `main()`, in der die Ausführung beginnt

Struktur eines C-Programms – am Beispiel

```
1  // include files
2  #include <led.h>
3
4  // global variables
5  LED nextLED = RED0;
6
7  // subfunction 1
8  LED lightLED(void) {
9    if (nextLED <= BLUE1) {
10     sb_led_on(nextLED++);
11   }
12   return nextLED;
13 }
```

```
14 // subfunction 2
15 void wait(void) {
16   volatile unsigned int i;
17   for (i=0; i<0xffff; i++)
18     ;
19 }
20
21 // main function
22 void main() {
23   while (lightLED() < 8) {
24     wait();
25   }
26 }
```

- Ein C-Programm besteht (üblicherweise) aus
  - Menge von globalen Variablen — `nextLED`, Zeile 5
  - Menge von (Sub-)Funktionen — `wait()`, Zeile 15
    - Menge von lokalen Variablen — `i`, Zeile 16
    - Menge von Anweisungen — `for`-Schleife, Zeile 17
  - Der Funktion `main()`, in der die Ausführung beginnt

Bezeichner [= Java]

```
1  // include files
2  #include <led.h>
3
4  // global variables
5  LED nextLED = RED0;
6
7  // subfunction 1
8  LED lightLED(void) {
9    if (nextLED <= BLUE1) {
10     sb_led_on(nextLED++);
11   }
12   return nextLED;
13 }
```

```
14 // subfunction 2
15 void wait(void) {
16   volatile unsigned int i;
17   for (i=0; i<0xffff; i++)
18     ;
19 }
20
21 // main function
22 void main() {
23   while (lightLED() < 8) {
24     wait();
25   }
26 }
```

- Vom Entwickler vergebener Name für ein Element des Programms
  - Element: Typ, Variable, Konstante, Funktion, Sprungmarke
  - Aufbau: **[** *A-Z, a-z,* _ **]** **[** *A-Z, a-z, 0-9,* _ **]** $*$
    - Buchstabe gefolgt von Buchstaben, Ziffern und Unterstrichen
    - Unterstrich als erstes Zeichen möglich, aber reserviert für Compilerhersteller
  - Ein Bezeichner muss vor Gebrauch deklariert werden

Schlüsselwörter [≈Java]

```
1  // include files
2  #include <led.h>
3
4  // global variables
5  LED nextLED = RED0;
6
7  // subfunction 1
8  LED lightLED(void) {
9    if (nextLED <= BLUE1) {
10     sb_led_on(nextLED++);
11   }
12   return nextLED;
13 }
```

```
14 // subfunction 2
15 void wait(void) {
16   volatile unsigned int i;
17   for (i=0; i<0xffff; i++)
18     ;
19 }
20
21 // main function
22 void main() {
23   while (lightLED() < 8) {
24     wait();
25   }
26 }
```

- Reservierte Wörter der Sprache
  (⇝ dürfen nicht als Bezeichner verwendet werden)
  - Eingebaute (*primitive*) Datentypen — `unsigned int`, `void`
  - Typmodifizierer — `volatile`
  - Kontrollstrukturen — `for`, `while`
  - Elementaranweisungen — `return`

Schlüsselwörter in C99

[Handout]

- Referenz: Liste der Schlüsselwörter (bis einschließlich C99)
  - `auto`, `_Bool`, `break`, `case`, `char`, `_Complex`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `_Imaginary`, `inline`, `int`, `long`, `register`, `restrict`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`

Literale [= Java]

```
1  // include files
2  #include <led.h>
3
4  // global variables
5  LED nextLED = RED0;
6
7  // subfunction 1
8  LED lightLED(void) {
9    if (nextLED <= BLUE1) {
10     sb_led_on(nextLED++);
11   }
12   return nextLED;
13 }
```

```
14 // subfunction 2
15 void wait(void) {
16   volatile unsigned int i;
17   for (i=0; i<0xffff; i++)
18     ;
19 }
20
21 // main function
22 void main() {
23   while (lightLED() < 8) {
24     wait();
25   }
26 }
```

- (Darstellung von) Konstanten im Quelltext
  - Für jeden primitiven Datentyp gibt es eine oder mehrere Literalformen
    - Bei Integertypen: dezimal (Basis 10: `65535`), hexadezimal (Basis 16, führendes `0x`: `0xffff`), oktal (Basis 8, führende `0`: `0177777`)
  - Der Programmierer kann jeweils die am besten geeignete Form wählen
    - `0xffff` ist handlicher als `65535`, um den Maximalwert einer vorzeichenlosen 16-Bit-Ganzzahl darzustellen

Anweisungen [= Java]

```
1  // include files
2  #include <led.h>
3
4  // global variables
5  LED nextLED = RED0;
6
7  // subfunction 1
8  LED lightLED(void) {
9    if (nextLED <= BLUE1) {
10     sb_led_on(nextLED++);
11   }
12   return nextLED;
13 }
```

```
14 // subfunction 2
15 void wait(void) {
16   volatile unsigned int i;
17   for (i=0; i<0xffff; i++)
18     ;
19 }
20
21 // main function
22 void main() {
23   while (lightLED() < 8) {
24     wait();
25   }
26 }
```

- Beschreiben den eigentlichen Ablauf des Programms
- Werden hierarchisch komponiert aus drei Grundformen
  - Einzelanweisung – Ausdruck gefolgt von ;
    - einzelnes Semikolon $\mapsto$ leere Anweisung
  - Block – Sequenz von Anweisungen, geklammert durch {...}
  - Kontrollstruktur, gefolgt von Anweisung

Ausdrücke [= Java]

```
1  // include files
2  #include <led.h>
3
4  // global variables
5  LED nextLED = RED0;
6
7  // subfunction 1
8  LED lightLED(void) {
9    if (nextLED <= BLUE1) {
10     sb_led_on(nextLED++);
11   }
12   return nextLED;
13 }
```

```
14 // subfunction 2
15 void wait(void) {
16   volatile unsigned int i;
17   for (i=0; i<0xffff; i++)
18     ;
19 }
20
21 // main function
22 void main() {
23   while (lightLED() < 8) {
24     wait();
25   }
26 }
```

- Gültige Kombination von Operatoren, Literalen und Bezeichnern
  - „Gültig“ im Sinne von Syntax und Typsystem
  - Vorrangregeln für Operatoren legen die Reihenfolge fest, in der Ausdrücke abgearbeitet werden ↪ 7–14
    - Auswertungsreihenfolge kann mit Klammern ( ) explizit bestimmt werden
    - Der Compiler darf Teilausdrücke in möglichst effizienter Folge auswerten

Überblick: Teil B Einführung in C

Was ist ein Datentyp? [↪ GDI, 03-2]

- **Datentyp** := (<*Menge von Werten*>, <*Menge von Operationen*>)
  - Literal — Wert im Quelltext ↪ 5–6
  - Konstante — Bezeichner für einen Wert
  - Variable — Bezeichner für Speicherplatz, der einen Wert aufnehmen kann
  - Funktion — Bezeichner für Sequenz von Anweisungen, die einen Wert zurückgibt

  ⇝ Literale, Konstanten, Variablen, Funktionen haben einen **(Daten-)Typ**

- Datentyp legt fest
  - Repräsentation der Werte im Speicher
  - Größe des Speicherplatzes für Variablen
  - Erlaubte Operationen

- Datentyp wird festgelegt
  - Explizit, durch Deklaration, Typ-Cast oder Schreibweise (Literale)
  - Implizit, durch „Auslassung“ (⇝ int schlechter Stil!)

Primitive Datentypen in C

- Ganzzahlen/Zeichen `char`, `short`, `int`, `long`, `long long` (C99)
  - Wertebereich: implementierungsabhängig [≠Java]
    Es gilt: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`
  - Jeweils als `signed`- und `unsigned`-Variante verfügbar
- Fließkommazahlen `float`, `double`, `long double`
  - Wertebereich: implementierungsabhängig [≠Java]
    Es gilt: `float` ≤ `double` ≤ `long double`
  - Ab C99 auch als `_Complex`-Datentypen verfügbar (für komplexe Zahlen)
- Leerer Datentyp `void`
  - Wertebereich: $\emptyset$
- Boolescher Datentyp `_Bool` (C99)
  - Wertebereich: $\{0, 1\}$ ($\hookleftarrow$ letztlich ein Integertyp)
  - Bedingungsausdrücke (z. B. `if(...)`) sind in C vom Typ `int`! [≠Java]

Integertypen [≈Java][↪ GDI, 03-2]

- Integertyp — Verwendung — Literalformen
  - `char` — kleine Ganzzahl oder Zeichen — `'A'`, `65`, `0x41`, `0101`
  - `short [int]` — Ganzzahl (int ist optional) — s. o.
  - `int` — Ganzzahl „natürlicher Größe“ — s. o.
  - `long [int]` — große Ganzzahl — `65L`, `0x41L`, `0101L`
  - `long long [int]` — sehr große Ganzzahl — `65LL`, `0x41LL`, `0101LL`
- Typ-Modifizierer — werden vorangestellt — Literal-Suffix
  - signed — Typ ist vorzeichenbehaftet (Normalfall) — -
  - unsigned — Typ ist vorzeichenlos — `U`
  - const — Variable des Typs kann nicht verändert werden — -
- Beispiele (Variablendefinitionen)

```
char a                = 'A';    // char-Variable, Wert 65 (ASCII: A)
const int b           = 0x41;   // int-Konstante, Wert 65 (Hex: 0x41)
long c                = 0L;     // long-Variable, Wert 0
unsigned long int d = 22UL;   // unsigned-long-Variable, Wert 22
```

Integertypen: Größe und Wertebereich [≠Java]

- Die interne Darstellung (Bitbreite) ist implementierungsabhängig

	Datentyp-Breite in Bit				
	Java	C-Standard	gcc_{IA32}	gcc_{IA64}	gcc_{AVR}
`char`	16	≥ 8	8	8	8
`short`	16	≥ 16	16	16	16
`int`	32	≥ 16	32	32	16
`long`	64	≥ 32	32	64	32
`long long`	-	≥ 64	64	64	64

- Der Wertebereich berechnet sich aus der Bitbreite
  - signed $\quad -(2^{Bits-1}-1) \longrightarrow +(2^{Bits-1}-1)$
  - unsigned $\quad 0 \longrightarrow +(2^{Bits}-1)$

Hier zeigt sich die C-Philosophie: Effizienz durch **Maschinennähe** ↪ 3–13

Die interne Repräsentation der Integertypen ist definiert durch die **Hardware** (Registerbreite, Busbreite, etc.). Das führt im Ergebnis zu **effizientem Code**.

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

- **Problem:** Breite ($\rightsquigarrow$ Wertebereich) der C-Standardtypen ist implementierungsspezifisch $\mapsto$ **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe $\mapsto$ **Problemnähe**
  - Wertebereich sicher, aber möglichst kompakt darstellen
  - Register definierter Breite n bearbeiten
  - Code unabhängig von Compiler und Hardware halten ($\rightsquigarrow$ Portierbarkeit)
- **Lösung:** Modul `stdint.h`
  - Definiert Alias-Typen: `int`n`_t` und `uint`n`_t` für $n \in \{8, 16, 32, 64\}$
  - Wird vom Compiler-Hersteller bereitgestellt

Wertebereich `stdint.h`-Typen							
`uint8_t`	0	$\rightarrow$	255	`int8_t`	-128	$\rightarrow$	$+127$
`uint16_t`	0	$\rightarrow$	65.535	`int16_t`	-32.768	$\rightarrow$	$+32.767$
`uint32_t`	0	$\rightarrow$	4.294.967.295	`int32_t`	$-2.147.483.648$	$\rightarrow$	$+2.147.483.647$
`uint64_t`	0	$\rightarrow$	$> 1,8 * 10^{19}$	`int64_t`	$< -9,2 * 10^{18}$	$\rightarrow$	$> +9,2 * 10^{18}$

Typ-Aliase mit `typedef` [≠Java]

- Mit dem `typedef`-Schlüsselwort definiert man einen Typ-Alias:

  `typedef` *Typausdruck* *Bezeichner*`;`

  - *Bezeichner* ist nun ein alternativer Name für *Typausdruck*
  - Kann überall verwendet werden, wo ein Typausdruck erwartet wird

```
// stdint.h (avr-gcc)                  // stdint.h (x86-gcc, IA32)
typedef unsigned char uint8_t;         typedef unsigned char  uint8_t;
typedef unsigned int  uint16_t;        typedef unsigned short uint16_t;
...                                    ...
```

```
// main.c
#include <stdint.h>

uint16_t counter = 0;     // global 16-bit counter, range 0-65535
...
typedef uint8_t Register; // Registers on this machine are 8-bit
...
```

Typ-Aliase mit `typedef` (Forts.) [≠Java]

- Typ-Aliase ermöglichen einfache problembezogene Abstraktionen
  - `Register` ist problemnäher als `uint8_t`
    ⇝ Spätere Änderungen (z. B. auf 16-Bit-Register) zentral möglich
  - `uint16_t` ist problemnäher als `unsigned char`
  - `uint16_t` ist sicherer als `unsigned char`

Definierte Bitbreiten sind bei der μC-Entwicklung sehr wichtig!

- Große Unterschiede zwischen Plattformen und Compilern
  ⇝ Kompatibilitätsprobleme
- Um Speicher zu sparen, sollte immer der **kleinstmögliche** Integertyp verwendet werden

Regel: Bei der systemnahen Programmierung werden Typen aus `stdint.h` verwendet!

Aufzählungstypen mit `enum` [≈Java]

- Mit dem `enum`-Schlüsselwort definiert man einen Aufzählungstyp über eine explizite Menge symbolischer Werte:

  `enum` *Bezeichner*$_{opt}$ `{` *KonstantenListe* `} ;`

- Beispiel
  - Definition:

```
enum eLED {RED0, YELLOW0, GREEN0, BLUE0,
           RED1, YELLOW1, GREEN1, BLUE1};
```

  - Verwendung:

```
enum eLED myLed = YELLOW0;  // enum necessary here!
...
sb_led_on(BLUE1);
```

- Vereinfachung der Verwendung durch `typedef`
  - Definition:

```
typedef enum eLED {RED0, YELLOW0, GREEN0, BLUE0,
                   RED1, YELLOW1, GREEN1, BLUE1} LED;
```

  - Verwendung:

```
LED myLed = YELLOW0;  // LED --> enum eLED
```


enum ↦ int [≠Java]

- Technisch sind enum-Typen Integers (int)
  - enum-Konstanten werden von 0 an durchnummeriert

```
typedef enum { RED0,      // value: 0
               YELLOW0,   // value: 1
               GREEN0,    // value: 2
               ··· } LED;
```

  - Es ist auch möglich, Werte direkt zuzuweisen

```
typedef enum { BUTTON0 = 4, BUTTON1 = 8 } BUTTON;
```

  - Man kann sie verwenden wie ints (z. B. mit ihnen rechnen)

```
sb_led_on(RED0 + 2);  // -> LED GREEN0 is on
sb_led_on(1);         // -> LED YELLOW0 is on
for( int led = RED0, led <= BLUE1; led++ )
  sb_led_off(led);    // turn off all LEDs
// Also possible...
sb_led_on(4711);      // no compiler/runtime error!
```

- ⇝ Es findet **keinerlei Typprüfung** statt!

Das entspricht der **C-Philosophie**! ↪ 3–13

Fließkommatypen [≈Java]

- Fließkommatyp — Verwendung — Literalformen
  - `float` — einfache Genauigkeit (≈ 7 St.) — `100.0F`, `1.0E2F`
  - `double` — doppelte Genauigkeit (≈ 15 St.) — `100.0`, `1.0E2`
  - `long double` — „erweiterte Genauigkeit“ — `100.0L` `1.0E2L`
- Genauigkeit / Wertebereich sind implementierungsabhängig [≠Java]
  - Es gilt: `float` ≤ `double` ≤ `long double`
  - `long double` und `double` sind auf vielen Plattformen identisch

„Effizienz durch Maschinennähe“ ↪ 3–13

Fließkommazahlen + μC-Plattform = $$$

- Oft keine Hardwareunterstützung für `float`-Arithmetik
  ⇝ **sehr teure** Emulation in Software (langsam, viel zusätzlicher Code)
- Speicherverbrauch von `float`- und `double`-Variablen ist **sehr hoch**
  ⇝ mindestens 32/64 Bit (`float`/`double`)

Regel: Bei der μ-Controller-Programmierung ist auf Fließkommaarithmetik **zu verzichten**!

Zeichen $\mapsto$ Integer [$\approx$Java]

- Zeichen sind in C ebenfalls Ganzzahlen (Integers) $\hookrightarrow$ 6–3
  - `char` gehört zu den Integer-Typen (üblicherweise 8 Bit = 1 Byte)
- Repräsentation erfolgt durch den ASCII-Code $\hookrightarrow$ 6–12
  - 7-Bit-Code $\mapsto$ 128 Zeichen standardisiert
    (die verbleibenden 128 Zeichen werden unterschiedlich interpretiert)
  - Spezielle Literalform durch Hochkommata
    `'A'` $\mapsto$ ASCII-Code von `A`
  - Nichtdruckbare Zeichen durch Escape-Sequenzen
    - Tabulator `'\t'`
    - Zeilentrenner `'\n'`
    - Backslash `'\\'`
- Zeichen $\mapsto$ Integer $\rightsquigarrow$ man kann mit Zeichen rechnen

```
char b = 'A' + 1;          // b: 'B'

int lower(int ch) {        // lower('X'): 'x'
  return ch + 0x20;
}
```

ASCII-Code-Tabelle (7 Bit)

ASCII $\mapsto$ *American Standard Code for Information Interchange*

NUL 00	**SOH** 01	**STX** 02	**ETX** 03	**EOT** 04	**ENQ** 05	**ACK** 06	**BEL** 07
BS 08	**HT** 09	**NL** 0A	**VT** 0B	**NP** OC	**CR** 0D	**SO** 0E	**SI** 0F
DLE 10	**DC1** 11	**DC2** 12	**DC3** 13	**DC4** 14	**NAK** 15	**SYN** 16	**ETB** 17
CAN 18	**EM** 19	**SUB** 1A	**ESC** 1B	**FS** 1C	**GS** 1D	**RS** 1E	**US** 1F
SP 20	**!** 21	**"** 22	**#** 23	**$** 24	**%** 25	**&** 26	**'** 27
(28	**)** 29	***** 2A	**+** 2B	**,** 2C	**-** 2D	**.** 2E	**/** 2F
0 30	**1** 31	**2** 32	**3** 33	**4** 34	**5** 35	**6** 36	**7** 37
8 38	**9** 39	**:** 3A	**;** 3B	**<** 3C	**=** 3D	**>** 3E	**?** 3F
@ 40	**A** 41	**B** 42	**C** 43	**D** 44	**E** 45	**F** 46	**G** 47
H 48	**I** 49	**J** 4A	**K** 4B	**L** 4C	**M** 4D	**N** 4E	**O** 4F
P 50	**Q** 51	**R** 52	**S** 53	**T** 54	**U** 55	**V** 56	**W** 57
X 58	**Y** 59	**Z** 5A	**[** 5B	**** 5C	**]** 5D	**^** 5E	**_** 5F
` 60	**a** 61	**b** 62	**c** 63	**d** 64	**e** 65	**f** 66	**g** 67
h 68	**i** 69	**j** 6A	**k** 6B	**l** 6C	**m** 6D	**n** 6E	**o** 6F
p 70	**q** 71	**r** 72	**s** 73	**t** 74	**u** 75	**v** 76	**w** 77
x 78	**y** 79	**z** 7A	**{** 7B	**|** 7C	**}** 7D	**~** 7E	**DEL** 7F

Zeichenketten (Strings) [≠Java]

- Ein String ist in C ein Feld (Array) von Zeichen
  - Repräsentation: Folge von Einzelzeichen, terminiert durch (letztes Zeichen): `NUL` (ASCII-Wert 0)
  - Speicherbedarf: (Länge + 1) Bytes
  - Datentyp: `char[]` oder `char*` (synonym)
- Spezielle Literalform durch doppelte Hochkommata:

- Beispiel (Linux)

```
#include <stdio.h>
char[] string = "Hello, World!\n";
int main(){
  printf(string);
  return 0;
}
```

Zeichenketten brauchen vergleichsweise viel Speicher und „größere“ Ausgabegeräte (z. B. LCD-Display).
⇝ Bei der μC-Programmierung spielen sie nur eine untergeordnete Rolle.

Ausblick: Komplexe Datentypen

- Aus einfachen Datentypen lassen sich (rekursiv) auch komplexe(re) Datentypen bilden
  - Felder (Arrays) $\hookrightarrow$ Sequenz von Elementen gleichen Typs [≈Java]

```
int intArray[4];          // allocate array with 4 elements
intArray[0] = 0x4711;     // set 1st element (index 0)
```

  - Zeiger $\hookrightarrow$ veränderbare Referenzen auf Variablen [≠Java]

```
int  a = 0x4711;          // a: 0x4711
int *b = &a;              // b: -->a (memory location of a)
int  c = *b;              // pointer dereference (c: 0x4711)
*b     = 23;              // pointer dereference (a: 23)
```

  - Strukturen $\hookrightarrow$ Verbund von Elementen bel. Typs [≠Java]

```
struct Point { int x; int y; };
struct Point p;           // p is Point variable
p.x = 0x47;               // set x-component
p.y = 0x11;               // set y-component
```

- Wir betrachten diese detailliert in späteren Kapiteln

Überblick: Teil B Einführung in C

Arithmetische Operatoren [= Java]

- Stehen für alle Ganzzahl- und Fließkommatypen zur Verfügung

$+$	Addition	
$-$	Subtraktion	
$\star$	Multiplikation	
/	Division	
unäres $-$	negatives Vorzeichen (z. B. $-a$)	$\leadsto$ Multiplikation mit -1
unäres $+$	positives Vorzeichen (z. B. $+3$)	$\leadsto$ kein Effekt

- Zusätzlich nur für Ganzzahltypen:

%	Modulo (Rest bei Division)

Inkrement-/Dekrement-Operatoren [=Java][↪ GDI, 07-02]

- Stehen für Ganzzahltypen und Zeigertypen zur Verfügung

++	Inkrement (Erhöhung um 1)
−−	Dekrement (Verminderung um 1)

- Linksseitiger Operator (Präfix) ++x bzw. --x
  - Erst wird der Inhalt von x verändert
  - Dann wird der (neue) Inhalt von x als Ergebnis geliefert
- Rechtsseitiger Operator (Postfix) x++ bzw. x--
  - Erst wird der (alte) Inhalt von x als Ergebnis geliefert
  - Dann wird der Inhalt von x verändert
- Beispiele

```
a = 10;
b = a++;  // b: 10, a: 11
c = ++a;  // c: 12, a: 12
```

Vergleichsoperatoren [=Java][↪ GDI, 03-09]

- Vergleichen von zwei Ausdrücken

$<$	kleiner
$<=$	kleiner gleich
$>$	größer
$>=$	größer gleich
$==$	gleich (zwei Gleichheitszeichen!)
$!=$	ungleich

- Beachte: Ergebnis ist vom Typ `int` [≠Java]
  - Ergebnis: *falsch* $\mapsto$ 0
    wahr $\mapsto$ 1
  - Man kann mit dem Ergebnis rechnen
- Beispiele

```
if (a >= 3) {···}
if (a == 3) {···}
return a * (a > 0);   // return 0 if a is negative
```


Logische Operatoren [≈Java][↪ GDI, 03-10]

- Verknüpfung von Wahrheitswerten (wahr / falsch), kommutativ

&&	„und“ (Konjunktion)	*wahr*	&&	*wahr*	→	*wahr*
		wahr	&&	*falsch*	→	*falsch*
		falsch	&&	*falsch*	→	*falsch*
||	„oder“ (Disjunktion)	*wahr*	||	*wahr*	→	*wahr*
		wahr	||	*falsch*	→	*wahr*
		falsch	||	*falsch*	→	*falsch*
!	„nicht“ (Negation, unär)		!	*wahr*	→	*falsch*
			!	*falsch*	→	*wahr*

- Beachte: Operanden und Ergebnis sind vom Typ `int` [≠Java]
  - Operanden (Eingangsparameter): $0 \mapsto$ *falsch*, $\neq 0 \mapsto$ *wahr*
  - Ergebnis: *falsch* $\mapsto 0$, *wahr* $\mapsto 1$

Logische Operatoren – Auswertung [=Java]

- Die Auswertung eines logischen Ausdrucks wird abgebrochen, sobald das Ergebnis feststeht
  - Sei `int a = 5; int b = 3; int c = 7;`

$$\underbrace{\underbrace{a > b}_{1} \;||\; \underbrace{a > c}_{?}}_{1}$$

⟵ wird nicht ausgewertet, da der erste Term bereits *wahr* ergibt

$$\underbrace{\underbrace{a > c}_{0} \;\&\&\; \underbrace{a > b}_{?}}_{0}$$

⟵ wird nicht ausgewertet, da der erste Term bereits *falsch* ergibt

- Kann überraschend sein, wenn Teilausdrücke Nebeneffekte haben

```
int a = 5; int b = 3; int c = 7;
if ( a > c && !func(b) ) {···}      // func() will not be called
```

Zuweisungsoperatoren [=Java]

- Allgemeiner Zuweisungsoperator (=)
  - Zuweisung eines Wertes an eine Variable
  - Beispiel: `a = b + 23`
- Arithmetische Zuweisungsoperatoren (+=, −=, ...)
  - Abgekürzte Schreibweise zur Modifikation des Variablenwerts
  - Beispiel: `a += 23` ist äquivalent zu `a = a + 23`
  - Allgemein: `a` *op*`= b` ist äquivalent zu `a = a` *op* `b`
    für $op \in \{ +, -, \star, \%, <<, >>, \&, \hat{}, | \}$
- Beispiele

```
int a = 8;
a += 8;     // a: 16
a %= 3;     // a: 1
```

Zuweisungen sind Ausdrücke!

- Zuweisungen können in komplexere Audrücke geschachtelt werden
  - Das Ergebnis eines Zuweisungsausdrucks ist der zugewiesene Wert

```
int a, b, c;
a = b = c = 1;  // c: 1, b: 1, a: 1
```

  - Die Verwendung von Zuweisungen in beliebigen Ausdrücken führt zu **Nebeneffekten**, die nicht immer offensichtlich sind

```
a += b += c;    // Value of a and b?
```

Besonders gefährlich: Verwendung von = statt ==

In C sind Wahrheitswerte Integers: $0 \mapsto$ *falsch*, $\emptyset \mapsto$ *wahr*

- Typischer „Anfängerfehler“ in Kontrollstrukturen:

```
if (a = 6) {···} else {···} // BUG: if-branch is always taken!!!
```

- Compiler beanstandet das Konstrukt nicht, es handelt sich um einen gültigen Ausdruck! $\leadsto$ Fehler wird leicht übersehen!

Bitoperationen [= Java]

- Bitweise Verknüpfung von Ganzzahltypen, kommutativ

&	bitweises „Und“ (Bit-Schnittmenge)	$1 \,\&\, 1 \rightarrow 1$ $1 \,\&\, 0 \rightarrow 0$ $0 \,\&\, 0 \rightarrow 0$
|	bitweises „Oder“ (Bit-Vereinigungsmenge)	$1 \mid 1 \rightarrow 1$ $1 \mid 0 \rightarrow 1$ $0 \mid 0 \rightarrow 0$
$\wedge$	bitweises „Exklusiv-Oder“ (Bit-Antivalenz)	$1 \wedge 1 \rightarrow 0$ $1 \wedge 0 \rightarrow 1$ $0 \wedge 0 \rightarrow 0$
~	bitweise Inversion (Einerkomplement, unär)	$\sim 1 \rightarrow 0$ $\sim 0 \rightarrow 1$

Bitoperationen (Forts.) [=Java]

- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

  `<<` bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)

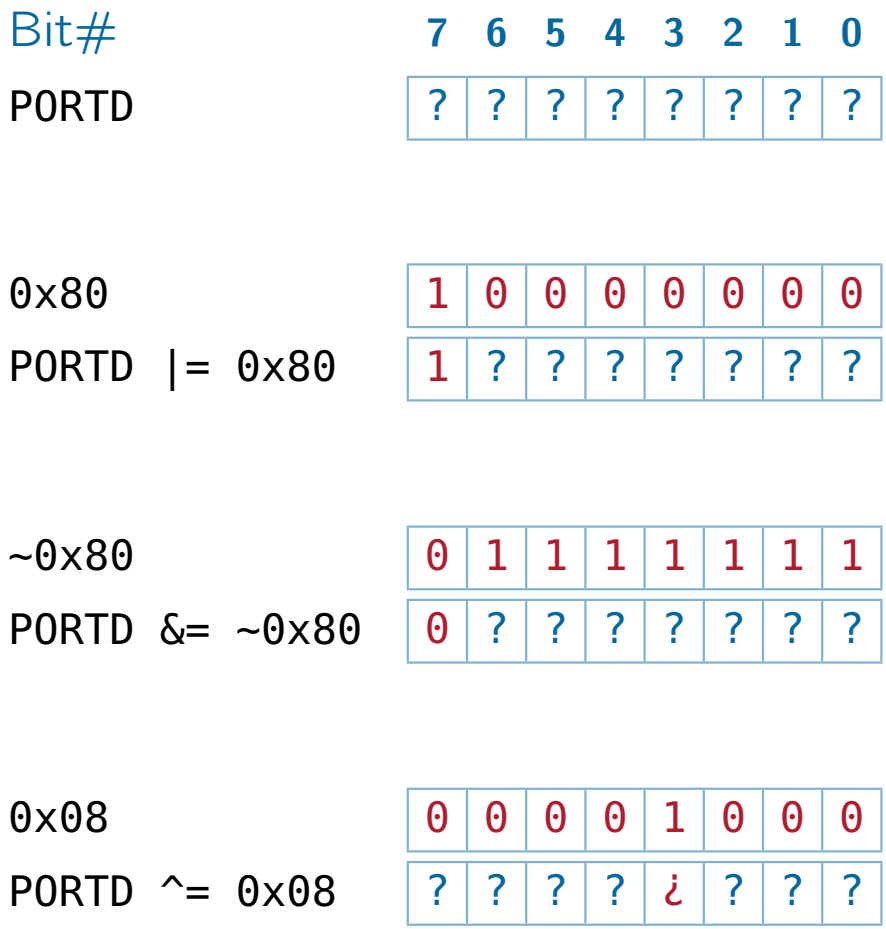
  `>>` bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (`x` sei vom Typ `uint8_t`)

Bit#	7	6	5	4	3	2	1	0	
`x=156`	1	0	0	1	1	1	0	0	`0x9c`
`~x`	0	1	1	0	0	0	1	1	`0x63`
`7`	0	0	0	0	0	1	1	1	`0x07`
`x | 7`	1	0	0	1	1	1	1	1	`0x9f`
`x & 7`	0	0	0	0	0	1	0	0	`0x04`
`x ^ 7`	1	0	0	1	1	0	1	1	`0x9B`
`x << 2`	0	1	1	1	0	0	0	0	`0x70`
`x >> 1`	0	1	0	0	1	1	1	0	`0x4e`

Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen



Bit 7 soll verändert werden, die anderen Bits jedoch erhalten bleiben!

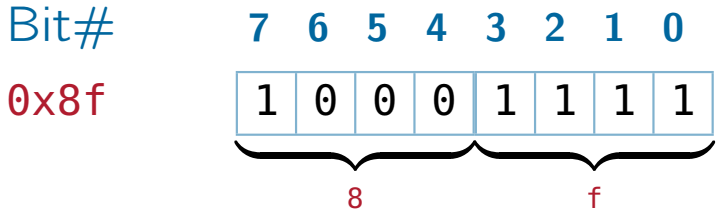
Setzen eines Bits durch **Ver-odern** mit Maske, in der nur das Zielbit 1 ist

Löschen eines Bits durch **Ver-unden** mit Maske, in der nur das Zielbit 0 ist

Invertieren eines Bits durch **Ver-xodern** mit Maske, in der nur das Zielbit 1 ist

Bitoperationen – Anwendung (Forts.)

- Bitmasken werden gerne als Hexadezimal-Literale angegeben



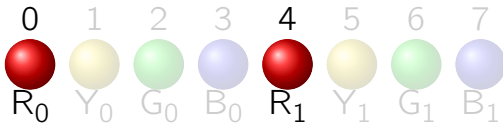
Jede Hex-Ziffer repräsentiert genau ein Halb-Byte (*Nibble*) ⇝ Verständlichkeit

- Für „Dezimal-Denker“ bietet sich die Linksschiebe-Operation an

```
PORTD |= (1<<7);    // set bit 7:  1<<7 --> 10000000
PORTD &= ~(1<<7);   // mask bit 7: ~(1<<7) --> 01111111
```

- Zusammen mit der Oder-Operation auch für komplexere Masken

```
#include <led.h>
void main() {
  uint8_t mask = (1<<RED0) | (1<<RED1);

  sb_led_set_all_leds (mask);

  while(1) ;
}
```

⇝ 

Bedingte Auswertung [≈Java][↪ GDI, 07-10]

- Formulierung von Bedingungen in Ausdrücken

  $Ausdruck_1$ **?** $Ausdruck_2$ **:** $Ausdruck_3$

  - Zunächst wird $Ausdruck_1$ ausgewertet
    - $Ausdruck_1 \neq 0$ (*wahr*) $\rightsquigarrow$ Ergebnis ist $Ausdruck_2$
    - $Ausdruck_1 = 0$ (*falsch*) $\rightsquigarrow$ Ergebnis ist $Ausdruck_3$
  - `?:` ist der einzige ternäre (dreistellige) Operator in C

- Beispiel

```
int abs(int a) {
  // if (a<0) return -a; else return a;
  return (a<0) ? -a : a;
}
```


Sequenzoperator

[≠ Java]

- Reihung von Ausdrücken

  $Ausdruck_1$ **,** $Ausdruck_2$

  - Zunächst wird $Ausdruck_1$ ausgewertet
    $\leadsto$ Nebeneffekte von $Ausdruck_1$ werden sichtbar
  - Ergebnis ist der Wert von $Ausdruck_2$

- Verwendung des Komma-Operators ist selten erforderlich!
  (Präprozessor-Makros mit Nebeneffekten)

Vorrangregeln bei Operatoren [≈Java][↪ GDI, 03-07]

	Klasse	Operatoren	Assoziativität

1	Funktionsaufruf, Feldzugriff
	Strukturzugriff
	Post-Inkrement/-Dekrement
		`x() x[]`
		`x.y x->y`
		`x++ x--`
			links → rechts

2	Prä-Inkrement/-Dekrement
	unäre Operatoren
	Adresse, Verweis (Zeiger)
	Typkonvertierung (cast)
	Typgröße
		`++x --x`
		`+x -x ~x !x`
		`& *`
		`(<Typ>)x`
		`sizeof(x)`
			rechts → links

3	Multiplikation, Division, Modulo	`* / %`	links → rechts
4	Addition, Subtraktion	`+ -`	links → rechts
5	Bitweises Schieben	`>> <<`	links → rechts
6	Relationaloperatoren	`< <= > >=`	links → rechts
7	Gleichheitsoperatoren	`== !=`	links → rechts
8	Bitweises UND	`&`	links → rechts
9	Bitweises OR	`|`	links → rechts
10	Bitweises XOR	`^`	links → rechts
11	Konjunktion	`&&`	links → rechts
12	Disjunktion	`||`	links → rechts
13	Bedingte Auswertung	`?:=`	rechts → links
14	Zuweisung	`= op=`	rechts → links
15	Sequenz	`,`	links → rechts

Typumwandlung in Ausdrücken

- Ein Ausdruck wird *mindestens* mit `int`-Wortbreite berechnet
  - `short`- und `signed char`-Operanden werden implizit „aufgewertet“ (↪ *Integer Promotion*)
  - Erst das Ergebnis wird auf den Zieldatentyp abgeschnitten/erweitert

```
int8_t a=100, b=3, c=4, res;  // range: -128 --> +127

res   = a * b  / c;           // promotion to int: 300 fits in!
```

int8_t: 75 — int: 300 — int: 75

- Generell wird die *größte* beteiligte Wortbreite verwendet ↪ 6–3

```
int8_t a=100, b=3, res;  // range: -128 --> +127
int32_t c=4;             // range: -2147483648 --> +2147483647

res   =  a * b   / c;    // promotion to int32_t
```

int8_t: 75 — int32_t: 300 — int32_t: 75

Typumwandlung in Ausdrücken (Forts.)

- Fließkomma-Typen gelten dabei als „größer“ als Ganzzahl-Typen

```
int8_t a=100, b=3, res;          // range: -128 --> +127

res  =  a * b  /  4.0  ; // promotion to double
```

int8_t: 75 | double: 300.0 | double: 4.0 | double: 75.0

- `unsigned`-Typen gelten dabei als „größer“ als `signed`-Typen

```
int s = -1, res;              // range: -32768 --> +32767
unsigned u = 1;               // range: 0 --> 65535

res  =    s     < u;    // promotion to unsigned: -1 --> 65535
```

int: 0 | unsigned: 65535 | unsigned: 0

↝ Überraschende Ergebnisse bei negativen Werten!

↝ Mischung von `signed`- und `unsigned`-Operanden vermeiden!

Typumwandlung in Ausdrücken – Typ-Casts

- Durch den Typ-Cast-Operator kann man einen Ausdruck gezielt in einen anderen Typ konvertieren

(*Typ***)** *Ausdruck*

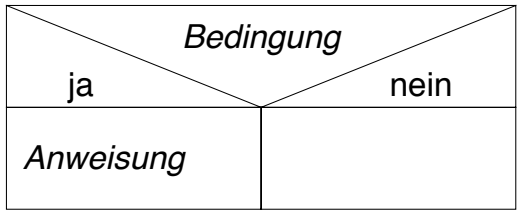
```
int s = -1, res;        // range: -32768 --> +32767
unsigned u = 1;         // range: 0 --> 65535

res  = s < (int) u ;    // cast u to int
int: 1      int: 1
      int: 1
```

Überblick: Teil B Einführung in C

Bedingte Anweisung [= Java][↪ GDI, 07-02]

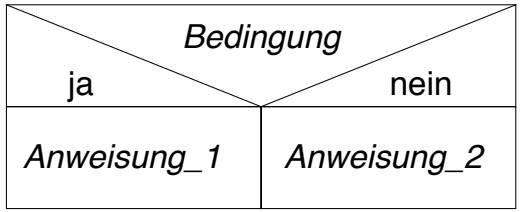
- `if`-Anweisung (bedingte Anweisung)

```
if ( Bedingung )
  Anweisung;
```



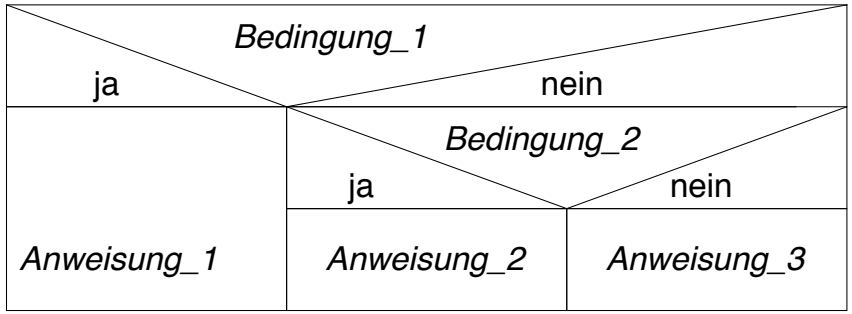
- `if-else`-Anweisung (einfache Verzweigung)

```
if ( Bedingung )
  Anweisung_1;
else
  Anweisung_2;
```



- `if-else-if`-Kaskade (mehrfache Verzweigung)

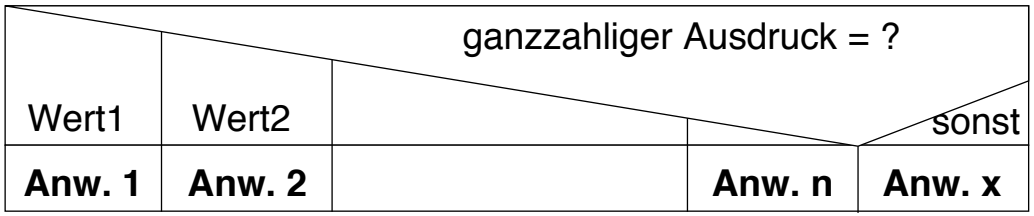
```
if ( Bedingung_1 )
  Anweisung_1;
else if ( Bedingung_2 )
  Anweisung_2;
else
  Anweisung_3;
```



Fallunterscheidung [= Java]

`switch`-Anweisung (Fallunterscheidung)

- Alternative zur `if`-Kaskade bei Test auf Ganzzahl-Konstanten



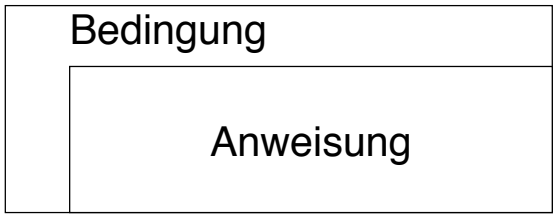

```
switch ( Ausdruck ) {
  case Wert_1:
    Anweisung_1;
    break;
  case Wert_2:
    Anweisung_2;
    break;
  ...
  case Wert_n:
    Anweisung_n;
    break;
  default:
    Anweisung_x;
}
```


Abweisende und nicht-abweisende Schleife [=Java]

Abweisende Schleife

[↪ GDI, 08-05]

- `while`-Schleife
- Null- oder mehrfach ausgeführt



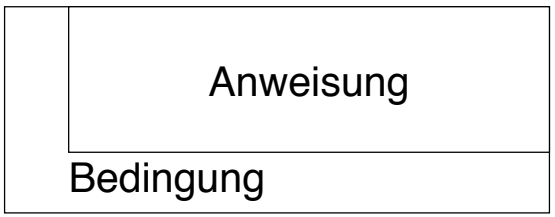

```
while( Bedingung )
  Anweisung;
```

```
while (
  sb_button_getState(BUTTON0)
  == BTNRELEASED
) {
  ··· // do unless button press.
}
```

Nicht-abweisende Schleife

[↪ GDI, 08-07]

- `do-while`-Schleife
- Ein- oder mehrfach ausgeführt



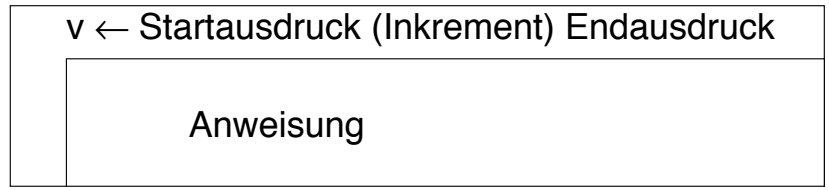

```
do
  Anweisung;
while( Bedingung );
```

```
do {
  ··· // do at least once
} while (
  sb_button_getState(BUTTON0)
  == BTNRELEASED
);
```

Zählende Schleife [=Java][↪ GDI, 08-02]

- `for`-Schleife (Laufanweisung)

```
for ( Startausdruck;
      Endausdruck;
      Inkrement–Ausdruck  )
  Anweisung;
```



- Beispiel (übliche Verwendung: *n* Ausführungen mit Zählvariable)

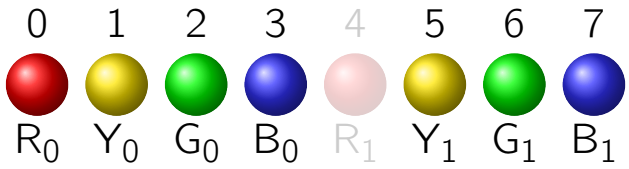
```
uint8_t sum = 0; // calc sum 1+...+10
for (uint8_t n = 1; n < 11; n++) {
  sum += n;
}
sb_7seg_showNumber( sum );
```

- Anmerkungen
  - Die Deklaration von Variablen (`n`) im *Startausdruck* ist erst ab C99 möglich
  - Die Schleife wird wiederholt, solange *Endausdruck* $\neq 0$ (*wahr*)
    $\rightsquigarrow$ die `for`-Schleife ist eine „verkappte“ `while`-Schleife

Schleifensteuerung [= Java][↪ GDI, 08-09]

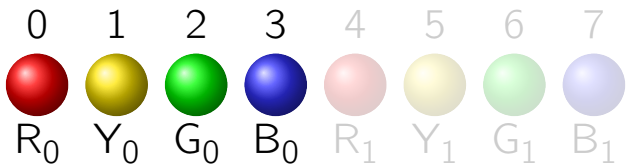
- Die `continue`-Anweisung beendet den aktuellen Schleifendurchlauf
  ⇝ Schleife wird mit dem nächsten Durchlauf fortgesetzt

```
for( uint8_t led=0; led < 8; ++led ) {
  if( led == RED1 ) {
    continue;          // skip RED1
  }
  sb_led_on(led);
}
```



- Die `break`-Anweisung verlässt die (innerste) Schleife
  ⇝ Programm wird *nach* der Schleife fortgesetzt

```
for( uint8_t led=0; led < 8; ++led ) {
  if( led == RED1 ) {
    break;           // break at RED1
  }
  sb_led_on(led);
}
```



Überblick: Teil B Einführung in C

Was ist eine Funktion?

- **Funktion** := Unterprogramm [↪ GDI, 11-01]
  - Programmstück (Block) mit einem Bezeichner
  - Beim Aufruf können Parameter übergeben werden
  - Bei Rückkehr kann ein Rückgabewert zurückgeliefert werden
- Funktionen sind elementare Programmbausteine
  - Gliedern umfangreiche Aufgaben in kleine, beherrschbare Komponenten
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)

Funktion ↦ Abstraktion ↪ 4–1

- Bezeichner und Parameter abstrahieren
  - Vom tatsächlichen Programmstück
  - Von der Darstellung und Verwendung von Daten
- Ermöglicht schrittweise Abstraktion und Verfeinerung

Beispiel

- Funktion (Abstraktion) `sb_led_set_all_leds()`

```
#include <led.h>
void main() {
  sb_led_set_all_leds( 0xaa );
  while(1) {}
}
```

- Implementierung in der `libspicboard`

```
void sb_led_set_all_leds(uint8_t setting)
{
  uint8_t i = 0;
  for (i = 0; i < 8; i++) {
    if (setting & (1<<i)) {
      sb_led_on(i);
    } else {
      sb_led_off(i);
    }
  }
}
```

Sichtbar: Bezeichner und formale Paramter

Unsichtbar: Tatsächliche Implementierung

Funktionsdefinition [≈Java]

- Syntax: *Typ Bezeichner* **(** *FormaleParam*$_{opt}$ **)** **{***Block***}**
  - *Typ* — Typ des Rückgabewertes der Funktion, `void` falls kein Wert zurückgegeben wird [=Java]
  - *Bezeichner* — Name, unter dem die Funktion aufgerufen werden kann ↪ 5–3 [=Java]
  - *FormaleParam*$_{opt}$ — Liste der formalen Parameter: *Typ*$_1$ *Bez*$_{1\,opt}$**,** **...,** *Typ*$_n$ *Bez*$_{n\,opt}$ (Parameter-Bezeichner sind optional) [=Java]
    `void`, falls kein Parameter erwartet wird [≠Java]
  - {*Block*} — Implementierung; formale Parameter stehen als lokale Variablen bereit [=Java]
- Beispiele:

```
int max( int a, int b ) {
  if (a>b) return a;
  return b;
}
```

```
void wait( void ) {
  volatile uint16_t w;
  for( w = 0; w<0xffff; w++ ) {
  }
}
```


Funktionsaufruf [=Java]

- Syntax: *Bezeichner* **(** *TatParam* **)**
  - *Bezeichner* — Name der Funktion, in die verzweigt werden soll [=Java]
  - *TatParam* — Liste der tatsächlichen Parameter (übergebene Werte, muss anzahl- und typkompatibel sein zur Liste der formalen Parameter) [=Java]
- Beispiele:

```
int x = max( 47, 11 );
```

Aufruf der `max()`-Funktion. `47` und `11` sind die tatsächlichen Parameter, welche nun den formalen Parametern `a` und `b` der `max()`-Funktion (↪ 9–3) zugewiesen werden.

```
char[] text = "Hello, World";
int x = max( 47, text );
```

Fehler: `text` ist nicht `int`-konvertierbar (tatsächlicher Parameter 2 passt nicht zu formalem Parameter `b` ↪ 9–3)

```
max( 48, 12 );
```

Der Rückgabewert darf ignoriert werden (was hier nicht wirklich Sinn ergibt)

Funktionsaufruf – Parameterübergabe [≠Java]

- Generelle Arten der Parameterübergabe [↪ GDI, 14-01]
  - *Call-by-value*: Die formalen Parameter sind Kopien der tatsächlichen Parameter. Änderungen in den formalen Parametern gehen mit Verlassen der Funktion verloren. **Dies ist der Normalfall in C.**
  - *Call-by-reference*: Die formalen Parameter sind Verweise (Referenzen) auf die tatsächlichen Parameter. Änderungen in den formalen Parametern betreffen auch die tatsächlichen Parameter. **In C nur indirekt über Zeiger möglich.** ↪ 13–5
- Des weiteren gilt
  - Arrays werden in C immer *by-reference* übergeben [=Java]
  - Die Auswertungsreihenfolge der Parameter ist undefiniert! [≠Java]

Funktionsaufruf – Rekursion [=Java]

- Funktionen können sich auch selber aufrufen (Rekursion)

```
int fak( int n ) {
  if (n > 1)
    return n * fak(n-1);
  return 1;
}
```

Rekursive Definition der Fakultätsfunktion.

Ein anschauliches, aber mieses Beispiel für den Einsatz von Rekursion!

Rekursion $\mapsto$ $$$

Rekursion verursacht erhebliche Laufzeit- und Speicherkosten!
Pro Rekursionsschritt muss:

- Speicher bereit gestellt werden für Rücksprungadresse, Parameter und alle lokalen Variablen
- Parameter kopiert und ein Funktionsaufruf durchgeführt werden

Regel: Bei der systemnahen Softwareentwicklung wird möglichst auf **Rekursion verzichtet!**

Funktionsdeklaration [≠Java]

- Funktionen müssen vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine voranstehende Definition beinhaltet bereits die Deklaration
  - Ansonsten (falls die Funktion „weiter hinten“ im Quelltext oder in einem anderen Modul definiert wird) muss sie explizit deklariert werden
- Syntax: *Bezeichner* **(** *FormaleParam* **) ;**
- Beispiel:

```
// Deklaration durch Definition
int max( int a, int b) {
  if(a > b) return a;
  return b;
}

void main() {
  int z = max( 47, 11 );
}
```

```
// Explizite Deklaration
int max( int, int );

void main() {
  int z = max( 47, 11 );
}

int max( int a, int b) {
  if(a > b) return a;
  return b;
}
```


Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein

Achtung: C erzwingt dies nicht!

- Es ist erlaubt **nicht-deklarierte** Funktionen aufzurufen (↪ implizite Deklaration)
- Derartige Aufrufe sind jedoch **nicht typsicher**
  - Compiler kennt die formale Parameterliste nicht
    ⇝ kann nicht prüfen, ob die tatsächlichen Parameter passen
  - Man kann **irgendwas** übergeben
- Moderne Compiler generieren immerhin eine **Warnung**
  ⇝ Warnungen des Compilers immer ernst nehmen!

Funktionsdeklaration (Forts.)

[≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
- **Beispiel:**

```
1  #include <stdio.h>
2
3  int main() {
4    double d = 47.11;
5    foo( d );
6    return 0;
7  }
8
9  void foo( int a, int b) {
10   printf( "foo: a:%d, b:%d\n", a, b);
11 }
```

5 Funktion `foo()` ist nicht **deklariert** ⇝ der Compiler **warnt**, aber akzeptiert beliebige tatsächliche Parameter

9 `foo()` ist **definiert** mit den formalen Parmetern `(int, int)`. Was immer an tatsächlichen Parametern übergeben wurde, wird entsprechend interpretiert!

10 **Was wird hier ausgegeben?**

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine Funktion, die mit leerer formaler Parameterliste deklariert wurde, akzeptiert ebenfalls beliebige Parameter ⇝ **keine Typsicherheit**
  - In diesem Fall warnt der Compiler **nicht!** Die Probleme bleiben!
- **Beispiel:**

```
#include <stdio.h>

void foo(); // "open" declaration

int main() {
  double d = 47.11;
  foo( d );
  return 0;
}

void foo( int a, int b) {
  printf( "foo: a:%d, b:%d\n", a, b);
}
```

Funktion `foo` wurde mit **leerer** formaler Parameterliste deklariert ⇝ dies ist formal ein **gültiger Aufruf!**

Funktionsdeklaration (Forts.) [≠Java]

- Funktionen ~~müssen~~ **sollten** vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine Funktion, die mit leerer formaler Parameterliste deklariert wurde, akzeptiert ebenfalls beliebige Parameter ⇝ **keine Typsicherheit**
  - In diesem Fall warnt der Compiler **nicht!** Die Probleme bleiben!

Achtung: Verwechslungsgefahr

- In Java deklariert `void foo()` eine **parameterlose** Methode
  - In C muss man dafür `void foo(void)` schreiben ↪ 9–3
- In C deklariert `void foo()` eine **offene** Funktion
  - Das macht nur in (sehr seltenen) Ausnahmefällen Sinn!
  - Schlechter Stil ⇝ Punktabzug

Regel: Funktionen werden stets **vollständig deklariert!**

Überblick: Teil B Einführung in C

Variablendefinition [≈Java]

- **Variable** := Behälter für Werte (↦ Speicherplatz)
- Syntax (Variablendefinition):

$$SK_{opt}\ \ Typ_{opt}\ \ Bez_1\ [\ =\ Ausdr_1]_{opt}\ [,\ Bez_2\ [\ =\ Ausdr_2]_{opt}\ ,\ \ldots]_{opt};$$

SK_{opt}	Speicherklasse der Variable, `auto`, `static`, oder leer	[≈Java]
Typ	Typ der Variable, `int` falls kein Typ angegeben wird (↦ schlechter Stil!)	[=Java] [≠Java]
Bez_i	Name der Variable	[=Java]
$Ausdr_i$	Ausdruck für die initiale Wertzuweisung; wird kein Wert zugewiesen so ist der Inhalt von nicht-`static`-Variablen **undefiniert**	[≠Java]

Variablendefinition (Forts.)

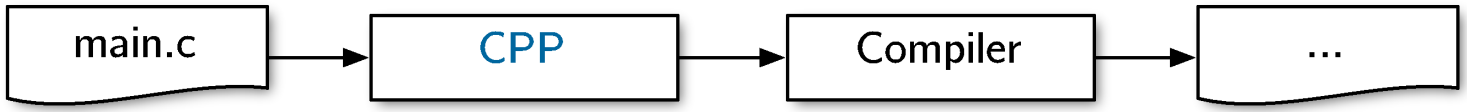
- Variablen können an verschiedenen Positionen definiert werden
  - Global — außerhalb von Funktionen, üblicherweise am Kopf der Datei
  - Lokal — zu Beginn eines `{ Blocks }`, direkt nach der öffnenden Klammer — C89
  - Lokal — überall dort, wo eine Anweisung stehen darf — C99

```
int a = 0;                      // a: global
int b = 47;                     // b: global

void main() {
  int a = b;                    // a: local to function, covers global a
  printf("%d", a);
  int c = 11;                   // c: local to function (C99 only!)
  for(int i=0; i<c; i++) {      // i: local to for-block (C99 only!)
    int a = i;                  // a: local to for-block,
  }                             //    covers function-local a
}
```

Mit globalen Variablen beschäftigen wir uns noch näher im Zusammenhang mit **Modularisierung** ↪ 12–5

Überblick: Teil B Einführung in C

Der C-Präprozessor [≠Java]



- Bevor eine C-Quelldatei übersetzt wird, wird sie zunächst durch einen Makro-Präprozessor bearbeitet
  - Historisch ein eigenständiges Programm (**CPP** = **C P**re**P**rocessor)
  - Heutzutage in die üblichen Compiler integriert
- Der CPP bearbeitet den Quellcode durch Texttransformation
  - Automatische Transformationen („Aufbereiten“ des Quelltextes)
    - Kommentaren werden entfernt
    - Zeilen, die mit \ enden, werden zusammengefügt
    - ...
  - Steuerbare Transformationen (durch den Programmierer)
    - Präprozessor-Direktiven werden evaluiert und ausgeführt
    - Präprozessor-Makros werden expandiert

Präprozessor-Direktiven [≠Java]

- **Präprozessor-Direktive** := Steueranweisung an den Präprozessor

`#include` <*Datei*>	**Inklusion:** Fügt den Inhalt von *Datei* an der aktuellen Stelle in den Token-Strom ein.
`#define` *Makro Ersetzung*	**Makrodefinition:** Definiert ein Präprozessor-Makro *Makro*. In der Folge wird im Token-Strom jedes Auftreten des Wortes *Makro* durch *Ersetzung* substituiert. *Ersetzung* kann auch leer sein.
`#if`(*Bedingung*), `#elif`, `#else`, `#endif`	**Bedingte Übersetzung:** Die folgenden Code-Zeilen werden in Abhängigkeit von *Bedingung* dem Compiler überreicht oder aus dem Token-Strom entfernt.
`#ifdef` *Makro*, `#ifndef` *Makro*	Bedingte Übersetzung in Abhängigkeit davon, ob *Makro* (z. B. mit `#define`) definiert wurde.
`#error` *Text*	**Abbruch:** Der weitere Übersetzungsvorgang wird mit der Fehlermeldung *Text* abgebrochen.

Der Präprozessor definiert letztlich eine eingebettete **Meta-Sprache**. Die Präprozessor-Direktiven (Meta-Programm) verändern das C-Programm (eigentliches Programm) vor dessen Übersetzung.

Präprozessor – Verwendungsbeispiele [≠Java]

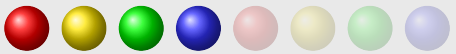
Einfache Makro-Definitionen

Leeres Makro (Flag)	`#define USE_7SEG`
Quelltext-Konstante	`#define NUM_LEDS (4)`
„Inline“-Funktion	`#define SET_BIT(m,b) (m | (1<<b))`

Präprozessor-Anweisungen werden **nicht** mit einem Strichpunkt abgeschlossen!

Verwendung

```
#if( (NUM_LEDS > 8) || (NUM_LEDS < 0) )
#  error invalid NUM_LEDS            // this line is not included
#endif

void enlighten(void) {
  uint8_t mask = 0, i;
  for (i = 0; i < NUM_LEDS; i++) {  // NUM_LEDS --> (4)
    mask = SET_BIT(mask, i);        // SET_BIT(mask, i) --> (mask | (1<<i))
  }
  sb_led_set_all_leds( mask );      // -->

#ifdef USE_7SEG
  sb_show_HexNumber( mask );        // -->
#endif

}
```



Präprozessor – Gefahren [≠Java]

- Funktionsähnliche Makros sind keine Funktionen!
  - Parameter werden nicht evaluiert, sondern textuell eingefügt
    Das kann zu unangenehmen Überraschungen führen

```
#define POW2(a) 1 << a
n = POW2( 2 ) * 3
```
<< hat geringere Präzedenz als *

⇝
```
n = 1 << 2 * 3
```

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

```
#define POW2(a) (1 << a)
n = POW2( 2 ) * 3
```
⇝
```
n = (1 << 2) * 3
```

  - Aber nicht alle

```
#define max(a,b) ((a > b) ? a : b)
n = max( x++, 7 )
```
a++ wird ggf. zweimal ausgewertet

⇝
```
n = ((x++ > 7) ? x++ : 7)
```

- Eine mögliche Alternative sind `inline`-Funktionen C99
  - Funktionscode wird eingebettet ⇝ ebenso effizient wie Makros

```
inline int max(int a, int b) {
  return (a > b) ? a : b;
}
```

Systemnahe Programmierung in C (SPiC)

Teil C Systemnahe Softwareentwicklung

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

http://www4.cs.fau.de/Lehre/SS16/V_SPIC

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μC-Systemarchitektur

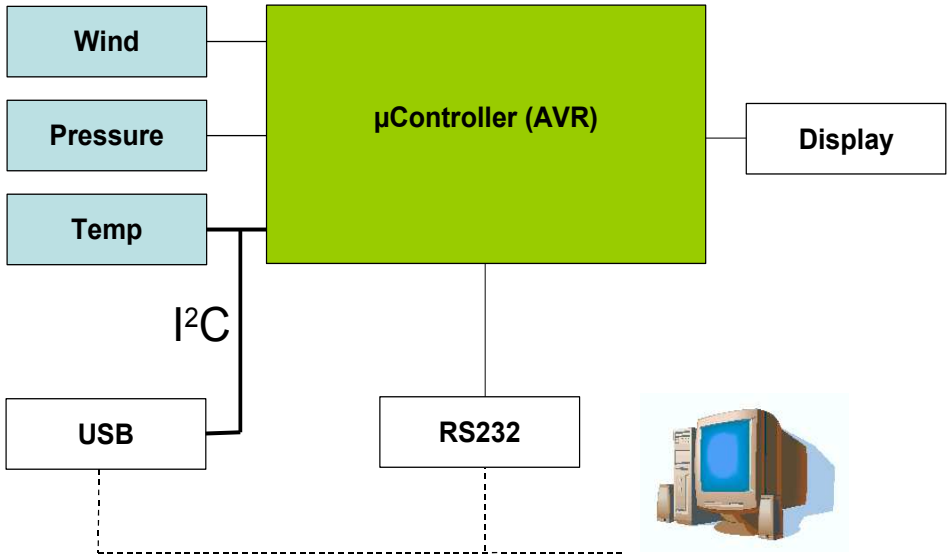
Softwareentwurf

- Softwareentwurf: Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
  - Ziel: Zerlegung des Problems in beherrschbare Einheiten
- Es gibt eine Vielzahl von Softwareentwurfs-Methoden
  - Objektorientierter Entwurf [↪ GDI, 01-01]
    - Stand der Kunst
    - Dekomposition in Klassen und Objekte
    - An Programmiersprachen wie C++ oder Java ausgelegt
  - Top-Down-Entwurf / Funktionale Dekomposition
    - Bis Mitte der 80er Jahre fast ausschließlich verwendet
    - Dekomposition in Funktionen und Funktionsaufrufe
    - An Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert

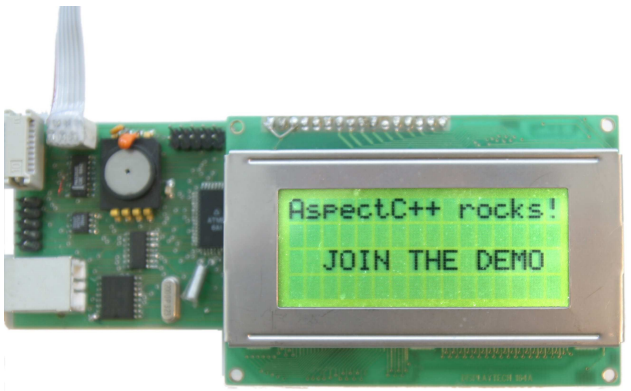
Systemnahe Software wird oft (noch) mit **Funktionaler Dekomposition** entworfen und entwickelt.

Beispiel-Projekt: Eine Wetterstation

- Typisches eingebettetes System
  - Mehrere Sensoren
    - Wind
    - Luftdruck
    - Temperatur
  - Mehrere Aktoren (hier: Ausgabegeräte)
    - LCD-Anzeige
    - PC über RS232
    - PC über USB
  - Sensoren und Aktoren an den μC angebunden über verschiedene Bussysteme
    - I^2C
    - RS232



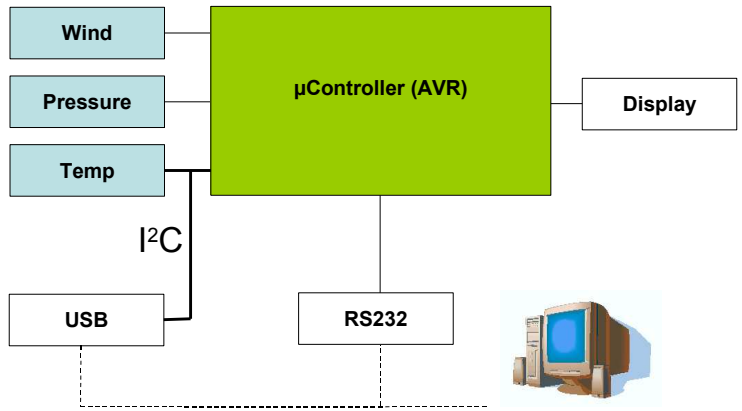
Wie sieht die **funktionale Dekomposition** der Software aus?



Funktionale Dekomposition: Beispiel

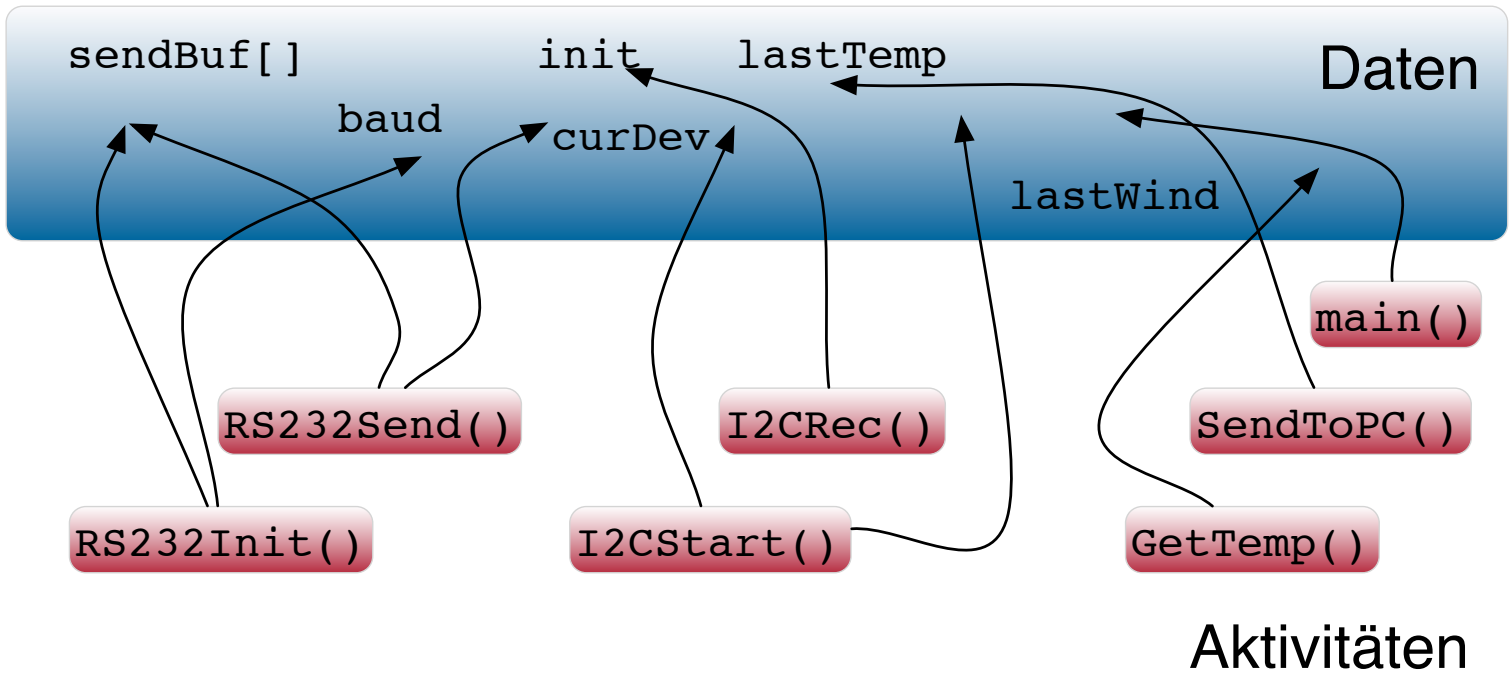
Funktionale Dekomposition der Wetterstation (Auszug):

1. Sensordaten lesen
   - 1.1 Temperatursensor lesen
     - 1.1.1 I²C-Datenübertragung initiieren
     - 1.1.2 Daten vom I²C-Bus lesen
   - 1.2 Drucksensor lesen
   - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
   - 3.1 Daten über RS232 versenden
     - 3.1.1 Baudrate und Parität festlegen (einmalig)
     - 3.1.2 Daten schreiben
   - 3.2 LCD-Display aktualisieren
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Aktivitäten, nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten $\rightsquigarrow$ mangelhafte Trennung der Belange



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Aktivitäten, nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten $\leadsto$ mangelhafte Trennung der Belange

Prinzip der **Trennung der Belange**

Dinge, die **nichts miteinander** zu tun haben, sind auch **getrennt** unterzubringen!

Trennung der Belange (*Separation of Concerns*) ist ein **Fundamentalprinzip** der Informatik
(wie auch jeder anderen Ingenieursdisziplin).

Zugriff auf Daten (Variablen)

- Variablen haben ↪ 10–1
  - Sichtbarkeit (*Scope*) „Wer kann auf die Variable zugreifen?“
  - Lebensdauer „Wie lange steht der Speicher zur Verfügung?“
- Wird festgelegt durch Position (Pos) und Speicherklasse (SK)

Pos	SK ↦	Sichtbarkeit	Lebensdauer
Lokal	*keine*, `auto`	Definition → Blockende	Definition → Blockende
	`static`	Definition → Blockende	Programmstart → Programmende
Global	*keine*	unbeschränkt	Programmstart → Programmende
	`static`	modulweit	Programmstart → Programmende

```
int a = 0;                    // a: global
static int b = 47;            // b: local to module

void f(void) {
  auto int a = b;             // a: local to function (auto optional)
                              //    destroyed at end of block
  static int c = 11;          // c: local to function, not destroyed
}
```


Zugriff auf Daten (Variablen) (Forts.)

- Sichtbarkeit und Lebensdauer sollten restriktiv ausgelegt werden
  - Sichtbarkeit so beschränkt wie möglich!
    - Überraschende Zugriffe „von außen“ ausschließen (Fehlersuche)
    - Implementierungsdetails verbergen (Black-Box-Prinzip, *information hiding*)
  - Lebensdauer so kurz wie möglich
    - Speicherplatz sparen
    - Insbesondere wichtig auf μ-Controller-Plattformen

↪ 1–4

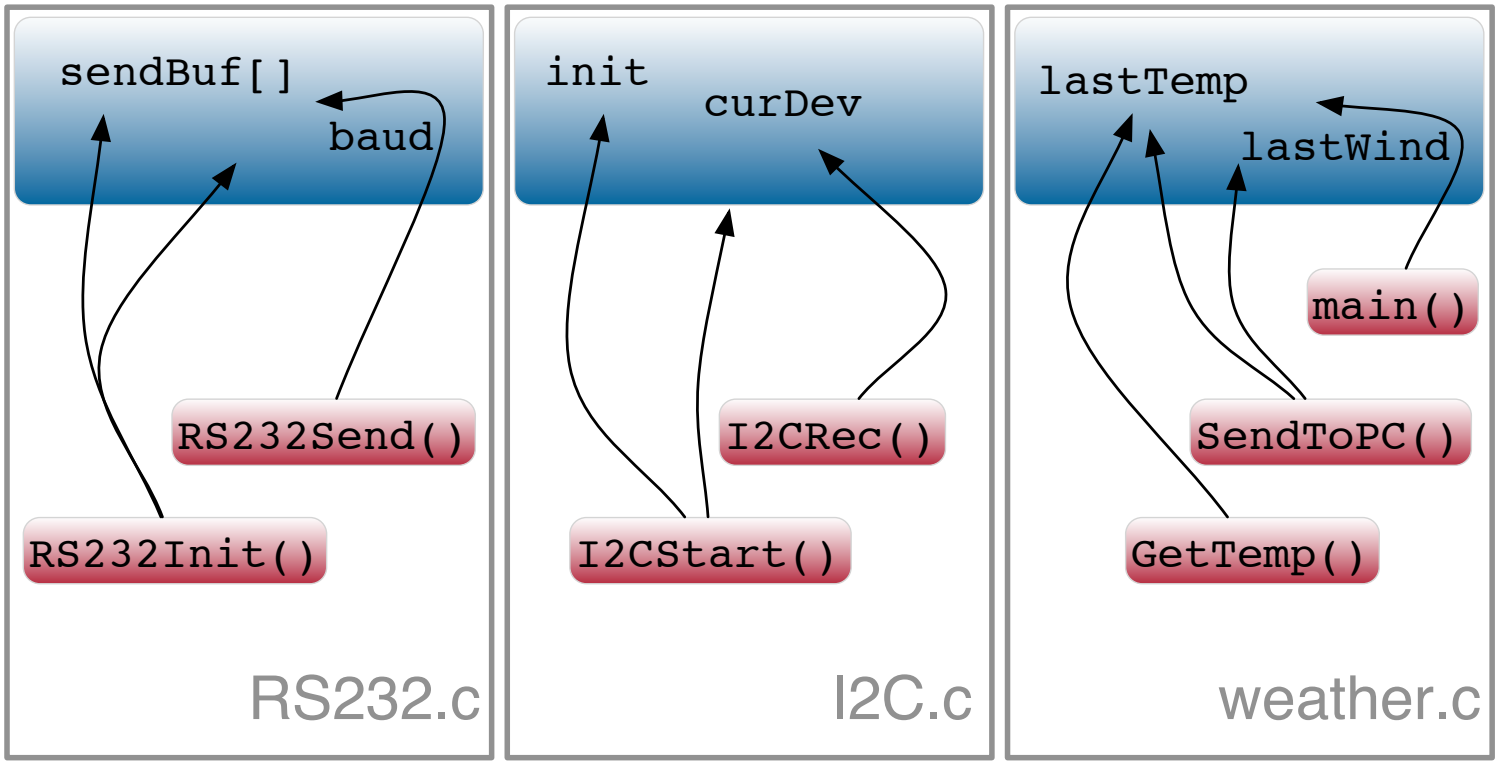
Konsequenz: Globale Variablen vermeiden!

- Globale Variablen sind überall sichtbar
- Globale Variablen belegen Speicher über die gesamte Programmlaufzeit

Regel: Variablen erhalten stets die **geringstmögliche Sichtbarkeit und Lebensdauer**

Lösung: Modularisierung

- Separation jeweils zusammengehöriger Daten und Funktionen in übergeordnete Einheiten $\rightsquigarrow$ **Module**



Was ist ein Modul?

- **Modul** := (<*Menge von Funktionen*>, (↦ „`class`“ in Java)
  <*Menge von Daten*>,
  <*Schnittstelle*>)
- Module sind größere Programmbausteine ↪ 9–1
  - Problemorientierte Zusammenfassung von Funktionen und Daten
    ⇝ Trennung der Belange
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)
    ⇝ Zugriff erfolgt ausschließlich über die Modulschnittstelle

Modul ↦ Abstraktion ↪ 4–1

- Die Schnittstelle eines Moduls abstrahiert
  - Von der tatsächlichen Implementierung der Funktionen
  - Von der internen Darstellung und Verwendung von Daten

Module in C

[≠Java]

- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein idiomatisch (über Konventionen) realisiert ↪ 3–12
  - Modulschnittstelle ↦ `.h`-Datei (enthält Deklarationen ↪ 9–7)
  - Modulimplementierung ↦ `.c`-Datei (enthält Definitionen ↪ 9–3)
  - Modulverwendung ↦ `#include <Modul.h>`

```
void RS232Init( uint16_t br );
void RS232Send( char ch );
...
```

RS232.h: **Schnittstelle / Vertrag (öffentl.)**
Deklaration der bereitgestellten Funktionen (und ggf. Daten)

```
#include <RS232.h>
static uint16_t   baud = 2400;
static char       sendBuf[16];
...
void RS232Init( uint16_t br) {
  ...
  baud = br;
}
void RS232Send( char ch ) {
  sendBuf[···] = ch;
  ...
}
```

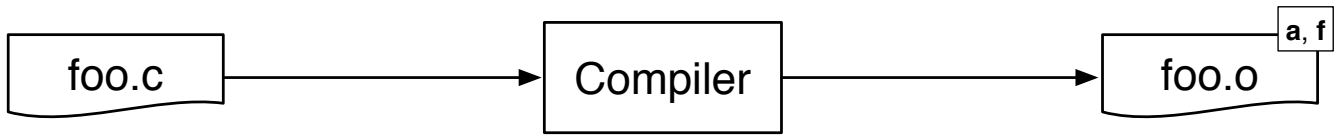
RS232.c: **Implementierung (nicht öffentl.)**
Definition der bereitgestellten Funktionen (und ggf. Daten)

Ggf. modulinterne Hilfsfunktionen und Daten (`static`)

Inklusion der eigenen Schnittstelle stellt sicher, dass der Vertrag eingehalten wird

Module in C – Export [≠Java]

- Ein C-Modul exportiert eine Menge von definierten Symbolen
  - Alle Funktionen und globalen Variablen (↦ „`public`“ in Java)
  - Export kann mit `static` unterbunden werden (↦ „`private`“ in Java)
    (↦ Einschränkung der Sichtbarkeit ↪ 12–5)
- Export erfolgt beim Übersetzungsvorgang (`.c`-Datei ⟶ `.o`-Datei)



Quelldatei (`foo.c`)

```
uint16_t a;
// public
static uint16_t b;
// private

void f(void)          // public
{ ... }
static void g(int)    // private
{ ... }
```

Objektdatei (`foo.o`)

Symbole **a** und **f** werden exportiert.

Symbole b und g sind `static` definiert und werden deshalb nicht exportiert.

Module in C – Import [≠Java]

- Ein C-Modul importiert eine Menge nicht-definierter Symbole
  - Funktionen und globale Variablen, die verwendet werden, im Modul selber jedoch nicht definiert sind
  - Werden beim Übersetzen als unaufgelöst markiert

Quelldatei (`bar.c`)

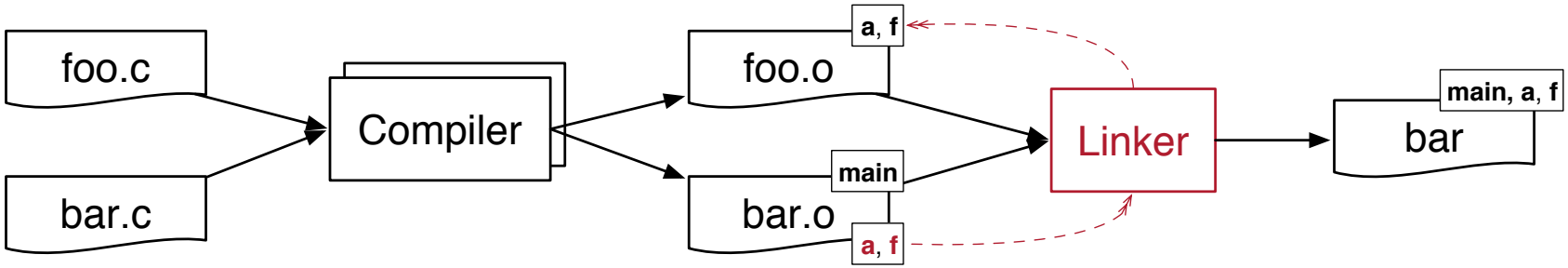
```
extern uint16_t a;
// declare
void f(void);       // declare

void main() {       // public
  a = 0x4711;       // use
  f();              // use
}
```

Objektdatei (`bar.o`)

Symbol **`main`** wird exportiert.
Symbole **`a`** und **`f`** sind unaufgelöst.

Module in C – Import (Forts.) [≠Java]

- Die eigentliche Auflösung erfolgt durch den Linker



Linken ist nicht typsicher!

- Typinformationen sind in Objektdateien nicht mehr vorhanden
- Auflösung durch den Linker erfolgt **ausschließlich** über die **Symbolnamen** (Bezeichner)

⇝ Typsicherheit muss beim **Übersetzen** sichergestellt werden

⇝ Einheitliche Deklarationen durch gemeinsame Header-Datei

Module in C – Header [≠Java]

- Elemente aus fremden Modulen müssen deklariert werden
  - Funktionen durch normale Deklaration ↪ 9–7

```
void f(void);
```

  - Globale Variablen durch `extern`

```
extern uint16_t a;
```

Das `extern` unterscheidet eine Variablendeklaration von einer Variablendefinition.

- Die Deklarationen erfolgen sinnvollerweise in einer Header-Datei, die von der Modulentwicklerin bereitgestellt wird
  - Schnittstelle des Moduls (↦ „`interface`“ in Java)
    - Exportierte Funktionen des Moduls
    - Exportierte globale Variablen des Moduls
    - Modulspezifische Konstanten, Typen, Makros
    - Verwendung durch Inklusion (↦ „`import`“ in Java)
  - Wird **auch vom Modul inkludiert**, um Übereinstimmung von Deklarationen und Definitionen sicher zu stellen (↦ „`implements`“ in Java)

Module in C – Header (Forts.)

[≠Java]

Modulschnittstelle: `foo.h`

```
// foo.h
#ifndef _FOO_H
#define _FOO_H

// declarations
extern uint16_t a;
void f(void);

#endif // _FOO_H
```

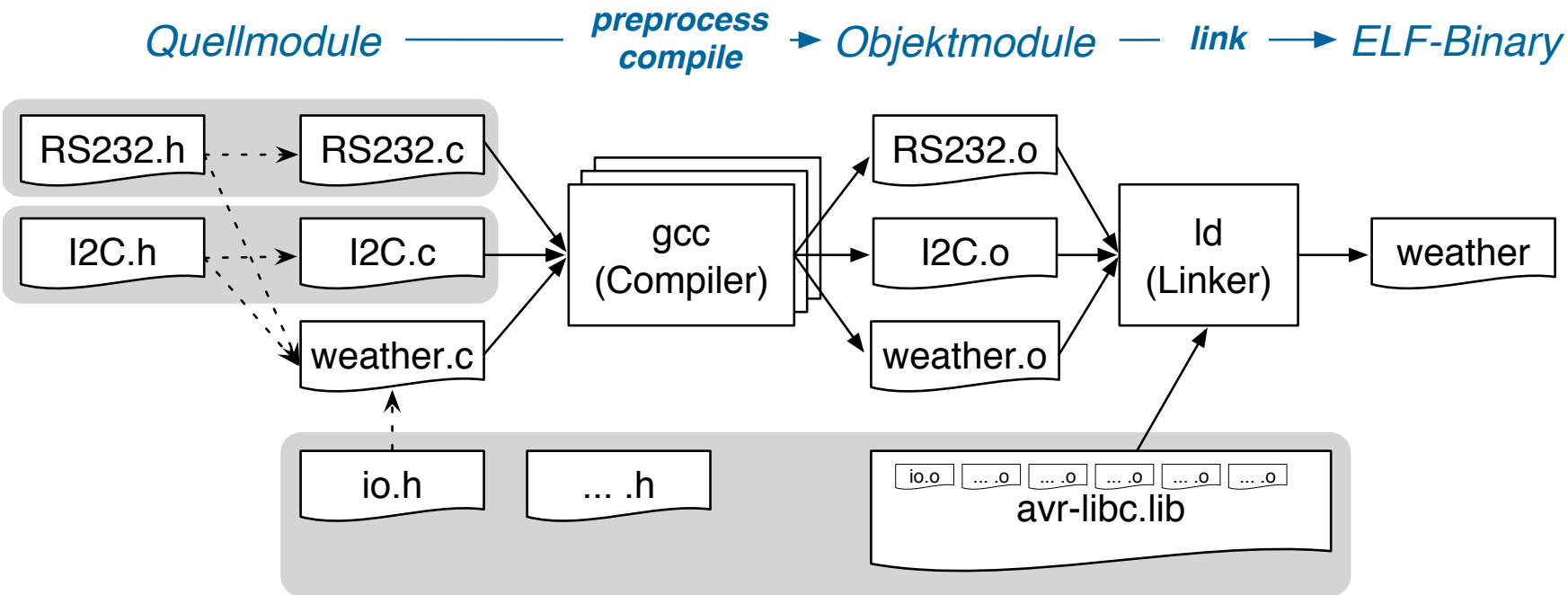
Modulimplementierung `foo.c`

```
// foo.c
#include <foo.h>


// definitions
uint16_t a;
void f(void){
  ...
}
```

Modulverwendung `bar.c`
(vergleiche ↪ 12–11)

```
// bar.c
extern uint16_t a;
void f(void);
#include <foo.h>

void main() {
  a = 0x4711;
  f();
}
```


Zurück zum Beispiel: Wetterstation



- Jedes Modul besteht aus Header- und Implementierungsdatei(en)
  - `.h`-Datei definiert die Schnittstelle
  - `.c`-Datei implementiert die Schnittstelle, inkludiert `.h`-Datei, um sicherzustellen, dass Deklaration und Definition übereinstimmen
- Modulverwendung durch Inkludieren der modulspezifischen `.h`-Datei
- Das Ganze funktioniert entsprechend bei Bibliotheken

Zusammenfassung

- Prinzip der Trennung der Belange $\leadsto$ Modularisierung
  - Wiederverwendung und Austausch wohldefinierter Komponenten
  - Verbergen von Implementierungsdetails

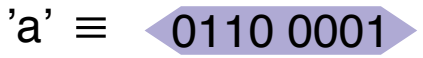
- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern idiomatisch durch Konventionen realisiert
  - Modulschnittstelle $\mapsto$ `.h`-Datei (enthält Deklarationen)
  - Modulimplementierung $\mapsto$ `.c`-Datei (enthält Definitionen)
  - Modulverwendung $\mapsto$ `#include <Modul.h>`
  - `private` Symbole $\mapsto$ als `static` definieren

- Die eigentliche Zusammenfügung erfolgt durch den Linker
  - Auflösung erfolgt ausschließlich über Symbolnamen
    $\leadsto$ **Linken ist nicht typsicher!**
  - Typsicherheit muss beim Übersetzen sichergestellt werden
    $\leadsto$ durch gemeinsame Header-Datei

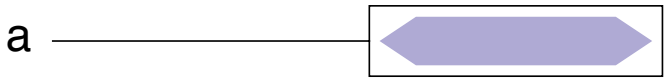
Überblick: Teil C Systemnahe Softwareentwicklung

Einordnung: Zeiger (*Pointer*)

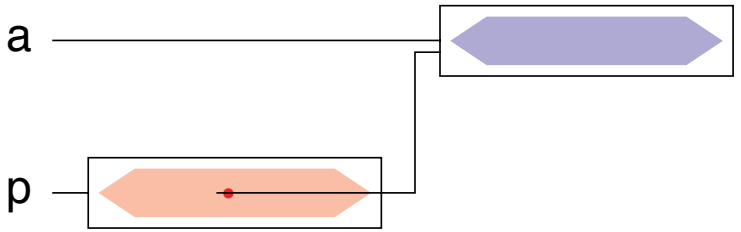
- **Literal:** `'a'`
  Darstellung eines Wertes



- **Variable:** `char a;`
  Behälter für einen Wert



- **Zeiger-Variable:** `char *p = &a;`
  Behälter für eine Referenz
  auf eine Variable



Zeiger (*Pointer*)

- Eine Zeigervariable (*Pointer*) enthält als Wert die Adresse einer anderen Variablen
  - Ein Zeiger verweist auf eine Variable (im Speicher)
  - Über die Adresse kann man indirekt auf die Zielvariable (ihren Speicher) zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
  - Funktionen können Variablen des Aufrufers verändern (*call-by-reference*) ↪ 9–5
  - Speicher lässt sich direkt ansprechen
  - Effizientere Programme

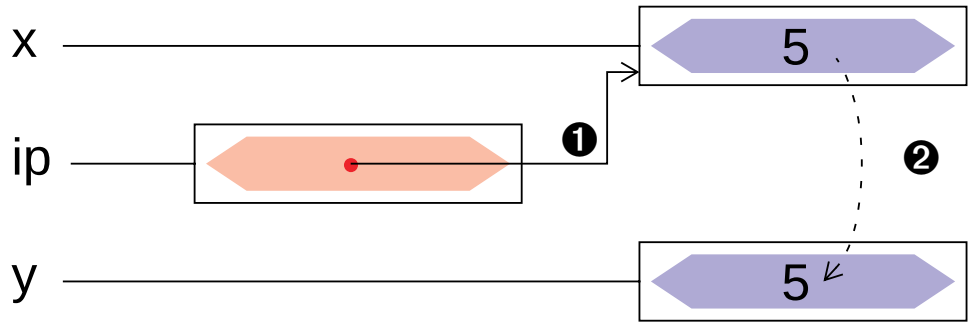
„Effizienz durch Maschinennähe“ ↪ 3–13

- Aber auch viele Probleme!
  - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
  - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

Definition von Zeigervariablen

- **Zeigervariable** := Behälter für Verweise ($\mapsto$ Adresse)
- Syntax (Definition): *Typ* * *Bezeichner* ;
- Beispiel

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



Adress- und Verweisoperatoren

- Adressoperator: **&x**
  Der unäre **&**-Operator liefert die Referenz ($\mapsto$ Adresse im Speicher) der Variablen **x**.
- Verweisoperator: ***y**
  Der unäre *-Operator liefert die Zielvariable ($\mapsto$ Speicherzelle / Behälter), auf die der Zeiger **y** verweist (Dereferenzierung).
- Es gilt: **(*(&x)) ≡ x**
  Der Verweisoperator ist die Umkehroperation des Adressoperators.

Achtung: Verwirrungsgefahr (*** *Ich seh überall Sterne* ***)

Das *-Symbol hat in C verschiedene Bedeutungen, **je nach Kontext**

1.	Multiplikation (binär):	`x * y`	in Ausdrücken
2.	Typmodifizierer:	`uint8_t *p1, *p2` `typedef char* CPTR`	in Definitionen und Deklarationen
3.	Verweis (unär):	`x = *p1`	in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung
$\rightsquigarrow$ * wird fälschlicherweise für ein Bestandteil des Bezeichners gehalten.

Zeiger als Funktionsargumente

- Parameter werden in C immer *by-value* übergeben ↪ 9–5
  - Parameterwerte werden in lokale Variablen der aufgerufenen Funktion kopiert
  - Aufgerufene Funktion kann tatsächliche Parameter des Aufrufers nicht ändern

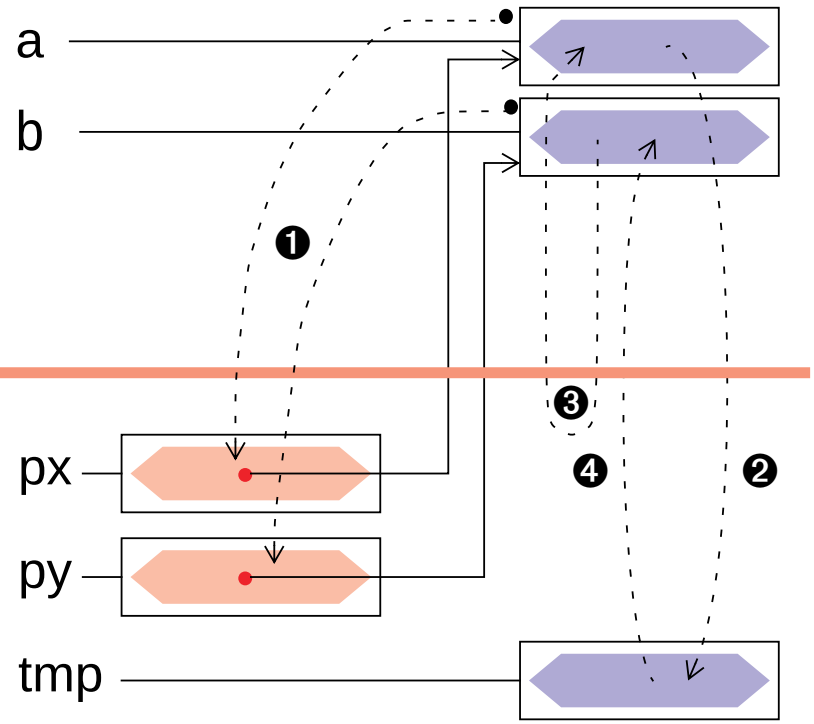
- Das gilt auch für Zeiger (Verweise) [↪ GDI, 14-01-01]
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des *-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

  ⇝ **Call-by-reference**

Zeiger als Funktionsargumente (Forts.)

Beispiel (Gesamtüberblick)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ❶
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```

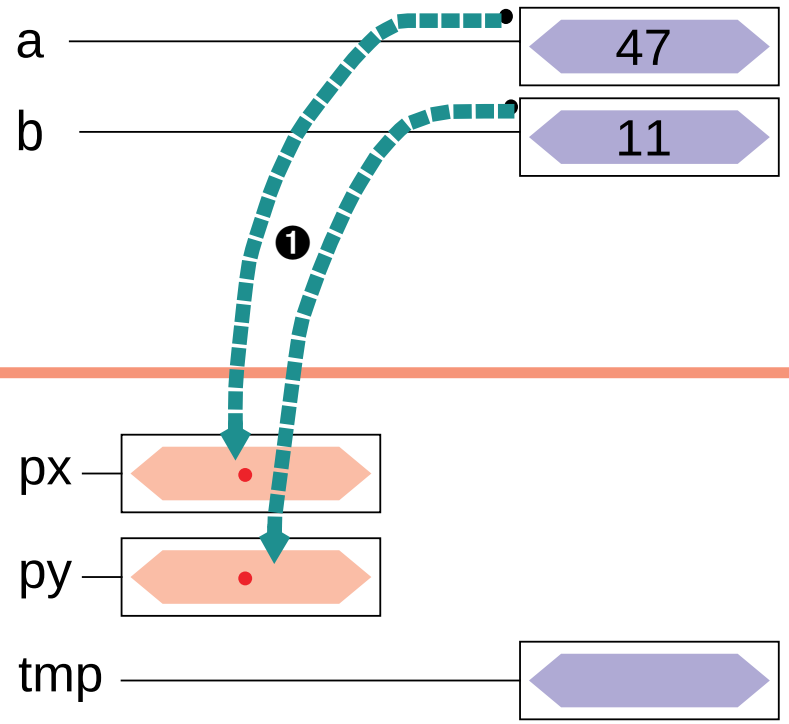


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b); ❶
```

```
void swap (int *px, int *py)
{
    int tmp;
```

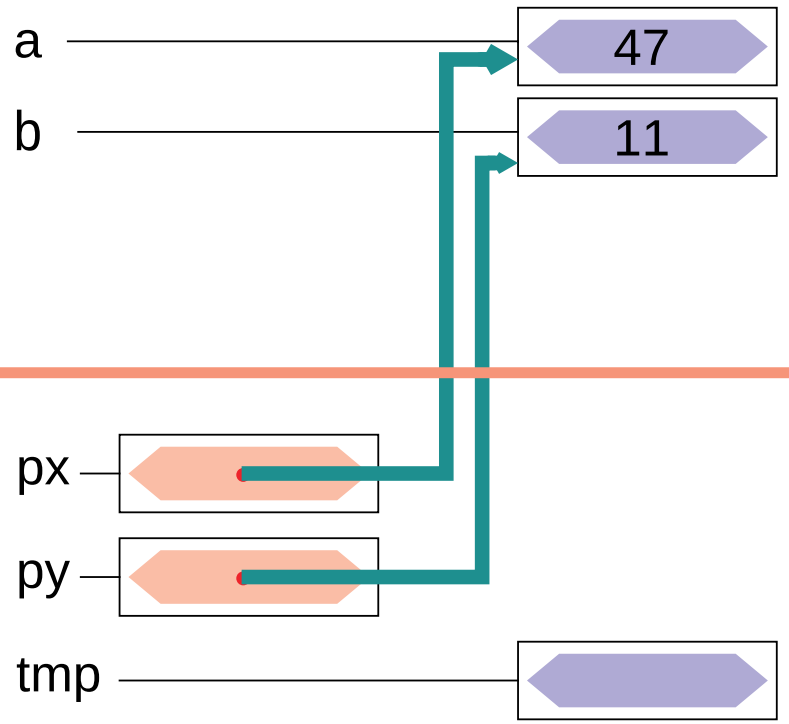


Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
```

```
void swap (int *px, int *py)
{
    int tmp;
```

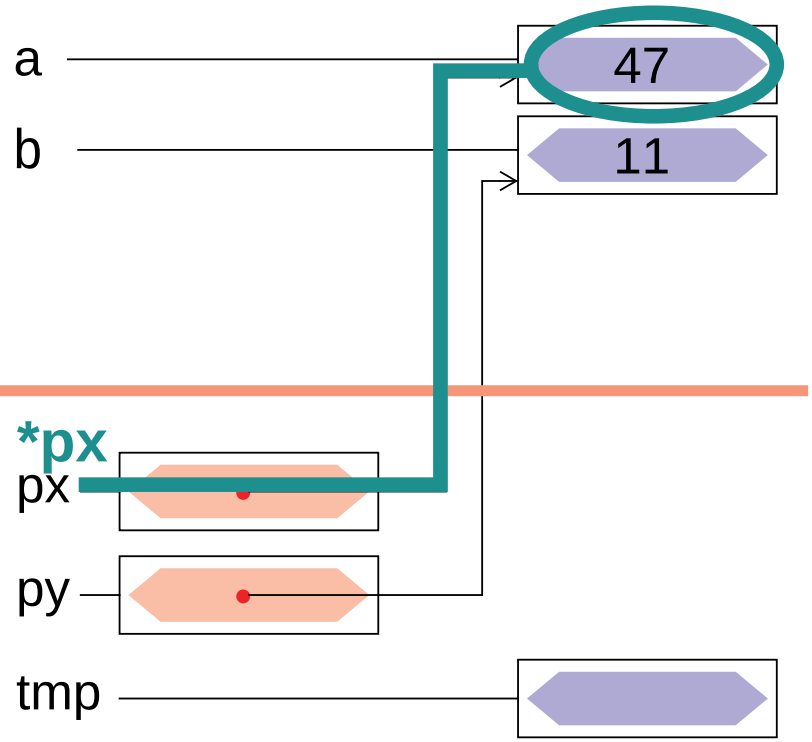


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
```

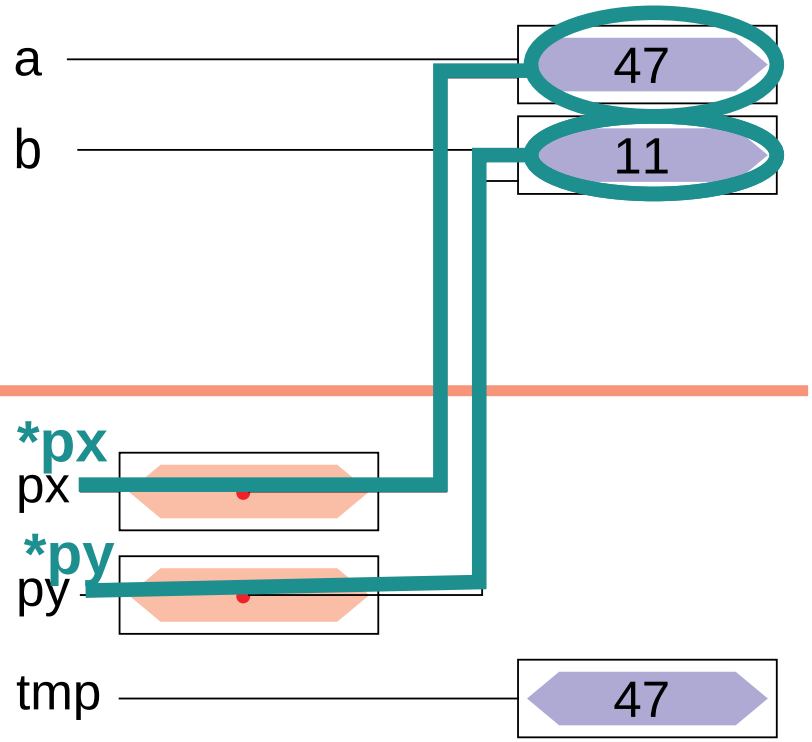


Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
```

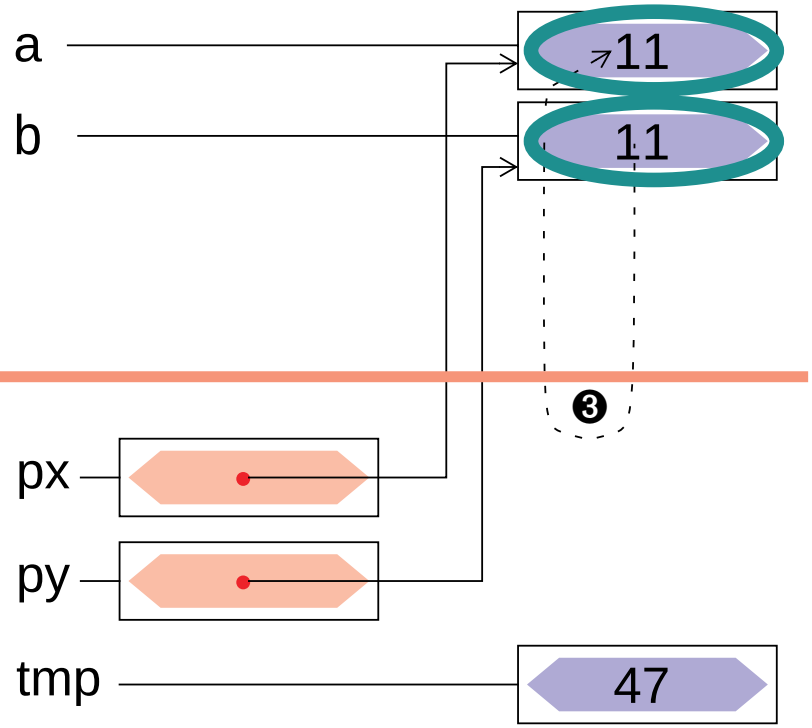


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
```

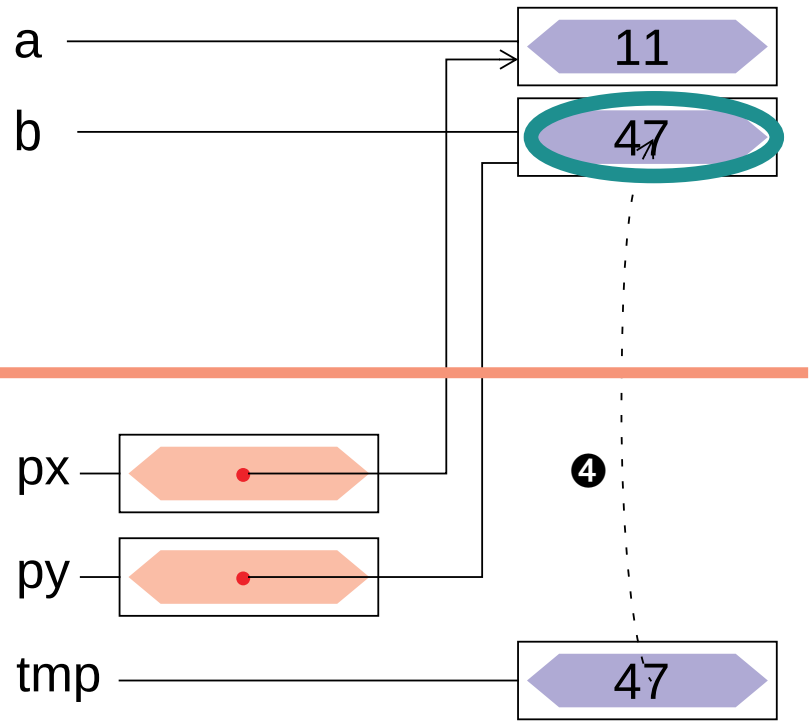


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

```
void swap (int *, int *);
int main() {
    int a=47, b=11;
    ...
    swap(&a, &b);
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



Einordnung: Felder (*Arrays*) [≈Java]

- **Feldvariable** := Behälter für eine Reihe von Werten desselben Typs
- Syntax (Definition): *Typ Bezeichner* **[** *IntAusdruck* **] ;**
  - *Typ* — Typ der Werte [=Java]
  - *Bezeichner* — Name der Feldvariablen [=Java]
  - *IntAusdruck* — **Konstanter** Ganzzahl-Ausdruck, definiert die Feldgröße (↦ Anzahl der Elemente). [≠Java]

    Ab **C99** darf *IntAusdruck* bei `auto`-Feldern auch **variabel** (d. h. beliebig, aber fest) sein.
- Beispiele:

```
static uint8_t LEDs[ 8*2 ];     // constant, fixed array size

void f( int n ) {
  auto char a[ NUM_LEDS * 2];   // constant, fixed array size
  auto char b[ n ];             // C99: variable, fixed array size
}
```


Feldinitialisierung

- Wie andere Variablen auch, kann ein Feld bei Definition eine initiale Wertzuweisung erhalten

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[5]     = { 1, 2, 3, 5, 7 };
```

- Werden zu wenig Initialisierungselemente angegeben, so werden die restlichen Elemente mit 0 initialisiert

```
uint8_t LEDs[4] = { RED0 };     // => { RED0, 0, 0, 0 }
int prim[5]     = { 1, 2, 3 };  // => { 1, 2, 3, 0, 0 }
```

- Wird die explizite Dimensionierung ausgelassen, so bestimmt die Anzahl der Initialisierungselemente die Feldgröße

```
uint8_t LEDs[]  = { RED0, YELLOW0, GREEN0, BLUE0 };
int prim[]      = { 1, 2, 3, 5, 7 };
```


Feldzugriff

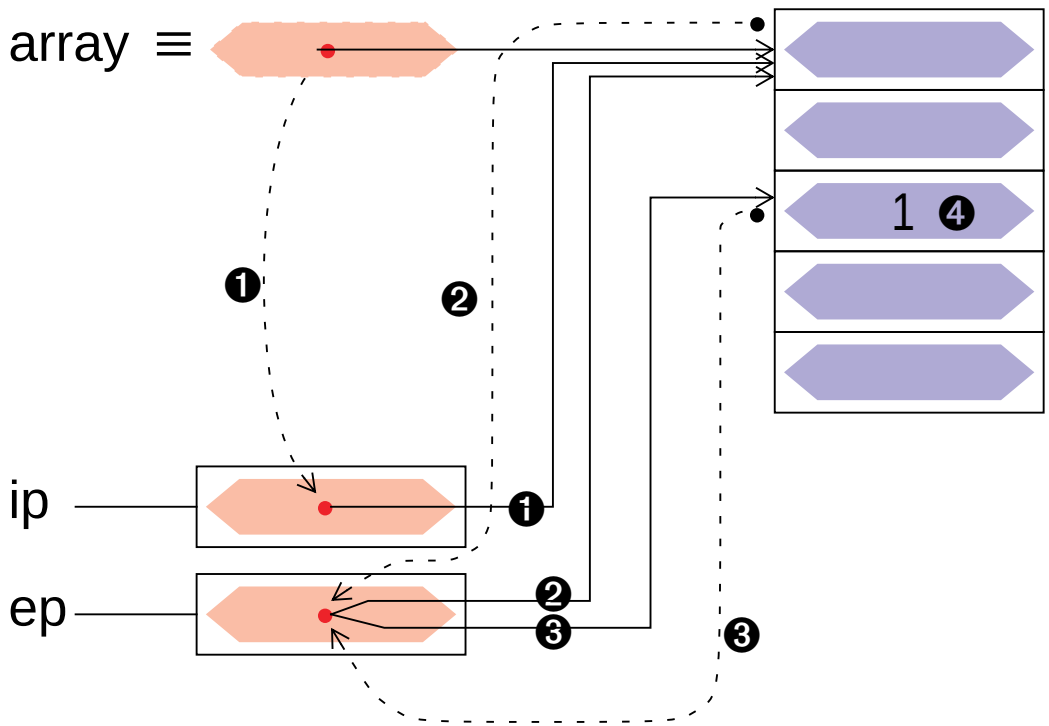
- Syntax: *Feld* **[** *IntAusdruck* **]** [=Java]
  - Wobei $0 \leq$ *IntAusdruck* $< n$ für $n =$ Feldgröße
  - **Achtung:** Feldindex wird nicht überprüft [≠Java]
    $\leadsto$ häufige Fehlerquelle in C-Programmen
- Beispiel

```
uint8_t LEDs[]  = { RED0, YELLOW0, GREEN0, BLUE0 };

LEDs[ 3 ] = BLUE1;

for( unit8_t i = 0; i < 4; ++i ) {
  sb_led_on( LEDs[ i ] );
}

LEDs[ 4 ] = GREEN1;   // UNDEFINED!!!
```

Felder sind Zeiger

- Ein Feldbezeichner ist syntaktisch äquivalent zu einem konstanten Zeiger auf das erste Element des Feldes: `array` ≡ `&array[0]`
  - Ein Alias – kein Behälter ⇝ Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

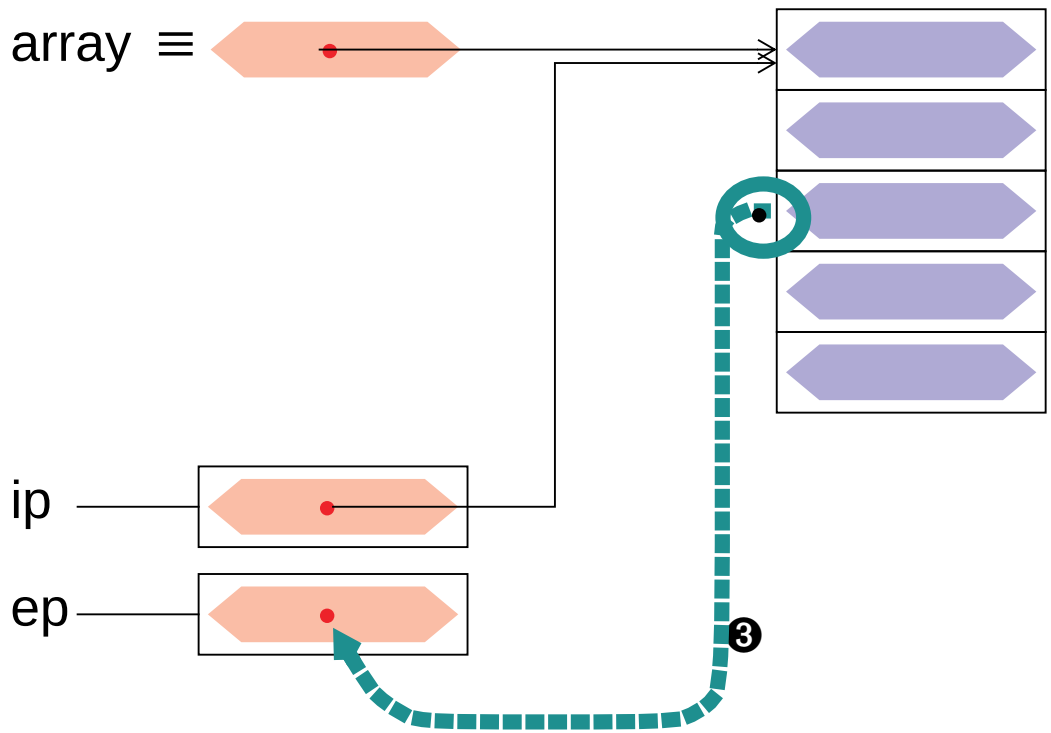
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Felder sind Zeiger

- Ein Feldbezeichner ist syntaktisch äquivalent zu einem konstanten Zeiger auf das erste Element des Feldes: `array` ≡ `&array[0]`
  - Ein Alias – kein Behälter ⇝ Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

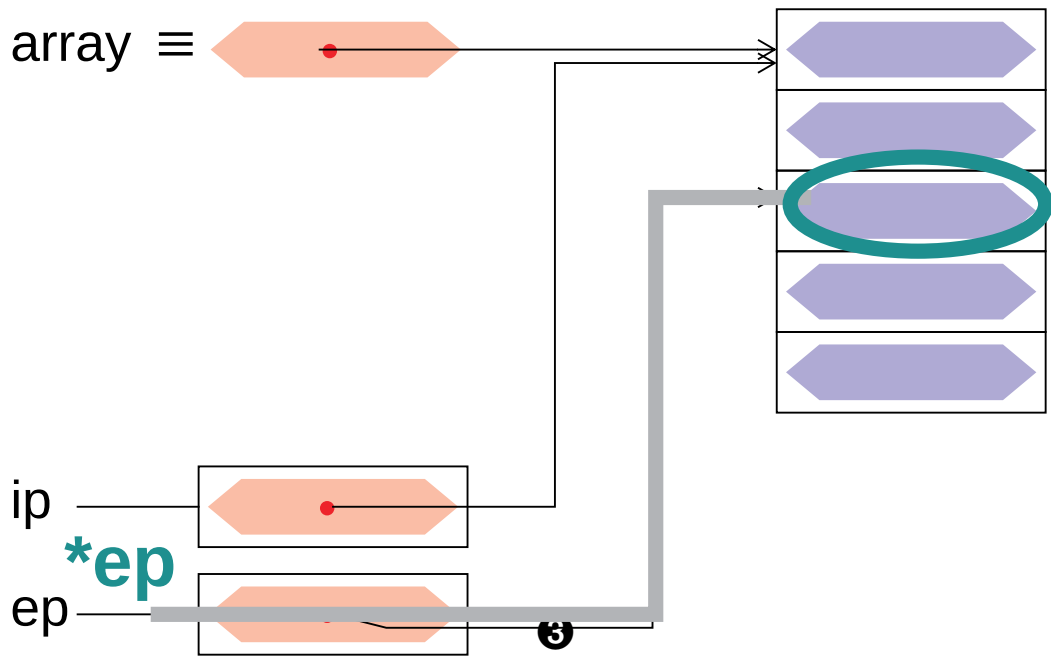
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```



Felder sind Zeiger

- Ein Feldbezeichner ist syntaktisch äquivalent zu einem konstanten Zeiger auf das erste Element des Feldes: `array` ≡ `&array[0]`
  - Ein Alias – kein Behälter ⇝ Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Zeiger sind Felder

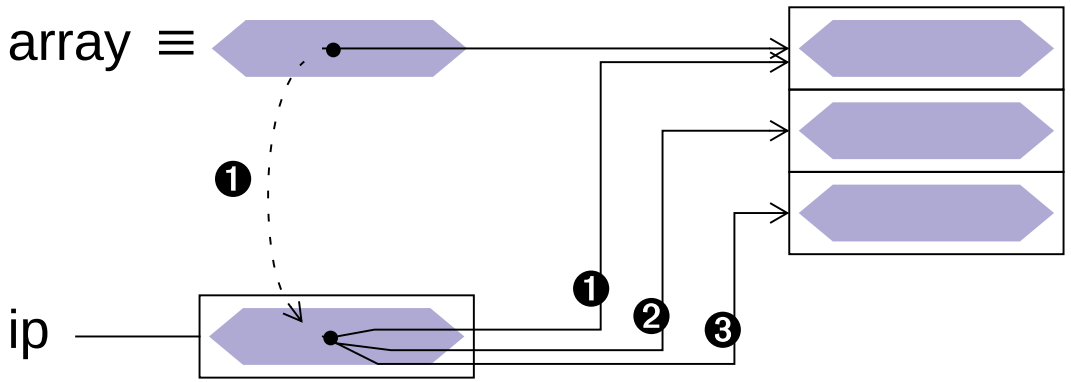
- Ein Feldbezeichner ist syntaktisch äquivalent zu einem konstanten Zeiger auf das erste Element des Feldes: `array` ≡ `&array[0]`
- Diese Beziehung gilt in beide Richtungen: `*array` ≡ `array[0]`
  - Ein Zeiger kann wie ein Feld verwendet werden
  - Insbesondere kann der **[ ]** - Operator angewandt werden ↪ 13–9
- Beispiel (vgl. ↪ 13–9)

```
uint8_t LEDs[]  = { RED0, YELLOW0, GREEN0, BLUE0 };

LEDs[ 3 ]    = BLUE1;
uint8_t *p   = LEDs;
for( unit8_t i = 0; i < 4; ++i ) {
  sb_led_on( p[ i ] );
}
```

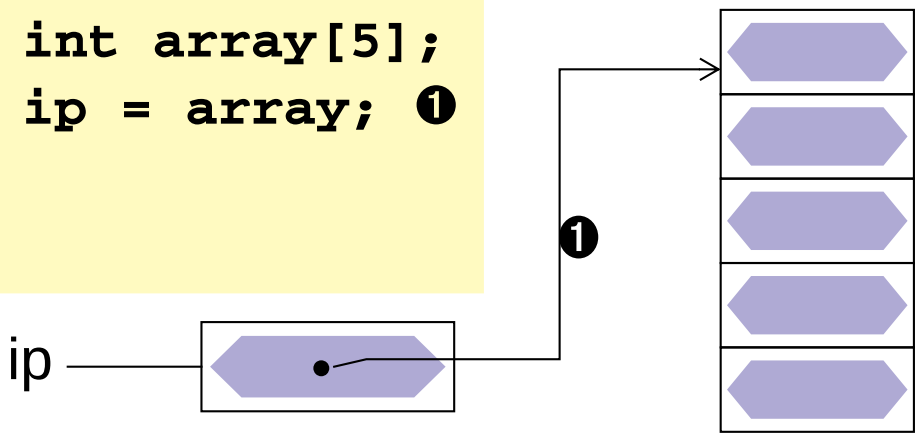
Rechnen mit Zeigern

- Im Unterschied zu einem Feldbezeichner ist eine Zeiger*variable* ein Behälter $\rightsquigarrow$ Ihr Wert ist veränderbar
- Neben einfachen Zuweisungen ist dabei auch Arithmetik möglich

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```




```
int array[5];
ip = array; ❶
```



`(ip+3)` ≡ `&ip[3]`

Bei der Zeigerarithmetik wird immer die Größe des Objekttyps mit berücksichtigt.

Zeigerarithmetik – Operationen

- Arithmetische Operationen
  - ++ Prä-/Postinkrement
    - ⇝ Verschieben auf das nächste Objekt
  - −− Prä-/Postdekrement
    - ⇝ Verschieben auf das vorangegangene Objekt
  - +, − Addition / Subtraktion eines `int`-Wertes
    - ⇝ Ergebniszeiger ist verschoben um n Objekte
  - − Subtraktion zweier Zeiger
    - ⇝ Anzahl der Objekte n zwischen beiden Zeigern (Distanz)
- Vergleichsoperationen: $<$, $<=$, $==$, $>=$, $>$, $!=$ ↪ 7–3
  - ⇝ Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen

Felder sind Zeiger sind Felder – Zusammenfassung

- In Kombination mit Zeigerarithmetik lässt sich in C jede Feldoperation auf eine äquivalente Zeigeroperation abbilden.

- Für `int i, array[N], *ip = array;` mit $0 \leq i < N$ gilt:

```
      array  ≡  &array[0]  ≡  ip          ≡  &ip[0]
     *array  ≡   array[0]  ≡  *ip         ≡  ip[0]
*(array + i)  ≡   array[i]  ≡  *(ip + i)  ≡  ip[i]
               array++  ≢  ip++
```

Fehler: `array` ist konstant!

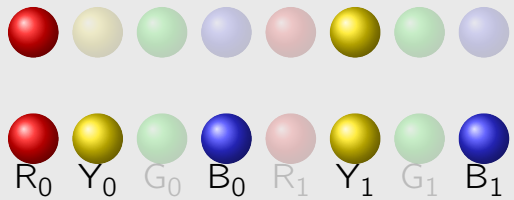
- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.
Der Feldbezeichner kann aber nicht verändert werden.

Felder als Funktionsparameter

- Felder werden in C **immer** als Zeiger übergeben [=Java]

  ⇝ *Call-by-reference*

```
static uint8_t LEDs[] = {RED0, YELLOW1};

void enlight( uint8_t *array,  unsigned n ) {
  for( unsigned i = 0; i < n; ++i )
    sb_led_on( array[i] );
}

void main() {
  enlight( LEDs, 2 );
  uint8_t moreLEDs[] = {YELLOW0, BLUE0, BLUE1};
  enlight( moreLEDs, 3);
}
```



- Informationen über die Feldgröße gehen dabei verloren!
  - Die Feldgröße muss explizit als Parameter mit übergeben werden
  - In manchen Fällen kann sie auch in der Funktion berechnet werden (z. B. bei Strings durch Suche nach dem abschließenden `NUL`-Zeichen)

Felder als Funktionsparameter (Forts.)

- Felder werden in C **immer** als Zeiger übergeben [=Java]
  ⇝ *Call-by-reference*
- Wird der Parameter als `const` deklariert, so kann die Funktion die Feldelemente nicht verändern ↦ Guter Stil! [≠Java]

```
void enlight( const uint8_t *array,  unsigned n ) {
  ...
}
```

- Um anzuzeigen, dass ein Feld (und kein „Zeiger auf Variable“) erwartet wird, ist auch folgende äquivalente Syntax möglich:

```
void enlight( const uint8_t array[],  unsigned n ) {
  ...
}
```

  - Achtung: Das gilt so nur bei Deklaration eines Funktionparameters
  - Bei Variablendefinitionen hat `array[]` eine völlig andere Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, ↪ 13–8)

Felder als Funktionsparameter (Forts.)

- Die Funktion `int strlen(const char *)` aus der Standardbibliothek liefert die Anzahl der Zeichen im übergebenen String

```
void main() {
  ...
  const char *string = "hallo";   // string is array of char
  sb_7seg_showNumber( strlen(string) );
  ...
}
```

Dabei gilt: "hallo" ≡ 

↪ 6–13

- Implementierungsvarianten

Variante 1: Feld-Syntax

```
int strlen( const char s[] ) {
  int n=0;
  while( s[n] != 0 )
    n++;
  return n;
}
```

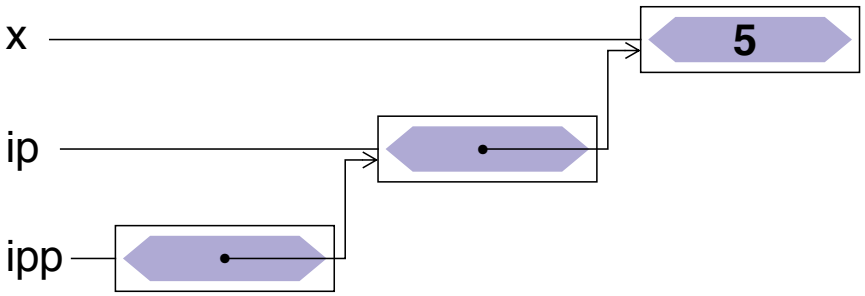
Variante 2: Zeiger-Syntax

```
int strlen( const char *s ) {
  const char *end = s;
  while( *end )
    end++;
  return end - s;
}
```

Zeiger auf Zeiger

- Ein Zeiger kann auch auf eine Zeigervariable verweisen

```
int x = 5;
int *ip = &x;

int **ipp = &ip;
/* → **ipp = 5 */
```



- Wird vor allem bei der Parameterübergabe an Funktionen benötigt
  - Zeigerparameter *call-by-reference* übergeben
    (z. B. `swap()`-Funktion für Zeiger)
  - Ein Feld von Zeigern übergeben

Zeiger auf Funktionen

- Ein Zeiger kann auch auf eine Funktion verweisen
  - Damit lassen sich Funktionen an Funktionen übergeben
    $\mapsto$ Funktionen höherer Ordnung
- Beispiel

```
// invokes job() every second
void doPeriodically( void (*job)(void) ) {
  while( 1 ) {
    job();       // invoke job
    for( volatile uint16_t i = 0; i < 0xffff; ++i )
      ;          // wait a second
  }
}

void blink( void ) {
  sb_led_toggle( RED0 );
}

void main() {
  doPeriodically( blink );  // pass blink() as parameter
}
```


Zeiger auf Funktionen (Forts.)

- Syntax (Definition): *Typ* **(** * *Bezeichner* **)(** $FormaleParam_{opt}$ **);**
  (sehr ähnlich zur Syntax von Funktionsdeklarationen) ↪ 9–3
  - *Typ* — Rückgabetyp der Funktionen, auf die dieser Zeiger verweisen kann
  - *Bezeichner* — Name des Funktionszeigers
  - $FormaleParam_{opt}$ — Formale Parameter der Funktionen, auf die dieser Zeiger verweisen kann: $Typ_1, \ldots, Typ_n$
- Ein Funktionszeiger wird genau wie eine Funktion verwendet
  - Aufruf mit *Bezeichner* **(** *TatParam* **)** ↪ 9–4
  - Adress- (&) und Verweisoperator (*) werden nicht benötigt ↪ 13–4
  - Ein Funktionsbezeichner ist ein konstanter Funktionszeiger

```
void blink( uint8_t which ) { sb_led_toggle( which ); }

void main() {
  void (*myfun)(uint8_t); // myfun is pointer to function
  myfun = blink;          // blink is constant pointer to function
  myfun( RED0 );          // invoke blink() via function pointer
  blink( RED0 );          // invoke blink()
}
```

Zeiger auf Funktionen (Forts.)

- Funktionszeiger werden oft für Rückruffunktionen (*Callbacks*) zur Zustellung asynchroner Ereignisse verwendet ($\mapsto$ „Listener“ in Java)

```
// Example: asynchronous button events with libspicboard
#include <avr/interrupt.h>          // for sei()
#include <7seg.h>                   // for sb_7seg_showNumber()
#include <button.h>                 // for button stuff

// callback handler for button events (invoked on interrupt level)
void onButton( BUTTON b, BUTTONEVENT e ) {
  static int8_t count = 1;
  sb_7seg_showNumber( count++ ); // show no of button presses
  if( count > 99 ) count = 1;    // reset at 100
}

void main() {
  sb_button_registerListener(    // register callback
    BUTTON0, BTNPRESSED,         //   for this button and events
    onButton                     //   invoke this function
  );
  sei();                         // enable interrupts (necessary!)
  while( 1 ) ;                   // wait forever
}
```

Zusammenfassung

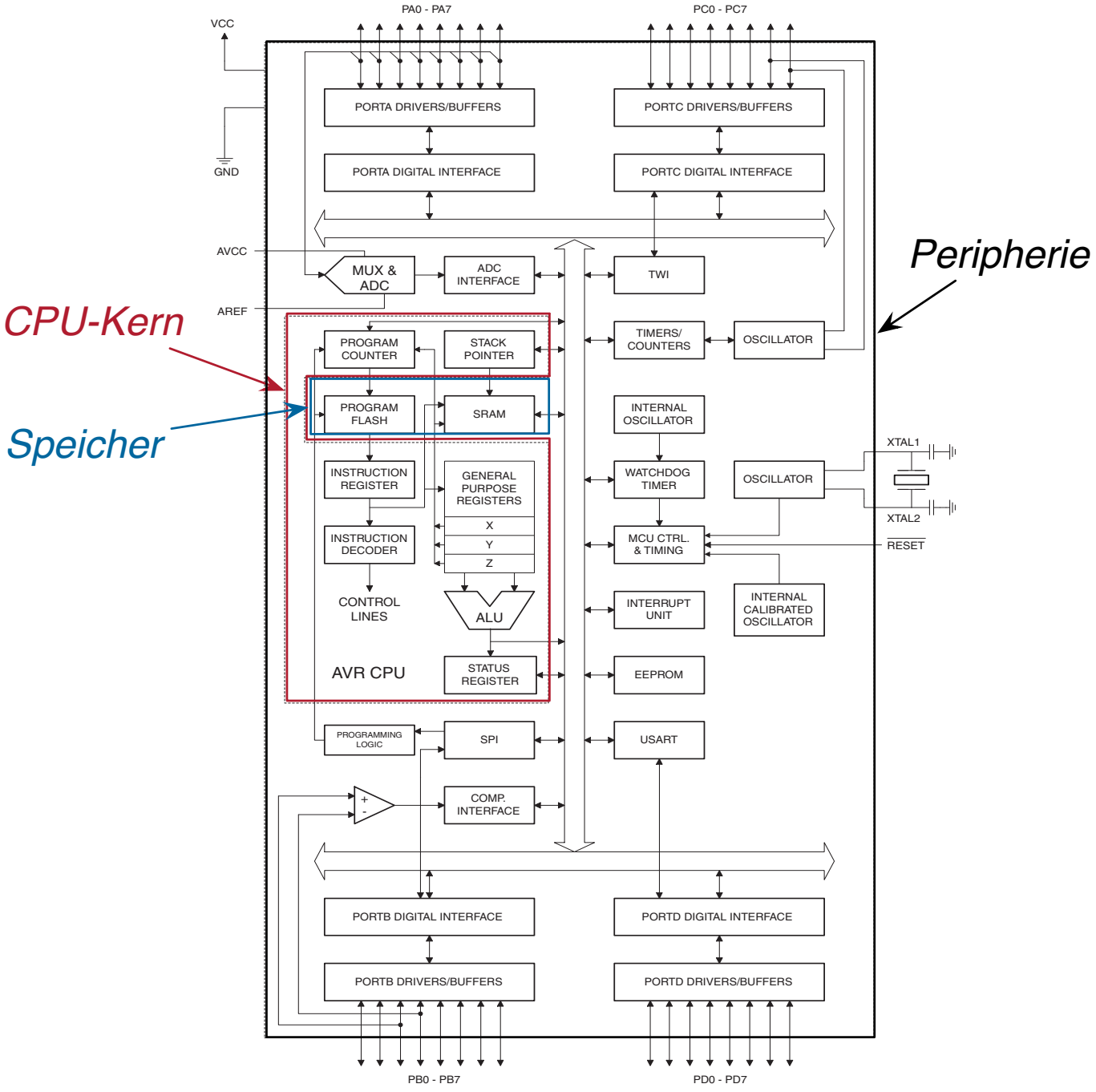
- Ein Zeiger verweist auf eine Variable im Speicher
  - Möglichkeit des indirekten Zugriffs auf den Wert
  - Grundlage für die Implementierung von *call-by-reference* in C
  - Grundlage für die Implementierung von Feldern
  - Wichtiges Element der Maschinennähe von C
  - **Häufigste Fehlerursache in C-Programmen**
- Die syntaktischen Möglichkeiten sind vielfältig (und verwirrend)
  - Typmodifizierer `*`, Adressoperator `&`, Verweisoperator `*`
  - Zeigerarithmetik mit `+`, `-`, `++` und `--`
  - syntaktische Äquivalenz zu Feldern (`[]` Operator)
- Zeiger können auch auf Funktionen verweisen
  - Übergeben von Funktionen an Funktionen
  - Prinzip der Rückruffunktion

Überblick: Teil C Systemnahe Softwareentwicklung

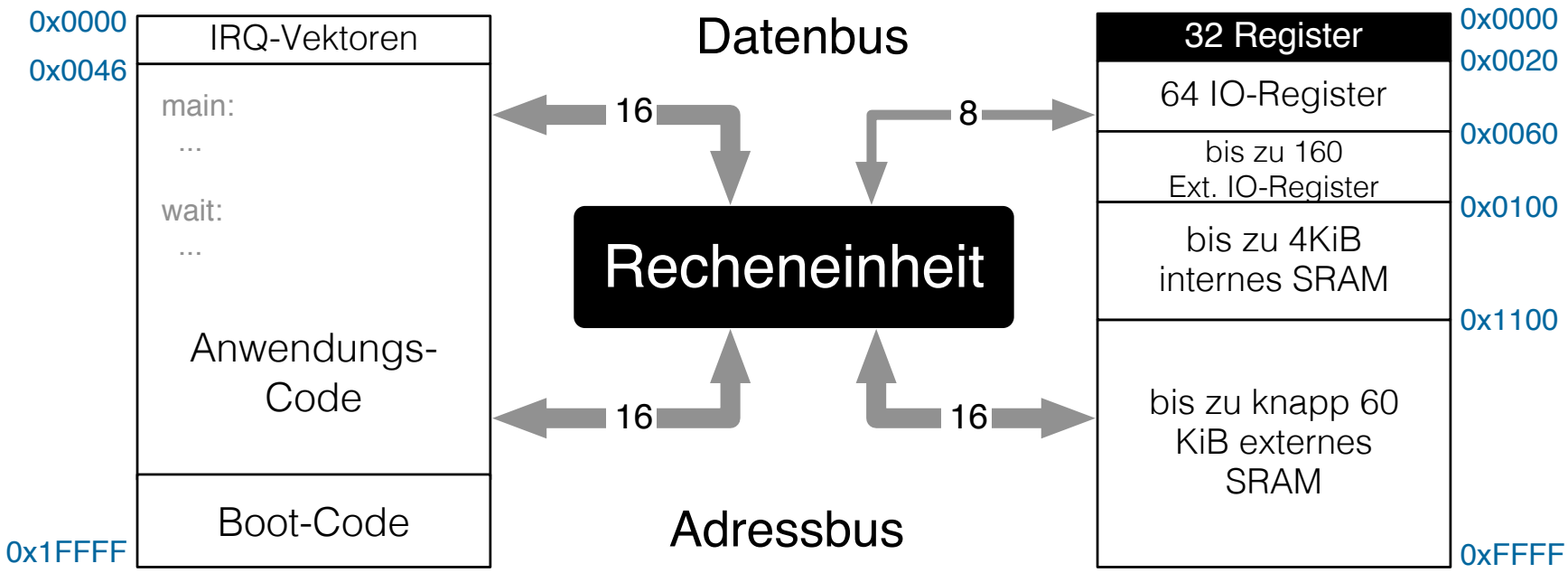
Was ist ein μ-Controller?

- **μ-Controller** := Prozessor + Speicher + Peripherie
  - Faktisch ein Ein-Chip-Computersystem $\longrightarrow$ SoC (*System-on-a-Chip*)
  - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher $\rightsquigarrow$ kostengünstiges Systemdesign
- Wesentliches Merkmal ist die (reichlich) enthaltene Peripherie
  - Timer/Counter (Zeiten/Ereignisse messen und zählen)
  - Ports (digitale Ein-/Ausgabe), A/D-Wandler (analoge Eingabe)
  - PWM-Generatoren (pseudo-analoge Ausgabe)
  - Bus-Systeme: SPI, RS-232, CAN, Ethernet, MLI, I^2C, ...
  - ...
- Die Abgrenzungen sind fließend: Prozessor $\longleftrightarrow$ μC $\longleftrightarrow$ SoC
  - AMD64-CPUs haben ebenfalls eingebaute Timer, Speicher (Caches), ...
  - Einige μC erreichen die Geschwindigkeit „großer Prozessoren“

Beispiel ATmega32: Blockschaltbild



Beispiel ATmega-Familie: CPU-Architektur



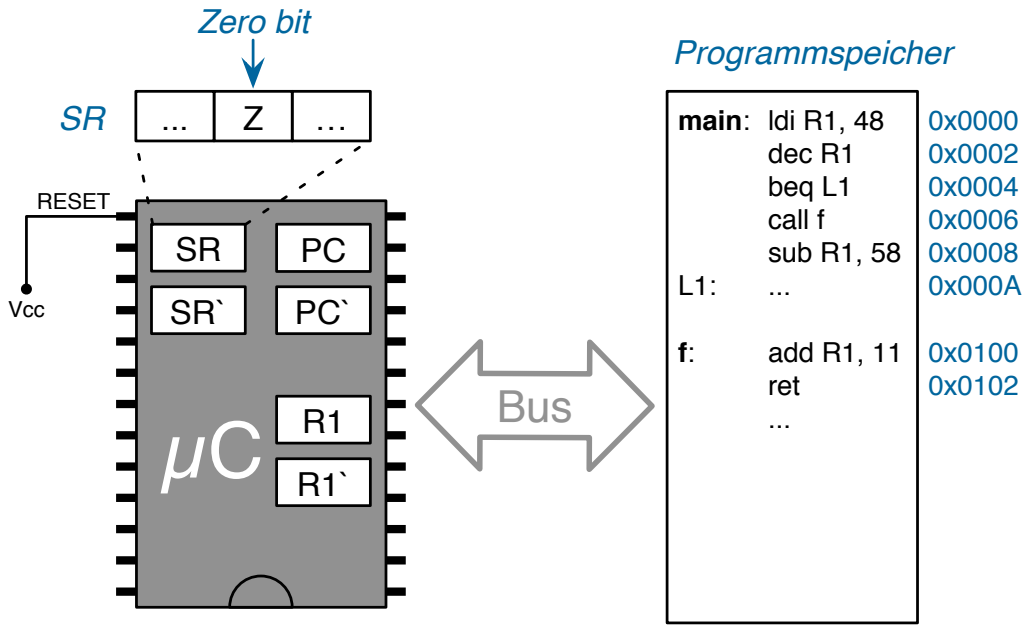
Programmadressraum (Flash)
1–128 KiB, 2-Byte-Elemente

Datenadressraum (SRAM)
0–64 KiB, 1-Byte-Elemente

- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingeblendet
  $\rightsquigarrow$ ansprechbar wie globale Variablen

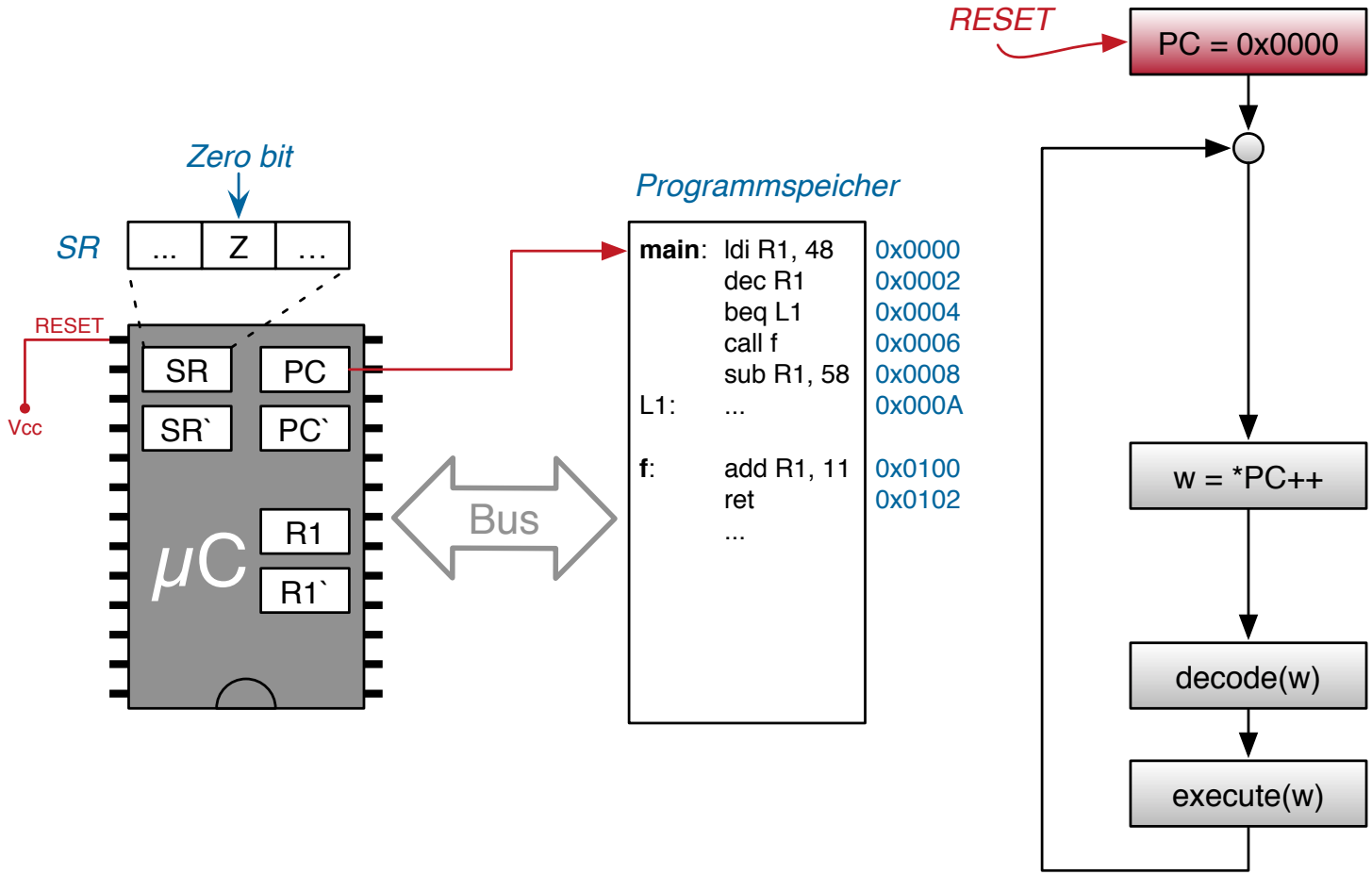
Zum Vergleich: PC basiert auf von-Neumann-Architektur [$\hookrightarrow$ GDI, 18-10] mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

Wie arbeitet ein Prozessor?

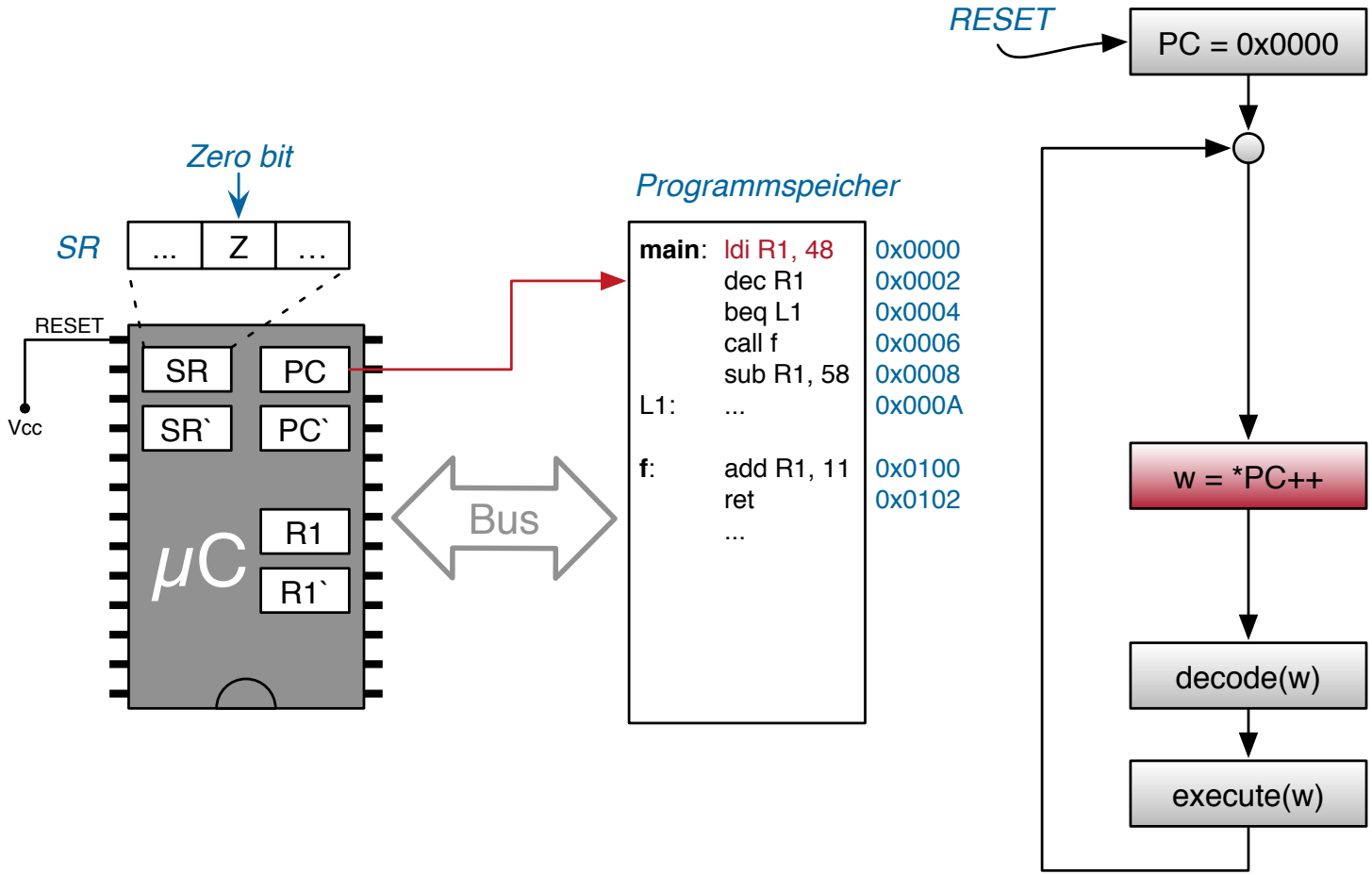


- Hier am Beispiel eines sehr einfachen Pseudoprozessors
  - Nur zwei Vielzweckregister (R1 und R2)
  - Programmzähler (PC) und Statusregister (SR) (+ „Schattenkopien“)
  - Kein Datenspeicher, kein Stapel $\rightsquigarrow$ Programm arbeitet nur auf Registern

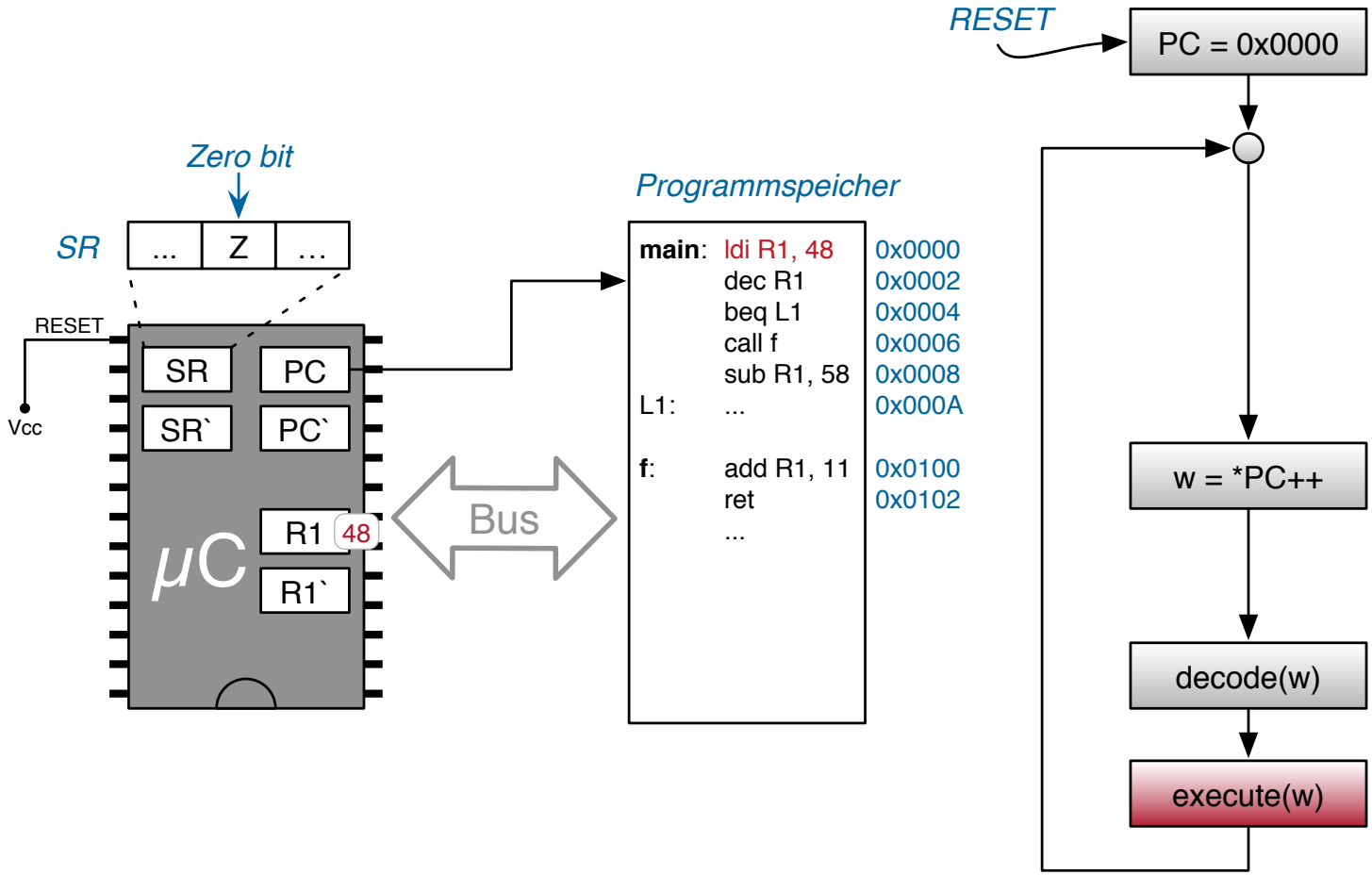
Wie arbeitet ein Prozessor?



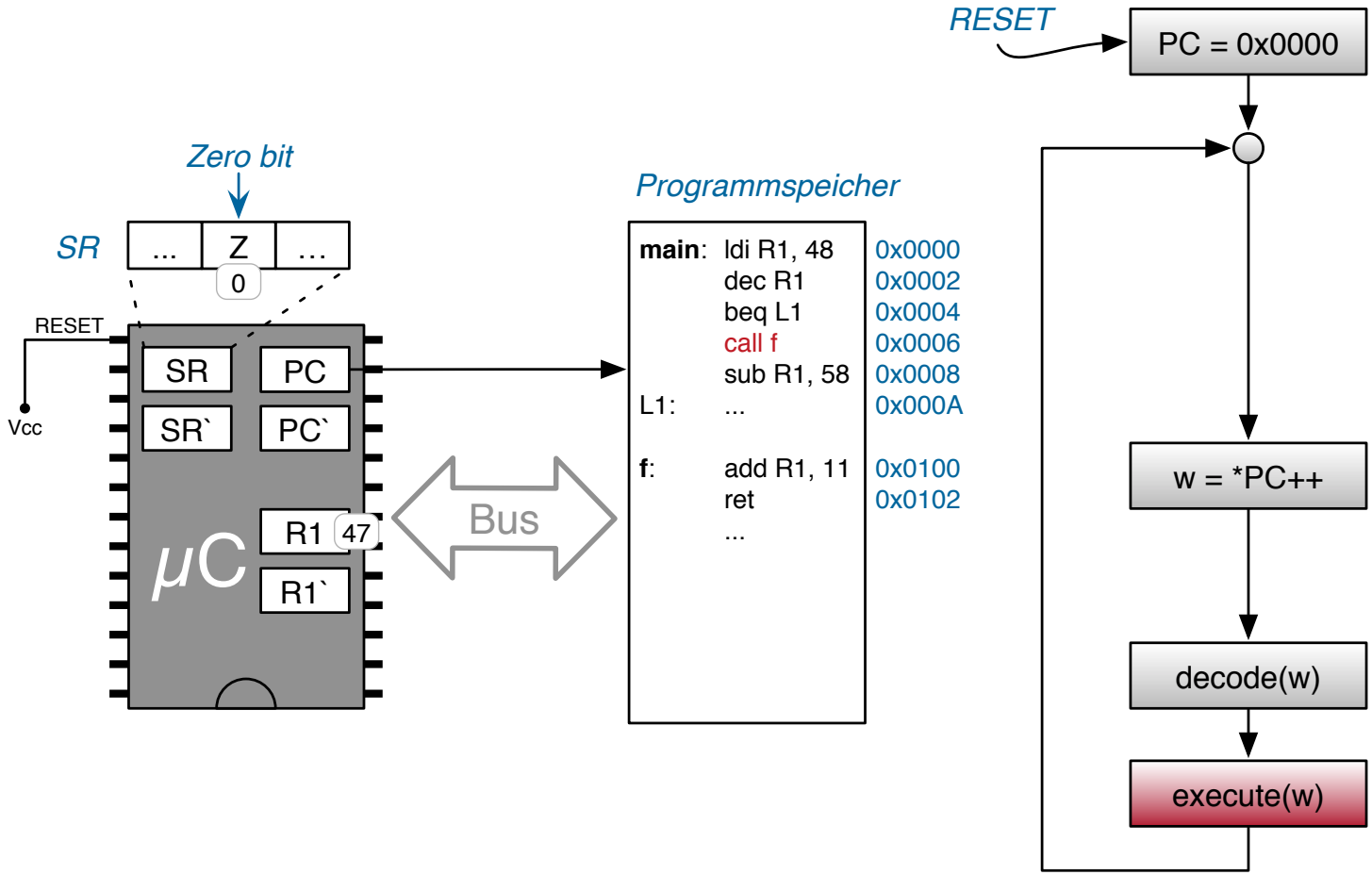
Wie arbeitet ein Prozessor?



Wie arbeitet ein Prozessor?



Wie arbeitet ein Prozessor?

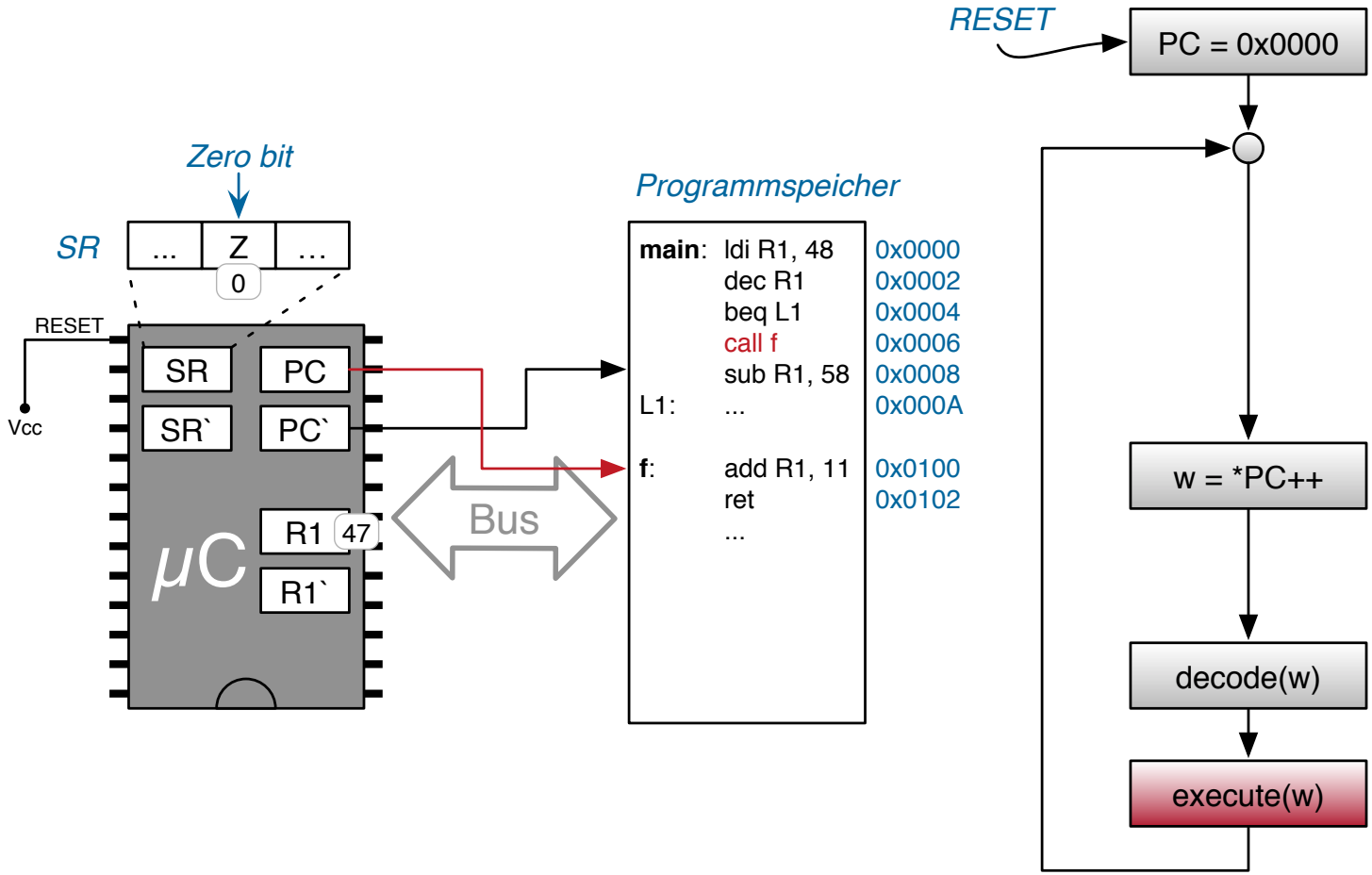


w: **dec** <R>
R –= 1
if(R == 0) Z = 1
else Z = 0

w: **beq** <lab>
if(Z) PC = lab

w: **call** <func>
PC` = PC
PC = func

Wie arbeitet ein Prozessor?

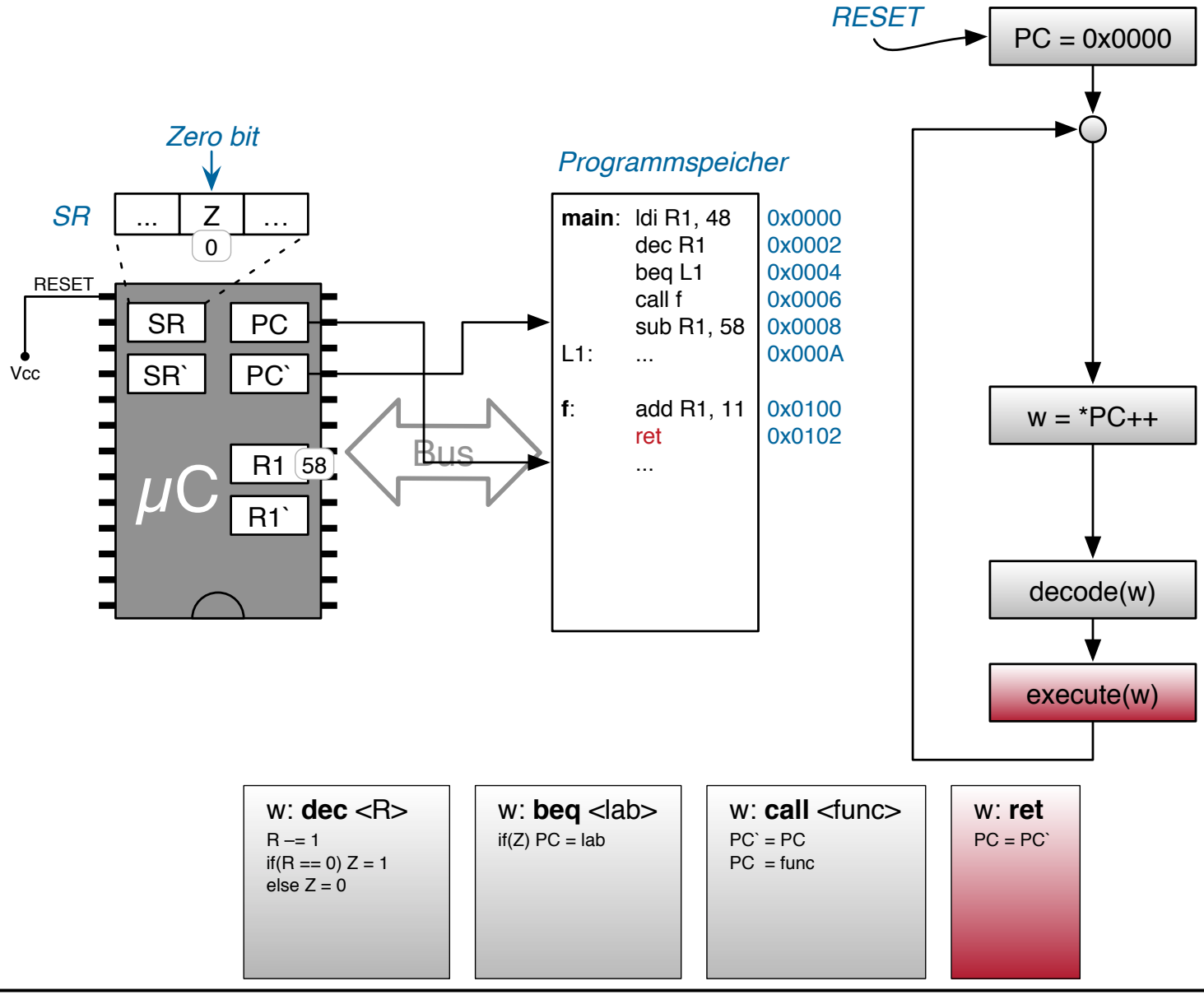


w: **dec** <R>
R –= 1
if(R == 0) Z = 1
else Z = 0

w: **beq** <lab>
if(Z) PC = lab

w: **call** <func>
PC` = PC
PC = func

Wie arbeitet ein Prozessor?



Peripheriegeräte

- **Peripheriegerät:** Hardwarekomponente, die sich „außerhalb“ der Zentraleinheit eines Computers befindet
  - Traditionell (PC): Tastatur, Bildschirm, ...
    ($\mapsto$ physisch „außerhalb“)
  - Allgemeiner: Hardwarefunktionen, die nicht direkt im Befehlssatz des Prozessors abgebildet sind
    ($\mapsto$ logisch „außerhalb“)
- Peripheriebausteine werden über I/O-Register angesprochen
  - Kontrollregister: Befehle an / Zustand der Peripherie wird durch Bitmuster kodiert (z. B. `DDRD` beim ATmega)
  - Datenregister: Dienen dem eigentlichen Datenaustausch
    (z. B. `PORTD`, `PIND` beim ATmega)
  - Register sind häufig für entweder nur Lesezugriffe (*read-only*) oder nur Schreibzugriffe (*write-only*) zugelassen

Peripheriegeräte: Beispiele

- Auswahl von typischen Peripheriegeräten in einem μ-Controller
  - **Timer/Counter** — Zählregister, die mit konfigurierbarer Frequenz (Timer) oder durch externe Signale (Counter) erhöht werden und bei konfigurierbarem Zählwert einen Interrupt auslösen.
  - **Watchdog-Timer** — Timer, der regelmäßig neu beschrieben werden muss oder sonst einen RESET auslöst („Totmannknopf“).
  - **(A)synchrone serielle Schnittstelle** — Bausteine zur seriellen (bitweisen) Übertragung von Daten mit synchronem (z. B. RS-232) oder asynchronem (z. B. I^2C) Protokoll.
  - **A/D-Wandler** — Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V $\mapsto$ 10-Bit-Zahl).
  - **PWM-Generatoren** — Bausteine zur Generierung von pulsweiten-modulierten Signalen (pseudo-analoge Ausgabe).
  - **Ports** — Gruppen von üblicherweise 8 Anschlüssen, die auf GND oder Vcc gesetzt werden können oder deren Zustand abgefragt werden kann. $\hookrightarrow$ 14–12

Peripheriegeräte – Register

- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
  - Memory-mapped: (Die meisten μC) — Register sind in den Adressraum eingeblendet; der Zugriff erfolgt über die Speicherbefehle des Prozessors (`load`, `store`)
  - Port-basiert: (x86-basierte PCs) — Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle `in`- und `out`-Befehle
- Die Registeradressen stehen in der Hardware-Dokumentation

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C	8
$3E ($5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8	11
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
$3C ($5C)	OCR0	Timer/Counter0 Output Compare Register								86
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	67
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	68

[1, S. 334]

Peripheriegeräte – Register (Forts.)

- Memory-mapped Register ermöglichen einen komfortablen Zugriff
  - Register $\mapsto$ Speicher $\mapsto$ Variable
  - Alle C-Operatoren stehen direkt zur Verfügung (z. B. `PORTD++`)
- Syntaktisch wird der Zugriff oft durch Makros erleichtert:

```
#define PORTD ( * (volatile uint8_t*)( 0x12 ) )
```

`0x12`: Adresse: int

`(volatile uint8_t*)(...)`: Adresse: volatile uint8_t* (Cast ↪ 7–17)

`*(...)`: Wert: volatile uint8_t (Dereferenzierung ↪ 13–4)

`PORTD` ist damit (syntaktisch) äquivalent zu einer `volatile uint8_t`-Variablen, die an Adresse `0x12` liegt

- Beispiel

```
#define PORTD (*(volatile uint8_t*)(0x12))

PORTD |= (1<<7);           // set D.7
uint8_t *pReg = &PORTD;    // get pointer to PORTD
*pReg &= ~(1<<7);          // use pointer to clear D.7
```

Registerzugriff und Nebenläufigkeit

- Peripheriegeräte arbeiten nebenläufig zur Software
  ⇝ Wert in einem Hardwareregister kann sich jederzeit ändern
- Dies widerspricht einer Annahme des Compilers
  - Variablenzugriffe erfolgen **nur** durch die aktuell ausgeführte Funktion
    ⇝ Variablen können in Registern zwischengespeichert werden

```
// C code
#define PIND (*(uint8_t*)(0x10))
void foo(void) {
  ...
  if( !(PIND & 0x2) ) {

    // button0 pressed
    ...
  }
  if( !(PIND & 0x4) ) {

    // button 1 pressed
    ...
  }
}
```

```
// Resulting assembly code

foo:
  lds   r24, 0x0010 // PIND->r24
  sbrc  r24, 1      // test bit 1
  rjmp L1
  // button0 pressed
  ...
L1:
  sbrc r24, 2       // test bit 2
  rjmp  L2

  ...
L2:
  ret
```

PIND wird nicht erneut aus dem Speicher geladen. Der Compiler nimmt an, dass der Wert in r24 aktuell ist.

Der `volatile`-Typmodifizierer

- **Lösung:** Variable `volatile` („*flüchtig, unbeständig*“) deklarieren
  - Compiler hält Variable nur so kurz wie möglich im Register
    - ⇝ Wert wird unmittelbar vor Verwendung gelesen
    - ⇝ Wert wird unmittelbar nach Veränderung zurückgeschrieben

```
// C code
#define PIND \
  (*(volatile uint8_t*)(0x10))
void foo(void) {
  ...
  if( !(PIND & 0x2) ) {

    // button0 pressed
    ...
  }

  if( !(PIND & 0x4) ) {

    // button 1 pressed
    ...
  }
}
```

```
// Resulting assembly code


foo:
  lds  r24, 0x0010 // PIND->r24
  sbrc r24, 1      // test bit 1
  rjmp L1
  // button0 pressed
  ...
L1:
  lds  r24, 0x0010 // PIND->r24
  sbrc r24, 2      // test bit 2
  rjmp L2

  ...
L2:
  ret
```

`PIND` ist `volatile` und wird deshalb vor dem Test erneut aus dem Speicher geladen.

Der `volatile`-Typmodifizierer (Forts.)

- Die `volatile`-Semantik verhindert viele Code-Optimierungen (insbesondere das Entfernen von scheinbar unnützem Code)
- Kann ausgenutzt werden, um aktives Warten zu implementieren:

```
// C code
void wait( void ){
  for( uint16_t i = 0; i<0xffff; i++)

}
```

volatile!

```
// Resulting assembly code
wait:
  // compiler has optimized
  // "unneeded" loop
  ret
```

Achtung: volatile $\mapsto$ **$$$**

Die Verwendung von `volatile` verursacht erhebliche Kosten

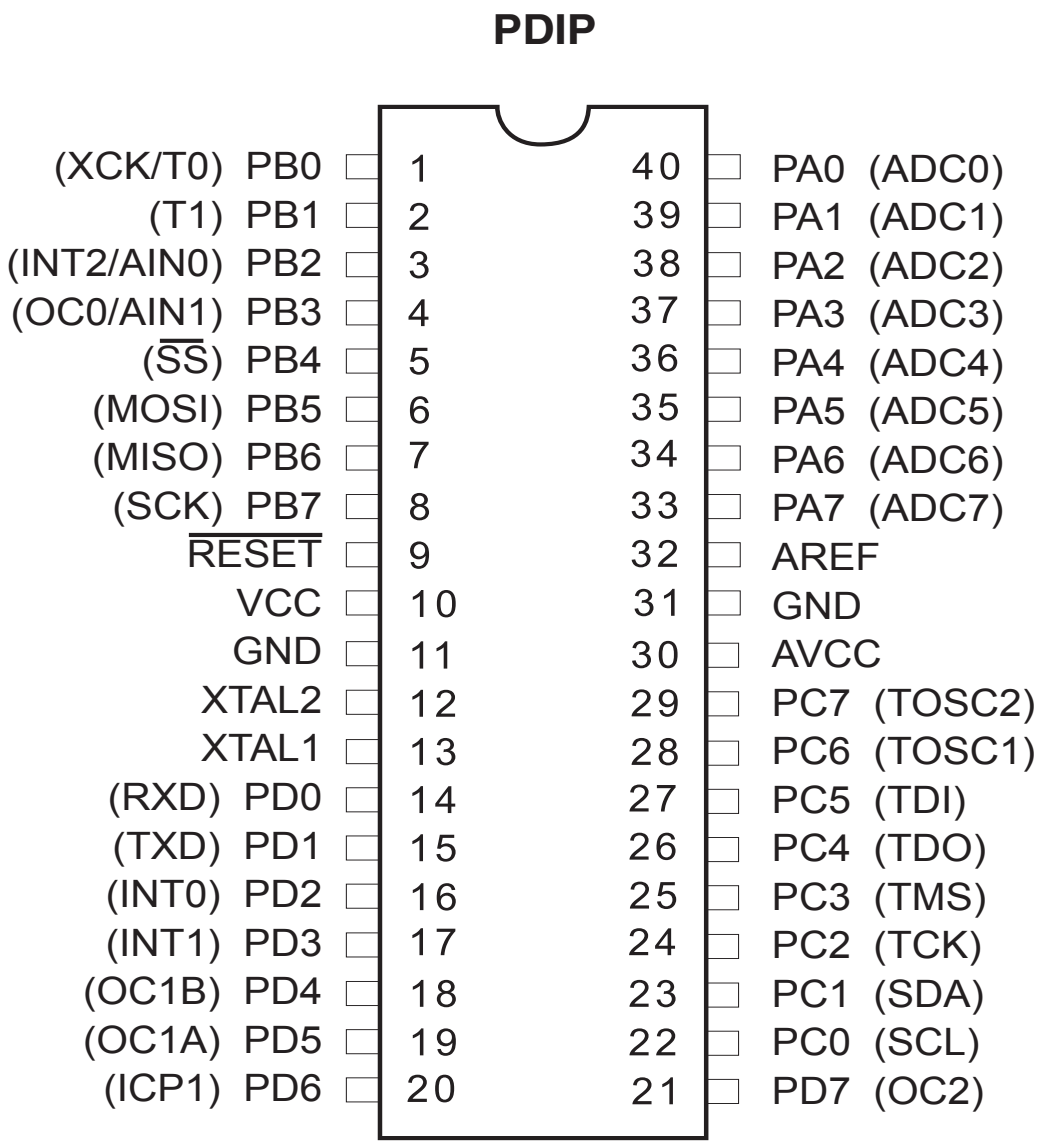
- Werte können nicht mehr in Registern gehalten werden
- Viele Code-Optimierungen können nicht durchgeführt werden

Regel: `volatile` wird nur in **begründeten Fällen** verwendet

Peripheriegeräte: Ports

- **Port** := Gruppe von (üblicherweise 8) digitalen Ein-/Ausgängen
  - Digitaler Ausgang: Bitwert $\mapsto$ Spannungspegel an μC-Pin
  - Digitaler Eingang: Spannungspegel an μC-Pin $\mapsto$ Bitwert
  - Externer Interrupt: Spannungspegel an μC-Pin $\mapsto$ Bitwert
    (bei Pegelwechsel) $\rightsquigarrow$ Prozessor führt Interruptprogramm aus
- Die Funktion ist üblicherweise pro Pin konfigurierbar
  - Eingang
  - Ausgang
  - Externer Interrupt (nur bei bestimmten Eingängen)
  - Alternative Funktion (Pin wird von anderem Gerät verwendet)

Beispiel ATmega32: Port/Pin-Belegung



Aus Kostengründen ist nahezu jeder Pin doppelt belegt, die Konfiguration der gewünschten Funktion erfolgt durch die Software.

Beim SPiCboard werden z. B. Pins 39–40 als ADCs konfiguriert, um Poti und Photosensor anzuschließen.

PORTA steht daher nicht zur Verfügung.

Beispiel ATmega32: Port-Register

- Pro Port x sind drei Register definiert (Beispiel für $x = D$)

- `DDRx` **Data Direction Register:** Legt für jeden Pin i fest, ob er als Eingang (Bit i=0) oder als Ausgang (Bit i=1) verwendet wird.

7	6	5	4	3	2	1	0
DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- `PORTx` **Data Register:** Ist Pin i als Ausgang konfiguriert, so legt Bit i den Pegel fest (0=GND sink, 1=Vcc source). Ist Pin i als Eingang konfiguriert, so aktiviert Bit i den internen Pull-Up-Widerstand (1=aktiv).

7	6	5	4	3	2	1	0
PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

- `PINx` **Input Register:** Bit i repräsentiert den Pegel an Pin i (1=*high*, 0=*low*), unabhängig von der Konfiguration als Ein-/Ausgang.

7	6	5	4	3	2	1	0
PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
R	R	R	R	R	R	R	R

Verwendungsbeispiele: ↪ 3–4 und ↪ 3–7 [1, S. 66]

Strukturen: Motivation

- Jeder Port wird durch *drei* globale Variablen verwaltet
  - Es wäre besser diese zusammen zu fassen
  - „problembezogene Abstraktionen“
  - „Trennung der Belange“

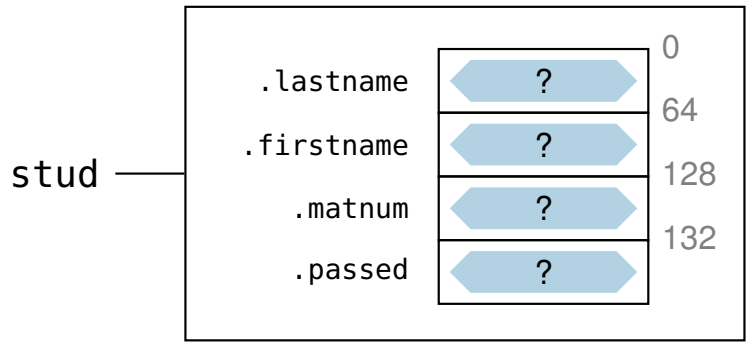
↪ 4–1
↪ 12–4

- Dies geht in C mit Verbundtypen (Strukturen)

```
// Structure declaration
struct Student {
  char  lastname[64];
  char  firstname[64];
  long  matnum;
  int   passed;
};

// Variable definition
struct Student stud;
```

Ein Strukturtyp fasst eine Menge von Daten zu einem gemeinsamen Typ zusammen.

Die Datenelemente werden hintereinander im Speicher abgelegt.



Strukturen: Variablendefinition und -initialisierung

- Analog zu einem Array kann eine Strukturvariable bei Definition elementweise initialisiert werden

↪ 13–8

```
struct Student {
  char  lastname[64];
  char  firstname[64];
  long  matnum;
  int   passed;
};
```

```
struct Student stud = { "Meier", "Hans",
                        4711, 0 };
```

Die Initialisierer werden nur über ihre Reihenfolge, nicht über ihren Bezeichner zugewiesen.
⇝ Potentielle Fehlerquelle bei Änderungen!

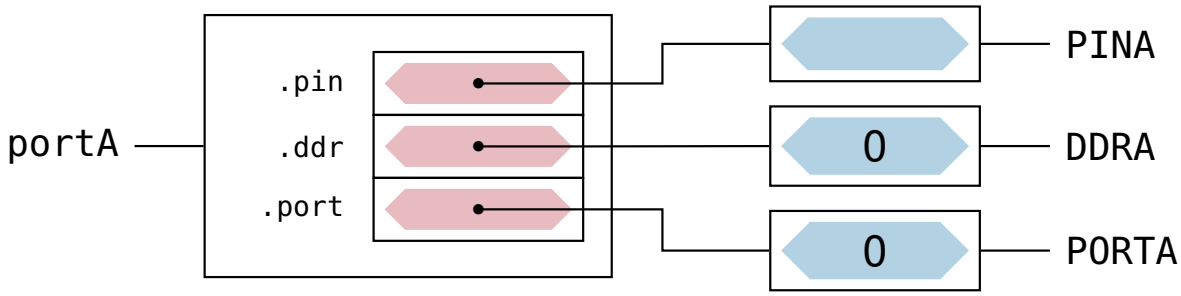
- Analog zur Definition von `enum`-Typen kann man mit `typedef` die Verwendung vereinfachen

↪ 6–8

```
typedef struct {
  volatile uint8_t *pin;
  volatile uint8_t *ddr;
  volatile uint8_t *port;
} port_t;
```

```
port_t portA = { &PINA, &DDRA, &PORTA };
port_t portD = { &PIND, &DDRD, &PORTD };
```


Strukturen: Elementzugriff

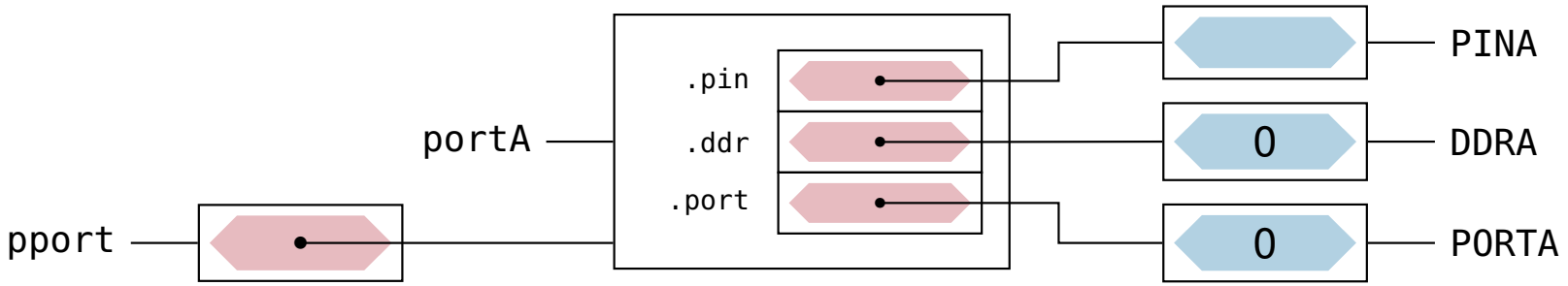


- Auf Strukturelemente wird mit dem `.`-Operator zugegriffen [≈Java]

```
port_t portA = { &PINA, &DDRA, &PORTA };

*portA.port = 0;    // clear all pins
*portA.ddr  = 0xff; // set all to input
```

Beachte: `.` hat eine höhere Priorität als `*`

Strukturen: Elementzugriff



- Bei einem Zeiger auf eine Struktur würde Klammerung benötigt

```
port_t * pport = &portA;  // p --> portA

*(*pport).port = 0;       // clear all pins
*(*pport).ddr  = 0xff;    // set all to output
```

- Mit dem `->`-Operator lässt sich dies vereinfachen `s->m` ≡ `(*s).m`

```
port_t * pport = &portA;  // p --> portA

*pport->port = 0;       // clear all pins
*pport->ddr  = 0xff;    // set all to output
```

-> hat **ebenfalls** eine höhere Priorität als *

Strukturen als Funktionsparameter

- Im Gegensatz zu Arrays werden Strukturen *by-value* übergeben

```
void initPort( port_t p ){
  *p.port = 0;         // clear all pins
  *p.ddr  = 0xff;      // set all to output

  p.port  = &PORTD;    // no effect, p is local variable
}

void main(){ initPort( portA ); ··· }
```

- Bei größeren Strukturen wird das **sehr ineffizient**
  - Z. B. `Student` (↪ 14–15): Jedes mal 134 Byte allozieren und kopieren
  - Besser man übergibt einen Zeiger auf eine konstante Struktur

```
void initPort( const port_t *p ){
  *p->port = 0;          // clear all pins
  *p->ddr  = 0xff;       // set all to output

  // p->port  = &PORTD;   compile-time error, *p is const!
}

void main(){ initPort( &portA ); ··· }
```


Bit-Strukturen: Bitfelder

- Strukturelemente können auf Bit-Granularität festgelegt werden
  - Der Compiler fasst Bitfelder zu passenden Ganzzahltypen zusammen
  - Nützlich, um auf einzelne Bit-Bereiche eines Registers zuzugreifen

- Beispiel
  - `MCUCR` **MCU Control Register:** Steuert Power-Management-Funktionen und Auslöser für externe Interrupt-Quellen `INT0` und `INT1`. [1, S. 36+69]

7	6	5	4	3	2	1	0
SE	**SM2**	**SM1**	**SM0**	**ISC11**	**ISC10**	**ISC01**	**ISC00**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

```
typedef struct {
  uint8_t ISC0  : 2;  // bit 0-1: interrupt sense control INT0
  uint8_t ISC1  : 2;  // bit 2-3: interrupt sense control INT1
  uint8_t SM    : 3;  // bit 4-6: sleep mode to enter on sleep
  uint8_t SE    : 1;  // bit 7  : sleep enable
} MCUCR_t;
```

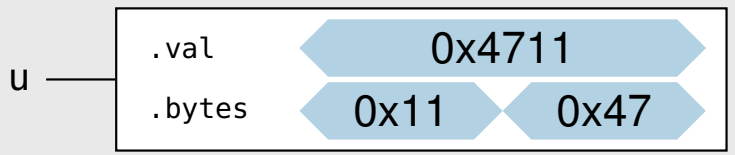

Unions

- In einer Struktur liegen die Elemente hintereinander im Speicher, in einer Union hingegen übereinander
  - Wert im Speicher lässt sich verschieden (Typ)-interpretieren
  - Nützlich für bitweise Typ-Casts

↪ 14–15

- Beispiel

```
void main(){
  union {
    uint16_t  val;
    uint8_t   bytes[2];
  } u;

  u.val = 0x4711;

  // show high-byte
  sb_7seg_showHexNumber( u.bytes[1] );
  ...
  // show  low-byte
  sb_7seg_showHexNumber( u.bytes[0] );
  ...
}
```



Unions und Bit-Strukturen: Anwendungsbeispiel

- Unions werden oft mit Bit-Feldern kombiniert, um ein Register wahlweise „im Ganzen“ oder bitweise ansprechen zu können

```
typedef union {
  volatile uint8_t  reg;  // complete register
  volatile struct {
    uint8_t ISC0  : 2;    // components
    uint8_t ISC1  : 2;
    uint8_t SM    : 3;
    uint8_t SE    : 1;
  };
} MCUCR_t;

void foo( void ) {
  MCUCR_t *mcucr = (MCUCR_t *) (0x35);
  uint8_t oldval = mcucr->reg;  // save register
  ...
  mcucr->ISC0 = 2;              // use register
  mcucr->SE   = 1;              // ···
  ...
  mcucr->reg  = oldval;         // restore register
}
```


Systemnahe Programmierung in C (SPiC)

Teil D Betriebssystemabstraktionen

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

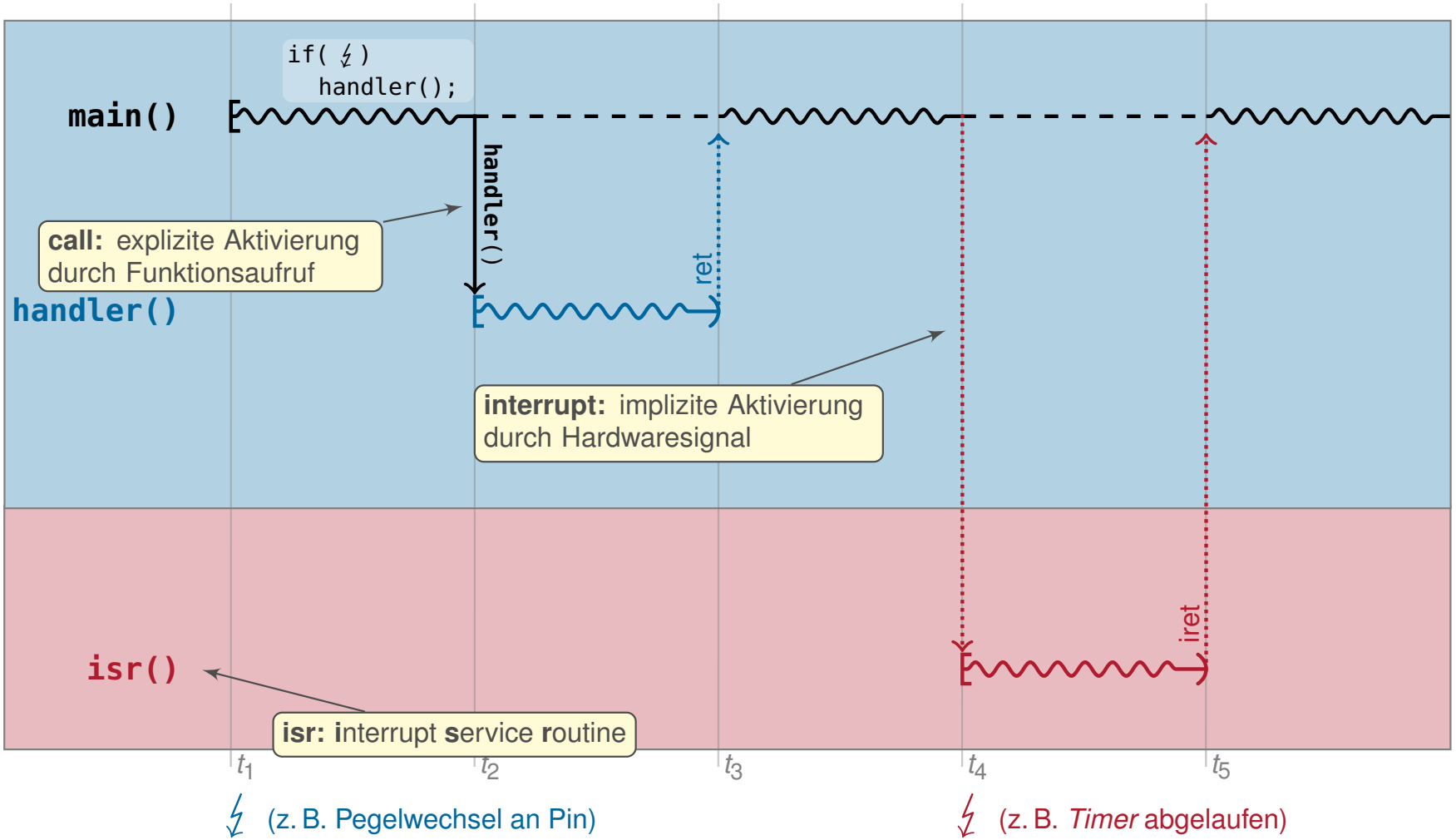
http://www4.cs.fau.de/Lehre/SS16/V_SPIC

Überblick: Teil D Betriebssystemabstraktionen

Ereignisbehandlung

- Bei einem Peripheriegerät tritt ein Ereignis (↯) auf ↪ 14–5
  - Signal an einem Port-Pin wechselt von *low* auf *high*
  - Ein *Timer* ist abgelaufen
  - Ein A/D-Wandler hat einen neuen Wert vorliegen
  - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren
  - **Polling:** Das Programm überprüft den Zustand regelmäßig und ruft ggf. eine Bearbeitungsfunktion auf.
  - **Interrupt:** Gerät „meldet“ sich beim Prozessor, der daraufhin in eine Bearbeitungsfunktion verzweigt.

Interrupt ↦ Funktionsaufruf „von außen“



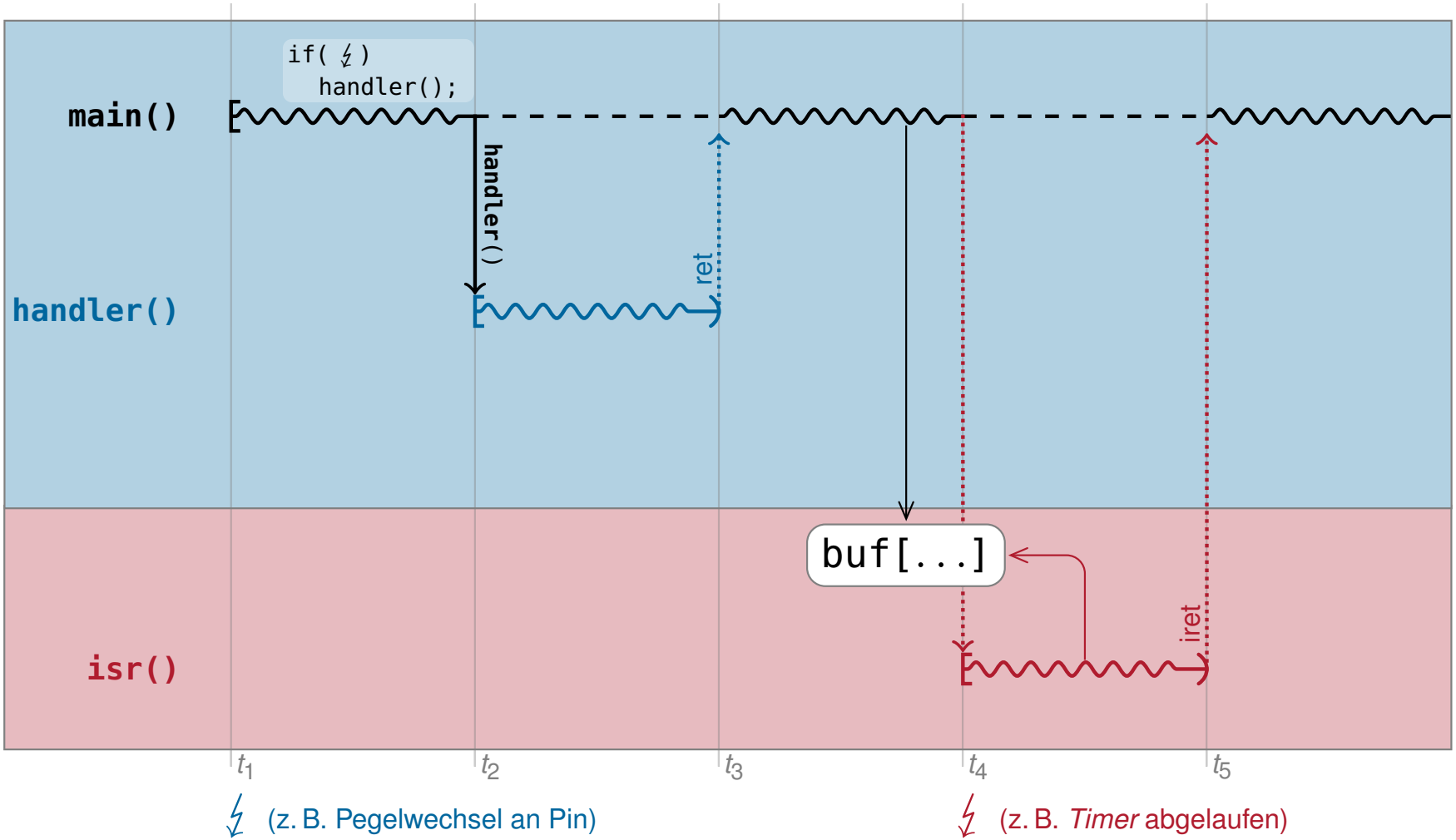
Polling vs. Interrupts – Vor- und Nachteile

- Polling ($\mapsto$ „Taktgesteuertes System“)
  - Ereignisbearbeitung erfolgt synchron zum Programmablauf
    - – Ereigniserkennung über das Programm „verstreut“ (Trennung der Belange)
    - – „Verschwendung“ von Prozessorzeit (falls anderweitig verwendbar)
    - – Hochfrequentes Pollen $\rightsquigarrow$ hohe Prozessorlast $\rightsquigarrow$ hoher Energieverbrauch
    - \+ Implizite Datenkonsistenz durch festen, sequentiellen Programmablauf
    - \+ Programmverhalten gut vorhersagbar

- Interrupts ($\mapsto$ „Ereignisgesteuertes System“)
  - Ereignisbearbeitung erfolgt asynchron zum Programmablauf
    - \+ Ereignisbearbeitung kann im Programmtext gut separiert werden
    - \+ Prozessor wird nur beansprucht, wenn Ereignis tatsächlich eintritt
    - – Höhere Komplexität durch Nebenläufigkeit $\rightsquigarrow$ Synchronisation erforderlich
    - – Programmverhalten schwer vorhersagbar

Beide Verfahren bieten spezifische Vor- und Nachteile
$\rightsquigarrow$ Auswahl anhand des konkreten Anwendungsszenarios

Interrupt $\mapsto$ unvorhersagbarer Aufruf „von außen“



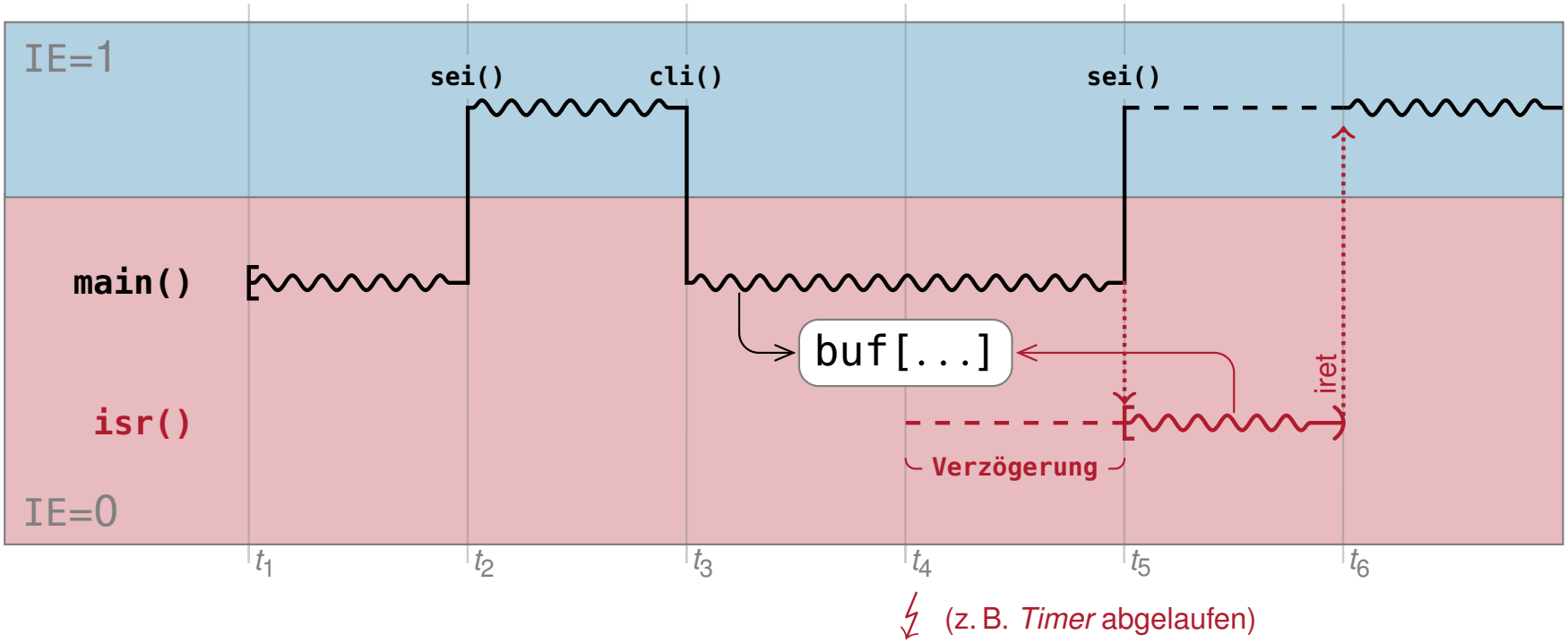
Interruptsperren

- Zustellung von Interrupts kann softwareseitig gesperrt werden
  - Wird benötigt zur Synchronisation mit ISRs
  - Einzelne ISR: Bit in gerätespezifischem Steuerregister
  - Alle ISRs: Bit (`IE`, *Interrupt Enable*) im Statusregister der CPU
- Auflaufende IRQs werden (üblicherweise) gepuffert
  - Maximal einer pro Quelle!
  - Bei längeren Sperrzeiten können IRQs verloren gehen!

IRQ $\mapsto$ *Interrupt ReQuest*

- Das `IE`-Bit wird beeinflusst durch:
  - Prozessor-Befehle: `cli`: `IE`$\leftarrow$0 (*clear interrupt*, IRQs gesperrt)
    `sei`: `IE`$\leftarrow$1 (*set interrupt*, IRQs erlaubt)
  - Nach einem RESET: `IE`=0 $\rightsquigarrow$ IRQs sind zu Beginn des Hauptprogramms gesperrt
  - Bei Betreten einer ISR: `IE`=0 $\rightsquigarrow$ IRQs sind während der Interruptbearbeitung gesperrt

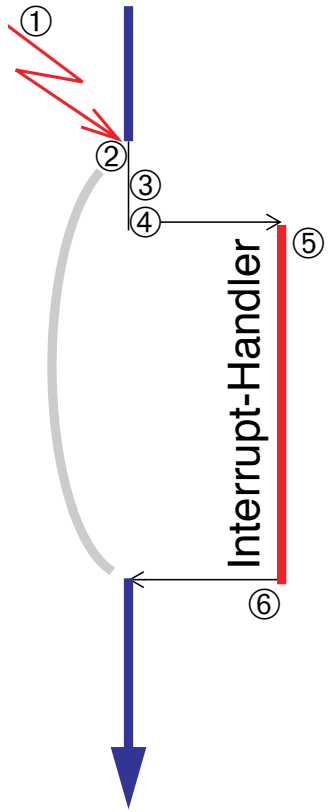
Interruptsperren: Beispiel



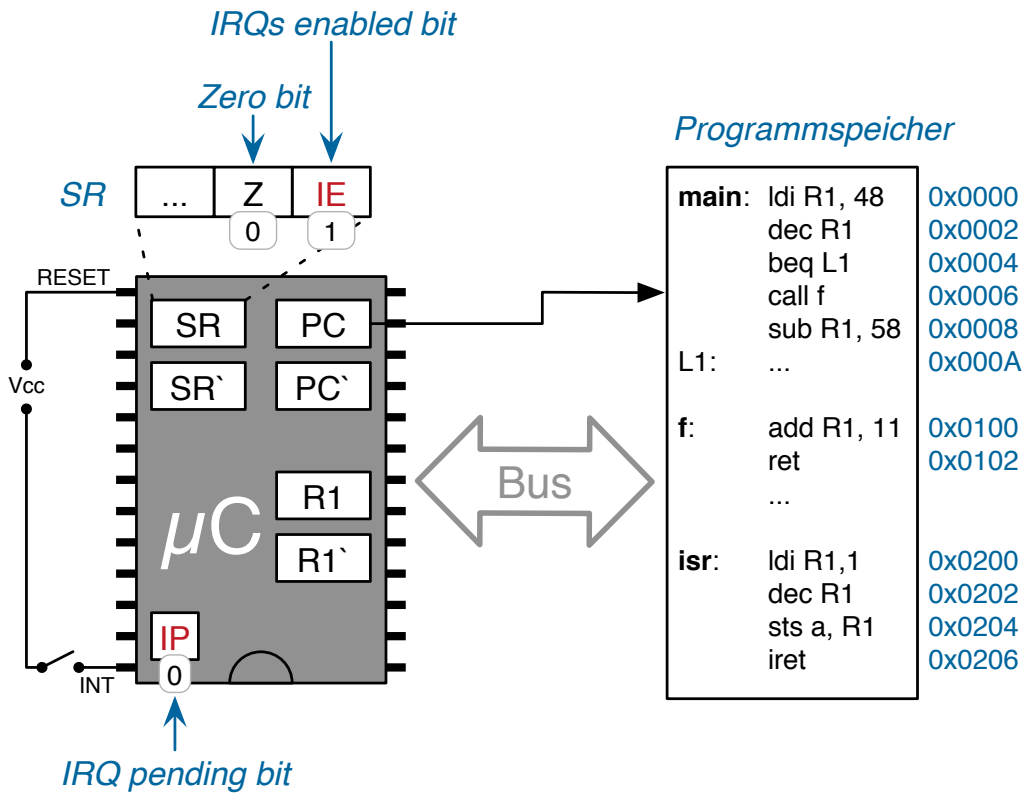
- t_1 Zu Beginn von `main()` sind IRQs gesperrt (`IE`=0)
- t_2, t_3 Mit `sei()` / `cli()` werden IRQs freigegeben (`IE`=1) / erneut gesperrt
- t_4 ↯ aber `IE`=0 ⇝ Bearbeitung ist unterdrückt, IRQ wird gepuffert
- t_5 `main()` gibt IRQs frei (`IE`=1) ⇝ gepufferter IRQ „schlägt durch“
- t_5–t_6 Während der ISR-Bearbeitung sind die IRQs gesperrt (`IE`=0)
- t_6 Unterbrochenes `main()` wird fortgesetzt

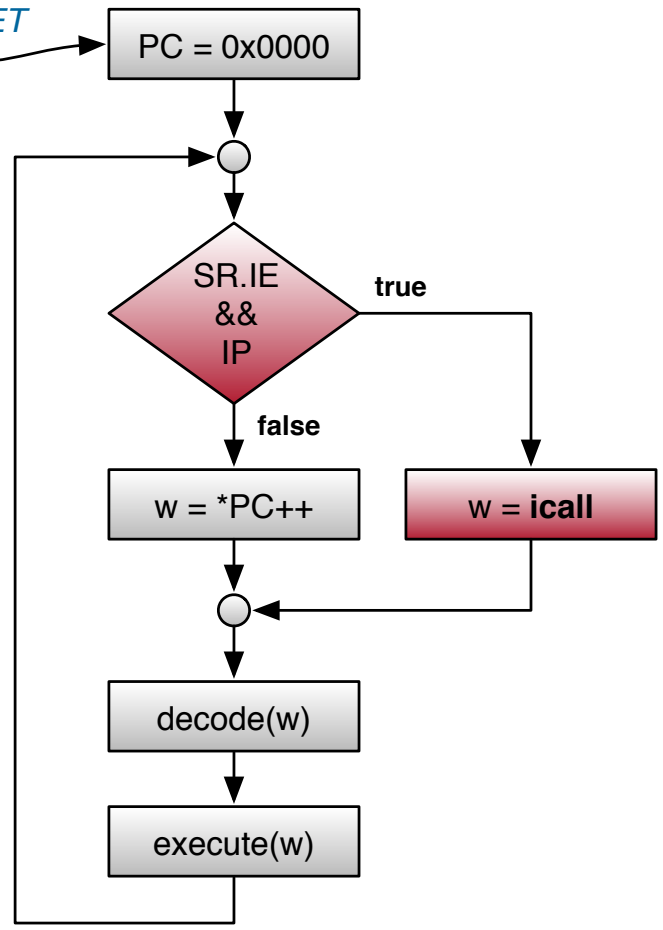
Ablauf eines Interrupts – Überblick

1. Gerät signalisiert Interrupt
   - Anwendungsprogramm wird „unmittelbar“ (vor dem nächsten Maschinenbefehl mit `IE`=1) unterbrochen
2. Die Zustellung weiterer Interrupts wird gesperrt (`IE`=0)
   - Zwischenzeitlich auflaufende Interrupts werden gepuffert (maximal einer pro Quelle!)
3. Registerinhalte werden gesichert (z. B. im Stapel)
   - PC und Statusregister automatisch von der Hardware
   - Vielzweckregister üblicherweise manuell in der ISR
4. Aufzurufende ISR (Interrupt-Handler) wird ermittelt
5. ISR wird ausgeführt
6. ISR terminiert mit einem „return from interrupt“-Befehl
   - Registerinhalte werden restauriert
   - Zustellung von Interrupts wird freigegeben (`IE`=1)
   - Das Anwendungsprogramm wird fortgesetzt



Ablauf eines Interrupts – Details





- Hier als Erweiterung unseres einfachen Pseudoprozessors ↪ 14–4
  - Nur eine Interruptquelle
  - Sämtliche Register werden von der Hardware gerettet

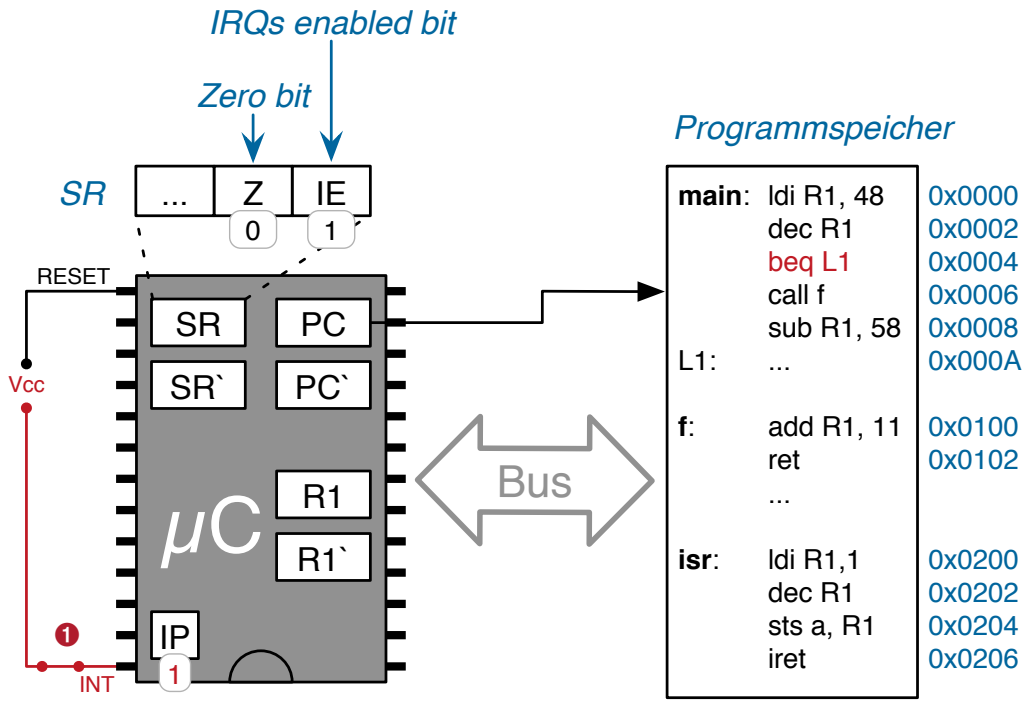
w: call <func>
PC` = PC
PC = func

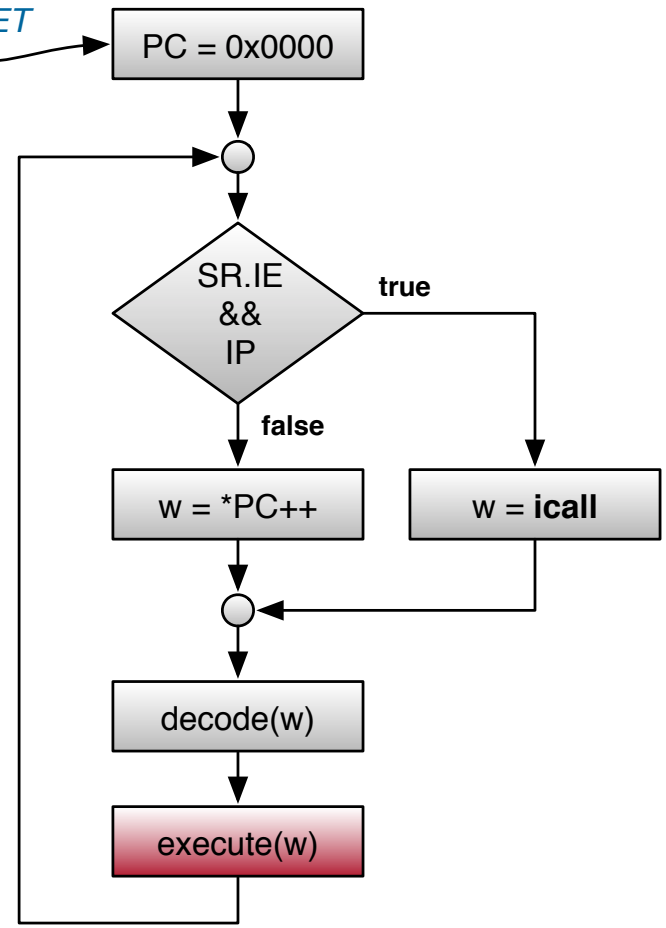
w: ret
PC = PC`

w: icall
SR` = SR
SR.IE = 0
IP = 0
PC` = PC
PC = isr
R1` = R1

w: iret
SR = SR`
PC = PC`
R1 = R1`

Ablauf eines Interrupts – Details

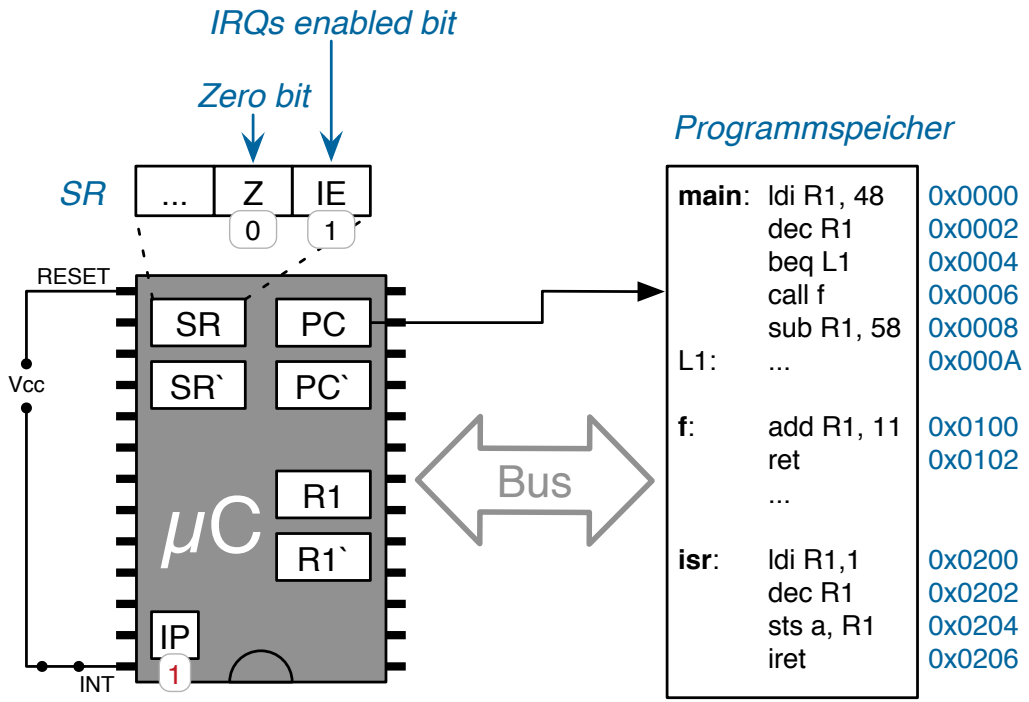




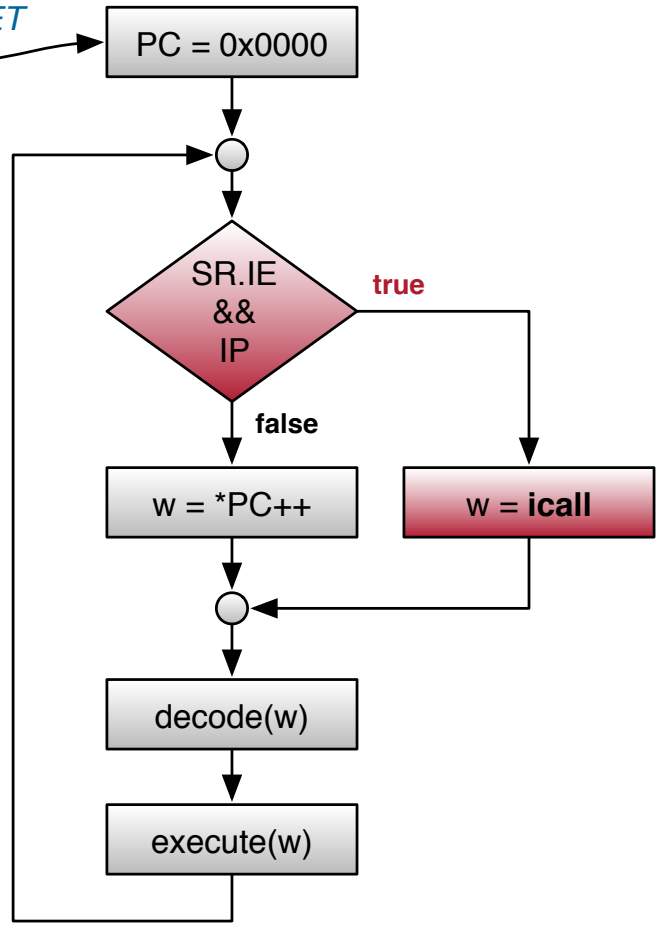
❶ Gerät signalisiert Interrupt (aktueller Befehl wird noch fertiggestellt)

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Ablauf eines Interrupts – Details

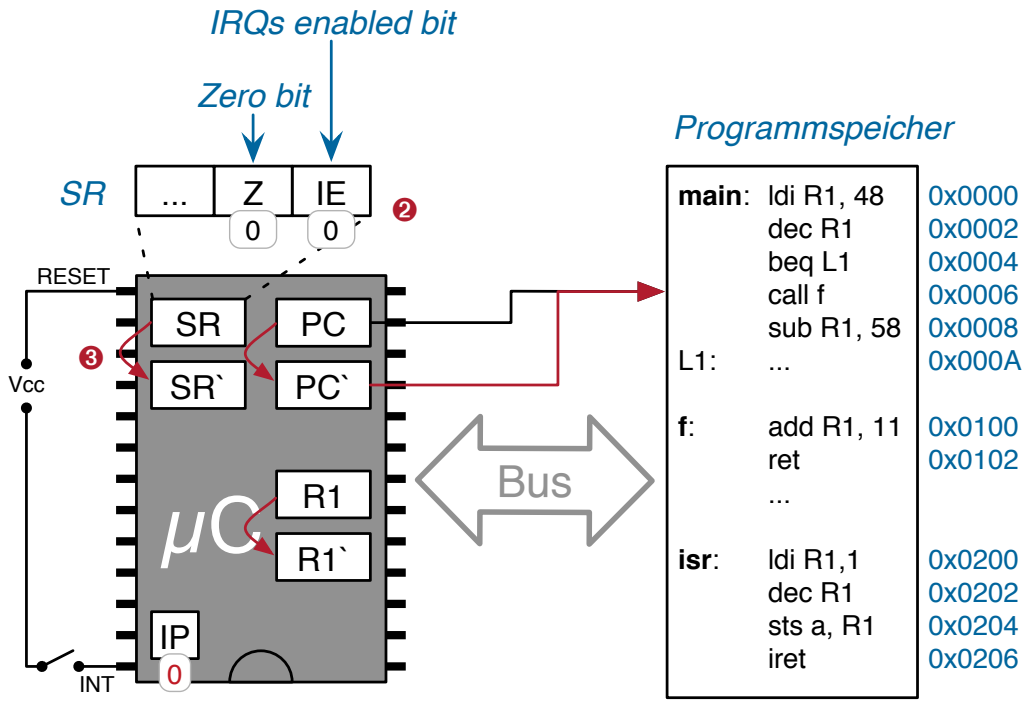


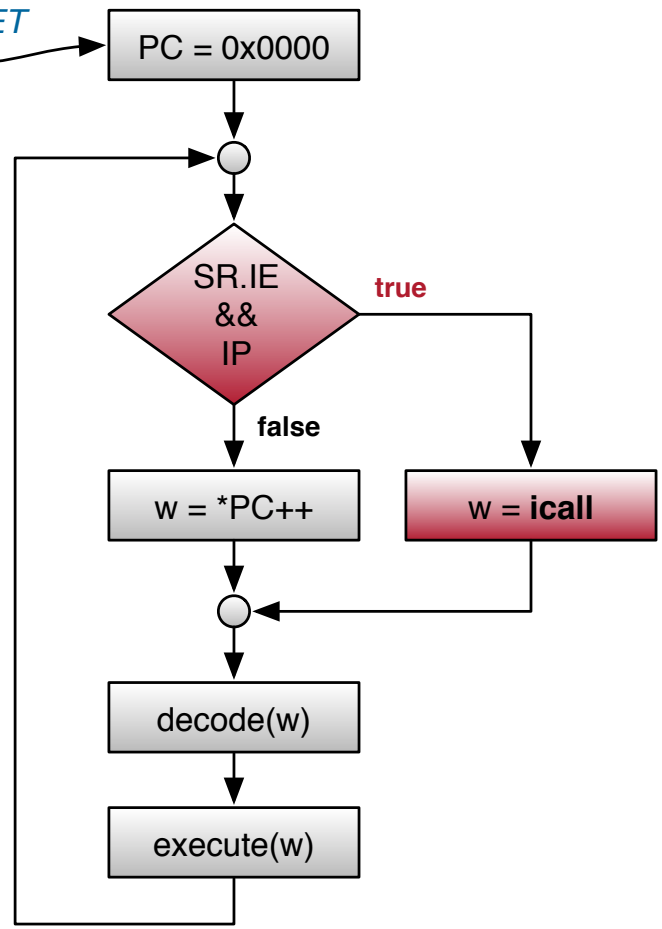
(Vor dem nächsten *instruction fetch* wird der Interruptstatus überprüft)



w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Ablauf eines Interrupts – Details





❷ Die Zustellung weiterer Interrupts wird verzögert
❸ Registerinhalte werden gesichert

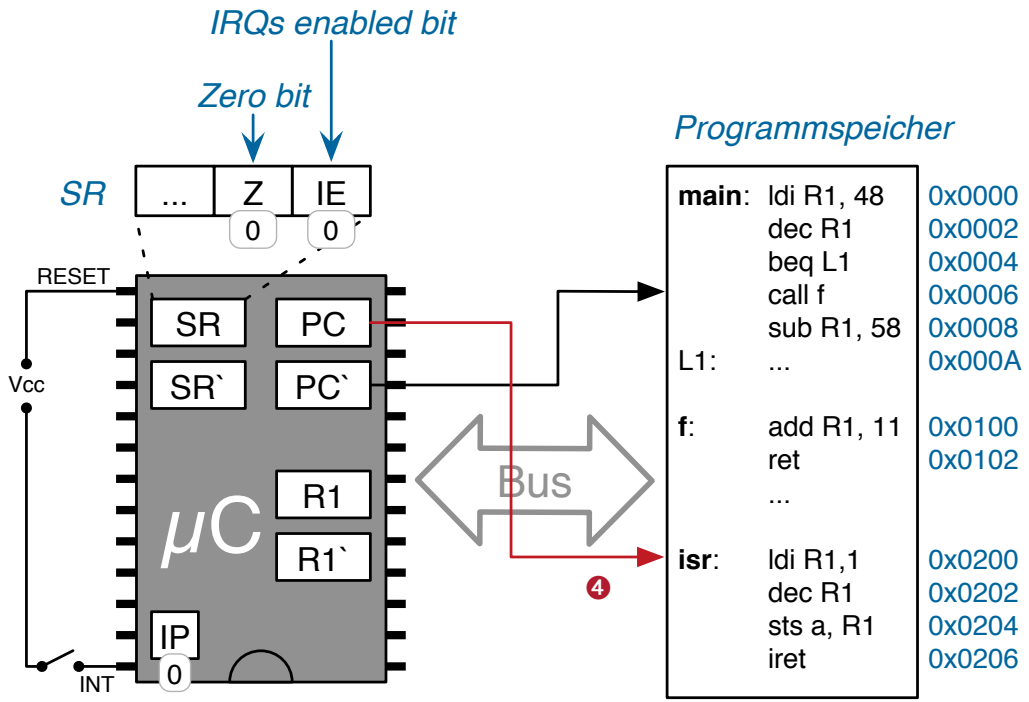
w: call <func>
PC` = PC
PC = func

w: ret
PC = PC`

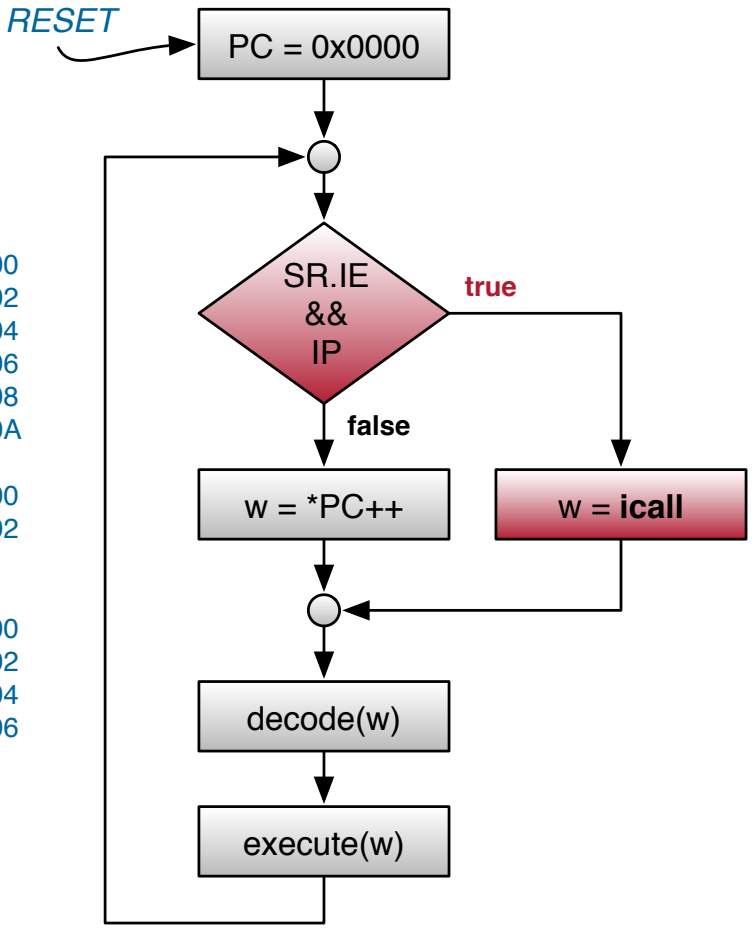
w: icall
SR` = SR
SR.IE = 0
IP = 0
PC` = PC
PC = isr
R1` = R1

w: iret
SR = SR`
PC = PC`
R1 = R1`

Ablauf eines Interrupts – Details



❹ Aufzurufende ISR wird ermittelt



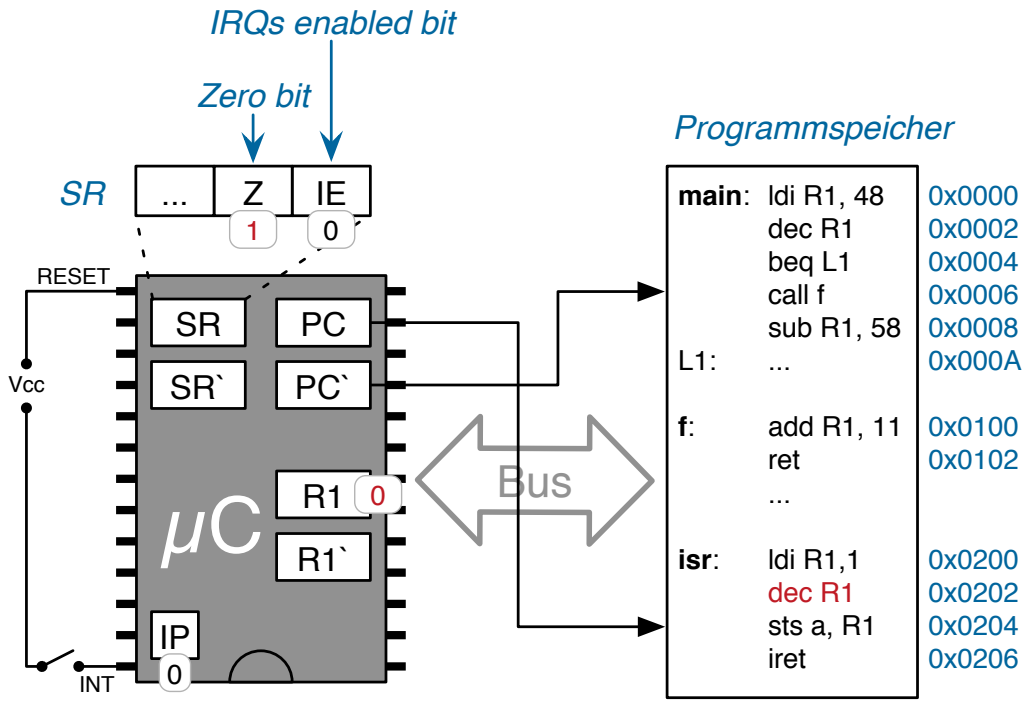
w: call <func>
PC` = PC
PC = func

w: ret
PC = PC`

w: icall
SR` = SR
SR.IE = 0
IP = 0
PC` = PC
PC = isr
R1` = R1

w: iret
SR = SR`
PC = PC`
R1 = R1`

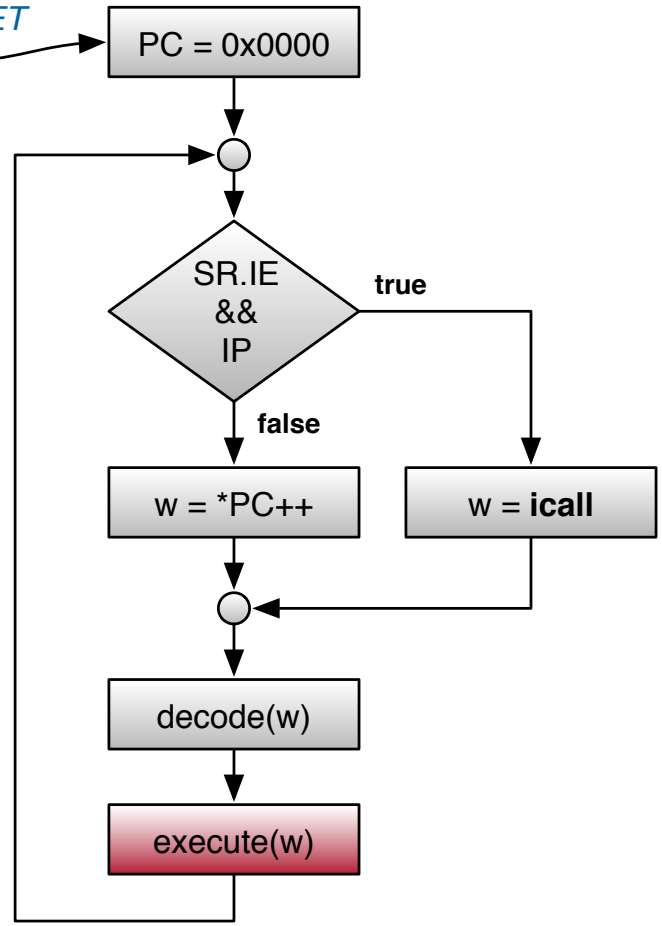
Ablauf eines Interrupts – Details



Programmspeicher

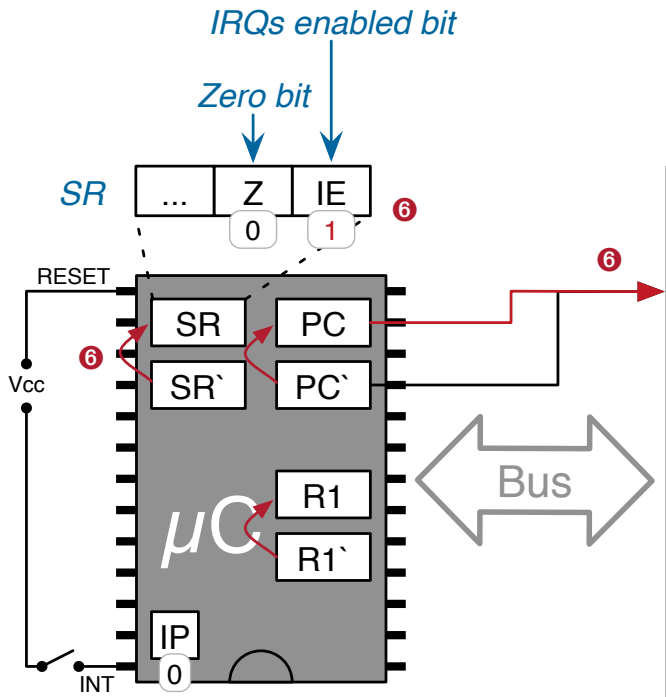
```
main:  ldi R1, 48      0x0000
       dec R1          0x0002
       beq L1          0x0004
       call f          0x0006
       sub R1, 58      0x0008
L1:    ...             0x000A

f:     add R1, 11      0x0100
       ret             0x0102
       ...

isr:   ldi R1,1        0x0200
       dec R1          0x0202
       sts a, R1       0x0204
       iret            0x0206
```

❺ ISR wird ausgeführt



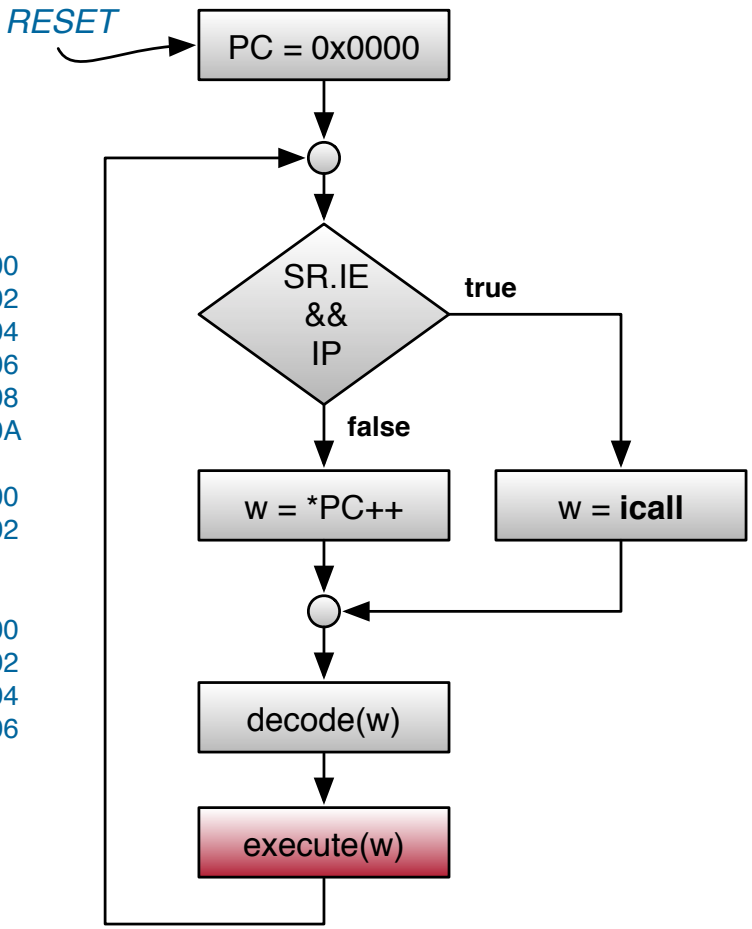
w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Ablauf eines Interrupts – Details



Programmspeicher

```
main:  ldi R1, 48      0x0000
       dec R1          0x0002
       beq L1          0x0004
       call f          0x0006
       sub R1, 58      0x0008
L1:    ...             0x000A

f:     add R1, 11      0x0100
       ret             0x0102
       ...

isr:   ldi R1,1        0x0200
       dec R1          0x0202
       sts a, R1       0x0204
       iret            0x0206
```

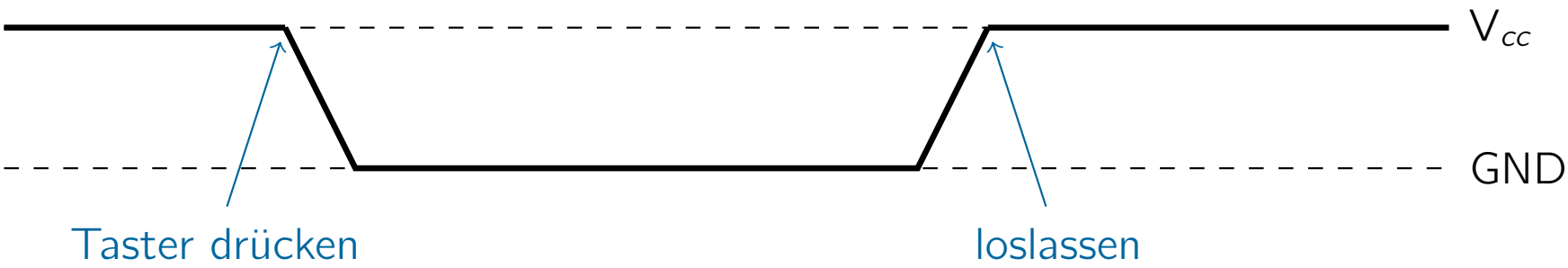


❻ ISR terminiert mit *iret*-Befehl
- Registerinhalte werden restauriert
- Zustellung von Interrupts wird reaktiviert
- Das Anwendungsprogramm wird fortgesetzt

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines idealisierten Tasters (*active low*)



- Flankengesteuerter Interrupt
  - Interrupt wird durch den Pegelwechsel (Flanke) ausgelöst
  - Häufig ist konfigurierbar, welche Flanke (steigend/fallend/beide) einen Interrupt auslösen soll
- Pegelgesteuerter Interrupt
  - Interrupt wird immer wieder ausgelöst, so lange der Pegel anliegt

Interruptsteuerung beim AVR ATmega

- IRQ-Quellen beim ATmega32 (IRQ $\mapsto$ *Interrupt ReQuest*)
  - 21 IRQ-Quellen
  - einzeln de-/aktivierbar
  - IRQ $\rightsquigarrow$ Sprung an Vektor-Adresse

[1, S. 45]

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

- Verschaltung SPiCboard
  ($\hookrightarrow$ 14–14 $\hookrightarrow$ 2–4)
  - `INT0` $\mapsto$ `PD2` $\mapsto$ Button0 (hardwareseitig entprellt)
  - `INT1` $\mapsto$ `PD3` $\mapsto$ Button1

Externe Interrupts: Register

Steuerregister für `INT0` und `INT1`

- `GICR` **General Interrupt Control Register:** Legt fest, ob die Quellen `INT`*i* IRQs auslösen (Bit `INT`*i*=1) oder deaktiviert sind (Bit `INT`*i*=0) [1, S. 71]

7	6	5	4	3	2	1	0
INT1	**INT0**	**INT2**	**–**	**–**	**–**	**IVSEL**	**IVCE**
R/W	R/W	R/W	R	R	R	R/W	R/W

- `MCUCR` **MCU Control Register:** Legt für externe Interrupts `INT0` und `INT1` fest, wodurch ein IRQ ausgelöst wird (Flanken-/Pegelsteuerung) [1, S. 69]

7	6	5	4	3	2	1	0
SE	**SM2**	**SM1**	**SM0**	**ISC11**	**ISC10**	**ISC01**	**ISC00**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Jeweils zwei *Interrupt-Sense-Control*-Bits (`ISC`*i*`0` und `ISC`*i*`1`) steuern dabei die Auslöser (Tabelle für `INT1`, für `INT0` gilt entsprechendes):

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Externe Interrupts: Verwendung

- **Schritt 1**: Installation der Interrupt-Service-Routine
  - ISR in Hochsprache $\rightsquigarrow$ Registerinhalte sichern und wiederherstellen
  - Unterstützung durch die `avrlibc`: Makro `ISR( SOURCE_vect )` (Modul `avr/interrupt.h`)

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

ISR( INT1_vect ) {  // invoked for every INT1 IRQ
  static uint8_t counter = 0;
  sb_7seg_showNumber( counter++ );
  if( counter == 100 ) counter = 0;
}

void main() {
  ...               // setup
}
```


Externe Interrupts: Verwendung (Forts.)

- **Schritt 2**: Konfigurieren der Interrupt-Steuerung
  - Steuerregister dem Wunsch entsprechend initialisieren
  - Unterstützung durch die `avrlibc`: Makros für Bit-Indizes (Modul `avr/interrupt.h` und `avr/io.h`)

```
...
void main() {
  DDRD  &= ~(1<<PD3);                  // PD3: input with pull-up
  PORTD |= (1<<PD3);
  MCUCR  &= ~(1<<ISC10 | 1<<ISC11);    // INT1: IRQ on level=low
  GICR  |= (1<<INT1);                  // INT1: enable
  ...
  sei();                               // global IRQ enable
  ...
}
```

- **Schritt 3**: Interrupts global zulassen
  - Nach Abschluss der Geräteinitialisierung
  - Unterstützung durch die `avrlibc`: Befehl `sei()` (Modul `avr/interrupt.h`)

Externe Interrupts: Verwendung (Forts.)

- **Schritt 4**: Wenn nichts zu tun, den Stromsparmodus betreten
  - Die `sleep`-Instruktion hält die CPU an, bis ein IRQ eintrifft
    - In diesem Zustand wird nur sehr wenig Strom verbraucht
  - Unterstützung durch die `avrlibc` (Modul `avr/sleep.h`):
    - `sleep_enable()` / `sleep_disable()`: Sleep-Modus erlauben / verbieten
    - `sleep_cpu()`: Sleep-Modus betreten


```
#include <avr/sleep.h>
...
void main() {
  ...
  sei();                                  // global IRQ enable
  while(1) {
    sleep_enable();
    sleep_cpu();                          // wait for IRQ
    sleep_disable();
  }
}
```

Atmel empfiehlt die Verwendung von `sleep_enable()` und `sleep_disable()` in dieser Form, um das Risiko eines „versehentliches“ Betreten des Sleep-Zustands (z. B. durch Programmierfehler oder Bit-Kipper in der Hardware) zu minimieren.

Nebenläufigkeit

Definition: Nebenläufigkeit

Zwei Programmausführungen A und B sind nebenläufig $(A|B)$, wenn für einzelne Instruktionen a aus A und b aus B nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird (a, b oder b, a).

- Nebenläufigkeit tritt auf durch
  - Interrupts
    $\leadsto$ IRQs können ein Programm an „beliebiger Stelle“ unterbrechen
  - Echt-parallele Abläufe (durch die Hardware)
    $\leadsto$ andere CPU / Peripherie greift „jederzeit“ auf den Speicher zu
  - Quasi-parallele Abläufe (z. B. Fäden in einem Betriebssystem)
    $\leadsto$ Betriebssystem kann „jederzeit“ den Prozessor entziehen
- **Problem:** Nebenläufige Zugriffe auf **gemeinsamen** Zustand

Nebenläufigkeitsprobleme

- Szenario
  - Eine Lichtschranke am Parkhauseingang soll Fahrzeuge zählen
  - Alle 60 Sekunden wird der Wert an den Sicherheitsdienst übermittelt

```
static volatile uint16_t cars;

void main() {
  while(1) {
    waitsec( 60 );
    send( cars );
    cars = 0;
  }
}
```

```
// photo sensor is connected
// to INT2

ISR(INT2_vect){
  cars++;
}
```

- Wo ist hier das Problem?
  - Sowohl `main()` als auch `ISR` **lesen und schreiben** `cars`
    ⇝ Potentielle *Lost-Update*-Anomalie
  - Größe der Variable `cars` **übersteigt die Registerbreite**
    ⇝ Potentielle *Read-Write*-Anomalie

Nebenläufigkeitsprobleme (Forts.)

- Wo sind hier die Probleme?
  - Lost-Update: Sowohl `main()` als auch `ISR` lesen und schreiben `cars`
  - Read-Write: Größe der Variable `cars` übersteigt die Registerbreite
- Wird oft erst auf der Assemblerebene deutlich

```
void main() {
  ...
    send( cars );
    cars = 0;
  ...
}
```

```
// photosensor is connected
// to INT2

ISR(INT2_vect){
  cars++;
}
```

```
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...
```

```
INT2_vect:
  ...                ; save regs
  lds r24,cars       ; load cars.lo
  lds r25,cars+1     ; load cars.hi
  adiw r24,1         ; add (16 bit)
  sts cars+1,r25     ; store cars.hi
  sts cars,r24       ; store cars.lo
  ...                ; restore regs
```


Nebenläufigkeitsprobleme: *Lost-Update*-Anomalie

```
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...
```

```
INT2_vect:
  ...                 ; save regs
  lds r24,cars
  lds r25,cars+1
  adiw r24,1
  sts cars+1,r25
  sts cars,r24
  ...                 ; restore regs
```

- Sei `cars`=5 und an **dieser Stelle** tritt der IRQ (↯) auf
  - `main` hat den Wert von `cars` (5) bereits in Register gelesen (Register $\mapsto$ lokale Variable)
  - `INT2_vect` wird ausgeführt
    - Register werden gerettet
    - `cars` wird inkrementiert $\leadsto$ `cars`=6
    - Register werden wiederhergestellt
  - `main` übergibt den **veralteten Wert** von `cars` (5) an `send`
  - `main` nullt `cars` $\leadsto$ **1 Auto ist „verloren“ gegangen**

Nebenläufigkeitsprobleme: *Read-Write*-Anomalie

```
main:
  ...
  lds r24,cars
  lds r25,cars+1
  rcall send
  sts cars+1,__zero_reg__
  sts cars,__zero_reg__
  ...
```

```
INT2_vect:
  ...                   ; save regs
  lds r24,cars
  lds r25,cars+1
  adiw r24,1
  sts cars+1,r25
  sts cars,r24
  ...                   ; restore regs
```

- Sei `cars`=255 und an **dieser Stelle** tritt der IRQ (↯) auf
  - `main` hat bereits `cars`=255 Autos mit `send` gemeldet
  - `main` hat bereits das High-Byte von `cars` genullt
    ⇝ `cars`=255, `cars.lo`=255, `cars.hi`=0
  - `INT2_vect` wird ausgeführt
    ⇝ `cars` wird gelesen und inkrementiert, **Überlauf ins High-Byte**
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1
  - `main` nullt das Low-Byte von `cars`
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1
    ⇝ Beim nächsten `send` werden **255 Autos zu viel gemeldet**

Interruptsperren: Datenflussanomalien verhindern

```
void main() {
  while(1) {
    waitsec( 60 );
    cli();
    send( cars );
    cars = 0;
    sei();
  }
}
```

kritisches Gebiet

- Wo genau ist das **kritische Gebiet**?
  - Lesen von `cars` und Nullen von `cars` müssen atomar ausgeführt werden
  - Dies kann hier mit **Interruptsperren** erreicht werden
    - `ISR` unterbricht `main`, aber nie umgekehrt $\rightsquigarrow$ asymmetrische Synchronisation
  - Achtung: Interruptsperren sollten **so kurz wie möglich** sein
    - Wie lange braucht die Funktion `send` hier?
    - Kann man `send` aus dem kritischen Gebiet herausziehen?

Nebenläufigkeitsprobleme (Forts.)

- Szenario, Teil 2 (Funktion `waitsec()`)
  - Eine Lichtschranke am Parkhauseingang soll Fahrzeuge zählen
  - Alle 60 Sekunden wird der Wert an den Sicherheitsdienst übermittelt

```
void waitsec( uint8_t sec ) {
  ...                  // setup timer
  sleep_enable();
  event = 0;
  while( !event ) { // wait for event
    sleep_cpu();    // until next irq
  }
  sleep_disable();
}
```

```
static volatile int8_t event;

// TIMER1 ISR
//   triggers when
//   waitsec() expires

ISR(TIMER1_COMPA_vect) {
  event = 1;
}
```

- Wo ist hier das Problem?
  - **Test, ob nichts zu tun ist**, gefolgt von **Schlafen, bis etwas zu tun ist**
    ⇝ Potentielle *Lost-Wakeup*-Anomalie

Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

```
void waitsec( uint8_t sec ) {
  ...                   // setup timer
  sleep_enable();
  event = 0;
  while( !event ) {
    sleep_cpu();
  }
  sleep_disable();
}
```

```
static volatile int8_t event;

// TIMER1 ISR
//   triggers when
//   waitsec() expires

ISR(TIMER1_COMPA_vect) {
  event = 1;
}
```

- Angenommen, an **dieser Stelle** tritt der Timer-IRQ (↯) auf
  - `waitsec` hat bereits festgestellt, dass `event` nicht gesetzt ist
  - `ISR` wird ausgeführt ⇝ `event` **wird gesetzt**
  - Obwohl `event` gesetzt ist, wird der Schlafzustand betreten
    ⇝ Falls kein weiterer IRQ kommt, **Dornröschenschlaf**

Lost-Wakeup: Dornröschenschlaf verhindern

```
1  void waitsec( uint8_t sec ) {
2    ...                  // setup timer
3    sleep_enable();
4    event = 0;
5    cli();
6    while( !event ) {
7      sei();                    kritisches Gebiet
8      sleep_cpu();
9      cli();
10   }
11   sei();
12   sleep_disable();
13 }
```

```
static volatile int8_t event;

// TIMER1 ISR
//   triggers when
//   waitsec() expires

ISR(TIMER1_COMPA_vect) {
  event = 1;
}
```

- Wo genau ist das **kritische Gebiet**?
  - Test auf Vorbedingung und Betreten des Schlafzustands (Kann man *das* durch Interruptsperren absichern?)
  - Problem: Vor `sleep_cpu()` müssen IRQs freigegeben werden!
  - Funktioniert dank spezieller Hardwareunterstützung:
    ⇝ Befehlssequenz `sei`, `sleep` wird von der CPU **atomar** ausgeführt

Zusammenfassung

- Interruptbearbeitung erfolgt asynchron zum Programmablauf
  - Unerwartet $\leadsto$ Zustandssicherung im Interrupt-Handler erforderlich
  - Quelle von Nebenläufigkeit $\leadsto$ Synchronisation erforderlich
- Synchronisationsmaßnahmen
  - Gemeinsame Zustandsvariablen als `volatile` deklarieren (immer)
  - Zustellung von Interrupts sperren: `cli`, `sei` (bei nichtatomaren Zugriffen, die mehr als einen Maschinenbefehl erfordern)
  - Bei längeren Sperrzeiten können IRQs verloren gehen!
- Nebenläufigkeit durch Interrupts ist eine **sehr große Fehlerquelle**
  - *Lost-Update* und *Lost-Wakeup* Probleme
  - indeterministisch $\leadsto$ durch Testen schwer zu fassen
- Wichtig zur Beherrschbarkeit: **Modularisierung** $\hookrightarrow$ 12–7
  - Interrupthandler und Zugriffsfunktionen auf gemeinsamen Zustand (`static` Variablen!) in eigenem Modul kapseln.

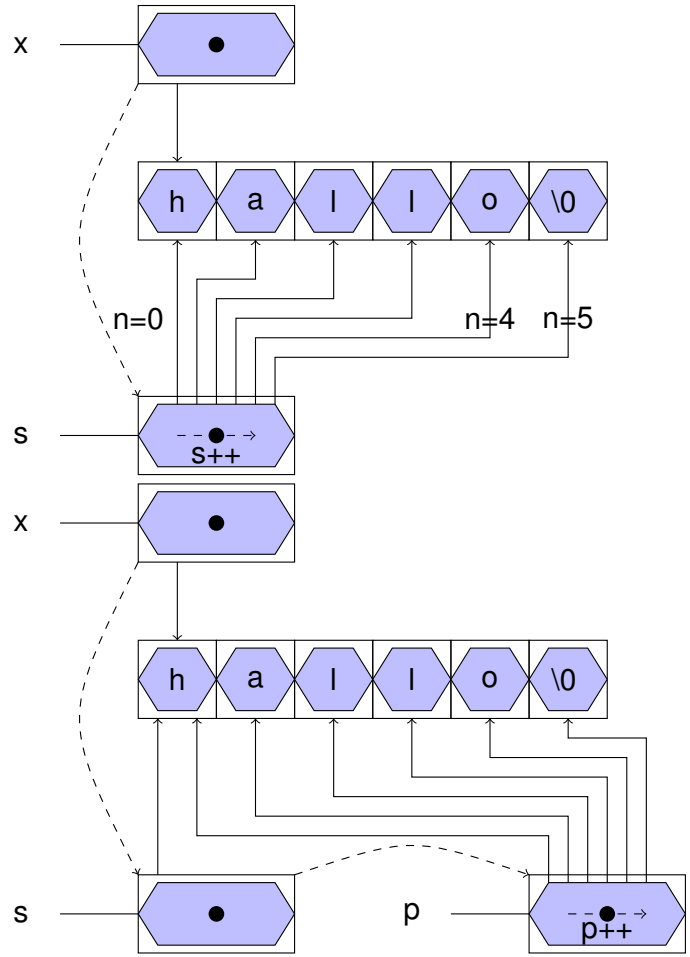
Überblick: Teil D Betriebssystemabstraktionen

Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln – Aufruf `strlen(x);`

```
/* 1. Version */
int strlen(const char *s)
{
    int n;
    for (n = 0; *s != '\0'; n++) {
        s++;
    }
    return n;
}
```

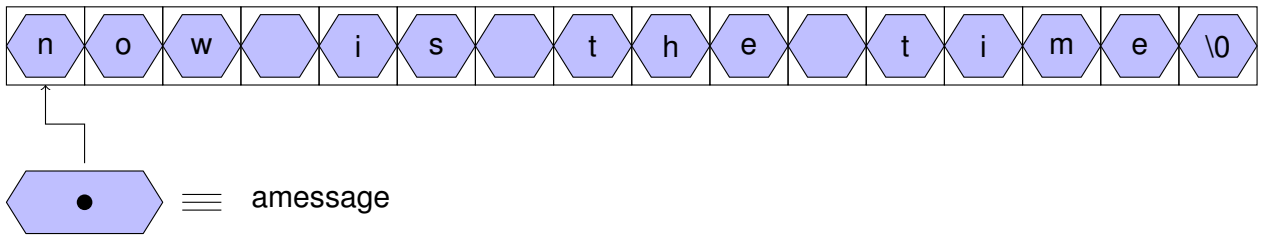
```
/* 2. Version */
int strlen(const char *s)
{
    const char *p = s;
    while (*p != '\0') {
        p++;
    }
    return p - s;
}
```



Zeiger, Felder und Zeichenketten (2)

- wird eine Zeichenkette zur Initialisierung eines `char`-Feldes verwendet, ist
  der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

```
char amessage[] = "now is the time";
```



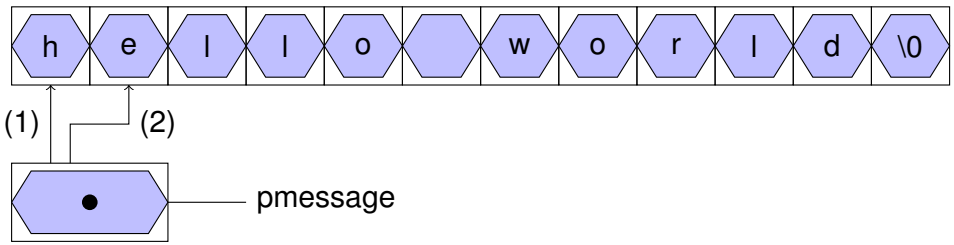
- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- `amessage` ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden

```
amessage[0] = 'h';
```


Zeiger, Felder und Zeichenketten (3)

- wird eine Zeichenkette zur Initialisierung eines `char`-Zeigers verwendet, ist der Zeiger eine
  Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
const char *pmessage = "hello world";    /*(1)*/
```

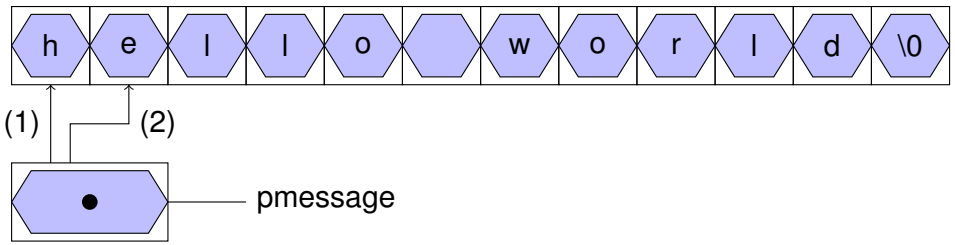



```
pmessage++;    /*(2)*/
printf("%s\n", pmessage); /* gibt "ello world" aus */
```

- die Zeichenkette selbst wird vom Compiler als konstanter Wert (String-Literal) im Speicher angelegt
- es wird ein Speicherbereich für einen Zeiger reserviert (z.B. 4 Byte) und mit der Adresse der Zeichenkette initialisiert

Zeiger, Felder und Zeichenketten (4)

```
const char *pmessage = "hello world";    /*(1)*/
```




```
pmessage++;    /*(2)*/
printf("%s\n", pmessage); /* gibt "ello world" aus */
```

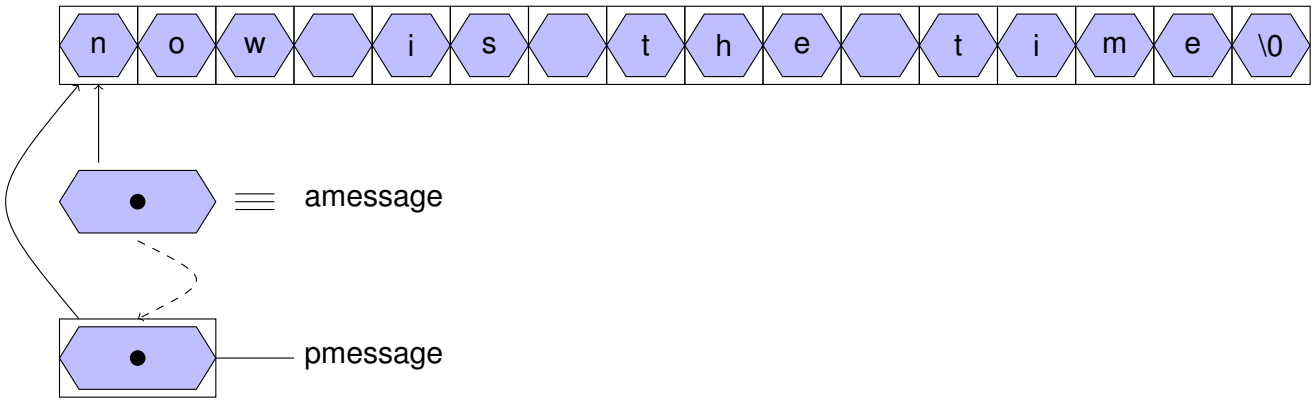
- `pmessage` ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf (`pmessage++;`)
- der Speicherbereich von `"hello world"` darf aber nicht verändert werden
  - der Compiler erkennt dies durch das Schlüsselwort `const` und verhindert schreibenden Zugriff über den Zeiger
  - manche Compiler legen solche Zeichenketten ausserdem im schreibgeschützten Speicher an (=> Speicherschutzverletzung beim Zugriff, falls der Zeiger nicht als `const`-Zeiger definiert wurde)

Zeiger, Felder und Zeichenketten (5)

- die Zuweisung eines `char`-Zeigers oder einer Zeichenkette an einen `char`-Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger `pmessage` lediglich die Adresse der Zeichenkette `"now is the time"` zu



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

Zeiger, Felder und Zeichenketten (6)

- Um eine ganze Zeichenkette einem anderen char-Feld zuzuweisen, muss sie kopiert werden: Funktion `strcpy` in der Standard-C-Bibliothek
- Implementierungsbeispiele:

```
/* 1. Version */
void strcpy(char s[], char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != '\0') {
        i++;
    }
}
```

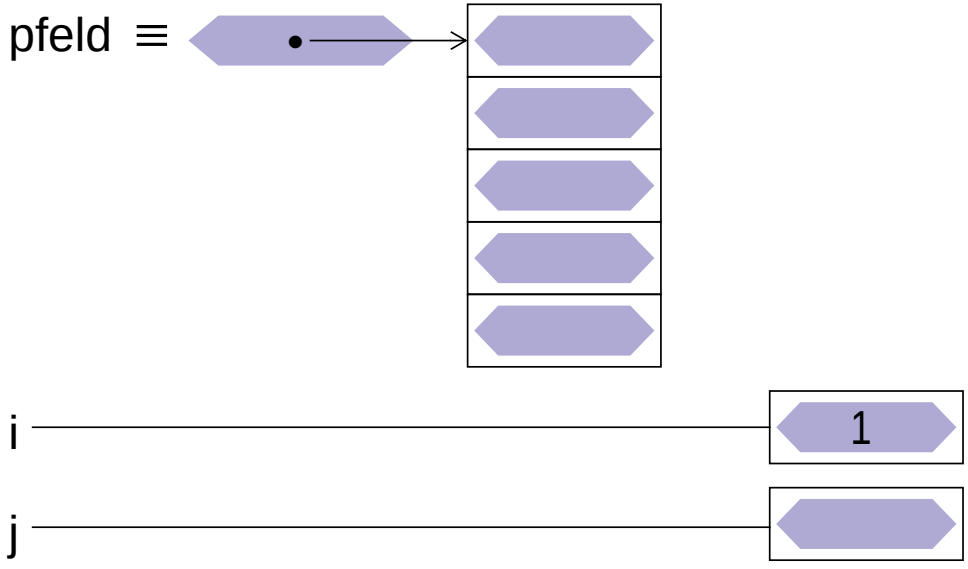
```
/* 2. Version */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++, t++;
    }
}
```

```
/* 3. Version */
void strcpy(char *s, char *t) {
    while (*s++ = *t++) {
    }
}
```

Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```



Felder von Zeigern

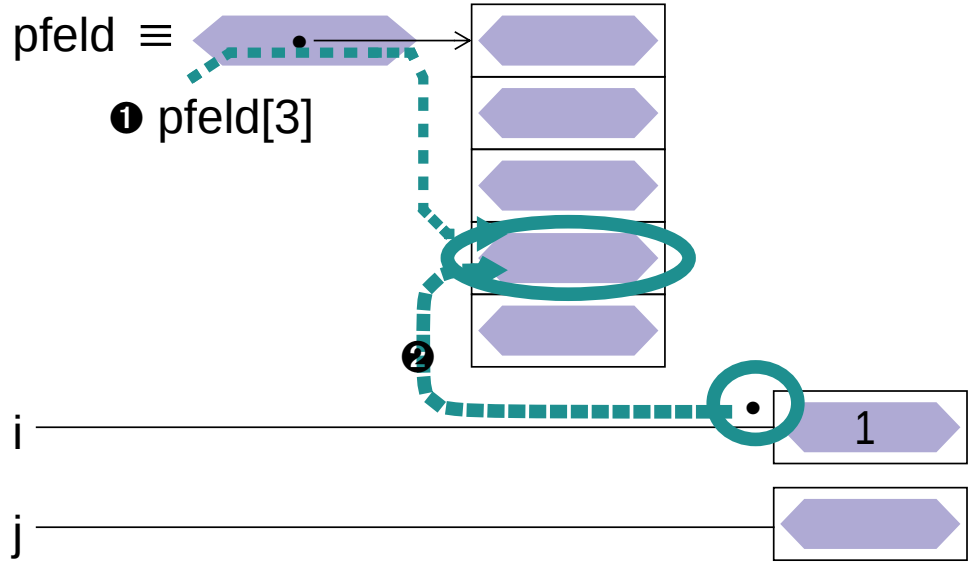
- Auch von Zeigern können Felder gebildet werden
- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```

- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

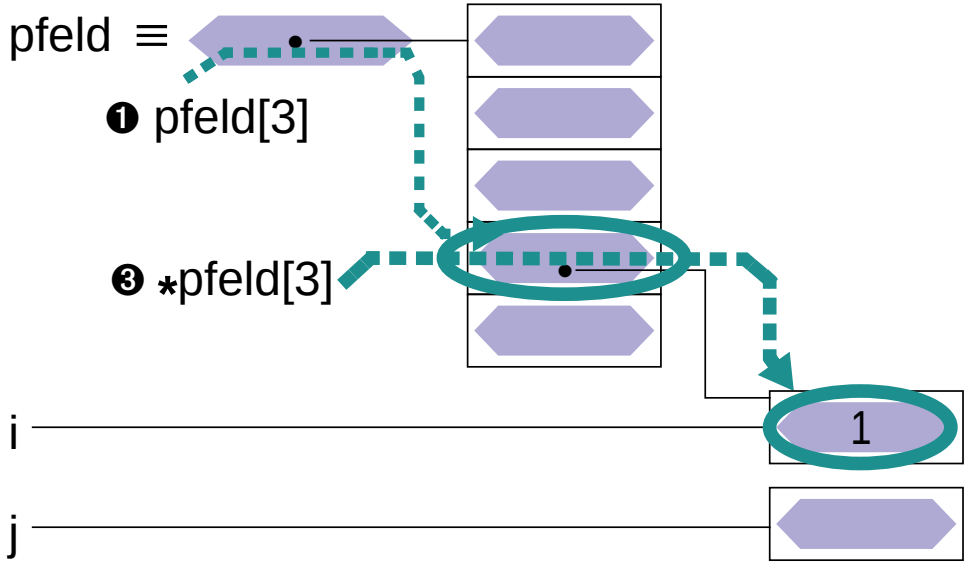
❶ ❷



Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```



- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

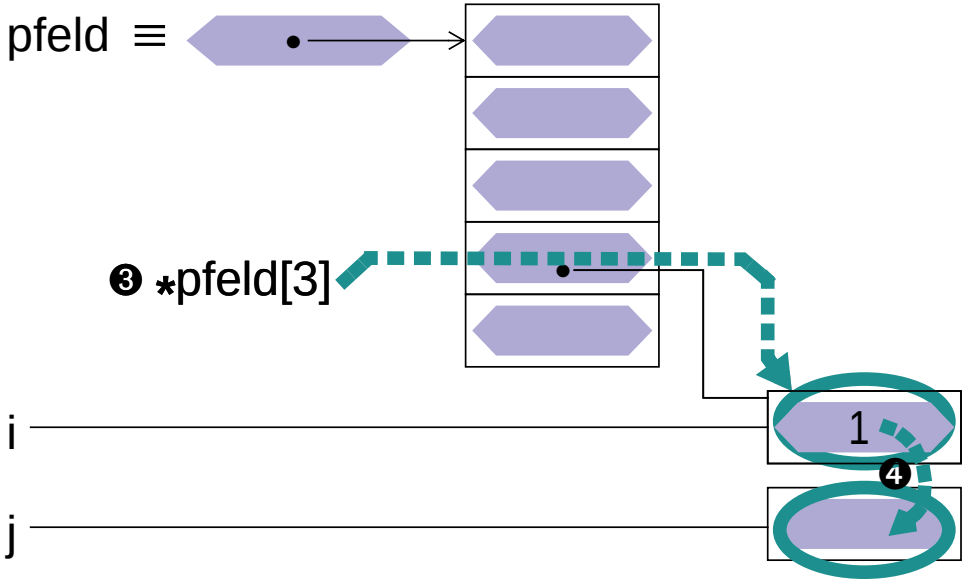
```
j = *pfeld[3];
```

❶
❸

Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```



- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

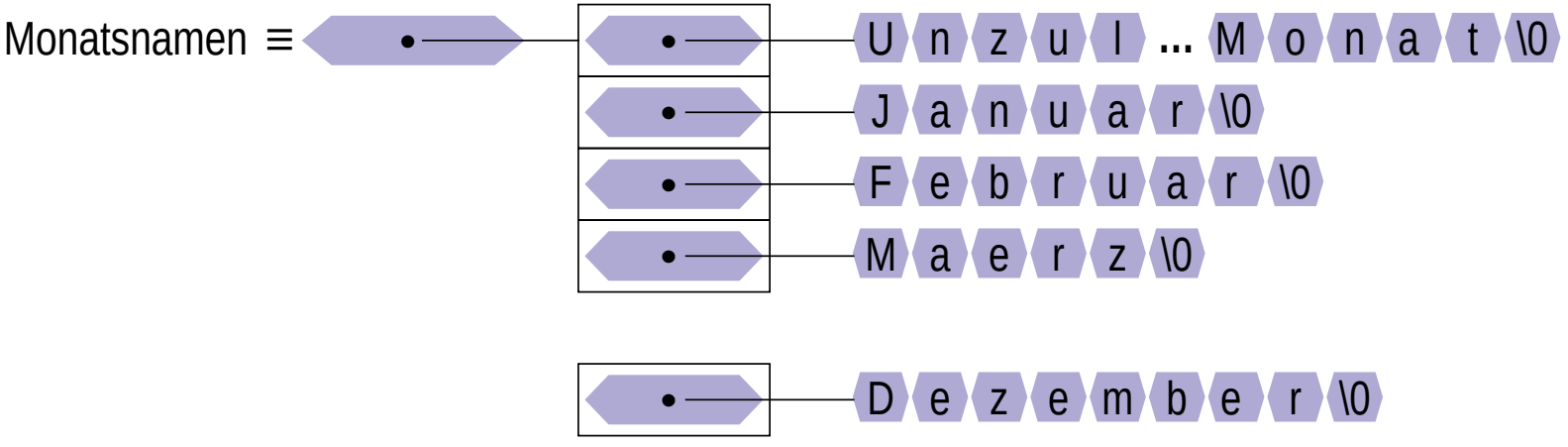
```
j = *pfeld[3];
```

❶ ❸ ❹ ❹

Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
        static char *Monatsnamen[] = {
                "Unzulaessiger Monat",
                "Januar",
        ...
                "Dezember"
        };

        return ( (n<0 || n>12) ?
                Monatsnamen[0] : Monatsnamen[n] );
}
```



Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion ***main( )*** durch zwei Aufrufparameter ermöglicht:

```
int
main (int argc, char *argv[])
{
     …
}
```

oder

```
int
main (int argc, char **argv)
{
     …
}
```

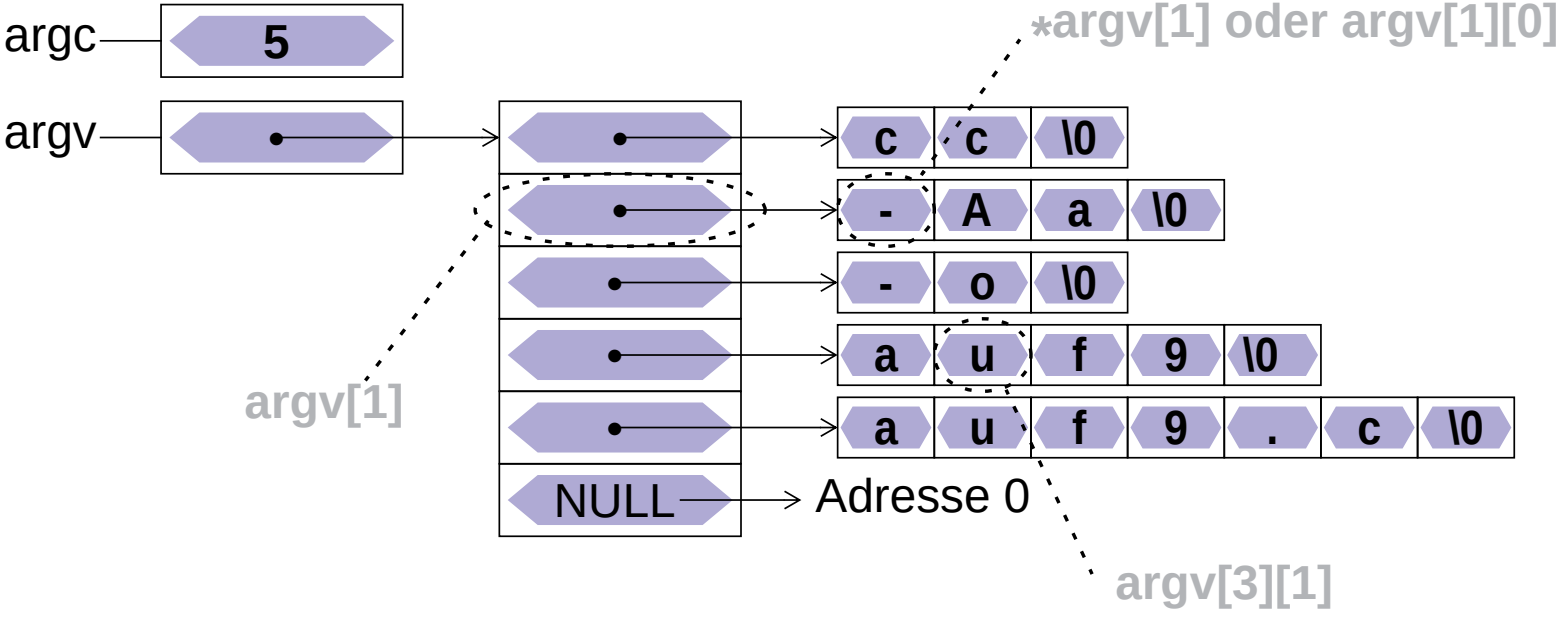
- der Parameter **`argc`** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **`argv`** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**`argv[0]`**)

Datenaufbau

Kommando: `cc -Aa -o auf9 auf9.c`

Datei cc.c:

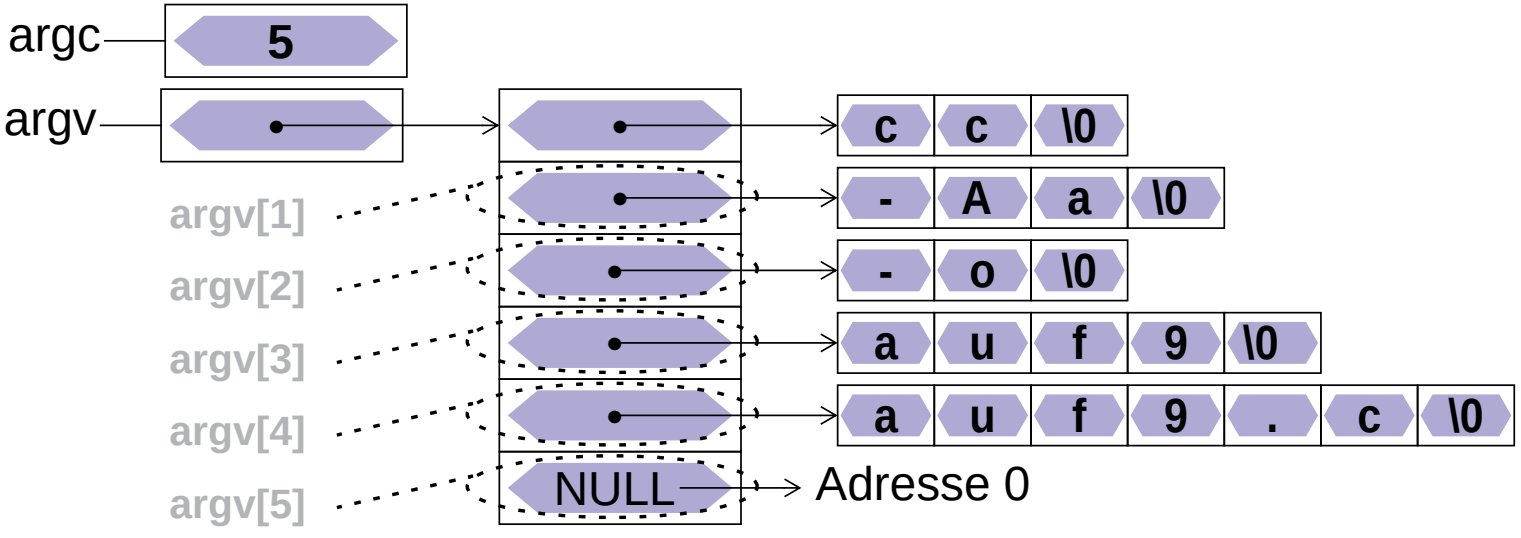
```
...
main(int argc, char *argv[]) {
...
```



Zugriff Beispiel: Ausgeben aller Argumente

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int main (int argc, char *argv[])
{   int i;
    for ( i=1; i<argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    ...
```



Verbund-Datentypen / Strukturen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit
  (ein zusammenfassender Datentyp)
- Strukturdeklaration

```
struct person {
    char name[20];
    int alter;
};
```

- Definition einer Variablen vom Typ der Struktur

```
struct person p1;
```

- Zugriff auf ein Element der Struktur

```
p1.alter = 20;
```

Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - ➤ Adresse einer Struktur mit &-Operator zu bestimmen
- Beispiele

```
struct person stud1;
struct person *pstud;
pstud = &stud1;                    /* ⇒ pstud → stud1 */
```

- Besondere Bedeutung zum Aufbau verketteter Strukturen
  - ➤ eine Strukur kann die Adresse einer weiteren Struktur desselben Typs enthalten

Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger
- Bekannte Vorgehensweise
  - `*`-Operator liefert die Struktur
  - `.`-Operator zum Zugriff auf Komponente
  - Operatorenvorrang beachten!

➥
```
(*pstud).alter = 21;
```
nicht so gut leserlich!

- Syntaktische Verschönerung

➥ `->`-Operator

```
pstud->alter = 21;
```


Rekursive Strukturen

- Strukturen in Strukturen sind erlaubt — aber
  - die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    - ➤ Problem: eine Struktur enthält sich selbst
  - die Größe eines Zeigers ist bekannt (meist 4 Byte)
    - ➤ eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

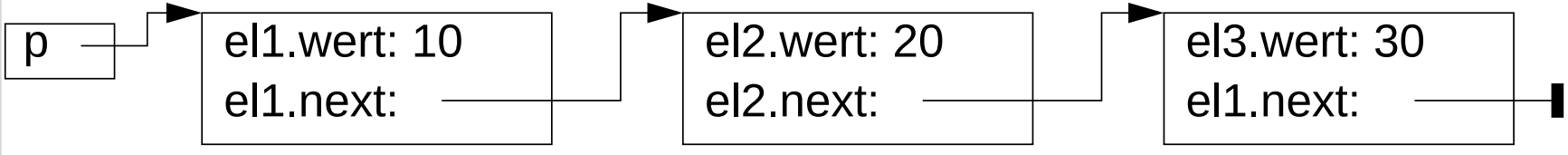
```
struct liste {
    struct student stud;
    struct liste *rest;
};
```

➥ Programmieren rekursiver Datenstrukturen
z. B. verkettete Listen

Verkettete Listen

- Mehrere Strukturen desselben Typs werden über Zeiger miteinander verkettet

```
struct liste { int wert; struct liste *next };
struct liste el1, el2, el3;
struct liste *p = &el1;
el1.wert = 10; el2.wert = 20; el3.wert = 30;
el1.next = &el2; el2.next = &el3; el3.next = NULL;
```



- Laufen über eine verkettete Liste

```
int summe = 0;
while (p != NULL) {
    summe += p-> wert;
    p = p->next;
}
```

Ein-/Ausgabe

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch "normale" Funktionen
  - Bestandteil der Standard-Funktionsbibliothek
  - einfache Programmierschnittstelle
  - effizient
  - portabel
  - betriebssystemnah
- Funktionsumfang
  - Öffnen/Schließen von Dateien
  - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
  - Formatierte Ein-/Ausgabe

Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
  - **`stdin`** Standardeingabe
    - normalerweise mit der Tastatur verbunden
    - Dateiende (**`EOF`**) wird durch Eingabe von **`CTRL-D`** am Zeilenanfang signalisiert
    - bei Programmaufruf in der Shell auf Datei umlenkbar

      ```
      prog <eingabedatei
      ```

      (bei Erreichen des Dateiendes wird **`EOF`** signalisiert)
  - **`stdout`** Standardausgabe
    - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden
    - bei Programmaufruf in der Shell auf Datei umlenkbar

      ```
      prog >ausgabedatei
      ```
  - **`stderr`** Ausgabekanal für Fehlermeldungen
    - normalerweise ebenfalls mit Bildschirm verbunden

Standard Ein-/Ausgabe (2)

- Pipes
  - die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden
    - ➤ Aufruf

      `prog1 | prog2`

! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar

- automatische Pufferung
  - Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **`NEWLINE`**-Zeichen (**`'\n'`**) an das Programm übergeben!

Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen
  - Zugriff auf Dateien
- Öffnen eines E/A-Kanals
  - Funktion fopen:

```
#include <stdio.h>
FILE *fopen(char *name, char *mode);
```

name Pfadname der zu öffnenden Datei

mode Art, wie die Datei geöffnet werden soll

"r"	zum Lesen
"w"	zum Schreiben
"a"	append: Öffnen zum Schreiben am Dateiende
"rw"	zum Lesen und Schreiben

  - Ergebnis von **fopen**:
    Zeiger auf einen Datentyp **FILE**, der einen Dateikanal beschreibt
    im Fehlerfall wird ein **NULL**-Zeiger geliefert

Öffnen und Schließen von Dateien (2)

- Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        FILE *eingabe;

        if (argv[1] == NULL) {
                fprintf(stderr, "keine Eingabedatei angegeben\n");
                exit(1);                /* Programm abbrechen */
        }

        if ((eingabe = fopen(argv[1], "r")) == NULL) {
                /* eingabe konnte nicht geoeffnet werden */
                perror(argv[1]);        /* Fehlermeldung ausgeben */
                exit(1);                /* Programm abbrechen */
        }

        ... /* Programm kann jetzt von eingabe lesen */
```

- Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

  - ➤ schließt E/A-Kanal **`fp`**

Zeichenweise Lesen und Schreiben

- Lesen eines einzelnen Zeichens
  - von der Standardeingabe

```
int getchar( )
```

  - von einem Dateikanal

```
int getc(FILE *fp )
```

    - ➤ lesen das nächste Zeichen
    - ➤ geben das gelesene Zeichen als **`int`**-Wert zurück
    - ➤ geben bei Eingabe von **`CTRL-D`** bzw. am Ende der Datei **`EOF`** als Ergebnis zurück

- Schreiben eines einzelnen Zeichens
  - auf die Standardausgabe

```
int putchar(int c)
```

  - auf einen Dateikanal

```
int putc(int c, FILE *fp )
```

    - ➤ schreiben das im Parameter **`c`** übergeben Zeichen
    - ➤ geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

Zeichenweise Lesen und Schreiben (2)

- Beispiel: copy-Programm, Aufruf: **`copy Quelldatei Zieldatei`**

Teil 1: Aufrufargumente auswerten

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        FILE *quelle, *ziel;
        int c;                  /* gerade kopiertes Zeichen */

        if (argc < 3) { /* Fehlermeldung, Abbruch */ }

        if ((quelle = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);/* Fehlermeldung ausgeben */
                exit(EXIT_FAILURE);/* Programm abbrechen */
        }

        if ((ziel = fopen(argv[2], "w")) == NULL) {
                /* Fehlermeldung, Abbruch */
        }

        while ( (c = getc(quelle)) != EOF ) {
                putc(c, ziel);
        }

        fclose(quelle);
        fclose(ziel);
}
```

Zeilenweise Lesen und Schreiben

- Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

  - liest Zeichen von Dateikanal **fp** in das Feld **s** bis entweder **n**-1 Zeichen gelesen wurden oder **'\n'** oder **EOF** gelesen wurde
  - s wird mit **'\0'** abgeschlossen (**'\n'** wird nicht entfernt)
  - gibt bei **EOF** oder Fehler **NULL** zurück, sonst s
  - für **fp** kann **stdin** eingesetzt werden, um von der Standardeingabe zu lesen

- Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

  - schreibt die Zeichen im Feld s auf Dateikanal fp
  - für **fp** kann auch **stdout** oder **stderr** eingesetzt werden
  - als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert

Formatierte Ausgabe

- Bibliotheksfunktionen — Schnittstelle

```
int printf(char *format, /* Parameter */ ... );
int fprintf(FILE *fp, char *format, /* Parameter */ ... );
int sprintf(char *s, char *format, /* Parameter */ ...);
int snprintf(char *s, int n, char *format, /* Parameter */ ...);
```

- Die statt ... angegebenen Parameter werden entsprechend der Angaben im **`format`**-String ausgegeben
  - bei **`printf`** auf der Standardausgabe
  - bei **`fprintf`** auf dem Dateikanal **`fp`**
    (für **`fp`** kann auch **`stdout`** oder **`stderr`** eingesetzt werden)
  - **`sprintf`** schreibt die Ausgabe in das **`char`**-Feld **`s`**
    (achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
  - **`snprintf`** arbeitet analog, schreibt aber maximal nur n Zeichen
    (**`n`** sollte natürlich nicht größer als die Feldgröße sein)

Formatierte Ausgabe (2)

- Zeichen im **`format`**-String können verschiedene Bedeutung haben
  - normale Zeichen: werden einfach auf die Ausgabe kopiert
  - Escape-Zeichen: z. B. **`\n`** oder **`\t`**, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
  - Format-Anweisungen: beginnen mit **`%`**-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem **`format`**-String aufbereitet werden soll

- Format-Anweisungen

  `%d`, **`%i`** **`int`** Parameter als Dezimalzahl ausgeben

  `%f` **`float`** Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben

  `%e` **`float`** Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben

  `%c` **`char`**-Parameter wird als einzelnes Zeichen ausgegeben

  `%s` **`char`**-Feld wird ausgegeben, bis **`'\0'`** erreicht ist

Formatierte Eingabe

- Bibliotheksfunktionen — Schnittstelle

```
int scanf(char *format, /* Parameter */ ...);
int fscanf(FILE *fp, char *format, /* Parameter */ ...);
int sscanf(char *s, const char *format, /* Parameter */ ...);
```

- Die Funktionen lesen Zeichen von **`stdin`** (**`scanf`**), **`fp`** (**`fscanf`**) bzw. aus dem **`char`**-Feld **`s`**.
- **`format`** gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. **`char`**-Felder bei Format **`%s`**), in die die Resultate eingetragen werden
- relativ komplexe Funktionalität, für Details siehe Manual-Seiten

Fehlerbehandlung

- Fast jeder Systemcall/Bibliotheksaufruf kann fehlschlagen
  - Fehlerbehandlung unumgänglich!
- Vorgehensweise:
  - Rückgabewerte von Systemcalls/Bibliotheksaufrufen abfragen
  - Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable **`errno`**
- Fehlermeldung kann mit der Funktion **`perror`** auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

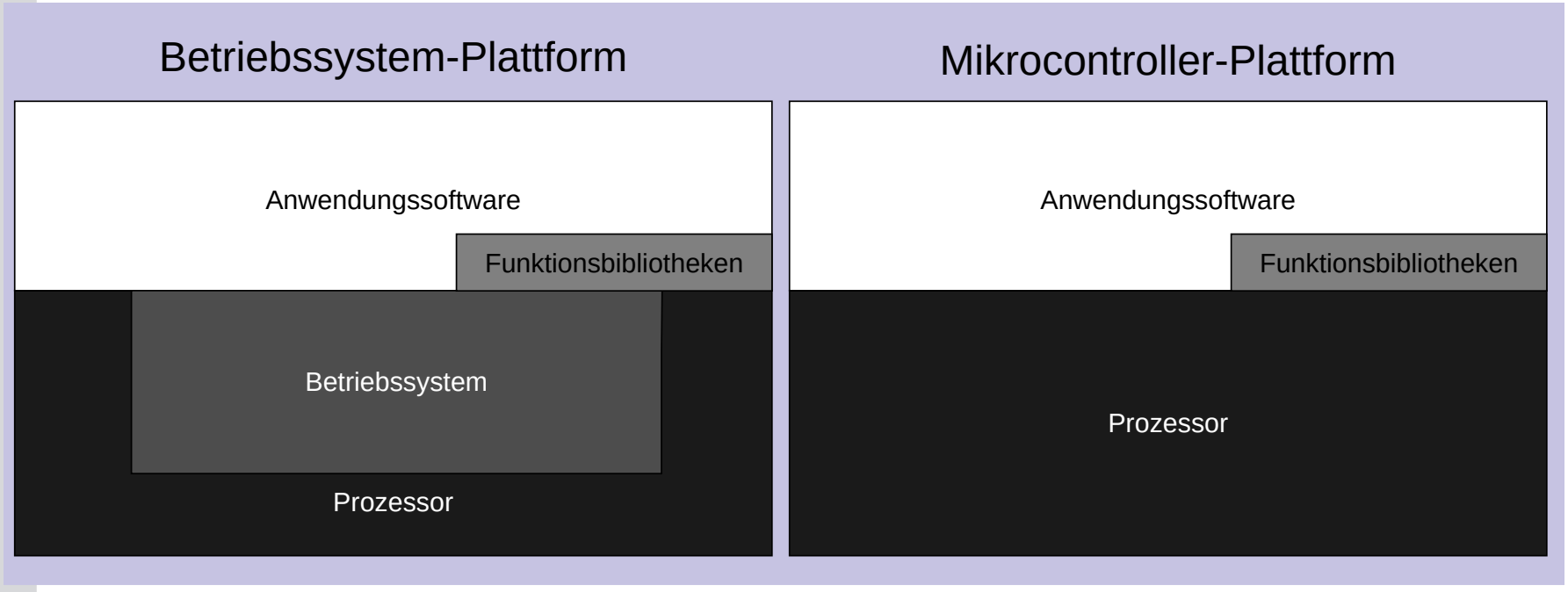
Überblick: Teil D Betriebssystemabstraktionen

Definition

- DIN 44300
  - „...die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die **Basis der möglichen Betriebsarten** des digitalen Rechensystems bilden und die insbesondere die **Abwicklung von Programmen steuern und überwachen.**“
- Andy Tanenbaum
  - „...eine Software-Schicht ..., die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle oder eine *virtuelle Maschine* anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware].“
- ★ Zusammenfassung:
  - Software zur Verwaltung und Virtualisierung der Hardwarekomponenten (Betriebsmittel)
  - Programm zur Steuerung und Überwachung anderer Programme

Betriebssystem-Plattform vs. Mikrocontroller

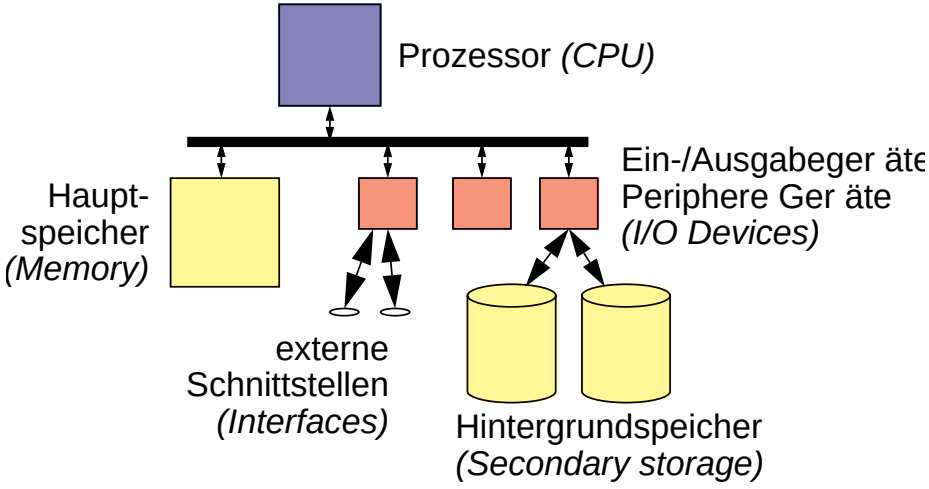
- Entscheidende Unterschiede:
  - Betriebssystem bietet zusätzliche Softwareinfrastruktur für die Ausführung von Anwendungen



  - ➤ Software-Abstraktionen (Prozesse, Dateien, Sockets, Geräte, …)
  - ➤ Schutzkonzepte
  - ➤ Verwaltungsmechanismen

Verwaltung von Betriebsmitteln

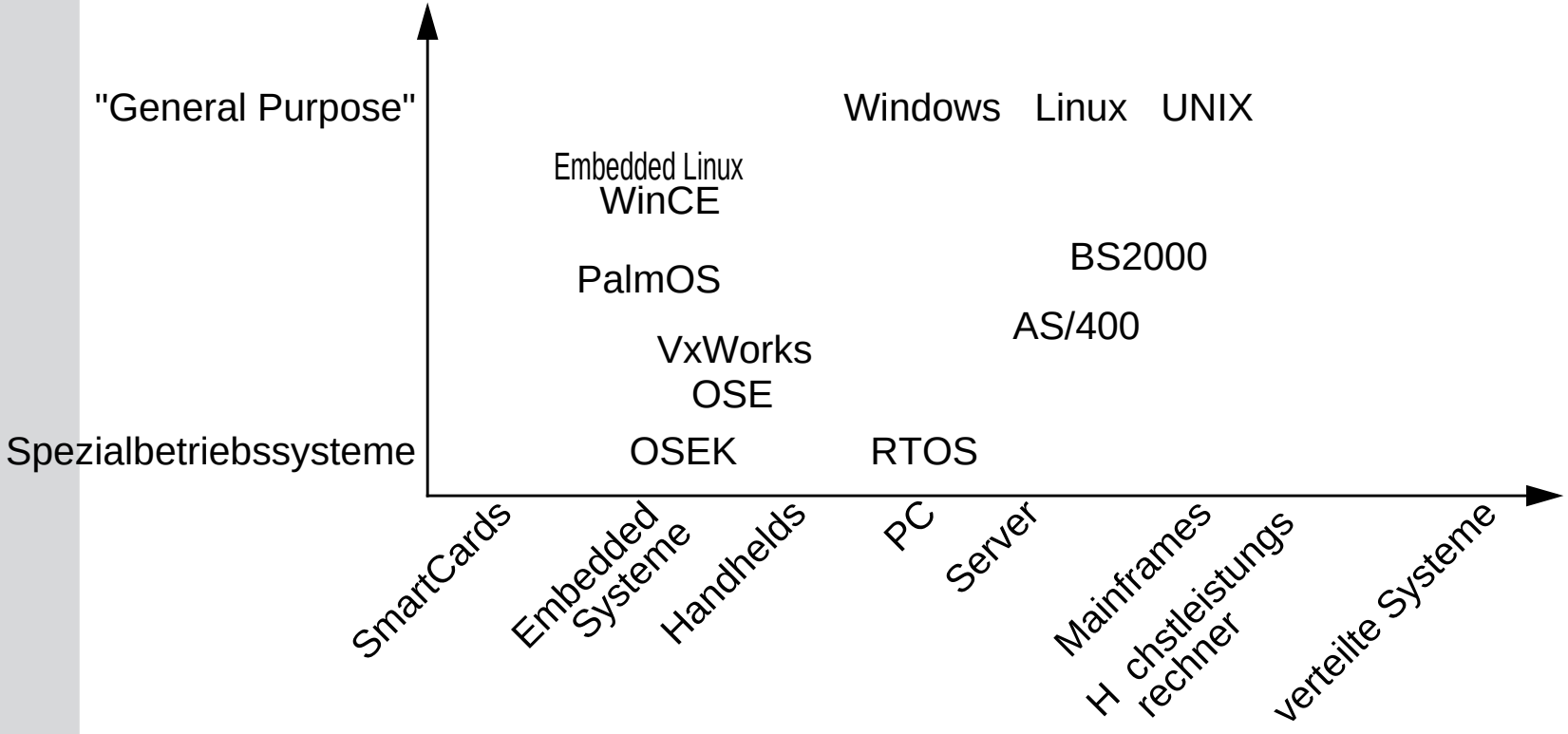
- Resultierende Aufgaben
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel



- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar in
  - aktive, zeitlich aufteilbare (Prozessor)
  - passive, nur exklusiv nutzbare (periphere Geräte, z.B. Drucker u.Ä.)
  - passive, räumlich aufteilbare (Speicher, Plattenspeicher u.Ä.)
- Unterstützung bei der Fehlererholung

Klassifikation von Betriebssystemen

- Unterschiedliche Klassfikationskriterien
  - Zielplattform
  - Einsatzzweck, Funktionalität



Klassifikation von Betriebssystemen (2)

- Wenigen "General Purpose"- und Mainframe/Höchstleistungsrechner-Betriebssystemen steht eine Vielzahl kleiner und kleinster Spezialbetriebssysteme gegenüber:

  C51, C166, C251, CMX RTOS, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, Hi Ross, Hynet-OS, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK Flex, OSEK Turbo, OSEK Plus, OSEKtime, Precise/MQX, Precise/RTCS, proOSEK, pSOS, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, ThreadX, RTA, RTX51, RTX251, RTX166, RTXC, Softune, SSXS RTOS, VRTX, VxWorks, …

➥ Einsatzbereich: Eingebettete Systeme, häufig Echtzeit-Betriebssysteme, über 50% proprietäre (in-house) Lösungen

- Alternative Klassifikation: nach Architektur

Betriebssystemarchitekturen

- Umfang zehntausende bis mehrere Millionen Befehlszeilen
  - Strukturierung hilfreich
- Verschiedene Strukturkonzepte
  - monolithische Systeme
  - geschichtete Systeme
  - Minimalkerne
  - Laufzeitbibliotheken (minimal, vor allem im Embedded-Bereich)
- Unterschiedliche Schutzkonzepte
  - ➤ kein Schutz
  - ➤ Schutz des Betriebssystems
  - ➤ Schutz von Betriebssystem und Anwendungen untereinander
  - ➤ feingranularer Schutz auch innerhalb von Anwendungen

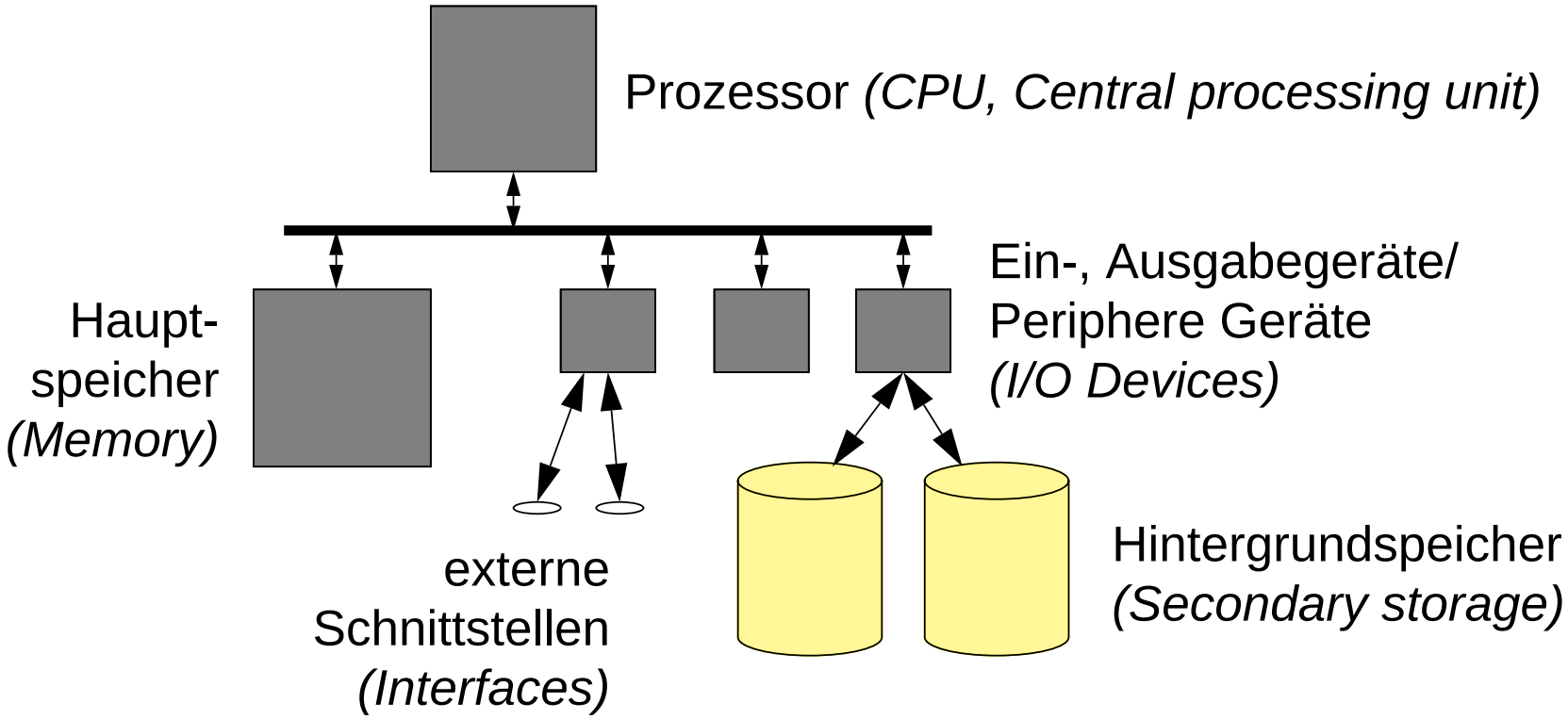
Betriebssystemkomponenten

- Speicherverwaltung
  - Wer darf wann welche Information wohin im Speicher ablegen?
- Prozessverwaltung
  - Wann darf welche Aufgabe bearbeitet werden?
- Dateisystem
  - Speicherung und Schutz von Langzeitdaten
- Interprozesskommunikation
  - Kommunikation zwischen Anwendungsausführungen bzw. Teilen einer parallel ablaufenden Anwendung
- Ein/Ausgabe
  - Kommunikation mit der "Außenwelt" (Benutzer/Rechner)

Überblick: Teil D Betriebssystemabstraktionen

Überblick

- Einordnung

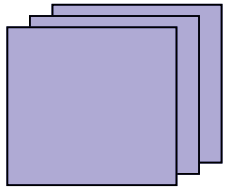


Überblick (2)

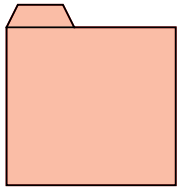
- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Betriebssystemabstraktion zur Nutzung von Hintergrundspeicher (z. B. Platten, SSD / Flash-Speicher, DVD / CD-ROM, Bandlaufwerke)
    - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
    - einheitliche Sicht auf den Hintergrundspeicher
- Wesentliche Elemente eines Dateisystems:
  - Dateien *(Files)*
  - Verzeichnissen / Katalogen *(Directories)*
  - Partitionen *(Partitions)*

Überblick (3)

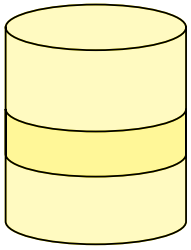
- Datei
  - speichert Daten oder Programme
- Verzeichnis / Katalog (*Directory*)
  - fasst Dateien (u. Verzeichnisse) zusammen
  - erlaubt Benennung der Dateien
  - ermöglicht Aufbau eines hierarchischen Namensraums
  - enthält Zusatzinformationen zu Dateien
- Partition
  - eine Menge von Verzeichnissen und deren Dateien
  - sie dienen zum physischen oder logischen Trennen von Dateimengen
    - ➤ *physisch:* Festplatte, Diskette
    - ➤ *logisch:* Teilbereich auf Platte oder CD



Dateien



Verzeichnis



Partition

Dateien

- Kleinste Einheit, in der etwas auf den Hintergrundspeicher geschrieben werden kann.

Dateiattribute

- *Name* — Symbolischer Name, vom Benutzer les- und interpretierbar
  - z. B. **`AUTOEXEC.BAT`**
- *Typ* — Für Dateisysteme, die verschiedene Dateitypen unterscheiden
  - z. B. sequenzielle Datei, zeichenorientierte Datei, satzorientierte Datei
- *Ortsinformation* — Wo werden die Daten physisch gespeichert?
  - Gerätenummer, Nummern der Plattenblocks

Dateiattribute (2)

- *Größe* — Länge der Datei in Größeneinheiten (z. B. Bytes, Blöcke, Sätze)
  - steht in engem Zusammenhang mit der Ortsinformation
  - wird zum Prüfen der Dateigrenzen z. B. beim Lesen benötigt
- *Zeitstempel* — z. B. Zeit und Datum der Erstellung, letzten Änderung
  - unterstützt Backup, Entwicklungswerkzeuge, Benutzerüberwachung etc.
- *Rechte* — Zugriffsrechte, z. B. Lese-, Schreibberechtigung
  - z. B. nur für den Eigentümer schreibbar, für alle anderen nur lesbar
- *Eigentümer* — Identifikation des Eigentümers
  - eventuell eng mit den Rechten verknüpft
  - Zuordnung beim Accounting (Abrechnung des Plattenplatzes)

Operationen auf Dateien

- Erzeugen *(Create)*
  - Nötiger Speicherplatz wird angefordert.
  - Verzeichniseintrag wird erstellt.
  - Initiale Attribute werden gespeichert.
- Schreiben *(Write)*
  - Identifikation der Datei
  - Daten werden auf Platte transferiert.
  - eventuell Nachfordern von Speicherplatz
  - eventuelle Anpassung der Attribute, z. B. Länge, Zugriffszeit
- Lesen *(Read)*
  - Identifikation der Datei
  - Daten werden von Platte gelesen.

Operationen auf Dateien (2)

- Positionieren des Schreib-/Lesezeigers für die nächste Schreib- oder Leseoperation *(Seek)*
  - Identifikation der Datei
  - In vielen Systemen wird dieser Zeiger implizit bei Schreib- und Leseoperationen positioniert.
  - Ermöglicht explizites Positionieren
- Verkürzen *(Truncate)*
  - Identifikation der Datei
  - Ab einer bestimmten Position wird der Inhalt entfernt (evtl. kann nur der Gesamtinhalt gelöscht werden).
  - Anpassung der betroffenen Attribute
- Löschen *(Delete)*
  - Identifikation der Datei
  - Entfernen der Datei aus dem Verzeichnis und Freigabe der Plattenblöcke

Verzeichnisse / Kataloge

- Ein Verzeichnis gruppiert Dateien und evtl. weitere Verzeichnisse
- Gruppierungsalternativen:
  - Verknüpfung mit der Benennung
    - Verzeichnis enthält Namen und Verweise auf Dateien und andere Verzeichnisse,
      z. B. *UNIX, Windows*
  - Gruppierung über Bedingung
    - Verzeichnis enthält Namen und Verweise auf Dateien, die einer bestimmten Bedingung gehorchen
      z. B. gleiche Gruppennummer in *CP/M*
      z. B. eigenschaftsorientierte und dynamische Gruppierung in *BeOS-BFS*
- Verzeichnis ermöglicht das Auffinden von Dateien
  - Vermittlung zwischen externer und interner Bezeichnung
    (Dateiname — Plattenblöcke)

Operationen auf Verzeichnissen

- Auslesen der Einträge *(Read, Read Directory)*
  - Daten des Verzeichnisinhalts werden gelesen und meist eintragsweise zurückgegeben
- Erzeugen und Löschen der Einträge erfolgt implizit mit der zugehörigen Dateioperation
- Erzeugen und Löschen von Verzeichnissen
  (Create and Delete Directory)

Attribute von Verzeichnissen

- Die meisten Dateiattribute treffen auch für Verzeichnisse zu
  - Name, Ortsinformationen, Größe, Zeitstempel, Rechte, Eigentümer

Dateisystem am Beispiel Linux/UNIX

- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
- Dateiattribute
  - das Betriebssystem verwaltet zu jeder Datei eine Reihe von Attributen (Rechte, Größe, Zugriffszeiten, Adressen der Datenblöcke, …)
  - die Attribute werden in einer speziellen Verwaltungsstrukur, dem *Dateikopf*, gespeichert
    - ➤ Linux/UNIX: *Inode*
    - ➤ Windows NTFS: *Master File Table*-Eintrag
- Namensraum
  - Flacher Namensraum: Inodes sind einfach durchnummeriert
  - Hierarchischer Namensraum: Verzeichnisstruktur bildet Datei- und Pfadnamen in einem Dateibaum auf Inode-Nummern ab

Namensräume

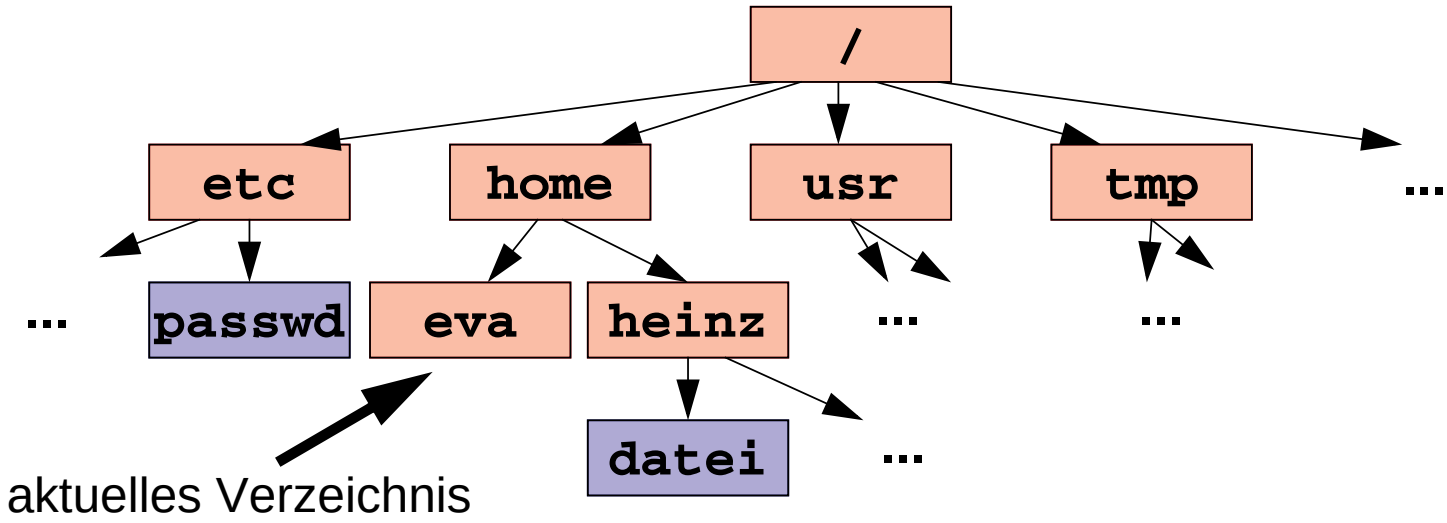
- Ein Namensraum definiert den Kontext, in dem ein Name eine Bedeutung hat ("Java" im Kontext Geografie, Programmiersprachen;
  "C" im Kontext Musik, Alphabet, Programmiersprachen)

- Flache Namensräume
  - jeder Name muss eindeutig sein
  - bei einer großen Menge von Elementen unübersichtlich und schwer zu verwalten

- Hierarchische Namensräume
  - Menge von Kontexten, die irgendwelche zu benennenden Elemente, auch Kontexte, enthalten
  - am Beispiel UNIX: baumförmig strukturierter hierarchischer Namensraum
    - Knoten (Kontexte) des Baums sind Verzeichnisse (*Directories*)
    - Blätter des Baums sind Verweise auf Dateien
  - jedem UNIX-Prozess ist zu jeder Zeit ein aktuelles Verzeichnis *(Current working directory)* zugeordnet

Pfadnamen

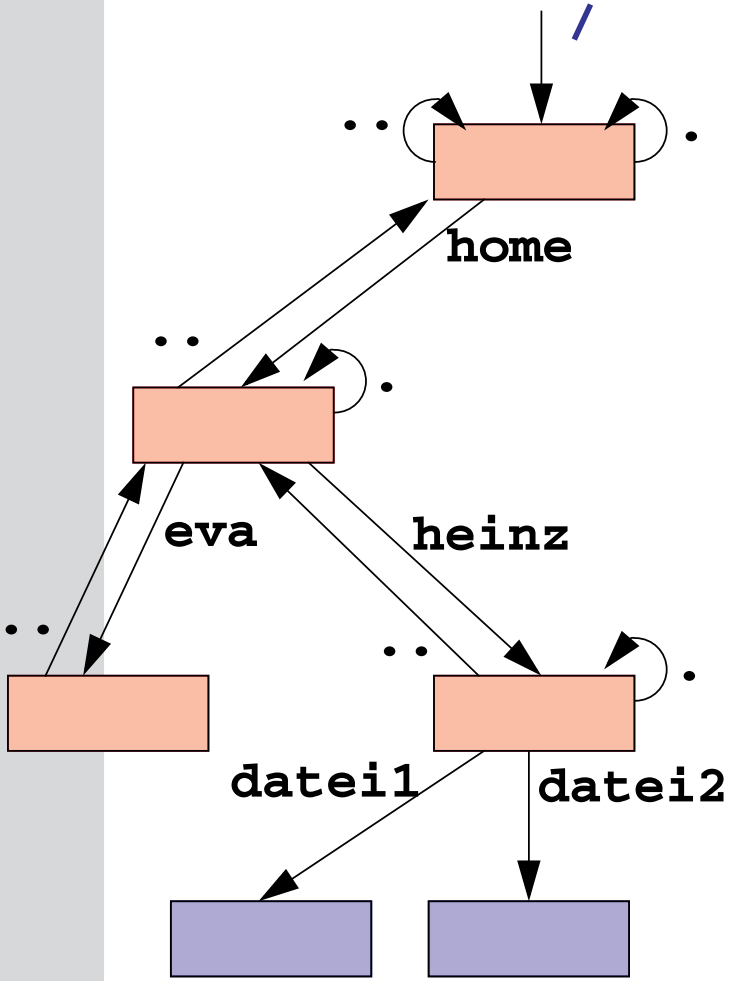
- Baumstruktur



- Pfade
  - z. B. „**`/home/heinz/datei`**“, „**`/tmp`**“, „**`../heinz/datei`**“
  - „**`/`**“ ist Trennsymbol *(Slash)*; beginnender „**`/`**“ bezeichnet Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellen Verzeichnis

Pfadnamen (2)

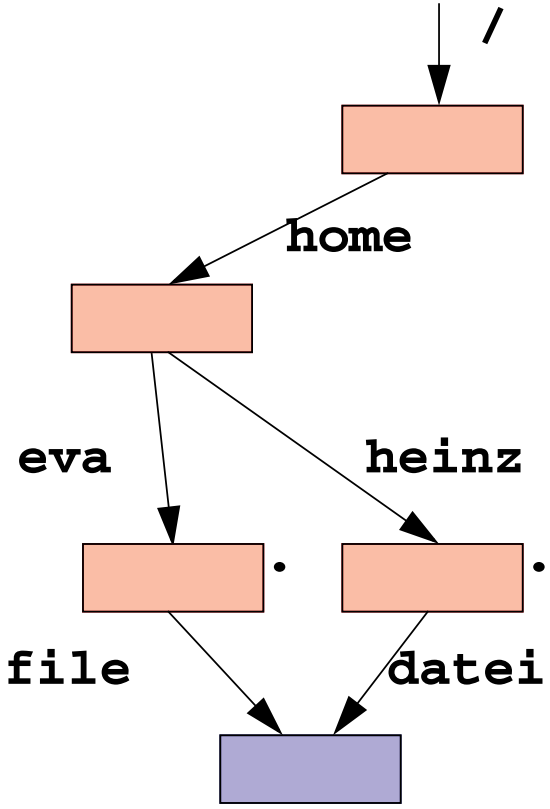
- Eigentliche Baumstruktur



- benannt sind nicht Dateien und Verzeichnisse, sondern die Verbindungen (*Links*) zwischen ihnen
  - Verzeichnisse und Dateien können auf verschiedenen Pfaden erreichbar sein
    z. B. **`../heinz/datei1`** und **`/home/heinz/datei1`**
  - Jedes Verzeichnis enthält
    - einen Verweis auf sich selbst (**`.`**) und
    - einen Verweis auf das darüberliegende Verzeichnis im Baum (**`..`**)
    - Verweise auf Dateien

Pfadnamen (3)

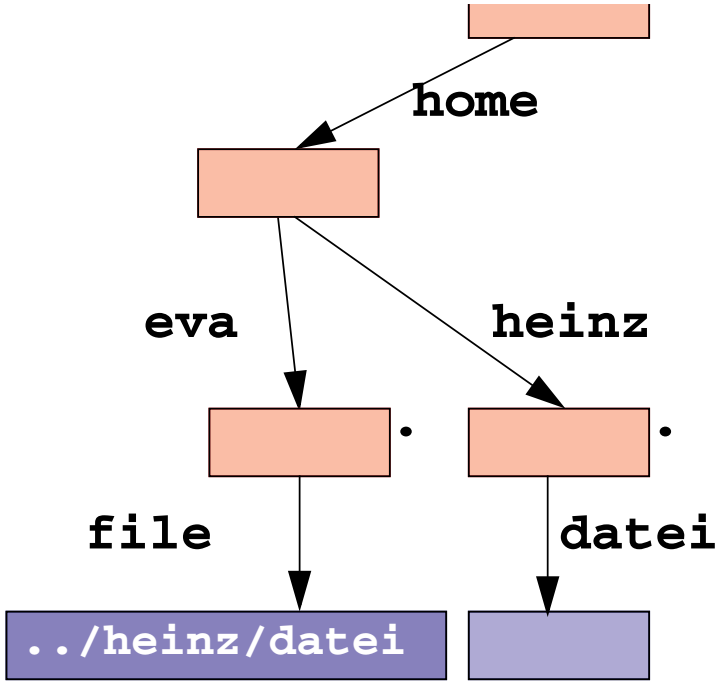
- Links *(Hard Links)*
  - Dateien können mehrere auf sich zeigende Verweise besitzen, sogenannte Hard-Links (nicht jedoch Verzeichnisse)



  - Die Datei hat zwei Einträge in verschiedenen Verzeichnissen, die völlig gleichwertig sind:
    `/home/eva/file`
    `/home/heinz/datei`
  - Datei wird erst gelöscht, wenn letzter Link gekappt wird.

Pfadnamen (4)

- Symbolische Namen *(Symbolic Links)*
  - Verweise auf einen anderen Pfadnamen
    (sowohl auf Dateien als auch Verzeichnisse)
  - Symbolischer Name bleibt auch bestehen, wenn Datei oder Verzeichnis nicht mehr existiert



  - Symbolischer Name enthält einen neuen Pfadnamen, der vom FS interpretiert wird.

Eigentümer und Rechte

- Eigentümer
  - Jeder Benutzer wird durch eindeutige Nummer (UID) repräsentiert
  - Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die durch eine eindeutige Nummer (GID) repräsentiert werden
  - Eine Datei oder ein Verzeichnis ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
  - Lesen, Schreiben, Ausführen (nur vom Eigentümer veränderbar)
  - einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Verzeichnissen
  - Lesen, Schreiben (Löschen u. Anlegen von Dateien etc.), Durchgangsrecht
  - Schreibrecht ist einschränkbar auf eigene Dateien („nur erweiterbarer“)

Inodes

- Attribute (Zugriffsrechte, Eigentümer, etc.) einer Datei und Ortsinformation über ihren Inhalt werden in **Inodes** gehalten
  - Inodes werden pro Partition nummeriert *(Inode number)*
- Inhalt eines Inode: Dateiattribute
  - Dateityp: Verzeichnis, normale Datei, Spezialdatei (z. B. Gerät)
  - Eigentümer und Gruppe
  - Zugriffsrechte
  - Zugriffszeiten: letzte Änderung *(mtime)*, letzer Zugriff *(atime)*, letzte Änderung des Inodes *(ctime)*
  - Anzahl der Hard links auf den Inode
  - Dateigröße (in Bytes)
  - Gerät (Disk, Partition), auf dem der Inode angelegt ist
  - Adressen der Datenblöcke des Dateiinhalts
  - bei Spezialdateien: Typ und laufende Nummer des zugeordneten Geräts

Inodes — Programmierschnittstelle: stat / lstat

- liefert Dateiattribute aus dem Inode
- Funktionsschnittstelle:

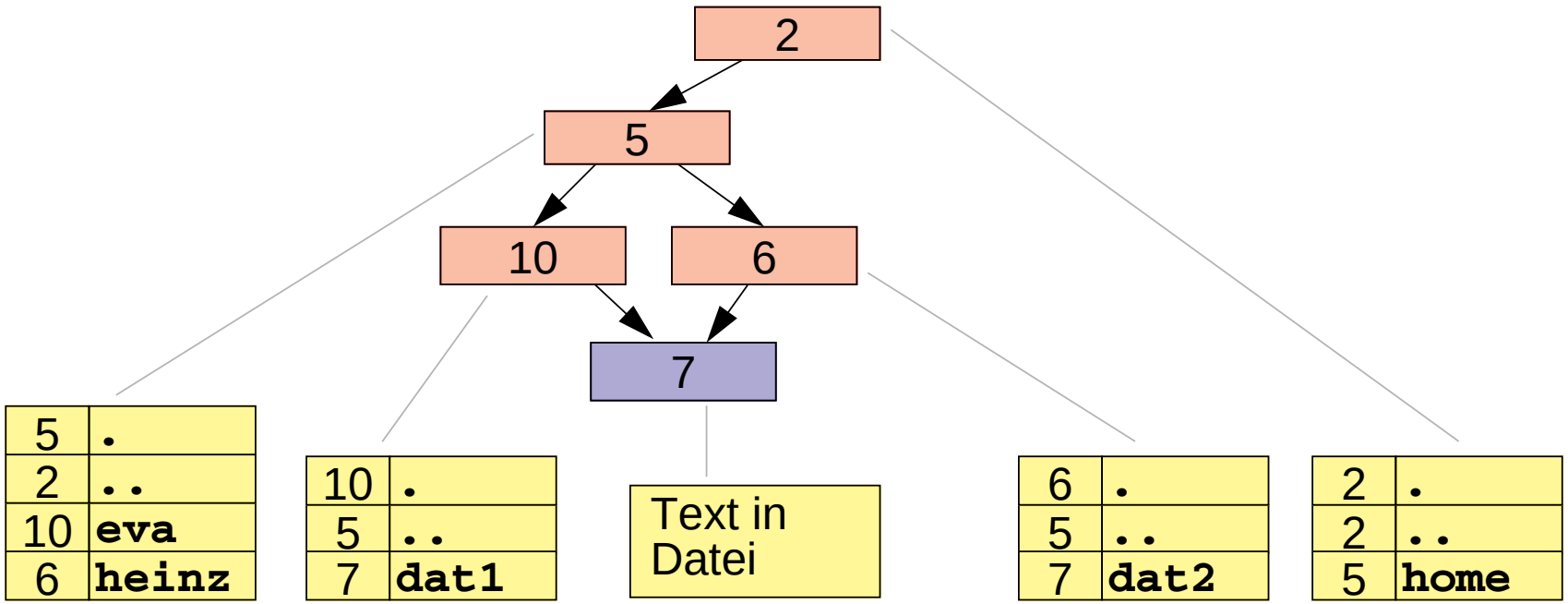
```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - **`path`**: Dateiname
  - **`buf`**: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```


Verzeichnisse, Pfadnamen und Inodes

- Verzeichnisse enthalten lediglich Paare von Namen und Inode-Nummern
  - Verzeichnisse bilden einen hierarchischen Namensraum über einem eigentlich flachen Namensraum (durchnumerierte Dateien)
  - Die Verzeichnisstrukturen mit ihren Funktionen agieren damit als *Nameserver*, der Pfadnamen in Inode-Nummern abbildet



  - Dateiattribute stehen im Inode, nicht im Verzeichnis! (oft benötigte Attribute, z. B. Dateigröße, können aber zusätzlich im Verzeichnis gehalten werden)

Programmierschnittstelle für Verzeichnisse

- Verzeichnisse verwalten
  - Erzeugen

```
int mkdir( const char *path, mode_t mode );
```

  - Löschen

```
int rmdir( const char *path )
```

  - Hard Link erzeugen

```
int link( const char *existing, const char *new );
```

  - Symbolischen Namen erzeugen

```
int symlink( const char *path, const char *new );
```

  - Verweis/Datei löschen

```
int unlink( const char *path );
```

- Symbolische Namen auslesen

```
int readlink( const char *path, void *buf, size_t bufsiz );
```


Programmierschnittstelle für Verzeichnisse (2)

- Verzeichnisse lesen (Schnittstelle der C-Bibliothek)
  - Verzeichnis öffnen:
    ```
    DIR *opendir( const char *path );
    ```
  - Verzeichniseinträge lesen:
    ```
    struct dirent *readdir( DIR *dirp );
    ```
  - Verzeichnis schließen:
    ```
    int closedir( DIR *dirp );
    ```

Verzeichnisse (2): opendir / closedir

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

- Argument von opendir
  - **`dirname`**: Verzeichnisname
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`DIR`** oder **`NULL`**

Verzeichnisse (3): readdir

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argumente
  - **`dirp`**: Zeiger auf **`DIR`**-Datenstruktur
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **`struct dirent`** oder **`NULL`** wenn fertig oder Fehler (**`errno`** vorher auf 0 setzen!)
- Probleme: Der Speicher für **`struct dirent`** wird von der Funktion readdir beim nächsten Aufruf wieder verwendet!
  - ➤ wenn Daten aus der Struktur (z. B. der Dateiname) länger benötigt werden, reicht es nicht, sich den zurückgegebenen Zeiger zu merken sondern es müssen die benötigten Daten kopiert werden

Verzeichnisse (4): struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

```
struct dirent {
    __ino_t d_ino;
    __off_t d_off;
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256];
};
```

Programmierschnittstelle für Dateien

- siehe C-Ein/Ausgabe (Schnittstelle der C-Bibliothek)
- C-Funktionen (fopen, printf, scanf, getchar, fputs, fclose, ...) verbergen die "eigentliche" Systemschnittstelle und bieten mehr "Komfort"
  - ➤ Systemschnittstelle: open, close, read, write

Spezialdateien

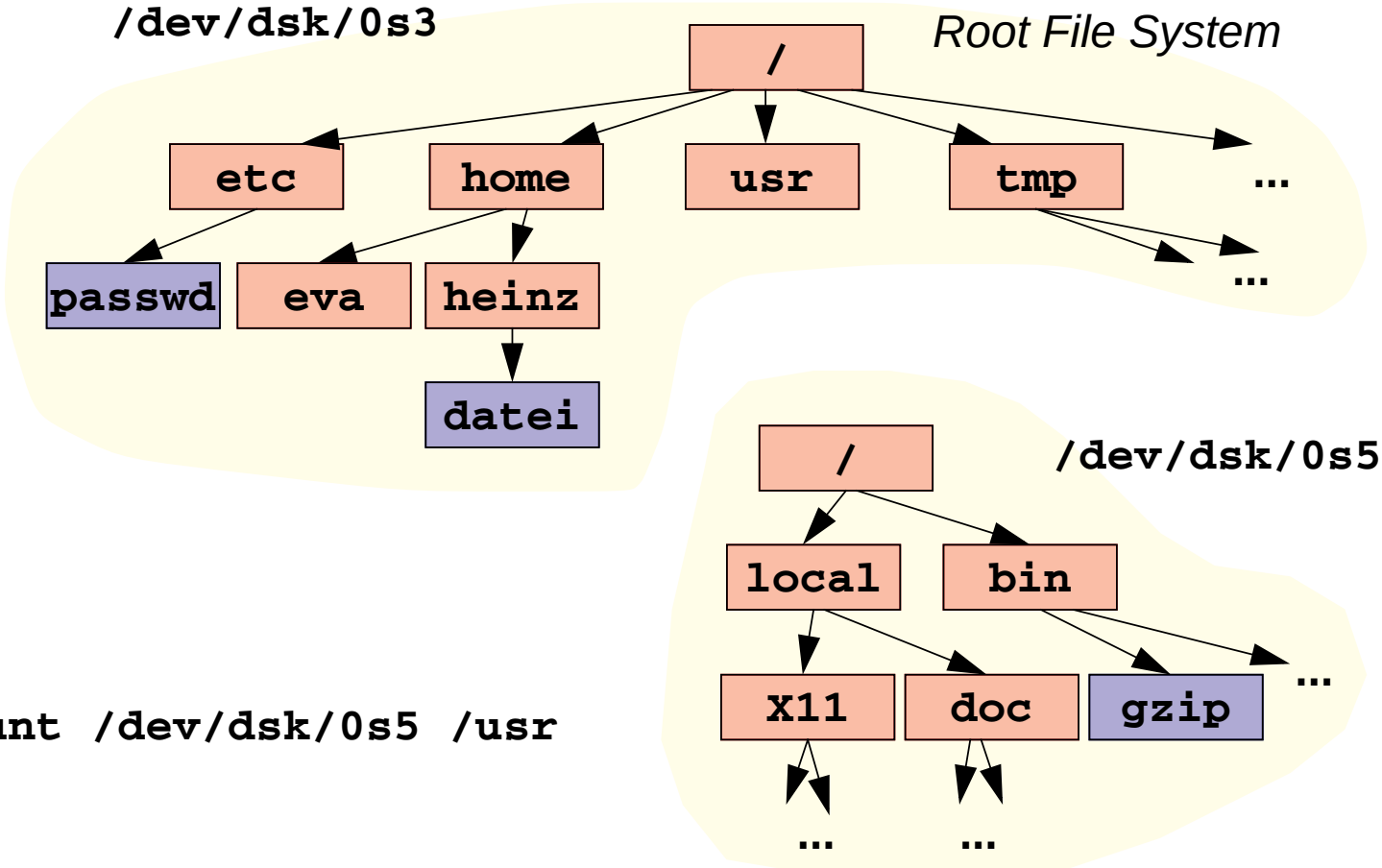
- Periphere Geräte werden als Spezialdateien repräsentiert
  - Geräte können wie Dateien mit Lese- und Schreiboperationen angesprochen werden
  - Öffnen der Spezialdateien schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch einen Treiber hergestellt wird
- Blockorientierte Spezialdateien
  - Plattenlaufwerke, Bandlaufwerke, Floppy Disks, CD-ROMs
- Zeichenorientierte Spezialdateien
  - Serielle Schnittstellen, Drucker, Audiokanäle etc.
  - blockorientierte Geräte haben meist auch eine zusätzliche zeichenorientierte Repräsentation

Partitionen

- jede Partition enthält einen eigenen Dateibaum
  (= "ein eigenes Dateisystem")
  - wird durch blockorientierte Spezialdatei repräsentiert
    (z. B. **`/dev/dsk/0s3`**)
- Bäume der Partitionen können zu einem homogenen Dateibaum zusammengebaut werden (Grenzen für Anwender nicht sichtbar!)
  - "Montieren" von Dateibäumen (*mounting*)
- Ein ausgezeichnetes Dateisystem ist das *Root File System*, dessen Wurzelverzeichnis gleichzeitig Wurzelverzeichnis des Gesamtsystems ist
  - Andere Dateisysteme können mit dem Befehl **`mount`** in das bestehende System hineinmontiert werden
  - Über *Network File System* (NFS) können auch Verzeichnisse anderer Rechner in einen lokalen Dateibaum hineinmontiert werden
    ➥ Grenzen zwischen Dateisystemen verschiedener Rechner weren unsichtbar

Montieren des Dateibaums

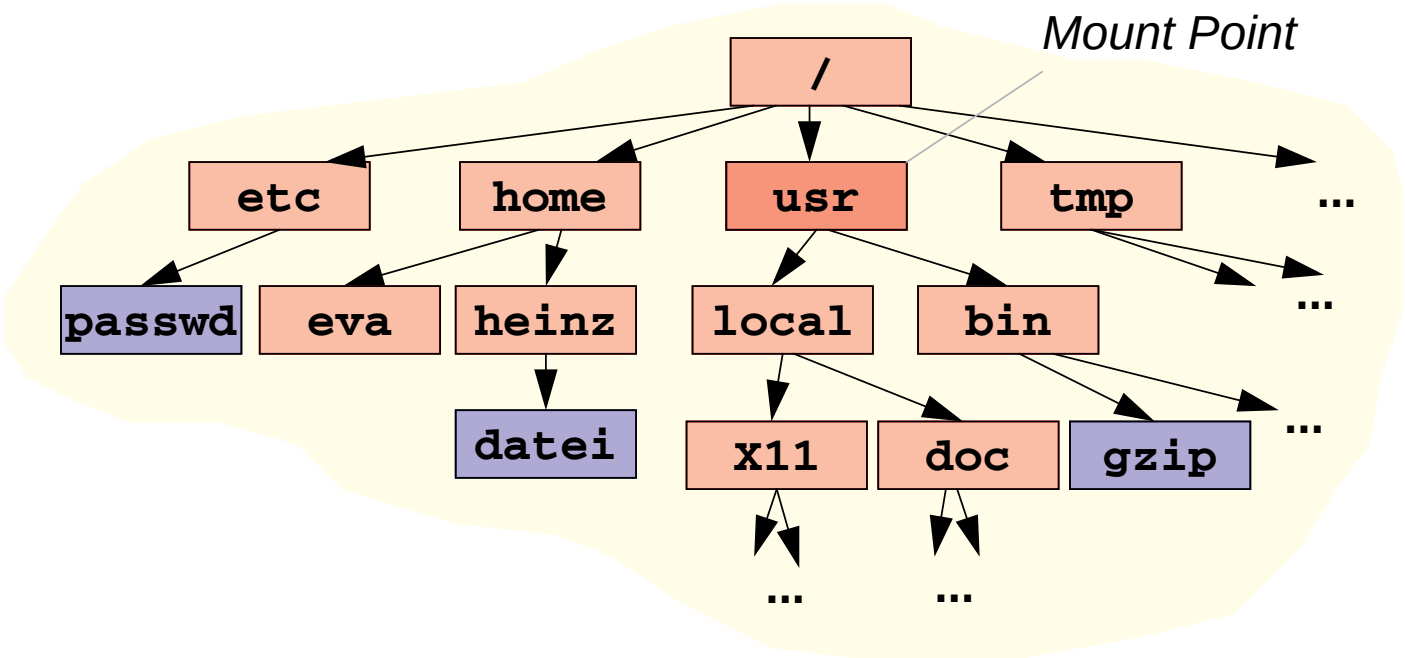
- Beispiel




```
mount /dev/dsk/0s5 /usr
```

Montieren des Dateibaums (2)

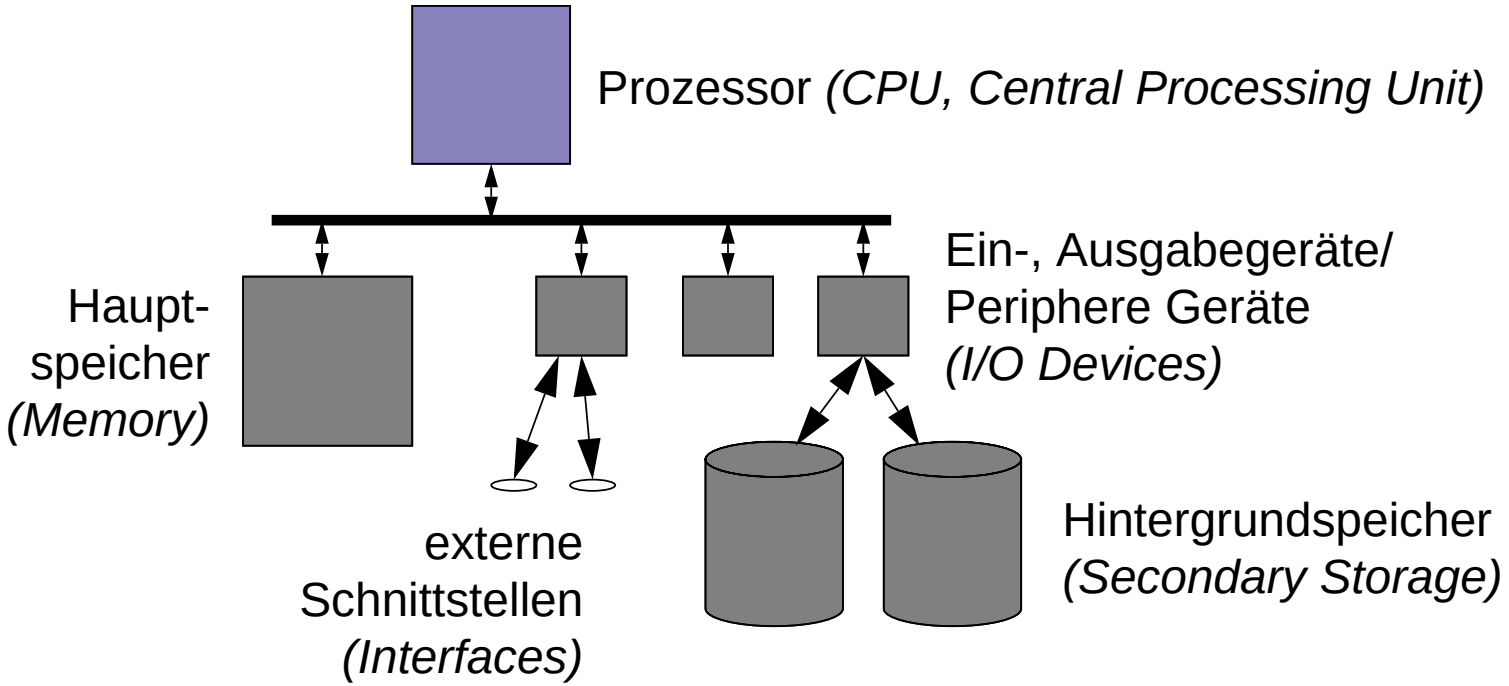
- Beispiel nach Ausführung des Montierbefehls



Überblick: Teil D Betriebssystemabstraktionen

Überblick

- Einordnung



Prozessor

- Register
  - Prozessor besitzt Steuer- und Vielzweckregister
  - Steuerregister:
    - Programmzähler *(Instruction Pointer)*
    - Stapelregister *(Stack Pointer)*
    - Statusregister
    - etc.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
  - Instruktion wird geladen und
  - ausgeführt
  - Programmzähler wird inkrementiert
  - dieser Vorgang wird ständig wiederholt

Prozessor (2)

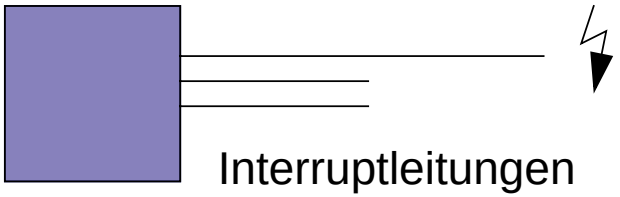
- Beispiel für Instruktionen

```
...
0010 5510000000    movl DS:$10, %ebx
0015 5614000000    movl DS:$14, %eax
001a 8a            addl %eax, %ebx
001b 5a18000000    movl %ebx, DS:$18
```

- Prozessor arbeitet in einem bestimmten Modus
  - Benutzermodus: eingeschränkter Befehlssatz
  - privilegierter Modus: erlaubt Ausführung privilegierter Befehle
    - ➤ Konfigurationsänderungen des Prozessors
    - ➤ Moduswechsel
    - ➤ spezielle Ein-, Ausgabebefehle

Prozessor (3)

- Unterbrechungen *(Interrupts)*



Signalisieren der Unterbrechung
(Interrupt Request; IRQ)

- ausgelöst durch ein Signal eines externen Geräts

  ➥ asynchron zur Programmausführung

  - ➤ Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
  - ➤ vorher werden alle Register einschließlich Programmzähler gesichert (z. B. auf dem Stack)
  - ➤ nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden
  - ➤ Unterbrechungen werden im privilegierten Modus bearbeitet

Prozessor (4)

- Ausnahmesituationen, Systemaufrufe *(Traps)*
  - ausgelöst durch eine Aktivität des gerade ausgeführten Programms
    - ➤ fehlerhaftes Verhalten
      (Zugriff auf ungültige Speicheradresse, ungültiger Maschinenbefehl, Division durch Null)
    - ➤ kontrollierter Eintritt in den privilegierten Modus
      (spezieller Maschinenbefehl - *Trap* oder *Supervisor Call*)
      – Implementierung der Betriebssystemschnittstelle

    ➥ synchron zur Programmausführung
  - Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
    - ➤ Ausnahmesituation wird geeignet bearbeitet (z. B. durch Abbruch der Programmausführung)
    - ➤ Systemaufruf wird durch Funktionen des Betriebssystems im privilegierten Modus ausgeführt (partielle Interpretation)

    ➤ Parameter werden nach einer Konvention übergeben (z. B. auf dem Stack)

Programme, Prozesse und Speicher

- **Programm:** Folge von Anweisungen
(hinterlegt beispielsweise als ausführbare Datei auf dem Hintergrundspeicher)

- **Prozess:** Betriebssystemkonzept
  - Programm, das sich in Ausführung befindet, und seine Daten
(Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - eine konkrete Ausführungsumgebung für ein Programm
Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...

- jeder Prozess bekommt einen eigenen virtuellen Adressraum zur Verfügung gestellt
  - eigener (virtueller) Speicherbereich von 0 bis 4 GB (32-Bit-Architekturen)
(bei 64-Bit-Architekturen auch beliebig viel mehr)
  - Datenbereiche von verschiedenen Prozessen und Betriebssystem sind gegeneinander geschützt
  - Datentransfer zwischen Prozessen nur durch Vermittlung des Betriebssystems möglich

Prozesse

- Bisherige Definition:
  - Programm, das sich in Ausführung befindet, und seine Daten
- eine etwas andere Sicht:
  - ein virtueller Prozessor, der ein Programm ausführt
    - Speicher → virtueller Adressraum
    - Prozessor → Zeitanteile am echten Prozessor
    - Interrupts → Signale
    - I/O-Schnittstellen → Dateisystem, Kommunikationsmechanismen
    - Maschinenbefehle → direkte Ausführung durch echten Prozessor
      oder partielle Interpretation von Trap-Befehlen durch Betriebssystemcode

Prozesse (2)

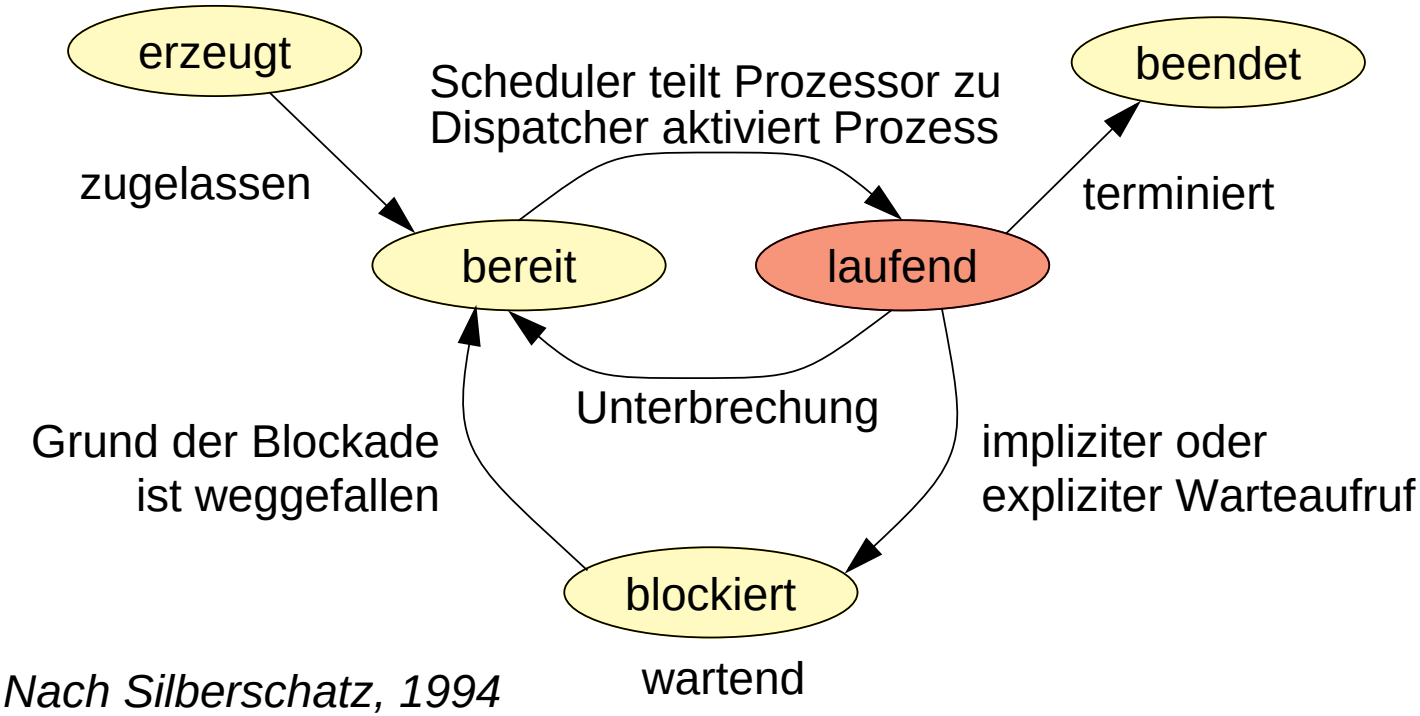
- Mehrprogrammbetrieb
  - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
  - steht nur ein echter Prozessor zur Verfügung, werden Zeitanteile der Rechenzeit an die Prozesse vergeben ***(Time Sharing System)***
  - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit zugeteilt bekommt, trifft das Betriebssystem ***(Scheduler)***
  - die Umschaltung zwischen Prozessen erfolgt durch das Betriebssystem ***(Dispatcher)***
  - Prozesse laufen nebenläufig
    (das ausgeführte Programm weiß nicht, an welchen Stellen auf einen anderen Prozess umgeschaltet wird)

Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** *(New)*
    Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** *(Ready)*
    Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** *(Running)*
    Prozess wird vom realen Prozessor ausgeführt
  - **Blockiert** *(Blocked/Waiting)*
    Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - **Beendet** *(Terminated)*
    Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

Prozesszustände (2)

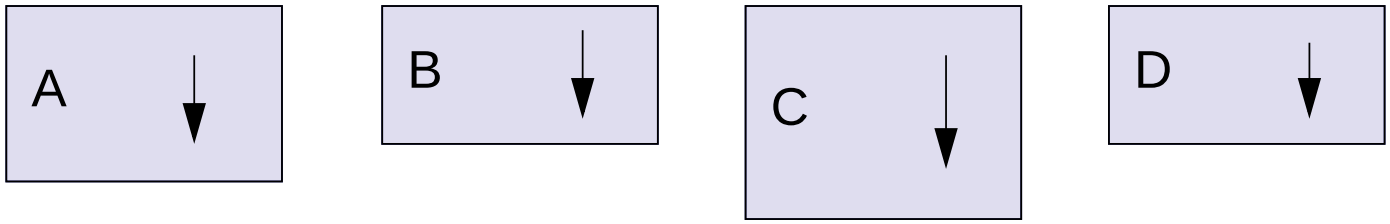
- Zustandsdiagramm



Nach Silberschatz, 1994

Prozesswechsel

- Konzeptionelles Modell

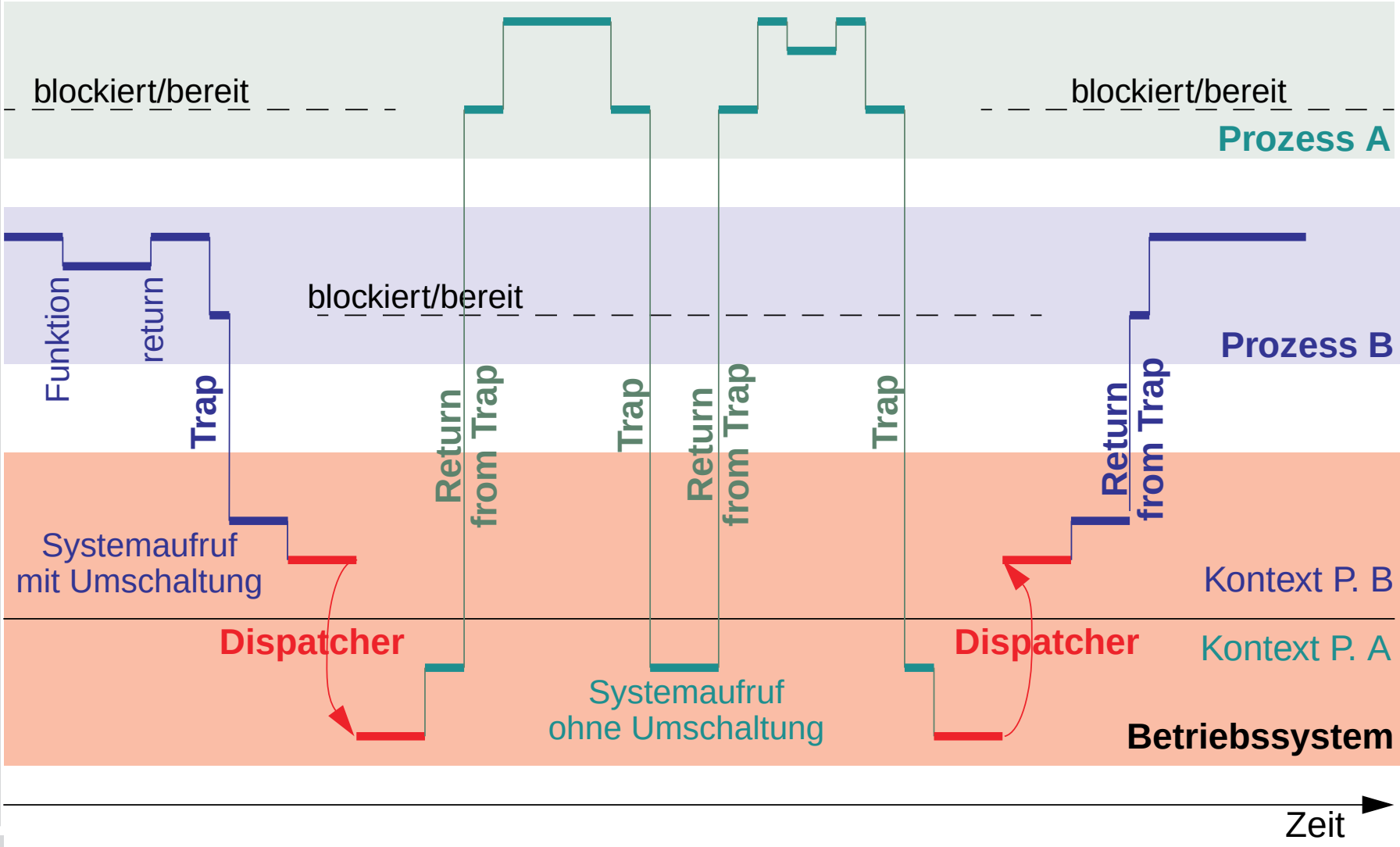


vier Prozesse mit eigenständigen Befehlszählern

- Umschaltung *(Context Switch)*
  - Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),
  - Auswahl des neuen Prozesses,
  - Ablaufumgebung des neuen Prozesses herstellen (z. B. Speicherabbildung, etc.),
  - gesicherte Register des neuen Prozesses laden und
  - Prozessor aufsetzen.

Prozesswechsel (2)

- Ablauf von zwei Prozessen in Benutzermodus und Kern mit Umschaltung



Prozesswechsel (3)

- Prozesskontrollblock *(Process Control Block; PCB)*
  - Datenstruktur des Betriebssystems,
    die alle nötigen Daten für einen Prozess hält.
    Beispielsweise in UNIX:
    - Prozessnummer *(PID)*
    - verbrauchte Rechenzeit
    - Erzeugungszeitpunkt
    - Kontext (Register etc.)
    - Speicherabbildung
    - Eigentümer *(UID, GID)*
    - Wurzelkatalog, aktueller Katalog
    - offene Dateien
    - …

Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
} else if( p > 0 ) {
    /* parent */
    ...
} else {
    /* error */
```

Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
} else if( p > 0 ) {
    /* parent */
    ...
} else {
    /* error */
```

Kind

```
pid_t p;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
} else if( p > 0 ) {
    /* parent */
    ...
} else {
    /* error */
```

Prozesserzeugung (2)

- Der Kind-Prozess ist eine perfekte Kopie des Vaters
  - gleiches Programm
  - gleiche Daten (gleiche Werte in Variablen)
  - gleicher Programmzähler (nach der Kopie)
  - gleicher Eigentümer
  - gleiches aktuelles Verzeichnis
  - gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
  - ...
- Unterschiede:
  - Verschiedene PIDs
  - **`fork()`** liefert verschiedene Werte als Ergebnis für Vater und Kind

Ausführen eines Programms (UNIX)

- Das von einem Prozess gerade ausgeführte Programm kann durch ein neues Programm ersetzt werden

```
int execv( const char *path, char *const argv[]);
```

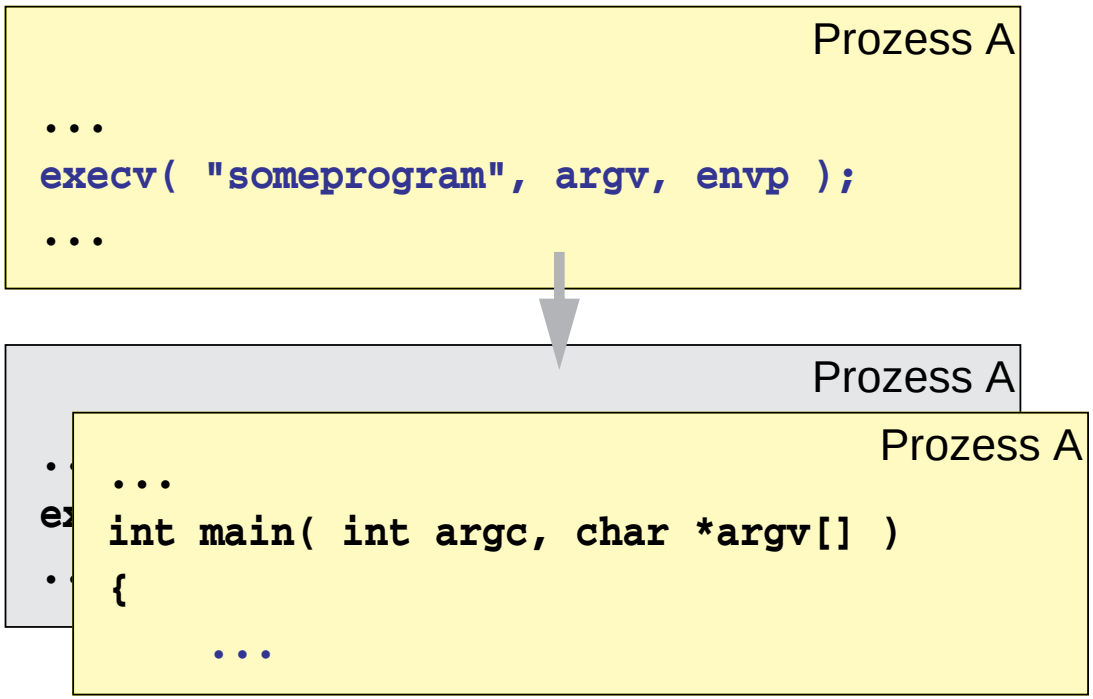
Prozess A

```
...
execv( "someprogram", argv, envp );
...
```

Ausführen eines Programms (UNIX)

- Das von einem Prozess gerade ausgeführte Programm kann durch ein neues Programm ersetzt werden

```
int execv( const char *path, char *const argv[]);
```



das zuvor ausgeführte Programm wird dadurch beendet.

- ➤ Es wird nur das Programm beendet, nicht aber der Prozess!!!

Operationen auf Prozessen (UNIX)

- Prozess beenden

```
void _exit( int status );
[ void exit( int status ); ]
```

- Prozessidentifikator

```
pid_t getpid( void );                    /* eigene PID */
pid_t getppid( void );                   /* PID des Vaterprozesses */
```

- Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

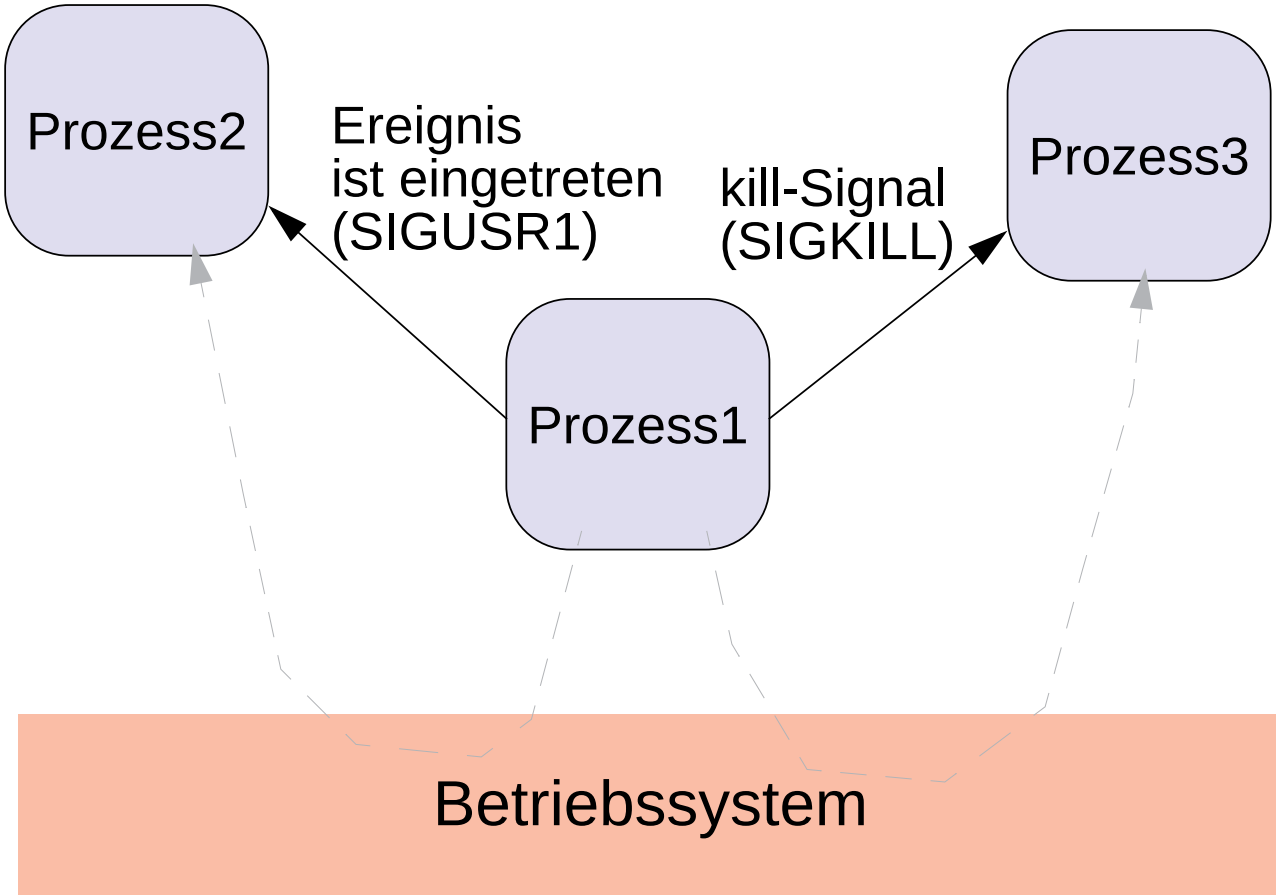
  - ➤ das Betriebssystem schreibt den Exit-Status des beendeten Kindprozesses in ein Byte der Variablen, auf die **`statusp`** zeigt.

Signalisierung des Systemkerns an einen Prozess

- Software-Implementierung der Hardware-Konzepte
  - **Interrupt**: asynchrones Signal aufgrund eines "externen" Ereignisses
    - CTRL-C auf der Tastatur gedrückt (Interrupt-Signal)
    - Timer abgelaufen
    - Kind-Prozess terminiert
    - …
  - **Trap:** synchrones Signal, ausgelöst durch die Aktivität des Prozesses
    - Zugriff auf ungültige Speicheradresse
    - Illegaler Maschinenbefehl
    - Division durch NULL
    - Schreiben auf eine geschlossene Kommunikationsverbindung
    - …

Kommunikation zwischen Prozessen

- ein Prozess will einem anderen ein Ereignis signalisieren

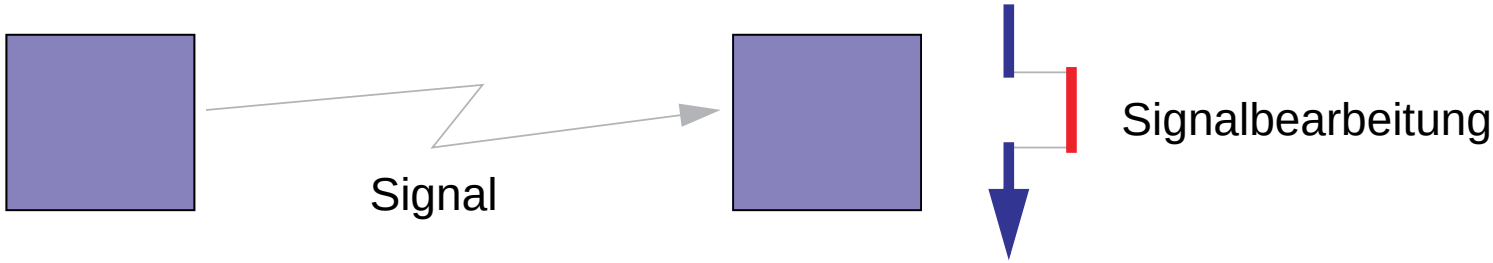


Reaktion auf Signale

- abort
  - erzeugt Core-Dump (Segmente + Registercontext) und beendet Prozess
- exit
  - beendet Prozess, ohne einen Core-Dump zu erzeugen
- ignore
  - ignoriert Signal
- stop
  - stoppt Prozess
- continue
  - setzt gestoppten Prozess fort
- signal handler
  - Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

POSIX Signalbehandlung

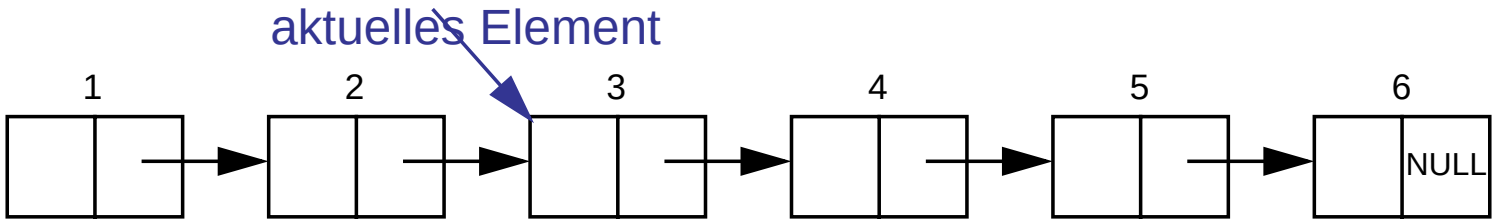
- Betriebssystemschnittstelle zum Umgang mit Signalen
- Signal bewirkt Aufruf einer Funktion (analog ISR)



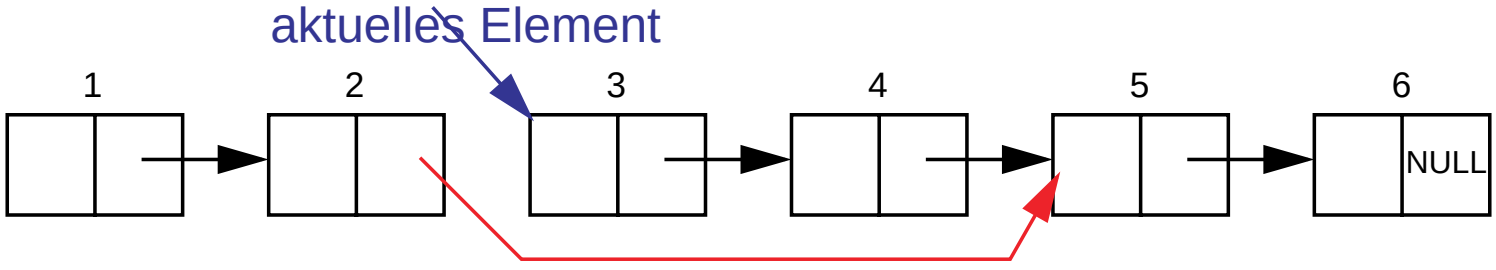
  - nach der Behandlung läuft der Prozess an der unterbrochenen Stelle weiter
- Systemschnittstelle
  - sigaction – Anmelden einer Funktion = Einrichten der ISR-Tabelle
  - sigprocmask – Blockieren/Freigeben von Signalen ≈ cli() / sei()
  - sigsuspend – Freigeben + passives Warten auf Signal + wieder Blockieren
    ≈ sei() + sleep_cpu() + cli()
  - kill – Signal an anderen Prozess verschicken

Signale und Nebenläufigkeit → Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller
- Beispiel:
  - main-Funktion läuft durch eine verkettete Liste



  - Prozess erhält Signal; Signalhandler entfernt Elemente 3 und 4 aus der Liste und gibt den Speicher dieser Elemente frei



Signale und Nebenläufigkeit → Race Conditions (2)

- zusätzliche Problem:
  - Signale können die Behandlung anderer Signale unterbrechen
  - Signale können Bibliotheksfunktionen unterbrechen, die nicht dafür eingerichtet sind
    - ➤ Funktionen `printf()` oder `getchar()`
    - ➤ siehe Funktion `readdir` in Kapitel 18

- Lösung:
  - ➤ Signal während Ausführung von kritischen Programmabschnitten blockieren!
  - ➤ kritische Bibliotheksfunktionen aus Signalbehandlungsfunktionen möglichst nicht aufrufen

- grundlegendes Problem
  man muss wissen, welche Funktion(en) in Bezug auf Nebenläufigkeit problematisch **(nicht reenttrant)** sind

Überblick: Teil D Betriebssystemabstraktionen

Speicherorganisation

```
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```

Wo kommt der Speicher für diese Variablen her?

- **Statische Allokation** – Reservierung beim Übersetzen / Linken
  - Betrifft alle globalen/statischen Variablen, sowie den Code ↪ 12–5
  - Allokation durch Platzierung in einer Sektion

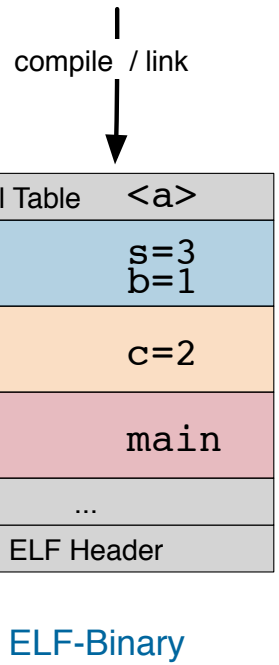
.text	– enthält den Programmcode	`main()`
.bss	– enthält alle mit 0 initialisierten Variablen	`a`
.data	– enthält alle mit anderen Werten initalisierten Variablen	`b,s`
.rodata	– enthält alle unveränderlichen Variablen	`c`

- **Dynamische Allokation** – Reservierung zur Laufzeit
  - Betrifft lokale auto-Variablen und explizit angeforderten Speicher

Stack	– enthält alle aktuell lebendigen auto-Variablen	`x,y,p`
Heap	– enthält explizit mit `malloc()` angeforderte Speicherbereiche	`*p`

Speicherorganisation auf einem μC

```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```



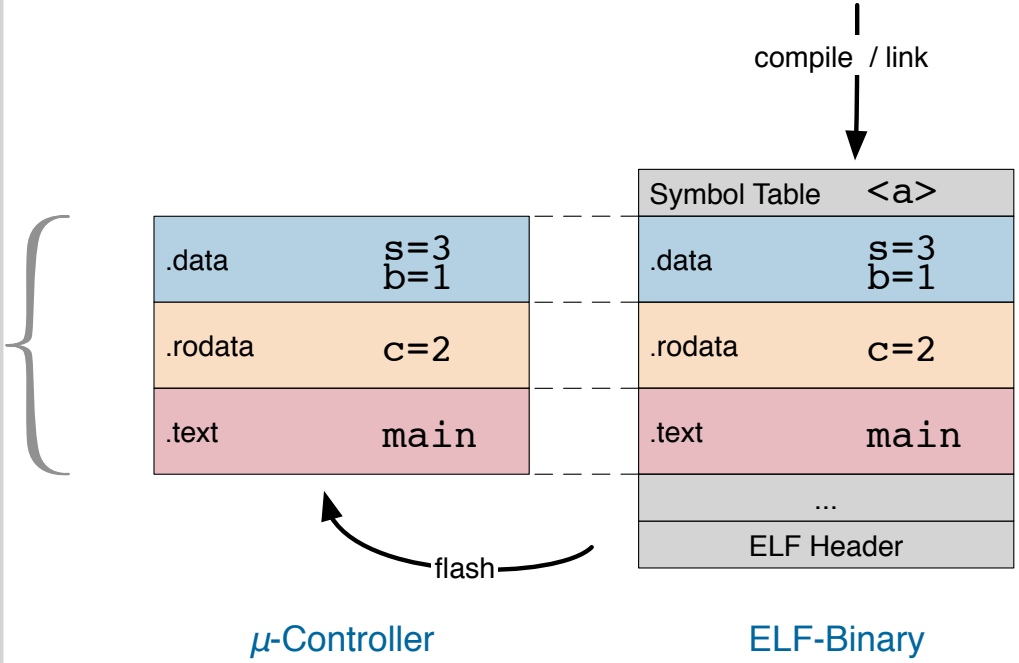
Quellprogramm

ELF-Binary

Beim Übersetzen und Linken werden die Programmelemente in entsprechenden Sektionen der ELF-Datei zusammen gefasst. Informationen zur Größe der `.bss`-Sektion landen ebenfalls in der Symboltabelle.

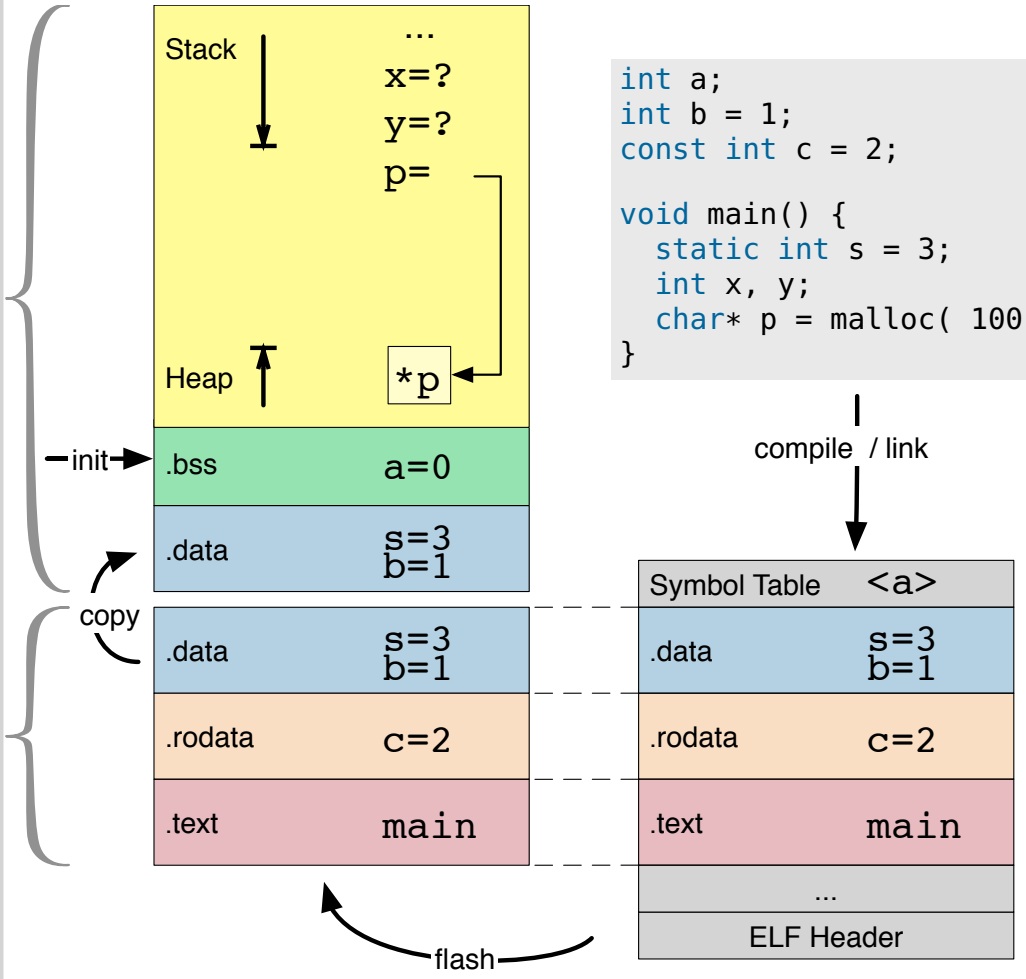
Speicherorganisation auf einem μC

```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```



Zur Installation auf dem μC werden `.text` und `.[ro]data` in den Flash-Speicher des μC geladen.

Speicherorganisation auf einem μC




```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```

Quellprogramm

Beim Systemstart wird das `.bss`-Segment im RAM angelegt und mit `0` initialisiert, das `.data`-Segment wird aus dem Flash ins RAM kopiert.
Das verbleibende RAM wird für den Stack und (falls vorhanden) den Heap verwendet.

Verfügt die Architektur über keinen Daten-Flashspeicher (beim ATmega der Fall ↪ 14–3), so werden konstante Variablen ebenfalls in `.data` abgelegt (und belegen zur Laufzeit RAM).

Speicherorganisation in einem UNIX-Prozess

- **Programm:** Folge von Anweisungen
- **Prozess:** Betriebssystemkonzept zur Ausführung von Programmen
  - Programm, das sich in Ausführung befindet, und seine Daten (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - Eine konkrete Ausführungsumgebung für ein Programm (Prozessor, Speicher, ...) $\mapsto$ vom Betriebssystem verwalteter *virtueller Computer*
- Jeder Prozess bekommt einen virtuellen Adressraum zugeteilt
  - 4 GB auf einem 32-Bit-System, davon bis zu 3 GB für die Anwendung
    - In das verbleibende GB werden Betriebssystem und *memory-mapped* Hardware (z. B. PCI-Geräte) eingeblendet
    - Daten des Betriebssystems werden durch Zugriffsrechte geschützt
  - Zugriff auf andere Prozesse ist nur über das Betriebssystem möglich
  - Virtueller Speicher wird durch das Betriebssystem auf physikalischen (Hintergrund-)Speicher abgebildet

Speicherorganisation in einem UNIX-Prozess (Forts.)

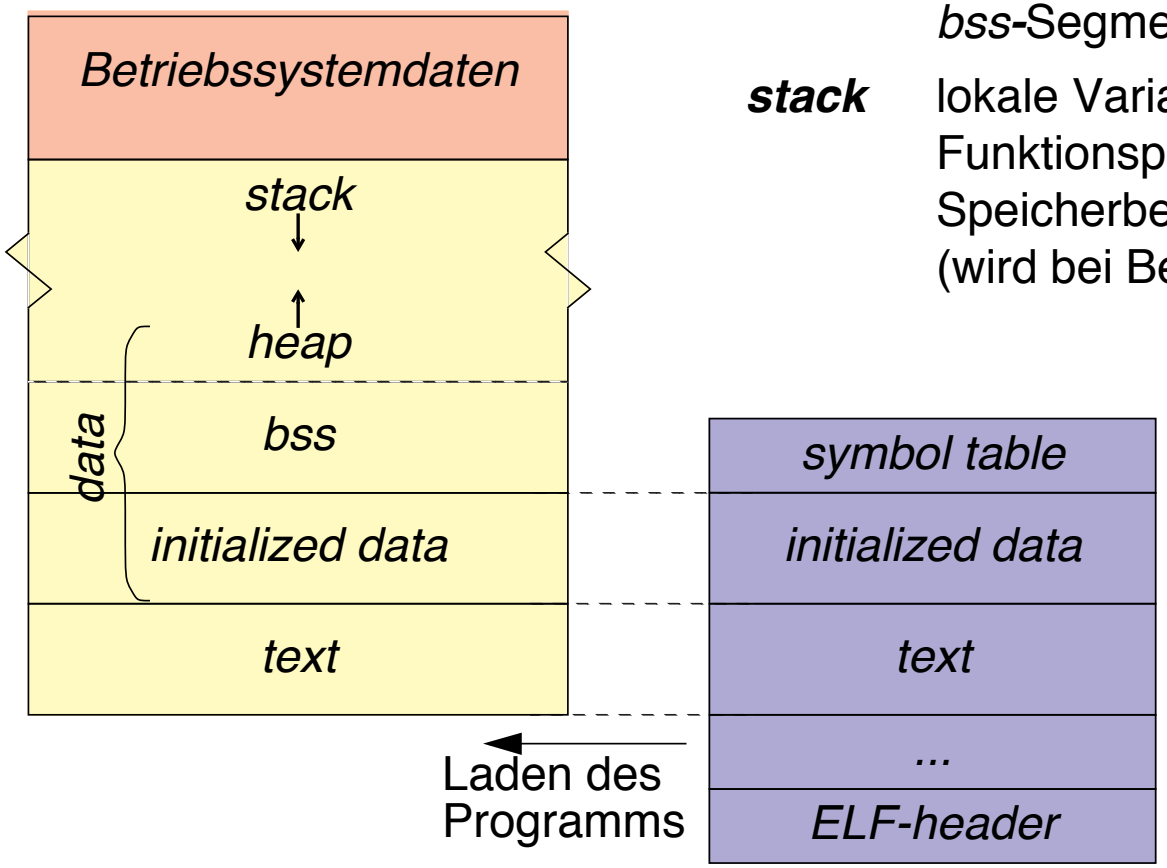
text Programmkode

data globale und static Variablen

bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (***sbrk(2)***, ***malloc(3)***)

stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

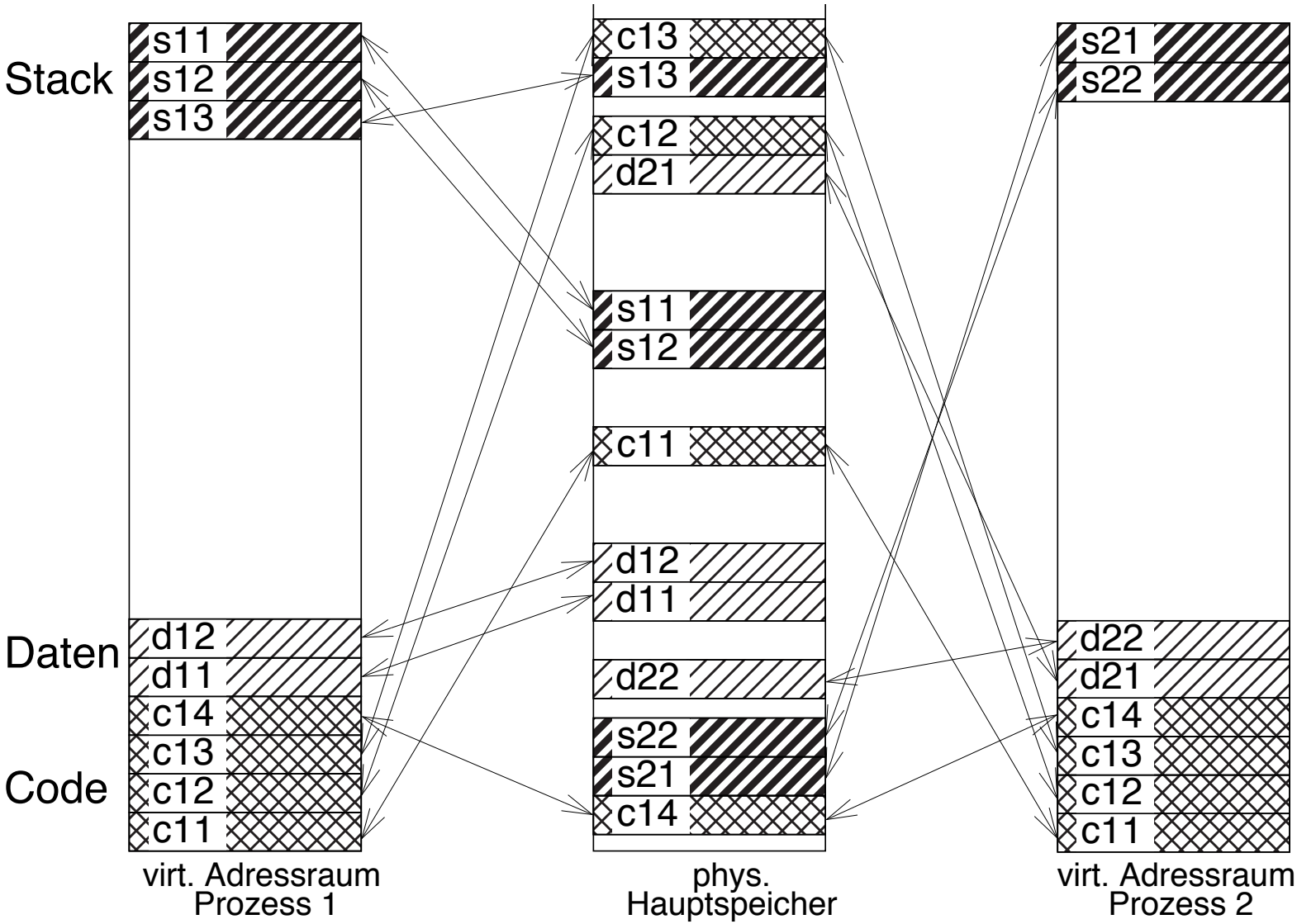


Aufbau eines Programms (ELF-Format)

Seitenbasierte Speicherverwaltung

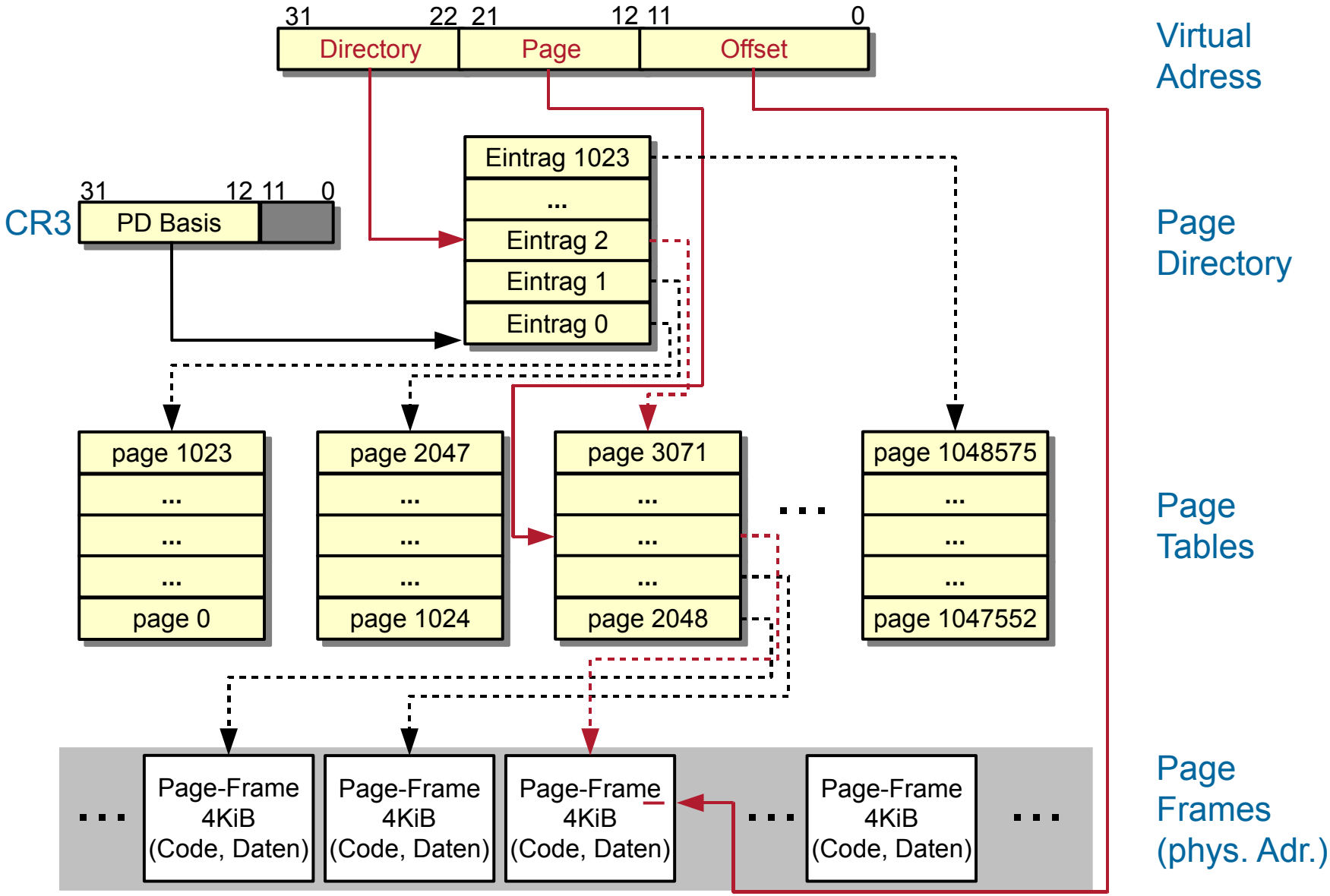
- Die Abbildung von virtuellem Speicher (*VS*) auf physikalischen Speicher (*PS*) erfolgt durch Seitenaddressierung (*Paging*)
  - *VS* eines Prozesses ist unterteilt in Speicherseiten (*Memory Pages*)
    - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
    - in dieser Granularität wird Speicher vom Betriebssystem zugewiesen
  - *PS* ist analog unterteilt in Speicherrahmen (*Page Frames*)
  - Abbildung: *Seite* $\mapsto$ *Rahmen* über eine Seitentabelle (*Page Table*)
    - Umrechnung *VS* auf *PS* bei jedem Speicherzugriff
    - Hardwareunterstützung durch MMU (*Memory Management Unit*)
    - Betriebssystem kann Seiten auf den Hintergrundspeicher auslagern
    - Abbildung ist nicht linkseindeutig: Seiten aus mehreren Prozesse können auf denselben Rahmen verweisen (z. B. gemeinsamer Programmcode)
- Seitenbasierte Speicherverwaltung ist auch ein Schutzkonzept
  - Seiten sind mit Zugriffsrechten versehen: *Read*, *Read–Write*, *Execute*
  - MMU überprüft bei der Umrechnung, ob der Zugriff erlaubt ist

Seitenbasierte Speicherverwaltung



Seitenbasierte Speicherverwaltung

Technische Sicht (IA32)



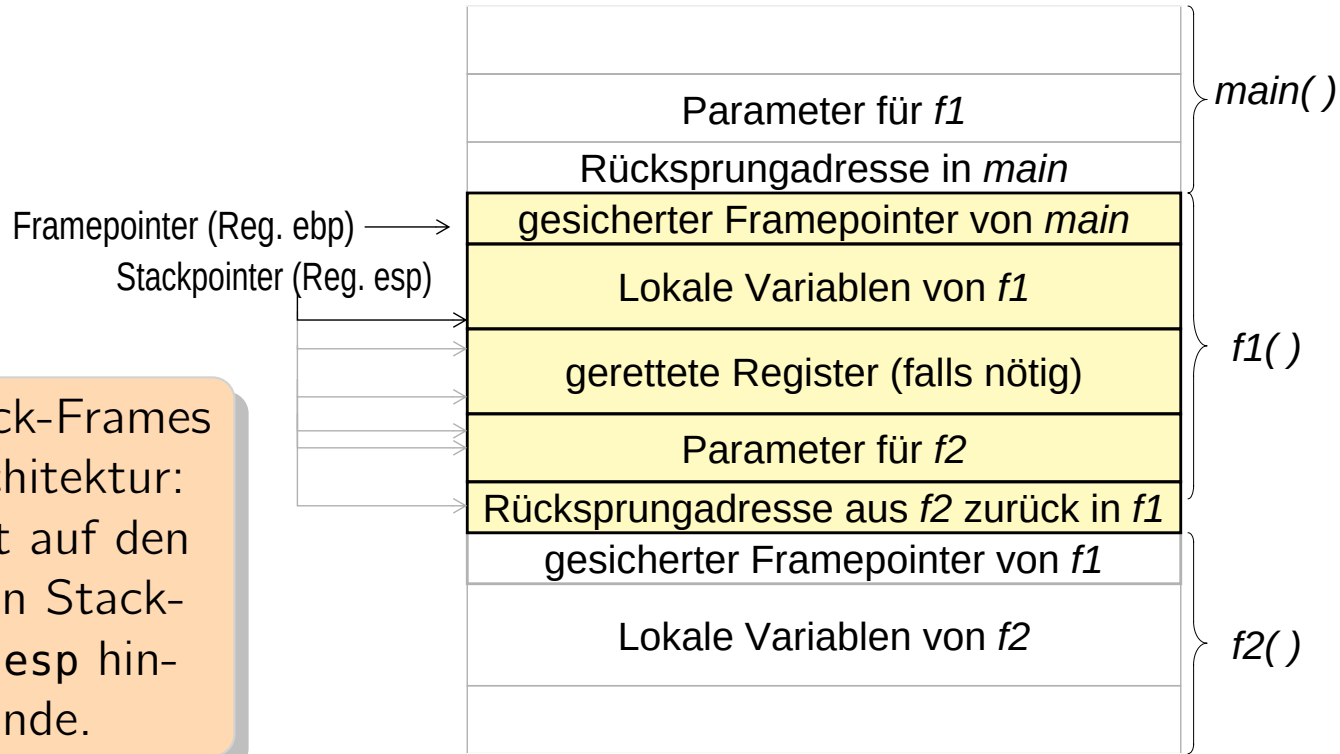
Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void* malloc( size_t n )` fordert einen Speicherblock der Größe *n* an; Rückgabe bei Fehler: 0-Zeiger (`NULL`)
  - `void free( void* pmem )` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```
#include <stdlib.h>
int* intArray( uint16_t n ) {    // alloc int[n] array
  return (int*) malloc( n * sizeof int );
}
void main() {
  int* array  = intArray(100);   // alloc memory for 100 ints
  if( array ) {                  // malloc() returns NULL on failure
    ...                          // if succeeded, use array
    array[99]   = 4711;
    ...
    free( array );               // free allocated block (** IMPORTANT! **)
  }
}
```


Dynamische Speicherallokation: Stack

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem Stack (Stapel, Keller) verwaltet
  - Prozessorregister `[e]sp` zeigt immer auf den nächsten freien Eintrag
  - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von Stack-Frames

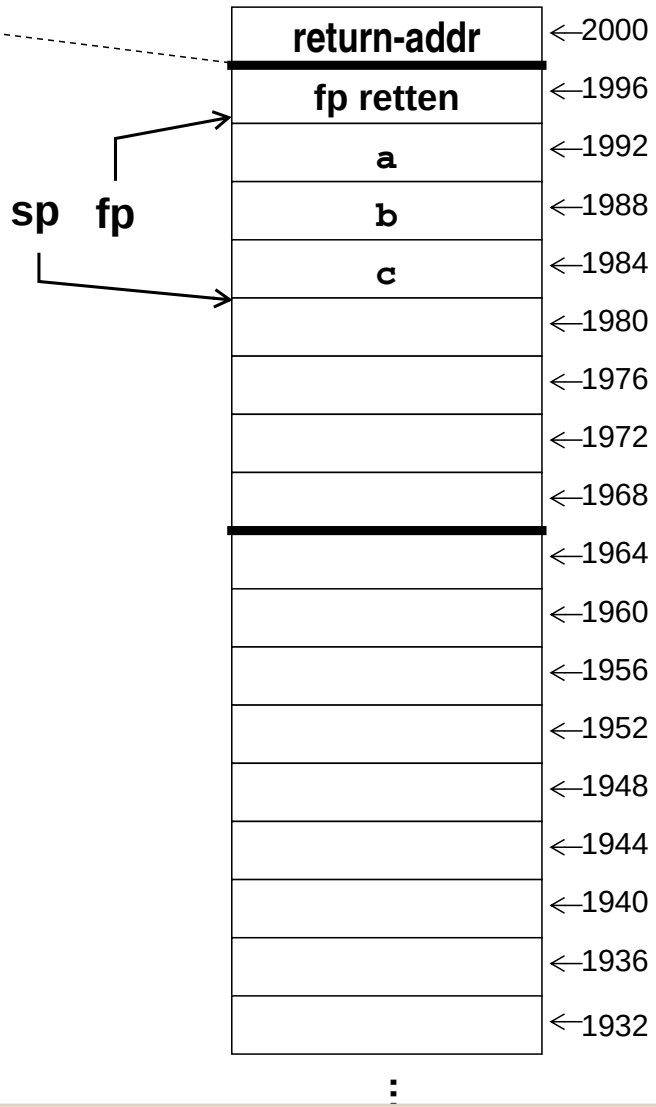


Aufbau eines Stack-Frames auf der IA-32-Architektur: Register **ebp** zeigt auf den Beginn des aktiven Stack-Frames; Register **esp** hinter das aktuelle Ende.

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

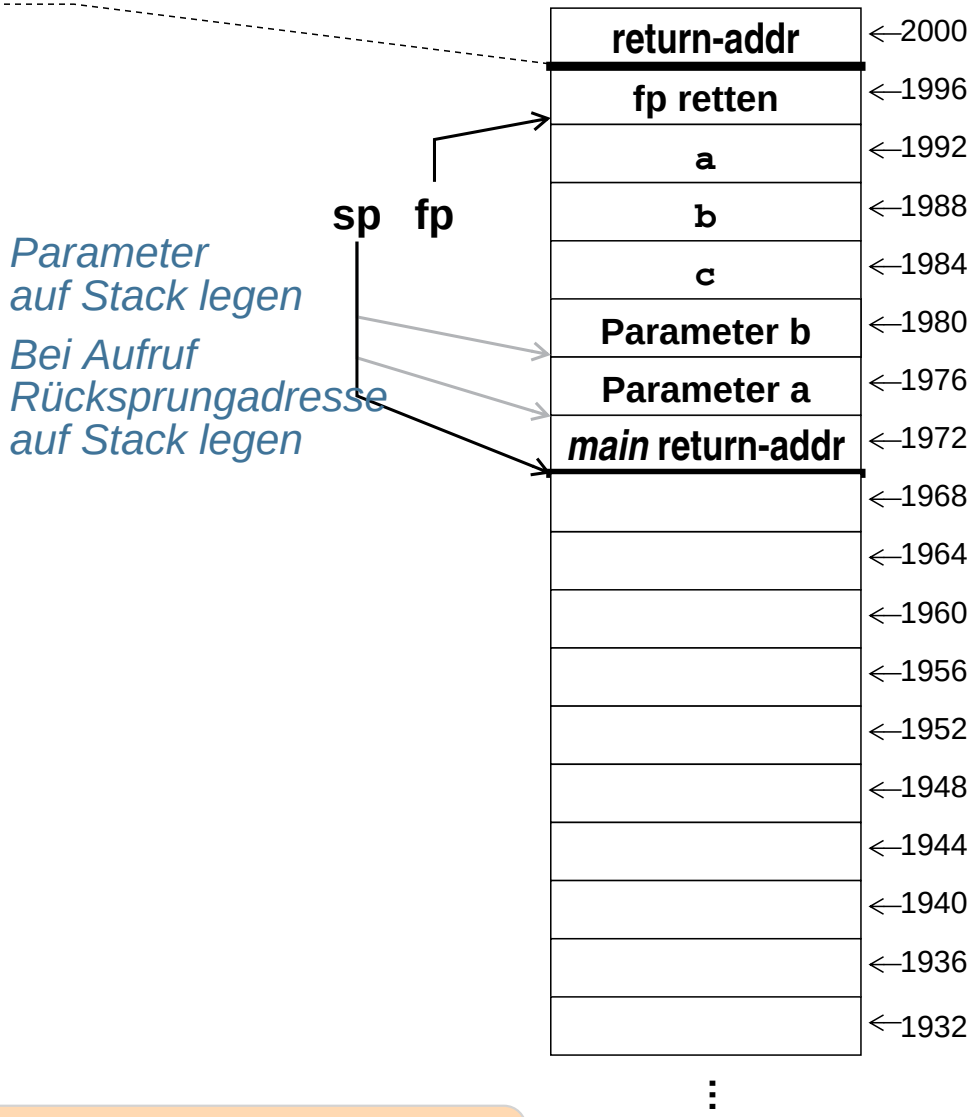
Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



Beispiel hier für 32-Bit-Architektur (4-Byte `int`s), `main()` wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

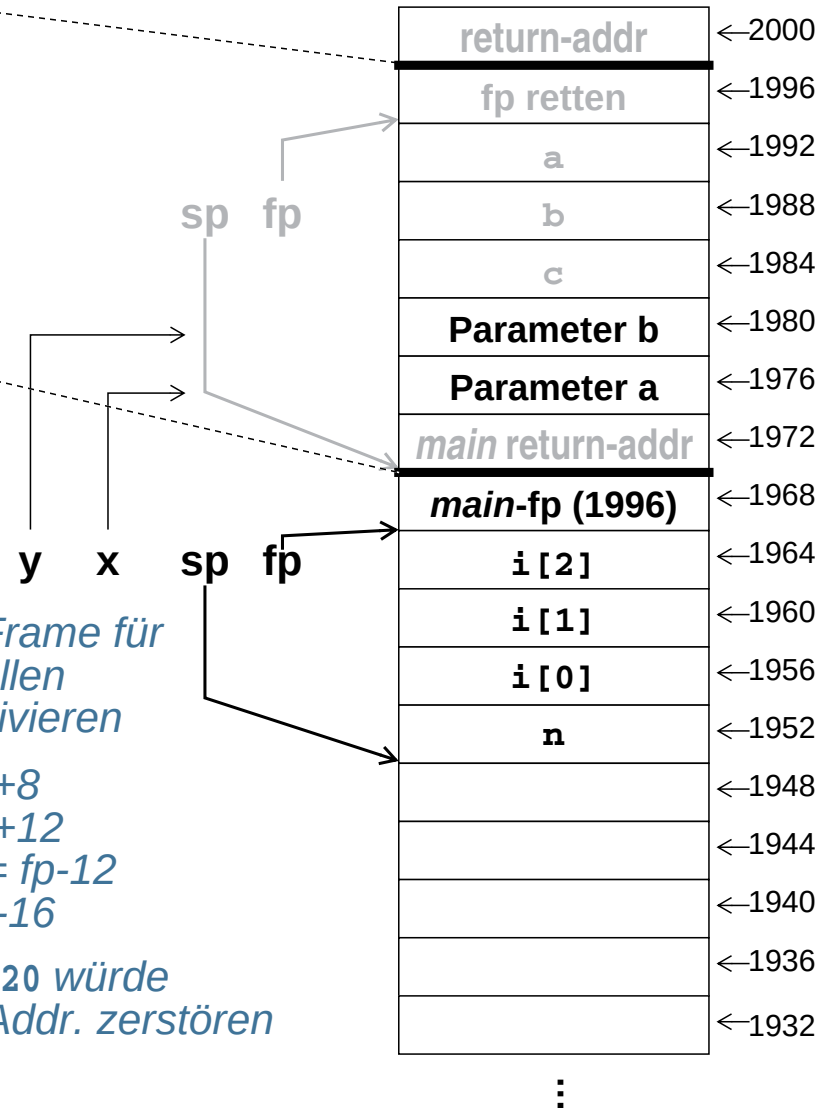


`main()` bereitet den Aufruf von `f1(int, int)` vor

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Stack-Frame für f1 erstellen und aktivieren

&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

`f1()` wurde soeben betreten

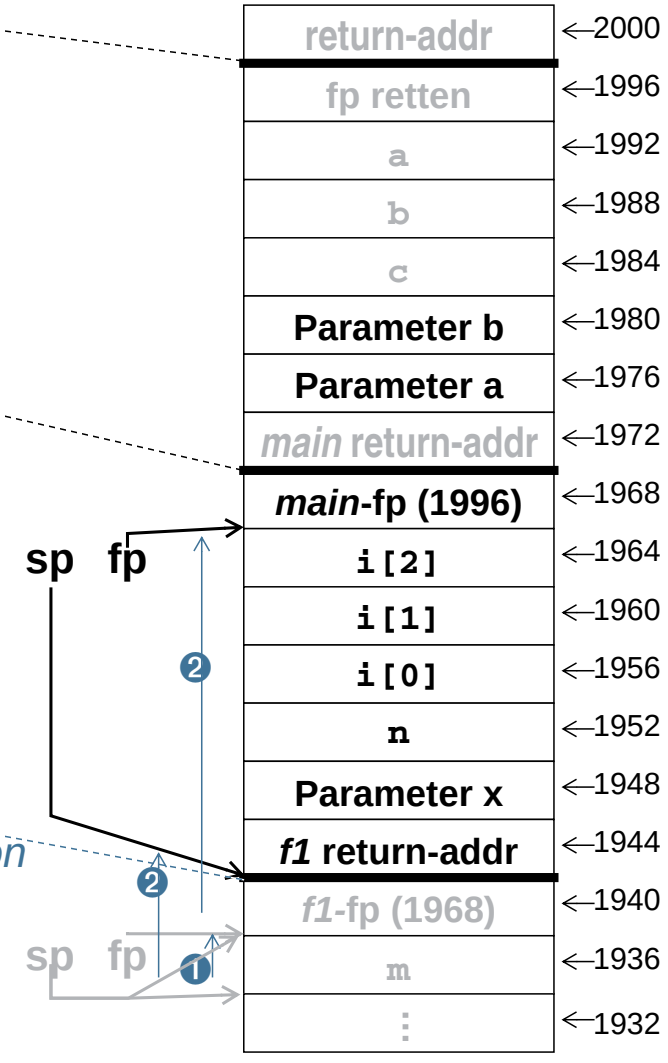
Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

Stack-Frame von f2 abräumen

❶ *sp = fp*

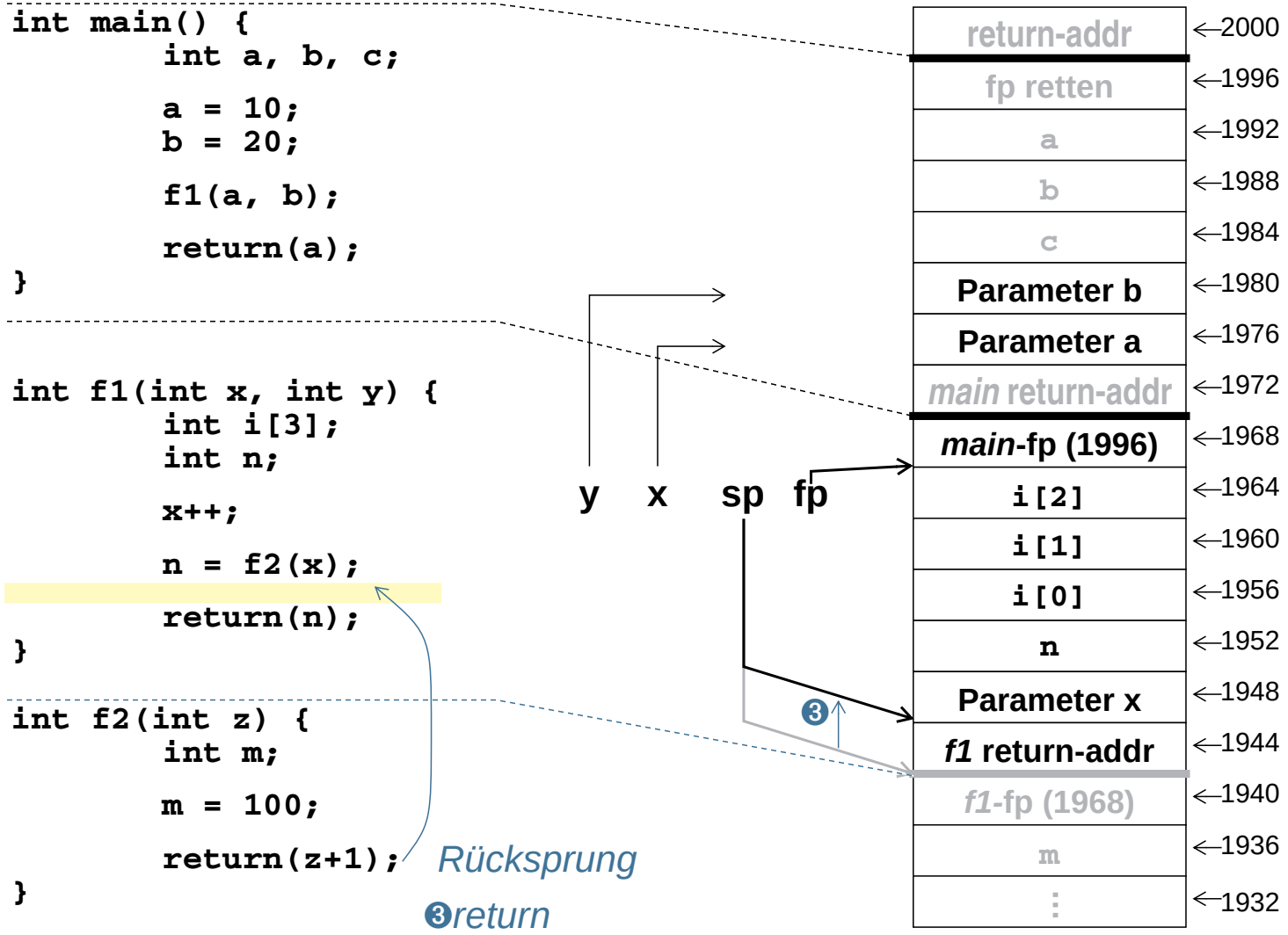
❷ *fp = pop(sp)*



f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

Stack-Aufbau bei Funktionsaufrufen

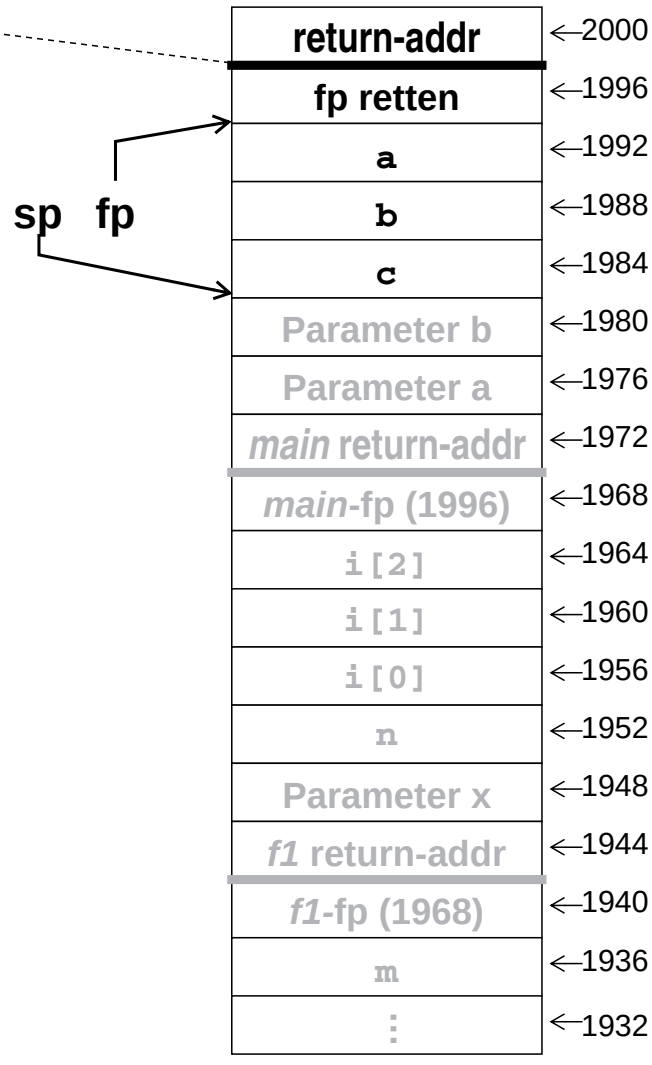
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

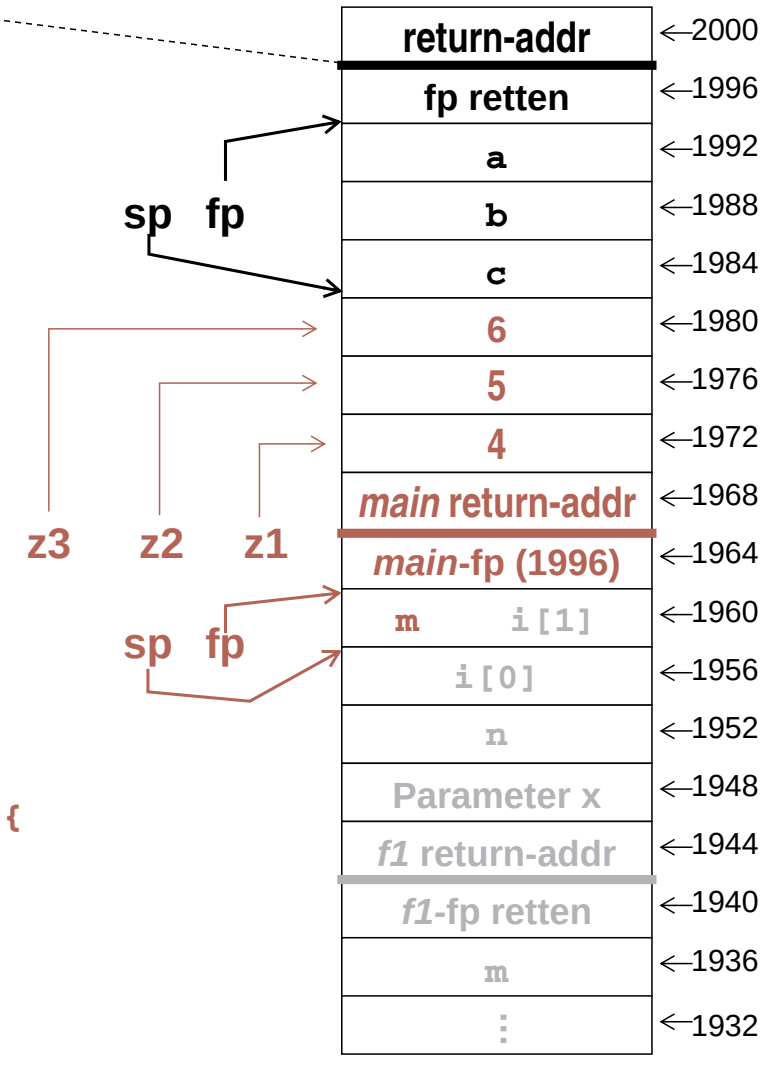


zurück in `main()`

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        f3(4,5,6);
}
```

was wäre, wenn man nach f1 jetzt eine Funktion f3 aufrufen würde?




```
int f3(int z1, int z2, int z3) {
        int m;

        return(m);
}
```

m wird nicht initialisiert ⇝„erbt“ alten Wert vom Stapel

Statische versus dynamische Allokation

- Bei der **μC-Entwicklung** wird statische Allokation bevorzugt
  - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
  - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ 1–4)

```
lohmann@faui48a:$ size sections.avr
text     data     bss     dec     hex filename
682      10       6       698     2ba sections.avr
```

Sektionsgrößen des Programms von ↪ 20–1

⇝ Speicher möglichst durch `static`-Variablen anfordern

- Regel der geringstmöglichen Sichtbarkeit beachten ↪ 12–6
- Regel der geringstmöglichen Lebensdauer „sinnvoll“ anwenden

- Ein Heap ist verhältnismäßig teuer ⇝ wird möglichst vermieden
  - Zusätzliche Speicherkosten durch Verwaltungsstrukturen und Code
  - Speicherbedarf zur Laufzeit schlecht abschätzbar
  - Risiko von Programmierfehlern und Speicherlecks

Statische versus dynamische Allokation (Forts.)

- Bei der Entwicklung für eine **Betriebssystemplattform** ist dynamische Allokation hingegen sinnvoll
  - **Vorteil:** Dynamische Anpassung an die Größe der Eingabedaten (z. B. bei Strings)
  - Reduktion der Gefahr von *Buffer-Overflow*-Angriffen

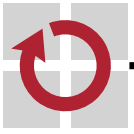
$\rightsquigarrow$ Speicher für Eingabedaten möglichst auf dem Heap anfordern

- Das Risiko von Programmierfehlern und Speicherlecks bleibt!

Überblick: Teil D Betriebssystemabstraktionen

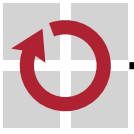
21 Kontrollfäden / Aktivitätsträger *(Threads)*

- Mehrere Prozesse zur Strukturierung von Problemlösungen
  - Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
    z. B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
    z. B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
  - Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
    - ➤ früher nur bei Hochleistungsrechnern (Aerodynamik, Wettervorhers.)
    - ➤ durch Multicoresysteme jetzt massive Verbreitung
  - Client-Server-Anwendungen unter UNIX:
    pro Anfrage wird ein neuer Prozess gestartet
    z. B. Webserver



Prozesse mit gemeinsamem Speicher

- Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse
- Nachteile
  - viele Betriebsmittel zur Verwaltung eines Prozesses notwendig
    - Dateideskriptoren
    - Speicherabbildung
    - Prozesskontrollblock
  - Prozessumschaltungen sind aufwändig
- Vorteil
  - in Multiprozessorsystemen sind echt parallele Abläufe möglich

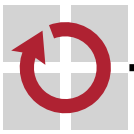


Threads in einem Prozess

★ **Lösungsansatz:**
Kontrollfäden / Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(Lightweight Processes, LWPs)*

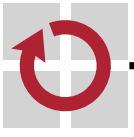
- Jeder Thread repräsentiert einen eigenen aktiven Ablauf:
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack
- eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln (gemeinsame Ausführungsumgebung)
  - Instruktionen
  - Datenbereiche (Speicher)
  - Dateien, etc.

➥ letztlich wird das Konzept des Prozesses aufgespalten:
eine Ausführungsumgebung für mehrere (parallele oder nebenläufige) Abläufe



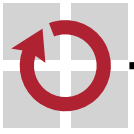
Threads (2)

- Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.
  - es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
  - Speicherabbildung muss nicht gewechselt werden.
  - alle Systemressourcen bleiben verfügbar.
- ein klassischer UNIX-Prozess ist ein Adressraum mit einem Thread
- Implementierungen von Threads
  - User-level Threads
  - Kernel-level Threads



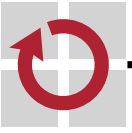
User-Level-Threads

- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - Realisierung durch Bibliotheksfunktionen
  - Betriebssystem sieht nur einen Kontrollfaden
- ★ Vorteile
  - keine Systemaufrufe zum Umschalten erforderlich
  - effiziente Umschaltung (einige wenige Maschinenbefehle)
  - Schedulingstrategie in der Hand des Anwendungsprogrammierers
- ▲ Nachteile
  - bei blockierenden Systemaufrufen bleibt die ganze Anwendung (und damit alle User-Level-Threads) stehen
  - kein Ausnutzen eines Multiprozessors möglich



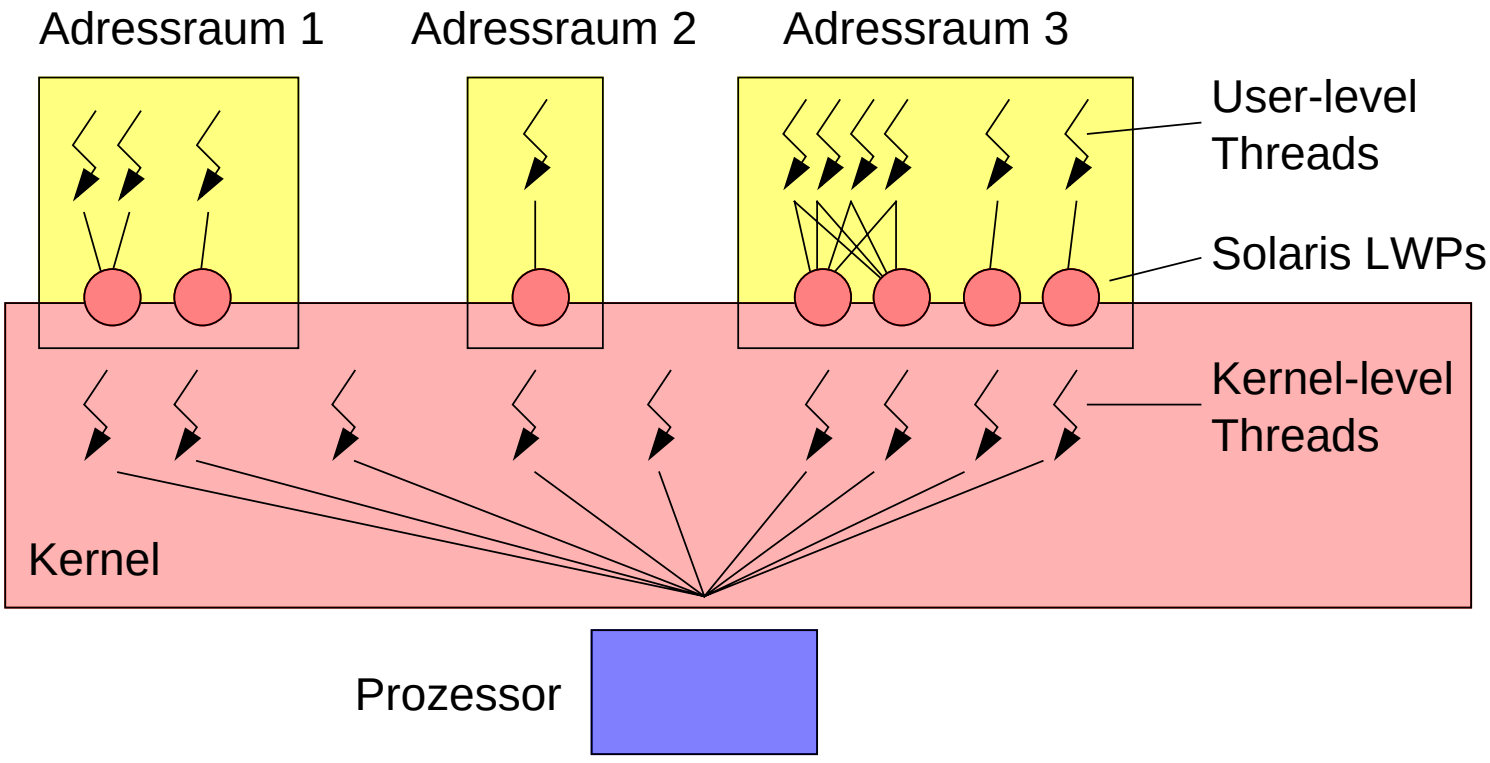
Kernel-Level-Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level-Threads
  - Betriebssystem schaltet zwischen Threads um
- ★ Vorteile
  - kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
  - Betriebssystem kann mehrere Threads einer Anwendung gleichzeitig auf verschiedenen Prozessoren laufen lassen
- ▲ Nachteile
  - weniger effizientes Umschalten zwischen Threads (Umschaltung in den Systemkern notwendig)
  - Schedulingstrategie meist durch Betriebssystem vorgegeben



Mischform: LWPs und Threads (Bsp. Solaris)

- Solaris kennt Kernel-, User-Level-Threads und LWPs



Nach Silberschatz, 1994

➤ wenige Kernel-level-Threads um Parallelität zu erreichen, viele User-level-Threads, um die unabhängigen Abläufe in der Anwendung zu strukturieren

Koordinierung / Synchronisation

- Threads arbeiten nebenläufig oder parallel auf Multiprozessor
- Threads haben gemeinsamen Speicher

➥ alle von Interrupts und Signalen bekannten Probleme beim Zugriff auf gemeinsame Daten treten auch bei Threads auf

★ Unterschied zwischen Threads und Interrupt-Service-Routinen bzw. Signal-Handler-Funktionen:

- "Haupt-Kontrollfaden" der Anwendung und eine ISR bzw. ein Signal-Handler sind nicht gleichberechtigt
  - ➤ ISR bzw. Signal-Handler unterbricht den Haupt-Kontrollfaden aber ISR bzw. Signal-Handler werden nicht unterbrochen
- zwei Threads sind gleichberechtigt
  - ➤ ein Thread kann jederzeit zugunsten eines anderen unterbrochen werden (Scheduler) oder parallel zu einem anderen arbeiten (MPS)

➥ Interrupts sperren oder Signale blockieren hilft nicht!

Koordinierungsprobleme

- Grundlegende Probleme
  - gegenseitiger Ausschluss (Koordinierung)
    - ➤ ein Thread möchte auf einen kritischen Datenbereich zugreifen und verhindern, dass andere Threads gleichzeitig zugreifen
  - gegenseitiges Warten (Synchronisation)
    - ➤ ein Thread will darauf warten, dass ein anderer einen bestimmten Bearbeitungsstand erreicht hat
- Komplexere Koordinierungsprobleme (Beispiele)
  - Bounded Buffer
    - ➤ Threads schreiben Daten in einen Pufferspeicher (meist als Feld implementiert), anderere entnehmen Daten
      (kritische Situationen: Zugriff auf den Puffer, Puffer leer, Puffer voll)
  - Philosophenproblem
    - ➤ ein Thread reserviert sich zuerst Zugriff auf Datenbereich 1, dann auf Datenbereich 2, der andere Thread umgekehrt
      - ➥ kann zu Verklemmung führen

Gegenseitiger Ausschluss *(mutual exclusion)*

- Einfache Implementierung durch *mutex*-Variablen

```
mutex m = 1;
volatile int counter = 0;
```

Thread 1

```
...
lock(&m);
counter++;
unlock(&m);
```

Thread 2

```
...
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
```

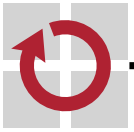
  - ➤ nur der Thread, der das *lock* gemacht hat, darf das *unlock* aufrufen!

- Realisierung (konzeptionell)

```
void lock (mutex *m) {
    while (*m == 0) {
        /* warten */
    }
    m = 0;
}
```

atomare Funktionen

```
void unlock (mutex *m) {
    *m = 1;
    /* ggf. wartende
       Threads wecken */
}
```

Zählende Semaphore

- Ein Semaphor (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach *Dijkstra*)
  - P-Operation *(proberen; passeren; wait; down)*
    - ➤ wartet bis Zugang frei

```
void P( int *s ) {
    while( *s <= 0 ) {
        /* warten */
    };
    *s= *s-1;
}
```

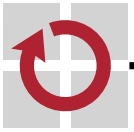
atomare Funktion

  - V-Operation *(verhogen; vrijgeven; signal; up)*
    - ➤ macht Zugang für anderen Thread / Prozess frei

```
void V( int *s ) {
    *s= *s+1;
    /* ggf wartende Threads/Prozesse wecken */
}
```

atomare Funktion

    - ➤ P und V müssen nicht vom selben Thread/Prozess aufgerufen werden!



Gegenseitiges Warten

- Implementierung mit Hilfe eines Semaphors

```
int barrier = 0;
int result;
```

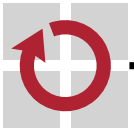
Thread 1

```
...
P(&barrier);
f1(result);
...
```

Thread 2

```
...
result = f2(...);
V(&barrier);
...
```

- ➤ Thread 2 läuft immer ungehindert durch
- ➤ Thread 1 blockiert an P, falls Thread 2 die V-Operation noch nicht ausgeführt hat (und wartet auf die V-Operation) – andernfalls läuft Thread 1 auch durch



Mutex im Detail: spin lock vs. sleeping lock

- spin lock
  - aktives Warten bis Mutex-Variable frei (= 1) wird
  - entspricht konzeptionell einem Pollen
  - Thread bleibt im Zustand *laufend*
- Problem: wenn nur ein Prozessor verfügbar ist, wird Rechenzeit vergeudet bis durch den Scheduler eine Umschaltung erfolgt
  - nur ein anderer, laufender Thread kann den Mutex frei geben
- sleeping lock
  - passives Warten
  - Thread geht in den Zustand *blockiert* (Schlafen, bis ein Ereignis eintrifft)
  - im Rahmen von unlock() wird der blockierte Thread in den Zustand *bereit* zurückgeführt
- Problem: bei sehr kurzen kritischen Abschnitten ist der Aufwand für das Blockieren/Aufwecken und die Umschaltungen unverhältnismäßig teuer

Implementierung von spin locks

- zentrales Problem: Atomarität von mutex-Abfrage und -Setzen

```
void lock (mutex *m) {
    while (*m == 0) { ;}
    m = 0;
}
```

- Lösung: spezielle Maschinenbefehle, die atomar eine Abfrage und Modifikation auf einer Hauptspeicherzelle ermöglichen
  - *Test-and-Set*, *Compare-and-Swap*, *Load-link/store-conditional, ...*
- Beispiel: *Test-and-set* – atomarer Maschinenbefehl mit folgender Wirkung
  - wenn zwei Threads den Befehl gleichzeitig ausführen wollen, sorgt die Hardware dafür, dass ein Thread den Befehl vollständig zuerst ausführt

```
bool test_and_set(bool *plock) {
    bool initial= *plock;
    *plock= TRUE;
    return initial;
}
```

- Ausgangssituation: **`*plock == FALSE`**
- Ergebnis von **`test_and_set`**:
  der Thread, der den Befehl zuerst ausführt, erhält **`FALSE`**,
  der andere **`TRUE`**

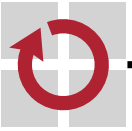
Implementierung von spin locks (2)

- Realisierung von mutex-Operationen mit dem *Test-and-Set*-Befehl

```
mutex m = FALSE;
```

```
void lock (mutex *m) {
  while(test_and_set(&m) ){ ; }
  /* got lock */
}
```

```
void unlock (mutex *m) {
     *m = FALSE;
}
```

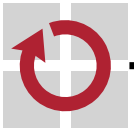


Implementierung von sleeping locks

- zwei Probleme
  - ➤ Konflikt mit einer zweiten lock-Operation: Atomarität von mutex-Abfrage und -Setzen
  - ➤ Konflikt mit einem unlock: *lost-wakeup*-Problem

```
void lock (mutex *m) {
    while (*m == 0) { sleep();}
    m = 0;
}
```

- Ursachen
  - (1) Prozessumschaltung während der lock-Operation
  - (2) Bei Multiprozessoren:
    gleichzeitige Ausführung von lock auf einem anderen Prozessor



Implementierung von sleeping locks (2)

- Behebung von Ursache (1): Prozessumschaltungen verhindern
  - Prozessumschaltung ist ein Funktion des Betriebssystem-Kerns
    - ➤ erfolgt im Rahmen eines Systemaufrufs
    - ➤ oder im Rahmen einer Interrupt-Behandlung

➥ lock/unlock werden ebenfalls im BS-Kern implementiert, BS-Kern läuft unter Interrupt-Sperre

```
void lock (mutex *m) {
   enter_OS(); cli();
   while (*m == 0) {
      block_thread_and_schedule();
   }
   m = 0;
   sei(); leave_OS();
}
```

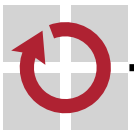
```
void unlock (mutex *m) {
   enter_OS(); cli();
   *m = 1;
   wakeup_waiting_theads();
   sei(); leave_OS();
}
```


Implementierung von sleeping locks (3)

- Behebung von Ursache (2): Parallele Ausführung auf anderem Prozessor verhindern
  - Problem (1) (Prozessumschaltungen) bleibt gleichzeitig bestehen
  - Gegenseitiger Ausschluss mit anderen Prozessoren durch spin locks

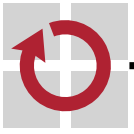
```
void lock (mutex *m) {
   enter_OS(); cli();
   spin_lock();
   while (*m == 0) {
      block_thread_and_schedule();
   }
   m = 0;
   spin_unlock();
   sei(); leave_OS();
}
```

```
void unlock (mutex *m) {
   enter_OS(); cli();
   spin_lock();
   *m = 1;
   wakeup_waiting_theads();
   spin_unlock();
   sei(); leave_OS();
}
```

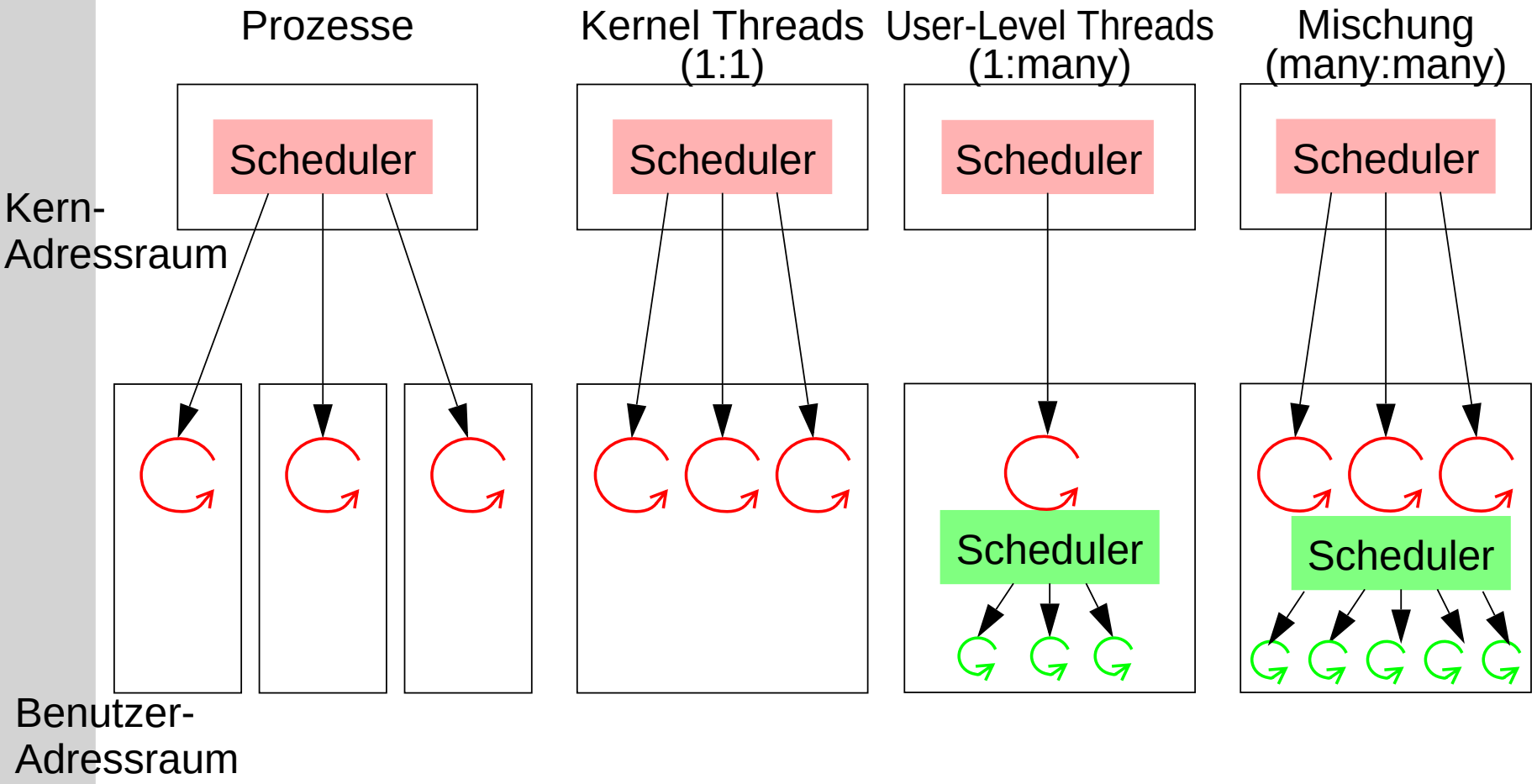



Thread-Konzepte in UNIX/Linux

- verschiedene Implementierungen von Thread-Paketen verfügbar
  - reine User-Level-Threads
    eine beliebige Zahl von User-Level-Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
  - reine Kernel-Level-Threads
    jedem auf User-Level sichtbaren Thread ist 1:1 ein Kernel-Level-Thread zugeordnet (*1:1*)
  - Mischungen: eine große Zahl von User-Level Threads wird auf eine kleinere Zahl von Kernel Threads abgebildet (*many:many*)
    + User-Level Threads sind billig
    + die Kernel Threads ermöglichen echte Parallelität auf einem Multiprozessor
    + wenn sich ein User-Level-Thread blockiert, dann ist mit ihm der Kernel-Level-Thread blockiert in dem er gerade abgewickelt wird — aber andere Kernel-Level-Threads können verwendet werden um andere, lauffähige User-Level-Threads weiter auszuführen



Thread-Konzepte in UNIX/Linux (2)



- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE-POSIX-Standard P1003.4a

pthread-Benutzerschnittstelle

- **Pthreads-Schnittstelle (Basisfunktionen)**:

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

- Funktionen in Pthreads-Bibliothek zusammengefasst

```
gcc ... -pthread
```


pthread-Benutzerschnittstelle (2)

- Thread-Erzeugung

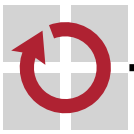
```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg)
```

`thread` Thread-Id

`attr` Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **`NULL`** für Standardattribute.

Thread wird erzeugt und ruft Funktion **`start_routine`** mit Parameter **`arg`** auf.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.



pthread-Benutzerschnittstelle (3)

- Thread beenden (bei return aus **`start_routine`** oder):

```
void pthread_exit(void *retval)
```

Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe pthread_join)

- Auf Thread warten und exit-Status abfragen:

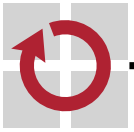
```
int pthread_join(pthread_t thread, void **retvalp)
```

Wartet auf den Thread mit der Thread-ID **thread** und liefert dessen Rückgabewert über **retvalp** zurück.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];

int main(int argc, char* argv[]) {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                                    (void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

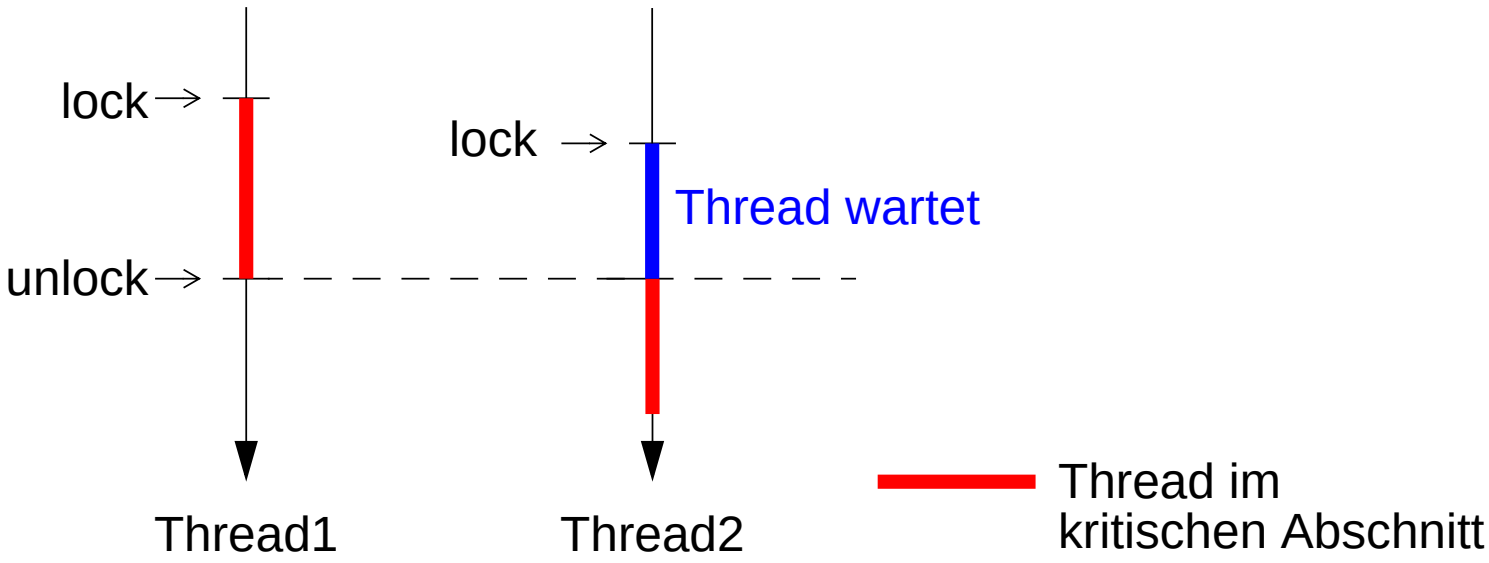
Pthreads-Koordinierung

- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
  - Implementierung durch den Systemkern
  - komplexe Datenstrukturen, aufwändig zu programmieren
  - für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache ***Mutex***-Variablen
  - gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

Pthreads-Koordinierung (2)

Mutexes

- Koordinierung von kritischen Abschnitten



Pthreads-Koordinierung (3)

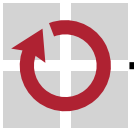
… Mutexes (2)

- Schnittstelle
  - Mutex erzeugen

```
pthread_mutex_t m1;
pthread_mutex_init(&m1, NULL);
```

  - Lock & unlock

```
pthread_mutex_lock(&m1);
... kritischer Abschnitt
pthread_mutex_unlock(&m1);
```

Pthreads-Koordinierung (4)

… Mutexes (3)

- Komplexere Koordinierungsprobleme können alleine mit Mutexes nicht implementiert werden

➥ Problem:
- Ein Mutex sperrt die eine komplexere Datenstruktur
- Der Zustand der Datenstruktur erlaubt die Operation nicht
- Thread muss warten, bis die Situation durch anderen Thread behoben wurde
- Blockieren des Threads an einem weiteren Mutex kann zu Verklemmungen führen

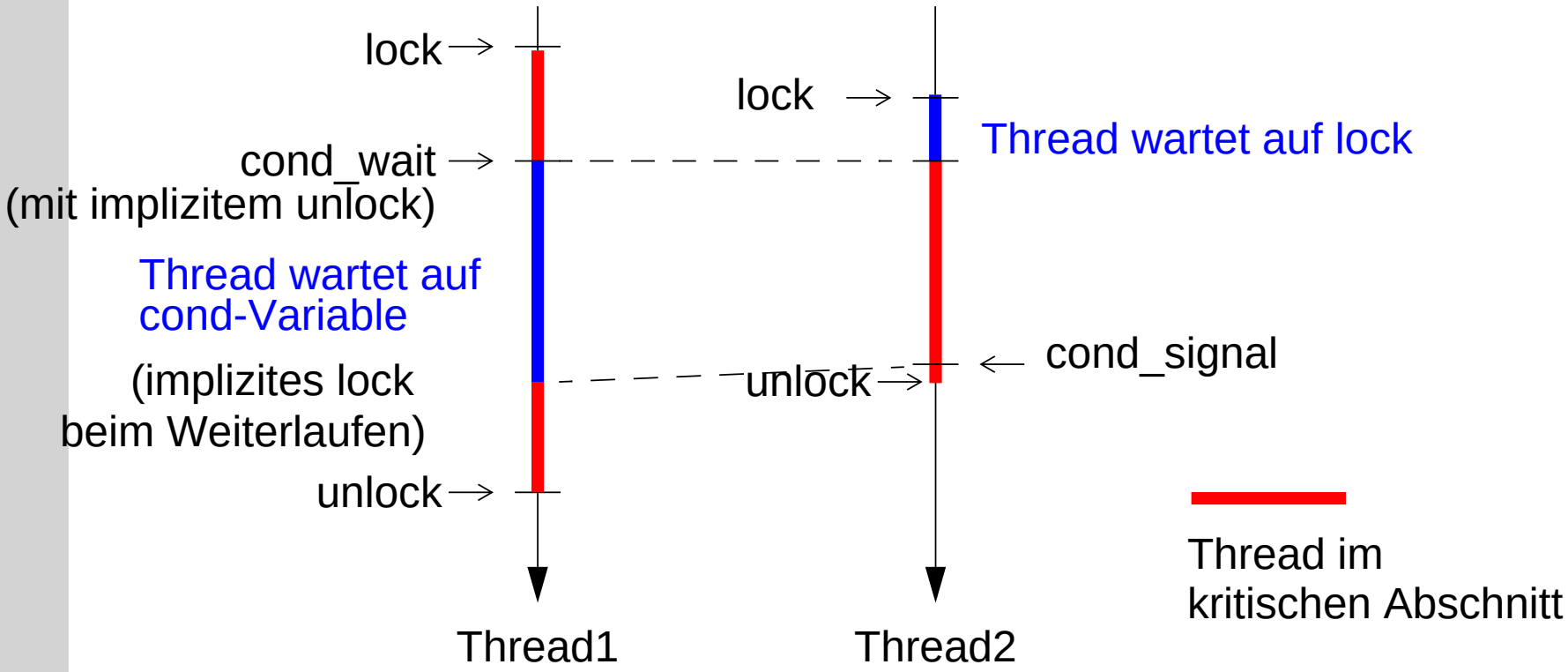
➥ Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus

➥ **Condition Variables**

Pthreads-Koordinierung (5)

★ Condition Variables

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



Pthreads-Koordinierung (6)

… Condition Variables (2)

- Realisierung
  - Thread reiht sich in Warteschlange der Condition Variablen ein
  - Thread gibt Mutex frei
  - Thread gibt Prozessor auf
  - Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
  - Deblockierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
  - Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden

Pthreads-Koordinierung (7)

… Condition Variables (3)

- Schnittstelle
  - Condition Variable erzeugen

```
pthread_cond_t c1;
pthread_cond_init(&c1,   NULL);
```

  - Beispiel: zählende Semaphore

P-Operation

```
void P(int *sem) {
  pthread_mutex_lock(&m1);
    while ( *sem == 0 )
      pthread_cond_wait
           (&c1, &m1);
    (*sem)--;
  pthread_mutex_unlock(&m1);
}
```

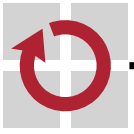
V-Operation

```
void V(int *sem) {
  pthread_mutex_lock(&m1);
    (*sem)++;
   pthread_cond_broadcast(&c1);
  pthread_mutex_unlock(&m1);
}
```


Pthreads-Koordinierung (8)

… Condition Variables (4)

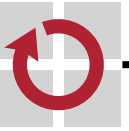
- Bei **`pthread_cond_signal`** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher
  - ➤ evtl. Prioritätsverletzung wenn nicht der höchstpriore gewählt wird
  - ➤ Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben
- Mit **`pthread_cond_broadcast`** werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert



Threads und Koordinierung in Java

- Thread-Konzept und Koordinierungsmechanismen sind in Java integriert
- Erzeugung von Threads über Thread-Klassen
  - Beispiel

```
class MyClass implements Runnable {
        public void run()  {
                System.out.println("Hello\n");
        }
}

....
MyClass o1 = new MyClass();     // create object
Thread t1 = new Thread(o1);     // create thread to run in o1

t1.start();                     // start thread

Thread t2 = new Thread(o1);     // create second thread to run in o1

t2.start();                     // start second thread
```



Threads und Koordinierung in Java (2)

★ Koordinierungsmechanismen

- Monitore: exklusive Ausführung von Methoden eines Objekts
  - es darf nur jeweils ein Thread die **`synchronized`**-Methoden betreten
  - Beispiel:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

- Conditions: gezieltes Freigeben des Monitors und Warten auf ein Ereignis