Grundlagen der Systemnahen Programmierung in C (GSPiC)

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Wintersemester 2012

http://www4.cs.fau.de/Lehre/WS12/V_GSPIC

Referenzen

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

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

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

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

[GDI] Elmar Nöth, Peter Wilke, and Stefan Steidl. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2010 (jährlich). URL: `http://www5.informatik.uni-erlangen.de/lectures/ws-1011/grundlagen-der-informatik-gdi/folien/`.

Referenzen (Forts.)

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

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

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

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

Veranstaltungsüberblick

Grundlagen der Systemnahen Programmierung in C (GSPiC)

Teil A Konzept und Organisation

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Wintersemester 2012

http://www4.cs.fau.de/Lehre/WS12/V_GSPIC

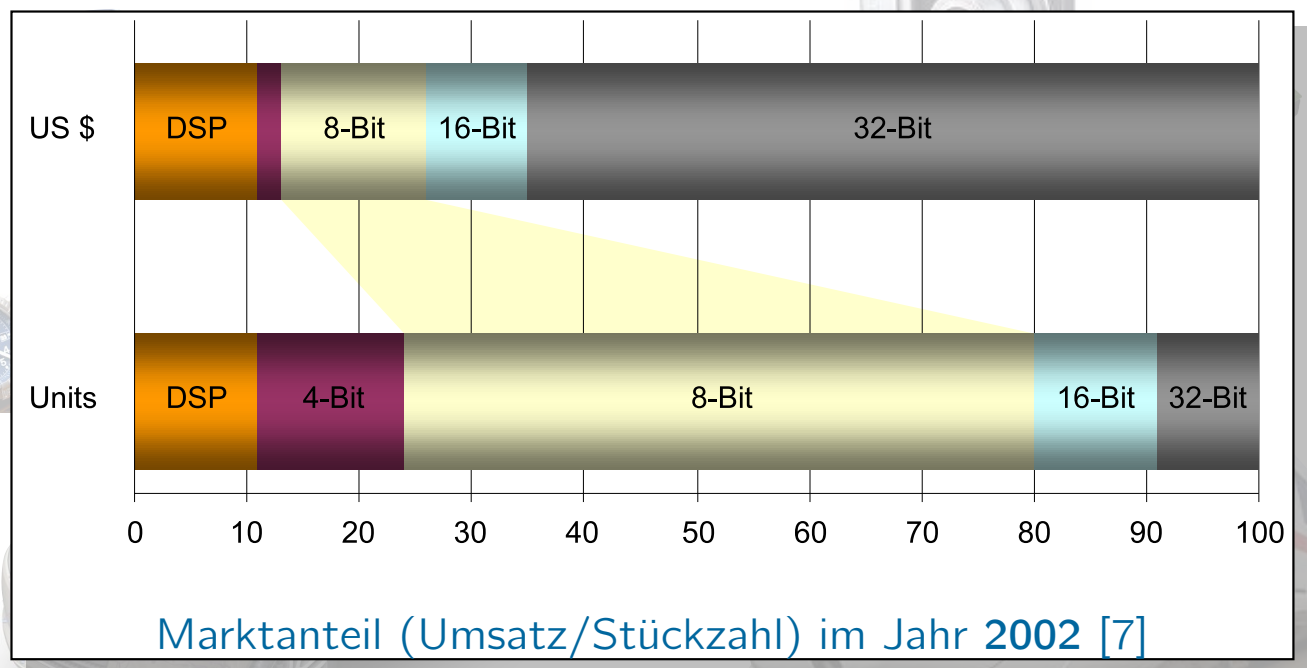
Überblick: Teil A Konzept und Organisation

Lernziele

- **Vertiefen** des Wissens über Konzepte und Techniken der Informatik für die Softwareentwicklung
  - Ausgangspunkt: Grundlagen der Informatik (GdI)
  - Schwerpunkt: Systemnahe Softwareentwicklung in C
- **Entwickeln** von Software in C für einen μ-Controller (μC)
  - SPiCboard-Lehrentwicklungsplattform mit ATmega-μC
  - 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 μ-Controller, **8-Bit oder kleiner** [6, 7]



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

Motivation: Eingebettete Systeme

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

Bei den oberen Zahlen ist gesunde Skepsis geboten

- Die Veröffentlichungen [6, 7] sind um die 10 Jahre alt!
- Man kann dennoch davon ausgehen, dass die relativen Größenordnungen nach wie vor stimmen
  - 2012 liegt der Anteil an 8-Bittern (vermutlich) noch weit über 50 Prozent
  - 4-Bitter dürften inzwischen jedoch weitgehend ausgestorben sein

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

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

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

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

Motivation: Die Sprache C

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

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

Motivation: GSPiC – Stoffauswahl und Konzept

- **Lehrziel:** Systemnahe Softwareentwicklung in C
  - Das ist ein sehr umfangreiches Feld: Hardware-Programmierung, Betriebssysteme, Middleware, Datenbanken, Verteilte Systeme, Übersetzerbau, ...
  - Dazu kommt dann noch das Erlernen der Sprache C selber
- **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 μ-Controller
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (EEI)

Vorlesungsskript

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

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

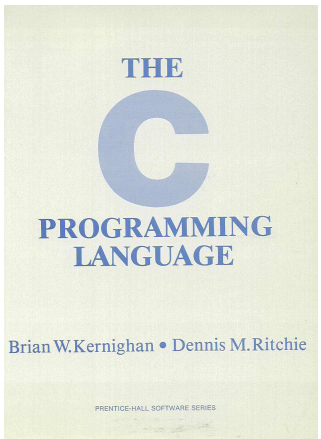
Literaturempfehlungen

[2] Für den Einstieg empfohlen:

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

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

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



Vorlesung

- Inhalt und Themen
  - Grundlegende Konzepte der systemnahen Programmierung
  - Einführung in die Programmiersprache C
    - Unterschiede zu Java
    - Modulkonzept
    - Zeiger und Zeigerarithmetik
  - Softwareentwicklung auf „der nackten Hardware“ (ATmega-μC)
    - Abbildung Speicher $\leftrightarrow$ Sprachkonstrukte
    - Unterbrechungen (*interrupts*) und Nebenläufigkeit
- Termin: Di 10:15–11:45, H14
  - insgesamt 9 Vorlesungstermine

↪ 2–7

Übungen

- 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 (AVR Studio)
    - Betreuung durch Übungsbetreuer
- Termin: Initial 5 Gruppen zur Auswahl
  - Anmeldung über Waffel (siehe Webseite): Heute, 14:15 – So, 20:00
  - Bei nur 2–3 Teilnehmern behalten wir uns eine Verteilung auf andere Gruppen vor. Ihr werdet in diesem Fall per E-Mail angeschrieben.

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

Programmieraufgaben

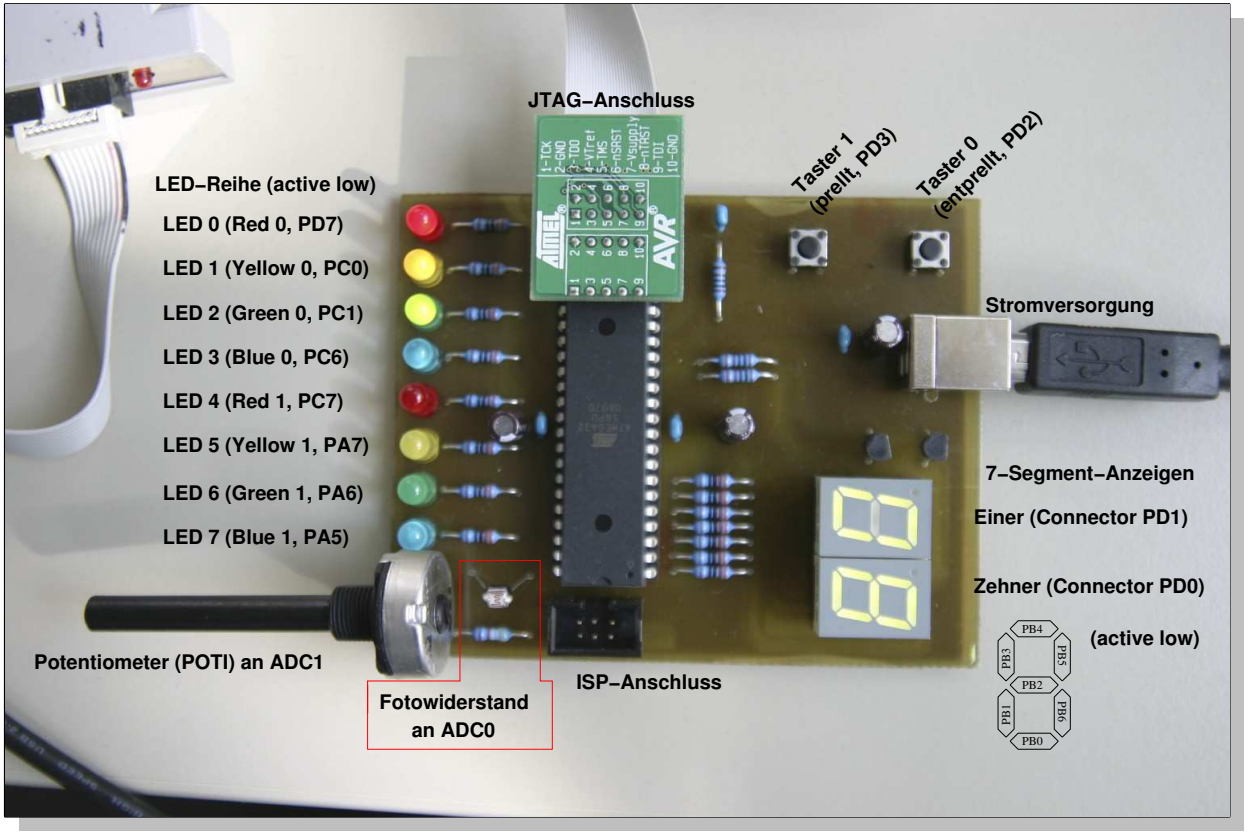
- Praktische Umsetzung des Vorlesungsstoffs
  - Fünf Programmieraufgaben (Abgabe ca. alle 14 Tage) ↪ 2–7
  - Bearbeitung wechselseitig alleine / mit Übungspartner
- Lösungen mit Abgabeskript am Rechner abgeben
  - Lösung wird durch Skripte überprüft
  - Wir korrigieren und bepunkten die Abgaben und geben sie zurück
  - Eine Lösung wird vom Teilnehmer an der Tafel erläutert (impliziert Anwesenheit!)
- ★ Abgabe der Übungsaufgaben ist freiwillig; ↪ 2–6
  es können jedoch bis zu **10% Bonuspunkte** für die Prüfungsklausur erarbeitet werden!

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

01-02-Konzept: 2012-10-09

Übungsplattform: Das SPiCboard

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



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

SPiCboard-Lötabend

- Die Fachschaften (EEI / ME) bieten einen „Lötabend“ an
  - Teilnahme ist freiwillig
  - (Erste) Löterfahrung sammeln beim Löten eines eigenen SPiCboards
- **Termin:** 23. Oktober, 18:15–21:00
- **Anmeldung:** über Waffel (siehe Webseite)
- **Kostenbeitrag:** SPiCBoard: ~~13 EUR~~ *bezahlt aus Studienbeiträgen*
  Progger: 22 EUR (falls gewünscht)

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

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

Prüfung und Modulnote

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

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

Semesterplanung

KW	Mo	Di	Mi	Do	Fr	Themen
42	15.10.	16.10. VL 1	17.10.	18.10.	19.10.	*Einführung, Organisation, Java nach C*
43	22.10.	23.10. VL 2	24.10.	25.10.	26.10.	*Abstraktion, Sprachüberblick, Datentypen*
44	29.10.	30.10. VL 3	31.10. A1 (Blink)	*01.11.*	02.11.	*Ausdrücke, Kontrollstrukturen*
45	05.11.	06.11. VL 4	07.11. A2 (Snake)	08.11.	09.11.	*Funktionen, Variablen, Präprozessor, Programmstruktur, Module*
46	12.11.	11.11.	14.11.	15.11.	16.11.	
47	19.11.	20.11. VL 5	21.11. A3 (Spiel)	22.11.	23.11.	*Zeiger*
48	26.11.	27.11. VL 6	28.11.	29.11.	30.11.	*Mikrocontroller-Systemarchitektur, volatile, Verbundtypen (struct, union)*
49	03.12.	04.12. A4 (LED)	05.12.	06.12.	07.12.	
50	10.12.	11.12.	12.12.	13.12.	14.12.	
51	17.12.	18.12.	19.12.	20.12.	21.12.	
52	24.12.	25.12. *Weihnachten*	26.12.	27.12.	28.12.	
01	31.12.	01.01. *Weihnachten*	02.01.	03.01.	04.01.	
02	07.01.	08.01. VL 7	09.01. A5 (Ampel)	10.01.	11.01.	*Interrupts, Nebenläufigkeit*
03	14.01.	15.01.	16.01.	17.01.	18.01.	
04	21.01.	22.01. VL 8	23.01.	24.01.	25.01.	*Speicherorganisation, Zusammenfassung*
05	28.01.	29.01. Wiederholung	30.01.	31.01.	01.02.	
06	04.02.	05.02. VL 9	06.02.	07.02.	08.02.	*Fragestunde*

http://www4.cs.fau.de/Lehre/WS12/V_GSPIC

Beteiligte Personen, LS Informatik 4

Dozenten Vorlesung

Daniel Lohmann

Jürgen Kleinöder

Organisatoren des Übungsbetriebs

Rainer Müller

Moritz Strübe

Beteiligte Personen, LS Informatik 4 (Forts.)

Techniker (Ausleihe SPiCboard und Debugger)

Harald Junggunst

Christian Preller

Daniel Christiani

Übungsleiter

Nils Ballmann

Robert Heyn

Martin Klüpfel

Markus Müller

Andreas Och

Grundlagen der Systemnahen Programmierung in C (GSPiC)

Teil B Einführung in C

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Wintersemester 2012

http://www4.cs.fau.de/Lehre/WS12/V_GSPIC

Überblick: Teil B Einführung in C

Das erste C-Programm

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

```c
#include <stdio.h>

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

- Übersetzen und Ausführen (auf einem UNIX-System)

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

Gar nicht so schwer :-)

Das erste C-Programm – Vergleich mit Java

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

```c
#include <stdio.h>

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

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

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


Das erste C-Programm – Erläuterungen [Handout]

- **C**-Version zeilenweise erläutert
  1. Für die Benutzung von `printf()` wird die **Funktionsbibliothek** `stdio.h` mit der **Präprozessor-Anweisung** `#include` eingebunden.
  3. Ein C-Programm startet in `main()`, einer **globalen Funktion** vom Typ `int`, die in genau einer **Datei** definiert ist.
  5. Die Ausgabe einer Zeichenkette erfolgt mit der **Funktion** `printf()`. (`\n` $\leadsto$ 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`. [$\hookrightarrow$ GDI, IV-191]
  6. Rückkehr zum Betriebssystem.

Das erste C-Programm für einen μ-Controller

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

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

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

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

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

μ-Controller-Programmierung ist „irgendwie anders“.

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

Das erste C-Programm für einen μ-Controller

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

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


μ-Controller-Programm – Erläuterungen [Handout]

- μ-Controller-Programm zeilenweise erläutert
  (Beachte Unterschiede zur Linux-Version ↪ 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 μ-Controller-Programm läuft **endlos** ⇝ `main()` terminiert nie.

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

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

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

Das zweite C-Programm – Eingabe unter Linux

- Benutzerinteraktion (Lesen eines Zeichens) unter Linux:

```
#include <stdio.h>

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

  printf("Press key: ");
  int key = getchar();

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

Die `getchar()`-Funktion liest ein Zeichen von der Standardeingabe (hier: Tastatur). Sie „wartet“ gegebenenfalls, bis ein Zeichen verfügbar ist. In dieser Zeit entzieht das Betriebssystem den Prozessor.

Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

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

Warten auf Tasterdruck – Erläuterungen [Handout]

- Benutzerinteraktion mit SPiCboard zeilenweise erläutert
  - 5 Wie die LED ist der Taster mit einem **digitalen IO-Pin** des μ-Controllers verbunden. Hier konfigurieren wir Pin 2 von Port D als **Eingang** durch **Löschen** des entsprechenden Bits im Register `DDRD`.
  - 6 Durch **Setzen** von Bit 2 im Register `PORTD` wird der interne Pull-Up-Widerstand (hochohmig) aktiviert, über den V_{CC} anliegt $\leadsto$ PD2 = *high*.
  - 13-14 **Aktive Warteschleife:** Wartet auf Tastendruck, d. h. solange PD2 (Bit 2 im Register `PIND`) *high* ist. Ein Tasterdruck zieht PD2 auf Masse $\leadsto$ Bit 2 im Register `PIND` wird *low* und die Schleife verlassen.

Zum Vergleich: Benutzerinteraktion als Java-Programm

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

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

Benutzerinteraktion als Java-Programm – Erläuterungen [Handout]

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

- Benutzerinteraktion in Java zeilenweise erläutert

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

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

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

  23-26 Der Knopfdruck wird durch eine `actionPerformed()`-Nachricht (Methodenaufruf) signalisiert.

Ein erstes Fazit: Von Java → C (Syntax)

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

Ein erstes Fazit: Von Java → C (Idiomatik)

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

Ein erstes Fazit: Von Java → C (Philosophie)

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

Ein erstes Fazit: μ-Controller-Programmierung

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

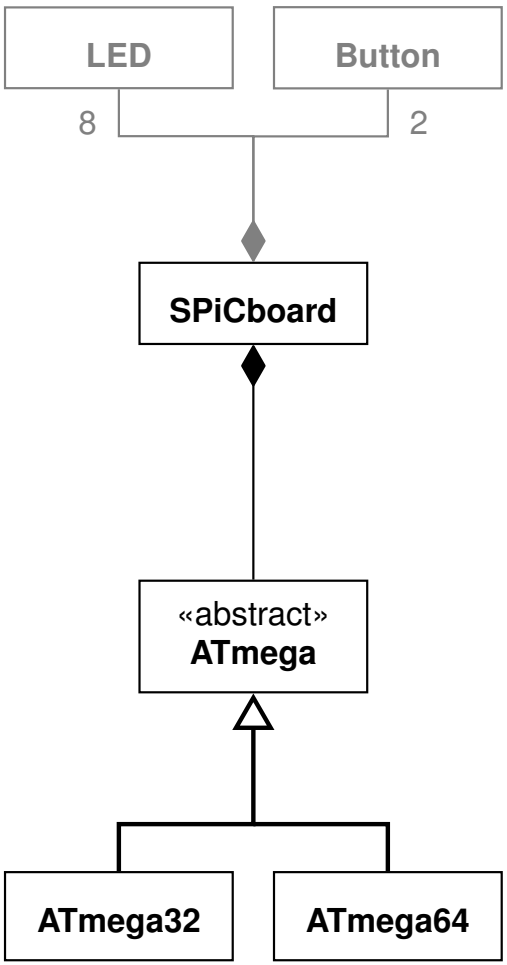
Die Maschinennähe von C zeigt sich insbesondere auch bei der μ-Controller-Programmierung!

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

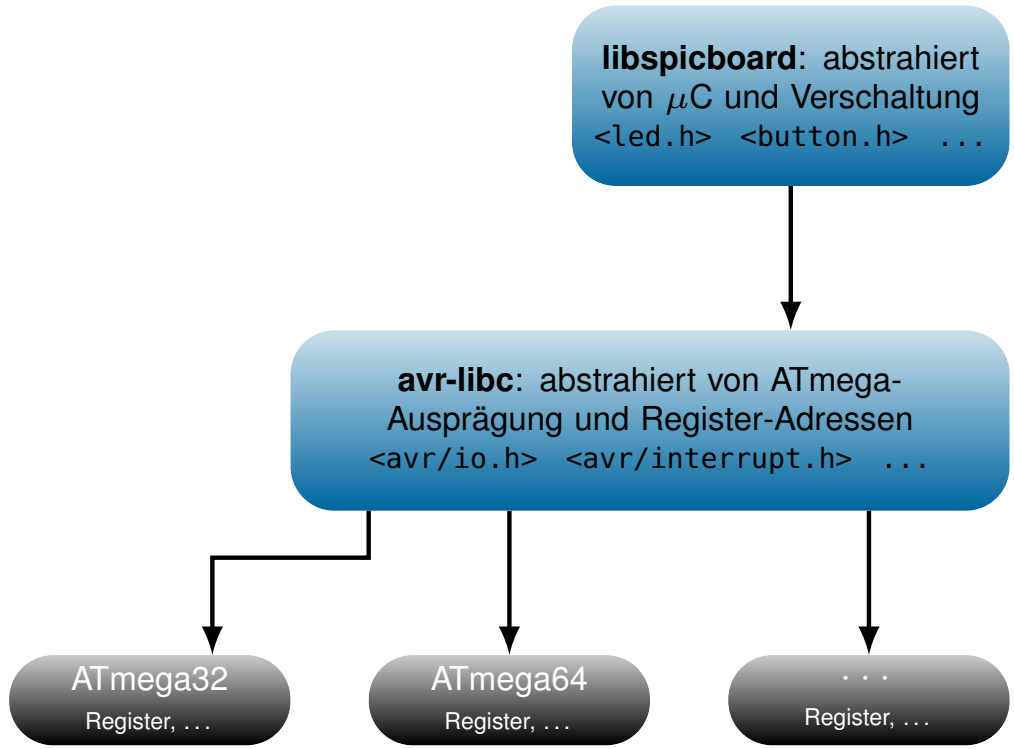
Ansatz: Mehr Abstraktion durch problemorientierte Bibliotheken

Abstraktion durch Softwareschichten: SPiCboard

Hardwaresicht



Softwareschichten



Abstraktion durch Softwareschichten: *LED → on* im Vergleich

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

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

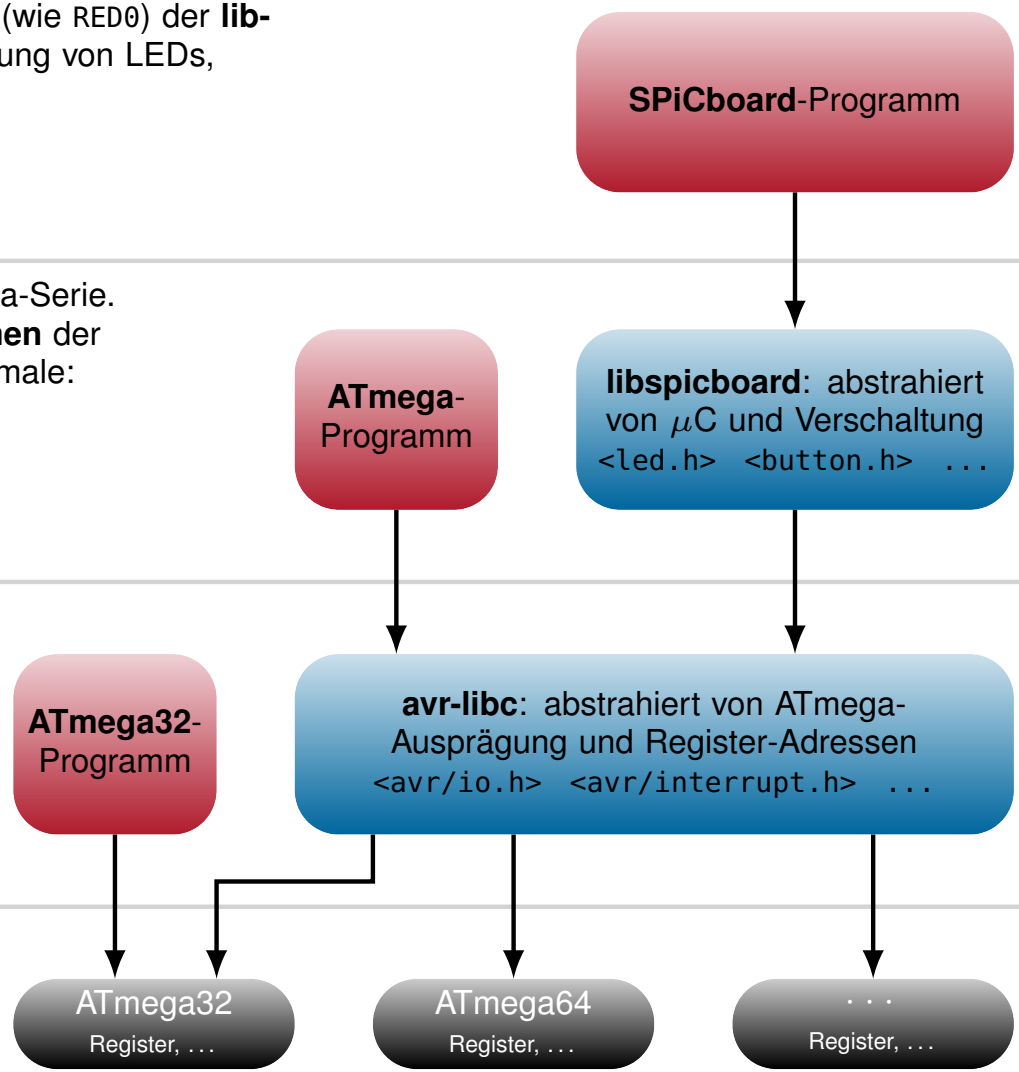
Programm läuft auf **jedem** μC der ATmega-Serie. Es verwendet **symbolische Registernamen** der **avr-libc** (wie `PORTD`) und allgemeine Merkmale:

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

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

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

Ziel: Schalte LED `RED0` auf SPiCboard an:



Abstraktion durch Softwareschichten: Vollständiges Beispiel

Bisher: Entwicklung mit avr-libc

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

void main() {
  // initialize hardware

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

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

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

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

(vgl. ↪ 3–8)

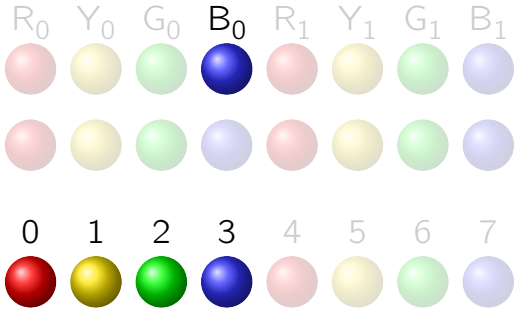
Nun: Entwicklung mit libspicboard

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

void main() {

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

  // greet user
  sb_led_on(RED0);

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

- Hardwareinitialisierung entfällt
- Programm ist einfacher und verständlicher durch **problemspezifische Abstraktionen**
  - Setze Bit 7 in `PORTD`
    ↦ `sb_set_led(RED0)`
  - Lese Bit 2 in `PORTD`
    ↦ `sb_button_getState(BUTTON0)`

Abstraktionen der libspicboard: Kurzüberblick

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

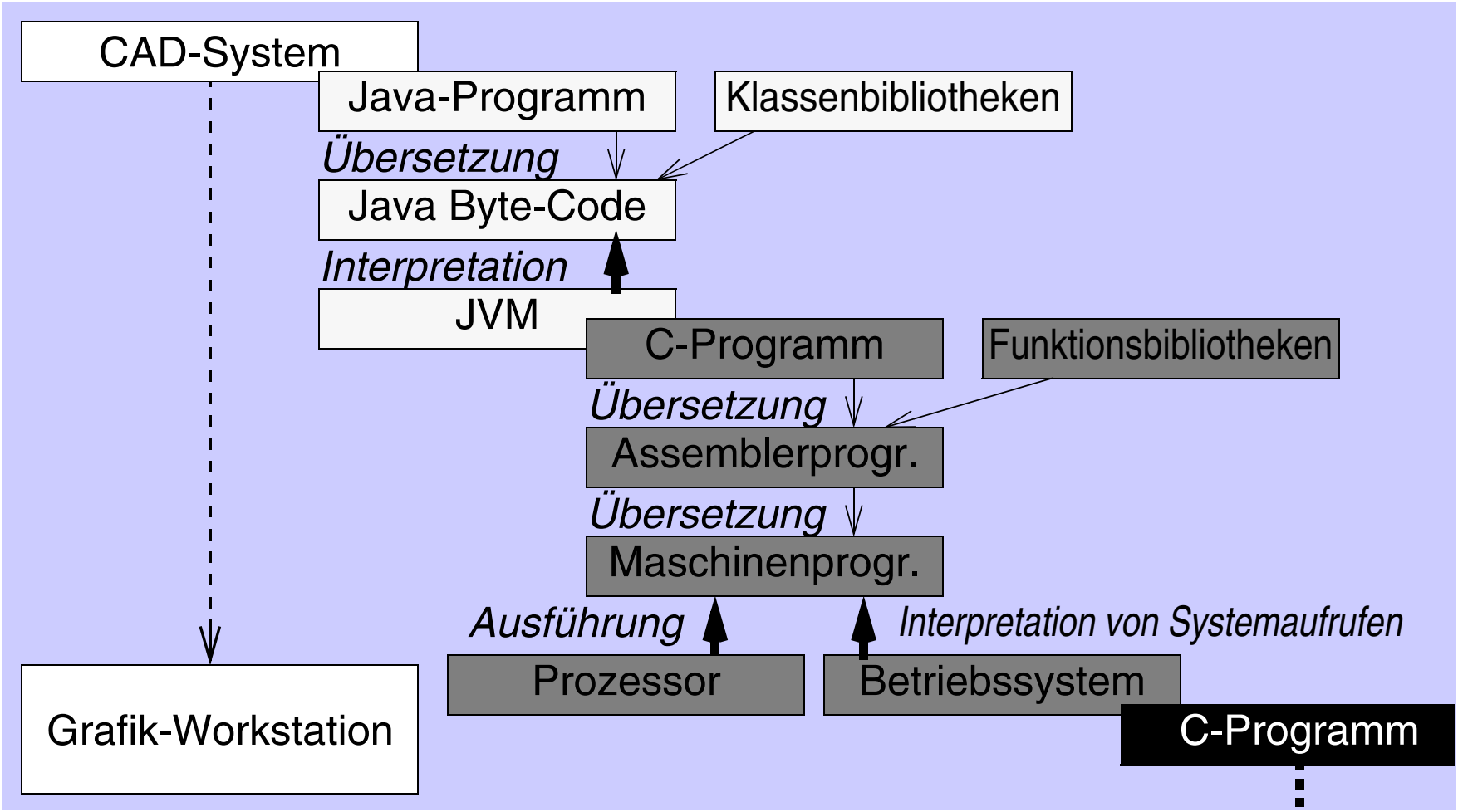


  - 7-Seg-Modul (`#include <7seg.h>`)
    - Ganzzahl $n \in \{-9 \ldots 99\}$ ausgeben: `sb_7seg_showNumber(47)` ⇝

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

Softwareschichten im Allgemeinen

Diskrepanz: Anwendungsproblem $\longleftrightarrow$ Abläufe auf der Hardware



Ziel: Ausführbarer Maschinencode

Die Rolle des Betriebssystems

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

Die Rolle des Betriebssystems (Forts.)

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

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

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

Beispiel: Fehlererkennung durch Betriebssystem

Linux: Division durch 0

```
1  #include <stdio.h>
2
3
4  int main(int argc, char** argv) {
5    int a = 23;
6    int b = 0;
7
8    b = 4711 / (a-23);
9    printf("Ergebnis: %d\n", b);
10
11   return 0;
12 }
```

Übersetzen und Ausführen ergibt:

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

⇝ 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

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

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

Bezeichner [= Java]

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

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

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

Schlüsselwörter [≈Java]

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

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

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

Schlüsselwörter in C99

[Handout]

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

Literale [= Java]

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

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

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

Anweisungen [= Java]

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

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

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

Ausdrücke [= Java]

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

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

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

Überblick: Teil B Einführung in C

Was ist ein Datentyp? [↪ GDI, II-11]

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

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

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

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

Primitive Datentypen in C

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

Integertypen [≈Java][↪ GDI, II-13]

Integertyp	Verwendung	Literalformen
`char`	kleine Ganzzahl oder Zeichen	`'A'`, `65`, `0x41`, `0101`
`short [int]`	Ganzzahl (int ist optional)	s. o.
`int`	Ganzzahl „natürlicher Größe“	s. o.
`long [int]`	große Ganzzahl	`65L`, `0x41L`, `0101L`
`long long [int]`	sehr große Ganzzahl	`65LL`, `0x41LL`, `0101LL`

Typ-Modifizierer	werden vorangestellt	Literal-Suffix
signed	Typ ist vorzeichenbehaftet (Normalfall)	-
unsigned	Typ ist vorzeichenlos	`U`
const	Variable des Typs kann nicht verändert werden	-

- Beispiele (Variablendefinitionen)

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

Integertypen: Größe und Wertebereich [≠Java]

- Die interne Darstellung (Bitbreite) ist implementierungsabhängig

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

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

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

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

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

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

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

Typ-Aliase mit `typedef` [≠Java]

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

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

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

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

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

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

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

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

Definierte Bitbreiten sind bei der μC-Entwicklung sehr wichtig!

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

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

Aufzählungstypen mit enum [≈Java]

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

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

- Beispiel
  - Definition:

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

  - Verwendung:

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

- Vereinfachung der Verwendung durch `typedef`
  - Definition:

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

  - Verwendung:

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


enum ↦ int [≠Java]

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

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

  - Es ist auch möglich, Werte direkt zuzuweisen

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

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

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

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

Das entspricht der **C-Philosophie**! ↪ 3–14

Fließkommatypen [≈Java][↪ GDI, II-14]

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

„Effizienz durch Maschinennähe“ ↪ 3–14

Fließkommazahlen + μC-Plattform = $$$

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

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

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

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

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

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

ASCII-Code-Tabelle (7 Bit)

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

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

Zeichenketten (Strings) [≠Java]

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

- Beispiel (Linux)

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

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

Ausblick: Komplexe Datentypen

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

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

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

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

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

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

- Wir betrachten diese detailliert in späteren Kapiteln

Überblick: Teil B Einführung in C

Arithmetische Operatoren [= Java]

- Stehen für alle Ganzzahl- und Fließkommatypen zur Verfügung

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

- Zusätzlich nur für Ganzzahltypen:

%	Modulo (Rest bei Division)

Inkrement-/Dekrement-Operatoren [=Java][↪ GDI, II-26]

- Stehen für Ganzzahltypen und Zeigertypen zur Verfügung

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

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

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

Vergleichsoperatoren [=Java][↪ GDI, II-29]

- Vergleichen von zwei Ausdrücken

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

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

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


Logische Operatoren [≈Java][↪ GDI, II-30]

- Verknüpfung von Wahrheitswerten (wahr / falsch), kommutativ

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

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

Logische Operatoren – Auswertung [=Java]

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

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

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

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

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

- Kann überraschend sein, wenn Teilausdrücke Nebeneffekte haben

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


Zuweisungsoperatoren [= Java]

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

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

Zuweisungen sind Ausdrücke!

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

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

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

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

Besonders gefährlich: Verwendung von = statt ==

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

- Typischer „Anfängerfehler“ in Kontrollstrukturen:

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

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

Bitoperationen [= Java][↪ GDI, II-32]

- Bitweise Verknüpfung von Ganzzahltypen, kommutativ

&	bitweises „Und“ (Bit-Schnittmenge)	1 & 1 → 1 1 & 0 → 0 0 & 0 → 0
|	bitweises „Oder“ (Bit-Vereinigungsmenge)	1 | 1 → 1 1 | 0 → 1 0 | 0 → 0
∧	bitweises „Exklusiv-Oder“ (Bit-Antivalenz)	1 ∧ 1 → 0 1 ∧ 0 → 1 0 ∧ 0 → 0
~	bitweise Inversion (Einerkomplement, unär)	~ 1 → 0 ~ 0 → 1

Bitoperationen (Forts.) [= Java][↪ GDI, II-32]

- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

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

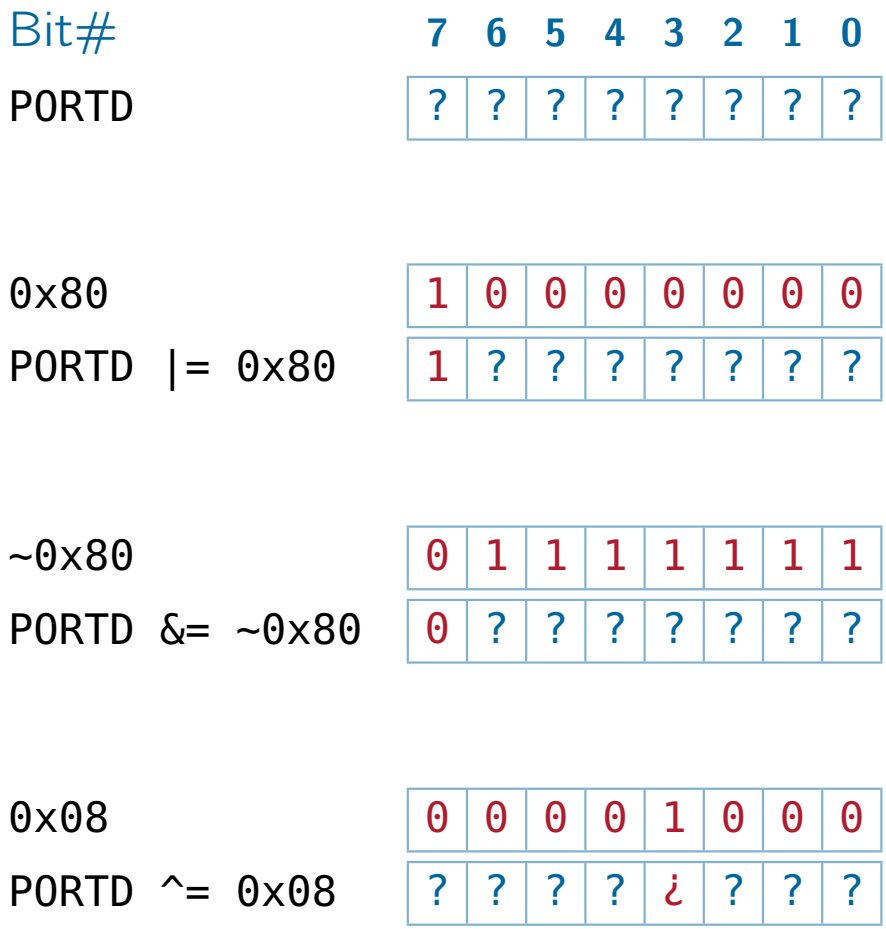
  `>>` bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

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

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

Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen



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

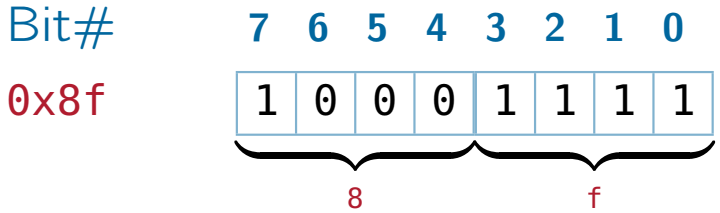
Setzen eines Bits durch **Ver-odern** mit Maske, in der nur das Zielbit 1 ist

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

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

Bitoperationen – Anwendung (Forts.)

- Bitmasken werden gerne als Hexadezimal-Literale angegeben



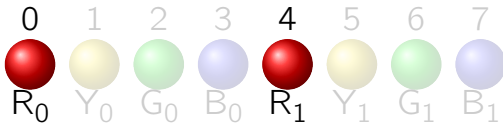
Jede Hex-Ziffer repräsentiert genau ein Halb-Byte (*Nibble*) ⇝ Verständlichkeit

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

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

- Zusammen mit der Oder-Operation auch für komplexere Masken

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

  sb_led_set_all_leds (mask);

  while(1) ;
}
```

⇝ 

Bedingte Auswertung [≈Java][↪ GDI, II-34]

- Formulierung von Bedingungen in Ausdrücken

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

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

- Beispiel

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


Sequenzoperator [≠Java]

- Reihung von Ausdrücken

  $Ausdruck_1$, $Ausdruck_2$

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

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

Vorrangregeln bei Operatoren [≈Java][↪ GDI, II-35]

	Klasse	Operatoren	Assoziativität

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

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

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

Typumwandlung in Ausdrücken

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

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

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

int8_t: 75 — int: 300 — int: 75

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

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

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

int8_t: 75 — int32_t: 300 — int32_t: 75

Typumwandlung in Ausdrücken (Forts.)

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

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

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

int8_t: 75 — double: 300.0 — double: 4.0 — double: 75.0

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

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

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

int: 0 — unsigned: 65535 — unsigned: 0

↝ Überraschende Ergebnisse bei negativen Werten!

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

Typumwandlung in Ausdrücken – Typ-Casts

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

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

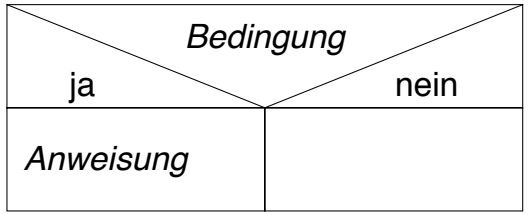
```
int s = -1, res;        // range: -32768 --> +32767
unsigned u = 1;         // range: 0 --> 65535

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


Überblick: Teil B Einführung in C

Bedingte Anweisung [= Java][↪ GDI, II-46]

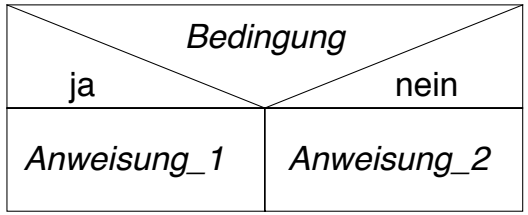
- `if`-Anweisung (bedingte Anweisung)

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



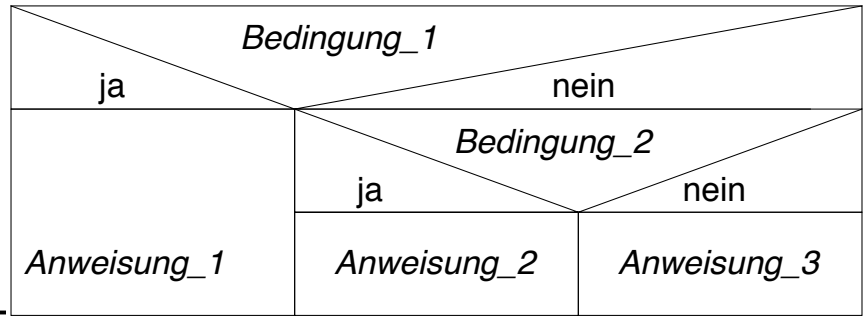
- `if-else`-Anweisung (einfache Verzweigung)

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



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

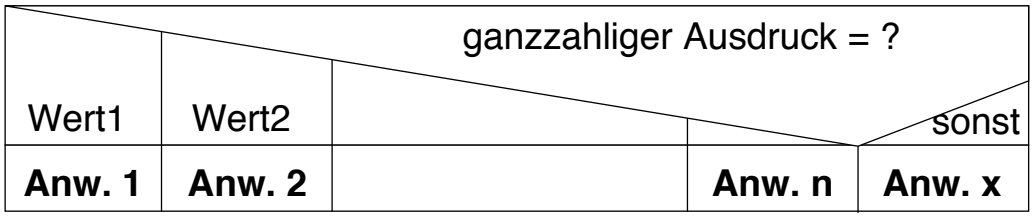
```
if ( Bedingung1 )
  Anweisung1;
else if ( Bedingung2 )
  Anweisung2;
else
  Anweisung3;
```



Fallunterscheidung [= Java][↪ GDI, II-49]

`switch`-Anweisung (Fallunterscheidung)

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



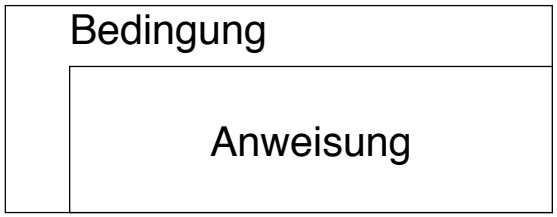

```
switch ( Ausdruck ) {
  case Wert_1:
    Anweisung_1;
    break;
  case Wert_2:
    Anweisung_2;
    break;
  ...
  case Wert_n:
    Anweisung_n;
    break;
  default:
    Anweisung_x;
}
```

Abweisende und nicht-abweisende Schleife [= Java]

Abweisende Schleife

[↪ GDI, II-57] [↪ GDI-Ü, II-9]

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



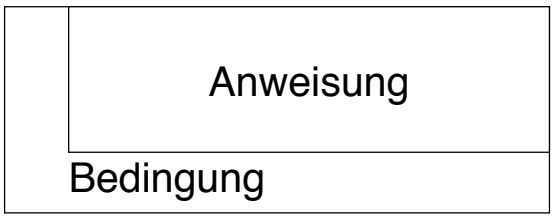

```
while( Bedingung )
  Anweisung;
```

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

Nicht-abweisende Schleife

[↪ GDI, II-58]

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



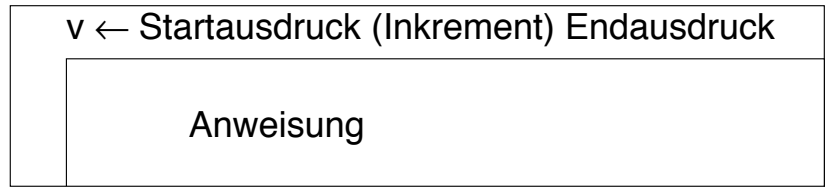

```
do
  Anweisung;
while( Bedingung );
```

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

Zählende Schleife [= Java][↪ GDI, II-51]

- `for`-Schleife (Laufanweisung)

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



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

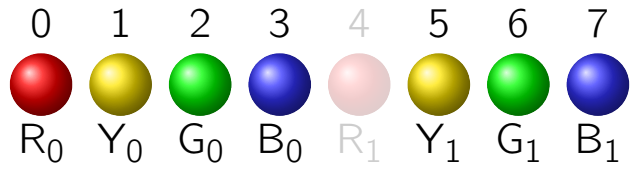
```
uint8_t sum = 0; // calc sum 1+...+10
for (uint8_t n = 1; n < 11; n++) {
  sum += n;
}
sb_7seg_showNumber( sum );
```

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

Schleifensteuerung [= Java][↪ GDI, II-54]

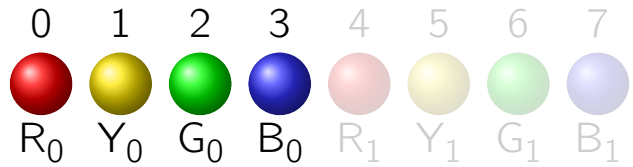
- Die `continue`-Anweisung beendet den aktuellen Schleifendurchlauf
  ⇝ Schleife wird mit dem nächsten Durchlauf fortgesetzt

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



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

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



Überblick: Teil B Einführung in C

Was ist eine Funktion?

- **Funktion** := Unterprogramm [↪ GDI, II-79]
  - Programmstück (Block) mit einem Bezeichner
  - Beim Aufruf können Parameter übergeben werden
  - Bei Rückkehr kann ein Rückgabewert zurückgeliefert werden
- Funktionen sind elementare Programmbausteine
  - Gliedern umfangreiche Aufgaben in kleine, beherrschbare Komponenten
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)

Funktion ↦ Abstraktion

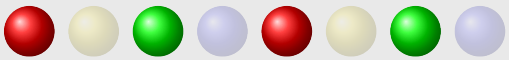
↪ 4–1

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

Beispiel

- Funktion (Abstraktion) `sb_led_set_all_leds()`

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



- Implementierung in der `libspicboard`

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

Sichtbar: Bezeichner und formale Paramter

Unsichtbar: Tatsächliche Implementierung

Funktionsdefinition [≈Java]

- Syntax: *Typ Bezeichner* **(** $FormaleParam_{opt}$ **)** **{***Block***}**
  - *Typ* — Typ des Rückgabewertes der Funktion, `void` falls kein Wert zurückgegeben wird [=Java]
  - *Bezeichner* — Name, unter dem die Funktion aufgerufen werden kann ↪ 5–3 [=Java]
  - $FormaleParam_{opt}$ — Liste der formalen Parameter: $Typ_1\ Bez_{1\,opt}$**,** ..., $Typ_n\ Bez_{n\,opt}$ (Parameter-Bezeichner sind optional) [=Java]
    `void`, falls kein Parameter erwartet wird [≠Java]
  - {*Block*} — Implementierung; formale Parameter stehen als lokale Variablen bereit [=Java]
- Beispiele:

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

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


Funktionsaufruf [=Java]

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

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

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

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

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

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

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

Funktionsaufruf – Parameterübergabe [≠Java]

- Generelle Arten der Parameterübergabe [↪ GDI, II-88]
  - *Call-by-value*: Die formalen Parameter sind Kopien der tatsächlichen Parameter. Änderungen in den formalen Parametern gehen mit Verlassen der Funktion verloren. **Dies ist der Normalfall in C.**
  - *Call-by-reference*: Die formalen Parameter sind Verweise (Referenzen) auf die tatsächlichen Parameter. Änderungen in den formalen Parametern betreffen auch die tatsächlichen Parameter. **In C nur indirekt über Zeiger möglich.** ↪ 13–5
- Des weiteren gilt
  - Arrays werden in C immer *by-reference* übergeben [=Java]
  - Die Auswertungsreihenfolge der Parameter ist undefiniert! [≠Java]

Funktionsaufruf – Rekursion [=Java]

- Funktionen können sich auch selber aufrufen (Rekursion)

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

Rekursive Definition der Fakultätsfunktion.

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

Rekursion $\mapsto$ $$$

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

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

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

Funktionsdeklaration [≠Java]

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

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

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

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

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

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

Funktionsdeklaration (Forts.)

[≠Java]

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

Achtung: C erzwingt dies nicht!

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

Funktionsdeklaration (Forts.)

[≠Java]

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

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

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

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

10 **Was wird hier ausgegeben?**

Funktionsdeklaration (Forts.) [≠Java]

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

```
#include <stdio.h>

void foo(); // "open" declaration

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

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

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

Funktionsdeklaration (Forts.) [≠Java]

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

Achtung: Verwechslungsgefahr

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

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

Überblick: Teil B Einführung in C

Variablendefinition [≈Java]

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

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

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

Variablendefinition (Forts.)

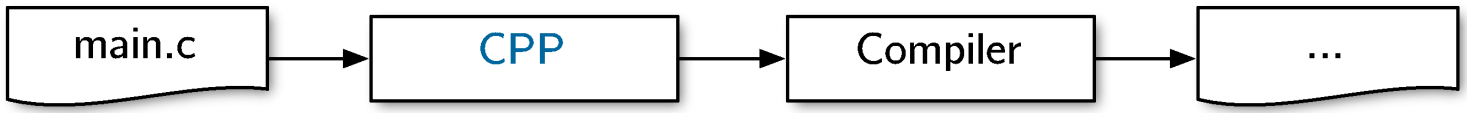
- Variablen können an verschiedenen Positionen definiert werden
  - Global — außerhalb von Funktionen, üblicherweise am Kopf der Datei
  - Lokal — zu Beginn eines `{ Blocks }`, direkt nach der öffnenden Klammer — C89
  - Lokal — überall dort, wo eine Anweisung stehen darf — C99

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

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

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

Überblick: Teil B Einführung in C

Der C-Präprozessor [≠Java]



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

Präprozessor-Direktiven [≠Java]

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

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

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

Präprozessor – Verwendungsbeispiele [≠Java]

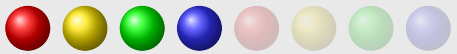
Einfache Makro-Definitionen

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

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

Verwendung

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

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

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

}
```



Präprozessor – Gefahren [≠Java]

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

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

<< hat geringere Präzedenz als *

⇝ `n = 1 << 2 * 3`

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

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

⇝ `n = (1 << 2) * 3`

  - Aber nicht alle

```
#define max(a,b) ((a > b) ? a : b)
n = max( x++, 7 )
```

a++ wird ggf. zweimal ausgewertet

⇝ `n = ((x++ > 7) ? x++ : 7)`

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

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


Grundlagen der Systemnahen Programmierung in C (GSPiC)

Teil C Systemnahe Softwareentwicklung

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Wintersemester 2012

http://www4.cs.fau.de/Lehre/WS12/V_GSPIC

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μC-Systemarchitektur

15 Nebenläufigkeit

16 Speicherorganisation

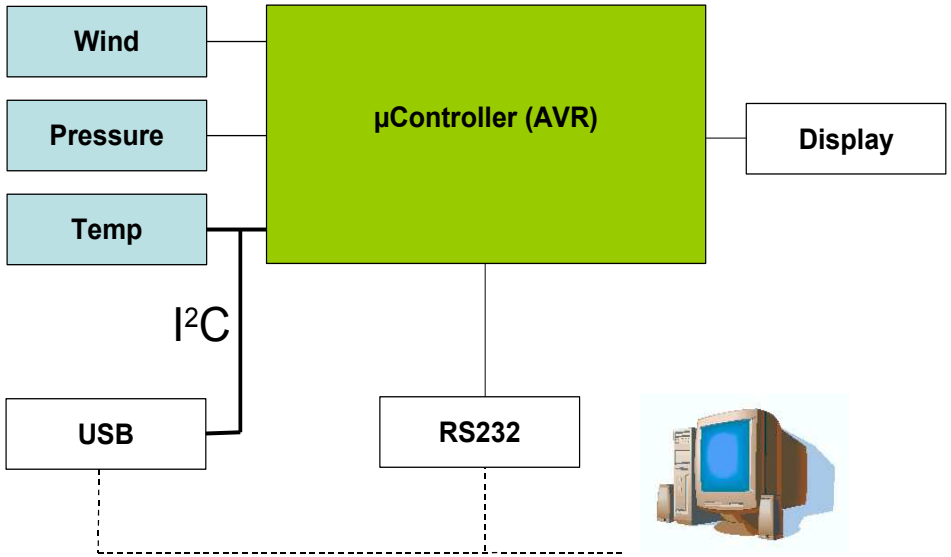
Softwareentwurf

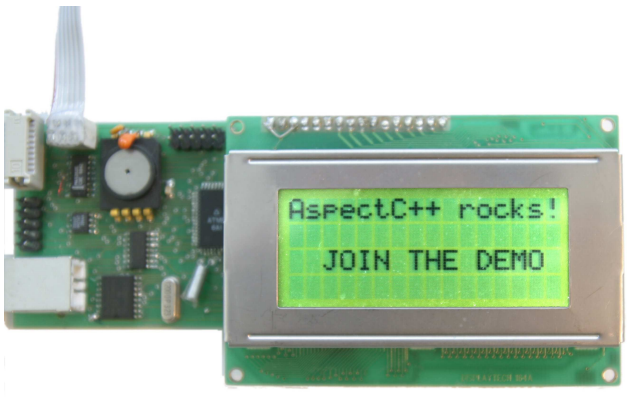
- Softwareentwurf: Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
  - Ziel: Zerlegung des Problems in beherrschbare Einheiten
- Es gibt eine Vielzahl von Softwareentwurfs-Methoden
  - Objektorientierter Entwurf [↪ GDI, IV]
    - Stand der Kunst
    - Dekomposition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

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

Beispiel-Projekt: Eine Wetterstation

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



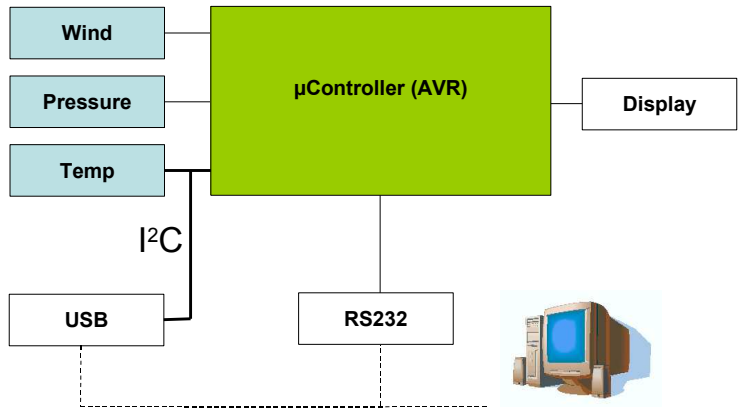


Wie sieht die **funktionale Dekomposition** der Software aus?

Funktionale Dekomposition: Beispiel

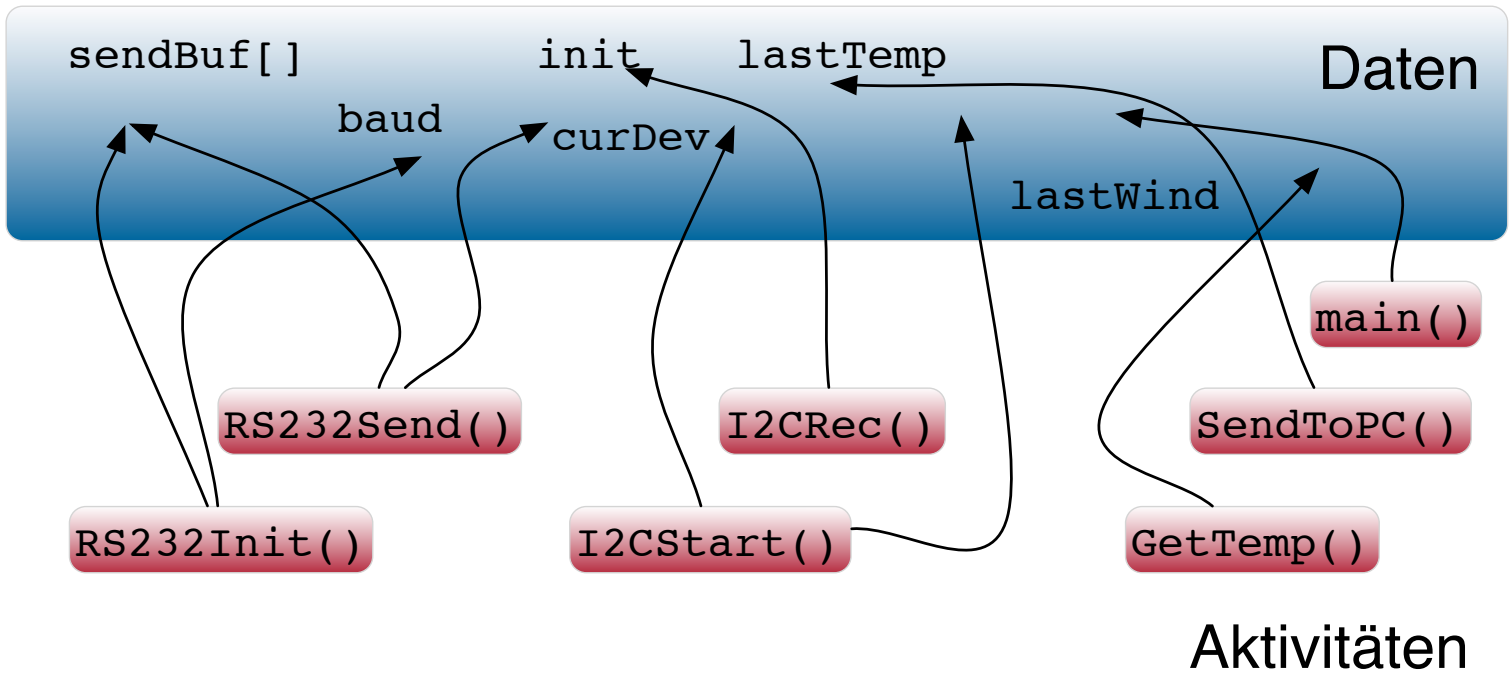
Funktionale Dekomposition der Wetterstation (Auszug):

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



Funktionale Dekomposition: Probleme

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



Funktionale Dekomposition: Probleme

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

Prinzip der **Trennung der Belange**

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

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

Zugriff auf Daten (Variablen)

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

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

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

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


Zugriff auf Daten (Variablen) (Forts.)

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

↪ 1–3

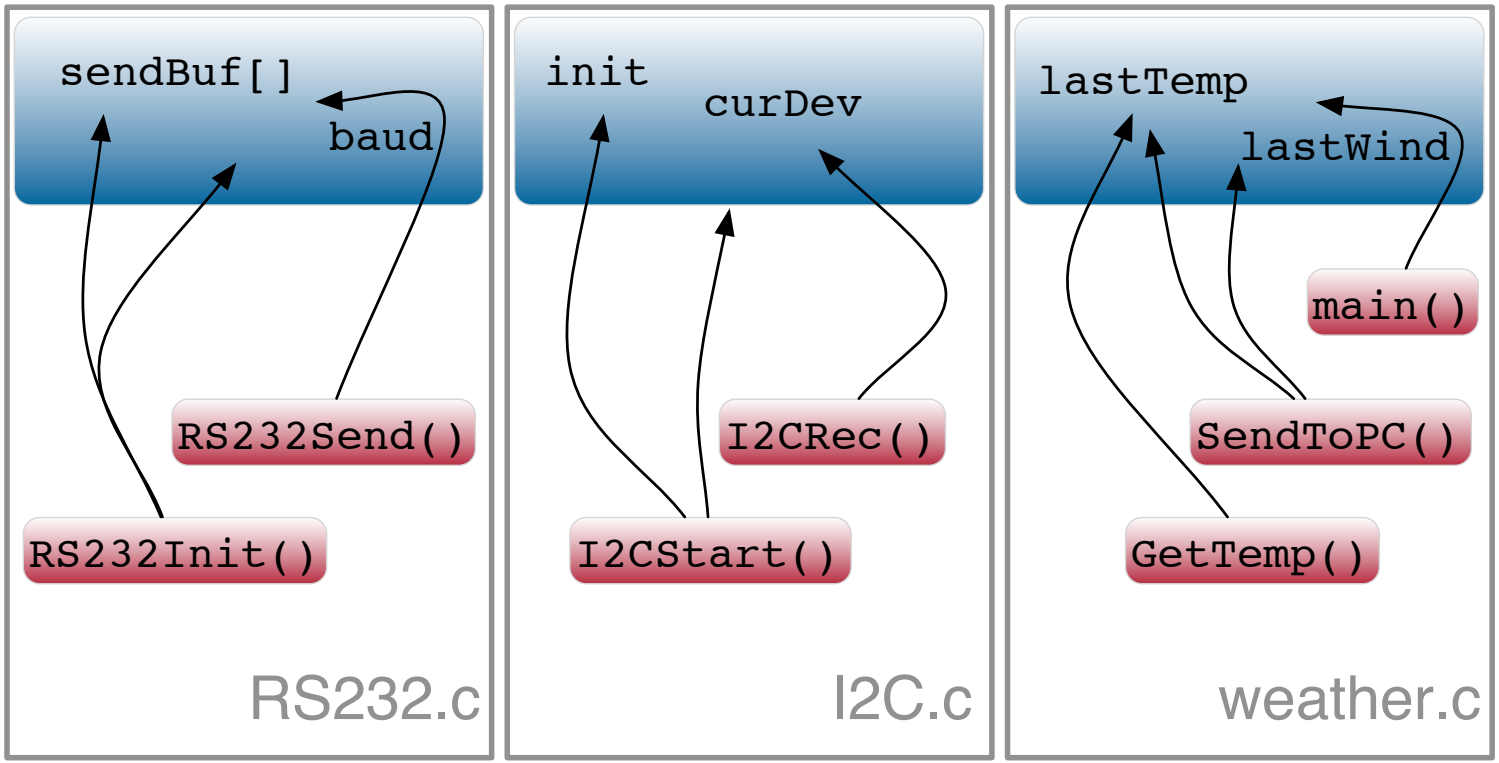
Konsequenz: Globale Variablen vermeiden!

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

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

Lösung: Modularisierung

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



Was ist ein Modul?

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

Modul ↦ Abstraktion ↪ 4–1

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

Module in C

[≠Java]

- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein idiomatisch (über Konventionen) realisiert ↪ 3–13
  - Modulschnittstelle ↦ `.h`-Datei (enthält Deklarationen ↪ 9–7)
  - Modulimplementierung ↦ `.c`-Datei (enthält Definitionen ↪ 9–3)
  - Modulverwendung ↦ `#include <Modul.h>`

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

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

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

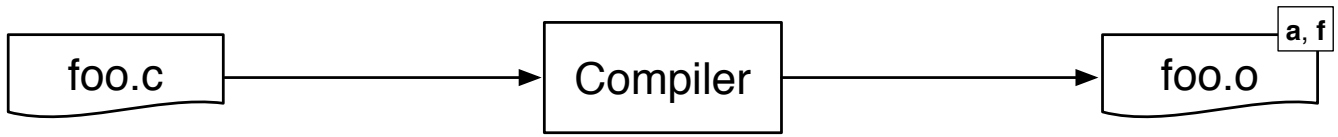
RS232.c: **Implementierung (nicht öffentl.)**
Definition der bereitgestellten Funktionen (und ggf. Daten)

Ggf. modulinterne Hilfsfunktionen und Daten (`static`)

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

Module in C – Export [≠Java]

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



Quelldatei (`foo.c`)

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

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

Objektdatei (`foo.o`)

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

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

Module in C – Import [≠Java]

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

Quelldatei (`bar.c`)

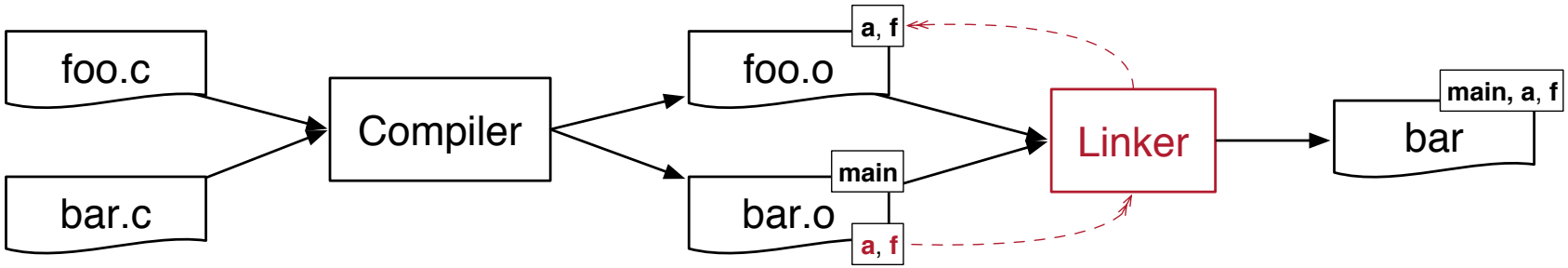
```
extern uint16_t a;
// declare
void f(void);       // declare

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

Objektdatei (`bar.o`)

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

Module in C – Import (Forts.) [≠Java]

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



Linken ist nicht typsicher!

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

⇝ Typsicherheit muss beim **Übersetzen** sichergestellt werden

⇝ Einheitliche Deklarationen durch gemeinsame Header-Datei

Module in C – Header [≠Java]

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

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

  - Globale Variablen durch `extern`

```
extern uint16_t a;
```

Das `extern` unterscheidet eine Variablendeklaration von einer Variablendefinition.

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

Module in C – Header (Forts.)

[≠Java]

Modulschnittstelle: `foo.h`

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

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

#endif // _FOO_H
```

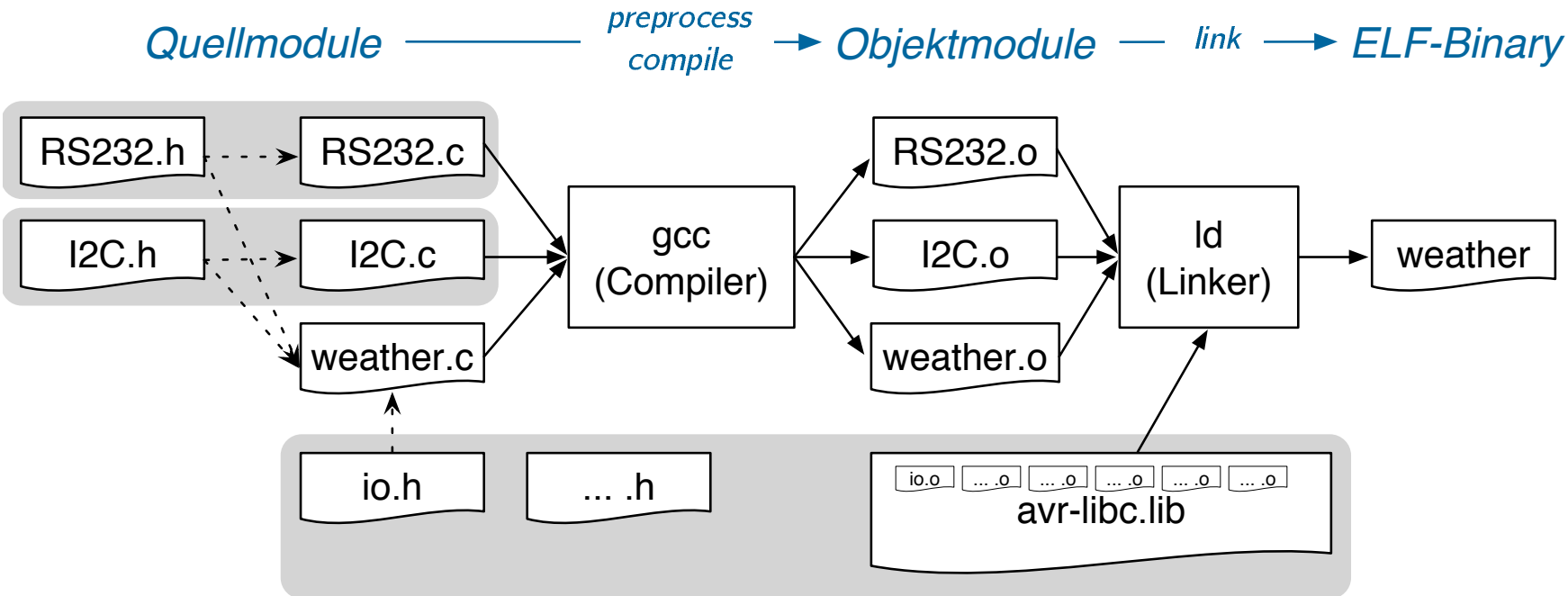
Modulimplementierung `foo.c`

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


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

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

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

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


Zurück zum Beispiel: Wetterstation



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

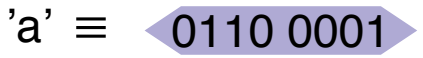
Zusammenfassung

- Prinzip der Trennung der Belange $\leadsto$ Modularisierung
  - Wiederverwendung und Austausch wohldefinierter Komponenten
  - Verbergen von Implementierungsdetails
- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern idiomatisch durch Konventionen realisiert
  - Modulschnittstelle $\mapsto$ `.h`-Datei (enthält Deklarationen)
  - Modulimplementierung $\mapsto$ `.c`-Datei (enthält Definitionen)
  - Modulverwendung $\mapsto$ `#include <Modul.h>`
  - `private` Symbole $\mapsto$ als `static` definieren
- Die eigentliche Zusammenfügung erfolgt durch den Linker
  - Auflösung erfolgt ausschließlich über Symbolnamen
    $\leadsto$ **Linken ist nicht typsicher!**
  - Typsicherheit muss beim Übersetzen sichergestellt werden
    $\leadsto$ durch gemeinsame Header-Datei

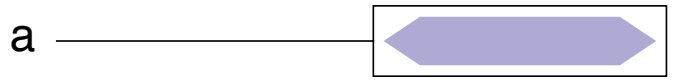
Überblick: Teil C Systemnahe Softwareentwicklung

Einordnung: Zeiger (*Pointer*)

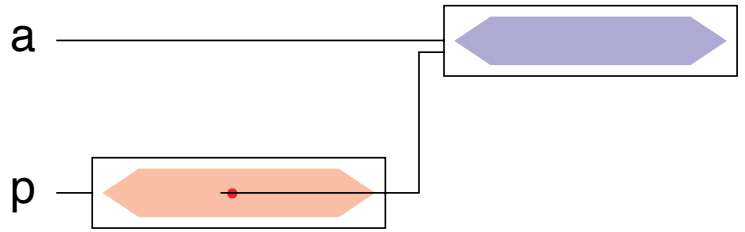
- **Literal:** `'a'`
  Darstellung eines Wertes



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



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



Zeiger (*Pointer*)

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

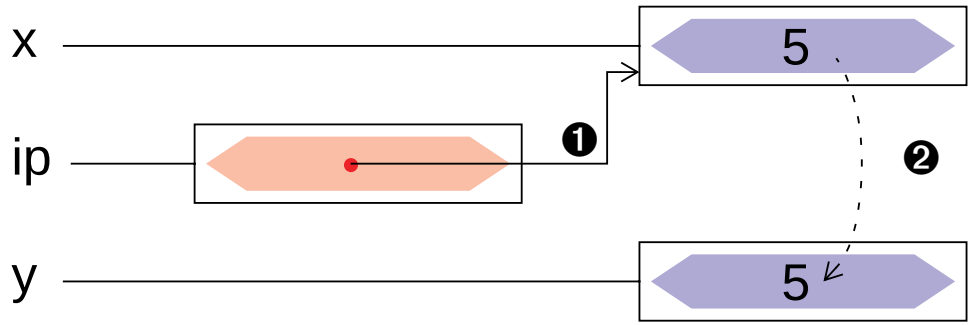
„Effizienz durch Maschinennähe“ ↪ 3–14

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

Definition von Zeigervariablen

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

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



Adress- und Verweisoperatoren

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

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

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

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

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

Zeiger als Funktionsargumente

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

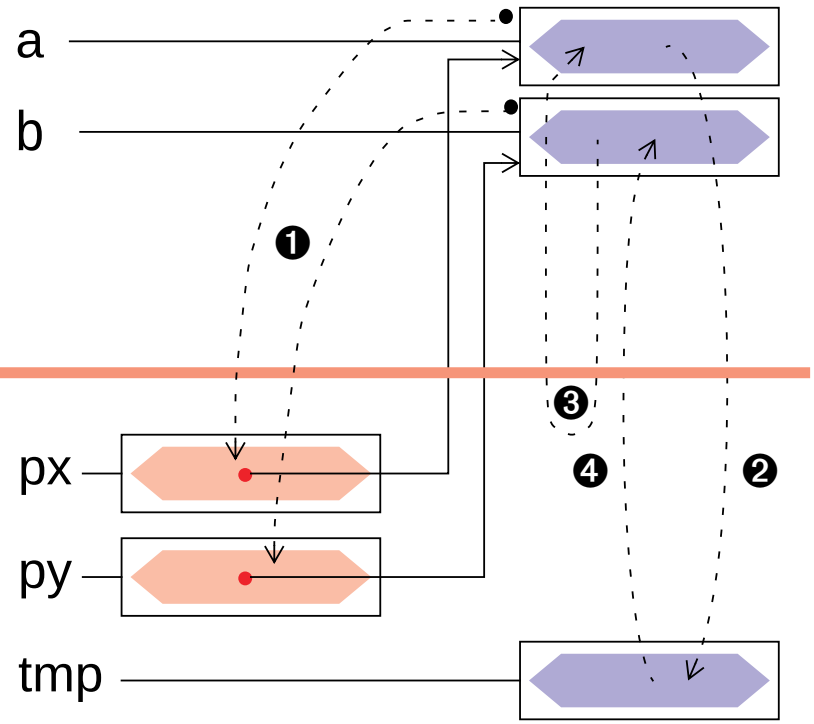
- Das gilt auch für Zeiger (Verweise) [↪ GDI, II-89]
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des *-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

  ⇝ **Call-by-reference**

Zeiger als Funktionsargumente (Forts.)

Beispiel (Gesamtüberblick)

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

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

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

}
```

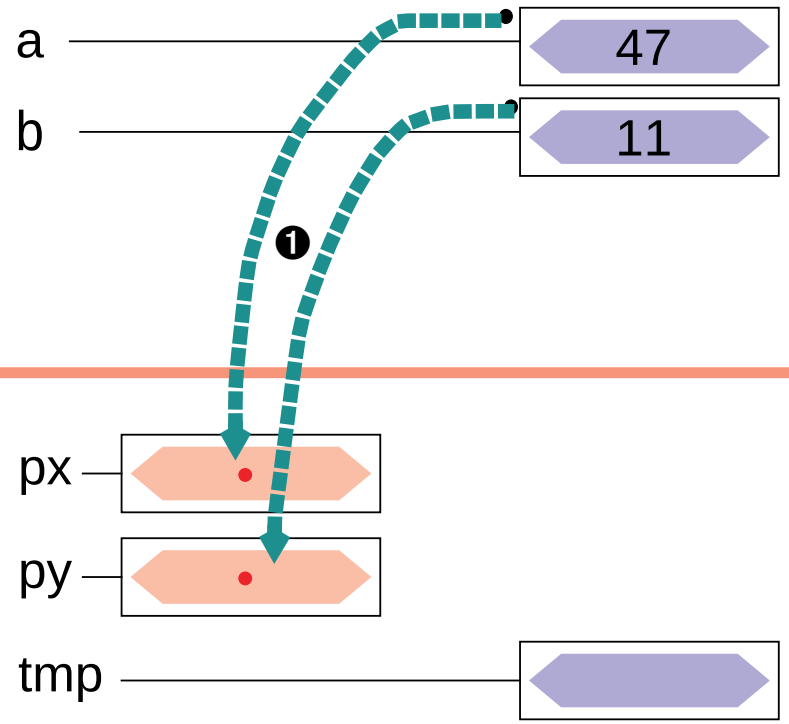


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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

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

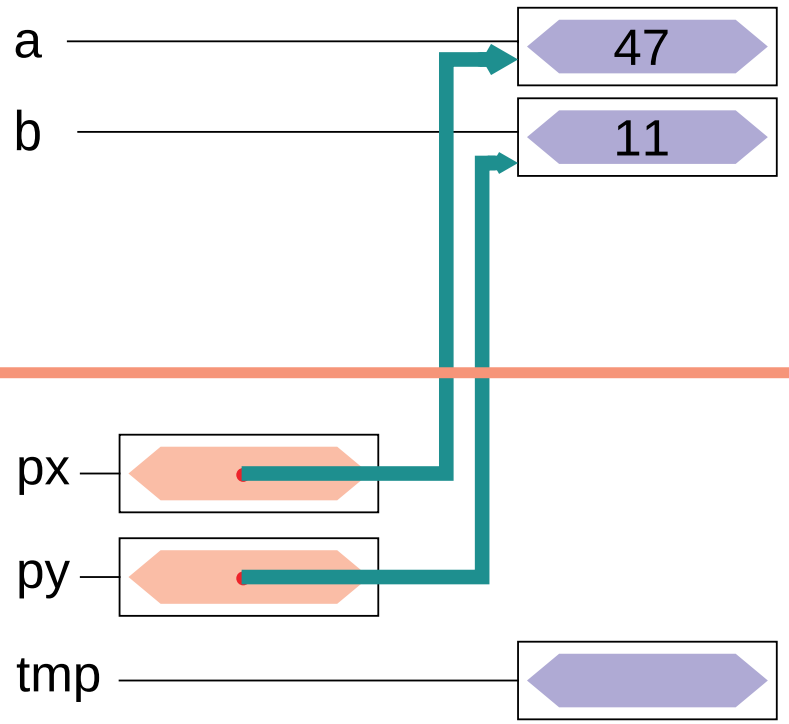


Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

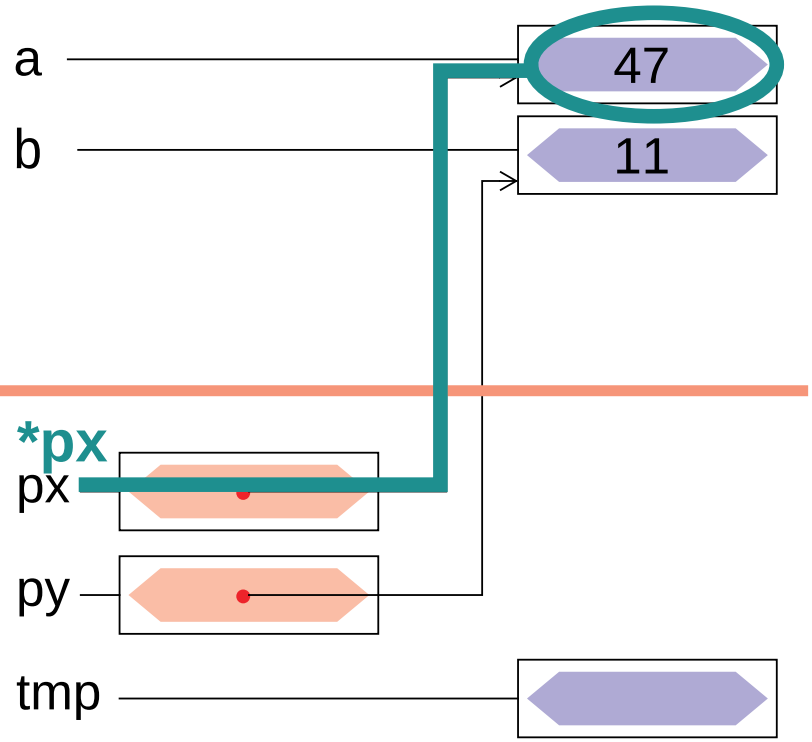


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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

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

    tmp = *px; ❷
```

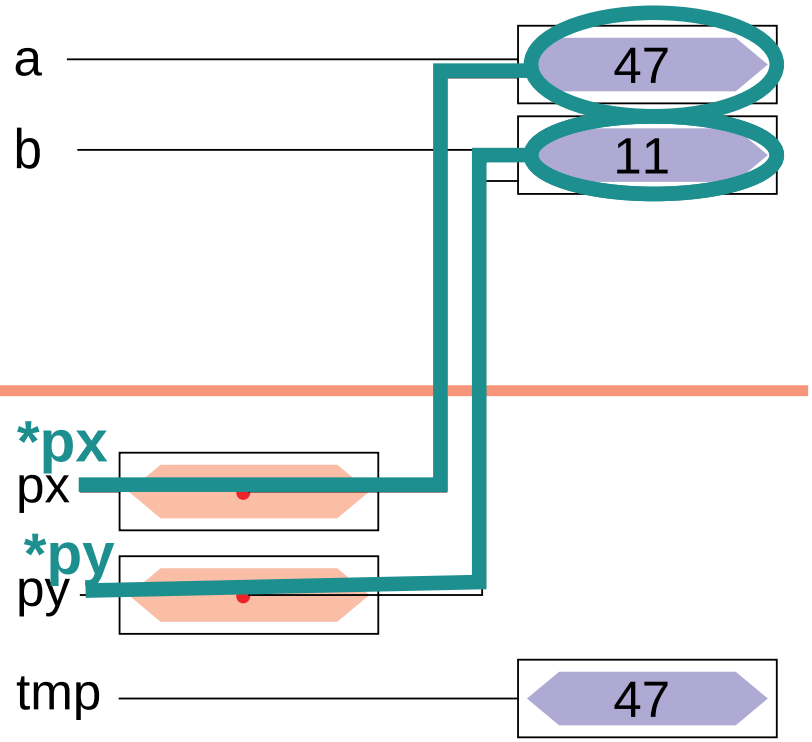


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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

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

    tmp = *px; ❷
    *px = *py; ❸
```

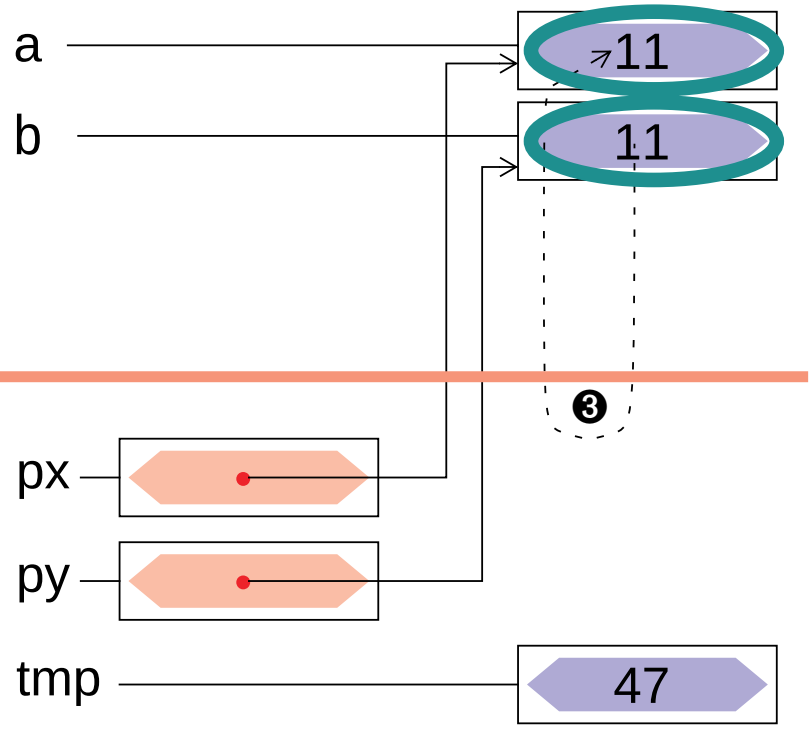


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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

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

    tmp = *px; ❷
    *px = *py; ❸
```

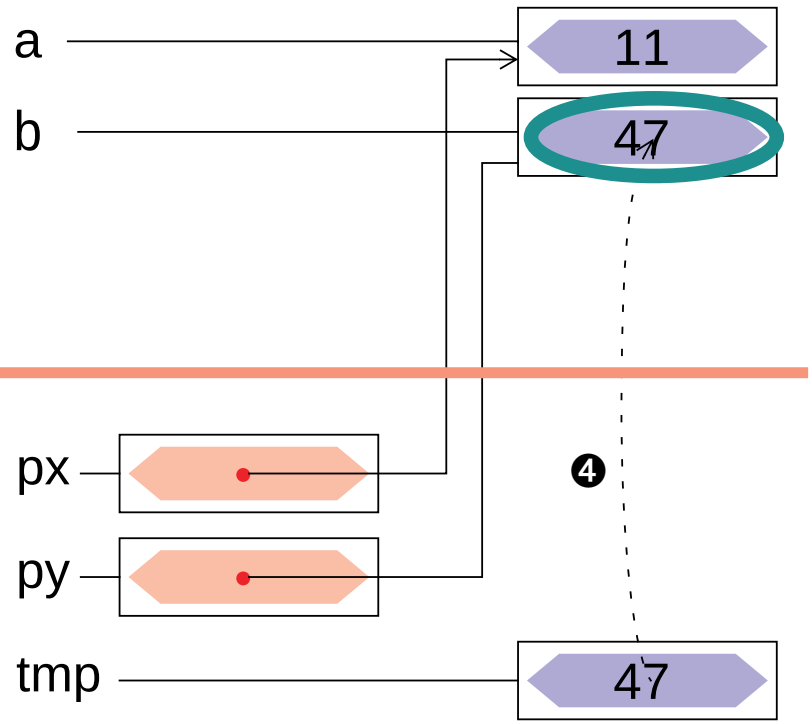


Zeiger als Funktionsargumente (Forts.)

Beispiel (Einzelschritte)

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

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

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

}
```



Einordnung: Felder (*Arrays*) [≈Java]

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

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

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

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


Feldinitialisierung

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

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

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

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

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

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


Feldzugriff

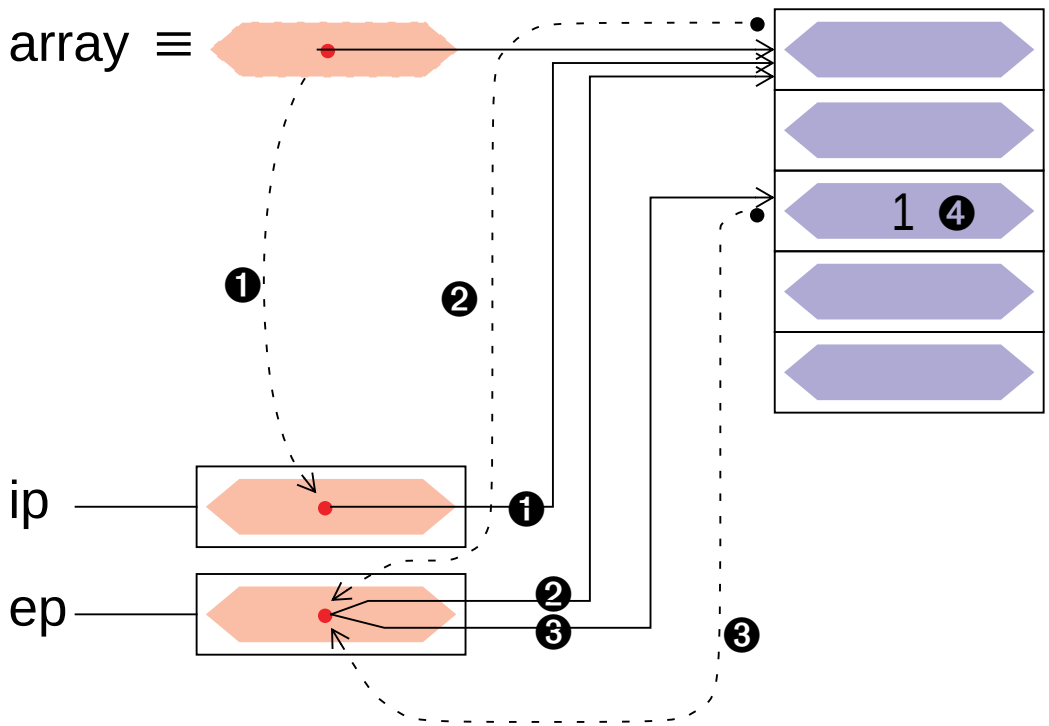
- Syntax: *Feld* **[** *IntAusdruck* **]** [=Java]
  - Wobei $0 \leq$ *IntAusdruck* $< n$ für $n =$ Feldgröße
  - **Achtung:** Feldindex wird nicht überprüft [≠Java]
    $\rightsquigarrow$ häufige Fehlerquelle in C-Programmen
- Beispiel

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

LEDs[ 3 ] = BLUE1;

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

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

Felder sind Zeiger

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

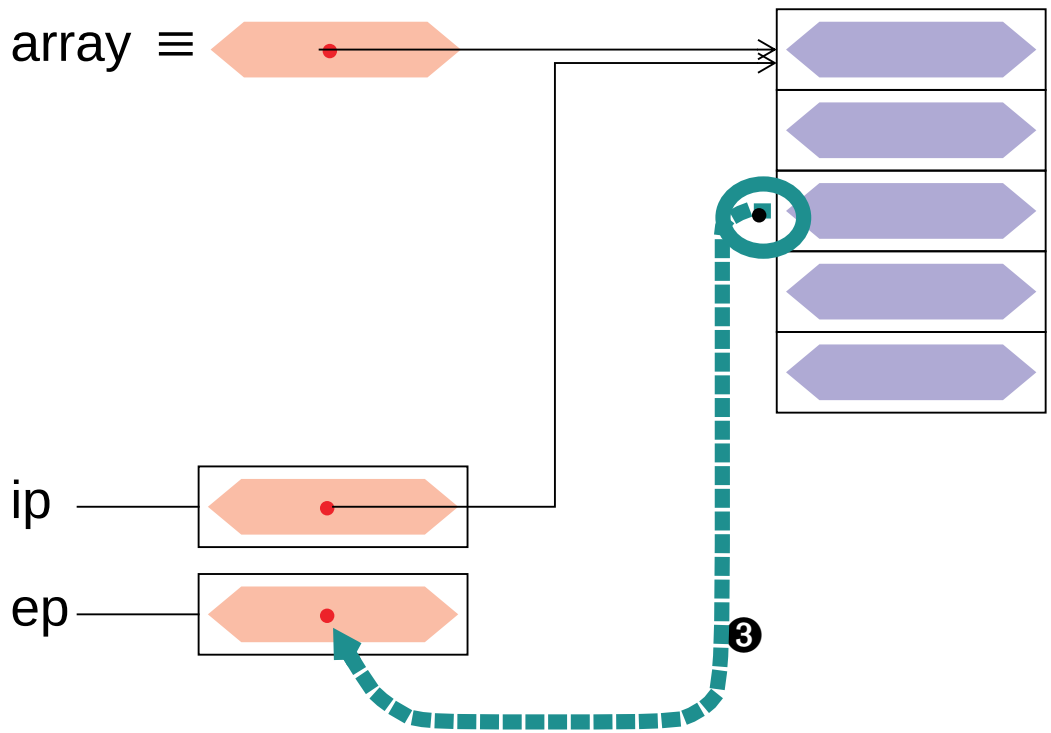
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Felder sind Zeiger

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

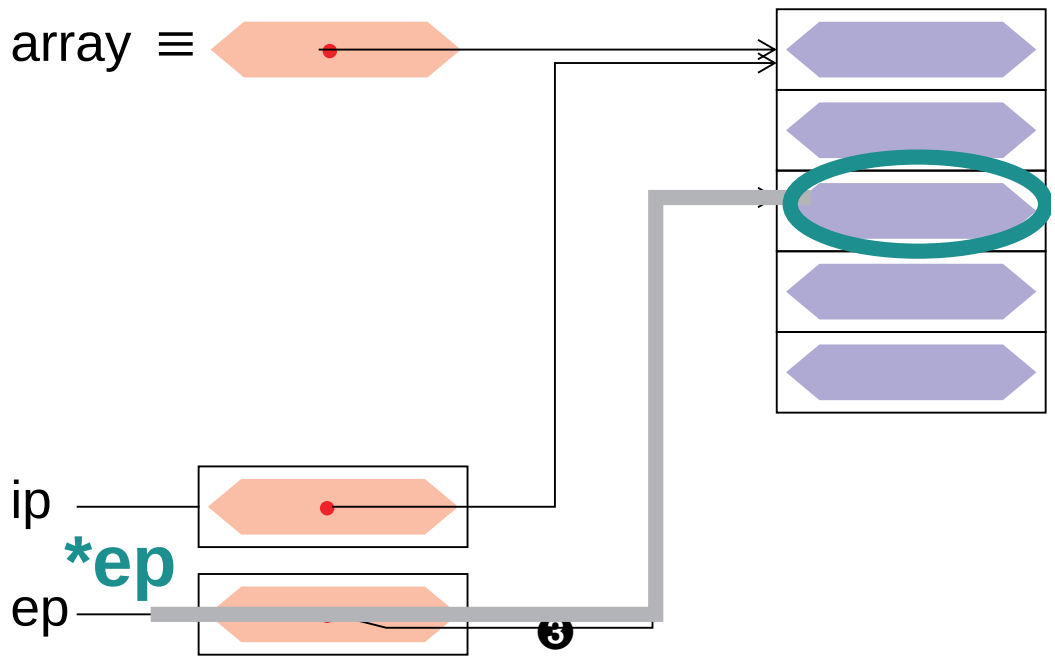
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```



Felder sind Zeiger

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

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Zeiger sind Felder

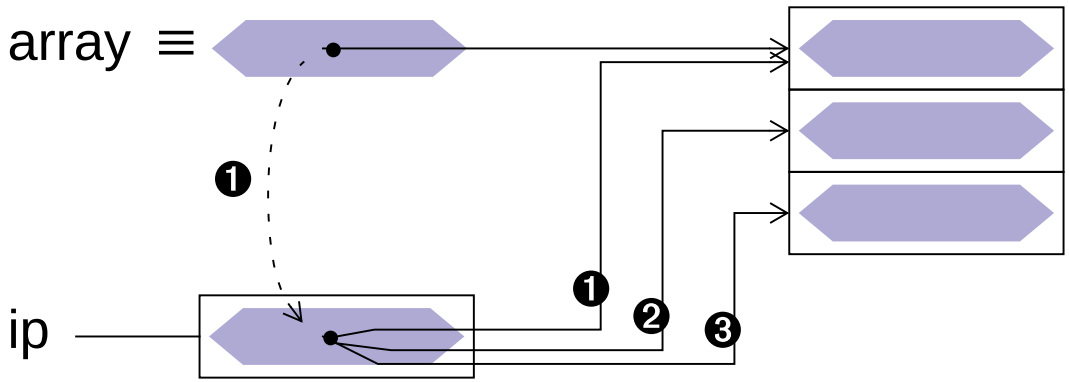
- Ein Feldbezeichner ist syntaktisch äquivalent zu einem konstanten Zeiger auf das erste Element des Feldes: `array` ≡ `&array[0]`
- Diese Beziehung gilt in beide Richtungen: `*array` ≡ `array[0]`
  - Ein Zeiger kann wie ein Feld verwendet werden
  - Insbesondere kann der **[ ]** - Operator angewandt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

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

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

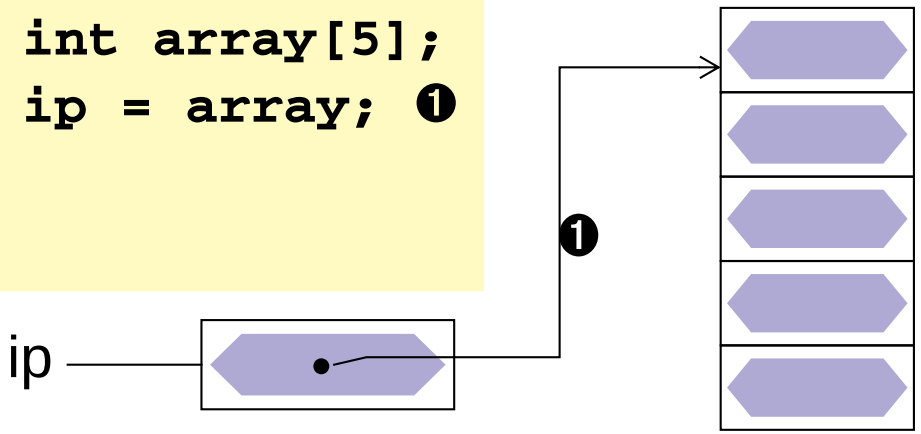
Rechnen mit Zeigern

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

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```




```
int array[5];
ip = array; ❶
```



`(ip+3)` ≡ `&ip[3]`

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

Zeigerarithmetik – Operationen

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

Felder sind Zeiger sind Felder – Zusammenfassung

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

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

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

Fehler: `array` ist konstant!

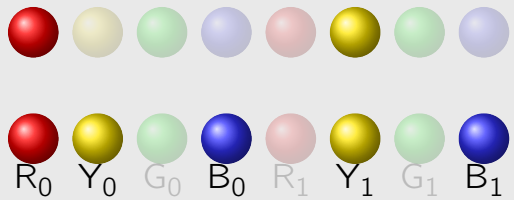
- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.
Der Feldbezeichner kann aber nicht verändert werden.

Felder als Funktionsparameter

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

  ⇝ *Call-by-reference*

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

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

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



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

Felder als Funktionsparameter (Forts.)

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

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

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

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

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

Felder als Funktionsparameter (Forts.)

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

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

Dabei gilt: "hallo" ≡ 

↪ 6–13

- Implementierungsvarianten

Variante 1: Feld-Syntax

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

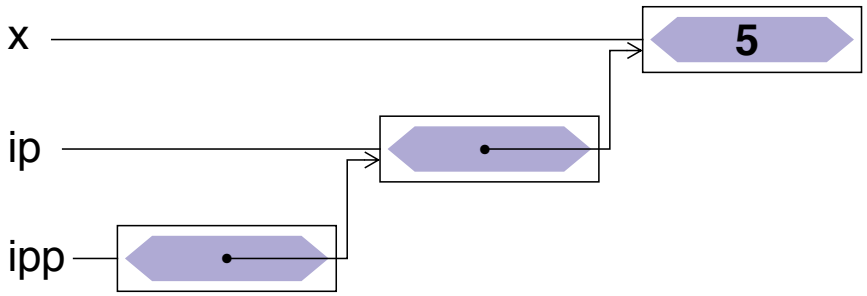
Variante 2: Zeiger-Syntax

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

Zeiger auf Zeiger

- Ein Zeiger kann auch auf eine Zeigervariable verweisen

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

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



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

Zeiger auf Funktionen

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

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

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

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


Zeiger auf Funktionen (Forts.)

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

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

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

Zeiger auf Funktionen (Forts.)

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

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

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

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

Zusammenfassung

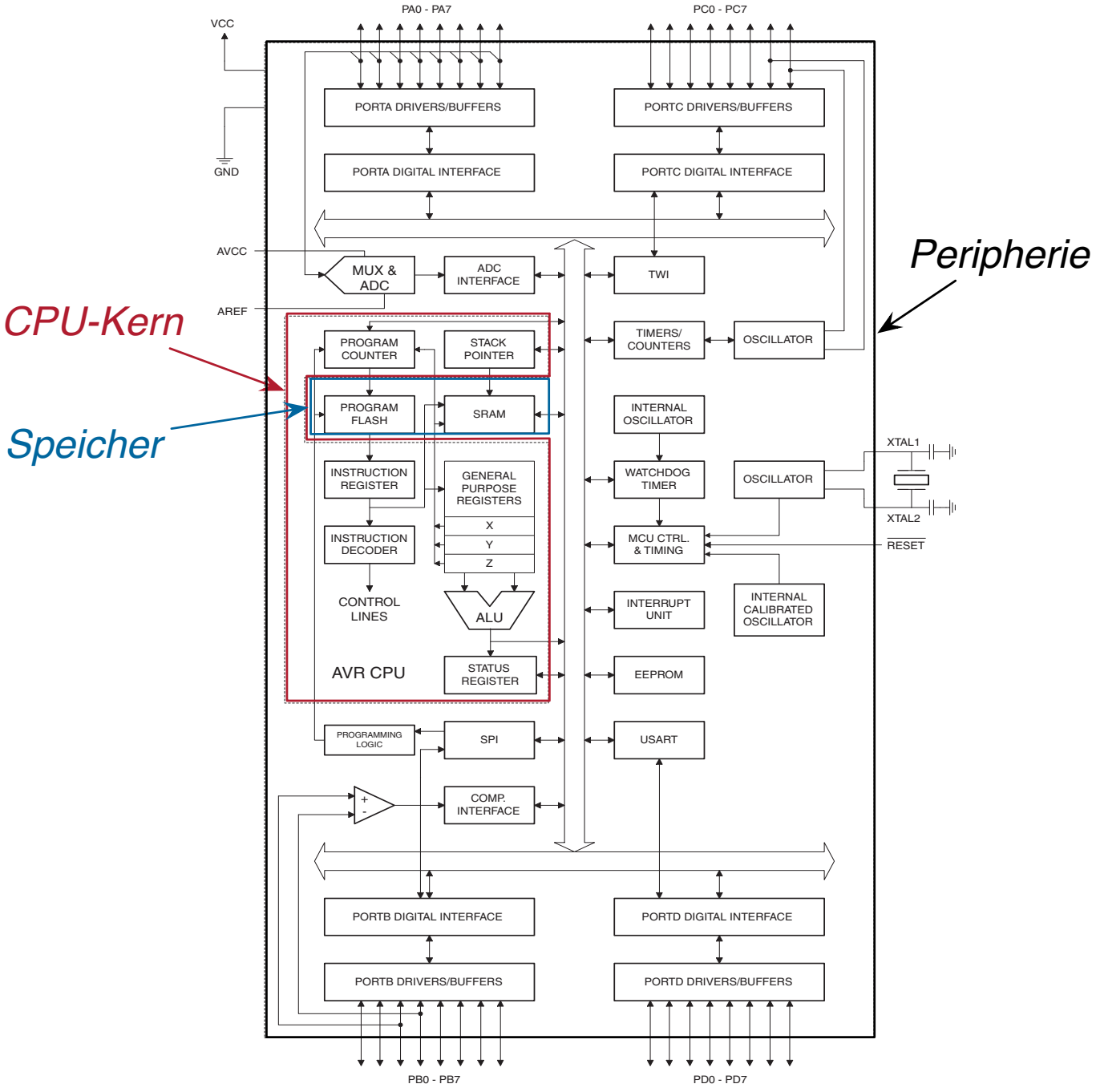
- Ein Zeiger verweist auf eine Variable im Speicher
  - Möglichkeit des indirekten Zugriffs auf den Wert
  - Grundlage für die Implementierung von *call-by-reference* in C
  - Grundlage für die Implementierung von Feldern
  - Wichtiges Element der Maschinennähe von C
  - **Häufigste Fehlerursache in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

Überblick: Teil C Systemnahe Softwareentwicklung

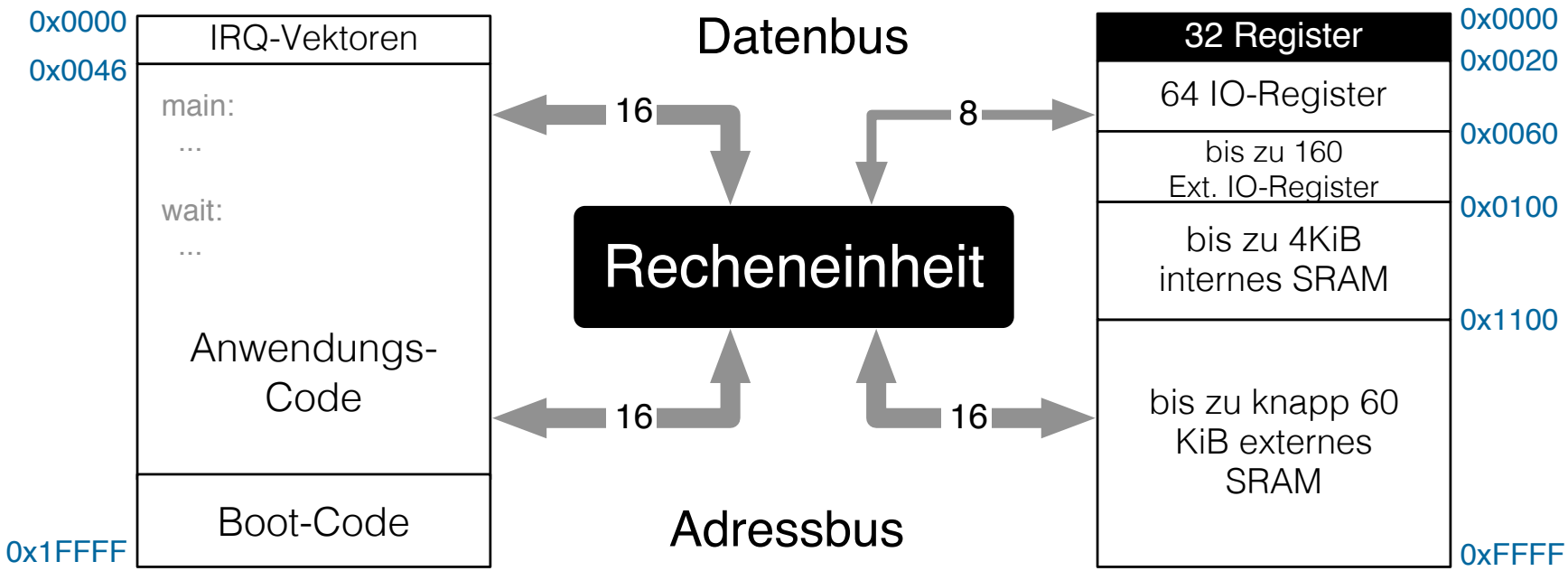
Was ist ein μ-Controller?

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

Beispiel ATmega32: Blockschaltbild



Beispiel ATmega-Familie: CPU-Architektur



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

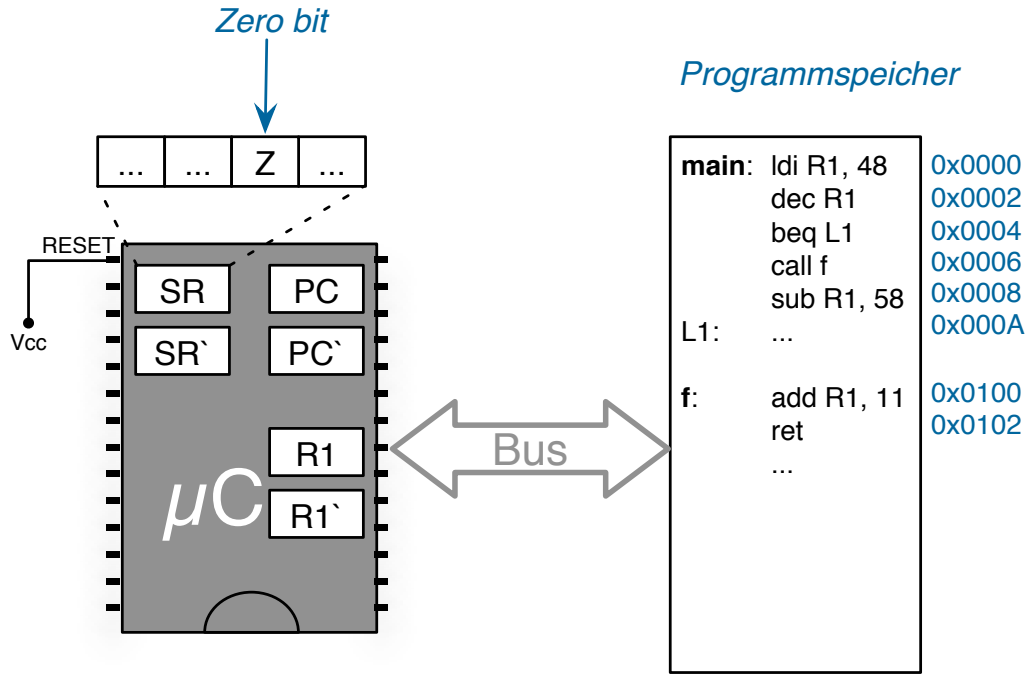
Datenadressraum (SRAM)
0–64 KiB, 1-Byte-Elemente

- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingeblendet

  ⇝ ansprechbar wie globale Variablen

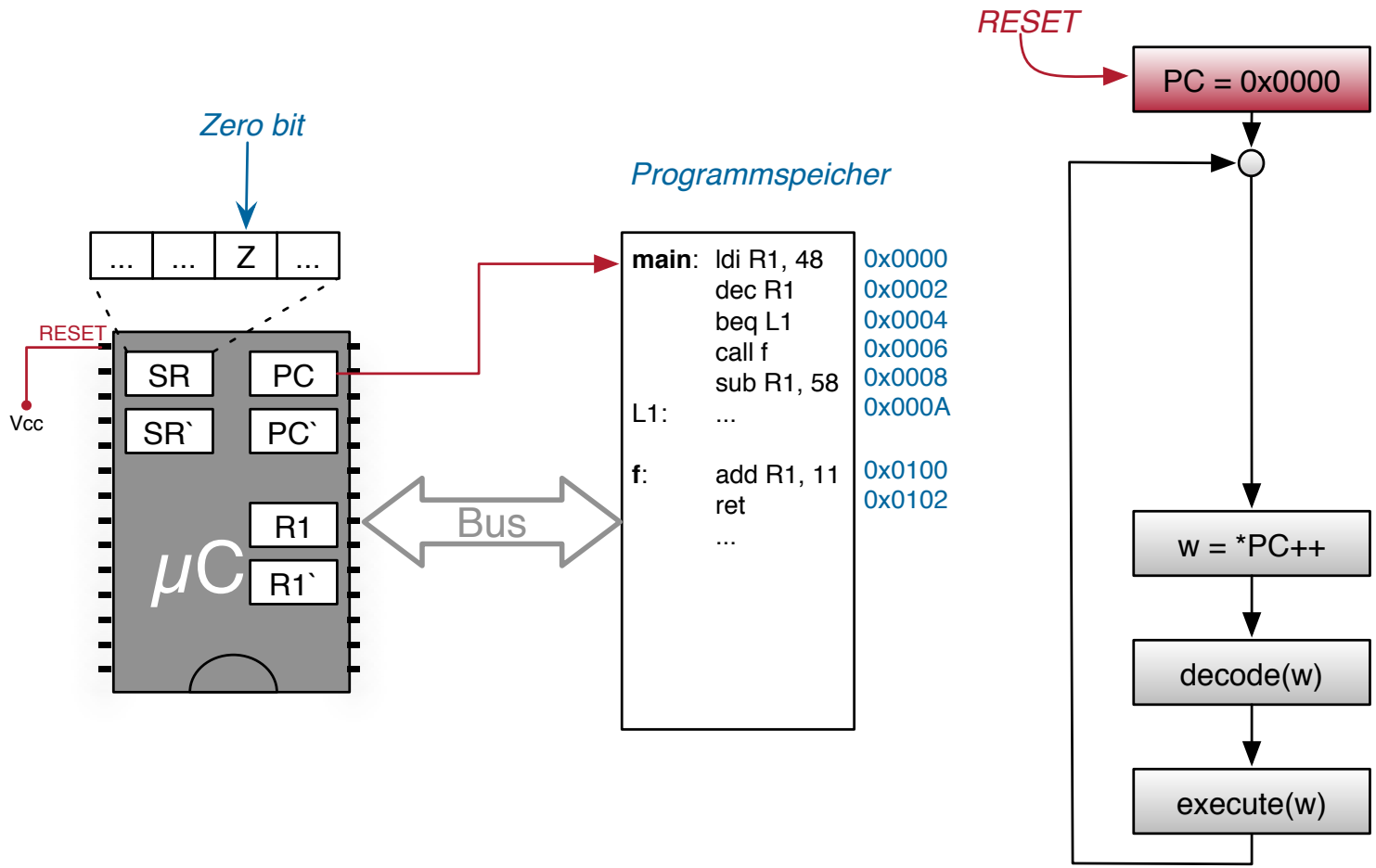
Zum Vergleich: PC basiert auf von-Neumann-Architektur [↪ GDI, VI-6] mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

Wie arbeitet ein Prozessor?

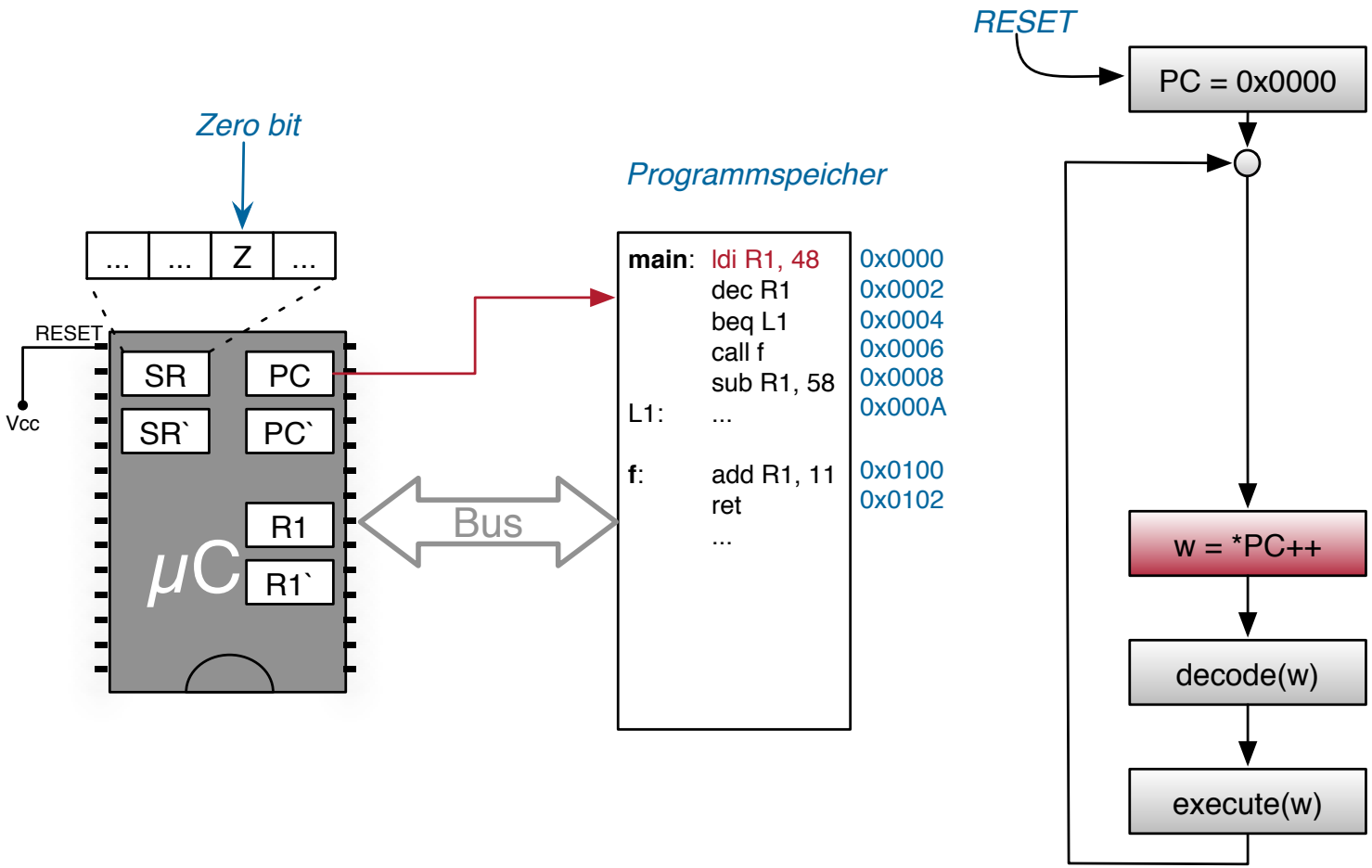


- Hier am Beispiel eines sehr einfachen Pseudoprozessors
  - Nur zwei Vielzweckregister (R1 und R2)
  - Programmzähler (PC) und Statusregister (SR) (+ „Schattenkopien“)
  - Kein Datenspeicher, kein Stapel $\rightsquigarrow$ Programm arbeitet nur auf Registern

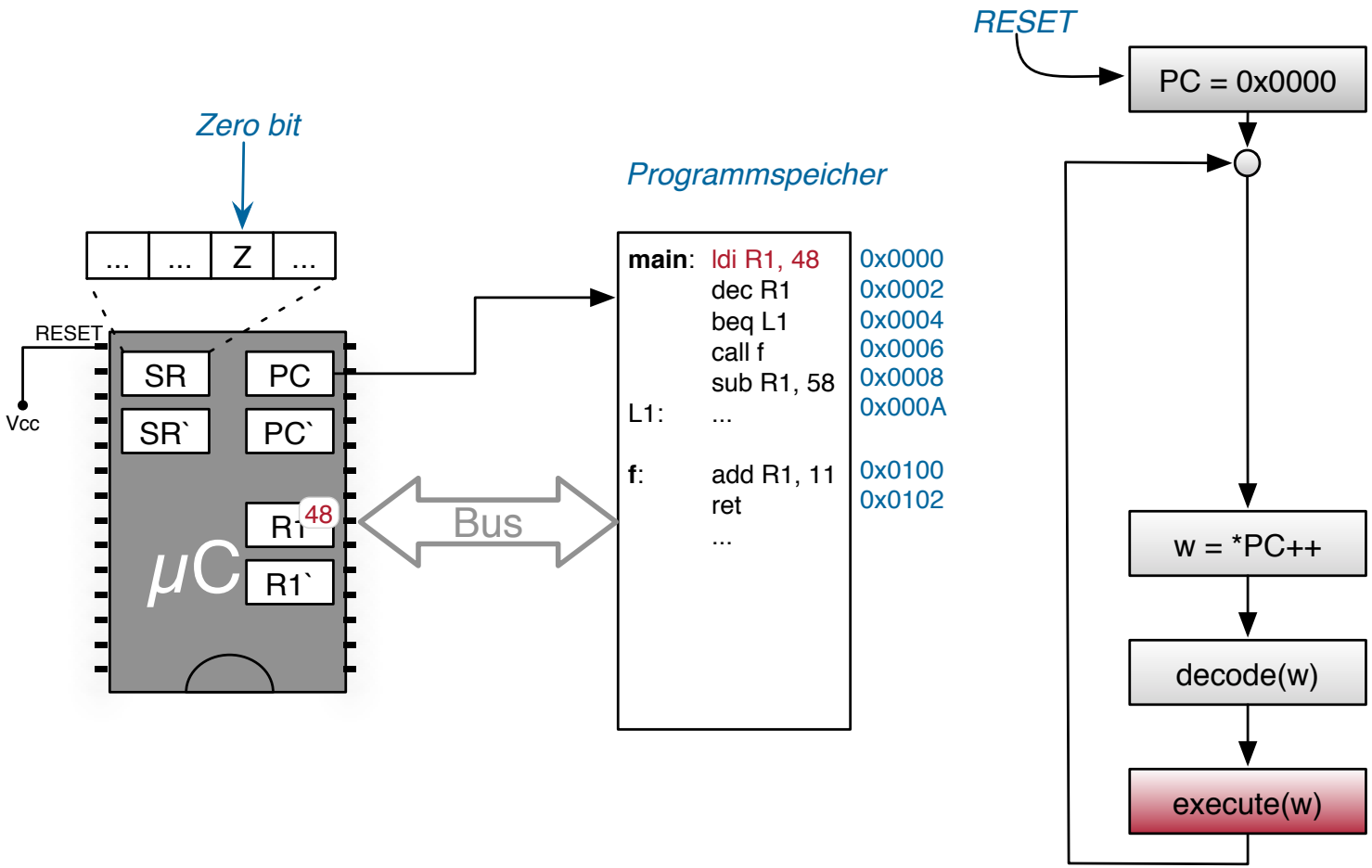
Wie arbeitet ein Prozessor?



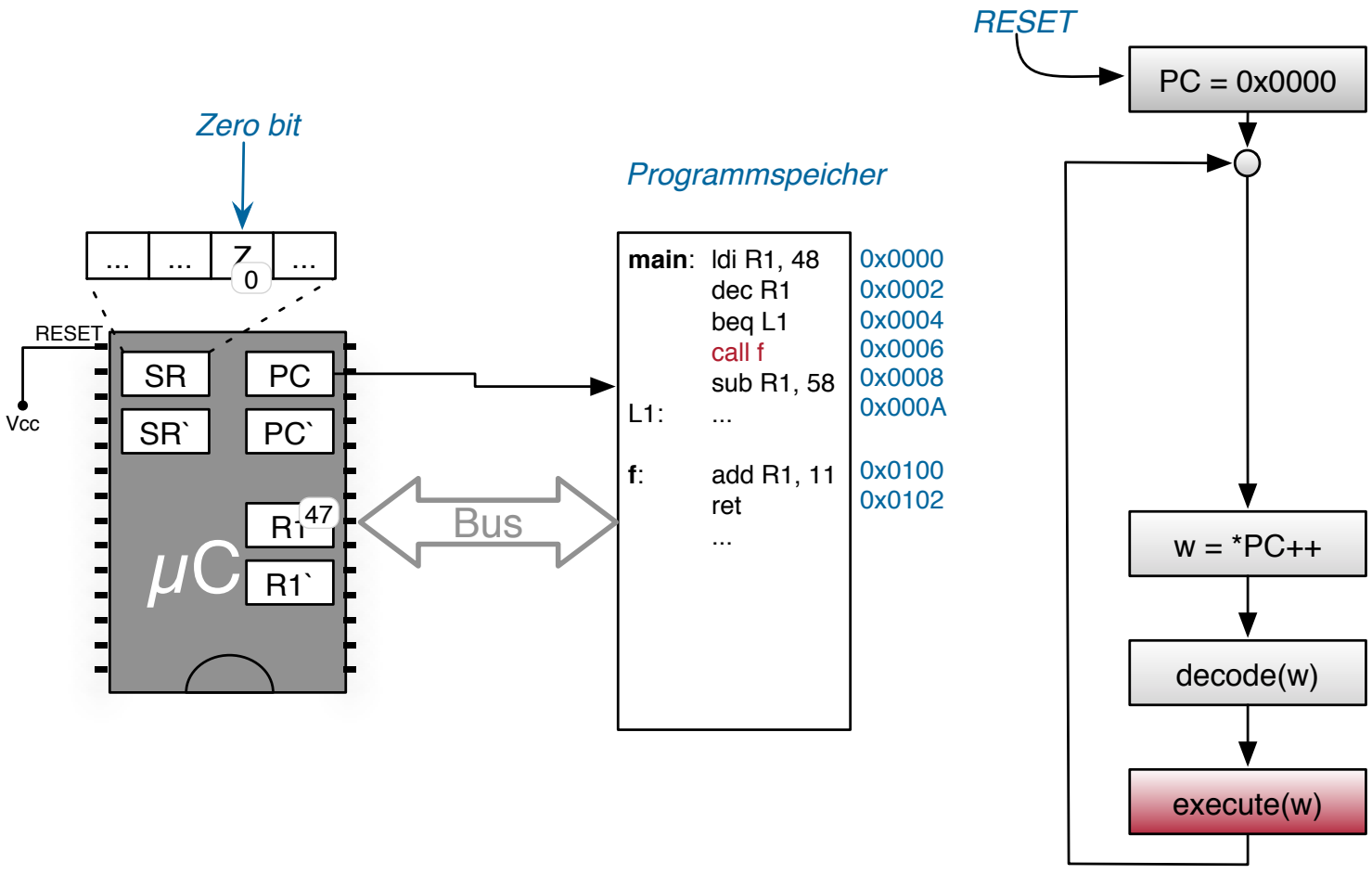
Wie arbeitet ein Prozessor?



Wie arbeitet ein Prozessor?



Wie arbeitet ein Prozessor?

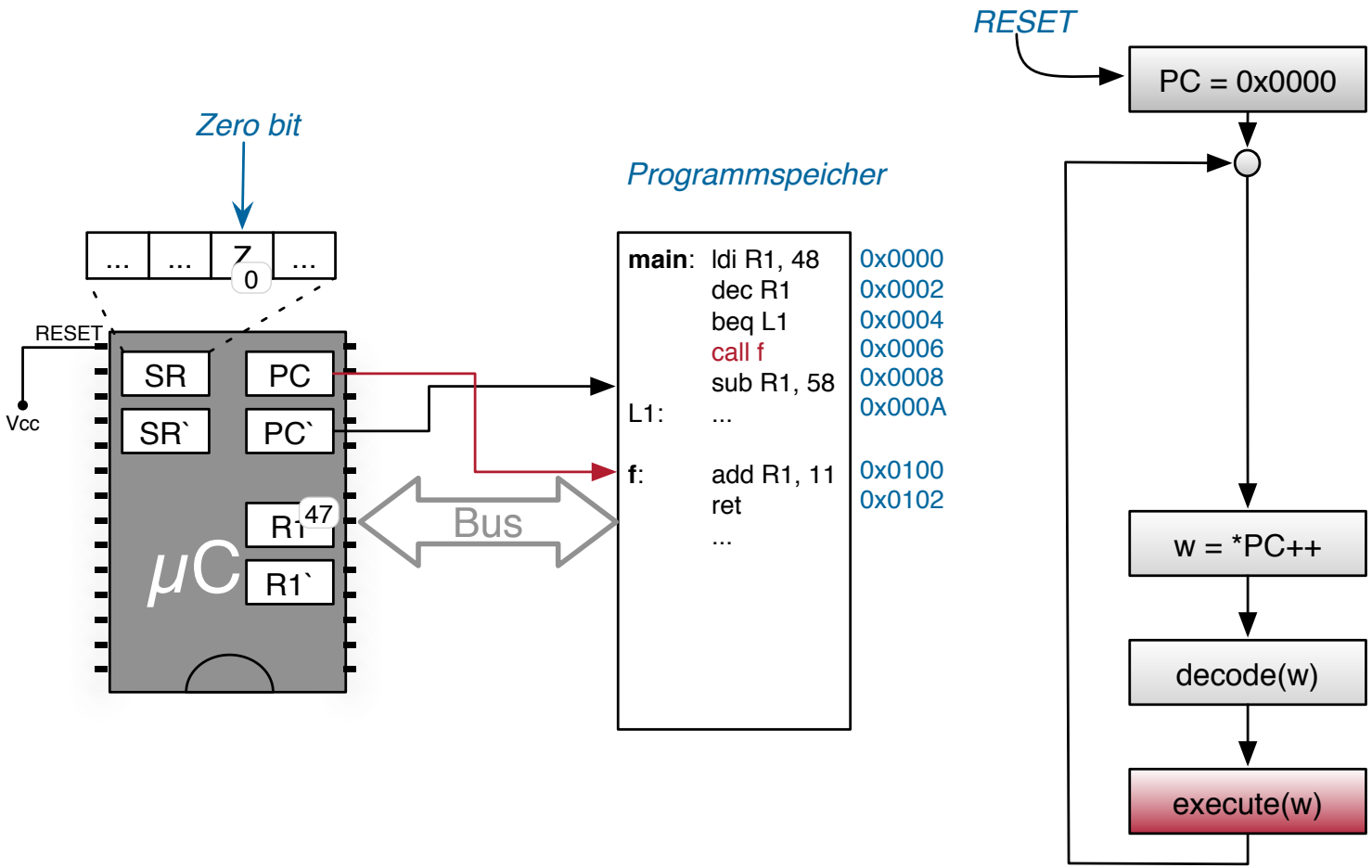


w: **dec** <R>
R –= 1
if(R == 0) Z = 1
else Z = 0

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

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

Wie arbeitet ein Prozessor?

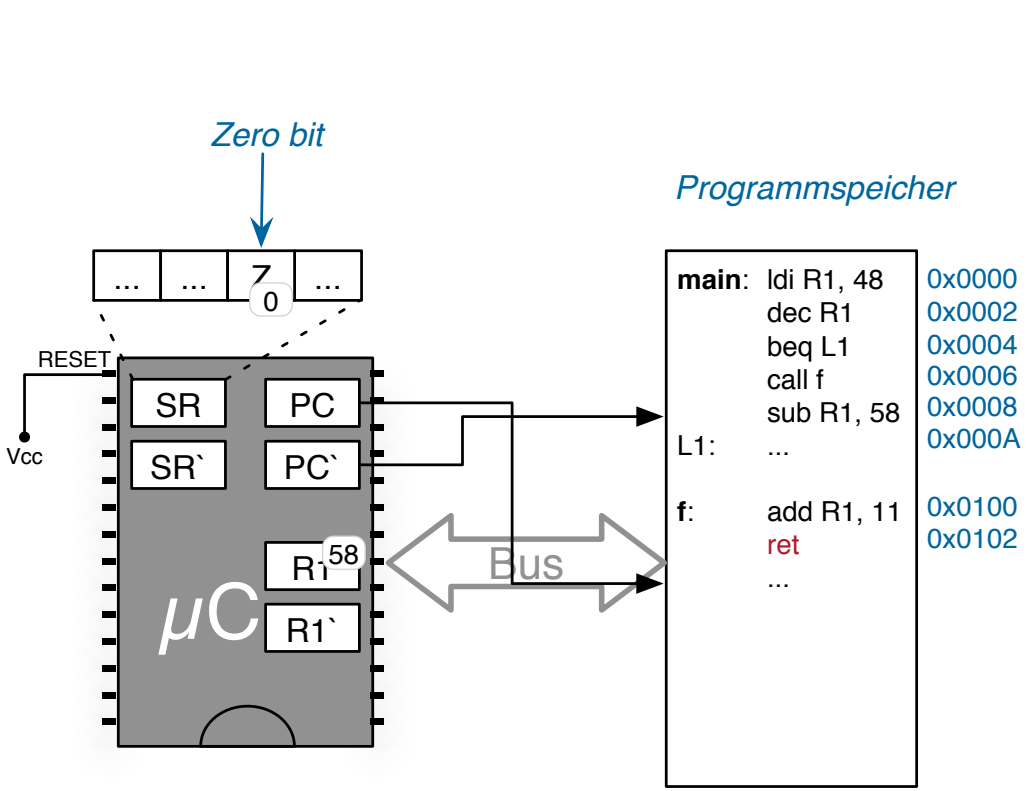


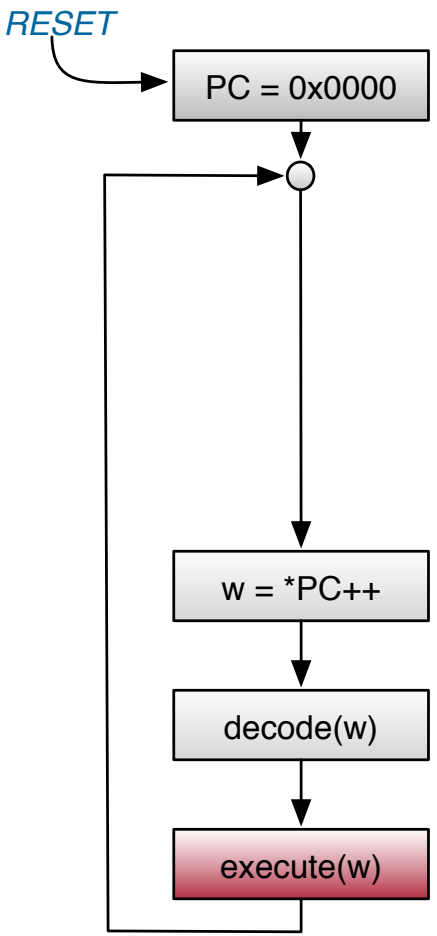
w: **dec** <R>
R –= 1
if(R == 0) Z = 1
else Z = 0

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

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

Wie arbeitet ein Prozessor?





w: **dec** <R>	w: **beq** <lab>	w: **call** <func>	w: **ret**
R –= 1 if(R == 0) Z = 1 else Z = 0	if (Z) PC = lab	PC` = PC PC = func	PC = PC`

Peripheriegeräte

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

Peripheriegeräte: Beispiele

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

Peripheriegeräte – Register

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

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

[1, S. 334]

Peripheriegeräte – Register (Forts.)

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

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

`0x12`: Adresse: int

`(volatile uint8_t*)(0x12)`: Adresse: volatile uint8_t* (Cast ↪ 7–17)

`*(volatile uint8_t*)(0x12)`: Wert: volatile uint8_t (Dereferenzierung ↪ 13–4)

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

- Beispiel

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

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

Registerzugriff und Nebenläufigkeit

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

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

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

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

```
// Resulting assembly code

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

  ...
L2:
  ret
```

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

Der `volatile`-Typmodifizierer

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

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

    // button0 pressed
    ...
  }

  if( !(PIND & 0x4) ) {

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

```
// Resulting assembly code


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

  ...
L2:
  ret
```

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

Der `volatile`-Typmodifizierer (Forts.)

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

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

volatile!

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

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

Die Verwendung von `volatile` verursacht erhebliche Kosten

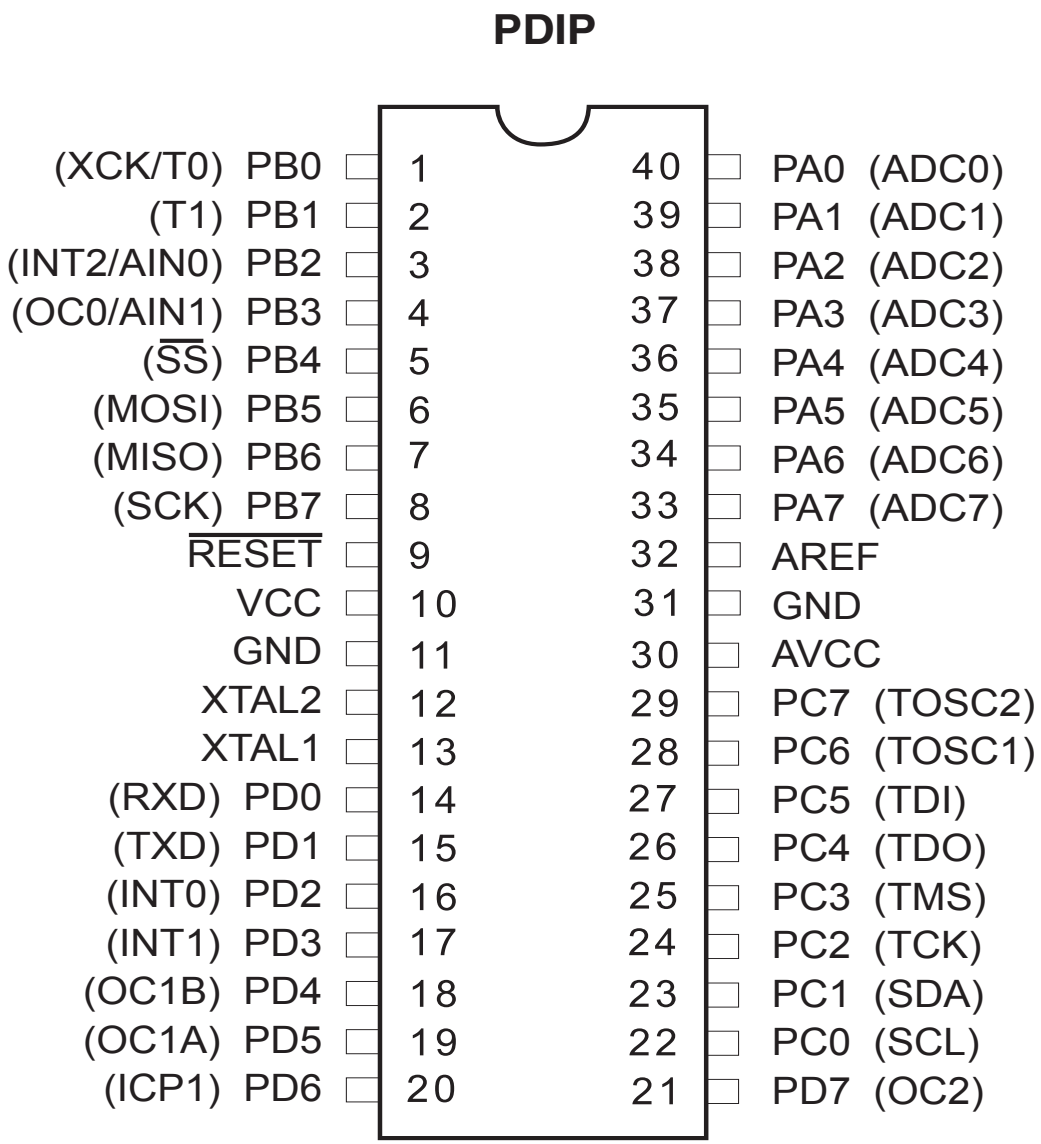
- Werte können nicht mehr in Registern gehalten werden
- Viele Code-Optimierungen können nicht durchgeführt werden

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

Peripheriegeräte: Ports

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

Beispiel ATmega32: Port/Pin-Belegung



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

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

PORTA steht daher nicht zur Verfügung.

Beispiel ATmega32: Port-Register

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

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

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

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

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

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

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

Verwendungsbeispiele: ↪ 3–5 und ↪ 3–8 [1, S. 66]

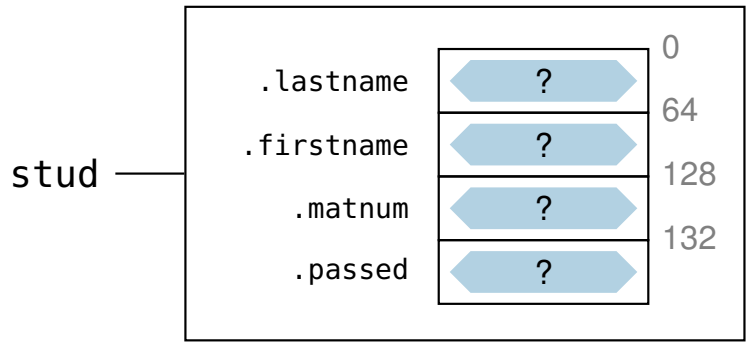
Strukturen: Motivation

- Jeder Port wird durch *drei* globale Variablen verwaltet
  - Es wäre besser diese zusammen zu fassen
  - „problembezogene Abstraktionen“ ↪ 4–1
  - „Trennung der Belange“ ↪ 12–4
- Dies geht in C mit Verbundtypen (Strukturen)

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

// Variable definition
struct Student stud;
```

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

Die Datenelemente werden hintereinander im Speicher abgelegt.



Strukturen: Variablendefinition und -initialisierung

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

↪ 13–8

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

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

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

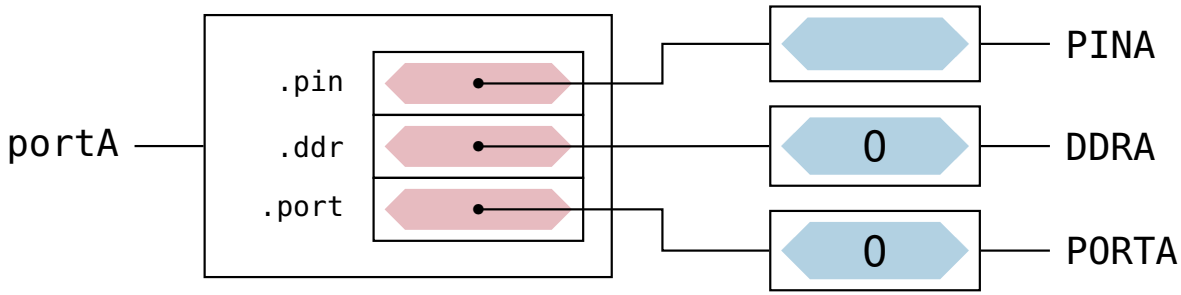
- Analog zur Definition von `enum`-Typen kann man mit `typedef` die Verwendung vereinfachen

↪ 6–8

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

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


Strukturen: Elementzugriff

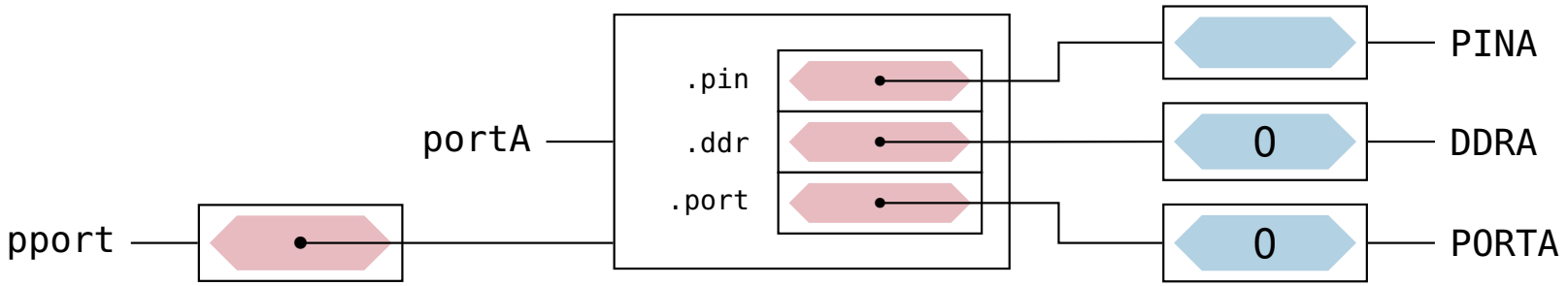


- Auf Strukturelemente wird mit dem `.`-Operator zugegriffen [≈Java]

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

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

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

Strukturen: Elementzugriff



- Bei einem Zeiger auf eine Struktur würde Klammerung benötigt

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

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

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

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

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

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

Strukturen als Funktionsparameter

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

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

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

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

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

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

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

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


Bit-Strukturen: Bitfelder

- Strukturelemente können auf Bit-Granularität festgelegt werden
  - Der Compiler fasst Bitfelder zu passenden Ganzzahltypen zusammen
  - Nützlich, um auf einzelne Bit-Bereiche eines Registers zuzugreifen
- Beispiel
  - `MCUCR` **MCU Control Register:** Steuert Power-Management-Funktionen und Auslöser für externe Interrupt-Quellen `INT0` und `INT1`. [1, S. 36+69]

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

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

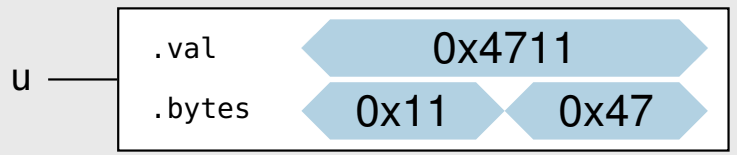

Unions

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

↪ 14–15

- Beispiel

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

  u.val = 0x4711;

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



Unions und Bit-Strukturen: Anwendungsbeispiel

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

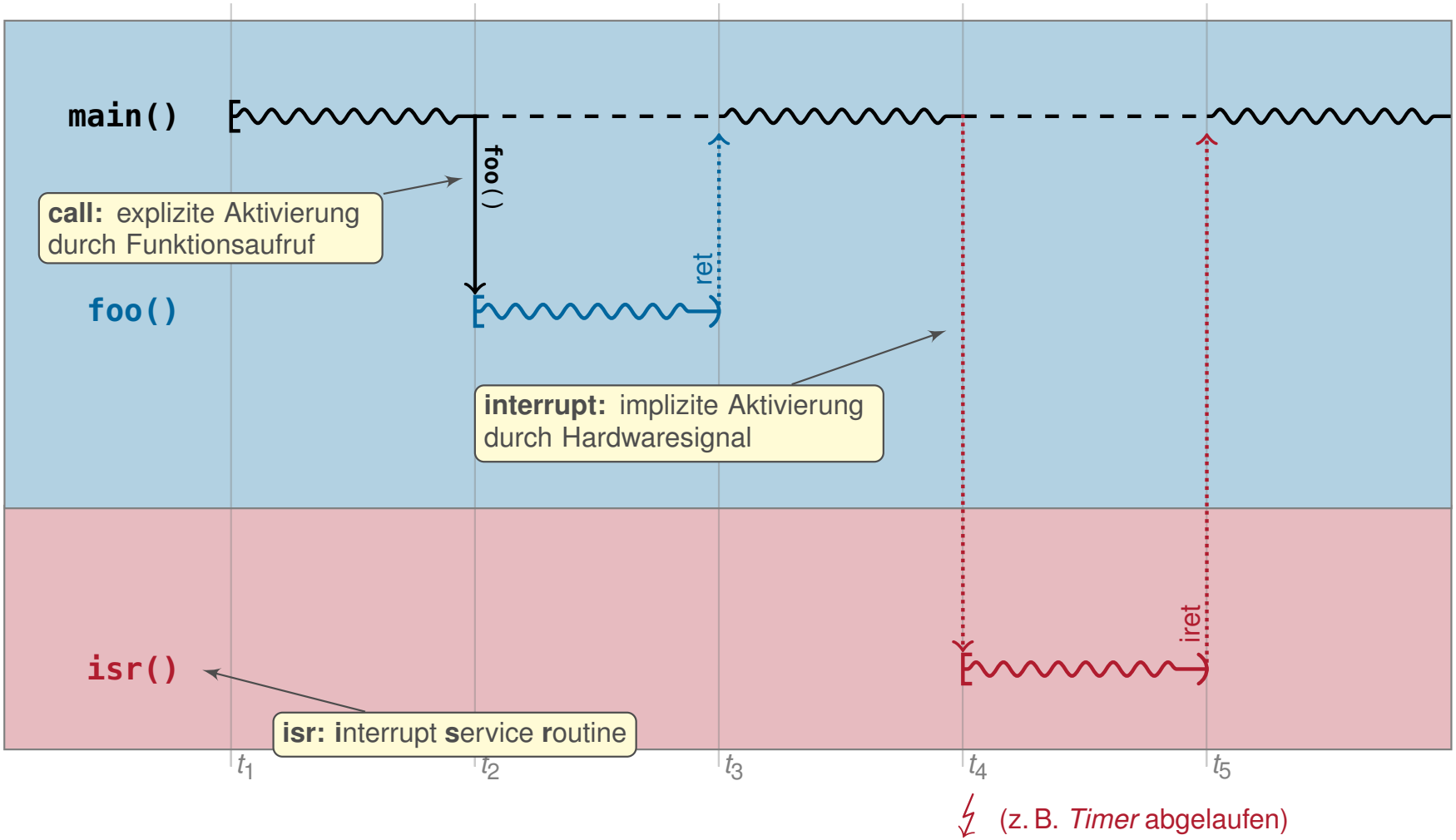
```
typedef union {
  volatile uint8_t  reg;  // complete register
  volatile struct {
    uint8_t ISC0  : 2;    // components
    uint8_t ISC1  : 2;
    uint8_t SM    : 3;
    uint8_t SE    : 1;
  };
} MCUCR_t;

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


Überblick: Teil C Systemnahe Softwareentwicklung

Ereignisbehandlung

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

Interrupt $\mapsto$ Funktionsaufruf „von außen“



Polling vs. Interrupts – Vor- und Nachteile

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

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

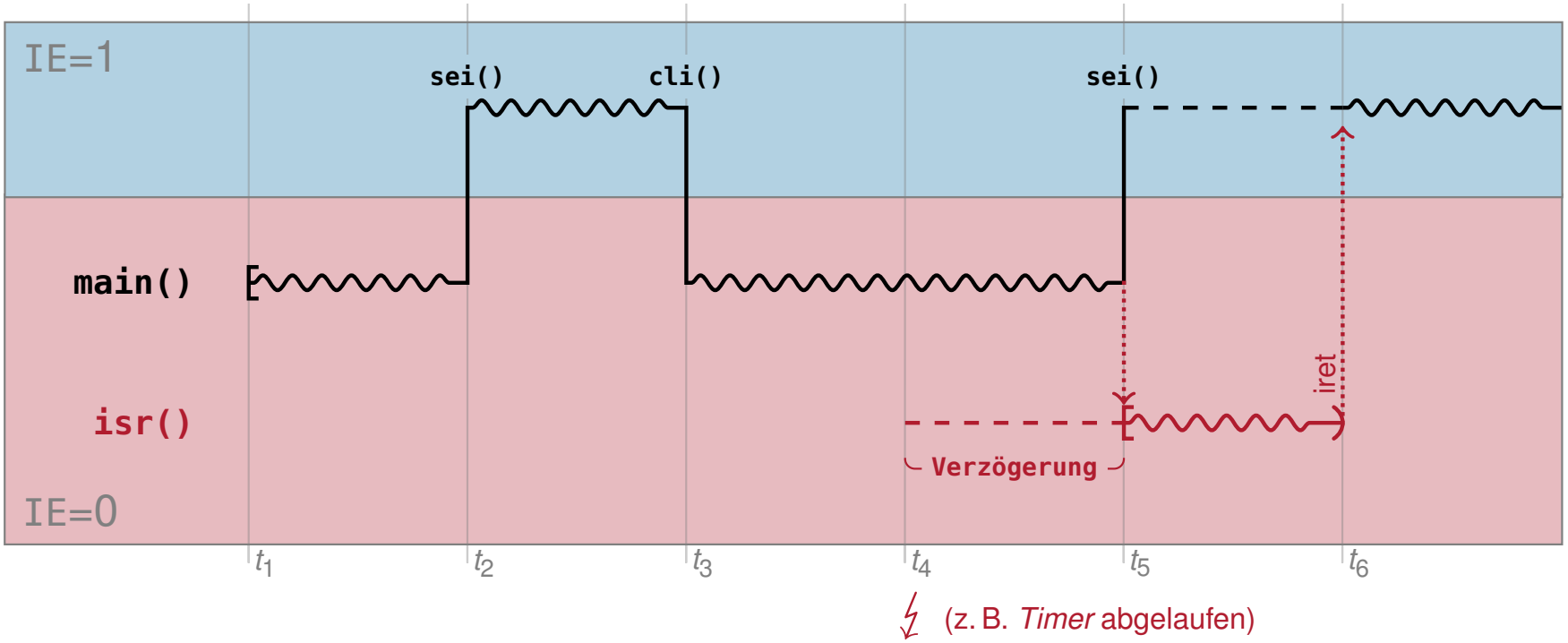
Interruptsperren

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

IRQ $\mapsto$ *Interrupt ReQuest*

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

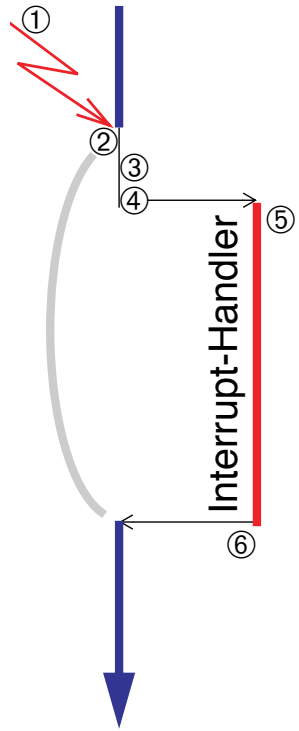
Interruptsperren: Beispiel



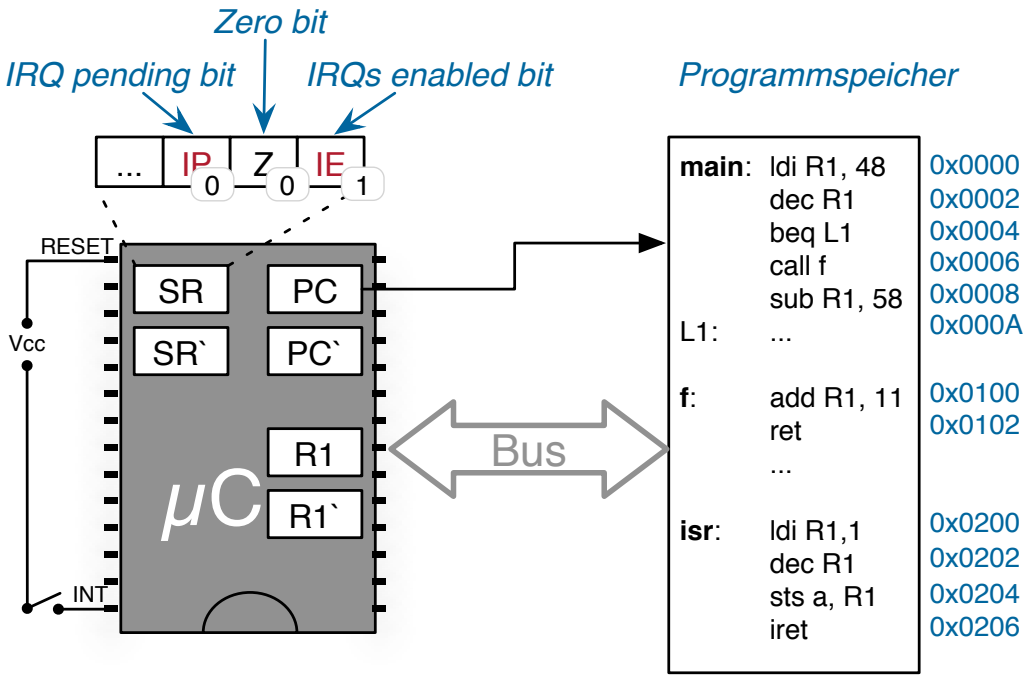
- t_1 Zu Beginn von `main()` sind IRQs gesperrt (`IE`=0)
- t_2, t_3 Mit `sei()` / `cli()` werden IRQs freigegeben (`IE`=1) / erneut gesperrt
- t_4 ↯ aber `IE`=0 ⇝ Bearbeitung ist unterdrückt, IRQ wird gepuffert
- t_5 `main()` gibt IRQs frei (`IE`=1) ⇝ gepufferter IRQ „schlägt durch“
- t_5–t_6 Während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

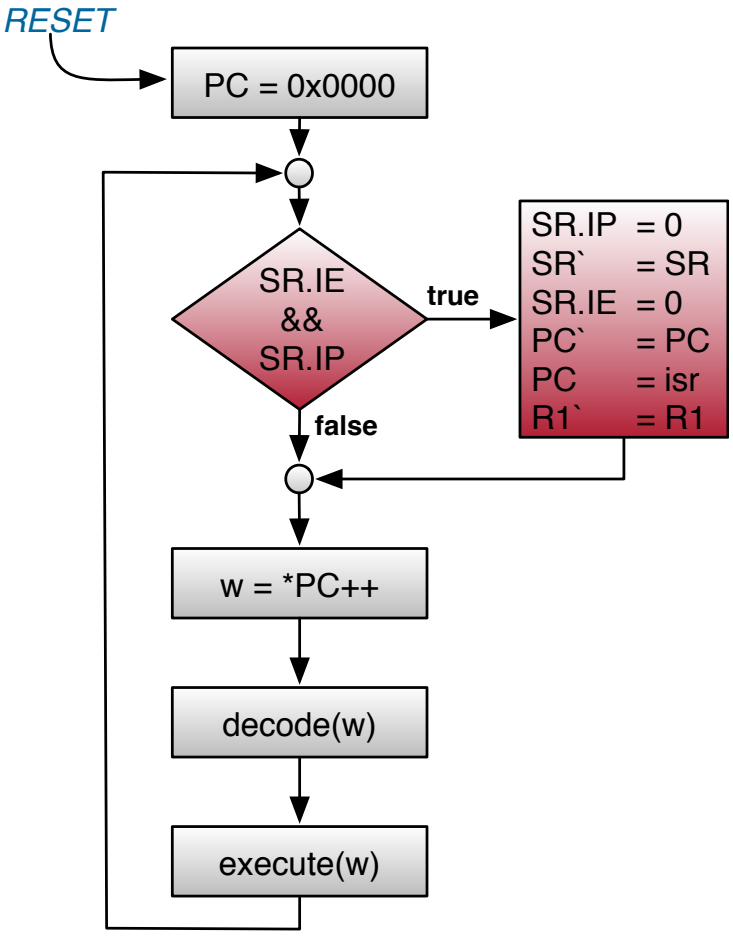
Ablauf eines Interrupts – Überblick

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



Ablauf eines Interrupts – Details





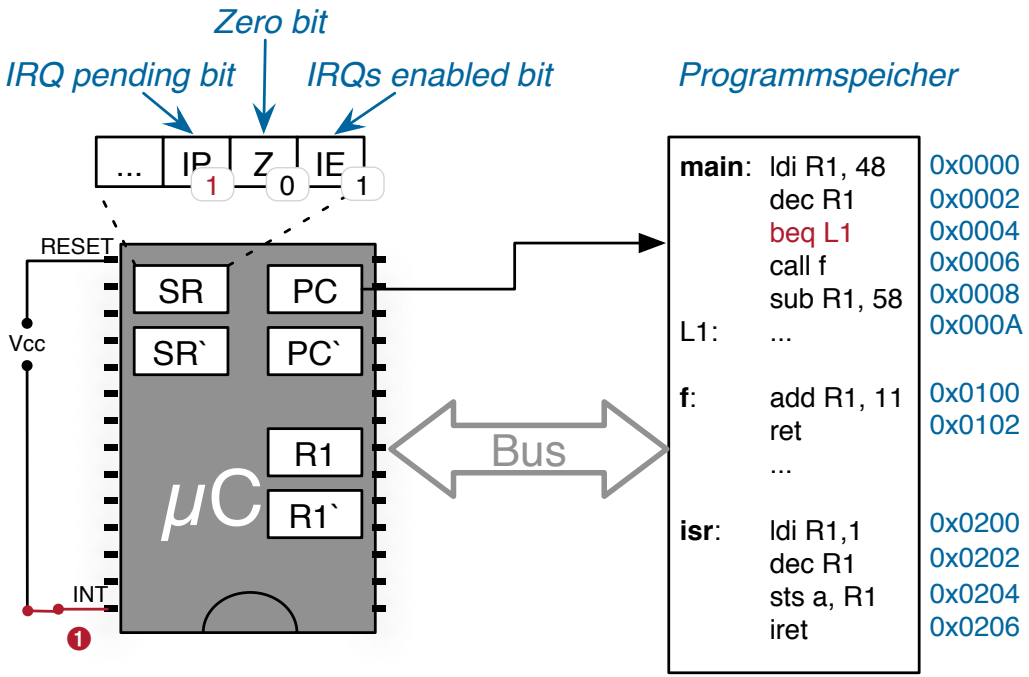
- Hier als Erweiterung unseres einfachen Pseudoprozessors ↪ 14–4
  - Nur eine Interruptquelle
  - Sämtliche Register werden von der Hardware gerettet

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

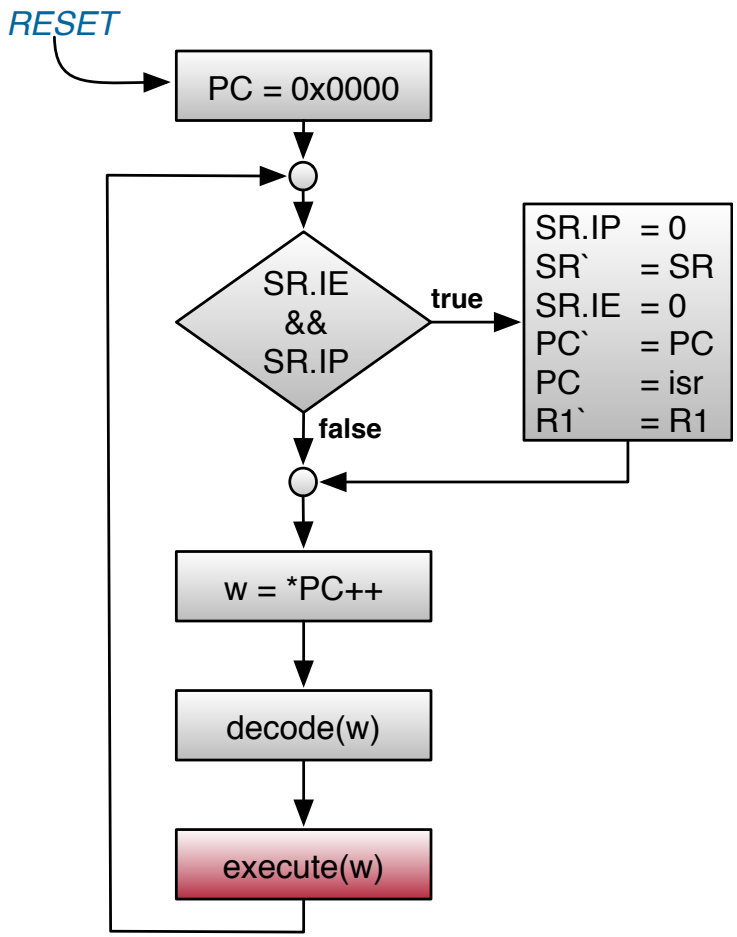
w: **ret**
PC = PC`

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

Ablauf eines Interrupts – Details



❶ Gerät signalisiert Interrupt (aktueller Befehl wird noch fertiggestellt)

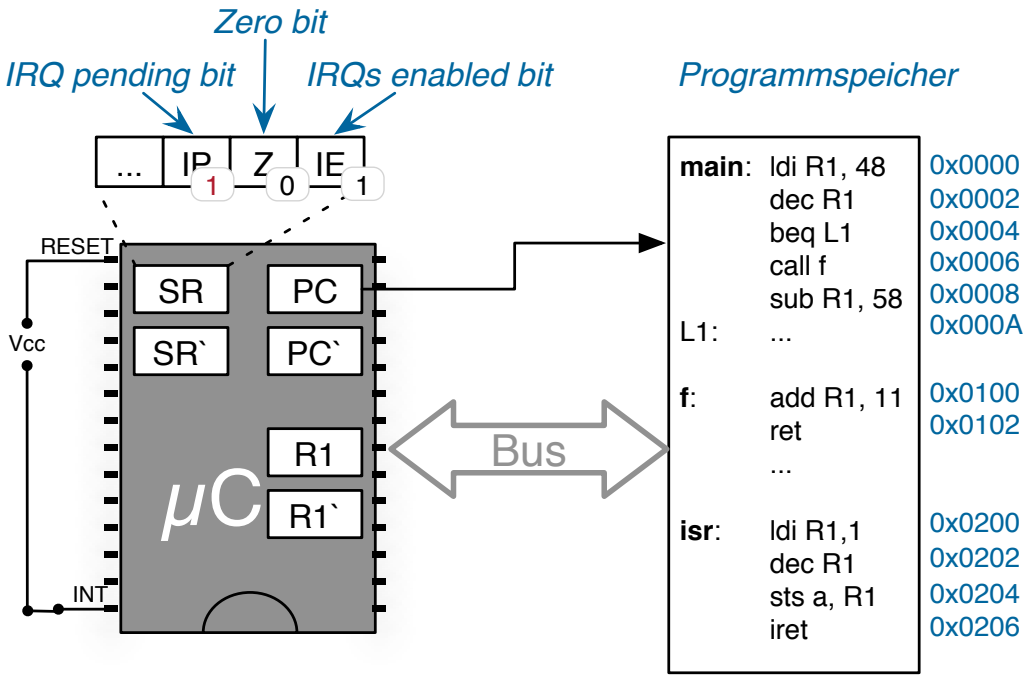


w: **call** <func>
PC` = PC
PC = func

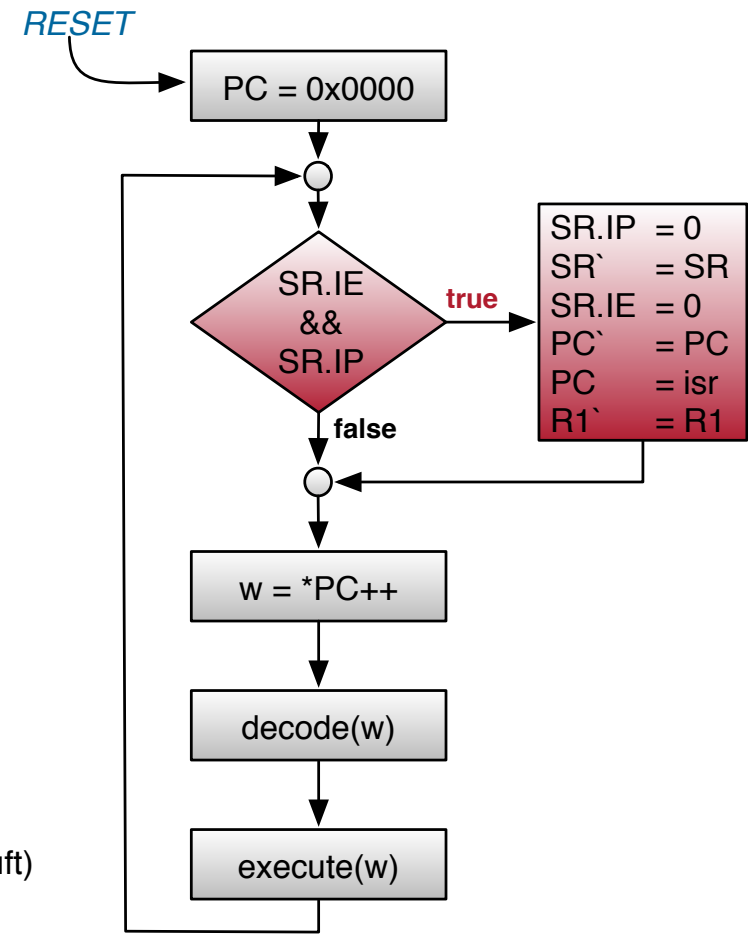
w: **ret**
PC = PC`

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

Ablauf eines Interrupts – Details



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

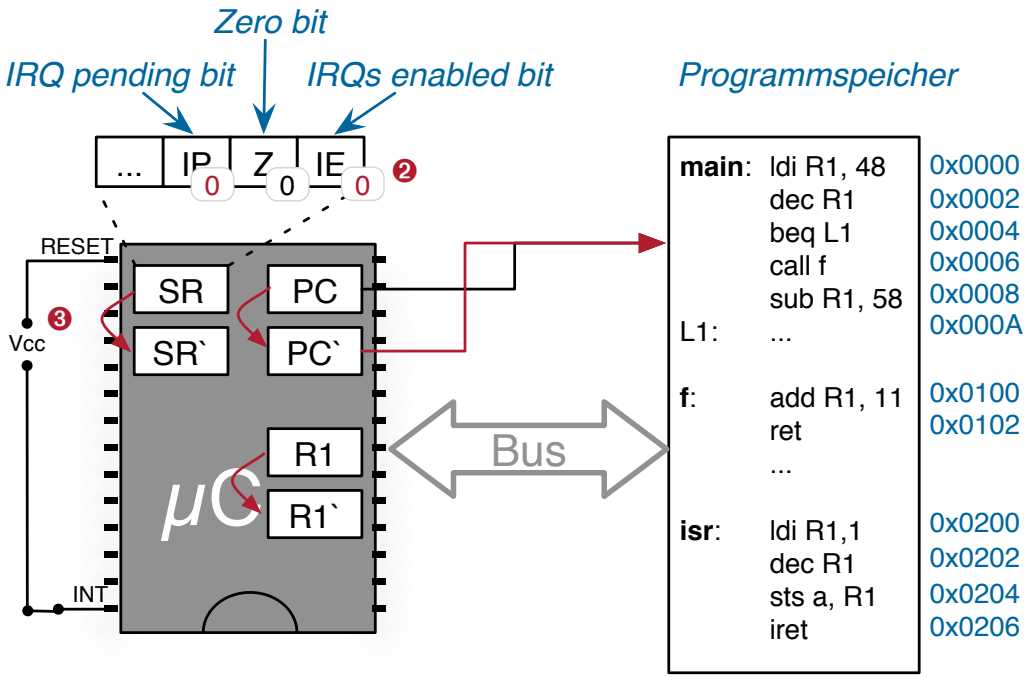


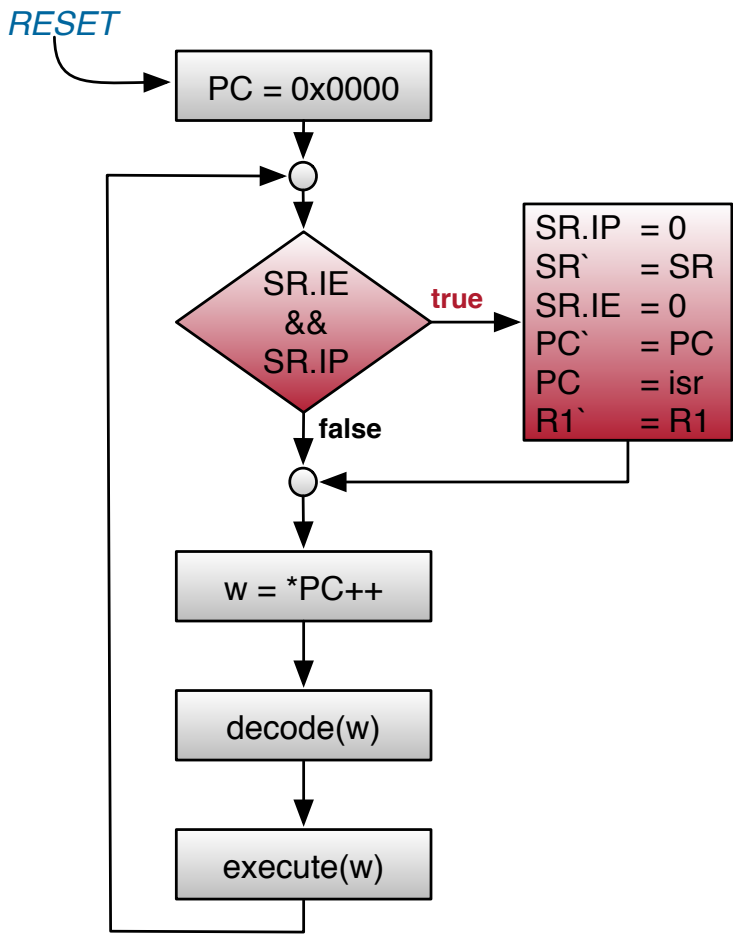
w: **call** <func>
PC` = PC
PC = func

w: **ret**
PC = PC`

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

Ablauf eines Interrupts – Details





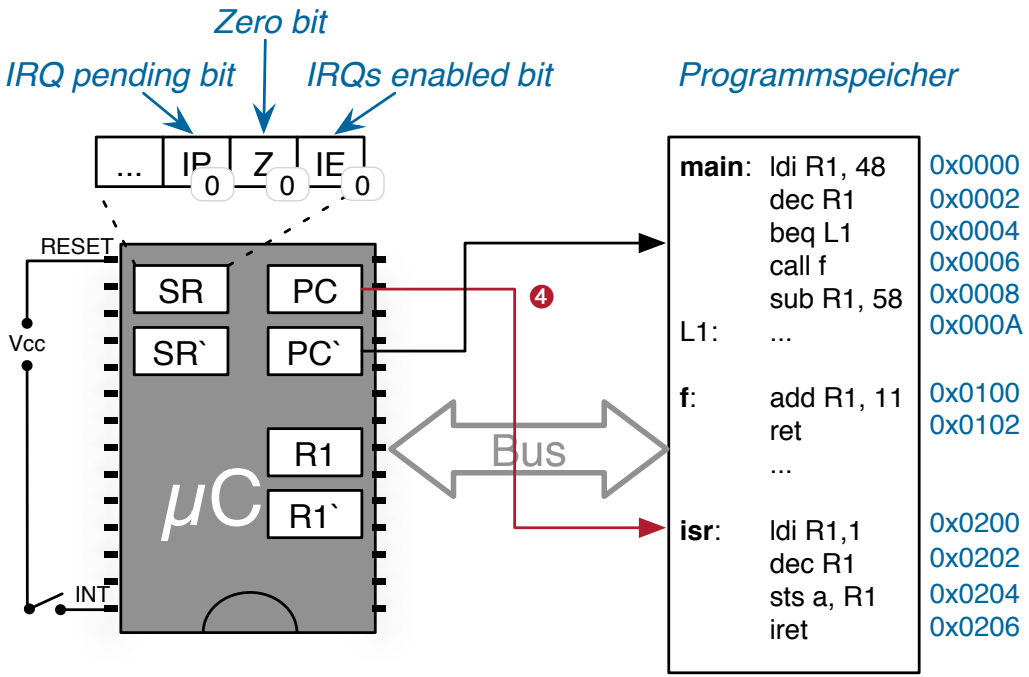
❷ Die Zustellung weiterer Interrupts wird verzögert
❸ Registerinhalte werden gesichert

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

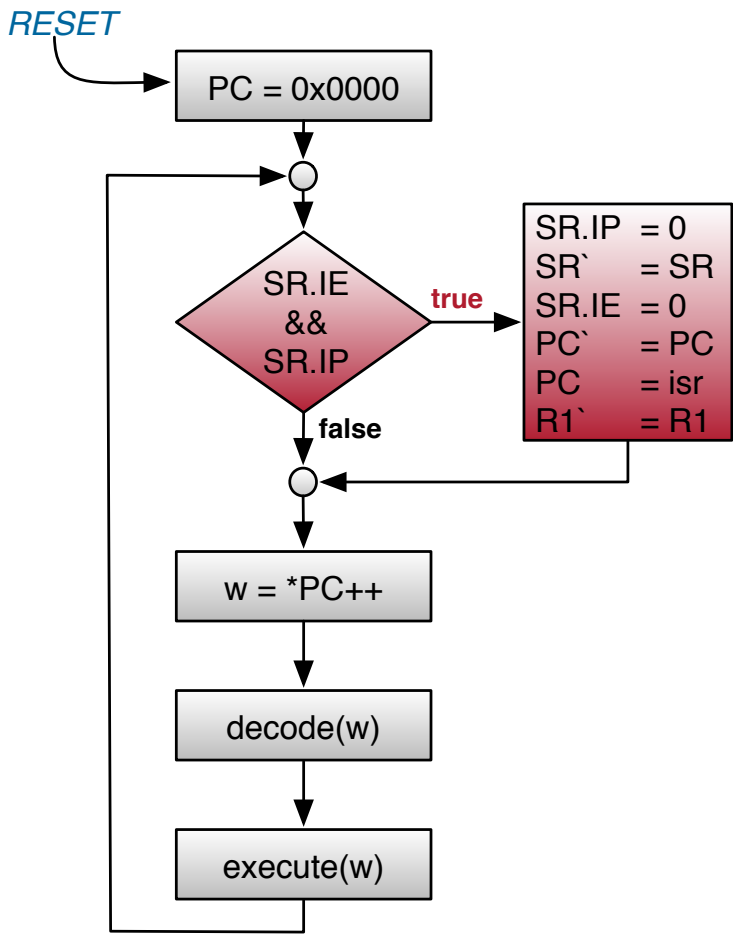
w: **ret**
PC = PC`

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

Ablauf eines Interrupts – Details



❹ Aufzurufende ISR wird ermittelt

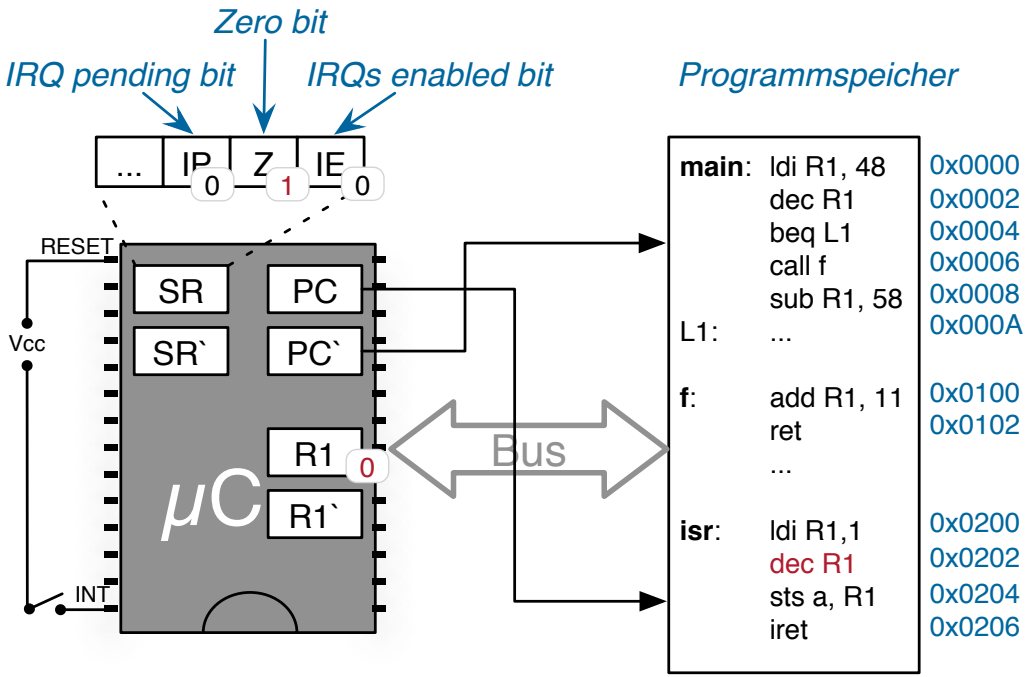


w: call <func>
PC` = PC
PC = func

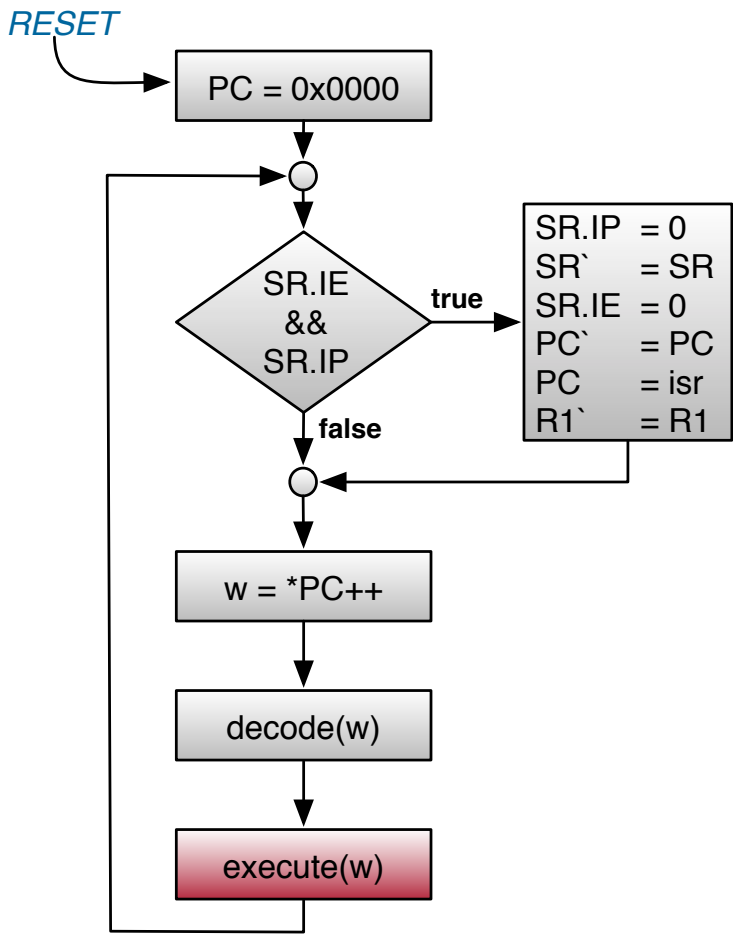
w: ret
PC = PC`

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

Ablauf eines Interrupts – Details



❺ ISR wird ausgeführt

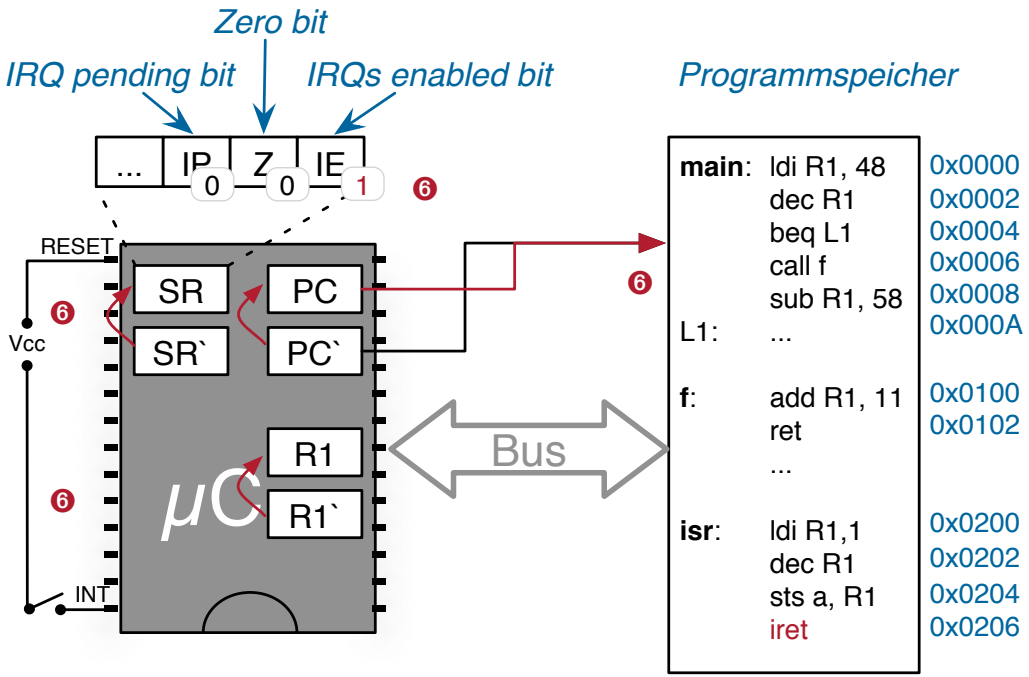


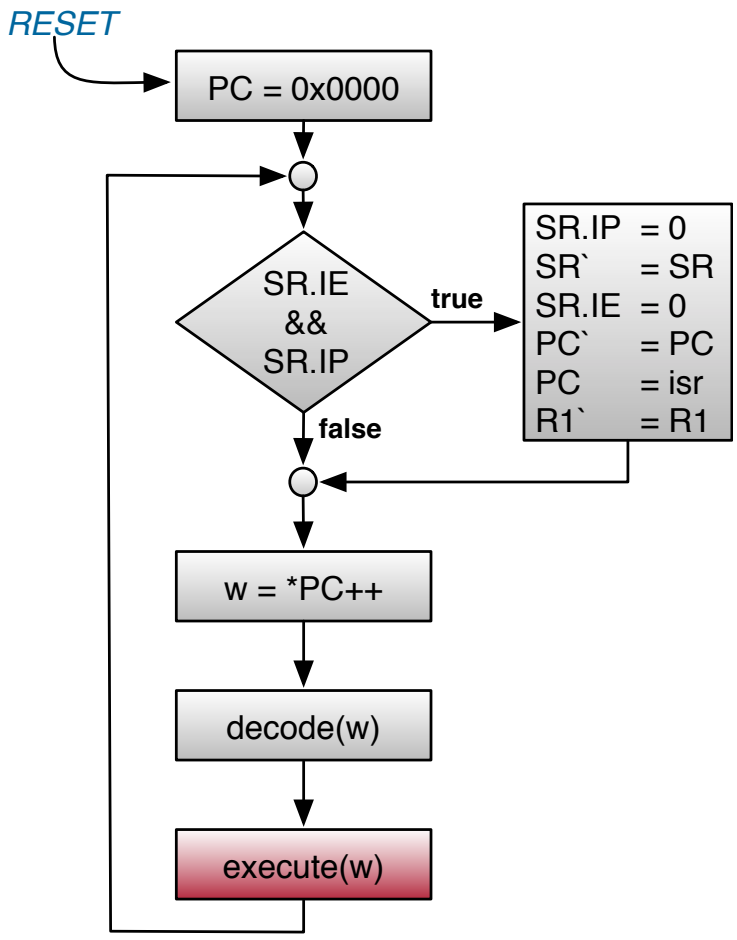
w: call <func>
PC` = PC
PC = func

w: ret
PC = PC`

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

Ablauf eines Interrupts – Details



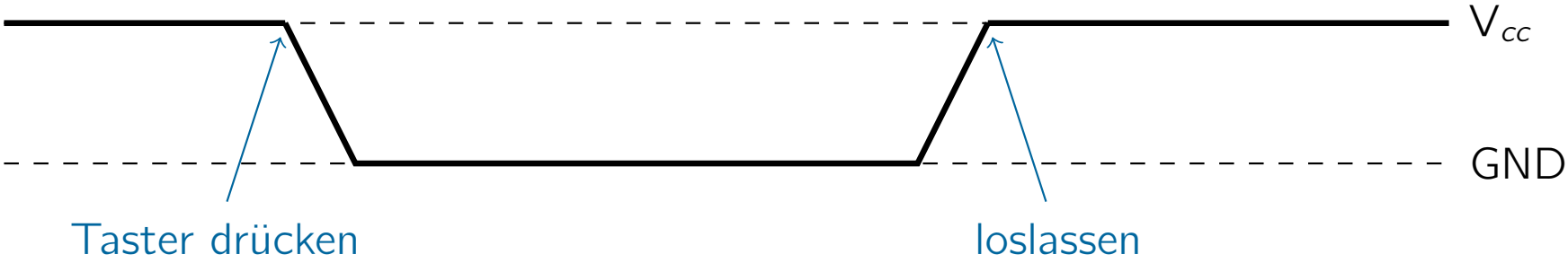


❻ ISR terminiert mit *iret*-Befehl
- Registerinhalte werden restauriert
- Zustellung von Interrupts wird reaktiviert
- Das Anwendungsprogramm wird fortgesetzt

w: **call** <func>	w: **ret**	w: **iret**
PC` = PC PC = func	PC = PC`	SR = SR` PC = PC` R1 = R1`

Pegel- und Flanken-gesteuerte Interrupts

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



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

Interruptsteuerung beim AVR ATmega

- IRQ-Quellen beim ATmega32 (IRQ $\mapsto$ *Interrupt ReQuest*)
  - 21 IRQ-Quellen [1, S. 45]
  - einzeln de-/aktivierbar
  - IRQ $\rightsquigarrow$ Sprung an Vektor-Adresse
- Verschaltung SPiCboard
  ($\hookrightarrow$ 14–14 $\hookrightarrow$ 2–4)
  - `INT0` $\mapsto$ `PD2` $\mapsto$ Button0 (hardwareseitig entprellt)
  - `INT1` $\mapsto$ `PD3` $\mapsto$ Button1

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

Externe Interrupts: Register

Steuerregister für `INT0` und `INT1`

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

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

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

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

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

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

Externe Interrupts: Verwendung

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

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

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

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


Externe Interrupts: Verwendung (Forts.)

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

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

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

Externe Interrupts: Verwendung (Forts.)

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


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

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

Nebenläufigkeit

Definition: Nebenläufigkeit

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

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

Nebenläufigkeitsprobleme

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

```
static volatile uint16_t cars;

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

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

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

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

Nebenläufigkeitsprobleme (Forts.)

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

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

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

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

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

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


Nebenläufigkeitsprobleme: *Lost-Update*-Anomalie

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

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

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

Nebenläufigkeitsprobleme: *Read-Write*-Anomalie

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

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

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

Interruptsperren: Datenflussanomalien verhindern

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

kritisches Gebiet

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

Nebenläufigkeitsprobleme (Forts.)

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

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

```
static volatile int8_t event;

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

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

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

Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

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

```
static volatile int8_t event;

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

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

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

Lost-Wakeup: Dornröschenschlaf verhindern

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

```
static volatile int8_t event;

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

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

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

Zusammenfassung

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

Überblick: Teil C Systemnahe Softwareentwicklung

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 globale und modullokale Variablen, sowie den Code
  - Allokation durch Platzierung in einer Sektion

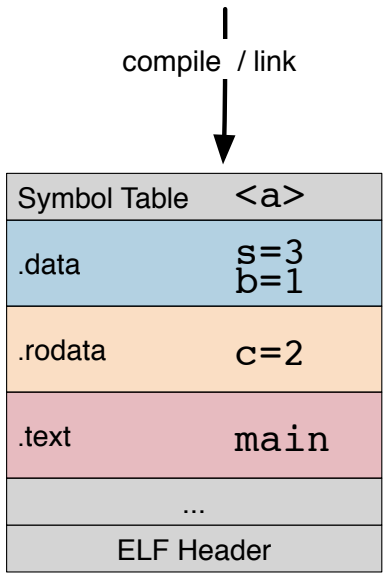
.text	– enthält den Programmcode	`main()`
.bss	– enthält alle uninitialisierten / mit 0 initialisierten Variablen	`a`
.data	– enthält alle initialisierten Variablen	`b,s`
.rodata	– enthält alle initialisierten unveränderlichen Variablen	`c`

- **Dynamische Allokation** – Reservierung zur Laufzeit
  - Betrifft lokale Variablen und explizit angeforderten Speicher

Stack	– enthält alle aktuell gültigen lokalen Variablen	`x,y,p`
Heap	– enthält explizit mit `malloc()` angeforderte Speicherbereiche	`*p`

Speicherorganisation auf einem μC

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

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



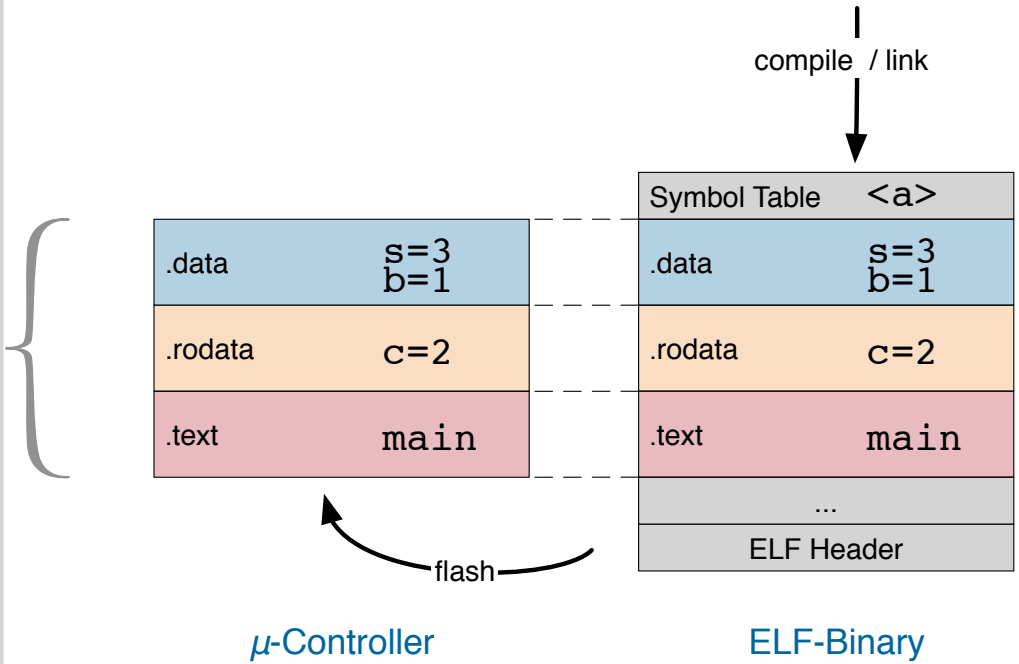
Quellprogramm

ELF-Binary

Beim Übersetzen und Linken werden die Programmelemente in entsprechenden Sektionen der ELF-Datei zusammen gefasst. Informationen zur Größe der `.bss`-Sektion landen ebenfalls in `.rodata`.

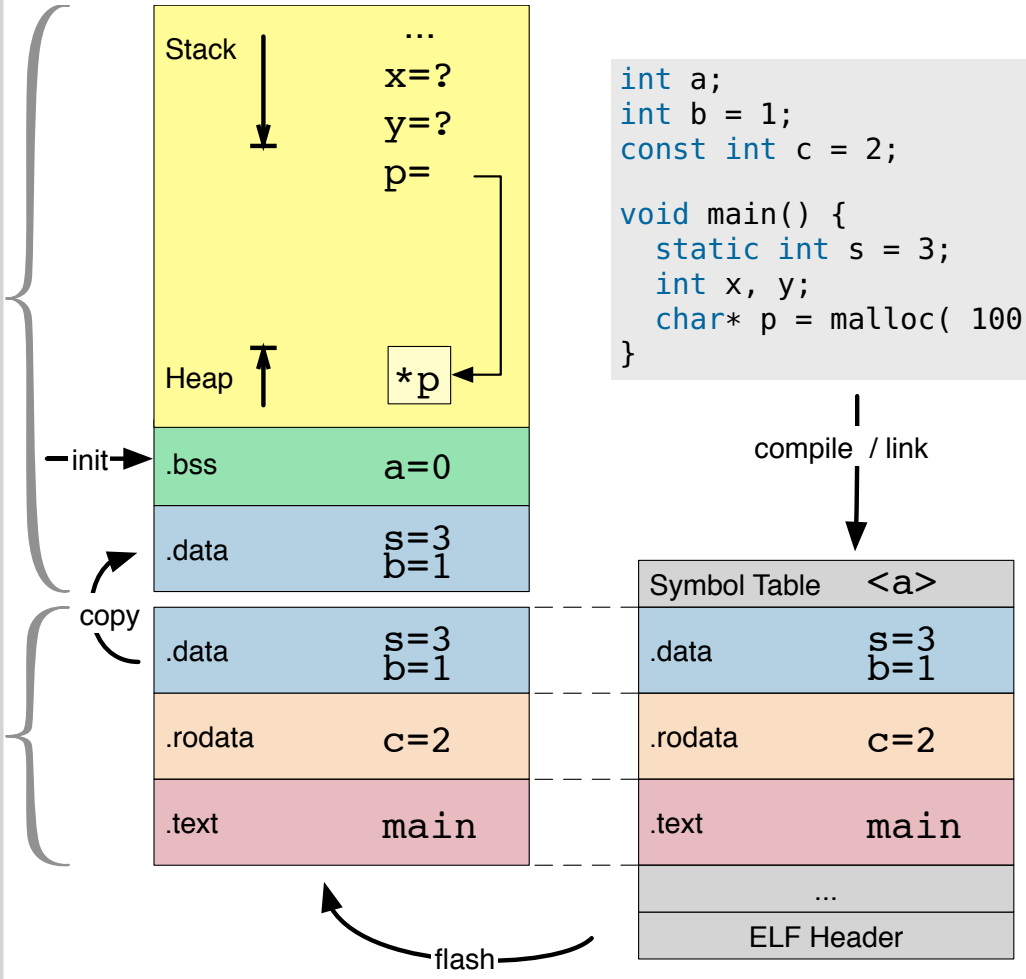
Speicherorganisation auf einem μC

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

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



Zur Installation auf dem μC werden `.text` und `.[ro]data` in den Flash-Speicher des μC geladen.

Speicherorganisation auf einem μC




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

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

Quellprogramm

Beim Systemstart wird das `.bss`-Segment im RAM angelegt und mit `0` initialisiert, das `.data`-Segment wird aus dem Flash ins RAM kopiert.
Das verbleibende RAM wird für den Stack und (falls vorhanden) den Heap verwendet.

Verfügt die Architektur über keinen Daten-Flashspeicher (beim ATmega der Fall ↪ 14–3), so werden konstante Variablen ebenfalls in `.data` abgelegt (und belegen zur Laufzeit RAM).

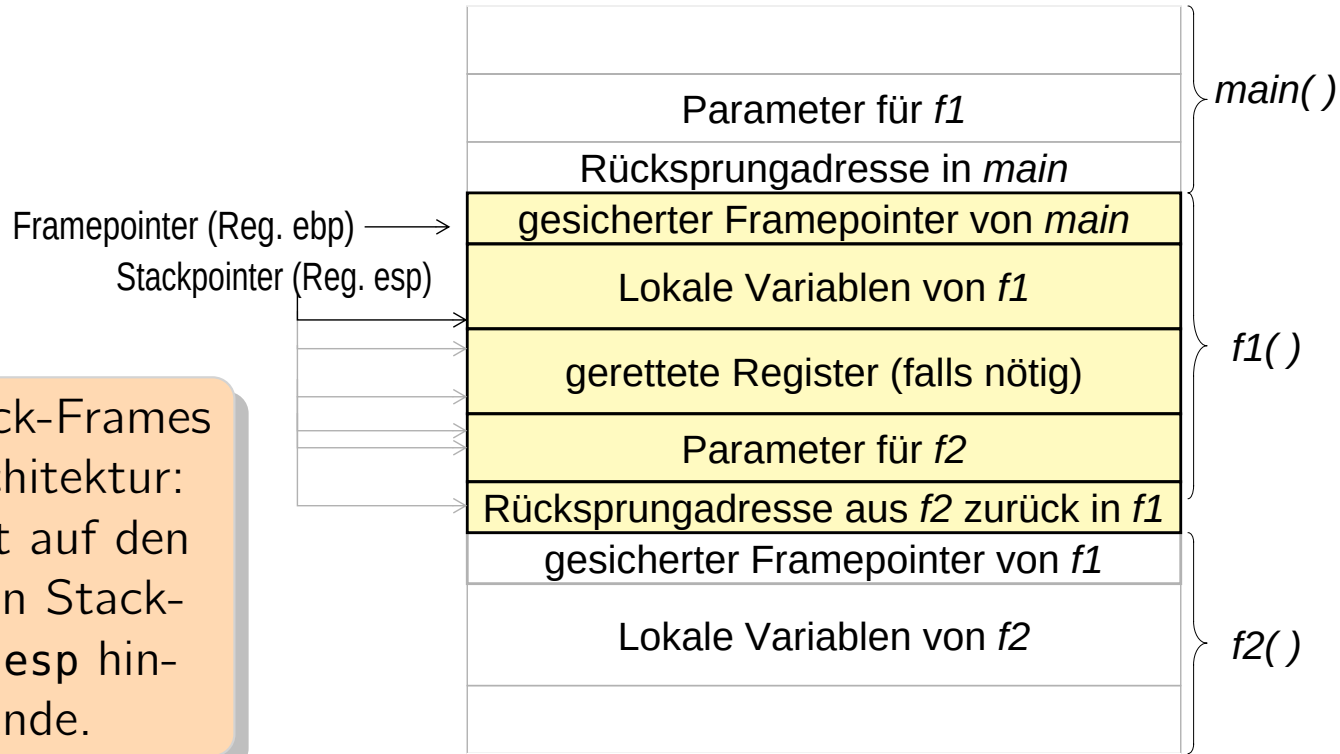
Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void* malloc( size_t n )` fordert einen Speicherblock der Größe n an; Rückgabe bei Fehler: 0-Zeiger (`NULL`)
  - `void free( void* pmem )` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

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


Dynamische Speicherallokation: Stack [↪ GDI, V-50]

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

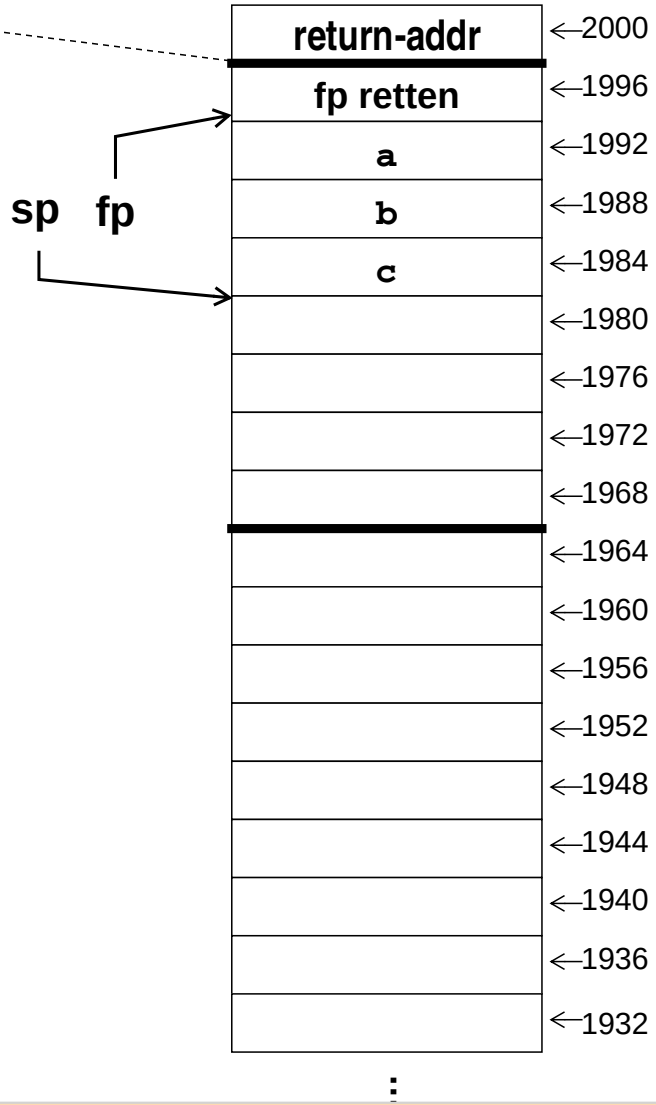


Aufbau eines Stack-Frames auf der IA-32-Architektur: Register **ebp** zeigt auf den Beginn des aktiven Stack-Frames; Register **esp** hinter das aktuelle Ende.

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

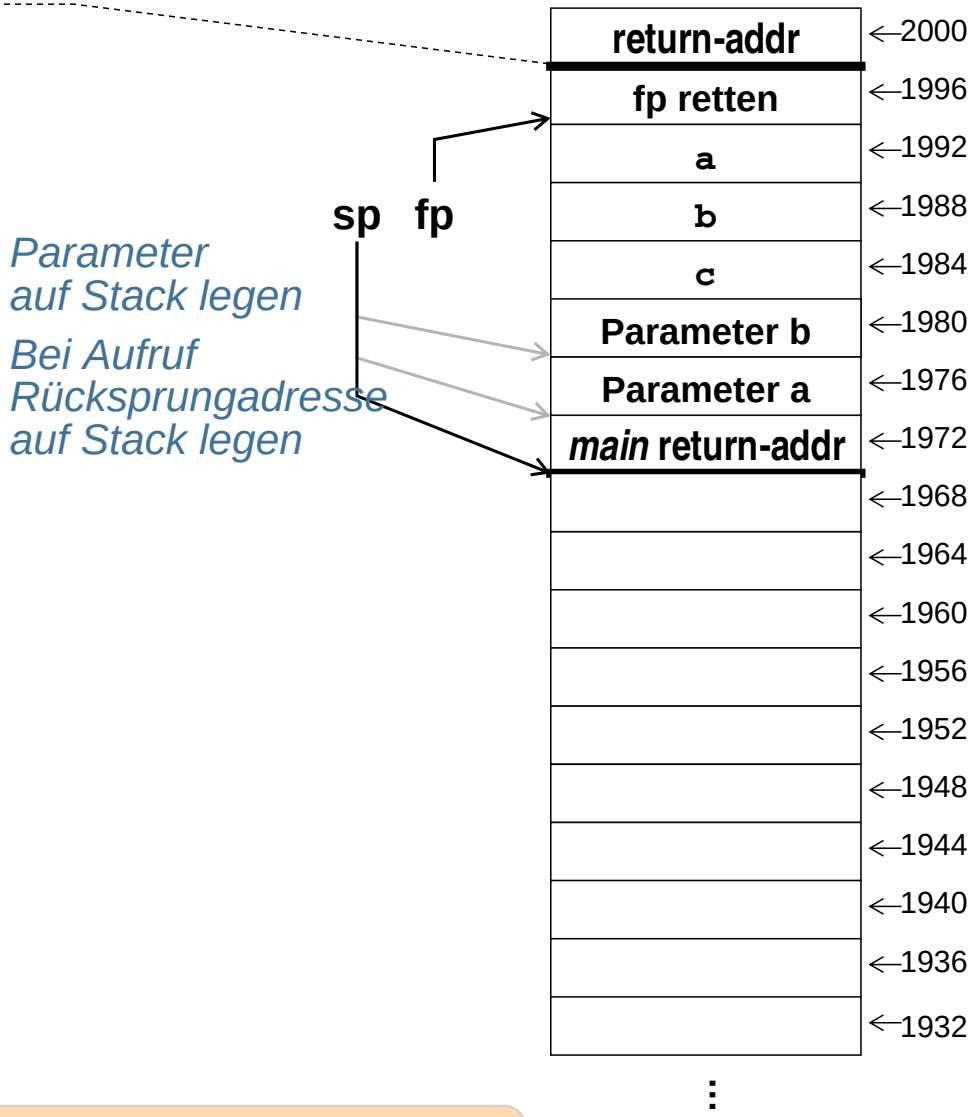
Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



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

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

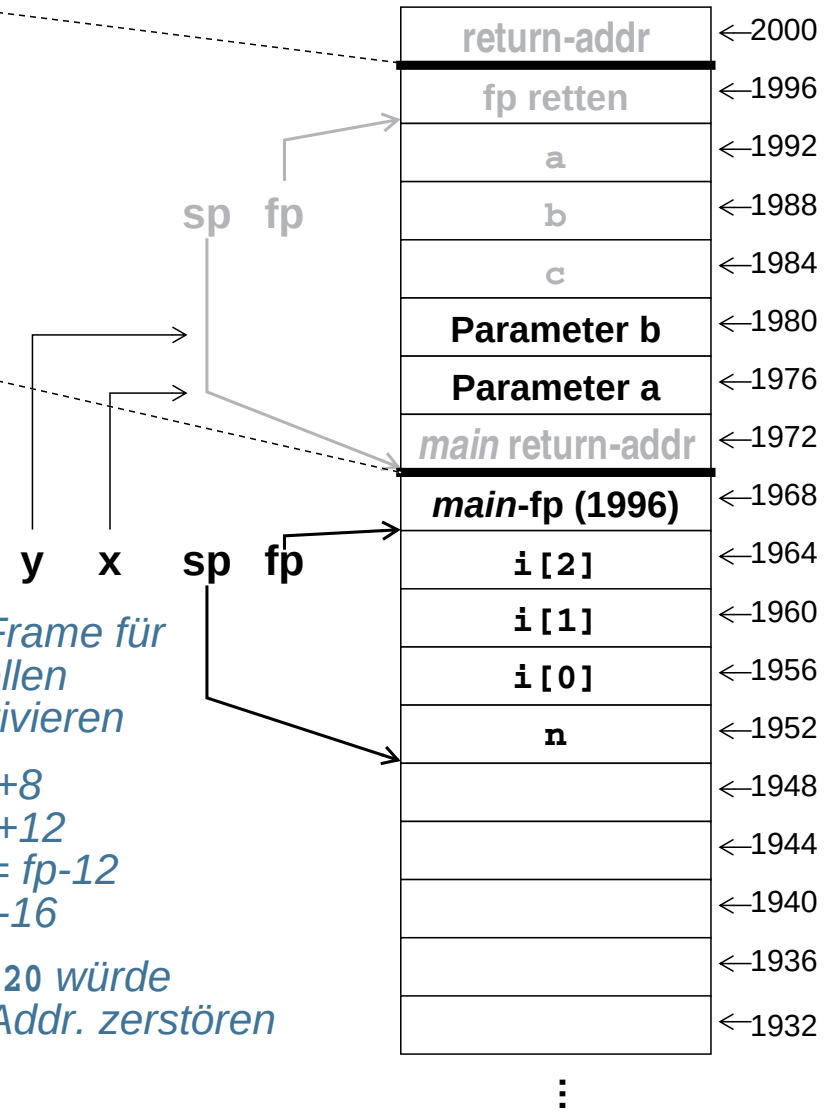


`main()` bereitet den Aufruf von `f1(int, int)` vor

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Stack-Frame für f1 erstellen und aktivieren

&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

`f1()` wurde soeben betreten

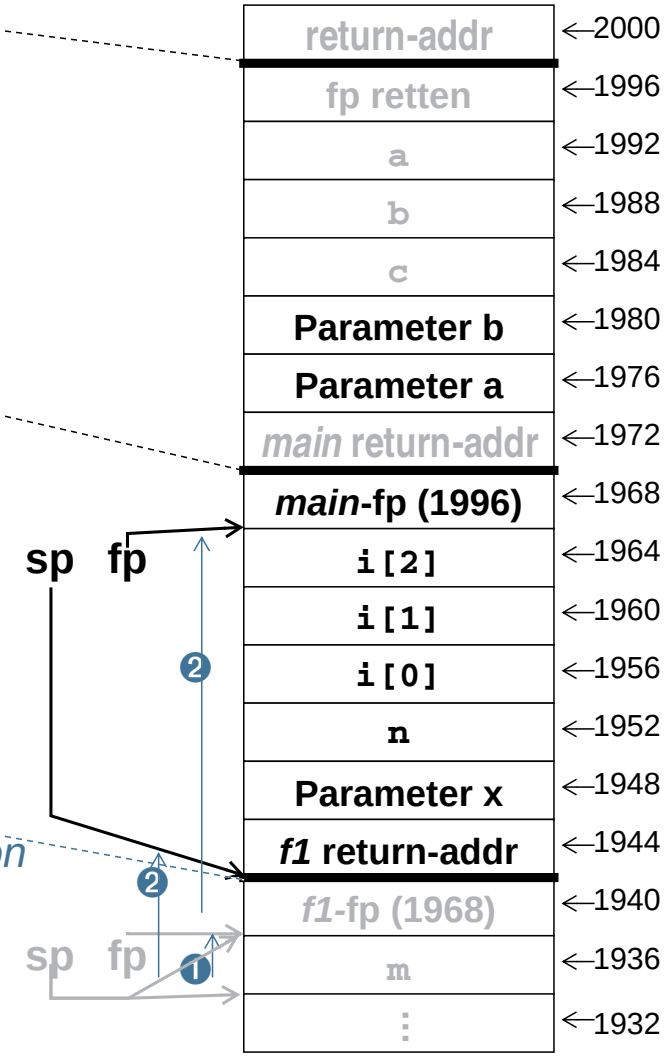
Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

Stack-Frame von f2 abräumen

❶ sp = fp

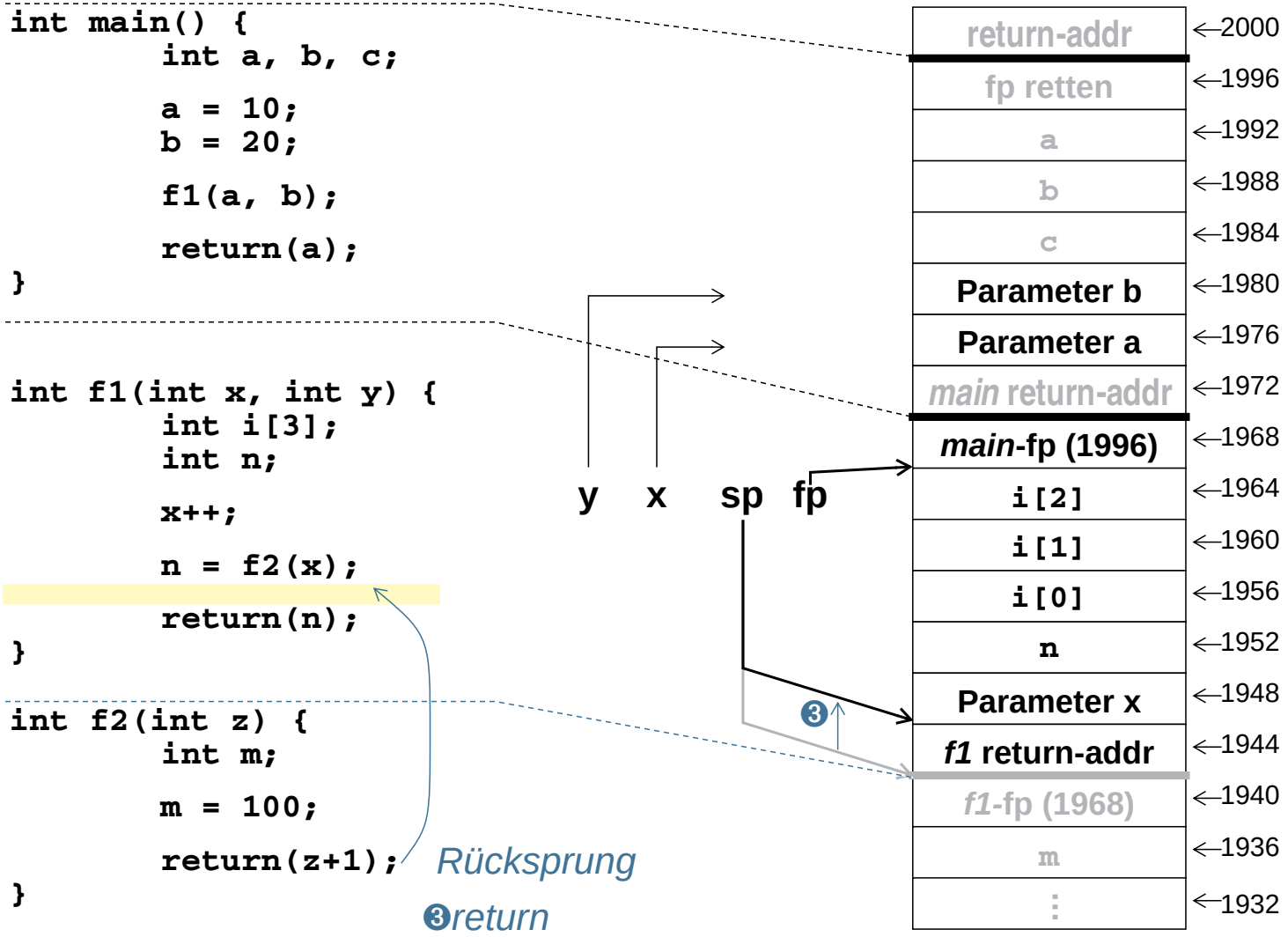
❷ fp = pop(sp)



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

Stack-Aufbau bei Funktionsaufrufen

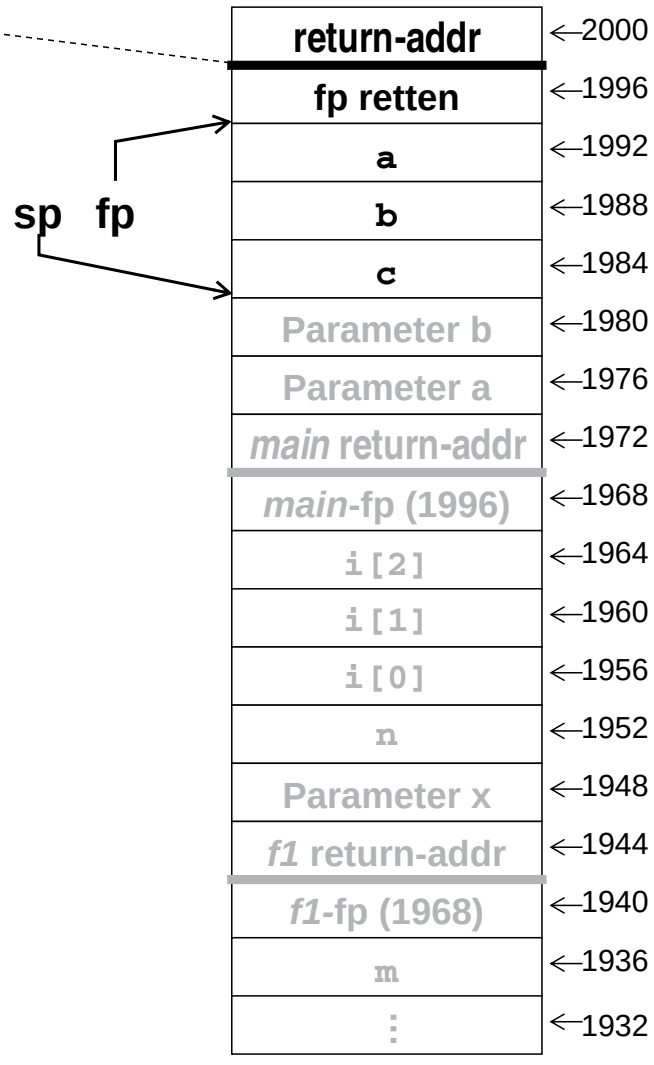
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

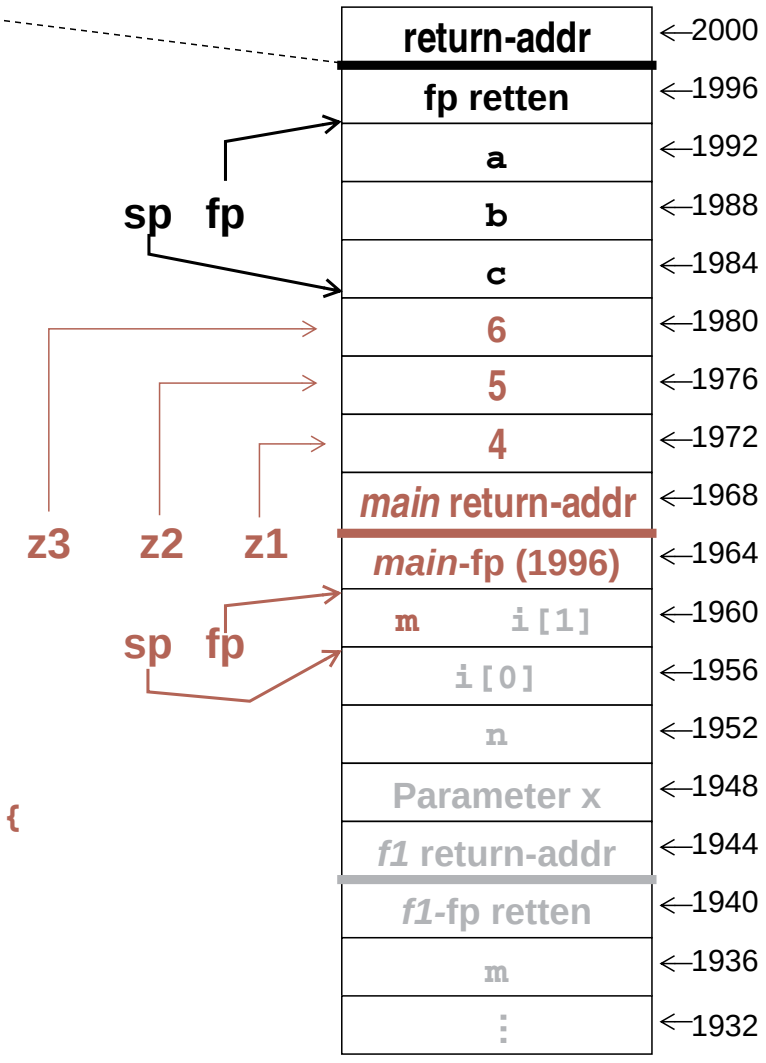


zurück in `main()`

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        f3(4,5,6);
}
```

was wäre, wenn man nach f1 jetzt eine Funktion f3 aufrufen würde?




```
int f3(int z1, int z2, int z3) {
        int m;

        return(m);
}
```

m wird nicht initialisiert ⇝„erbt“ alten Wert vom Stapel

Statische versus dynamische Allokation

- Bei der **μC-Entwicklung** wird statische Allokation bevorzugt
  - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
  - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ 1–3)

```
lohmann@faui48a:$ size sections.avr
text    data     bss     dec     hex filename
682     10       6       698     2ba sections.avr
```

Sektionsgrößen des Programms von ↪ 16–1

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