Systemnahe Programmierung in C

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Skriptum_handout

Referenzen

[1] *ATmega328PB 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. Atmel Corporation. Okt. 2015.

[GDI] Frank Bauer. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2015 (jährlich). URL: `https://gdi.cs.fau.de/w15/material`.

[2] Joachim Goll und Manfred Dausmann. *C als erste Programmiersprache*. (Als E-Book aus dem Uninetz verfügbar). Springer Vieweg, 2014. ISBN: 978-3-8348-2271-0. URL: `https://link.springer.com/book/10.1007/978-3-8348-2271-0`.

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

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

[5] Dennis MacAlistair Ritchie und Ken Thompson. "The Unix Time-Sharing System". In: *Communications of the ACM* 17.7 (Juli 1974), S. 365–370. DOI: `10.1145/361011.361061`.

[6] David Tennenhouse. "Proactive Computing". In: *Communications of the ACM* (Mai 2000), S. 43–45.

Referenzen (Forts.)

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

Veranstaltungsüberblick

Teil A: Konzept und Organisation

1 Einführung

2 Organisation

Teil B: Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Teil C: Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Veranstaltungsüberblick (2)

Teil D: Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Teil E: Speicher

33 Dynamische Speicherallokation

34 Speicherorganisation

35 Speicherorganisation – Stack

Systemnahe Programmierung in C

Teil A Konzept und Organisation

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Skriptum_handout

Überblick: Teil A Konzept und Organisation

1 Einführung
- Ziele der Lehrveranstaltung
- Warum μ-Controller?
- Warum C?
- Literatur

2 Organisation
- Vorlesung
- Übungen
- Lötabend
- Prüfung

Einleitung

- **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) und eine Betriebssystem-Plattform (Linux)
  - SPiCboard-Lehrentwicklungsplattform mit ATmega-μC
  - Praktische Erfahrungen in hardware- und systemnaher Softwareentwicklung machen
- **Verstehen** der technologischen Sprach- und Hardwaregrundlagen für die Entwicklung systemnaher Software
  - Die Sprache C verstehen und einschätzen können
  - Umgang mit Nebenläufigkeit und Hardwarenähe
  - Umgang mit den Abstraktionen eines Betriebssystems (Dateien, Prozesse, ...)

Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren werden in **eingebetteten Systemen** verbaut [6]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und μ-Controller, **8-Bit oder kleiner** [6, 7]

Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren werden in **eingebetteten Systemen** 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`)

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

Type	Flash	SRAM	IO	Timer 8/16	UART	SPI	ADC	PWM	EUR
ATTINY13	1 KiB	64 B	6	1/-	-	-	1*4	-	**2,20**
ATTINY2313	2 KiB	128 B	18	1/1	-	1	-	-	**2,99**
ATMEGA48	4 KiB	512 B	23	2/1	1	1	8*10	6	**2,40**
ATMEGA16	16 KiB	1024 B	32	2/1	1	1	8*10	4	**6,40**
ATMEGA32	32 KiB	2048 B	32	2/1	1	1	8*10	4	**5,40**
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	8	**–**
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	8	**19,80**
ATMEGA256	256 KiB	8192 B	86	2/2	4	1	16*10	16	**15,50**

ATmega-Varianten (Auswahl) und Handelspreise (Reichelt Elektronik, April 2023)

- 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: SPiC – 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

- **Ansatz**
  - Konzentration auf zwei Domänen
    - μ-Controller-Programmierung
    - Softwareentwicklung für die Linux-Systemschnittstelle
  - Gegensatz μC-Umgebung $\leftrightarrow$ Betriebssystemplattform erfahren
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (EEI, ME, ...)

Motivation: SPiC

Jede(r) soll am Ende der Veranstaltung abschätzen können,

- was ein μ-Controller (nicht) kann,
- wie aufwändig es ist, ihn zu programmieren,
- was ein Betriebssystem (nicht) bietet,
- wie aufwändig es ist, es zu nutzen.

Jede(r) soll in der Lage sein, ggf. mit einem Informatiker zusammenzuarbeiten...

Vorlesungsskript

- Das Handout der Vorlesungsfolien wird online zur Verfügung gestellt
  - Kapitel einzeln oder als gesamtes Skriptum verfügbar
  - Handout enthält (in geringem Umfang) zusätzliche Informationen

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

Literaturempfehlungen

[2] Für den Einstieg empfohlen:

Joachim Goll und Manfred Dausmann. *C als erste Programmiersprache*. (Als E-Book aus dem Uninetz verfügbar). Springer Vieweg, 2014. ISBN: 978-3-8348-2271-0. URL: `https://link.springer.com/book/10.1007/978-3-8348-2271-0`

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

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

Überblick: Teil A Konzept und Organisation

1 Einführung

2 Organisation

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
  - Softwareentwicklung auf „einem Betriebssystem“ (Linux)
    - Betriebssystem als Ausführungsumgebung für Programme
    - Abstraktionen und Dienste eines Betriebssystems

Vorlesung

- 36 Themenabschnitte
  - Foliensätze auf WWW-Server
  - Reihenfolge/Termin aus dem Semesterplan ersichtlich
  - → Voraussetzung für erfolgreiche Bearbeitung der Übungsaufgaben
- Fragen zur Vorlesung
  - am besten **sofort** stellen
  - im StudOn Thread *Fragen zur Vorlesung*
  - Beantwortung im Forum oder während der wöchentlichen Vorlesung/Tafelübung
- Ende des Semesters Klausurfragestunde
- **Tafel- und Rechnerübungen sind keine Ersatzvorlesungen!**

Vorlesung

- Screencasts aus SS20
- Vorlesungsaufzeichnung aus SS19

existieren...

siehe

- WWW-Seite
- Video-Portal Uni

...sind aber z.T. (etwas) veraltet!

Referenz ist die aktuelle Präsenzvorlesung!

Übungen

- Tafelübung und Rechnerübung
  - Tafelübungen
    - Ausgabe und Erläuterung der Programmieraufgaben
    - Gemeinsame Entwicklung einer Lösungsskizze
    - Besprechung der Lösungen
  - Rechnerübungen
    - selbstständige Programmierung
    - Umgang mit Entwicklungwerkzeug
    - Betreuung durch Übungsbetreuer
- Termin: 8 + 1 Gruppen zur Auswahl
  - Anmeldung über Waffel ab Donnerstag 18.04.2024 um 18:00 Uhr (siehe Webseite)
  - Keine Unterscheidung zwischen SPiC / GSPiC / InfoEEI
  - **Separate** Übung für SLP

Zur Übungsteilnahme wird ein gültiger Login im Linux-CIP gebraucht!

Übungen

WARNING!

Im WS wird es **keine** Übungen für Wiederholer mehr geben!

WARNING!

Programmieraufgaben

- Praktische Umsetzung des Vorlesungsstoffs
  - Acht Programmieraufgaben ↪ ??
  - Teilweise Gruppenabgaben
- Lösungen per SPiC-IDE 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; ↪ 2–10
es können jedoch bis zu **10% Bonuspunkte**
für die Prüfungsklausur erarbeitet werden!

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

Unabhängig davon ist die Bearbeitung der Übungen **dringend empfohlen!**

Bestehensquote der Klausur

nach Aktivität der Teilnehmer bei den Übungsaufgaben

keine oder weniger als die Hälfte der Übungsaufgaben abgegeben

mindestens die Hälfte der Übungsaufgaben abgegeben

Übungsplattform: Das SPiCboard

- ATmega328-μC
- USB-Anschluss
- 8 LEDs
- 2 7-Seg-Elemente
- 2 Taster
- 1 Potentiometer
- 1 Fotosensor

  optional:
- OLED Display

- Ausleihe während Rechnerübung
- Oder noch besser ↪ selber Löten
- Alternativ: Entwicklung im Simulator

SPiCboard-Lötabend

- Die FSI EEI, FSI ME sowie das FabLab bieten einen „Lötabend“ für die Teilnehmer der Veranstaltung an
  - Teilnahme ist freiwillig
  - (Erste) Löterfahrung sammeln beim Löten eines eigenen SPiCboards
  - Voraussichtlich insgesamt 4 Termine (in KW 18/19)
- Anmeldung über Waffel **notwendig**, da begrenzte Plätze:
  ab Donnerstag 18.04.2024 um 18:00 Uhr (siehe Webseite)
- Kostenlos für Teilnehmer dieser Veranstaltung
  (finanziert aus Studienzuschüssen)

Der bei der Anmeldung gewählte Termin ist verbindlich!

Prüfung und Modulnote

- Prüfung (Klausur)
  - Termin: voraussichtlich Anfang August
  - Dauer: 60 min (GSPiC) bzw. 90 min (SPiC und InfoEEI)
  - Inhalt: Fragen zum Vorlesungsstoff + Programmieraufgabe(n)

- Klausurnote $\mapsto$ Modulnote
  - Bestehensgrenze (in der Regel): 50% der möglichen Klausurpunkte (KP)
  - Falls bestanden ist eine Notenverbesserung möglich durch Bonuspunkte aus den Programmieraufgaben
    - Basis (Minimum): 20% der möglichen Übungspunkte (ÜP)
    - Bonuspunkte werden gleichmäßig auf Intervall [20%;80%] der möglichen ÜP aufgeteilt

  $\rightsquigarrow$ 80%-100% der möglichen ÜP $\mapsto$ +10% der möglichen KP

Bei Fragen oder Problemen

- Vorlesungs- und Übungsfolien konsultieren
- Häufig gestellte Fragen (FAQ) und Antworten siehe Webseite
- Rechnerübungen
- StudOn Forum
  → `https://www.studon.fau.de/studon/goto.php?target=frm_5700999`
- Bei speziellen Fragen Mail an Mailingliste
  → alle Übungsleiter i4spic@lists.cs.fau.de (inhaltlich)
  → wiss. Mitarbeiter i4spic-orga@lists.cs.fau.de (organisatorisch)

Systemnahe Programmierung in C

Teil B Einführung in C

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Skriptum_handout

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Das erste C-Programm

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

```
#include <stdio.h>

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

- Übersetzen und Ausführen (auf einem UNIX-System)

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

Gar nicht so schwer :-)

Das erste C-Programm – Vergleich mit Java

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

```c
#include <stdio.h>

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

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

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

Das erste C-Programm – Erläuterungen [Handout]

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

Das erste C-Programm für einen μ-Controller

Vorbemerkung:

- DDRx: Data Direction Register
- PINx: Port Input Register
- PORTx: Port Ouput Register

(jeweils 8 Bit)

Das erste C-Programm für einen μ-Controller

Vorbemerkung:

- LED leuchtet nicht:
  - DDRD Bit 6: '1' (Output)
  - PORTD Bit 6: '1' (5V)
- LED leuchtet:
  - DDRD Bit 6: '1' (Output)
  - PORTD Bit 6: '0' (0V)

Das erste C-Programm für einen μ-Controller

- „Hello World“ für AVR ATmega (SPiCboard)

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

void main(void) {
  /* initialize hardware: LED on port D pin 6, active low */
  DDRD  |= (1<<6);  /* PD6 is used as output */
  PORTD |= (1<<6);  /* PD6: high --> LED is off */

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

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

μ-Controller-Programmierung ist „irgendwie anders“.

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

Das erste C-Programm für einen μ-Controller

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

```
1  #include <avr/io.h>
2
3  void main(void) {
4    /* initialize hardware: LED on port D pin 6, active low */
5    DDRD  |= (1<<6);  /* PD6 is used as output */
6    PORTD |= (1<<6);  /* PD6: high --> LED is off */
7
8    /* greet user */
9    PORTD &= ~(1<<6); /* PD6: low --> LED is on */
10
11   /* wait forever */
12   while (1) {
13   }
14 }
```

μ-Controller-Programm – Erläuterungen [Handout]

- μ-Controller-Programm zeilenweise erläutert
  (Beachte Unterschiede zur Linux-Version ↪ 3–3)

  1 Für den Zugriff auf Hardware-Register (`DDRD`, `PORTD`, bereitgestellt als **globale Variablen**) wird die **Funktionsbibliothek** `avr/io.h` mit `#include` eingebunden.

  3 Die `main()`-Funktion hat **keinen Rückgabewert** (Typ `void`). Ein μ-Controller-Programm läuft **endlos** ⇝ `main()` terminiert nie.

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

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

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

Das zweite C-Programm – Eingabe unter Linux

- Benutzerinteraktion (Lesen eines Zeichens) unter Linux:

```
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Press key: ");
  char 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.

Das zweite C-Programm für einen μ-Controller

Vorbemerkung:

- Initialisierung:
  - DDRD Bit 2: '0' (Input)
  - PORTD Bit 2: '1' (Pullup eingeschaltet)
- Erkennung:
  - PIND Bit 2: '1' => Taster nicht gedrückt
  - PIND Bit 2: '0' => Taster gedrückt

Das zweite C-Programm – Eingabe mit μ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

```
1  #include <avr/io.h>
2
3  void main(void) {
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 >> 2) & 1) {
14   }
15
16   /* greet user */
17   PORTD &= ~(1 << 6); /* PD6: low --> LED is on */
18
19   /* wait forever */
20   while (1) {
21   }
22 }
```

Warten auf Tasterdruck – Erläuterungen [Handout]

- Benutzerinteraktion mit SPiCboard zeilenweise erläutert
  - 5 Wie die LED ist der Taster mit einem **digitalen IO-Pin** des μ-Controllers verbunden. Hier konfigurieren wir Pin 2 von Port D als **Eingang** durch **Löschen** des entsprechenden Bits im Register `DDRD`.
  - 6 Durch **Setzen** von Bit 2 im Register `PORTD` wird der interne Pull-Up-Widerstand (hochohmig) aktiviert, über den V_{CC} anliegt $\rightsquigarrow$ 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

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

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

Benutzerinteraktion als Java-Programm – Erläuterungen [Handout]

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

- Benutzerinteraktion in Java zeilenweise erläutert

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

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

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

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

Ein erstes Fazit: Von Java → C (Syntax)

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

Ein erstes Fazit: Von Java → C (Idiomatik)

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

Ein erstes Fazit: Von Java → C (Philosophie)

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

Ein erstes Fazit: μ-Controller-Programmierung

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

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

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

Ansatz: Mehr Abstraktion durch problemorientierte Bibliotheken

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Abstraktion durch Softwareschichten: SPiCboard

Problemnähe ↑ — Maschinennähe ↓

Hardwaresicht

Softwareschichten

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

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

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

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

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

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

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

Ziel: Schalte LED `RED0` auf SPiCboard an:

Problemnähe ↑ — Maschinennähe ↓

04-Abstraktion: 2024-04-11

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 >> 2) & 1) {
  }

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

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

(vgl. ↪ 3–11)

Nun: Entwicklung mit libspicboard

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

void main(void) {

  /* wait until Button0 is pressed */
  while (sb_button_getState(BUTTON0)
                          != 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_led_on(RED0)`
  - Lese Bit 2 in `PORTD`
    ↦ `sb_button_getState(BUTTON0)`

Abstraktionen der libspicboard: Kurzüberblick

- Ausgabe-Abstraktionen (Auswahl)
  - LED-Modul (`#include <led.h>`)
    - LED einschalten: `sb_led_on(BLUE0)` ⇝
    - LED ausschalten: `sb_led_off(BLUE0)` ⇝
    - Alle LEDs ein-/ausschalten: `sb_led_setMask(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)` ↦ `BUTTONSTATE_{PRESSED,RELEASED}`
  - 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;
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(void) {
  int a = 23;
  int b;
  sei();
  b = 4711 / (a - 23);
  sb_7seg_showNumber(b);

  while (1) {}
}
```

Ausführen ergibt:

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

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Struktur eines C-Programms – allgemein

```
1  // include files
2  #include ...
3
4  // global variables
5  ... variable1 = ...
6
7  // subfunction 1
8  ... subfunction_1(...) {
9    // local variables
10   ... variable1 = ...
11   // statements
12   ...
13 }
14 // subfunction n
15 ... subfunction_n(...) {
16
17   ...
18
19 }
20
21 // main function
22 ... main(...) {
23
24   ...
25
26 }
```

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

Struktur eines C-Programms – am Beispiel

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

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(void) {
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(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, in der Ausdrücke abgearbeitet werden ↪ 7–14
    - Auswertungsreihenfolge kann mit Klammern ( ) explizit bestimmt werden
    - Der Compiler darf Teilausdrücke in möglichst effizienter Folge auswerten

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Was ist ein Datentyp?

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

Integertypen [≈Java]

- 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`	64	≥ 32	32	64	32
`long long`	-	≥ 64	64	64	64

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

Hier zeigt sich die C-Philosophie: Effizienz durch **Maschinennähe** ↪ 3–17

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

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

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

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

Typ-Aliase mit `typedef` [≠Java]

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

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

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

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

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

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

Typ-Aliase mit `typedef` (Forts.)

[≠ Java]

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

Definierte Bitbreiten sind bei der μC-Entwicklung sehr wichtig!

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

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

Aufzählungstypen mit enum [≈Java]

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

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

- Beispiel
  - Definition:

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

  - Verwendung:

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

- Vereinfachung der Verwendung durch `typedef`
  - Definition:

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

  - Verwendung:

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

enum ↦ int [≠ Java]

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

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

  - Es ist auch möglich, Werte direkt zuzuweisen

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

  - Man kann sie verwenden wie `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**! ↪ 3–17

Fließkommatypen [≈ Java]

- 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“ ↪ 3–17

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
- Spezielle Literalform durch doppelte Hochkommata:

`"Hi!"` ↦

'H'	'i'	'!'	0

← abschließendes 0-Byte

- Beispiel (Linux)

```
#include <stdio.h>

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

int main(void) {
  printf("%s", string);
  return 0;
}
```

Zeichenketten brauchen vergleichsweise viel Speicher und „größere“ Ausgabegeräte (z. B. LCD-Display).
⇝ Bei der 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

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

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, 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
```

Vergleichsoperatoren [=Java][↪ GDI, 03-09]

- Vergleichen von zwei Ausdrücken

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

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

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

Logische Operatoren [≈Java][↪ GDI, 03-10]

- Verknüpfung von Wahrheitswerten (wahr / falsch), kommutativ

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

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

Logische Operatoren – Auswertung [=Java]

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

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

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

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

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

- Kann überraschend sein, wenn Teilausdrücke Nebeneffekte haben

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

Zuweisungsoperatoren [= Java]

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

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

Zuweisungen sind Ausdrücke!

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

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

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

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

Besonders gefährlich: Verwendung von = statt ==

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

- Typischer „Anfängerfehler“ in Kontrollstrukturen:

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

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

Bitoperationen [= Java]

- Bitweise Verknüpfung von Ganzzahltypen, kommutativ

&	bitweises „Und“ (Bit-Schnittmenge)	$1 \,\&\, 1 \rightarrow 1$ $1 \,\&\, 0 \rightarrow 0$ $0 \,\&\, 0 \rightarrow 0$
|	bitweises „Oder“ (Bit-Vereinigungsmenge)	$1 \mid 1 \rightarrow 1$ $1 \mid 0 \rightarrow 1$ $0 \mid 0 \rightarrow 0$
∧	bitweises „Exklusiv-Oder“ (Bit-Antivalenz)	$1 \wedge 1 \rightarrow 0$ $1 \wedge 0 \rightarrow 1$ $0 \wedge 0 \rightarrow 0$
~	bitweise Inversion (Einerkomplement, unär)	$\sim 1 \rightarrow 0$ $\sim 0 \rightarrow 1$

Bitoperationen (Forts.) [= Java]

- 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`	?	?	?	?	?	?	?	?
`0x80`	1	0	0	0	0	0	0	0
`PORTD |= 0x80`	1	?	?	?	?	?	?	?
`~0x80`	0	1	1	1	1	1	1	1
`PORTD &= ~0x80`	0	?	?	?	?	?	?	?
`0x08`	0	0	0	0	1	0	0	0
`PORTD ^= 0x08`	?	?	?	?	¿	?	?	?

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

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

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

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

Bitoperationen – Anwendung (Forts.)

- Bitmasken werden gerne als Hexadezimal-Literale angegeben

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(void) {
  uint8_t mask = (1<<RED0) | (1<<RED1);

  sb_led_setMask (mask);

  while(1) ;
}
```

⇝

Bedingte Auswertung [≈Java][↪ GDI, 07-10]

- 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
    $\rightsquigarrow$ Nebeneffekte von $Ausdruck_1$ werden sichtbar
  - Ergebnis ist der Wert von $Ausdruck_2$

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

Vorrangregeln bei Operatoren [≈Java][↪ GDI, 03-07]

	Klasse	Operatoren	Assoziativität

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

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

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

Typumwandlung in Ausdrücken

- Eine Operation wird *mindestens* mit `int`-Wortbreite berechnet
  - `short`- und `signed char`-Operanden werden implizit „aufgewertet“ (↪ *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.)

- 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 (Forts.)

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

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

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

int8_t: 75 | int: 100 | int: 3 | double 4.0 | int: 300 | double: 300.0 | double: 75.0

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    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 (res)

int: 1 ((int) u)

int: 1 (s < (int) u)

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Goto-Anweisung

```
    ...
    goto Label
    ...
Label:
    ...

    ...
Label:
    ...
    goto Label
    ...
```

- `goto`-Anweisungen werden selten gebraucht
- `goto` in früheren JAVA-Versionen enthalten
- Edgar Dijkstra: „Go To Statement Considered Harmful"
- sie erzeugen leicht „Spagetti-Code"
- Label darf nicht letzte Anweisung in Funktion sein

`goto`- und `if (...) goto`-Anweisungen sind die einzigen Kontrollstrukturen, die die Hardware wirklich ausführen kann (wichtig für das spätere Verständnis von z.B. Unterbrechungen).

Bedingte Anweisung [= Java]

- `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]

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

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

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

Abweisende und nicht-abweisende Schleife [= Java]

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

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

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

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

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

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

Zählende Schleife [= Java]

- `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*)
    $\leadsto$ die `for`-Schleife ist eine „verkappte“ `while`-Schleife

Schleifensteuerung [= Java]

- 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);
}
```

Schleifen & goto-Anweisungen

Alle Schleifentypen lassen sich **semantisch-äquivalent** in Sequenzen mit goto-Anweisungen umbauen. Beispiel:

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

```
  uint8_t led = 0;
  goto test;
loop:
  if (led == RED1)
    goto next;
  sb_led_on(led);
next:
  led++;
test:
  if (led < 8)
    goto loop;
end:
```

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Was ist eine Funktion?

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

Funktion ↦ Abstraktion ↪ 4–1

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

Beispiel

- Funktion (Abstraktion) `sb_led_setMask()`

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

- Implementierung in der `libspicboard`

```
void sb_led_setMask(uint8_t setting)
```

Sichtbar: Bezeichner und formale Parameter

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

    } else {
      sb_led_off(i);
    }
  }
}
```

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*$_{n\,opt}$ [=Java]
    (Parameter-Bezeichner sind optional)
    `void`, falls kein Parameter erwartet wird [≠Java]
  - {*Block*} — Implementierung; formale Parameter stehen als lokale Variablen bereit [=Java]
- Beispiele:

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

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

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, 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]

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: *Typ 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;
}
```

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(void) {
4    double d = 47.11;
5    foo(d);
6    return 0;
7  }
8
9  void foo(int a, int b) {
10   printf("foo: a:%d, b:%d\n", a, b);
11 }
```

5 Funktion `foo()` ist nicht **deklariert** ⇝ 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(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!**

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

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

Variablendefinition [≈Java]

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

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

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

Variablendefinition (Forts.)

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

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

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

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

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

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)
    - Kommentare werden entfernt
    - Zeilen, die mit \ enden, werden zusammengefügt
    - ...
  - Steuerbare Transformationen (durch den Programmierer)
    - Präprozessor-Direktiven werden evaluiert und ausgeführt
    - Präprozessor-Makros werden expandiert

Präprozessor-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 < 0 || 8 < NUM_LEDS
#  error invalid NUM_LEDS           // this line is not included
#endif

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

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

}
```

Präprozessor – Gefahren [≠ Java]

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

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

<< hat geringere Präzedenz als *

⇝ `n = 1 << 2 * 3`

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

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

⇝ `n = (1 << 2) * 3`

  - Aber nicht alle

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

a++ wird ggf. zweimal ausgewertet

⇝ `n = ((x++ > 7) ? x++ : 7)`

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

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

Systemnahe Programmierung in C

Teil C Systemnahe Softwareentwicklung

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Skriptum_handout

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Softwareentwurf

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

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

Beispiel-Projekt: Eine Wetterstation

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

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

Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

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

Funktionale Dekomposition: Probleme

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

Funktionale Dekomposition: Probleme

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

Prinzip der Trennung der Belange

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

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

Zugriff auf Daten (Variablen)

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

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

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

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

Zugriff auf Daten (Variablen) (Forts.)

- Sichtbarkeit und Lebensdauer sollten restriktiv ausgelegt werden
  - Sichtbarkeit so beschränkt wie möglich!
    - Überraschende Zugriffe „von außen“ ausschließen (Fehlersuche)
    - Implementierungsdetails verbergen (Black-Box-Prinzip, *information hiding*)
  - Lebensdauer so kurz wie möglich
    - Speicherplatz sparen
    - Insbesondere wichtig auf μ-Controller-Plattformen ↪ 1–4

Konsequenz: Globale Variablen vermeiden!

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

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

Lösung: Modularisierung

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

Was ist ein Modul?

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

Modul ↦ Abstraktion ↪ 4–1

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

Module in C [≠ Java]

- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein idiomatisch (über Konventionen) realisiert ↪ 3–16
  - Modulschnittstelle ↦ `.h`-Datei (enthält Deklarationen ↪ 9–7)
  - Modulimplementierung ↦ `.c`-Datei (enthält Definitionen ↪ 9–3)
  - Modulverwendung ↦ `#include <Modul.h>`

```
extern void Init(uint16_t br);
extern void Send(char ch);
...
```

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

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

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

Ggf. modulinterne Hilfsfunktionen und Daten (`static`)

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

Module in C – Export [≠ Java]

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

Quelldatei (`foo.c`)

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

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

Objektdatei (`foo.o`)

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

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

Module in C – Import [≠ Java]

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

Quelldatei (`bar.c`)

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

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

Objektdatei (`bar.o`)

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

Module in C – Import (Forts.)

[≠Java]

- Die eigentliche Auflösung erfolgt durch den Linker

Linken ist nicht typsicher!

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

⇝ Typsicherheit muss beim **Übersetzen** sichergestellt werden

⇝ Einheitliche Deklarationen durch gemeinsame Header-Datei

Module in C – Header [≠ Java]

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

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

  - Globale Variablen durch `extern`

    ```
    extern uint16_t a;
    ```

Das `extern` unterscheidet eine Variablendeklaration von einer Variablendefinition.

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

Module in C – Header (Forts.)

[≠ Java]

Modulschnittstelle: `foo.h`

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

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

#endif // _FOO_H
```

Modulimplementierung `foo.c`

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


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

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

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

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

Zurück zum Beispiel: Wetterstation

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

Zusammenfassung

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

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Einordnung: Zeiger (*Pointer*)

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

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

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

Zeiger (*Pointer*)

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

„Effizienz durch Maschinennähe“ ↪ 3–17

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

Definition von Zeigervariablen

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

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```

Adress- und Verweisoperatoren

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

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

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

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

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

Zeiger als Funktionsargumente

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

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

    ⇝ **Call-by-reference**

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Gesamtüberblick)

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

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

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

}
```

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

    tmp = *px; ❷
```

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

    tmp = *px; ❷
    *px = *py; ❸
```

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

    tmp = *px; ❷
    *px = *py; ❸
```

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

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

}
```

Einordnung: Felder (*Arrays*) [≈Java]

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

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

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

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

Feldinitialisierung

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

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

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

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

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

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

Feldzugriff

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

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

LEDs[3] = BLUE1;

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

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

Felder sind Zeiger

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

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

Felder sind Zeiger

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

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```

Felder sind Zeiger

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

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

Zeiger sind Felder

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

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

LEDs[3]    = BLUE1;
uint8_t *p = LEDs;

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

Rechnen mit Zeigern

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

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```

```
int array[5];
ip = array; ❶
```

`(ip+3) ≡ &ip[3]`

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

Zeigerarithmetik – Operationen

- Arithmetische Operationen

  ++ Prä-/Postinkrement
  ⇝ Verschieben auf das nächste Objekt

  −− Prä-/Postdekrement
  ⇝ Verschieben auf das vorangegangene Objekt

  +, − Addition / Subtraktion eines `int`-Wertes
  ⇝ Ergebniszeiger ist verschoben um n Objekte

  − Subtraktion zweier Zeiger
  ⇝ Anzahl der Objekte n zwischen beiden Zeigern (Distanz)

- Vergleichsoperationen: $<$, $<=$, $==$, $>=$, $>$, $!=$ ↪ 7–3
  ⇝ Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen

Felder sind Zeiger sind Felder – Zusammenfassung

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

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

`array`	$\equiv$	`&array[0]`	$\equiv$	`ip`	$\equiv$	`&ip[0]`
`*array`	$\equiv$	`array[0]`	$\equiv$	`*ip`	$\equiv$	`ip[0]`
`*(array + i)`	$\equiv$	`array[i]`	$\equiv$	`*(ip + i)`	$\equiv$	`ip[i]`
		`array++`	$\not\equiv$	`ip++`		

Fehler: `array` ist konstant!

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

Felder als Funktionsparameter

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

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

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

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

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

Felder als Funktionsparameter (Forts.)

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

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

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

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

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

Felder als Funktionsparameter (Forts.)

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

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

Dabei gilt: "hallo" ≡

↪ 6–13

- Implementierungsvarianten

Variante 1: Feld-Syntax

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

Variante 2: Zeiger-Syntax

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

Zeiger auf Zeiger

- Ein Zeiger kann auch auf eine Zeigervariable verweisen

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

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

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

Zeiger auf Funktionen

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

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

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

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

Zeiger auf Funktionen (Forts.)

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

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

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

Zeiger auf Funktionen (Forts.)

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

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

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

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

Zusammenfassung

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

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Strukturen: Motivation

- Gehören Variable „irgendwie“ zusammen,
  - wäre es auch besser diese zusammenzufassen
  - „problembezogene Abstraktionen“ ↪ 4–1
  - „Trennung der Belange“ ↪ 12–4
- Dies geht in C mit Verbundtypen (Strukturen)

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

// Variable definition
struct Student stud;

// Pointer definition
struct Student *pstud;
```

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

Die Datenelemente werden hintereinander im Speicher abgelegt.

Strukturen: Variablendefinition und -initialisierung

- Analog zu einem Array kann eine Strukturvariable bei Definition elementweise initialisiert werden ↪ 13–8

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

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

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

- Analog zur Definition von `enum`-Typen kann man mit `typedef` die Verwendung vereinfachen ↪ 6–8

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

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

Strukturen: Elementzugriff

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

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

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

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

Strukturen: Elementzugriff

- Bei einem Zeiger auf eine Struktur würde Klammerung benötigt

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

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

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

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

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

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

Strukturen als Funktionsparameter

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

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

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

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

- Bei größeren Strukturen wird das **sehr ineffizient**
  - Z. B. `Student` (↪ 14–1): Jedes mal 134 Byte allozieren und kopieren
  - Besser man übergibt einen Zeiger auf eine konstante Struktur

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

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

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

Bit-Strukturen: Bitfelder

- Strukturelemente können auf Bit-Granularität festgelegt werden
  - Der Compiler fasst Bitfelder zu passenden Ganzzahltypen zusammen
  - Nützlich, um auf einzelne Bit-Bereiche eines Registers zuzugreifen

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

7	6	5	4	3	2	1	0
				ISC11	ISC10	ISC01	ISC00
				R/W	R/W	R/W	R/W

```
typedef struct {
  uint8_t ISC0     : 2;     // bit 0-1: interrupt sense control INT0
  uint8_t ISC1     : 2;     // bit 2-3: interrupt sense control INT1
  uint8_t reserved : 4;     // bit 4-7: reserved for future use
} EICRA_t;
```

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Was tut ein Compiler?

Aufgabe des Compilers: Zerlegung des Programms in kleine, vom μ-Controller ausführbare Befehle

Beispiel 1: Zerlegung eines Ausdrucks

```
int a, b, c, d;

a = b + c * abs(d - 1);
```

```
int r0, r1, r2, r3;
int a, b, c, d;

r0 = b;
r1 = c;
r3 = d;
r3 -= 1;
r2 = abs(r3);
r1 *= r2;
r0 += r1;
a = r0;
```

`a, b, ...` : „Speichervariablen"

`r0, r1, ...` : „Registervariablen"

15-MC-Vorbemerkungen: 2024-04-09

Was tut ein Compiler?

Aufgabe des Compilers: Zerlegung des Programms in kleine, vom μ-Controller ausführbare Befehle

Beispiel 2: Zerlegung einer Kontrollstruktur (1. Schritt)

```
if (n != 0) {
    for (i = 0; i != 10; i++)
        output();
    }
}
```

```
if (n != 0) {
    i = 0;
    while (i != 10) {
        output();
        i++;
    }
}
```

Was tut ein Compiler?

Aufgabe des Compilers: Zerlegung des Programms in kleine, vom μ-Controller ausführbare Befehle

Beispiel 2: Zerlegung einer Kontrollstruktur (2. Schritt)

```
if (n != 0) {
    i = 0;
    while (i != 10) {
        output();
        i++;
    }
}
```

```
if (n != 0) {
    i = 0;
    goto test;
loop:
    output();
    i++;
test:
    if (i != 10) goto loop;
}
```

Was tut ein Compiler?

Aufgabe des Compilers: Zerlegung des Programms in kleine, vom μ-Controller ausführbare Befehle

Beispiel 2: Zerlegung einer Kontrollstruktur (3. Schritt)

```
if (n != 0) {
    i = 0;
    goto test;
loop:
    output();
    i++;
test:
    if (i != 10) goto loop;
}
```

```
        if (n == 0) goto endif;
        i = 0;
        goto test;
    loop:
        output();
        i++;
    test:
        if (i != 10) goto loop;
    endif:
```

Was tut ein Compiler?

Aufgabe des Compilers: Zerlegung des Programms in kleine, vom μ-Controller ausführbare Befehle

Beispiel 2: Zerlegung einer Kontrollstruktur (3. Schritt)

```
        if (n == 0) goto endif;
        i = 0;
        goto test;
    loop:
        output();
        i++;
    test:
        if (i != 10) goto loop;
    endif:
```

```
        r0 = n;
        if (r0 == 0) goto endif;
        r0 = 0;
        i = r0;
        goto test;
    loop:
        output();
        r0 = i;
        r0++;
        i = r0;
    test:
        r0 = i;
        if (r0 != 10) goto loop;
    endif:
```

Was tut ein Compiler?

Aufgabe des Compilers: Zerlegung des Programms in kleine, vom μ-Controller ausführbare Befehle:

- `rN = const;`
- `rN = var;`
- `rN` op`= const;`
- `rN` op`= rN;`
- `rN = func(...);`
- `var = rN;`
- `goto label;`
- `if (rN` op `const) goto label;`
- `if (rN` op `rM) goto label;`
- `return rN;`
- ...

Was tut ein Compiler?

Typische, vom μ-Controller ausführbare Befehle (Beispiele):

C-Code	Mnemonic	
`rN++;`	`inc rN`	increment
`rN--;`	`dec rN`	decrement
`rN` = const;	`ldi rN`, const	load immediate
`rN` = var;	`ld rN`, var	load
`rN` += const;	`addi rN,` const	add immediate
`rN` -= const;	`subi rN,` const	subtract immediate
`rN += rM;`	`add rN, rM`	add
`rN -= rM;`	`sub rN, rM`	sub
`rN` = func();	`call` func	call function
`var = rN;`	`st var, rN`	store
`goto` label;	`jmp` label	jump
`if (rN == rM) goto` label;	`cmp rN, rM` `beq label`	compare branch if equal
...	...	

Vorhandene Befehle siehe Handbuch zum Prozessor/μ-Controller.

15-MC-Vorbemerkungen: 2024-04-09

Was tut ein Compiler?

Beispielprogramm:

vereinfachter C-Code		Assembler-Code	
	`r0 = n;`		`ld r0, n`
	`if (r0 == 0) goto endif;`		`cmpi r0, 0`
			`beq endif`
	`r0 = 0;`		`ldi r0, 0`
	`i = r0;`		`st i, r0`
	`goto test;`		`jmp test`
`loop:`	`output();`	`loop:`	`call output`
	`r0 = i;`		`ld r0, i`
	`r0++;`		`inc r0`
	`i = r0;`		`st i, r0`
`test:`	`r0 = i;`	`test:`	`ld r0, i`
	`if (r0 != 10) goto loop;`		`cmpi r0, 10`
			`bneq loop`
`endif:`		`endif:`	

Was tut ein Compiler?

Beispielprogramm:

vereinfachter C-Code		Assembler-Code	
	uint8_t n;	10	
	uint8_t i;	11	
	...	...	
	r0 = n;	20	ld r0, 10
	if (r0 == 0) goto endif;	21	cmpi r0, 0
		22	beq 33
	r0 = 0;	23	ldi r0, 0
	i = r0;	24	st 11, r0
	goto test;	25	jmp 30
loop:	output();	26	call 70
	r0 = i;	27	ld r0, 11
	r0++;	28	inc r0
	i = r0;	29	st 11, r0
test:	r0 = i;	30	ld r0, 11
	if (r0 != 10) goto loop;	31	cmpi r0, 10
		32	bneq 26
endif:		33	
	...	...	...
	output(...)	70	...

Was tut ein Assembler?

Beispielprogramm:

Assembler-Code		Binär-Code
...		...
20	ld r0, 10	0a4f
21	cmpi r0, 0	a77f
22	beq 33	77bc
23	ldi r0, 0	87ee
24	st 11, r0	7439
25	jmp 30	30af
26	call 70	dd33
27	ld r0, 11	75ca
28	inc r0	9e88
29	st 11, r0	11f2
30	ld r0, 11	ad8f
31	cmpi r0, 10	54e1
32	bneq 26	98e4
...	...	...

Codierung der Befehle siehe Handbuch zum Prozessor/μ-Controller.

Program Counter / Instruction Pointer

Program Counter (PC) oder Instruction Pointer (IP):
Register, das die Nummer der Speicherzelle enthält, die den nächsten auszuführenden Befehl enthält

```
PC = 24

                    ...
                    21   cmpi r0, 0
                    22   beq 33
                    23   ldi r0, 0
          PC -->    24   st 11, r0
                    25   jmp 30
                    26   call 70
                    27   ld r0, 11
                    ...  ...
```

Bemerkungen

Diese Folien

- sind *wichtig für das Verständnis* der nächsten Vorlesungen
  - C-Code wird vom C-Compiler in kleinere Einheiten zerlegt
  - kleinere Einheiten können in Befehle für den μ-Controller übersetzt werden
  - Befehle werden vom Assembler in binären Code übersetzt
  - Befehle werden vom μ-Controller gemäß dem Program Counter Schritt-für-Schritt abgearbeitet
- sind *nicht Prüfungs-relevant*

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Beispiel ATmega32: Blockschaltbild

Beispiel ATmega-Familie: CPU-Architektur

- Harvard-Architektur (getrennter Speicher für Code und Daten)
- Peripherie-Register sind in den Speicher eingeblendet
  $\rightsquigarrow$ ansprechbar wie globale Variablen

Zum Vergleich: PC basiert auf von-Neumann-Architektur mit gemeinsamem Speicher; I/O-Register verwenden einen speziellen I/O-Adressraum.

Wie arbeitet ein Prozessor?

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

Wie arbeitet ein Prozessor?

Wie arbeitet ein Prozessor?

Wie arbeitet ein Prozessor?

Wie arbeitet ein Prozessor?

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

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

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

Wie arbeitet ein Prozessor?

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

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

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

Wie arbeitet ein Prozessor?

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

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

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

w: **ret**
PC = PC`

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Was ist ein μ-Controller?

- μ-**Controller** := Prozessor + Speicher + Peripherie
  - Faktisch ein Ein-Chip-Computersystem $\longrightarrow$ SoC (*System-on-a-Chip*)
  - Häufig verwendbar ohne zusätzliche externe Bausteine, wie z. B. Taktgeneratoren und Speicher $\rightsquigarrow$ kostengünstiges Systemdesign
- Wesentliches Merkmal sind die (reichlich) enthaltenen Ein-/Ausgabe-Komponenten (Peripherie)
- Die Abgrenzungen sind fließend: Prozessor $\longleftrightarrow$ μC $\longleftrightarrow$ SoC
  - AMD64-CPUs haben ebenfalls eingebaute Timer, Speicher (Caches), ...
  - Einige μC erreichen die Geschwindigkeit „großer Prozessoren“

Beispiel ATmega32: Blockschaltbild

Peripheriegeräte

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

Peripheriegeräte: Beispiele

- Auswahl von typischen Peripheriegeräten in einem μ-Controller
  - Timer/Counter — Zählregister, die mit konfigurierbarer Frequenz (Timer) oder durch externe Signale (Counter) erhöht werden und bei konfigurierbarem Zählwert einen Interrupt auslösen.
  - Watchdog-Timer — Timer, der regelmäßig neu beschrieben werden muss oder sonst einen RESET auslöst („Totmannknopf“).
  - (A)synchrone serielle Schnittstelle — Bausteine zur seriellen (bitweisen) Übertragung von Daten mit synchronem (z. B. RS-232) oder asynchronem (z. B. I^2C) Protokoll.
  - A/D-Wandler — Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V $\mapsto$ 10-Bit-Zahl).
  - PWM-Generatoren — Bausteine zur Generierung von pulsweiten-modulierten Signalen (pseudo-analoge Ausgabe).
  - Ports — Gruppen von üblicherweise 8 Anschlüssen, die auf GND oder Vcc gesetzt werden können oder deren Zustand abgefragt werden kann. $\hookrightarrow$ 17–11
  - Bus-Systeme — SPI, RS-232, CAN, Ethernet, MLI, I^2C, ...

Peripheriegeräte – Register

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

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

[1]

Peripheriegeräte – Register (Forts.)

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

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

7	6	5	4	3	2	1	0
DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

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

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

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

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

Verwendungsbeispiele: ↪ 3–7 und ↪ 3–11 [1]

Peripheriegeräte – Register (Forts.)

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

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

- `0x12`: Adresse: int
- `(volatile uint8_t *)`: Adresse: volatile uint8_t *(Cast ↪ 7–19)
- `*(...)`: Wert: volatile uint8_t (Dereferenzierung ↪ 13–4)

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

- Beispiel

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

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

Registerzugriff und Nebenläufigkeit

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

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

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

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

```
// Resulting assembly code

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

  ...
L2:
  ret
```

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

Der `volatile`-Typmodifizierer

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

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

    // button0 pressed
    ...
  }

  if (! (PIND & 0x4)) {

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

```
// Resulting assembly code


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

  ...
L2:
  ret
```

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

Der `volatile`-Typmodifizierer (Forts.)

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

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

volatile!

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

Die Verwendung von `volatile` verursacht erhebliche Kosten

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

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

Peripheriegeräte: Ports

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

Beispiel ATmega328PB: Port/Pin-Belegung

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

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

Diese Pins stehen daher für PORTC nicht zur Verfügung.

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Ereignisbehandlung

- Bei einem Peripheriegerät tritt ein Ereignis (↯) auf ↪ 17–3
  - Signal an einem Port-Pin wechselt von *low* auf *high*
  - Ein *Timer* ist abgelaufen
  - Ein A/D-Wandler hat einen neuen Wert vorliegen
  - ...
- Wie bekommt das Programm das (nebenläufige) Ereignis mit?
- Zwei alternative Verfahren
  - **Polling:** Das Programm überprüft den Zustand regelmäßig und ruft ggf. eine Bearbeitungsfunktion auf.
  - **Interrupt:** Gerät „meldet“ sich beim Prozessor, der daraufhin in eine Bearbeitungsfunktion verzweigt.

Interrupt ↦ Funktionsaufruf „von außen“

Polling vs. Interrupts – Vor- und Nachteile

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

Beide Verfahren bieten spezifische Vor- und Nachteile
⇝ Auswahl anhand des konkreten Anwendungsszenarios

Interrupt ↦ unvorhersagbarer Aufruf „von außen“

Interruptsperren

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

IRQ ↦ *Interrupt ReQuest*

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

Interruptsperren: Beispiel

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

Ablauf eines Interrupts – Überblick

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

Ablauf eines Interrupts – Details

- Hier als Erweiterung unseres einfachen Pseudoprozessors ↪ 16–3
  - Nur eine Interruptquelle
  - Sämtliche Register werden von der Hardware gerettet

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Ablauf eines Interrupts – Details

❶ Gerät signalisiert Interrupt (aktueller Befehl wird noch fertiggestellt)

Ablauf eines Interrupts – Details

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

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Ablauf eines Interrupts – Details

❷ Die Zustellung weiterer Interrupts wird verzögert
❸ Registerinhalte werden gesichert

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Ablauf eines Interrupts – Details

❹ Aufzurufende ISR wird ermittelt

Ablauf eines Interrupts – Details

❺ ISR wird ausgeführt

Ablauf eines Interrupts – Details

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

w: **call** <func>	w: **ret**	w: **icall**	w: **iret**
PC` = PC PC = func	PC = PC`	SR` = SR SR.IE = 0 IP = 0 PC` = PC PC = isr R1` = R1	SR = SR` PC = PC` R1 = R1`

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Pegel- und Flanken-gesteuerte Interrupts

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

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

Interruptsteuerung beim AVR ATmega

- IRQ-Quellen beim ATmega328PB (IRQ ↦ *Interrupt ReQuest*)
  - 45 IRQ-Quellen [1, S. 78]
  - einzeln de-/aktivierbar
  - IRQ ⇝ Sprung an Vektor-Adresse

Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Coutner2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Coutner0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI0 STC	SPI1 Serial Transfer Complete
19	0x0024	USART0_RX	USART0 Rx Complete
20	0x0026	USART0_UDRE	USART0, Data Register Empty
21	0x0028	USART0_TX	USART0, Tx Complete
22	0x002A	ADC	ADC Conversion Complete

- Verschaltung SPiCboard
  (↪ 17–12 ↪ 2–8)
  - `INT0` ↦ `PD2` ↦ Button0 (hardwareseitig entprellt)
  - `INT1` ↦ `PD3` ↦ Button1

Externe Interrupts: Register

- Steuerregister für `INT0` und `INT1`
  - `EIMSK` **External Interrupt Mask Register:** Legt fest, ob die Quellen `INT`*i* IRQs auslösen (Bit `INT`*i*=1) oder deaktiviert sind (Bit `INT`*i*=0) [1, S. 84]

7	6	5	4	3	2	1	0
						INT1	INT0
						R/W	R/W

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

7	6	5	4	3	2	1	0
				ISC11	ISC10	ISC01	ISC00
				R/W	R/W	R/W	R/W

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

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

Externe Interrupts: Verwendung

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

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

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

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

Externe Interrupts: Verwendung (Forts.)

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

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

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

Externe Interrupts: Verwendung (Forts.)

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

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

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

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 Verbundtypen

15 μC-Systemarchitektur – Vorbemerkungen

16 μC-Systemarchitektur – Prozessor

17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Nebenläufigkeit

Definition: Nebenläufigkeit

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

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

Nebenläufigkeitsprobleme

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

```
static volatile uint16_t cars;

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

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

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

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

Nebenläufigkeitsprobleme (Forts.)

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

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

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

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

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

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

Nebenläufigkeitsprobleme: *Lost-Update*-Anomalie

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

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

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

Nebenläufigkeitsprobleme: *Read-Write*-Anomalie

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

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

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

Interruptsperren: Datenflussanomalien verhindern

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

kritisches Gebiet

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

Nebenläufigkeitsprobleme (Forts.)

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

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

```
static volatile int8_t event;

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

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

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

Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

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

```
static volatile int8_t event;

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

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

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

Lost-Wakeup: Dornröschenschlaf verhindern

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

```
static volatile int8_t event;

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

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

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

Zusammenfassung

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

Systemnahe Programmierung in C

Teil D Betriebssystemabstraktionen

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Skriptum_handout

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln – Aufruf `strlen(x);`

```
/* 1. Version */
int strlen(const char *s)
{
    int n;
    for (n = 0; *s != '\0'; n++) {
        s++;
    }
    return n;
}
```

```
/* 2. Version */
int strlen(const char *s)
{
    const char *p = s;
    while (*p != '\0') {
        p++;
    }
    return p - s;
}
```

Zeiger, Felder und Zeichenketten (Forts.)

- wird eine Zeichenkette zur Initialisierung eines `char`-Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

```
char amessage[] = "now is the time";
```

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

```
amessage[0] = 'h';
```

Zeiger, Felder und Zeichenketten (Forts.)

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

```
const char *pmessage = "hello world";    /*(1)*/
```

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

- die Zeichenkette selbst wird vom Compiler als konstanter Wert (String-Literal) im Speicher angelegt
- es wird ein Speicherbereich für einen Zeiger reserviert (z.B. 4 Byte) und mit der Adresse der Zeichenkette initialisiert

Zeiger, Felder und Zeichenketten (4)

```
const char *pmessage = "hello world";    /*(1)*/
```

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

- `pmessage` ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf (`pmessage++;`)
- der Speicherbereich von `"hello world"` darf aber nicht verändert werden
  - der Compiler erkennt dies durch das Schlüsselwort `const` und verhindert schreibenden Zugriff über den Zeiger
  - manche Compiler legen solche Zeichenketten ausserdem im schreibgeschützten Speicher an (=> Speicherschutzverletzung beim Zugriff, falls der Zeiger nicht als `const`-Zeiger definiert wurde)

Zeiger, Felder und Zeichenketten (5)

- die Zuweisung eines `char`-Zeigers oder einer Zeichenkette an einen `char`-Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger `pmessage` lediglich die Adresse der Zeichenkette `"now is the time"` zu

- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

Zeiger, Felder und Zeichenketten (6)

- Um eine ganze Zeichenkette einem anderen char-Feld zuzuweisen, muss sie kopiert werden: Funktion `strcpy` in der Standard-C-Bibliothek
- Implementierungsbeispiele:

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

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

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

Felder von Zeigern

Auch von Zeigern können Felder gebildet werden

- Deklaration

```
int *pfeld[5];
int i = 1;
int j;
```

Felder von Zeigern (Forts.)

Auch von Zeigern können Felder gebildet werden

- Deklaration

```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriff auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

Felder von Zeigern (Forts.)

Auch von Zeigern können Felder gebildet werden

- Deklaration

```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriff auf einen Zeiger des Feldes

```
pfeld[3] = &i;
```

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

```
j = *pfeld[3];
```

Felder von Zeigern (Forts.)

Beispiel: Definition und Initialisierung eines Zeigerfeldes:

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

    return (n < 1 || 12 < n) ? monatsname[0] : monatsname[n];
}
```

Argumente aus der Kommandozeile

- beim Aufruf eines Programms können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion `main()` durch zwei Aufrufparameter ermöglicht (beide Varianten gleichwertig):

```
int
main(int argc, char *argv[])
{
    ...
}
```

```
int
main(int argc, char **argv)
{
    ...
}
```

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

Argumente aus der Kommandozeile

- Kommando:

```
gcc -Wall -o test test.c
```

- C-Datei:

```
...
int main(int argc, char *argv[])
...
```

```
...
int main(int argc, char **argv)
...
```

Argumente – Beispiel

Beispiel: **echo**-Programm

```
~> echo Hallo SPiC!
Hallo SPiC!
~>
```

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");

    return 0;
}
```

Verbund-Datentypen / Strukturen

- Zusammenfassen mehrerer Daten zu einer Einheit
- Struktur-Deklaration

```
struct person {
    char name[20];
    int age;
};
```

- Definition einer Variablen vom Typ der Struktur

```
struct person p1;
```

- Zugriff auf ein Element der Struktur

```
strcpy(p1.name, "Peter Pan");
p1.age = 12;
```

Zeiger auf Strukturen

- Konzept analog zu „Zeiger auf Variable"
  - Adresse einer Struktur mit &-Operator zu bestimmen
- Beispiel

```
struct person stud1;
struct person *pstud;
pstud = &stud1;
```

- Besondere Bedeutung beim Aufbau verketteter Strukturen (Listen, Bäume, ...)
  - eine Struktur kann Adressen weiterer Strukturen desselben Typs enthalten

Zeiger auf Strukturen (Forts.)

- Zugriff auf Strukturkomponenten über Zeiger
- bekannte Vorgehensweise
  - „*"-Operator liefert die Struktur
  - „.”-Operator liefert ein Element der Struktur
  - **Aber:** Operatorenvorrang beachten!

```
(*pstud).age = 21;
```

- syntaktische Verschönerung
  - „->”-Operator

```
pstud->age = 21;
```

Verschachtelte/verkettete Strukturen

- Strukturen in Strukturen sind erlaubt – aber:
  - die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    => Struktur kann sich nicht selbst enthalten
  - die Größe eines Zeigers ist bekannt
    => Struktur kann Zeiger auf gleiche Struktur enthalten
  - Beispiele:

Verkettete Liste:

```
struct list {
    struct list *next;
    struct person stud;
};

struct list *head;
```

Baum:

```
struct tree {
    struct tree *left;
    struct tree *right;
    struct person stud;
};

struct tree *root;
```

Verkettete Listen

- Mehrere Strukturen desselben Typs werden über Zeiger miteinander verkettet

```
struct list { struct list *next; int val; };

struct list el1, el2, el3;
struct list *head;

head = &el1;
el1.next = &el2; el2.next = &el3; el3.next = NULL;
el1.val = 10;    el2.val = 20;    el3.val = 30;
```

- Laufen über eine verkettete Liste

```
int sum = 0;
for (struct list *curr = head; curr != NULL; curr = curr->next) {
    sum += curr->val;
}
```

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Ein-/Ausgabe

- E/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch „normale” Funktionen
  - Bestandteil der Standard-Bibliothek
  - einfache Programmierschnittstelle
  - effizient
  - portabel
  - betriebssystem-nah
- Funktionsumfang
  - Öffnen/Schließen von Dateien
  - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
  - formatierte Ein-/Ausgabe

Standard-Ein-/Ausgabe

Jedes C-Programm erhält beim Start automatisch 3 E/A-Kanäle:

`stdin`: Standard-Eingabe

- normalerweise mit der Tastatur verbunden
- „Dateiende" (`EOF`) wird durch Eingabe von `CTRL-D` am Zeilenanfang signalisiert
- bei Programmaufruf in der Shell auf Datei umlenkbar

```
~> prog < eingabedatei
```

`stdout`: Standard-Ausgabe

- normalerweise mit Bildschirm (bzw. dem Fenster in dem das Programm gestartet wurde) verbunden
- bei Programmaufruf in der Shell auf Datei umlenkbar

```
~> prog > ausgabedatei
```

`stderr`: Ausgabekanal für Fehlermeldungen

- normalerweise ebenfalls mit Bildschirm verbunden

Standard-Ein-/Ausgabe (Forts.)

- Pipes
  - Die Standardausgabe eines Programmes kann mit der Standardeingabe eines anderen Programms verbunden werden:

    ```
    ~> prog1 | prog2
    ```

    Die Umlenkung von Standard-E/A-Kanälen ist für die aufgerufenen Programme weitgehend unsichtbar.
- automatische Pufferung
  - Eingaben von der Tastatur werden normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem `NEWLINE`-Zeichen (`'\n'`) an das Programm übergeben!
  - Ausgaben an den Bildschirm werden vom Programm normalerweise zeilenweise zwischengespeichert und erst beim `NEWLINE`-Zeichen wirklich auf den Bildschirm geschrieben!

Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen
  - Zugriff auf Dateien
- Öffnen eines E/A-Kanals
  - Funktion `fopen` (File Open)
- Schließen eines E/A-Kanals
  - Funktion `fclose` (File Close)

Öffnen und Schließen von Dateien (Forts.)

- Schnittstelle `fopen`

```
#include <stdio.h>

FILE *fopen(const char *name, const char *mode);
```

`name`: Pfadname der zu öffnenden Datei

`mode`: Art, wie Datei zu öffnen ist

- `"r"`: zum Lesen (read)
- `"w"`: zum Schreiben (write)
- `"a"`: zum Schreiben am Dateiende (append)
- `"rw"`: zum Lesen und Schreiben (read/write)

- öffnet Datei `name`
- Ergebnis von `fopen`: Zeiger auf einen Datentyp `FILE`, der einen Dateikanal beschreibt; im Fehlerfall `NULL`

Öffnen und Schließen von Dateien (Forts.)

- Schnittstelle `fclose`

```
#include <stdio.h>

int fclose(FILE *fp);
```

  - schließt E/A-Kanal `fp`
  - Ergebnis ist entweder `0` (kein Fehler aufgetreten) oder `EOF` im Falle eines Fehlers

Öffnen und Schließen von Dateien – Beispiel

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

int main(void)
{
    FILE *fp; int ret;

    fp = fopen("test.dat", "w"); /* Open "test.dat" for writing. */
    if (fp == NULL) {
        /* Error */
        perror("test.dat"); /* Print error message. */
        exit(EXIT_FAILURE); /* Terminate program. */
    }

    ... /* Program can now write to file "test.dat". */

    ret = fclose(fp); /* Close file. */
    if (ret == EOF) {
        /* Error */
        perror("test.dat"); /* Print error message. */
        exit(EXIT_FAILURE); /* Terminate program. */
    }

    return EXIT_SUCCESS;
}
```

Zeichenweises Lesen und Schreiben

- Lesen eines einzelnen Zeichens
  - von der Standardeingabe

```
#include <stdio.h>
int getchar(void);
```

  - aus einer Datei

```
#include <stdio.h>
int fgetc(FILE *fp);
```

  - lesen das nächste Zeichen
  - geben das Zeichen als `int`-Wert zurück
  - geben bei Eingabe von CTRL-D bzw. am Ende der Datei `EOF` als Ergebnis zurück
- Schreiben eines einzelnen Zeichens
  - auf die Standardausgabe

```
#include <stdio.h>
int putchar(int c);
```

  - in eine Datei

```
#include <stdio.h>
int fputc(int c, FILE *fp);
```

  - schreiben das Zeichen `c`
  - geben im Fehlerfall `EOF` als Ergebnis zurück

Zeichenweises Lesen und Schreiben – Beispiel

Kopierprogramm:

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

int main(int argc, char *argv[]) {
    FILE *src, *dst;
    int c;

    if (argc != 3) { ... }

    if ((src = fopen(argv[1], "r")) == NULL) { ... }
    if ((dst = fopen(argv[2], "w")) == NULL) { ... }

    while ((c = fgetc(src)) != EOF) {
        if (fputc(c, dst) == EOF) { ... }
    }

    if (fclose(dst) == EOF) { ... }
    if (fclose(src) == EOF) { ... }

    return EXIT_SUCCESS;
}
```

Zeilenweises Lesen und Schreiben

- Lesen einer Zeile

```
#include <stdio.h>
char *fgets(char *buf, int bufsize, FILE *fp);
```

  - liest Zeichen aus Dateikanal `fp` in das `char`-Feld `buf` bis entweder `bufsize-1` Zeichen gelesen wurden oder `'\n'` oder `EOF` gelesen wurde
  - `s` wird mit `'\0'` abgeschlossen (`'\n'` wird nicht entfernt)
  - gibt bei `EOF` oder Fehler `NULL` zurück
  - für `fp` kann `stdin` eingesetzt werden, um von der Standardeingabe zu lesen

- Schreiben einer Zeile

```
#include <stdio.h>
int fputs(char *buf, FILE *fp);
```

  - schreibt die Zeichen im Feld `s` auf Dateikanal `fp`
  - gibt im Fehlerfall `EOF` zurück
  - für `fp` kann auch `stdout` oder `stderr` eingesetzt werden

Formatierte Ausgabe

- Schnittstelle

```
#include <stdio.h>
int printf(char *format, ...);
int fprintf(FILE *fp, char *format, ...);
int sprintf(char *buf, char *format, ...);
int snprintf(char *buf, int bufsize, char *format, ...);
```

- Die statt `...` angegebenen Parameter werden entsprechend der Angaben im `format`-String ausgegeben
  - bei `printf` auf der Standardausgabe
  - bei `fprintf` auf dem Dateikanal `fp`
    (für `fp` kann auch `stdout` oder `stderr` eingesetzt werden)
  - `sprintf` schreibt die Ausgabe in das `char`-Feld `buf`
    (achtet dabei aber nicht auf das Feldende => Pufferüberlauf möglich!)
  - `snprintf` arbeitet analog, schreibt aber nur maximal `bufsize` Zeichen
    (`bufsize` sollte natürlich nicht größer als die Feldgröße sein)

Formatierte Ausgabe (Forts.)

- Zeichen im `format`-String können verschiedene Bedeutung haben
  - normale Zeichen:
    werden einfach in die Ausgabe kopiert
  - Escape-Zeichen:
    z.B. `\n` oder `\t` werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
  - Format-Anweisungen:
    beginnen mit `%`-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem `format`-String aufbereitet werden soll
- für genauere Informationen siehe Manuals (`man 3 printf`, ...)

Formatierte Ausgabe (Forts.)

- Format-Anweisungen

%d, %i: `int`-Parameter als Dezimalzahl ausgeben
%ld, %li: entsprechend für `long int`

%f: `float`-Parameter als Fließkommazahl ausgeben (z.B. `13.153534`)
%lf: entsprechend für `double`

%e: `float`-Parameter als Fließkommazahl in 10er-Potenz-Schreibweise ausgeben (z.B. `2.71456e+02`)
%le: entsprechend für `double`

%c: `char`-Parameter als einzelnes Zeichen ausgeben
%s: `char`-Feld wird ausgegeben, bis `'\0'` erreicht ist

%%: ein `%`-Zeichen wird ausgegeben

...: ...

Formatierte Ausgabe – Beispiel

```
int tag = 25;
int monat = 6;
int jahr = 2009;
char *name = "Michael Jackson";
printf("Am %d.%d.%d starb\n%s.\n",
       tag, monat, jahr, name);

printf("\n");

double pi = asin(1.0) * 2.0;
double e = exp(1.0);
fprintf(stdout,
        "Wichtige Werte sind:\n");
fprintf(stdout,
        "pi=%lf und e=%lf\n", pi, e);
```

```
~> ./test
Am 25.6.2009 starb
Michael Jackson.

Wichtige Werte sind:
pi=3.141593 und e=2.718282
~>
```

Formatierte Eingabe

- Schnittstelle

```
#include <stdio.h>

int scanf(char *format, ...);
int fscanf(FILE *fp, char *format, ...);
int sscanf(char *buf, char *format, ...);
```

Format-String analog zur formatierten Ausgabe.
Für genauere Informationen siehe Manuals (`man 3 scanf`, ...).

Aber: da Werte gelesen werden sollen, müssen Zeiger auf die zu beschreibenden Variablen übergeben werden!

Formatierte Eingabe – Beispiel

```
double pi, e;
int ret;

ret = scanf("pi=%lf, e=%lf\n", &pi, &e);
if (ret != 2) {
    fprintf(stderr, "Bad input!\n");
    exit(EXIT_FAILURE);
}
printf("I got\n\tpi=%lf\n\te=%lf\n", pi, e);
```

```
~> ./test
3.14 2.718
Bad input!
~>
```

```
~> ./test
pi=3.14, e=2.718
I got
        pi=3.140000
        e=2.718000
~>
```

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Fehlerbehandlung

- Fast jeder Systemaufruf/Bibliotheksaufruf kann fehlschlagen

  => **Fehlerbehandlung unumgänglich!**

- Ziel:

  Es darf kein Programm ohne Fehlermeldung abstürzen!

Fehlerbehandlung

- Vorgehensweise:
  - Rückgabewert von Systemaufruf/Bibliotheksaufruf abfragen
  - Im Fehlerfall (häufig durch Rückgabewert -1 oder `NULL` angezeigt): Fehlercode steht in globaler Variablen `errno`
- Fehlermeldung kann mit der Funktion `perror` auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

- Zwischenergebnisse auf Plausibilität überprüfen

```
#include <assert.h>
void assert(int condition);
```

Wenn Bedingung `condition` nicht „wahr" ist, wird das Programm mit Fehlermeldung abgebrochen.

Fehlerbehandlung (Forts.)

- Fehlerbehandlung dem Kontext anpassen; Beispiele
  - Fehler aufgrund von Benutzer-Fehlern
    (z.B. Benutzer gibt falschen Dateinamen oder falsche URL ein)
    - Benutzer auf Fehler hinweisen
    - Benutzer neue Eingabe ermöglichen
    - fehlgeschlagenen Programmteil wiederholen
  - Fehler aufgrund fehlender Ressourcen
    (z.B. Speicher oder Platte voll)
    - Benutzer auf Fehler hinweisen
    - Benutzer Möglichkeit geben „aufzuräumen"
    - fehlgeschlagenen Programmteil wiederholen
  - Programmierfehler
    (z.B. Zwischenergebnisse falsch)
    - Fehlermeldung ausgeben
    - Programm abbrechen
  - ...

Fehlerbehandlung – Beispiel

```
...

assert(argv[1] != NULL);

/* Open file for writing. */
FILE *fp = fopen(argv[1], "w");
if (fp == NULL) {
    perror(argv[1]);
    exit(EXIT_FAILURE);
}

/* Write to file. */
...

/* Close file. */
int ret = fclose(fp);
if (ret == EOF) {
    perror("fclose");
    exit(EXIT_FAILURE);
}

...
```

```
~> ./test
test.c:9: main: Assertion
          'argv[1] != NULL' failed.
~>
```

```
~> ./test /etc/shadow
/etc/shadow: Permission denied
~>
```

```
~> ./test hallo.txt
fclose: Quota exceeded
~>
```

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Definition „Betriebssystem”

- DIN 44300
  - „... die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.”
- Andy Tannenbaum
  - „... eine Software-Schicht ..., die alle Teile eines Systems verwaltet und dem Benutzer eine Schnittstelle oder eine virtuelle Maschine anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware].”
- Zusammenfassung:
  - Software zur Verwaltung und Virtualisierung der Hardware-Komponenten (Betriebsmittel)
  - Programm zur Steuerung und Überwachung anderer Programme

Überblick

Bisher:

- **Ein** Programm, das
- **alleine**,
- **beim Booten** gestartet
- mit **Hardware-Zugriffen**
- seine Umgebung steuert.

Quelle: www.wikipedia.org

Jetzt:

- **Mehrere** Programme, die
- **nebenläufig**,
- **dynamisch** gestartet/beendet
- über **definierte E-/A-Funktionen**
- ihre Umgebung steuern.

Multitasking

Existiert mehr als eine Anwendung auf einem System („Multitasking"),

- müssen sich die Anwendungen abstimmen,
  - wer wann die/eine CPU bekommt,
  - wer welche Speicherbereiche verwenden darf,
  - wer welche Plattenblöcke verwenden darf,
  - wer welchen Teil des Bildschirms beschreiben darf,
  - ...

Da keine Anwendung für sich allein z.B. entscheiden kann, welche Speicherbereiche noch ungenutzt sind, muss es **gemeinsam benutzte Methoden und Zustandsvariablen** geben.

- muss sichergestellt sein, dass
  - sich alle Anwendungen an die Abmachungen halten (auch die versehentlich/absichtlich fehlerhaft programmierten!)

Hardware-Erweiterungen müssen Zugriffe auf unerlaubte Speicherbereiche bzw. E-/A-Geräte verhindern.

Betriebssystem-Kern

- „**gemeinsam benutzte Methoden und Zustandsvariablen**"
  - **Betriebssystem-Kern** („Kern", „System-Kern", „Kernel")
- „**Hardware-Erweiterungen**"
  - **Priviligierungsstufen** („Privilege Level", „Ringe")
  - **Speicherschutz** („Memory Management Unit" („MMU"))

Schichtenmodell

Definition
„**Betriebssystem**":

Betriebssystem:
Kern zusammen mit
Hilfsprogrammen

oder

Betriebssystem:
nur Kern

Priviligierungsstufen

- unpriviligierte Ebenen
  („Anwendungsebene", „User-Ebene", „User-Ring")
  - darf „normale" CPU-Instruktionen ausführen,
  - darf auf den ihr zugewiesenen Speicher zugreifen,
  - darf Betriebssystem-Methoden aufrufen
- priviligierte Ebene
  („System-Ebene", „Kern-Ebene", „Ring 0")
  - darf alle CPU-Instruktionen ausführen,
  - darf auf jeden Speicherbereich zugreifen,
  - darf Speicherschutz umkonfigurieren,
  - darf auf E-/A-Geräte zugreifen

Wechsel in die priviligierte Ebene durch

- System-Aufrufe („System Calls", „Traps")
- Unterbrechnungen („Interrupts")
- Ausnahmen („Exceptions")

System-Aufrufe

Beispiel: Anwendung braucht mehr Speicher
Schritte:

- Anwendung berechnet, wieviel Speicher sie benötigt,
- legt Parameter in CPU-Registern ab,
- wechselt mit spezieller CPU-Instruktion in den Kern,
  (=> ab jetzt priviligiert!)
- liest Parameter aus CPU-Registern,
- reserviert Speicher für sich,
- programmiert MMU um,
- legt Ergebnis in CPU-Registern ab,
- wechselt mit spezieller CPU-Instruktion zurück in Anwender-Ebene,
  (=> ab jetzt wieder unpriviligiert!)
- und holt Ergebnis aus CPU-Register.

Verwaltung von Betriebsmitteln

- Aufgaben des Betriebssystem-Kerns
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel

- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar in
  - aktive, zeitlich aufteilbare (Prozessor)
  - passive, nur exklusiv nutzbare (periphere Geräte, z.B. Drucker u.Ä.)
  - passive, räumlich aufteilbare (Speicher, Plattenspeicher u.Ä.)
- Unterstützung bei der Fehlererholung

Klassifikation von Betriebssystemen

- Unterschiedliche Klassifikationskriterien
  - Zielplattform
  - Einsatzzweck
  - Funktionalität

Klassifikation von Betriebssystemen (2)

- Wenigen „General Purpose"-, Mainframe- und Höchstleistungsrechner-Betriebssystemen steht eine Vielzahl kleiner und kleinster Spezialbetriebssysteme gegenüber:

  `C51, C166, C251, CMX RTOS, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, Hi Ross, Hynet-OS, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK Flex, OSEK Turbo, OSEK Plus, OSEKtime, Pricise/MQX, Pricise/RTCS, proOSEK, SOS, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, ThreadX, RTA, RTX51, RTX251, RTX166, RTXC, Softune, SSXS RTOS, VRTX, VxWorks, ...`

  Einsatzbereich: Eingebettete Systeme, häufig Echtzeit-Betriebssysteme, über 50% proprietäre (in-house) Lösungen

- Alternative Klassifikation: nach Architektur

Betriebssystem-Architekturen

- Umfang: zehntausende bis mehrere Millionen Befehlszeilen
  => Strukturierung hilfreich
- Verschiedene Strukturkonzepte
  - Laufzeitbibliotheken (minimal, vor allem im Embedded-Bereich)
  - monolithische Systeme
  - geschichtete Systeme
  - Minimalkerne
- Unterschiedliche Schutzkonzepte
  - kein Schutz
  - Schutz des Betriebssystems
  - Schutz des Betriebssystems und Anwendungen untereinander
  - feingranularer Schutz auch innerhalb von Anwendungen

Betriebssystem-Komponenten

- Speicherverwaltung
  - Wer darf Informationen wohin im Speicher ablegen?
- Prozessverwaltung
  - Wann wird welche Aufgabe bearbeitet?
- Dateisystem
  - Speicherung und Schutz von Langzeitdaten
- Interprozesskommunikation
  - Kommunikation zwischen verschiedenen Anwendungen bzw. parallel ablaufenden Anwendungsteilen
- Ein-/Ausgabe
  - Kommunikation mit der „Außenwelt" (Benutzer/Rechner)

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Überblick

- Einordnung

Überblick (2)

Anwendungsprogramm:

- liest/schreibt Dateiinhalte
- liest/schreibt Verzeichnisinhalte

Dateisystem:

- liest/schreibt Blöcke

Block-Gerätetreiber:

- liest/schreibt I/O-Register

Hardware:

- liest/schreibt Bytes von/auf Datenträger

Hardware

- Speichermedien (z.B. Platten, SSD / Flash-Speicher, DVD, CD-ROM) mit Unterschieden; Beispiele
  - Blockgrößen:
    - Festplatten: 512 Bytes/Block
    - CDs: 2048 Bytes/Block
    - Flash: 4096 Bytes/Block
  - Nutzung der Blöcke
    - Flash-Speicher hat nur begrenzte Anzahl von Schreibzyklen pro Block => gleichmäßig beschreiben
    - Festplatten können auf benachbarte Blöcke jeweils schneller zugreifen
  - Größe der Medien (typ.)
    - CD-ROM: ca. 750 MByte
    - DVD: ca. 8,5 GByte
    - Festplatte: ca. 4 TByte
    - SSD: ca. 500 GByte

Block-Gerätetreiber

Beispiel: PC-IDE-Festplatten-Treiber (vereinfacht):

```
void block_read(uint32_t nr, uint8_t buf[]) {
    /* Read 1 data block. */
    IDE_COUNT = 1;

    /* Set block number. */
    IDE_BLK0 = (nr >>  0) & 0xff;
    IDE_BLK1 = (nr >>  8) & 0xff;
    IDE_BLK2 = (nr >> 16) & 0xff;
    IDE_BLK3 = (nr >> 24) & 0xff;

    /* Send command. */
    IDE_CMD = IDE_READ;

    /* Wait for READY bit set. */
    while (! (IDE_STATUS & IDE_READY)) { /* Wait... */ }

    /* Read data. */
    for (i = 0; i < 512; i++) {
        buf[i] = IDE_DATA;
    }
}
```

Dateisystem

- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Benutzer muss sich nicht um die Ansteuerung und Verwaltung verschiedener Speichermedien kümmern
  - einheitliche Sicht auf den Hintergrundspeicher
- Wesentliche Elemente eines Dateisystems:
  - Dateien (Files)
  - Verzeichnisse / Kataloge (Directories)
  - Partitionen (Partitions)

Dateisystem (2)

- Datei (File)
  - speichert Daten oder Programme
  - enthält Zusatzinformationen

Datei

- Verzeichnis / Katalog (Directory)
  - fasst Dateien (u. Verzeichnisse) zusammen
  - erlaubt Benennung der Dateien
  - ermöglicht Aufbau eines hierarchischen Namensraums

Verzeichnis

- Partition (Partition)
  - eine Menge von Verzeichnissen und deren Dateien
  - sie dienen zum physikalischen oder logischen Trennen von Dateimengen
    - physisch: Festplatte, Diskette
    - logisch: Teilbereich auf Platte oder CD

Partition

Dateien

- Kleinste Einheit, in der etwas auf den Hintergrundspeicher geschrieben werden kann.
- Unterscheidung:
  - eigentliche Daten (Bild, Text, Programm, ...)
  - Metadaten (Erstellungsdatum, Eigentümer, Zugriffsrechte, ...)

Metadaten / Dateiattribute:

Name: Symbolischer Name, vom Benutzer les- und interpretierbar

- z.B. `AUTOEXEC.BAT`

Typ: Für Dateisysteme, die verschiedene Dateitypen unterscheiden

- z.B. sequenzielle Datei, zeichenorientierte Datei, satzorientierte Datei

Ort: Wo werden die Daten physisch gespeichert?

- Nummern der Plattenblöcke

Dateiattribute (2)

Größe: Länge der Datei in Größeneinheiten (z.B. Bytes, Blöcke, Sätze)

- steht in engem Zusammenhang mit der Ortsinformation
- wird zum Prüfen der Dateigrenzen z.B. beim Lesen benötigt

Zeitstempel: z.B. Zeit und Datum der Erstellung, letzten Änderung

- für Backup, Entwicklungswerkzeuge, Benutzerüberwachung, ...

Rechte: Zugriffsrechte, z.B. Lese- und Schreibberechtigung

- z.B. nur für den Eigentümer schreibbar, für alle anderen nur lesbar

Eigentümer: Identifikation des Eigentümers

- eventuell eng mit den Rechten verknüpft
- Zuordnung beim Accouting (Abrechnung von Plattenplatz)

Operationen auf Dateien

- **Erzeugen (Create)**
  - Nötiger Speicherplatz wird angefordert
  - Verzeichniseintrag wird erstellt
  - Initiale Attribute werden gespeichert
- **Schreiben (Write)**
  - Identifikation der Datei
  - eventuell Nachfordern von Speicherplatz
  - Daten werden auf Platte geschrieben
  - eventuell Anpassung der Attribute (z.B. Länge der Datei, Zeitpunkt der letzten Änderung)
- **Lesen (Read)**
  - Identifikation der Datei
  - Daten werden von Platte gelesen
  - eventuell Anpassung der Attribute (z.B. Zugriffszeit)

Operationen auf Dateien

- **Positionieren** des Schreib-/Lesezeigers für die nächste Schreib- bzw. Leseoperation **(Seek)**
  - Identifikation der Datei
  - In vielen Systemen wird dieser Zeiger implizit bei Schreib- und Leseoperationen positioniert
  - Ermöglicht explizites Positionieren
- **Verkürzen (Truncate)**
  - Identifikation der Datei
  - Ab einer bestimmten Position (oder ab Anfang) wird der Inhalt der Datei gelöscht
  - eventuell Freigeben von Speicherplatz
  - Anpassung der Attribute (z.B. Länge der Datei, Zeitpunkt der letzten Änderung)
- **Löschen (Delete)**
  - Identifikation der Datei
  - Entfernen der Datei aus dem Verzeichnis und Freigabe der Plattenblöcke

Verzeichnisse / Kataloge

- Ein Verzeichnis gruppiert Dateien und evtl. weitere Verzeichnisse
- Gruppierungsalternativen
  - Verknüpfung mit Benennung
    - Verzeichnis enthält Namen und Verweise auf Dateien und andere Verzeichnisse (z.B. UNIX, Windows)
  - Gruppierung über Bedingung
    - Verzeichnis enthält Namen und Verweise auf Dateien, die einer bestimmten Bedingung gehorchen:
      – z.B. gleiche Gruppennummer in CP/M
      – z.B. eigenschaftsorientierte und dynamische Gruppierung in BeOS-BFS
- Verzeichnis ermöglicht das Auffinden von Dateien
  - Vermittlung zwischen externer und interner Bezeichnung (Dateiname – Plattenblöcke)

Operationen auf Verzeichnissen

- **Lesen** der Einträge (**Read, Read Directory)**
  - Daten des Verzeichnisses werden gelesen und meist eintragsweise zurückgegeben
- **Erzeugen** und **Löschen** der Einträge erfolgt implizit beim Anlegen bzw. Löschen der Dateien
- **Erzeugen** von Verzeichnissen **(Create, Create Directory)**
- **Löschen** von Verzeichnissen **(Delete, Delete Directory)**

Attribute von Verzeichnissen

- Die meisten Attribute von Dateien treffen auch auf Verzeichnissen zu
  - Name, Ortsinformation, Größe, Zeitstempel, Rechte, Eigentümer, ...

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Dateisystem am Beispiel Linux/UNIX

- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
- Dateiattribute
  - das Betriebssystem verwaltet zu jeder Datei eine Reihe von Attributen (Rechte, Größe, Zugriffszeiten, Datenblöcke, ...)
  - die Attribute werden in einer speziellen Verwaltungsstruktur, dem *Dateikopf*, gespeichert
    - Linux/UNIX: *Inode*
    - Windows NTFS: *Master File Table*-Eintrag
- Namensraum
  - flacher Namensraum: Inodes sind einfach durchnummeriert
  - hierarchischer Namensraum: Verzeichnisstruktur bildet Datei- und Pfadnamen in einem Dateibaum auf Inode-Nummern ab

Dateisystem-Struktur

- Struktur auf Medium (vereinfacht)

- `mkfs` legt leere Struktur an; `fsck` überprüft Struktur

Pfadnamen

- Baumstruktur

- Pfade
  - z.B. `/home/adam/datei`, `/tmp`, `../adam/datei`
  - `/` ist Trennsymbol (*Slash*)
  - beginnender `/` ist Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellen Verzeichnis

Pfadnamen (2)

- eigentliche „Baumstruktur”

- Beispiel Pfadauflösung „`../adam/datei`”:
  - 77 + „`../adam/datei`” $\rightsquigarrow$ 25 + „`adam/datei`”
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60

Pfadnamen (3)

- Es können mehrere Verweise (**Hard Links**) auf eine Datei existieren:

- Beispiel Pfadauflösung „`adam/datei`":
  - 25 + „`adam/datei`" $\rightsquigarrow$ 98 + „`datei`"
  - 98 + „`datei`" $\rightsquigarrow$ 60
- Beispiel Pfadauflösung „`eva/test`":
  - 25 + „`eva/test`" $\rightsquigarrow$ 77 + „`test`"
  - 77 + „`test`" $\rightsquigarrow$ 60
- Datei wird gelöscht, wenn keine Verweise auf sie mehr existieren

Pfadnamen (4)

- Es können mehrere symbolische Verweise (**Symbolic Links**) auf eine Datei oder ein Verzeichnis existieren:

- Beispiel Pfadauflösung „`eva/test`":
  - 25 + „`eva/test`" $\rightsquigarrow$ 77 + „`test`"
  - 77 + „`test`" $\rightsquigarrow$ 99 $\rightsquigarrow$ 77 + „`../adam/datei`"
  - 77 + „`../adam/datei`" $\rightsquigarrow$ 25 + „`adam/datei`"
  - 25 + „`adam/datei`" $\rightsquigarrow$ 98 + „`datei`"
  - 98 + „`datei`" $\rightsquigarrow$ 60
- Symbolischer Name kann auch bestehen, wenn Datei oder Verzeichnis noch nicht bzw. nicht mehr existiert.

Eigentümer und Rechte

- Eigentümer
  - Jeder Eigentümer wird durch eindeutige Nummer (UID) repräsentiert
  - Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die jeweils durch eine eindeutige Nummer (GID) repräsentiert werden
  - Eine Datei oder ein Verzeichnis ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
  - Lesen, Schreiben, Ausführen (nur vom Eigentümer änderbar)
  - Einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Verzeichnissen
  - Lesen, Schreiben (Anlegen und Löschen von Dateien/Verzeichnissen), Durchgangsrecht
  - Schreibrecht ist einschränkbar auf eigene Dateien

Inodes

- Attribute (Zugriffsrechte, Eigentümer, usw.) einer Datei, eines Verzeichnisses werden in **Inodes** gespeichert (vereinfacht):

```
int st_mode;        /* Typ und Zugriffrechte */
int st_nlink;       /* Anzahl der Hard Links */
int st_uid;         /* Eigentuemer */
int st_gid;         /* Gruppe */
long st_size;       /* Laenge der Datei in Bytes */
int st_block[...]; /* Liste der (indirekten) Bloecke */
time_t st_atime;    /* Letzter Lesezeitpunkt */
time_t st_mtime;    /* Letzter Modifikationszeitpunkt */
time_t st_ctime;    /* Letzte Aenderung an Attributen */
```

- Jede Inode hat eine Nummer und einen Speicherort (Platte/Partition):

```
int st_ino;         /* Inode-Nummer */
int st_dev;         /* Platte/Partition-Nummer */
```

Programmierschnittstelle für Inodes

- `stat`, `lstat` liefern Dateiattribute aus Inodes
- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - `path`: Pfadname
  - `buf`: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert
  - 0, wenn OK
  - -1, wenn Fehler (`errno`-Variable enthält Fehlernummer)
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerbehandlung...! */
printf("Inode-Nummer: %d\n", buf.st_ino);
```

Programmierschnittstelle für Verzeichnisse

- Verzeichnisse, Links verwalten
  - Erzeugen (eines leeren Verzeichnisses)

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

  - Löschen (eines leeren Verzeichnisses)

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

  - Hard Link anlegen

```
int link(const char *existing, const char *new);
```

  - Symbolischen Link anlegen

```
int symlink(const char *existing, const char *new);
```

  - Link löschen (und damit ggf. auch Datei)

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

  - Symbolischen Link auslesen

```
int readlink(const char *path, char *buf, int size);
```

Programmierschnittstelle für Verzeichnisse (2)

- Verzeichnisse lesen (Schnittstelle des Linux-Kerns)
  - open(2), getdents(2), close(2)
  - Linux-spezifisch und damit nicht portabel
- Verzeichnisse lesen (Schnittstelle der C-Bibliothek)
  - Verzeichnis öffnen

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

  - einen Eintrag lesen

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

  - Verzeichnis schließen

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

  - Struktur `struct dirent` (vereinfacht)

```
struct dirent {
    int d_ino;                  /* Inode-Nummer */
    char d_name[NAME_MAX + 1]; /* Name */
};
```

Verzeichnisse (3): opendir/closedir

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *path);
int closedir(DIR *dirp);
```

- Argument von `opendir`:
  - `path`: Verzeichnisname
- Rückgabewert von `opendir`:
  - Zeiger auf Datenstruktur vom Typ `DIR`, wenn OK
  - `NULL`, wenn Fehler (`errno`-Variable enthält Fehlernummer)

Verzeichnisse (4): readdir

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

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

- Argument:
  - `dirp`: Zeiger auf `DIR`-Datenstruktur
- Rückgabewert:
  - Zeiger auf Datenstruktur vom Typ `struct dirent`, wenn OK
  - NULL, wenn Verzeichnis zu Ende gelesen wurde (`errno`-Variable nicht verändert)
  - NULL, wenn Fehler aufgetreten ist (`errno`-Variable enthält Fehlercode)
- Hinweis: Der Speicher für `struct dirent` wird u.U. beim nächsten `readdir`-Aufruf überschrieben!

Verzeichnisse (5): Beispiel

- Ausgabe der Dateinamen im aktuellen Verzeichnis:

```
#include <sys/types.h>
#include <dirent.h>

DIR *dirp;
struct dirent *de;
int ret;

dirp = opendir(".");                    // akt. Verz. oeffnen
if (dirp == NULL) ...                   // Fehler

while (1) {
  errno = 0;
  de = readdir(dirp);                   // Eintrag lesen
  if (de == NULL && errno != 0) ... // Fehler
  if (de == NULL) break;                // Ende erreicht

  printf("%s\n", de->d_name);
}

ret = closedir(dirp);                   // Verz. schliessen
if (ret < 0) ...                        // Fehler
```

Programmierschnittstelle für Dateien

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <fcntl.h>

int open(const char *path, int flags, ...);

int close(int fd);

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Dateien (2): Beispiel

- Kopierprogramm

```
#include <fcntl.h>

int ret;

int src_fd = open("src", O_RDONLY);
if (src_fd < 0) ...          // Fehler
int dst_fd = open("dst", O_CREAT | O_TRUNC | O_WRONLY, 0777);
if (dst_fd < 0) ...          // Fehler

while (1) {
  char buf[1024];
  len = read(src_fd, buf, sizeof(buf));
  if (len < 0) ...           // Fehler
  if (len == 0) break;
  ret = write(dst_fd, buf, len);
  if (ret < 0) ...           // Fehler
}

ret = close(dst_fd);
if (ret < 0) ...             // Fehler
ret = close(src_fd);
if (ret < 0) ...             // Fehler
```

Dateien (3)

- `write`-Aufruf muss
  - den File-Deskriptor überprüfen (Datei geöffnet, Datei beschreibbar?)
  - die Pufferadresse/-länge überprüfen
  - den/die zu beschreibenden Blöcke des Mediums ermitteln
  - den/die Blöcke vom Medium lesen (wenn nicht ganzer Block geschrieben wird)
  - die entsprechenden Bytes im gelesenen Block überschreiben
  - den/die Blöcke auf das Medium zurückübertragen
  - die Attribute anpassen (Datum letzte Modifikation, Länge der Datei)
  - und ist ein Betriebssystem-Aufruf
- => `write` ist eine zeitlich teure Operation (`read` entsprechend)!
- => Besser: viele Bytes (am Besten: Vielfache der Blockgröße) am Stück lesen/schreiben
- => `fopen`-, `fclose`-, `fread`-, `fwrite`-, `getchar`-, `putchar`-, `fscanf`-, `fprintf`-, ... -Funktionen aus der C-Bibliothek benutzen!

Spezialdateien

- Periphere Geräte (Platte, Drucker, CD, Terminal, Scanner, ...) werden als Spezialdateien repräsentiert (`/dev/sda`, `/dev/lp0`, `/dev/cdrom0`, `/dev/tty`, ...)
- in Inode steht
  - Typ:
    - Block-orientiertes Gerät (Platte, CD, DVD, SSD, ...)
    - Zeichen-orientiertes Gerät (Drucker, Terminal, Scanner, ...)
  - statt Blocknummern:
    - Major-Number: Typ des Gerätes (Platte, Drucker, ...)
    - Minor-Number: Nummer des Gerätes (3. Drucker, 5. Terminal, ...)
- Öffnen der Geräte schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch Treiber hergestellt wird
- Geräte können dann mit `read`-, `write`- und `ioctl`-Operationen angesprochen werden

Spezialdateien (2): Beispiel

- Ausgabe auf Drucker

```
#include <linux/lp.h>
int fd, ret;

/* Verbindung zum Drucker 0 herstellen. */
fd = open("/dev/lp0", O_WRONLY);
if (fd < 0) ...

/* Druckerstatus abfragen. */
ret = ioctl(fd, LPGETSTATUS, &state);
if (ret < 0) ...
if (state & LP_POUTPA) {
  fprintf(stderr, "Out of paper!\n"); exit(1);
}

/* Auf Drucker schreiben. */
ret = write(fd, "Hallo, Drucker!\n\f", 17);
if (ret < 0) ...

/* Verbindung abbauen. */
ret = close(fd);
if (ret < 0) ...
```

Partitionen

- jede Festplatte kann als Ganzes ein Dateisystem enthalten
  - Festplatte entspricht dann einer Partition
- jede Festplatte kann aber auch unterteilt werden in mehrere Partitionen
  - erster Block der Platte enthält Partitionstabelle
  - Partitionstabelle enthält Informationen
    - wieviele Partitionen existieren
    - wie groß die jeweiligen Partitionen sind
    - wo sie beginnen
- jede Partition
  - wird durch eine Spezialdatei repräsentiert; z.B.
    - `/dev/sda`, `/dev/sdb` (ganze Platte)
    - `/dev/sda1`, `/dev/sda2`, `/dev/sdb1` (Teile der jeweiligen Platte)
  - enthält eigenes Dateisystem

Partitionen (2)

- Bäume der Partitionen können zu einem homogenen Dateibaum zusammengesetzt werden (Grenzen für Anwender nicht sichtbar!)
  - „Montieren" von Dateibäumen (*mounting*)
- Ein ausgezeichnetes Dateisystem ist das *Root File System*, dessen Wurzelverzeichnis gleichzeitig das Wurzelverzeichnis des Gesamtsystems ist
  - Andere Dateisysteme können mit dem `mount`-Befehl in das bestehende System hineinmontiert werden bzw. mit dem `umount`-Befehl wieder entfernt werden.
  - Über das *Network File System* (NFS) können auch Verzeichnisse anderer Rechner in einen lokalen Dateibaum hineinmontiert werden => Grenzen zwischen Dateisystemen verschiedener Rechner werden unsichtbar

Montieren des Dateibaumes

- Beispiel

```
mount /dev/sda3 /home
```

Montieren des Dateibaumes (2)

- Nach Ausführung des Montierbefehls

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Überblick

- **Mehrere** Programme, die
- **nebenläufig**,
- **dynamisch** gestartet/beendet
- über **definierte E/A-Funktionen**
- ihre Umgebung steuern.

Quelle: www.wikipedia.org

Jedes laufende Programm bekommt Hardware zugeteilt:

- CPU (Zeitanteile)
- Speicher (Teil des Gesamtspeichers)

und kann Betriebssystem-Kern-Funktionen aufrufen.

Definitionen

Programm: Folge von Anweisungen

Prozess: laufendes Programm mit seinen Daten

Hinweis: ein Programm kann sich mehrfach in Ausführung befinden!

Prozesse

- Definition „Prozess”: laufendes Programm mit seinen Daten
- eine etwas andere Sicht:

Mikrocontroller-Prozess	UNIX-/Windows/...-Prozess
Prozessor	Zeitanteile am echten Prozessor
Speicher	virtueller Speicher
Interrupts	Signale
E/A-Geräte	E/A-Betriebssystem-Funktionen

Prozesse (2)

- Mehrprogrammbetrieb („Multitasking”)
  - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
  - stehen weniger Prozessoren zur Verfügung, als Prozesse ausgeführt werden sollen, werden Zeitanteile der Rechenzeit an die Prozesse vergeben (**Time Sharing System**)
  - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit bekommt, trifft der Betriebssystem-Kern (**Scheduler**)
  - die Umschaltung zwischen Prozessen erfolgt durch den Betriebssystem-Kern (**Dispatcher**)
  - laufende Prozesse wissen nicht, an welchen Stellen auf andere Prozesse umgeschaltet wird

Prozesszustände

Ein Prozess befindet sich in einem der folgenden Zustände

Erzeugt: (New)
Prozess wurde erzeugt, besitzt aber noch nicht alle zum Laufen notwendigen Betriebsmittel

Bereit: (Ready)
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zu laufen

Laufend: (Running)
Prozess wird vom realen Prozessor ausgeführt

Blockiert: (Blocked)
Prozess wartet auf ein Ereignis (Fertigstellung einer Ein- oder Ausgabeoperation)

Beendet: (Terminated)
Prozess ist beendet, seine Betriebsmittel sind noch nicht alle freigegeben

Prozesszustände (2)

- Zustandsdiagramm mit Übergängen:

Nach Silberschatz, 1994

Prozesswechsel

- Jeder Prozess hat Zustand/Kontext
  - Registerinhalte des Prozessors
  - Inhalte der Speicherbereiche
  - offene Dateien, aktuelles Verzeichnis, ...
- Beim Prozesswechsel (Context Switch)
  - wird der Inhalt der Prozessorregister abgespeichert,
  - ein neuer Prozess ausgewählt,
  - die Ablaufumgebung des neuen Prozesses hergestellt
    - Umprogrammierung der MMU
    - Wechsel der offenen Dateien, des aktuellen Verzeichnisses, ...
  - werden die gesicherten Register des neuen Prozesses geladen.

Prozesswechsel

- Ablauf von zwei Prozessen in Benutzermodus und Kern mit Umschaltung

Prozesskontrollblock

- Prozesskontrollblock (Process Control Block – PCB)

  Datenstruktur des Betriebssystem-Kerns, die alle notwendigen Daten für einen Prozess enthält.

  Beispiel UNIX:
  - Prozess-ID (PID)
  - Prozesszustand (Laufend, Bereit, ...)
  - Register
  - Speicherabbildung
  - Eigentümer (UID, GID)
  - Wurzelverzeichnis, aktuelles Verzeichnis
  - offene Dateien
  - ...

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Beispiel: UNIX

- Überblick:

fork: Erzeugung von neuen Prozessen

exit: Terminieren von Prozessen

waitpid: Warten auf das Terminieren von Prozessen

Programmierschnittstelle

- Duplizieren des gerade laufenden Prozesses

```
#include <unistd.h>

pid_t fork(void);
```

- Terminieren des aktuellen Prozesses

```
#include <stdlib.h>

void exit(int status);
```

- Warten auf das Terminieren eines anderen Prozesses

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Beispiel

```
pid_t pid, ret;
int status;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(1);

case 0: /* Child */
    do_child_work();
    exit(13);

default: /* Parent */
    do_parent_work();
    ret = waitpid(pid, &status, 0);
    /*
     * In case of no error:
     * ret == pid
     * WIFEXITED(status) == 1
     * WEXITSTATUS(status) == 13
     */
    break;
}
```

Prozesserzeugung

- Der Kind-Prozess ist Kopie des Eltern-Prozesses
  - gleiches Programm
  - gleiche Daten (Variablen-Inhalte)
  - gleicher Programmzähler
  - gleiches aktuelles Verzeichnis, Wurzelverzeichnis
  - gleiche geöffnete Dateien
- einzige Unterschiede
  - verschiedene Prozess-IDs
  - Rückgabewert von `fork`

Ausführung von Programmen

- Das von einem Prozess ausgeführte Programm kann durch ein neues Programm ersetzt werden:

```
#include <unistd.h>

int execv(const char *path, char *argv[]);
int execl(const char *path, char *arg0, ...);
```

Beispiel:

```
...                /* Process A */
argv[0] = "ls";
argv[1] = "-l";
argv[2] = NULL;
execv("/bin/ls", argv);
/* Should not be reached. */
```

=>

```
...                /* Process A */
int
main(int argc, char *argv[])
{
    ...
}
```

Das zuvor laufende Programm wird beendet, das neue gestartet.

Es wird nur das Programm ausgetauscht.

Der Prozess läuft weiter!

Start eines Programms

- Beispiel: Start des Programms `./prog` mit Parametern `-a` und `-b`

... als Vordergrund-Prozess:

```
pid_t pid;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(EXIT_FAILURE);

case 0: /* Child */
    execl("./prog", "prog",
            "-a", "-b", NULL);
    perror("./prog");
    exit(EXIT_FAILURE);

default: /* Parent */
    waitpid(pid, NULL, 0);
    break;
}
```

... als Hintergrund-Prozess:

```
pid_t pid;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(EXIT_FAILURE);

case 0: /* Child */
    execl("./prog", "prog",
            "-a", "-b", NULL);
    perror("./prog");
    exit(EXIT_FAILURE);

default: /* Parent */
    /* No "waitpid" here! */
    break;
}
```

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Signale

- Mikrocontroller kann auf nebenläufige Ereignisse (Interrupts) mit Interrupt-Service-Routinen reagieren.
- Ähnliches Konzept auf Prozess-Ebene: **Signale**

Signale (2)

Interrupt: asynchrones Signal aufgrund eines „externen" Ereignisses

- CTRL-C auf der Tastatur gedrückt
- Timer abgelaufen
- Kind-Prozess terminiert
- ...

Exception: synchrones Signal, ausgelöst durch die Aktivität des Prozesses

- Zugriff auf ungültige Speicheradresse
- Illegaler Maschinenbefehl
- Division durch 0
- Schreiben auf eine geschlossene Kommunikationsverbindung
- ...

Kommunikation: ein Prozess will einem anderen ein Ereignis signalisieren

Signale (3)

CTRL-C:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
^C
~>
```

Illegaler Speicherzugriff:

```
int main(void)
{
    *(int *) NULL = 0;
    return 0;
}
```

```
~> ./test
Segmentation fault
~>
```

Inter-Prozess-Kommunikation:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
Terminated
~>
```

```
~> killall test
~>
```

Reaktion auf Signale

abort:

erzeugt Core-Dump (Speicher- und Registerinhalte werden in Datei `./core` geschrieben) und beendet Prozess
Standardeinstellung für alle Exceptions (zum nachträglichen Debuggen)

exit:

beendet Prozess (ohne Core-Dump)
Standardeinstellung für z.B. CTRL-C, Kill-Signal

ignore:

Signal wird ignoriert
Standardeinstellung für alle „unwichtigen" Signale (z.B. Kindprozess terminiert, Größe des Terminal-Fensters hat sich geändert)

...

Reaktion auf Signale (2)

...

handler:

Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

nie Standardeinstellung, da Programm-abhängig

stop:

stoppt Prozess

Standardeinstellung für Stop-Signal

continue:

setzt Prozess fort

Standardeinstellung für Continue-Signal

Reaktion über System-Aufruf (`sigaction`) änderbar

Programmierschnittstelle

- Einstellen der Signalbehandlungsfunktion
  (entspricht dem Setzen der ISR-Funktion)

```
#include <signal.h>

int sigaction(int sig, struct sigaction *new, struct sigaction *old);
```

`struct sigaction` enthält:

```
void (*sa_handler)(int sig); /* handler function
                                or SIG_DFL or SIG_IGN */
sigset_t sa_mask;            /* list of blocked signals while
                                handler is executed */
int sa_flags;                /* 0 or SA_RESTART ... */
```

- ...

Programmierschnittstelle

- ...
- Blockieren/Freigeben von Signalen
  (entspricht `cli()`, `sei()`)

```
#include <signal.h>

int sigprocmask(int how, sigset_t *nmask, sigset_t *omask);
```

  - `SIG_BLOCK`: angegebene Signale blockieren
  - `SIG_UNBLOCK`: angegebene Signale deblockieren
  - `SIG_SETMASK`: Signalmaske setzen
- Freigeben + Passives Warten auf Signal + wieder Blockieren
  (entspricht `sei(); sleep_cpu(); cli();`)

```
#include <signal.h>

int sigsuspend(sigset_t *mask);
```

- ...

Programmierschnittstelle

- ...
- Erstellen einer leeren Signal-Liste

```
#include <signal.h>

int sigemptyset(sigset_t *mask);
```

- Erstellen einer vollen Signal-Liste

```
int sigfillset(sigset_t *mask);
```

- Hinzufügen eines Signals zu einer Signal-Liste

```
int sigaddset(sigset_t *mask, int sig);
```

- Entfernen eines Signals aus einer Signal-Liste

```
int sigdelset(sigset_t *mask, int sig);
```

Programmierschnittstelle

- Typische Signale:

SIGSEGV: „Segmentation Fault" (ungültiger Speicherzugriff)
SIGINT: „Interrupt" (CTRL-C)
SIGALRM: „Alarm" (Timer abgelaufen)
SIGCHLD: „Child" (Kindprozess terminiert)
SIGTERM: „Terminate" (Abbruch des Prozesses; abfangbar)
SIGKILL: „Kill" (Abbruch des Prozesses; nicht abfangbar)

Signal-Beispiel 1

```c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // CTRL-C signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGINT, &sa, NULL);

  for (int i = 0; ; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

```c
static void handler(int sig)
{
    char s[] = "CTRL-C!\n";
    write(STDOUT_FILENO,
          s, strlen(s));
}
```

(Fehlerbehandlung weggelassen...)

```
~> ./test
...
146431
146432
146433
14^CCTRL-C!
6434
146435
146436
...
~>
```

Signal-Beispiel 2

```c
int main(void)
{
  // Call handler when
  // timer signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGALRM, &sa, NULL);

  // Send timer signal every sec.
  struct itimerval it;
  it.it_value.tv_sec  = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec  = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);

  // Wait for timer ticks.
  sigset_t mask;
  sigemptyset(&mask);
  while (1) sigsuspend(&mask);
}
```

```c
static void handler(int sig)
{
  write(STDOUT_FILENO,
        "Tick\n", 5);
}
```

(Fehlerbehandlung weggelassen...)

```
~> ./test
Tick
Tick
Tick
^C
~>
```

Signal-Beispiel 3

```c
#include <signal.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // I/O signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGIO, &sa, NULL);

  // Send I/O signal when
  // STDIN can be read.
  int flags = fcntl(STDIN_FILENO,
      F_GETFL);
  flags |= O_ASYNC;
  fcntl(STDIN_FILENO, F_SETFL,
      flags);

  while (1) sleep(1);
}
```

```c
static void handler(int sig)
{
  char buf[256];
  int len;

  // Read chars from STDIN.
  len = read(STDIN_FILENO, buf,
          sizeof(buf));

  // Handle chars in buf.
  ...
}
```

(Fehlerbehandlung weggelassen...)

Signale und Nebenläufigkeit

- Signale erzeugen Nebenläufigkeit innerhalb von Prozessen
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller

Nebenläufigkeitsbeispiel

```c
int main(void) {
  struct sigaction sa;
  struct itimerval it;
  /* Setup timer tick handler. */
  sa.sa_handler = tick;
  sa.sa_flags = 0;
  sigfillset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);
  /* Setup timer. */
  it.it_value.tv_sec = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);
  /* Print time while working. */
  while (1) {
    int s = sec, m = min, h = hour;
    printf("%02d:%02d:%02d\n", h, m, s);
    do_work();
  }
}
```

↓ hier Signal

```c
volatile int hour = 0;
volatile int min = 0;
volatile int sec = 0;

static void tick(int sig) {
  sec++;
  if (60 <= sec) {
    sec = 0; min++;
  }
  if (60 <= min) {
    min = 0; hour++;
  }
  if (24 <= hour) {
    hour = 0;
  }
}
```

```
~> ./test
...
23:59:59
00:59:59
00:00:00
...
```

← hier Problem!

(Fehlerbehandlung weggelassen...)

Lösungen Nebenläufigkeitsbeispiel

1. Lösung

```
sigset_t nmask, omask;

/* Block SIGALRM. */
sigemptyset(&nmask);
sigaddset(&nmask, SIGALRM);
sigprocmask(SIG_BLOCK,
            &nmask, &omask);

/* Get current time. */
int s = sec, m = min, h = hour;

/* Restore signal mask. */
sigprocmask(SIG_SETMASK,
            &omask, NULL);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

2. Lösung

```
/* Get current time. */
int s, m, h;
do {
  s = sec;
  m = min;
  h = hour;
} while (s != sec
      || m != min
      || h != hour);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

Weitere Lösungen existieren...

Nebenläufigkeitsprobleme

- Zusätzliches Problem:
  interne Funktionsweise von Bibliotheksfunktionen i.A. unbekannt
- Beispiel 1:
  `printf` fügt Zeichen in Puffer ein
  => Nutzung von `printf` im Hauptprogramm *und* in Signal-Behandlungsfunktion u.U. gefährlich
- Beispiel 2:
  `malloc` durchsucht Liste nach freiem Speicherbereich; `free` fügt Block in Liste ein
  => Nutzung von `malloc`/`free` im Hauptprogramm *und* in Signal-Behandlungsfunktion u.U. gefährlich
- Lösung:
  - Signale während der Ausführung kritischer Bereiche blockieren oder
  - keine unbekannten Bibliotheksfunktionen aus Signal-Behandlungsfunktionen heraus aufrufen

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Prozesse

- Mehrere Prozesse zur Strukturierung von Problemlösungen
  Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
  - z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
  - z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Web-Browser)
  - z.B. Client-Server-Anwendungen;
    pro Anfrage wird ein neuer Prozess gestartet (Web-Server)
- Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
  - früher nur bei Hochleistungsrechnern (Aerodynamik, Wettervorhersage)
  - durch Multicore-Systeme jetzt massive Verbreitung

Beispiel: Berechnung einer Wetterkarte

- Berechnung der Wetterkarte muss so schnell wie möglich erfolgen

Quelle: www.wetterdienst.de

- Ansatz: Mehrere Prozessoren berechnen jeweils einen Teil der Karte

Beispiel: Berechnung einer Wetterkarte (2)

- Z.B. Berechnung der Wetterkarte aufgeteilt auf 4 Prozessoren:

- Alle Prozessoren greifen auf einen gemeinsamen Speicher zu, in dem das Ergebnis berechnet wird.

Prozesse mit gemeinsamen Speicher

- Nutzung von gemeinsamen Speicher durch mehrere Prozesse

```
char *ptr = mmap(NULL, NBYTES, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (ptr == MAP_FAILED) ... // Fehler

for (i = 0; i < NPROCESSES; i++) {
    pid[i] = fork();
    switch (pid[i]) {
    case -1: ...            // Fehler
    case 0:
        do_work(i, ptr);
        _exit(0);
    default:;
    }
}
for (i = 0; i < NPROCESSES; i++) {
    ret = waitpid(pid[i], NULL, 0);
    if (ret < 0) ...        // Fehler
}

ret = munmap(ptr, NBYTES);
if (ret < 0) ...            // Fehler
```

Beispiel: Vektorlänge

- Berechnung der Länge/Norm eines N-Elemente-Vektors mit einem Prozess:

```
#include <math.h>

double
veclen(double vec[])
{
    double sum = 0.0;

    for (int i = 0; i < N; i++) {
        sum += vec[i] * vec[i];
    }

    return sqrt(sum);
}
```

Beispiel: Vektorlänge (2)

- Berechnung der Länge eines N-Elemente-Vektors mit vier Prozessen:

```
double veclen(double vec[]) {
    pid_t pid[4];
    double *ptr = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                       MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    for (int p = 0; p < 4; p++) {
        if ((pid[p] = fork()) == 0) {
            double sum = 0.0;
            for (int i = p * N / 4; i < (p + 1) * N / 4; i++)
                sum += vec[i] * vec[i];
            ptr[p] = sum;
            _exit(0);
        }
    }
    for (int p = 0; p < 4; p++)
        waitpid(pid[p], NULL, 0);
    double sum = 0.0;
    for (int p = 0; p < 4; p++)
        sum += ptr[p];
    munmap(ptr, 4096);
    return sqrt(sum);
}
```

Beispiel: Vektorlänge (3)

- Hinweis: Beispiel unvollständig
  - `#includes` fehlen
  - Fehlerbehandlung fehlt
  - ...
- Trotzdem sieht man
  - Programmierung sehr viel aufwändiger
  - Programm sehr viel unübersichtlicher
  - eigentlicher Algorithmus kaum noch erkennbar
- Ergebnis ernüchternd
  - Aufwand lohnt sich bei aktuellen Rechnern erst ab etwa $N = 100000$

Prozesse mit gemeinsamen Speicher (2)

Vorteil der obigen Lösung: in Multiprozessorsystemen sind **echt-parallele Abläufe möglich**

aber

jeder Prozess hat eigene Betriebsmittel

- Speicherabbildung
- Rechte
- offene Dateien
- Wurzel- und aktuelles Verzeichnis
- ...

=> **Prozess-Erzeugung, Prozess-Terminierung und Prozess-Umschaltungen sind teuer**

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Multiprozessor-Systeme

In Multiprozessorsystemen sind echt-parallele Abläufe möglich

aber

Prozess-Erzeugung, Prozess-Terminierung und Prozess-Umschaltungen sind teuer

Für die **Praxis** heißt das:

- nur selten Prozesse erzeugen/terminieren
- nicht mehr Prozesse erzeugen als echte Prozessoren vorhanden

oder

statt teuren Prozessen einfache **Fäden (Threads)** verwenden

Fäden in einem Prozess

Lösung: **mehrere** Aktivitätsträger (Fäden, Threads) in **einer** Ausführungsumgebung

- jeder **Faden (Thread)** repräsentiert einen eigenen aktiven Ablauf
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack (für lokale Variablen)
- gemeinsame **Ausführungsumgebung** stellt Menge von Betriebsmitteln zur Verfügung
  - Speicherabbildung
  - Rechte
  - offene Dateien
  - Wurzel- und aktuelles Verzeichnis
  - ...

Fäden in einem Prozess (2)

- Das Konzept eines Prozesses wird aufgespalten in eine **Ausführungsumgebung** und ein oder mehrere **Aktivitätsträger**
- Ein klassischer UNIX-Prozess ist ein Aktivitätsträger in einer Ausführungsumgebung

Fäden in einem Prozess (3)

- Erzeugen/Terminieren eines Fadens in einem Prozess erheblich billiger als das Erzeugen/Terminieren eines Prozesses (keine eigenen Betriebsmittel)
- Umschalten zwischen Fäden innerhalb eines Prozesses erheblich billiger als das Umschalten zwischen Prozessen
  - es müssen nur die Register und der Programmzähler gewechselt werden (entspricht in etwa einem Funktionsaufruf)
  - Speicherabbildung muss nicht gewechselt werden (Cache-Inhalte bleiben gültig!)

Koordinierung / Synchronisierung

- Fäden arbeiten nebenläufig/parallel und haben gemeinsamen Speicher
  => alle von Unterbrechungen und Signalen bekannten Probleme beim Zugriff auf gemeinsame Daten treten auch bei Fäden auf
- Unterschied zwischen Fäden und Interrupt-Service-Routinen bzw. Signal-Handler-Funktionen:
  - „Haupt-Faden" der Anwendung und eine ISR bzw. ein Signal-Handler sind nicht gleichberechtigt
    - ISR bzw. Signal-Handler unterbricht den Haupt-Faden aber ISR bzw. Signal-Handler werden nicht unterbrochen
  - zwei Fäden sind gleichberechtigt
    - ein Faden kann jederzeit zugunsten eines anderen unterbrochen werden (Scheduler) oder parallel zu einem anderen laufen (MPS)

=> Unterbrechungen sperren oder Signale blockieren hilft nicht!

Koordinierung / Synchronisierung (2)

- Grundlegende Probleme
  - gegenseitiger Ausschluss (**Koordinierung**)
    Beispiel:
    Ein Faden möchte einen Datensatz lesen und verhindern, dass ein anderer Faden ihn währenddessen verändert.
  - gegenseitiges Warten (**Synchronisierung**)
    Beispiel:
    Ein Faden wartet auf andere Fäden, die jeweils Teilergebnisse berechnen sollen, die dann zusammengefasst werden.

Koordinierung / Synchronisierung (3)

- Komplexe Koordinierungs-/Synchronisierungsprobleme (Beispiel)
  - **Bounded Buffer**:
    Fäden schreiben Daten in Pufferspeicher, andere entnehmen Daten; kritische Situationen:
    - Zugriff auf den Puffer
    - Puffer leer / voll

Element einfügen:

- Warten, bis Platz im Puffer
- Warten, bis kein anderer Faden mehr den Puffer liest/schreibt
- Puffer beschreiben
- Signalisieren, dass neues Element im Puffer

Element herausnehmen:

- Warten, bis Element im Puffer
- Warten, bis kein anderer Faden mehr den Puffer liest/schreibt
- Puffer auslesen
- Signalisieren, dass Platz im Puffer

Gegenseitiger Ausschluss (Mutual Exclusion)

- Einfache Implementierung durch **mutex**-Variablen

```
volatile int m = 0; /* 0: free; 1: locked */
volatile int counter = 0;
```

```
...        /* Thread 1 */
lock(&m);
counter++;
unlock(&m);
...
```

```
...        /* Thread 2 */
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
...
```

Nur der Faden, der `lock` aufgerufen hat, darf `unlock` aufrufen!

- Realisierung (nur konzeptionell!)

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

```
void unlock(volatile int *m) {
    *m = 0;
}
```

`lock` (und ggf. `unlock`) müssen **atomar** ausgeführt werden!

Zählende Semaphore

- Ein `Semaphor` (griech. Zeichenträger) ist eine Datenstruktur mit zwei Operationen (nach *Dijkstra*):
  - `P-Operation` (*proberen; passeren; wait; down*)

```
void P(volatile int *s) {
    while (*s <= 0) {
        /* Wait/sleep... */
    }
    *s -= 1;
}
```

  - `V-Operation` (*verhogen; vrijgeven; signal; up*)

```
void V(volatile int *s) {
    *s += 1;
    /* Wakeup... */
}
```

`P` und `V` müssen **atomar** ausgeführt werden!

`P` und `V` müssen nicht vom selben Faden aufgerufen werden.

Bounded Buffer (2)

Bounded-Integer-Buffer-Beispiel:

```
#define N 1000
volatile int mutex = 0;
volatile int alloc = 0, free = N;
volatile int head = 0, tail = 0;
volatile int buf[N];
```

Element einfügen:

```
void put(int x) {

    P(&free);
    lock(&mutex);
    buf[head] = x;
    head = (head + 1) % N;
    unlock(&mutex);
    V(&alloc);

}
```

Element herausnehmen:

```
int get(void) {
    int x;
    P(&alloc);
    lock(&mutex);
    x = buf[tail];
    tail = (tail + 1) % N;
    unlock(&mutex);
    V(&free);
    return x;
}
```

Spin Lock vs. Sleeping Lock

- **Spin Lock**
  - aktives Warten, bis Mutex-Variable frei (= 0) wird
  - entspricht konzeptionell einem Pollen
  - Faden bleibt im Zustand „laufend"

  Problem: wenn nur ein Prozessor verfügbar ist, wird Rechenzeit vergeudet, bis durch den Scheduler eine Umschaltung erfolgt

  - nur ein anderer, laufender Faden kann den Mutex freigeben
- **Sleeping Lock**
  - passives Warten
  - Faden geht in den Zustand „blockiert"
  - im Rahmen von `unlock` wird der blockierte Faden in den Zustand „bereit" zurückgeführt

  Problem: bei sehr kurzen kritischen Abschnitten ist der Aufwand für das Blockieren/Aufwecken und die Umschaltung unverhältnismäßig teuer

Implementierung Spin Lock

- zentrales Problem: Atomarität von mutex-Abfrage und -Setzen

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

kritischer Abschnitt

- Lösung: spezielle Maschinenbefehle, die atomar eine Abfrage und eine Modifikation einer Hauptspeicherzelle ermöglichen
  - *Test-and-Set*, *Compare-and-Swap*, *Load-Link/Store-Conditional*, ...

Implementierung Sleeping Lock

- zwei Probleme:

1. Konflikt mit einer zweiten `lock`-Operation:
   Atomarität von mutex-Abfrage und -Setzen

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

kritischer Abschnitt 1

2. Konflikt mit einem `unlock`: *lost-wakeup*-Problem

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

kritischer Abschnitt 2

- Ursachen:

1. Prozessumschaltung während der `lock`-Operation
2. Echt-parallel laufende `lock`- und/oder `unlock`-Operationen

Implementierung Sleeping Lock (2)

- Behebung von Ursache (1):
  Prozessumschaltungen verhindern
  - Prozessumschaltung ist eine Funktion des BS-Kerns
    - erfolgt im Rahmen eines BS-Aufrufs (z.B. `exit`)
    - oder im Rahmen einer Unterbrechungs-Behandlung (z.B. Zeitscheiben-Unterbrechung)

  => `lock`/`unlock` werden ebenfalls im BS-Kern implementiert; BS-Kern mit Unterbrechungs-Sperre

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    *m = 0;
    wakeup_waiting_threads();
    sei();
    leave_OS();
}
```

Implementierung Sleeping Lock (3)

- Behebung von Ursache (2):
  Parallele Ausführung auf anderem Prozessor verhindern

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    spin_unlock();
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    *m = 0;
    wakeup_waiting_threads();
    spin_unlock();
    sei();
    leave_OS();
}
```

- `P()` und `V()` ähnlich

Überblick: Teil D Betriebssystemabstraktionen

21 Ergänzungen – Zeiger

22 Ergänzungen – Ein-/Ausgabe

23 Ergänzungen – Fehlerbehandlung

24 Betriebssysteme

25 Dateisysteme – Einleitung

26 Dateisysteme – UNIX

27 Programme und Prozesse

28 Programme und Prozesse – UNIX

29 Signale

30 Multiprozessoren

31 Nebenläufige Fäden

32 Nebenläufige Fäden – Praxis

Beispiel: POSIX Threads (pthread)

- Programmierschnittstelle standardisiert: **pthread-Bibliothek** (IEEE-POSIX-Standard P1003.4a)
- pthread-Schnittstelle (Basisfunktionen):

  `pthread_create:` Faden erzeugen
  `pthread_exit:` Faden beendet sich selbst
  `pthread_join:` auf Ende eines Fadens warten
  `...:`
- Funktionen in pthread-Bibliothek zusammengefasst

```
gcc ... -pthread ...
```

pthread-Schnittstelle

- Faden-Erzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *param);
```

- Parameter

  `tid`: Zeiger auf Variable, in der die Faden-ID abgelegt werden soll.

  `attr`: Zeiger auf Attribute (z.B. Stack-Größe) des Fadens. `NULL` für Standard-Attribute.

  `func`, `param`: Der neu erzeugte Faden führt die Funktion `func` mit dem Parameter `param` aus.

- Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode (ähnlich `errno`) zurückgeliefert.

pthread-Schnittstelle

- Faden beenden (bei `return` aus `func` oder):

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Der Faden wird beendet und `retval` wird als Rückgabewert zurückgeliefert (siehe `pthread_join`.

- Auf Faden warten und `pthread_exit`-Status abfragen:

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **retvalp);
```

Wartet auf den Faden mit der Faden-ID `tid` und liefert dessen Rückgabewert über `retvalp` zurück.
Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode (ähnlich `errno`) zurückgeliefert.

pthread-Beispiel

- Beispiel (Multiplikation Matrix mit Vektor; $\vec{c} = A\vec{b}$):

```
double a[100][100], b[100], c[100];

static void *mult(void *ci) {
    int i = (double *) ci - c;

    double sum = 0.0;
    for (int j = 0; j < 100; j++) {
        sum += a[i][j] * b[j];
    }
    c[i] = sum;
    return NULL;
}

int main(void) {
    pthread_t tid[100];

    for (int i = 0; i < 100; i++) {
        pthread_create(&tid[i], NULL, mult, &c[i]);
    }
    for (int i = 0; i < 100; i++) {
        pthread_join(tid[i], NULL);
    }
}
```

pthread-Koordinierung und -Synchronisierung

- Koordinierung durch Mutex-Variablen
  - Erzeugung von Mutex-Variablen

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

  - `lock`-Operation

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *m);
```

  - `unlock`-Operation

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *m);
```

pthread-Beispiel

- Mutex-Beispiel:

```
volatile int counter = 0;
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

```
...      /* Thread 1 */
pthread_mutex_lock(&m);
counter++;
pthread_mutex_unlock(&m);
...
```

```
...      /* Thread 2 */
pthread_mutex_lock(&m);
printf("counter = %d\n", counter);
counter = 0;
pthread_mutex_unlock(&m);
...
```

pthread-Koordinierung und -Synchronisierung (2)

- Synchronisierung durch Bedingungs-Variablen (Condition Variable)
  - auf eine Bedingung kann gewartet werden (sleep)
  - eine Bedingung kann signalisiert werden (wakeup)
  - Erzeugung einer Condition-Variablen

```
pthread_cond_t c;
pthread_cond_init(&c, NULL);
```

  - auf eine Bedingung warten

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

  - eine Bedingung signalisieren

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *c);
int pthread_cond_broadcast(pthread_cond_t *c);
```

`pthread_cond_signal` weckt *einen* Faden, `pthread_cond_broadcast` weckt *alle* auf die Bedingung wartenden Fäden auf

pthread-Beispiel (2)

- Beispiel: zählende Semaphore

```
pthread_mutex_t m;
pthread_cond_t c;

pthread_mutex_init(&m, NULL);
pthread_cond_init(&c, NULL);
```

```
void P(volatile int *s) {
    pthread_mutex_lock(&m);
    while (*s == 0) {
        pthread_cond_wait(&c, &m);
    }
    *s -= 1;
    pthread_mutex_unlock(&m);
}
```

```
void V(volatile int *s) {
    pthread_mutex_lock(&m);
    *s += 1;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```

Fäden, Koordinierung, Synchronisierung in Java

- Faden-Konzept, Koordinierung und Synchronisierung in Java integriert
- Erzeugung von Fäden über die Thread-Klasse; Beispiel:

```
class MyClass implements Runnable {
    public void run() {
        System.out.println("Hello!");
    }
}
...
MyClass o = new MyClass();    // create object
Thread t1 = new Thread(o);    // create thread to run in o
t1.start();                   // start thread
Thread t2 = new Thread(o);    // create second thread
t2.start();                   // start second thread
```

Fäden, Koordinierung, Synchronisierung in Java (2)

- Koordinierung und Synchronisierung über jedes beliebige Objekt
  - Koordinierung über `synchronized`-Blöcke

```
synchronized(obj) {
    ...
}
```

    Ein solcher Block ruft zu Block-Beginn ein `lock` auf das Objekt `obj` auf, führt die angegebenen Anweisungen aus, und ruft vor dem Verlassen des Blockes das entsprechende `unlock` auf.
  - Synchronisierung über `wait`, `notify` und `notifyAll`

    `obj.wait()`: wartet auf die Signalisierung einer Bedingung auf dem angegebenen Objekt `obj`.

    `obj.notify()`: signalisiert eine Bedingung auf dem angegebenen Objekt `obj` an *einen* wartenden Faden.

    `obj.notifyAll()`: signalisiert eine Bedingung auf dem angegebenen Objekt `obj` *allen* wartenden Fäden.

Fäden, Koordinierung, Synchronisierung in Java (3)

- Beispiel Koordinierung und Synchronisierung:

```
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public void P() {
        synchronized(this) {
            while (s == 0)
                this.wait();
            s--;
        }
    }
    public void V() {
        synchronized(this) {
            s++;
            this.notifyAll();
        }
    }
}
```

Entspricht dem pthread-Beispiel...

Fäden, Koordinierung, Synchronisierung in Java (4)

- Vereinfachte Schreibweise (entspricht „Monitor"-Konzept):

```
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public synchronized void P() {
        while (s == 0) {
            wait();
        }
        s--;
    }
    public synchronized void V() {
        s++;
        notifyAll();
    }
}
```

Systemnahe Programmierung in C

Teil E Speicher

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Skriptum_handout

Überblick: Teil E Speicher

33 Dynamische Speicherallokation

34 Speicherorganisation

35 Speicherorganisation – Stack

Größe von Typen/Objekten

- Größe elementarer Typen bekannt; z.B.:
  - `char`: 1 Byte
  - `uint16_t`: 2 Byte
  - `uint32_t`: 4 Byte
  - ...
- Größe von Datenstrukturen:
  - **Felder**: N-elementiges Array braucht N-mal den Platz eines Elements
  - **Strukturen**: Struktur braucht (mindestens) den Platz aller Elemente
  - **Unions**: Union braucht den Platz des größten Elements

Größe ermittelbar mit:

```
sizeof type
```

bzw.

```
sizeof var
```

`sizeof`-Operator liefert Wert vom Typ `size_t`.

Dynamische Speicherallokation: Heap

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

```
#include <stdlib.h>

int *intArray(size_t n) { /* alloc int[n] array */
    return (int *) malloc(n * sizeof int);
}

void main(void) {
    int *array = intArray(100); /* alloc memory for 100 ints */
    if (array == NULL) { /* error handling... */ }
    ...
    array[99] = 4711; /* use array */
    ...
    free(array); /* free allocated block (** IMPORTANT! **) */
}
```

Dynamische Speicherallokation: Verkette Liste

Beispiel: Allozieren eines Listenelementes und Einfügen in Liste:

```
struct list_elem {
    struct list_elem *next;
    int num;
}
struct list_elem *head = NULL;

void add_to_list(int num) {
    struct list_elem *elem;

    /* Allocate memory for element. */
    elem = (struct list_elem *) malloc(sizeof(*elem));
    if (elem == NULL) { /* Error handling... */ }

    /* Fill object. */
    elem->num = num;

    /* Add element to list. */
    elem->next = head;
    head = elem;
}
```

Dynamische Speicherallokation: Verkette Liste (Forts.)

Beispiel: Herausnehmen eines Listenelementes aus Liste und Freigeben:

```
int remove_from_list(void) {
    /* Get element. */
    struct list_elem *elem = head;

    if (elem == NULL) {
        return -1; /* List empty. */
    }

    /* Remove element from list. */
    head = elem->next;

    /* Get info from element. */
    int num = elem->num;

    /* Free memory of element. */
    free(elem);

    return num;
}
```

Dynamische Speicherallokation: Strings

Beispiel: Duplizieren eines Strings:

```
char *strdup(const char *s) {
    /* Calculate size of string. */
    /* ** IMPORTANT **: "+ 1" for '\0' at end! */
    size_t size = strlen(s) + 1;

    /* Allocate memory. */
    char *p = (char *) malloc(size * sizeof(char));
    if (p == NULL) {
        return NULL; /* Out of memory. */
    }

    /* Copy string. */
    strcpy(p, s);

    return p;
}
```

Überblick: Teil E Speicher

33 Dynamische Speicherallokation

34 Speicherorganisation

35 Speicherorganisation – Stack

Speicherorganisation

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

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

Wo kommt der Speicher für diese Variablen her?

- **Statische Allokation** – Reservierung beim Übersetzen / Linken
  - Betrifft alle globalen/statischen Variablen, sowie den Code ↪ 12–5
  - Allokation durch Platzierung in einer Sektion

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

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

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

Speicherorganisation auf einem μC

Speicherorganisation mit Betriebssystem

Speicherorganisation mit Betriebssystem (Forts.)

Speicherorganisation mit Betriebssystem (Forts.)

Überblick: Teil E Speicher

33 Dynamische Speicherallokation

34 Speicherorganisation

35 Speicherorganisation – Stack

Dynamische Speicherallokation – Stack

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem Stack (Stapel, Keller) verwaltet
- Stack ist Teil des normalen Hauptspeichers
- Prozessorregister `sp` „**Stack Pointer**" zeigt immer auf das zuletzt abgelegte Datum (architekturabhängig)
- Stack „wächst" „von oben nach unten" (architekturabhängig)

=> `sp` zeigt damit immer auf den Anfang des genutzten Teil des Stacks

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Anlegen von a, b, c

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Schreiben von a, b

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Berechnen der Parameter

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Speichern der Rückkehradresse

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Start f1

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Anlegen von i[0]...i[2]

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Inkrementieren von x

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Berechnen des Parameters

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Speichern der Rückkehradresse

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Start f2

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Anlegen von m

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Schreiben von m

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Entfernen von m

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Rücksprung

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Entfernen des Parameters

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Entfernen von i[0]...i[2]

Dynamische Speicherallokation – Stack

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

    a = 10;
    b = 20;
    f1(a, b + 1);
    b = f3(a);
    return b;
}

void f1(int x, int y) {
    int i[3];
    x++;
    f2(x);
}

void f2(int z) {
    int m;
    m = 100;
}

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

Rückkehr

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Entfernen der Parameter

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Berechnen des Parameters

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Speichern der Rückkehradresse

Dynamische Speicherallokation – Stack

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

   a = 10;
   b = 20;
   f1(a, b + 1);
   b = f3(a);
   return b;
}

void f1(int x, int y) {
   int i[3];
   x++;
   f2(x);
}

void f2(int z) {
   int m;
   m = 100;
}

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

Start f3