Systemnahe Programmierung in C

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Sommersemester 2026

http://sys.cs.fau.de/lehre/ss26

Skriptum_handout

Referenzen

[1] *ATmega328PB 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*. Atmel Corporation. Okt. 2015. URL: https://sys.cs.fau.de/extern/lehre/ss25/spic/uebung/spicboard/Atmel-42397-8-bit-AVR-Microcontroller-ATmega328PB_Datasheet.pdf.

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

[2] Manfred Dausmann, Ulrich Bröckl, Dominic Schoop u. a. *C als erste Programmiersprache: Vom Einsteiger zum Fortgeschrittenen*. (Als E-Book aus dem Uninetz verfügbar; PDF-Version unter /proj/i4spic/pub/material/). Vieweg+Teubner, 2010. ISBN: 978-3834812216. URL: https://www.springerlink.com/content/978-3-8348-1221-6/#section=813748&page=1.

[3] 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] Jens Gustedt. *Modern C*. Manning, 2024. URL: https://inria.hal.science/hal-02383654.

Skriptum_handout

Referenzen (Forts.)

[5] Brian "Beej Jorgensen" Hall. *Beej's Guide to C Programming*. 2025. URL: https://beej.us/guide/bgc/.

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

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

[8] 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.

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

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

Skriptum_handout

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

Lecture Overview

Part A: Konzept und Organisation

1 Einführung

2 Organisation

Part 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

Part 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

Skriptum_handout

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

Lecture Overview (2)

Part 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

Part 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, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Sommersemester 2026

http://sys.cs.fau.de/lehre/ss26

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, ...)

Introduction

- **Deepen** knowledge of concepts and techniques of computer science and software development
  - Starting point: Algorithms, Programming, and Data Representation
  - Main focus: System-Level Programming (SLP) in C

- **Development** of software in C for a µ-Controller (µC) and an operating-system platform (Linux)
  - SPiCboard learning development platform with an ATmega-µC
  - Practical experience in hardware and system-level software development

- **Understanding** of language and hardware basics for the development of system-level software
  - Being able to understand and assess the language C and
  - Dealing with concurrency and hardware orientation
  - Dealing with the abstractions of an operating system (files, processes, . . .)

Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren werden in **eingebetteten Systemen** verbaut [9]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und µ-Controller, **8-Bit oder kleiner** [9, 10]

Motivation: Eingebettete Systeme

- **Omnipräsent:** **98–99 Prozent** aller Prozessoren werden in **eingebetteten Systemen** verbaut [9]
- **Kostensensitiv:** **70–80 Prozent** aller produzierten Prozessoren sind DSPs und µ-Controller, **8-Bit oder kleiner** [9, 10]
- **Relevant:** **25 Prozent** der Stellenanzeigen für EE-Ingenieure enthalten die Stichworte *embedded* oder *automotive* (`http://stepstone.com`)

Motivation: Embedded Systems

- **Omnipresent:** **98–99 percent** of processors are being used in **embedded systems** [9]
- **Cost-sensitive:** **70–80 percent** of all produced processors are DSPs and µ-Controller, based on **8-bit architecture or smaller** [9, 10]

Motivation: Embedded Systems

- **Omnipresent:** **98–99 percent** of processors are being used in **embedded systems** [9]
- **Cost-sensitive:** **70–80 percent** of all produced processors are DSPs and µ-Controller, based on **8-bit architecture or smaller** [9, 10]
- **Relevant:** **25 percent** of job offers for EE engineers do contain the terms *embedded* or *automotive* (`http://stepstone.com`)

Motivation: Embedded Systems

Source: IC Insights 2014 *McClean Report*

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	-	**1,02**
ATTINY2313	2 KiB	128 B	18	1/1	-	1	-	-	**0,90**
ATMEGA48	4 KiB	512 B	23	2/1	1	1	8*10	6	**2,95**
ATMEGA16	16 KiB	1024 B	32	2/1	1	1	8*10	4	**2,10**
ATMEGA32	32 KiB	2048 B	32	2/1	1	1	8*10	4	**2,20**
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	8	**6.84**
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	8	**6.10**
ATMEGA256	256 KiB	8192 B	86	2/2	4	1	16*10	16	**14.50**

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: The ATmega-µC Family (8-bit)

Type	Flash	SRAM	IO	Timer 8/16	UART	SPI	ADC	PWM	EUR
ATTINY13	1 KiB	64 B	6	1/-	-	-	1*4	-	**1,02**
ATTINY2313	2 KiB	128 B	18	1/1	-	1	-	-	**0,90**
ATMEGA48	4 KiB	512 B	23	2/1	1	1	8*10	6	**2,95**
ATMEGA16	16 KiB	1024 B	32	2/1	1	1	8*10	4	**2,10**
ATMEGA32	32 KiB	2048 B	32	2/1	1	1	8*10	4	**2,20**
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	8	**6.84**
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	8	**6.10**
ATMEGA256	256 KiB	8192 B	86	2/2	4	1	16*10	16	**14.50**

ATmega variants (selection) and market prices (Reichelt Elektronik, April 2026)

- Becomes visible: **resource scarcity**
  - **Flash** (memory for program code and constant data) is **scarce**
  - **RAM** (memory for runtime variables) is **extremely scarce**
  - few bytes “wasted” ⇝ significantly higher cost per piece

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 [6, 8]

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

Motivation: C as a Language

- System-level software development mostly uses **C**.
  - **Why C?** (and not Python/Java/Scala/<*favorite language*>)
- C stands for a multitude of important features
  - **Runtime efficiency** (CPU)
    - Translated C code runs on the processor directly
    - No checks for programming errors at runtime
  - **Space efficiency** (memory)
    - Code and data can be stored rather compactly
    - No checks for data access at runtime
  - **Directness** (machine orientation)
    - C allows for direct access to memory and registers
  - **Portability**
    - There is a C compiler for **every platform**
    - C was invented (1973), to implement the OS UNIX portable [6, 8]

⇝ **C** is the **lingua franca** of system-level programming!

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 ↔ Betriebssystemplattform erfahren
  - Konzepte und Techniken an kleinen Beispielen lehr- und erfahrbar
  - **Hohe Relevanz** für die Zielgruppe (EEI, ME, ...)

Motivation: SLP – Syllabus and Concept

- **Teaching objective:** system-level programming in C
  - This is a really broad field: hardware programming, operating systems, middleware, databases, distributed systems, compiler construction, ...
  - Additionally, we have the goal of learning the language C itself

- **Approach**
  - Concentration on two domains
    - µC programming
    - Software development for Linux system interface
  - Experience contrast µC environment ↔ operating system (OS)
  - Concepts and techniques taught and experienced with the help of various examples
  - **High relevance** for the target audience (electrical & mechanical engineering, ...)

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...

Motivation: SLP

At the end of the lecture, everyone should be able to assess,

- what a µC can (not) do,
- how labor-intensive & beneficial µC programming is,
- what an OS does (not) provide,
- how labor-intensive & beneficial it is to use a µC.

Everyone should be able to work with a computer scientist, if necessary...

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!**

Lecture Notes

- This handout of the lecture notes will be provided online.
  - Chapters are available as individual files
  - The handout contains (some) additional information

- **However, the handout cannot be used as a substitute for making your own notes!**

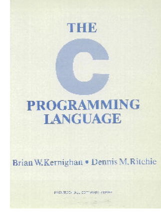
Literaturempfehlungen

[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`

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

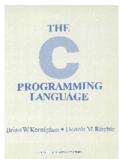
Brian W. Kernighan und Dennis MacAlistair Ritchie. *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 1988. ISBN: 978-8120305960



Literature Recommendations

[7] standard book (more suitable as a reference):

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

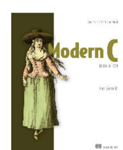


[5] open-access book (guide for audience with basic programming knowledge):

Brian "Beej Jorgensen" Hall. *Beej's Guide to C Programming*. 2025. URL: https://beej.us/guide/bgc/

[4] open-access book (covers modern C standards):

Jens Gustedt. *Modern C*. Manning, 2024. URL: https://inria.hal.science/hal-02383654



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 ↔ Sprachkonstrukte
    - Unterbrechungen (*interrupts*) und Nebenläufigkeit
  - Softwareentwicklung auf „einem Betriebssystem“ (Linux)
    - Betriebssystem als Ausführungsumgebung für Programme
    - Abstraktionen und Dienste eines Betriebssystems

Lecture

- Content and topics
  - Basic concepts of system-level programming
  - Introduction to the programming language C
    - differences compared to Python/Java
    - modular concept
    - pointers and pointer arithmetic
  - **“Bare-metal”** software development directly on hardware (ATmega µC)
    - mapping of storage ↔ language constructs
    - **interrupts** & **concurrency**
  - Software development on **operating system (OS)**: Linux
    - operating system as a runtime environment for programs
    - abstractions and services of an operating system

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!**

Lecture

- 36 sections
  - slides on the web server `sys.cs.fau.de`
  - dates: see **semester overview**
  - $\rightarrow$ requirement for successful handling of the exercises
- Questions on the lecture
  - ideally ask **immediately**
  - in following lecture
- Q&A at the end of the term
- **Lecture does not replace the tutorials and hands-on exercises!**

Vorlesung

- Screencasts aus SS20
- Vorlesungsaufzeichnung aus SS19

existieren...

siehe

- WWW-Seite
- Video-Portal Uni

Teil A: Konzept und Organisation 1.2 - Kapitel 2: Organisation [SS2020]

Dr. Volkmar Sieh
2020-04-20
IdM-login

...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
    - Abgabegespräche für Bonuspunkte
- Termin:
  - Anmeldung über Waffel ab Donnerstag 18.04.2024 um 18:00 Uhr (siehe Webseite)
  - Bitte für T01 anmelden
  - Anmeldung freiwillig, nur für Bonuspunkte nötig
  - **Separate** Übung für SLP

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

Exercises

- Tutorial and hands-on exercise
  - Tutorial (Tafelübung)
    - distribution of and additional information for the programming assignments
    - joined development of an outline for the solution
    - discussion of the solution the subsequent week
  - Hands-on exercise (Rechnerübung)
    - independent programming
    - working with development tools
    - support from an exercise supervisor
    - discussion of your submission for bonus points
- Appointment:
  - registration via Waffel (see website)
  - **seperate** group only for SLP

Valid login for the Linux-CIP (computer lab) required for participation in exercises!

Übungen

WARNING!

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

WARNING!

Exercises

WARNING!

There will be **no tutorials & exercises** during the winter term for students who failed the exam

WARNING!

Programmieraufgaben

- Praktische Umsetzung des Vorlesungsstoffs
  - Acht Programmieraufgaben ↪ ??
  - Teilweise Gruppenabgaben
- Lösungen per SPiC-IDE abgeben
  - Lösung wird durch Skripte überprüft
  - Ein Tutor wird die Abgabe mit Euch besprechen
  - LLMs sind zur Bearbeitung **nicht** gestattet (ChatGPT, Perplexity, etc.)

★ Abgabe der Übungsaufgaben ist freiwillig; ↪ 2–12
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!**

Programming Assignments

- Practically apply lecture contents
  - **eight programming assignments** ↪ ??
  - including assignments in groups
- Solutions must be submitted in the SPiC-IDE
  - your solution is validated with the help of scripts
  - a teaching assistant will discuss your submission (more info in the upcoming tutorial)
  - using LLMs to solve the assignments is **not** permitted (ChatGPT, Perplexity, etc.)

★ Participation in the programming assignments is NOT mandatory;

↪ 2–12

however you can earn up to **10% extra points** for the exam!

☠ Plagiarising will lead to losing **ALL extra points**.

Nonetheless, the participation in the assignments is **highly recommended!**

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
SS25	66 % (mit ⌀ 3,2)	86 % (mit ⌀ 2,2)
SS24	58 % (mit ⌀ 3,2)	91 % (mit ⌀ 2,4)
SS23	48 % (mit ⌀ 3,4)	83 % (mit ⌀ 2,3)
SS22	33 % (mit ⌀ 3,5)	69 % (mit ⌀ 2,7)
SS21	38 % (mit ⌀ 3,3)	87 % (mit ⌀ 2,4)

0 % 20 % 40 % 60 % 80 % 100 %

Bestehensquote

keine oder weniger als die Hälfte der Übungsaufgaben abgegeben

mindestens die Hälfte der Übungsaufgaben abgegeben

Passing Rate of the Exam (SPiC)

By activity of the participants in the programming assignments.

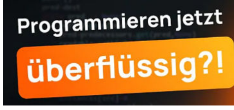
	submitted none or less than half of the assignments	**submitted** at least half of the assignments
ST25	66 % (with ∅ 3,2)	86 % (with ∅ 2,2)
ST24	58 % (with ∅ 3,2)	91 % (with ∅ 2,4)
ST23	48 % (with ∅ 3,4)	83 % (with ∅ 2,3)
ST22	33 % (with ∅ 3,5)	69 % (with ∅ 2,7)
ST21	38 % (with ∅ 3,3)	87 % (with ∅ 2,4)

0 % 20 % 40 % 60 % 80 % 100 %

Passing rate

Vorbemerkungen KI

Programmieren mit KI



Quelle: YouTube

unsicherer Code:
: KI-generierter Code kann versteckte Sicherheitslücken enthalten

Verlust Fähigkeiten:
: Entwickler verlassen sich zunehmend auf KI

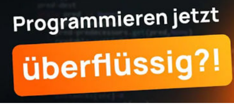
Wartungsaufwand:
: KI-generierter Code führt zu schwer wartbarer Codebasis

Over-Reliance:
: blindes Vertrauen auf KI statt Probleme zu verstehen

Fazit: KI ist ein mächtiges Werkzeug, ersetzt aber (noch) nicht die Notwendigkeit von Expertenwissen und sorgfältiger Code-Review.

Vorbemerkungen KI

Programmieren mit KI



Quelle: YouTube

Fazit: KI ist ein mächtiges Werkzeug, ersetzt aber (noch) nicht die Notwendigkeit von Expertenwissen und sorgfältiger Code-Review.

Wer Code von anderen (oder einer KI) reviewen will, muss selber programmieren können!

Übungsplattform: Das SPiCboard

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

optional:

- OLED Display

I²C Anschlüsse
OLED1 OLED2
GND VCC SCL SDA VCC GND SCL SDA

Serielle Schnittstelle
TX GND RX

LED-Reihe (active low)
LED 0 Red 0 an PD6
LED 1 Yellow 0 an PD5
LED 2 Green 0 an PD4
LED 3 Blue 0 an PD7
LED 4 Red 1 an PB0
LED 5 Yellow 1 an PB1
LED 6 Green 1 an PC3
LED 7 Blue 1 an PC2

Schieberegister
RCK an PB2, SCK an PB5 und SER an PB3

7 Segment Anzeige (active low)
Zehner Bit 7 an
Einer Bit 7 aus

Fototransistor an ADC 1

Button 1 prellt an PD3

Button 0 entprellt an PD2

Taster (active low)

Potentiometer an ADC 0

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

Exercise Platform: the SPiCboard

- ATmega328-µC
- USB port
- 8 LEDs
- 2 7-segment elements
- 2 buttons
- 1 potentiometer
- 1 photo sensor
- *optional:*
- OLED display

- can be borrowed during hands-on exercises
- better option: ↪ solder one yourself!
- alternatively: development in simulator, which is integrated into the IDE

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
  - Insgesamt 3 Termine (in KW 17)
- Anmeldung über Waffel **notwendig**, da begrenzte Plätze:
  ab Donnerstag 16.04.2026 um 18:00 Uhr (siehe Webseite)
- Kostenlos für Teilnehmer dieser Veranstaltung
  (finanziert aus Studienzuschüssen)

Der bei der Anmeldung gewählte Termin ist verbindlich!

SPiCboard-Soldering Night

- The FSI EEI, FSI ME and the FabLab offer a “soldering night” for the participants of the SLP lecture.
  - participation is not mandatory
  - you can gain (first) soldering experience while building your own SPiCboard
  - there will be 3 appointments
- Registration via Waffel **necessary** since places are limited: from April 16th, 6pm
- Participation is free of charge for SLP students (materials are funded from tuition fees)

The date you choose to register is binding!

Prüfung und Modulnote

- Prüfung (Klausur)
  - Termin: voraussichtlich Ende Juli / 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

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

Exam and Final Grade

- Exam (written test)
  - date: expected in end of July / early August
  - length: 90 min (SLP)
  - contents: questions on the lecture + programming exercise

- Exam grade $\mapsto$ final grade
  - (Usually) 50% of the exam's maximum possible points (EP) are necessary to pass.
  - Only if you passed, your grade can be improved by your bonus points from the programming exercises.
    - minimum: 20% of possible bonus points (BP)
    - bonus points get divided in equal parts to match the interval [50%;80%] of possible BP

  $\rightsquigarrow$ having 80%-100% of possible BP $\mapsto$ +10% of the maximum EP

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 i4slp@i4.cs.fau.de (inhaltlich)
  → wiss. Mitarbeiter i4slp-orga@i4.cs.fau.de (organisatorisch)

Chat für euren Austausch:

`https://to.chat.fau.de/#/room/#spic:fau.de`

If there are Questions or Problems

- Take a look at the lecture or tutorial slides
- Consult the FAQ on our website
- Hands-on exercise
- Only if you still have no answer or in special cases, write an email to
  → all tutorial advisors i4spic@lists.cs.fau.de (content-related)
  → all academic staff (of this lecture) i4spic-orga@lists.cs.fau.de (organizational questions)

  Chatroom for students:
  `https://to.chat.fau.de/#/room/#spic:fau.de`

Systemnahe Programmierung in C

Teil B Einführung in C

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Sommersemester 2026

http://sys.cs.fau.de/lehre/ss26

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 :-)

The First C Program

- The most famous program in the world in **C**

```
#include <stdio.h>

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

- Compilation and execution (on a UNIX system)

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

Not that complicated at all :-)

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;
  }
}
```

The First C Program – a Comparison to Java

- The most famous program of the world in **C**

```c
#include <stdio.h>

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

- The most famous program of the world in **Java**

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

A Comparison to Hello World in Python

- The most famous program of the world in **Python**

```
print('Hello World')
```

- **Python**
  - Usually execution in an interpreter
  - High-level abstraction level
  - Numerous built-in functions
  - No explicit include directive required for built-in function
- **C**
  - Native execution on machine
  - No interpreter required
  - Machine orientation
  - Explicit include statements for used functions

Das erste C-Programm – Erläuterungen

- **C**-Version zeilenweise erläutert
  1. To use the function `printf()`, the **library** `stdio.h` is included by the **preprocessor instruction** `#include`.
  3. A C program starts with `main()`, a **global function** of type `int`, which is defined in exactly one **file**.
  5. The string is output using the **function** `printf()`. (`\n` $\leadsto$ new line)
  6. Return to the operating system with **return value**. In this case, 0 indicates that no error occurred.

- **Java**-Version zeilenweise erläutert
  1. To use the **class** `out`, the **package** `System` is included by the `import` instruction.
  2. Each Java program consists of at least one **class**.
  3. Each Java program starts with the function `main()`, a **static method** of type `void`, which is defined in exactly one **class**.
  5. The output of one string takes place in the **method** `println()` from the class `out`, which is from the package `System`.
  6. Return to the operating system.

Das erste C-Programm für einen μ-Controller

Vorbemerkung:

Port A

0 1 2 3 4 5 6 7

DDRA PINA PORTA

Port D

DDRD
PIND
PORTD

Mikro-
Controller

DDRB
PINB
PORTB

Port B

DDRC PINC PORTC

0 1 2 3 4 5 6 7

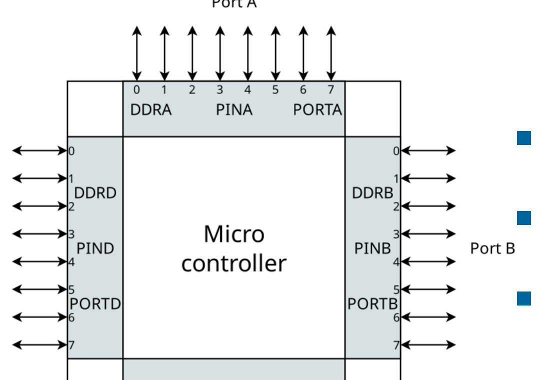
Port C

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

(jeweils 8 Bit)

The First C Program for a µ-Controller

Preliminary information:

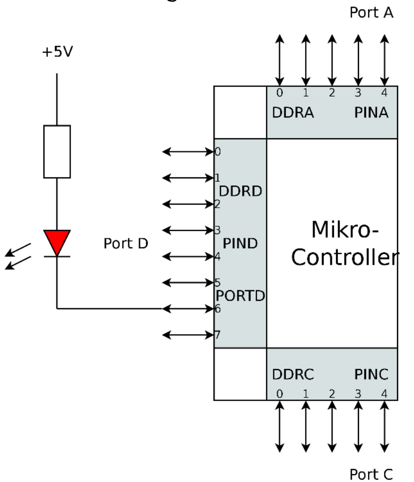


- DDRx: data direction register
- PINx: port input register
- PORTx: port output register

(of size 8 bits each)

Das erste C-Programm für einen μ-Controller

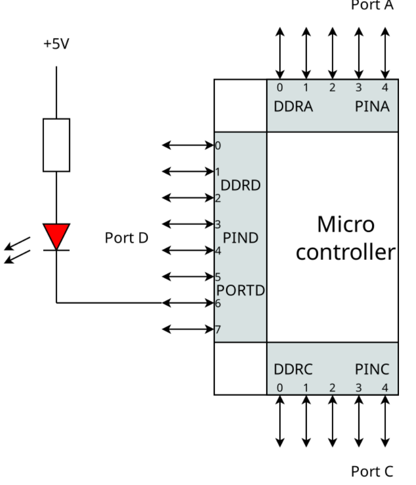
Vorbemerkung:



- 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)

The First C Program for a µ-Controller

Background information:



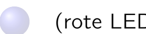
- LED is not lit:
  - DDRD bit 6: '1' (output)
  - PORTD bit 6: '1' (5V)
- LED lights up:
  - DDRD bit 6: '1' (output)
  - PORTD bit 6: '0' (0V)

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) ⇝ Übung
- Ausführen (SPiCboard):  (rote LED leuchtet)

The First C Program for a µ-Controller

- “Hello world” for AVR ATmega (SPiCboard)

```c
#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 programming is “somewhat different”.

- Compilation and **flashing** (with SPiC-IDE) ⤳ Exercises
- Execution (SPiCboard): (red LED lit)

Das erste C-Programm für einen µ-Controller

- „Hello World“ für AVR ATmega (vgl. ↪ 3–2)

```
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 }
```

The First C Program for a µ-Controller

- "Hello world" for AVR ATmega (compare ↪ 3–2)

```
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

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

  1 To access the hardware registers (`DDRD`, `PORTD`, provided as **global variables**), the **library** `avr/io.h` is included with `#include`.

  3 The `main()`-function has **no return value** (type `void`). A µ-Controller program runs **indefinitely** ⇝ `main()` does not terminate.

  5-6 First, the **hardware** is initialized (d. h. put in a predefined state). For this, **single bits** in certain **hardware registers** have to be changed.

  9 The interaction with the environment (in this case: switching on the LED) takes place by **manipulating single bits** in hardware registers.

  12-13 There will be **no return** to an operating system (which one?). The endless loop assures that `main()` does not terminate.

µ-Controller-Program – Explanations

- µ-Controller-program is explained line by line
  (note the differences to the Linux-version ↪ 3–2)
  1 To access the hardware registers (`DDRD`, `PORTD`, provided as **global variables**), the **library** `avr/io.h` is included with `#include`.
  3 The `main()`-function has **no return value** (type `void`). A µ-Controller program runs **indefinitely** ⇝ `main()` does not terminate.
  5-6 First, the **hardware** is initialized (d. h. put in a predefined state). For this, **single bits** in certain **hardware registers** have to be changed.
  9 The interaction with the environment (in this case: switching on the LED) takes place by **manipulating single bits** in hardware registers.
  12-13 There will be **no return** to an operating system (which one?). The endless loop assures that `main()` does not terminate.

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.

The Second C Program – Input with Linux

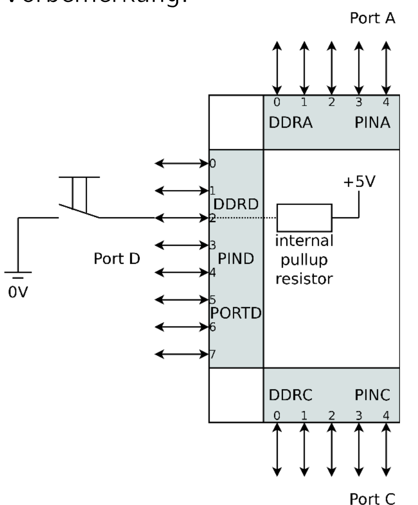
- user interaction (reading one character) with Linux:

```
#include <stdio.h>

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

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

The **`getchar()`**-function reads one character from the standard input (here: keyboard). The function "waits", if necessary, until a character is available.

Das zweite C-Programm für einen µ-Controller

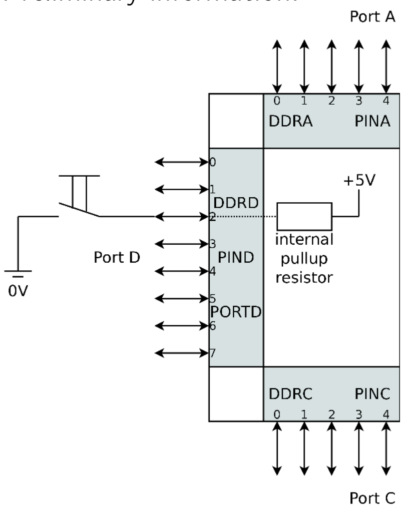
Vorbemerkung:



- 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

The Second C Program for a µ-Controller

Preliminary information:



- Initializing:
  - DDRD bit 2: '0' (input)
  - PORTD bit 2: '1' (pull-up switched on)
- Detection:
  - PIND bit 2: '1' => button not pressed
  - PIND bit 2: '0' => button pressed

Das zweite C-Programm – Eingabe mit µ-Controller

- Benutzerinteraktion (Warten auf Tasterdruck) auf dem SPiCboard:

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

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

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

  /* wait until PD2 -> low (button is pressed) */
  while ((PIND >> 2) & 1) {
  }

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

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

The Second C Program – Input with µ-Controller

- User interaction (waiting for a button to be pressed) on the 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 Just like the LED, the button is connected to a **digital IO pin** of the µ-Controller. We now configure pin 2 of port D as an **input** by **clearing** the corresponding bits in the register `DDRD`.

  6 By **setting** bit 2 in the register `PORTD` to 1, the internal pull-up resistor (high resistance) is activated. Which is connected to $V_{CC} \leadsto$ PD2 = *high*.

  13-14 **Active waiting:** waits for a button to be pressed, d. h. while PD2 (bit 2 in the register `PIND`) is *high*. When the button is pressed, PD2 is pulled to ground $\leadsto$ bit 2 in the register `PIND` is now *low* and the loop is exited.

Zum Vergleich: Benutzerinteraktion als Java-Programm

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

public class Input implements ActionListener {
  private JFrame frame;

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

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

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

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

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

As a Reference: User Interaction as a Java Program

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

public class Input implements ActionListener {
  private JFrame frame;

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

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

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

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

- 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 The class `Input` implements an **interface** to receive interaction events.
  - 10-12 The program behavior is implemented with the help of a multitude of **objects** (`frame`, `button`, `input`), which are created during initialization.
  - 20 The created `button`-object sends a message to the `input`-object.
  - 23-26 The button press is signaled by an `actionPerformed()`-message (method call).

- The program cannot be compared to its counterpart in C directly.
  - It uses the (already known to you) **object-oriented paradigm**, which is typical for Java.
  - This difference shall be emphasised here.

- User interaction in Java explained line by line

  5 The class `Input` implements an **interface** to receive interaction events.

  10-12 The program behavior is implemented with the help of a multitude of **objects** (`frame`, `button`, `input`), which are created during initialization.

  20 The created `button`-object sends a message to the `input`-object.

  23-26 The button press is signaled by an `actionPerformed()`-message (method call).

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), ...

1st Takeaway: Java/Python → C (Syntax)

- Java/Python and C have similar **syntax**
  (Syntax: "What do valid programs of the language look like?")
- C syntax was used as a reference for the development of Java/Python
  ⇝ many language elements are similar or identical
  - blocks, loops, conditions, statements, literals
  - these elements will be looked at in detail in the following chapters
- Major elements from Java/Python are **not** present in C
  - classes, packages, objects, 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

1st Takeaway: Java/Python → C (Idiomatic)

- There are major **idiomatic** differences
  (Idiomatic: "What do programs of the language usually look like?")
- **Java: object-oriented paradigm**
  - Central question: From which things is a problem made of?
  - Segmentation of the problem in **classes** and objects
  - Hierarchy by inheritance and aggregation
  - Program flow by interaction between objects
  - Re-usability through extensive class libraries
- **C: imperative paradigm**
  - Central question: From which steps is the problem made up?
  - Segmentation of the problem in **functions** and variables
  - Hierarchy by breakdown into functions
  - Program flow through calls between functions
  - Re-usability through function libraries

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

1st Takeaway: Java/Python → C (Philosophy)

- There are **philosophical** differences as well
  (Philosophy: "Basic ideas and concepts of a language")
- **Java:** Security and portability due to **abstracting from machine**
  - Compilation for virtual machines (JVM)
  - Extensive checks for programming errors during runtime
    - range overflow, division by 0, ...
  - Problem-centric memory model
    - Only type-safe memory accesses, automatic garbage collection during runtime.
- **C:** efficiency and lightweight due to **machine orientation**
  - Compilation for concrete hardware architecture
  - No checks for programming errors during runtime
    - some errors are caught by the operating system – if present
  - Memory model directly maps to the machine
    - pointers provide direct memory access
    - coarse-grained access protection and automatic garbage collection (at processor level) by an OS – if present

Ein erstes Fazit: µ-Controller-Programmierung

C ↦ Maschinennähe ↦ µC-Programmierung

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

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

Ansatz: Mehr Abstraktion durch problemorientierte Bibliotheken

A First Takeaway: μ-Controller Programming

C ↦ machine orientation ↦ μC programming

The machine orientation of the language C especially shows when looking at μ-Controller programming!

- Only one program is running
  - On RESET the program is loaded directly from flash memory
  - Hardware has to be initialized by the program first
  - Shall never terminate (z. B. with the help of an infinite loop in `main()`)
- The solution is implemented close to the machine
  - Direct manipulation of single bits in hardware registers
  - Therefore detailed knowledge of *electrical wiring* is needed
  - No support of an operating system (like Linux)
  - Usually a low level of abstraction ⇝ error-prone... *but fast*

Approach: Higher abstraction with problem-oriented libraries

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

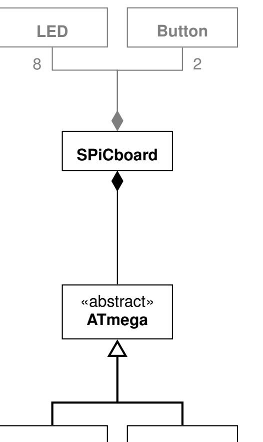
9 Funktionen

10 Variablen

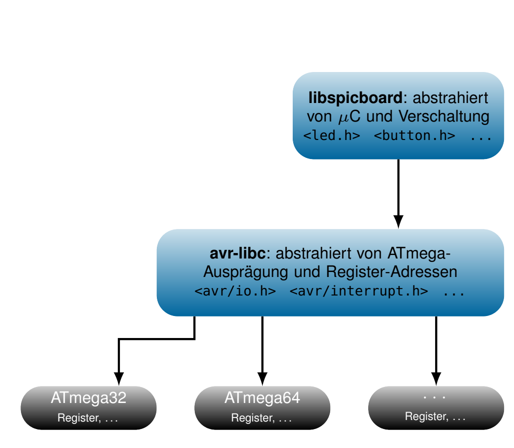
11 Präprozessor

Abstraktion durch Softwareschichten: SPiCboard

Hardwaresicht

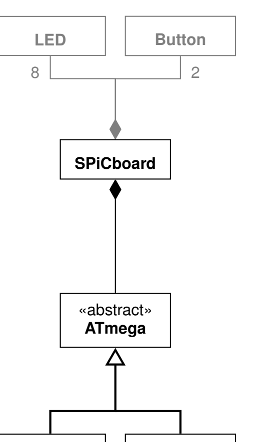


Softwareschichten

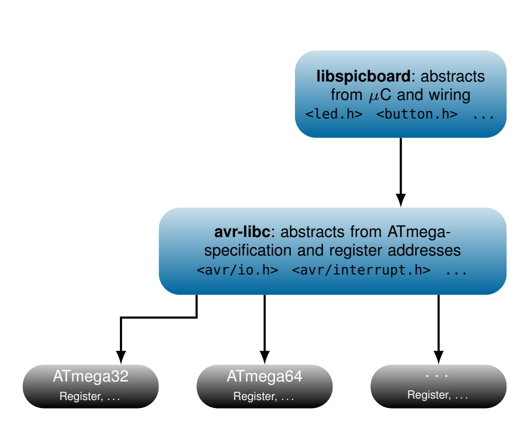


Abstraction by Software Layers: SPiCboard

Hardware view



Software layers



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

Problemnähe ↑ — Maschinennähe ↓

Programm läuft nur auf dem **SPiCboard**. Es verwendet Funktionen (wie `sb_led_on()`) und Konstanten (wie `RED0`) der **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:

Abstraction by Software Layers: Comparing *LED → on*

Program only runs on the **SPiCboard**. It uses functions (like `sb_led_on()`) and constants (like `RED0`) of the **libspicboard** that represent concrete wiring of LEDs, buttons, etc. with the μC:

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

Program runs on **each** μC of the ATmega-series. It uses **symbolic register names** of the **avr-libc** (like `PORTD`) and general characteristics:

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

Program only runs on **ATmega32**. It uses register addresses specific to the **ATmega32** (like `0x12`) and characteristics:

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

Goal: Switch on LED `RED0` on the SPiCboard:

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)`

Abstraction by Software Layers: Complete Example

Until now: development with 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) {
  }
}
```

(ref. ↪ 3–11)

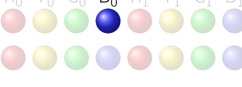
Now: development with libspicboard

```
#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) {
  }
}
```

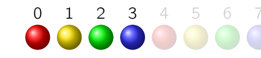
- Hardware initialization not no longer needed
- Program simpler to understand due to **problem-specific abstraction**
  - setting bit 6 in `PORTD` ↦ `sb_led_on(RED0)`
  - reading bit 2 in `PORTD` ↦ `sb_button_getState(BUTTON0)`

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) $\mapsto$ BUTTONSTATE_{PRESSED,RELEASED}
  - ADC-Modul (#include <adc.h>)
    - Potentiometer-Stellwert abfragen: sb_adc_read(POTI) $\mapsto$ $\{0 \ldots 1023\}$

Abstraction of the libspicboard: Short Overview

- Output abstractions (selection)
  - LED module (`#include <led.h>`)
    - switch LED on: `sb_led_on(BLUE0)` ⇝
    - switch LED off: `sb_led_off(BLUE0)` ⇝
    - switching all LEDs on or off: `sb_led_setMask(0x0f)` ⇝

  - 7-segment module (`#include <7seg.h>`)
    - showing an integer $n \in \{-9 \ldots 99\}$: `sb_7seg_showNumber(47)` ⇝

- Input abstractions (selection)
  - Button module (`#include <button.h>`)
    - reading the button state: `sb_button_getState(BUTTON0)` ↦ `BUTTONSTATE_{PRESSED,RELEASED}`
  - ADC module (`#include <adc.h>`)
    - reading the value of the potentiometer: `sb_adc_read(POTI)` ↦ $\{0 \ldots 1023\}$

Softwareschichten im Allgemeinen

Problemnähe ↔ Maschinennähe

Diskrepanz: Anwendungsproblem ⟷ Abläufe auf der Hardware

CAD-System
Java-Programm
Klassenbibliotheken
Übersetzung
Java Byte-Code
Interpretation
JVM
C-Programm
Funktionsbibliotheken
Übersetzung
Assemblerprogr.
Übersetzung
Maschinenprogr.
Ausführung
Interpretation von Systemaufrufen
Prozessor
Betriebssystem
C-Programm
Grafik-Workstation

Ziel: Ausführbarer Maschinencode

Software Layers in General

Discrepancy: application problem ⟷ hardware processes

Goal: executable machine code

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**.

The Role of the Operating System

- **User view:** Environment for starting, controlling, and combining applications
  - Shell, graphical user interface
    - z. B. bash, Windows
  - Communication between applications and users
    - z. B. with files
- **Application view:** Function libraries with abstractions for easier software development
  - Generic in-/output of data
    - z. B. on printers, serial interfaces, in files
  - Persistent storage and transfer of data
    - z. B. by the file system, over TCP/IP sockets
  - Management of memory and other resources
    - z. B. execution time on CPU

The Role of the Operating System (Forts.)

- **System view:** Software layers for multiplexing hardware resources (↪ multi-user mode)
  - Parallel handling of program instances with **process concepts**
    - virtual memory ↪ own 32-/64-bit address space
    - virtual processor ↪ scheduled/preempted transparently
    - virtual in/output devices ↪ can be piped in files, sockets, . . .
  - Isolation of program instances with **process concepts**
    - automatic garbage collection at the end of process life
    - detection/prevention of memory access to other processes
  - **Partial protection** from critical programming errors
    - detection of *some* invalid memory accesses (z. B. access to address `0`)
    - detection of *some* invalid operations (z. B. `div/0`)

µC programming without operating system platform ⇝ **no protection**

- Operating system **protects the programmer less** from bugs compared to z. B. Python.
- For the µC programming, we even have to **give up this protection**.
- 8/16-bit µC often have **no hardware support** for protection.

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:

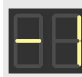
```
gcc error-linux.c -o error-linux
./error-linux
Floating point exception
```

⇝ 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**.

Example: Error Detection by the Operating System

Linux: Division by 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("Result: %d\n", b);
10
11   return 0;
12 }
```

Compilation and execution yield:

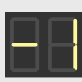
```
gcc error-linux.c -o error-linux
./error-linux
Floating point exception
```

⤳ program gets **terminated**.

SPiCboard: Division by 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) {}
}
```

Execution yields:



⤳ Program continues execution with **incorrect data**.

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

Structure of a C Program – General

```
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 }
```

- A C program (usually) consists of
  - a set of global variables
  - a set of (sub-)functions
    - a set of local variables
    - a set of instructions
  - the function `main()`, which is the entry point for any execution

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

Structure of a C Program – an Example

```
 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  }
```

- A C program (usually) consists of
  - a set of global variables — `nextLED`, line 5
  - a set of (sub-)functions — `wait()`, line 15
    - a set of local variables — `i`, line 16
    - a set of instructions — `for`-loop, line 17
  - the function `main()`, which is the entry point for any execution

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

Identifier [=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 }
```

- Names given by the developer for certain elements of the program
  - element: type, variable, constant, function, label (i.e., jump target)
  - structure: **[** *A-Z, a-z, _* **] [** *A-Z, a-z, 0-9, _* **]** ∗
    - one letter, followed by a combination of letters, numbers and underscores
    - underscore can be used as a first symbol, however, this is usually reserved for compiler manufacturers
  - every identifier has to be declared prior to being used

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`

Keywords [≈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 }
```

- Reserved words/keywords of the language
(⇝ must not be used as an identifier)
  - embedded (*primitive*) types — `unsigned int`, `void`
  - type modifiers — `volatile`
  - control structures — `for`, `while`
  - elementary instructions — `return`

05-Sprachueberblick: 2026-04-13

Schlüsselwörter in C99

- 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`

Keywords in C99

[Handout]

- Reference: list of keywords (up until 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

Literals [=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 }
```

- (Expression of) constants in the code
  - For every primitive data type, there is at least one literal form.
    - for integers: decimal (base 10: `65535`), hexadecimal (base 16, leading `0x`: `0xffff`), octal (base 8, leading `0`: `0177777`)
  - The programmer can then choose the most suitable form.
    - `0xffff` is more convenient than `65535` to represent the maximal value of a 16-bit integer

Anweisungen [=Java]

```c
// include files
#include <led.h>

// global variables
LED nextLED = RED0;

// subfunction 1
LED lightLED(void) {
  if (nextLED <= BLUE1) {
    sb_led_on(nextLED++);
  }
  return nextLED;
}

// subfunction 2
void wait(void) {
  volatile unsigned int i;
  for (i = 0; i < 0xffff; i++)
    ;
}

// main function
void main() {
  while (lightLED() < 8) {
    wait();
  }
}
```

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

Instructions [=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  }
```

- Describe the actual control flow of the program
- They are hierarchically made up of three basic forms
  - single instruction – expression followed by ; (contrast: Python's line break)
    - single semicolon ↦ empty instruction
  - block – sequence of instructions, wrapped in {...} (contrast: Python's indentation along with colon :)
  - control structures, followed by instructions

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

Expressions [=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  }
```

- Valid combination of operators, literals, and identifiers
  - "valid" in the sense of syntax and types
  - priority rules for operators determine the order, in which the expressions get handled ↪ 7–14
    - order of execution can be explicitly forced with the help of brackets ( )
    - the compiler is allowed to evaluate partial expressions in the most efficient order

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!)

What Exactly is a Data Type?

- **Data type** := (<*set of values*>, <*set of operations*>)
  - Literal — value in the source code ↪ 5–6
  - Constant — identifier for a value
  - Variable — identifier for a memory address, where a value can be stored
  - Function — identifier for a sequence of instructions, which will return a value

  ⇝ literals, constants, variables, functions all have a **(data) type**

- The data type determines
  - the representation of the value in memory
  - the **size** which gets occupied by the variable in storage
  - which **operations** are permitted

- The data type gets determined
  - explicitly, by declaration, type cast, or notation (literals)
  - implicitly, by "omitting" (⇝ int bad style!)

Primitive Datentypen in C

- Ganzzahlen/Zeichen `char`, `short`, `int`, `long`, `long long` (C99)
  - Wertebereich: implementierungsabhängig [≠Java]

    Es gilt: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`
  - Jeweils als `signed`- und `unsigned`-Variante verfügbar
- Fließkommazahlen `float`, `double`, `long double`
  - Wertebereich: implementierungsabhängig [≠Java]

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

Primitive Data Types in C

- Integers/characters `char`, `short`, `int`, `long`, `long long` (C99)
  - range of values: dependent on implementation [≠Java]
    still: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`
  - both available in `signed` or `unsigned` version
- Floating-point numbers `float`, `double`, `long double`
  - range of values: dependent on implementation [≠Java]
    still: `float` ≤ `double` ≤ `long double`
  - From C99 onwards, they are available as `_Complex` data types (for complex numbers).
- Empty data type `void`
  - range of values: $\emptyset$
- Boolean `_Bool` (C99)
  - range of values: $\{0, 1\}$ (↩ actually only an integer type)
  - conditional expressions (z. B. `if(...)`) are of type `int`! [≠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
```

Integer Types [≈Java]

- Integer type — usage — literal from
  - char — small integer or character — 'A', 65, 0x41, 0101
  - short [int] — integer (int is optional) — s. o.
  - int — integer of "natural size" — s. o.
  - long [int] — big integer — 65L, 0x41L, 0101L
  - long long [int] — really big integer — 65LL, 0x41LL, 0101LL
- Type modifier — is prefixed — literal suffix
  - signed — type is signed (standard case) — -
  - unsigned — type does not have a sign — U
  - const — variable cannot be changed — -
- Examples (definition of variables)

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

Integertypen: Größe und Wertebereich [≠Java]

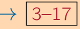
- Die interne Darstellung (Bitbreite) ist implementierungsabhängig

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

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



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

Integer Types: Size and Range of Values [≠Java]

- The internal representation (width in bits) is dependent on implementation

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

- The range of values can be calculated from the width in bits
  - signed $-(2^{bits-1}-1) \longrightarrow +(2^{bits-1}-1)$
  - unsigned $0 \longrightarrow +(2^{bits}-1)$

The philosophy of C is obvious: Efficiency by **machine orientation**

Internal representation of integer types is defined by the **hardware** (width of registers, bus, etc.). This yields code that is i. a. **more efficient**.

Integertypen: Maschinennähe ⟶ Problemnähe

- **Problem:** Breite ($\leadsto$ 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 ($\leadsto$ Portierbarkeit)
- **Lösung:** Modul `stdint.h`
  - Definiert Alias-Typen: `int`*n*`_t` und `uint`*n*`_t` für $n \in \{8, 16, 32, 64\}$
  - Wird vom Compiler-Hersteller bereitgestellt

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

Integer Types: Machine Orientation ⟶ Problem Orientation

- **Problem:** width ($\leadsto$ range of values) of C standard types is dependent on implementation
  $\mapsto$ **machine orientation**
- **Often needed:** Integer types of specific size
  $\mapsto$ **problem orientation**
  - represent range of values safely, but as memory efficient as possible
  - dealing with registers of defined width n
  - keeping code independent of compiler and hardware ($\leadsto$ portability)
- **Solution:** module `stdint.h`
  - defines alias types: `int`n`_t` and `uint`n`_t` for $n \in \{8, 16, 32, 64\}$
  - is provided by compiler developers

range of values for `stdint.h`-types							
`uint8_t`	0	→	255	`int8_t`	−128	→	+127
`uint16_t`	0	→	65 535	`int16_t`	−32 768	→	+32 767
`uint32_t`	0	→	4 294 967 295	`int32_t`	−2 147 483 648	→	+2 147 483 647
`uint64_t`	0	→	$> 1.8 * 10^{19}$	`int64_t`	$< -9.2 * 10^{18}$	→	$> +9.2 * 10^{18}$

Typ-Aliase mit `typedef` [≠Java]

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

  typedef *Typausdruck Bezeichner* ;

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

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

```
// stdint.h (x86-gcc, IA32)
typedef unsigned char  uint8_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
...
```

Type Aliases with `typedef` [≠Java]

- With help of the keyword **`typedef`**, possibility to define a type alias:

  `typedef` *alias* *identifier*;

  - *identifier* is now an alternative name for a *type expression*
  - It can be used at any place a type expression is expected.

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

```
// stdint.h (x86-gcc, IA32)
typedef unsigned char  uint8_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!

Type Aliases with `typedef` (Forts.) [≠Java]

- Type aliases enable easy problem-specific abstractions
  - `register` is closer to the problem than `uint8_t`
    ⇝ later (z. B. with 16-bit-registers) modification possible
  - `uint16_t` is closer to the problem than `unsigned char`
  - `uint16_t` is safer than `unsigned char`

Defined bit widths are crucial for µC development!

- Major differences between platforms and compilers
  ⇝ compatibility problems
- To save memory, the **smallest possible** integer type should always be used!

Rule: For system-level programming, types from `stdint.h` are used!

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
```

Enumeration Types with enum

- With help of the keyword **enum**, an enumeration type is defined, consisting of an explicit set of symbolic values:

  enum *identifier*$_{opt}$ { *listofconstants* } ;

- Example
  - definition:

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

  - usage:

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

- Simplification with **typedef**
  - definition:

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

  - usage:

```
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

enum ↦ int [≠Java]

- enum types are technically nothing else than integers (int)
  - enum constants get enumerated, starting from 0

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

  - possibility to explicitly assign values:

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

  - they can be used like ints (z. B. arithmetic operations)

```
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!
```

- ⇝ There will be **no type checks**!

This conforms to **C philosophy**!

Fließkommatypen [≈Java]

- Fließkommatyp | Verwendung | Literalformen

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**!

Floating-Point (FP) Types [≈Java]

- FP type — usage — literal form

FP type	usage	literal form
`float`	single precision	`100.0F`, `1.0E2F`
`double`	double precision	`100.0`, `1.0E2`
`long double`	“extended precision“	`100.0L` `1.0E2L`

- Precision / range of values are implementation-dependent [≠Java]
  - still: `float` $\leq$ `double` $\leq$ `long double`
  - `long double` and `double` are identical on most platforms

“efficiency by machine orientation“

Floats + µC platform = $$$

- Often, µCs have no native hardware support for `float` arithmetic.
  $\leadsto$ **really expensive** emulation in software (slow, much additional code)
- Memory demand of `float`- and `double` variables is **quite high**
  $\leadsto$ at least 32/64 bits (`float`/`double`)

Rule: When programming a µ-Controller, floating-point arithmetic **should be avoided**!

Zeichen $\mapsto$ 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 $\mapsto$ 128 Zeichen standardisiert
    (die verbleibenden 128 Zeichen werden unterschiedlich interpretiert)
  - Spezielle Literalform durch Hochkommata
    `'A'` $\mapsto$ ASCII-Code von `A`
  - Nichtdruckbare Zeichen durch Escape-Sequenzen
    - Tabulator `'\t'`
    - Zeilentrenner `'\n'`
    - Backslash `'\\'`
- Zeichen $\mapsto$ Integer $\leadsto$ man kann mit Zeichen rechnen

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

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

Character ↦ Integer

- In C, characters are integers ↪ 6–3
  - `char` is part of the integer types (usually 8 bits = 1 byte)
- Representation takes place with ASCII code ↪ 6–12

  - 7-bit code ↦ 128 standardized characters
    (the remaining 128 characters can be interpreted differently)
  - special literal form with single quote marks
    `'A'` ↦ ASCII code of `A`
  - non-printable characters with escape sequences
    - tab character `'\t'`
    - line separator `'\n'`
    - backslash `'\\'`
- character ↦ integer ⇝ characters can be used in operations

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

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

ASCII-Code-Tabelle (7 Bit)

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

NUL 00	SOH 01	STX 02	ETX 03	EOT 04	ENQ 05	ACK 06	BEL 07
BS 08	HT 09	NL 0A	VT 0B	NP 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

ASCII-Code Table (7 bit)

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

NUL 00	SOH 01	STX 02	ETX 03	EOT 04	ENQ 05	ACK 06	BEL 07
BS 08	HT 09	NL 0A	VT 0B	NP 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.

Strings [≠Java]

- In C, a string is an array of characters.
  - representation: sequence of single characters, terminated by (last character): `NUL` (ASCII value 0)
  - memory demand: (length + 1) bytes
- Special literal form with double quotes:

"Hi!" ↦ | 'H' | 'i' | '!' | 0 | ⟵ terminating 0 byte

- Example (Linux)

```c
#include <stdio.h>

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

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

Strings need relatively much memory and “larger” output devices (z. B. LCD display).
⇝ For µC programming they only have a minor significance.

06-Datentypen: 2026-04-13

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.

Outlook: Complex Data Types

- From primitive data types, complex data types can be created (recursively)
  - Arrays ↪ element sequence (same type) [≈Java/Python]

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

  - Pointers ↪ modifiable reference to a variable [≠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)
```

  - Structures ↪ composition of elements of any type [≠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
```

- We take a closer look at this in later chapters.

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	
*	Multiplikation	
/	Division	
unäres −	negatives Vorzeichen (z. B. $-a$)	⇝ Multiplikation mit −1
unäres +	positives Vorzeichen (z. B. +3)	⇝ kein Effekt

- Zusätzlich nur für Ganzzahltypen:

%	Modulo (Rest bei Division)

Arithmetic Operators

[=Java]

- Can be used with all integer and floating-point types

+	addition	
−	subtraction	
*	multiplication	
/	division	
unary −	negative sign (z. B. $-a$)	⇝ multiplication with −1
unary +	positive sign (z. B. +3)	⇝ no effect

- Additionally only for integer types:

%	modulo (remainder of 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
```

Increment/Decrement Operators

- Available for integer types and pointers [≠Python]

++	increment (increase by 1)
−−	decrement (decrease by 1)

- Left-side operator (prefix) `++x` or `--x`
  - first, the value of variable `x` gets changed
  - then, the (new) value of `x` is used
- Right-side operator (postfix) `x++` or `x--`
  - first, the (old) value of `x` is used
  - then, the value of `x` gets changed
- Examples

```
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
```

Comparison Operators [≠Python]

- Comparison of two expressions

<	less
<=	less or equal
>	greater
>=	greater or equal
==	identical (two equal signs!)
!=	unequal

- Note: The result is of type `int` [≠Python]
  - Result: *false* $\mapsto$ 0
    true $\mapsto$ 1
  - The result can be used for calculations
- Examples

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

- 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* $\mapsto 1$

Logical Operators [≠Python]

- Combining logical values (true / false), commutative

&&	`and` in Python (conjunction)	*true* && *true* → *true* *true* && *false* → *false* *false* && *false* → *false*
||	`or` in Python (disjunction)	*true* || *true* → *true* *true* || *false* → *true* *false* || *false* → *false*
!	`not` in Python (negation, unary)	! *true* → *false* ! *false* → *true*

- Note: operands and result are of type `int` [≠Python]
  - Operand (input parameter): $0 \mapsto$ *false*, $\neq 0 \mapsto$ *true*
  - Result: *false* $\mapsto 0$, *true* $\mapsto 1$

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;`

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

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

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

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

- Kann überraschend sein, wenn Teilausdrücke Nebeneffekte haben

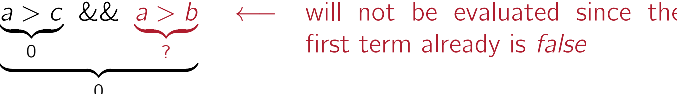
```
int a = 5; int b = 3; int c = 7;
if ( a > c && !func(b) ) {···}      // func() will not be called
```

Logical Operators – Evaluation

- The evaluation of a logical expression is terminated as soon as the result is known
  - Let `int a = 5; int b = 3; int c = 7;`

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

⟵ will not be evaluated since the first term already is *true*



- This *short-circuit evaluation* can have surprising results if subexpressions have side effects!

```
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 $op \in \{ +, -, \star, /, \%, \texttt{<<}, \texttt{>>}, \&, \hat{}, | \}$
- Beispiele

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

Assignment Operators [=Python]

- General assignment operator (=)
  - assigns a value to a variable
  - example: `a = b + 23`
- Arithmetic assignment operators (+=, −=, ...)
  - shortened notation for modifying the value of a variable
  - example: `a += 23` is equivalent to `a = a + 23`
  - generally: `a` *op*`=` `b` is equivalent to `a = a` *op* `b`

    for $op \in \{ +, -, \star, /, \%, <<, >>, \&, \hat{}, | \}$
- Examples

```c
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!

Assignments are Expressions!

- Assignments can be nested in more complex expressions
  - The result of an assignment is the assigned value.

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

  - The use of assignments in arbitrary expressions leads to **side effects**, which are not always obvious.

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

Particularly dangerous: use of = instead of ==

In C, logical values are integers: 0 $\mapsto$ *false*, $\emptyset \mapsto$ *true*

- In Python: syntax error
- typical “rookie mistake” of control structures:

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

- Compiler possibly gives **no warning** about the construct as it is a valid expression! $\leadsto$ Programming bug is quite easy to miss!

07-Operatoren: 2026-04-13

Bitoperationen [=Java]

- Bitweise Verknüpfung von Ganzzahltypen, kommutativ

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

Bit Operations [=Java]

- Bitwise operations of integers, commutative

Operator	Description	Example
&	bitwise "and" (bit intersection)	1 & 1 → 1 1 & 0 → 0 0 & 0 → 0
|	bitwise "or" (bit unification)	1 | 1 → 1 1 | 0 → 1 0 | 0 → 0
^	bitwise "exclusive or" (bit antivalence)	1 ^ 1 → 0 1 ^ 0 → 1 0 ^ 0 → 0
~	bitwise inversion (one's complement, unary)	~ 1 → 0 ~ 0 → 1

- Schiebeoperationen auf Ganzzahltypen, nicht kommutativ

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

- Beispiele (x sei vom Typ `uint8_t`)

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

Bit Operations (Forts.) [=Java]

- Shift operators on integers, not commutative

<<	bitwise left shift (on the right side, 0 bits are “inserted”)
>>	bitwise right shift (on the left side, 0 bits are “inserted”)

- Examples (let `x` be of type `uint8_t`)

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

Bitoperationen – Anwendung

- Durch Verknüpfung lassen sich gezielt einzelne Bits setzen/löschen

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

07-Operatoren: 2026-04-13

Bit Operations – Usage

- By combining these operations, single bits are set/unset.

bit#	7	6	5	4	3	2	1	0	
PORTD	?	?	?	?	?	?	?	?	Bit 7 shall be changed without altering other bits!
0x80	1	0	0	0	0	0	0	0	One bit gets set by **or-operation** with a mask that only contains a 1 bit at the desired position
PORTD |= 0x80	1	?	?	?	?	?	?	?	
~0x80	0	1	1	1	1	1	1	1	One bit gets unset (set to 0) by **and-operation** with a mask that only contains a 0 bit at the desired position.
PORTD &= ~0x80	0	?	?	?	?	?	?	?	
0x08	0	0	0	0	1	0	0	0	Inversion of one bit by **xor-operation** with a mask that only contains a 1 bit at the desired position.
PORTD ^= 0x08	?	?	?	?	¿	?	?	?	

07-Operatoren: 2026-04-13

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*) $\leadsto$ Verständlichkeit

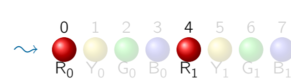
- Für „Dezimal-Denker“ bietet sich die Linksschiebe-Operation an

```
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) ;
}
```

$\leadsto$



- Bit masks are usually given as hexadecimal literals.

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

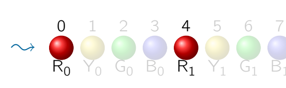
A hex digit represents a half byte: *nibble*

- For "thinkers in decimals", the left-shift notation is more suitable

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

- Combined with the or-operation, shifting ones works for complex masks

```
#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*) $\leadsto$ Ergebnis ist $Ausdruck_2$
    - $Ausdruck_1 = 0$ (*falsch*) $\leadsto$ 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;
}
```

Conditional Evaluation [≈Python]

- Formulation of conditions in expressions

  expression$_1$ **?** *expression*$_2$ **:** *expression*$_3$

  - first, *expression*$_1$ gets evaluated
    - *expression*$_1 \neq 0$ (*true*) $\leadsto$ *expression*$_2$ is the result
    - *expression*$_1 = 0$ (*false*) $\leadsto$ *expression*$_3$ is the result
  - `?:` is the only ternary (three-part) operator in C

- Example C

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

Python

- `value_if_true if condition else value_if_false`
- more readable, but longer

Sequenzoperator [≠Java]

- Reihung von Ausdrücken
  $\mathit{Ausdruck}_1$ **,** $\mathit{Ausdruck}_2$
  - Zunächst wird $\mathit{Ausdruck}_1$ ausgewertet
    $\leadsto$ Nebeneffekte von $\mathit{Ausdruck}_1$ werden sichtbar
  - Ergebnis ist der Wert von $\mathit{Ausdruck}_2$

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

Sequence Operator [≠Python]

- Sequencing of expressions

  expression$_1$ **,** *expression*$_2$

  - first, *expression*$_1$ gets evaluated
    ⇝ side effects of *expression*$_1$ are visible for *expression*$_2$
  - the value of *expression*$_2$ is the result

- Use of the comma operator is often not required!
  (C-preprocessor macros with side effects)

Vorrangregeln bei Operatoren

	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

Associativity Rules of Operators [≈Java]

	class	operators	associativity

1	function call, array access
	structure access
	post-increment/-decrement
		`x() x[]`
		`x.y x->y`
		`x++ x--`
			left → right

2	pre-increment/-decrement
	unary operators
	address, pointer
	type conversion (cast)
	type size
		`++x --x`
		`+x -x ~x !x`
		`& *`
		`(<Typ>)x`
		`sizeof(x)`
			right → left

3	multiplication, division, modulo	`* / %`	left → right
4	addition, subtraction	`+ -`	left → right
5	bitwise shifts	`>> <<`	left → right
6	relational operators	`< <= > >=`	left → right
7	equality operators	`== !=`	left → right
8	bitwise AND	`&`	left → right
9	bitwise OR	`|`	left → right
10	bitwise XOR	`^`	left → right
11	conjunction	`&&`	left → right
12	disjunction	`||`	left → right
13	conditional evaluation	`?:=`	right → left
14	assignment	`=` *op*`=`	right → left
15	sequence	`,`	left → right

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!
```

Type Promotion in Expressions

- Operations are calculated *at least* with `int`-width
  - `short`- and `signed char`-operands are “promoted” implicitly (↪ *Integer Promotion*)
  - Only the result will then be promoted/cut off to match the target type

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

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

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
```

- res: int8_t: 75
- a: int: 100
- b: int: 3
- a * b: int: 300
- int32_t: 300
- a * b / c: int32_t: 75

Type Promotion in Expressions (Forts.)

- In general, the *largest* involved width is used

```
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
```

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


07-Operatoren: 2026-04-13

Type Casting in Expressions (Forts.)

- Floating-point types are considered to be “larger” than integer types
- All floating-point operations are *at least* calculated with `double` width

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

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

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
```


⇝ Überraschende Ergebnisse bei negativen Werten!

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

Type Promotion in Expressions (Forts.)

- unsigned types are also considered "larger" than signed types

```
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

⇝ Surprising results when using negative values!

⇝ Avoid mixing signed and unsigned operands!

07-Operatoren: 2026-04-13

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
```

Type Casting in Expressions – Type Casts

- By using the type cast operator, an expression is converted into a target type.
- Casting is explicit type promotion.

(*type***)** *expression*

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

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

`res` → int: 1; `(int) u` → int: 1; `s < (int) u` → int: 1

Überblick: Teil B Einführung in C

3 Java versus C

4 Softwareschichten und Abstraktion

5 Sprachüberblick

6 Einfache Datentypen

7 Operatoren und Ausdrücke

8 Kontrollstrukturen

9 Funktionen

10 Variablen

11 Präprozessor

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).

goto Instruction

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

- `goto` statement is rarely used in clean code
- Edgar Dijkstra: „Go To Statement Considered Harmful"

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

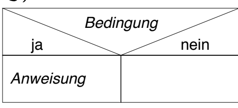
- `goto` leads to hard-to-read code
- A label must not be the function's last statement

- `goto` and `if (...) goto` statements are the only control structures that the hardware can directly execute.
- This aspect is essential for understanding interrupts!

Bedingte Anweisung [=Java]

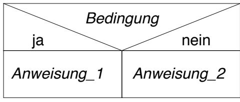
- if-Anweisung (bedingte Anweisung)

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



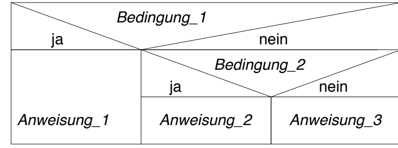
- if-else-Anweisung (einfache Verzweigung)

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



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

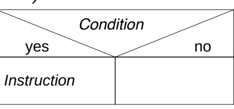
```
if (Bedingung_1)
    Anweisung_1;
else if (Bedingung_2)
    Anweisung_2;
else
    Anweisung_3;
```



Conditional Statements

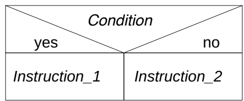
- Minor differences in syntax between C and Java/Python
- `if` statement (conditional statement)

```
if (condition)
    instruction;
```



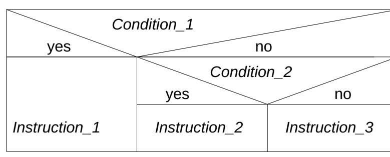
- `if-else` statement (two branches)

```
if (condition)
    instruction1;
else
    instruction2;
```

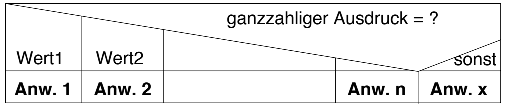


- `if-else-if` cascade (multiple branches)

```
if (condition1)
    instruction1;
else if (condition2)
    instruction2;
else
    instruction3;
```



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




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

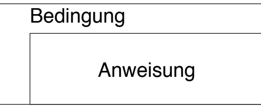
- `switch` statement (case selection)
  - alternative to `if` cascade when testing for integer values
  - Python: `match`/`case`

integer expression = ?				
Value1	Value2			else
Inst. 1	**Inst. 2**		**Inst. n**	**Inst. x**

```
switch (expression) {
case value1:
    instruction1;
    break;
case value2:
    instruction2;
    break;
...
case valuen:
    instructionn;
    break;
default:
    instructionx;
}
```

Abweisende Schleife

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



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

```
while (
  sb_button_getState(BUTTON0)
      == RELEASED
) {
  ··· // repeat until the button is pre
}
```

Nicht-abweisende Schleife

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

Anweisung
Bedingung

`do`
    Anweisung`;`

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

08-Kontrollstrukturen: 2026-04-13

Pre-Condition and Post-Condition Loops

- Pre-condition loop
  - `while`-loop
  - executed zero or more times

Condition	
	Instruction

`while(`*condition*`)`
    instruction`;`

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

- Post-condition loops
  - `do-while` loops
  - executed once or more

	Instruction
Condition	

`do`
    instruction`;`
`while(`*condition*`);`

```
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

Iterating Loops

- C: `for` loop has an explicitly managed counter
- Python: for item in iterable

```
for (starting_expression;
        terminating_expression;
        incrementing_expression)
  instruction;
```

v ← Start expr. (increment) end expr.
Instruction

- Example (usually: *n* executions with counter variable)

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

- Remarks
  - Declaring a variable (`n`) in the *starting_expression* is only possible from C99 onwards.
  - The loop is repeated as long as *terminating_expression* $\neq 0$ (*true*)
    $\leadsto$ the `for` loop is a more explicit `while` loop

- 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);
}
```

0	1	2	3	4	5	6	7
R_0	Y_0	G_0	B_0	R_1	Y_1	G_1	B_1

- 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);
}
```

0	1	2	3	4	5	6	7
R_0	Y_0	G_0	B_0	R_1	Y_1	G_1	B_1

Controlling Loops [=Python]

- The current iteration of the loop can be terminated with the `continue` instruction.

  ⤳ The loop continues with the next iteration

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

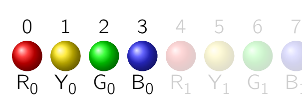
0 1 2 3 4 5 6 7

R_0 Y_0 G_0 B_0 R_1 Y_1 G_1 B_1

- The execution of the *innermost loop* is terminated with the `break` instruction.

  ⤳ The program resumes execution *after* the loop

```
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:
```

Loops & goto Instructions

- All loop types have semantically equivalent sequences with goto statements
- Example:

```
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

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

What is a Function?

- **Function** := subprogram
  - program piece (block) that has an identifier
  - parameter can be passed when calling the function
  - a return value can be passed after finishing
- Functions are elementary program pieces
  - structure extensive tasks into smaller, manageable components
  - enable a simple reuse of components
  - enable a simple exchange of components
  - hide implementation details: **black-box** principle

function ↦ abstraction ↪ 4–1

- Identifier and parameters abstract
  - from the actual program piece
  - from the representation and usage of data
- Enables a step-by-step abstraction and refinement

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

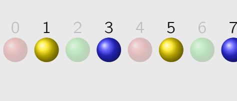
```
{
  uint8_t i = 0;
  for (i = 0; i < 8; i++) {
    if ((setting >> i) & 1) {
      sb_led_on(i);
    } else {
      sb_led_off(i);
    }
  }
}
```

Unsichtbar: Tatsächliche Implementierung

Example

- Function (abstraction) `sb_led_setMask()`

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



- Implementation in `libspicboard`

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

visible: identifier & formal parameters

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

not visible: actual implementation

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++) {
  }
}
```

Function Definitions

- Syntax: *type identifier* **(** $formalParam_{opt}$ **)** **{***block***}**
  - *type* — type of the returned value, `void` if nothing is returned
  - *identifier* — name of the function, which is used to call it ↪ 5–3
  - $formalParam_{opt}$ — list of formal parameters: $type_1\ id_{1\ opt}$, . . ., $type_n\ id_{n\ opt}$ (parameter identifiers are optional) `void`, if no parameter is expected
  - {*block*} — implementation; formal parameters can be used as local variables
- Examples:

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

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

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

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

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

Function Calls

- Syntax: *identifier* **(** *actParam* **)**
  - *identifier* — name of the function to be called
  - *actParam* — list of actual parameters (passed values, have to be compatible in type and count to the list of formal parameters) [=Python]
- Examples:

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

Call of the `max()` function. `47` and `11` are the actual parameters, which are mapped to the formal parameters `a` and `b` of the `max()`-function (↪ 9–3).

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

Error: `text` is not promotable to type `int` actual parameter 2 does not match definition of formal parameter `b` (↪ 9–3)

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

The returned value can be ignored (even though it makes no sense here)

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. ↪

- Des weiteren gilt
  - Arrays werden in C immer *by-reference* übergeben [=Java]
  - Die Auswertungsreihenfolge der Parameter ist undefiniert! [≠Java]

09-Funktionen: 2026-04-13

Function Calls – Passing of Parameters [≠Python]

- General types of parameter passing
  - *call by value* — Formal parameters are copies of the actual parameters. Changes made to the formal parameters are lost when the function is exited.
    This is the standard case in C.
  - *call by reference* — Formal parameters are references to the actual parameters. Changes made to the formal parameters directly affect the actual parameters as well.
    In C possible with the help of pointers.

↪ 13–5

- Note:
  - arrays are always passed *by reference* [≈Python]
  - the order in which parameters are evaluated is undefined!

- Funktionen können sich auch selber aufrufen (Rekursion)

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

Rekursive Definition der Fakultätsfunktion.

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

Rekursion $\mapsto$ $$$

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

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

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

Function Calls – Recursion [=Python]

- Functions can call themselves (recursion)

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

Recursive definition of the factorial function.

A descriptive but really bad example for the use of recursion!

recursion $\mapsto$ $$$

Recursion leads to a significant runtime and memory cost!
For each recursion step:

- Memory has to be provided for: return address, parameters and all local variables
- Parameters are copied, and a function call is performed

Rule: When possible, **avoid any recursion** when writing system-level code!

Funktionsdeklaration [≠Java]

- Funktionen müssen vor ihrem ersten Aufruf im Quelltext deklariert ($\mapsto$ 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:

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

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

```c
// 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;
}
```

Function Declaration

- Functions have to be declared (↦ made known) in the source code prior to being used
  - When a function is defined earlier, this definition serves as declaration
  - Otherwise, (if the function is implemented "further below" in the source code or is defined in another module) it has to be declared explicitly
- Syntax: *type identifier* **(** *formalParam* **)** **;**
- Example:

```c
// declaration by definition
int max(int a, int b) {
  if (a > b) return a;
  return b;
}

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

```c
// explicit declaration
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!

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

```c
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?**

Function Declaration (Forts.)

- Functions ~~shall~~ **should** be declared in the code prior to being used

Attention: C does not enforce this!

- Possibility to call functions that are **not declared** (↪ implicit declaration)
- Such calls are however **not type-safe**
  - the compiler does not know the list of formal parameters
    ⇝ it cannot verify whether the actual parameters match
  - Possibility to pass **anything**
- Modern compilers at least generate a **warning**
  ⇝ Always take compiler warnings seriously!

Function Declaration (Forts.)

- Functions ~~shall~~ **should** be declared in the code prior to being used
- **Example:**

```
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 Function `foo()` is not **declared** ⇝ the compiler **warns** but accepts any actual parameters

9 `foo()` is **defined** with formal parameters `(int, int)`. Everything that is passed as actual parameters will be interpreted as `int`!

10 **What will be printed?**

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!**

09-Funktionen: 2026-04-13

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!**

Function Declaration (Forts.)

- Functions ~~shall~~ **should** be declared in the code prior to being used
  - Functions that are declared with an empty list of formal parameters will also accept any parameter ⇝ **no type safety**
  - The compiler does **not** warn in this case. The problems remain!
- **Example:**

```c
#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
}
```

Function `foo()` has been declared with **empty** list of formal parameters ⇝ this is a formal **valid call!**

Function Declaration (Forts.)

- Functions ~~shall~~ **should** be declared in the code prior to being used
  - Functions that are declared with an empty list of formal parameters will also accept any parameter ⇝ **no type safety**
  - The compiler does **not** warn in this case. The problems remain!

Attention: Risk of confusion

- In Java, `void foo()` defines a **parameterless** method
- In Python, `def foo():` defines a **parameterless** method
  - In C, one has to explicitly write `void foo(void)` ↪
- In C, `void foo()` declares an **open** function
  - This is only useful in (rare) cases!
  - Generally it is considered bad style ⇝ point deduction in exam!

Rule: Functions always need to be **declared fully!**

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

- **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,	[=Java]
	`int` falls kein Typ angegeben wird (↦ schlechter Stil!)	[≠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]

Definition of Variables [≠Python]

- **variable** := container for values (↦ memory space)
- Syntax (definition of variables):

  $sc_{opt}\ type_{opt}\ id_1\ [= expr_1]_{opt}\ [, id_2\ [= expr_2]_{opt}\ , \ldots]_{opt}$;

  - sc_{opt} — storage class of the variable, `auto`, `static`, or none
  - $type$ — type of the variable, `int` if no type is given (↦ bad style!)
  - id_i — name of the variable
  - $expr_i$ — expression for initial value; if no value gets assigned, the content of non-`static` variables is **undefined**

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, shadows 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,
  }                            //    shadows function-local a
}
```

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

Definition of Variables (Forts.)

- Variables can be defined at different positions
  - global — outside of any function, usually at the beginning of the file
  - local — at the beginning of a `{ block }`, directly after the opening curly bracket — C89
  - local — anywhere where an expression is valid — C99

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

void main(void) {
  int a = b;                  // a: local to function, shadows 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,
  }                           //    shadows function-local a
}
```

We will have a closer look at global variables when talking about **modularization** ↪ 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]

main.c → CPP → Compiler → ...

- Bevor eine C-Quelldatei übersetzt wird, wird sie zunächst durch einen Makro-Präprozessor bearbeitet
  - Historisch ein eigenständiges Programm (**CPP** = **C** **Pre****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

The C Preprocessor

- Before a C source file is compiled, it is processed by the macro preprocessor
  - in the past, a stand-alone program (**CPP** = **C** **P**re**P**rocessor)
  - nowadays, integrated into compilers
- The CPP edits the source code by text transformations
  - automatic transformation ("clean-up" of the source code)
    - comments are deleted
    - lines ending with \ are put together
    - ...
  - controllable transformations (by the programmer)
    - preprocessor directives are evaluated and executed
    - preprocessor macros are expanded

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.

Preprocessor Directives [≠Python]

- **preprocessor directive** := control expression for the preprocessor

`#include` <*file*>	**Inclusion:** The contents of *file* are included at this exact place into the token stream.
`#define` *macro replacement*	**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. The *replacement* can also be empty.
`#if` *condition*, `#elif`, `#else`, `#endif`	**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.
`#ifdef` *macro*, `#ifndef` *macro*	Conditional compilation dependent on (defined/not defined) *macro* (z. B. with `#define`).
`#error` *text*	**Abort:** The compilation procedure is aborted with the error message *text*.

> The preprocessor defines an embedded **meta language**. All preprocessor directives (i.e., the meta program) modify the C program (i.e., actual program) prior to actual compilation.

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);          // --> 0F
#endif

}
```

Preprocessor – Example [≠Python]

- Simple macro definitions

empty macro (flag)	`#define USE_7SEG`
source-code constant	`#define NUM_LEDS (4)`
"inline" function	`#define SET_BIT(m, b) (m	(1 << b))`

Preprocessor directives are **not** followed by a semicolon!

- Usage

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

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

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

  - Aber nicht alle

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

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

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

Preprocessor – Risks [≠Python]

- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are expanded textually. Since CPP misses C semantics, expansions can lead to unwanted surprises.

```
#define POW2(a) 1 << a                << has lower precedence than *
n = POW2(2) * 3                      ⇝ n = 1 << 2 * 3
```

  - Some problems can be avoided by the correct use of brackets

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

  - However, not all

```
#define max(a, b) ((a > b) ? a : b)   a++ will be potentially evaluated twice
n = max(x++, 7)                      ⇝ n = ((x++ > 7) ? x++ : 7)
```

- A possible alternative are real `inline` functions C99
  - function's body is directly inserted ⇝ as efficient as macros

```
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, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Sommersemester 2026

http://sys.cs.fau.de/lehre/ss26

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.

Software Design

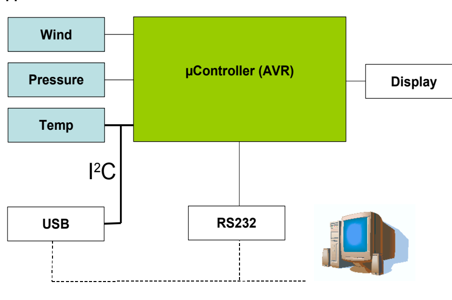
- Software design: general considerations about program's structure **before** the actual programming/implementation
  - Goal: Partitioning of the problem in manageable sub-problems

- There exists a multitude of different approaches for software design
  - Object-oriented approach
    - decomposition into classes and objects
    - designed for Python, Java, or C++
  - Top-down design / functional decomposition
    - state-of-the-art approach until the mid 80s
    - decomposition into functions and function calls
    - design constraints for FORTRAN, COBOL, Pascal, or C

> System-level software is still designed with the **functional decomposition** in mind.

Beispiel-Projekt: Eine Wetterstation

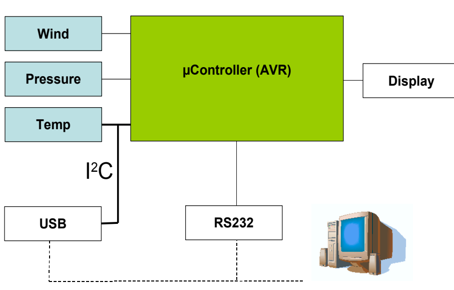
- Typisches eingebettetes System
  - Mehrere Sensoren
    - Wind
    - Luftdruck
    - Temperatur
  - Mehrere Aktuatoren (hier: Ausgabegeräte)
    - LCD-Anzeige
    - PC über RS232
    - PC über USB
  - Sensoren und Aktuatoren an den µC angebunden über verschiedene Bussysteme
    - I^2C
    - RS232



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

Example Project: A Weather Station

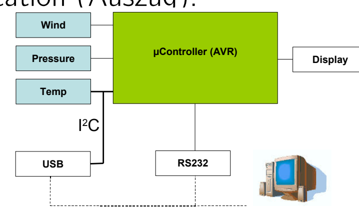
- Typical embedded system
  - multiple sensors
    - air speed
    - air pressure
    - temperature
  - multiple actuators (here: output devices)
    - LCD-screen
    - PC via RS232
    - PC via USB
  - Sensors and actuators are connected to the µC via different bus systems
    - I^2C
    - RS232



What does **functional decomposition** of the software look like?

Funktionale Dekomposition: Beispiel

Funktionale Dekomposition der Wetterstation (Auszug):

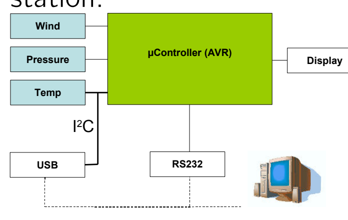


1. Sensordaten lesen
   - 1.1 Temperatursensor lesen
     - 1.1.1 I^2C-Datenübertragung initiieren
     - 1.1.2 Daten vom I^2C-Bus lesen
   - 1.2 Drucksensor lesen
   - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
   - 3.1 Daten über RS232 versenden
     - 3.1.1 Baudrate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

Functional Decomposition: Example

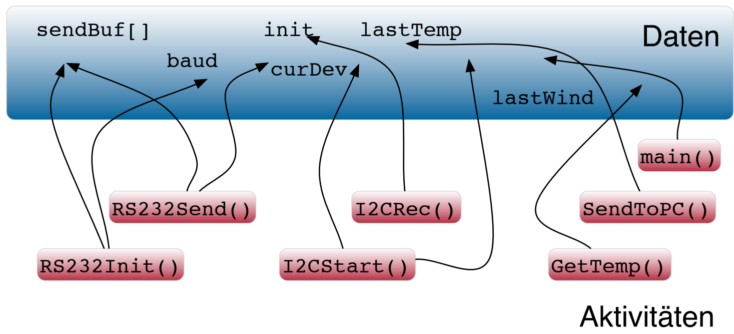
Functional decomposition of the weather station:

1. read sensor data
   1.1 read the temperature sensor
      1.1.1 initialize I^2C data transfer
      1.1.2 read data from the I^2C-bus
   1.2 read the pressure sensor
   1.3 read the air speed sensor
2. process data (z. B. smoothing)
3. output data
   3.1 sending data via RS232
      3.1.1 choose baud rate and parity (once)
      3.1.2 write data
   3.2 refresh the LCD
4. wait and eventually re-start again with step 1



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 ⇝ 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 $\leadsto$ mangelhafte Trennung der Belange

Prinzip der Trennung der Belange

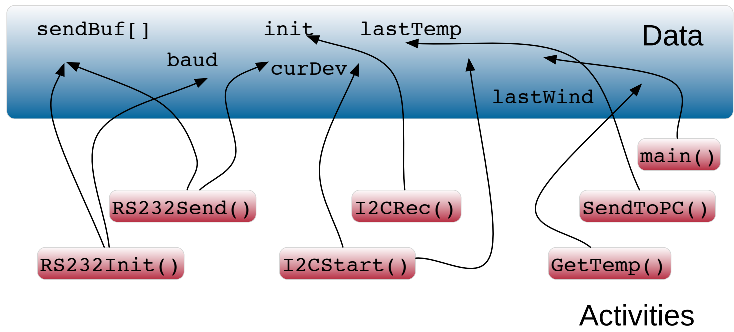
Dinge, die **nichts miteinander** zu tun haben, sind auch **getrennt** unterzubringen!

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

Functional Decomposition: Problems

- The obtained decomposition does only account for the structure of the activities; however, not for the structure of the data
- Risk: Functions "wildly" work on a vast amount of unstructured data

$\leadsto$ inadequate separation of concerns



Functional Decomposition: Problems

- The obtained decomposition does only account for the structure of the activities; however, not for the structure of the data
- Risk: Functions “wildly” work on a vast amount of unstructured data
  $\leadsto$ inadequate separation of concerns

> **Principle of separation of concerns**
>
> Parts that have **nothing in common** with each other should be placed **separately**!
>
> *Separation of concerns* is a **fundamental principle**
> in computer science (likewise in each other engineering discipline).

Zugriff auf Daten (Variablen)

- Variablen haben
  - 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

```c
int a = 0;                  // a: global          ("public")
static int b = 47;          // b: local to module ("private")

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
}
```

Access to Data (Variables)

- Variables have ↪ 10–1
  - Scope “Who can access the variable?”
  - Lifespan “How long is the memory accessible?”
- These are determined by their position (pos) and storage class (sc)

pos	sc ↦	scope	lifespan
local	*none*, `auto`	definition → end of block	definition → end of block
	`static`	definition → end of block	program start → program end
global	*none*	unrestricted	program start → program end
	`static`	whole module	program start → program end

```c
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
}
```

12-Module: 2026-04-13

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**

Acces to Data (Variables) (Forts.)

- Scope and lifespan should be chosen restrictively
  - Scope as restricted as possible!
    - prevent unwanted access from other modules (debug)
    - hide information of implementation (black-box principle, *information hiding*)
  - Lifespan as short as possible!
    - save memory space
    - especially relevant for μ-Controller platforms

↪ 1–4

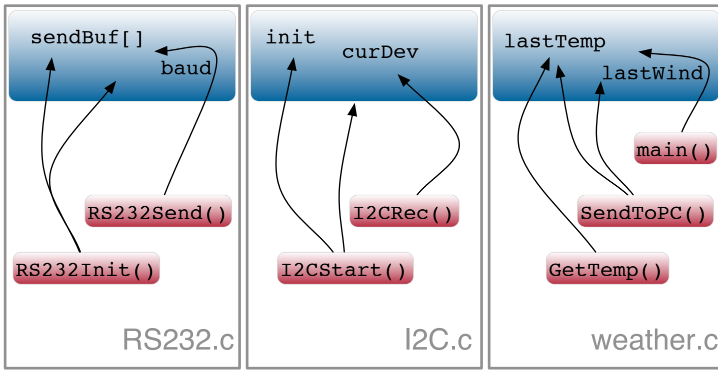
Consequence: Avoid global variables!

- global variables are visible everywhere
- global variables require memory for the entire program execution

Rule: Declaration of variables with **minimal scope & lifespan**

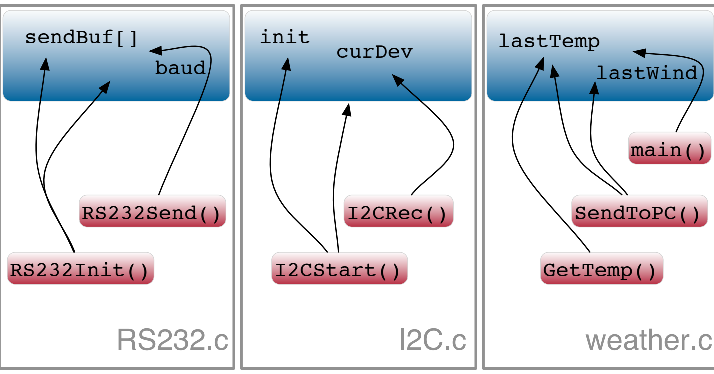
Lösung: Modularisierung

- Separation jeweils zusammengehöriger Daten und Funktionen in übergeordnete Einheiten ⇝ **Module**



Solution: Modularization

- Decomposition of related data & functions into dedicated, surrounding units $\leadsto$ **modules**



Was ist ein Modul?

- **Modul** := (<*Menge von Funktionen*>, <*Menge von Daten*>, <*Schnittstelle*>) (↦ „`class`“ in Java)
- Module sind größere Programmbausteine ↪

  - 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 ↪

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

What is a Module?

- **module** := (<*set of functions*>, <*set of data*>,
  <*interface*>)
- Modules are larger components of programs ↪
  - problem-oriented aggregation of functions and data
    ⇝ separation of concerns
  - enable easy reuse of components
  - enable simple exchange of components
  - hide information of implementation: **black-box** principle
    ⇝ access only by means of the module's interface

Module ↦ Abstraction ↪

- The interface of a module abstracts
  - from the actual implementation of the functions
  - from the internal representation and use of data

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>`

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

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

```c
#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

12-Module: 2026-04-13

Modules in C

- In C, the modules are not part of the language itself, instead it is handled solely idiomatically (by using conventions) ↪ 3–16
  - module interface ↦ .h-file (contains declarations ↪ 9–7)
  - module implementation ↦ .c-file (contains definitions ↪ 9–3)
  - module usage ↦ `#include <module.h>`

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

RS232.h: Interface / Contract (public)
Declaration of provided functions (and data)

```c
#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;
  ...
}
```

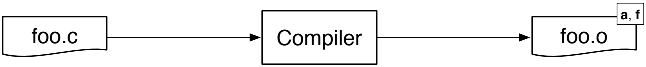
RS232.c: Implementation (not public)
Definition of provided functions (and data)

Possible module-internal helper functions and variables (`static`)

Inclusion of the own interface ensures that the contract is adhered to

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)

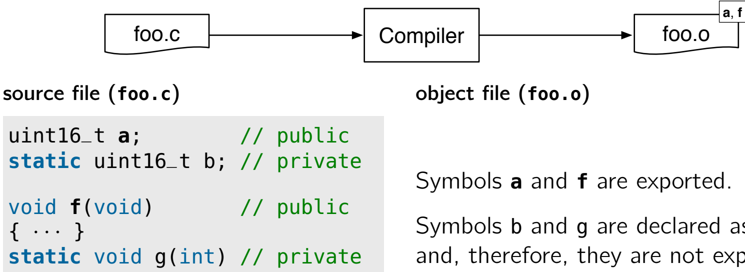
Symbole **a** und **f** werden exportiert.

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

12-Module: 2026-04-13

Modules in C – Export

- C module exports a set of defined symbols
  - all functions and global variables
  - export can be prevented with `static` (↦ "__" **convention** in Python) (↦ restriction of scope ↪ 12–5)
- Export takes place during compilation (`.c` file ⟶ `.o` file)



source file (`foo.c`)

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

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

object file (`foo.o`)

Symbols **a** and **f** are exported.

Symbols `b` and `g` are declared as `static` and, therefore, they are not exported.

- 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.

Modules in C – Import

- C module imports a set of not-defined symbols
  - functions and global variables that are used but not defined in the module itself
  - during compilation, they are marked as unresolved

source file (bar.c)

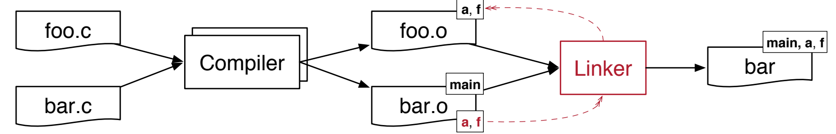
```
extern uint16_t a;   // declare
extern void f(void); // declare

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

object file (bar.o)

Symbol **main** is exported.
Symbols **a** and **f** are unresolved.

Module in C – Import (Forts.) [≠Java]

- Die eigentliche Auflösung erfolgt durch den Linker



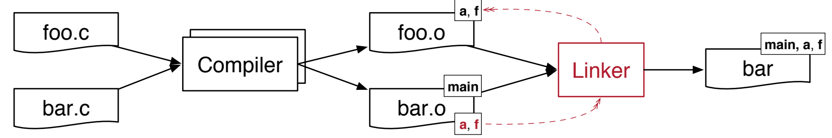
Linken ist nicht typsicher!

- 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

- The actual resolution is performed by the linker



Linking is **not type safe!**

- Information about types is not anymore present in the object files
- Resolution by the linker takes place **exclusively** via **names of symbols** (identifier)

⇝ type safety has to be ensured during **compilation**

⇝ uniform declaration with the help of a common header file

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)

Modules in C – Header [≠Python]

- Elements from other modules have to be declared
  - functions with the `extern` declaration ↪ 9–7

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

  - global variables with `extern`

```
extern uint16_t a;
```

> The keyword `extern` differentiates between a declaration and definition of a variable.

- Declarations are usually part of the header file, which module developers make available
  - interface of the module
    - exported functions of the module
    - exported global variables of the module
    - module-specific constants, types, and macros
    - usage by including (↦ "`import`" in Python)
  - is **included by the module itself** to ensure a match of declaration and definition

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;      (durchgestrichen)
extern void f(void);    (durchgestrichen)
#include <foo.h>

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

module interface: `foo.h`

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

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

#endif // _FOO_H
```

module implementation `foo.c`

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


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

module usage `bar.c`
(compare for ↪ 12–11)

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

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

Zurück zum Beispiel: Wetterstation

Quellmodule —— ***preprocess compile*** → *Objektmodule* — ***link*** → *ELF-Binary*

RS232.h
RS232.c
I2C.h
I2C.c
weather.c
gcc (Compiler)
RS232.o
I2C.o
weather.o
ld (Linker)
weather
io.h
... .h
io.o ... .o ... .o ... .o ... .o ... .o
avr-libc.lib

- 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

Back to the Example: Weather Station

Source modules — preprocess compile → Object modules — link → ELF binary

RS232.h, RS232.c
I2C.h, I2C.c
weather.c
gcc (compiler)
RS232.o, I2C.o, weather.o
ld (linker)
weather
io.h, ...h
io.o, ...o, ...o, ...o, ...o, ...o
avr-libc.lib

- Each module consists of a header and one or more implementation file(s)
  - `.h` file **defines the interface**
  - `.c` file **implements the interface**, includes the `.h`-file to ensure a match of declaration and definition
- Usage of the module by including the specific `.h` file
- This is similar for libraries

Zusammenfassung

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

Summary

- Principle of separation of concerns ⤳ modularization
  - reuse and exchange of well-defined components
  - hiding of implementation details
- In C, the concept of modules is not part of the language. Therefore, it is realized idiomatically by conventions.
  - module interface ↦ `.h`-file (contains declarations)
  - module implementation ↦ `.c`-file (contains definitions)
  - use of module ↦ `#include <module.h>`
  - private symbols ↦ define as `static`
- The actual combination is done by the linker
  - resolution exclusively by symbol names
    ⤳ **Linking is not type safe!**
  - type safety has to be ensured during compilation
    ⤳ with the help of a common header file

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

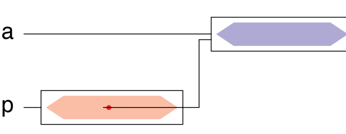
Einordnung: Zeiger (*Pointer*)

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

'a' ≡ 0110 0001

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

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

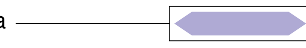


Classification: Pointers

- **Literal:** `'a'`
  representation of a value

'a' ≡ 0110 0001

- **Variable:** `char a;`
  container for a value



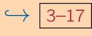
- **Pointer variable:**
  `char *p = &a;`
  container for a reference
  to a variable

a

p

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!

Pointers

- A *pointer* variable contains a memory address of a different variable as its value
  - A pointer points to another variable (in memory)
  - With the address, an indirect access to the target variable (its memory) is possible
- Therefore pointers are of **major relevance** for C programming
  - Functions now can change variables of the caller (*call by reference*) ↪ 9–5
  - Memory can be addressed directly
  - More efficient programs

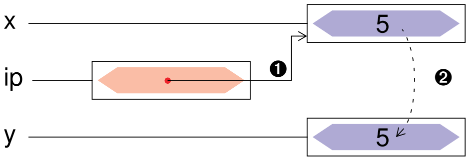
“Efficiency by machine orientation” ↪ 3–17

- However, pointers lead to many problems!
  - Structure of programs gets complicated (which functions can access which variables?)
  - Pointers are the **most common cause for errors** in C programs!

Definition von Zeigervariablen

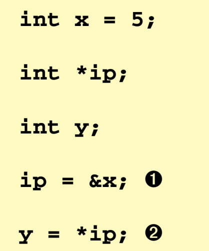
- **Zeigervariable** := Behälter für Verweise ($\mapsto$ Adresse)
- Syntax (Definition): *Typ* ******Bezeichner***;**
- Beispiel

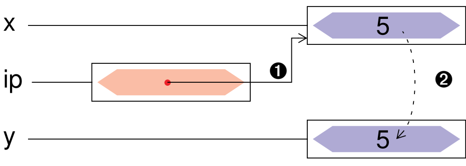
```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



Definition of Pointer Variables

- **Pointer variable** := container for reference ($\mapsto$ address)
- Syntax (definition): *type* ******identifier*;
- Example





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.

Address and Reference Operators

- *Address-of operator*: **&x** — The unary **&** operator provides the reference (↦ address in memory) of the variable **x**.

  also named *address operator*, memory aid: **&**ddress operator

- *Dereference operator*: **∗y** — The unary ∗ operator provides the target variable (↦ memory cell / container), to which the pointer **y** points (dereferencing).

- Valid code: **(∗(&x))** ≡ **x** — The reference operator is the inverse operation to the address operator.

Attention: Risk of Confusion (*** *I see stars everywhere* ***)

The ∗ symbol has different meanings in C **depending on the context**:

1. Multiplication (binary):	`x * y`	in expressions
2. Type modifier:	`uint8_t *p1, *p2` `typedef char *CPTR`	in definitions and declarations
3. Reference (unary):	`x = *p1`	in expressions

In particular 2. and 3. often cause confusion

⇝ * is erroneously considered as part of the identifier.

13-Zeiger: 2026-04-13

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**

Pointers as Function Arguments

- In C, parameters are always passed *by value* ↪ 9–5
  - Values of parameters are copied to local variables of the called function
  - The called function cannot change the actual parameters of the calling function

- This is also true for pointers (references)
  - The called function receives a copy of the address reference
  - With help of the ***** operators, the target variable can be accessed and its value can be changed

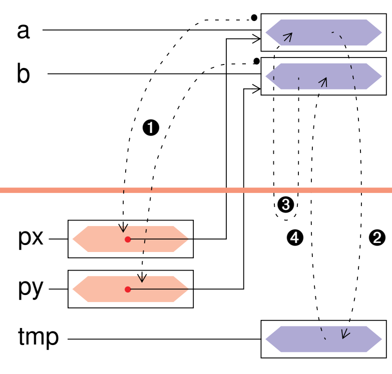
    ⇝ **call by reference**

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

}
```



13-Zeiger: 2026-04-13

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;
```


13-Zeiger: 2026-04-13

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;
```

a 47
b 11
px
py
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
```

a → 11
b → 11
❸
px
py
tmp → 47

Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

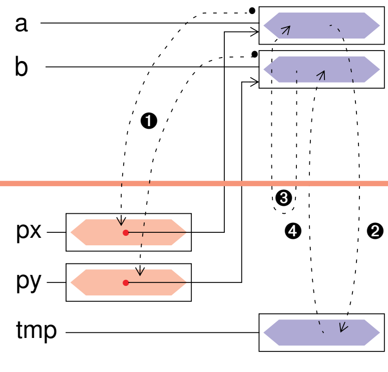
```
void swap (int *px, int *py)
{
    int tmp;

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

}
```

Pointers as Function Arguments (Forts.)

- Example (overview)

```
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

}
```



Pointers as Function Arguments (Forts.)

- Example (step by step)

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

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

Pointers as Function Arguments (Forts.)

- Example (step by step)

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

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

a 47
b 11
px
py
tmp

Pointers as Function Arguments (Forts.)

- Example (step by step)

```
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
```

a 47
b 11
*px
px
py
tmp

Pointers as Function Arguments (Forts.)

- Example (step by step)

```
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
```

Pointers as Function Arguments (Forts.)

- Example (step by step)

```
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
```


13-Zeiger: 2026-04-13

Pointers as Function Arguments (Forts.)

- Example (step by step)

```
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
}
```

Arrays

- **Array variable** := container for a list of values of same type
- Syntax (definition): *type identifier* **[** *IntExpression* **] ;**
  - *type* — type of the values [=Python]
  - *identifier* — name of the array variable [=Python]
  - *IntExpression* — **constant** integer expression, defines the size of the array (↦ number of elements). [≠Python]

    From **C99** onwards, the *IntExpression* of `auto` arrays can be chosen **variably** (d. h. arbitrary, but constant).
- Example:

```
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 };
```

Array Initialization

- Like other variables, an array can receive a set of initial values during definition

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

- If not all initializing elements are given, the remainder is initialized with 0

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

- If the explicit dimension of the array is omitted, the number of initializing elements determines the size

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

Feldzugriff

- Syntax: *Feld* **[** *IntAusdruck* **]** [=Java]
  - Wobei $0 \leq \mathit{IntAusdruck} < n$ 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!!!
```

13-Zeiger: 2026-04-13

Access to Arrays

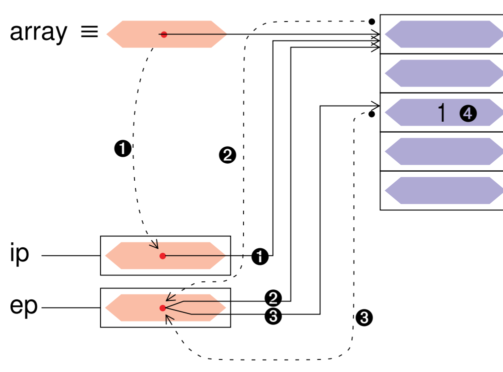
- Syntax: *array* **[** *IntExpression* **]** [=Python]
  - With $0 \leq \mathit{IntExpression} < n$ for $n =$ size of the array
  - **Attention:** The index is not checked [≠Python]
    ⇝ common cause for errors in C programs
- Example

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

LEDs[3] = BLUE1;

for (uint8_t i = 0; i < 4; i++) {
  sb_led_on(LEDs[i]);
}

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

Felder sind Zeiger

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

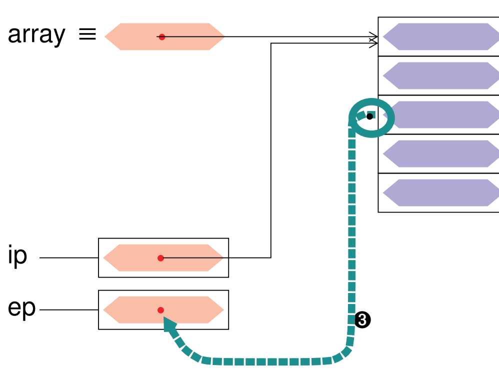
```
int array[5];

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)

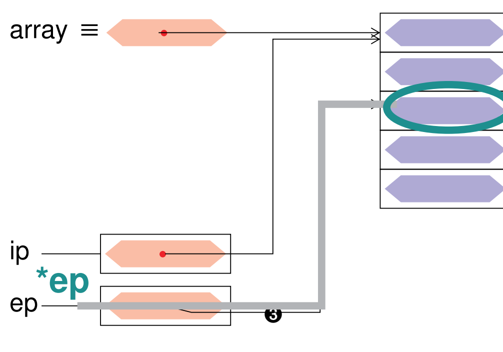
```
int array[5];

int *ip = array; ➊

int *ep;
ep = &array[0]; ➋

ep = &array[2]; ➌
```



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; ➊

int *ep;
ep = &array[0]; ➋

ep = &array[2]; ➌

*ep = 1; ➍
```

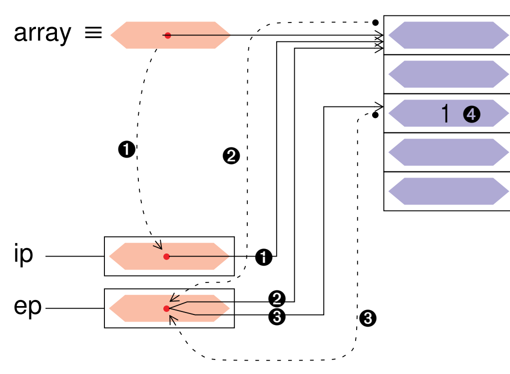


13-Zeiger: 2026-04-13

Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (overview)

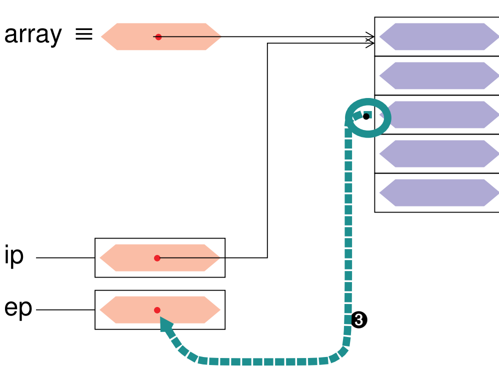
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` ≡ `&array[0]`
  - An alias – not a container ⇝ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (step by step)

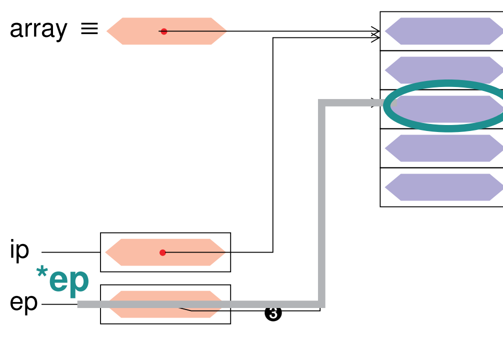
```
int array[5];

int *ip = array; ➀

int *ep;
ep = &array[0]; ➁

ep = &array[2]; ➌
```



Arrays are Pointers

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` $\equiv$ `&array[0]`
  - An alias – not a container $\leadsto$ value cannot be changed
  - Via such a pointer, the indirect access to array cells is possible
- Example (step by step)

```
int array[5];

int *ip = array; ➊

int *ep;
ep = &array[0]; ➋

ep = &array[2]; ➌

*ep = 1; ➍
```



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]);
}
```

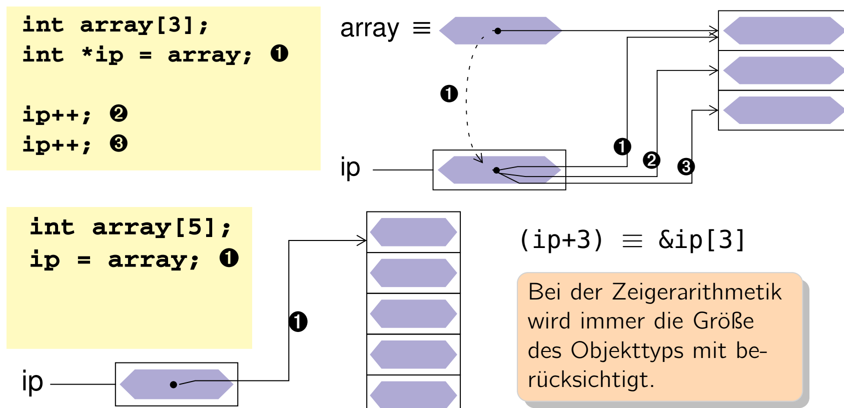
Pointers are Arrays

- The identifier of an array is syntactically equivalent to a constant pointer to the first element of the array: `array` $\equiv$ `&array[0]`
- This relation is valid in both directions: `*array` $\equiv$ `array[0]`
  - A pointer can be used like an array
  - In particular, the **[ ]**-operator can be used ↪ 13–9
- Example (see ↪ 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 $\leadsto$ Ihr Wert ist veränderbar
- Neben einfachen Zuweisungen ist dabei auch Arithmetik möglich



Arithmetic Operations on Pointers

- In contrast to the identifier of an array, a pointer *variable* is a container $\leadsto$ its value can be modified
- Besides simple assignments, arithmetic operations are possible

```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



```
int array[5];
ip = array; ❶
```


`(ip+3)` ≡ `&ip[3]`

When using arithmetic operations on pointers, the size of the type of one object is always taken into account.

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

Pointer Arithmetic – Operations

- Arithmetic operations

  $++$ pre/post increment

  $\leadsto$ shift to the next object

  $--$ pre/post decrement

  $\leadsto$ shift to previous object

  $+, -$ addition / subtraction of an `int` value

  $\leadsto$ resulting pointer is moved by n objects

  $-$ subtraction of two pointers

  $\leadsto$ number of objects n between the pointers (distance)

- Comparison operators: $<, <=, ==, >=, >, !=$ ↪ 7–3

  $\leadsto$ pointers can be compared and ordered like integers

Felder sind Zeiger sind Felder – Zusammenfassung

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

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

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

Fehler: `array` ist konstant!

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

Arrays are Pointers are Arrays – Summary

- In combination with arithmetic operations for pointers, each array operation can be mapped to an equivalent pointer operation.

- For `int i, array[N], *ip = array;` with $0 \leq i < N$ holds:

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

Error: `array` is constant!

- In contrary, pointer operations can be represented by array operations.
  However, the identifier of the array cannot be modified.

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);
}
```

R_0 Y_0 G_0 B_0 R_1 Y_1 G_1 B_1

- 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)

Arrays as Function Arguments

- Arrays are **always** passed as pointers in C

```
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);
}
```

R_0 Y_0 G_0 B_0 R_1 Y_1 G_1 B_1

- Information on size of the array is lost!
  - The size has to passed explicitly as another parameter
  - In some cases, the size can be calculated inside the function (z. B. by searching for the terminating `NUL` symbol at the end of a string)

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)

Arrays as Function Arguments (Forts.)

- Arrays are **always** passed as pointers in C
- If the parameter is declared as `const`, the function cannot modify the elements of the array ↦ good style!

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

- To clarify, that an array (and not a "pointer to a variable") is expected, one can use the following equivalent syntax:

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

- Attention: This is only valid for declaring function parameters
- For defining variables, `array[]` has a entirely different meaning (identifying size of the array from list of initializers ↪ 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" ≡ h | a | l | l | o | \0

↪ 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;
}
```

Arrays as Function Arguments (Forts.)

- The function `int strlen(const char *)` from the standard library provides the number of characters of the passed string

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

It holds: "hello" ≡ h e l l o \0 ↪ 6–13

- Variants of implementation

option 1: array syntax

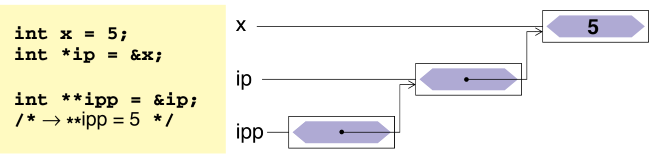
```
int strlen(const char s[]) {
  int n = 0;
  while (s[n] != '\0')
    n++;
  return n;
}
```

option 2: pointer 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



- 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

Pointers to Pointers

- A pointer can point to another pointer variable

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

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


- This is particularly useful for passing parameters to functions
  - pointer parameter is passed *call by reference*
    (z. B. `swap()` function for pointers)
  - passing an array of pointers

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
}
```

Pointers to Functions

- A pointer can point to a function
  - With this feature, functions are passed as parameters to other functions
    ↦ functions of higher order
- Example

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

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

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

Zeiger auf Funktionen (Forts.)

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

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

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

Pointers to Functions (Forts.)

- Syntax (definition): *type* **(****identifier* **)(***formalParam$_{opt}$***);**
  (similar to function definitions)

↪ 9–3

  - *type* — return value of the functions the pointer can point to
  - *identifier* — name of the function pointer
  - *formalParam$_{opt}$* — formal parameters of the functions the pointer can point to: *type*$_1$,..., *type*$_n$

- A function pointer is used in the same way as a function
  - call with *identifier***(***actParam***)** ↪ 9–4
  - address (&) and reference operator (*) are not required ↪ 13–4
  - an identifier of a function is a constant pointer to that function

```
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
}
```

Pointers to Functions (Forts.)

- Function pointers are often used for callback functions to deliver asynchronous events ($\mapsto$ “listener” pattern)

```
// 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

Summary

- A pointer references a variable in memory
  - possibility for indirect access to a value
  - basis for implementation of *call-by-reference* in C
  - basis for implementation of arrays
  - **important part of the machine orientation** of C
  - **most common cause for errors in C programs!**
- The syntactical possibilities are diverse (and confusing)
  - type modifier *, address operator &, reference operator *
  - pointer arithmetic with `+`, `-`, `++`, and `--`
  - syntactical equivalence between pointers and arrays (`[]` Operator)
- Pointers can point to functions
  - pass functions to functions
  - principle of callback functions

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

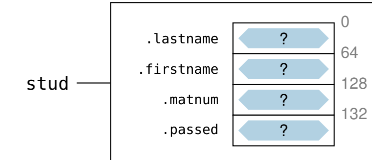
Strukturen: Motivation

- 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)

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

// Variable definition
struct Student stud;

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

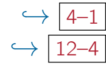
Ein Strukturtyp fasst eine Menge von Daten zu einem gemeinsamen Typ zusammen.

Die Datenelemente werden hintereinander im Speicher abgelegt.



Structs: Motivation

- If variables “somehow” belong together,
  - intuitive approach to structure together
  - problem-specific abstraction
  - separation of concerns


↪ 4–1
↪ 12–4

- C makes this possible with composite types: structs

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

// Variable definition
struct Student stud;

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

A structure type combines a set of data with a common type.

The elements are placed subsequently in memory.

stud

		offset
.lastname	?	0
.firstname	?	64
.matnum	?	128
.passed	?	132

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 };
```

Structs: Variable Definitions and Initialization

- Similar to an array, a structure variable is initialized during definition ↪ 13–8

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

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

The initializers are only assigned by order, not by their identifier. ⇝ potential source of errors when changing code!

- In analogy to the definition of `enum` types, the use is simplified with the keyword `typedef` ↪ 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

portA — .pin → PINA; .ddr → 0 DDRA; .port → 0 PORTA

- 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 *

Structs: Access to Elements

portA
.pin
.ddr
.port
PINA
0
DDRA
0
PORTA

- The elements of a structure is accessed with the .-operator [≈Python]

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

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

Note: . has higher precedence than *

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 *

Structs: Access to Elements

pport portA .pin .ddr .port PINA 0 DDRA 0 PORTA

- Parentheses are required when working with pointers to structures

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

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

- The **->** operator simplifies this access: `s->m` ≡ `(*s).m`

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

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

-> **also** has a higher precedence than *

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); ··· }
```

14-Strukturen: 2026-04-13

Structs as Function Arguments

- In contrast to arrays, structs are passed *by value*

```
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); ··· }
```

- This is **highly inefficient** when working with larger structures
  - z. B. `Student` (↪ 14–1): 134 byte have to be allocated and copied with each function call
  - better solution: pass a pointer to the constant structure

```
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;
```

Bit-Structures: Bit Arrays

- Single structure elements can be (un)set with granularity of bits
  - the compiler combines bit fields as suitable integer types
  - useful for accessing a range of bits of a register

- Example
  - EICRA **External Interrupt Control Register A**
    Controls triggers for external interrupt sources `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"

What does a µC understand? Compiler's Job?

- **µ-Controller cannot directly work on variables in memory**
- **µ-Controller can only make arithmetic operations on registers**
- assembly code contains **load-from** & **store-to sequences**
- compiler's job: decomposition of the program into smaller instructions that can be executed by a µ-Controller.
- example 1: decomposition of an expression

```
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, ...` : "variables in memory"

`r0, r1, ...` : "variables in registers"

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:
```

Example: Decomposing Operations

- example 2: decomposition of a control structure

```
    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;`
- ...

Typical Operations on Registers

- abbreviated with `r1` or alternatively with `R1`
- here: `N` possible registers
- `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.

Typical Assembly Instructions of µ-Controllers

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
...	...	

- all available instructions: see manual of processor/µ-Controller

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:`	

15-MC-Vorbemerkungen: 2026-04-13

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	`...`

C Code & Assembly Code

- example program:

	C code	assembly 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.

Assembler's Job: Encoding Instructions

- example program:

assembly code		binary 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
...	...	...

- encoding of instructions is listed in µ-Controller's manual

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

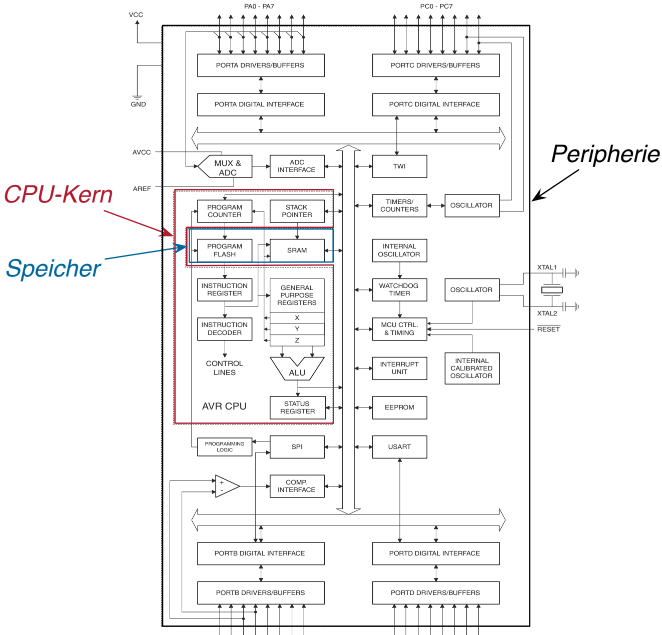
17 μC-Systemarchitektur – Peripherie

18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Beispiel ATmega32: Blockschaltbild



Example ATmega32: Block Circuit Diagram

CPU core

Memory

Peripherals

PA0 - PA7
PC0 - PC7
VCC
GND
AVCC
AREF
PORTA DRIVERS/BUFFERS
PORTA DIGITAL INTERFACE
PORTC DRIVERS/BUFFERS
PORTC DIGITAL INTERFACE
MUX & ADC
ADC INTERFACE
TWI
PROGRAM COUNTER
STACK POINTER
TIMERS/ COUNTERS
OSCILLATOR
PROGRAM FLASH
SRAM
INTERNAL OSCILLATOR
XTAL1
INSTRUCTION REGISTER
GENERAL PURPOSE REGISTERS
X
Y
Z
WATCHDOG TIMER
OSCILLATOR
XTAL2
INSTRUCTION DECODER
MCU CTRL. & TIMING
RESET
CONTROL LINES
ALU
INTERRUPT UNIT
INTERNAL CALIBRATED OSCILLATOR
AVR CPU
STATUS REGISTER
EEPROM
PROGRAMMING LOGIC
SPI
USART
COMP. INTERFACE
PORTB DIGITAL INTERFACE
PORTD DIGITAL INTERFACE
PORTB DRIVERS/BUFFERS
PORTD DRIVERS/BUFFERS
PB0 - PB7
PD0 - PD7

Beispiel ATmega-Familie: CPU-Architektur

Adresse	Programmadressraum (Flash)
0x0000	IRQ-Vektoren
0x0046	main: … wait: … Anwendungs-Code
	Boot-Code
0x1FFFF	

Datenbus: 16 (Flash ↔ Recheneinheit), 8 (Recheneinheit ↔ SRAM)

Recheneinheit

Adressbus: 16 (Flash ↔ Recheneinheit), 16 (Recheneinheit ↔ SRAM)

Adresse	Datenadressraum (SRAM)
0x0000	32 Register
0x0020	64 IO-Register
0x0060	bis zu 160 Ext. IO-Register
0x0100	bis zu 4KiB internes SRAM
0x1100	bis zu knapp 60 KiB externes SRAM
0xFFFF	

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

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

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

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

Example ATmega Family: CPU Architecture

Address	Program address space (Flash)
0x0000	IRQ vectors
0x0046	main: … wait: … Application code
	Boot code
0x1FFFF	

Data bus

16 / 8

Computing unit

16 / 16

Address bus

Address	Data address space (SRAM)
0x0000	32 registers
0x0020	64 IO registers
0x0060	up to 160 ext. IO registers
0x0100	up to 4KiB internal SRAM
0x1100	up to approx. 60 KiB external SRAM
0xFFFF	

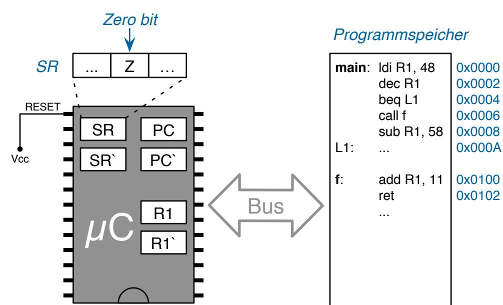
Program address space (Flash)
1–128 KiB, 2-Byte elements

Data address space (SRAM)
0–64 KiB, 1-Byte elements

- Harvard architecture (memory for code and data is separated)
- peripheral registers are integrated into the memory
  ⇝ can be accessed like global variables

In comparison: PC is based on a von-Neumann-architecture that uses shared memory; I/O registers use a special I/O address space.

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 ⇝ Programm arbeitet nur auf Registern

Wie arbeitet ein Prozessor?

Wie arbeitet ein Prozessor?

Zero bit

SR | ... | Z | ...

RESET

Vcc

SR | PC

SR` | PC`

μC | R1 | R1`

Bus

Programmspeicher

main:	ldi R1, 48	0x0000
	dec R1	0x0002
	beq L1	0x0004
	call f	0x0006
	sub R1, 58	0x0008
L1:	...	0x000A
f:	add R1, 11	0x0100
	ret	0x0102
	...	

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

Wie arbeitet ein Prozessor?

Zero bit

SR | ... | Z | ...

RESET

Vcc

SR PC
SR` PC`
R1 48
R1`

μC

Bus

Programmspeicher

main:	ldi R1, 48	0x0000
	dec R1	0x0002
	beq L1	0x0004
	call f	0x0006
	sub R1, 58	0x0008
L1:	...	0x000A
f:	add R1, 11	0x0100
	ret	0x0102
	...	

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

Wie arbeitet ein Prozessor?

Zero bit

SR | ... | Z | ... |
0

RESET
Vcc

SR PC
SR` PC`
R1 47
R1`
μC

Bus

Programmspeicher

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

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

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

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?

Zero bit

SR | ... | Z | ... |
Z = 0

RESET
Vcc

SR PC
SR` PC`
R1 58
R1`
µC

Bus

Programmspeicher

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

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

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

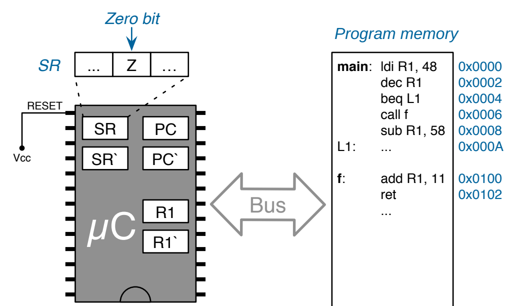
w: dec <R>
R –= 1
if(R == 0) Z = 1
else Z = 0

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

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

w: ret
PC = PC`

How Does a Processor Work?



- example assumes a simplified pseudo processor
  - only one multi-purpose register (R1)
  - program counter (PC) and status register (SR) (+ "shadow copies")
  - no data memory, no stack $\leadsto$ program only works on registers

How Does a Processor Work?

Zero bit

SR | ... | Z | ... |

RESET

Vcc

µC

SR | PC

SR` | PC`

R1

R1`

Bus

Program memory

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

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

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

How Does a Processor Work?

Zero bit

SR | ... | Z | ...

RESET

Vcc

SR PC

SR` PC`

μC

R1

R1`

Bus

Program memory

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

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

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

How Does a Processor Work?

RESET → PC = 0x0000

w = *PC++

decode(w)

execute(w)

Zero bit

SR: | ... | Z | ... |

RESET

Vcc

μC: SR, PC, SR`, PC`, R1 (48), R1`

Bus

Program memory

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

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

How Does a Processor Work?

RESET → PC = 0x0000 → w = *PC++ → decode(w) → execute(w) → (loop back)

Zero bit

SR: ... | Z | ... (Z = 0)

RESET

Vcc

SR PC
SR`  PC`
R1 47
R1`

μC

Bus

Program memory

main:	ldi R1, 48	0x0000
	dec R1	0x0002
	beq L1	0x0004
	call f	0x0006
	sub R1, 58	0x0008
L1:	...	0x000A
f:	add R1, 11	0x0100
	ret	0x0102
	...	

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

How Does a Processor Work?

RESET → PC = 0x0000

Zero bit

SR | ... | Z | ... |

Z = 0

RESET

Vcc

μC: SR, PC, SR`, PC`, R1 (47), R1`

Bus

Program memory

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

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

w = *PC++

decode(w)

execute(w)

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

How Does a Processor Work?

Zero bit

SR | ... | Z | ... |
0

RESET

Vcc

SR PC

SR` PC`

R1 58

R1`

µC

Bus

Program memory

main:	ldi R1, 48	0x0000
	dec R1	0x0002
	beq L1	0x0004
	call f	0x0006
	sub R1, 58	0x0008
L1:	...	0x000A
f:	add R1, 11	0x0100
	ret	0x0102
	...	

RESET

PC = 0x0000

w = *PC++

decode(w)

execute(w)

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

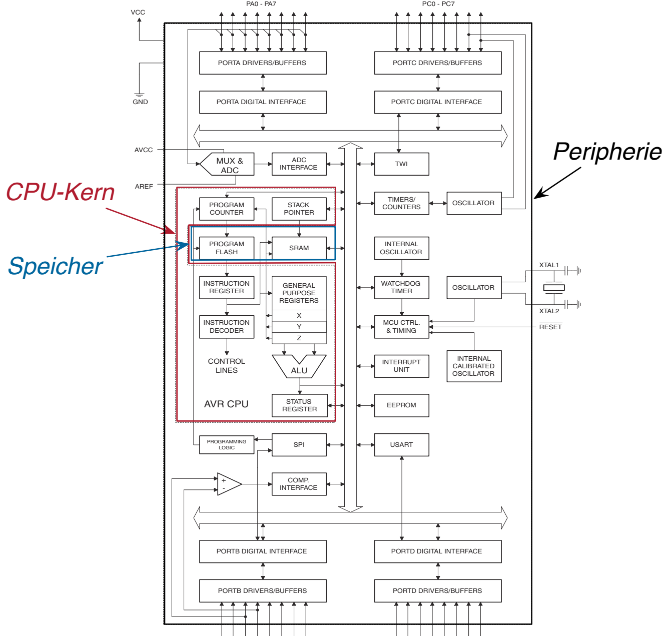
Was ist ein µ-Controller?

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

What is a µ-Controller?

- **µ-Controller** := processor + memory + **peripherals**
  - in fact, computer system on one chip ⟶ *System-on-a-Chip* (SoC)
  - often usable without additional external components, like z. B. timers and memory ⇝ cost-efficient system design
- main features are (plentiful) integrated input/output components (peripherals)
- distinctions are not fixed: processor ⟷ µC ⟷ SoC
  - AMD64 CPUs also have included timers, memory (cache), ...
  - some µC execute at speeds close to "large processors"

Beispiel ATmega32: Blockschaltbild



Example ATmega32: Block Circuit Diagram

CPU core

Memory

Peripherals

PA0 - PA7
PC0 - PC7
VCC
GND
AVCC
AREF
PORTA DRIVERS/BUFFERS
PORTA DIGITAL INTERFACE
PORTC DRIVERS/BUFFERS
PORTC DIGITAL INTERFACE
MUX & ADC
ADC INTERFACE
TWI
PROGRAM COUNTER
STACK POINTER
TIMERS/ COUNTERS
OSCILLATOR
PROGRAM FLASH
SRAM
INTERNAL OSCILLATOR
XTAL1
INSTRUCTION REGISTER
GENERAL PURPOSE REGISTERS
X
Y
Z
WATCHDOG TIMER
OSCILLATOR
XTAL2
INSTRUCTION DECODER
MCU CTRL. & TIMING
RESET
CONTROL LINES
ALU
INTERRUPT UNIT
INTERNAL CALIBRATED OSCILLATOR
AVR CPU
STATUS REGISTER
EEPROM
PROGRAMMING LOGIC
SPI
USART
COMP. INTERFACE
PORTB DIGITAL INTERFACE
PORTD DIGITAL INTERFACE
PORTB DRIVERS/BUFFERS
PORTD DRIVERS/BUFFERS
PB0 - PB7
PD0 - PD7

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

Peripheral Devices

- **peripheral device:** hardware component that is located “outside” of the central unit of a computer
  - traditional (laptop): keyboard, monitor, ... (↦ physically “outside”)
  - in general: hardware functions that are not directly mapped into the processor’s instruction set (↦ logically “outside”)

- peripheral components are addressed via I/O registers
  - control registers: instructions to control/query state of peripheral is encoded by bit patterns (z. B. **`DDRD`** for ATmega)
  - data registers: required for exchange of data (z. B. **`PORTD`**, **`PIND`** for ATmega)
  - registers are often only available as *read-only* or *write-only*

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. I²C) Protokoll.
A/D-Wandler	Bausteine zur momentweisen oder kontinuierlichen Diskretisierung von Spannungswerten (z. B. 0–5V ↦ 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. ↪ 17–11
Bus-Systeme	SPI, RS-232, CAN, Ethernet, MLI, I²C, ...

Peripheral Devices: Examples

- typical peripheral devices of µ-Controllers
  - timer/counter — counting registers that are incremented with a defined frequency (timer) or by external signals (counter) and that trigger an interrupt at a configurable counting value.
  - watchdog timer — timer that has to be written to regularly otherwise a RESET is triggered ("dead man's button").
  - (a)synchronous serial interface — component for serial (bitwise) exchange of data with a synchronous (z. B. RS-232) or asynchronous (z. B. I^2C) protocol.
  - A/D converter — component for one-time/continuous discretization of voltage values (z. B. 0–5V $\mapsto$ 10-bit integer).
  - PWM generators — component for generating pulse-width–modulated signals, pseudo-analog (D/A) output.
  - ports — groups of usually 8 ports which can be set to GND or Vcc and whose states can be monitored. ↪ 17–11
  - bus systems — 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]

Peripheral Devices – Registers

- different architectures for accessing I/O registers
  - memory-mapped: (most µC) — Registers are integrated into address space; access with memory instructions of the processor (`load`, `store`)
  - port-based: (x86-based laptops) — Registers are organized in a separate I/O address space; access with special `in`- and `out`-instructions
- addresses of the registers are listed in the hardware's documentation

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 $x = D$)

- **DDR**x **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

- **PORT**x **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

- **PIN**x **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]

- for every port x, three registers are defined (example for $x = D$)
  - DDRx **Data Direction Register:** Determines for every pin i whether it is used as input (bit i=0) or output (bit i=1).

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:** If pin i is configured to be an output, bit i determines the voltage level (0=GND sink, 1=Vcc source). If pin i is configured to be an input, bit i activates the internal pull-up resistor (1=active).

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 represents the voltage level at pin i (1=*high*, 0=*low*), independent of the data direction of the register.

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

Examples: ↪ 3–7 and ↪ 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 *) 0x12` — Adresse: volatile uint8_t *(Cast ↪ 7–19)
- `(*(volatile uint8_t *) 0x12)` — Wert: volatile uint8_t (Dereferenzierung ↪ 13–4)

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

- Beispiel

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

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

Peripheral Devices – Registers (Forts.)

- Memory-mapped registers enable convenient access
  - register ↦ memory ↦ variable
  - all C operators are directly available (z. B. `PORTD++`)

- Syntactically, preprocessor macros simplify the access:

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


Therefore `PORTD` is syntactically-equivalent to a `volatile uint8_t`-variable that is stored at address `0x12`.

- Example

```
#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
```

17-MC-Peripherie: 2026-04-13

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.

Access to Registers and Concurrency

- Peripheral devices operate concurrently to the software
  ⤳ Value in hardware registers can change anytime
- This change in contrast to the assumption of the compiler
  - Access to variables only takes place by the currently executed function
    ⤳ Variables are temporarily stored in registers

```
// 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 is not again loaded from memory. The compiler assumes the value in r24 to still be accurate.

17-MC-Peripherie: 2026-04-13

Details of Assembly Instructions

- knowledge how to program assembly instructions not necessary [1]
- however, semantics of assembly instructions necessary for *understanding concurrency*
- `lds`
  - load direct s from SRAM
  - load variable from memory to a register (for operations)
- `sbrc`
  - skip if bit in register cleared
  - test condition, depending on condition's result, skip following instruction
- `rjmp`
  - relative jump
- `ret`
  - return address is popped from the stack
  - return from a function call

Der volatile-Typmodifizierer

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

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

        // button0 pressed
        ...
    }

    if (! (PIND & 0x4)) {

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

```
// Resulting assembly code


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

...
L2:
    ret
```

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

The `volatile` Type Modifier

- **Solution:** declare variable as `volatile` (*"transient, changeable"*)
  - Compiler minimizes the time, the variable is held in registers
    - ⇝ value is read *immediately before* use
    - ⇝ value is written *immediately after* modification

```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 is declared as `volatile`. It is therefore loaded from memory before each test.

Der volatile-Typmodifizierer (Forts.)

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

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

}
```

volatile!

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

Achtung: volatile ↦ **$$$**

Die Verwendung von volatile verursacht erhebliche Kosten

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

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

The volatile Type Modifier (Forts.)

- volatile semantics prevent many code optimizations;
  in particular the removal of apparently unnecessary code
- volatile can be used to implement active waiting:

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

}
```

volatile!

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

Attention: volatile ↦ **$$$**

The use of volatile causes considerable runtime penalties

- Values cannot be stored in registers any longer
- Most code optimizations cannot be performed any longer

Rule: Use volatile only in **justified scenarios**

17-MC-Peripherie: 2026-04-13

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 $\leadsto$ 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)

Peripheral Devices: Ports

- **Port** := group of (usually 8) digital inputs/outputs
  - Digital output: bit value $\mapsto$ voltage level at µC pin
  - Digital input: voltage level at µC pin $\mapsto$ bit value
  - External interrupt: (on voltage change) voltage level at µC pin $\mapsto$ bit value $\leadsto$ processor executes interrupt program
- This function is usually configurable per pin
  - Input
  - Output
  - External interrupt (only for some inputs)
  - Alternative functions (pin used by another device)

Beispiel ATmega328PB: Port/Pin-Belegung

Power
Ground
Programming/debug
Digital
Analog
Crystal/CLK

32 PD2 (PTCXY/INT0/OC3B/OC4B)
31 PD1 (PTCXY/OC4A/TXD0)
30 PD0 (PTCXY/OC3A/RXD0)
29 PC6 ($\overline{\text{RESET}}$)
28 PC5 (ADC5/PTCY/SCL0)
27 PC4 (ADC4/PTCY/SDA0)
26 PC3 (ADC3/PTCY)
25 PC2 (ADC2/PTCY)

1 (OC2B/INT1/PTCXY) PD3
2 (XCK0/T0/PTCXY) PD4
3 (SDA1/ICP4/ACO/PTCXY) PE0
4 VCC
5 GND
6 (SCL1/T4/PTCXY) PE1
7 (XTAL1/TOSC1) PB6
8 (XTAL2/TOSC2) PB7

24 PC1 (ADC1/PTCY/SCK1)
23 PC0 (ADC0/PTCY/MISO1)
22 PE3 (ADC7/PTCY/T3/MOSI1)
21 GND
20 AREF
19 PE2 (ADC6/PTCY/ICP3/$\overline{\text{SS1}}$)
18 AVCC
17 PB5 (PTCXY/XCK0/SCK0)

9 (OC0B/T1/PTCXY) PD5
10 (OC0A/PTCXY/AIN0) PD6
11 (PTCXY/AIN1) PD7
12 (ICP1/CLKO/PTCXY) PB0
13 (OC1A/PTCXY) PB1
14 ($\overline{\text{SS0}}$/OC1B/PTCXY) PB2
15 (MOSI0/TXD1/OC2A/PTCXY) PB3
16 (MISO0/RXD1/PTCXY) PB4

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.

Example ATmega328PB: Port/Pin Assignment

Power
Ground
Programming/debug
Digital
Analog
Crystal/CLK

Pin	Assignment
1	(OC2B/INT1/PTCXY) PD3
2	(XCK0/T0/PTCXY) PD4
3	(SDA1/ICP4/ACO/PTCXY) PE0
4	VCC
5	GND
6	(SCL1/T4/PTCXY) PE1
7	(XTAL1/TOSC1) PB6
8	(XTAL2/TOSC2) PB7
9	(OC0B/T1/PTCXY) PD5
10	(OC0A/PTCXY/AIN0) PD6
11	(PTCXY/AIN1) PD7
12	(ICP1/CLKO/PTCXY) PB0
13	(OC1A/PTCXY) PB1
14	(SS0/OC1B/PTCXY) PB2
15	(MOSI0/TXD1/OC2A/PTCXY) PB3
16	(MISO0/RXD1/PTCXY) PB4
17	PB5 (PTCXY/XCK0/SCK0)
18	AVCC
19	PE2 (ADC6/PTCY/ICP3/SS1)
20	AREF
21	GND
22	PE3 (ADC7/PTCY/T3/MOSI1)
23	PC0 (ADC0/PTCY/MISO1)
24	PC1 (ADC1/PTCY/SCK1)
25	PC2 (ADC2/PTCY)
26	PC3 (ADC3/PTCY)
27	PC4 (ADC4/PTCY/SDA0)
28	PC5 (ADC5/PTCY/SCL0)
29	PC6 (RESET)
30	PD0 (PTCXY/OC3A/RXD0)
31	PD1 (PTCXY/OC4A/TXD0)
32	PD2 (PTCXY/INT0/OC3B/OC4B)

For reasons of cost efficiency nearly every pin is assigned twice. The configuration of the respective functionality takes place in software.

Z. B. pins 23–24 are configured as ADCs at the SPiCBoard to connect potentiometer and photo sensor.

Those pins are therefore not available for PORTC.

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 Handling

- An interrupt (↯) occurs when a peripheral device signals ↪ 17–3
  - a level change at a port pin *low* to *high*
  - the expiration of a *timer*
  - the completion of an A/D conversion (new value available)
  - ...
- How is the program notified about the (concurrent) event?
- Two alternative procedures
  - **Polling:** The program regularly checks a state and calls a handler function if necessary.
  - **Interrupt:** Device "notifies" the processor; subsequently, the processor branches into a handler function.

Interrupt ↦ Funktionsaufruf „von außen“

```
if( ⚡ )
  handler();
```

main()

handler()

isr()

handler()

ret

iret

call: explizite Aktivierung durch Funktionsaufruf

interrupt: implizite Aktivierung durch Hardwaresignal

isr: **i**nterrupt **s**ervice **r**outine

t_1 t_2 t_3 t_4 t_5

⚡ (z. B. Pegelwechsel an Pin)

⚡ (z. B. *Timer* abgelaufen)

Interrupt ↦ Function Call "from Outside"

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

Polling vs. Interrupts – (Dis-)Advantages

- **Polling** (↦ "polling-based system")
  - Processing of events synchronously to the program flow
    - – Detection of events scattered everywhere (missing separation of concerns)
    - – Wasting processing resources (if usable for other things)
    - – High polling frequency ⇝ high processor load ⇝ high energy consumption
    - + Implicit consistency in data by sequential program flow
    - + Program behavior predictable

- **Interrupts** (↦ "event-triggered system")
  - Processing of events asynchronous to the program flow
    - + Event handlers can be easily separated in the source code
    - + Processor is only triggered when an event occurs
    - – Higher complexity by concurrency ⇝ synchronization required
    - – Program behavior unpredictable

> Both methods provide specific (dis-)advantages
> ⇝ Which one to choose depends on concrete scenario

Interrupt ↦ unvorhersagbarer Aufruf „von außen“

```
if( ⚡ )
  handler();
```

main()

handler()

isr()

handler()

ret

buf[...]

iret

t_1 t_2 t_3 t_4 t_5

⚡ (z. B. Pegelwechsel an Pin)

⚡ (z. B. *Timer* abgelaufen)

Interrupt ↦ Unpredictable Call "from Outside"

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

Disabling Interrupts

- Notification about new interrupts can be disabled by software
  - Used for synchronization with ISRs
  - Single ISR: Bit in device-specific control register
  - All ISRs: Bit (**IE**, *Interrupt Enable*) in a status register of CPU
- Pending IRQs are (usually) buffered
  - At most one interrupt (per source)!
  - During longer disabled time spans, IRQs can be missed!

IRQ ↦ *Interrupt ReQuest*

- The **IE** bit is affected by:
  - processor instructions: `cli`: **IE**←0 (*clear interrupt*, IRQs disabled)
    `sei`: **IE**←1 (*set interrupt*, IRQs enabled)
  - after a RESET: **IE**=0 ⇝ IRQs are always disabled at the begin of the main program
  - when entering an ISR: **IE**=0 ⇝ IRQs are disabled during handling of other interrupts

18-IRQ-Konzept: 2026-04-14

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

Interrupt Blocking: Example

t_1	At the begin of `main()`, all IRQs are disabled (`IE`=0)
t_2, t_3	With `sei()` / `cli()` IRQs can be enabled (`IE`=1) / disabled
t_4	ϟ but `IE`=0 ⇝ handling is blocked, IRQ is buffered
t_5	`main()` unblocks IRQs (`IE`=1) ⇝ buffered IRQ is executed
$t_5 - t_6$	During handling of the ISR, all IRQs are blocked again (`IE`=0)
t_6	Interrupted `main()` is resumed

Ablauf eines Interrupts – Überblick

❶ Gerät signalisiert Interrupt

- Anwendungsprogramm wird „unmittelbar“ (vor dem nächsten Maschinenbefehl mit `IE`=1) unterbrochen

❷ Die Zustellung weiterer Interrupts wird gesperrt (`IE`=0)

- Zwischenzeitlich auflaufende Interrupts werden gepuffert (maximal einer pro Quelle!)

❸ Registerinhalte werden gesichert (z. B. im Stapel)

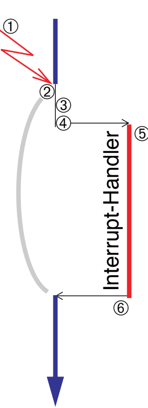
- PC und Statusregister automatisch von der Hardware
- Vielzweckregister üblicherweise manuell in der ISR

❹ Aufzurufende ISR (Interrupt-Handler) wird ermittelt

❺ ISR wird ausgeführt

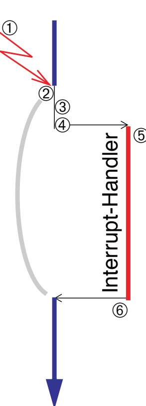
❻ ISR terminiert mit einem „return from interrupt“-Befehl

- Registerinhalte werden restauriert
- Zustellung von Interrupts wird freigegeben (`IE`=1)
- Das Anwendungsprogramm wird fortgesetzt



Procedure of an Interrupt – Overview

1. Device signals an interrupt
   - Current program is "immediately" interrupted (prior to the next machine instruction, with `IE`=1)
2. Notification of further interrupts is blocked (`IE`=0)
   - Interrupts that occur during this time are buffered (at most once per source!)
3. Content of registers is stored (z. B. on the stack)
   - PC and status registers automatically by the hardware
   - Multi-purpose registers usually manually in the ISR
4. Determination of to be called ISR (interrupt handler)
5. ISR is executed
6. ISR terminates with "return from interrupt" instruction
   - Content of registers is restored
   - Notification of interrupts again unblocked/enabled (`IE`=1)
   - Program is resumed



Ablauf eines Interrupts – Details

IRQs enabled bit

Zero bit

SR: ... | Z (0) | IE (1)

RESET, Vcc, INT

μC: SR, PC, SR`, PC`, R1, R1`, IP (0)

IRQ pending bit

Bus

Programmspeicher

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

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

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

RESET → PC = 0x0000

SR.IE && IP — true → w = **icall**; false → w = *PC++

decode(w)

execute(w)

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`

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

Ablauf eines Interrupts – Details

IRQs enabled bit

Zero bit

SR

...	Z	IE
	0	1

RESET

Vcc

INT

SR PC

SR` PC`

R1

R1`

IP 1

μC

Bus

Programmspeicher

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

RESET → PC = 0x0000

SR.IE && IP — **true** → w = **icall**; **false** → w = *PC++

decode(w)

execute(w)

➊ Gerät signalisiert Interrupt (aktueller Befehl wird noch fertiggestellt)

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

w: ret
PC = PC`

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

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

Ablauf eines Interrupts – Details

IRQs enabled bit

Zero bit

SR

...	Z	IE
	0	1

RESET

Vcc

INT

SR PC

SR` PC`

R1

R1`

μC

IP 1

Bus

Programmspeicher

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

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

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

RESET

PC = 0x0000

SR.IE && IP

true

false

w = *PC++

w = **icall**

decode(w)

execute(w)

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

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

w: ret
PC = PC`

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

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

Ablauf eines Interrupts – Details

RESET → PC = 0x0000

IRQs enabled bit

Zero bit

SR	...	Z	IE
		0	0

RESET, Vcc, INT

SR, PC, SR`, PC`, R1, R1`, IP 0, µC

Bus

Programmspeicher

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

SR.IE && IP — **true** → w = **icall**; **false** → w = *PC++

decode(w)

execute(w)

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

IRQs enabled bit
Zero bit
SR | ... | Z | IE
0 0
RESET
Vcc
INT
SR PC
SR` PC`
R1
R1`
µC
IP 0
Bus

Programmspeicher

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

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

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

RESET → PC = 0x0000
SR.IE && IP
true → w = **icall**
false → w = *PC++
decode(w)
execute(w)

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

w: ret
PC = PC`

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

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

❹ Aufzurufende ISR wird ermittelt

Ablauf eines Interrupts – Details

IRQs enabled bit

Zero bit

SR | ... | Z | IE |
Z = 1, IE = 0

RESET

Vcc

INT

SR | PC
SR` | PC`
R1 = 0
R1`
IP = 0

μC

Bus

Programmspeicher

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

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

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

RESET → PC = 0x0000

SR.IE && IP

true → w = **icall**

false → w = *PC++

decode(w)

execute(w)

❺ ISR wird ausgeführt

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

IRQs enabled bit
Zero bit
SR | ... | Z | IE
Z = 0, IE = 1 ❻

RESET, Vcc, INT
SR, SR`, PC, PC`, R1, R1`, IP (0) — μC ❻

Bus

Programmspeicher

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

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

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

RESET → PC = 0x0000
SR.IE && IP — true → w = **icall**; false → w = *PC++
decode(w)
execute(w)

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

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

w: ret
PC = PC`

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

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

Procedure of an Interrupt – Details

IRQs enabled bit

Zero bit

SR | ... | Z 0 | IE 1

RESET, Vcc, INT

SR, PC, SR`, PC`, R1, R1`, IP 0, μC

IRQ pending bit

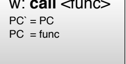
Bus

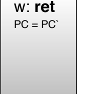
Program memory

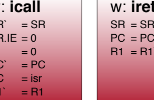
```
main: ldi R1, 48     0x0000
      dec R1         0x0002
      beq L1         0x0004
      call f         0x0006
      sub R1, 58     0x0008
L1:   ...            0x000A

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

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

RESET → PC = 0x0000 → SR.IE && IP ? true: w = **icall** / false: w = *PC++ → decode(w) → execute(w)

- Here as an extension of the simplified pseudo processor ↪ 16–3
  - Only one source for interrupts
  - All registers are saved by the hardware







Procedure of an Interrupt – Details

IRQs enabled bit
Zero bit
SR | ... | Z | IE
0 1
RESET
SR PC
SR` PC`
μC
R1
R1`
IP
1
Vcc
❶
INT
Bus

Program memory

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

RESET
PC = 0x0000
SR.IE && IP
true
false
w = *PC++
w = **icall**
decode(w)
execute(w)

❶ Device signals an interrupt (current instruction will be finalised)

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

w: ret
PC = PC`

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

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

Procedure of an Interrupt – Details

IRQs enabled bit

Zero bit

SR | ... | Z 0 | IE 1

RESET

Vcc

INT

SR | PC

SR` | PC`

R1

R1`

IP 1

μC

Bus

Program memory

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

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

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

RESET → PC = 0x0000

SR.IE && IP

true → w = **icall**

false → w = *PC++

decode(w)

execute(w)

(Before the next *instruction fetch* the interrupt state is checked)

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`

Procedure of an Interrupt – Details

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

❷ The delivery of further interrupts is stalled
❸ Register contents get saved

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

w: ret
PC = PC`

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

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

Procedure of an Interrupt – Details

IRQs enabled bit

Zero bit

SR | ... | Z 0 | IE 0

RESET

Vcc

INT

µC: SR, PC, SR`, PC`, R1, R1`, IP 0

Bus

Program memory

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

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

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

❹ ISR to be called is determined

RESET → PC = 0x0000

SR.IE && IP — true → w = **icall**; false → w = *PC++

decode(w)

execute(w)

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`

Procedure of an Interrupt – Details

❺ ISR gets executed

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`

Procedure of an Interrupt – Details

❻ ISR terminates with *iret*-instruction
- Register contents are restored
- Delivery of interrupts is reactivated
- Program is resumed

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

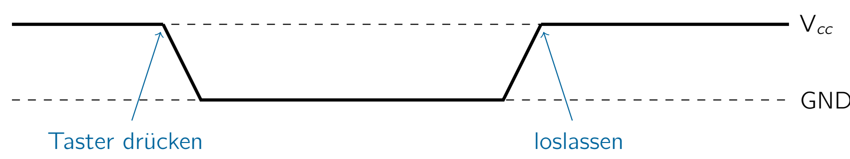
18 Unterbrechungen

19 Unterbrechungen – Beispiel

20 Unterbrechungen – Nebenläufigkeit

Pegel- und Flanken-gesteuerte Interrupts

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

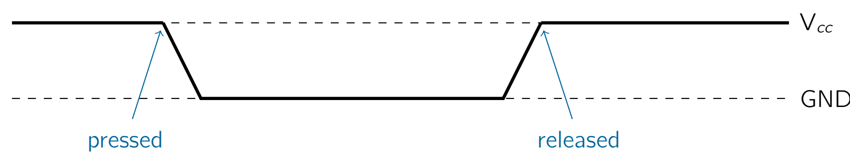


- Flankengesteuerter Interrupt
  - Interrupt wird durch den Pegelwechsel (Flanke) ausgelöst
  - Häufig ist konfigurierbar, welche Flanke (steigend/fallend/beide) einen Interrupt auslösen soll

- Pegelgesteuerter Interrupt
  - Interrupt wird immer wieder ausgelöst, so lange der Pegel anliegt

Level- and Edge-Triggered Interrupts

- Example: Signal of an idealized button (*active low*)



- Edge-triggered interrupt
  - interrupt is triggered by a change of voltage (edge)
  - configurable which edge (rising/falling/both) triggers an interrupt
- Level-triggered interrupt
  - interrupt is triggered as long as a specific voltage level is present

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

- Verschaltung SPiCboard
  (↪ 17–12 ↪ 2–10)
  - **INT0** ↦ **PD2** ↦ Button0 (hardwareseitig entprellt)
  - **INT1** ↦ **PD3** ↦ Button1

Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/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

Interrupt Control for AVR ATmega

- IRQ sources for the ATmega328PB (IRQ ↦ *Interrupt ReQuest*)
  - 45 IRQ sources [1, S. 78]
  - individual (de)activation
  - IRQ ⇝ jump to vector address
- Wiring SPiCboard
  (↪ 17–12 ↪ 2–10)
  - **`INT0`** ↦ **`PD2`** ↦ Button0 (debounced by hardware)
  - **`INT1`** ↦ **`PD3`** ↦ Button1

Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/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

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.

External Interrupts: Registers

- Control registers for `INT0` and `INT1`
  - **EIMSK** **External Interrupt Mask Register:** Determines whether the sources INT*i* IRQs trigger (bit INT*i*=1) or are deactivated (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:** Determines for external interrupts INT0 and INT1 the *cause* for activation (edge-/level-triggered) [1, S. 83]

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

Thereby, two *Interrupt-Sense-Control* bits (ISC*i*0 and ISC*i*1) control the exact trigger (table for INT1, INT0 has analogous table):

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

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

External Interrupts: Usage

- **Step 1**: Installation of interrupt-service routine
  - ISR in high level language ⇝ store and restore contents of registers
  - Support by `avrlibc`: Macro `ISR(SOURCE_vect)` (module `avr/interrupt.h`)

```c
#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
}
```

19-IRQ-Beispiel: 2026-04-13

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`)

External Interrupts: Usage (Forts.)

- **Step 2**: Configuration of interrupt control
  - Initialize control registers as required
  - Support by `avrlibc`: macros for bit indices (module `avr/interrupt.h` and `avr/io.h`)

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

- **Step 3**: Globally enable interrupts
  - After finishing the device initialization
  - Support by `avrlibc`: Instruction `sei()` (module `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

Green IT

```
#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.

External Interrupts: Usage (Forts.)

- **Step 4**: If there is nothing left to do, enter power-saving mode
  - The `sleep` instruction halts the CPU until an IRQ occurs
    - This state only has a comparably low power demand
  - Support by `avrlibc` (module `avr/sleep.h`):
    - `sleep_enable()` / `sleep_disable()`: sleep mode is activated/deactivated
    - `sleep_cpu()`: sleep mode entered

Green IT

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

Atmel recommends the use of `sleep_enable()` and `sleep_disable()` in this way to minimize the risk of an "accidental" activation of the sleep mode (z. B. by programming errors or switched bits in hardware).

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
    ⇝ IRQs können ein Programm an „beliebiger Stelle“ unterbrechen
  - Echt-parallele Abläufe (durch die Hardware)
    ⇝ andere CPU / Peripherie greift „jederzeit“ auf den Speicher zu
  - Quasi-parallele Abläufe (z. B. Fäden in einem Betriebssystem)
    ⇝ Betriebssystem kann „jederzeit“ den Prozessor entziehen

- **Problem:** Nebenläufige Zugriffe auf **gemeinsamen** Zustand

Concurrency

Definition: Concurrency

Two executions A and B of a program are considered to be concurrent ($A|B$), if for every single instruction a of A and b of B it is not determined, whether a or b is executed first (a, b or b, a).

- Concurrency is induced by
  - Interrupts
    $\leadsto$ IRQs can interrupt a program at an “arbitrary point”
  - Real-parallel sequences (by the hardware)
    $\leadsto$ other CPU / peripheral devices access the memory at “anytime”
  - Quasi-parallel sequences (z. B. threads in an operating system)
    $\leadsto$ OS can preempt tasks “anytime”
- **Problem:** Concurrent access to a **shared state**

Nebenläufigkeitsprobleme

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

```
static volatile uint16_t cars;

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

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

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

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

Problems with Concurrency

- Scenario
  - a light gate at the entrance of a parking lot counts cars
  - every 60 seconds, the value is transferred to security agency

```
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++;
}
```

- Where does the problem occur?
  - both `main()` as well as `ISR` **read and write** `cars`
    ⇝ potential *lost-update anomaly*
  - size of the variable `cars` **is larger than one register**
    ⇝ potential *read-write* anomaly

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
```

Problems with Concurrency (Forts.)

- Where are the problems here?
  - lost-update: both **main()** as well as **ISR** read and write **cars**
  - read-write: size of the variable **cars** is larger than one register
- problem only becomes obvious when looking at the assembly level

```
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
```

20-IRQ-Nebenlaeufigkeit: 2026-04-14

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**

Concurrency Problems: *Lost-Update* Anomaly

```
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
```

- Let `cars`=5 and let the IRQ (⚡) occur at **this point**
  - `main` already read the value of `cars` (5) from the register (register $\mapsto$ local variable)
  - `INT2_vect` is executed
    - registers are saved
    - `cars` is incremented $\leadsto$ `cars`=6
    - registers are restored
  - `main` passes the **old value** of `cars` (5) to `send`
  - `main` sets `cars` to zero $\leadsto$ **1 car is "lost"**

20-IRQ-Nebenlaeufigkeit: 2026-04-14

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**

```
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
```

- Let `cars`=255 and let the IRQ (↯) occur at **this point**
  - `main` has already transmitted `cars`=255 with `send`
  - `main` has already set the high byte of `cars` to zero
    ⇝ `cars`=255, `cars.lo`=255, `cars.hi`=0
  - `INT2_vect` is executed
    ⇝ `cars` is read and incremented, **overflow in the high byte**
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1
  - `main` sets the low byte of `cars` to zero
    ⇝ `cars`=256, `cars.lo`=0, `cars.hi`=1
    ⇝ During the next `send`, `main` will transmit **too many cars (255 cars)**

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 ⇝ 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?

Interrupt Locks: Avoid Data-Flow Anomalies

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

critical region

- Where exactly is the **critical region**?
  - Reading of `cars` and setting it to zero have to be executed atomically
  - This can be forced by using **interrupt locks**
    - `ISR` interrupts `main`, never the other way round
      $\leadsto$ asymmetric synchronization (also unilateral synchronization)
  - Attention: keep regions with blocked interrupts **as short as possible**
    - How long does the function `send` take?
    - Can `send` be excluded from the critical region?

Nebenläufigkeitsprobleme (Forts.)

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

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

```
static volatile int8_t event;

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

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

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

Problems with Concurrency (Forts.)

- Scenario, part 2 (function `waitsec()`)
  - a light gate at the entrance of a parking lot should count cars
  - every 60 seconds, the value is transferred to security agency

```
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;
}
```

- Where exactly does the problem occur?
  - **Test, whether sth. is to be done**, followed by **sleeping until there is sth. to do**
    ⇝ Potential lost-wakeup anomaly

Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

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

```c
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**

Concurrency Problems: *Lost-Wakeup*-Anomaly

```
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;
}
```

- Suppose, at **this point** a timer-IRQ (↯) occurs
  - `waitsec` already determined that `event` is not set
  - `ISR` is executed ⇝ `event` **is set to 1**
  - Even though `event` is set to 1, the sleep state is entered
    ⇝ If no further IRQ occurs, **sleeping forever**

20-IRQ-Nebenlaeufigkeit: 2026-04-14

Lost-Wakeup: Dornröschenschlaf verhindern

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

```
static volatile int8_t event;

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

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

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

Lost-Wakeup: Prevention of Deep Sleep

```
 1  void waitsec(uint8_t sec) {
 2    ...                 // setup timer
 3    sleep_enable();
 4    event = 0;
 5    cli();
 6    while (! event) {
 7      sei();              critical region
 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;
}
```

- Where exactly is the **critical region** located?
  - evaluation of the condition and entry of the sleeping state (Can *this* be solved by interrupt blocking?)
  - problem: the IRQs have to be unblocked prior to `sleep_cpu()`!
  - works thanks to specific **hardware support**:
    $\leadsto$ sequence `sei`, `sleep` is executed as an **atomic instruction**

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**

  - Interrupthandler und Zugriffsfunktionen auf gemeinsamen Zustand (`static` Variablen!) in eigenem Modul kapseln.

Summary

- Handling of interrupts is asynchronous to the program flow
  - unexpected ⇝ current state has to be saved in the interrupt handler
  - source of concurrency ⇝ synchronization required
- Measures for synchronization
  - shared variables shall (always) be declared as `volatile`
  - blocking arrival of interrupts: `cli`, `sei` (when working with non-atomic accesses that translate to more than one machine instruction)
  - Locking for longer times leads to the loss of IRQs!
- Concurrency induced by interrupts is **enormous source for errors**
  - *lost-update* and *lost-wakeup* problems
  - indeterministic ⇝ cannot efficiently be tested for
- Important for handling complexity: **modularization** ↪ 12–7

  - Interrupt handler and functions accessing a shared state (`static` variables!) should be encapsulated in their own module

Systemnahe Programmierung in C

Teil D Betriebssystemabstraktionen

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Sommersemester 2026

http://sys.cs.fau.de/lehre/ss26

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;
}
```

x
h a l l o \0
n=0 n=4 n=5
s
s++
x
h a l l o \0
s p
p++

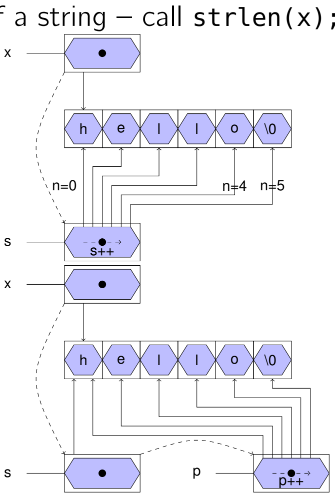
21-Misc-Zeiger: 2026-04-13

Pointers, Arrays, and Strings

- Strings are arrays of single characters (`char`) that are internally terminated by the `'\0'`-character
- Example: Determining the length of a string – call `strlen(x);`

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

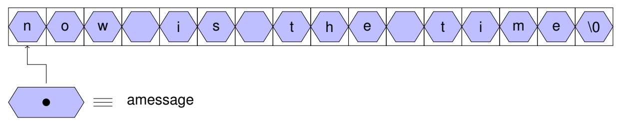
```
/* 2. Version */
int strlen(const char *s)
{
    const char *p = s;
    while (*p != '\0') {
        p++;
    }
    return p - s;
}
```



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";
```



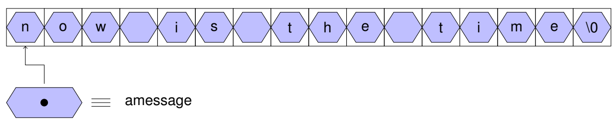
- es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
- `amessage` ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
- der Inhalt des Speicherbereichs kann aber modifiziert werden

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

Pointer, Arrays and Strings (Forts.)

- If a string is used for the initialization of a **char**-array, the identifier of the array is a constant pointer to the start of the string

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



  - a memory space of size 16 bytes is allocated and the characters are copied to this area
  - **amessage** is a *constant pointer* to the start of the memory area, this pointer cannot be modified
  - however, the *contents* of the memory area can be modified

```
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); /* prints "ello world" */
```

- 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

Pointer, Arrays and Strings (Forts.)

- If a string is used for the initialization of a `char` pointer, the pointer is a variable that is initialized with the starting address of the string

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



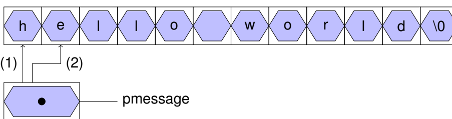
```
pmessage++;    /*(2)*/
printf("%s\n", pmessage); /* prints "ello world" */
```

- the string itself is placed in memory as a constant value (string literal) by the compiler
- the memory space for a pointer is reserved (z. B. 4 byte) and then initialized with the address of the string

21-Misc-Zeiger: 2026-04-13

Zeiger, Felder und Zeichenketten (4)

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




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

- `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)

Pointer, Arrays and Strings (4)

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



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

- `pmessage` is a variable pointer that is initialized with a certain address, but can be modified (`pmessage++;`)
- it is not allowed to modify the memory area of "`hello world`"
  - the compiler detects this use of the keyword `const` and prevents write access via the pointer
  - some compilers place such strings in the write-protected area of the memory (⇒ memory-protection violation when the content is accessed and the pointer has not been declared as a `const` pointer)

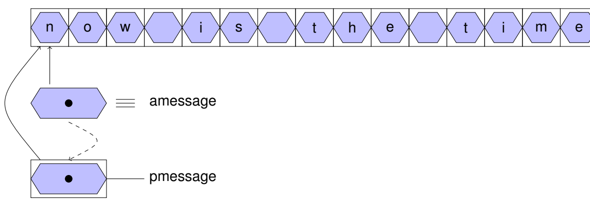
21-Misc-Zeiger: 2026-04-13

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



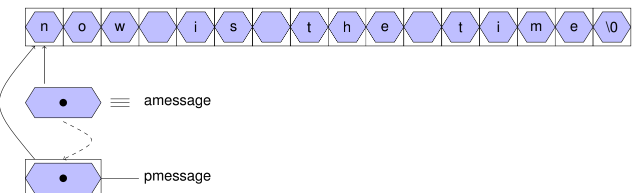
- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

Pointer, Arrays and Strings (5)

- Assigning a `char` pointer or string to another `char` pointer does copy the string!

```
pmessage = amessage;
```

The pointer **`pmessage`** only is assigned the address of the string **`"now is the time"`**.



- When passing a string as an actual parameter to a function, the function only receives a copy of the pointer to the string

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++) {
    }
}
```

Pointer, Arrays and Strings (6)

- To assign a whole string to another **char** array, the string has to be copied: Function **strcpy** from the standard C library
- Examples for implementation:

```
/* 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++;
    }
}
```

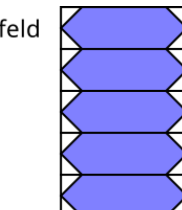
```
/* 3. Version */
void strcpy(char *s, char *t) {
    while (*s++ = *t++) {
    }
}
```

Felder von Zeigern

Auch von Zeigern können Felder gebildet werden

- Deklaration

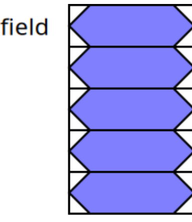
```
int *pfeld[5];
int i = 1;
int j;
```



21-Misc-Zeiger: 2026-04-13

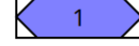
Pointer Arrays

Arrays of pointers can also be created



- Declaration

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



Felder von Zeigern (Forts.)

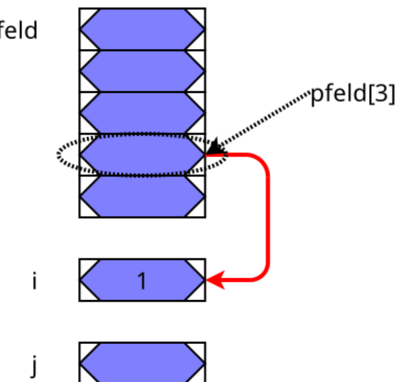
Auch von Zeigern können Felder gebildet werden

- Deklaration

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

- Zugriff auf einen Zeiger des Feldes

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



Pointer Arrays (Forts.)

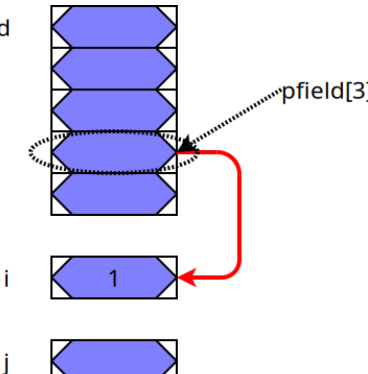
Arrays of pointers can be created also

- Declaration

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

- Access to a pointer of the array

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



Felder von Zeigern (Forts.)

Auch von Zeigern können Felder gebildet werden

- Deklaration

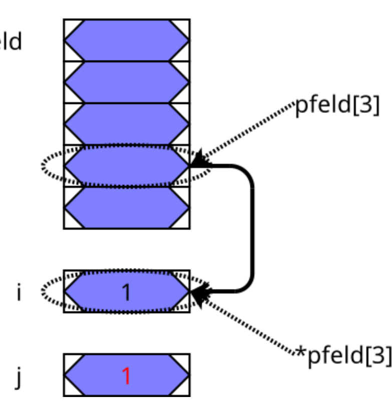
```
int *pfeld[5];
int i = 1;
int j;
```

- Zugriff auf einen Zeiger des Feldes

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

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

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



Pointer Arrays (Forts.)

Arrays of pointers can be created also

- Declaration

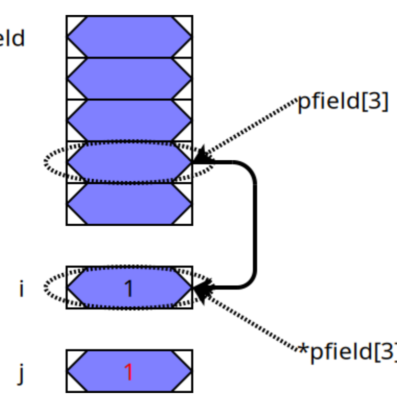
```
int *pfield[5];
int i = 1;
int j;
```

- Access to a pointer of the array

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

- Access to the object that the pointer of the array points to

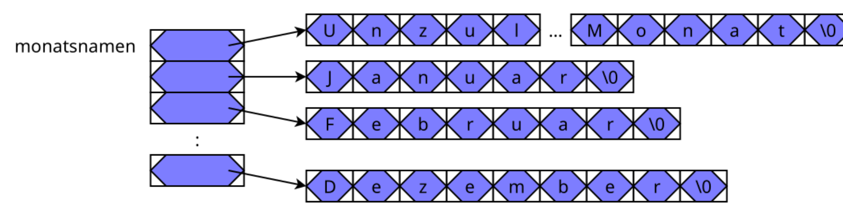
```
j = *pfield[3];
```



Felder von Zeigern (Forts.)

Beispiel: Definition und Initialisierung eines Zeigerfeldes:

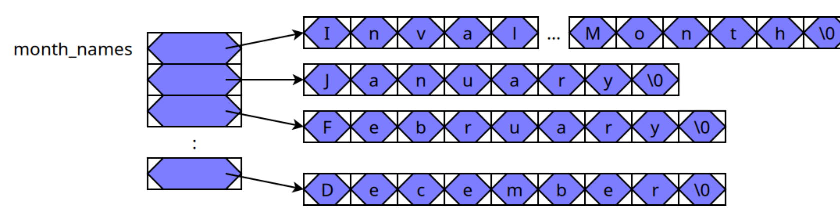
```
const char *
month_name(int n)
{
    static const char *monatsname[] = {
        "Unzulaessiger Monat",
        "Januar",
        ...
        "Dezember"
    };

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



Pointer Arrays (Forts.)

Example: Definition and initialization of a pointer array

```
const char *
month_name(int n)
{
    static const char *name_of_month[] = {
        "invalid month",
        "January",
        ...
        "December"
    };

    return (n < 1 || 12 < n) ? name_of_month[0] : name_of_month[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]`**)

Arguments from the Command Line

- Usually, when a program is called, arguments are passed to the program
- The access to these arguments is provided in the function **main()** by two parameters (both variants are equivalent):

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

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

- The parameter **argc** contains the number of arguments that were given when calling the program
- The parameter **argv** is a field of pointers to the respective arguments (strings)
- The name of the program is always passed as the first argument (**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)
...
```

argv[1][0]
argc 5
argv
g c c \0
- W a l l \0
- o \0
t e s t \0
t e s t . c \0
NULL
argv[1]
argv[3][1]

Arguments from the Command Line

- Command:

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

- C-file:

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

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

argc 5
argv

argv[1]
argv[1][0]
argv[3][1]

g	c	c	\0

-	W	a	l	l	\0

-	o	\0

t	e	s	t	\0

t	e	s	t	.	c	\0

NULL

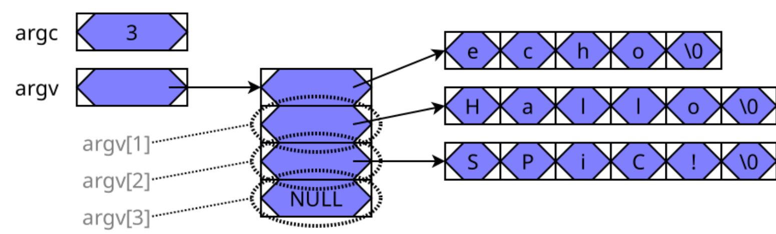
21-Misc-Zeiger: 2026-04-13

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;
}
```

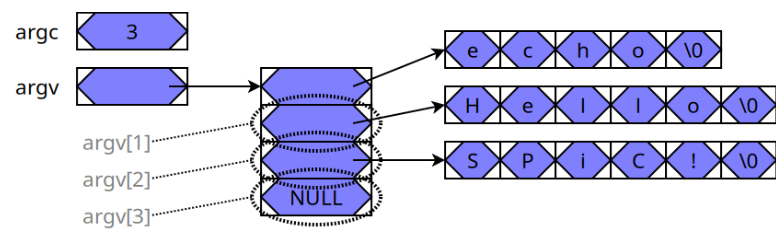


Arguments – Example

Example: `echo` program

```
~> echo Hello SLP!
Hello SLP!
~>
```

```c
#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;
```

Composite Data Types / Structures

- Combination of multiple values to one structure
- Declaration of structure

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

- Definition of a variable of type `struct`

```
struct person p1;
```

- Access to an element of the structure

```
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

Pointers to Structures

- Concept in analogy to “pointer to variable”
  - Address of a structure determined with the & (address-of) operator
- Example

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

- Especially useful when building linked structures (lists, trees, ...)
  - a structure can contain addresses to further structures of the same (and other) types

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;
```

Pointers to Structures (Forts.)

- Access to components of the structure via the pointer
- Known approach
  - "∗"-operator yields structure itself
  - "."-operator yields an element of the structure
  - **However:** Keep in mind the order of the operators!

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

- Syntactically nicer:
  - "->"-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;
```

Nested/Linked Structures

- Structures inside of structures are allowed – however:
  - the structure's size has to be statically determined by the compiler
    ⇒ structure cannot contain itself
  - the size of a pointer is always known
    ⇒ structure can contain a pointer to the same structure
  - Examples:

Linked list:

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

struct list *head;
```

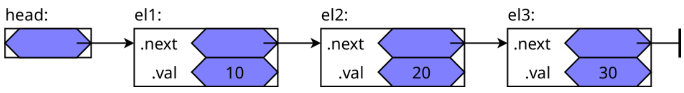
Tree:

```
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;
```



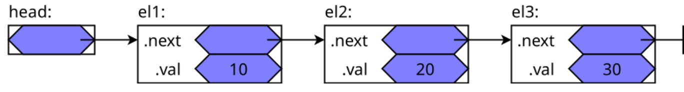
- Laufen über eine verkettete Liste

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

Linked Lists

- Multiple structures of the same type can be linked via pointers

```
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;
```



- Iterating over a linked list

```
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

Input/Output

- I/O functionality is *not part* of the programming language
- Realized by “normal” (library) functions
  - part of the *standard library*
  - simple programming interface
  - efficient
  - portable
  - close to the operation system
- Features
  - open/close files
  - read/write single characters, lines, or arbitrary blocks of data
  - formatted input/output

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 Input/Output

Every C program has *three I/O channels*, assigned automatically upon starting:

`stdin`: standard input

- usually connected to the keyboard
- "end of file" (**`EOF`**) is signaled by input of **`CTRL-D`** at the begin of a line
- this can be redirected to a file upon calling the program

```
~> prog < inputfile
```

`stdout`: standard output

- usually connected to the display (or the window from which the program was started)
- this can be redirected to a file upon calling the program

```
~> prog > outputfile
```

`stderr`: output channel for error messages

- usually also connected to the display

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!

Standard Input/Output (Forts.)

- Pipes
  - The standard *output* of a program can be connected with the standard *input* of another program:

    ```
    ~> prog1 | prog2
    ```

    The redirection of the standard I/O channels is invisible for the called program.
- Automatic buffering
  - Input from the keyboard is usually buffered line-by-line by the operating system and only passed to the program when a **NEWLINE** symbol (**'\n'**) occurs!
  - Output for the display is usually buffered line-by-line by the program and only written to the display when a **NEWLINE** symbol occurs!

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)

Opening and Closing Files

- Besides the standard I/O channels, a program can open further I/O channels
  - access to files
- Opening an I/O channel
  - function **`fopen`** (file open)
- Closing an I/O channel
  - function **`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`

22-Misc-IO: 2026-04-13

Opening and Closing Files (Forts.)

- Interface **fopen**

```
#include <stdio.h>

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

name: path name of the file to be opened
mode: mode how the file has to be opened

- "r": read
- "w": write
- "a": write at the end of the file (append)
- "rw": read and write

- opens file **name**
- result of **fopen**: pointer to a data type **FILE** that describes a file channel; on error **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

Opening and Closing Files (Forts.)

- Interface `fclose`

```
#include <stdio.h>

int fclose(FILE *fp);
```

  - closes I/O channel `fp`
  - result is either `0` (no errors) or `EOF` if an error occured

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;
}
```

Opening and Closing Files – Example

```
#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

Reading and Writing single Characters

- Reading a single character
  - from standard input

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

  - from a file

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

  - read the next character
  - return the character as **int** value
  - return **EOF** at the end of file or when CRTL-D is pressed
- Writing a single character
  - to the standard output

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

  - into a file

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

  - write the character **c**
  - return **EOF** in case of an error

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;
}
```

Reading and Writing single Characters – Example

Copy program:

```
#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

Reading and Writing Line-by-Line

- Reading one line

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

  - reads characters from the file channel `fp` into the `char` array `buf` until either `bufsize-1` characters have been read or `'\n'` or `EOF`
  - `s` (returned string) is terminated by `'\0'` (`'\n'` is not removed)
  - returns `NULL` on `EOF` or when an error occurs
  - for `fp`, `stdin` can be used to read from the standard input

- Writing one line

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

  - writes the characters from the array `s` to the file channel `fp`
  - returns `EOF` when an error occurs
  - for `fp` `stdout` or `stderr` can be used

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)

Formatted Output

- Interface

```
#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, ...);
```

- The parameters given as `...` are outputted according to the specifications in the `format` string
  - when using `printf` to the standard output channel
  - when using `fprintf` to the file channel `fp` (`fp` can be substituted by `stdout` or `stderr`)
  - `sprintf` writes the output into the `char`-array `buf` (but does not consider the length of the array ⇒ buffer overflow possible!)
  - `snprintf` works analogously, but writing at most `bufsize` characters (`bufsize` therefore should not be greater than the size of the array!)

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`, ...)

Formatted Output (Forts.)

- Characters in the `format` string have different meanings
  - normal (printable) characters:
    are copied to the output
  - escape characters:
    z. B. `\n` or `\t` are substituted by the corresponding characters in the output (here: new line or tabulator)
  - format instructions:
    start with `%` character and describe, how the corresponding parameter in the list after the `format` string has to be interpreted
- For more specific information refer to the 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

...: ...

Formatted Output (Forts.)

- Format-instructions

%d, %i: output **`int`** parameter as a decimal number
%ld, %li: comparable behavior for **`long int`**

%f: output **`float`** parameter as floating point number
(z. B. **`13.153534`**)
%lf: comparable behavior for **`double`**

%e: output **`float`** parameter as a floating point number with powers of 10 (z. B. **`2.71456e+02`**)
%le: comparable behavior for **`double`**

%c: output **`char`** parameter as single character
%s: output **`char`** array until **`'\0'`** is reached

%%: output a **`%`** character

...: ...

Formatierte Ausgabe – Beispiel

```c
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
~>
```

Formatted Output – Example

```c
int day = 25;
int month = 6;
int year = 2009;
char *name = "Michael Jackson";
printf("On %d/%d/%d\n%s died.\n",
       month, day, year, name);

printf("\n");

double pi = asin(1.0) * 2.0;
double e = exp(1.0);
fprintf(stdout,
        "Important value are:\n");
fprintf(stdout,
        "pi=%lf and e=%lf\n", pi, e);
```

```
~> ./test
On 6/25/2009
Michael Jackson died.

Important value are:
pi=3.141593 and 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!

Formatted Input

- Interface

```
#include <stdio.h>

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

Format string analogously works to the formatted output.
For more specific information, read the manuals (`man 3 scanf`, ...).

But: since values have to be read, *pointers* to the variables have to be passed to the functions (mimic call-by-reference semantics with C's call-by-value approach)!

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
~>
```

Formatted Input – Example

```c
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!

Error Handling

- Nearly every system call or library call can fail.

  $\Rightarrow$ **Error handling is inevitable!**

- Goal:

  No program should crash without an error message!

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.

Error Handling

- Approach
  - Always check the return value of system/library calls
  - In the case of an error (usually indicated by -1 or `NULL`): error code is stored in the global variable `errno`
- Error message can be printed to `stderr` with the function `perror`:

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

- Check temporary results for plausibility

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

If `condition` is not “true”, the program is aborted with an error message.

23-Misc-Fehler: 2026-04-13

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
  - ...

Error Handling (Forts.)

- Error handling has to be adapted to the context:
  - Errors due to user errors
    (z. B. user inputs wrong file name or wrong URL)
    - notify the user about the error
    - allow a new input
    - **repeat** the failed part of the program
  - Errors due to lack of resources
    (z. B. memory or disk is full)
    - notify the user about the error
    - allow the user to "clean up"
    - **repeat** the failed part of the program
  - Programming error
    (z. B. computed result is wrong)
    - output an error message
    - **abort** program
  - ...

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
~>
```

Error Handling – Example

```
...

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

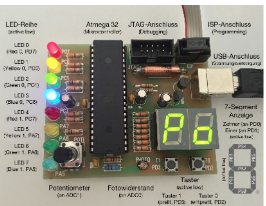
Definition “Operating System”

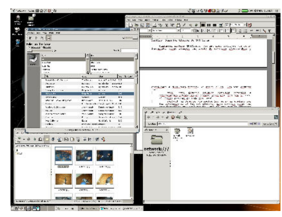
- DIN 44300
  - “... the programs of a digital computing system which, together with the properties of the computing system, form the basis of the possible operating modes of the digital computing system and that particularly control and monitor the execution of programs”
- Andrew S. Tannenbaum
  - “... a *software layer* ..., that manages all parts of a system and provides the user with an interface or virtual machine that is easier to understand and program [than the bare hardware].”
- Conclusion:
  - Software for managing and *virtualizing* the hardware components (i.e., resources)
  - Program for *controlling & monitoring* other programs

Überblick

Bisher:

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





Quelle: www.wikipedia.org

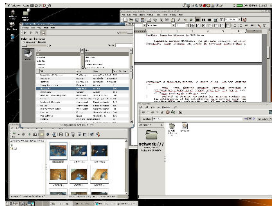
Jetzt:

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

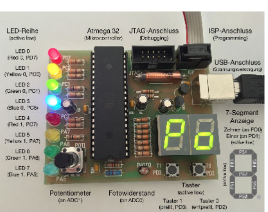
Overview

Previous lectures:

- **One** program that
- controls its environment,
- **alone**,
- started **during boot**,
- with **hardware accesses**.



Source: www.wikipedia.org



Now:

- **Multiple** programs that
- control their environment,
- **concurrently**,
- started/stopped **dynamically**,
- via **defined I/O functions**.

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.

Multitasking

- If more than one application exists on a system ("multitasking"), the applications have to coordinate
  - who and when can access the/one *CPU*,
  - who can use which *memory* areas,
  - who can use which part of the *disk/hard drive*,
  - who is allowed to display which part on the *screen*,
  - ...

Since no application can decide on its own, for example, which areas in the memory are still unused, **shared methods and state variables** are required.

- It has to be ensured that
  - all applications meet the agreements (even those, that are (un)intentionally programmed erroneously!)
- **Hardware extensions** have to restrict access to unauthorized memory areas or I/O devices.

Betriebssystem-Kern

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

Operating System Kernel

- **Shared methods and state variables**
  - **Operating-system kernel** (also kernel or system kernel)
- **Hardware extensions**
  - **Levels of privilege**: “rings”
  - **Memory protection**: memory-management unit (MMU)

Schichtenmodell

Definition
„**Betriebssystem**":

Betriebssystem:
Kern zusammen mit
Hilfsprogrammen

oder

Betriebssystem:
nur Kern

Layer Model

Definition "**operating system**":

Operating system:
Kernel *and* auxiliary programs

or (depending on perspective)

Operating system:
Only the kernel

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”)
- Unterbrechungen („Interrupts”)
- Ausnahmen („Exceptions”)

Levels of Privilege

- Unprivileged layers (“application layer”, “user layer”, “user ring”)
  - may execute “normal” CPU instructions
  - may access its assigned memory areas
  - may call OS functions
- Privileged layer (“system layer”, “kernel layer”, “ring 0”)
  - may execute all CPU instructions
  - may access every memory area
  - may reconfigure the memory protection
  - may access I/O devices

Switch to privileged layer by

- System calls or traps
- Interrupts
- 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.

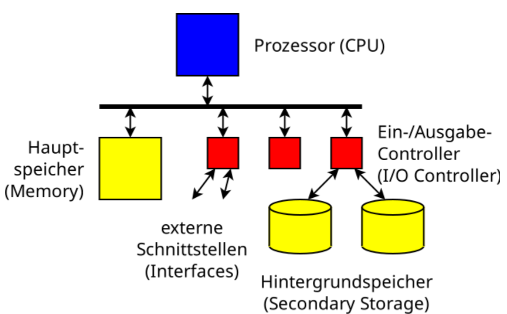
System Calls

Example: Application needs *more memory resources*, step by step:

- application calculates how much more memory is needed
- stores parameter in CPU registers
- switches in the kernel with a special CPU instruction,
  ($\Rightarrow$ from now on privileged!)
- reads parameters from the CPU register
- reserves more memory for itself
- reprograms the MMU
- stores the result in CPU registers
- switches back to the application layer with special CPU instructions,
  ($\Rightarrow$ from now on unprivileged again!)
- retrieves result from the CPU register

Verwaltung von Betriebsmitteln

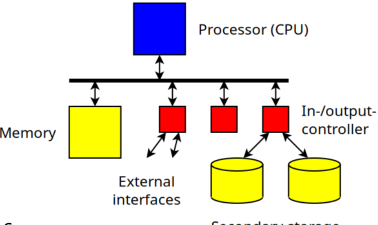
- Aufgaben des Betriebssystem-Kerns
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel



- 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

Management of Resources

- Responsibilities of the operating-system kernel
  - **multiplexing** of resources for multiple users or applications
  - providing **protections**
  - providing **abstractions** for easier handling of resources



- Enable a coordinated shared usage of resources, which can be classified:
  - active, **timely divisible** (processor)
  - passive, only **exclusively usable** (peripheral devices, z. B. printers, etc.)
  - passive, **spatially divisible** (memory, disk space, etc.)
- Support for recovering from errors (segmentation fault)

Klassifikation von Betriebssystemen

- Unterschiedliche Klassifikationskriterien
  - Zielplattform
  - Einsatzzweck
  - Funktionalität

Classification of Operating Systems

- Different criteria for classification
  - target platform
  - intended use
  - functionality

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

Classification of Operating Systems (2)

- Compared to only a small number of "general purpose"-, mainframe- and high-performance computer operating systems, there exits a multitude of small and smallest specialized operating systems:

  `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, ...`

  Usage: embedded systems, often real-time systems, more than 50% proprietary (in-house) solutions

- Alternative classification: by architecture

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

Architecture of Operating Systems

- Scope: tens of thousands or even *multiple millions of code lines*
  ⇒ structuring required
- Various structural concepts
  - runtime libraries (minimal, mostly used for embedded systems)
  - monolithic systems
  - layered systems
  - microkernels (minimal kernels)
- Various protection concepts
  - no protection (μ-Controller)
  - protection of the operating system
  - protection of the operating system and between applications
  - fine-grained protection within application

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)

Components of Operating Systems

- Memory management
  - Who is allowed to store information in memory?
- Process management
  - When is which task scheduled?
- File system
  - storage and protection of long-term data
- Inter-process communication (IPC)
  - communication between different applications or between executed parts (running in parallel) of an application
- Input/Output
  - communication with the “world outside” (user/computer)

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

Prozessor (CPU)

Haupt-
speicher
(Memory)

Ein-/Ausgabe-
Controller
(I/O Controller)

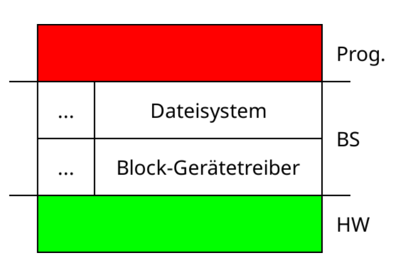
externe
Schnittstellen
(Interfaces)

Hintergrundspeicher
(Secondary Storage)

Overview

- Classification

Überblick (2)



Anwendungsprogramm:

- liest/schreibt Dateiinhalte
- liest/schreibt Verzeichnisinhalte

Dateisystem:

- liest/schreibt Blöcke

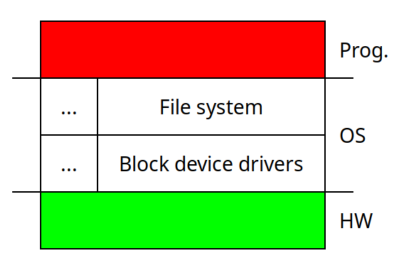
Block-Gerätetreiber:

- liest/schreibt I/O-Register

Hardware:

- liest/schreibt Bytes von/auf Datenträger

Overview (2)



Application:

- reads/writes file contents
- reads/writes directory contents

File system:

- reads/writes blocks

Block device driver:

- reads/writes I/O register

Hardware:

- reads/writes bytes from/to a disk

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

Hardware

- Storage medium (z. B. hard disks, SSD/flash storage, DVD, CD-ROM) with differences; examples
  - size of **blocks**:
    - hard disks: 512 bytes/block
    - CDs: 2048 bytes/block
    - flash: 4096 bytes/block
  - usage of the blocks
    - flash storage only has a limited amount of write cycles for each block ⇒ evenly distribute data
    - hard disks can access neighboring blocks faster than others
  - (typical) size of the medium
    - CD-ROM: approx. 750 MByte
    - DVD: approx. 8.5 GByte
    - hard disk: approx. 4 TByte
    - SSD: approx. 1 TByte

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;
    }
}
```

Block Device Drivers

Example: PC-IDE hard-disk driver (simplified):

```
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

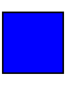
- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Benutzer muss sich nicht um die Ansteuerung und Verwaltung verschiedener Speichermedien kümmern
  - einheitliche Sicht auf den Hintergrundspeicher
- Wesentliche Elemente eines Dateisystems:
  - Dateien (Files)
  - Verzeichnisse /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

File System

- File systems **persistently store data and programs** in files
  - user does not have to care about accessing and managing different storage types
  - unified view of the background storage
- Essential parts of a file system:
  - files
  - directories
  - 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

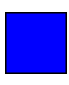
Verzeichnis

- 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

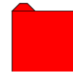
Partition

File System (2)

- File
  - stores data or programs
  - contains additional information



- Directory
  - combines files and other directories
  - makes it possible to give names to files
  - enables an hierarchical name space



- Partition
  - a set of directories and their files
  - are used to physically or logically split amounts of data
    - physical: hard disk, floppy
    - logical: partitioned area on a disk or CD

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

Files

- Smallest unit that can be written to the persistent storage
- Classification:

1. actual files (image, text, program, ...)
2. meta data (date of creation, owner, access permissions, ...)

- Meta data/file attributes:

Name: symbolic name, readable by the user

- z. B. `AUTOEXEC.BAT`

Type: For file systems that differentiate between file types

- z. B. sequential file, character-oriented file, record-oriented file

Place: Where are the files stored physically?

- number of the block on the disk

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)

File Attributes (2)

Size: Length of the file in different units of measure (z. B. bytes, blocks, records)

- closely related to the information about storage location
- is used to verify the bounds of the file, z. B. when reading it

Time stamp: z. B. time and data of creation, last change

- for backups, development tools, monitoring users, ...

Permissions: access permissions (z. B. read & write permissions)

- z. B. only writable by the owner, other users have read-only access

Owner: identification of the owner

- closely related to the permissions
- allocation for accounting (users' quota of disk space)

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)

Operations on Files

- **Create**
  - required storage space is requested
  - entry in the directory is created
  - initial attributes are stored
- **Write**
  - identification of the file
  - possibly request more storage space (reallocation)
  - data is written to the disk
  - possibly modification of the attributes (z. B. length of the file, time of last change)
- **Read**
  - identification of the file
  - data is read from the disk
  - possibly modification of the attributes (z. B. time of last access)

Operationen auf Dateien (Forts.)

- **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

Operations on Files (Forts.)

- **Positioning** of the write/read pointer for the next write or read operation **(seek)**
  - identification of the file
  - in many systems, this positioning takes place automatically when a read or write operation is requested
  - enables explicit positioning
- **Truncate**
  - identification of the file
  - beginning at a certain position (or from the start) the contents of the file get deleted
  - storage is possibly freed
  - modification of the attributes (z. B. length of the file, time of last change)
- **Delete**
  - identification of the file
  - removing of the file from the directory and release of blocks on the disk

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)

Directories/Catalogs

- A directory groups files and other directories
- Alternatives for grouping
  - Linking by naming
    - directory contains names and references to files and other directories (z. B. UNIX, Windows)
  - Grouping by conditions
    - directory contains names and references to files that comply with certain conditions:
      - z. B. same group number in CP/M
      - z. B. attributes-dependent grouping or dynamic grouping in BeOS-BFS
- Directories enable locating files
  - link between internal and external identification
    (name of the file – blocks on the disk)

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, ...

Operations on Directories

- **Reading** of directory content
  - data of the directory is read and typically returned entry-by-entry
- **Creating** and **Deleting** takes place implicitly during creation and deletion of files
- **Creation** of directories
- **Deletion** of directories

Attributes of directories

- Most attributes of file also apply for directories
  - name, storage location, size, time stamps, permission, owner, ...

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

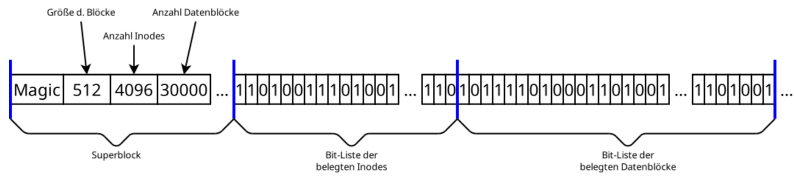
- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
- Dateiattribute
  - das Betriebssystem verwaltet zu jeder Datei eine Reihe von Attributen (Rechte, Größe, Zugriffszeiten, Datenblöcke, ...)
  - die 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

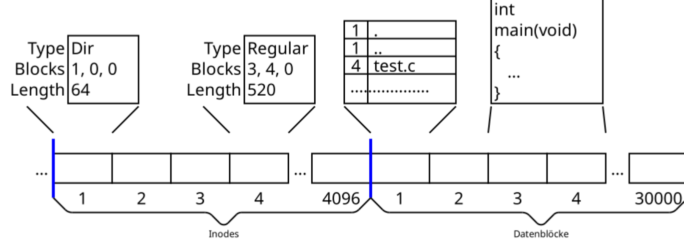
File System – Example Linux/UNIX

- File
  - simple, unstructured set of bytes
  - arbitrary content; content is transparent for the operating system
  - can be dynamically extended
- File attributes
  - the operating system manages a set of attributes for each file (permissions, size, time of access, data blocks, ...)
  - the attributes are saved in a special management structures, called the *file header*
    - Linux/UNIX: *Inode*
    - Windows NTFS: *Master File Table* entry
- Namespace
  - flat namespace: inodes are enumerated
  - hierarchical namespace: directory structure maps the file and path names to the inode numbers

Dateisystem-Struktur

- Struktur auf Medium (vereinfacht)

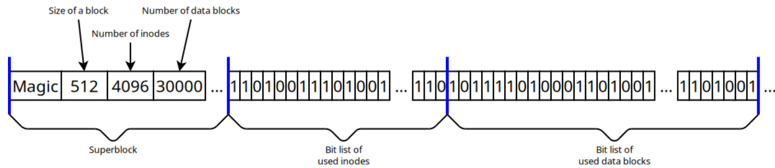


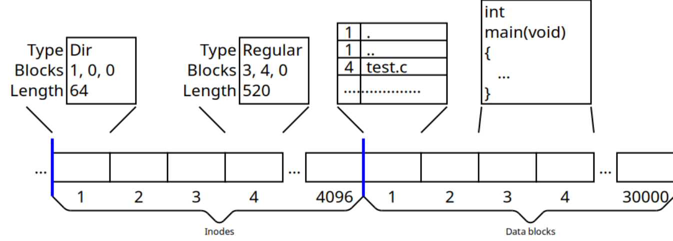


- **mkfs** legt leere Struktur an; **fsck** überprüft Struktur

File-System Structure

- Structure on medium (simplified)

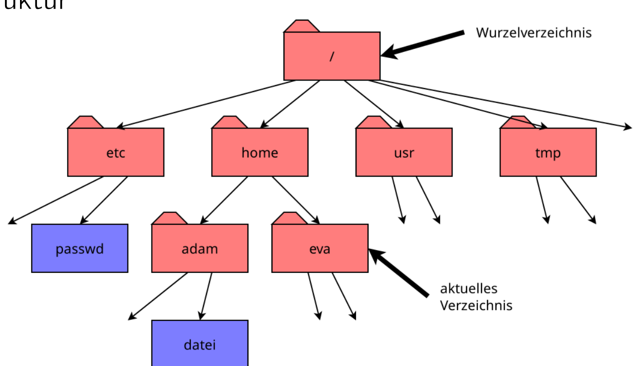




- `mkfs` creates an empty structure; `fsck` verifies the structure

Pfadnamen

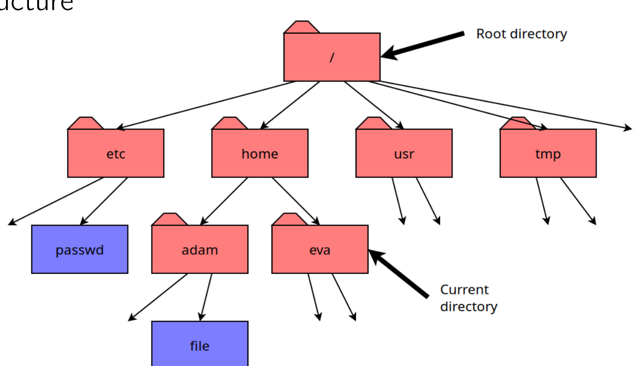
- Baumstruktur



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

Path Names

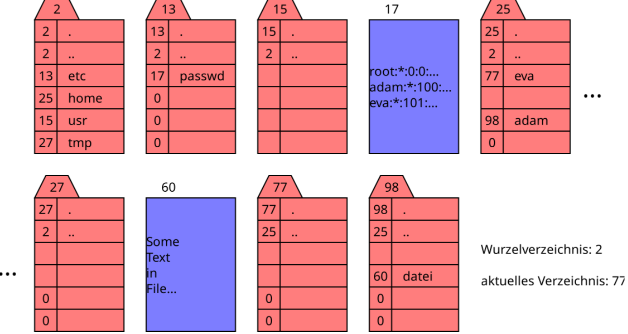
- Tree structure



- Paths
  - z. B. `/home/adam/datei`, `/tmp`, `../adam/datei`
  - `/` represents separator (*forward slash*)
  - beginning `/` stands for the root directory; else, implicit start in the current working directory

Pfadnamen (2)

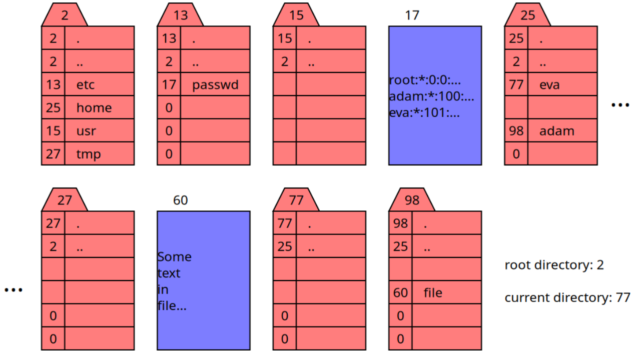
- eigentliche „Baumstruktur"



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

Path Names (2)

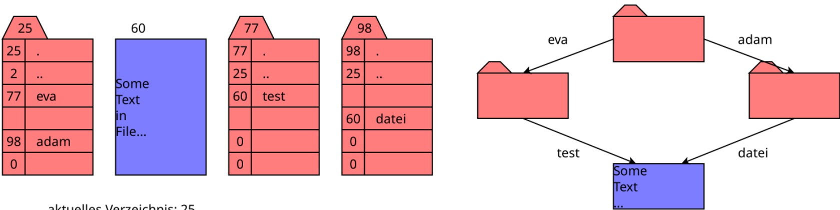
- Actual “tree structure”



- Example, resolving the path “`../adam/datei`”:
  - 77 + “`../adam/datei`” ⇝ 25 + “`adam/datei`”
  - 25 + “`adam/datei`” ⇝ 98 + “`datei`”
  - 98 + “`datei`” ⇝ 60

Pfadnamen (3)

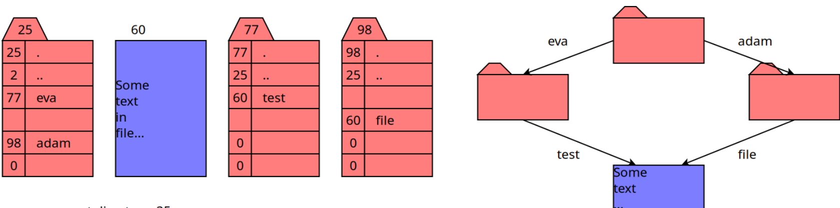
- Es können mehrere Verweise (**Hard Links**) auf eine Datei existieren:



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

Path Names (3)

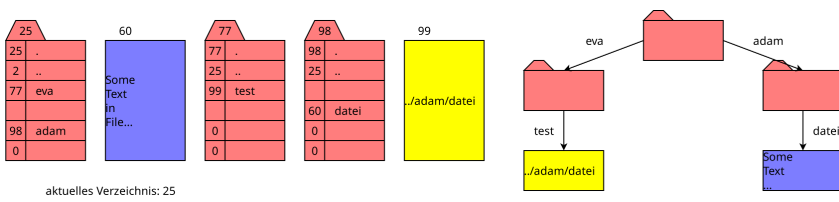
- There can exist multiple references (**hard links**) to the same file:



- Example, resolving the path "`adam/datei`":
  - 25 + "`adam/datei`" $\rightsquigarrow$ 98 + "`datei`"
  - 98 + "`datei`" $\rightsquigarrow$ 60
- Example, resolving the path "`eva/test`":
  - 25 + "`eva/test`" $\rightsquigarrow$ 77 + "`test`"
  - 77 + "`test`" $\rightsquigarrow$ 60
- A file is deleted, if no more references exist that refer to it.

Pfadnamen (4)

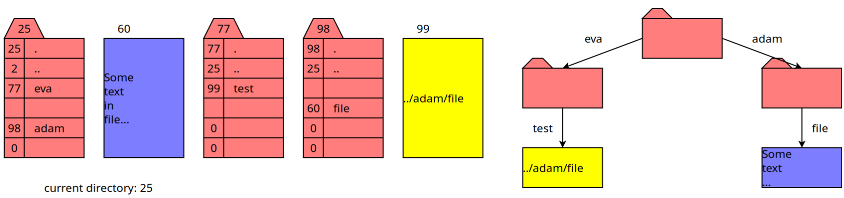
- Es können mehrere symbolische Verweise (**Symbolic Links**) auf eine Datei oder ein Verzeichnis existieren:



- 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.

Path Names (4)

- There can exist multiple symbolic references (**symbolic links**) to a file or directory:



- Example, resolving the path "`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
- A symbolic name persists even if the file or directory does not yet exist or does not longer exist.

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

Owners and Permissions

- Owner
  - Each owner is represented by a unique user identification number (UID).
  - A user can belong to one or more user groups, each of which is represented by a unique group identification number (GID).
  - A file or directory is mapped to exactly one user and one group.
- Permissions on files
  - Read, write, execute (only by the owner)
  - Can be individually modified for the owner, for members of the group or for all others.
- Permission on directories
  - Read, write (creating and deleting of files/directories), right of pass-through
  - Write permission can be restricted for each file individually.

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 */
```

Inodes

- Attributes (permissions, owners, etc.) of a file, or of a directory are stored in the so called **inodes** (simplified):

```
int st_mode;        /* type and permissions */
int st_nlink;       /* number of Hard Links */
int st_uid;         /* owner */
int st_gid;         /* group */
long st_size;       /* length of the file in bytes */
int st_block[...];  /* list of the (indirect) blocks */
time_t st_atime;    /* last time of reading */
time_t st_mtime;    /* last time of modification */
time_t st_ctime;    /* last change of attributes */
```

- **File system** assigns each inode a number and a storage location (disk/partition):

```
int st_ino;         /* inode number */
int st_dev;         /* Disk/partition number */
```

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);
```

Programming Interface for Inodes

- `stat`, `lstat` return file attributes from an inode
- Function interface:

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

- Arguments:
  - `path`: path name
  - `buf`: pointer to a buffer, which will be filled with the information from the inode
- Return value:
  - 0 if OK
  - -1 if an error occurred (`errno` variable contains error number)
- Example:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Error handling...! */
printf("Number of the inode: %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);
```

Programming Interface for Directories

- Handling links or directories
  - Creating (an empty directoy)

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

  - Deleting (an empty directory)

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

  - Creating a hard link

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

  - Creating a symbolic link

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

  - Deleting a link (and possibly the file)

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

  - Reading a symbolic link

    ```
    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 */
};
```

Programming Interface for Directories (2)

- Reading directories (interface of the Linux kernel)
  - open(2), getdents(2), close(2)
  - Linux-specific
- Reading directories (interface of the C library)
  - Opening a directory

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

  - Reading an entry

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

  - Closing a directory

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

  - Structure `struct dirent` (simplified)

```
struct dirent {
    int d_ino;                  /* Number of the inode */
    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)

Directories (3): opendir/closedir

- Function interface:

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

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

- Argument of **opendir**:
  - **path**: name of the directory
- Return value of **opendir**:
  - Pointer to a structure of type **DIR** if OK
  - **NULL** if an error occurred (**errno** variable contains error number)

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!

Directories (4): `readdir`

- Function interface:

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

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

- Argument:
  - `dirp`: pointer to `DIR` data structure
- Return value:
  - Pointer to data structure of type `struct dirent`, if OK
  - NULL, if directory has been written entirely (`errno`-variable remains unchanged)
  - NULL, if an error occurred (`errno` variable contains error number)
- Note: The memory for `struct dirent` can possibly be overwritten by the next `readdir` call!

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
```

Directories (5): Example

- Output all file names in the current directory ("."):

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

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

dirp = opendir(".");                    // opening cur. dir
if (dirp == NULL) ...                   // error

while (1) {
  errno = 0;
  de = readdir(dirp);                   // reading entry
  if (de == NULL && errno != 0) ...     // error
  if (de == NULL) break;                // end reached

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

ret = closedir(dirp);                   // closing directory
if (ret < 0) ...                        // error
```

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);
```

Programming Interface for Files

- Function interface:

```
#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

```c
#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
```

Files (2): Example

- Copy program

```
#include <fcntl.h>

int ret;

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

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

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

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!

Files (3)

- A call to the `write` function has to
  - verify the file descriptor whether file is opened and/or writable
  - verify the length and address of the buffer
  - determine the block(s) of the medium which have to be written to
  - read the block(s) from the medium (unless the whole block needs to be written)
  - overwrite the required bytes in the block(s)
  - transfer the block(s) back to the medium
  - modify the attributes (last change, length of the file)
- `write` is therefore a system call
- ⇒ `write` is a costly operation, (`read` analogously)!
- ⇒ Improvement: read/write many bytes at once (ideally: multiples of the block size)
- ⇒ Use `fopen`, `fclose`, `fread`, `fwrite`, `getchar`, `putchar`, `fscanf`, `fprintf`, ... functions from the C library!

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

Special Files

- Peripheral devices (disks, printers, CD, terminal, scanner, ...) are represented as special files (`/dev/sda`, `/dev/lp0`, `/dev/cdrom0`, `/dev/tty`, ...)
- Their inode contains
  - Type:
    - Block-oriented devices (drives, CD, DVD, SSD, ...)
    - Character-oriented devices (printer, terminal, scanner, ...)
  - Instead of block numbers:
    - Major number: type of the device (disk, printer, ...)
    - Minor number: number of the device (3rd printer, 5th terminal, ...)
- Opening a device creates a possibly exclusive connection to the device, provided by drivers
- Devices can be accessed with **read**, **write**, and **ioctl** operations

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) ...
```

Special Files (2): Example

- Output to a printer

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

/* Establish connection to printer 0. */
fd = open("/dev/lp0", O_WRONLY);
if (fd < 0) ...

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

/* Write to the printer. */
ret = write(fd, "Hallo, Drucker!\n\f", 17);
if (ret < 0) ...

/* Close connection. */
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

Partitions

- Each disk can contain a file system as a whole
  - the disk then corresponds to a single partition
- However, each disk can be divided into multiple partitions
  - first block of the disk contains partition table
  - partition table contains information about
    - how many partitions exist
    - how large these partitions are
    - where they start
- Each partition
  - is represented by a special file; z. B.
    - `/dev/sda`, `/dev/sdb` (whole disk)
    - `/dev/sda1`, `/dev/sda2`, `/dev/sdb1` (parts of the disk)
  - contains a single file system

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

Partitions (2)

- Trees for each partition can be combined to a single homogeneous file tree (boundaries not visible for the user!)
  - “mounting” of file trees
- A single specified file system is the *root file system*, whose root directory is the root directory for the whole system
  - Other file systems can be mounted to the existing file system with the `mount` operation and removed from it with the `umount` operation.
  - With the help of *network file system* (NFS) even directories of other computers can be mounted to the local file system.
    $\Rightarrow$ *“hidden” boundaries* between file systems of different computers

Montieren des Dateibaumes

- Beispiel

/dev/sda1

Root File System

/

etc

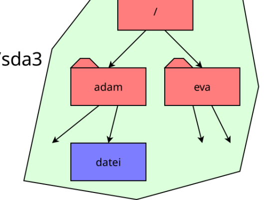
home

usr

tmp

passwd

/dev/sda3




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

Mounting to the File Tree

- Example

/dev/sda1

Root File System

/

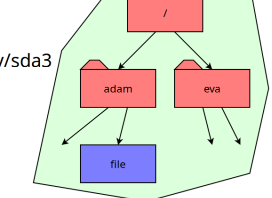
etc

home

usr

tmp

passwd




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

Montieren des Dateibaumes (2)

- Nach Ausführung des Montierbefehls

Mounting to the File Tree (2)

- After execution of the mount operation

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

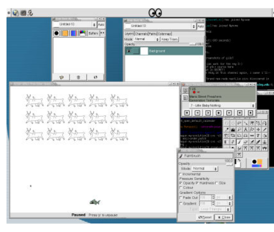
30 Multiprozessoren

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

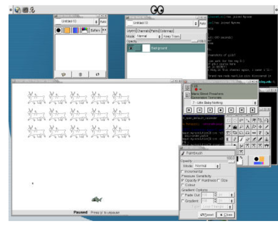
Jedes laufende Programm bekommt Hardware zugeteilt:

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

und kann Betriebssystem-Kern-Funktionen aufrufen.

Overview

- **Multiple** Programs that
- run **concurrently**,
- are **dynamically** started/stopped,
- control their environment
- via **defined I/O functions**.



Source: www.wikipedia.org

Each running program gets hardware assigned:

- CPU (time shares)
- memory (parts of the main memory)
- ... and can call operating-system–kernel functions

Definitionen

Programm: Folge von Anweisungen

Prozess: laufendes Programm mit seinen Daten

Hinweis: ein Programm kann sich mehrfach in Ausführung befinden!

Definitions

Program: set of instructions

Process: running program and its data

Hint: one program can be in execution multiple times (e.g., PDF viewer)!

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

Processes

- Definition "process": running program with its data
- Different point of view:

microcontroller process	UNIX-/Windows/... process
processor	time shares of the physical processor
memory	virtual memory
interrupts	signals
I/O devices	I/O operating-system functions

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

Processes (2)

- Multi-program operation (“multitasking”)
  - multiple processes can be executed virtually simultaneously
  - if there are less processors then there are running processes, time shares for using a processor are given to the processes: **time-sharing system**
  - the OS kernel the decides which process receives how much computing time: **scheduling**
  - the switch between processes takes place by the OS kernel: **dispatching**
  - running processes do not know at which point a subsequent process is dispatched

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

Process States

A process is always in one of the following states

New (or created):
: Process has been created but does not have all necessary resources to run

Ready:
: Process has all necessary resources (except CPU) and is ready for execution/running

Running:
: Process is executed by a physical processor

Waiting (or blocked):
: Process waits for an event (completion of an I/O operation)

Terminated:
: Process is terminated but not all of its resources are yet freed

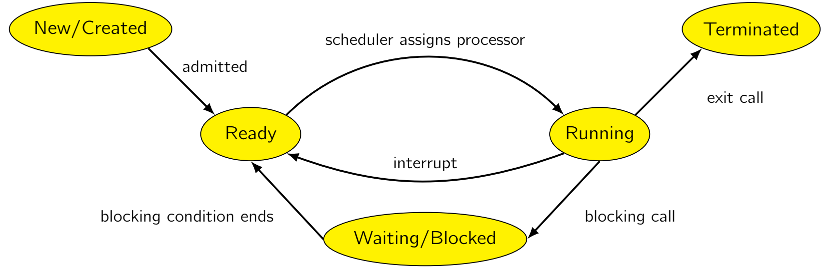
Prozesszustände (2)

- Zustandsdiagramm mit Übergängen:

Nach Silberschatz, 1994

Process States (2)

- State diagram with transitions:



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.

Context Switch

- Each process has a context (i.e., its state)
  - contents of processor registers
  - contents of memory areas
  - open files, current directory, ...
- When switching a process (context switch)
  - the contents of the processor registers are saved,
  - a new process is selected,
  - the execution environment for the new process is established
    - reprogramming of the MMU
    - change of the open files and current working directory, ...
  - the stored registers of the new process are loaded

Prozesswechsel

- Ablauf von zwei Prozessen in Benutzermodus und Kern mit Umschaltung

blockiert/bereit
blockiert/bereit
Prozess A
blockiert/bereit
Prozess B
Funktion
return
Trap
Return from Trap
Trap
Return from Trap
Trap
Return from Trap
Systemaufruf mit Umschaltung
Kontext P. B
Dispatcher
Dispatcher
Kontext P. A
Systemaufruf ohne Umschaltung
Betriebssystem

Context Switch

- control flow of two processes in user mode and kernel

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
  - ...

Process Control Block

- Process Control Block (PCB)

  Data structure of the kernel that contains all necessary data for a process

  Example UNIX:
  - process ID (PID)
  - process state (running, ready, ...)
  - register
  - memory mapping
  - owner (UID, GID)
  - root directory, working directory
  - open files
  - ...

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

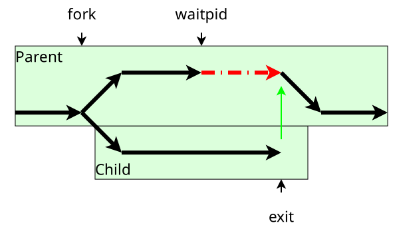
Beispiel: UNIX

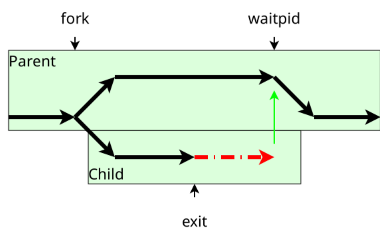
- Überblick:

  fork: Erzeugung von neuen Prozessen

  exit: Terminieren von Prozessen

  waitpid: Warten auf das Terminieren von Prozessen

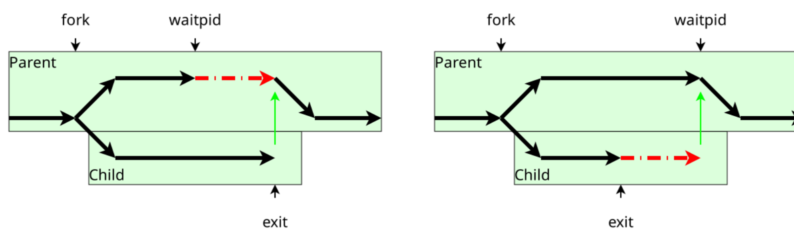




Example: UNIX

- Overview:

fork: creates new processes
exit: terminates a process
waitpid: waits for the termination of a process



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);
```

Programming Interface

- Duplicating the currently running process

```
#include <unistd.h>

pid_t fork(void);
```

- Termination of the currently running process

```
#include <stdlib.h>

void exit(int status);
```

- Waiting for the termination of a different process

```
#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;
}
```

Example

```
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`

Process Creation

- The child process is a copy of the parent process
  - same program
  - same data (contents of variables)
  - same program counter
  - same current and root directories
  - same open files
- Only difference
  - different process IDs
  - returned value from `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!

Execution of Programs

- The program that is executed by a process can be replaced by an other program:

```
#include <unistd.h>

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

Example:

```
...                /* 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[])
{
    ...
}
```

The previously running program is terminated, the new one is started.
Only the **program is replaced**.
Still the **same process is running**!

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;
}
```

Starting a Program

- Example: Starting of the program `./prog` with parameters `-a` and `-b`

... as a foreground process:

```
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;
}
```

... as a background process:

```
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**

Signals

- Microcontrollers react to concurrent events (interrupts) with so-called interrupt service routines
- A similar concept exists on the level of processes: **signals**

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

Signals (2)

Interrupt: **asynchronous** signal triggered by an “external” event

- CTRL-C pressed on the keyboard
- timer expired
- child process terminated
- ...

Exception: **synchronous** signal triggered by an activity of a process

- access to invalid memory address
- illegal machine instruction
- division by 0
- write operation to a closed communication connection
- ...

Communication: a process sends an event to another process

Signale (3)

CTRL-C:

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

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

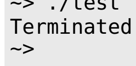
Illegaler Speicherzugriff:

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

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

Inter-Prozess-Kommunikation:

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



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

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

Signals (3)

CTRL-C:

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

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

Inter-process communication:

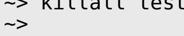
```
int main(void)
{
    while (1) {
    }
}
```

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

Illegal memory access:

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

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



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

29-Signale: 2026-04-13

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)

...

Reaction to Signals

abort:

creates a *core dump* (contents from memory and registers are saved in the file `./core`) and terminates the process

default setting for all exceptions (to make debugging easier)

exit:

terminates the process (without core dump)

default setting for z. B. CTRL-C, kill signal

ignore:

signal gets ignored

default setting for all “unimportant” signals (z. B. child process terminated, size of the terminal window has changed)

...

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

Reaction to Signals (2)

...

handler:

call to a signal handler function, continuation afterwards

no default setting possible since handling requires program-dependent behavior

stop:

stops the process

default setting for stop signal

continue:

continues a process

default setting for continue signal

reaction can be changed with system call (`sigaction`)

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 ... */
```

- ...

Programming Interface

- Configuration of the signal handler function
  (equivalent to configuring a function as ISR)

```
#include <signal.h>

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

`struct sigaction` contains:

```
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);
```

- ...

Programming Interface

- ...
- For solving concurrency problems: blocking/unblocking of signals necessary, *equivalent to* µ-Controller's `cli()`, `sei()`

```
#include <signal.h>

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

  - **`SIG_BLOCK`**: block all given signals
  - **`SIG_UNBLOCK`**: unblock all given signals
  - **`SIG_SETMASK`**: set a signal mask

- Unblocking + passive waiting for signal + blocking again, *equivalent to* µ-Controller's `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);
```

Programming Interface

- ...
- Creating an empty signal list

```
#include <signal.h>

int sigemptyset(sigset_t *mask);
```

- Creating a full signal list

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

- Adding one signal to an existing signal list

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

- Removing one signal from an existing signal list

```
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)

Programming Interface

- Typical signals:

  SIGSEGV: “segmentation fault” (invalid access to memory)
  SIGINT: “interrupt” (CTRL-C)
  SIGALRM: “alarm” (timer expired)
  SIGCHLD: “child” (child process terminated)
  SIGTERM: “terminate” (termination of the process; possible to handle in program)
  SIGKILL: “kill” (termination of the process; impossible to handle in program)

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 Example 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));
}
```

(error handling omitted...)

```
~> ./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 Example 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);
}
```

(error handling omitted...)

```
~> ./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...)

Signal Example 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.
  ...
}
```

(error handling omitted...)

Signale und Nebenläufigkeit

- Signale erzeugen Nebenläufigkeit innerhalb von Prozessen
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller

Signals and Concurrency

- Signals create **concurrency** inside processes
- Resulting problems are **analogous to concurrency of interrupts** on a microcontroller platform
- For example: lost update, lost wakeup, ...

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

(Fehlerbehandlung weggelassen...)

```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!

Example of Concurrency

```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();
  }
}
```

↓ signal here

(error handling omitted...)

```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
...
```

← problem here!

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...

Solution for the Concurrency Example

1. Solution

```c
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. Solution

```c
/* 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);
```

More solutions exist...

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

Problems with Concurrency

- Additional problem:
  internal functionality of library functions unknown in general
- Example 1:
  `printf` inserts character into a buffer
  $\Rightarrow$ use of `printf` in the main program *and* in the signal handler is dangerous
- Example 2:
  `malloc` searches list for free memory area; `free` inserts a block into the list
  $\Rightarrow$ use of `malloc`/`free` in the main program *and* in the signal handler is dangerous
- Solutions:
  - block signals during the execution of **critical sections** or
  - no unknown library functions is called from inside a signal handler function

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

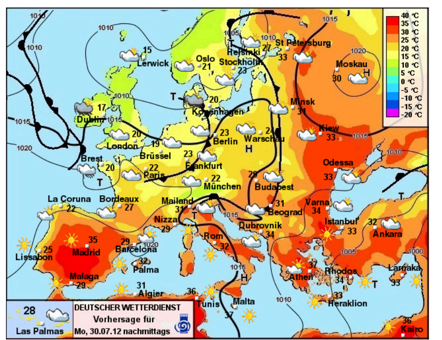
- Mehrere Prozesse zur Strukturierung von Problemlösungen
  Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
  - z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
  - z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Web-Browser)
  - 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

Processes

- Multiple **processes** for structuring of solutions
- Tasks of an application can be modeled easier when divided into **multiple cooperating subprocesses**
  - z. B. applications with multiple windows (one process per window)
  - z. B. applications with many concurrent tasks (web browser)
  - z. B. client server applications;
    for each request a new process gets started (web server)
- **Multiprocessor** systems can only be used efficiently with multiple processes running in parallel
  - in the past this was only viable for high-performance computers (aerodynamics, weather prediction)
  - with today's multi-core systems very common

Beispiel: Berechnung einer Wetterkarte

- Berechnung der Wetterkarte muss so schnell wie möglich erfolgen

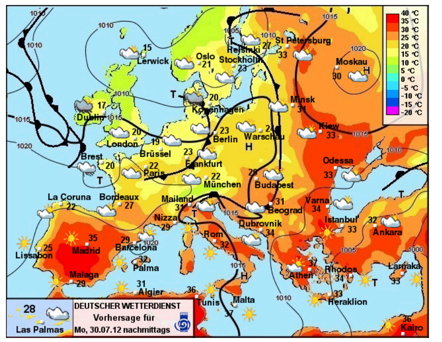


Quelle: www.wetterdienst.de

- Ansatz: Mehrere Prozessoren berechnen jeweils einen Teil der Karte

Example: Computing of Weather Map

- Computation of a weather map has to be as fast as possible

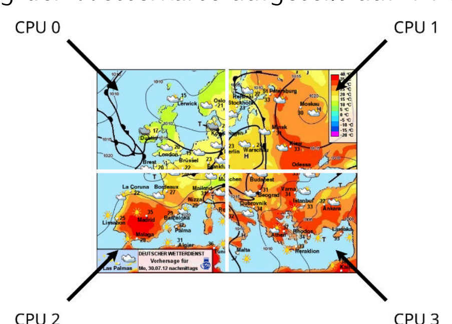


Source: www.wetterdienst.de

- Approach: multiple processes compute distinct parts of the map

Beispiel: Berechnung einer Wetterkarte (2)

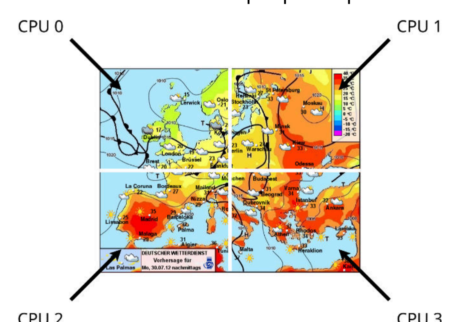
- Z.B. Berechnung der Wetterkarte aufgeteilt auf 4 Prozessoren:



- Alle Prozessoren greifen auf einen gemeinsamen Speicher zu, in dem das Ergebnis berechnet wird.

Example: Computing of Weather Map (2)

- Z. B. computation of a weather map split up between 4 processors:



- All processes access a shared memory area for computing the result

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
```

Processes with Shared Memory

- Use of *shared memory* by multiple processes

```
char *ptr = mmap(NULL, NBYTES, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (ptr == MAP_FAILED) ... // Error

for (i = 0; i < NPROCESSES; i++) {
    pid[i] = fork();
    switch (pid[i]) {
    case -1: ...            // Error
    case 0:
        do_work(i, ptr);
        _exit(0);
    default:;
    }
}
for (i = 0; i < NPROCESSES; i++) {
    ret = waitpid(pid[i], NULL, 0);
    if (ret < 0) ...        // Error
}

ret = munmap(ptr, NBYTES);
if (ret < 0) ...            // Error
```

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);
}
```

Example: Length of a Vector

- Calculation of the length/norm of a vector with N elements in one process:

```
#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);
}
```

Example: Length of a Vector (2)

- Compute of the length/norm of a vector with N elements with 4 processes:

```
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
  - `#include`s 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 $N = 100000$

Example: Length of a Vector (3)

- Note that example is incomplete
  - `#include` instructions missing
  - error handling missing
  - ...
- Nonetheless, example illustrates
  - programming is more complex
  - program structure less straight-forward
  - parallel algorithm is harder to understand
- Benefits can be unintuitive
  - Significant overhead expenses for forking processes
  - The additional overhead for forking is only beneficial for very large vectors, that is, for values of N greater than 100 000 (depending on the actual machine)

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**

Processes with Shared Memory (2)

Advantage of the solution above: in multiprocessor systems, **physically parallel execution** is possible

BUT

each process needs its own resources

- memory mapping
- permissions
- open files
- root and working directory
- ...

⇒ **creation, termination, and switching of processes is expensive**

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

Multi Processor Systems

In multiprocessor systems, physically parallel execution is possible

but

process creation, termination and, switching are expensive!

For **practical applications**, we therefore should take into account:

- only few processes should be created/terminated
- *never create more processes than there are physical processors*

or

instead of expensive processes use more lightweight, simple **threads**

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
  - ...

Threads in a Process

Solution: **multiple** threads in **one** execution environment

- Each **thread** has for its own execution
  - individual program counter
  - individual set of registers
  - individual stack (for local variables)
- **Shared execution environment** provides a set of resources
  - memory mapping
  - permissions
  - open files
  - root and working directory
  - ...

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

Threads in a Process (2)

- The concept of a process is split up into one **execution environment** and one or more **threads**
- A classical UNIX process is a thread in an execution environment

Fäden in einem Prozess (3)

- Erzeugen/Terminieren eines Fadens in einem Prozess erheblich billiger als das Erzeugen/Terminieren eines Prozesses (weniger 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!)

Threads in a Process (3)

- *Creation/termination of a thread* are less expensive compared to creating/terminating a process (less individual resources required)
- *Switching between threads* inside one process is also cheaper than switching between processes
  - only the registers and the program counter have to be changed (similar to a function call)
  - memory mapping does not have to be changed (cached content remains valid!)

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!

Coordination / Synchronization

- Threads work concurrent/parallel and have shared memory
  ⇒ all problems occurring when dealing with signals and interrupts and accessing shared data also exist
- *Differences* between threads and ISRs/signal handling functions:
  - "main thread" of an application and an ISR/signal handling function are unequal in their behavior
    - ISRs/signal handlers function interrupts the main thread but ISRs/signals are not interrupted by themselves
  - two threads are equal
    - a thread can always be interrupted in favor of an other thread by the scheduler or be run in parallel to another one

⇒ It is insufficient to block signals!

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.

Coordination / Synchronization (2)

- Basic problems
  - mutual exclusion (**coordination**)
    Example:
    A thread wants to read a set of data and prevent other threads from changing the data in this time.
  - mutual waiting (**synchronization**)
    Example:
    A thread waits for an other thread so that they can combine partial results that each thread has computed.

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

Coordination / Synchronization (3)

- Example of complex problem with coordination and synchronization
  - **Bounded buffer**
  - Threads write data into a buffer, others remove data from it; critical situations:
    - access to the buffer
    - buffer empty/full

Inserting an element:

- wait until there is free space
- wait until no other thread reads/writes from/to the buffer
- write into the buffer
- send signal that there is a new element in the buffer

Removing an element:

- wait until an element is in the buffer
- wait until no other thread reads/writes
- read from the buffer
- send signal that there is free space in the buffer

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!

Mutual Exclusion

- Simple implementation with **mutex** variables

```
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);
...
```

Only the thread that called `lock` is allowed to call `unlock`!

- Realization (only conceptual!)

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

```
void unlock(volatile int *m) {
    *m = 0;
}
```

`lock` (and `unlock`) have to be **executed atomically**!

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.

Counting Semaphores

- A **semaphore** (greek. character carrier) is a data structure with two instructions (refer *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 and V have to be executed **atomically**!

P and V do not have to be called from the same thread.

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;
}
```

Bounded Buffer (2)

Bounded integer buffer example:

```
#define N 1000
volatile int mutex = 0;
volatile int alloc = 0, free = N;
volatile int head = 0, tail = 0;
volatile int buf[N];
```

Inserting element:

```
void put(int x) {

    P(&free);
    lock(&mutex);
    buf[head] = x;
    head = (head + 1) % N;
    unlock(&mutex);
    V(&alloc);

}
```

Removing element:

```
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

Spin Lock vs. Sleeping Lock

- **Spin lock**
  - active waiting until mutex variable is free ($= 0$)
  - conceptually similar to polling
  - thread stays in the state *running*

  Problem: when there is only one processor available, computation time is wasted until the scheduler schedules a switch
  - only another running thread can free the mutex variable
- **Sleeping Lock**
  - passive waiting
  - thread changes state to *blocked*
  - when `unlock` occurs, the blocked thread changes to the state *ready*

  Problem: for really short critical sections the expenses for blocking/waking up and switching are disproportionately expensive

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*, ...

Implementation Spin Lock

- Main problem: atomicity of mutex request and setting

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

critical section

- Solution: special *machine instructions* that enable to atomically request and modify a cell in the main memory
  - *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

31-Threads: 2026-04-13

Implementation Sleeping Lock

- Two problems:

1. Conflict with a second `lock` operation:
   Atomicity of mutex request and setting

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

critical section 1

2. Conflict with second `unlock`: *lost-wakeup* problem

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

critical section 2

- Scenarios:

1. switching of processes during a `lock` operation
2. actually parallel running `lock`- and/or `unlock` operations

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

```c
void lock(volatile int *m)
{
    enter_OS();
    cli();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    sei();
    leave_OS();
}
```

```c
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    *m = 0;
    wakeup_waiting_threads();
    sei();
    leave_OS();
}
```

Implementation Sleeping Lock (2)

- Solution scenario (1):
  prevent process switches
  - process switches are functions of the OS kernel
    - takes place in the context of system calls (z. B. `exit`)
    - or in the context of an interrupt handler (z. B. time-slice expiration interrupt)

  ⇒ `lock`/`unlock` are implemented in the OS kernel; OS kernel has preemption avoidance

```
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

Implementation Sleeping Lock (3)

- Solution scenario (2):
  Prevent parallel execution on another processor

```
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()` and `V()` similar

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 ...
```

Example: POSIX Threads (pthread)

- Standardized programming interface: **pthread library** (IEEE-POSIX-Standard P1003.4a)
- pthread interface (basic functions):

  `pthread_create:` create a new thread

  `pthread_exit:` thread can terminate itself

  `pthread_join:` wait for the end of a thread

  `...:`

- Functions are combined in the pthread library

```
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 Interface

- Thread creation

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *param);
```

- Parameters

`tid`: Pointer to a variable that will store the ID of the thread.

`attr`: Pointer to attributes (z. B. size of the stack) for the thread.
`NULL` if standard attributes are chosen.

`func`, `param`: The newly created thread will execute the function `func` with parameter `param`.

- The returned value usually is 0. In case of an error, an error code (similar to `errno`) is returned.

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 Interface

- Terminating a thread (on **return** from inside **func** or):

```
#include <pthread.h>

void pthread_exit(void *retval);
```

  The thread is terminated and **retval** is returned (see **pthread_join**.

- Waiting for a thread and checking the **pthread_exit**-status:

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **retvalp);
```

  Waits for the thread with given thread ID **tid** and returns its return value via **retvalp**.
  The returned value is 0. In case of an error, an error code (similar to **errno**) is returned.

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 Example

- Example (matrix-vector multiplication; $\vec{c} = A\vec{b}$):

```
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 Coordination & Synchronization

- Coordination via mutex (mutual exclusion) variables
  - Creation of mutex variables

```
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 Example

- Mutex example:

```
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 Coordination & Synchronization (2)

- Synchronization with *condition variables*
  - variables used for waiting for termination (sleep)
  - a termination is signaled with such variables (wakeup)
  - creation of a condition variable

```
pthread_cond_t c;
pthread_cond_init(&c, NULL);
```

  - waiting for a condition

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

  - signaling of a condition

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *c);
int pthread_cond_broadcast(pthread_cond_t *c);
```

`pthread_cond_signal` wakes up *one* thread, `pthread_cond_broadcast` wakes up *all* threads waiting for the condition

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);
}
```

pthread Example (2)

- Example: counting 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
```

Threads, Coordination, and Synchronization in Java

- Thread concept, coordination, and synchronization are integrated into Java
- Creation of threads via a thread class; example:

```
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.

Threads, Coordination and Synchronization in Java (2)

- Coordination and synchronization can take place in Java with the help of any object
  - Coordination via **synchronized** blocks

```
synchronized(obj) {
    ...
}
```

    Such a block calls a **lock** for the given object **obj** at the beginning and then executes the given instructions. Before leaving the block, the corresponding **unlock** is called.

  - Synchronization via **wait**, **notify** and **notifyAll**

    obj.wait(): Waits for the signal of a termination on the given object **obj**.

    obj.notify(): Signals the termination on the given object **obj** to *a single* waiting thread.

    obj.notifyAll(): Signals the terminationn on the given object **obj** to *all* waiting threads.

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();
        }
    }
}
```

E

dem pthread-Beispiel...

Threads, Coordination and Synchronization in Java (3)

- Example coordination and synchronization:

```
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();
        }
    }
}
```

In analogy to the pthread example...

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();
    }
}
```

Threads, Coordination and Synchronization in Java (4)

- Simplified notation (corresponds to the “monitor” concept):

```
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();
    }
}
```

Synchronization Primitives in Python

- Mutual exclusion in Python

```
lock = asyncio.Lock()

async with lock:
    '''
    from here on, we can exclusively
    access a shared state
    '''
```

- equivalent to

```
lock = asyncio.Lock()

await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Systemnahe Programmierung in C

Teil E Speicher

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Sommersemester 2026

http://sys.cs.fau.de/lehre/ss26

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`.

Size of Types and Objects

- Size of standard types is known; e.g.:
  - `char`: 1 byte
  - `uint16_t`: 2 bytes
  - `uint32_t`: 4 bytes
  - ...
- Size of data structures:
  - **Arrays**: An array of *N* elements needs *N* times the space of one element
  - **Struts**: Structs need (at least) the space of all elements combined

Their size can be determined:

```
sizeof type
```

or

```
sizeof var
```

`sizeof`-operator returns a value of type `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! **) */
}
```

Dynamic Allocation of Memory: Heap

- **Heap** := RAM memory that is explicitly used by the program
  - lifespan independent of program structure (i.e., blocks)
- Allocation and freeing via two basic operations
  - `void *malloc(size_t n)` allocates a memory area of size *n*; returns `NULL` pointer on error
  - `void free(void *pmem)` frees a previously with `malloc()` allocated memory area
- Example

```
#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;
}
```

Dynamic Allocation of Memory: Linked Lists

Example: Allocation and insertion of a list element into a list

```
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;
}
```

Dynamic Allocation of Memory: Linked Lists (Forts.)

Example: Removing and freeing of a list element

```
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;
}
```

Dynamic Allocation of Memory: Strings

Example: Duplicating a string

```
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

Organization of Memory

```
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)
}
```

Where does the memory for these variables come from?

- Static allocation – allocation during compilation / linking
  - Concerns all global/static variables and the code itself ↪ 12–5
  - Allocation by getting placed into a section

.text	– contains program code	`main()`
.bss	– contains all variables initialized with 0	`a`
.data	– contains all variables initialized with other values	`b,s`
.rodata	– contains all constant variables	`c`

- Dynamic allocation – allocated during runtime
  - Concerns all local automatic variables and explicitly allocated memory

Stack	– contains all auto variables that are currently alive	`x,y,p`
Heap	– contains with `malloc()` explictly allocated memory areas	`*p`

Speicherorganisation auf einem µC


```c
int a;
int b = 1;
const int c = 2;

void main(void) {
    static int s = 3;
    int x, y;
    char *p = malloc(100);
}
```

Organization of Memory on a µC

```
int a;
int b = 1;
const int c = 2;

void main(void) {
    static int s = 3;
    int x, y;
    char *p = malloc(100);
}
```

Speicherorganisation mit Betriebssystem

```
int a;
int b = 1;
const int c = 2;

void main(void) {
    static int s = 3;
    int x, y;
    char *p = malloc(100);
}
```

Organization of Memory with an OS

```
int a;
int b = 1;
const int c = 2;

void main(void) {
   static int s = 3;
   int x, y;
   char *p = malloc(100);
}
```


Skriptum _handout

Speicherorganisation mit Betriebssystem (Forts.)

I/O-Devices

Stack
Heap

.bss ← init — RAM (read/write)

.data

.rodata — RAM (read-only)

.text

Symbol-Tabelle
.data
.rodata
.text
Header

...

compile
link

boot

Quell-Programm
(Datei)

ELF-Binary
(Datei)

Stack
Heap

.bss ← init — RAM (read/write)

.data

.rodata — RAM (read-only)

.text

Symbol-Tabelle
.data
.rodata
.text
Header

...

compile
link

exec

Quell-Programm
(Datei)

ELF-Binary
(Datei)

PC
(Speicher)

Skriptum_handout

Organization of Memory with an OS (Forts.)

I/O-Devices

Stack
Heap
.bss ← init
.data
.rodata
.text

RAM (read/write)
RAM (read-only)

...

Symbol table
.data
.rodata
.text
Header

compile
link

boot

Source program
(file)

ELF binary
(file)

Stack
Heap
.bss ← init
.data
.rodata
.text

RAM (read/write)
RAM (read-only)

...

Symbol table
.data
.rodata
.text
Header

compile
link

exec

Source program
(file)

ELF binary
(file)

PC
(memory)

Skriptum_handout

Speicherorganisation mit Betriebssystem (Forts.)

Betriebssystem

I/O-Devices
Stack
Heap
.bss
.data
.rodata
.text

laufende Anwendung

Stack
Heap
.bss
.data
.rodata
.text

MMU

virtueller Speicher

Betriebssystem

I/O-Devices
Stack
Heap
.bss
.data
.rodata
.text

Anwendung 1

Stack
Heap
.bss
.data
.rodata
.text

Anwendung 2

Stack
Heap
.bss
.data
.rodata
.text

...

Anwendung N

Stack
Heap
.bss
.data
.rodata
.text

physikalischer Speicher

Skriptum_handout

Organization of Memory with an OS (Forts.)

Skriptum _handout

Überblick: Teil E Speicher

33 Dynamische Speicherallokation

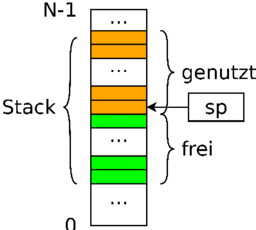
34 Speicherorganisation

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)

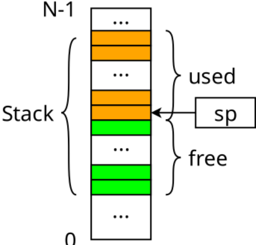
  => `sp` zeigt damit immer auf den Anfang des genutzten Teil des Stacks



Dynamic Allocation of Memory – Stack

- Local variables, function parameters, and return addresses are organized by the compiler on the stack
- The stack is part of the main memory
- The processor register `sp` **stack pointer** always points to the last allocated memory on stack
- The stack "grows" "top to bottom"

$\Rightarrow$ `sp` always points to the start of the stack's used part



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;
}
```

Inhalt	Adresse
...	
	2000
?	1996
?	1992
?	1988
?	1984
?	1980
?	1976
?	1972
?	1968
?	1964
?	1960
?	1956
?	1952
?	1948
?	1944
?	1940
?	1936
?	1932
?	1928
?	1924
...	

sp → 2000

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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;
}
```

Content	Address
...	
	2000
?	1996
?	1992
?	1988
?	1984
?	1980
?	1976
?	1972
?	1968
?	1964
?	1960
?	1956
?	1952
?	1948
?	1944
?	1940
?	1936
?	1932
?	1928
?	1924
...	

sp → 2000

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

	...	
		2000
c	?	1996
b	?	1992
a	?	1988 ← sp
	?	1984
	?	1980
	?	1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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;
}
```

Creating a, b, c

	...	
		2000
c	?	1996
b	?	1992
a	?	1988
	?	1984
	?	1980
	?	1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

35-Speicher-Stack: 2026-04-13

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;
}
```

Variable	Inhalt	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988 ← sp
	?	1984
	?	1980
	?	1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

Schreiben von a, b

Dynamic Allocation of Memory – 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;
}
```

Writing of a, b

	...	
		2000
c	?	1996
b	20	1992
a	10	1988 ← sp
	?	1984
	?	1980
	?	1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

35-Speicher-Stack: 2026-04-13

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;
}
```

Variable	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	21	1984
	10	1980
	?	1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Berechnen der Parameter

Dynamic Allocation of Memory – 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;
}
```

Calculating parameters

35-Speicher-Stack: 2026-04-13

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

	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	21	1984
	10	1980
		1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Dynamic Allocation of Memory – 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;
}
```

		Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	21	1984
	10	1980
		1976
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Storing the return address

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

Variable	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	10	1980
		1976 ← sp
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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

Variable	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	10	1980
		1976 ← sp
	?	1972
	?	1968
	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

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]

	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	10	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Dynamic Allocation of Memory – 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;
}
```

	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	10	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Creating 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

		Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp → 1964

Dynamic Allocation of Memory – 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;
}
```

Incrementing x

	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	?	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp → 1964

35-Speicher-Stack: 2026-04-13

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

	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Dynamic Allocation of Memory – 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;
}
```

Calculation of the parameter

Name	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
	?	1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

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

Variable	Inhalt	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
		1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp → 1956

Dynamic Allocation of Memory – 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;
}
```

	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
		1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Storing the return address

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

Variable	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp → 1956

Dynamic Allocation of Memory – 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

Variable	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp → 1956

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

Variable	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
m	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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;
}
```

Variable	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
m	?	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Creating 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

Variable	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
m	100	1952 ← sp
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

Dynamic Allocation of Memory – 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;
}
```

Writing of m

Variable	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
m	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp → 1952

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

Dynamic Allocation of Memory – 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;
}
```

	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
z	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Removing 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

	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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;
}
```

Return

	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

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;
}
```

Variable	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Entfernen des Parameters

Dynamic Allocation of Memory – 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;
}
```

	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
i[2]	?	1972
i[1]	?	1968
i[0]	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Removing the 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;
}
```

	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Entfernen von i[0]...i[2]

Dynamic Allocation of Memory – 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;
}
```

Removing i[0]...i[2]

Var	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

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

		Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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;
}
```

Name	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
y	21	1984
x	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Return

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

	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	21	1984
	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

35-Speicher-Stack: 2026-04-13

Dynamic Allocation of Memory – 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;
}
```

Removing the parameters

	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	21	1984
	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

35-Speicher-Stack: 2026-04-13

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

	Wert	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	10	1984
	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Dynamic Allocation of Memory – 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;
}
```

Calculating the parameter

	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	10	1984
	11	1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

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;
}
```

Variable	Inhalt	Adresse
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	10	1984
		1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Speichern der Rückkehradresse

Dynamic Allocation of Memory – 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;
}
```

	Value	Address
	...	
		2000
c	?	1996
b	20	1992
a	10	1988
	10	1984
		1980
		1976
	?	1972
	?	1968
	?	1964
	11	1960
		1956
	100	1952
	?	1948
	?	1944
	?	1940
	?	1936
	?	1932
	?	1928
	?	1924
	...	

sp

Storing the return address

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

		Wert	Adresse
		...	
			2000
	c	?	1996
z3	b	20	1992
z2	a	10	1988
z1		10	1984
			1980
			1976
		?	1972
		?	1968
		?	1964
		11	1960
			1956
		100	1952
		?	1948
		?	1944
		?	1940
		?	1936
		?	1932
		?	1928
		?	1924
		...	

sp → 1980

Dynamic Allocation of Memory – 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

		Value	Address
		...	
			2000
	c	?	1996
z3	b	20	1992
z2	a	10	1988
z1		10	1984
			1980
			1976
		?	1972
		?	1968
		?	1964
		11	1960
			1956
		100	1952
		?	1948
		?	1944
		?	1940
		?	1936
		?	1932
		?	1928
		?	1924
		...	

sp → 1980

35-Speicher-Stack: 2026-04-13