Grundlagen der Systemnahen Programmierung in C (GSPiC)

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2014

http://www4.cs.fau.de/Lehre/SS14/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.

[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`.

V_GSPIC_handout

Referenzen (Forts.)

[GDI] Volkmar Sieh and Frank Bauer. *Grundlagen der Informatik*. Vorlesung. (Referenzen beziehen sich auf Druckversion). Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Informatik, 2012 (jährlich). URL: `https://gdi.cs.fau.de/w12/material`.

[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.

V_GSPIC_handout

Veranstaltungsüberblick

V_GSPIC_handout

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

Sommersemester 2014

http://www4.cs.fau.de/Lehre/SS14/V_GSPIC

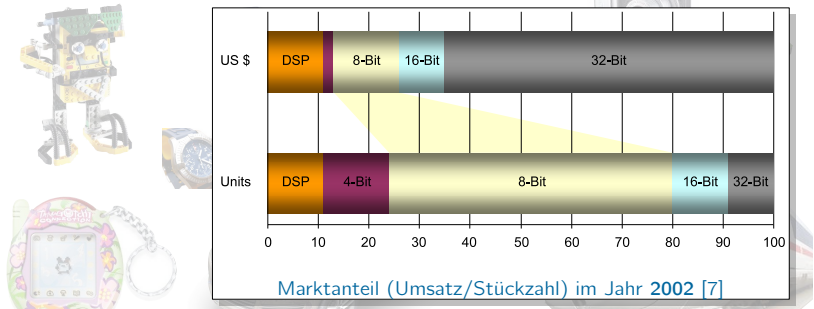
Überblick: Teil A Konzept und Organisation

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
  - 2014 liegt der Anteil an 8-Bittern (vermutlich) noch 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	I²C	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]

$\rightsquigarrow$ **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!**

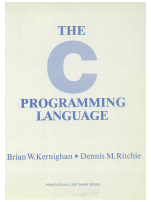
Literaturempfehlungen

[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: D0 12:15–13:45, H9
  - insgesamt 9 Vorlesungstermine $\hookrightarrow$ 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 10 Gruppen zur Auswahl
  - Anmeldung über Waffel (siehe Webseite): Heute, 18:00 – Fr, 18:00
  - Bei zu wenigen Teilnehmern behalten wir uns eine Verteilung auf andere Gruppen vor. Ihr werdet in diesem Fall per E-Mail angeschrieben.

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

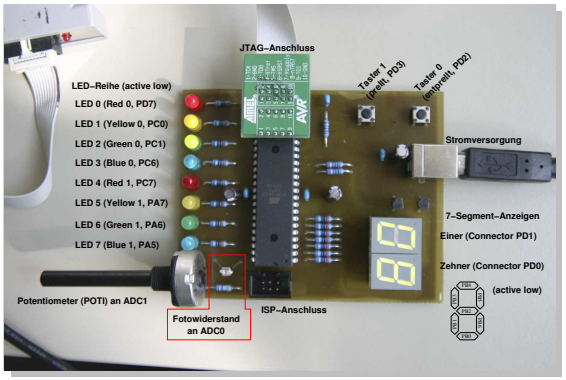
Programmieraufgaben

- Praktische Umsetzung des Vorlesungsstoffs
  - 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 kann jedoch eine Notenverbesserung bis **0,7 Notenpunkte** für die Prüfungsklausur erarbeitet werden!

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

Übungsplattform: Das SPiCboard

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



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

SPiCboard-Lötabend

- Die Fachschaften (EEI / ME) bieten einen „Lötabend“ an
  - Teilnahme ist freiwillig
  - (Erste) Löterfahrung sammeln beim Löten eines eigenen SPiCboards
- **Termine:** 14.–16. und 23.–25. April, 18:15–21:00
- **Anmeldung:** über Waffel (ab Mi. 18:00, siehe Webseite)
- **Kostenbeitrag:** SPiCBoard: 2 EUR (subventioniert)
  Progger: 30 EUR (optional!)

Die Progger (ISPs) können ggfs. auch gebraucht erworben werden.
Thread im EEI-Forum: `http://eei.fsi.fau.de/forum/post/3208`

Prüfung und Modulnote

- Prüfung (Klausur)
  - Termin: voraussichtlich Ende Juli / Anfang August
  - Dauer: 60 min
  - Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe
- Klausurnote ↦ 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 7,14% der möglichen ÜP ↦ $-0,1$ Klausurnote

  ⇝ 100% der möglichen ÜP ↦ $-0,7$ bei der Klausurnote

Semesterplanung

KW	Mo	Di	Mi	Do	Fr	Themen
15	07.04.	08.04.	09.04.	10.04.	11.04.	*Einführung, Organisation, Java nach C*
			VL 1	VL 2		*Abstraktion, Sprachüberblick, Datentypen*
16	14.04.	15.04.	16.04.	17.04.	18.04.	
	A1 (Blink)			*Gründonnerstag/ Karfreitag*		
17	21.04.	22.04.	23.04.	24.04.	25.04.	*Ausdrücke, Kontrollstrukturen, Funktionen*
	Ostern			VL 3		
18	28.04.	29.04.	30.04.	01.05.	02.05.	
	A2 (Snake)			*Tag der Arbeit*		
19	05.05.	06.05.	07.05.	08.05.	09.05.	*Funktionen (Forts.), Variablen, Präprozessor, Programmstruktur, Module*
				VL 4		
20	12.05.	13.05.	14.05.	15.05.	16.05.	*Zeiger*
	A3 (Spiel)			VL 5		
21	19.05.	20.05.	21.05.	22.05.	23.05.	*Mikrocontroller-Systemarchitektur, volatile, Verbundtypen (struct, union)*
				VL 6		
22	26.05.	27.05.	28.05.	29.05.	30.05.	
	A4 (LED)			*Himmelf.*		
23	02.06.	03.06.	04.06.	05.06.	06.06.	*Interrupts, Nebenläufigkeit*
				VL 7		
24	09.06.	10.06.	11.06.	12.06.	13.06.	
	Pfingsten/Berg					
25	16.06.	17.06.	18.06.	19.06.	20.06.	
	A5 (Ampel)			*Fronleich.*		
26	23.06.	24.06.	25.06.	26.06.	27.06.	*Speicherorganisation, Zusammenfassung*
				VL 8		
27	30.06.	01.07.	02.07.	03.07.	04.07.	
	Wiederholung					
28	07.07.	08.07.	09.07.	10.07.	11.07.	*Fragestunde*
				VL 9		

http://www4.cs.fau.de/Lehre/SS14/V_GSPIC

Beteiligte Personen, LS Informatik 4

Dozenten Vorlesung

Daniel Lohmann

Jürgen Kleinöder

Organisatoren des Übungsbetriebs

Rainer Müller

Moritz Strübe

01-02-Konzept: 2014-04-03

Beteiligte Personen, LS Informatik 4 (Forts.)

Techniker (Ausleihe SPiCboard und Debugger)

Harald Junggunst

Christian Preller

Daniel Christiani

01-02-Konzept: 2014-04-03

Beteiligte Personen, LS Informatik 4 (Forts.)

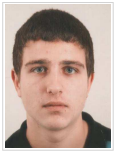
Übungsleiter

Maximilian Deckelmann

Michael Dzierza

Jonas Fuchs

Lucas Grunenberg



Christian Hintea

Florian Huber

Felix Kreyß

Markus Müller

Lars Schneider

Andreas Tschaftari

01-02-Konzept: 2014-04-03

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

Sommersemester 2014

http://www4.cs.fau.de/Lehre/SS14/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
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4   // greet user
5   printf("Hello World!\n");
6   return 0;
7 }
```

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

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


Das erste C-Programm – Erläuterungen [Handout]

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

- **Java**-Version zeilenweise erläutert
  1 Für die Benutzung der **Klasse** `out` wird das **Paket** `System` mit der `import`-Anweisung eingebunden.
  2 Jedes Java-Programm besteht aus mindestens einer **Klasse**.
  3 Jedes Java-Programm startet in `main()`, einer **statischen Methode** vom Typ `void`, die in genau einer **Klasse** definiert ist.
  5 Die Ausgabe einer Zeichenkette erfolgt mit der **Methode** `println()` aus der Klasse `out` aus dem Paket `System`. [↪ GDI, 13-18]
  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) ⇝ 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:

```c
#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:

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

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

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

  // wait until PD2 -> low (button is pressed)
  while(PIND & (1<<2))
    ;

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

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


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 $\rightsquigarrow$ PD2 = *high*.

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

Zum Vergleich: Benutzerinteraktion als Java-Programm

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

public class Input implements ActionListener {
  private JFrame frame;

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

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

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

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

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

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

Abstraktion durch Softwareschichten: *LED* $\rightarrow$ *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)

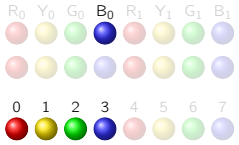
Nun: Entwicklung mit libspicboard

```
#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` $\mapsto$ `sb_set_led(RED0)`
  - Lese Bit 2 in `PORTD` $\mapsto$ `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)` ⇝



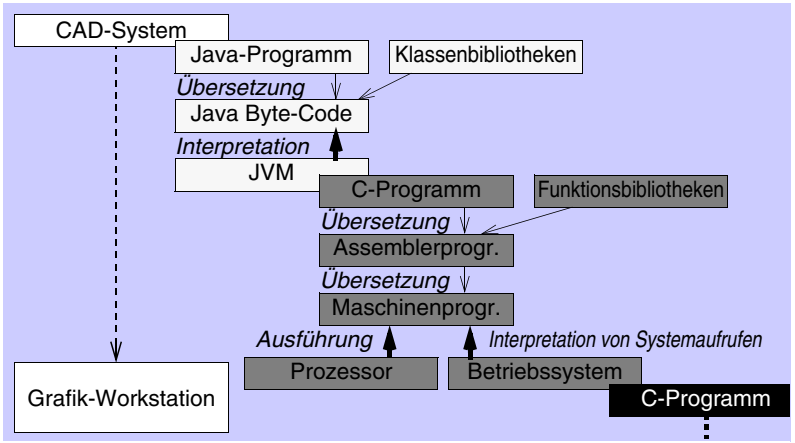
  - 7-Seg-Modul (`#include <7seg.h>`)
    - Ganzzahl $n \in \{-9 \ldots 99\}$ ausgeben: `sb_7seg_showNumber(47)` ⇝

- 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

Problemnähe ↑ Maschinennähe ↓

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

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

Beispiel: Fehlererkennung durch Betriebssystem

Linux: Division durch 0

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

Übersetzen und Ausführen ergibt:

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

⇝ Programm wird **abgebrochen**.

SPiCboard: Division durch 0

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

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

  while(1){}
}
```

Ausführen ergibt:

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

Überblick: Teil B Einführung in C

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 ↦ 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, 02-13]

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

  ⇝ Literale, Konstanten, Variablen, Funktionen haben einen **(Daten-)Typ**
- Datentyp legt fest
  - Repräsentation der Werte im Speicher
  - Größe des Speicherplatzes für Variablen
  - Erlaubte Operationen
- Datentyp wird festgelegt
  - Explizit, durch Deklaration, Typ-Cast oder Schreibweise (Literale)
  - Implizit, durch „Auslassung“ (⇝ int schlechter Stil!)

Primitive Datentypen in C

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

Integertypen [≈Java][↪ GDI, 02-14]

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

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 ⟶ Problemnähe

- **Problem:** Breite (⇝ Wertebereich) der C-Standardtypen ist implementierungsspezifisch ↦ **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe ↦ **Problemnähe**
  - Wertebereich sicher, aber möglichst kompakt darstellen
  - Register definierter Breite n bearbeiten
  - Code unabhängig von Compiler und Hardware halten (⇝ 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	→	255	`int8_t`	−128	→	+127
`uint16_t`	0	→	65.535	`int16_t`	−32.768	→	+32.767
`uint32_t`	0	→	4.294.967.295	`int32_t`	−2.147.483.648	→	+2.147.483.647
`uint64_t`	0	→	$> 1,8 * 10^{19}$	`int64_t`	$< -9,2 * 10^{18}$	→	$> +9,2 * 10^{18}$

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 `int`s (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, 02-18]

- 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 ↦ Integer [≈Java]

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

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

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


ASCII-Code-Tabelle (7 Bit)

ASCII ↦ *American Standard Code for Information Interchange*

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 0C	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:

`"Hi!"` ↦ | `'H'` | `'i'` | `'!'` | `0` | ← abschließendes 0-Byte

- Beispiel (Linux)

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

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

Ausblick: Komplexe Datentypen

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

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

  - Zeiger ↪ 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 ↪ 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. $-a$)	$\rightsquigarrow$ Multiplikation mit -1
unäres $+$	positives Vorzeichen (z. B. $+3$)	$\rightsquigarrow$ kein Effekt

- Zusätzlich nur für Ganzzahltypen:

%	Modulo (Rest bei Division)

Inkrement-/Dekrement-Operatoren [=Java][↪ GDI, 02-34]

- 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, 02-38]

- Vergleichen von zwei Ausdrücken

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

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

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


Logische Operatoren [≈Java][↪ GDI, 02-39]

- 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 ↦ *falsch*, ≠0 ↦ *wahr*
  - Ergebnis: *falsch* ↦ 0, *wahr* ↦ 1

Logische Operatoren – Auswertung [=Java]

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

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

$\underbrace{\underbrace{a > c}_{0} \;\&\&\; \underbrace{a > b}_{?}}_{0}$ ⟵ wird nicht ausgewertet, da der erste Term bereits *falsch* ergibt

- Kann überraschend sein, wenn Teilausdrücke Nebeneffekte haben

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


Zuweisungsoperatoren [=Java]

- Allgemeiner Zuweisungsoperator (=)
  - Zuweisung eines Wertes an eine Variable
  - Beispiel: `a = b + 23`
- Arithmetische Zuweisungsoperatoren (+=, −=, . . .)
  - Abgekürzte Schreibweise zur Modifikation des Variablenwerts
  - Beispiel: `a += 23` ist äquivalent zu `a = a + 23`
  - Allgemein: `a` *op*`= b` ist äquivalent zu `a = a` *op* `b`
    für *op* ∈ { +, −, ⋆, %, <<, >>, &, ^, | }
- 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! $\rightsquigarrow$ Fehler wird leicht übersehen!

Bitoperationen [=Java][↪ GDI, 02-41]

- 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, 02-41]

- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

<<	bitweises Linksschieben (rechts werden 0-Bits „nachgefüllt“)
>>	bitweises Rechtsschieben (links werden 0-Bits „nachgefüllt“)

- Beispiele (x sei vom Typ `uint8_t`)

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

Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

Bit#	7	6	5	4	3	2	1	0	
PORTD	?	?	?	?	?	?	?	?	Bit 7 soll verändert werden, die anderen Bits jedoch erhalten bleiben!
0x80	1	0	0	0	0	0	0	0	Setzen eines Bits durch **Ver-odern** mit Maske, in der nur das Zielbit 1 ist
PORTD |= 0x80	1	?	?	?	?	?	?	?	
~0x80	0	1	1	1	1	1	1	1	Löschen eines Bits durch **Ver-unden** mit Maske, in der nur das Zielbit 0 ist
PORTD &= ~0x80	0	?	?	?	?	?	?	?	
0x08	0	0	0	0	1	0	0	0	Invertieren eines Bits durch **Ver-xodern** mit Maske, in der nur das Zielbit 1 ist
PORTD ^= 0x08	?	?	?	?	¿	?	?	?	

Bitoperationen – Anwendung (Forts.)

- Bitmasken werden gerne als Hexadezimal-Literale angegeben

Bit#	7	6	5	4	3	2	1	0
0x8f	1	0	0	0	1	1	1	1
	8				f			

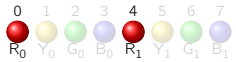
Jede Hex-Ziffer repräsentiert genau ein Halb-Byte (*Nibble*) ⇝ Verständlichkeit

- 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, 02-44]

- Formulierung von Bedingungen in Ausdrücken

  Ausdruck$_1$ **?** *Ausdruck*$_2$ **:** *Ausdruck*$_3$
  - Zunächst wird *Ausdruck*$_1$ ausgewertet
    - *Ausdruck*$_1 \neq 0$ (*wahr*) ⇝ Ergebnis ist *Ausdruck*$_2$
    - *Ausdruck*$_1 = 0$ (*falsch*) ⇝ 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
    ⇝ 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, 02-45]

	Klasse	Operatoren	Assoziativität

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

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

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

Typumwandlung in Ausdrücken

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

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

res   = a * b / c;            // promotion to int: 300 fits in!
int8_t: 75    int: 300
                  int: 75
```

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

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

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


Typumwandlung in Ausdrücken (Forts.)

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

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

res   =  a * b   /   4.0  ; // promotion to double
int8_t: 75    double: 300.0    double: 4.0
                  double: 75.0
```

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

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

res  =     s     < u;    // promotion to unsigned: -1 --> 65535
int: 0    unsigned: 65535
              unsigned: 0
```

↝ Überraschende Ergebnisse bei negativen Werten!
↝ Mischung von `signed`- und `unsigned`-Operanden vermeiden!

Typumwandlung in Ausdrücken – Typ-Casts

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

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

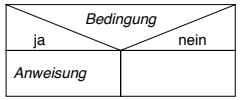
```
int s = -1, res;      // range: -32768 --> +32767
unsigned u = 1;       // range: 0 --> 65535

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


Überblick: Teil B Einführung in C

Bedingte Anweisung [=Java][↪ GDI, 03-11]

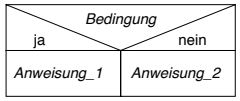
- `if`-Anweisung (bedingte Anweisung)

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



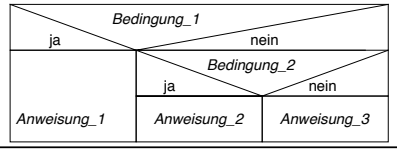
- `if-else`-Anweisung (einfache Verzweigung)

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



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

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



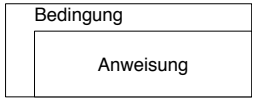
Fallunterscheidung [=Java][↪ GDI, 03-15]

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


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


Abweisende und nicht-abweisende Schleife [=Java]

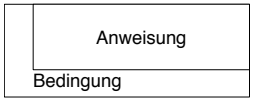
- Abweisende Schleife
  [↪ GDI, 03-26]
  - `while`-Schleife
  - Null- oder mehrfach ausgeführt




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

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

- Nicht-abweisende Schleife
  [↪ GDI, 03-27]
  - `do-while`-Schleife
  - Ein- oder mehrfach ausgeführt



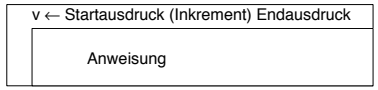

```
do
  Anweisung;
while( Bedingung );
```

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


Zählende Schleife [=Java][↪ GDI, 03-19]

- `for`-Schleife (Laufanweisung)

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



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

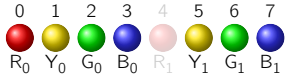
```
uint8_t sum = 0; // calc sum 1+...+10
for (uint8_t n = 1; n < 11; n++) {
  sum += n;
}
sb_7seg_showNumber( sum );
```

- 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, 03-23]

- 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, 04-10]
  - 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, 04-26]
  - *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 ↦ $$$

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 [= $Ausdr_2$]$_{opt}$, . . .]$_{opt}$;
  - 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.)

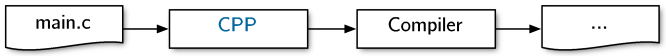
- Variablen können an verschiedenen Positionen definiert werden
  - Global — außerhalb von Funktionen, üblicherweise am Kopf der Datei
  - Lokal — zu Beginn eines `{ Blocks }`, direkt nach der öffnenden Klammer C89
  - Lokal — überall dort, wo eine Anweisung stehen darf C99

```
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                    << hat geringere Präzedenz als *
n = POW2( 2 ) * 3                  ⇝ n = 1 << 2 * 3
```

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

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

  - Aber nicht alle

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

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

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