

# Grundlagen der Systemnahen Programmierung in C (GSPiC)

**Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2018

[http://www4.cs.fau.de/Lehre/SS18/V\\_GSPiC](http://www4.cs.fau.de/Lehre/SS18/V_GSPiC)



- [1] *ATmega328PB 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. Atmel Corporation. Okt. 2015.
- [GDI] Frank Bauer. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2015 (jährlich). URL: <https://gdi.cs.fau.de/w15/material>.
- [2] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop u. a. *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: <https://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1>.
- [3] Brian W. Kernighan und Dennis MacAlistair Ritchie. *The C Programming Language*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1978.
- [4] Brian W. Kernighan und Dennis MacAlistair Ritchie. *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1988. ISBN: 978-8120305960.
- [5] Dennis MacAlistair Ritchie und Ken Thompson. "The Unix Time-Sharing System". In: *Communications of the ACM* 17.7 (Juli 1974), S. 365–370. DOI: 10.1145/361011.361061.



- [6] David Tennenhouse. "Proactive Computing". In: *Communications of the ACM* (Mai 2000), S. 43–45.
- [7] Jim Turley. "The Two Percent Solution". In: *embedded.com* (Dez. 2002). <http://www.embedded.com/story/0EG2002121750039>, visited 2011-04-08.



# Veranstaltungsüberblick

## Teil A: Konzept und Organisation

1 Einführung

2 Organisation

## Teil B: Einführung in C

3 Java versus C

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

15 Nebenläufigkeit

16 Speicherorganisation



# Grundlagen der Systemnahen Programmierung in C (GSPiC)

## Teil A Konzept und Organisation

**Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2018

[http://www4.cs.fau.de/Lehre/SS18/V\\_GSPiC](http://www4.cs.fau.de/Lehre/SS18/V_GSPiC)



# Überblick: Teil A Konzept und Organisation

## 1 Einführung

- Ziele der Lehrveranstaltung
- Warum  $\mu$ -Controller?
- Warum C?
- Literatur

## 2 Organisation

- Vorlesung
- Übung
- Lötabend
- Prüfung
- Semesterüberblick

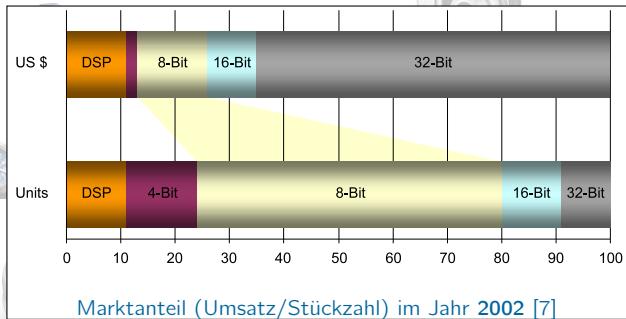


- **Vertiefen** des Wissens über Konzepte und Techniken der Informatik für die Softwareentwicklung
  - Ausgangspunkt: Grundlagen der Informatik (GdI)
  - Schwerpunkt: Systemnahe Softwareentwicklung in C
- **Entwickeln** von Software in C für einen  $\mu$ -Controller ( $\mu$ C)
  - SPiCboard-Lehrentwicklungsplattform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



# 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 **mehr als 10 Jahre** alt!
- Man kann dennoch davon ausgehen, dass die **relativen Größenordnungen** nach wie vor stimmen
  - 2016 liegt der Anteil an 8-Bitern (vermutlich) noch bei 40 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	SPI	ADC	PWM	EUR
ATTINY13	1 KiB	64 B	6	1/-	-	-	-	1*4	-	0,86
ATTINY2313	2 KiB	128 B	18	1/1	-	1	-	-	-	0,99
ATMEGA48	4 KiB	512 B	23	2/1	1	1	8*10	6	1,40	
ATMEGA16	16 KiB	1024 B	32	2/1	1	1	8*10	4	2,05	
ATMEGA32	32 KiB	2048 B	32	2/1	1	1	8*10	4	3,65	
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	8	5,70	
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	8	7,35	
ATMEGA256	256 KiB	8192 B	86	2/2	4	1	16*10	16	8,99	

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

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



# Motivation: Die Sprache C

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



~> **C** ist die **lingua franca** der systemnahen Softwareentwicklung!



- **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
- **Herausforderung:** Umfang der Veranstaltung (nur 2,5 ECTS)
  - Für Vorlesung und Übung eigentlich zu wenig
  - Veranstaltung soll trotzdem einen **hohen praktischen Anteil** haben
- **Ansatz:** Konzentration auf die Domäne  $\mu$ -Controller
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (EEI)



- 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!**



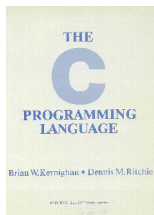
## [2] Für den Einstieg empfohlen:

Manfred Dausmann, Ulrich Bröckl, Dominic Schoop u. a. *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: <https://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1>



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

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



# Überblick: Teil A Konzept und Organisation

1 Einführung

**2 Organisation**



- 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
- Termin: Montag, 12:15 – 13:45, H8
  - insgesamt 9 Vorlesungstermine

 2-8





- Kombinierte Tafel- und Rechnerübung (jeweils im Wechsel)
  - Tafelübungen
    - Ausgabe und Erläuterung der Programmieraufgaben
    - Gemeinsame Entwicklung einer Lösungsskizze
    - Besprechung der Lösungen
  - Rechnerübungen
    - selbstständige Programmierung
    - Umgang mit Entwicklungswerkzeug (Atmel Studio)
    - Betreuung durch Übungsbetreuer
- Termin: 8 Gruppen zur Auswahl
  - Anmeldung über Waffel (siehe Webseite) ab heute (Montag), 19:30 nach dem Windhundverfahren.
  - 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 gültiger Login im Linux-CIP gebraucht!



# Programmieraufgaben

- Praktische Umsetzung des Vorlesungsstoffs
  - Fünf Programmieraufgaben ↔ 2-8
  - Bearbeitung teilweise 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**; es können jedoch bis zu **10% Bonuspunkte** für die Prüfungsklausur erarbeitet werden!

↔ 2-7

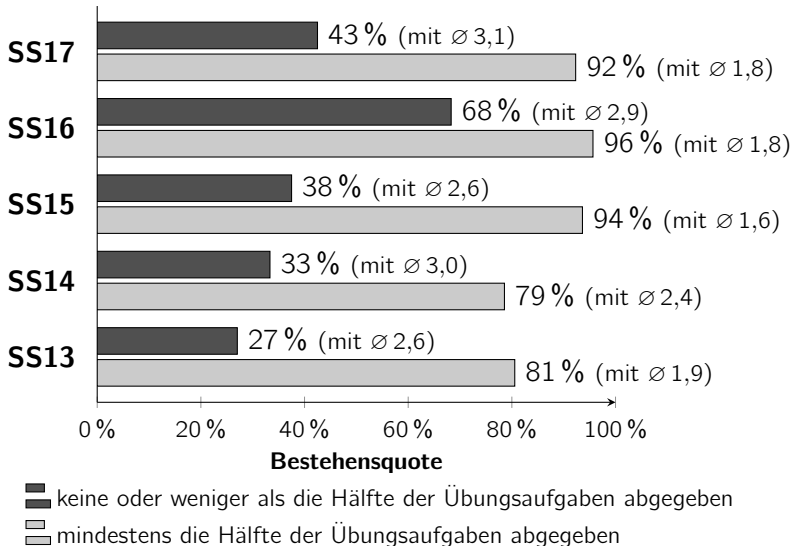
Plagiate können zum **Verlust aller Bonuspunkte** führen.

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



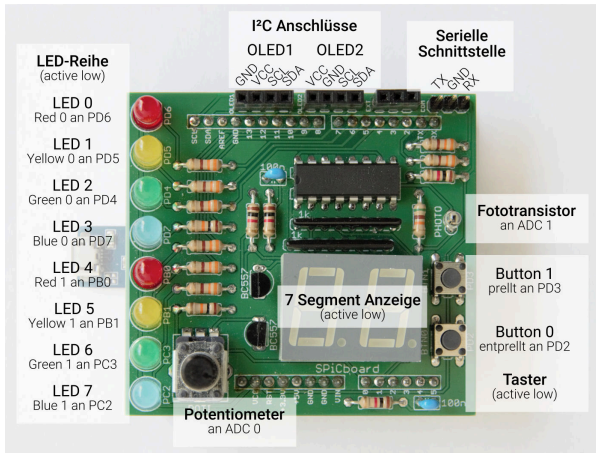
# Bestehensquote der Klausur

nach Aktivität der Teilnehmer bei den Übungsaufgaben



# Übungsplattform: Das SPiCboard

- ATmega328- $\mu$ C
- USB-Anschluss
- 8 LEDs
- 2 7-Seg-Elemente
- 2 Taster
- 1 Potentiometer
- 1 Fotosensor
- optional:
  - OLED Display



- Ausleihe während Rechnerübung möglich
- Oder noch besser  $\leftrightarrow$  selber Löten



- Die FSI EEI, FSI ME sowie das FabLab bieten einen „Lötabend“ für die Teilnehmer der Veranstaltung an
  - Teilnahme ist freiwillig
  - (Erste) Löterfahrung sammeln beim Löten eines eigenen SPiCboards
  - Voraussichtlich insgesamt 6 Termine (in KW 16 & 17)
- **Anmeldung:** über Waffel **notwendig**, da begrenzte Plätze:  
Morgen (Dienstag), 20:15 – Freitag, 11:00
- **Kostenbeitrag:** kostenlos (für Teilnehmer dieser Veranstaltung),  
finanziert aus Studienzuschüssen

**Der bei der Anmeldung gewählte Termin ist verbindlich!**



- Prüfung (Klausur)
  - Termin: voraussichtlich Ende Juli / Anfang August
  - Dauer: 60 min (bzw. 90 min bei InfoEEI mit GdI)
  - Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe
- Klausurnote  $\mapsto$  Modulnote
  - Bestehensgrenze (in der Regel): 50% der möglichen Klausurpunkte (KP)
  - Falls **bestanden** ist eine Notenverbesserung möglich durch Bonuspunkte aus den Programmieraufgaben
    - Basis (Minimum): 50% der möglichen Übungspunkte (ÜP)
    - Jede weiteren 5% der möglichen ÜP  $\mapsto$  +1% der möglichen KP
    - $\rightsquigarrow$  100% der möglichen ÜP  $\mapsto$  +10% der möglichen KP



## Semesterplanung

KW	Mo	Di	Mi	Do	Fr	Themen	Kapitel im Skript
15	09.04. VL 1	10.04.	11.04.	12.04.	13.04.	Einführung, Organisation, Java nach C	1, 2, 3
16	16.04. VL 2	17.04.	18.04. A1 (Blink)	19.04.	20.04.	Abstraktion, Sprachüberblick, Datentypen	4, 5, 6
17	23.04. VL 3	24.04.	25.04. A2 (Snake)	26.04.	27.04.	Ausdrücke, Kontrollstrukturen, Funktionen	7, 8, 9
18	30.04. VL 4	01.05. Feiertag	02.05.	03.05.	04.05.	Variablen, Präprozessor, Programmstruktur, Module	10, 11, 12
19	07.05. VL 5	08.05.	09.05.	10.05. Himmelf.	11.05.	Zeiger	13
20	14.05. VL 6	15.05.	16.05. A3 (LED-Modul)	17.05.	18.05.	Mikrocontroller-Systemarchitektur, volatile, Verbundtypen	14
21	21.05. Pfingsten/Berg	22.05.	23.05.	24.05.	25.05.		
22	28.05.	29.05.	30.05.	31.05. Fronleich.	01.06.		
23	04.06. VL 7	05.06.	06.06. A4 (Spiel)	07.06.	08.06.	Interrupts, Nebenläufigkeit	15
24	11.06.	12.06.	13.06.	14.06.	15.06.		
25	18.06. VL 8	19.06.	20.06. A5 (Ampel)	21.06.	22.06.	Speicherorganisation, Zusammenfassung	16
26	25.06.	26.06.	27.06.	28.06.	29.06.		
27	02.07.	03.07.	04.07.	05.07.	06.07.	Wiederholung / Bonusaufgabe	
28	09.07. VL 9	10.07.	11.07.	12.07.	13.07.	Fragestunde	

# Bei Fragen oder Problemen

---

- Vorlesungs- und Übungsfolien konsultieren
- Häufig gestellte Fragen (FAQ) und Antworten siehe Webseite  
→ [http://www4.cs.fau.de/Lehre/SS18/V\\_GSPIC](http://www4.cs.fau.de/Lehre/SS18/V_GSPIC)  
→ Übungen  
→ FAQ
- Allgemeine Fragen zu Übungsaufgaben etc. im EEI-Forum posten  
→ <https://eei.fsi.uni-erlangen.de/forum/forum/16>
- Bei speziellen Fragen Mail an Mailingliste (alle Übungsleiter)  
→ [i4spic@cs.fau.de](mailto:i4spic@cs.fau.de)





# Grundlagen der Systemnahen Programmierung in C (GSPiC)

## Teil B Einführung in C

**Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2018

[http://www4.cs.fau.de/Lehre/SS18/V\\_GSPiC](http://www4.cs.fau.de/Lehre/SS18/V_GSPiC)



# Überblick: Teil B Einführung in C

**3 Java versus C**

**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)

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

Gar nicht so  
schwer :-)



# Das erste C-Programm – Vergleich mit Java

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

```
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, 01-10]
- 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 6, active low
    DDRD  |= (1<<6); // PD6 is used as output
    PORTD |= (1<<6); // PD6: high --> LED is off

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

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

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

- Übersetzen und **Flashen** (mit Atmel 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 6, active low
5     DDRD  |= (1<<6); // PD6 is used as output
6     PORTD |= (1<<6); // PD6: high --> LED is off
7
8     // greet user
9     PORTD &= ~(1<<6); // PD6: low --> LED is on
10
11    // wait forever
12    while(1){
13    }
14 }
```



- $\mu$ -Controller-Programm zeilenweise erläutert  
(Beachte Unterschiede zur Linux-Version  $\leftrightarrow$  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 6, active low
9     DDRD  |= (1<<6); // PD6 is used as output
10    PORTD |= (1<<6); // PD6: 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<<6); // PD6: 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, ...
  - **Problemnahes** Speichermodell
    - Nur typsichere Speicherzugriffe, automatische Bereinigung zur Laufzeit
- **C:** Effizienz und Leichtgewichtigkeit durch **Maschinennähe**
  - Übersetzung für **konkrete Hardwarearchitektur**
  - **Keine** Überprüfung von Programmfehlern zur Laufzeit
    - Einige Fehler werden vom Betriebssystem abgefangen – **falls vorhanden**
  - **Maschinennahes** Speichermodell
    - Direkter Speicherzugriff durch **Zeiger**
    - Grobgranularer Zugriffsschutz und automatische Bereinigung (auf Prozessebene) durch das Betriebssystem – **falls vorhanden**





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  $\rightsquigarrow$  fehleranfällig, aufwändig

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



# Überblick: Teil B Einführung in C

3 Java versus C

**4 Softwareschichten und Abstraktion**

**5 Sprachüberblick**

**6 Einfache Datentypen**

**7 Operatoren und Ausdrücke**

**8 Kontrollstrukturen**

**9 Funktionen**

**10 Variablen**

**11 Präprozessor**

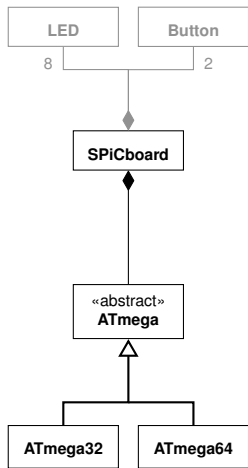


# Abstraktion durch Softwareschichten: SPiCboard

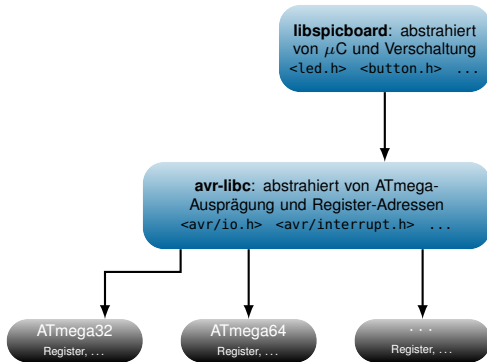
↑ Problemnähe

↓ Maschinennähe

## Hardwareansicht



## 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);
```

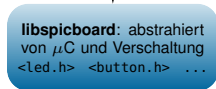
Programm läuft auf **jedem**  $\mu\text{C}$  der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

```
#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:



## Bisher: Entwicklung mit avr-libc

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

void main(void) {
    // initialize hardware

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

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

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

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

(vgl. ↔ 3-8)

## Nun: Entwicklung mit libspicboard

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

void main(void) {

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

    // greet user
    sb_led_on(RED0);

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

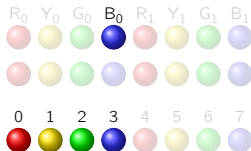
- Hardwareinitialisierung entfällt
- Programm ist einfacher und verständlicher durch **problemspezifische Abstraktionen**
  - Setze Bit 6 in PORTD  
↳ `sb_set_led(RED0)`
  - Lese Bit 2 in PORTD  
↳ `sb_button_getState(BUTTON0)`



## ■ Ausgabe-Abstraktionen (Auswahl)

### ■ LED-Modul (`#include <led.h>`)

- LED einschalten: `sb_led_on(BLUE0)`  $\rightsquigarrow$
- LED ausschalten: `sb_led_off(BLUE0)`  $\rightsquigarrow$
- Alle LEDs ein-/ausschalten:  
`sb_led_set_all_leds(0x0f)`  $\rightsquigarrow$



### ■ 7-Seg-Modul (`#include <7seg.h>`)

- Ganzzahl  $n \in \{-9 \dots 99\}$  ausgeben:  
`sb_7seg_showNumber(47)`  $\rightsquigarrow$



## ■ Eingabe-Abstraktionen (Auswahl)

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

- Button-Zustand abfragen:  
`sb_button_getState(BUTTON0)`  $\mapsto$  `BUTTONSTATE_{PRESSED, RELEASED}`

### ■ ADC-Modul (`#include <adc.h>`)

- Potentiometer-Stellwert abfragen:  
`sb_adc_read(POTI)`  $\mapsto$  `{0 ... 1023}`

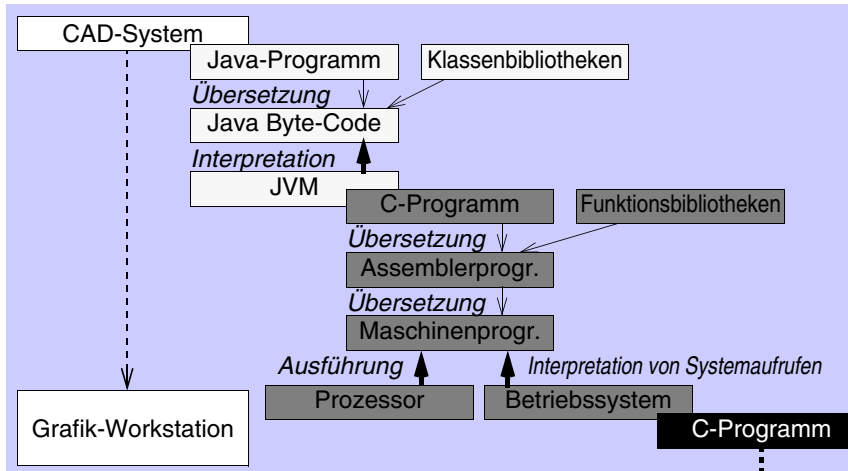


# Softwareschichten im Allgemeinen

↑ Problemnähe

↓ Maschinennähe

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



**Ziel:** Ausführbarer Maschinencode

04-Abstraktion: 2018-04-05



- **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.  $\text{div}/0$ )

*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

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 }
20
21 // main function
22 ... main(...) {
23     ...
24     ...
25     ...
26 }
```

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



# Struktur eines C-Programms – am Beispiel

```
1 // include files
2 #include <led.h>
3
4 // global variables
5 LED nextLED = RED0;
6
7 // subfunction 1
8 LED lightLED(void) {
9     if (nextLED <= BLUE1) {
10         sb_led_on(nextLED++);
11     }
12     return nextLED;
13 }
14 // subfunction 2
15 void wait(void) {
16     volatile unsigned int i;
17     for (i=0; i<0xffff; i++)
18         ;
19 }
20
21 // main function
22 void main() {
23     while (lightLED() < 8) {
24         wait();
25     }
26 }
```

## ■ Ein C-Programm besteht (üblicherweise) aus

- Menge von **globalen Variablen** nextLED, Zeile 5
- Menge von **(Sub-)Funktionen** wait(), Zeile 15
  - Menge von **lokalen Variablen** i, Zeile 16
  - Menge von **Anweisungen** for-Schleife, Zeile 17
- Der Funktion **main()**, in der die Ausführung beginnt



```
1 // include files
2 #include <led.h>
3
4 // global variables
5 LED nextLED = RED0;
6
7 // subfunction 1
8 LED lightLED(void) {
9     if (nextLED <= BLUE1) {
10         sb_led_on(nextLED++);
11     }
12     return nextLED;
13 }
14 // subfunction 2
15 void wait(void) {
16     volatile unsigned int i;
17     for (i=0; i<0xffff; i++)
18         ;
19 }
20
21 // main function
22 void main() {
23     while (lightLED() < 8) {
24         wait();
25     }
26 }
```

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



```

1 // include files
2 #include <led.h>
3
4 // global variables
5 LED nextLED = RED0;
6
7 // subfunction 1
8 LED lightLED(void) {
9     if (nextLED <= BLUE1) {
10         sb_led_on(nextLED++);
11     }
12     return nextLED;
13 }
14 // subfunction 2
15 void wait(void) {
16     volatile unsigned int i;
17     for (i=0; i<0xffff; i++)
18         ;
19 }
20
21 // main function
22 void main() {
23     while (lightLED() < 8) {
24         wait();
25     }
26 }

```

## ■ Reservierte Wörter der Sprache

(↷ dürfen nicht als Bezeichner verwendet werden)

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



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

## ■ (Darstellung von) Konstanten im Quelltext

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



```
1 // include files
2 #include <led.h>
3
4 // global variables
5 LED nextLED = RED0;
6
7 // subfunction 1
8 LED lightLED(void) {
9     if (nextLED <= BLUE1) {
10         sb_led_on(nextLED++);
11     }
12     return nextLED;
13 }
14 // subfunction 2
15 void wait(void) {
16     volatile unsigned int i;
17     for (i=0; i<0xffff; i++)
18         ;
19 }
20
21 // main function
22 void main() {
23     while (lightLED() < 8) {
24         wait();
25     }
26 }
```

- Beschreiben den eigentlichen **Ablauf** des Programms
- Werden hierarchisch komponiert aus drei Grundformen
  - Einzelanweisung – **Ausdruck** gefolgt von **;**
    - einzelnes Semikolon ↦ 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 // subfunction 2
15 void wait(void) {
16     volatile unsigned int i;
17     for (i=0; i<0xffff; i++)
18         ;
19 }
20
21 // main function
22 void main() {
23     while (lightLED() < 8) {
24         wait();
25     }
26 }
```

## ■ Gültige Kombination von Operatoren, Literalen und Bezeichnern

- „Gültig“ im Sinne von Syntax und Typsystem
- Vorrangregeln für Operatoren legen die Reihenfolge fest, ↔ 7-14  
in der Ausdrücke abgearbeitet werden
  - 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

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 ↔ 5-6
  - **Konstante** Bezeichner für einen Wert
  - **Variable** Bezeichner für Speicherplatz, der einen Wert aufnehmen kann
  - **Funktion** Bezeichner für Sequenz von Anweisungen, die einen Wert zurückgibt
- ↪ Literale, Konstanten, Variablen, Funktionen haben einen **(Daten-)Typ**
- Datentyp legt fest
  - Repräsentation der Werte im Speicher
  - Größe des Speicherplatzes für Variablen
  - Erlaubte Operationen
- Datentyp wird festgelegt
  - Explizit, durch Deklaration, Typ-Cast oder Schreibweise (Literale)
  - Implizit, durch „Auslassung“ (↪ `int` schlechter Stil!)



# Primitive Datentypen in C

- Ganzzahlen/Zeichen `char`, `short`, `int`, `long`, `long long` (C99)
  - Wertebereich: implementierungsabhängig [≠Java]  
Es gilt: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`
  - Jeweils als `signed`- und `unsigned`-Variante verfügbar
- Fließkommazahlen `float`, `double`, `long double`
  - Wertebereich: implementierungsabhängig [≠Java]  
Es gilt: `float` ≤ `double` ≤ `long double`
  - Ab C99 auch als `_Complex`-Datentypen verfügbar (für komplexe Zahlen)
- Leerer Datentyp `void`
  - Wertebereich: ∅
- 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/A32	gcc/A64	gcc/AVR
char	16	≥ 8	8	8	8
short	16	≥ 16	16	16	16
int	32	≥ 16	32	32	16
long	64	≥ 32	32	64	32
long long	-	≥ 64	64	64	64

- Der Wertebereich berechnet sich aus der Bitbreite

- signed  $-(2^{Bits-1}-1) \rightarrow +(2^{Bits-1}-1)$
- unsigned  $0 \rightarrow +(2^{Bits}-1)$

Hier zeigt sich die C-Philosophie: Effizienz durch **Maschinennähe**  $\leftrightarrow$  3-14

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





# Integertypen: Maschinennähe → Problemnähe

- **Problem:** Breite ( $\leadsto$  Wertebereich) der C-Standardtypen ist implementierungsspezifisch → **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe → **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 → 255	<code>int8_t</code>	-128 → +127
<code>uint16_t</code>	0 → 65.535	<code>int16_t</code>	-32.768 → +32.767
<code>uint32_t</code>	0 → 4.294.967.295	<code>int32_t</code>	-2.147.483.648 → +2.147.483.647
<code>uint64_t</code>	0 → $> 1,8 * 10^{19}$	<code>int64_t</code>	$< -9,2 * 10^{18}$ → $> +9,2 * 10^{18}$



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

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

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



- Typ-Aliase 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!
```

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

Das entspricht der  
**C-Philosophie!** ↪

3-14



- Fließkommattyp      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“  $\leftrightarrow$  3-14

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

- Oft keine Hardwareunterstützung für **float**-Arithmetik
  - $\rightsquigarrow$  **sehr teure** Emulation in Software (langsam, viel zusätzlicher Code)
- Speicherverbrauch von **float**- und **double**-Variablen ist **sehr hoch**
  - $\rightsquigarrow$  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  $\rightsquigarrow$  man kann mit Zeichen rechnen

```
char b = 'A' + 1;           // b: 'B'
int lower(int ch) {        // lower('X'): 'x'
    return ch + 0x20;
}
```



# ASCII-Code-Tabelle (7 Bit)

ASCII → *American Standard Code for Information Interchange*

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

"Hi!" → 

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

 ← abschließendes 0-Byte

- Beispiel (Linux)

```
#include <stdio.h>

char string[] = "Hello, World!\n";

int main(void) {
    printf("%s", 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)  $\leftrightarrow$  Sequenz von Elementen gleichen Typs [ $\approx$ Java]

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

- Zeiger  $\leftrightarrow$  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  $\leftrightarrow$  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

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

&&	„und“	<i>wahr</i> && <i>wahr</i> $\rightarrow$ <i>wahr</i>
	(Konjunktion)	<i>wahr</i> && <i>falsch</i> $\rightarrow$ <i>falsch</i>
		<i>falsch</i> && <i>falsch</i> $\rightarrow$ <i>falsch</i>

	„oder“	<i>wahr</i>    <i>wahr</i> $\rightarrow$ <i>wahr</i>
	(Disjunktion)	<i>wahr</i>    <i>falsch</i> $\rightarrow$ <i>wahr</i>
		<i>falsch</i>    <i>falsch</i> $\rightarrow$ <i>falsch</i>

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

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

- Operanden (Eingangparameter):  $0 \mapsto$  *falsch*  
 $\neq 0 \mapsto$  *wahr*

- Ergebnis: *falsch*  $\mapsto$  0  
*wahr*  $\mapsto$  1



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

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

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

$\underbrace{a > c}_{0} \ \&\& \ \underbrace{a > b}_{?}$  ← 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 **Nebenwirkungen**, 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!  $\rightsquigarrow$  Fehler wird leicht übersehen!



## ■ Bitweise Verknüpfung von Ganzzahltypen, kommutativ

&	bitweises „Und“ (Bit-Schnittmenge)	$1 \& 1 \rightarrow 1$
		$1 \& 0 \rightarrow 0$
		$0 \& 0 \rightarrow 0$

	bitweises „Oder“ (Bit-Vereinigungsmenge)	$1   1 \rightarrow 1$
		$1   0 \rightarrow 1$
		$0   0 \rightarrow 0$

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

~	bitweise Inversion (Einerkomplement, unär)	$\sim 1 \rightarrow 0$
		$\sim 0 \rightarrow 1$



- 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
---	---	---	---	---	---	---	---

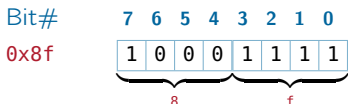
  
PORTD ^= 0x08        

?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

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

$Ausdruck_1 ? Ausdruck_2 : Ausdruck_3$

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

- Beispiel

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



- Reihung von Ausdrücken  
 $Ausdruck_1$  ,  $Ausdruck_2$ 
  - Zunächst wird  $Ausdruck_1$  ausgewertet  
     $\rightsquigarrow$  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 Strukturzugriff Post-Inkrement/-Dekrement	x() x[] x.y x->y x++ x--	links → rechts
2	Prä-Inkrement/-Dekrement unäre Operatoren Adresse, Verweis (Zeiger) Typkonvertierung (cast) Typgröße	++x --x +x -x ~x !x & * (<Typ>)x sizeof(x)	rechts → links
3	Multiplikation, Division, Modulo	* / %	links → rechts
4	Addition, Subtraktion	+ -	links → rechts
5	Bitweises Schieben	>> <<	links → rechts
6	Relationaloperatoren	< <= > >=	links → rechts
7	Gleichheitsoperatoren	== !=	links → rechts
8	Bitweises UND	&	links → rechts
9	Bitweises OR		links → rechts
10	Bitweises XOR	^	links → rechts
11	Konjunktion	&&	links → rechts
12	Disjunktion		links → rechts
13	Bedingte Auswertung	?:=	rechts → links
14	Zuweisung	= op=	rechts → links
15	Sequenz	,	links → rechts



# Typumwandlung in Ausdrücken

- Eine Operation wird *mindestens* mit `int`-Wortbreite berechnet
  - `short`- und `signed char`-Operanden werden implizit „aufgewertet“ ( $\hookrightarrow$  *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!
```

Diagram illustrating the integer promotion process:

- `a` (int: 100) and `b` (int: 3) are promoted to `int` and multiplied to get `int: 300`.
- The result `int: 300` is then divided by `c` (int: 4) to get the final result `int: 75`.



- Generell wird die *größte* beteiligte Wortbreite verwendet

↔ 6-3

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

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

Diagram illustrating the type promotion process:

- `int8_t: 75` (under `res`)
- `int: 100` (under `a`)
- `int: 3` (under `b`)
- `int: 300` (under `a * b`)
- `int32_t: 300` (under `a * b / c`)
- `int32_t: 75` (under `res`)



# Typumwandlung in Ausdrücken (Forts.)

- Fließkomma-Typen gelten dabei als „größer“ als Ganzzahl-Typen
- Alle Fließkomma-Operationen werden *mindestens* mit **double**-Wortbreite berechnet

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

res = a * b / 4.0f ; // promotion to double
int8_t: 75      int: 100   int: 3      double 4.0
                └──────────┘
                int: 300
                └──────────┘
                double: 300.0
                └──────────────────┘
                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 type casting:*

The expression `res = s < (int) u;` is annotated with curly braces and labels:

- A brace under `res` is labeled `int: 1`.
- A brace under `(int)` is labeled `int: 1`.
- A brace under `u` is labeled `int: 1`.
- A larger brace under the entire expression `(int) u` is labeled `int: 1`.



# Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

**8 Kontrollstrukturen**

**9 Funktionen**

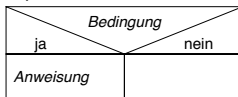
**10 Variablen**

**11 Präprozessor**



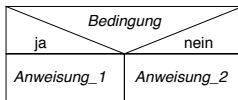
- **if**-Anweisung (bedingte Anweisung)

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



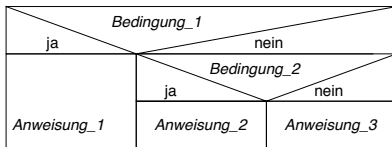
- **if-else**-Anweisung (einfache Verzweigung)

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



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

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





- **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;  
}
```

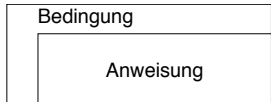


# Abweisende und nicht-abweisende Schleife [=Java]

## ■ Abweisende Schleife

[↔ GDI, 08-05]

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



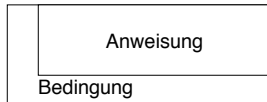
`while`(*Bedingung*)  
*Anweisung*;

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

## ■ Nicht-abweisende Schleife

[↔ GDI, 08-07]

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



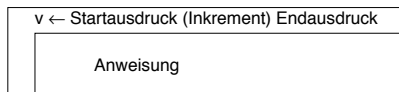
`do`  
*Anweisung*;  
`while`(*Bedingung*);

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



## ■ for-Schleife (Laufrichtung)

```
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

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

## Funktion ↔ Abstraktion

↔ 4-1

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



# Beispiel

- Funktion (Abstraktion) `sb_led_set_all_leds()`

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



- Implementierung in der `libspicboard`

```
void sb_led_set_all_leds(uint8_t setting)
```

**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<sub>opt</sub>* 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 Werte, muss anzahl- und typkompatibel sein zur Liste der formalen Parameter) [=Java]

## Beispiele:

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

Aufruf der `max()`-Funktion. 47 und 11 sind die **tatsächlichen Parameter**, welche nun den formalen Parametern `a` und `b` der `max()`-Funktion ( $\leftrightarrow$  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`  $\leftrightarrow$  9-3 )

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

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



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



- 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 ↦ \$\$\$

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

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

void main(void) {
    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(void) {
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**  $\leadsto$  der Compiler **warn**t, 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(void) {
    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

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 (↪ Speicherplatz)

- Syntax (Variablendefinition):

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

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



- Variablen können an verschiedenen Positionen definiert werden
  - Global                      außerhalb von Funktionen,  
   üblicherweise am Kopf der Datei
  - Lokal                        zu Beginn eines { Blocks },                      C89  
   direkt nach der öffnenden Klammer
  - 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

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

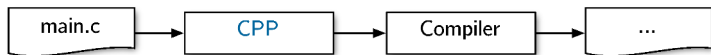
8 Kontrollstrukturen

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)
    - Kommentare 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 < 0 || 8 < NUM_LEDS)
# 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           << hat geringere Präzedenz als *
n = POW2(2) * 3                 ~ 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)   a++ wird ggf. zweimal ausgewertet
n = max(x++, 7)                     ~ 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;
}
```





# Grundlagen der Systemnahen Programmierung in C (GSPiC)

## Teil C Systemnahe Softwareentwicklung

**Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2018

[http://www4.cs.fau.de/Lehre/SS18/V\\_GSPiC](http://www4.cs.fau.de/Lehre/SS18/V_GSPiC)



# Überblick: Teil C Systemnahe Softwareentwicklung

**12 Programmstruktur und Module**

**13 Zeiger und Felder**

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

**15 Nebenläufigkeit**

**16 Speicherorganisation**



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

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



# Beispiel-Projekt: Eine Wetterstation

## ■ Typisches eingebettetes System

### ■ Mehrere Sensoren

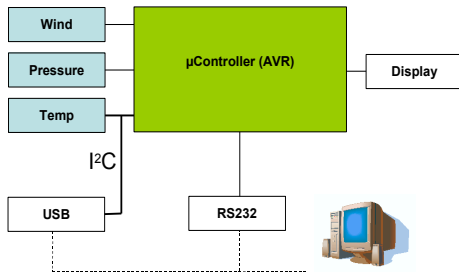
- Wind
- Luftdruck
- Temperatur

### ■ Mehrere Aktuatoren (hier: Ausgabegeräte)

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

### ■ Sensoren und Aktuatoren an den $\mu\text{C}$ angebunden über verschiedene Bussysteme

- I<sup>2</sup>C
- RS232



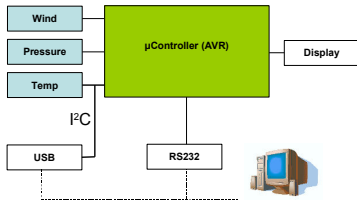
Wie sieht die **funktionale Dekomposition** der Software aus?



# Funktionale Dekomposition: Beispiel

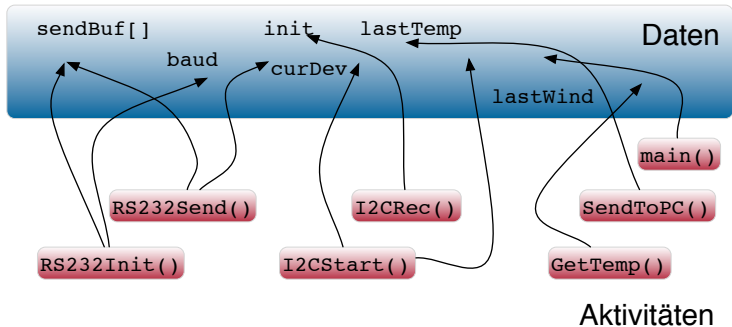
## Funktionale Dekomposition der Wetterstation (Auszug):

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  $\rightsquigarrow$  mangelhafte Trennung der Belange



# Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der **Aktivitäten**, nicht jedoch die die Struktur der **Daten**
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten  $\rightsquigarrow$  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>		Definition → Blockende	Definition → Blockende
	<b>static</b>		Definition → Blockende	Programmstart → Programmende
Global	<i>keine</i>		unbeschränkt	Programmstart → Programmende
	<b>static</b>		modulweit	Programmstart → Programmende

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

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





- 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

↔ 1-4

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

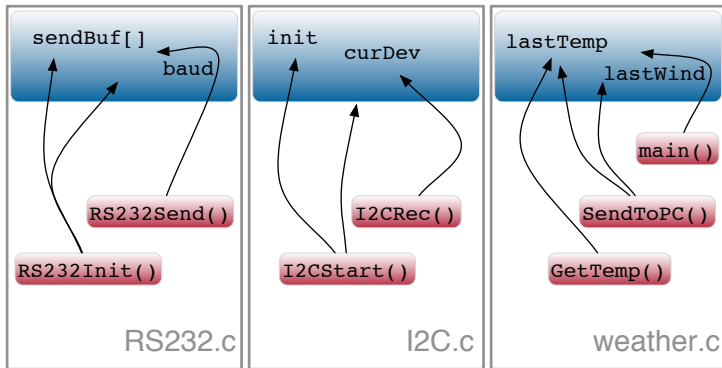
- Globale Variablen sind überall sichtbar
- Globale Variablen belegen Speicher über die gesamte Programmlaufzeit

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



# Lösung: Modularisierung

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



# Was ist ein Modul?

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

## Modul $\mapsto$ Abstraktion

$\leftrightarrow$  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, ↔ 3-13  
sondern rein **idiomatisch** (über **Konventionen**) realisiert
- 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;
    ...
}
```

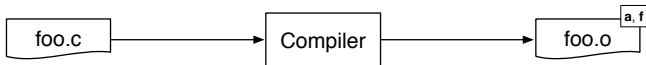
**RS232.c:** **Implementierung (nicht öffentl.)**  
Definition der bereitgestellten Funktionen (und ggf. Daten)

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**)

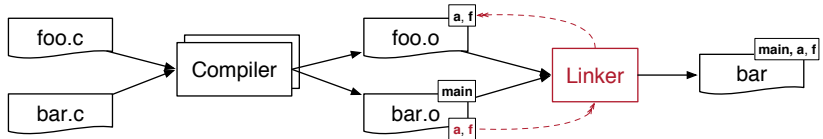
```
extern uint16_t a;  
// declare  
void f(void);      // declare  
  
void main() {      // public  
    a = 0x4711;    // use  
    f();           // use  
}
```

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

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



- Die eigentliche Auflösung erfolgt durch den **Linker**



## Linken ist **nicht typsicher!**

- Typinformationen sind in Objektdateien nicht mehr vorhanden
- Auflösung durch den Linker erfolgt **ausschließlich** über die **Symbolnamen** (Bezeichner)
- ↪ Typsicherheit muss beim **Übersetzen** sichergestellt werden
- ↪ Einheitliche Deklarationen durch gemeinsame Header-Datei

- 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

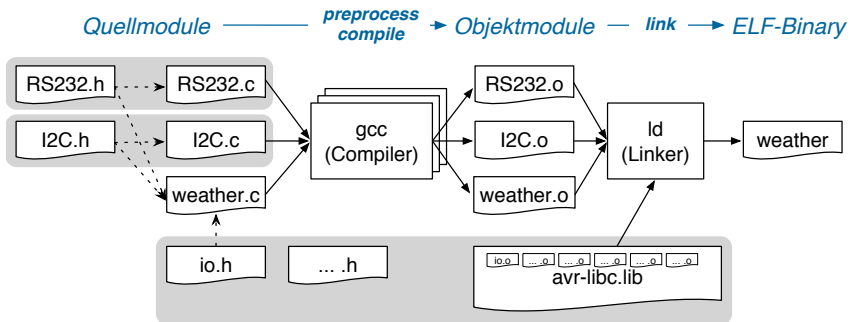
(vergleiche ↔ 12-11)

```
// 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



# Zusammenfassung

- Prinzip der Trennung der Belange  $\rightsquigarrow$  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
    - $\rightsquigarrow$  **Linken ist nicht typsicher!**
  - Typsicherheit muss beim Übersetzen sichergestellt werden
    - $\rightsquigarrow$  durch gemeinsame Header-Datei



# Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

**13 Zeiger und Felder**

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

**15 Nebenläufigkeit**

**16 Speicherorganisation**



# Einordnung: Zeiger (*Pointer*)

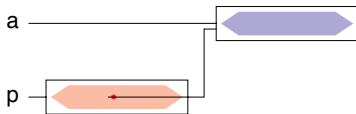
- **Literal:** 'a'  
Darstellung eines Wertes

'a'  $\equiv$  

- **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*)
  - Speicher lässt sich direkt ansprechen
  - Effizientere Programme
- Aber auch viele Probleme!
  - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
  - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

↪ 9-5

„Effizienz durch  
Maschinennähe“

↪ 3-14



# Definition von Zeigervariablen

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

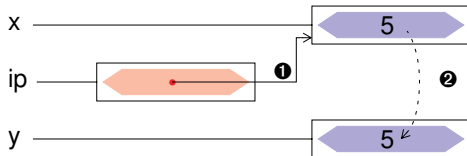
```
int x = 5;
```

```
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.





# Zeiger als Funktionsargumente

- Parameter werden in C immer *by-value* übergeben ↔ 9-5
  - Parameterwerte werden in lokale Variablen der aufgerufenen Funktion kopiert
  - Aufgerufene Funktion kann tatsächliche Parameter des Aufrufers nicht ändern
  
- Das gilt auch für Zeiger (Verweise) [↔ GDI, 14-01-01]
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des \*-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

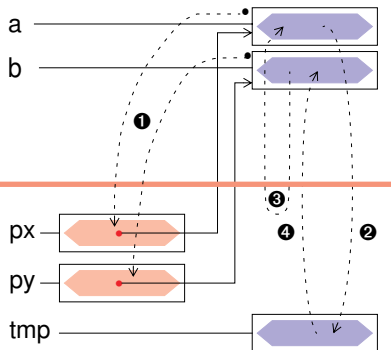
↪ **Call-by-reference**



## ■ Beispiel (Gesamtüberblick)

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

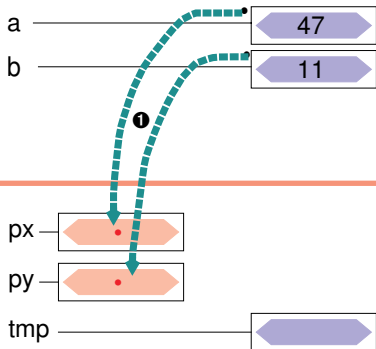
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ❷  
    *px = *py; ❸  
    *py = tmp; ❹  
}
```



## ■ Beispiel (Einzelschritte)

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

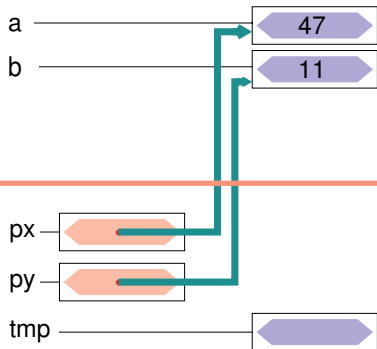
```
void swap (int *px, int *py)  
{  
    int tmp;
```



## ■ Beispiel (Einzelschritte)

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

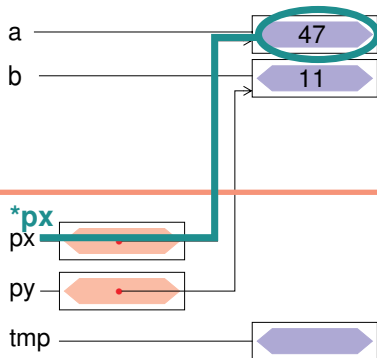
```
void swap (int *px, int *py)  
{  
    int tmp;  
    ...  
}
```



## ■ Beispiel (Einzelschritte)

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

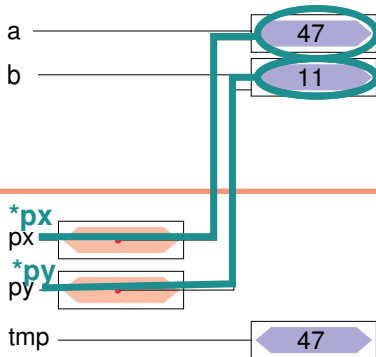
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②
```



## ■ Beispiel (Einzelschritte)

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

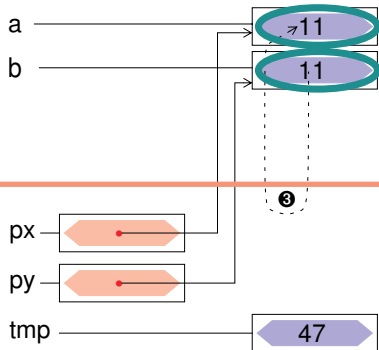
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
}
```



## ■ Beispiel (Einzelschritte)

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

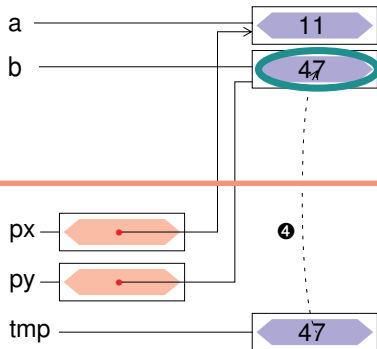
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
}
```



## ■ 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 (→ Anzahl der Elemente).                      [≠Java]  
Ab **C99** darf *IntAusdruck* bei **auto**-Feldern auch **variabel** (d. h. beliebig, aber fest) sein.
- Beispiele:

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

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



# Feldinitialisierung

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

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

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

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

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

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



# Feldzugriff

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

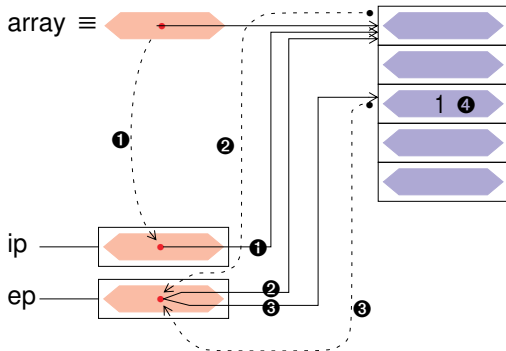
```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };  
LEDs[3] = BLUE1;  
for (uint8_t i = 0; i < 4; i++) {  
    sb_led_on(LEDs[i]);  
}  
LEDs[4] = GREEN1; // UNDEFINED!!!
```



# Felder sind Zeiger

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

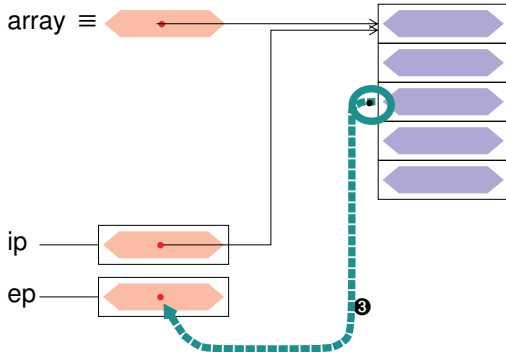
```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



# Felder sind Zeiger

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

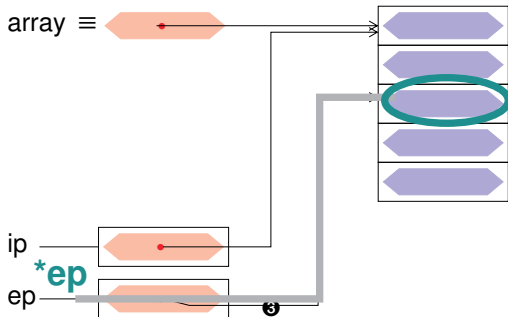
```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array`  $\equiv$  `&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 ↪ 13-9
- Beispiel (vgl. ↪ 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]);  
}
```

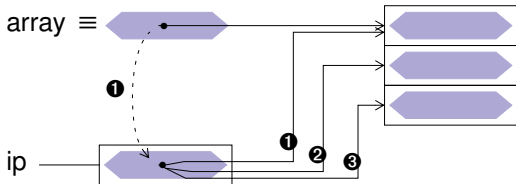


# Rechnen mit Zeigern

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

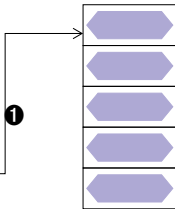
```
int array[3];  
int *ip = array; ❶
```

```
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-/Postdecrement  
↷ Verschieben auf das vorangegangene Objekt
- +, - Addition / Subtraktion eines `int`-Wertes  
↷ Ergebniszeiger ist verschoben um  $n$  Objekte
  - Subtraktion zweier Zeiger  
↷ Anzahl der Objekte  $n$  zwischen beiden Zeigern (Distanz)

## ■ Vergleichsoperationen: `<`, `<=`, `==`, `>=`, `>`, `!=`

↷ 7-3

- ↷ Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen



# Felder sind Zeiger sind Felder – Zusammenfassung

- In Kombination mit Zeigerarithmetik lässt sich in C **jede** Feldoperation auf eine äquivalente Zeigeroperation abbilden.
- Für `int i, array[N], *ip = array;` mit  $0 \leq i < N$  gilt:

```
array    ≡ &array[0]  ≡ ip      ≡ &ip[0]
*array   ≡ array[0]   ≡ *ip     ≡ ip[0]
*(array + i) ≡ array[i] ≡ *(ip + i) ≡ ip[i]
          array++ ≠ ip++
          Fehler: array ist konstant!
```

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



# Felder als Funktionsparameter

- Felder werden in C **immer** als Zeiger übergeben

[=Java]

↪ *Call-by-reference*

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

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

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



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



# Felder als Funktionsparameter (Forts.)

- Felder werden in C **immer** als Zeiger übergeben [=Java]  
↳ *Call-by-reference*

- Wird der Parameter als **const** deklariert, so kann die Funktion die Feldelemente **nicht verändern** → Guter Stil! [≠Java]

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

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

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

- **Achtung:** Das gilt so nur bei Deklaration eines 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

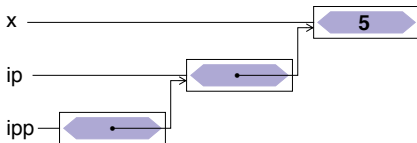
```
int strlen(const char *s) {  
    const char *end = s;  
    while (*end != '\0')  
        end++;  
    return end - s;  
}
```



# 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 )(FormaleParamopt);`  
(sehr ähnlich zur Syntax von Funktionsdeklarationen) ↔ 9-3
  - *Typ* Rückgabetyt 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:  $Typ_1, \dots, Typ_n$
- Ein Funktionszeiger wird genau wie eine Funktion verwendet
  - Aufruf mit `Bezeichner(TatParam)` ↔ 9-4
  - Adress- (&) und Verweisoperator (\*) werden nicht benötigt ↔ 13-4
  - Ein Funktionsbezeichner ist ein konstanter Funktionszeiger

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

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





- 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_registerCallback(     // register callback
        BUTTON0, BUTTONEVENT_PRESSED, // 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**

**15 Nebenläufigkeit**

**16 Speicherorganisation**



# Was ist ein $\mu$ -Controller?

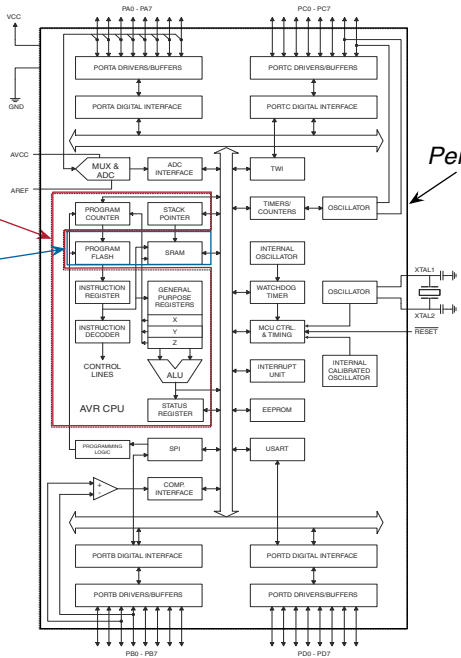
- **$\mu$ -Controller** := Prozessor + Speicher + Peripherie
  - Faktisch ein Ein-Chip-Computersystem  $\rightarrow$  SoC (*System-on-a-Chip*)
  - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher  $\rightsquigarrow$  kostengünstiges Systemdesign
- Wesentliches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



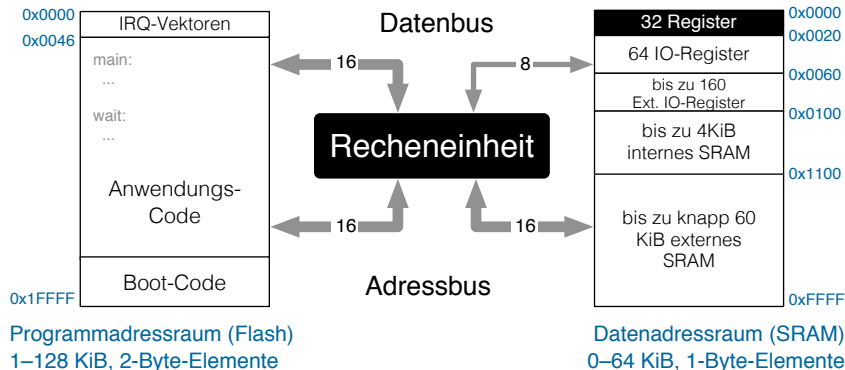
# Beispiel ATmega32: Blockschaltbild

*CPU-Kern*  
*Speicher*

*Peripherie*



# Beispiel ATmega-Familie: CPU-Architektur

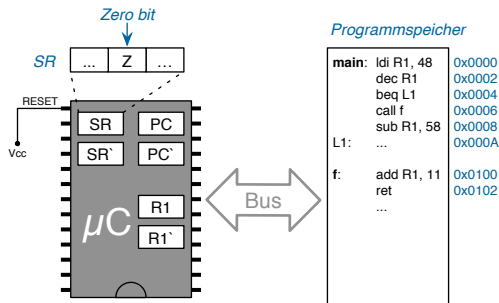


- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingebündelt  
↪ ansprechbar wie globale Variablen

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



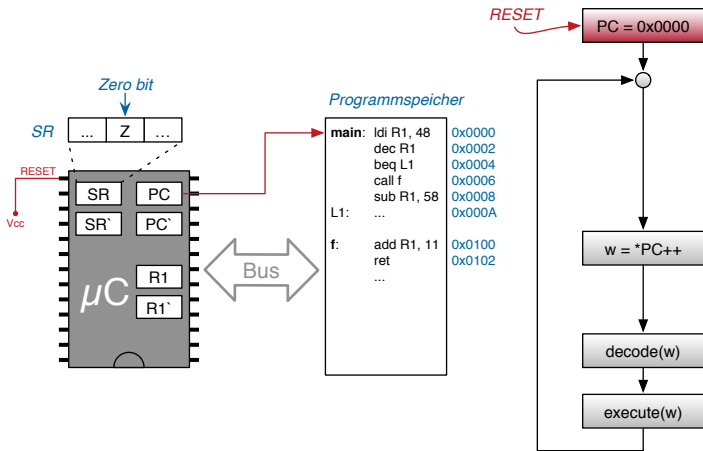
# Wie arbeitet ein Prozessor?



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

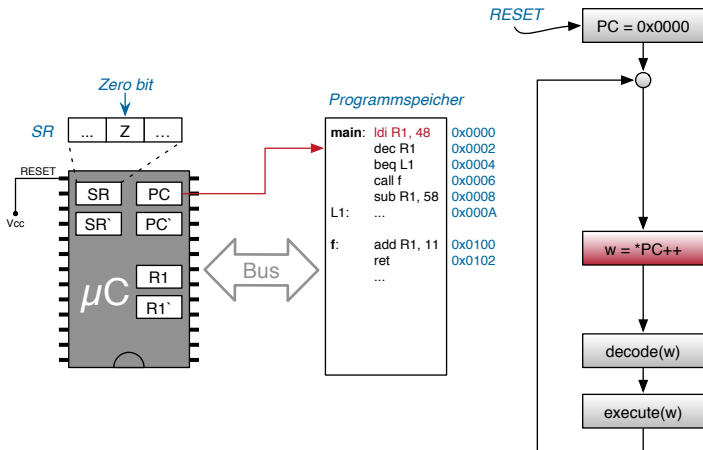


# Wie arbeitet ein Prozessor?

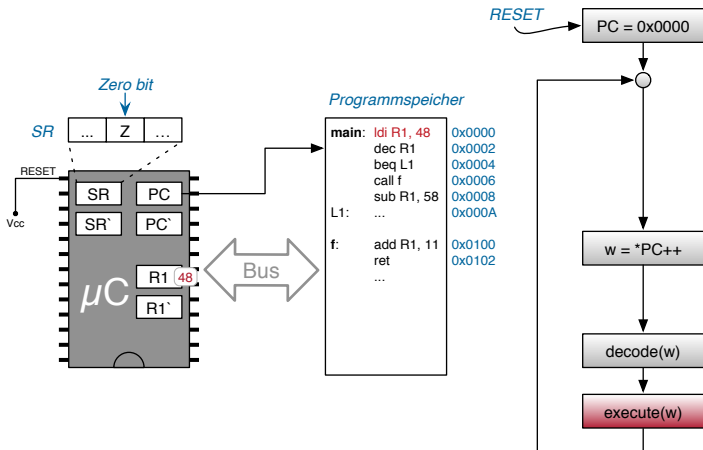




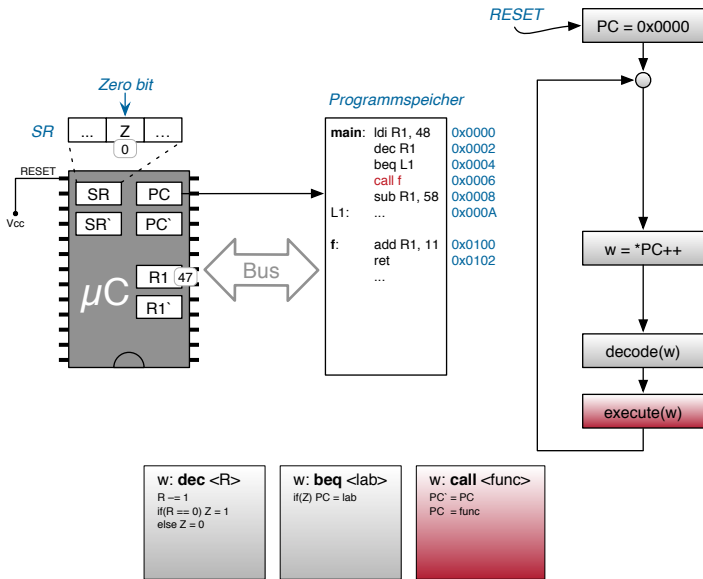
# Wie arbeitet ein Prozessor?



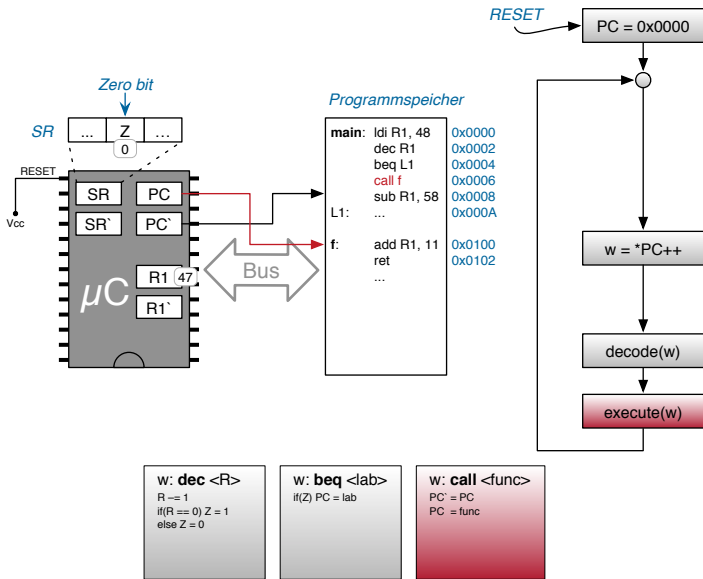
# Wie arbeitet ein Prozessor?



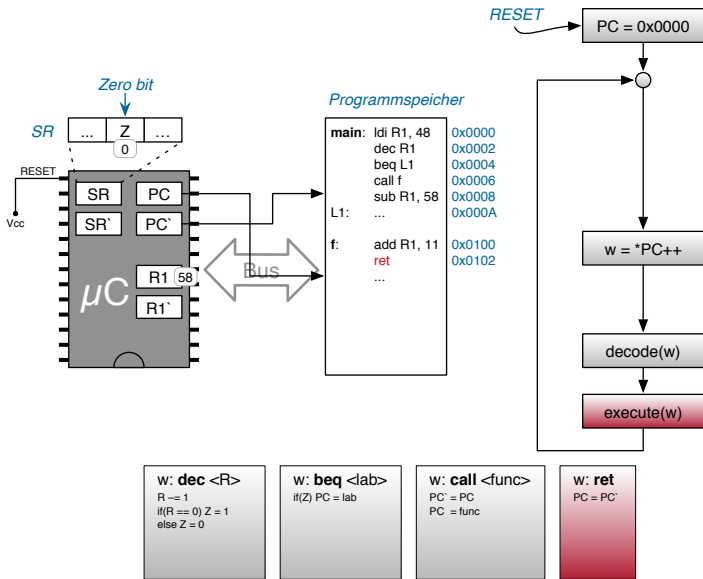
# Wie arbeitet ein Prozessor?



# Wie arbeitet ein Prozessor?



# Wie arbeitet ein Prozessor?



- **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.  $\leftrightarrow$  14–12



# Peripheriegeräte – Register

- Es gibt verschiedene Architekturen für den Zugriff auf I/O-Register
  - Memory-mapped: Register sind in den Adressraum eingebunden; 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. 327]





- 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 )
```

Diagramm zur Analyse des Makros `PORTD`:

- Die Adresse `0x12` ist als `Adresse: int` markiert.
- Die gesamte Expression `*(volatile uint8_t *) 0x12` ist als `Adresse: volatile uint8_t *(Cast  $\leftrightarrow$  7-19)` markiert.
- Die gesamte Expression `*(volatile uint8_t *) 0x12` ist als `Wert: volatile uint8_t (Dereferenzierung  $\leftrightarrow$  13-4)` markiert.

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

- Beispiel

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

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



# Registerzugriff und Nebenläufigkeit

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

```
// C code
#define PIND (*(uint8_t*) 0x10)
void foo(void) {
    ...
    if (! (PIND & 0x2)) {
        // button0 pressed
        ...
    }
    if (! (PIND & 0x4)) {
        // button 1 pressed
        ...
    }
}
```

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

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



# Der volatile-Typmodifizierer

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

```
// C code
#define PIND \
  (*(volatile uint8_t*) 0x10)
void foo(void) {
  ...
  if (! (PIND & 0x2)) {
    // button0 pressed
    ...
  }

  if (! (PIND & 0x4)) {
    // button 1 pressed
    ...
  }
}
```

```
// Resulting assembly code

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

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

L2:
  ret
```

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



# Der volatile-Typmodifizierer (Forts.)

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

```
// C code                                // Resulting assembly code
void wait(void) {                          wait:
    for (uint16_t i = 0; i < 0xffff; i++); // compiler has optimized
}                                           // "unneeded" loop
                                           ret
```

**volatile!**

## **Achtung:** `volatile` ↦ \$\$\$

Die Verwendung von `volatile` verursacht erhebliche **Kosten**

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

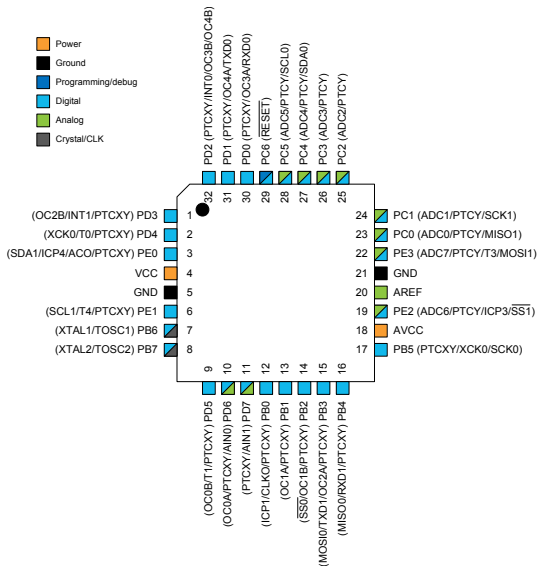
**Regel:** `volatile` wird nur in **begründeten Fällen** verwendet



- **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)  $\rightsquigarrow$  Prozessor führt Interruptprogramm aus
- Die Funktion ist üblicherweise pro Pin konfigurierbar
  - Eingang
  - Ausgang
  - Externer Interrupt (nur bei bestimmten Eingängen)
  - Alternative Funktion (Pin wird von anderem Gerät verwendet)



# Beispiel ATmega328PB: Port/Pin-Belegung



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

Beim SPiCboard werden z. B. **Pins 23–24 als ADCs konfiguriert**, um Poti und Photosensor anzuschließen.

PORTC steht daher für diese Pins **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
DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
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/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Verwendungsbeispiele:  $\leftrightarrow$  3-5 und  $\leftrightarrow$  3-8 [1, S. 92]



# 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)

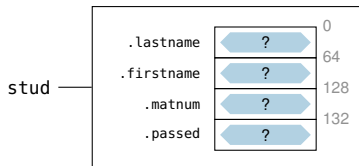
↪ 4-1  
↪ 12-4

```
// 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

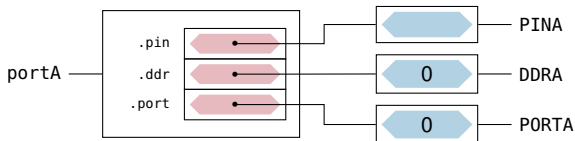
↔ 6-8

```
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]

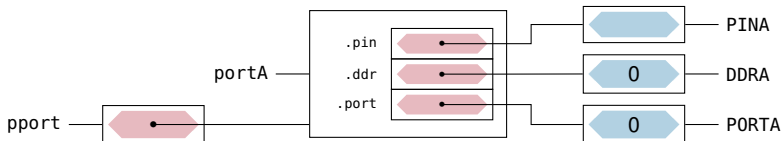
```
port_t portA = { &PINA, &DDRA, &PORTA };
```

```
*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(void) { initPort(portA); ... }
```

- Bei größeren Strukturen wird das **sehr ineffizient**
  - Z. B. Student ( $\leftrightarrow$  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(void) { initPort(&portA); ... }
```

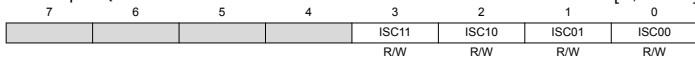


# Bit-Strukturen: Bitfelder

- 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

- EICRA **External Interrupt Control Register A** Steuert Auslöser für externe Interrupt-Quellen INT0 und INT1. [1, S. 83]



```
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 reserved : 4;    // bit 4-7: reserved for future use
} EICRA_t;
```

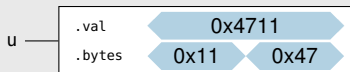


# Unions

- 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(void) {  
    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 reserved : 4;
    };
} EICRA_t;

void foo(void) {
    EICRA_t *eicra = (EICRA_t *) 0x69;
    uint8_t oldval = eicra->reg; // save register
    ...
    eicra->ISC0 = 2; // use register
    eicra->ISC1 = 1; // ...
    ...
    eicra->reg = oldval; // restore register
}
```



# Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14  $\mu$ C-Systemarchitektur

**15 Nebenläufigkeit**

**16 Speicherorganisation**

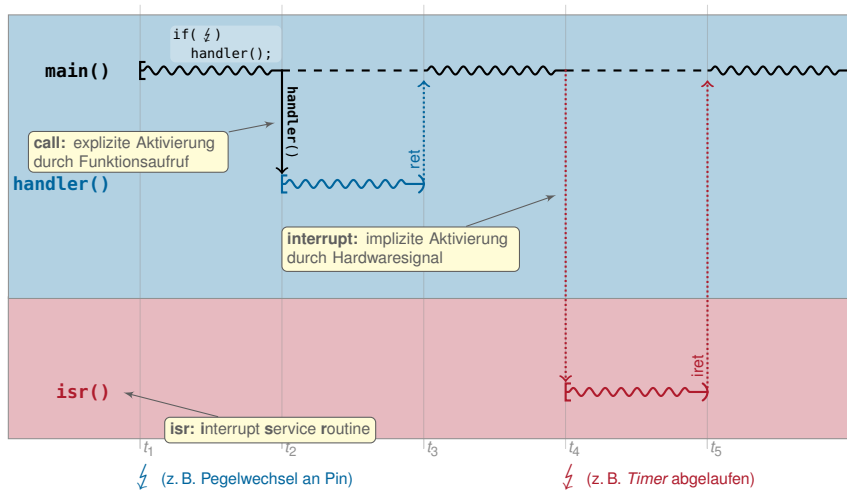




- Bei einem **Peripheriegerät** tritt ein Ereignis (⚡) auf ↔ 14-5
  - Signal an einem Port-Pin wechselt von *low* auf *high*
  - Ein *Timer* ist abgelaufen
  - Ein A/D-Wandler hat einen neuen Wert vorliegen
  - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren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



# Interrupt $\mapsto$ Funktionsaufruf „von außen“



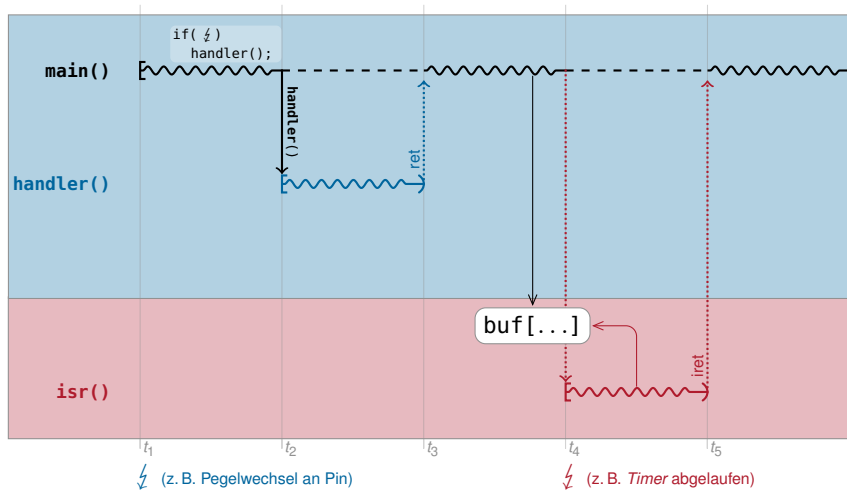
# Polling vs. Interrupts – Vor- und Nachteile

- Polling (↳ „Taktgesteuertes System“)
  - Ereignisbearbeitung erfolgt **synchron** zum Programmablauf
    - Ereigniserkennung über das Programm „verstreut“ (Trennung der Belange)
    - „Verschwendung“ von Prozessorzeit (falls anderweitig verwendbar)
    - Hochfrequentes Pollen  $\leadsto$  hohe Prozessorlast  $\leadsto$  **hoher Energieverbrauch**
    - + Implizite Datenkonsistenz durch festen, sequentiellen Programmablauf
    - + Programmverhalten gut vorhersagbar
- Interrupts (↳ „Ereignisgesteuertes System“)
  - Ereignisbearbeitung erfolgt **asynchron** zum Programmablauf
    - + Ereignisbearbeitung kann im Programmtext gut separiert werden
    - + Prozessor wird nur beansprucht, wenn Ereignis tatsächlich eintritt
    - Höhere Komplexität durch Nebenläufigkeit  $\leadsto$  Synchronisation erforderlich
    - Programmverhalten **schwer vorhersagbar**

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



# Interrupt $\mapsto$ unvorhersagbarer Aufruf „von außen“



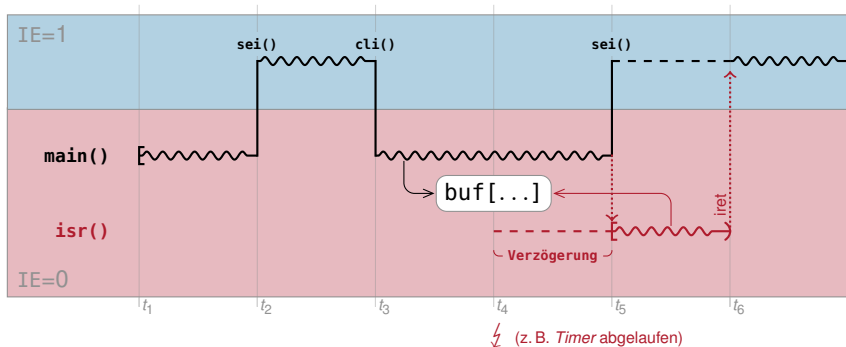
# Interruptsperrn

- Zustellung von Interrupts kann softwareseitig **gesperrt** werden
  - Wird benötigt zur **Synchronisation** mit ISRs
  - Einzelne ISR: Bit in gerätespezifischem Steuerregister
  - Alle ISRs: Bit (**IE**, *Interrupt Enable*) im Statusregister der CPU
- Auflaufende IRQs werden (üblicherweise) gepuffert
  - Maximal einer pro Quelle!
  - **Bei längeren Sperrzeiten können IRQs verloren gehen!**
- Das **IE**-Bit wird beeinflusst durch:
  - Prozessor-Befehle: `cli`:  $IE \leftarrow 0$  (*clear interrupt*, IRQs gesperrt)  
`sei`:  $IE \leftarrow 1$  (*set interrupt*, IRQs erlaubt)
  - Nach einem RESET:  $IE=0 \rightsquigarrow$  IRQs sind zu Beginn des Hauptprogramms gesperrt
  - Bei Betreten einer ISR:  $IE=0 \rightsquigarrow$  IRQs sind während der Interruptbearbeitung gesperrt

IRQ  $\mapsto$  *Interrupt ReQuest*



# Interruptsperrn: Beispiel

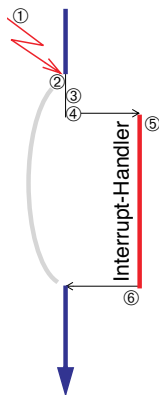


- $t_1$  Zu Beginn von `main()` sind IRQs gesperrt (`IE=0`)
- $t_2, t_3$  Mit `sei()` / `cli()` werden IRQs freigegeben (`IE=1`) / erneut gesperrt
- $t_4$  ⚡ aber `IE=0`  $\leadsto$  Bearbeitung ist unterdrückt, IRQ wird gepuffert
- $t_5$  `main()` gibt IRQs frei (`IE=1`)  $\leadsto$  gepufferter IRQ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

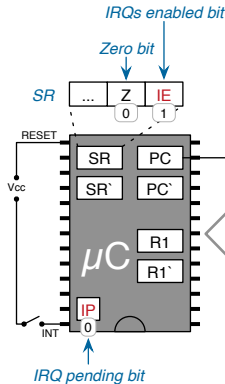


# Ablauf eines Interrupts – Überblick

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



# Ablauf eines Interrupts – Details



## Programmspeicher

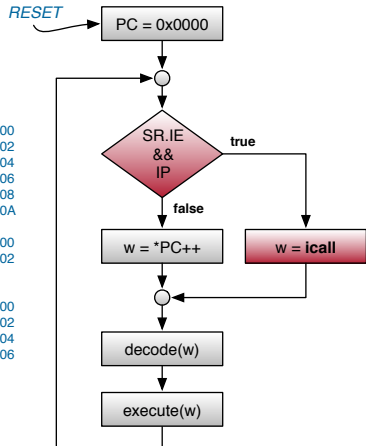
```

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

L1:    ...

f:     add R1, 11  0x0100
       ret       0x0102

isr:   Idi R1, 1  0x0200
       dec R1    0x0202
       sts a, R1 0x0204
       irt      0x0206
    
```



■ Hier als Erweiterung unseres einfachen Pseudoprozessors  $\leftrightarrow$  14-4

- Nur eine Interruptquelle
- Sämtliche Register werden von der Hardware gerettet

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

**w: ret**  
 PC = PC'

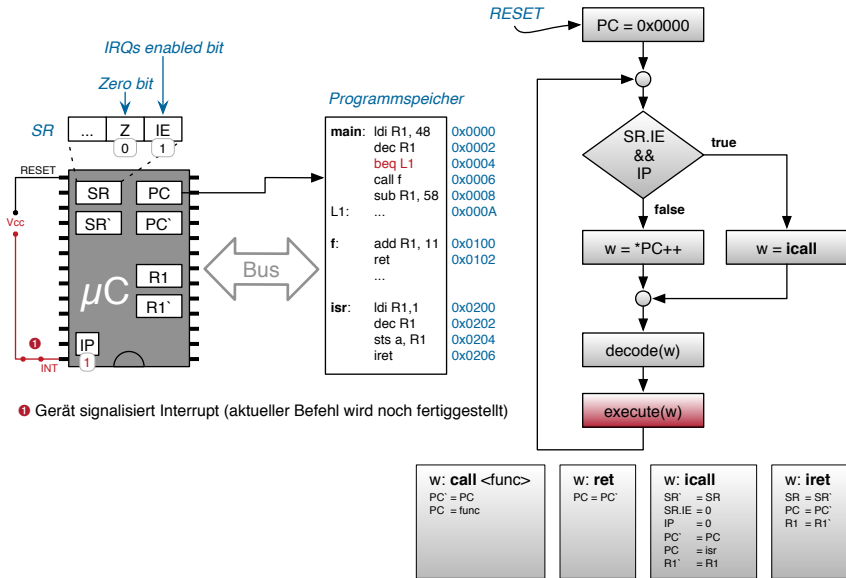
**w: icall**  
 SR' = SR  
 SR.IE = 0  
 IP = 0  
 PC' = PC  
 PC = isr  
 R1' = R1

**w: irt**  
 SR = SR'  
 PC = PC'  
 R1 = R1'

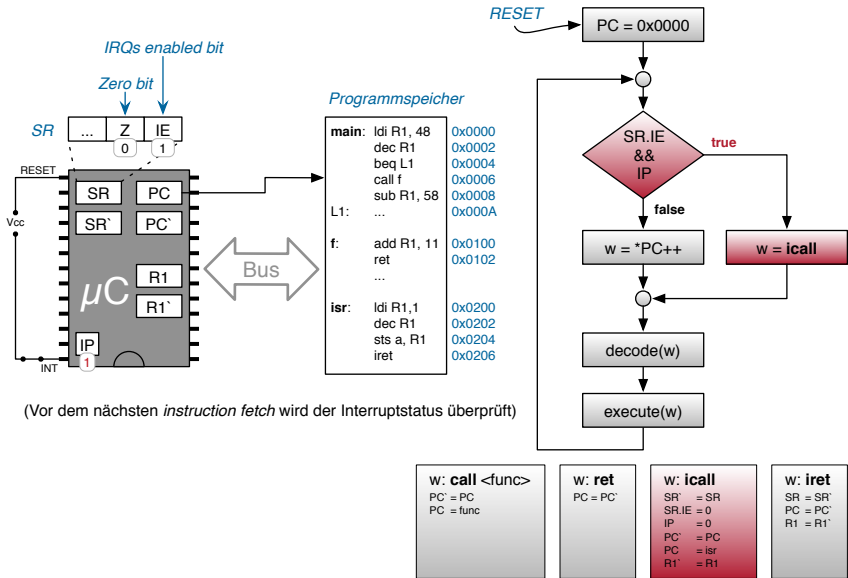




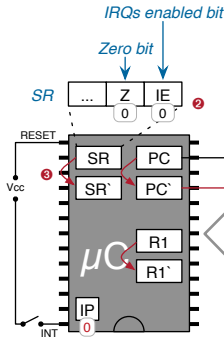
# Ablauf eines Interrupts – Details



# Ablauf eines Interrupts – Details



# Ablauf eines Interrupts – Details



## Programmspeicher

```

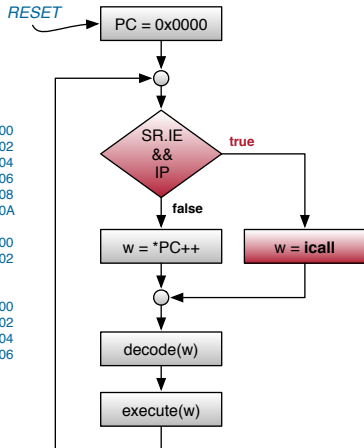
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

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

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

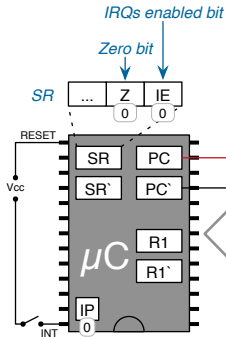
- ② Die Zustellung weiterer Interrupts wird verzögert
- ③ Registerinhalte werden gesichert



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



# Ablauf eines Interrupts – Details



4 Aufzurufende ISR wird ermittelt

Programmspeicher

```

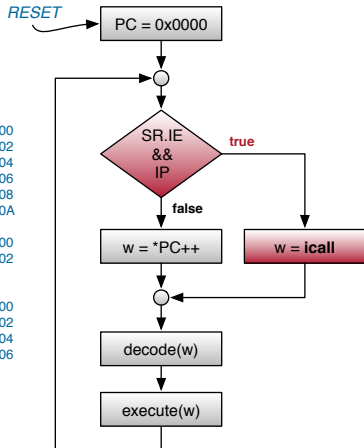
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A

L1:   ...

f:    add R1, 11 0x0100
      ret       0x0102

...

ISR: ldi R1, 1 0x0200
      dec R1   0x0202
      sts a, R1 0x0204
      ired    0x0206
    
```



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

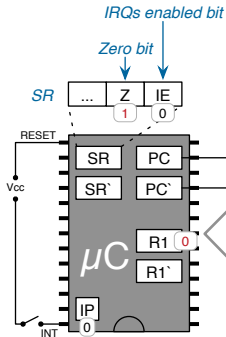
w: **ret**  
PC = PC'

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

w: **ired**  
SR = SR'  
PC = PC'  
R1 = R1'



# Ablauf eines Interrupts – Details

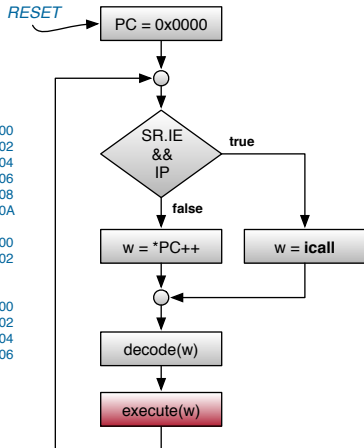


ISR wird ausgeführt

Programmspeicher

```

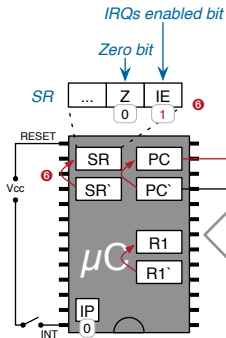
main: ldi R1, 48 0x0000
      dec R1    0x0002
      beq L1   0x0004
      call f   0x0006
      sub R1, 58 0x0008
      ...     0x000A
L1:   ...
f:    add R1, 11 0x0100
      ret      0x0102
...
isr: ldi R1, 1  0x0200
      dec R1    0x0202
      sts a, R1 0x0204
      iret     0x0206
    
```



<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: iret</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



# Ablauf eines Interrupts – Details



## Programmspeicher

```

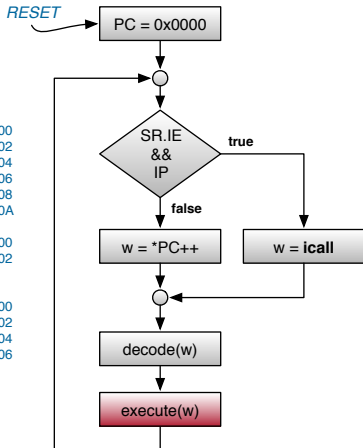
main:  Idi R1, 48  0x0000
       dec R1    0x0002
       beq L1    0x0004
       call f    0x0006
       sub R1, 58 0x0008
       ...      0x000A

L1:    ...

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

isr:   Idi R1, 1  0x0200
       dec R1    0x0202
       sts a, R1 0x0204
       ired     0x0206
    
```

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

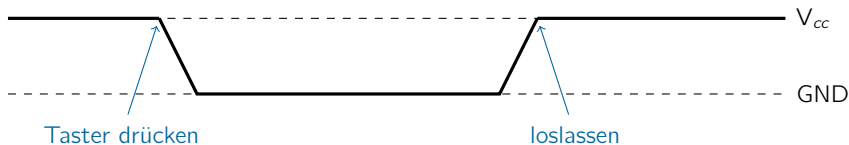


<b>w: call &lt;func&gt;</b> PC' = PC PC = func	<b>w: ret</b> PC = PC'	<b>w: icall</b> SR' = SR SR.IE = 0 IP = 0 PC' = PC PC = isr R1' = R1	<b>w: ired</b> SR = SR' PC = PC' R1 = R1'
--	---------------------------	--	--



# Pegel- und Flanken-gesteuerte Interrupts

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



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



# Interruptsteuerung beim AVR ATmega

## ■ IRQ-Quellen beim ATmega328PB

- 45 IRQ-Quellen
- einzeln de-/aktivierbar
- IRQ  $\rightsquigarrow$  Sprung an Vektor-Adresse

(IRQ  $\rightsquigarrow$  Interrupt ReQuest)

[1, S. 78]

## ■ Verschaltung SPiCboard

(  $\leftrightarrow$  14-14  $\leftrightarrow$  2-5 )

- INT0  $\rightsquigarrow$  PD2  $\rightsquigarrow$  Button0 (hardwareseitig entprellt)
- INT1  $\rightsquigarrow$  PD3  $\rightsquigarrow$  Button1

Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Counter1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI0_STC	SPI1 Serial Transfer Complete
19	0x0024	USART0_RX	USART0 Rx Complete
20	0x0026	USART0_UDRE	USART0, Data Register Empty
21	0x0028	USART0_TX	USART0, Tx Complete
22	0x002A	ADC	ADC Conversion Complete

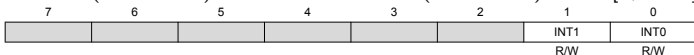




# Externe Interrupts: Register

## ■ Steuerregister für INT0 und INT1

- **EIMSK** **External Interrupt Mask Register:** Legt fest, ob die Quellen  $INT_i$  IRQs auslösen (Bit  $INT_i=1$ ) oder deaktiviert sind (Bit  $INT_i=0$ ) [1, S. 84]



- **EICRA** **External Interrupt Control Register A:** Legt für externe Interrupts INT0 und INT1 fest, wodurch ein IRQ ausgelöst wird (Flanken-/Pegelsteuerung) [1, S. 83]



Jeweils zwei *Interrupt-Sense-Control*-Bits ( $ISC_i0$  und  $ISC_i1$ ) steuern dabei die Auslöser (Tabelle für INT1, für INT0 gilt entsprechendes):

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



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

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

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

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



## ■ Schritt 2: Konfigurieren der Interrupt-Steuerung

- Steuerregister dem Wunsch entsprechend initialisieren
- Unterstützung durch die avrlibc: Makros für Bit-Indizes (Modul avr/interrupt.h und avr/io.h)

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

## ■ Schritt 3: Interrupts global zulassen

- Nach Abschluss der Geräteinitialisierung
- Unterstützung durch die avrlibc: Befehl sei() (Modul avr/interrupt.h)



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



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

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



## Definition: Nebenläufigkeit

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

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



# Nebenläufigkeitsprobleme

## ■ Szenario

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

```
static volatile uint16_t cars;

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

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

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

## ■ Wo ist hier das Problem?

- Sowohl main() als auch ISR **lesen und schreiben** cars
  - ↪ Potentielle *Lost-Update*-Anomalie
- Größe der Variable cars **übersteigt die Registerbreite**
  - ↪ Potentielle *Read-Write*-Anomalie



# Nebenläufigkeitsprobleme (Forts.)

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

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

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

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

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

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



# Nebenläufigkeitsprobleme: *Lost-Update*-Anomalie

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

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



- Sei cars=5 und an **dieser Stelle** tritt der IRQ (⚡) auf
  - main hat den Wert von cars (5) bereits in Register gelesen (Register → lokale Variable)
  - INT2\_vect wird ausgeführt
    - Register werden gerettet
    - cars wird inkrementiert ~ cars=6
    - Register werden wiederhergestellt
  - main übergibt den **veralteten Wert** von cars (5) an send
  - main nullt cars ~ **1 Auto ist „verloren“ gegangen**





# Nebenläufigkeitsprobleme: *Read-Write-Anomalie*

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

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

- Sei cars=255 und an **dieser Stelle** tritt der IRQ (⚡) auf
  - main hat bereits cars=255 Autos mit send gemeldet
  - main hat bereits das **High-Byte** von cars genullt
    - ↪ cars=255, cars.lo=255, cars.hi=0
  - INT2\_vect wird ausgeführt
    - ↪ cars wird gelesen und inkrementiert, **Überlauf ins High-Byte**
    - ↪ cars=256, cars.lo=0, cars.hi=1
  - main nullt das **Low-Byte** von cars
    - ↪ cars=256, cars.lo=0, cars.hi=1
    - ↪ Beim nächsten send werden **255 Autos zu viel gemeldet**



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

kritisches Gebiet

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



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

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

```
static volatile int8_t event;

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

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

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



# Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

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

```
static volatile int8_t event;  
  
// TIMER1 ISR  
// triggers when  
// waitsec() expires  
  
ISR(TIMER1_COMPA_vect) {  
    event = 1;  
}
```

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



# Lost-Wakeup: Dornröschenschlaf verhindern

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

```
static volatile int8_t event;
// TIMER1 ISR
// triggers when
// waitsec() expires
ISR(TIMER1_COMPA_vect) {
    event = 1;
}
```

## ■ Wo genau ist das **kritische Gebiet**?

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



# Zusammenfassung

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



# Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14  $\mu$ C-Systemarchitektur

15 Nebenläufigkeit

**16 Speicherorganisation**



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

void main() {
    static int s = 3; // s: local, static, initialized
    int x, y;        // x: local, auto; y: local, auto
    char *p = malloc(100); // p: local, auto; *p: heap (100 byte)
}
```

Wo kommt der Speicher für diese Variablen her?

## ■ Statische Allokation – Reservierung beim Übersetzen / Linken

- Betrifft alle globalen/statischen Variablen, sowie den Code
- Allokation durch Platzierung in einer **Sektion**

↪ 12-5

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

## ■ Dynamische Allokation – Reservierung zur Laufzeit

- Betrifft lokale auto-Variablen und explizit angeforderten Speicher

<b>Stack</b>	– enthält alle <b>aktuell lebendigen</b> auto-Variablen	<code>x,y,p</code>
<b>Heap</b>	– 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 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 der Symboltabelle.



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

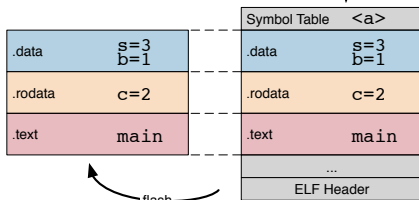
```
int a;           // a: global, uninitialized
int b = 1;      // b: global, initialized
const int c = 2; // c: global, const

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

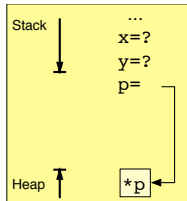
Zur Installation auf dem  $\mu\text{C}$  werden `.text` und `.[ro]data` in den Flash-Speicher des  $\mu\text{C}$  geladen.



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

RAM

Flash / ROM



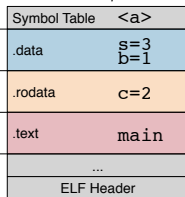
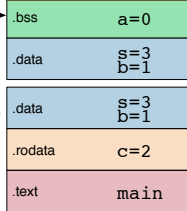
```

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



$\mu\text{-Controller}$

ELF-Binary

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).



# 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

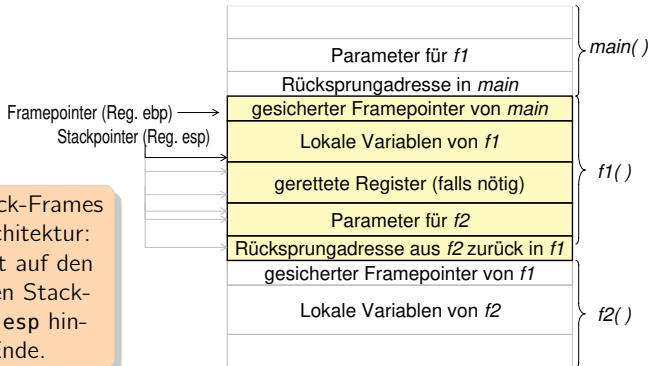
```
#include <stdlib.h>
int *intArray(uint16_t n) { // alloc int[n] array
    return (int *) malloc(n * sizeof int);
}
void main(void) {
    int *array = intArray(100); // alloc memory for 100 ints
    if (array != NULL) { // malloc() returns NULL on failure
        ... // if succeeded, use array
        array[99] = 4711;
        ...
        free(array); // free allocated block (** IMPORTANT! **)
    }
}
```



# Dynamische Speicherallokation: Stack

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem **Stack** (Stapel, Keller) verwaltet
  - Prozessorregister [e]sp zeigt immer auf den nächsten freien Eintrag
  - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von **Stack-Frames**

Aufbau eines Stack-Frames auf der IA-32-Architektur: Register ebp zeigt auf den Beginn des aktiven Stack-Frames; Register esp hinter das aktuelle Ende.

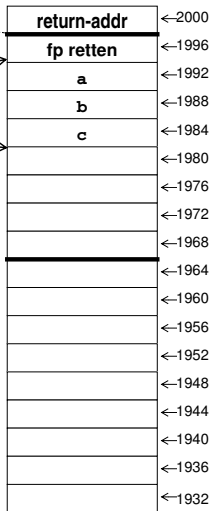


# Stack-Aufbau bei Funktionsaufrufen

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

*Stack-Frame für  
main erstellen  
&a = fp-4  
&b = fp-8  
&c = fp-12*

sp fp



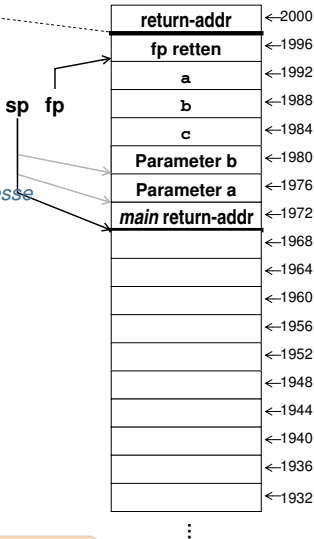
Beispiel hier für 32-Bit-Architektur (4-Byte ints), main() wurde soeben betreten



# 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

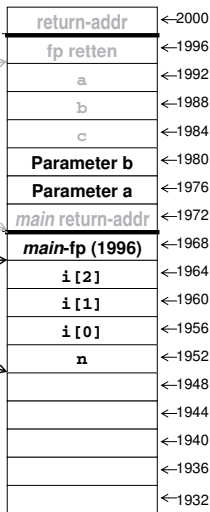
```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

```
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



⋮

f1() wurde soeben betreten





# Stack-Aufbau bei Funktionsaufrufen

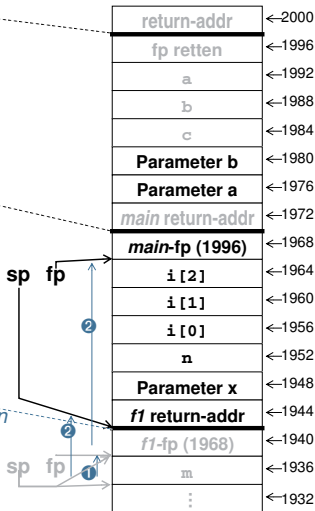
```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

```
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

y x sp fp

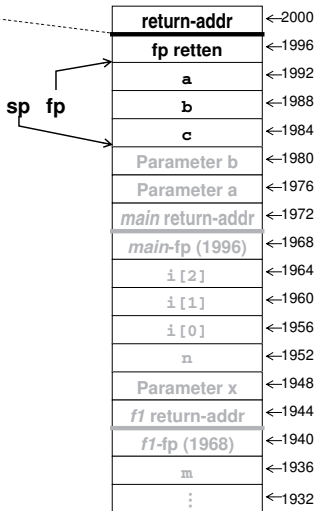
return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
Parameter b	←1980
Parameter a	←1976
main return-addr	←1972
main-fp (1996)	←1968
i [2]	←1964
i [1]	←1960
i [0]	←1956
n	←1952
Parameter x	←1948
f1 return-addr	←1944
f1-fp (1968)	←1940
m	←1936
⋮	←1932

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()

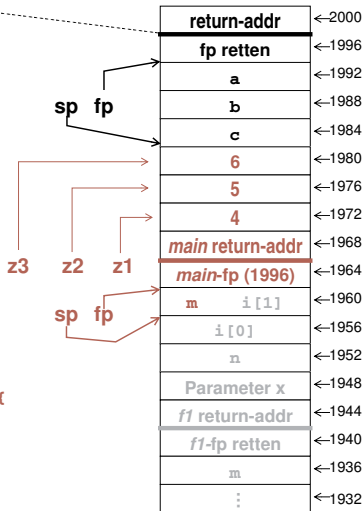


# Stack-Aufbau bei Funktionsaufrufen

```
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  $\mu$ **C-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!  $\rightarrow$  1-4)

```
~> size sections.avr
```

```
text      data      bss      dec      hex filename
682       10         6       698     2ba sections.avr
```

Sektionsgrößen des  
Programms von  $\rightarrow$  16-1

- $\rightsquigarrow$  Speicher möglichst durch **static**-Variablen anfordern
  - Regel der geringstmöglichen Sichtbarkeit beachten  $\rightarrow$  12-6
  - Regel der geringstmöglichen Lebensdauer „sinnvoll“ anwenden
- Ein Heap ist **verhältnismäßig teuer**  $\rightsquigarrow$  wird möglichst vermieden
  - Zusätzliche Speicherkosten durch Verwaltungsstrukturen und Code
  - Speicherbedarf zur Laufzeit schlecht abschätzbar
  - Risiko von Programmierfehlern und Speicherlecks

