

# Grundlagen der Systemnahen Programmierung in C (GSPiC)

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2017

[http://www4.cs.fau.de/Lehre/SS17/V\\_GSPIC](http://www4.cs.fau.de/Lehre/SS17/V_GSPIC)

V\_GSPIC\_handout



## Referenzen (Forts.)

- [6] David Tennenhouse. "Proactive Computing". In: *Communications of the ACM* (May 2000), pp. 43–45.
- [7] Jim Turley. "The Two Percent Solution". In: *embedded.com* (Dec. 2002). <http://www.embedded.com/story/0EG20021217S0039>, visited 2011-04-08.

V\_GSPIC\_handout



© kls GSPiC (SS17)

## Referenzen

- [1] *ATmega328PB 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. Atmel Corporation. Oct. 2015.
- [GDI] Frank Bauer. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2015 (jährlich). URL: <https://gdi.cs.fau.de/w15/material>.
- [2] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop, 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](https://doi.org/10.1145/361011.361061).

V\_GSPIC\_handout



© kls GSPiC (SS17)

## Veranstaltungsüberblick

### Teil A: Konzept und Organisation

#### 1 Einführung

#### 2 Organisation

### Teil B: Einführung in C

#### 3 Java versus C – Erste Beispiele

#### 4 Softwareschichten und Abstraktion

#### 5 Sprachüberblick

#### 6 Einfache Datentypen

#### 7 Operatoren und Ausdrücke

#### 8 Kontrollstrukturen

### 9 Funktionen

### 10 Variablen

### 11 Präprozessor

### Teil C: Systemnahe Softwareentwicklung

#### 12 Programmstruktur und Module

#### 13 Zeiger und Felder

#### 14 µC-Systemarchitektur

#### 15 Nebenläufigkeit

#### 16 Speicherorganisation

V\_GSPIC\_handout



© kls GSPiC (SS17)

# Grundlagen der Systemnahen Programmierung in C (GSPiC)

## Teil A Konzept und Organisation

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2017

[http://www4.cs.fau.de/Lehre/SS17/V\\_GSPiC](http://www4.cs.fau.de/Lehre/SS17/V_GSPiC)

V\_GSPiC\_handout



### 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  $\mu$ -Controller ( $\mu$ C)
  - SPIboard-Lehrentwicklungsplattform mit ATmega- $\mu$ C
  - **Praktische Erfahrungen** in hardwarenaher Softwareentwicklung machen
- **Verstehen** der technologischen Sprach- und Hardwaregrundlagen für die Entwicklung systemnaher Software
  - Die Sprache C verstehen und einschätzen können
  - Umgang mit Nebenläufigkeit und Hardwarenähe

V\_GSPiC\_handout



## Überblick: Teil A Konzept und Organisation

### 1 Einführung

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

### 2 Organisation

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

V\_GSPiC\_handout

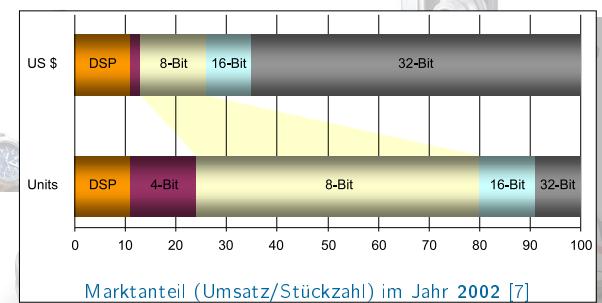


### Motivation: Eingebettete Systeme

- **Omnipräsent:**
- **Kostensensitiv:**

**98–99 Prozent** aller Prozessoren wurden im Jahr 2000 in einem **eingebetteten System** verbaut [6]

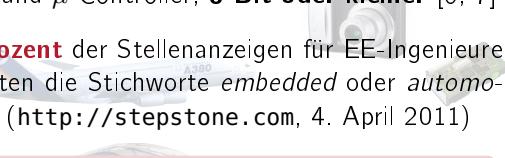
**70–80 Prozent** aller produzierten Prozessoren sind DSPs und  $\mu$ -Controller, **8-Bit oder kleiner** [6, 7]



01-02-Konzept: 2017-04-24



## Motivation: Eingebettete Systeme

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

Bei den oberen Zahlen ist gesunde Skepsis geboten

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

01-402-Konzept: 2017-04-24



## 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)
    - Übersetzer 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!

01-402-Konzept: 2017-04-24



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

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

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

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

01-402-Konzept: 2017-04-24



## 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  $\mu$ -Controller
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (EEI)

V\_GSPiC\_handout



## Vorlesungsskript

- Das Handout der Vorlesungsfolien wird online und als 4 x 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!**

01-02-Konzept: 2017-04-24



## 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- $\mu$ C)
    - Abbildung Speicher  $\leftrightarrow$  Sprachkonstrukte
    - Unterbrechungen (*interrupts*) und Nebenläufigkeit
- Termin: Do 12:15–13:45, H8
  - Einzeltermin am 24. April (Mo), 12:15–13:45, H8
  - insgesamt 9 Vorlesungstermine

V\_GSPiC\_handout



$\leftrightarrow$  [2-7]

## Literaturempfehlungen

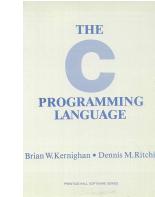
- [2] Für den Einstieg empfohlen:

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



## 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 (Atmel Studio)
    - Betreuung durch Übungsbetreuer
- Termin: Initial 8 Gruppen zur Auswahl
  - Anmeldung über Waffel (siehe Webseite): Heute, 18:00 – So, 18:00
  - Bei zu wenigen Teilnehmern behalten wir uns eine Verteilung auf andere Gruppen vor. Ihr werdet in diesem Fall per E-Mail angeschrieben.

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

01-02-Konzept: 2017-04-24



# Programmieraufgaben

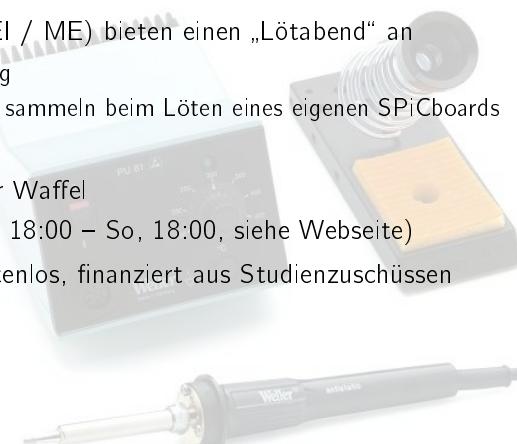
- Praktische Umsetzung des Vorlesungsstoffs
  - Fünf Programmieraufgaben (Abgabe ca. alle 14 Tage) → [2-7]
  - Bearbeitung wechselseitig alleine / mit Übungspartner
- Lösungen mit Abgabeskript am Rechner abgeben
  - Lösung wird durch Skripte überprüft
  - Wir korrigieren und bepunkteten die Abgaben und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

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



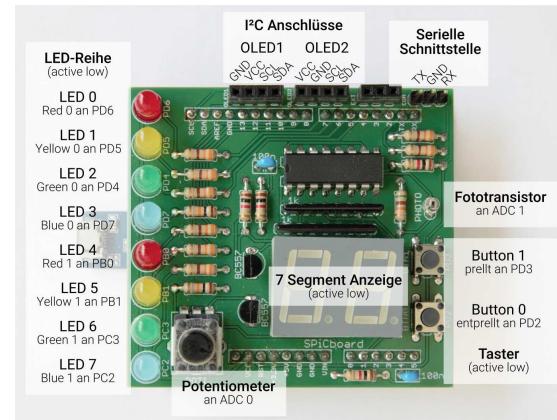
# SPiCboard-Lötabend

- Die Fachschaften (EEI / ME) bieten einen „Lötabend“ an
  - Teilnahme ist freiwillig
  - (Erste) Löterfahrung sammeln beim Löten eines eigenen SPiCboards
- **Anmeldung:** über Waffel  
(Di, 18:00 – So, 18:00, siehe Webseite)
- **Kostenbeitrag:** kostenlos, finanziert aus Studienzuschüssen



## Übungsplattform: Das SPiCboard

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



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



## 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 5% der möglichen ÜP → +1% der möglichen KP
  - 100% der möglichen ÜP → +10% der möglichen KP





## Beteiligte Personen, LS Informatik 4

### Semesterplanung

KW	Mo	Di	Mi	Do	Fr	Sa	Themen
17	24.04.	25.04.	26.04.	27.04.	28.04.		Erstellung, Optimierung, Anpassung Abstraktion, Spezialisierung, Darlegungen
18	01.05.	02.05.	03.05.	04.05.	05.05.		Ausprägen, Konkretisieren, Funktions
19	08.05.	09.05.	10.05.	11.05.	12.05.		Aufgaben, Praktische, Theoretische, Programmatischer Module
20	15.05.	16.05.	17.05.	18.05.	19.05.		Zusage
21	22.05.	23.05.	24.05.	25.05.			
22	29.05.	30.05.	31.05.	01.06.	02.06.		
23	05.06.	06.06.	07.06.	08.06.	09.06.		
24	12.06.	13.06.	14.06.	15.06.	16.06.		
25	19.06.	20.06.	21.06.	22.06.	23.06.		
26	26.06.	27.06.	28.06.	29.06.	30.06.		
27	02.07.	03.07.	04.07.	05.07.	07.07.		
28	10.07.	11.07.	12.07.	13.07.	14.07.		
29	17.07.	18.07.	19.07.	20.07.	21.07.		
30	24.07.	25.07.	26.07.	27.07.	28.07.		
							Prüfung
							VL 9

<http://www4.cs.fau.de/Lehre/SS17/VL.GSPIC>

## Beteiligte Personen, LS Informatik 4 (Forts.)

### Techniker (Ausleihe SPiCboard)



Harald Jungjunist



Christian Preller



Daniel Christiani

### Dozenten Vorlesung



Volkmar Sieh



Jürgen Kleinöder

### Organisatoren des Übungsbetriebs



Rainer Müller



Sebastian Maier



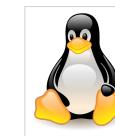
Heiko Janker

## Beteiligte Personen, LS Informatik 4 (Forts.)

### Übungsleiter



Constantin Baumann



Fabian Dechent



Tom Endres



Lukas Geyer



Julius Gohsrich



Jonas Herterich



Mike Schmidt



Tim Zunner

## Bei Fragen oder Problemen

- Vorlesungs- und Übungsfolien konsultieren
- Häufig gestellte Fragen (FAQ) und Antworten siehe Webseite  
→ [http://www4.cs.fau.de/Lehre/SS17/V\\_GSPIC](http://www4.cs.fau.de/Lehre/SS17/V_GSPIC)  
→ Übungen  
→ FAQ
- Allgemeine Fragen zu Übungsaufgaben etc. im EEI-Forum posten  
→ <https://eei.fsi.uni-erlangen.de/forum/forum/16>
- Bei speziellen Fragen Mail an Mailingliste (alle Übungsleiter)  
→ [i4spic@cs.fau.de](mailto:i4spic@cs.fau.de)

01-402-Konzept: 2017-04-24



## Überblick: Teil B Einführung in C

### 3 Java versus C – Erste Beispiele

### 4 Softwareschichten und Abstraktion

### 5 Sprachüberblick

### 6 Einfache Datentypen

### 7 Operatoren und Ausdrücke

### 8 Kontrollstrukturen

### 9 Funktionen

### 10 Variablen

### 11 Präprozessor

V\_GSPiC\_handout



03-04-ErsteSchritte: 2017-03-03

# Grundlagen der Systemnahen Programmierung in C (GSPiC)

## Teil B Einführung in C

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

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2017

[http://www4.cs.fau.de/Lehre/SS17/V\\_GSPIC](http://www4.cs.fau.de/Lehre/SS17/V_GSPIC)

## Das erste C-Programm

- Das berühmteste Programm der Welt in C

```
#include <stdio.h>

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

- Übersetzen und Ausführen (auf einem UNIX-System)

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

Gar nicht so  
schwer :-)

## Das erste C-Programm – Vergleich mit Java

### Das berühmteste Programm der Welt in C

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

### Das berühmteste Programm der Welt in Java

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

03-04-ErsteSchritte: 2017-05-03



© kls GSPiC (Teil B, SS17) 3 Java versus C – Erste Beispiele | 3.1 Ausgabe

3-2

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

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

```
1 #include <avr/io.h>
2
3 void main() {
4     // initialize hardware: LED on port D pin 6, active low
5     DDRD |= (1<<6); // PD6 is used as output
6     PORTD |= (1<<6); // PD6: high --> LED is off
7
8     // greet user
9     PORTD &= ~(1<<6); // PD6: low --> LED is on
10
11    // wait forever
12    while(1){
13    }
}
```

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

03-04-ErsteSchritte: 2017-05-03

### Übersetzen und **Flashen** (mit Atmel Studio)

~ Übung

### Ausführen (SPiCboard):



(rote LED leuchtet)



© kls GSPiC (Teil B, SS17) 3 Java versus C – Erste Beispiele | 3.1 Ausgabe

3-4

## Das erste C-Programm – Erläuterungen

[Handout]

### C-Version zeilenweise erläutert

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

### Java-Version zeilenweise erläutert

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

© kls GSPiC (Teil B, SS17) 3 Java versus C – Erste Beispiele | 3.1 Ausgabe

3-3

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

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

```
1 #include <avr/io.h>
2
3 void main() {
4     // initialize hardware: LED on port D pin 6, active low
5     DDRD |= (1<<6); // PD6 is used as output
6     PORTD |= (1<<6); // PD6: high --> LED is off
7
8     // greet user
9     PORTD &= ~(1<<6); // PD6: low --> LED is on
10
11    // wait forever
12    while(1){
13    }
}
```

03-04-ErsteSchritte: 2017-05-03

© kls GSPiC (Teil B, SS17) 3 Java versus C – Erste Beispiele | 3.1 Ausgabe

3-5

- 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.

03-04-ErtesSchritte: 2017-05-03



## 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 6, active low
9     DDRD |= (1<<6); // PD6 is used as output
10    PORTD |= (1<<6); // PD6: high --> LED is off
11
12    // wait until PD2 -> low (button is pressed)
13    while(PIND & (1<<2))
14        ;
15
16    // greet user
17    PORTD &= ~(1<<6); // PD6: low --> LED is on
18
19    // wait forever
20    while(1)
21        ;
22 }
```

03-04-ErtesSchritte: 2017-05-03



- 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;
}
```

03-04-ErtesSchritte: 2017-05-03

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.

## Warten auf Tasterdruck – Erläuterungen

- 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 **Vcc** anliegt ↪ **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 ↪ Bit 2 im Register **PIND** wird **low** und die Schleife verlassen.

03-04-ErtesSchritte: 2017-05-03

## 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 }
```

03-04-ErsteSchritte: 2017-05-03

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



## 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), ...

03-04-ErsteSchritte: 2017-05-03



## 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 Interaktionsergebnisse 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.

03-04-ErsteSchritte: 2017-05-03



## 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**

03-04-ErsteSchritte: 2017-05-03



## 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, ...
  - **Problemanahes** 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**

03-04-ErsteSchnitte:2017-05-03



© kls GSPiC (Teil B, SS17) 3 Java versus C – Erste Beispiele | 3.3 Erstes Fazit

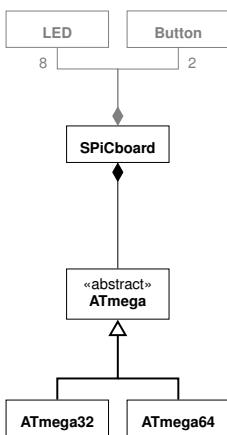
3-14

## Abstraktion durch Softwareschichten: SPIBoard

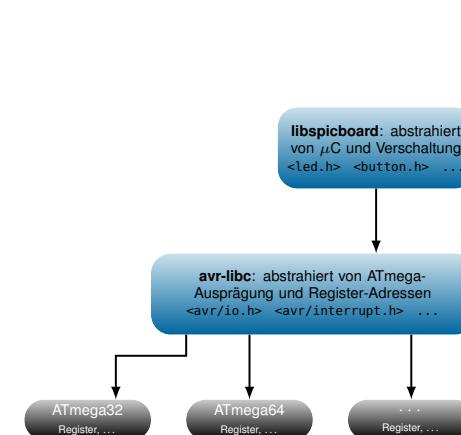
03-04-ErsteSchnitte:2017-05-03



### Hardwaresicht



### Softwareschichten



03-04-ErsteSchnitte:2017-05-03



4 Softwareschichten und Abstraktion | 4.1 Funktionsbibliotheken

4-1

## Ein erstes Fazit: µ-Controller-Programmierung

### C → Maschinennähe → 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 Hardweregistern
  - Detailliertes Wissen über die elektrische Verschaltung erforderlich
  - Keine Unterstützung durch Betriebssystem (wie etwa Linux)
  - Allgemein geringes Abstraktionsniveau → fehleranfällig, aufwändig

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

03-04-ErsteSchnitte:2017-05-03



© kls GSPiC (Teil B, SS17) 3 Java versus C – Erste Beispiele | 3.3 Erstes Fazit

3-15

## Abstraktion durch Softwareschichten: LED → on im Vergleich

Programm läuft nur auf dem **SPIBoard**. Es verwendet Funktionen (wie `sb.led.on()`) und Konstanten (wie `REDO`) der **libspicboard**, welche die konkrete Verschaltung von LEDs, Tastern, usw. mit dem µC repräsentieren:

```
#include <led.h>
...
sb.led.on(REDO);
```

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 `REDO` auf SPIBoard an:



**SPIBoard-Programm**

**ATmega-Programm**

**libspicboard:** abstrahiert von µC und Verschaltung



© kls GSPiC (Teil B, SS17)

4 Softwareschichten und Abstraktion | 4.1 Funktionsbibliotheken

4-2

## Abstraktion durch Softwareschichten: Vollständiges Beispiel

**Bisher:** Entwicklung mit avr-libc

```
#include <avr/io.h>
void main(void) {
    // initialize hardware

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

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

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

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

03-04-ErsteSchritte: 2017-05-03



© kls GSPiC (Teil B, SS17) 4 Softwareschichten und Abstraktion | 4.1 Funktionsbibliotheken

**Nun:**

Entwicklung mit libspicboard

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

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

    // greet user
    sb_led_on(RED0);

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

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

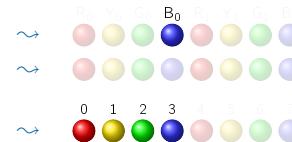
(vgl. ↪ 3-8)

## 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)`



↪ 47

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

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

↪ 47

### Eingabe-Abstraktionen (Auswahl)

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

- Button-Zustand abfragen:  
↳ `sb_button_getState(BUTTON0)`

↪ `BUTTONSTATE_{PRESSED, RELEASED}`

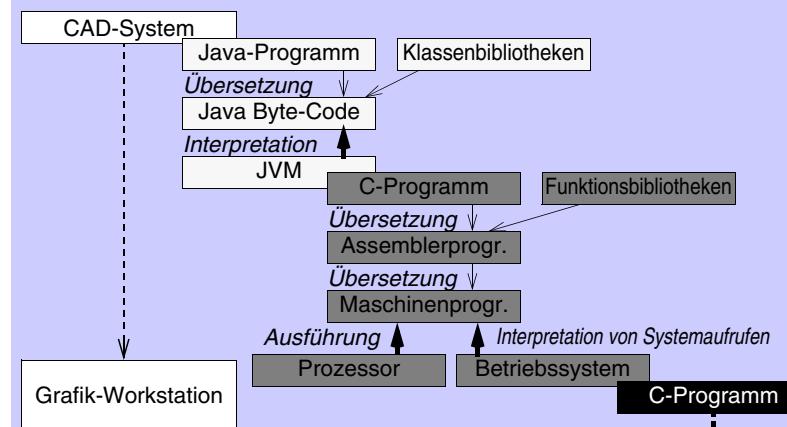
#### ■ ADC-Modul (#include <adc.h>)

- Potentiometer-Stellwert abfragen:  
↳ `sb_adc_read(POTI)`

↪  $\{0 \dots 1023\}$

## Softwareschichten im Allgemeinen

**Diskrepanz:** Anwendungsproblem ↔ Abläufe auf der Hardware



**Ziel:** Ausführbarer Maschinencode

03-04-ErsteSchritte: 2017-05-03



© kls GSPiC (Teil B, SS17) 4 Softwareschichten und Abstraktion | 4.2 Allgemein

03-04-ErsteSchritte: 2017-05-03

## 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-/Auszug 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

© kls GSPiC (Teil B, SS17) 4 Softwareschichten und Abstraktion | 4.2 Allgemein

4-6

## Die Rolle des Betriebssystems (Forts.)

- **Systemseicht:** Softwareschicht zum Multiplexen der Hardware (↔ Mehrbenutzerbetrieb)
  - Parallele Abarbeitung von Programminstanzen durch **Prozesskonzept**
    - Virtueller Speicher ↔ eigener 32-/64-Bit-Adressraum
    - Virtueller Prozessor ↔ wird transparent zugewiesen 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**.

03-04-Ersteschnitt:2017-05-03



© kls GSPiC (Teil B, SS 17) 4 Softwareschichten und Abstraktion | 4.2 Allgemein

4-7

## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

V\_GSPiC\_handout



## 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.
```

**SPiBoard:** Division durch 0

```
1 #include <7seg.h>
2 #include <avr/interrupt.h>
3
4 void main() {
5     int a = 23;
6     int b = 0;
7     sei();
8     b = 4711 / (a-23);
9     sb_7seg_showNumber(b);
10
11    while(1){}
12 }
```

Ausführen ergibt:



~ Programm setzt Berechnung fort mit falschen Daten.

© kls GSPiC (Teil B, SS 17) 4 Softwareschichten und Abstraktion | 4.2 Allgemein

4-8

## 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

© kls GSPiC (Teil B, SS 17) 5 Sprachüberblick | 5.1 Grundstruktur eines C-Programms

5-1

05-Sprachüberblick:2017-02-16



## Struktur eines C-Programms – am Beispiel

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

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

05-Sprachüberblick: 2017-02-16



© kls GSPiC (Teil B, SS17)

5 Sprachüberblick | 5.1 Grundstruktur eines C-Programms

5-2

## Schlüsselwörter

[≈Java]

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

- **Reservierte Wörter** der Sprache  
(≈ dürfen nicht als Bezeichner verwendet werden)
  - Eingegebene (*primitive*) Datentypen
  - Typmodifizierer
  - Kontrollstrukturen
  - Elementaranweisungen

05-Sprachüberblick: 2017-02-16



© kls GSPiC (Teil B, SS17)

5 Sprachüberblick | 5.2 Syntaktische Grundelemente

5-4

## Bezeichner

[=Java]

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

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

05-Sprachüberblick: 2017-02-16



© kls GSPiC (Teil B, SS17)

5 Sprachüberblick | 5.2 Syntaktische Grundelemente

5-3

## 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**

05-Sprachüberblick: 2017-02-16



© kls GSPiC (Teil B, SS17)

5 Sprachüberblick | 5.2 Syntaktische Grundelemente

5-5

## Literele

## [=Java]

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

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

### (Darstellung von) Konstanten im Quelltext

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



## Anweisungen

## [=Java]

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

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

### Beschreiben den eigentlichen Ablauf des Programms

- Werden hierarchisch komponiert aus drei Grundformen
  - Einzelanweisung – **Ausdruck** gefolgt von **;**
    - einzelnes Semikolon → leere Anweisung
  - **Block** – Sequenz von Anweisungen, geklammert durch **{...}**
  - **Kontrollstruktur**, gefolgt von Anweisung



## Ausdrücke

## [=Java]

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

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

### Gültige Kombination von Operatoren, Literalen und Bezeichnern

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



## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor



- **Datentyp** := (*<Menge von Werten>*, *<Menge von Operationen>*)
  - **Literal** Wert im Quelltext  $\leftrightarrow$  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“ ( $\sim$  int schlechter Stil!)



## Integertypen

[≈Java] [→ GDI, 03-2]

- **Integertyp** Verwendung Literalformen
 

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



## 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} ( $\leftrightarrow$  letztlich ein Integertyp)
  - Bedingungsausdrücke (z. B. **if(...)**) sind in C vom Typ **int!** [≠Java]



## Integertypen: Größe und Wertebereich

[≠Java]

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

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



Hier zeigt sich die C-Philosophie: Effizienz durch **Maschinennähe**  $\rightarrow$  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: `intn_t` und `uintn_t` für  $n \in \{8, 16, 32, 64\}$
  - Wird vom Compiler-Hersteller bereitgestellt

Wertebereich `stdint.h`-Typen

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

06-Datentypen: 2017-05-03



© kls GSPiC (Teil B, SS17)

6 Einfache Datentypen | 6.2 Ganzahltypen: int und Co

6-5

## 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!

06-Datentypen: 2017-05-03



© kls GSPiC (Teil B, SS17)

6 Einfache Datentypen | 6.3 Typ-Pseudonyme: `typedef`

6-7

## 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
...
```

06-Datentypen: 2017-05-03



© kls GSPiC (Teil B, SS17)

6 Einfache Datentypen | 6.3 Typ-Pseudonyme: `typedef`

6-6

## Aufzählungstypen mit `enum`

[≈ Java]

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

```
enum Bezeichneropt { KonstantenListe } ;
```

- Beispiel

- Definition:

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

- Verwendung:

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

- Vereinfachung der Verwendung durch `typedef`

- Definition:

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

- Verwendung:

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

06-Datentypen: 2017-05-03



© kls GSPiC (Teil B, SS17)

6 Einfache Datentypen | 6.4 Aufzählungstypen: `enum`

6-8

## enum $\mapsto$ int

[ $\neq$ Java]

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

```
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!
```

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

Das entspricht der  
C-Philosophie!  $\leadsto$  [3-14]

© kls GSPiC (Teil B, SS17)

6 Einfache Datentypen | 6.4 Aufzählungstypen: enum

6-9

## Zeichen $\mapsto$ Integer

[ $\approx$ Java]

- Zeichen sind in C ebenfalls Ganzzahlen (Integers)  $\leadsto$  [6-3]
  - char gehört zu den Integer-Typen (üblicherweise 8 Bit = 1 Byte)

- Repräsentation erfolgt durch den ASCII-Code  $\leadsto$  [6-12]

- 7-Bit-Code  $\mapsto$  128 Zeichen standardisiert  
(die verbleibenden 128 Zeichen werden unterschiedlich interpretiert)

- Spezielle Literalform durch Hochkommata

'A'  $\mapsto$  ASCII-Code von A

- Nichtdruckbare Zeichen durch Escape-Sequenzen

- Tabulator 't'

- Zeilentrenner 'n'

- Backslash '\'

- Zeichen  $\mapsto$  Integer  $\leadsto$  man kann mit Zeichen rechnen

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

© kls GSPiC (Teil B, SS17)

6 Einfache Datentypen | 6.6 Zeichen und Zeichenketten

6-11

## Fließkommatypen

[ $\approx$ Java]

- Fließkommatyp Verwendung Literalformen

■ <b>float</b>	einfache Genauigkeit ( $\approx$ 7 St.)	100.0F, 1.0E2F
■ <b>double</b>	doppelte Genauigkeit ( $\approx$ 15 St.)	100.0, 1.0E2
■ <b>long double</b>	„erweiterte Genauigkeit“	100.0L 1.0E2L

- Genauigkeit / Wertebereich sind **implementierungsabhängig** [ $\neq$ Java]

- Es gilt: **float  $\leq$  double  $\leq$  long double**

„Effizienz durch  
Maschinennähe“  $\leadsto$  [3-14]

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

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

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

© kls GSPiC (Teil B, SS17) 6 Einfache Datentypen | 6.5 Fließkommazahltypen: float und double 6-10

## ASCII-Code-Tabelle (7 Bit)

ASCII  $\mapsto$  American Standard Code for Information Interchange

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

© kls GSPiC (Teil B, SS17) 6 Einfache Datentypen | 6.6 Zeichen und Zeichenketten

6-12

- Ein String ist in C ein Feld (Array) von Zeichen
  - Repräsentation: Folge von Einzelzeichen, terminiert durch (letztes Zeichen): **NUL** (ASCII-Wert 0)
  - Speicherbedarf: (Länge + 1) Bytes
- Spezielle Literalform durch doppelte Hochkommata:  
"Hi!" → 

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

 ← abschließendes 0-Byte
- Beispiel (Linux)
 

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

06-Datentypen: 2017-05-03

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



## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

V\_GSPiC\_handout



- 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

06-Datentypen: 2017-05-03



## Arithmetische Operatoren

- Stehen für alle Ganzzahl- und Fließkommatypen zur Verfügung
 

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
unäres -	negatives Vorzeichen (z. B. $-a$ ) ↪ Multiplikation mit $-1$
unäres +	positives Vorzeichen (z. B. $+3$ ) ↪ kein Effekt
- Zusätzlich nur für Ganzzahltypen:
 

%	Modulo (Rest bei Division)
---	----------------------------

07-Operatoren: 2017-03-24



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

- Stehen für Ganzzahltypen und Zeigertypen zur Verfügung

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

- Linksseitiger Operator (Präfix)       $++x$  bzw.  $--x$

- Erst wird der Inhalt von  $x$  verändert
  - Dann wird der (neue) Inhalt von  $x$  als Ergebnis geliefert

- Rechtsseitiger Operator (Postfix)       $x++$  bzw.  $x--$

- Erst wird der (alte) Inhalt von  $x$  als Ergebnis geliefert
  - Dann wird der Inhalt von  $x$  verändert

- Beispiele

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

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.1 Arithmetische Operatoren

7-2

## Logische Operatoren [=Java][→ GDI, 03-10]

- Verknüpfung von Wahrheitswerten (wahr / falsch), kommutativ

„und“ (Konjunktion)	wahr $\&&$ wahr $\rightarrow$ wahr
	wahr $\&&$ falsch $\rightarrow$ falsch
	falsch $\&&$ falsch $\rightarrow$ falsch

„oder“ (Disjunktion)	wahr $\ $ wahr $\rightarrow$ wahr
	wahr $\ $ falsch $\rightarrow$ wahr
	falsch $\ $ falsch $\rightarrow$ falsch

„nicht“ (Negation, unär)	! wahr $\rightarrow$ falsch
	! falsch $\rightarrow$ wahr

- Beachte: Operanden und Ergebnis sind vom Typ **int**      [=Java]

- Operanden      0  $\rightarrow$  falsch

(Eingangsparameter):  $\neq 0 \rightarrow$  wahr

- Ergebnis:      falsch  $\rightarrow$  0

wahr  $\rightarrow$  1

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.2 Relationale Operatoren

7-4

## Vergleichsoperatoren [=Java][→ GDI, 03-09]

- Vergleichen von zwei Ausdrücken

<	kleiner
$\leq$	kleiner gleich
>	größer
$\geq$	größer gleich
$\equiv$	gleich (zwei Gleichheitszeichen!)
$\neq$	ungleich

- Beachte: Ergebnis ist vom Typ **int**      [=Java]

- Ergebnis: falsch  $\leftrightarrow$  0  
wahr  $\leftrightarrow$  1

- Man kann mit dem Ergebnis rechnen

- Beispiele

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

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.2 Relationale Operatoren

7-3

## 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;**

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

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

- Kann **überraschend** sein, wenn Teilausdrücke **Nebeneffekte** haben

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

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.2 Relationale Operatoren

7-5

## 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  $op \in \{ +, -, \star, \%, <<, >>, \&, ^, | \}$
- Beispiele

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

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17)

7 Operatoren und Ausdrücke | 7.3 Zuweisungsoperatoren

7-6

## Bitoperationen

### [=Java]

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

1	$\wedge$	1	→	0
1	$\wedge$	0	→	1
0	$\wedge$	0	→	0
  - $\sim$  bitweise Inversion  
(Einerkomplement, unär)  

$\sim$	1	→	0
$\sim$	0	→	1

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17)

7 Operatoren und Ausdrücke | 7.4 Bitoperationen

7-8

## Zuweisungen sind Ausdrücke!

- Zuweisungen können in komplexere Ausdrücke geschachtelt werden
  - Das Ergebnis eines Zuweisungsausdrucks ist der zugewiesene Wert
- ```
int a, b, c;  
a = b = c = 1; // c: 1, b: 1, a: 1
```
- Die Verwendung von Zuweisungen in beliebigen Ausdrücken führt zu **Nebeneffekten**, die nicht immer offensichtlich sind
- ```
a += b += c; // Value of a and b?
```

### Besonders gefährlich: Verwendung von = statt ==

In C sind Wahrheitswerte Integers: 0  $\mapsto$  falsch, 1  $\mapsto$  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!

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17)

7 Operatoren und Ausdrücke | 7.3 Zuweisungsoperatoren

7-7

## Bitoperationen (Forts.)

- 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
$\sim x$	0	1	1	0	0	0	1	1
7	0	0	0	0	0	1	1	1
$x   7$	1	0	0	1	1	1	1	1
$x & 7$	0	0	0	0	0	1	0	0
$x ^ 7$	1	0	0	1	1	0	1	1
$x << 2$	0	1	1	1	0	0	0	0
$x >> 1$	0	1	0	0	1	1	1	0

0x9c 0x63 0x07 0x9f 0x04 0x9b 0x70 0x4e

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17)

7 Operatoren und Ausdrücke | 7.4 Bitoperationen

7-9

## Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

Bit# 7 6 5 4 3 2 1 0  
PORTD [?] [?] [?] [?] [?] [?] [?] [?]

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

0x80 [1 0 0 0 0 0 0 0]  
PORTD |= 0x80 [1 ? ? ? ? ? ? ?]

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

~0x80 [0 1 1 1 1 1 1 1]  
PORTD &= ~0x80 [0 ? ? ? ? ? ? ?]

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

0x08 [0 0 0 0 1 0 0 0]  
PORTD ^= 0x08 [? ? ? ? ? ? ? ?]

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

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.4 Bitoperationen

7-10

## Bedingte Auswertung

[≈Java] [→ GDI, 07-10]

- Formulierung von Bedingungen in Ausdrücken

*Ausdruck<sub>1</sub> ? Ausdruck<sub>2</sub> : Ausdruck<sub>3</sub>*

- Zunächst wird *Ausdruck<sub>1</sub>* ausgewertet

- *Ausdruck<sub>1</sub> ≠ 0* (wahr) ↪ Ergebnis ist *Ausdruck<sub>2</sub>*
- *Ausdruck<sub>1</sub> = 0* (falsch) ↪ Ergebnis ist *Ausdruck<sub>3</sub>*

- ?: 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;
}
```

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.5 Weitere Operatoren

7-12

## 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) ;
}
```



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.4 Bitoperationen

7-11

## Sequenzoperator

[≠Java]

- Reihung von Ausdrücken

*Ausdruck<sub>1</sub> , Ausdruck<sub>2</sub>*

- Zunächst wird *Ausdruck<sub>1</sub>* ausgewertet

↪ Nebeneffekte von *Ausdruck<sub>1</sub>* werden sichtbar

- Ergebnis ist der Wert von *Ausdruck<sub>2</sub>*

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

07-Operatoren: 2017-03-24



© kls GSPiC (Teil B, SS17) 7 Operatoren und Ausdrücke | 7.5 Weitere Operatoren

7-13

Klasse	Operatoren	Assoziativität
1 Funktionsaufruf, Feldzugriff Strukturzugriff Post-Inkrement/-Dekrement	<code>x() x[]</code> <code>x.y x-&gt;y</code> <code>x++ x--</code>	links → rechts
2 Prä-Inkrement/-Dekrement unäre Operatoren Adresse, Verweis (Zeiger) Typkonvertierung (cast) Typgröße	<code>++x --x</code> <code>+x -x ~x !x</code> <code>&amp; *</code> <code>(&lt;Typ&gt;)x</code> <code>sizeof(x)</code>	rechts → links
3 Multiplikation, Division, Modulo	<code>* / %</code>	links → rechts
4 Addition, Subtraktion	<code>+ -</code>	links → rechts
5 Bitweises Schieben	<code>&gt;&gt; &lt;&lt;</code>	links → rechts
6 Relationaloperatoren	<code>&lt; &lt;= &gt; &gt;=</code>	links → rechts
7 Gleichheitsoperatoren	<code>== !=</code>	links → rechts
8 Bitweises UND	<code>&amp;</code>	links → rechts
9 Bitweises OR	<code> </code>	links → rechts
10 Bitweises XOR	<code>^</code>	links → rechts
11 Konjunktion	<code>&amp;&amp;</code>	links → rechts
12 Disjunktion	<code>  </code>	links → rechts
13 Bedingte Auswertung	<code>? :=</code>	rechts → links
14 Zuweisung	<code>= op=</code>	rechts → links
15 Sequenz	<code>,</code>	links → rechts



## Typumwandlung in Ausdrücken (Forts.)

- 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
int: 100
int: 3
int: 300
int32_t: 300
int32_t: 75
```



## Typumwandlung in Ausdrücken

- Eine Operation 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: 100
int: 3
int: 4
int: 300
int: 75
```



## Typumwandlung in Ausdrücken (Forts.)

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

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

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

int8_t: 75
int: 100
int: 3
double: 4.0
double: 300.0
double: 75.0
```



## Typumwandlung in Ausdrücken (Forts.)

- `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   s   < u;      unsigned: 65535
int: 0
```

~ Überraschende Ergebnisse bei negativen Werten!

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

07-Operatoren: 2017-03-24



## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

### 8 Kontrollstrukturen

### 9 Funktionen

### 10 Variablen

### 11 Präprozessor

V\_GSPiC\_handout



## 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   s   < int: 1 u;      int: 1
```

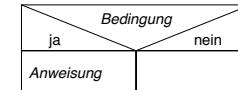
07-Operatoren: 2017-03-24



## Bedingte Anweisung [=Java][→ GDI, 07-02]

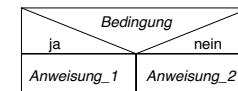
- `if`-Anweisung (bedingte Anweisung)

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



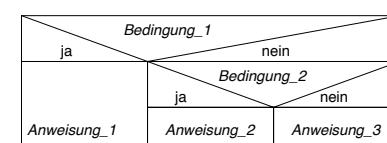
- `if-else`-Anweisung (einfache Verzweigung)

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



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

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



08-Kontrollstrukturen: 2017-05-03





## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

### 9 Funktionen

### 10 Variablen

### 11 Präprozessor

v\_GSPiC\_handout



## Beispiel

### Funktion (Abstraktion) `sb_led_set_all_leds()`

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



### Implementierung in der libspicboard

```
void sb_led_set_all_leds(uint8_t setting)  Sichtbar: Bezeichner und
{                                            formale Parameter
    uint8_t i = 0;
    for (i = 0; i < 8; i++) {
        if (setting & (1<<i)) {
            sb_led_on(i);
        } else {
            sb_led_off(i);
        }
    }
}
```

Unsichtbar: Tatsächliche Implementierung

09-Funktionen: 2017-05-03



## Was ist eine Funktion?

### ■ **Funktion** := Unterprogramm [↔ GDI, 11-01]

- Programmstück (Block) mit einem **Bezeichner**
- Beim Aufruf können **Parameter** übergeben werden
- Bei Rückkehr kann ein **Rückgabewert** zurückgeliefert werden

### ■ Funktionen sind elementare Programmbausteine

- Gliedern umfangreiche Aufgaben in kleine, beherrschbare Komponenten
- Ermöglichen die einfache Wiederverwendung von Komponenten
- Ermöglichen den einfachen Austausch von Komponenten
- Verbergen Implementierungsdetails (**Black-Box-Prinzip**)

### Funktion → Abstraktion

↔ 4-1

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

09-Funktionen: 2017-05-03



## Funktionsdefinition

[≈ Java]

### ■ Syntax: `Typ Bezeichner ( FormaleParamopt ) {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<sub>opt</sub>** Liste der formalen Parameter:  $Typ_1 Bez_{1, opt}, \dots, Typ_n Bez_{n, opt}$  (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++) {
    }
}
```

09-Funktionen: 2017-05-03



## Funktionsaufruf

[=Java]

### Syntax: Bezeichner ( TatParam )

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

### Beispiele:

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

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

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

**Fehler:** text ist nicht int-konvertierbar (tatsächlicher Parameter 2 passt nicht zu formalem Parameter b → 9-3)

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

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

09-Funktionen: 2017-05-03



© kls GSPiC (Teil B, SS17) 9 Funktionen | 9.3 Aufruf

9-4

## 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!**

09-Funktionen: 2017-05-03



© kls GSPiC (Teil B, SS17) 9 Funktionen | 9.3 Aufruf

09-Funktionen: 2017-05-03



9-6

## Funktionsaufruf – Parameterübergabe

[≠Java]

### Generelle Arten der Parameterübergabe

[→ GDI, 14-01]

#### ■ Call-by-value

Die formalen Parameter sind Kopien der tatsächlichen Parameter. Änderungen in den formalen Parametern gehen mit Verlassen der Funktion verloren. **Dies ist der Normalfall in C.**

#### ■ Call-by-reference

Die formalen Parameter sind Verweise (Referenzen) auf die tatsächlichen Parameter. Änderungen in den formalen Parametern betreffen auch die tatsächlichen Parameter.

**In C nur indirekt über Zeiger möglich.** → 13-5

### Des weiteren gilt

- Arrays werden in C immer *by-reference* übergeben

[=Java]

- Die Auswertungsreihenfolge der Parameter ist **undefiniert!**

[≠Java]



© kls GSPiC (Teil B, SS17) 9 Funktionen | 9.3 Aufruf

9-5

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

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

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

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



© kls GSPiC (Teil B, SS17) 9 Funktionen | 9.4 Deklaration

9-7

- 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!

09-Funktionen: 2017-05-03



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

**Beispiel:**

```
#include <stdio.h>

void foo(); // "open" declaration

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

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

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

09-Funktionen: 2017-05-03



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

**Beispiel:**

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

- Funktion foo() ist nicht **deklariert** → der Compiler **warnt**, aber akzeptiert beliebige tatsächliche Parameter
- foo() ist **definiert** mit den formalen Parametern (int, int). Was immer an tatsächlichen Parametern übergeben wurde, wird entsprechend interpretiert!
- Was wird hier ausgegeben?**

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

**Achtung: Verwechslungsgefahr**

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

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

## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

### 10 Variablen

### 11 Präprozessor



## 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

## Variablendefinition

[≈Java]

- **Variable** := Behälter für Werte (→ Speicherplatz)
- Syntax (Variablendefinition):  
 $SK_{opt} \ Typ_{opt} \ Bez_1 [= Ausdr_1]_{opt} \ [, \ Bez_2 [= Ausdr_2]_{opt} \ , \dots]_{opt}$ 
  - $SK_{opt}$  Speicherklasse der Variable, `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]



## Überblick: Teil B Einführung in C

3 Java versus C – Erste Beispiele

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor



