

# Systemnahe Programmierung in C (SPiC)

**Daniel Lohmann, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2012

<http://www4.cs.fau.de/Lehre/SS12/V-SPiC>



- [1] *ATmega32 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. 8155-AVR-07/09. Atmel Corporation. July 2009.
- [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.
- [GDI] Elmar Nöth, Peter Wilke, and Stefan Steidl. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2010 (jährlich). URL: <http://www5.informatik.uni-erlangen.de/lectures/ws-1011/grundlagen-der-informatik-gdi/folien/>.



- [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.
  
- [GDI-Ü] Stefan Steidl, Marcus Prümmer, and Markus Mayer. *Grundlagen der Informatik*. Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2010 (jährlich). URL: <http://www5.informatik.uni-erlangen.de/lectures/ws-1011/grundlagen-der-informatik-gdi/uebung/>.
  
- [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/0EG20021217S0039>, visited 2011-04-08.



# Veranstaltungsüberblick

## Teil A: Konzept und Organisation

1 Einführung

2 Organisation

## Teil B: Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

## Teil C: Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14  $\mu$ C-Systemarchitektur

## Teil D: Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

20 Speicherorganisation

21 Prozesse II

22 Betriebssysteme II



# Systemnahe Programmierung in C (SPiC)

## Teil A Konzept und Organisation

**Daniel Lohmann, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2012

[http://www4.cs.fau.de/Lehre/SS12/V\\_SPiC](http://www4.cs.fau.de/Lehre/SS12/V_SPiC)



# Überblick: Teil A Konzept und Organisation

## 1 Einführung

- 1.1 Ziele der Lehrveranstaltung
- 1.2 Warum  $\mu$ -Controller?
- 1.3 Warum C?
- 1.4 Literatur

## 2 Organisation

- 2.1 Vorlesung
- 2.2 Übung
- 2.3 Lötabend
- 2.4 Prüfung
- 2.5 Semesterüberblick

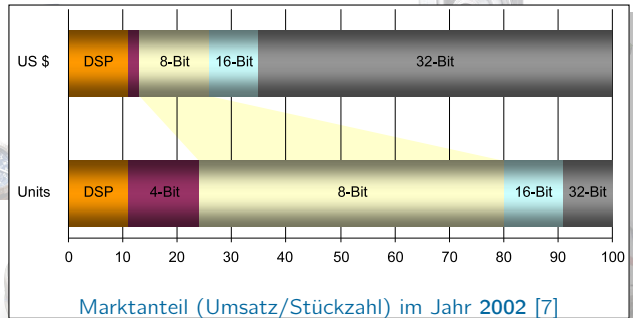


- **Vertiefen** des Wissens über Konzepte und Techniken der Informatik für die Softwareentwicklung
  - Ausgangspunkt: Grundlagen der Informatik (GdI)
  - Schwerpunkt: Systemnahe Softwareentwicklung in C
- **Entwickeln** von Software in C für einen  $\mu$ -Controller ( $\mu$ C) und eine Betriebssystem-Plattform (Linux)
  - SPiCboard-Lehrentwicklungsplattform mit ATmega- $\mu$ C
  - **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, ...)



# 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  $\mu$ -Controller, **8-Bit oder kleiner** [6, 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  $\mu$ -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 **um die 10 Jahre** alt!
- Man kann dennoch davon ausgehen, dass die **relativen Größenordnungen** nach wie vor stimmen
  - 2012 liegt der Anteil an 8-Bitern (vermutlich) noch weit über 50 Prozent
  - 4-Bitter dürften inzwischen jedoch weitgehend ausgestorben sein



# Motivation: Die ATmega- $\mu$ C-Familie (8-Bit)

| Type       | Flash   | SRAM   | IO | Timer 8/16 | UART | I <sup>2</sup> C | AD   | Price (€) |
|------------|---------|--------|----|------------|------|------------------|------|-----------|
| ATTINY11   | 1 KiB   |        | 6  | 1/-        | -    | -                | -    | 0.31      |
| ATTINY13   | 1 KiB   | 64 B   | 6  | 1/-        | -    | -                | 4*10 | 0.66      |
| ATTINY2313 | 2 KiB   | 128 B  | 18 | 1/1        | 1    | 1                | -    | 1.06      |
| ATMEGA4820 | 4 KiB   | 512 B  | 23 | 2/1        | 2    | 1                | 6*10 | 1.26      |
| ATMEGA8515 | 8 KiB   | 512 B  | 35 | 1/1        | 1    | -                | -    | 2.04      |
| ATMEGA8535 | 8 KiB   | 512 B  | 32 | 2/1        | 1    | 1                | -    | 2.67      |
| ATMEGA169  | 16 KiB  | 1024 B | 54 | 2/1        | 1    | 1                | 8*10 | 4.03      |
| ATMEGA64   | 64 KiB  | 4096 B | 53 | 2/2        | 2    | 1                | 8*10 | 5.60      |
| ATMEGA128  | 128 KiB | 4096 B | 53 | 2/2        | 2    | 1                | 8*10 | 7.91      |

ATmega-Varianten (Auswahl) und Großhandelspreise (DigiKey 2006)

- 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!



- **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
    - $\mu$ -Controller-Programmierung
    - Softwareentwicklung für die Linux-Systemschnittstelle
  - Gegensatz  $\mu$ C-Umgebung  $\leftrightarrow$  Betriebssystemplattform erfahren
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (ME)



- Das Handout der Vorlesungsfolien wird online und als 4 × 1-Ausdruck auf Papier zur Verfügung gestellt
  - Ausdrücke 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!**



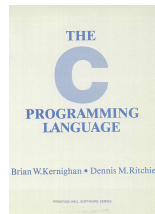
## [2] Für den Einstieg empfohlen:

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



## ■ 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- $\mu$ C)
  - Abbildung Speicher  $\leftrightarrow$  Sprachkonstrukte
  - Unterbrechungen (*interrupts*) und Nebenläufigkeit
- Softwareentwicklung auf „einem Betriebssystem“ (Linux)
  - Betriebssystem als Ausführungsumgebung für Programme
  - Abstraktionen und Dienste eines Betriebssystems

## ■ Termin: Fr 12:15–13:43, H4

- Einzeltermin am 19. April (Do), 10:15–11:45, H10
- insgesamt 15 Vorlesungstermine

 2-7



## ■ Tafel- und Rechnerübungen

### ■ Tafelübungen

- Ausgabe und Erläuterung der Programmieraufgaben
- Gemeinsame Entwicklung einer Lösungsskizze
- Besprechung der Lösungen

### ■ Rechnerübungen

- selbstständige Programmierung
- Umgang mit Entwicklungswerkzeug (AVR Studio)
- Betreuung durch Übungsbetreuer

## ■ Termin: Initial 6 Gruppen zur Auswahl

- Anmeldung über Waffel (siehe Webseite): Heute, 12:00 – So, 20:00
- 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

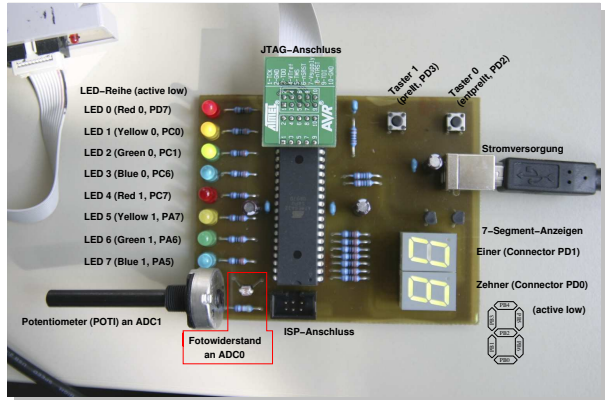
- Praktische Umsetzung des Vorlesungsstoffs
  - Acht Programmieraufgaben ↪ 2-7
  - Bearbeitung wechselseitig alleine / mit Übungspartner
- Lösungen mit Abgabeskript am Rechner abgeben
  - Lösung wird durch Skripte überprüft
  - Wir korrigieren und bepunkteten die Abgaben und geben sie zurück
  - Eine Lösung wird vom Teilnehmer an der Tafel erläutert (impliziert Anwesenheit!)
- ★ 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- $\mu$ C
- JTAG-Anschluss
- 8 LEDs
- 2 7-Seg-Elemente
- 2 Taster
- 1 Potentiometer
- 1 Fotosensor



- Ausleihe zur Übungsbearbeitung möglich
- Oder noch besser  $\hookrightarrow$  selber Löten

- Die Fachschaften (EEI / ME) bieten fünf „Lötabende“ an
  - Teilnahme ist freiwillig
  - (Erste) Lötterfahrung sammeln beim Löten eines eigenen SPiCboards
- **Termine:** 23. – 27. April, jeweils 18:00–21:00
- **Anmeldung:** über Waffel (siehe Webseite)
- **Kostenbeitrag:** SPiCBoard: ~~13~~ *13 EUR bezahlt aus Studienbeiträgen*  
Progger: 22 EUR (falls gewünscht)

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

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



- 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
    - $\leadsto$  100% der möglichen ÜP  $\mapsto$  +10% der möglichen KP





## Semesterplanung

| KW | Mo             | Di         | Mi          | Do     | Fr     | Themen                                                                          |
|----|----------------|------------|-------------|--------|--------|---------------------------------------------------------------------------------|
| 16 | 16.04.         | 17.04.     | 18.04.      | 19.04. | 20.04. | Einführung, Organisation, Java nach C, Abstraktion, Sprachüberblick, Datentypen |
|    |                |            |             | VL1    | VL2    |                                                                                 |
| 17 | 23.04.         | 24.04.     | 25.04.      | 26.04. | 27.04. | Variablen, Ausdrücke, Kontrollstrukturen, Funktionen, Makros                    |
|    |                | A1 (Blink) |             |        | VL3    |                                                                                 |
| 18 | 30.04.         | 01.05.     | 02.05.      | 03.05. | 04.05. | Mikrocontroller-Systemarchitektur, -Softwareentwicklung                         |
|    |                | T.d.A.     | A2 (Snake)  |        | VL4    |                                                                                 |
| 19 | 07.05.         | 08.05.     | 09.05.      | 10.05. | 11.05. | Programmstruktur, Module, komplexe Datentypen                                   |
|    |                |            | A3 (Spiel)  |        | VL5    |                                                                                 |
| 20 | 14.05.         | 15.05.     | 16.05.      | 17.05. | 18.05. | Zeiger und Felder                                                               |
|    |                |            | Himmelfahrt |        | VL 6   |                                                                                 |
| 21 | 21.05.         | 22.05.     | 23.05.      | 24.05. | 25.05. | Nebenläufigkeit                                                                 |
|    | A4(LED)        |            |             |        | VL 7   |                                                                                 |
| 22 | 28.05.         | 29.05.     | 30.05.      | 31.05. | 01.06. | Speicherorganisation                                                            |
|    | Pfingsten/Berg |            | A5 (Ampel)  |        | VL 8   |                                                                                 |
| 23 | 04.06.         | 05.06.     | 06.06.      | 07.06. | 08.06. | Dateisysteme                                                                    |
|    |                |            | Fronleichn. |        | VL 9   |                                                                                 |
| 24 | 11.06.         | 12.06.     | 13.06.      | 14.06. | 15.06. | Prozesse                                                                        |
|    | A6 (PrintDir)  |            |             |        | VL 10  |                                                                                 |
| 25 | 18.06.         | 19.06.     | 20.06.      | 21.06. | 22.06. | Signale                                                                         |
|    | A7 (fish)      |            |             |        | VL 11  |                                                                                 |
| 26 | 25.06.         | 26.06.     | 27.06.      | 28.06. | 29.06. | Threads, Koordinierung                                                          |
|    | A8 (tqsh)      |            |             |        | VL 12  |                                                                                 |
| 27 | 02.07.         | 03.07.     | 04.07.      | 05.07. | 06.07. | pthreads, Threads in Java                                                       |
|    |                |            |             |        | VL 13  |                                                                                 |
| 28 | 09.07.         | 10.07.     | 11.07.      | 12.07. | 13.07. | Wdh                                                                             |
|    | Wdh.           |            |             |        | VL 14  |                                                                                 |
| 29 | 16.07.         | 17.07.     | 18.07.      | 19.07. | 20.07. | Wdh                                                                             |
|    | Wdh.           |            |             |        | VL 15  |                                                                                 |

## Dozenten Vorlesung



Daniel Lohmann



Jürgen Kleinöder

## Organisatoren des Übungsbetriebs



Wanja Hofer



Moritz Strübe



Rainer Müller

## Techniker (Ausleihe SPiCboard und Debugger)



Harald Junggunst



Christian Preller



Daniel Christiani



## Übungsleiter



Martin Klüpfel



Sebastian  
Schinabeck



Sebastian Ehrenfels



# Systemnahe Programmierung in C (SPiC)

## Teil B Einführung in C

**Daniel Lohmann, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2012

[http://www4.cs.fau.de/Lehre/SS12/V\\_SPiC](http://www4.cs.fau.de/Lehre/SS12/V_SPiC)



# Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor



# Das erste C-Programm

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

```
#include <stdio.h>

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

- Übersetzen und Ausführen (auf einem UNIX-System)

```
lohmann@latte:~/src$ gcc -o hello hello-linux.c
lohmann@latte:~/src$ ./hello
Hello World!
lohmann@latte:~/src$
```

Gar nicht so  
schwer :-)



# Das erste C-Programm – Vergleich mit Java

## ■ Das berühmteste Programm der Welt in **C**

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     // greet user
5     printf("Hello World!\n");
6     return 0;
7 }
```

## ■ Das berühmteste Programm der Welt in **Java**

```
1 import java.lang.System;
2 class Hello {
3     public static void main(String[] args) {
4         /* greet user */
5         System.out.println("Hello World!");
6         return;
7     }
8 }
```



## ■ 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, IV-191](#)]
- 6 Rückkehr zum Betriebssystem.



# Das erste C-Programm für einen $\mu$ -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){
    }
}
```

$\mu$ -Controller-Programmierung  
ist „irgendwie anders“.

- Übersetzen und **Flashen** (mit AVR Studio) ~ Übung
- Ausführen (SPiCboard):  (rote LED leuchtet)



# Das erste C-Programm für einen $\mu$ -Controller

- „Hello World“ für AVR ATmega (vgl. [↪ 3-1](#))

```
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 }
```



- $\mu$ -Controller-Programm zeilenweise erläutert  
(Beachte Unterschiede zur Linux-Version  $\hookrightarrow$  3-3)
  - 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  $\mu$ -Controller-Programm läuft **endlos**  $\rightsquigarrow$  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.



- 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.



- 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 }
```



- Benutzerinteraktion mit SPiCboard zeilenweise erläutert
  - 5 Wie die LED ist der Taster mit einem **digitalen IO-Pin** des  $\mu$ -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  $\rightsquigarrow$  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  $\rightsquigarrow$  Bit 2 im Register **PIND** wird *low* und die Schleife verlassen.



# Zum Vergleich: Benutzerinteraktion als Java-Programm

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

```
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 }
```



- 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, ...
  - **Problemnahe** 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: $\mu$ -Controller-Programmierung

C  $\mapsto$  Maschinennähe  $\mapsto$   $\mu$ C-Programmierung

Die **Maschinennähe** von C zeigt sich insbesondere auch bei der  $\mu$ -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  $\leadsto$  fehleranfällig, aufwändig

**Ansatz:** Mehr Abstraktion durch **problemorientierte Bibliotheken**

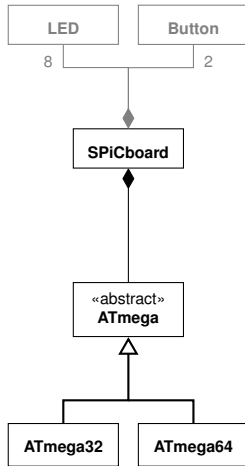


# Abstraktion durch Softwareschichten: SPiCboard

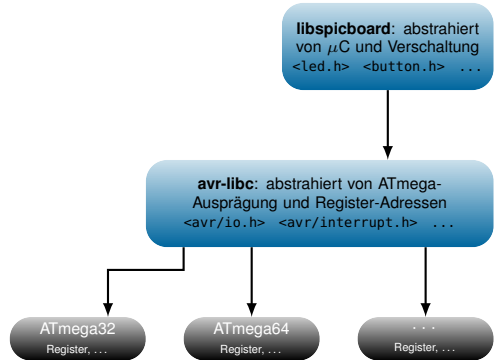
Problemnähe

Maschinennähe

## Hardwareseite



## Softwareschichten



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

Problemnähe ↑

↓ Maschinennähe

Programm läuft nur auf dem **SPiCboard**. Es verwendet Funktionen (wie `sb_led_on()`) und Konstanten (wie `RED0`) der **lib-spicboard**, welche die konkrete Verschaltung von LEDs, Tastern, usw. mit dem  $\mu\text{C}$  repräsentieren:

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

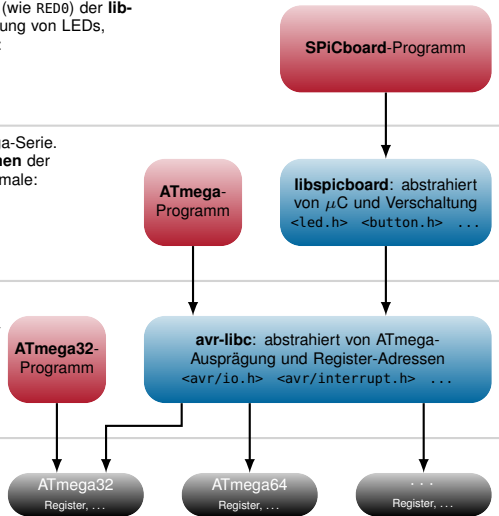
Programm läuft auf **jedem**  $\mu\text{C}$  der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

```
#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. ↪ 3-8)

## Nun: Entwicklung mit libspicboard

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

void main() {

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

    // greet user
    sb_led_on(LED0);

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

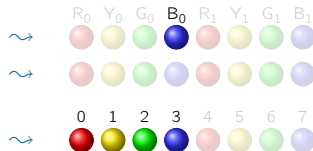
- Hardwareinitialisierung entfällt
- Programm ist einfacher und verständlicher durch **problemspezifische Abstraktionen**
  - Setze Bit 7 in PORTD  
    ↪ `sb_set_led(LED0)`
  - Lese Bit 2 in PORTD  
    ↪ `sb_button_getState(BUTTON0)`



## ■ 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  $n \in \{-9 \dots 99\}$  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...1023}

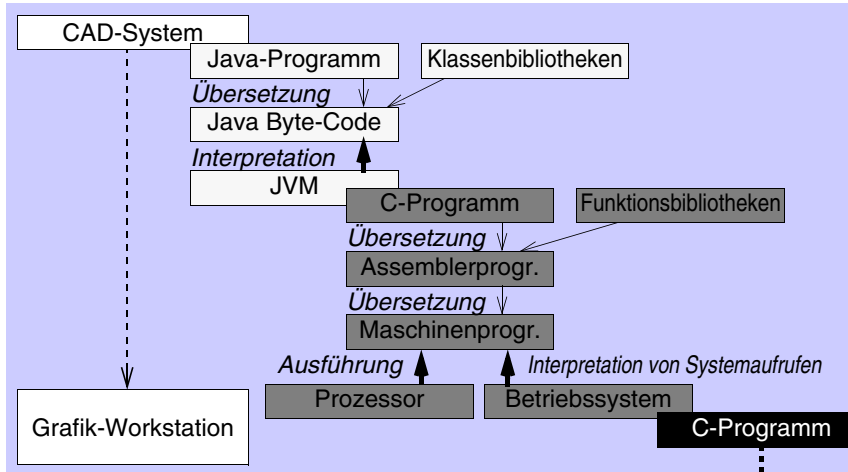


# Softwareschichten im Allgemeinen

Problemnähe

Maschinennähe

**Diskrepanz:** Anwendungsproblem  $\longleftrightarrow$  Abläufe auf der Hardware



**Ziel:** Ausführbarer Maschinencode

- **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



- **Systemsicht:** Softwareschicht zum Multiplexen der Hardware (↔ Mehrbenutzerbetrieb)
  - Parallele Abarbeitung von Programminstanzen durch **Prozesskonzept**
    - Virtueller Speicher ↔ eigener 32-/64-Bit-Adressraum
    - Virtueller Prozessor ↔ wird transparent zugeteilt und entzogen
    - Virtuelle Ein-/Ausgabe-Geräte ↔ 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 ~> **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

→ 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:



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



# Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

**5 Sprachüberblick**

**6 Einfache Datentypen**

**7 Operatoren und Ausdrücke**

**8 Kontrollstrukturen**

**9 Funktionen**

**10 Variablen**

**11 Präprozessor**



# 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 // main function
20 ... main(...) {
21     ...
22 }
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
15 // subfunction 2
16 void wait() {
17     volatile unsigned int i;
18     for (i=0; i<0xffff; i++)
19         ;
20 }
21
22 // main function
23 void main() {
24     while (lightLED() < 8) {
25         wait();
26     }
27 }
```

## ■ 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



```
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
15 // subfunction 2
16 void wait() {
17     volatile unsigned int i;
18     for (i=0; i<0xffff; i++)
19         ;
20 }
21
22 // main function
23 void main() {
24     while (lightLED() < 8) {
25         wait();
26     }
27 }
```

- 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



```
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
15 // subfunction 2
16 void wait() {
17     volatile unsigned int i;
18     for (i=0; i<0xffff; i++)
19         ;
20 }
21
22 // main function
23 void main() {
24     while (lightLED() < 8) {
25         wait();
26     }
27 }
```

## ■ Reservierte Wörter der Sprache (~ dürfen nicht als Bezeichner verwendet werden)

- Eingebaute (*primitive*) Datentypen unsigned int, void
- Typmodifizierer volatile
- Kontrollstrukturen for, while
- Elementaranweisungen return



- 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`



```

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() {
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



```
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() {
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



```
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
15 // subfunction 2
16 void wait() {
17     volatile unsigned int i;
18     for (i=0; i<0xffff; i++)
19         ;
20 }
21
22 // main function
23 void main() {
24     while (lightLED() < 8) {
25         wait();
26     }
27 }
```

## ■ 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

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

**6 Einfache Datentypen**

**7 Operatoren und Ausdrücke**

**8 Kontrollstrukturen**

**9 Funktionen**

**10 Variablen**

**11 Präprozessor**



## ■ **Datentyp** := (*<Menge von Werten>*, *<Menge von Operationen>*)

- **Literal** Wert im Quelltext
- **Konstante** Bezeichner für einen Wert
- **Variable** Bezeichner für Speicherplatz, der einen Wert aufnehmen kann
- **Funktion** Bezeichner für Sequenz von Anweisungen, die einen Wert zurückgibt

↔ 5-6

↪ 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: ∅
- Boolescher Datentyp `_Bool` (C99)
  - Wertebereich: {0, 1} (↔ letztlich ein Integertyp)
  - Bedingungsausdrücke (z. B. `if(...)`) sind in C vom Typ `int`! [≠Java]



| Integertyp                     | Verwendung                                    | Literalformen        |
|--------------------------------|-----------------------------------------------|----------------------|
| ■ <code>char</code>            | kleine Ganzzahl oder Zeichen                  | 'A', 65, 0x41, 0101  |
| ■ <code>short [int]</code>     | Ganzzahl ( <code>int</code> ist optional)     | s. o.                |
| ■ <code>int</code>             | Ganzzahl „natürlicher Größe“                  | s. o.                |
| ■ <code>long [int]</code>      | große Ganzzahl                                | 65L, 0x41L, 0101L    |
| ■ <code>long long [int]</code> | sehr große Ganzzahl                           | 65LL, 0x41LL, 0101LL |
| ■ Typ-Modifizierer             | werden vorangestellt                          | Literal-Suffix       |
| ■ <code>signed</code>          | Typ ist vorzeichenbehaftet (Normalfall)       | -                    |
| ■ <code>unsigned</code>        | Typ ist vorzeichenlos                         | U                    |
| ■ <code>const</code>           | 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
```



- Die interne Darstellung (Bitbreite) ist **implementierungsabhängig**

|           | Datentyp-Breite in Bit |            |          |          |         |
|-----------|------------------------|------------|----------|----------|---------|
|           | Java                   | C-Standard | gcc/IA32 | gcc/IA64 | gcc/AVR |
| char      | 16                     | $\geq 8$   | 8        | 8        | 8       |
| short     | 16                     | $\geq 16$  | 16       | 16       | 16      |
| int       | 32                     | $\geq 16$  | 32       | 32       | 16      |
| long      | 32                     | $\geq 32$  | 32       | 64       | 32      |
| long long | -                      | $\geq 64$  | 64       | 64       | 64      |

- Der Wertebereich berechnet sich aus der Bitbreite

|            |                   |   |                   |
|------------|-------------------|---|-------------------|
| ■ signed   | $-(2^{Bits-1}-1)$ | → | $+(2^{Bits-1}-1)$ |
| ■ unsigned | 0                 | → | $+(2^{Bits}-1)$   |

Hier zeigt sich die C-Philosophie: Effizienz durch **Maschinennähe** → 3-14

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



# Integertypen: Maschinennähe $\rightarrow$ Problemnähe

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

Wertebereich `stdint.h`-Typen

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



- 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)
typedef unsigned char uint8_t;
typedef unsigned int  uint16_t;
...
```

```
// stdint.h (x86-gcc, IA32)
typedef unsigned char uint8_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 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  $\mu$ C-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!



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

```
enum Bezeichneropt { 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
```



- 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!
```

- $\rightsquigarrow$  Es findet **keinerlei Typprüfung** statt!

Das entspricht der

**C-Philosophie!**  $\hookrightarrow$  3-14



- Fließkommatyp      Verwendung      Literalformen
  - **float**      einfache Genauigkeit ( $\approx 7$  St.)      100.0F, 1.0E2F
  - **double**      doppelte Genauigkeit ( $\approx 15$  St.)      100.0, 1.0E2
  - **long double**      „erweiterte Genauigkeit“      100.0L 1.0E2L
- Genauigkeit / Wertebereich sind **implementierungsabhängig** [ $\neq$  Java]
  - Es gilt: **float**  $\leq$  **double**  $\leq$  **long double**
  - **long double** und **double** sind auf vielen Plattformen identisch

„Effizienz durch Maschinennähe“  $\hookrightarrow$  3-14

### Fließkommazahlen + $\mu$ C-Plattform = \$\$\$

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

**Regel:** Bei der  $\mu$ -Controller-Programmierung ist auf Fließkommaarithmetik **zu verzichten!**



- 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  $\leadsto$  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 → *American Standard Code for Information Interchange*

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



- 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:

"Hi!" → 

|     |     |     |   |
|-----|-----|-----|---|
| 'H' | 'i' | '!' | 0 |
|-----|-----|-----|---|

 ← abschließendes 0-Byte

- 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  $\mu$ C-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 [ $\approx$ Java]

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

- Zeiger  $\hookrightarrow$  veränderbare Referenzen auf Variablen [ $\neq$ 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 [ $\neq$ 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

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

**7 Operatoren und Ausdrücke**

**8 Kontrollstrukturen**

**9 Funktionen**

**10 Variablen**

**11 Präprozessor**



## ■ Stehen für alle Ganzzahl- und Fließkommatypen zur Verfügung

+ Addition

− Subtraktion

★ Multiplikation

/ Division

unäres − negatives Vorzeichen (z. B.  $-a$ )  $\rightsquigarrow$  Multiplikation mit  $-1$

unäres + positives Vorzeichen (z. B.  $+3$ )  $\rightsquigarrow$  kein Effekt

## ■ Zusätzlich nur für Ganzzahltypen:

% Modulo (Rest bei Division)



- 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
```



### ■ Vergleichen von zwei Ausdrücken

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

### ■ **Beachte:** Ergebnis ist vom Typ `int`

[≠ Java]

- Ergebnis: *falsch* ↦ 0  
*wahr* ↦ 1

- Man kann mit dem Ergebnis rechnen

### ■ Beispiele

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



## ■ Verknüpfung von Wahrheitswerten (wahr / falsch), kommutativ

|                   |                        |                                                                           |
|-------------------|------------------------|---------------------------------------------------------------------------|
| <b>&amp;&amp;</b> | „und“<br>(Konjunktion) | <i>wahr</i> <b>&amp;&amp;</b> <i>wahr</i> $\rightarrow$ <i>wahr</i>       |
|                   |                        | <i>wahr</i> <b>&amp;&amp;</b> <i>falsch</i> $\rightarrow$ <i>falsch</i>   |
|                   |                        | <i>falsch</i> <b>&amp;&amp;</b> <i>falsch</i> $\rightarrow$ <i>falsch</i> |

|           |                         |                                                                   |
|-----------|-------------------------|-------------------------------------------------------------------|
| <b>  </b> | „oder“<br>(Disjunktion) | <i>wahr</i> <b>  </b> <i>wahr</i> $\rightarrow$ <i>wahr</i>       |
|           |                         | <i>wahr</i> <b>  </b> <i>falsch</i> $\rightarrow$ <i>wahr</i>     |
|           |                         | <i>falsch</i> <b>  </b> <i>falsch</i> $\rightarrow$ <i>falsch</i> |

|          |                             |                                                  |
|----------|-----------------------------|--------------------------------------------------|
| <b>!</b> | „nicht“<br>(Negation, unär) | <b>!</b> <i>wahr</i> $\rightarrow$ <i>falsch</i> |
|          |                             | <b>!</b> <i>falsch</i> $\rightarrow$ <i>wahr</i> |

## ■ **Beachte:** Operanden und Ergebnis sind vom Typ **int** [ $\neq$ Java]

■ Operanden  
(Eingangsparameter):  $0 \mapsto \text{falsch}$   
 $\neq 0 \mapsto \text{wahr}$

■ Ergebnis:  
 $\text{falsch} \mapsto 0$   
 $\text{wahr} \mapsto 1$



- Die Auswertung eines logischen Ausdrucks wird **abgebrochen**, sobald das Ergebnis feststeht

■ Sei `int a = 5;` `int b = 3;` `int c = 7;`

$\underbrace{a > b}_{1} \parallel \underbrace{a > c}_{?} \leftarrow$  wird nicht ausgewertet, da der erste Term bereits *wahr* ergibt

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



- 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  $op \in \{ +, -, *, \%, \ll, \gg, \&, ^, | \}$
- Beispiele

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



# Zuweisungen sind Ausdrücke!

- Zuweisungen können in komplexere Ausdrücke 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!



## ■ Bitweise Verknüpfung von Ganzzahltypen, kommutativ

|   |                                       |                        |
|---|---------------------------------------|------------------------|
| & | bitweises „Und“<br>(Bit-Schnittmenge) | $1 \& 1 \rightarrow 1$ |
|   |                                       | $1 \& 0 \rightarrow 0$ |
|   |                                       | $0 \& 0 \rightarrow 0$ |

|  |                                             |                       |
|--|---------------------------------------------|-----------------------|
|  | bitweises „Oder“<br>(Bit-Vereinigungsmenge) | $1   1 \rightarrow 1$ |
|  |                                             | $1   0 \rightarrow 1$ |
|  |                                             | $0   0 \rightarrow 0$ |

|          |                                               |                            |
|----------|-----------------------------------------------|----------------------------|
| $\wedge$ | bitweises „Exklusiv-Oder“<br>(Bit-Antivalenz) | $1 \wedge 1 \rightarrow 0$ |
|          |                                               | $1 \wedge 0 \rightarrow 1$ |
|          |                                               | $0 \wedge 0 \rightarrow 0$ |

|   |                                               |                        |
|---|-----------------------------------------------|------------------------|
| ~ | bitweise Inversion<br>(Einerkomplement, unär) | $\sim 1 \rightarrow 0$ |
|   |                                               | $\sim 0 \rightarrow 1$ |



## ■ 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 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|---|---|---|---|---|---|---|---|
| PORTD | ? | ? | ? | ? | ? | ? | ? | ? |

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

|               |   |   |   |   |   |   |   |   |
|---------------|---|---|---|---|---|---|---|---|
| 0x80          | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PORTD  = 0x80 | 1 | ? | ? | ? | ? | ? | ? | ? |

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

|                |   |   |   |   |   |   |   |   |
|----------------|---|---|---|---|---|---|---|---|
| ~0x80          | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| PORTD &= ~0x80 | 0 | ? | ? | ? | ? | ? | ? | ? |

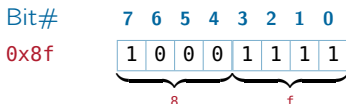
Löschen eines Bits durch **Ver-unden** mit Maske, in der nur das Zielbit 0 ist

|               |   |   |   |   |   |   |   |   |
|---------------|---|---|---|---|---|---|---|---|
| 0x08          | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| PORTD ^= 0x08 | ? | ? | ? | ? | ? | ? | ? | ? |

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



- 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) ;
}
```



- Formulierung von Bedingungen in Ausdrücken

$Ausdruck_1 \text{ ? } Ausdruck_2 \text{ : } Ausdruck_3$

- Zunächst wird  $Ausdruck_1$  ausgewertet
  - $Ausdruck_1 \neq 0$  (*wahr*)  $\leadsto$  Ergebnis ist  $Ausdruck_2$
  - $Ausdruck_1 = 0$  (*falsch*)  $\leadsto$  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;  
}
```



- 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)



|    | Klasse                                                                                                           | Operatoren                                             | Assoziativität |
|----|------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|----------------|
| 1  | Funktionsaufruf, Feldzugriff<br>Strukturzugriff<br>Post-Inkrement/-Dekrement                                     | x() x[]<br>x.y x->y<br>x++ x--                         | links → rechts |
| 2  | Prä-Inkrement/-Dekrement<br>unäre Operatoren<br>Adresse, Verweis (Zeiger)<br>Typkonvertierung (cast)<br>Typgröße | ++x --x<br>+x -x ~x !x<br>& *<br>(<Typ>)x<br>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 --> +2147483648

res = a * b / c; // promotion to int32_t
int8_t: 75      int32_t: 300
                └───┘
                int32_t: 75
```



- 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
```

Diagram illustrating the cast operation:

$\underbrace{\text{res}}_{\text{int: 1}} = s < \underbrace{(\text{int}) u}_{\text{int: 1}}$

The cast operation  $(\text{int}) u$  is shown with a bracket underneath it, indicating the result is of type `int`.



# Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

**8 Kontrollstrukturen**

**9 Funktionen**

**10 Variablen**

**11 Präprozessor**



## ■ if-Anweisung (bedingte Anweisung)

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

| Bedingung |      |
|-----------|------|
| ja        | nein |
| Anweisung |      |

## ■ if-else-Anweisung (einfache Verzweigung)

```
if ( Bedingung )
    Anweisung1;
else
    Anweisung2;
```

| Bedingung              |                        |
|------------------------|------------------------|
| ja                     | nein                   |
| Anweisung <sub>1</sub> | Anweisung <sub>2</sub> |

## ■ if-else-if-Kaskade (mehrfache Verzweigung)

```
if ( Bedingung1 )
    Anweisung1;
else if ( Bedingung2 )
    Anweisung2;
else
    Anweisung3;
```

| Bedingung <sub>1</sub> |                        |                        |
|------------------------|------------------------|------------------------|
| ja                     | nein                   |                        |
| Anweisung <sub>1</sub> | Bedingung <sub>2</sub> |                        |
|                        | ja                     | nein                   |
|                        | Anweisung <sub>2</sub> | Anweisung <sub>3</sub> |



- **switch**-Anweisung (Fallunterscheidung)
  - Alternative zur **if**-Kaskade bei Test auf Ganzzahl-Konstanten

| ganzzahliger Ausdruck = ? |        |  |        |        |
|---------------------------|--------|--|--------|--------|
| Wert1                     | Wert2  |  |        | sonst  |
| Anw. 1                    | Anw. 2 |  | Anw. n | Anw. x |

```

switch ( Ausdruck ) {
  case Wert1:
    Anweisung1;
    break;
  case Wert2:
    Anweisung2;
    break;
  ...
  case Wertn:
    Anweisungn;
    break;
  default:
    Anweisungx;
}

```

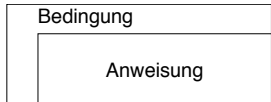


# Abweisende und nicht-abweisende Schleife [=Java]

## ■ Abweisende Schleife

[↪ GDI, II-57] [↪ GDI-Ü, II-9]

- **while**-Schleife
- Null- oder mehrfach ausgeführt



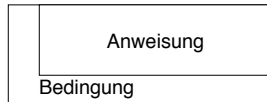
**while**( *Bedingung* )  
    *Anweisung*;

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

## ■ Nicht-abweisende Schleife

[↪ GDI, II-58]

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



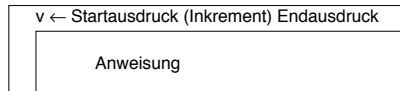
**do**  
    *Anweisung*;  
**while**( *Bedingung* );

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



### ■ **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*)  
    ↪ die **for**-Schleife ist eine „verkappte“ **while**-Schleife



- 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

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

**9 Funktionen**

**10 Variablen**

**11 Präprozessor**



# Was ist eine Funktion?

- **Funktion** := Unterprogramm [↔ GDI, II-79]
  - 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)
```

**Sichtbar:**

Bezeichner und  
formale Parameter

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

**Unsichtbar:**

Tatsächliche  
Implementierung



- Syntax:  $Typ\ Bezeichner\ (FormaleParam_{opt})\ \{Block\}$ 
  - *Typ* Typ des Rückgabewertes der Funktion, [=Java]  
`void` falls kein Wert zurückgegeben wird
  - *Bezeichner* Name, unter dem die Funktion aufgerufen werden kann ↪ 5-3  
[=Java]
  - $FormaleParam_{opt}$  Liste der formalen Parameter:  
 $Typ_1\ Bez_{1\ opt}, \dots, Typ_n\ Bez_{n\ opt}$  [=Java]  
 (Parameter-Bezeichner sind optional)  
`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++ ) {
    }
}
```



## ■ Syntax: *Bezeichner* ( *TatParam* )

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

## ■ 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)



- Generelle Arten der Parameterübergabe [↪ GDI, II-88]
  - *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]



- 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!**



- 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;
}
```



- 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!



- 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 Parametern (`int`, `int`). Was immer an tatsächlichen Parametern übergeben wurde, wird entsprechend interpretiert!
- 10 **Was wird hier ausgegeben?**



- 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!**



- 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

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

**10 Variablen**

**11 Präprozessor**



- **Variable** := Behälter für Werte ( $\mapsto$  Speicherplatz)

- Syntax (Variablendefinition):

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

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



- 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

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

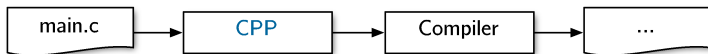
8 Kontrollstrukturen

9 Funktionen

10 Variablen

**11 Präprozessor**





- Bevor eine C-Quelldatei übersetzt wird, wird sie zunächst durch einen Makro-Präprozessor bearbeitet
  - Historisch ein eigenständiges Programm (**CPP** = **C PreProcessor**)
  - 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-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.



## ■ 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
}
```



## ■ 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

**Daniel Lohmann, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2012

[http://www4.cs.fau.de/Lehre/SS12/V\\_SPiC](http://www4.cs.fau.de/Lehre/SS12/V_SPiC)



# Überblick: Teil C Systemnahe Softwareentwicklung

**12 Programmstruktur und Module**

**13 Zeiger und Felder**

**14  $\mu$ C-Systemarchitektur**



- 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, IV]
    - 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

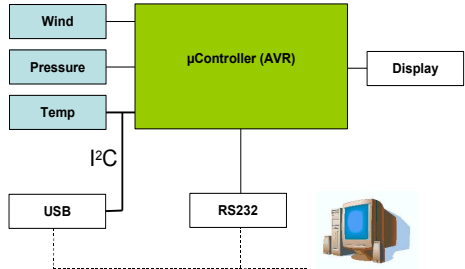
- Wind
- Luftdruck
- Temperatur

### ■ Mehrere Aktoren (hier: Ausgabegeräte)

- LCD-Anzeige
- PC über RS232
- PC über USB

### ■ Sensoren und Aktoren an den $\mu C$ angebunden über verschiedene Bussysteme

- I<sup>2</sup>C
- 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 I<sup>2</sup>C-Datenübertragung initiieren

##### 1.1.2 Daten vom I<sup>2</sup>C-Bus lesen

#### 1.2 Drucksensor lesen

#### 1.3 Windsensor lesen

### 2. Daten aufbereiten (z. B. glätten)

### 3. Daten ausgeben

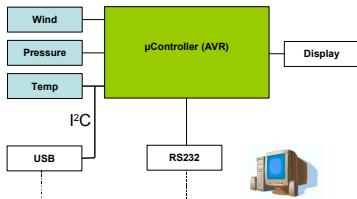
#### 3.1 Daten über RS232 versenden

##### 3.1.1 Baudrate und Parität festlegen (einmalig)

##### 3.1.2 Daten schreiben

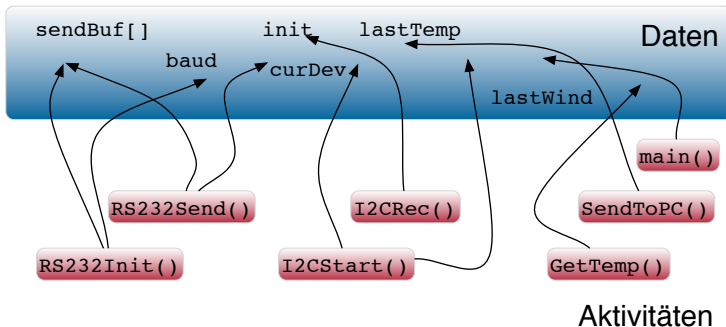
#### 3.2 LCD-Display aktualisieren

### 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 ~ mangelhafte Trennung der Belange



- 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 → 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  | <i>keine</i> , <b>auto</b><br><b>static</b> |   | Definition → Blockende<br>Definition → Blockende | Definition → Blockende<br>Programmstart → Programmende       |
| Global | <i>keine</i><br><b>static</b>               |   | unbeschränkt<br>modulweit                        | Programmstart → Programmende<br>Programmstart → Programmende |

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

void f() {
    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
}
```



- 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  $\mu$ -Controller-Plattformen

↔ 1-3

## **Konsequenz:** Globale Variablen vermeiden!

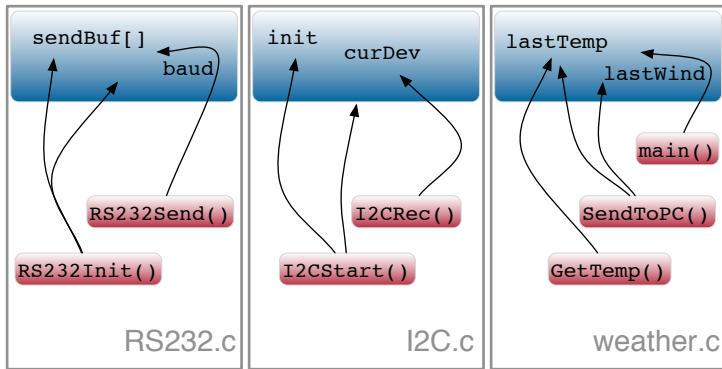
- Globale Variablen sind überall sichtbar
- Globale Variablen belegen Speicher über die gesamte Programmlaufzeit

**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>*, *<Menge von Daten>*, *<Schnittstelle>*) (↦ „**class**“ in Java)
- 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



- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein **idiomatisch** (über **Konventionen**) realisiert ↪ 3–13
  - 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 );  
...
```

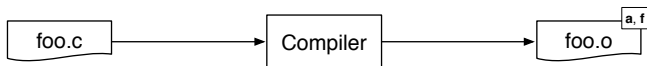
**RS232.h: Schnittstelle / Vertrag (öffentl.)**  
Deklaration der bereitgestellten Funktionen (und ggf. Daten)

```
#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



- 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.



- 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**)

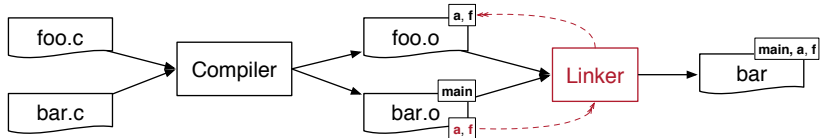
```
extern uint16_t a;  
// declare  
void f(void);      // declare  
  
void main() {      // public  
    a = 0x4711;    // use  
    f();           // use  
}
```

## Objektdatei (**bar.o**)

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



- Die eigentliche Auflösung erfolgt durch den **Linker** [↔ GDI, VI-158]



## 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



- 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)



## Modulschnittstelle: foo.h

```
// foo.h
#ifndef _F00_H
#define _F00_H

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

#endif // _F00_H
```

## Modulimplementierung foo.c

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

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

## Modulverwendung bar.c

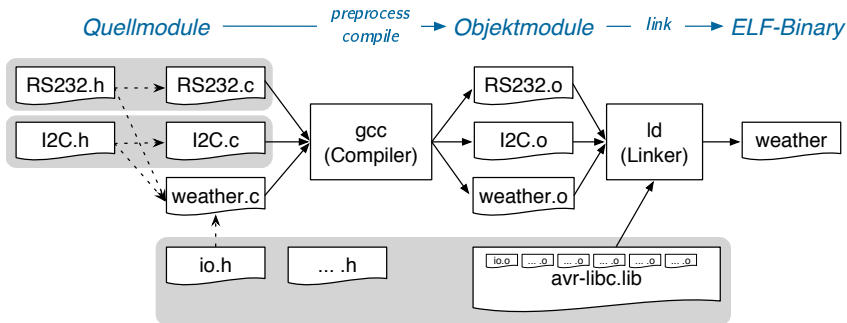
(vergleiche ↪ 12-11)

```
// 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



- 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

12 Programmstruktur und Module

**13 Zeiger und Felder**

**14  $\mu$ C-Systemarchitektur**



# Einordnung: Zeiger (*Pointer*)

- **Literal:** 'a'

Darstellung eines Wertes

'a'  $\equiv$  0110 0001

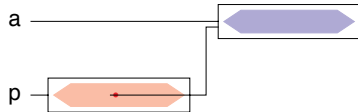
- **Variable:** `char a;`

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 ↪ 3-14
- Aber auch viele Probleme!
  - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
  - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

„Effizienz durch  
Maschinennähe“

↪ 3-14



# Definition von Zeigervariablen

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

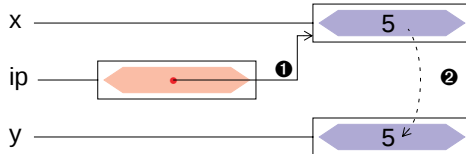
```
int x = 5;
```

```
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))  $\equiv$  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**      in Definitionen und  
                                 **typedef char\* CPTR**      Deklarationen
3. Verweis (unär):      **x = \*p1**      in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung

$\leadsto$  **\*** wird fälschlicherweise für ein Bestandteil des Bezeichners gehalten.



- 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, II-89]
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des \*-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

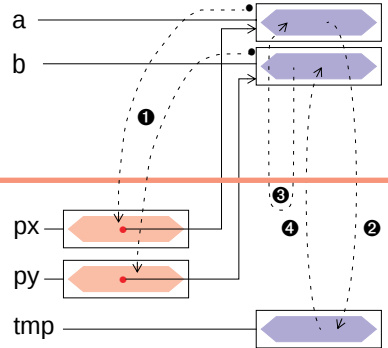
↪ **Call-by-reference**



## ■ Beispiel (Gesamtüberblick)

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

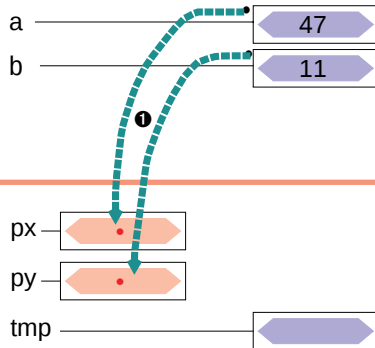
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ❷  
    *px = *py; ❸  
    *py = tmp; ❹  
}
```



## ■ Beispiel (Einzelschritte)

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

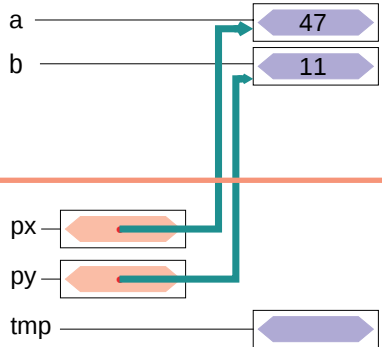
```
void swap (int *px, int *py)  
{  
    int tmp;
```



## ■ Beispiel (Einzelschritte)

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

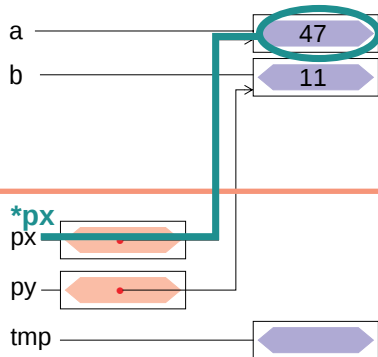
```
void swap (int *px, int *py)  
{  
    int tmp;  
    ...  
}
```



## ■ 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; ②  
}
```



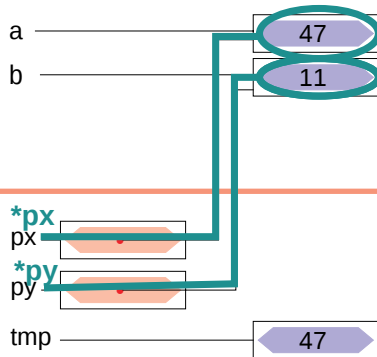
## ■ Beispiel (Einzelschritte)

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

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

```
    tmp = *px; ②
```

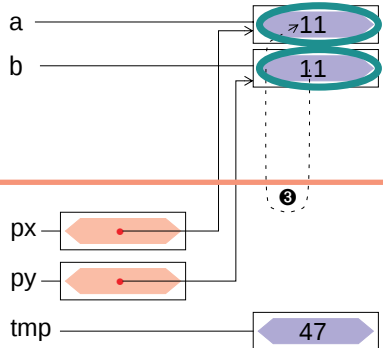
```
    *px = *py; ③
```



## ■ Beispiel (Einzelschritte)

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

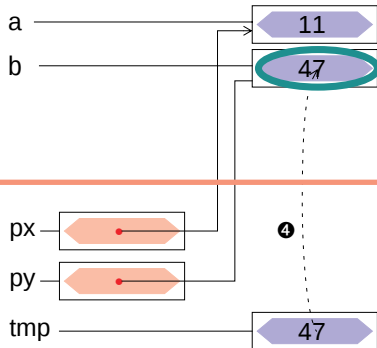
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
}
```



## ■ 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; ②  
    *px = *py; ③  
    *py = tmp; ④  
}
```



- **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 ( $\rightarrow$  Anzahl der Elemente).                      [ $\neq$ 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 };
```



## ■ Syntax: *Feld* [ *IntAusdruck* ]

[=Java]

■ Wobei  $0 \leq \text{IntAusdruck} < n$  für  $n = \text{Feldgröße}$

■ **Achtung:** Feldindex wird nicht überprüft  
    ↪ häufige Fehlerquelle in C-Programmen

[≠Java]

## ■ Beispiel

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

```
LEDs[ 3 ] = BLUE1;
```

```
for( uint8_t i = 0; i < 4; ++i ) {  
    sb_led_on( LEDs[ i ] );  
}
```

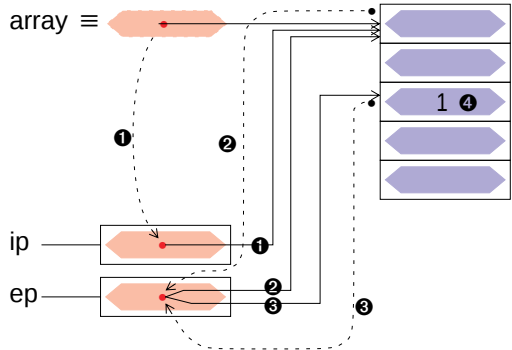
```
LEDs[ 4 ] = GREEN1;    // UNDEFINED!!!
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes:  $\text{array} \equiv \&\text{array}[0]$ 
  - Ein Alias – kein Behälter  $\rightsquigarrow$  Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes:  $\text{array} \equiv \&\text{array}[0]$ 
  - Ein Alias – kein Behälter  $\rightsquigarrow$  Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

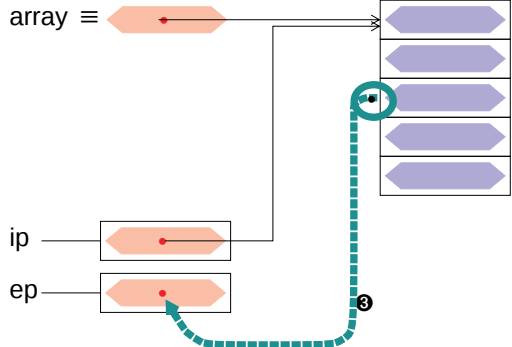
```
int array[5];
```

```
int *ip = array; ①
```

```
int *ep;
```

```
ep = &array[0]; ②
```

```
ep = &array[2]; ③
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes:  $\text{array} \equiv \&\text{array}[0]$ 
  - Ein Alias – kein Behälter  $\rightsquigarrow$  Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];
```

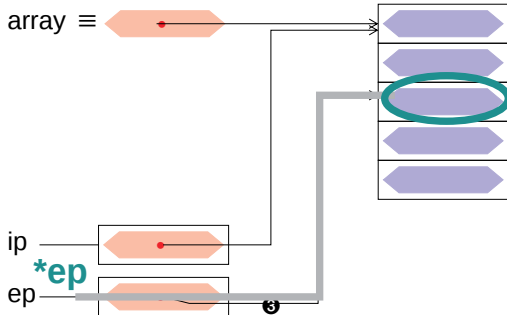
```
int *ip = array; ①
```

```
int *ep;
```

```
ep = &array[0]; ②
```

```
ep = &array[2]; ③
```

```
*ep = 1; ④
```



# Zeiger sind Felder

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

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };  
  
LEDs[ 3 ]      = BLUE1;  
uint8_t *p     = LEDs;  
for( uint8_t i = 0; i < 4; ++i ) {  
    sb_led_on( p[ i ] );  
}
```

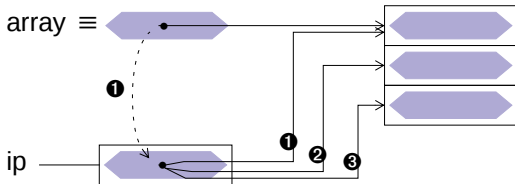


# Rechnen mit Zeigern

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

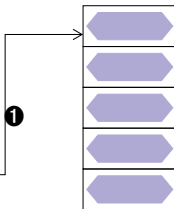
```
int array[3];  
int *ip = array; ❶
```

```
ip++; ❷  
ip++; ❸
```



```
int array[5];  
ip = array; ❶
```

ip



$(ip+3) \equiv \&ip[3]$

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



## ■ 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)

13-Zeiger: 2012-04-18



- 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 Funktionsparameters
- Bei Variablendefinitionen hat **array[]** eine **völlig andere** Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, ↪ 13-8)



- 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"  $\equiv$    $\leftrightarrow$  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

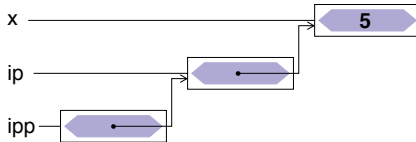
```
int strlen( const char *s ) {  
    const char *end = s;  
    while( *end )  
        end++;  
    return end - s;  
}
```



# 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
    - 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
}
```



- Syntax (Definition): *Typ* ( \* *Bezeichner* ) ( *FormaleParam<sub>opt</sub>* );  
(sehr ähnlich zur Syntax von Funktionsdeklarationen) ↪ 9-3

- *Typ* Rückgabebetyp der *Funktionen*, auf die dieser Zeiger verweisen kann
- *Bezeichner* Name des *Funktionszeigers*
- *FormaleParam<sub>opt</sub>* Formale Parameter der *Funktionen*, auf die dieser Zeiger verweisen kann: *Typ<sub>1</sub>, ..., Typ<sub>n</sub>*

- 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()
}
```



- Funktionszeiger werden oft für **Rückruffunktionen** (*Callbacks*) zur Zustellung asynchroner Ereignisse verwendet (→ „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, BTNPPRESSED,      //   for this button and events
        onButton                    //   invoke this function
    );
    sei();                           // enable interrupts (necessary!)
    while( 1 ) ;                     // wait forever
}
```



- 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

12 Programmstruktur und Module

13 Zeiger und Felder

**14  $\mu$ C-Systemarchitektur**

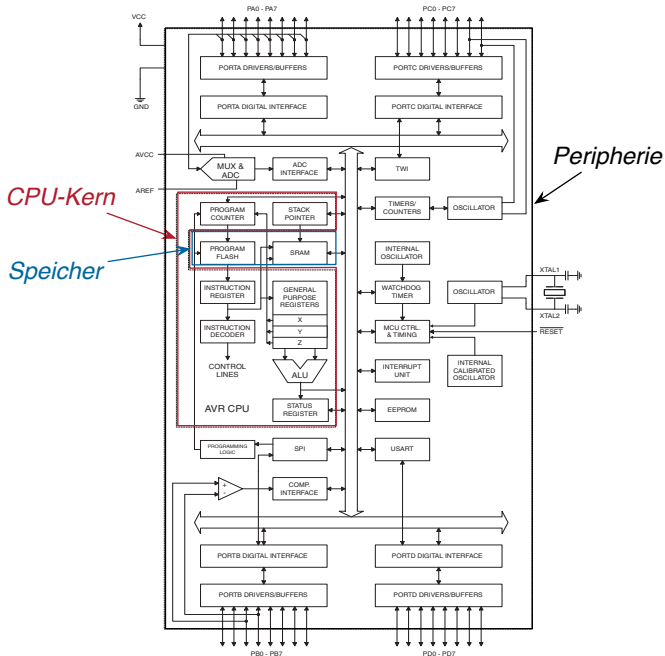


# Was ist ein $\mu$ -Controller?

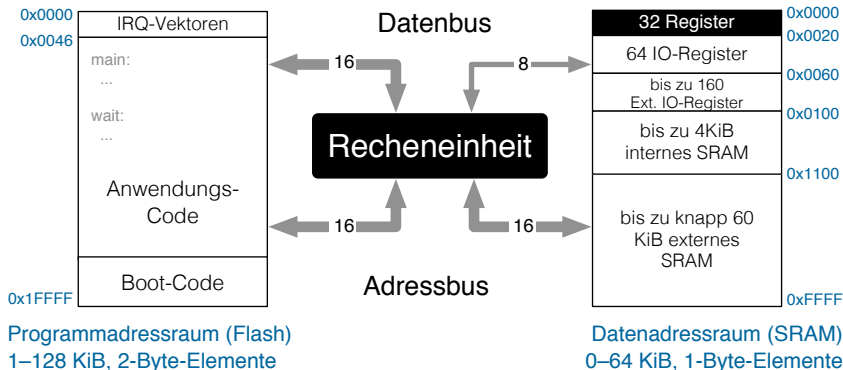
- **$\mu$ -Controller** := Prozessor + Speicher + Peripherie
  - Faktisch ein Ein-Chip-Computersystem  $\rightarrow$  SoC (*System-on-a-Chip*)
  - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher  $\leadsto$  kostengünstiges Systemdesign
- Wesentliches Merkmal 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<sup>2</sup>C, ...
  - ...
- Die Abgrenzungen sind fließend: Prozessor  $\longleftrightarrow \mu\text{C} \longleftrightarrow \text{SoC}$ 
  - AMD64-CPU's haben ebenfalls eingebaute Timer, Speicher (Caches), ...
  - Einige  $\mu\text{C}$  erreichen die Geschwindigkeit „großer Prozessoren“



# Beispiel ATmega32: Blockschaltbild



# Beispiel ATmega-Familie: CPU-Architektur

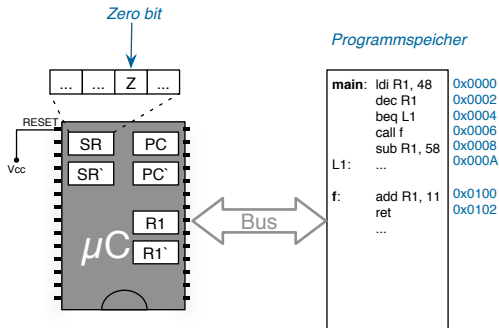


- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingebündet  
~ ansprechbar wie globale Variablen

Zum Vergleich: PC basiert auf von-Neumann-Architektur [↔ GDI, VI-6] mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

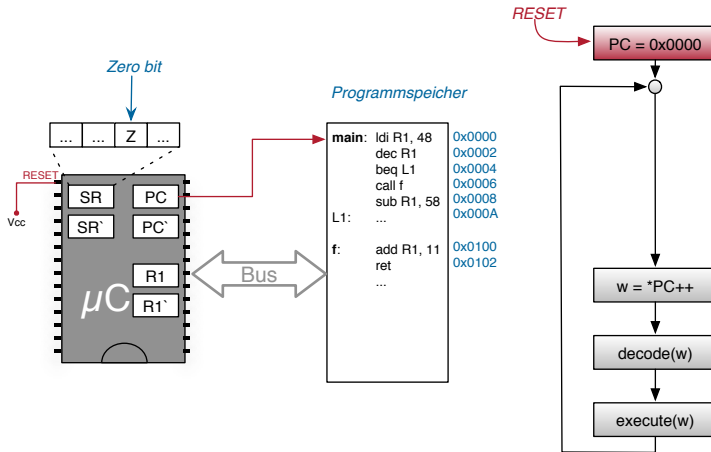


# Wie arbeitet ein Prozessor?

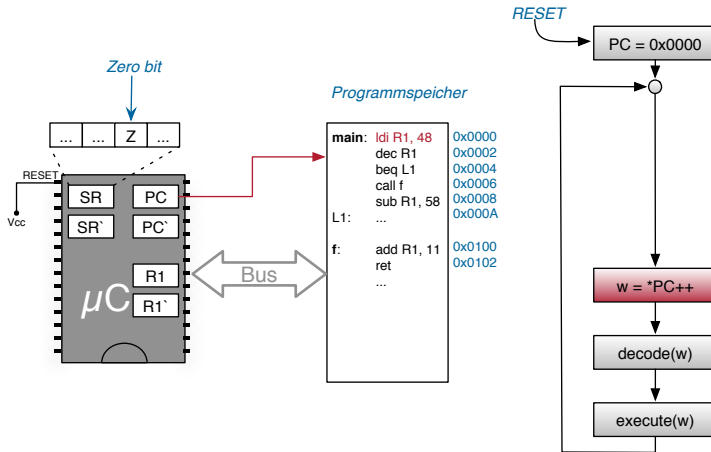


- Hier am Beispiel eines sehr einfachen Pseudoprozessors
  - Nur zwei Vielzweckregister (R1 und R2)
  - Programmzähler (PC) und Statusregister (SR) (+ „Schattenkopien“)
  - Kein Datenspeicher, kein Stapel → Programm arbeitet nur auf Registern

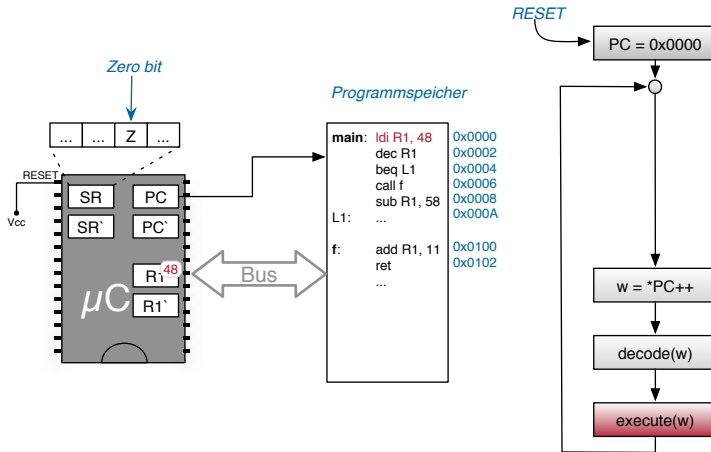
# Wie arbeitet ein Prozessor?



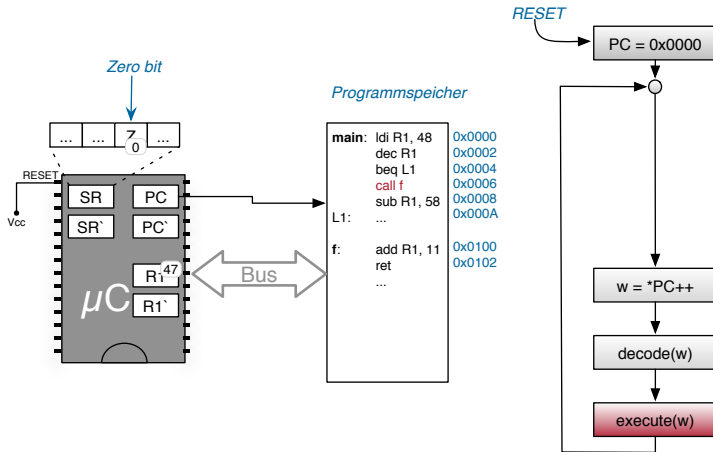
# Wie arbeitet ein Prozessor?



# Wie arbeitet ein Prozessor?



# Wie arbeitet ein Prozessor?

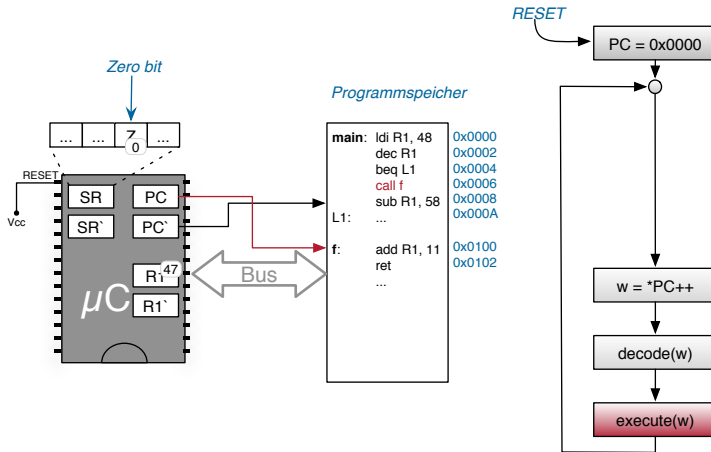


**w: dec <R>**  
 $R \leftarrow R - 1$   
 if (  $R == 0$  )  $Z = 1$   
 else  $Z = 0$

**w: beq <lab>**  
 if (  $Z$  )  $PC = \text{lab}$

**w: call <func>**  
 $PC' = PC$   
 $PC = \text{func}$

# Wie arbeitet ein Prozessor?

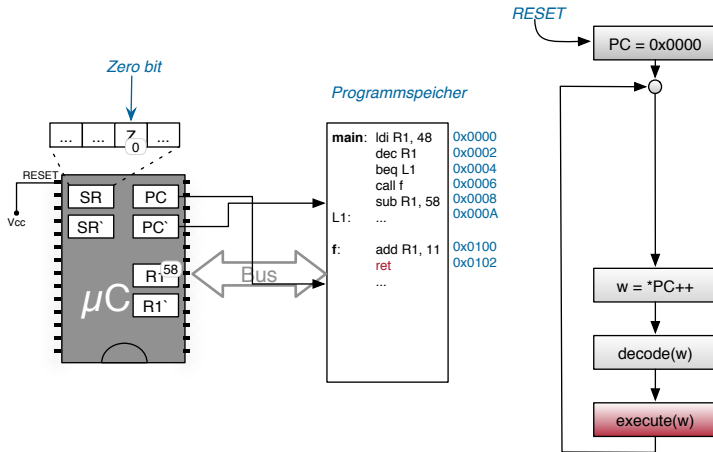


**w: dec <R>**  
 R ← 1  
 if (R == 0) Z = 1  
 else Z = 0

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

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

# Wie arbeitet ein Prozessor?



**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

**w: ret**  
 PC = PC'

- **Peripheriegerät:** Hardwarekomponente, die sich „außerhalb“ der Zentraleinheit eines Computers befindet
  - Traditionell (PC): Tastatur, Bildschirm, ...  
(→ physisch „außerhalb“)
  - Allgemeiner: Hardwarefunktionen, die nicht direkt im Befehlssatz des Prozessors abgebildet sind  
(→ 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



- Auswahl von typischen Peripheriegeräten in einem  $\mu$ -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<sup>2</sup>C) Protokoll.
  - A/D-Wandler Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V  $\leftrightarrow$  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



- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
  - Memory-mapped: Register sind in den Adressraum eingeblendet; der Zugriff erfolgt über die Speicherbefehle des Prozessors (**load**, **store**)  
(Die meisten  $\mu\text{C}$ )
  - Port-basiert: Register sind in einem eigenen I/O-Adressraum organisiert; der Zugriff erfolgt über spezielle **in**- und **out**-Befehle  
(x86-basierte PCs)
- 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]



- 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 ) )
```

Adresse: int  
Adresse: volatile uint8\_t\* (Cast  $\hookrightarrow$  7-17)  
Wert: volatile uint8\_t (Dereferenzierung  $\hookrightarrow$  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          // button1 pressed
    ...
L2:
    ret
```

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



- 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
    // "nonsensical" loop away
    ret
```

## **Achtung:** `volatile` ↪ \$\$\$

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



- **Port** := Gruppe von (üblicherweise 8) digitalen Ein-/Ausgängen
  - Digitaler Ausgang: Bitwert  $\mapsto$  Spannungspegel an  $\mu$ C-Pin
  - Digitaler Eingang: Spannungspegel an  $\mu$ C-Pin  $\mapsto$  Bitwert
  - Externer Interrupt: Spannungspegel an  $\mu$ C-Pin  $\mapsto$  Bitwert  
(bei Pegelwechsel)  $\leadsto$  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

## PDIP

|                 |      |    |               |
|-----------------|------|----|---------------|
| (XCK/T0) PB0    | □ 1  | 40 | □ PA0 (ADC0)  |
| (T1) PB1        | □ 2  | 39 | □ PA1 (ADC1)  |
| (INT2/AIN0) PB2 | □ 3  | 38 | □ PA2 (ADC2)  |
| (OC0/AIN1) PB3  | □ 4  | 37 | □ PA3 (ADC3)  |
| (SS) PB4        | □ 5  | 36 | □ PA4 (ADC4)  |
| (MOSI) PB5      | □ 6  | 35 | □ PA5 (ADC5)  |
| (MISO) PB6      | □ 7  | 34 | □ PA6 (ADC6)  |
| (SCK) PB7       | □ 8  | 33 | □ PA7 (ADC7)  |
| RESET           | □ 9  | 32 | □ AREF        |
| VCC             | □ 10 | 31 | □ GND         |
| GND             | □ 11 | 30 | □ AVCC        |
| XTAL2           | □ 12 | 29 | □ PC7 (TOSC2) |
| XTAL1           | □ 13 | 28 | □ PC6 (TOSC1) |
| (RXD) PD0       | □ 14 | 27 | □ PC5 (TDI)   |
| (TXD) PD1       | □ 15 | 26 | □ PC4 (TDO)   |
| (INT0) PD2      | □ 16 | 25 | □ PC3 (TMS)   |
| (INT1) PD3      | □ 17 | 24 | □ PC2 (TCK)   |
| (OC1B) PD4      | □ 18 | 23 | □ PC1 (SDA)   |
| (OC1A) PD5      | □ 19 | 22 | □ PC0 (SCL)   |
| (ICP1) PD6      | □ 20 | 21 | □ PD7 (OC2)   |

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:  $\hookrightarrow$  3-5 und  $\hookleftarrow$  3-8

[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“
- Dies geht in C mit **Verbundtypen** (Strukturen)

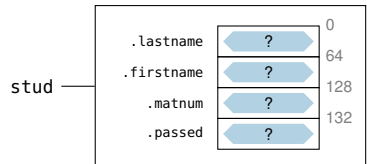


```
// Structure declaration
struct Student {
    char    lastname[64];
    char    firstname[64];
    long    matnum;
    int     passed;
};

// 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

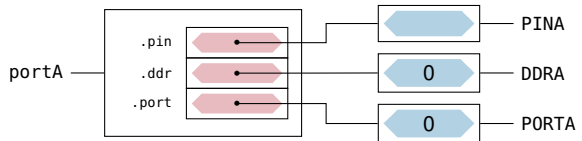
↪ 6-8

```
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]

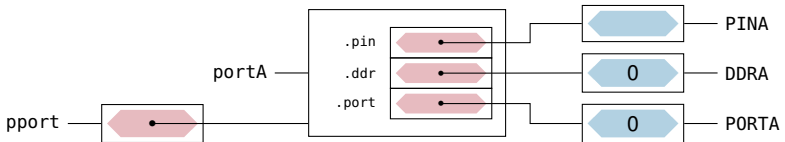
```
port_t portA = { &PINA, &DDRA, &PORTA };
```

```
*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  $s \rightarrow m \equiv (*s).m$

```
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.ddd = 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->ddd = 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

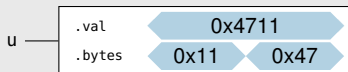
- In einer Struktur liegen die Elemente **hintereinander** im Speicher, in einer Union hingegen **übereinander**
  - Wert im Speicher lässt sich verschieden (Typ)-interpretieren
  - Nützlich für bitweise Typ-Casts
- Beispiel

↪ 14-15

```
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] );
    ...
}
```



47

11



# 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

**Daniel Lohmann, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2012

[http://www4.cs.fau.de/Lehre/SS12/V\\_SPiC](http://www4.cs.fau.de/Lehre/SS12/V_SPiC)



# Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

20 Speicherorganisation

21 Prozesse II

22 Betriebssysteme II



## Programmablauf in einer $\mu$ C-Umgebung

- Programm läuft "nackt" auf der Hardware
  - ➔ Compiler und Binder müssen ein vollständiges Programm erzeugen
    - keine Betriebssystemunterstützung zur Laufzeit
  - Funktionalität muss entweder vom Anwender programmiert werden oder in Form von Funktionsbibliotheken zum Programm dazugebunden werden
  - Umgang mit "lästigen Programmierdetails" (z. B. bestimmte Bits setzen) wird durch Makros erleichtert
- Es wird genau ein Programm ausgeführt
  - Programm kann zur Laufzeit "niemanden stören"
  - Fehler betreffen nur das Programm selbst
  - keine Schutzmechanismen notwendig
    - ➔ **ABER:** Fehler ohne direkte Auswirkung werden leichter übersehen



- generell bei Mikrocontrollern mehrere Möglichkeiten
  - Programm ist schon da (ROM)
  - Bootloader-Programm ist da, liest Anwendung über serielle Schnittstelle ein und speichert sie im Programmspeicher ab
  - spezielle Hardware-Schnittstelle
    - "jemand anderes" kann auf Speicher zugreifen
    - Beispiel: JTAG  
spezielle Hardware-Komponente im AVR-Chip, die Zugriff auf die Speicher hat und mit der man über spezielle PINs kommunizieren kann



# Programm starten

- Reset bewirkt Ausführung des Befehls an Adresse 0x0000
  - dort steht ein Sprungbefehl auf die Speicheradresse einer start-Funktion, die nach einer Initialisierungsphase die main-Funktion aufruft
  - alternativ: Sprungbefehl auf Adresse des Bootloader-Programms, Bootloader lädt Anwendung, initialisiert die Umgebung und springt dann auf main-Adresse

## Fehler zur Laufzeit

- Zugriff auf ungültige Adresse
  - es passiert nichts:
    - Schreiben geht in's Leere
    - Lesen ergibt zufälliges Ergebnis
- ungültige Operation auf nur-lesbare / nur-schreibbare Register/Speicher
  - hat keine Auswirkung



# Interrupts

- An einer Peripherie-Schnittstelle tritt ein Ereignis auf
  - Spannung wird angelegt
  - Zähler ist abgelaufen
  - Gerät hat Aufgabe erledigt (z. B. serielle Schnittstelle hat Byte übertragen, A/D-Wandler hat neuen Wert vorliegen)
  - Gerät hat Daten für die Anwendung bereit stehen (z. B. serielle Schnittstelle hat Byte empfangen)
- ? wie bekommt das Programm das mit?
  - Zustand der Schnittstelle regelmäßig überprüfen (= **Polling**)
  - Schnittstelle meldet sich von sich aus beim Prozessor und unterbricht den Programmablauf (= **Interrupt**)



## ■ Polling

- + Pollen erfolgt **synchron** zum Programmablauf, Programm ist in dem Moment auf das Ereignis vorbereitet
- Pollen erfolgt explizit im Programm und meistens umsonst — Rechenzeit wird verschwendet
- Polling-Funktionalität ist in den normalen Programmablauf eingestreut — und hat mit der "eigentlichen" Funktionalität dort meist nichts zu tun

## ■ Interrupts

- + Interrupts melden sich nur, wenn tatsächlich etwas zu erledigen ist
- + Interrupt-Bearbeitung ist in einer Funktion kompakt zusammengefasst
- Interrupts unterbrechen den Programmablauf irgendwo (**asynchron**), sie könnten in dem Augenblick stören
  - ➡ durch die Interrupt-Bearbeitungensteht **Nebenläufigkeit**



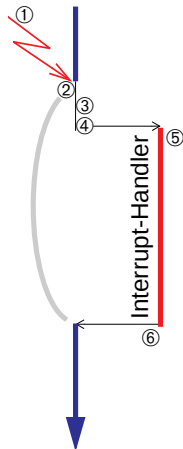
# Implementierung von Interrupts

- typischerweise mehrere Interrupt-Quellen
- Interrupt-Vektor
  - Speicherbereich (Tabelle), der für jeden Interrupt Informationen zur Bearbeitung enthält
    - Maschinenbefehl  
(typischerweise ein Sprungbefehl auf eine Adresse, an der eine Bearbeitungsfunktion (***Interrupt-Handler***) steht)  
oder
    - Adresse einer Bearbeitungsfunktion
  - feste Position im Speicher — ist im Prozessorhandbuch nachzulesen
- Maskieren von Interrupts
  - Bit im Prozessor-Statusregister schaltet den Empfang aller Interrupts ab
  - zwischenzeitlich eintreffende Interrupts werden gepuffert (nur einer!)
  - die Erzeugung einzelner Interrupts kann am jeweiligen Gerät unterbunden werden



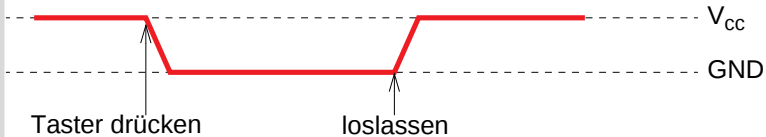
# Interrupts: Ablauf auf Hardware-Ebene

- ① Gerät löst Interrupt aus, Ablauf des Anwendungsprogramms wird unmittelbar unterbrochen
  - ② weitere Interrupts werden deaktiviert
  - ③ aktuelle Position im Programm wird gesichert
  - ④ Eintrag im Interrupt-Vektor ermitteln
  - ⑤ Befehl wird ausgeführt bzw. Funktion aufrufen (= Sprung in den Interrupt-Handler)
  - ⑥ am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts
- ! Der Interrupt-Handler muss alle Register, die er ändert am Anfang sichern und vor dem Rücksprung wieder herstellen!**



# Pegel- und Flanken-gesteuerte Interrupts

## ■ Beispiel: Signal eines (idealisierten) Tasters



## ■ Flanken-gesteuert

- Interrupt wird durch die Flanke (= Wechsel des Pegels) ausgelöst
- welche Flanke einen Interrupt auslöst kann bei manchen Prozessoren konfiguriert werden

## ■ Pegel-gesteuert

- solange ein bestimmter Pegel anliegt (hier Pegel = GND) wird immer wieder ein Interrupt ausgelöst



# Nebenläufigkeit – Überblick

- Definition von Nebenläufigkeit:  
*zwei Programmausführungen sind nebenläufig, wenn für zwei einzelne Befehle a und b aus beiden Ausführungen nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird*
- Nebenläufigkeit tritt auf
  - bei Interrupts
  - bei parallelen Abläufen (gleichzeitige Ausführung von Code in einem Mehrprozessorsystem mit Zugriff auf den gleichen Speicher)
  - bei quasi-parallelen Abläufen (wenn ein Betriebssystem verschiedenen Prozesse den Prozessor jeweils für einen Zeitraum zuteilt und ihn nach Ablauf der Zeit wieder entzieht)
- Problem:
  - was passiert, wenn die nebenläufigen Ausführungen auf die gleichen Daten im Speicher zugreifen?



# Nebenläufigkeit durch Interrupts

- Interrupts unterbrechen Anwendungsprogramme "irgendwo"
- Interrupts haben Zugriff auf den gleichen Speicher
- Szenario:
  - eine Lichtschranke soll Fahrzeuge zählen und alle 10 Sekunden soll der Wert ausgegeben werden

```
static volatile uint16_t a;

void main(void) {
    volatile uint32_t i;
    while(1) {
        for (i=0; i<2000000; i++)
            /* Zählen dauert 10 Sek. */;
        print(a);
        a=0;
    }
}
```

```
/* Lichtschranken-
   Interrupt */
void count(void) {
    a++;
}
```



# Nebenläufigkeit durch Interrupts (2)

- Auf C-Ebene führt die Interrupt-Behandlung nur einen Befehl aus: `a++`
  - nur scheinbar ein Befehl
  - auf Maschinencode-Ebene (Bsp. AVR) sieht die Sache anders aus

```
...  
    print(a);  
    a=0;  
...
```

```
...  
.L5:  
    lds r24,a  
    lds r25,(a)+1  
    rcall print  
    sts (a)+1,__zero_reg__  
    sts a,__zero_reg__  
    rjmp .L2  
...
```

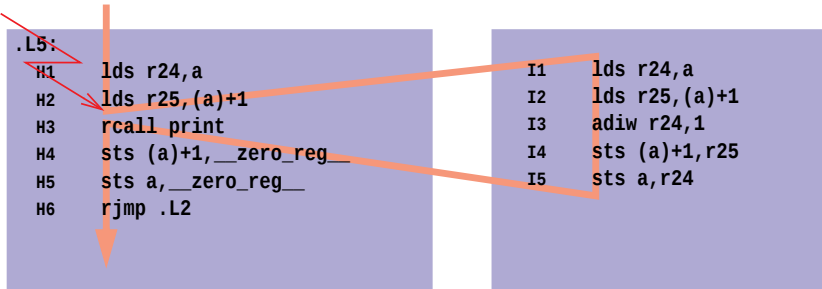
```
void count(void) {  
    a++;  
}
```

```
...  
    lds r24,a  
    lds r25,(a)+1  
    adiw r24,1  
    sts (a)+1,r25  
    sts a,r24  
...
```



# Nebenläufigkeit durch Interrupts (3)

- Annahme1: Interrupt trifft folgendermaßen ein:



- Folge: ein Fahrzeug wird nicht gezählt

➔ **Lost-Update-Problem**

- Details des Szenarios zeigen mehrere Problemstellen:

- uint16-Wert wird in zwei Schritten in zwei Register geladen (uint32: 4 Register)
- Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben



# Nebenläufigkeit durch Interrupts (3)

- Skizze zu Annahme 1, a habe initial den Wert 5

| Code-zeile | Variable a  |              | Prozessor-Register |     | gesicherte Registerinhalte |     | Ausgabe von print |
|------------|-------------|--------------|--------------------|-----|----------------------------|-----|-------------------|
|            | oberes Byte | unteres Byte | r25                | r24 | r25                        | r24 |                   |
| initial    | 00          | 05           |                    |     |                            |     |                   |
| H1         |             | 05           |                    | 05  |                            |     |                   |
| H2         | 00          |              | 00                 |     |                            |     |                   |
| INT        |             |              | 00                 | 05  | 00                         | 05  |                   |
| I1         |             | 05           |                    | 05  |                            |     |                   |
| I2         | 00          |              | 00                 | 05  |                            |     |                   |
| I3         |             |              | 00                 | 06  |                            |     |                   |
| I4         | 00          |              | 00                 |     |                            |     |                   |
| I5         |             | 06           |                    | 06  |                            |     |                   |
| ret        |             |              | 00                 | 05  | 00                         | 05  |                   |
| H3         |             |              | 00                 | 05  |                            |     | 5                 |
| H4         | 00          |              |                    |     |                            |     |                   |
| H5         |             | 00           |                    |     |                            |     |                   |



# Nebenläufigkeit durch Interrupts (4)

- Annahme 2: Interrupt trifft folgendermaßen ein:



- Folge: möglicherweise werden 255 Fahrzeuge zuviel gezählt
  - Variable a ist auf 2 Register verteilt → a = 0 nicht atomar  
zuerst wird obere Hälfte auf 0 gesetzt
  - falls a++ im Interrupt-Handler a zufällig von 255 auf 256 zählt  
→ Bitüberlauf vom "unteren" in's "obere" Register
  - nach Interrupt wird nur noch untere Hälfte auf 0 gesetzt → a = 256



# Nebenläufigkeit durch Interrupts (4)

- Skizze zu Annahme 2, a habe initial den Wert 255

| Code-zeile | Variable a  |              | Prozessor-Register |     | gesicherte Registerinhalte |     | Ausgabe von print |
|------------|-------------|--------------|--------------------|-----|----------------------------|-----|-------------------|
|            | oberes Byte | unteres Byte | r25                | r24 | r25                        | r24 |                   |
| initial    | 00          | ff           |                    |     |                            |     |                   |
| H1         |             | ff           |                    | ff  |                            |     |                   |
| H2         | 00          |              | 00                 |     |                            |     |                   |
| H3         |             |              | 00                 | ff  |                            |     | 255               |
| H4         | 00          |              |                    |     |                            |     |                   |
| INT        |             |              | 00                 | ff  | 00                         | ff  |                   |
| I1         |             | ff           |                    | ff  |                            |     |                   |
| I2         | 00          |              | 00                 | ff  |                            |     |                   |
| I3         |             |              | 01                 | 00  |                            |     |                   |
| I4         | 01          |              | 01                 |     |                            |     |                   |
| I5         |             | 00           |                    | 00  |                            |     |                   |
| ret        |             |              | 00                 | ff  | 00                         | ff  |                   |
| H5         | 01          | 00           |                    |     |                            |     |                   |



# Nebenläufigkeit durch Interrupts (5)

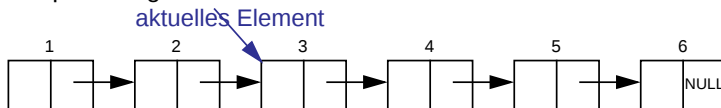
- weiteres Problem bei Zugriff auf globale Variablen:
  - AVR stellt 32 Register zur Verfügung
  - Compiler optimiert Code und vermeidet Speicherzugriffe wenn möglich
    - Variablen werden möglichst in Registern gehalten
  - Registerinhalte werden bei Interrupt gesichert und am Ende restauriert
    - Änderungen der Interrupt-Funktion an einer Variablen gehen beim Restaurieren der Register wieder verloren
- Lösung für dieses Problem:
  - Compiler muss Variablen vor jedem Zugriff aus dem Speicher laden und anschließend zurückschreiben
    - Attribut `volatile`  

```
volatile uint16_t a;
```
  - Nachteil: Code wird umfangreicher und langsamer
    - nur einsetzen wo unbedingt notwendig!

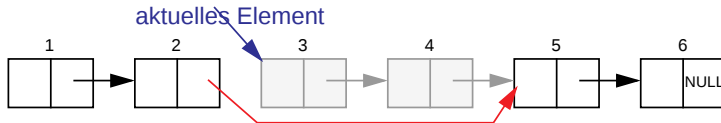


# Nebenläufigkeitsprobleme allgemein

- Zugriff auf gemeinsame Daten ist bei nebenläufigen Ausführungen generell kritisch
  - selbst bei einfachen Variablen (siehe vorheriges Beispiel)
  - Problem bei komplexeren Datenstrukturen  
(z. B. Einketten einer Struktur in verkettete Liste)  
noch gravierender: Datenstruktur kann völlig zerstört werden
- Beispiel: Programm läuft durch eine verkettete Liste



- Interrupt-Handler oder parallel laufendes Programm entfernt Elemente 3 und 4 und gibt den Speicher dieser Elemente frei



# Umgang mit Nebenläufigkeitsproblemen

- Gemeinsame Daten möglichst vermeiden
  - Interrupt-Funktionen sollten weitgehend auf eigenen Daten arbeiten
  - Parallele Abläufe sollten ebenfalls möglichst eigene Datenbereiche haben
- Kommunikation zwischen Anwendungsabläufen erfordert aber oft gemeinsame Daten
  - solche Daten sollten deutlich hervorgehoben werden  
z. B. durch entsprechenden Namen

```
volatile uint16_t INT_zaeher;
```

- betrifft nur globale Variablen
- lokale Variablen sind unkritisch  
(nur in der jeweiligen Funktion sichtbar)
- Zugriff auf solche Daten sollte in der Anwendung möglichst begrenzt sein  
(z. B. nur in bestimmten Funktionen,  
gemeinsames Modul mit Interrupt-Handlern, vgl. Kap. 12)



# Umgang mit Nebenläufigkeitsproblemen (2)

- Zugriffskonflikte mit Interrupt-Handlern verhindern
  - das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
    - Beispiel AVR:  
Funktionen `cli()` (blockiert alle Interrupts)  
und `sei()` (erlaubt Interrupts)
  - Problem: Interrupt-Verluste bei Interrupt-Sperren
    - trifft ein Interrupt während der Sperre ein, wird im zugehörigen Register das entsprechende Bit gesetzt
    - treffen weitere Interrupts ein, geht diese Information verloren
- ➡ Zeitraum von Interruptsperren muss möglichst kurz bleiben!
  - alternativ kann es sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift  
(hängt vom Einzelfall und von Details der Hardware ab!)



# Umgang mit Nebenläufigkeitsproblemen (3)

## ■ Warten auf einen Interrupt

- Häufiges Szenario: im Programm soll auf ein bestimmtes Ereignis gewartet werden, das durch einen Interrupt signalisiert wird
  - Warten erfolgt meist passiv (Sleep-Modus des Prozessors)
- Problem: Abfrage ob Ereignis bereits eingetreten ist, ist ein kritischer Zugriff auf gemeinsame Daten mit der Interrupt-Behandlung

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();

    while(1) {
        while(event == 0) { /* Warte auf Ereignis */
            sleep_cpu();
        }
        /* bearbeite Ereignis */
        ...
    }
}
```

- Synchronisation erforderlich?



# Umgang mit Nebenläufigkeitsproblemen (4)

- ... Warten auf einen Interrupt
- Was passiert, wenn der Interrupt an dieser Stelle eintrifft?

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();

    while(1) {
        while(event == 0) { /* Warte auf Ereignis */
            sleep_cpu();
        }
        /* bearbeite Ereignis */
        ...
    }
}
```



## ↳ Lost-Wakeup-Problem



# Umgang mit Nebenläufigkeitsproblemen (5)

- ... Warten auf einen Interrupt

- kritischer Abschnitt

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();

    while(1) {
        kritischer Abschnitt
        while(event == 0) { /* Warte auf Ereignis */
            sleep_cpu(); ← Interrupt!
        }

        /* bearbeite Ereignis */
        ...
    }
}
```

- können hier Interruptsperrn helfen?



# Umgang mit Nebenläufigkeitsproblemen (6)



... Warten auf einen Interrupt

■ Problem: Interruptsperre muss vor dem `sleep_cpu()` aufgehoben werden

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
    sleep_enable();

    while(1) {
        cli(); kritischer Abschnitt
        while(event == 0) { /* Warte auf Ereignis */
            sei();          ← Interrupt!
            sleep_cpu();
        }

        /* bearbeite Ereignis */
        ...
    }
}
```

- aber Interrupt darf nicht zwischen `sei()` und `sleep_cpu()` kommen
- Lösung: `sei()` und die Folgeanweisung werden atomar ausgeführt (Hardware-Mechanismus des AVR-Prozessors!)



# Umgang mit Nebenläufigkeitsproblemen (7)

## ■ Einseitige Synchronisation

### ■ Besonderheit bei Nebenläufigkeit durch Interrupts:

- der Interrupt kann den normalen Programmablauf unterbrechen
- aber nicht umgekehrt
- ➡ die Interruptbehandlung wird nie unterbrochen  
(höchstens durch Interrupts mit höherer Priorität)

## ■ Mehrseitige Synchronisation

### ■ Standardsituation bei parallelen Abläufen (z. B. auf Mehrkern-Prozessoren)

- Interruptsperrern helfen hier nicht

### ■ Lösungen

- spezielle atomare Maschinenbefehle  
(z. B. test-and-set oder compare-and-swap bei Intel-Architekturen)
- Software-Synchronisation (lock-Variablen, Semaphore, etc.)
- Kommunikation mittels Nachrichten statt gemeinsamer Daten



# Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

**16 Ergänzungen zur Einführung in C**

**17 Betriebssysteme I**

**18 Dateisysteme**

**19 Prozesse I**

**20 Speicherorganisation**

**21 Prozesse II**

**22 Betriebssysteme II**

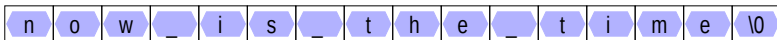


## Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (**char**), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind

- 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";
```



amessage  $\equiv$

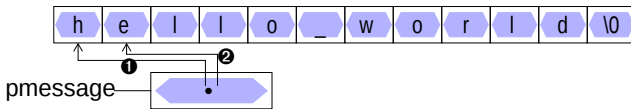
- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- amessage ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden  
`amessage[0] = 'h';`



# ... Zeiger, Felder und Zeichenketten (2)

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

```
char *pmessage = "hello world";
```



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

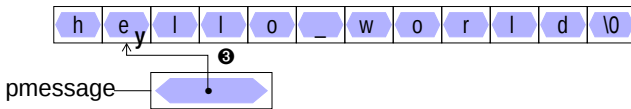
- es wird ein Speicherbereich für einen Zeiger reserviert (z. B. 4 Byte) und der Compiler legt die Zeichenkette hello world an irgendeiner Adresse im Speicher des Programms ab
- pmessage ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf  
pmessage++;



# ... 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

```
char *pmessage = "hello world";
```



```
*pmessage = 'y'; ③
```

- der Speicherbereich von `hello world` darf aber nicht verändert werden
  - manche Compiler legen solche Zeichenketten in schreibgeschütztem Speicher an
    - ➔ Speicherschutzverletzung beim Zugriff
  - sonst funktioniert der Zugriff obwohl er nicht erlaubt ist
    - ➔ Programm funktioniert nur in manchen Umgebungen

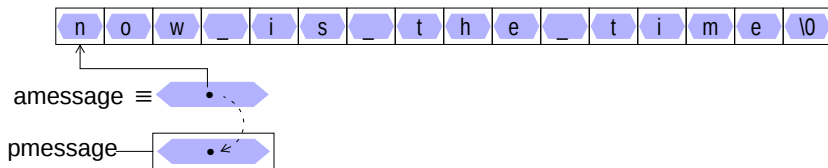


## ... Zeiger, Felder und Zeichenketten (4)

- 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



## ■ Zeichenketten kopieren

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

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

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



# 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; ②
```

①

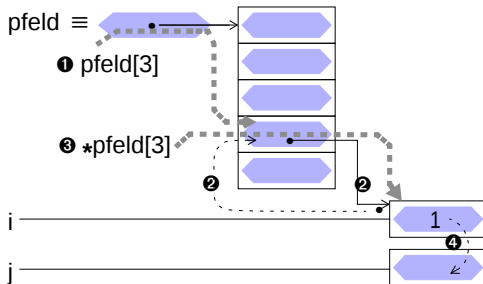
- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3]; ④
```

①

③

④

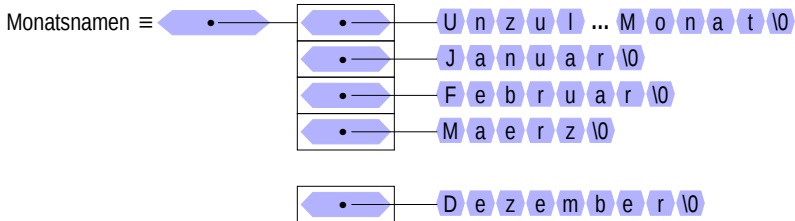


# Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

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

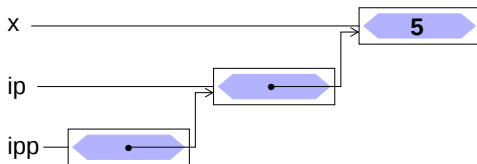
    return ( (n<0 || n>12) ?
             Monatsnamen[0] : Monatsnamen[n] );
}
```



# Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist

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



- wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muss (z. B. swap-Funktion für Zeiger)



# 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)  
{  
    ...  
}
```

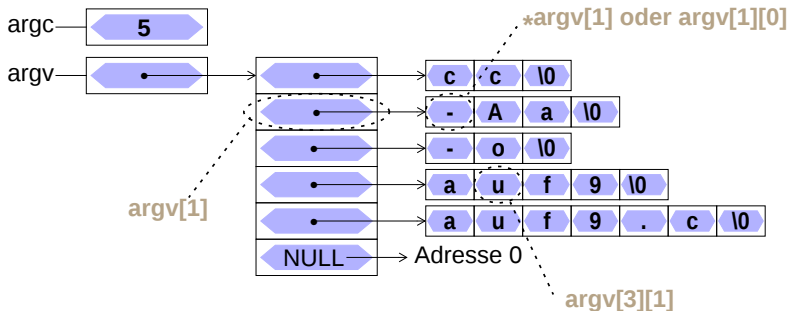
- der Parameter **argc** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **argv** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**argv[0]**)



# Datenaufbau

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

Datei cc.c:  
...  
`main(int argc, char *argv[]) {`  
...



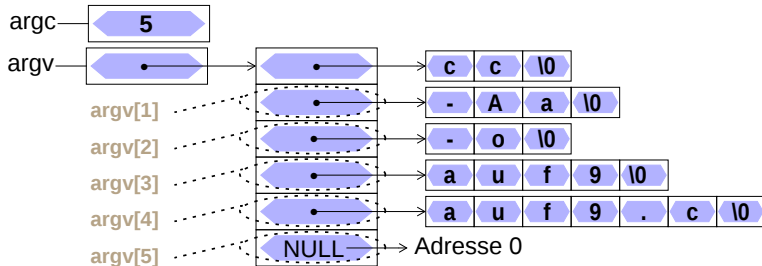
# Zugriff

## Beispiel: Ausgeben aller Argumente (1)

- 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%c", argv[i], (i < argc-1) ? ' ':'\n' );  
    }  
    ...  
}
```

1. Version



# Zugriff

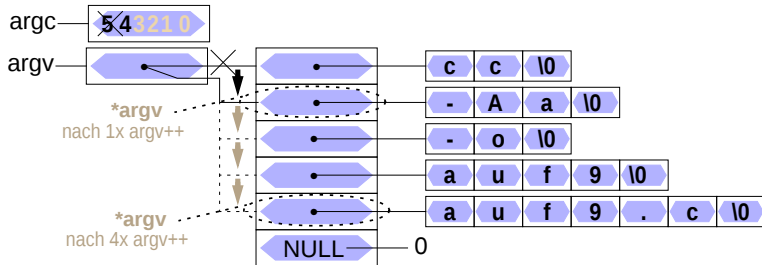
## Beispiel: Ausgeben aller Argumente (2)

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

```
int  
main (int argc, char **argv)  
{ while (--argc > 0) {  
    argv++;  
    printf("%s%c", *argv, (argc>1) ? ' ' : '\n' );  
}  
...
```

linkseitiger Operator:  
erst dekrementieren,  
dann while-Bedingung prüfen  
→ Schleife läuft für argc=4,3,2,1

2. Version



# Verbund-Datentypen / Strukturen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit
- 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



# 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



# Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

**17 Betriebssysteme I**

**18 Dateisysteme**

**19 Prozesse I**

**20 Speicherorganisation**

**21 Prozesse II**

**22 Betriebssysteme II**



## Was sind Betriebssysteme?

### ■ DIN 44300

- „...die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechanlage 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].“

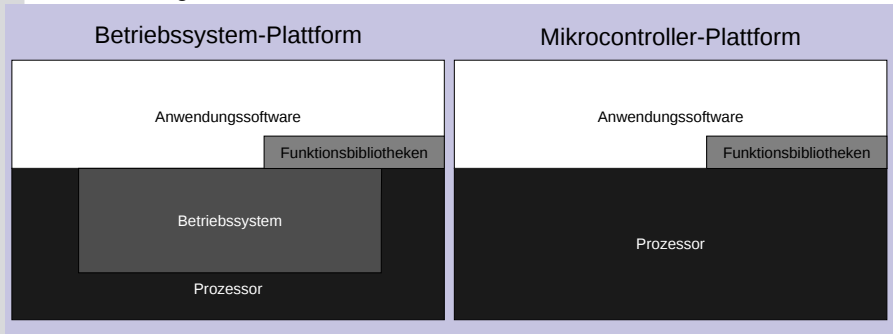
### ★ Zusammenfassung:

- Software zur Verwaltung und Virtualisierung der Hardwarekomponenten (Betriebsmittel)
- Programm zur Steuerung und Überwachung anderer Programme



# Betriebssystem-Plattform vs. Mikrocontroller

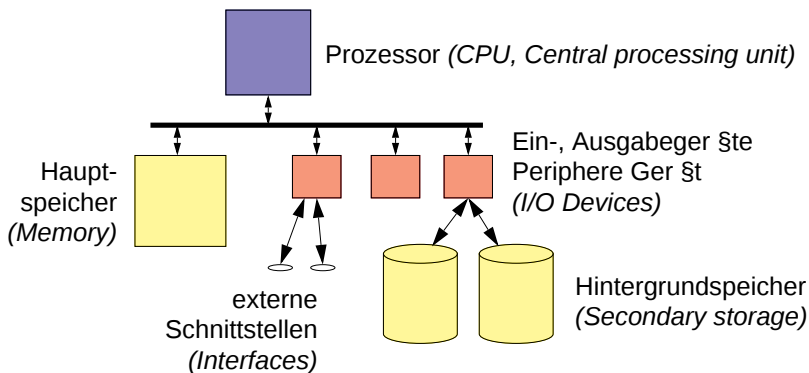
- Entscheidende Unterschiede:
  - Betriebssystem bietet zusätzliche Softwareinfrastruktur für die Ausführung von Anwendungen



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



# Verwaltung von Betriebsmitteln



# Verwaltung von Betriebsmitteln (2)

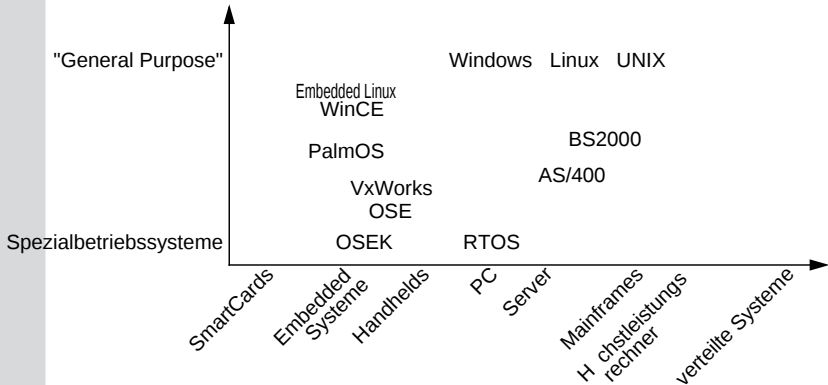
- Resultierende Aufgaben
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel
- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar 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 Klassifikationskriterien

- 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



- 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

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

**18 Dateisysteme**

**19 Prozesse I**

**20 Speicherorganisation**

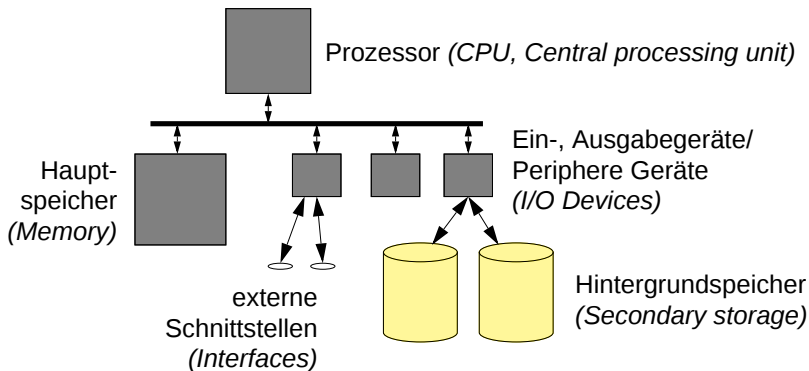
**21 Prozesse II**

**22 Betriebssysteme II**



## Allgemeine Konzepte

### ■ Einordnung



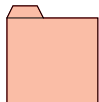
# Allgemeine Konzepte (2)

- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Betriebssystemabstraktion zur Nutzung von Hintergrundspeichern (z.B. Platten, CD-ROM, Bandlaufwerke)
    - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
    - einheitliche Sicht auf den Hintergrundspeicher
- Dateisysteme bestehen aus
  - Dateien (*Files*)
  - Katalogen (*Directories*)
  - Partitionen (*Partitions*)

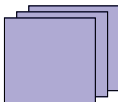


# Allgemeine Konzepte (3)

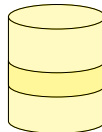
- Datei
  - speichert Daten oder Programme
- Katalog / Verzeichnis (*Directory*)
  - erlaubt Benennung der Dateien
  - enthält Zusatzinformationen zu Dateien
- Partitionen
  - eine Menge von Katalogen und deren Dateien
  - sie dienen zum physischen oder logischen Trennen von Dateimengen.



Katalog



Dateien



Partition



## Überblick

---

- 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, Umlenkung durch **<**
    - Dateiende (**EOF**) wird durch Eingabe von **CTRL - D** am Zeilenanfang signalisiert
  - **stdout** Standardausgabe
    - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden, Umlenkung durch **>**
  - **stderr** Ausgabekanal für Fehlermeldungen
    - normalerweise ebenfalls mit Bildschirm verbunden
- 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`

- Prototyp:

```
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() {
    FILE *eingabe;
    char dateiname[256];

    printf("Dateiname: ");
    scanf("%s\n", dateiname);

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

    ... /* Programm kann jetzt von eingabe lesen */
    ... /* z. B. mit c = getc(eingabe) */
}
```

## ■ 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( )
```

➤ 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

■ von einem Dateikanal

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

## ■ Schreiben eines einzelnen Zeichens

■ auf die Standardausgabe

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

➤ schreiben das im Parameter **c** übergeben Zeichen

➤ geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

■ auf einen Dateikanal

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



# Zeichenweise Lesen und Schreiben (2)

## ■ Beispiel: copy-Programm

```
#include <stdio.h>
```

Teil 1: Dateien öffnen

```
int main() {
    FILE *quelle;
    FILE *ziel;
    char quelldatei[256], zieldatei[256];
    int c;                                /* gerade kopiertes Zeichen */

    printf("Quelldatei und Zieldatei eingeben: ");
    scanf("%s %s\n", quelldatei, zieldatei);

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

    if ((ziel = fopen(zieldatei, "w")) == NULL) {
        perror(zieldatei); /* Fehlermeldung ausgeben */
        exit(1);           /* Programm abbrechen */
    }

    /* ... */
}
```



# Zeichenweise Lesen und Schreiben (3)

... Beispiel: copy-Programm  
— Fortsetzung

```
/* ... */                                     Teil 2: kopieren  
  
while ( (c = getc(quelle)) != EOF ) {  
    putc(c, ziel);  
}  
  
fclose(quelle);  
fclose(ziel);  
}
```



## ■ Bibliotheksfunktionen — Prototypen (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  
-> potentielle Sicherheitsprobleme!)
- **snprintf** arbeitet analog, schreibt aber maximal nur **n** Zeichen  
(**n** sollte natürlich nicht größer als die Feldgröße sein)



- 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** oder **double** Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
- %e** **float** oder **double** 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



## ■ Bibliotheksfunktionen — Prototypen (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, hier nur Kurzüberblick  
für Details siehe Manual-Seiten



# Formatierte Eingabe — Eingabe-Daten

- *White space* (Space, Tabulator oder Newline \n) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
  - *white space* wird in beliebiger Menge einfach überlesen
  - Ausnahme: bei Format-Anweisung **%c** wird auch *white space* eingelesen
- Alle anderen Daten in der Eingabe müssen zum **format**-String passen oder die Interpretation der Eingabe wird abgebrochen
  - wenn im format-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
  - wenn im Format-String eine Format-Anweisung (%...) angegeben ist, muß in der Eingabe etwas hierauf passendes auftauchen
    - ➡ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen
- Die **scanf**-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte
- Detail siehe Manual-Seite (man scanf)



## ■ Datei

- einfache, unstrukturierte Folge von Bytes
- beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
- dynamisch erweiterbar

## ■ Katalog

- baumförmig strukturiert
  - Knoten des Baums sind Kataloge
  - Blätter des Baums sind Verweise auf Dateien
- jedem UNIX-Prozess ist zu jeder Zeit ein aktueller Katalog (*Current working directory*) zugeordnet

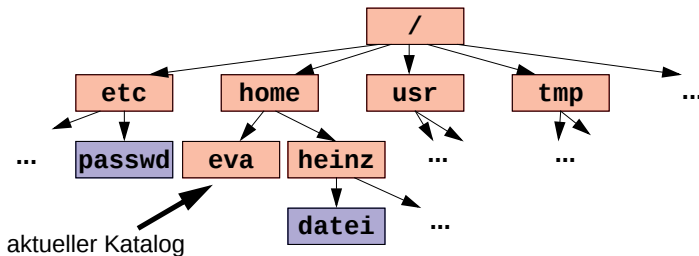
## ■ Partitionen

- jede Partition enthält einen eigenen Dateibaum
- Bäume der Partitionen werden durch "mounten" zu einem homogenen Dateibaum zusammengebaut (Grenzen für Anwender nicht sichtbar!)



# Pfadnamen

## Baumstruktur



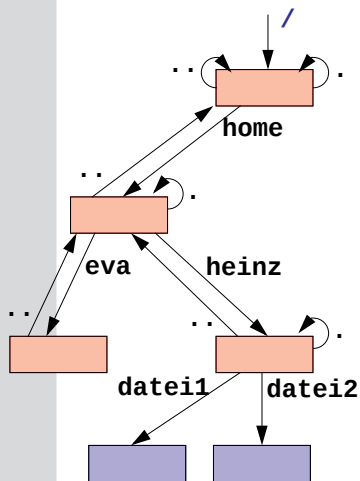
## Pfade

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



# Pfadnamen (2)

## ■ Eigentliche Baumstruktur



▲ benannt sind nicht Dateien und Kataloge, sondern die Verbindungen (*Links*) zwischen ihnen

- Kataloge und Dateien können auf verschiedenen Pfaden erreichbar sein  
z. B. **../heinz/datei1** und **/home/heinz/datei1**
- Jeder Katalog enthält
  - einen Verweis auf sich selbst (.) und
  - einen Verweis auf den darüberliegenden Katalog im Baum (..)
  - Verweise auf Dateien



## ■ Kataloge verwalten

### ■ Erzeugen

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

### ■ Löschen

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

## ■ Kataloge lesen (Schnittstelle der C-Bibliothek)

### ➤ Katalog öffnen:

```
DIR *opendir( const char *path );
```

### ➤ Katalogeinträge lesen:

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

### ➤ Katalog schließen:

```
int closedir( DIR *dirp );
```



# Kataloge (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**



# Kataloge (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



# Kataloge (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];  
};
```

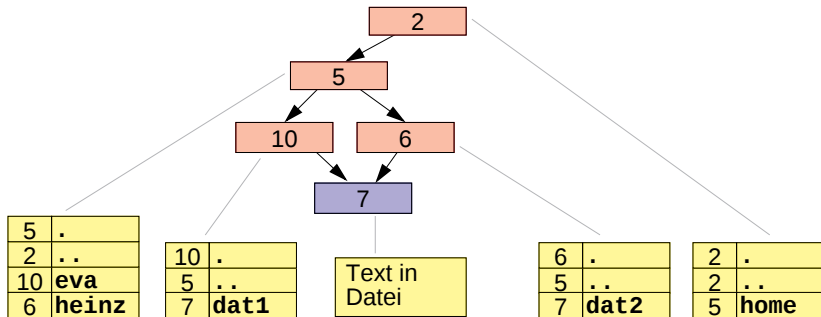
## Programmierschnittstelle für Dateien

- 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



# Inodes

- Attribute (Zugriffsrechte, Eigentümer, etc.) einer Datei und Ortsinformation über ihren Inhalt werden in **Inodes** gehalten
  - Inodes werden pro Partition numeriert (*Inode number*)
- Kataloge enthalten lediglich Paare von Namen und Inode-Nummern
  - Kataloge bilden einen hierarchischen Namensraum über einem eigentlich flachen Namensraum (durchnummerierte Dateien)



# Inodes (2)

---

- Inhalt eines Inode
  - Dateityp: Katalog, normale Datei, Spezialdatei (z.B. Gerät)
  - Eigentümer und Gruppe
  - Zugriffsrechte
  - Zugriffszeiten: letzte Änderung (*mtime*), letzter Zugriff (*atime*), letzte Änderung des Inodes (*ctime*)
  - Anzahl der Hard links auf den Inode
  - Dateigröße (in Bytes)
  - Adressen der Datenblöcke des Datei- oder Kataloginhalts



# Inodes — Programmierschnittstelle: stat / lstat

- liefert Datei-Attribute 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);
```



# Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

**19 Prozesse I**

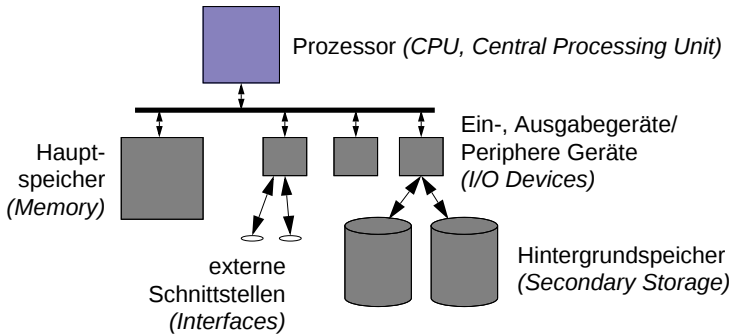
**20 Speicherorganisation**

**21 Prozesse II**

**22 Betriebssysteme II**



## ■ Einordnung



- 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
```

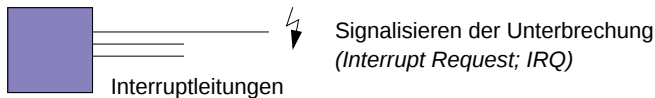
## ■ Prozessor arbeitet in einem bestimmten Modus

- Benutzermodus: eingeschränkter Befehlssatz
- privilegierter Modus: erlaubt Ausführung privilegierter Befehle
  - Konfigurationsänderungen des Prozessors
  - Moduswechsel
  - spezielle Ein-, Ausgabebefehle



# Prozessor (3)

## ■ Unterbrechungen (*Interrupts*)



### ■ 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 2 GB (oder mehr bis 4 GB)
  - 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



## ■ 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)

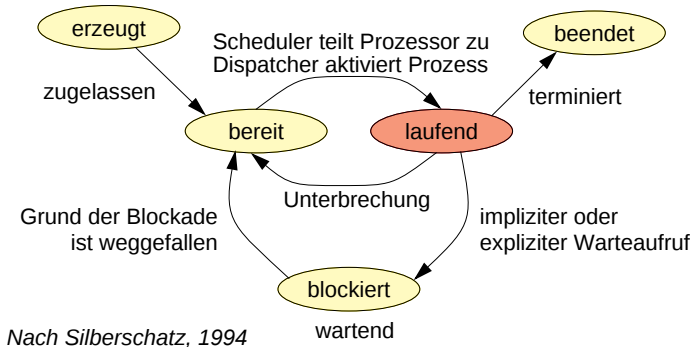


- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** (*New*)  
Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** (*Ready*)  
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** (*Running*)  
Prozess wird 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



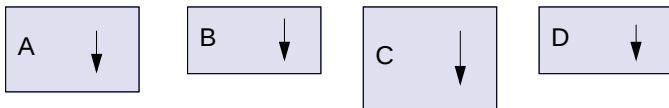
# Prozesszustände (2)

## ■ Zustandsdiagramm



# 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.



## K1.pdf: 2011-07-05



## ■ 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 );
```

```
pid_t p;                Vater
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
}
```



# Prozesserzeugung (UNIX)

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

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

```
pid_t p;                                Vater
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
}
```

```
pid_t p;                                Kind
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* 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

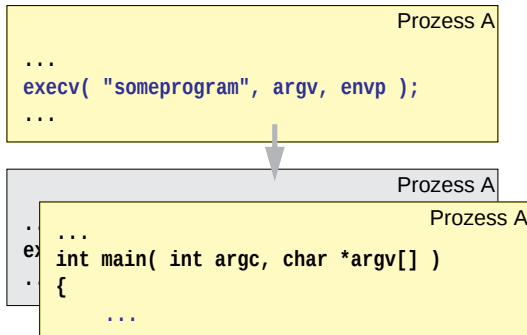
```
...  
execv( "someprogram", argv, envp );  
...
```



# 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.



# 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 );
```



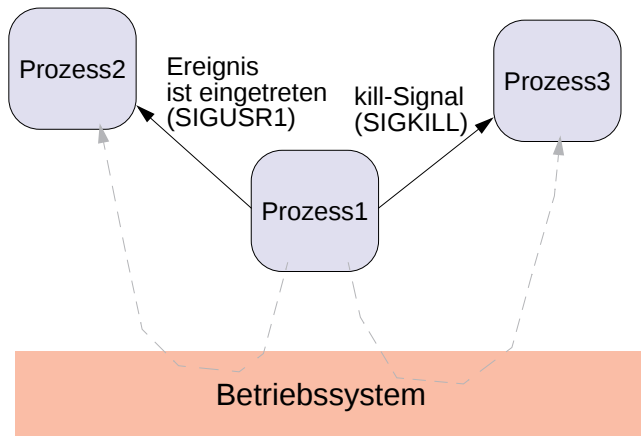
## Signalisierung des Systemkerns an einen Prozess

- 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

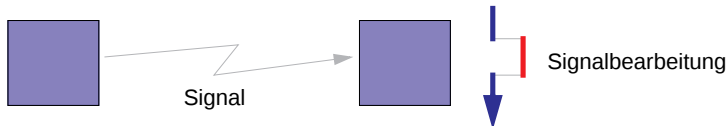
---

- 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

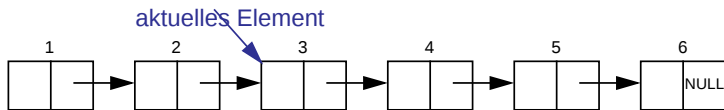
- sigaction – Anmelden einer Funktion = Einrichten der ISR-Tabelle
- sigprocmask – Blockieren/Freigeben von Signalen  $\approx$  cli() / sei()
- sigsuspend – Freigeben + passives Warten auf Signal + wieder Blockieren  $\approx$  sei() + sleep\_cpu() + cli()
- kill – Signal an anderen Prozess verschicken



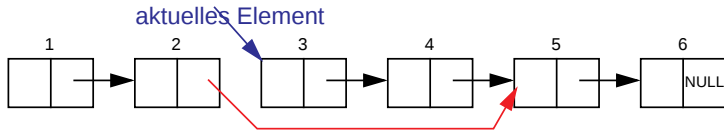
# Signale und Nebenläufigkeit → Race Conditions

- 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



# Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

**20 Speicherorganisation**

**21 Prozesse II**

**22 Betriebssysteme II**



```
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 globale und modullokalen Variablen, sowie den Code
- Allokation durch Platzierung in einer [Sektion](#)

|                      |                                                                    |                     |
|----------------------|--------------------------------------------------------------------|---------------------|
| <code>.text</code>   | – enthält den Programmcode                                         | <code>main()</code> |
| <code>.bss</code>    | – enthält alle uninitialisierten / mit 0 initialisierten Variablen | <code>a</code>      |
| <code>.data</code>   | – enthält alle initialisierten Variablen                           | <code>b,s</code>    |
| <code>.rodata</code> | – enthält alle initialisierten unveränderlichen Variablen          | <code>c</code>      |

## ■ Dynamische Allokation – Reservierung zur Laufzeit

- Betrifft lokale Variablen und explizit angeforderten Speicher

|                    |                                                                            |                    |
|--------------------|----------------------------------------------------------------------------|--------------------|
| <code>Stack</code> | – enthält alle <b>aktuell gültigen</b> lokalen Variablen                   | <code>x,y,p</code> |
| <code>Heap</code>  | – enthält explizit mit <code>malloc()</code> angeforderte Speicherbereiche | <code>*p</code>    |



# Speicherorganisation auf einem $\mu\text{C}$

```
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)
}
```

compile / link

Quellprogramm

| Symbol Table | <a>        |
|--------------|------------|
| .data        | s=3<br>b=1 |
| .rodata      | c=2        |
| .text        | main       |
| ...          |            |
| ELF Header   |            |

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 .rodata.



# Speicherorganisation auf einem $\mu\text{C}$

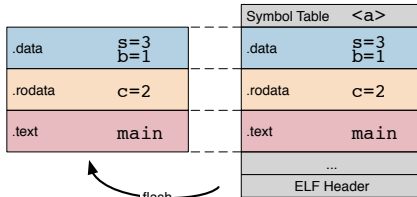
```
int a;           // a: global, uninitialized
int b = 1;       // b: global, initialized
const int c = 2; // c: global, const

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)
}
```

compile / link

Quellprogramm

Flash / ROM



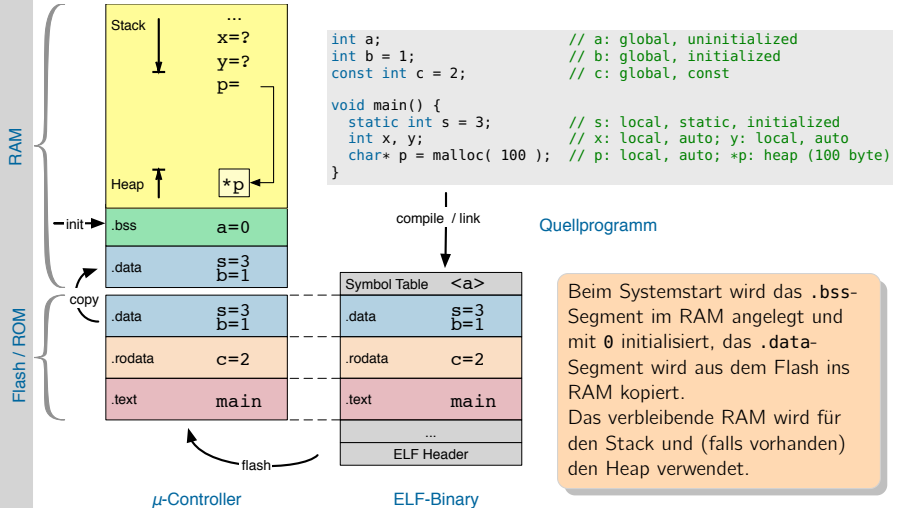
$\mu\text{-Controller}$

ELF-Binary

Zur Installation auf dem  $\mu\text{C}$  werden `.text` und `.[ro]data` in den Flash-Speicher des  $\mu\text{C}$  geladen.



# Speicherorganisation auf einem $\mu\text{C}$



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  $\leftrightarrow$  14-3), so werden konstante Variablen ebenfalls in `.data` abgelegt (und belegen zur Laufzeit RAM).

- **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**, ...) → 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



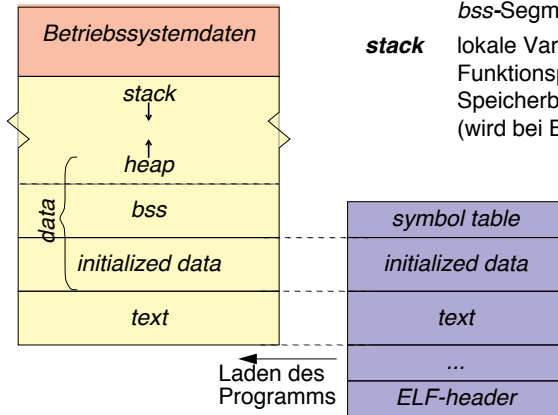
# Speicherorganisation in einem UNIX-Prozess (Forts.)

**text**    Programmcode  
**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)**)

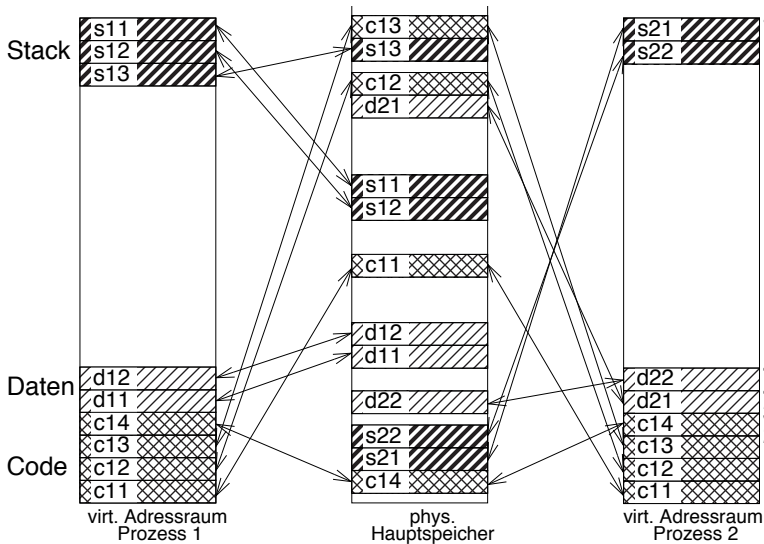
**stack**    lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

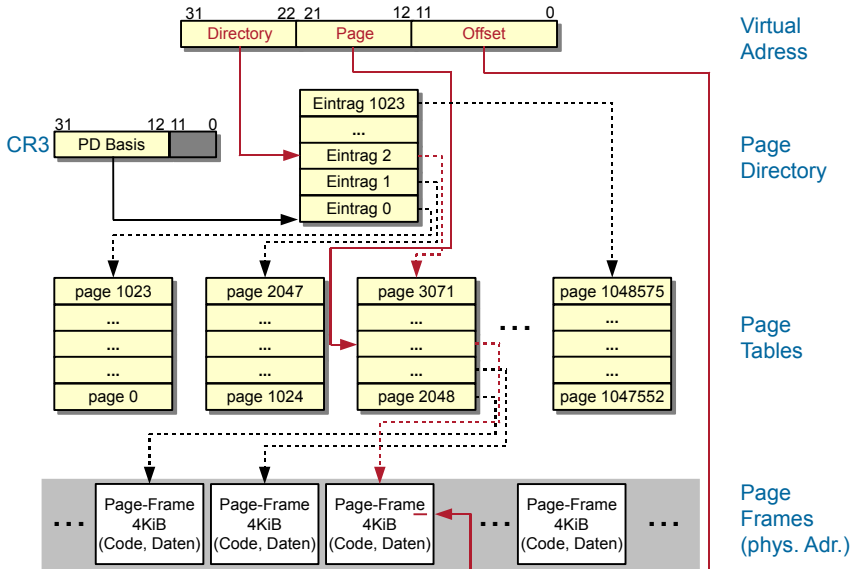


Aufbau eines Programms  
ELF-Format)

- Die Abbildung von virtuellem Speicher (VS) auf physikalischen Speicher (PS) erfolgt durch **Seitenadressierung** (*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







# 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

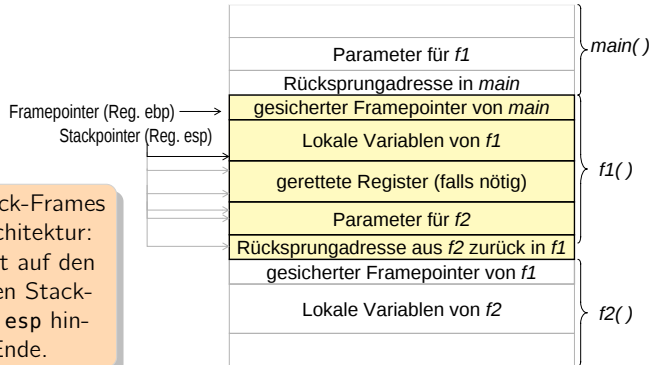
```
#include <stdlib.h>

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! **)
    }
}
```



- 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*

sp fp

|             |       |
|-------------|-------|
| return-addr | ←2000 |
| fp retten   | ←1996 |
| a           | ←1992 |
| b           | ←1988 |
| c           | ←1984 |
|             | ←1980 |
|             | ←1976 |
|             | ←1972 |
|             | ←1968 |
|             | ←1964 |
|             | ←1960 |
|             | ←1956 |
|             | ←1952 |
|             | ←1948 |
|             | ←1944 |
|             | ←1940 |
|             | ←1936 |
|             | ←1932 |

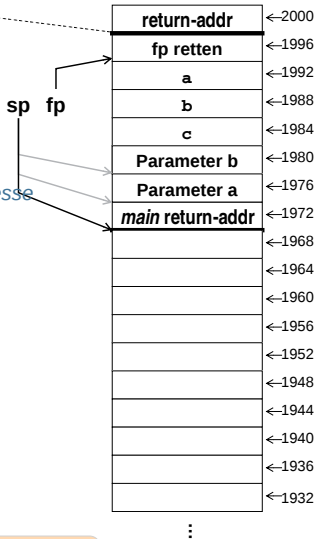
Beispiel hier für 32-Bit-Architektur (4-Byte ints), main() wurde soeben betreten



# Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

*Parameter  
auf Stack legen  
Bei Aufruf  
Rücksprungadresse  
auf Stack legen*



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

|                  |       |
|------------------|-------|
| return-addr      | ←2000 |
| fp retten        | ←1996 |
| a                | ←1992 |
| b                | ←1988 |
| c                | ←1984 |
| Parameter b      | ←1980 |
| Parameter a      | ←1976 |
| main return-addr | ←1972 |
| main-fp (1996)   | ←1968 |
| i [2]            | ←1964 |
| i [1]            | ←1960 |
| i [0]            | ←1956 |
| n                | ←1952 |
|                  | ←1948 |
|                  | ←1944 |
|                  | ←1940 |
|                  | ←1936 |
|                  | ←1932 |

⋮

f1() wurde soeben betreten



# Stack-Aufbau bei Funktionsaufrufen

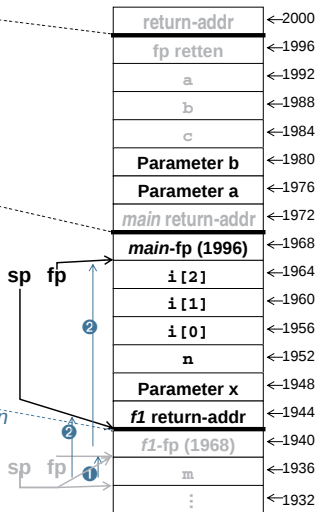
```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

```
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

- ①  $sp = fp$
- ②  $fp = pop(sp)$



f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

# Stack-Aufbau bei Funktionsaufrufen

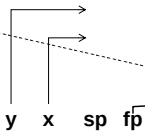
```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

```
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);  
}
```

Rücksprung  
③ return

|                  |       |
|------------------|-------|
| return-addr      | ←2000 |
| fp retten        | ←1996 |
| a                | ←1992 |
| b                | ←1988 |
| c                | ←1984 |
| Parameter b      | ←1980 |
| Parameter a      | ←1976 |
| main return-addr | ←1972 |
| main-fp (1996)   | ←1968 |
| i [2]            | ←1964 |
| i [1]            | ←1960 |
| i [0]            | ←1956 |
| n                | ←1952 |
| Parameter x      | ←1948 |
| f1 return-addr   | ←1944 |
| f1-fp (1968)     | ←1940 |
| m                | ←1936 |
| ⋮                | ←1932 |

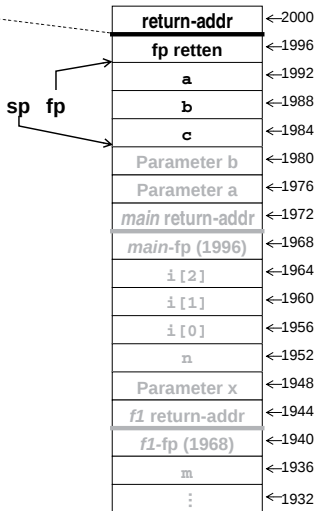


f2() wird verlassen



# Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```



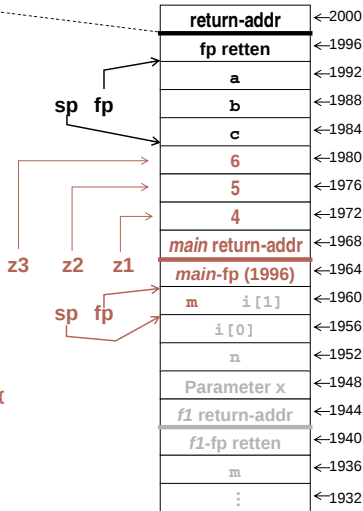
zurück in main()

# 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! → 1-3)

```
lohmann@fau148a:~$ 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



- 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
- Speicher für Eingabedaten möglichst auf dem Heap anfordern
  - Das **Risiko von Programmierfehlern und Speicherlecks bleibt!**



# Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

20 Speicherorganisation

**21 Prozesse II**

**22 Betriebssysteme II**



- 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

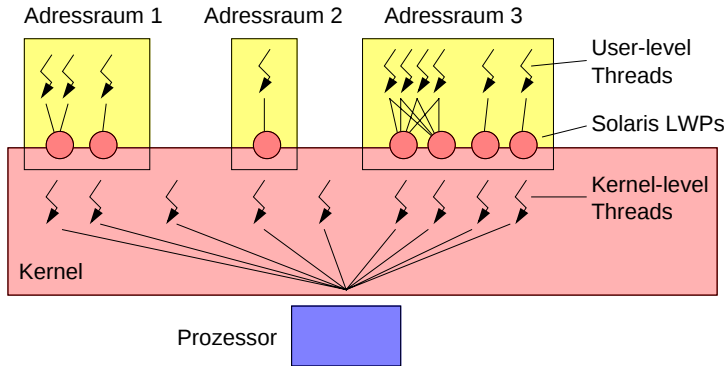
## ▲ Nachteile

- 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



- 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!



## ■ 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), andere 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_threads();  
    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();  
}
```

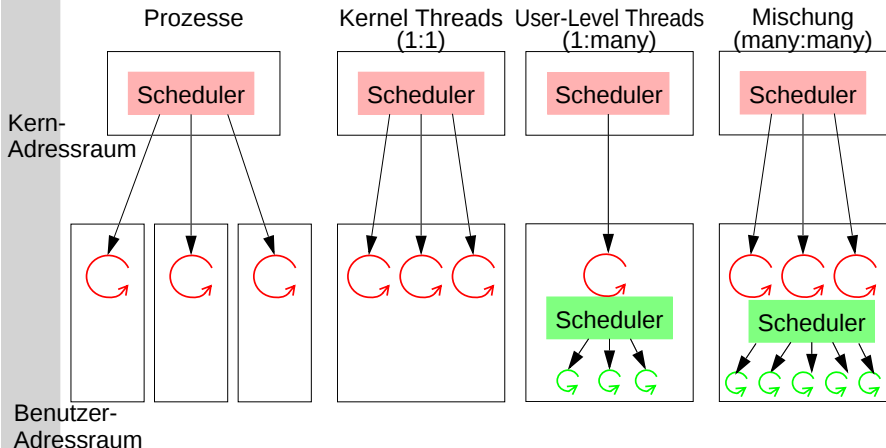
```
void unlock (mutex *m) {  
    enter_OS(); cli();  
    spin_lock();  
    *m = 1;  
    wakeup_waiting_threads();  
    spin_unlock();  
    sei(); leave_OS();  
}
```



- 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

## ■ Pthreads-Schnittstelle (Basisfunktionen):

|                       |                                                    |
|-----------------------|----------------------------------------------------|
| <i>pthread_create</i> | Thread erzeugen & Startfunktion angeben            |
| <i>pthread_exit</i>   | Thread beendet sich selbst                         |
| <i>pthread_join</i>   | Auf Ende eines anderen Threads warten              |
| <i>pthread_self</i>   | Eigene Thread-Id abfragen                          |
| <i>pthread_yield</i>  | 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

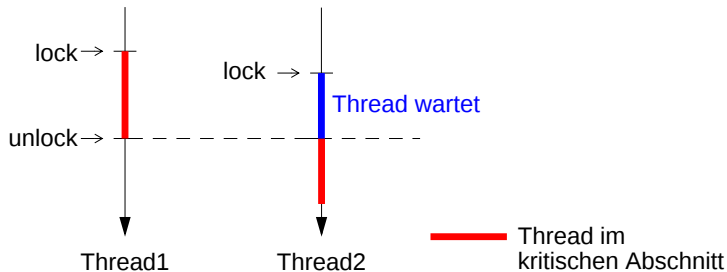
- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphore-Operationen zur Verfügung
  - Implementierung durch den Systemkern
  - komplexe Datenstrukturen, aufwändig zu programmieren
  - für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache **Mutex**-Variablen
  - gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*



# Pthreads-Koordinierung (2)

## ★ Mutexes

### ■ Koordination von kritischen Abschnitten



...

## 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);
```



## ... 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

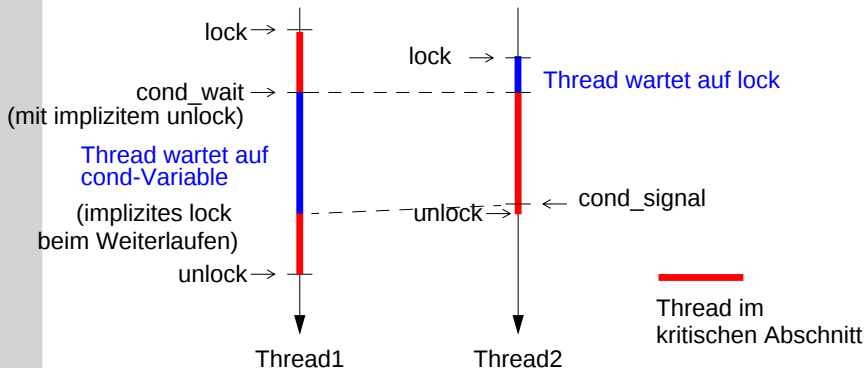
➡ **Condition Variables**



# 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



...

## 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



...

## 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);  
}
```



## ... 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 höchstpriorie 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



# Vorläufiges Ende...

---

Die weiteren Kapitel sind leider noch nicht verfügbar

