



EINFÜHRUNG

RECHNERSTRUKTUREN & EMBEDDED SYSTEMS

MATTHIAS JANETSCHEK
SS 2025

Inhaltsverzeichnis



Organisatorisches

Einführung

Beispielanwendung

Organisatorisches

Einführung

Beispielanwendung

Pflichttermine



1. Webinartermin:	Mi, 28.05.2025	18:00 – 20:30
2. Webinartermin:	Mi, 04.06.2025	18:00 – 20:30
3. Webinartermin:	Mi, 11.06.2025	18:00 – 20:30
4. Webinartermin:	Mi, 18.06.2025	18:00 – 20:30
5. Webinartermin:	Mi, 25.06.2025	18:00 – 20:30
6. Webinartermin:	Mi, 02.07.2025	18:00 – 20:30
Präsenztag:	Fr, 11.07.2025	09:00 – 13:45
Klausur:	Fr, 11.07.2025	14:30 – 15:30

- Die Übungszettel werden immer am Mittwoch kurz vor dem Webinar freigeschalten.
- Die **Soft-Deadline** ist immer **am Tag vor dem nächsten Webinar um 23:55**.
- Die **Hard-Deadline** ist immer kurz vor **Beginn der Nachbesprechung der Hausübung**.

Hausübungen



Die Übungszettel werden über JupyterHub veröffentlicht:

- Sakai ⇒ Kursseite “Rechnerstrukturen & Embedded Systems” ⇒ JupyterHub in der linken Sidebar.
- In Jupyter den Tab “Assignments” aktivieren und den neuesten Übungszettel herunterladen.
- In Jupyter sollte dann ein neuer Ordner mit dem Namen des Assignments auftauchen.

Nach Ausfüllen des Übungszettel, muss dieser über Jupyter abgeschickt werden:

- Wenn gefordert, Quellcode-Dateien in den Übungszettel-Ordner hochladen.
- Dann wieder den Tab “Assignments” aktivieren und auf “Submit” klicken.

Hausübungen

Benotung



- Für Hausübungen gilt das “Best Effort” Prinzip.
- Hausübungen sollen dazu dienen, neue Konzepte auszuprobieren und Praxis zu sammeln.
 - Dass dabei Fehler gemacht werden, ist ganz normal
 - Durch Fehler lernt man!
 - Daher immer danach die Musterlösung anschauen und mit der eigenen Lösung vergleichen.
- Daher wird die Mühe belohnt, die Hausübungen zu machen und nicht die Fehlerfreiheit!
 - Solange das Programm kompiliert und halbwegs das richtige tut, gilt die Hausübung als erfolgreich abgeschlossen.
 - Automatische Tests dienen dazu, Fehler und mögliche Verbesserungen aufzuzeigen, nicht zur Benotung.

Hausübungen

Benotung



Um die volle Punktzahl zu erreichen, müssen folgende Punkte erfüllt werden:

- 1 Die entsprechende Python Variable muss auf `True` gesetzt sein.
 - Wenn sie auf `False` gesetzt ist, dann gibt es **gar keine Punkte!**
 - Auf richtigen Datentypen achten!
 - Ich habe nicht die Zeit, zu überprüfen, ob sie richtig gesetzt wurde.

```
# Weisen Sie der untenstehenden Variable den Wert True zu, sobald Sie die Aufgabe erfolgreich erledigt haben!
# Datentyp: bool
exercise_1_1_solved = True
```

- 2 Die Namensgebung der Abgaben wurde laut Angabe umgesetzt.
 - Achtet darauf, dass ihr alle Dateien richtig benannt habt!
 - Ich habe nicht die Zeit, auf falsch benannte Dateien zu überprüfen!

Um die volle Punktzahl zu erreichen, müssen folgende Punkte erfüllt werden:

- ① Das Programm muss **ohne Fehler und ohne Warnings** kompilieren.
- ② Das Programm muss **ohne Fehler und Abstürze** ausgeführt werden können.
- ③ Das Programm darf die Lösung **nicht vortäuschen**.
 - In diesem Fall gibt es auch **gar keine Punkte!**
 - z.B. mit `printf()` eine schon vorberechnete Folge von Primzahlen ausgeben statt tatsächlich Primzahlen zu berechnen.

Nachbesprechung der Hausübungen



Keine Anwesenheitspflicht!

Nachbesprechung 1.Hausübung:	Do, 05.06.2025	ab 16:00
Nachbesprechung 2.Hausübung:	Do, 12.06.2025	ab 16:00
Nachbesprechung 3.Hausübung:	Do, 19.06.2025	ab 16:00
Nachbesprechung 4.Hausübung:	Do, 26.06.2025	ab 16:00
Nachbesprechung 5.Hausübung:	Fr, 04.07.2025	ab 16:00
Nachbesprechung 6.Hausübung:	Mi, 09.07.2025	ab 16:00

- Für Fragen, die nicht im Webinar bzw. Forum geklärt werden konnten, bieten wir auch “So wird’s gemacht”-Sessions an.
- Bedarf bitte im Forum anmelden (mit **konkreten Fragen**).
- Termine nach Vereinbarung (meist direkt nach der Nachbesprechung der Hausübung).

- Zusammensetzung Gesamtnote:
 - Hausübung (24%)
 - 6 Übungszettel
 - Jeder Übungszettel zählt gleich viel (4%)
 - Schlussklausur (76%)
 - Multiple Choice und offene Fragen
- Für eine positive Note müssen insgesamt mind. 60% erreicht werden.
- Weitere Informationen siehe Prüfungsordnung.

- Bei Fragen zum Vorlesungsstoff bitte das Sakai-Forum benutzen.
 - Alle sind herzlich eingeladen die Fragen zu beantworten
- Bei Fragen direkt an mich bitte das Message-Tool in Sakai benutzen.
- Bei Kritik und Anregungen bitte das entsprechende Sakai-Unterforum benutzen.
 - Postings dort sind anonym!
- Bei Problemen mit / Fragen zu JupyterHub bitte das entsprechende Sakai-Unterforum benutzen.
- Bei Problemen mit Adobe Connect / BigBlueButton im entsprechenden Sakai-Unterforum auf weitere Anweisungen warten.

Diese Lehrveranstaltung passiert hauptsächlich auf folgender Literatur:

- Andrew S. Tanenbaum und Todd Austin, **Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner**, 6., aktualisierte Auflage 2014, Pearson Verlag
- Karsten Berns, Bernd Schürmann und Mario Trapp, **Eingebettete Systeme: Systemgrundlagen und Entwicklung eingebetteter Software**, 1. Auflage 2010, Springer Verlag

Noch Fragen?

Organisatorisches

Einführung

Beispielanwendung

Zwei verschiedene Themengebiete:

- Rechnerstrukturen
 - Wie funktioniert eine CPU
 - Technische Realisierung
 - Assemblerprogrammierung
- Embedded Systems
 - Was sind "Embedded Systems"
 - Mikrocontroller
 - Entwicklung von Software für Mikrocontroller

Was werden wir in dieser LV machen:

- Wir werden uns hauptsächlich mit Mikrocontrollern beschäftigen.
- Theorie
 - Bestandteile eines Mikrocontrollers
 - Wie funktioniert eine CPU (nur sehr oberflächlich)
 - Assemblerprogrammierung (nur sehr oberflächlich)
- Praxis
 - Praktische Softwareentwicklung auf einem Mikrocontroller

Einführung

Was sind "Embedded Systems"?



Definition (Eingebettetes System, laut Wikipedia)

Ein **eingebettetes System** (auch englisch **embedded system**) ist ein elektronischer Rechner oder auch Computer, der in **einen technischen Kontext eingebunden (eingebettet) ist**.

Dabei übernimmt der Rechner entweder **Überwachungs-, Steuerungs- oder Regelfunktionen** oder ist für eine Form der **Daten- bzw. Signalverarbeitung** zuständig, beispielsweise beim Ver- bzw. Entschlüsseln, Codieren bzw. Decodieren oder Filtern.

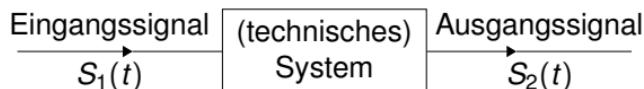
Einführung

Eingebettete Systeme sind “technische Systeme”



Definition (technisches System)

Laut Systemtheorie ist ein (technisches) System ein (mathematisches) Modell, welches die **Abbildung von Eingangssignalen auf Ausgangssignalen** (auch **Transformationsfunktion** genannt) beschreibt.



Einführung

Beispiele für “Embedded Systems”



- Fahrassistsysteme in Autos



- Haushaltsgeräte



- Medizinische Geräte



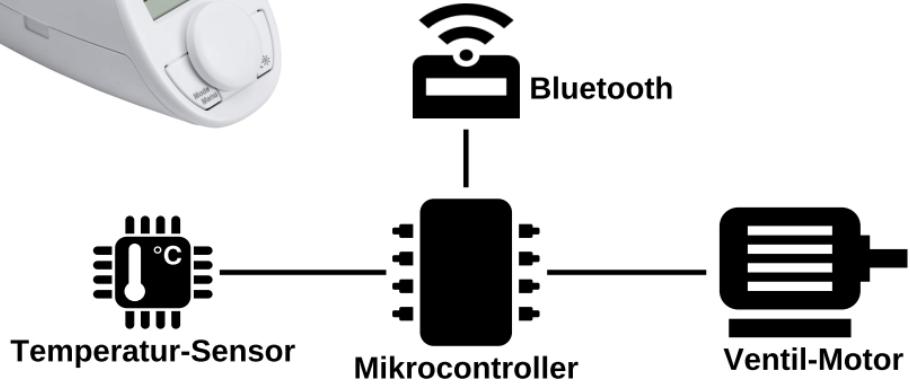
- Beleuchtung



- ...

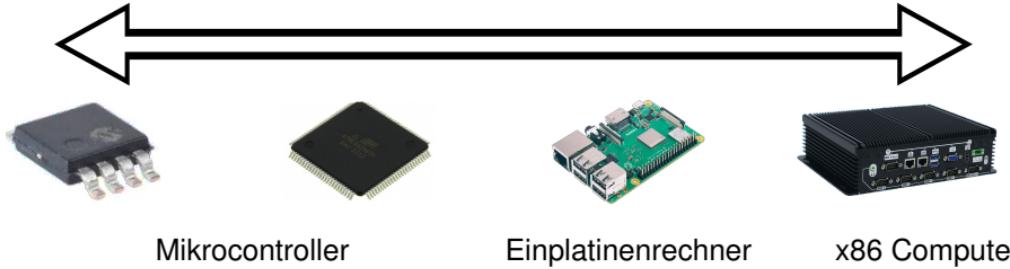
Beispiele für “Embedded Systems”

Beispiel: Smarter Heizkörperthermostat



Einführung

Welche Computer werden eingesetzt?



- Simple Rechenwerke
 - Knappe Ressourcen
 - i.d.R. ohne Betriebssysteme
 - Sehr energieeffizient
-
- Vollwertige Rechenwerke
 - Ausreichend Ressourcen
 - i.d.R. mit Betriebssysteme
 - Weniger energieeffizient

Einführung

Was ist ein Mikrocontroller?

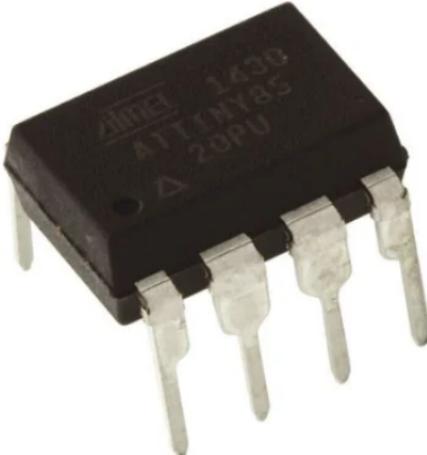


Mikrocontroller sind einfache Ein-Chip-Computersysteme

- Auf dem Chip sind alle für den Betrieb wesentlichen Bauteile (CPU, RAM, Speicher, Peripherie, ...) enthalten.
- Zum Betrieb muss man oft nur noch den Chip mit Strom versorgen.
- Solche Systeme nennt man auch **System-on-a-Chip (SoC)**
 - Vorsicht: Nicht jedes SoC ist auch ein Mikrocontroller!
 - Man redet auch von SoC, wenn mehrere, aber nicht alle Komponenten auf einem Chip zusammengefasst werden (e.g. bei Einplatinenrechnern).
 - Grenze ist oft fließend!
- Das Gegenstück ist der **Mikroprozessor** (Chip enthält **nur die CPU**).
 - Auch hier ist die Grenze oft fließend!

Was ist ein Mikrocontroller?

Beispiel: Atmel AVR ATtiny 85



- Ist ein vollwertiger Computer
- Enthält alle benötigten Bauteile
 - 8-bit CPU (bis zu 20 MHz)
 - Arbeitsspeicher (512 Byte SRAM)
 - Programmspeicher (8KB Flash)
 - I/O-Hardware
- Sobald er mit Strom versorgt wird,
fängt er zu arbeiten an.

Vorteile von Mikrocontroller im Vergleich zu dedizierten elektrischen Schaltungen:

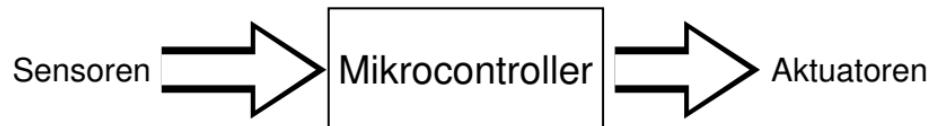
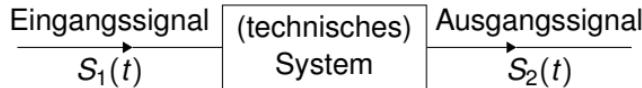
- **Billiger** (Economies of scale: Dedizierte Schaltungen für bestimmten Zweck haben weit geringere Stückzahlen als universell einsetzbare Mikrocontroller)
- **Flexibler** (In Software können Features einfacher geändert werden)
- **Nachrüstbar** (Durch ein Software-Update können neue Features nachträglich hinzugefügt werden)
- **Wartbarer** (Durch ein Software-Update können Fehler einfach behoben werden)

Mikrocontroller sind “**general-purpose**” (**Mehrzwecks-**) Bausteine:

- Müssen in vielen verschiedenen Szenarios verwendbar sein.
- Müssen vielen unterschiedlichen Anforderungen gerecht werden.
- Bestehen aus hochkonfigurierbaren Bausteinen, die für unterschiedliche Zwecke verwendet werden können.

Einführung

Mikrocontroller im Kontext von eingebetteten Systemen



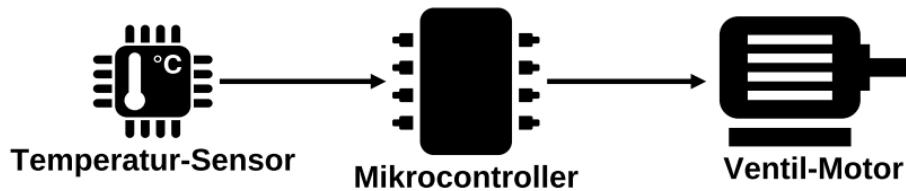
Eingangssignal: Messwerte der Sensoren

Ausgangssignal: Stellwerte der Aktuatoren

Transformationsfunktion: Programm, das auf den Mikrocontroller läuft

Mikrocontroller und eingebettete Systeme

Beispiel: Smarter Heizkörperthermostat



Einführung

Olimex EduArdu Board



Für den praktischen Teil verwenden wir das EduArdu Board von Olimex:

- Arduino-kompatibles Board speziell für die Lehre
- Atmel AVR ATmega32u4 Mikrocontroller

Verbaute Sensoren:

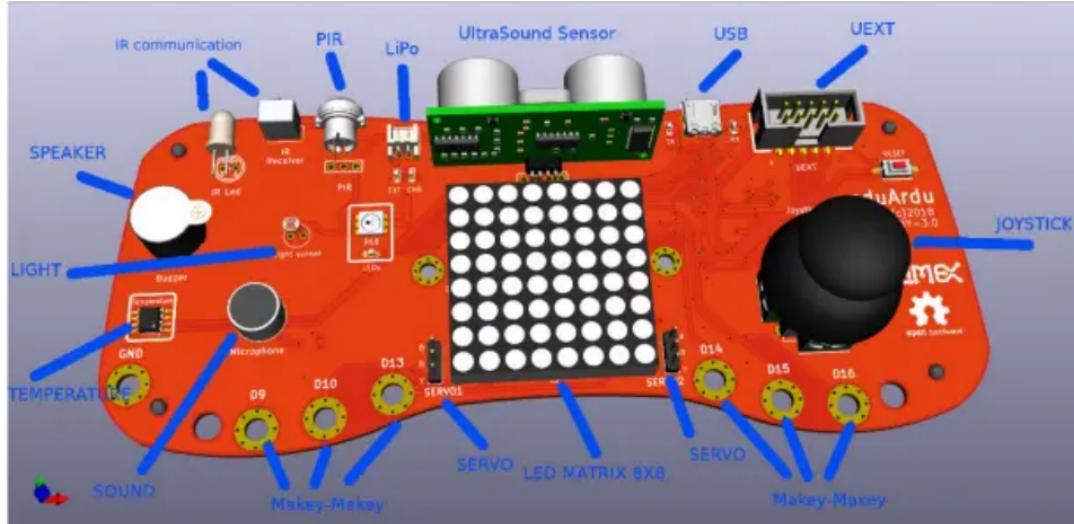
- Temperatur
- Helligkeit
- Mikrofon
- Infrarot-Empfänger
- Bewegung (PIR)
- Abstand (Ultraschall)
- Joystick

Verbaute Aktuatoren:

- Buzzer
- LED
- RGB-LED
- 8x8 LED-Matrix
- 2 Servo-Anschlüsse
- Infrarot-Sender

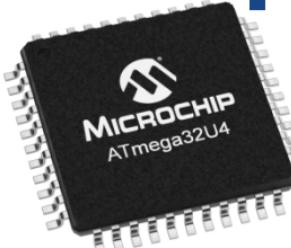
Einführung

Olimex EduArdu Board



Atmel AVR ATmega32u4

- 8-bit RISC CPU mit bis zu 16MHz
- 2.5 KB SRAM Arbeitsspeicher
- 32 KB Flash Programmspeicher
- 1 KB EEPROM (für persistente Daten)
- 26 konfigurierbare I/O Pins
- Dedizierte Unterstützung für USB 2.0
- Kommunikations-Peripherie:
 - Universal Asynchronous Receiver Transmitter (UART)
 - Serial Peripheral Interface (SPI)
 - Inter-Integrated Circuit (I^2C), auch Two-Wire Interface (TWI) genannt
- Sonstige Peripherie:
 - 1x 8-bit Timer, 1x 10-bit Timer, 2x 16-bit Timer
 - 10-bit A/D-Umwandler mit 12 Kanälen
- Betriebsspannung: 2.7 - 5.5 Volt



- **Vorsicht: Ganzzahlen haben nicht die gewohnte Größe!**
 - Der C-Standard gibt nur Mindestgrößen vor!
 - z.B. bei AVR-Mikrocontrollern: `sizeof(int) = 2 Byte`
- Damit man sich die entsprechenden Größen nicht ständig merken muss, wird i.d.R. das Headerfile `stdint.h` verwendet.
- Definiert entsprechende `typedefs`, die die Größe des Datentyps in Bit im Namen kodieren.
 - z.B. `uint8_t` - vorzeichenlose 8-bit Ganzzahl
 - z.B. `int16_t` - vorzeichenbehaftete 16-bit Ganzzahl
 - z.B. `uint32_t` - vorzeichenlose 32-bit Ganzzahl

Organisatorisches

Einführung

Beispielanwendung

Beispielanwendung

Pseudocode



Wir wollen ein Programm entwickeln, dass eine LED abhängig vom Umgebungslicht ansteuert:

```
while true do
    helligkeit = readLightSensor()
    if helligkeit < schwellwert then
        switchLed(true)                      # Helligkeit unter Schwellwert
        # Schalte LED an
    else
        switchLed(false)                     # Helligkeit über Schwellwert
        # Schalte LED aus
    end if
end while
```

Beispielanwendung

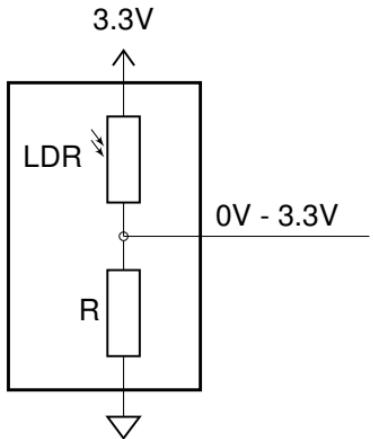
Helligkeit auslesen



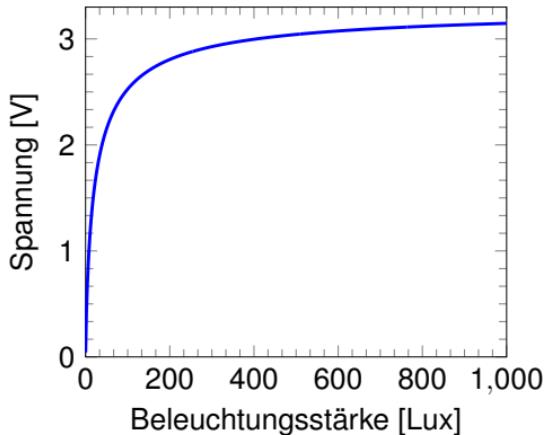
- Der Helligkeitssensor auf dem EduArdu-Board ist ein sogenannter **Photowiderstand** (englisch: **Light Dependent Resistor, LDR**)
 - Je mehr Licht auf den Sensor fällt, desto kleiner wird sein Widerstand.
 - Ein Mikrocontroller kann i.d.R. nur Spannungen messen.
 - Der Sensor kann nicht direkt mit dem Mikrocontroller verbunden werden.
 - Es braucht eine elektrische Schaltung, die die Widerstandsänderung in eine Spannungsänderung umwandelt.
- ⇒ Mit einem zweiten Widerstand kann man einen lichtabhängigen Spannungsteiler bauen.
- Je mehr Licht auf den Sensor fällt, desto größer ist die Spannung.
 - Wenn wir im Folgenden von einem Lichtsensor reden, dann meinen wir diesen Spannungsteiler.

Helligkeit auslesen

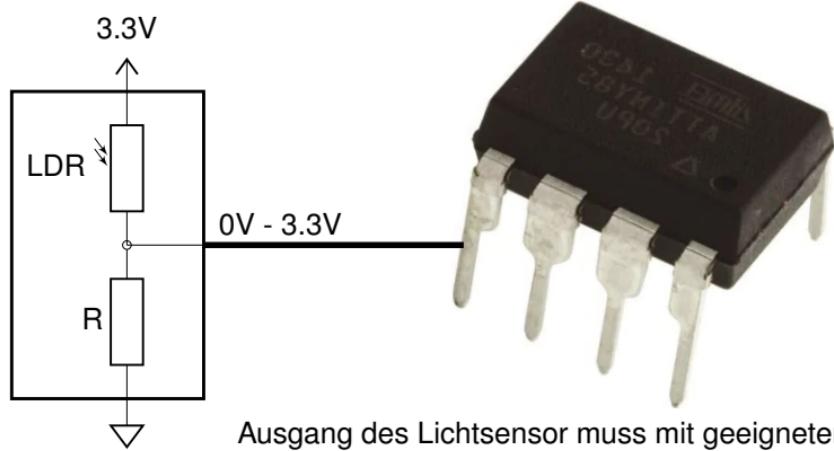
Lichtsensor



**Transformationsfunktion
Lichtsensor ($R=10\text{k}\Omega$)**



Helligkeit auslesen Verbindung mit Mikrocontroller



Ausgang des Lichtsensor muss mit geeignetem
Mikrocontroller-Pin verbunden werden.

Helligkeit auslesen

Spannung messen



Wie misst ein Mikrocontroller Spannungen?

- Digitaleingang:
 - Kann nur zwei Zustände unterscheiden: **High** (hohe Spannung) und **Low** (niedrige bzw. keine Spannung).
- Analogeingang:
 - Ein **Analog/Digital-Wandler** (englisch: **Analog-Digital Converter, ADC**) wandelt die anliegende Spannung in einen numerischen Wert um.
 - Numerischer Wert gibt Verhältnis von anliegender Spannung zu einer Referenzspannung an.
 - z.B.: 10-bit ADC mit 3.3V Referenzspannung:
 - 10-bit \Rightarrow Wertebereich: 0 -1023
 - 0V \rightarrow 0, 1.65V \rightarrow 511, 3.3V \rightarrow 1023
 - Generelle Formel: $ADC = V_{IN} \cdot 1023 / V_{REF}$

Helligkeit auslesen

ADC konfigurieren



- Bevor der ADC verwendet werden kann, muss er konfiguriert werden.
- Mikrocontrollerkomponenten werden i.d.R. über sogenannte [I/O Register](#) konfiguriert.
 - Register sind spezielle Speicherstellen.
- Wie können Daten in I/O Registern gespeichert bzw. ausgelesen werden?
 - 1. Möglichkeit: Unter Verwendung von speziellen Maschinenbefehlen
 - 2. Möglichkeit: Werden in den Hauptspeicher gemappt ([Memory Mapped I/O](#))
 - Meistens wird Memory Mapped I/O verwendet.
 - Jedem Register wird eine fixe Hauptspeicheradresse zugewiesen.

ADC konfigurieren

I/O Register



- Ein I/O Register kann man sich als eine Art Schalttafel vorstellen.
 - Ein einzelnes Bit repräsentiert einen Schalter mit 2 Stellungen.
 - Bitgruppen repräsentieren Schalter mit mehr als 2 Stellungen.

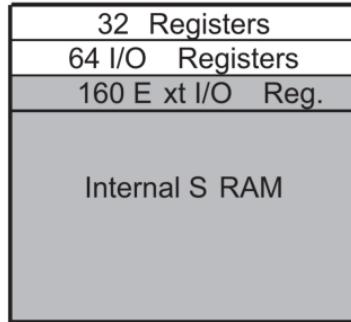


ADC konfigurieren

Memory Mapped I/O



Data Memory



\$0000 - \$001F

\$0020 - \$005F

\$0060 - \$00FF

ISRAM start : \$0100

ISRAM end : \$05FF / \$0AFF



ADC konfigurieren

Memory Mapped I/O



- Um ein I/O Register zu lesen wird einfach eine bestimmte Hauptspeicheradresse gelesen.
- Um ein I/O Register zu schreiben wird einfach ein Wert an eine bestimmte Hauptspeicheradresse geschrieben.

Lesen/Schreiben des ADMUX-Registers mittels Pointer

```
// I/O-Register ADMUX hat die Hauptspeicheradresse 0x7C
uint8_t* register_ptr = (uint8_t*)0x7C;

// Speichern des Registerwerts in einer Variable
uint8_t register_value = *register_ptr;

// Schreiben eines neuen Registerwerts
*register_ptr = 0xFF;
```

- Das Verwenden eines Pointers verursacht aber Overhead, darum wird i.d.R. direkt die Hauptspeicheradresse angegeben.

Direktes Lesen/Schreiben des ADMUX-Registers

```
// Speichern des Registerwerts in einer Variable  
uint8_t register_value = (*(volatile uint8_t*)0x7C);  
  
// Schreiben eines neuen Registerwerts  
(*(volatile uint8_t*)0x7C) = 0xFF;
```

- `volatile` verhindert, dass der Compiler unser Programm kaputt “optimiert”.

- Normalerweise müsst ihr nicht selber die Hauptspeicheradresse angeben, sondern ihr bindet ein Headerfile ein, in dem entsprechende Makros definiert sind.

Direktes Lesen/Schreiben des ADMUX-Registers

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

// Speichern des Registerwerts in einer Variable
uint8_t register_value = ADMUX;

// Schreiben eines neuen Registerwerts
ADMUX = 0xFF;
```

- Der ADC wird über drei 8-bit I/O Register konfiguriert:

ADMUX: ADC Multiplexer Selection Register

ADCSRA: ADC Control and Status Register A

ADCSR B: ADC Control and Status Register B

- Das Ergebnis der Umwandlung wird über zwei 8-bit I/O Register (die zusammen ein 16-bit Register bilden) ausgelesen:

ADCH: ADC Data Register - High Byte

ADCL: ADC Data Register - Low Byte

ADC Multiplexer Selection Register - ADMUX:

Bit	7	6	5	4	3	2	1	0
Read/Write	REFS1 R/W	REFS0 R/W	ADLAR R/W	MUX4 R/W	MUX3 R/W	MUX2 R/W	MUX1 R/W	MUX0 R/W
Initial Value	0	0	0	0	0	0	0	0

Bit 7:6 Reference Selection Bits (Auswahl der Referenzspannung)

Bit 5 ADC Left Adjust Result (Wie wird das Ergebnis in ADCH/ADCL gespeichert)

Bit 4:0 Analog Channel Selection Bits (Auswahl des Analogeingangs)

Helligkeit auslesen

ADC konfigurieren



ADC Control and Status Register A – ADCSRA:

Bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bit 7 ADC Enable (Schaltet den ADC ein)

Bit 6 ADC Start Conversion (Startet die Umwandlung)

Bit 5 ADC Auto Trigger Enable (Automatischer Start der Umwandlung)

Bit 4 ADC Interrupt Flag (Löst die Interruptroutine aus)

Bit 3 ADC Interrupt Enable (Schaltet die Interruptroutine ein)

Bit 2:0 ADC Prescaler Select Bits (Teiler der ADC Frequenz)

Helligkeit auslesen

ADC konfigurieren



ADC Control and Status Register B – ADCSRB:

Bit	7	6	5	4	3	2	1	0
	ADHSM	ACME	MUX5	–	ADTS3	ADTS2	ADTS1	ADTS0
Read/Write	R/W	R/W	R	R	R	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bit 7 ADC High Speed Mode (Schaltet den High Speed Modus ein)

Bit 6 Analog Comparator Multiplexer Enable (Auswahl des Kanals für den Analog Comparator)

Bit 5 Analog Channel Additional Selection Bits (Erweitert die MUX4:0 Bits im ADMUX Register)

Bit 4 Nicht verwendet

Bit 3:0 ADC Auto Trigger Source (Auslöser der automatischen Umwandlung)

ADC konfigurieren

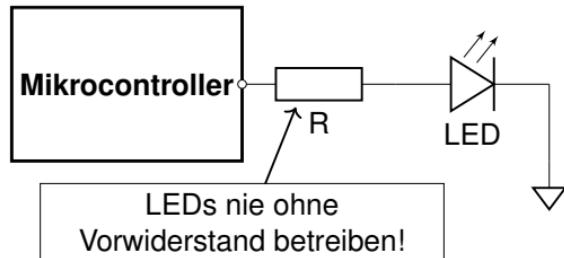
Wie setzt/löscht man einzelne Bits in einem Register?



- Für das Setzen bzw. Löschen einzelner Bits verwendet man die sog. **bitweisen Operatoren**
 - Bitweises Und: &
 - Bitweises Oder: |
- Im Gegensatz zu den logischen Operatoren && und || arbeiten diese auf den einzelnen Bits.
- **Näheres ist im Selbststudium zu erarbeiten.**

Beispielanwendung LED ein-/ausschalten

- Die LED ist mit einem digitalen Ausgang verbunden:
 - Ausgang ist **HIGH** (3.3V) ⇒ Strom fließt, d.h. LED leuchtet
 - Ausgang ist **LOW** (0V) ⇒ kein Strom fließt, d.h. LED leuchtet nicht



- Achtung: Digitale Ausgänge nie mit mehr als 10mA-20mA belasten!
 - Bei größeren Strömen muss Transistor oder Relay verwendet werden.

Fast alle Mikrocontroller-Pins lassen sich als digitalen Eingang oder Ausgang verwenden.

- Werden auch **General Purpose Input/Output Pins (GPIO-Pins)** genannt.
- Viele Pins haben auch Mehrfachfunktionen.
 - z.B.: Für den I²C Bus sind bestimmte Pins vorgesehen.
 - Wird I²C nicht verwendet, dann können diese Pins als externe Interrupt-Quelle verwendet werden.
 - Oder sie können als digitale Ein- oder Ausgänge verwendet werden.

LED ein-/ausschalten

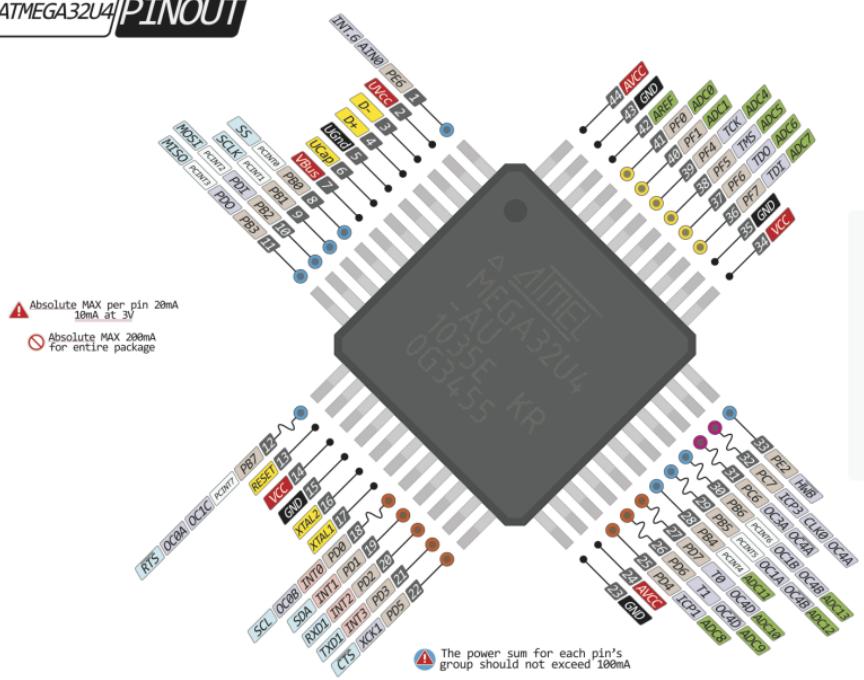
General Purpose Input/Output Pins



- Mehrere GPIO-Pins (bis zu 8) werden zu **Ports** zusammengefasst.
 - Ports werden durch Buchstaben bezeichnet (z.B. Port A, Port B, ...)
- Die Konfiguration der Pins erfolgt wieder über Register.
 - Alle Pins eines Ports werden über dasselbe Register konfiguriert.
 - Pro Port gibt es 3 Register (x steht für Portbuchstabe):
 - DDR_x** Pin-Richtung (Eingang oder Ausgang)
 - PORT_x** Pin-Status (bei Ausgängen) bzw. Pullup (bei Eingängen)
 - PIN_x** Pin-Status (bei Eingängen)
- Die möglichen Funktionen der einzelnen Pins sind im **Pinout** niedergeschrieben.

ATMEGA32U4 PINOUT

®



- Power
- GND
- Serial Pin
- Analog Pin
- Control
- Pin Change Int
- Physical Pin
- Port Pin
- Pin Function
- Ext Interrupt
- ~PWM Pin
- Port Power

LED ein-/ausschalten General Purpose Input/Output Pins



Siehe Hands-On Beispiel für konkreten Code

(Quellcode auf Sakai verfügbar)

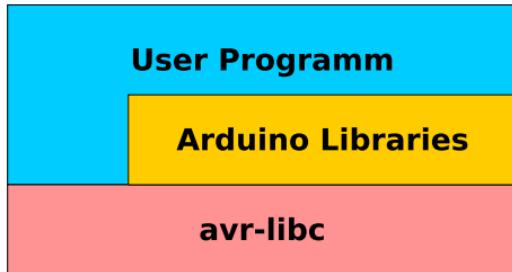
Arduino

Für was steht “Arduino”?



Arduino war primär der Name einer Firma (Arduino LLC)

- Baut und vertreibt Mikrocontroller-Boards
 - Schaltpläne sind unter Open-Source Lizenz verfügbar
 - Daher viele Klonen von anderen Firmen erhältlich
 - Richten sich vor allem an Programmieranfänger:innen und Bastler:innen
- Zur Programmierung wurde ein Software-Ökosystem entwickelt
 - Arduino Programmbibliothek ist eine C++ high-level Bibliothek
 - Arduino IDE ist eine Entwicklungsumgebung speziell für Programmieranfänger:innen
 - Viele externe Bibliotheken für Sensoren oder ähnliches verfügbar
- Unter **Arduino** wird mittlerweile das gesamte Ökosystem aus Hardware und Software verstanden



avr-libc: Low-level Bibliothek, die eine Teilmenge der C Standardbibliothek und AVR-spezifische Funktionen implementiert.

Arduino: High-level Bibliothek, die auf der avr-libc aufbaut und eine einfach zu verwendende C++ API anbietet.

1 EINFÜHRUNG

▼ Bitweise Operatoren



Was sind bitweise Operatoren?

Bitweise Operatoren arbeiten **nicht auf ganzen Zahlenwerten**, sondern **direkt auf deren Bits**.

Besonders wichtig bei **Registerzugriffen, Maskierung, Flags setzen/löschen, Portsteuerung usw.**



Übersicht der wichtigsten bitweisen Operatoren

Operator	Bedeutung	Beispiel	Ergebnis (binär)
&	Bitweises UND	0b1100 & 0b1010	0b1000
	Bitweises ODER	0b1100 0b0101	0b1110
^	Bitweises XOR (Exklusiv)	0b1100 ^ 0b1010	0b0110
~	Bitweise Negation (NOT)	~0b1100	0b0011*
<<	Bitweise Linksverschiebung	0b0001 << 2	0b0100
>>	Bitweise Rechtsverschiebung	0b0100 >> 2	0b0001

- Vorzeichen beachten: ~ kehrt **alle Bits** um, auch das Vorzeichenbit!



Typische Anwendung: Registermanipulation

● Bit setzen

```
PORTB |= (1 << PB3); // Setzt Bit 3 in PORTB auf 1
```

● Bit löschen

```
PORTB &= ~(1 << PB3); // Setzt Bit 3 in PORTB auf 0
```

● Bit umschalten (toggle)

```
PORTB ^= (1 << PB3); // Kippt Bit 3 (0 → 1, 1 → 0)
```

● Bit prüfen

```
if (PINB & (1 << PB3)) {  
    // Bit 3 ist HIGH  
}
```



Bitmasken & Flags

Maske:

Eine Zahl, bei der nur bestimmte Bits auf 1 gesetzt sind, z. B.:

```
#define BITMASK_LED (1 << PB0)
```

Flags kombinieren:

```
uint8_t flags = (1 << FLAG_A) | (1 << FLAG_C); // A und C gesetzt
```



Tipps & Stolperfallen

- $\sim 0x00 = 0xFF$, aber $\sim 0x00U$ ergibt ggf. 0xFFFFFFFF (wegen Typ)
- Bei **signed** Typen können $>>$ und \sim unerwartete Ergebnisse liefern
- Nutze **uint8_t**, **uint16_t**, **uint32_t** aus stdint.h für Klarheit



Praktisches Beispiel: Taster abfragen und LED steuern

```
if (PIND & (1 << PD2)) {  
    PORTB |= (1 << PB0); // LED an  
} else {  
    PORTB &= ~(1 << PB0); // LED aus  
}
```



Quellen:

1. [Rheinwerk Openbook – Bitweise Operatoren in C](#)
2. [Arduino Playground – Bitmath](#)
3. [mikrocontroller.net – Bitmanipulation](#)
4. [Beuche – Bitoperationen in der Mikrocontrollertechnik](#)



Zusammenfassung

Ziel	Operator / Technik
Bit setzen	$ \text{value}$
Bit löschen	$\text{value} \&= \sim(1 << n)$

Ziel	Operator / Technik
Bit umschalten (toggle)	<code>value ^= (1 << n)</code>
Bit prüfen	<code>if (value & (1 << n))</code>
Bitmaske erstellen	<code>#define MASK (1 << BIT_POS)</code>
Bits verschieben	<code>value << n, value >> n</code>



MESSEN, STEUERN & REGELN

RECHNERSTRUKTUREN & EMBEDDED SYSTEMS

MATTHIAS JANETSCHEK
SS 2025

Inhaltsverzeichnis



Einführung

Signalverarbeitung

Messen

Steuern

Regeln

Inhaltsverzeichnis



Einführung

Signalverarbeitung

Messen

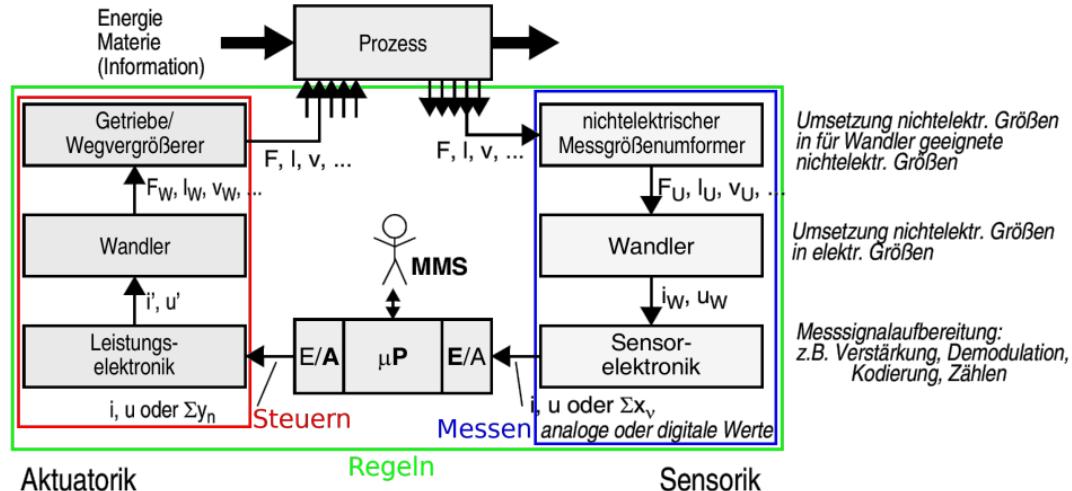
Steuern

Regeln

Zu den wichtigsten Aufgaben eines Mikrocontrollers gehören:

- **Messen**
 - Informationen über die “Welt” sammeln
- **Steuern**
 - Den Zustand der “Welt” ändern
- **Regeln**
 - Messen und Steuern in einem geordneten Prozess mit Rückkoppelung

Die Grundlage der Mess- und Steuertechnik bildet die [Signalverarbeitung](#).



Inhaltsverzeichnis



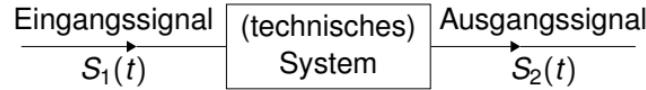
Einführung

Signalverarbeitung

Messen

Steuern

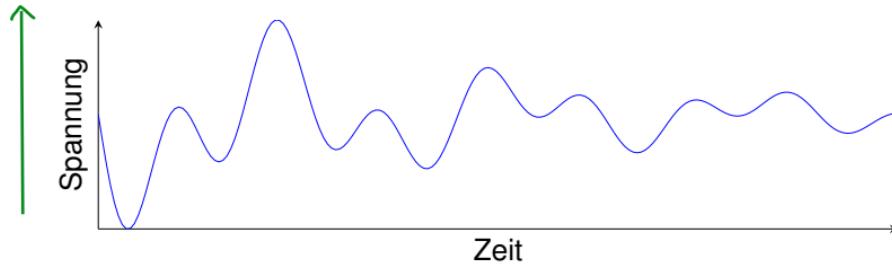
Regeln



Definition (Signal)

Unter einem **Signal** versteht man die **Darstellung einer Information** durch eine **zeitveränderliche physikalische**, insbesondere elektrische **Größe**, z. B. Strom, Spannung, Feldstärke. Die Information wird durch einen **Parameter dieser Größe kodiert**, z. B. Amplitude, Phase, Frequenz, Impulsdauer.

Beispiel: Signalverlauf Helligkeitssensor



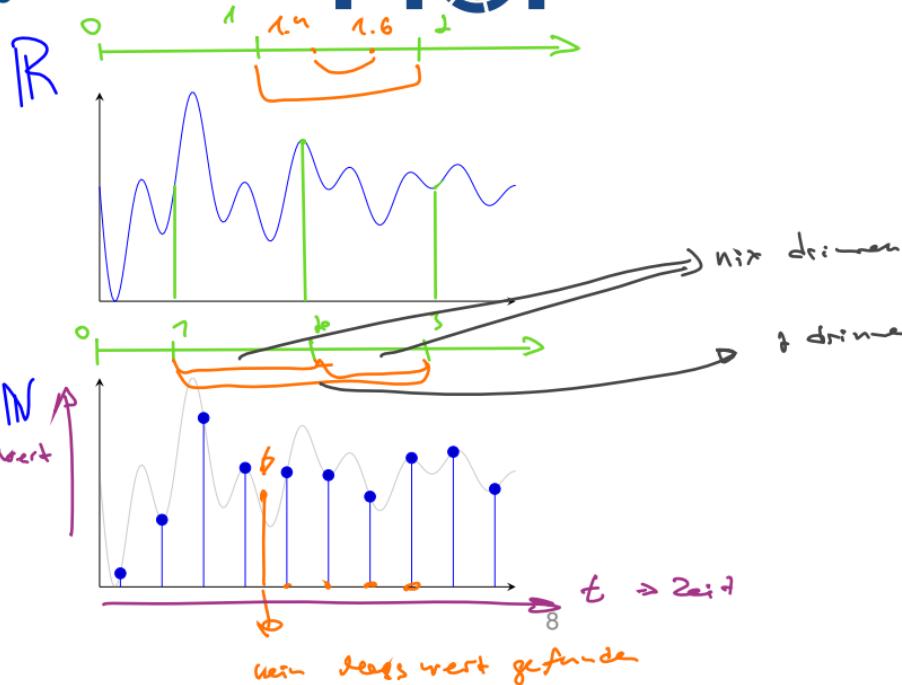
Signalverarbeitung

Kontinuierliche und diskrete Signale



Kontinuierliches Signal

- Zu jedem beliebigen Zeitpunkt ist ein Signalwert definiert.



Diskretes Signal

- Es ist nur für bestimmte, endlich viele Zeitpunkte ein Signalwert definiert.

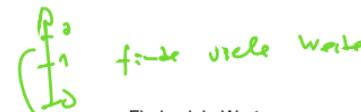
Die Attribute "kontinuierlich" und "diskret" können sowohl auf die Zeitachse, wie auch auf den Wertebereich angewendet werden.

zeitkontinuierlich: Zu jedem beliebigen Zeitpunkt ist ein Signalwert definiert.

zeitdiskret: Es ist nur für bestimmte, endlich viele Zeitpunkte ein Signalwert definiert.

wertkontinuierlich: Der Signalwert kann beliebige Werte annehmen (vgl. reelle Zahlen).

wertdiskret: Der Signalwert kann nur bestimmte, endlich viele Werte annehmen (vgl. natürliche Zahlen)

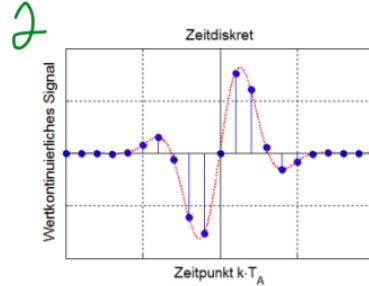


Finde viele Werte

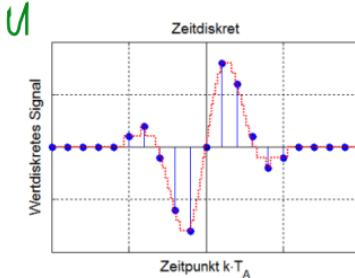
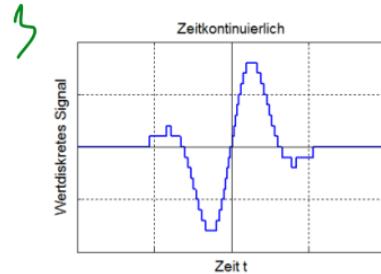


Signalverarbeitung

Kontinuierliche und diskrete Signale

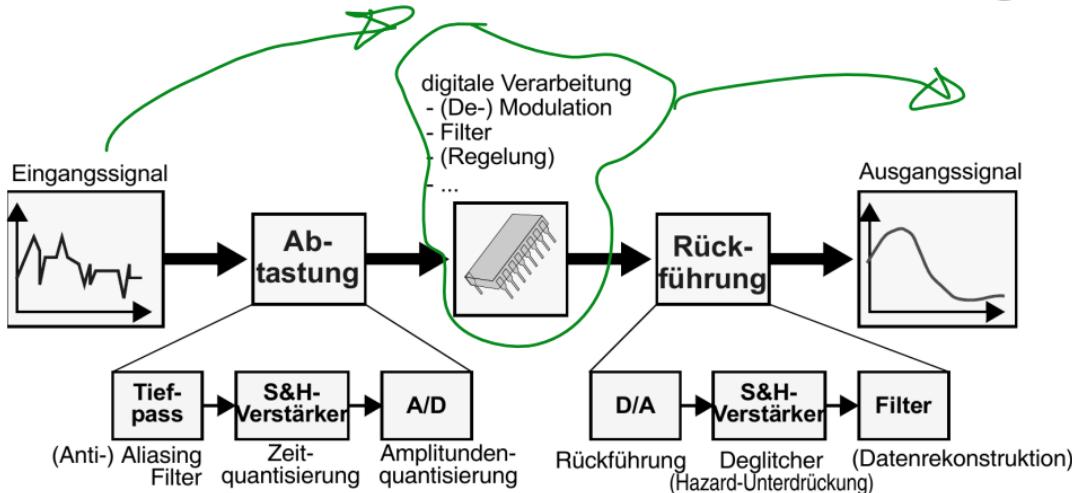


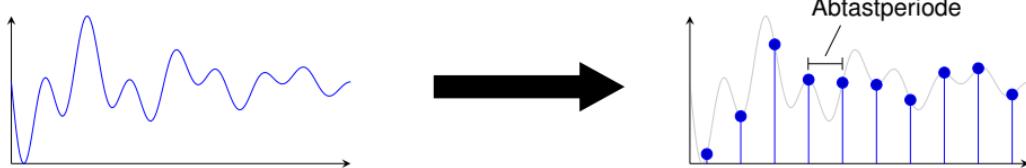
1. Sowohl zu jedem Zeitpunkt einen Messwert hat und diese Messwert kann beliebig viele Werte annehmen
2. Wir haben nur zur bestimmten Zeitpunkten Messwerte, aber diese Werte können beliebig viele Werte annehmen
3. Zu jeden zeitpunkt wir haben einen Messwert. Diese Werte können aber nur endlich viele bestimmte Werte annehmen. Treppenstufe erkennen.
4. Nur zur bestimmten Zeitpunkt Messwerte und diese Messwerte können nur bestimmte Werte annehmen



Signalverarbeitung

Signalverarbeitungsprozess





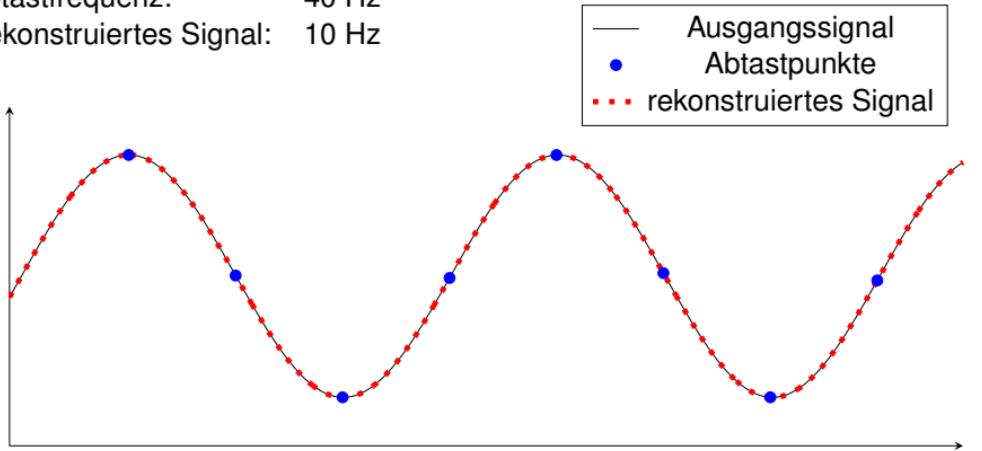
Ein Mikrocontroller kann nur **zeitdiskrete** und **wertdiskrete** Signalen verarbeiten.

- Ein kontinuierliches Signal muss daher **diskretisiert** werden.
- Hierzu wird das Signal **abgetastet**, d.h. zu bestimmten, meist äquidistanten Zeitpunkten wandelt der A/D-Wandler das Signal in einen digitalen Wert um.
- Diesen Vorgang nennt man auch **Quantisierung**.
 - Zeitquantisierung: zeitkontinuierlich \Rightarrow zeitdiskret
 - Amplituden- bzw. Wertquantisierung: wertkontinuierlich \Rightarrow wertdiskret

Abtastung

Welche Abtastfrequenz?

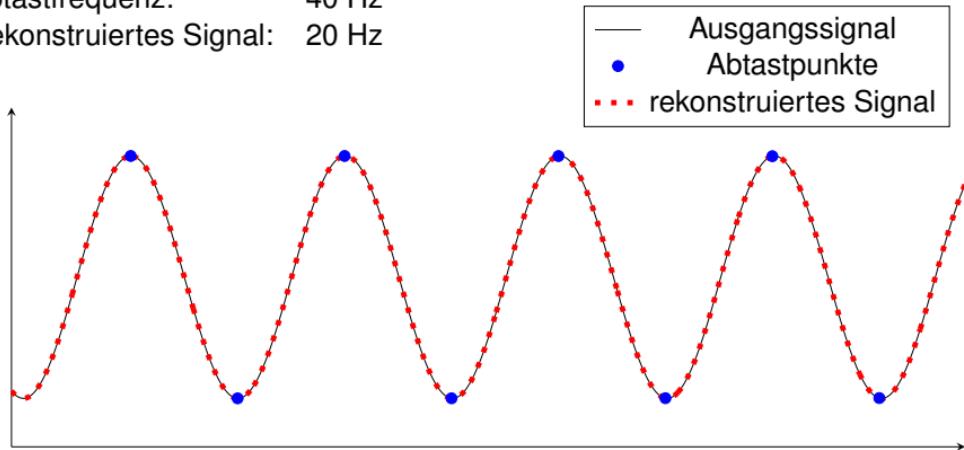
Ausgangssignal: 10 Hz
Abtastfrequenz: 40 Hz
Rekonstruiertes Signal: 10 Hz



Abtastung

Welche Abtastfrequenz?

Ausgangssignal: 20 Hz
Abtastfrequenz: 40 Hz
Rekonstruiertes Signal: 20 Hz



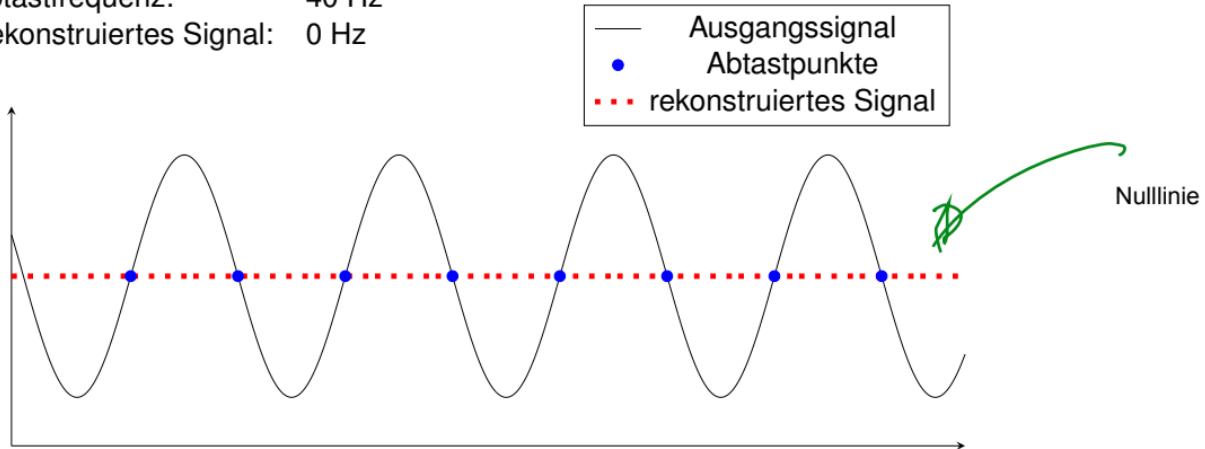
Abtastung

Welche Abtastfrequenz?

Ausgangssignal: 20 Hz (um Viertelperiode verschoben)

Abtastfrequenz: 40 Hz

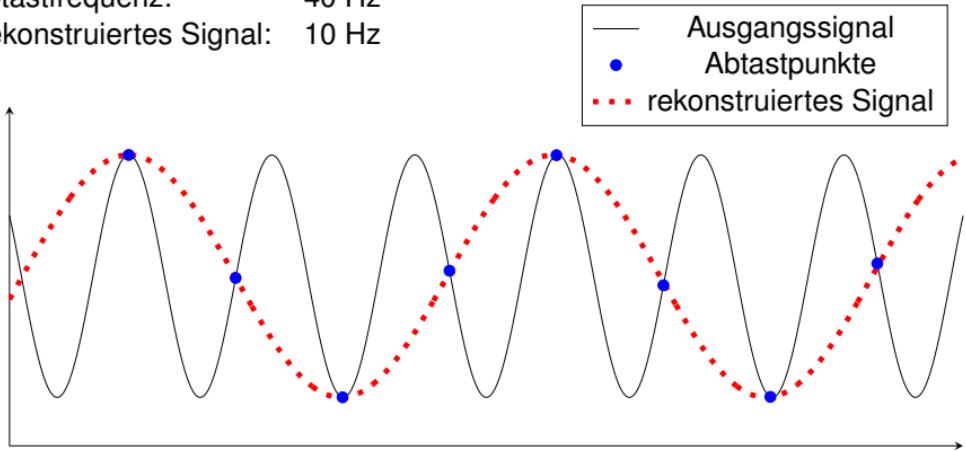
Rekonstruiertes Signal: 0 Hz



Abtastung

Welche Abtastfrequenz?

Ausgangssignal: 30 Hz
Abtastfrequenz: 40 Hz
Rekonstruiertes Signal: 10 Hz

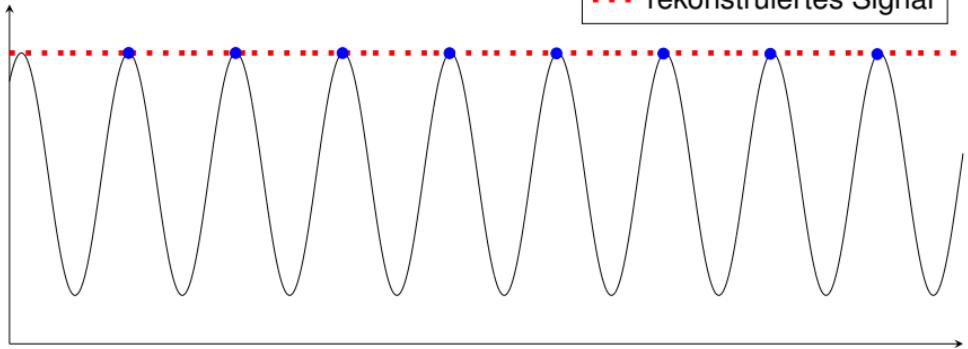


Abtastung

Welche Abtastfrequenz?

Ausgangssignal: 40 Hz
Abtastfrequenz: 40 Hz
Rekonstruiertes Signal: 0 Hz

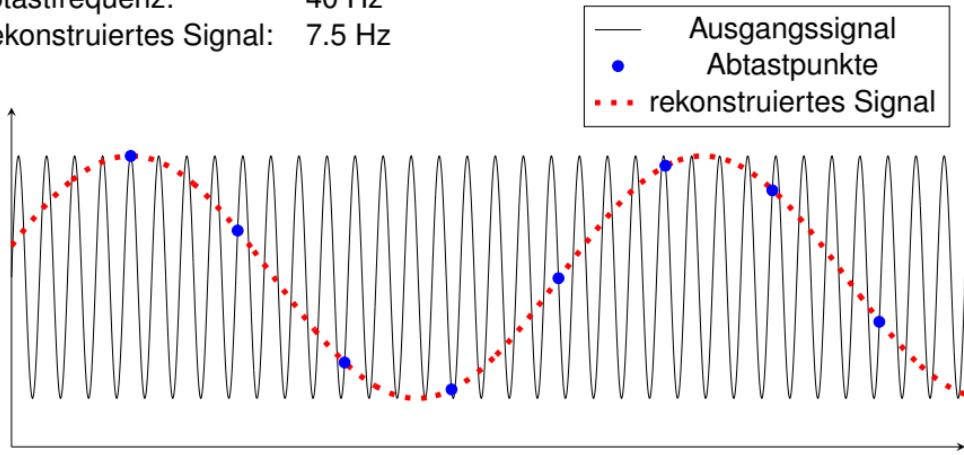
— Ausgangssignal
● Abtastpunkte
- - - rekonstruiertes Signal



Abtastung

Welche Abtastfrequenz?

Ausgangssignal: 153 Hz
Abtastfrequenz: 40 Hz
Rekonstruiertes Signal: 7.5 Hz



Welche Abtastfrequenz?

Abtasttheorem (Nyquist-Shannon-Theorem)



NUR GRÖSSER / MEHR ALS $2 \cdot f_{\max}$!!!!!

Abtasttheorem (Nyquist-Shannon-Theorem)

Ein auf f_{\max} **bandbegrenztes Signal** kann aus einer Folge von äquidistanten Abtastwerten exakt rekonstruiert werden, wenn die **Abtastfrequenz größer als $2 \cdot f_{\max}$** ist.



- Auf f_{\max} bandbegrenzt: es kommt keine Frequenz größer als f_{\max} vor.
- In der Praxis kommen noch Messfehler, nicht optimale Kennlinien und Rauschen hinzu.
 - ⇒ Signal lässt sich nicht mehr exakt rekonstruieren.
 - ⇒ Je größer die Abtastfrequenz, desto genauer ist die Rekonstruktion.
 - z.B. CD: $f_{\max} = 20\text{kHz}$, Abtastfrequenz = 44.1 kHz, Faktor = 2.21
 - z.B. Super Audio CD: $f_{\max} = 100\text{kHz}$, Abtastfrequenz = 2822 kHz, Faktor = 28.22

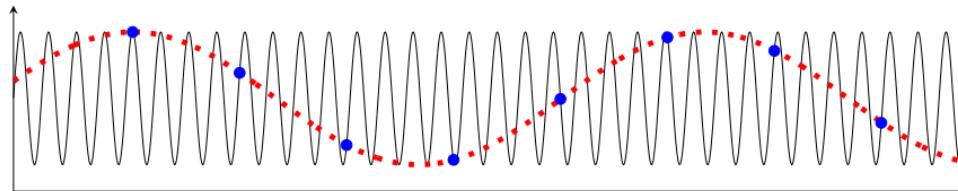
Welche Abtastfrequenz?

Alias-Effekt



Wenn das abzutastende Signal Frequenzanteile enthält, die höher als die halbe Abtastfrequenz sind, kommt es zu Fehlern.

- Im rekonstruierten Signal kommen dann Frequenzen vor, die im Originalsignal nicht vorkommen.



Diese Fehler nennt man **Alias-Effekt** (auch **Aliasing-Effekt** oder kurz **Aliasing**).

Häufige Aliasing-Quelle: **Rauschen**

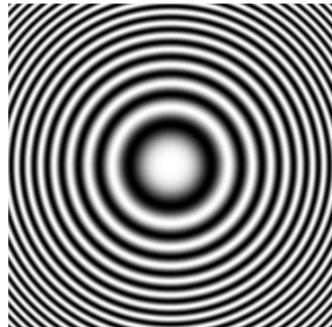
- Deshalb Signale VOR der Digitalisierung durch einen Tiefpass filtern (**Anti-Aliasing-Filter**)!

Welche Abtastfrequenz?

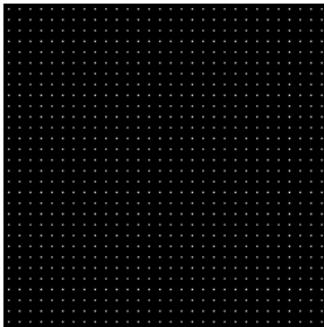
Alias-Effekt



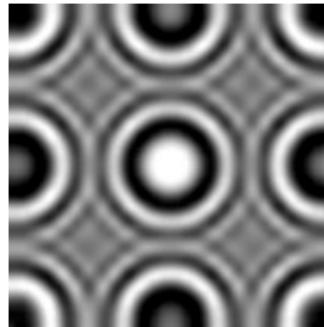
Aliasing tritt nicht nur bei elektrischen Signalen auf!



Originalbild



Abtastpunkte (30x30)



Rekonstruiertes Bild

Ein [Analog-Digital-Wandler](#) (A/D-Wandler bzw. [ADC](#)) wandelt ein anliegendes Spannungssignal in einen numerischen Wert um.

- Numerischer Wert gibt Verhältnis der anliegenden Spannung zu einer Referenzspannung wieder.
- Für numerischen Wert stehen nur endlich viele Bits zur Verfügung.
- Anzahl an zur Verfügung stehenden Bits bestimmt Qualität der Quantisierung.
 - Je mehr Bits, desto genauer kann das Signal rekonstruiert werden.
- Die Anzahl der Bits nennt man auch [Auflösung](#) des A/D-Wandlers.

Abtastung

Amplitudenquantisierung

8 Werte Können wir annehmen

Beispiel: 3-bit A/D-Wandler

Referenzspannung: 7 V
Eingangssignal: 0 – 7 V

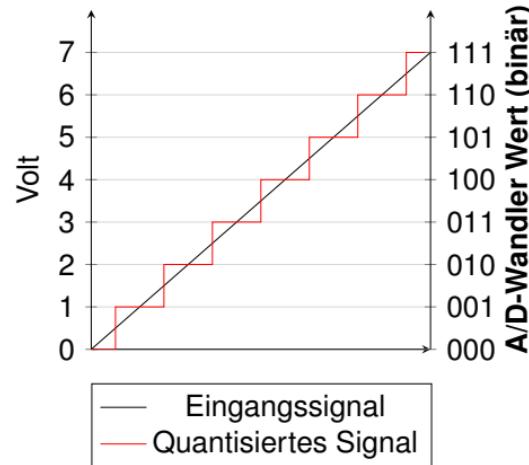
Formeln:

- $Wert_{ADC} = \text{round}(V_{in} \cdot \frac{2^n - 1}{V_{Ref}})$

⇒

- $\tilde{V}_{in} = Wert_{ADC} \cdot \frac{V_{Ref}}{2^n - 1}$
- Auflösung = $\frac{V_{Ref}}{2^n - 1}$
 - Man kann die Auflösung auch in Volt angeben. $\rightarrow \text{bit} = 1 \text{ V}$

$Wert_{ADC}$... A/D-Wandler Wert
V_{in}	... tatsächliche Eingangsspannung
\tilde{V}_{in}	... quantisierte Eingangsspannung
V_{Ref}	... Referenzspannung
n	... Auflösung in Bits



für schwarzes Signal finde ich beliebigviele Werte
für rotes: 0, 1, 2, 3 .. 7

3 bit = 1 V

Amplitudenquantisierung

Beispielmessungen

V_{in} : 2.6 V
 $Wert_{ADC}$: $round(2.6) = 3$

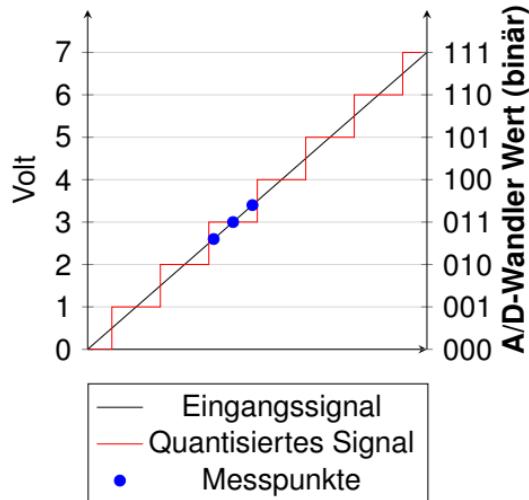
\tilde{V}_{in} : 3 V
Differenz $V_{in} - \tilde{V}_{in}$: -0.4 V

V_{in} : 3 V
 $Wert_{ADC}$: $round(3) = 3$

\tilde{V}_{in} : 3 V
Differenz $V_{in} - \tilde{V}_{in}$: 0 V

V_{in} : 3.4 V
 $Wert_{ADC}$: $round(3.4) = 3$

\tilde{V}_{in} : 3 V
Differenz $V_{in} - \tilde{V}_{in}$: 0.4 V



Definition (Quantisierungsfehler)

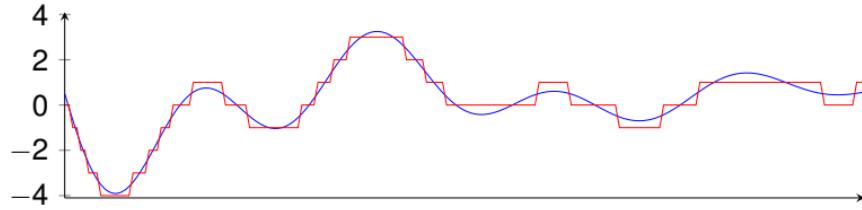
Der Quantisierungsfehler e_q (auch Quantisierungsabweichung oder Quantisierungsrauschen genannt) ist die Differenz zwischen dem Eingangssignal V_{in} und dem quantisierten Signal \tilde{V}_{in} .

- $-\frac{R_q}{2} < e_q \leq \frac{R_q}{2}$ (R_q ... Auflösung in Volt)

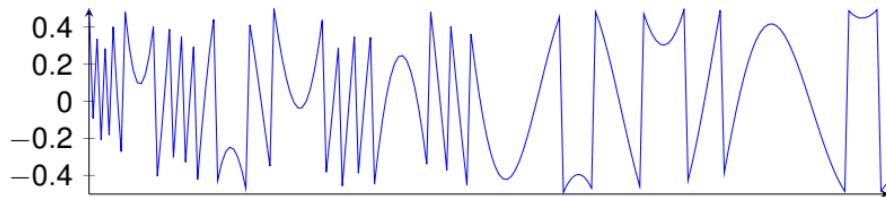
Amplitudenquantisierung

Quantisierungsfehler

Original- und quantisiertes Signal

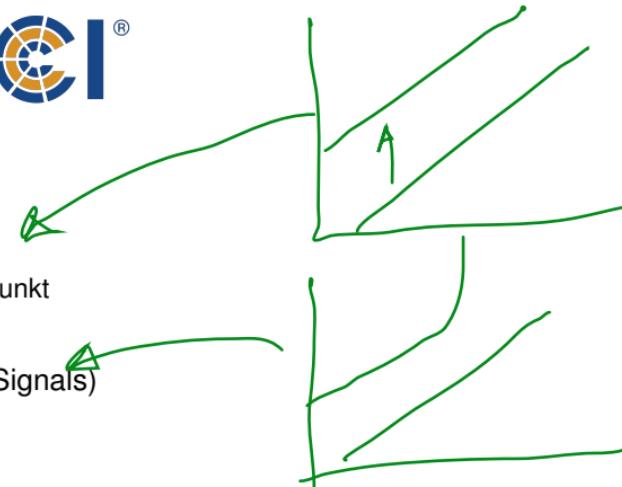


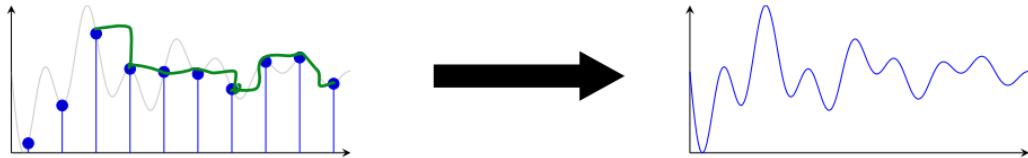
Quantisierungsrauschen



Weitere Fehler:

- Offsetfehler (Konstante Verschiebung des quantisierten Signals).
 - Gegenmaßnahme: Kalibration mit bekannter Spannung (ein Messungspunkt ausreichend).
- Nichtlineare Fehler (Nicht konstante Verschiebung des quantisierten Signals)
 - Gegenmaßnahme: Kalibration mit mehreren Messpunkten.
- Elektromagnetische Interferenzen von anderen Bauteilen
 - Gegenmaßnahme: Spezieller "Schlafmodus" schaltet nicht wichtige Bauteile ab.
- ...





Die **Rückführung** bzw. **Rekonstruktion** ist die Umkehrung der Abtastung:

- Ein **diskretes** Signal wird in ein **kontinuierliches** Signal umgewandelt.
- Die Umwandlung **zeitdiskret** \Rightarrow **zeitkontinuierlich** ist einfach
- Die Umwandlung **wertdiskret** \Rightarrow **wertkontinuierlich** ist schwierig
 - Mithilfe von sog. **Rekonstruktionsfiltern** können die Abtastwerte interpoliert werden.

Ein [Digital/Analog-Wandler](#) (D/A-Wandler bzw. **DAC**) wandelt einen numerischen Wert in eine analoge Spannung um.

- Numerischer Wert gibt Verhältnis der zu erzeugenden Spannung zur maximalen Spannung wieder.
- Für numerischen Wert stehen nur endlich viele Bits zur Verfügung.
- Anzahl an zur Verfügung stehenden Bits bestimmt Qualität der erzeugten Spannung.
 - Je mehr Bits desto näher kommt man an die gewünschten Spannung hin.
- Die Anzahl der Bits nennt man auch [Auflösung](#) des D/A-Wandlers.

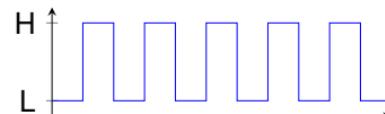
Dieselben Probleme wie beim A/D-Wandler:

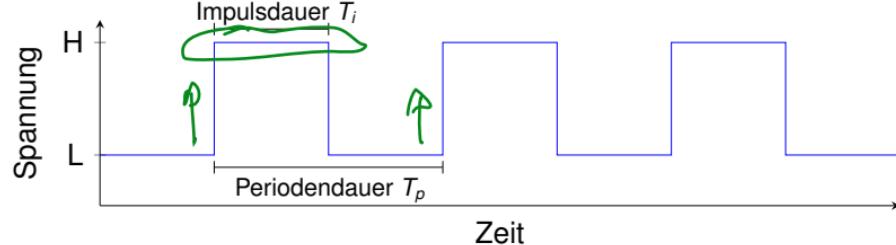
- Endliche Anzahl an möglichen Spannungspegeln führt zu Fehlern bei der Signalrekonstruktion.
 - Erzeugung eines echten wertkontinuierlichen Signals nicht möglich.
- Differenz zwischen gewünschter Spannung und tatsächlicher Spannung nennt man wieder **Quantisierungsfehler**.
- Was über Fehler beim A/D-Wandler gesagt wurde, trifft auch hier zu.

Definition (Pulsweitenmodulation)

Die Pulsweitenmodulation (kurz PWM) ist eine Modulationsart, bei der eine technische Größe (i.d.R. elektrische Spannung) zwischen zwei Werten hin und her wechselt. Dabei wird bei konstanter Frequenz die Pulsweite eines Rechteckpulses moduliert.

- Ist ein digitales Signal,
- daher leicht zu erzeugen.
- Wird von jedem Mikrocontroller unterstützt.
- Vielfältig einsetzbar.
- Eines der am häufigsten verwendeten Signalformen.





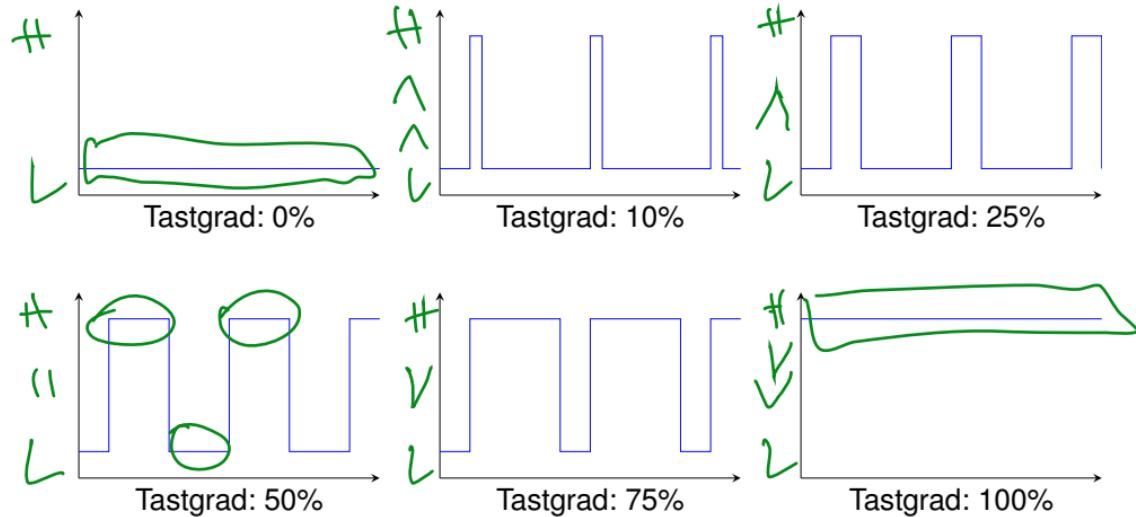
Definition (Tastgrad)

Als **Tastgrad** (auch **Aussteuergrad**, englisch **Duty Cycle**) bezeichnet man das **Verhältnis der Impulsdauer T_i zur Periodendauer T_p .**

- Tastgrad = $\frac{T_i}{T_p}$ (wird oft in Prozent angegeben)
- Periodendauer: fixiert, von Frequenz abhängig
- Impulsdauer: variabel, $0 \leq T_i \leq T_p$

Pulsweitenmodulation

Tastgrad



Pulsweitenmodulation

Einsatzgebiete



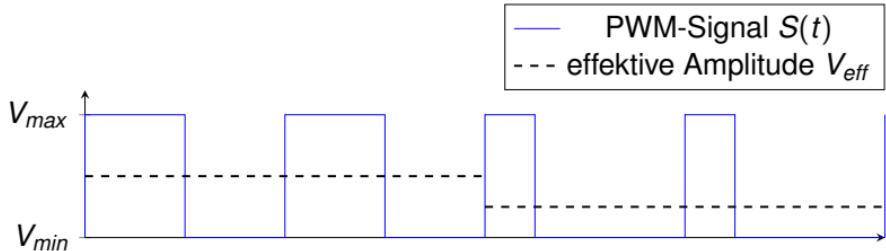
- Datenübertragung
 - Impulsdauer codiert Daten
- Steuerungstechnik
 - Dies ist das Haupteinsatzgebiet von PWM-Signalen in Embedded Systems
 - Steuerung von analogen Geräten mittels eines digitalen Signals
 - Erzeugung eines (quasi-)wertkontinuierlichen Signals in der Leistungselektronik
- Messtechnik
 - z.B.: A/D-Wandler, D/A-Wandler
- ...

Man nutzt die **Trägheit** von beteiligten Komponenten aus:

- z.B.: Motoren: Wenn man Stromzufuhr zu einem Motor kappt, bleibt er nicht sofort stehen.
 - z.B.: Menschliches Auge: Wenn man eine Lampe schnell genug ein- und ausschaltet, bekommt das Auge nicht mit.
 - z.B.: Tiefpass-Filter: Hohe Frequenzen werden gefiltert, daher wird das PWM-Signal geglättet.
- ⇒ Träge Komponenten “sehen” nur die **effektive Amplitude** eines PWM-Signals.

Pulsweitenmodulation

Effektive Amplitude



Um die **effektive Amplitude** eines PWM-Signals zu erhalten, wird das **Integral** des PWM-Signal berechnet:

- $V_{eff} = \int_{T_0}^{T_0+T_p} S(t) dt = V_{max} \cdot \frac{T_i}{T_p}$
- Beispiel: Motor wird mit PWM-Signal betrieben
 - $V_{max} = 5V$ und Tastgrad = 50%
 - ⇒ Motor wird effektiv mit 2.5V betrieben

Inhaltsverzeichnis



Einführung

Signalverarbeitung

Messen

Steuern

Regeln

Definition (Messen)

Unter **Messen** versteht man den Prozess der **Informationsgewinnung** über ein Mess-Objekt. Durch geeignete Sensorik wird eine **physikalische Größe** erfasst und in eine von einem **Mikrocontroller verarbeitbare Form** gebracht.

Definition (Sensor)

Ein **Sensor** (aus dem lateinischen *sensus*, übersetzt *Sinn*) ist eine Vorrichtung zum **Erfassen von physikalischen Größen**. Sie empfängt ein Signal oder Stimulus und reagiert darauf mit einem Ausgangssignal.

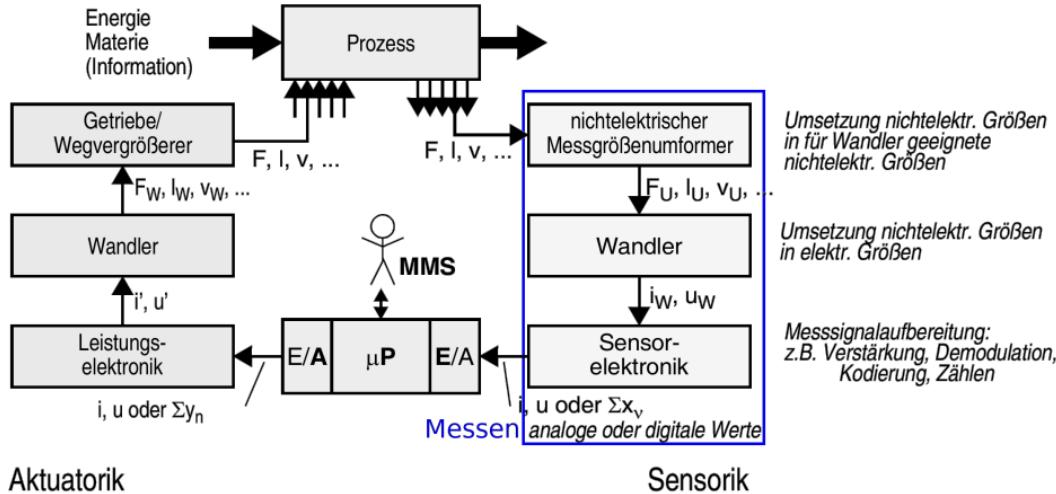
Jede reale Messung ist mit einem Messfehler behaftet.

Man unterscheidet zwei verschiedene Arten von Messfehlern:

- **Systematische Fehler**
 - Werden durch den Sensor selber verursacht (z.B. falsche Eichung).
 - Lassen sich durch sorgfältige Untersuchung beseitigen bzw. kompensieren.
- **Zufällige Fehler**
 - Treten zufällig auf (sog. statistische Fehler).
 - Können durch wiederholte Messung beseitigt bzw. kompensiert werden.

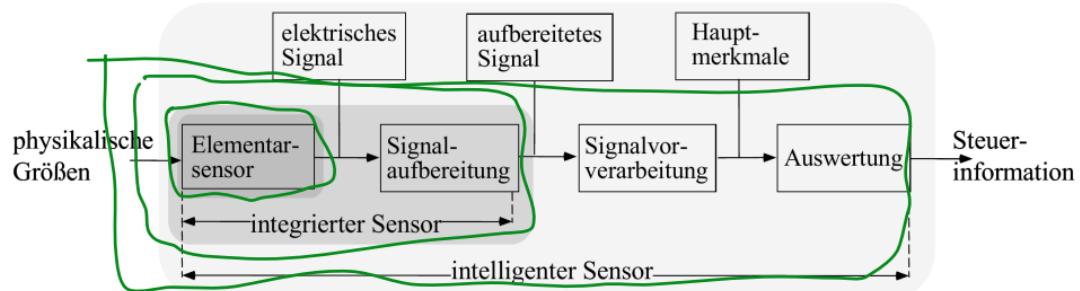
Messen

Messglied



Bei Sensoren kann man unterscheiden, "wo" gemessen wird:

- **Extrinsische/Externe Sensoren**
 - Ermitteln Informationen über die externe "Welt".
 - z.B.: Kamera, Mikrofon, Abstandssensoren, ...
- **Intrinsische/Interne Sensoren**
 - Ermitteln den internen Systemzustand.
 - z.B.: Lagesensor, Drehzahlmesser, ...



Elementarsensor: Wandelt die Messgröße in ein elektrisches Signal um.

Integrierter Sensor: Elementarsensor + Signalaufbereitung (Verstärkung, Filterung, ...)

Intelligenter Sensor: Integrierter Sensor + rechnergesteuerte Auswertung
(Digitalisierung, statistische Auswertung, ...)

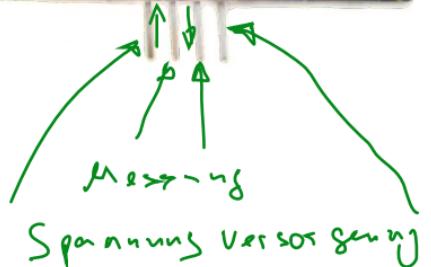
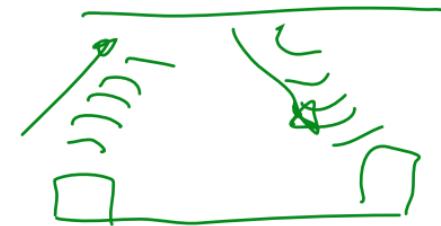
Messen

Sensor-Beispiel: Ultraschall-Abstandssensor HC-SR04



Auf dem EduArdu-Board ist ein HC-SR04 Ultraschall-Abstandssensor verbaut.

- Extrinsic, integrated sensor
- Determines distance to an obstacle using ultrasonic pulses.
- From the time between sending pulses and receiving reflected pulses, the distance is calculated.
- Range: 2cm - 4m



VCC Versorgungsspannung

TRIG Trigger-Eingang

ECHO Echo-Ausgang

GND Masse

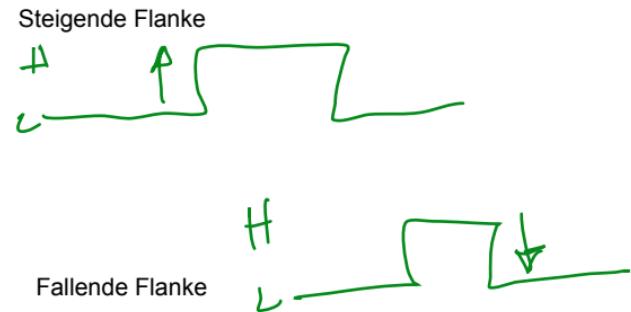
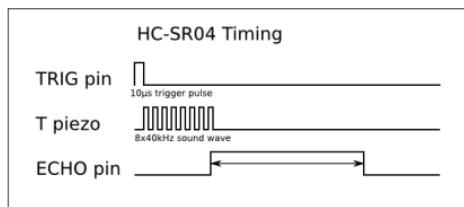
Messen

Sensor-Beispiel: Ultraschall-Abstandssensor HC-SR04



Funktionsweise:

- ① TRIG wird für mind. $10 \mu s$ auf HIGH geschalten. Fallende Flanke löst die Ultraschall-Impulse aus.
- ② Sobald die Impulse gesendet wurden, geht ECHO sofort auf HIGH.
- ③ Wenn die reflektierten Impulse empfangen werden, geht ECHO auf LOW.
- ④ Die Distanz D zum Hindernis berechnet sich durch: $D = \frac{T}{2} \cdot c_s$
(T ... gemessene Zeit, c_s ... Schallgeschwindigkeit)
- ⑤ Das Messintervall sollte nicht kleiner als 60 ms sein



Fehlerquellen:

- Systematische Fehler:
 - Die Schallgeschwindigkeit ist abhängig von Temperatur, Luftdruck und Luftfeuchtigkeit.
 - Diese Werte müssten zusätzlich ermittelt werden, um die exakte Schallgeschwindigkeit bestimmen zu können.
 - Oft wird die Zeitmessung gestartet, sobald der Trigger-Impuls losgeschickt wurde, anstatt auf die steigende Flanke von ECHO zu warten. Dies trägt auch zum systematischen Fehler bei.
- Zufällige Fehler:
 - Lokale Luftdruckschwankungen und andere zufällige Störeffekte sorgen für schwankende Messwerte bei gleichbleibender Distanz.
 - Mittelwert aus mehreren Messungen liefert besseres Ergebnis, und Bestimmung der Standardabweichung erlaubt Aussage über durchschnittlichen Messfehler.

Auf dem EduArdu-Board ist ein TCN75A Temperatursensor verbaut.

- Extrinsischer, intelligenter Sensor
- Bestimmt die Temperatur anhand von Spannungsunterschieden in der Flussspannung von Halbleiterelementen.
- Die gesamte Signalaufbereitung und -verarbeitung erledigt der Sensor selber.
- Wir bekommen direkt die Temperatur in Celcius zurück.
- Zusätzlich gibt es weitere Features: Auto-Modus, Alarm-Funktion, Schlaffunktion, ...

Auf dem EduArdu-Board ist ein Mikrofon verbaut.

- Extrinsischer Elementarsensor
- Wandelt mithilfe einer Membran und eines Magneten Schallwellen in elektrische Impulse um.
- Elektrischen Impulse sind aber zu schwach, um direkt vom A/D-Wandler umgewandelt zu werden.
- Benötigt daher eine eigene Mikrofon-Verstärker-Schaltung zur Signalauflaufbereitung.
- Signalverarbeitung ist Aufgabe des Mikrocontrollers (bzw. der darauf laufenden Software).

Inhaltsverzeichnis



Einführung

Signalverarbeitung

Messen

Steuern

Regeln

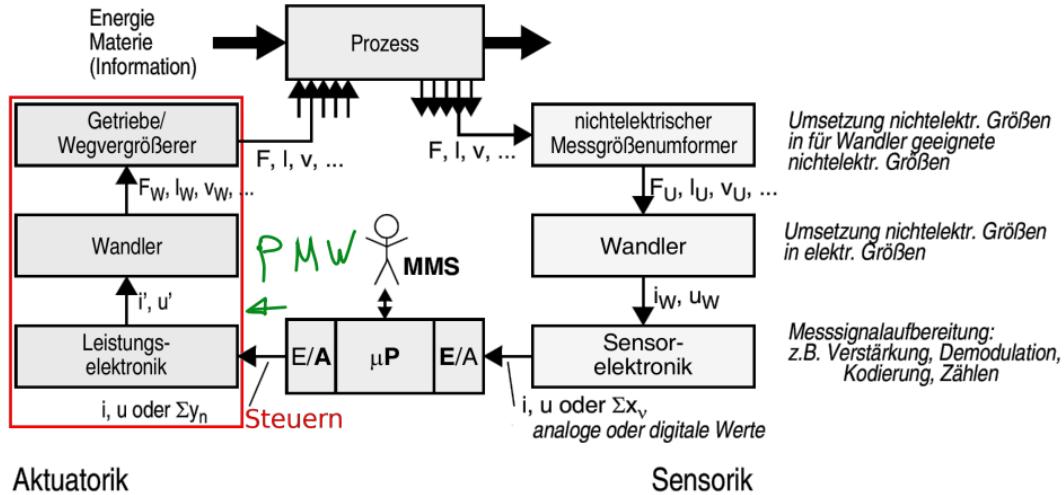
Definition (Steuern)

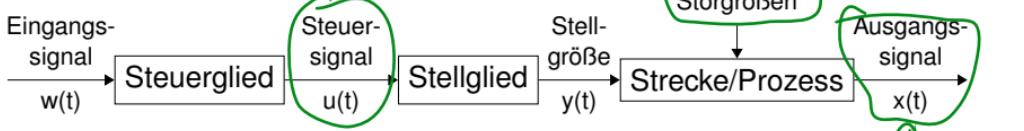
Unter **Steuern** versteht man die **gerichtete Beeinflussung des Verhaltens** technischer Systeme.

Definition (Aktuator)

Ein **Aktuator** ist eine Einheit, die ein elektisches Signal (**Steuersignal**) in **mechanische Bewegungen bzw. Veränderungen physikalischer Größen** (z.B. Druck, Temperatur, Licht) umsetzt.

- Binäre Steuerung
 - Kennt nur *zwei Stellungen*: meist *Ein* oder *Aus*
 - z.B.: Ventil, einfache Lampe, ...
- Analoge Steuerung
 - Wird durch ein *wertkontinuierliches* Analogsignal gesteuert.
 - bzw. durch ein quasi “analoges” PWM-Signal
 - z.B.: Motor, Lampe mit Dimmer, ...
- Digitale Steuerung
 - Wird durch digitale Signale gesteuert, die *mehr als 2 Zustände* codieren können.
 - z.B.: intelligente Aktuatoren, Lampe mit Digitaleingang, ...





Beschreibung Komponenten (am Beispiel eines Antriebssystems):

Eingangssignal: Gewünschte Geschwindigkeit

Steuerglied: Berechnet Tastgrad für PWM-Signal

Steuersignal: PWM-Signal in Mechanische bewegung

Stellglied: Setzt Steuersignal in Stellgröße um (Verstärkung)

Stellgröße: Motorspannung (verstärktes PWM-Signal)

Strecke/Prozess: Motor & Getriebe & Straße

Störgrößen: Verursachen Abweichungen von der Zielgröße (Beladung, Reifen- und Strassenzustand, ...)

Ausgangssignal: Tatsächliche Geschwindigkeit

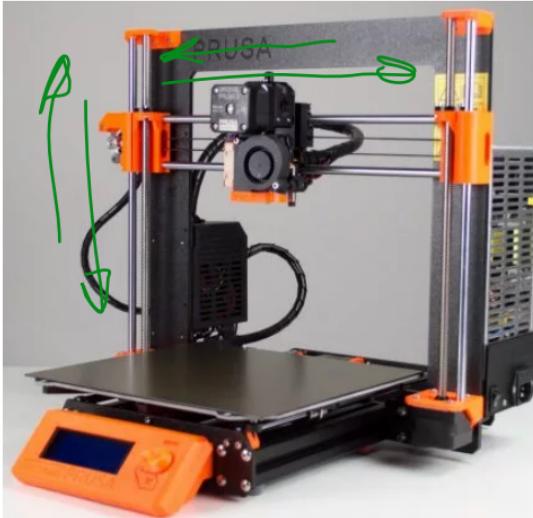
Problem:

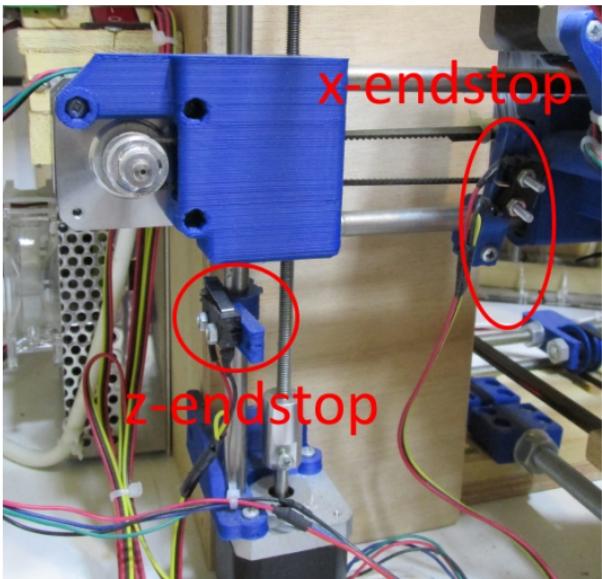
- Aufgrund der Störgrößen kann die tatsächliche Geschwindigkeit von der gewünschten abweichen.
- Abweichungen können nicht erkannt werden, da keine Rückkoppelung vorhanden ist.
- Daher kann sie auch nicht korrigiert werden.

Open-Loop Beispiel: Achse eines 3D-Druckers

Die Bewegungssachsen eines 3D-Druckers sind klassischerweise als Open-Loop Controller konzipiert.

- Beim Start des Druckvorgangs wird die Startposition angefahren.
- Das Erreichen der Startposition wird durch sog. Endstops detektiert (Schalter an den Achsenenden).
- Ausgehend von der Startposition wird blind navigiert.

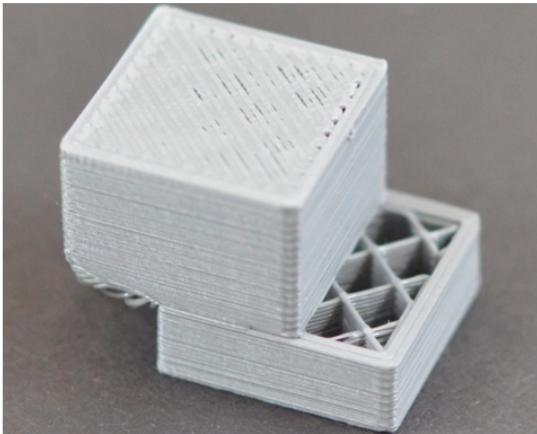




Open-Loop Beispiel: Achse eines 3D-Druckers

Im Fall von Störungen (z.B. lockere Keilriemen, böswillige Akteur:innen):

- Der 3D-Drucker bekommt nichts mit.
- Der Druckvorgang wird fortgesetzt, als wäre nichts geschehen.
- Das Druckobjekt weist verschobene Schichten auf (sog. [Layer Shifting](#)).



Inhaltsverzeichnis



Einführung

Signalverarbeitung

Messen

Steuern

Regeln

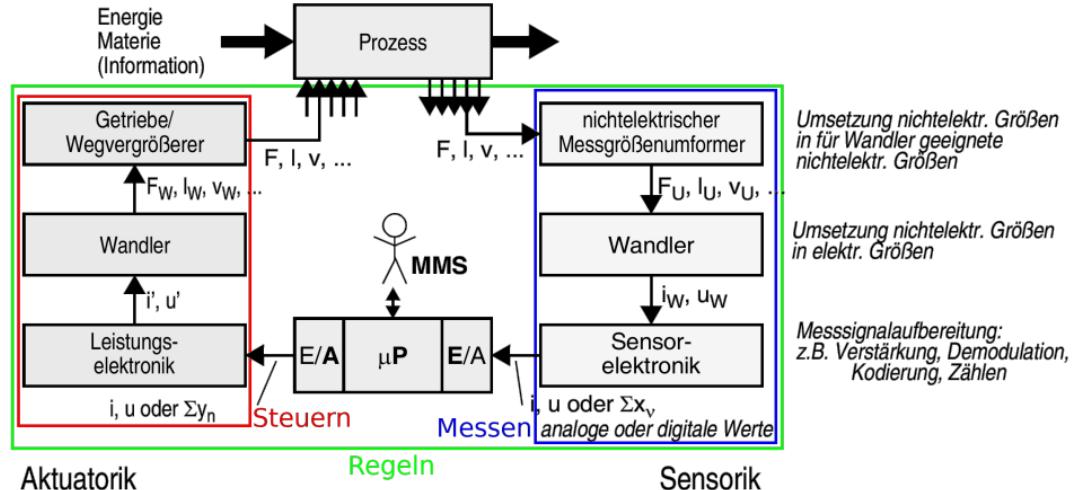
Definition (Regeln)

Unter **Regeln** versteht man die **gerichtete Beeinflussung des Verhaltens** technischer Systeme unter Zuhilfenahme eines **Rückkanals**, der kontinuierlich den **Ist-Zustand mit dem Soll-Zustand vergleicht**.

Festgestellte Abweichungen vom Sollzustand (die sog. **Regelabweichungen**) werden durch ein Nachjustieren des Steuersignals korrigiert.

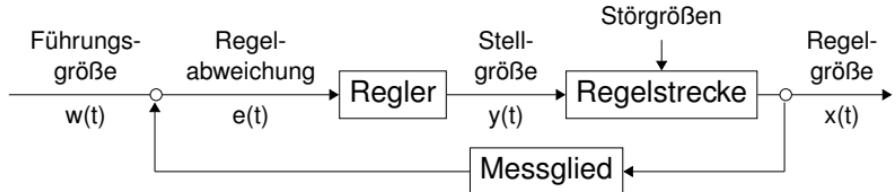
Definition (Regelkreis)

Als **Regelkreis** wird der **in sich geschlossene Wirkungsablauf** für die Beeinflussung des Verhaltens eines technischen Systems bezeichnet. Die physikalische Größen, die als Maß für das Verhalten dienen, nennt man **Regelgrößen**. Den gewünschten Soll-Zustand nennt man **Führungsgrößen**.



Regeln

Einfacher Regelkreis, Closed-Loop Controller



- 1 Der Ist-Zustand (Regelgröße) wird mit dem Soll-Zustand (Führungsgröße) verglichen.
- 2 Der Regler berechnet aus der Regelabweichung eine Stellgröße.
- 3 Die Stellgröße und die Störgrößen verursachen über die Regelstrecke eine Veränderung der Regelgröße.
- 4 Der neue Ist-Zustand wird mit dem Soll-Zustand verglichen.

Der Regler muss auf die Regelstrecke exakt abgestimmt sein:

- Der Einfluss der Stellgröße auf die Regelstrecke muss berücksichtigt werden.
- Der Einfluss der Störgrößen auf die Regelstrecke berücksichtigt werden.
- Die Verzögerung der Regelstrecke muss berücksichtigt werden.

Regeln

Einfacher Regelkreis, Closed-Loop Controller

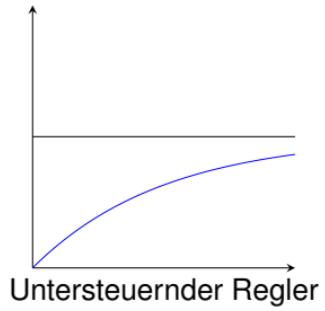
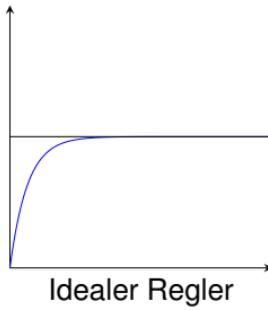
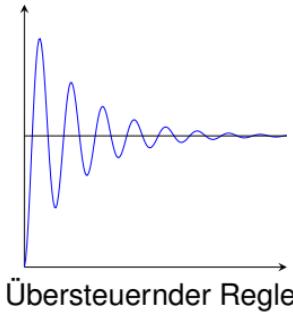


Eigenschaften eines guten Regelkreises:

- gutes Führungsverhalten
 - Der Soll-Zustand soll möglichst schnell erreicht werden ohne zu übersteuern oder untersteuern.
- stabil
 - Der erreichte Soll-Zustand soll möglichst gut gehalten werden.
- gutes Störverhalten
 - Große Ausschläge der Störgrößen dürfen nicht zu Instabilitäten führen.
- robust
 - Schleichende Parameteränderungen (z.B. durch Verschleiß) dürfen nicht zu Instabilitäten führen.

Regeln

Einfacher Regelkreis, Closed-Loop Controller



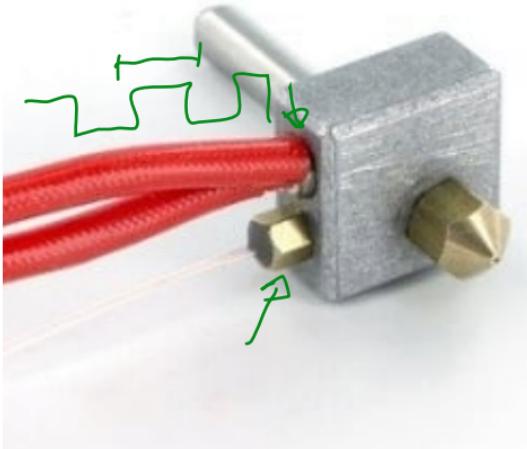
Regeln

Closed-Loop Beispiel: Heizelement eines 3D-Druckers



Die Heizungssteuerung eines 3D-Druckers klassischerweise als Closed-Loop Controller konzipiert.

- Im Druckkopf ist ein Heizelement und ein Thermistor (temperaturabhängiger Widerstand) verbaut.
- Die gemessene Temperatur wird laufend mit der gewünschten Temperatur verglichen.
- Zur Ansteuerung des Heizelements wird ein PWM-Signal verwendet.



Am häufigsten wird ein sogenannter **PID-Regler** verwendet,
welcher ein 3-in-1 Regler ist:

Proportionalregler: Reagiert sofort und unmittelbar auf normale Änderungen.

Integralregler: Merkt sich vergangene Regelabweichungen und reagiert verzögert.

Differentialregler: Reagiert auf sehr starke Änderungen.

Näheres ist im Selbststudium zu erarbeiten.

2 MESSEN, STEUERN & REGELN

▼ PID-Regler – Überblick

Ein **PID-Regler** ist ein weit verbreiteter **Regelalgorithmus** in der Automatisierungstechnik, der die Differenz zwischen einem gewünschten Sollwert und dem aktuellen Istwert (die **Regelabweichung**) auswertet und daraus ein Steuersignal berechnet, um die Regelgröße zu beeinflussen.

PID steht für:

- **P** = Proportionalanteil
- **I** = Integralanteil
- **D** = Differentialanteil

Ziel: **schnelles, genaues, stabiles und robustes Regelverhalten**

◆ P – Proportionalanteil

- Berechnet:

$$P = K_p * e(t)$$

- Reagiert **sofort proportional** zur aktuellen Regelabweichung $e(t)$.
- Je größer K_p , desto schneller die Reaktion.
- **Problem bei alleiniger Nutzung:** Bleibende Regelabweichung (stationärer Fehler), bei zu großem K_p : Instabilität (Überschwingen).

◆ I – Integralanteil

- Berechnet:

$$I = K_i * \int e(t) dt$$

- **Integriert** die Regelabweichung über die Zeit.
- Sorgt dafür, dass die **bleibende Regelabweichung** verschwindet (regelt auf 0).
- **Nachteil:** Kann zu langsamer Reaktion und **Schwingneigung** führen (Integratorüberschwingung).

◆ D – Differentialanteil

- Berechnet:

$$D = K_d * d(e(t)) / dt$$

- **Prognostiziert** zukünftige Entwicklungen der Regelabweichung (aufgrund ihrer Änderungsrate).
- Dämpft schnelle Änderungen, z.B. durch **Störungen**.
- **Vorsicht:** Sehr empfindlich gegenüber Rauschen im Signal.

⚙ Gesamte Reglerformel (kontinuierlich)

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)dt + K_d \cdot \frac{de(t)}{dt}$$

Diskrete Form (für Mikrocontroller)

In Mikrocontrollern (z. B. Arduino) wird der Regler **diskretisiert**, da Zeit in Schritten verarbeitet wird:

$$u[n] = K_p \cdot e[n] + K_i \cdot \sum e[n] \cdot T + K_d \cdot \frac{e[n] - e[n-1]}{T}$$

Dabei:

- T: Abtastzeit
- e[n]: Regelabweichung im aktuellen Schritt

Tunen eines PID-Reglers

Ziel: Optimale Werte für K_p, K_i, K_d finden. Methoden:

1. **Manuell** (Trial and Error)
2. **Ziegler-Nichols-Methode**
3. **Softwareunterstützt** (z. B. MATLAB, Simulink, Autotuning)

Wirkung der Anteile auf das Systemverhalten

Anteil	Wirkung	Vorteil	Nachteil
P	Reagiert sofort	Schnelles Ansprechen	Bleibende Regelabweichung
I	Beseitigt Regelabweichung	Keine bleibende Abweichung	Langsam, kann schwingen
D	Reagiert auf Änderung	Dämpft schnelle Änderungen	Rausch-empfindlich

Anwendung auf Mikrocontroller

Beispiel aus der **Microchip AppNote (AVR221)**:

- Implementierung eines diskreten PID-Reglers auf tinyAVR und megaAVR.
- Gleitkommaoperationen sind oft zu langsam → **Ganzzahlarithmetik bevorzugt**
- Empfehlung: Festkommaformate verwenden (z. B. Q15).
- Wichtig: **Begrenzung** der Ausgänge (Sättigung) und **Anti-Windup** beim I-Anteil

Praxisbeispiel: Heizbettregelung eines 3D-Druckers (siehe Vorlesung)

- Solltemperatur vs. gemessene Temperatur
- PID-Regler berechnet PWM-Signal für das Heizelement
- Ziel: Schnelles Erreichen und Halten der Solltemperatur trotz Störungen (z. B. Luftzug)

Quellenübersicht

1. [atwillys.de – Erklärung mit Formeln & Diagrammen](#)
2. [Arduino Forum – anschauliche, nicht-wissenschaftliche Erklärung](#)
3. [RN-Wissen – Regelungstechnik Übersicht](#)

4. [Microchip AppNote AVR221 – Diskrete PID-Regler in Embedded Systemen](#)

▼ Mikrocontrollerbaustein: Timer / Counter

Timer (auch **Zähler**) sind **essenzielle Hardware-Module** in Mikrocontrollern wie dem AVR, um zeitbezogene Aufgaben **ohne permanente CPU-Belastung** durchzuführen.



Grundfunktion

Ein **Timer/Counter** ist ein Register, das automatisch **hochzählt (Inkrementierung)** in regelmäßigen Abständen, meist ausgelöst durch:

- einen internen **Taktgeber** (z. B. Systemtakt oder Prescaler),
- externe Signale (z. B. an einem Pin, bei Counter-Modus).

Wenn ein definierter Wert erreicht ist (z. B. **Überlauf oder Vergleichswert**), kann:

- ein **Interrupt ausgelöst** werden,
- ein **Pin verändert** werden,
- oder ein **neuer Zyklus gestartet** werden.



Aufbau eines AVR-Timers

Ein Timer besteht meist aus folgenden Komponenten:

- **Zählregister (TCNTx)**: enthält den aktuellen Zählerstand
- **Steuerregister (TCCR_x)**: konfiguriert den Modus (Normal, PWM, CTC ...)
- **Compare-Register (OCR_x)**: Vergleichswert für Auslösung von Events
- **Interrupt-Flags (TIFR_x)**: Status, ob ein Event ausgelöst wurde
- **Prescaler**: teilt den Takt herunter, um langsamere Timerereignisse zu ermöglichen



Modi von Timern

Modus	Beschreibung	Beispiel-Anwendung
Normal Mode	Zählt bis Überlauf (0xFF oder 0xFFFF), löst dann Overflow-Interrupt aus	einfache Zeitmessung
CTC (Clear Timer on Compare)	Zählt bis zu einem Vergleichswert (OCR _x), dann Reset + optional Interrupt	präzise Zeitbasis für PWM
PWM (Fast/Phase Correct)	Erzeugt ein PWM-Signal an einem Ausgangspin, je nach Compare-Wert	Motorsteuerung, Dimmen von LEDs
Counter Mode (extern)	Zählt externe Ereignisse statt Taktzyklen	Schrittzähler, Frequenzmessung



Prescaler

Prescaler bestimmen die Geschwindigkeit des Timer-Takts:

- z. B. bei einem 16 MHz Takt und einem Prescaler von 64 \rightarrow Timer-Takt = 250 kHz
 - Je größer der Prescaler, desto **langsamer** der Timer (und umgekehrt)
 - Typische Werte: 1, 8, 64, 256, 1024
-



Typen von AVR-Timern

Timer	Breite	Besonderheiten
Timer0	8 Bit	Standard für Zeitverzögerungen
Timer1	16 Bit	Genauere Zeiten, Servo-PWM
Timer2	8 Bit	Unabhängig von Systemtakt, für RTC verwendbar



Interrupts

Timer können **Interrupts** auslösen:

- **Overflow Interrupt:** wenn der Timer überläuft
- **Compare Match Interrupt:** wenn TCNTx == OCRx
- Ermöglicht **nicht-blockierende Zeitsteuerung**

Beispiel:

```
ISR(TIMER1_COMPA_vect) {
    // Code, der alle x ms ausgeführt wird
}
```



Zeitberechnung (Beispiel)

Gegeben:

- Systemtakt: 16 MHz
- Prescaler: 64
- Timer = 8 Bit (max = 255)

Zeit für einen Überlauf:

$$\text{Timer-Takt} = \frac{16 \text{ MHz}}{64} = 250,000 \text{ Hz} \Rightarrow 1 \text{ Tick} = 4 \mu\text{s}$$

$$\text{Überlauf nach} = 256 \cdot 4 \mu\text{s} = 1.024 \text{ ms}$$



Typische Anwendungen

- PWM-Erzeugung (z. B. Servo, LED-Dimmung)
- präzise Zeitverzögerungen
- Echtzeituhr (RTC)
- Frequenzmessung (Counter-Modus)

- Software-Timer via Interrupt
-



Beispiel: Delay mit Timer0 im CTC-Modus

```
// 1ms Takt mit 16MHz Takt und Prescaler 64  
OCR0A = 250;  
TCCR0A = (1<<WGM01); // CTC Mode  
TCCR0B = (1<<CS01) | (1<<CS00); // Prescaler 64  
TIMSK0 = (1<<OCIE0A); // Enable Compare Interrupt
```



Quellenübersicht

1. [AVR-GCC: Timer und Counter bei Wikibooks](#)
2. [AVR-GCC Tutorial – mikrocontroller.net](#)
3. [AVR-Tutorial: Timer – mikrocontroller.net](#)
4. [RN-Wissen: Timer/Counter \(AVR\)](#)

▼ delay() vs. Timer auf Arduino



delay(ms)

- Einfach, aber blockierend

- delay() pausiert den Code für eine bestimmte Zeit (z. B. delay(1000) = 1 Sekunde).
- **Problem:** Während der Delay-Zeit wird der **gesamte Sketch blockiert**.
- Das bedeutet:
 - Keine Reaktion auf Benutzereingaben
 - Keine parallelen Aktionen möglich
 - Keine Sensorabfragen oder Display-Updates

Fazit: Gut für **einfache Sketche**, aber **ungeeignet für komplexe Anwendungen** mit mehreren Aufgaben.



Timer als Alternative zur delay()-Funktion

Mikrocontroller wie der ATmega328P (Arduino Uno) haben eingebaute **Timer**, die unabhängig vom Hauptprogramm laufen und regelmäßig Interrupts auslösen können.

Vorteile:

- **Nicht-blockierend:** Hauptprogramm läuft weiter
- **Multitaskingähnliches Verhalten** (z. B. Blinken und Sensorlesen gleichzeitig)
- **Präziser** als delay(), da unabhängig von CPU-Last



Implementierungsmöglichkeiten

1.

Millis()-basierte Zeitsteuerung (nicht direkt Timer, aber timer-gestützt)

```
unsigned long vorher = 0;  
const int intervall = 1000;  
  
void loop() {  
    if (millis() - vorher >= intervall) {  
        vorher = millis();  
        // Aktion alle 1000 ms  
    }  
}
```

✓ nicht blockierend

✗ Ungenau bei sehr kleinen Intervallen

✗ nicht echtzeitfähig

2.

Hardware-Timer mit Interrupts

Beispiel mit

Timer1

:

```
#include <avr/io.h>  
#include <avr/interrupt.h>  
  
void setup() {  
    noInterrupts();      // Interrupts deaktivieren  
    TCCR1A = 0;          // Normalmodus  
    TCCR1B = 0;  
    TCNT1 = 0;  
    OCR1A = 15624;       // 1s bei 16 MHz Takt und Prescaler 1024  
    TCCR1B |= (1 << WGM12); // CTC-Modus  
    TCCR1B |= (1 << CS12) | (1 << CS10); // Prescaler 1024  
    TIMSK1 |= (1 << OCIE1A); // Enable Timer Compare Interrupt  
    interrupts();         // Interrupts aktivieren  
}  
  
ISR(TIMER1_COMPA_vect) {  
    // Diese Funktion wird alle 1s aufgerufen  
}
```

✓ Hochpräzise

✓ Echtzeitfähig

✗ Aufwändiger in der Einrichtung

 Nur wenige Timer verfügbar (3 bei Arduino Uno)

Vergleichstabelle

Feature	delay()	millis()-Vergleich	Hardware-Timer/ISR
Blockiert Code?	 Ja	 Nein	 Nein
Genaugigkeit	Mittel	Gut	Sehr gut
Echtzeitfähig	 Nein	 Eher nicht	 Ja
Einfache Nutzung	 Ja	 Ja	 Nein
Multitasking	 Nein	 Eingeschränkt	 Ja

Tipp: Bibliotheken zur Vereinfachung

Für Arduino gibt es Libraries, die Timer einfacher machen:

- TimerOne, TimerThree
- MsTimer2
- SimpleTimer (für millis()-basierte Zeitsteuerung)

Beispiel mit TimerOne:

```
#include <TimerOne.h>

void blink() {
    digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
}

void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
    Timer1.initialize(1000000); // 1 Sekunde
    Timer1.attachInterrupt(blink);
}
```

Quellen

1. [Arduino ProjectHub – delay\(\) vs Timer](#)
2. [StartHardware – Timer mit Arduino als Alternative zu delay\(\).](#)

▼ Interrupts – Was ist das?

Ein **Interrupt** (Unterbrechung) ist ein **Signal an den Mikrocontroller**, das ihn veranlasst, seine aktuelle Programmausführung zu **unterbrechen** und eine **spezielle Routine** (Interrupt Service Routine, ISR) auszuführen.

Ziel: **Schnelle, reaktive Reaktion auf Ereignisse**, ohne dass ständig darauf gewartet werden muss (kein "Polling").



Arten von Interrupts

Art	Beschreibung	Beispiel
Hardware-Interrupt	Durch Peripherie ausgelöst	Taster, Timer, UART-Eingang

Art	Beschreibung	Beispiel
Software-Interrupt	Durch Programmcode ausgelöst	sei(), cli(), asm("int")
Externe Interrupts	Signal an spez. Pin (INTx)	Taster an INT0
Interne Interrupts	Von interner Hardware ausgelöst	Timer-Überlauf, ADC fertig, USART RX



Ablauf eines Interrupts

1. Ein Ereignis (z. B. ein Pin ändert seinen Zustand) **löst Interrupt aus.**
2. Aktuelles Programm wird **pausiert.**
3. Der **Program Counter (PC)** wird gespeichert.
4. Die **Interrupt Service Routine (ISR)** wird ausgeführt.
5. Nach der ISR kehrt das Programm an die **unterbrochene Stelle zurück** (automatisch mit RETI).



Interrupts in AVR / Arduino

Aktivieren und Deaktivieren:

```
sei(); // Global Interrupts aktivieren
cli(); // Global Interrupts deaktivieren
```

! Achtung: Viele ISR-nahe Codeabschnitte müssen

atomar



Beispiel: ISR bei Tasterdruck

```
ISR(INT0_vect) {
    // Wird ausgelöst bei Flanke an INT0 (Pin D2 auf Uno)
    digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
}

void setup() {
    pinMode(2, INPUT_PULLUP);      // INT0-Pin
    pinMode(LED_BUILTIN, OUTPUT);
    EICRA |= (1 << ISC01);      // Fallende Flanke an INT0
    EIMSK |= (1 << INT0);       // INT0 aktivieren
    sei();                      // Interrupts global aktivieren
}
```



Interrupt-Vektoren (AVR)

Jede Interruptquelle hat ihren eigenen **Vektor** (eine Adresse im Flash):

- INT0_vect, TIMER1_OVF_vect, ADC_vect, ...
- AVR-GCC erlaubt die einfache Definition über ISR(...)



Wichtige Regeln für ISR

1. **So kurz wie möglich** schreiben – keine Delays, keine Serial-Ausgaben!
2. Globale Variablen, die in ISR und loop() verwendet werden, als volatile markieren!

```
volatile uint8_t flag = 0;
```

- 1.
2. Möglichst **keine komplexen Rechenoperationen** in der ISR durchführen.
3. **Atomare Zugriffe** auf gemeinsam genutzte Variablen (z. B. mit cli() / sei() umschließen).



Typische Anwendungen

- **Taster-Eingaben** (entprellt per Software)
- **Timer-basierte Aufgaben** (z. B. Millisekudentakt)
- **UART-Empfang** (nicht blockierend)
- **Encoder-Auswertung**
- **Reaktionsschnelle Steuerungen** (z. B. Not-Aus)



Interrupt vs. Polling

Aspekt	Polling	Interrupt
CPU-Auslastung	Hoch	Niedrig
Reaktionszeit	Verzögert	Sofort (asynchron)
Komplexität	Einfach	Etwas komplexer
Stromverbrauch	Höher	Geringer (Sleep möglich)
Kontrolle	Ständig aktiv	Eventgesteuert



Quellen

1. [Wikipedia: Interrupt](#)
2. [mikrocontroller.net: Interrupt](#)
3. [RN-Wissen: Interrupt](#)
4. [Mini-Kurs Uni Regensburg: Interrupts](#)



SOFTWAREENTWICKLUNG AUF EMBEDDED SYSTEMS

RECHNERSTRUKTUREN & EMBEDDED SYSTEMS

**MATTHIAS JANETSCHKEK
SS 2025**

Eigenschaften von eingebetteter Software

Aufbau eingebetteter Systeme

Softwareentwicklung für eingebettete Systeme

Eigenschaften von eingebetteter Software

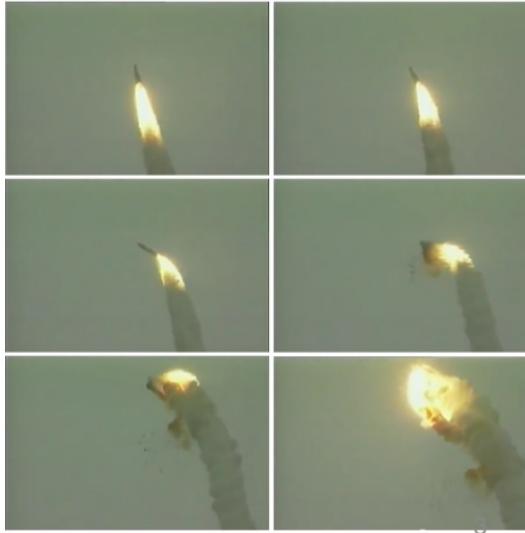
Aufbau eingebetteter Systeme

Softwareentwicklung für eingebettete Systeme

Beispiel: Erstflug der Ariane 5

Teuerster Softwarefehler der Geschichte

- Am 4. Juni 1996 explodierte die Ariane 5 Rakete bei ihrem Erstflug.
 - Gesamtschaden: 280 Mio. Euro (andere Schätzungen bis zu 500 Mio. Euro).
 - Einjährige Verzögerung des Ariane 5 Programms.
- Die Fehlerursache war die Software der Inertialen Navigationssysteme (INS).
 - Ein Zahlenüberlauf wurde nicht korrekt behandelt (bei Umwandlung von double nach short).
 - Der Hauptcomputer hat die Fehlermeldungen für echte Messdaten gehalten.



Beispiel: Erstflug der Ariane 5

Ursache



- Die betroffene Software wurde nahezu unverändert und **ungeprüft** von der Ariane 4 übernommen.
 - Mit entsprechenden Simulationen hätte man den Fehler gefunden.
 - Auf eine korrekte Behandlung des Zahlenüberlaufs wurde aus Ressourcengründen verzichtet.
 - Bei der Ariane 4 kam es nie zu einem Überlauf.
 - Entsprechende Dokumentation war nicht vorhanden.
 - Ariane 5 hatte aber eine höhere Horizontalgeschwindigkeit als Ariane 4.
 - Und diese höhere Geschwindigkeit verursachte den Zahlenüberlauf.
- ⇒ Das Hauptproblem war, dass die betroffene Software ohne korrekte Überprüfung aus dem Gesamtkontext gerissen wurde.

Eigenschaften von eingebetteter Software

Besonderheiten



- Integraler Bestandteil eines technischen Systems
 - Erfüllt dedizierte Aufgabe als **Teil eines Gesamtsystems**
 - Oft für spezielle Hardware entwickelt und darauf optimiert
 - Oft auf spezifische Sensoren/Aktuatoren und deren Schnittstellen, Ungenauigkeiten und Besonderheiten zugeschnitten
 - Muss im Gesamtkontext betrachtet werden
- Oft keine “Software von der Stange”, sondern **Maßanfertigung**
 - Nicht einfach auf andere Systeme übertragbar
- Entwicklung braucht oft **intensives Domänenwissen**
 - z.b.: Entwicklung von Fahrassistenzsystemen benötigt Kenntnisse über physikalische Prinzipien der Fahrdynamik

Beispiel: Golfkrieg 1991

Patriot vs SCUD



- Im 2. Golfkrieg wurden das erste Mal Patriot Systeme zur Abwehr von ballistischen Raketen (SCUD-Raketen) verwendet.
 - Ballistische Raketen sind sehr schnell (SCUD-Raketen: 1500 m/s)
 - Genaues zeitliches Timing ist beim Auffangen entscheidend.
- Zeit wurde als vergangene Zeit (in Zehntel-Sekunden) seit Systemstart gemessen.
 - Als Ganzzahl abgespeichert.
 - Je länger das System läuft, desto größer wird diese Zahl.



Beispiel: Golfkrieg 1991

Fehler nach längerer Laufzeit



- Für Positionsberechnungen des Ziels musste Zeit in eine Gleitkommazahl konvertiert werden.
 - Dabei wird die Zeit mit 0.1 multipliziert.
 - Die Binärdarstellung von 0.1 ist unendlich und muss abgeschnitten werden.
 - Bei 24-Bit Registern ergibt das einen Fehler von 0.000000095 Sekunden pro Zehntelsekunde.
- ⇒ Mit wachsender Betriebsdauer wird Fehler immer größer.
 - Fehler war seit 11. Februar 1991 bekannt bekannt, aber erste Einschätzung war "der normale Patriotbenutzer lässt das System nicht länger als acht Stunden laufen".
 - Software-Update wurde am 16. Februar 1991 veröffentlicht.
 - Entsprechender Hinweis wurde am 21. Februar 1991 an Benutzer verschickt.

Beispiel: Golfkrieg 1991

Katastrophe



- Konsequenz: "Am 25. Februar 1991 verfehlte eine amerikanische Patriot-Rakete eine anfliegende irakische Scud-Rakete. Die Scud schlug in eine Armeebaracke ein, tötete 28 Soldaten und verletzte ca. 100 Personen".
- Die betreffende Patriot-Batterie war zu dem Zeitpunkt schon seit 100 Stunden in Betrieb (und natürlich nicht gepatcht).
 - Fehler: $0.000000095 \cdot 100 \cdot 60 \cdot 60 \cdot 10 = 0.34$ Sekunden
 - In 0.34 Sekunden fliegt eine SCUD-Rakete ca. 500 Meter.

- Muss oft auf **Langzeitbetrieb ausgelegt sein.**
 - Benötigt robuste Fehlerbehandlung, die das Gesamtsystem nicht beeinträchtigt.
 - In der Testphase muss auch auf Fehler geachtet werden, die erst nach längerer Betriebsdauer auftreten.
- Oft in **sicherheitskritischen Bereichen** im Einsatz.
 - Fehler können neben finanziellen Schaden auch Menschenleben kosten.
- Benötigt entsprechendes **Patch-Management** und **Risk-Management**.
 - Potentielle Auswirkungen von Fehlern müssen korrekt eingeschätzt werden.
 - Fehlerbehebung muss mit entsprechender Ernsthaftigkeit betrieben werden.
 - Entsprechende Prozesse müssen etabliert werden.

Beispiel: Prusa 3D-Drucker Firmware

Platzprobleme bei Firmware-Entwicklung



- Original Prusa MK2 und MK3 3D-Drucker verwenden 8-bit Mikrocontroller.
 - AVR ATmega 2560: 8KB SRAM, 256 KB Flash
 - USB-to-Serial Chip: AVR ATmega 32u4
- Schon bestehende Features belegen einen Großteil des Programmspeichers
- Implementierung neuer Features wird dadurch zur Herausforderung
 - Zuerst muss entsprechend Platz im Programmspeicher geschaffen werden.
 - *“Eine unserer Lieblingsmethoden, um Platz zu sparen, ist das **Umschreiben eines älteren Codes in eine intelligenter, schlankere Form.**”¹*

¹https://blog.prusa3d.com/de/prusa-dev-diaries-4-das-besten-aus-der-8-bit-firmware-herausholen_51082/

- Muss **stark begrenzte Hardwareressourcen** (Speicher, Rechenleistung, Energie) optimal nutzen können
 - Ressourcensparsamkeit ein wichtiges Thema
- Oft **hardwarenahe, low-level Programmierung**
 - Oft kein bzw. sehr schlankes Betriebssystem
- Hohe Anforderungen an die **Verfügbarkeit, Zuverlässigkeit und Betriebssicherheit**
 - System muss in "Echtzeit" reagieren können
 - Gefährdung von Menschenleben muss ausgeschlossen werden
 - Wartung oft nicht einfach möglich.

Beispiel: Sonos S1

End-of-Life für Sonos S1 Systeme



- Sonos S1 Systeme haben seit Mai 2020 den End-of-Life Status erreicht.
 - Funktionieren prinzipiell noch weiter.
 - Aber bekommen keine weiteren Updates und neuen Features,
 - sondern nur mehr Bugfixes und Security Patches.
- Komplette Trennung zwischen Alt- und Neu-Geräte
 - Separate Apps für Alt- und Neu-Geräte.
 - Interoperabilität nicht mehr gegeben.
- Solche Entscheidungen können auch technischen Gründe haben.
 - Z.B. Hardware nicht mehr leistungsfähig genug für neue Features.

Bei **Massenprodukten** werden **Hardwarekosten möglichst gering** gehalten:

- Umso größer die Stückzahl, desto geringer wirken sich Entwicklungskosten auf den Stückpreis aus.
- Kosten für Hardware wirken sich aber direkt aus.
- Daher wird höherer Entwicklungsaufwand in Kauf genommen, um Software für beschränkte Hardware zu optimieren.
- Es wird dabei versucht, die **Hardware bis auf das Äußerste auszureizen**.
- Zu einem späteren Zeitpunkt hardware-intensive neue Features hinzuzufügen ist dann oft nicht mehr möglich.

Eingebettete Systeme haben oft ein sehr beschränktes Energiebudget:

- **Batteriebetriebene** Systeme sollen möglichst lange durchhalten können.
 - Energieverbrauch muss oft im Milliwattbereich liegen.
 - Software kann in diesem Bereich einen sehr großen Einfluss haben.
- "Verbrauchte" Energie muss in Form von **Wärme** wieder abgeführt werden.
 - Hitzeableitung über Kühlungssysteme oft nur sehr aufwendig oder gar nicht möglich.
 - Oft extremere Betriebsbedingungen als bei "normalen" Computersystemen (z.B. bis zu 85°C Außentemperatur).

Eigenschaften von eingebetteter Software

Verlässlichkeit



“Verlässlichkeit” kennt drei Teilgebiete:

- **Zuverlässigkeit**

- Wahrscheinlichkeit, dass eine vorgegebene Funktion unter vorgegebenen Bedingungen für einen bestimmten Zeitraum durchgängig fehlerfrei erfüllt wird.
- Metrik: Mean Time Between Failures (MTBF) - Der Erwartungswert der Betriebsdauer zwischen zwei aufeinanderfolgenden Ausfällen.
- Höhere Anforderungen an eingebettete Systeme als an “normale” Computersysteme.

wie lange funktioniert ohne dass Ausfall passiert

- **Verfügbarkeit**

- Anteil an der Betriebsdauer innerhalb dessen das System seine Funktion erfüllt.
- Verfügbarkeit = $\frac{\text{Gesamtzeit} - \text{Ausfallzeit}}{\text{Gesamtzeit}}$

Wie lang dauert es bis ich diesen Ausfall gefixt habe
Prozentzahl wann mein System funktioniert

• Funktionale Sicherheit

- Nicht akzeptierbare Risiken (z.B.: Gefährdung von Menschenleben) müssen weitgehend ausgeschlossen sein.
- Andere Risiken (z.B.: Verletzungsrisiko) müssen auf ein akzeptierbares Maß reduziert werden.
- Risikomaß = Eintrittswahrscheinlichkeit · Kosten
- Bei Sicherheitskritischen Systemen (z.B.: Autopilot bei Flugzeugen) ein sehr wichtiges Thema.

Definition (Echtzeit, DIN 44300)

Unter **Echtzeit** versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten **ständig betriebsbereit** sind, derart, dass die Verarbeitungsergebnisse **innerhalb einer vorgegebenen Zeitspanne verfügbar** sind.

Eingebettete Systeme müssen häufig strikten Zeitanforderungen genügen:

- Nichteinhaltung kann die funktionale Sicherheit gefährden.
- z.B. Airbag: Zwischen Aufprall und Auslösung liegen gerade mal 10-15 Millisekunden.
- siehe nächstes Webinar

Eigenschaften von eingebetteter Software

Aufbau eingebetteter Systeme

Softwareentwicklung für eingebettete Systeme

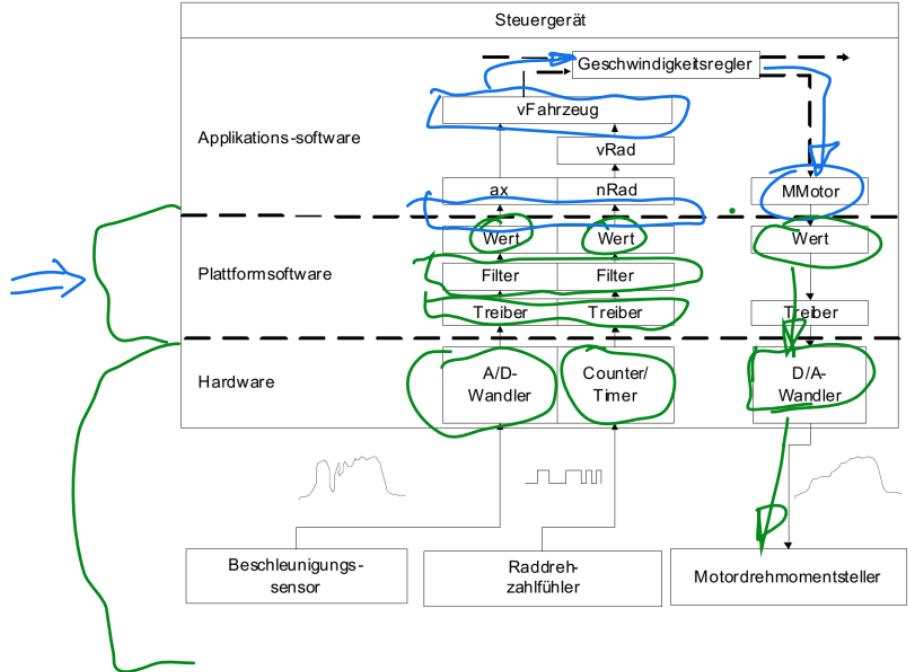
Grobe Einteilung in zwei elementare Bestandteile:

- **Plattformsoftware:** z.B. Betriebssystemen, MacOS, Linux, Windows
 - Hardware-Abstraktionsschicht
 - Beinhaltet Schnittstellen und Treibersoftware für die jeweilige Hardware.
 - Kapselung von low-level Aufgaben und Bereitstellung entsprechender Services
 - Übernimmt betriebssystemspezifische Aufgaben
- **Applikationssoftware:**
 - Setzt auf der Plattformsoftware auf
 - Möglichst Hardware-unabhängig
 - Implementiert die eigentliche Funktionalität

Aufbau eingebetteter Systeme

Plattform- und Applikationssoftware

Möglichst leicht codierbar und auf die andere Systeme transportierbar



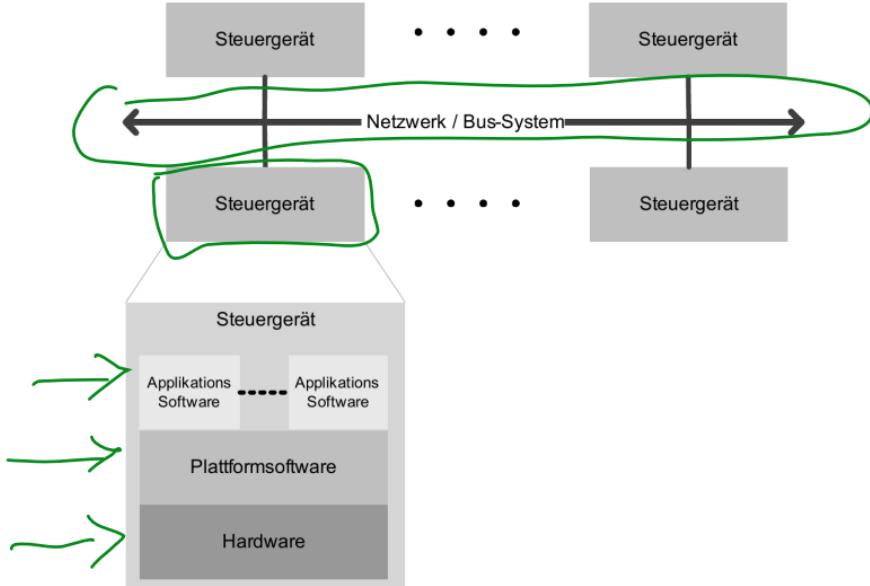
Eingebettete Systeme bestehen oft aus **mehreren unabhängigen Teilsystemen**:

- Teilsysteme nennt man **Steuergerät** oder **Electronic Control Unit (ECU)**
 - Eigenständige Rechenknoten
- Durch ein Kommunikationsnetzwerk miteinander verbunden
 - I.d.R. als Bussystem ausgelegt
 - siehe nächstes Webinar
- **Eindeutig definierte Schnittstellen** zwischen Teilsystemen
 - “Teile-und-Herrsche”-Prinzip
- Teilsysteme oft von unabhängigen Teams entwickelt und implementiert

unabhängige Teilsysteme

Aufbau eingebetteter Systeme

Eingebettetes System als Verteiltes System



Eigenschaften von eingebetteter Software

Aufbau eingebetteter Systeme

Softwareentwicklung für eingebettete Systeme

Im Großen und Ganzen nicht viel Unterschied zur "normalen"

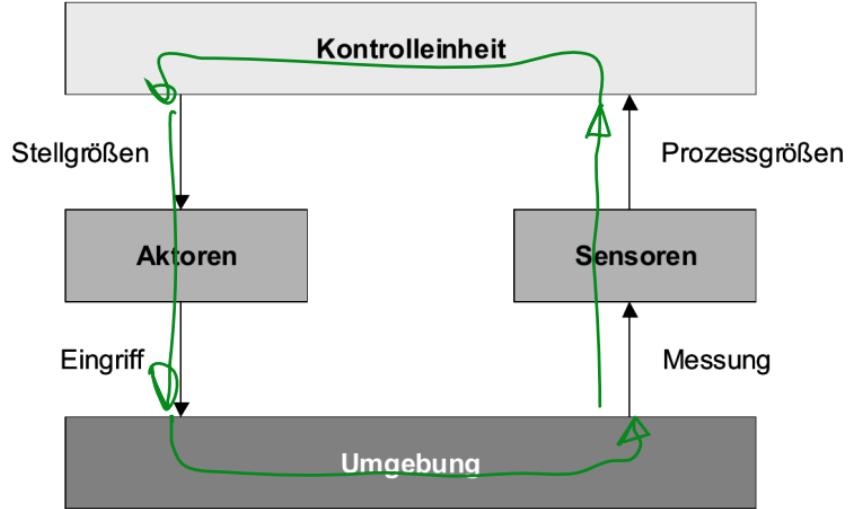
Softwareentwicklung:

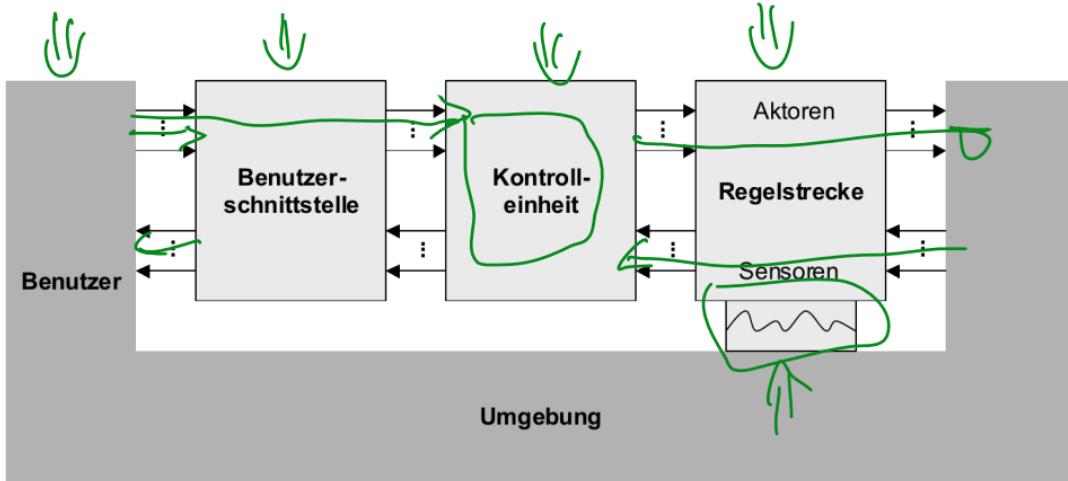
- Dieselben Vorgehensmodelle
 - z.B.: Wasserfallmodell, V-Model
- Dieselben Werkzeuge
 - z.B.: IDE, Versionsverwaltung

Aber im Kleinen große Unterschiede aufgrund der Besonderheiten:

- Andere Optimierungsrichtlinien
 - z.B. Platzbedarf und Energie statt Geschwindigkeit
 - Verwendung des Compilerflags ~~-Os~~ statt ~~-O2~~ / ~~-O3~~.
weil viel zu viel Programmspeicher verbraucht wird
- Andere Vorangehensweisen und Best Practices
 - z.B.: Zuviel Objektorientierung verursacht zuviel Overhead

-Os
s - steht für size





Wie können Sensorwerte bzw. das Eintreten von Ereignissen abgefragt werden?

Zwei prinzipielle Möglichkeiten:

- **Polling**

- Zyklische Abfrage des entsprechenden Werts.
- z.B. Ist ADC fertig: `while (ADCSRA & (1 << ADSC));`

- **Ereignissteuerung**

- Wert muss nicht ständig abgefragt werden, sondern Programm wird aktiv bei Änderungen informiert.
- Beim Eintreten eines Ereignisses wird meist eine entsprechende Funktion mit entsprechenden Parametern aufgerufen (die sog. **Callback-Funktion**).

- Das Kontinuierliche Abfragen von Sensoren o.ä. ohne Pause nennt man Busy-Waiting (auch Busy-Looping oder Spinning genannt).

Busy-Waiting auf das Ergebnis einer ADC-Umwandlung

```
while (ADCSRA & (1 << ADSC)) {}
```

- Busy-Waiting zählt zu den sog. Anti-Pattern.
 - Sollte man normalerweise nicht verwenden.
 - Verschwendet Rechenzeit bzw. Energie, die man sinnvoller einsetzen könnte.

- Prinzipiell sollte man bei Polling immer **kurze Pausen** einplanen.
 - Während der Pause kann man andere Aufgaben erledigen,
 - bzw. den Prozessor schlafen legen um Energie zu sparen.

Polling mit Pause

```
while (Ereignis noch nicht eingetreten) {  
    sleep(250); // Schlafe 250 ms  
}
```

- In manchen Situationen macht aber Busy-Waiting durchaus Sinn.
 - Wenn die Wartezeit sehr kurz ist (z.b. eine normale ADC-Umwandlung benötigt 13 ADC-Taktzyklen, bei 125 kHz sind das 104 μ s).
 - Wenn sehr schnell auf Ereignisse reagiert werden muss.

- “Klassische” Programme sind meist eine Liste von Befehlen, die von Anfang bis Ende abgearbeitet werden.
 - Man startet beim Problem.
 - Programmbefehle werden Schritt für Schritt abgearbeitet.
 - Am Ende hat man die Lösung und das Programm beendet sich.
- Eingebettete Systeme hingegen warten auf das Eintreffen eines Ereignisses, behandeln dieses und warten dann auf das nächste Ereignis.
 - z.B. Button wurde gedrückt.
 - z.B. Temperatur überschreitet Sollwert.
 - z.B. Bewegungsmelder registriert Bewegung.

- Daher kommt häufig Ereignisorientierte Programmierung zum Einsatz.
 - Kein lineares Programm das von vorne nach hinten durchlaufen wird.
 - Meist ist kein Ende in Sicht, sondern das Programm läuft in einer Endlosschleife.
 - Stattdessen werden beim Eintritt eines Ereignisses sog. Ereignisbehandlungs Routinen (englisch Event Handler) ausgeführt.
 - Konzeptionell starke Ähnlichkeit mit Exception-Handling. ähnlich zu try catch block

- Oft kommt die sog. **Inversion of Control** (*Umkehrung der Steuerung bzw. Steuerungsumkehr*) zum Einsatz.
 - Ist ein **Design Pattern** für das Design von Anwendungen.
 - Programmierer:innen implementieren nicht selber den Hauptprogrammfluss.
 - Stattdessen implementieren sie Code-Schnipsel, die auf Ereignisse reagieren.
 - Der Hauptprogrammfluss wird meist von generischen Frameworks vorgegeben.
- Neben eingebetteten Systemen findet man dieses Design Pattern sehr oft bei **GUI-Programmen**.
 - GUI-Programme reagieren meist auf Benutzereingaben.

Umsetzung mittels Event-Loop

```
void customEventHandler(Event e) {...}  
  
int main() {  
    registerEventHandler(customEventHandler);  
    ...  
    while (Event e = getNextEvent()) {  
        executeEventHandler(e);  
    }  
}
```

A hand-drawn style diagram highlights specific parts of the code. A blue curly brace encloses the entire definition of the `customEventHandler` function. A green curly brace encloses the `main` function body, specifically the registration of the event handler and the main loop structure.

- Auf unterster Ebene kommen **Interrupts** zur Ereignissesteuerung zum Einsatz.
 - Sind in Hardware (meist CPU) implementiert.
 - Kann man sich wie "Hardware"-Exceptions vorstellen.
 - Unterbrechen den aktuellen Programmfluss und springen zu einer fixierten Befehlsadresse.
 - Sind konfigurierbar und können ein- und ausgeschalten werden.
- Werden bei verschiedensten internen und externen Ereignissen ausgelöst.
 - z.B. bei Fehlerzuständen (Division by Zero, ...)
 - z.B. regelmäßige Interrupts für Zeitmessungen.
 - z.B. Verfügbarkeit von Daten (ADC-Umwandlung ist fertig, ...).
 - z.B. Veränderung von I/O-Pins (Spannung fällt von HIGH nach LOW, ...).
 - ...

z.B. millis() funktion

Ablauf Interrupt:

- ① Programm wird normal ausgeführt.
- ② Beim Auftreten des Interrupts wird die Ausführung des Programms gestoppt.
 - Kann an beliebiger Stelle unterbrochen werden!
 - Kritische Stellen können geschützt werden, indem man Interrupts global ausschaltet (`cli()`) und danach wieder einschaltet (`sei()`)
- ③ Interrupt-Handler (die sog. [Interrupt Service Routine](#)) wird ausgeführt.
- ④ Nach Beenden der Interrupt Service Routine wird die Ausführung des normalen Programms fortgesetzt.

Interrupts

Interrupt Service Routine



- Interrupt Service Routinen sind **zeitkritisch!**
 - Meist sind andere Interrupts währenddessen ausgeschalten.
 - Interrupt-Anforderungen könnten dann verlorengehen oder verspätet auslösen.
 - Deshalb so wenig Zeit wie möglich in Interrupt-Service-Routinen verbringen!
- Die Behandlung von Interrupts wird deswegen oft ausgelagert.
 - Z.B. In der Interrupt Service Routine nur notwendige Daten zwischenspeichern und ein Flag setzen.
 - In der Hauptschleife dann das Flag überprüfen und dort die eigentliche Ereignisbehandlung durchführen.

Interrupt Service Routine

Auslagern der Interrupt-Behandlung



```
bool interruptFlag = false;  
int interruptData = 0;  
  
void interrupt_handler() {  
    interruptFlag = true;  
    interruptData = getInterruptData();  
}  
  
void main() {  
    while (true) {  
        if (interruptFlag) {  
            interruptFlag = false;  
            // Do something with interruptData  
        }  
    }  
}
```

Die Zeit welche wir hier verbringen muss möglichst gering sein

Vor- und Nachteile Polling:

- + Braucht keine Hardware-Unterstützung.
- + Einfach in Software umzusetzen.
- + Verursacht bei linearen Programmen weniger Overhead.
- Ineffiziente Ressourcennutzung.
- Oft nicht deterministisch. → ich kann nicht genau sagen wann Ereigniss eingetreten ist. Es liegt immer zwischen sleep(z.B. 250)

Vor- und Nachteile Interrupts:

- + Effiziente Ressourcennutzung.
- + Deterministisch.
- + Verursacht bei ereignis-orientierten Programmen weniger Overhead.
- Benötigt Hardware-Unterstützung.
- Schwieriger in Software umzusetzen.

Polling vs. Ereignissteuerung

Pull vs. Push



- Oft hört man auch die Begriffe **Pull** und **Push**.
 - **Pull**: Ich muss mir aktiv "etwas" holen.
 - **Push**: Sobald "etwas" verfügbar ist wird es automatisch zugestellt.
- z.B. **Push-Notifications** bei Smartphones: Benachrichtigungen werden automatisch auf das Smart Phone "gepusht".
- z.B. **Pull- vs Push-APIs**:
 - **Pull-API**: Programmierschnittstelle, die hauptsächlich auf Polling setzt.
 - **Push-API**: Programmierschnittstelle, die hauptsächlich auf Ereignissteuerung mittels Callback-Funktionen setzt.
- Diese Begriffe werden auch abseits der Softwareentwicklung verwendet:
 - **Pull- vs Push-Marketing**
 - **Pull- vs Push-Logistics**

Pull - ich hole etwas
Push - drücke etwas

Generelle Tipps und Tricks:

- Datentypen "passend" wählen
 - z.B.: Wenn Wert zwischen 0 und 100, dann `char` und nicht `int` wählen.
- Redundanzen möglichst vermeiden
 - Lieber abgeleitete Attribute mehrmals berechnen als unnötig Arbeitsspeicher verbrauchen.
- Strings im Programmspeicher statt im Arbeitsspeicher ablegen
 - Mikrocontroller sind oft Harvard-Architekturen (dazu später mehr)
 - Siehe Hands-On-Session
- Variablen/Objekte möglichst am Stack anlegen
 - Heap-Objekte brauchen Pointer, um darauf zuzugreifen.
 - Heap-Verwaltungsinformationen verbrauchen Arbeitsspeicher.
 - Speicherverbrauch schlechter nachvollziehbar.
- Ressourcensparsamere Algorithmen bevorzugen.

C++ spezifische Tipps und Tricks: dynamische Bindung vermeiden

- "virtual"-Methoden möglichst vermeiden.
 - Virtual Function Table verbraucht Arbeitsspeicher.
- Weitverzweigte Vererbungshierarchien möglichst vermeiden.
 - Unnötig vererbte Attribute verbrauchen Arbeitsspeicher
- Polymorphismus weitgehend vermeiden.
 - Benötigt "virtual"-Methoden
 - Funktioniert nur mit Pointern
 - C++-Templates bieten u.U. bessere Möglichkeiten
- C++ auf Mikrocontrollern ist (war?) im professionellen Umfeld oft verpönt.
 - Genauso wie z.B. MicroPython (Python auf Mikrocontrollern)
 - oder JavaScript for Microcontrollers (Espruino)

Generelle Tipps und Tricks:

- Mikrocontrollerkomponenten ausschalten, wenn nicht mehr benötigt.
 - z.B.: ADC ausschalten, sobald Umwandlung beendet.
- Mikrocontroller so oft wie möglich schlafen legen
 - AVR Mikrocontroller kennen mehrere "Sleep Modes".
 - "Polling mit Pausen" statt "Busy-Waiting" kann Batterilaufzeit enorm verlängern.
 - Siehe Hands-On-Session.
- Unnötige Berechnungen vermeiden
 - Wenn möglich Ereignissteuerung verwenden.
 - Ansonsten Polling-Interval entsprechend anpassen.
 - z.B.: Es reicht Button-Status alle 50 ms abzufragen.

Beispiel: Energieverbrauch bei Batteriebetrieb

- AVR ATMega32u4 Mikrocontroller wird mit AA Batterien betrieben
 - 2 x AA Batterien seriell geschalten: ~3V Spannung, ~2000mAh Kapazität
- Energiebedarf Mikrocontroller:
 - Active: ~1.2 mA Volle Leistung: CPU, Timer, Peripherie laufen. Mikrocontroller rechnet aktiv, reagiert auf Eingaben, führt Code aus.
 - Idle: ~0.3 mA CPU pausiert, aber Timer/Peripherie laufen weiter. Gut für zeitgesteuerte Wartezustände, z. B. mit Timer-Interrupts.
 - Standby: ~35 µA Fast alles abgeschaltet, nur minimale Logik (z. B. RTC oder externer Interrupt) bleibt aktiv. Ideal für schnelles Aufwachen mit minimalem Verbrauch.
 - Power-Down: ~0.3 µA Maximale Energiesparung, alles ausgeschaltet außer vielleicht ein externer Interrupt oder Watchdog. Aufwecken nur durch spezielle Ereignisse.
- Laufzeit:
 - Active: ~69 Tage
 - Idle: ~277 Tage
 - Standby: ~6.5 Jahre
 - Power-Down: ~761 Jahre

💡 Wann welchen Modus verwenden?

Ziel	Empfohlener Modus
Rechnen, Sensor lesen	Active
Auf Timer-Interrupt warten	Idle
Auf Uhrzeit oder bestimmte Pins reagieren	Standby
Lange schlafen, extrem geringer Stromverbrauch	Power-Down

Beispiel: Batteriebetriebene Fernbedienung

- Reagiert auf Benutzer:inneneingabe
 - Wenn Knopf gedrückt wird, dann wird Befehl (über IR, Bluetooth, ...) gesendet.
- Sollte möglichst lange mit einer Batterieladung auskommen.
 - Sollte sich möglichst lange im Power-Down Modus befinden
 - Und alle nicht benötigen Komponenten immer ausschalten.
- Hier bietet sich Ereignisorientierte Programmierung mit Interrupts an.

Lineares Programm mit Busy-Waiting bzw. Polling

```
int main() {
    while (true) {
        if (is_button_pressed ()) {
            send_command ();
        }
    }
}
```

- Führt kontinuierlich Befehle aus.
- Kommt nie bzw. kaum in den Power-Down Mode.

Interrupt-gesteuertes Programm

```
void interrupt_handler() {
    send_command();
}

int main() {
    enable_io_interrupt();
    while (true) {
        activate_power_down_mode();
    }
}
```

- Befindet sich den Großteil der Zeit im Power-Down Mode.
- Erst wenn tatsächlich ein Button gedrückt wird, wird Code ausgeführt.

Bei sicherheitskritischen eingebetteten Systemen:

- **Kaum agile Vorgehensweisen.**
 - Wasserfallmodell hat noch sehr große Relevanz!
- **Sehr strikte Coding Standards und Best Practices**
 - Code muss leicht lesbar und verständlich sein!
- **Viele Code Reviews**
 - Kosten pro Zeile Code sehr groß!
- **Formale Verifikation ein großes Thema**
 - Richtigkeit des Codes wird mit logischen und formalen Methoden bewiesen.
- **Codeänderungen ein sehr langer Prozess**
 - Bugs können Menschenleben kosten!

Zuverlässigkeit und Verfügbarkeit in eingebetteten Systemen:

- **Code muss robust sein!**
 - Fehler dürfen nicht sofort zu einem Absturz führen
- **Abstürze müssen zuverlässig erkannt werden!**
 - Benötigt entsprechende Mechanismen.
 - z.B. Ein selbstfahrende Auto sollte sofort gestoppt werden, wenn das Steuerprogramm abstürzt.
- Bei Abstürzen muss Verfügbarkeit schnellstmöglichst wieder hergestellt werden!
 - Braucht Mechanismus, um Mikrocontroller bzw Programme automatisch neu zu starten.

Verlässlichkeit

Totmannschalter



- Ein Totmannschalter ist eine Sicherheitseinrichtung, die die Handlungsfähigkeit von Bedienpersonal überprüfen soll.
 - z.B. muss regelmäßig ein Knopf betätigt werden um zu signalisieren, dass man noch "am Leben" ist.
- Eingebettete System kennen auch "Totmannschalter"-ähnliche Mechanismen.
- Z.B. (Hardware)-Watchdog:
 - (Hardware)-Timer zählt ständig von einem definierten Wert nach unten.
 - Sobald der Timer 0 erreicht, wird das System neu gestartet.
 - Um Neustart zu verhindern, muss Watchdog regelmäßig resetet werden.
- Z.B. Ständiges Senden von Steuerbefehlen:
 - Steuerbefehle müssen ständig wiederholt gesendet werden.
 - Wenn der letzte Steuerbefehl zu lange her ist, wird das System sofort gestoppt.
 - Autonome Fahrzeuge und Roboter verwenden z.B. diesen Mechanismus.

3 SOFTWAREENTWICKLUNG AUF EMBEDDED SYSTEMS

▼ Multi-Tasking auf dem Arduino



Was ist Multi-Tasking auf dem Arduino?

Ein Arduino führt **nur einen Befehl zur Zeit** aus – es gibt **kein echtes Multithreading** oder parallele Prozesse wie in einem Betriebssystem.

Aber: Man kann durch clevere **zeitgesteuerte, nicht-blockierende Programmierung** mehrere Aufgaben **scheinbar gleichzeitig** erledigen.

Das nennt man **Kooperatives Multitasking**.



1. Vermeidung von delay() – Grundlage für Multitasking

Problem mit

delay()

:

- Blockiert den gesamten Codefluss
- Während des Delays kann **keine andere Aufgabe** ausgeführt werden

Lösung:

millis()

verwenden

```
unsigned long vorher = 0;  
unsigned long intervall = 1000;  
  
void loop() {  
    if (millis() - vorher >= intervall) {  
        vorher = millis();  
        // Aufgabe ausführen  
    }  
}
```

So kann man **mehrere Aufgaben parallel organisieren**, z. B.:

- LED blinken

- Sensor lesen
 - Seriell schreiben
-



2. Aufgaben trennen: State Machines

Statt alles in eine große loop() zu schreiben, wird jede Aufgabe als eigene „**State Machine**“ umgesetzt:

- Jede Aufgabe hat ihren **eigenen Timer (millis)**
- Jeder Teil läuft unabhängig in jedem Schleifendurchlauf

Beispiel: Zwei LEDs mit unterschiedlichem Blinkmuster

```
unsigned long previousMillis1 = 0;
unsigned long previousMillis2 = 0;
const long interval1 = 1000;
const long interval2 = 300;

void loop() {
    if (millis() - previousMillis1 >= interval1) {
        previousMillis1 = millis();
        digitalWrite(led1, !digitalRead(led1));
    }

    if (millis() - previousMillis2 >= interval2) {
        previousMillis2 = millis();
        digitalWrite(led2, !digitalRead(led2));
    }
}
```

✓ Kein Blocking

✓ Zwei unabhängig laufende Aufgaben



3. Zustandsspeicherung – „Finite State Machines“

Für komplexere Aufgaben z. B.:

- Menüsteuerung
- Animationssequenzen
- Serielle Kommunikation

verwaltet man **Zustände in Variablen**, um gezielt bestimmte Aktionen auszuführen.

```
enum State { IDLE, BLINKING, SENDING };
State currentState = IDLE;
```

Dann in loop():

```
switch (currentState) {
    case IDLE:
        // warten auf Input
        break;
```

```

case BLINKING:
    // LEDs an/aus mit millis()
    break;
case SENDING:
    // Daten senden
    break;
}

```



Best Practices für Arduino Multitasking

Technik	Vorteil	Anmerkung
millis() statt delay()	Nicht-blockierend	Grundvoraussetzung
Jede Aufgabe hat eigenen Timer	Trennung der Logik	Klar und wartbar
Zustände speichern (State Machines)	Strukturierter Code	Besonders für komplexe Abläufe
Keine globale Logik	Saubere, getrennte Aufgaben	Vermeidet Chaos
ISR oder Interrupts sparsam einsetzen	Für Zeitkritisches	Immer nur kurz ausführen!

⚠ Kein echtes Multithreading

- Arduino (ATmega328P) hat **kein Betriebssystem**
- Keine Prozesse, keine Threads
- Keine automatische Parallelität → nur **kooperatives Zeitmanagement**
- Aber für viele Anwendungen *völlig ausreichend*

📚 Quellen: Adafruit-Tutorials

1. [Part 1 – millis\(\) statt delay\(\)](#).
2. [Part 2 – State Machines und mehrere Aufgaben](#)
3. [Part 3 – Strukturierter Ansatz & erweiterte Steuerung](#)

✓ Zusammenfassung

Multitasking auf dem Arduino = **intelligentes Zeitmanagement**, nicht echter Parallelismus.

Durch:

- millis()-basierte Steuerung
 - klare Trennung von Aufgaben
 - Zustandsautomaten
- ... lassen sich komplexe Abläufe elegant realisieren.

▼ “Watchdog” bei Mikrocontrollern



Was ist ein Watchdog?

Ein **Watchdog-Timer** ist eine **Sicherheitsfunktion** in Mikrocontrollern, die automatisch einen **Systemneustart** auslöst, wenn sich das Programm **aufhängt** oder **nicht mehr reagiert**.

Prinzip:

- Der Watchdog ist ein **Timer**, der ständig abläuft.
 - Das Programm muss ihn regelmäßig **zurücksetzen („füttern“)**
 - Bleibt das Zurücksetzen aus → **automatischer Reset** des Mikrocontrollers.
-



Warum ist das wichtig?

- Verhindert Dauerfehler oder Totalausfälle (z. B. durch Endlosschleifen)
 - Erhöht **Verfügbarkeit und Zuverlässigkeit**
 - Wichtig bei **sicherheitskritischen oder unbeaufsichtigten Systemen**
 - Typisches Beispiel: **Totmannschalter-Ersatz**
-



Ablauf eines Watchdogs (vereinfacht)

[System läuft] → [WDT wird regelmäßig zurückgesetzt] → Alles ok

[System hängt] → [WDT nicht zurückgesetzt] → Timer läuft ab → Reset



Watchdog bei AVR (z. B. ATmega328P / Arduino)

Besonderheiten:

- Läuft **unabhängig vom Hauptprogramm** (eigener Taktgeber)
 - Muss explizit **aktiviert, konfiguriert und zurückgesetzt** werden
 - Achtung: **Reset durch WDT ist "harter" Reset** – wie Power-Off
-



Beispiel (AVR-GCC)

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

int main(void) {
    wdt_enable(WDTO_2S); // Watchdog auf 2 Sekunden setzen

    while (1) {
        // Hauptcode

        wdt_reset(); // Watchdog zurücksetzen
    }
}
```

Typische Zeitstufen:

- 15 ms, 30 ms, 60 ms, ... bis 8 s (je nach Chip)
-



Sicherheitsmaßnahmen

- Initialisierung des WDT sehr wichtig (z. B. im Bootloader)
- Bei falscher Konfiguration: "**Watchdog-Reset-Schleife**"
- Auffangen über die MCUSR-Flags nach dem Neustart:

```
if (MCUSR & (1<<WDRF)) {  
    // Watchdog-Reset erkannt  
    MCUSR &= ~(1<<WDRF);  
}
```



Arduino-Hinweis

Arduino selbst **aktiviert keinen Watchdog automatisch**.

Muss per Low-Level-C-Code erfolgen oder über spezielle Libraries wie avr/wdt.h.



Arduino Resetet bei WDT ohne Vorwarnung!

Kein Schutz für offene Dateien oder Speicher.

📚 Quellen

1. [Wikipedia: Watchdog](#)
 2. [mikrocontroller.net – AVR-GCC Watchdog-Tutorial](#)
-

✓ Zusammenfassung

Merkmal	Beschreibung
Funktion	Neustart bei Hänger/Fehler
Vorteil	Höhere Zuverlässigkeit
Konfiguration	Zeitfenster + Aktivierung
Rücksetzung	per <code>wdt_reset()</code>
Risiko bei Fehlern	Endlos-Reset möglich
Anwendung	Robotik, Safety, batteriebetriebene Geräte

▼ „Sleep Mode“ bei Mikrocontrollern



Was ist der Sleep Mode?

Ein **Sleep Mode** (Schlafmodus) ist ein Stromsparmodus eines Mikrocontrollers.

Darin wird der Prozessor **ganz oder teilweise abgeschaltet**, um **Energie zu sparen**, während bestimmte Module (Timer, Interrupts, etc.) ggf. **weiterlaufen**.



Warum Sleep Modes verwenden?

- **Längere Batterielaufzeit** bei Embedded Systems
- **Weniger Wärmeentwicklung**
- In vielen Anwendungen (z. B. Sensor-Nodes, Fernbedienungen, IoT) ist der Prozessor **die meiste Zeit untätig** → Schlafen spart massiv Strom!



Sleep-Modi bei AVR-Mikrocontrollern (z. B. ATmega328P)

Modus	CPU	Peripherie	Stromverbrauch	Aufweckbar durch
Active	✓	✓	Hoch (~1 mA)	–
Idle	✗	✓	Mittel (~0.3 mA)	Timer, UART, INT
ADC Noise Reduction	✗	✓ (nur ADC)	Niedrig	ADC
Power-Down	✗	✗	Sehr gering (~0.3 µA)	Externer INT, WDT
Power-Save	✗	✓ (Timer2)	Sehr gering	Timer2, INT
Standby/Extended	✗	⚠ Teils aktiv	Sehr gering	INT, Timer



Ablauf zur Nutzung eines Sleep Modes

1. **Modus setzen** mit `set_sleep_mode(...)`
2. **Sleep aktivieren** mit `sleep_enable()`
3. **Interrupts erlauben** mit `sei()`
4. **Prozessor schlafen legen** mit `sleep_cpu()`
5. **Nach dem Aufwachen:** `sleep_disable()` aufrufen (optional)



Beispiel (AVR-C mit <avr/sleep.h>)

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

void setup() {
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    sleep_enable();
    sei(); // Interrupts erlauben
}

void loop() {
    sleep_cpu(); // Mikrocontroller schlafen legen
```

```
// Wacht durch Interrupt wieder auf  
}
```



Aufwecken aus dem Sleep

Nur bestimmte Quellen können je nach Modus den Mikrocontroller wieder aufwecken:

- **Externe Interrupts** (z. B. Taster)
- **Timer** (z. B. mit Timer2 im Power-Save)
- **Watchdog Timer** (regelmäßiger Timeout)
- **ADC** (bei ADC Noise Reduction)



Wichtige Hinweise

- Nicht jeder Interrupt kann aus jedem Sleep-Mode wecken!
- Im Power-Down funktioniert z. B. nur:
 - **INT0 / INT1**
 - **Watchdog**
- Timer 0 und 1 schlafen mit – **kein millis() oder PWM in Power-Down**



Beispiel: Batterie-Fernbedienung

```
void setup() {  
    attachInterrupt(digitalPinToInterrupt(2), wakeUp, LOW);  
}  
  
void loop() {  
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);  
    sleep_enable();  
    sleep_cpu(); // schläft hier  
    sleep_disable();  
  
    send_command(); // nach dem Aufwachen  
}  
  
void wakeUp() {  
    // leer: Interrupt weckt nur  
}
```

 So bleibt die CPU **die meiste Zeit ausgeschaltet** → Batterien halten Jahre!



Typische Anwendungen

- Wetterstationen
 - Bewegungsmelder
 - drahtlose Sensoren
 - Datenlogger
 - Fernbedienungen
-

Quellen

1. mikrocontroller.net – Sleep Modes
 2. elektro.turanis.de – Praxisprojekt mit Sleep
-

Zusammenfassung

Vorteil	Wirkung
Weniger Energieverbrauch	längere Batterielaufzeit
Weniger Wärme	stabilerer Betrieb
Flexible Energiemodi	angepasst an Anwendung
Ereignisgesteuertes Aufwachen	hohe Effizienz + Reaktionsfähigkeit

▼ Entprellung (Debouncing)



Was ist Entprellung?

Entprellung bezeichnet Maßnahmen zur **Vermeidung von Störsignalen**, die beim **mechanischen Schalten** (z. B. von Tastern oder Relais) entstehen.



Problem:

- Mechanische Kontakte "prellen" beim Schließen/Öffnen → mehrere elektrische Impulse in Millisekunden
 - Ohne Entprellung wird z. B. ein Knopfdruck mehrfach erkannt ⇒ **Fehlverhalten**
-



Typische Symptome ohne Entprellung:

- LED flackert beim Tastendruck
 - Menü springt mehrere Einträge weiter
 - Zähler zählt mehrfach statt einmal
-



Lösungen: Entprellung in Hardware oder Software



1. Hardware-Entprellung

Schaltung mit Widerstand & Kondensator (RC-Glied)

Vorteil	Nachteil
Schnell, unabhängig vom Code	Zusätzliche Bauteile nötig

Typische Werte:

- $R = 10\text{k}\Omega$, $C = 100\text{nF} \rightarrow \text{Entprellzeit} \sim 1\text{ ms}$

💡 Ideal für:

- Schaltung mit wenig Ressourcen
- hohe Sicherheit (z. B. Not-Aus)



2. Software-Entprellung

Variante A:

Zeitverzögerung (delay-based)

```
if (digitalRead(tasterPin) == LOW) {
    delay(30); // 30 ms warten
    if (digitalRead(tasterPin) == LOW) {
        // gültiger Tastendruck
    }
}
```

✓ Einfach

✗ Blockierend (ungeeignet für Multitasking)

Variante B:

Nicht-blockierende Entprellung mit millis()

```
unsigned long lastDebounceTime = 0;
const int debounceDelay = 30;
bool lastState = HIGH;
bool confirmedState = HIGH;

void loop() {
    bool current = digitalRead(tasterPin);

    if (current != lastState) {
        lastDebounceTime = millis();
    }

    if ((millis() - lastDebounceTime) > debounceDelay) {
        if (current != confirmedState) {
            confirmedState = current;
            if (confirmedState == LOW) {
                // Taste wurde gedrückt
            }
        }
    }
}
```

```
lastState = current;  
}
```

- ✓ Nicht-blockierend
- ✓ Gut für Multitasking (z. B. LEDs, Sensoren gleichzeitig)
- ✗ Etwas mehr Code

Variante C:

Zustandsautomat (State Machine)

→ Für komplexe Eingabelogik (z. B. lang/kurz drücken, doppel-klick)



Vergleich: Hardware vs Software

Kriterium	Hardware	Software
Einfachheit	✓ bei 1–2 Tastern	✓ bei vielen Tastern
Flexibilität	✗ kaum anpassbar	✓ parametrierbar
Ressourcen	✗ braucht Bauteile	✓ braucht nur Code
Genauigkeit	✓ stabil	✓ bei gutem Timer

💡 Anwendungstipp

Vermeide delay() in produktiven Systemen!

Nutze:

- millis()-basierte Entprellung oder
- externe Hardware bei sicherheitskritischen Anwendungen.

📚 Quelle

mikrocontroller.net – Entprellung

✓ Zusammenfassung

Ziel	Lösung
Taste 1× statt 5× erfassen	Entprellung nötig
Schnell, stabil	RC-Glied (Hardware)
Flexibel, Multitasking	millis()-basiert (Software)
Kein Blockieren	Kein delay() verwenden

▼ Timer



Was ist ein Timer im Mikrocontroller?

Ein **Timer** (auch „Zähler“, engl. *Timer/Counter*) ist ein **Hardwaremodul**, das automatisch **zählt** – entweder:

- auf Basis eines **internen Takts** (z. B. Systemtakt), oder

- bei **externen Ereignissen** (Counter-Modus)



Grundprinzip

Ein Timer zählt **Schritt für Schritt nach oben**, typischerweise bis:

- er **überläuft** (z. B. von 255 auf 0 bei 8 Bit), oder
- er einen **Vergleichswert (Compare Match)** erreicht.

Dann kann ein **Interrupt ausgelöst** werden.



Komponenten eines AVR-Timers (z. B. ATmega328P)

- **TCNTx**: Zählregister (aktueller Stand)
- **OCR_x**: Compare-Register (Vergleichswert)
- **TCCR_{nA/B}**: Steuerregister (Modus, Prescaler)
- **TIFR_n, TIMSK_n**: Flags / Interrupt-Steuerung



Modi eines Timers

Modus	Beschreibung
Normal	Zählt bis Überlauf → löst ggf. Overflow-Interrupt aus
CTC	Clear Timer on Compare Match (bei OCR _x = TCNT _x)
PWM	Erzeugt Rechtecksignale mit variablem Tastverhältnis



Prescaler (Vorteiler)

Timer arbeiten nicht direkt mit dem CPU-Takt (z. B. 16 MHz), sondern oft mit **verzögertem Takt** durch Prescaler:

- Typische Werte: 1, 8, 64, 256, 1024
- z. B.: 16 MHz / 64 = **250 kHz Timer-Takt**

Dadurch lassen sich z. B. **1ms-Interrupts** generieren, selbst mit 8-Bit Timern.



Beispiel: 1Hz Interrupt mit Timer1 (AVR)

```
void setupTimer1() {
    cli(); // Interrupts deaktivieren
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1 = 0;
```

```

OCR1A = 15624;           // 1 Hz bei 16 MHz, Prescaler 1024
TCCR1B |= (1 << WGM12); // CTC-Modus
TCCR1B |= (1 << CS12) | (1 << CS10); // Prescaler 1024
TIMSK1 |= (1 << OCIE1A); // Compare Interrupt aktivieren
sei();                  // Interrupts aktivieren
}

ISR(TIMER1_COMPA_vect) {
    // Wird 1x pro Sekunde ausgeführt
}

```



Zeitberechnung:

$$T_{\text{Overflow}} = \frac{\text{Prescaler} \cdot (\text{MaxZähler} + 1)}{\text{CPU-Takt}}$$

Beispiel:

- 8-Bit Timer (0...255), Prescaler 64, Takt 16 MHz

→

$$T = \frac{64 \cdot 256}{16 \times 10^6} = 1.024 \text{ ms}$$



Was kann man mit Timern tun?

- **Blink-LEDs** unabhängig von `delay()`
- **PWM-Signale** erzeugen (z. B. für Dimmer, Motorsteuerung)
- **Zeitstempel / Echtzeituhr**
- **Software-Timer**: Aufgaben zyklisch ausführen
- **Encoder-Eingänge zählen** (Counter-Modus)
- **Sampling-Raten** definieren (z. B. für ADC)

Quelle

[Heise-Blog: Timer, Counter und Interrupts](#)

Zusammenfassung

Element	Beschreibung
Timer	zählt mit Systemtakt oder externem Takt
Prescaler	verlangsamt Takt für längere Zeitintervalle
Modus CTC	Zählt bis Vergleichswert, dann Reset
Interrupts	ermöglichen zeitgesteuerte Aktionen
Anwendung	PWM, Blinken, Scheduling, Messung



ECHTZEITSYSTEME UND KOMMUNIKATIONSNETZWERKE

RECHNERSTRUKTUREN & EMBEDDED SYSTEMS

MATTHIAS JANETSCHKE
SS 2025

Inhaltsverzeichnis



Echtzeitsysteme

Kommunikationsnetzwerke

Hands-On Session: Temperatursensor TCN75A

Inhaltsverzeichnis



Echtzeitsysteme

Kommunikationsnetzwerke

Hands-On Session: Temperatursensor TCN75A

Definition (Echtzeit, DIN 44300)

Unter **Echtzeit** versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten **ständig betriebsbereit** sind, derart, dass die Verarbeitungsergebnisse **innerhalb einer vorgegebenen Zeitspanne verfügbar** sind.

Eingebettete Systeme müssen häufig strikten Zeitanforderungen genügen:

- Nichteinhaltung kann die funktionale Sicherheit gefährden.
- z.B. Airbag: Zwischen Aufprall und Auslösung liegen gerade mal 10-15 Millisekunden.

Man unterscheidet zwischen

- **Harten Echtzeitsystemen**

- Garantiert, dass die definierte Reaktionszeit **niemals** überschritten wird.
- Nichteinhaltung hat oft gravierende Auswirkungen (z.B. Menschen kommen zu Schaden).
- Sicherheitskritische Systeme sind i.d.R. immer harte Echtzeitsysteme.

- **Weichen Echtzeitsystemen**

- Die definierte Reaktionszeit wird nur **statistisch** garantiert (best effort).
- Im statistischen Mittel wird die Reaktionszeit eingehalten, es gibt aber Ausreißer.
- Nichteinhaltung hat keine gravierenden Auswirkungen (z.B. Bild- oder Tonaussetzer bei Multimediasystemen)
- Nicht sicherheitskritische Systeme oft nur als weiche Echtzeitsysteme realisiert.

Echtzeitsysteme

Paradigmen



Bei der Realisierung unterscheidet man zwischen zwei Paradigmen

- **Zeitgesteuerte Systeme**

- System arbeitet mit einem **fest vorgegebenen Zeitplan**.
- Jeder Aufgabe wird ein Zeitslot zugeteilt.
- Ereignisse werden meist mittels Polling abgefragt.

- **Ereignisgesteuerte Systeme**

- Es wird schnellstmöglichst auf ein (**externes**) **Ereignis reagiert**.
- Wenn Ereignis eintritt, wird meist ein Interrupt ausgelöst und die Interrupt Service Routine gestartet.

Paradigmen

Zeitgesteuerte Systeme



Vorteile:

- Gut planbar (sofern Bearbeitungszeit der Aufgaben bekannt).
- Systemressourcen können effizient genutzt werden.
- Systemüberlastung kann weitgehend ausgeschlossen werden.
- Mehrere gleichzeitige Ereignisse sind kein Problem.



Nachteile:

- Nicht deterministisch (Man weiß nicht, wann genau reagiert wird, nur in welchen Zeitraum).
- Höherer Planungsaufwand.

Vorteile:

- Deterministisch (Zeit zwischen Ereignis und Reaktion kann meist genau bestimmt werden).
- Es kann mit sehr geringem Zeitverlust reagiert werden.
- Intuitiv implementierbar.

Nachteile:

- Schlecht planbar.
- Mehrere gleichzeitige Ereignisse können zur Systemüberlastung führen.
- Systemressourcen nicht immer effizient nutzbar, da Leistungsreserven eingeplant werden müssen.

Echtzeitsysteme

Prozessbasierte Systeme



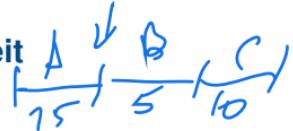
Zeitslots



Ist ein Spezialfall der Zeitgesteuerte Systeme

- Auf komplexen Systemen laufen i.d.R. gleichzeitig mehrere Prozesse ab.
 - Oft haben diese Prozesse auch unterschiedliche Prioritäten.
- Betriebssystem (bzw. der Scheduler) weist diesen Prozessen Zeitslots zu.
 - Im Grunde ein zeitgesteuertes System.
 - Höhere Prioritäten werden bevorzugt
- Normale Betriebssysteme erfüllen i.d.R. schon weiche Echtzeit.
 - Solange die Prioritäten passen.
- Harte Echtzeitbetriebssysteme unterscheiden sich oft nicht wesentlich von normalen Betriebssystemen.
 - Verwenden anderen Scheduler.
 - Verwenden deterministische Locking Mechanismen.
 - Granularere Kernel-Locks und mehr Preemption-Möglichkeiten im Kernel.
 - Oft kann aber harte Echtzeit nur für höhere Prioritäten sichergestellt werden!

Name	PID	Benutzername	CPU	Arbeitspeicher	Basispriorität	UAC-Virtualisierung
amdRendrSr.exe	2204	SYSTEM	00	1.712 K	Normal	Nicht zugelassen...
ApplicationFrameHo... AppVClient.exe	9852 4904	MCI SYSTEM	00 00	8.096 K 15.168 K	Normal Normal	Nicht zugelassen...
AppVStreamingUX.exe	7444	MCI	00	27.312 K	Normal	Nicht zugelassen...
armsvc.exe	2812	SYSTEM	00	804 K	Normal	Nicht zugelassen...
atiexec.exe	5852	SYSTEM	00	2.332 K	Normal	Nicht zugelassen...
atiexecxx.exe	2224	SYSTEM	00	1.024 K	Normal	Nicht zugelassen...
audiogd.exe	3332	Lokaler Di...	01	6.928 K	Normal	Nicht zugelassen...
connectdetector.exe	9976	MCI	00	1.760 K	Normal	Nicht zugelassen...
csrss.exe	676	SYSTEM	00	1.200 K	Normal	Nicht zugelassen...
cssrs.exe	816	SYSTEM	00	1.776 K	Normal	Nicht zugelassen...
ctfmon.exe	8744	MCI	00	4.860 K	Hoch	Nicht zugelassen...
dllhost.exe	10720	SYSTEM	00	1.732 K	Normal	Nicht zugelassen...
dllhost.exe	7096	MCI	00	2.504 K	Normal	Nicht zugelassen...
dwm.exe	1540	DWM-1	00	527.484 K	Hoch	Nicht zugelassen...
ETDService.exe	3076	SYSTEM	00	848 K	Normal	Nicht zugelassen...
explorer.exe	8096	MCI	00	48.028 K	Normal	Nicht zugelassen...
explorer.exe	8348	MCI	00	22.700 K	Normal	Nicht zugelassen...
firefox.exe	6188	MCI	01	448.632 K	Echtzeit	Nicht zugelassen...
firefox.exe	6900	MCI	02	390.848 K	Normal	Nicht zugelassen...
firefox.exe	9456	MCI	00	7.404 K	Normal	Nicht zugelassen...
firefox.exe	9980	MCI	00	32.328 K	Niedrig	Nicht zugelassen...
firefox.exe	9064	MCI	00	69.596 K	Normal	Nicht zugelassen...
firefox.exe	10372	MCI	00	4.840 K	Normal	Nicht zugelassen...
firefox.exe	10392	MCI	05	585.224 K	Normal	Nicht zugelassen...
firefox.exe	10544	MCI	00	9.732 K	Normal	Nicht zugelassen...
firefox.exe	11144	MCI	00	7.484 K	Normal	Nicht zugelassen...
firefox.exe	11152	MCI	00	5.024 K	Normal	Nicht zugelassen...
firefox.exe	1200	MCI	00	9.664 K	Normal	Nicht zugelassen...
firefox.exe	2588	MCI	00	9.684 K	Normal	Nicht zugelassen...
fontdrvhost.exe	1084	UMFD-0	00	1.300 K	Normal	Nicht zugelassen...
HDHks.exe	5580	SYSTEM	00	2.512 K	Normal	Nicht zugelassen...
HDHks.exe	9280	MCI	00	1.360 K	Normal	Nicht zugelassen...
HDStat.exe	9520	MCI	00	16.012 K	Normal	Nicht zugelassen...
igfxCUIService.exe	2240	SYSTEM	00	1.440 K	Normal	Nicht zugelassen...



- Harte Echtzeitfähigkeit kann nur formal/mathematisch bewiesen werden.
 - Bei zeitgesteuerten Systemen Berechnung der maximalen Reaktionszeit aus bekannten Zeitplan und Bearbeitungszeit der Ereignisbehandlung.
 - Bei ereignisgesteuerten Systemen Berechnung der Reaktionszeit aus Zeit bis Interruptauslösung und Laufzeit der Interrupt Service Routine.
 - Berücksichtigung gleichzeitiger Ereignisse durch Berechnung des Worst Case.
- Weiche Echtzeitfähigkeit wird durch statistische Auswertungen bewiesen.
 - Zeit von Ereignis bis Reaktion wird durch viele Versuche bestimmt.
 - Danach Bestimmung von Mittelwert und Standardabweichung und Analyse der Ausreißer.

Echtzeitsysteme

Kommunikationsnetzwerke

Hands-On Session: Temperatursensor TCN75A

Kommunikationsnetzwerke

Definition



Definition (Kommunikationsnetzwerke)

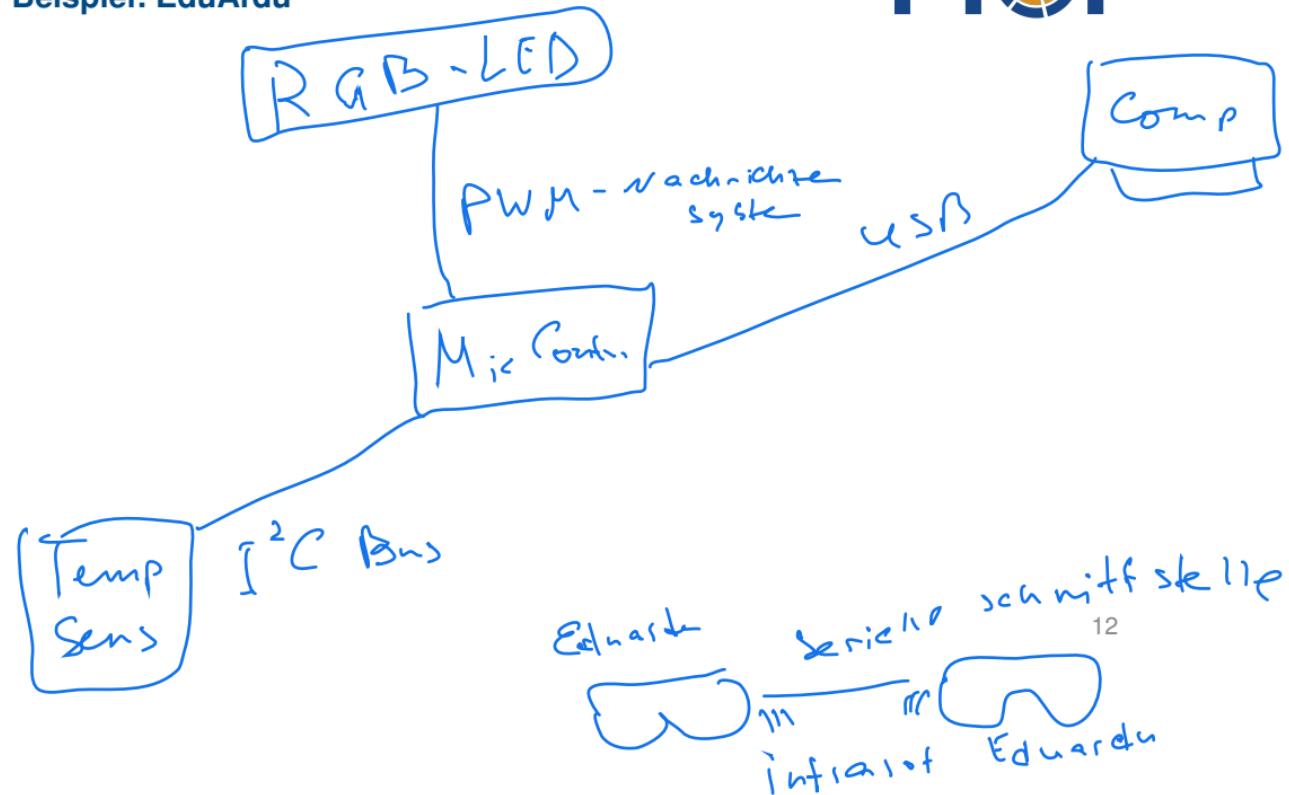
Ein Kommunikationsnetzwerk ist eine Einrichtung bzw. Infrastruktur zum Austausch von Nachrichten zwischen mehreren Teilnehmer:innen.

Für eingebettete Systeme sollten folgende Eigenschaften erfüllt werden:

- Sicherheit (soll keinen Schaden verursachen)
- Zuverlässigkeit (soll nicht ausfallen)
- Robustheit (soll unempfindlich gegen Störungen sein)
- Echtzeitfähigkeit (soll Nachrichten innerhalb einer definierten Zeitspanne zustellen)

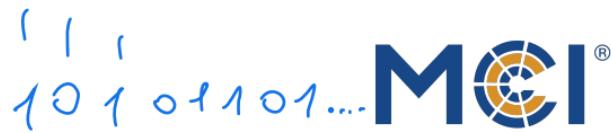
Kommunikationsnetzwerke

Beispiel: EduArdu



Kommunikationsnetzwerke

Eigenschaften



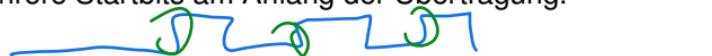
wo fängt 1 an und wo hört 1 auf
wo fängt 0 an und wo hört 0 auf

Wie werden Übertragungen synchronisiert:

Synchron: Übertragung einzelner Bits wird durch globales **Taktsignal** zeitlich synchronisiert (oft durch eigene Taktleitung).



Asynchron: Kommunikationspartner:innen arbeiten mit lokalen Taktgebern, die vor jeder Übertragung synchronisiert werden müssen. Oft geschieht das durch ein oder mehrere Startbits am Anfang der Übertragung.

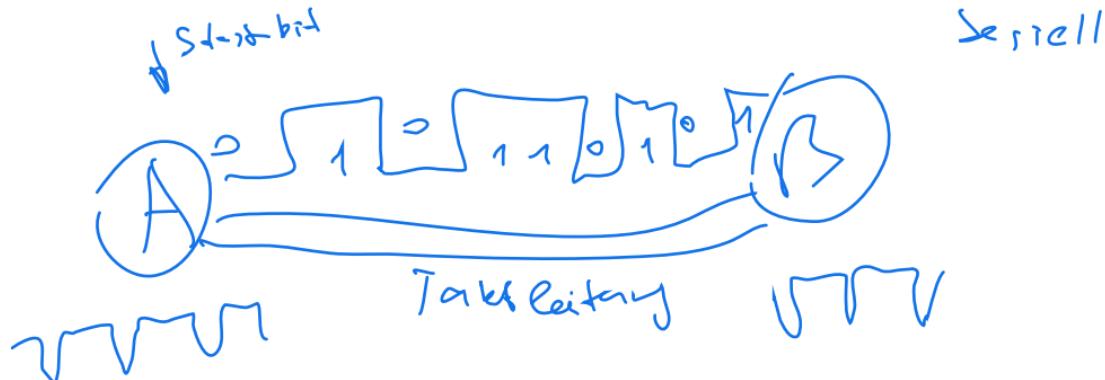


Wie werden Bits übertragen:

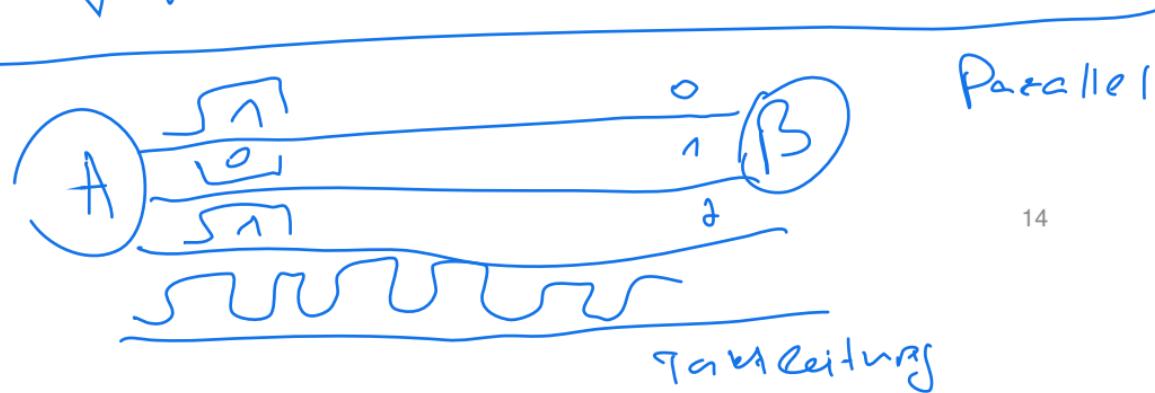
Seriell: Bits werden hintereinander über einen Datenkanal gesendet. Kann synchron oder asynchron passieren.

Parallel: Bits werden gleichzeitig über mehrere Datenkanäle gesendet. Kann nur synchron passieren.

hintereinander



Serial



Parallel

14

synchronisierung sehr schwierig

Kommunikationsnetzwerke

Eigenschaften



Kommunikationsrichtung:

Unidirektional: Es kann nur in eine Richtung kommuniziert werden.

Radio, TV, KANN MAN NICHT ZURÜCKSENDEN

Bidirektional: Es kann in beide Richtungen kommuniziert werden.

Arten von bidirektonaler Kommunikation:

Halbduplex: Es kann entweder empfangen oder gesendet werden. Funk

Vollduplex: Es kann gleichzeitig empfangen und gesendet werden.

Nachrichtenempfänger:innen:

Broadcast: Nachricht wird an alle Teilnehmer:innen gesendet.

Unicast: Nachricht wird nur an eine:n Teilnehmer:in gesendet.

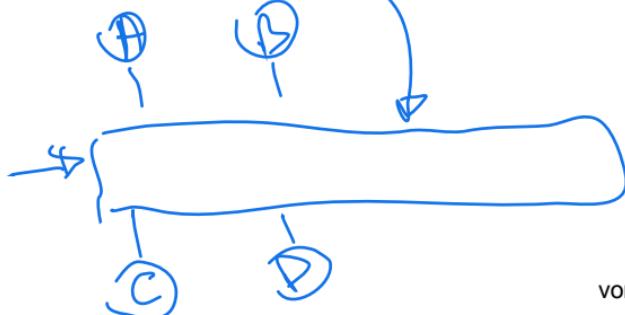
Multicast: Nachricht wird an eine definierte Gruppe von Teilnehmer:innen gesendet.

Kommunikationsnetzwerke

Arten von Kommunikationsnetzwerken

Für eingebettete Systeme gibt es zwei relevante Arten:

- Punkt-zu-Punkt Verbindungen
 - Exklusive Direktverbindung zwischen zwei Kommunikationspartner:innen.
- Bus-Systeme
 - Von mehreren Kommunikationspartner:innen gemeinsam genutzter Übertragungsweg.



von mehrere Teilnehmer möglich gleichzeitige Verwendung



nur für A und B

Arten von Kommunikationsnetzwerken

Punkt-zu-Punkt Verbindungen

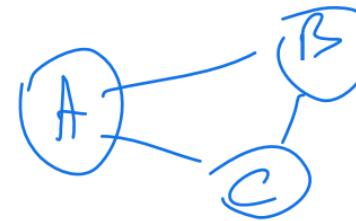


Vorteile:

- Echtzeitfähigkeit praktisch garantiert.
- Keine Störung durch andere Kommunikationspartner:innen.
- Übertragungsleistung effizient nutzbar.

Nachteile:

- Aufwendige Verkabelung
 - Pro Kommunikationspartner:in eigene Kabel notwendig.
- Schlechte Ressourcennutzung
 - Pro Kommunikationspartner:in mehrere exklusive Mikrocontroller-Pins notwendig.



Arten von Kommunikationsnetzwerken

Bus-Systeme



Vorteile:

- Effiziente Verkabelung
 - Kommunikationsteilnehmer:innen nutzen gemeinsam dieselben Kabel
- Effiziente Ressourcennutzung
 - Kommunikationsteilnehmer:innen nutzen gemeinsam dieselben Mikrocontroller-Pins.

Nachteile:

- Ohne weitere Maßnahmen nicht echtzeitfähig.
- Störung durch andere Kommunikationspartner:innen möglich.
- Übertragungsleistung nicht immer effizient nutzbar.

Bus-Systeme dominieren in eingebetteten Systemen:

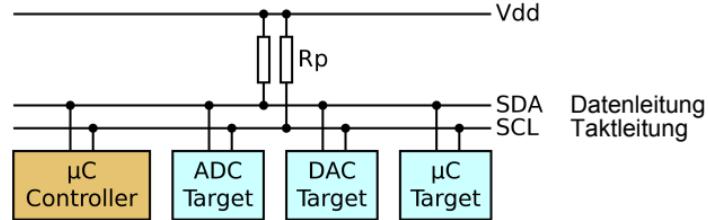
- Effiziente Verkabelung und Ressourcennutzung sind die ausschlaggebenden Argumente.
- Nachteile können durch zusätzliche Maßnahmen weitgehend ausgeglichen werden.
 - Echtzeitfähigkeit durch entsprechende Protokolle.
 - Störungen durch entsprechende Maßnahmen (z.B. Prüfsummen, Access-Guards, ...).
 - Verminderte Übertragungsleistung durch Overprovisioning.

- Punkt-zu-Punkt Verbindung:
 - Serielle Schnittstelle (UART bzw. USART)
 - ...
- Bus-Systeme:
 - Inter-Integrated Circuit (I²C) bzw. Two-Wire Interface (TWI)
 - Serial Peripheral Interface (SPI)
 - Controller Area Network (CAN-Bus)
 - Modbus
 - ...

- In der Technik gibt es noch viele veraltete und diskriminierende Begriffe.
 - z.B. Master / Slave, Whitelist / Blacklist, ...
- Seit einigen Jahren Bestrebungen, diese durch neutrale Begriffe zu ersetzen.
 - Noch gibt es leider keine einheitlichen Ersatzbegriffe.
 - Je nach Kontext unterschiedliche Ersatzbegriffe.
- **Whitelist/Blacklist ⇒ Allowlist/Denylist bzw. Inclusive List/Exclusive List**
- **Master/Slave:**
 - bei verteilten Systemen: **Primary/Secondary** bzw. **Main/Replica**
 - bei Bus-Systemen: **Controller/Target** bzw. **Leader/Follower**

- Im Folgenden werde ich die alten Begriffe nur kurz erwähnen.
 - Da immer noch viel Dokumentation mit den alten Begriffen im Umlauf ist.
- Ansonsten werde ich hauptsächlich neue diskriminierungsfreie Begriffe verwenden.
 - Inklusion ist wichtig und die Zukunft!

- Synchroner serieller Zweidraht-Bus
 - bidirektional, halbduplex
 - Eine Takteleitung (**SCL**)
 - Eine Datenleitung (**SDA**)
- **I²C** steht für *Inter-Integrated Circuit*
 - Von Philips Anfang der 1980er Jahre entwickelt, um Chips in Fernsehgeräten zu steuern.
- **TWI** steht für *Two-Wire-Interface*
 - Aus lizenzirechtlichen Gründen eingeführte Alternativbezeichnung.

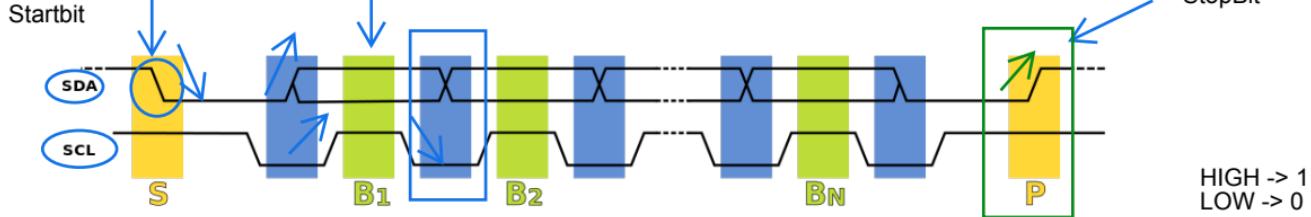


- Funktioniert nach dem Controller/Target-Prinzip (früher Master/Slave-Prinzip)
 - I.d.R ein einzelner Controller (Multi-Controller Betrieb aber auch möglich).
 - Kommunikation geht immer vom “Controller” aus.
 - “Targets” kommunizieren nur, wenn vom “Controller” aufgefordert.
- “Targets” werden durch eindeutige 7-bit Adressen identifiziert.
 - Protokollerweiterung ermöglicht 10-bit Adressen.
- Übertragungsgeschwindigkeit: 0 Bit/s – 5 MBit/s
- Nachfolgeprotokoll: I³C
 - Erstveröffentlichung Ende 2017
 - Hauptsächlich höhere Übertragungsgeschwindigkeit: 12.5 Mbit/s – 33 Mbit/s
 - Begrenzte Abwärtskompatibilität zu I²C.

Beispiel: I²C bzw. TWI



Timing



StopBit

HIGH -> 1
LOW -> 0

- Takt wird immer vom “Controller” gesteuert.
- Idle-Zustand:
 - SCL: Durchgehend HIGH
 - SDA: Durchgehend HIGH
- Start-Signal:
 - SCL: Durchgehend HIGH
 - SDA: Fallende Flanke (HIGH → LOW), vom Controller gesteuert

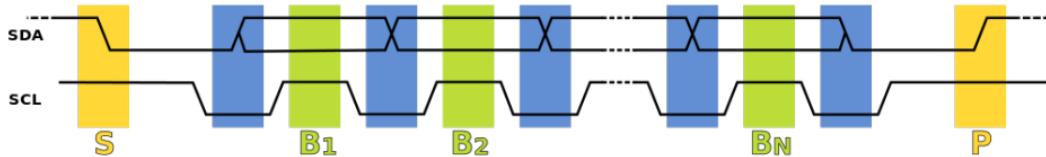
26

SCL steht für Serial Clock Line und dient als Takteleitung, die das Taktsignal für die Datenübertragung liefert.

SDA steht für Serial Data Line und ist die Datenleitung, über die die eigentlichen Daten zwischen den Geräten ausgetauscht werden.

Beispiel: I²C bzw. TWI

Timing



- Datenbits:
 - HIGH → 1, LOW → 0
 - Pegel darf sich nicht ändern solange SCL auf HIGH.
 - Änderung nur möglich wenn SCL auf LOW.
 - Es werden immer 8 Bits gesendet, gefolgt vom einem ACK- (LOW) bzw. NACK-Bit (HIGH)
- Stopp-Signal:
 - SCL: Durchgehend HIGH
 - SDA: Steigende Flanke (LOW → HIGH)

Beispiel: I²C bzw. TWI Adressierung

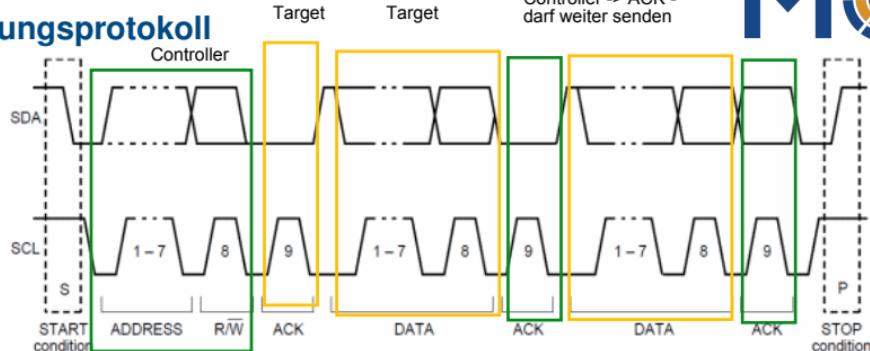


- Der "Controller" sendet immer zu Beginn einer Übertragung die "Target"-Adresse
 - Damit die "Targets" wissen, wer an der Reihe ist
- Die Adresse ist 7-bit groß
 - 112 mögliche "Target"-Adressen
 - 16 Spezialadressen (z.B. für Broadcast)
- Das 8. Bit gibt den Lese/Schreib-Modus an
 - LOW: Daten werden zum "Target" geschrieben
 - HIGH: Daten werden vom "Target" gelesen
- "Target" antwortet mit ACK-Bit (SDA auf LOW)
 - Wenn kein "Target" angeforderte Adresse hat, wird automatisch NACK-Bit gesendet (SDA auf HIGH), da HIGH Grundzustand ist.

wenn 8. Bit LOW -> wird geschrieben
8. Bit HIGH -> target gelesen

Beispiel: I²C bzw. TWI

Übertragungsprotokoll



- “Controller” sendet “Target”-Adresse und Lese-/Schreib-Bit
 - “Target” antwortet mit ACK-Bit
- Lese-Modus
 - “Target” sendet 8-Datenbits
 - “Controller” sendet ACK (weitere Daten erwartet) bzw. NACK (letztes Byte)
- Schreib-Modus:
 - “Controller” sendet 8-Datenbits
 - “Target” antwortet mit ACK

Beispiel: I²C bzw. TWI

Kommunikation mit TCN75A Temperatursensor



Auslesen passiert über I²C

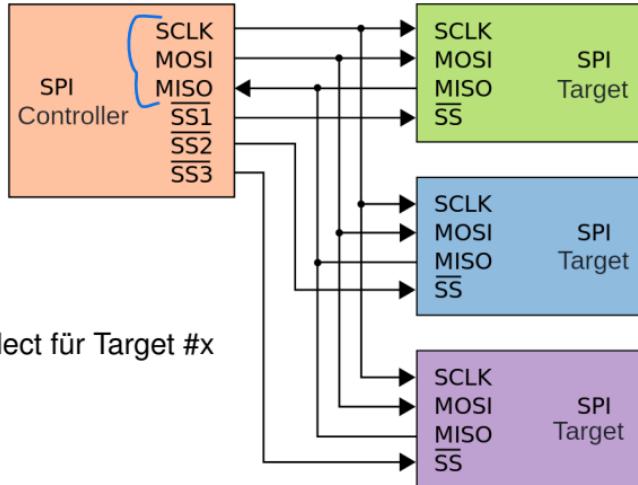
- Beispiel eines **intelligenten Sensoren**
 - Messauswertung und -verarbeitung findet direkt am Sensor statt
 - Man muss nur das fertige Messergebnis auslesen
- Besitzt neben der Temperaturmessung noch Zusatzfunktionen
 - Alarmierung bei Schwellwertüberschreitung (mit optionalen Filter)
 - Einstellbare Auflösung der Temperaturmessung (9-, 10-, 11-, oder 12-bit)
 - Stromsparfunktionen
- Kommunikation mittels **System Management Bus (SMBus)**
 - Eingeführt 1995 von Intel für die Kommunikation zwischen Komponenten am Motherboard.
 - Weitgehend **identisch zu I²C**.
- Näheres siehe Hands-On Session

- SPI steht für *Serial Peripheral Interface*
- Synchroner serieller Bus mit drei gemeinsamen Leitungen
 - bidirektional, vollduplex
 - Eine Takteleitung (**SCLK** bzw. **SCK**) Serial Clock
 - Zwei Datenleitungen
 - **MOSI**: Microcontroller Out, Sensor In (früher Master Out, Slave In)
 - **MISO**: Microcontroller In, Sensor Out (früher Master In, Slave Out)
- “lockerer” de-facto Standard mit vielen Einstellmöglichkeiten

- Funktioniert auch nach dem **Controller/Target-Prinzip**
- “Targets” werden durch exklusive **Sensor-Select** Leitungen (**SS**, früher **Slave-Select**) ausgewählt
 - Oft Aktive-LOW (d.h. zum Auswählen wird digitaler Ausgang auf 0 gesetzt)
 - Auch Kaskadierender Betriebsmodus möglich

Beispiel: SPI

Schaltbild: Controller und 3 Targets



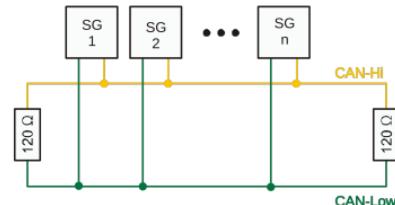
SPI geht schnellen. Braucht man für grössere Datenmenge Übetragung

- **CAN-Bus** steht für *Controller Area Network*

- Wird hauptsächlich im automotiven Bereich eingesetzt.
- 1983 von Bosch entwickelt und 1986 zusammen mit Intel vorgestellt.
- Zweck war die Reduzierung von Kabelbäumen (Damals konnte die Gesamtlänge aller Kabel in Kraftfahrzeugen ohne CAN bis zu 2 km betragen).

- Asynchroner serieller Bus

- bidirektional, halbduplex
- Zwei differenzielle Datenleitungen



es lokale Taktgeber gibt, die mit jede Nachricht synchronisieren müssen
seriell -> Daten werden hintereinander übertragen
halbduplex -> entweder senden oder empfangen

jeder kan gleichzeitig senden

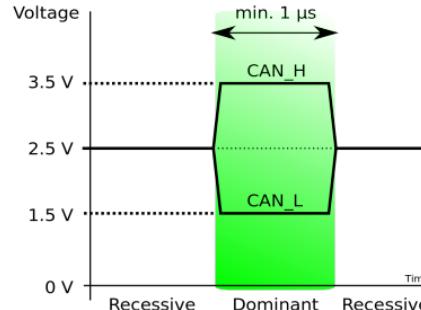
- Funktioniert nach dem **Multi-Controller Prinzip**
 - Mehrere gleichberechtigte Kommunikationspartner:innen
 - Braucht Mechanismus, um gleichzeitige Übertragungen zu erkennen und aufzulösen.
- Nachrichten werden durch eindeutige Identifier gekennzeichnet.
 - 11-bit (Base frame format) oder 20-bit (Extended frame format)
 - Nachrichten werden als Broadcast-Nachrichten versendet.
 - Anhand des Nachrichten-Identifier können Teilnehmer:innen für sie interessante Nachrichten identifizieren.

Beispiel: CAN-Bus

Differenzielle Datenleitungen



- Bits werden durch Spannungsunterschied zwischen Datenleitungen kodiert
 - 0: Spannungsunterschied vorhanden
 - 1: Kein Spannungsunterschied
- Datenübertragung:
 - Bei gleichzeitiger Übertragung werden rezessive Bits durch dominante überschrieben.
 - Logische 0: dominant
 - Logische 1: rezessiv
- Datenübertragungsrate:
 - High-Speed CAN: max. 1 MBit/s
 - Low-Speed CAN: max. 125 KBit/s



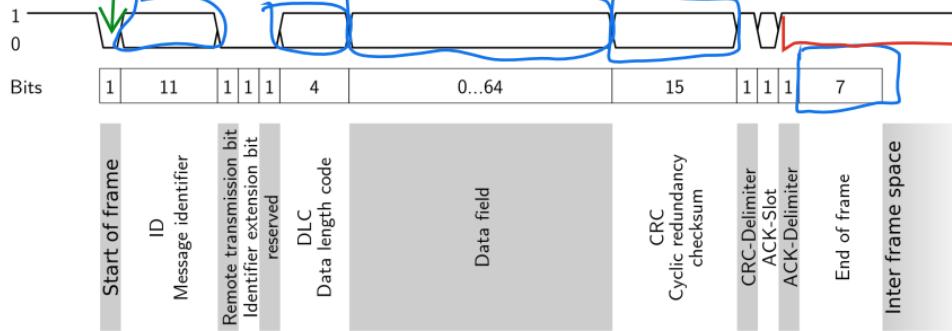
Beispiel: CAN-Bus

Nachrichtenformat



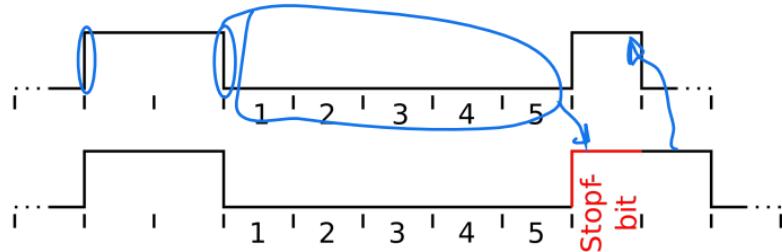
Richtig

rezeßiv ist immer 0



- Start-of-Frame Bit dient der Synchronisation der Taktgeber
 - Idle-Zustand: Logische 1
 - Start-Bit: Logische 0
- End-of-Frame Bits (7 rezessive Bits) signalisieren das Ende der Nachricht.
 - 7 rezessive Bits hintereinander kommen innerhalb der Nachricht niemals vor.

Beispiel: CAN-Bus Bitstopfen (Bit-Stuffing)



- Flanken bei Bitwechsel werden zur fortlaufenden Synchronisation verwendet.
- Problem: Bei Folge von gleichen Bits kommen keine Flanken vor.
- Lösung: Bitstopfen
 - Nach fünf gleichen Bits wird ein inverses Bit eingefügt.
 - Empfänger:in ignoriert sechstes Bit nach fünf gleichen Bits in Folge.

Beispiel: CAN-Bus

Kollisionserkennung und -behandlung



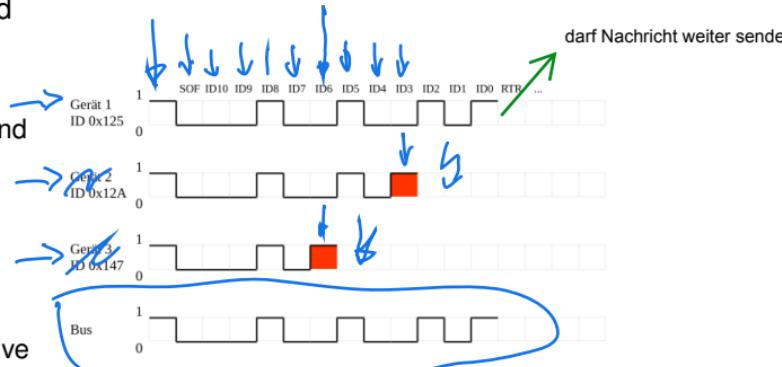
- Beginn von Nachrichtenübertragungen wird durch Startbit erkannt.
 - Bei erkannter Nachrichtenübertragung darf kein anderer mehr senden.
- Gleichzeitiger Beginn mehrerer Nachrichtenübertragungen trotzdem möglich.
 - Braucht Mechanismus zur Vergabe der Nutzungsrechte (**Arbitrierung**)
- Arbitrierung funktioniert über die Nachrichten-ID.
 - Niedrigere ID hat höhere Priorität.
 - Als Verfahren kommt **Carrier Sense Multiple Access/Collision Resolution (CSMA/CR)** zum Einsatz.
 - Verwendet **Bitarbitrierung**.

Beispiel: CAN-Bus CSMA/CR und Bitarbitrierung



- Während Nachrichtenübertragung wird jedes gesendete Bit mit Buszustand verglichen.

- Abweichungen bedeuten, dass jemand anderes auch sendet und man Arbitrierungsverfahren verloren hat.



- Direkt nach dem Startbit wird die Nachrichten-ID übertragen.

- Dominante Bits überschreiben rezessive Bits.
⇒ Niedrigere ID gewinnt deswegen.

- Controller/Target Systeme
 - Controller/Target Systeme sind i.d.R. deterministisch.
 - Und daher meist automatisch echtzeitfähig (abhängig von Anforderungen wie z.B. Übertragungsgeschwindigkeit).
- CSMA-Systeme
 - CSMA-Systeme sind i.d.R. nicht deterministisch.
 - Und daher nicht ohne weitere Maßnahmen echtzeitfähig.
 - Bitarbitrierung beim CAN-Bus garantiert Echtzeit nur für höchste Priorität.
- Zeitslot-Systeme
 - Zeitslot-Systeme sind i.d.R. deterministisch.
 - Und daher auch meist automatisch echtzeitfähig (wieder abhängig von Anforderungen).

Inhaltsverzeichnis



Echtzeitsysteme

Kommunikationsnetzwerke

Hands-On Session: Temperatursensor TCN75A

Hands-On Session: TCN75A

Configuration des TCN75A



- Wird über Register ausgelesen und konfiguriert
 - Register werden über SMBus gelesen und geschrieben
- Besitzt 4 Register:
 - Ambient Temperature Register (16-bit) Sensor to Controller
 - Sensor Configuration Register (8-bit) Gehr in beide Richtungen
 - Temperature Hysteresis Register (16-bit) Für Alarm Funktion, kann auch in beide Richtungen lesen und schreiben
 - Temperature Limit-Set Register (16-bit)

Hands-On Session: TCN75A

Configuration des TCN75A

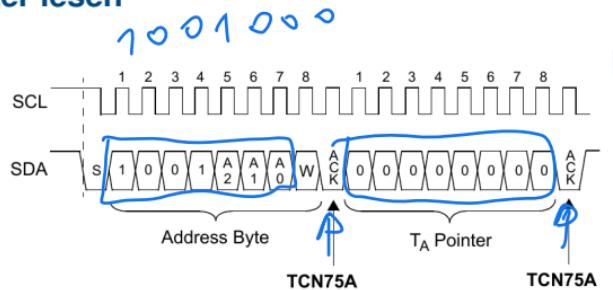


TABLE 5-1: BIT ASSIGNMENT SUMMARY FOR ALL REGISTERS

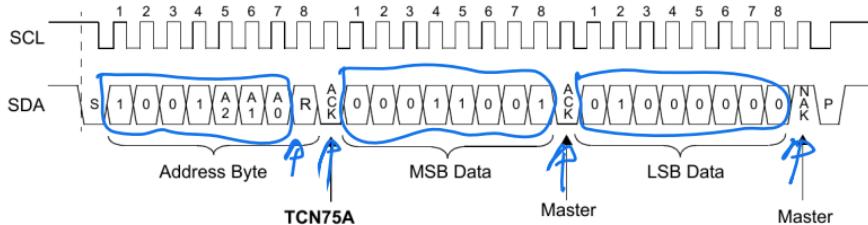
Register Pointer P1 P0	MSB/ LSB	Bit Assignment							
		7	6	5	4	3	2	1	0
Ambient Temperature Register (T_A)									
0 0	MSB	Sign	2^6°C	2^5°C	2^4°C	2^3°C	2^2°C	2^1°C	2^0°C
	LSB	$2^{-1^\circ\text{C}}$	$2^{-2^\circ\text{C}}$	$2^{-3^\circ\text{C}}$	$2^{-4^\circ\text{C}}$	0	0	0	0
Sensor Configuration Register (CONFIG)									
0 1	LSB	One-Shot	Resolution		Fault Queue		ALERT Polarity	COMP/INT	Shutdown
Temperature Hysteresis Register (T_{HYST})									
1 0	MSB	Sign	2^6°C	2^5°C	2^4°C	2^3°C	2^2°C	2^1°C	2^0°C
	LSB	$2^{-1^\circ\text{C}}$	0	0	0	0	0	0	0
Temperature Limit-Set Register (T_{SET})									
1 1	MSB	Sign	2^6°C	2^5°C	2^4°C	2^3°C	2^2°C	2^1°C	2^0°C
	LSB	$2^{-1^\circ\text{C}}$	0	0	0	0	0	0	0

Configuration des TCN75A

Register lesen

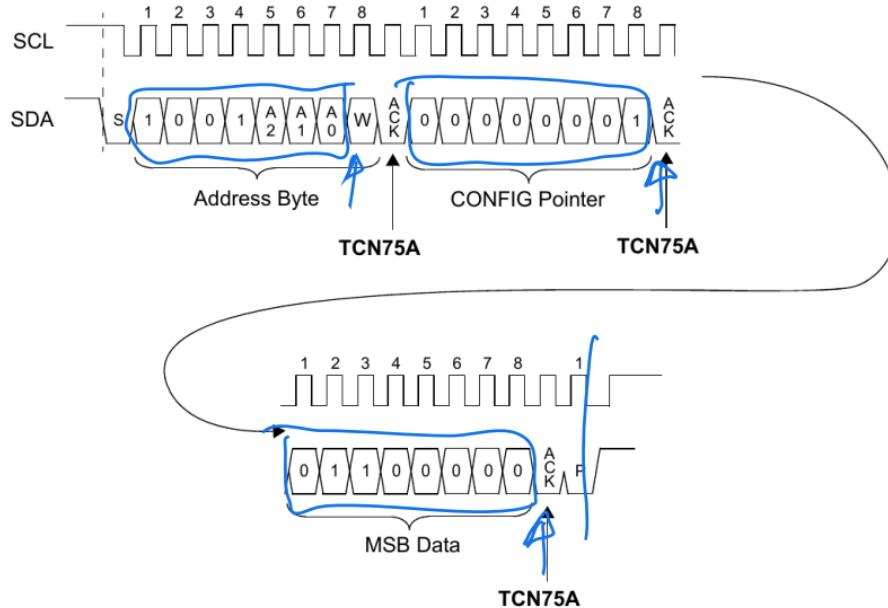


Note:
It is not necessary to select the register pointer if it was set from the previous read/write.
(see Section 4.1.1)



Configuration des TCN75A

Register schreiben



4 ECHTZEITSYSTEME UND KOMMUNIKATIONSNETZWERKE

▼ Echtzeit und Echtzeitsysteme

Was bedeutet "Echtzeit"?

Ein **Echtzeitsystem** ist ein Computersystem, bei dem die **Korrektheit nicht nur vom Ergebnis**, sondern auch von der **Zeit abhängt**, in der das Ergebnis geliefert wird.



Definition (nach DIN 44300):

"Ein Echtzeitsystem ist ein System, in dem die **Verarbeitung der Daten durch das System innerhalb der durch die Außenwelt vorgegebenen Zeitbeschränkung**



Echtzeit ≠ Schnell

Ein System kann **langsam**, aber dennoch **echtzeitfähig** sein – solange es **immer innerhalb der garantierten Zeitgrenzen** antwortet.



Arten von Echtzeitsystemen

Typ	Beschreibung	Beispiel
Harte Echtzeit	Verpasste Fristen führen zu Systemversagen	Airbag, Herzschrittmacher
Feste Echtzeit	Fristen müssen eingehalten werden , aber nicht katastrophal bei Fehler	Industrie-Roboter
Weiche Echtzeit	Fristüberschreitung reduziert Qualität , aber kein Ausfall	Audio-/Videowiedergabe



Merkmale von Echtzeitsystemen

- **Deterministisches Zeitverhalten** (vorhersagbar)
- **Fristen sind bekannt und messbar**
- **Reaktionszeit ist entscheidend**
- Ressourcen (CPU, RAM) sind oft **begrenzt**
- Einsatz in sicherheitskritischen oder zeitkritischen Anwendungen



Typische Anwendungen

- Luft- und Raumfahrt
- Automobilindustrie (ABS, ESP, Airbag)
- Medizintechnik (z. B. Infusionspumpe)
- Industrieautomation (z. B. SPS, Robotik)
- Embedded Audio/Video (mit weicher Echtzeit)



Echtzeitsysteme in der Software

- **Echtzeitbetriebssysteme (RTOS):** z. B. FreeRTOS, VxWorks, RTEMS
- **Scheduler mit festen Prioritäten:** z. B. Rate Monotonic, EDF (Earliest Deadline First)
- Keine spekulative Ausführung, kein Paging, kein Caching ohne Kontrolle
- **Interrupt-Latenzen müssen kontrollierbar sein**



Herausforderungen bei Echtzeitsystemen

Herausforderung	Beispiel
Interrupt-Jitter	Varierende Reaktionszeit auf IRQs
Nicht-deterministisches OS	Windows, Linux ohne RT-Patches
Unvorhersehbare Laufzeit	z. B. durch Cache Misses, Pipelining
Prioritätsinversion	Niedrige Aufgabe blockiert hohe

→ Lösung: Prioritätsvererbung, deterministische Scheduler



Zusammenfassung

Merkmal	Echtzeitsysteme
Definition	Ergebnis muss innerhalb einer festen Zeit vorliegen
Arten	Hart, fest, weich
Typische Anwendung	Kritische Steuer- und Regelaufgaben
Software	RTOS, deterministisches Scheduling
Fehler bei Frist	kritisch (hart), tolerierbar (weich)

Quellen

1. [Wikipedia – Echtzeitsystem](#)
2. [embedded-software-engineering.de – Echtzeit-Grundlagen](#)

▼ Bus-Systeme

Was ist ein Bus(-System) in der Datenverarbeitung?

Ein **Bus** ist ein **gemeinsamer Übertragungsweg**, über den mehrere Komponenten eines Rechners **Daten, Adressen und Steuerinformationen austauschen**.

Er verbindet z. B. **CPU, RAM, Peripherie, Controller**.



Bestandteile eines Bus-Systems

Bus-Typ	Aufgabe	Beispiel-Datenbreite
Datenbus	Überträgt Daten	8, 16, 32, 64 Bit
Adressbus	Überträgt Adressen (wohin?)	z. B. 32 Bit → 4 GB Adressraum
Steuerbus	Überträgt Steuersignale	z. B. Lese-/Schreibsignale



Arten von Bus-Systemen (nach Struktur)

Art	Beschreibung	Beispiel
Parallelbus	Mehrere Leitungen gleichzeitig	Datenbus auf Platine
Serieller Bus	Bits nacheinander übertragen	SPI, I2C, USB
Ein-/Mehrpunkt	Alle Teilnehmer am selben Bus / sternförmig	I2C (mehrpunkt), USB (sternförmig)



Typische Bus-Systeme (in Embedded & PC)

Bus-System	Typ	Eigenschaften
I ² C	seriell	2 Leitungen, langsam, leicht, günstig, Adressierung
SPI	seriell	schneller, mehrere Chip-Selects
UART	seriell	einfache Punkt-zu-Punkt-Kommunikation
CAN	seriell	robust, automotive, Fehlererkennung integriert
USB	seriell	Hot-Plug, Plug&Play, Master-Slave
PCIe	seriell	Hochgeschwindigkeitsbus im PC-Bereich



Buszugriffsverfahren

Wenn mehrere Geräte auf denselben Bus zugreifen wollen, braucht man ein **Arbitrationsverfahren**:

Verfahren	Beschreibung
Zentral (Master)	Ein Gerät bestimmt Zugriff (z. B. I ² C)
Verteilt	Jeder darf nach bestimmten Regeln
Prioritätsgesteuert	Geräte mit höherer Priorität zuerst



Probleme von Bussystemen

Problem	Ursache
Bus-Kollision	Zwei senden gleichzeitig
Latenz	Bei vielen Teilnehmern langsam
Störungen/EMI	Besonders bei langen Leitungen
Flaschenhals	Geteilter Pfad limitiert Bandbreite



Bus in Embedded-Systemen

- Mikrocontroller haben oft:
 - **I²C/SPI/UART** für externe Sensoren/Displays
 - **Interne Busse** (z. B. Harvard-Architektur → getrennt für Code und Daten)
- Buskommunikation erfolgt über **Registerzugriffe**



Zusammenfassung

Merkmal	Bedeutung
Bus	Gemeinsames Verbindungssystem
Typen	Daten-, Adress-, Steuerbus
Strukturen	Seriell/Parallel, Punkt-zu-Punkt oder Bus
Systeme	I ² C, SPI, UART, CAN, USB, PCIe
Probleme	Kollision, Latenz, Störungen



- [Wikipedia – Bus \(Datenverarbeitung\)](#)

▼ I²C-Bus



Was ist I²C?

I²C (gesprochen: „I-two-C“ oder „I-squared-C“) ist ein serieller Kommunikationsbus, der von Philips entwickelt wurde, um mehrere ICs über nur zwei Leitungen zu verbinden.



Hauptmerkmale:

Eigenschaft	Beschreibung
Anzahl Leitungen	2 (SDA = Daten, SCL = Takt)
Kommunikationstyp	seriell, synchron (Takt von Master)
Teilnehmer	1 Master, mehrere Slaves
Adressierung	7-Bit (Standard), 10-Bit (erweitert)
Busgeschwindigkeit	Standard (100 kHz), Fast (400 kHz), High-Speed (3.4 MHz)



Physikalischer Aufbau

- **SDA** (Serial Data): Bi-direktionale Datenleitung
- **SCL** (Serial Clock): Takteleitung, gesteuert vom Master
- Beide Leitungen benötigen **Pull-Up-Widerstände** (z. B. 4.7 kΩ)

```
Master --- SDA ---+--- Slave 1  
                   --- SCL ---+--- Slave 2  
                   ...
```



Ablauf einer I²C-Kommunikation

1. **START-Bedingung**: SDA wechselt von HIGH → LOW, während SCL HIGH ist
2. **Adresse + R/W-Bit** senden (7 Bit + 1 Bit Read/Write)
3. **ACK**: Empfänger sendet Bestätigung (ACK-Bit)
4. **Datenübertragung** (8 Bit + ACK)
5. **STOP-Bedingung**: SDA wechselt von LOW → HIGH, während SCL HIGH ist



Adressierung

- 7-Bit-Adresse = 128 mögliche Geräteadressen (0x00–0x7F)
- 10-Bit optional, aber selten
- Viele ICs haben feste oder konfigurierbare I²C-Adressen



Arduino: I²C Beispiel mit Wire.h

```
#include <Wire.h>

void setup() {
    Wire.begin(); // als Master starten
    Wire.beginTransmission(0x3C); // Adresse des Slaves
    Wire.write("Hallo");
    Wire.endTransmission();
}

void loop() {}
```

Slave (z. B. zweites Arduino-Board):

```
#include <Wire.h>
```

```

void setup() {
    Wire.begin(0x3C); // eigene Adresse
    Wire.onReceive(receiveData);
}

void receiveData(int bytes) {
    while (Wire.available()) {
        char c = Wire.read();
        Serial.print(c);
    }
}

```



Vorteile und Nachteile von I²C

Vorteile	Nachteile
Nur 2 Leitungen für viele Geräte	Langsamer als SPI
Adressierung per Software	Adresskonflikte möglich
Integrierte Bestätigung (ACK/NACK)	Leitungslänge begrenzt (cm-m)
Multimaster möglich (aber selten genutzt)	Timing empfindlich



Typische I²C-Geräte

- Temperatursensoren (z. B. LM75)
- Echtzeituhr (RTC, z. B. DS1307)
- OLED-/LCD-Displays
- EEPROMs
- IMUs (Beschleunigungs-/Lagesensoren)

Quellen

1. [Wikipedia – I²C](#)
2. [fmh-studios.de – I²C-Bus Theorie](#)
3. [DeepBlueEmbedded – Arduino I²C Tutorial](#)
4. [Instructables – Arduino and I2C](#)



Zusammenfassung

Merkmal	Wert
Bus-Typ	Seriell, synchron
Leitungen	2 (SDA, SCL)
Geräte	1 Master, viele Slaves
Geschwindigkeit	100 kHz – 3.4 MHz

Merkmal	Wert
Adressierung	7 oder 10 Bit
Bibliothek (Arduino)	Wire.h

▼ Serial Peripheral Interface (SPI)



Was ist SPI?

SPI (Serial Peripheral Interface) ist ein **schnelles, synchrones, serielles Kommunikationsprotokoll**, das typischerweise für die Verbindung eines **Masters** mit einem oder mehreren **Slaves** verwendet wird.

Es wurde ursprünglich von **Motorola** entwickelt.



Grundprinzip

SPI überträgt **Bits gleichzeitig in beide Richtungen** (Voll-Duplex) über **vier Leitungen**:

Leitung	Name	Funktion
MOSI	Master Out, Slave In	Daten vom Master zum Slave
MISO	Master In, Slave Out	Daten vom Slave zum Master
SCLK	Serial Clock	Takt vom Master
SS	Slave Select (CS)	Aktivierung eines Slaves (LOW)

Jeder Slave braucht einen **eigenen SS-Pin** am Master.



Merkmale von SPI

Eigenschaft	Wert
Kommunikation	Synchron (Takt gesteuert)
Anzahl Leitungen	Mind. 4 (MOSI, MISO, SCLK, SS)
Richtung	Voll-Duplex
Geschwindigkeit	Bis mehrere MHz (z. B. 10+ MHz)
Adressierung	Keine – Selektion über SS



SPI-Datenübertragung

1. Master zieht SS-Leitung des gewünschten Slaves **auf LOW**
 2. Master erzeugt Takt (SCLK)
 3. Daten werden **gleichzeitig gesendet & empfangen** (MOSI & MISO)
 4. Nach Ende der Übertragung wird SS wieder **auf HIGH** gesetzt
- ➡ Jeder **Taktimpuls** schiebt 1 Bit weiter



SPI-Modi (Clock Polarity/Phase)

SPI kennt **vier Modi**, definiert durch **CPOL** (Polarity) und **CPHA** (Phase):

Modus	CPOL	CPHA	Beschreibung
0	0	0	Daten auf steigender Flanke gelesen
1	0	1	Daten auf fallender Flanke gelesen
2	1	0	Daten auf fallender Flanke gelesen
3	1	1	Daten auf steigender Flanke gelesen

► Master & Slave müssen denselben Modus verwenden!



SPI in der Praxis (z. B. Arduino)

```
#include <SPI.h>

void setup() {
    SPI.begin();          // Initialisiert SPI als Master
    digitalWrite(SS, LOW); // Slave auswählen
    SPI.transfer(0x42);   // 1 Byte senden
    digitalWrite(SS, HIGH); // Slave abwählen
}

void loop() {}
```



Typische SPI-Geräte

- Displays (z. B. OLED, TFT)
- SD-Karten
- Flash-Speicher (EEPROM, NOR)
- ADCs / DACs
- Funkmodule (nRF24L01, LoRa)



Vorteile von SPI

Vorteil	Nachteil
Sehr schnell und effizient	Mehr Leitungen nötig
Einfach zu implementieren (HW/SW)	Kein standardisiertes Protokoll
Vollduplex möglich	Kein eingebautes Adresssystem
Beliebige Datenbreite (oft 8 Bit)	Keine automatische Fehlererkennung



Quellen

1. [Wikipedia – SPI](#)

2. [mikrocontroller.net – SPI](#)

3. [RN-Wissen – SPI](#)



Zusammenfassung

Merkmal	Beschreibung
Bus-Typ	Seriell, synchron, Master-Slave
Leitungen	MOSI, MISO, SCLK, SS
Kommunikation	Vollduplex, schnell
Protokollstruktur	Kein Standard – oft gerätespezifisch
Einsatz	Displays, SD-Karten, Sensoren

▼ CAN-Bus (Controller Area Network)



Was ist der CAN-Bus?

Der **CAN-Bus (Controller Area Network)** ist ein robustes, serielles Kommunikationsprotokoll, das speziell für den **Austausch von Daten zwischen Steuergeräten (ECUs)** in Fahrzeugen entwickelt wurde.

➡ Ursprünglich von **Bosch (1983)** für die Automobilindustrie entwickelt.



Wesentliche Eigenschaften

Merkmal	Beschreibung
Bus-Typ	Seriell, mehrpunktfähig (Multi-Master)
Teilnehmer	Bis zu ~100 Geräte (typisch 10–50)
Übertragung	Nachrichtenbasiert , nicht gerätebasiert
Geschwindigkeit	Bis 1 Mbit/s (CAN 2.0), bis 5–8 Mbit/s (CAN-FD)
Leitung	2-Draht-System : CAN_H und CAN_L
Zugriffsmethode	CSMA/CR (Carrier Sense Multiple Access with Collision Resolution)



Wie funktioniert die Kommunikation?

- **Broadcast-System**: Alle Nachrichten gehen an alle Teilnehmer
 - Nachrichten enthalten **keine Empfängeradresse**, sondern eine **ID**
 - Geräte **filtern selbst**, ob sie auf die Nachricht reagieren
-



Wichtige Konzepte



CAN-Frames

Feld	Funktion
ID (Identifier)	Priorität der Nachricht (je kleiner, desto höher)
Daten	0–8 Byte (CAN 2.0) oder bis 64 Byte (CAN-FD)
CRC	Fehlererkennung (Zyklische Redundanzprüfung)
ACK	Empfänger bestätigen korrekte Nachrichten



Fehlererkennung & Zuverlässigkeit

- **Mehrere Mechanismen** zur Erkennung von:
 - Bitfehlern
 - CRC-Fehlern
 - Format- und Stuffing-Fehlern

➡ Hohe Zuverlässigkeit – ideal für **sicherheitskritische Anwendungen**



Arbitrierung: Wer darf senden?

- **Mehrere Teilnehmer dürfen gleichzeitig senden**
- ID entscheidet über **Priorität**
- Wer beim Vergleich ein **dominantes Bit verliert, hört auf zu senden**

➡ Keine Kollision, nur **verliererloses Arbitrieren**



CAN vs. andere Busse

Merkmal	CAN-Bus	I ² C / SPI
Teilnehmerzahl	Hoch (10–100)	Gering (meist < 10)
Adressierung	Nachricht/ID-basiert	Geräteadresse
Robustheit	Sehr hoch (automotive-tauglich)	Gering bis mittel
Fehlererkennung	Integriert (CRC, ACK, ...)	Manuell
Geschwindigkeit	Mittel (1 Mbit/s Standard)	Höher (SPI), niedriger (I ² C)



Typische Anwendungen

- Fahrzeuge (PKW, LKW, Bus, Bahn)
- Industriesteuerung
- Medizintechnik
- Gebäudeautomation

- Robotik (z. B. CANopen, CAN-FD)

Quellen

- [Wikipedia – Controller Area Network \(CAN\)](#)
- [CSSelectronics – CAN Bus Explained Simply](#)

Zusammenfassung

Aspekt	Inhalt
Protokolltyp	Seriell, nachrichtenorientiert
Leitung	2-Draht: CAN_H & CAN_L
Kommunikation	Multimaster, Broadcast
Fehlererkennung	CRC, ACK, Bitmonitoring
Einsatzgebiet	Automotive, Industrie, Robotik
Vorteil	Robust, priorisiert, kollisionsfrei

▼ Vergleichstabelle für CAN vs. I²C vs. SPI



Bus-Vergleichstabelle: CAN vs. I²C vs. SPI

Merkmal	CAN	I ² C	SPI
Typ	Seriell, nachrichtenbasiert	Seriell, adressbasiert	Seriell, Master-Slave
Leitungen	2 (CAN_H, CAN_L)	2 (SDA, SCL)	mind. 4 (MOSI, MISO, SCLK, SS)
Max. Geschwindigkeit	1 Mbit/s (CAN 2.0), bis 8 Mbit/s (CAN-FD)	100–400 kHz (Standard/Fast), bis 3.4 MHz (High-Speed)	10+ Mbit/s typisch (theoretisch >50 Mbit/s)
Teilnehmerzahl	bis zu 100	typisch <10	wenige (wegen CS-Leitung)
Adressierung	über IDs in Nachrichten	über 7-/10-Bit-Adresse	keine, per Chip Select (CS)
Kommunikationsrichtung	Multimaster, Voll-Duplex über Nachrichten	Master-Slave, Halbduplex	Voll-Duplex
Fehlererkennung	Integriert (CRC, ACK, Bit-Monitoring)	Einfaches ACK/NACK	keine eingebaut
Arbitrierung / Zugriff	CSMA/CR (Prioritätsbasiert)	Master-basiert	Master-gesteuert
Robustheit / Störsicherheit	Sehr hoch (automotive-tauglich)	Gering bis mittel	Mittel
Datenstruktur	Frame mit ID, Daten (0–64 Byte)	Bytestream	Bytestream
Implementierungskomplexität	Hoch (Controller nötig, z. B. MCP2515)	Einfach	Mittel
Typische Anwendungen	Fahrzeugtechnik, Industrieautomation	Sensoren, Displays, RTC	Displays, SD-Karten, Flash, DACs



Fazit nach Anwendungsfall:

Anwendungsszenario	Empfehlung
Viele Geräte, hohe Robustheit nötig	<input checked="" type="checkbox"/> CAN
Wenige Geräte, einfache Verdrahtung	<input checked="" type="checkbox"/> I ² C
Hoher Datendurchsatz, kurze Strecken	<input checked="" type="checkbox"/> SPI



RECHNERARCHITEKTUR

RECHNERSTRUKTUREN & EMBEDDED SYSTEMS

MATTHIAS JANETSCHKEK
SS 2025

Einführung

Aufbau von Computersystemen

Methoden zur Leistungssteigerung

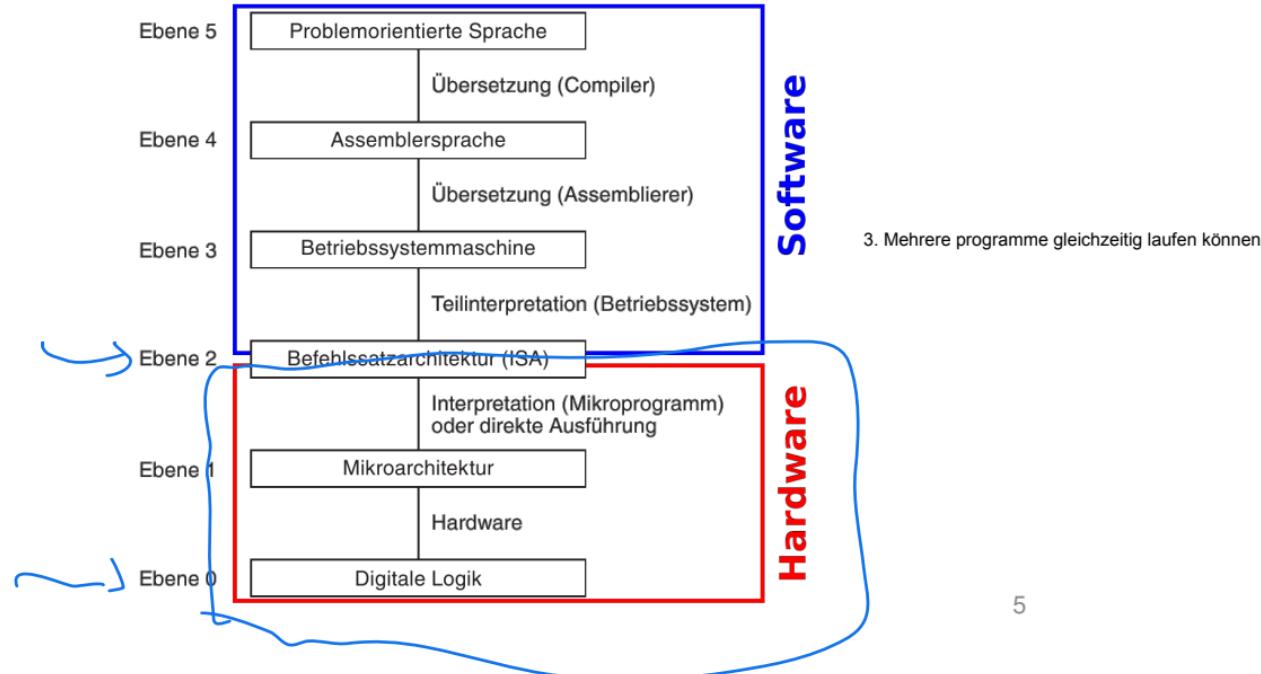
Einführung

Aufbau von Computersystemen

Methoden zur Leistungssteigerung

- Computer oder Digitalrechner sind Maschinen, die Aufgaben lösen, indem sie gegebene Befehle ausführen.
- Eine Befehlsequenz, die eine bestimmte Aufgabe löst, nennt man **Programm**.
- Die elektronischen Schaltungen eines Computers verstehen eine **begrenzte Anzahl einfacher Befehle**.
 - Alle Programme müssen in diese einfachen Befehle konvertiert werden, bevor sie ausgeführt werden können.
- Die Gesamtheit der elementaren Befehle eines Computers bildet die sogenannte **Maschinensprache**.
 - Es wird versucht, diese Befehle so einfach wie möglich zu halten.

- Mit Maschinensprachen zu arbeiten ist für Menschen schwierig und umständlich.
⇒ Stattdessen werden Hochsprachen verwendet, die dann in Maschinensprache konvertiert werden.
- Ein Computer ist als **Folge von Abstraktionsschichten** aufgebaut.
 - Jede Schicht erfüllt eine bestimmte Aufgabe.
 - Jede Schicht ist auf der jeweils darunterliegenden aufgebaut.
 - Komplexität wird so beherrschbar ("Teile-und-Herrsche"-Prinzip)
- Dieser Ansatz wird **strukturierte Computerorganisation** genannt.



Ebene 5: Problemorientierte Sprache

- Speziell auf Anwendungsprogrammierer:innen zugeschnitten.

Ebene 4: Assemblersprache

- Symbolischer Maschinencode

Ebene 3: Betriebssystemmaschine

- Betriebssystemspezifische Funktionen
- z.B. Speicherorganisation, Multitasking, ...

Ebene 2: Befehlssatzarchitektur (ISA)

- Heimat des “Maschinencode”
- Schnittstelle zwischen Hardware und Software

Ebene 1: Mikroarchitekturebene

- Optionale Schicht
- Setzt Maschinencode in interne Mikrobefehle um
- z.B. um CISC-Befehle auf RISC-Rechnern ausführen zu können

Ebene 0: Digitale Logik

- Die eigentliche Hardware-Schaltung, die die Mikro- bzw. Maschinenbefehle ausführt
- Hardware-Schaltung wird aus einzelnen Logik-Gattern aufgebaut

Einführung

Aufbau von Computersystemen

Methoden zur Leistungssteigerung

Definition (Logikgatter, laut Wikipedia)

Ein **Logikgatter**, auch nur **Gatter**, (engl. **(logic) gate**) ist eine Anordnung (heutzutage praktisch immer eine **elektronische Schaltung**) zur **Realisierung einer booleschen Funktion**.

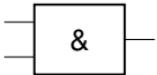
- Binäre Eingangssignale werden zu binären Ausgangssignalen verarbeitet.
- **Teile-und-Herrsche Prinzip:** Einfache Schaltungen dienen als Bausteine für komplexere Schaltungen
 - Aus Grundoperationen (AND, OR, NOT, ...) kann man einen Addierer bauen.
 - Aus Addierer & Co. kann man eine Arithmetic Logic Unit (ALU) bauen.
 - Aus der ALU, Registerspeichern, Befehlsdecoder, ... kann man eine CPU bauen.
 - Aus CPU, Arbeit- und Sekundärspeicher, ... kann man einen Computer bauen.

Logikgatter

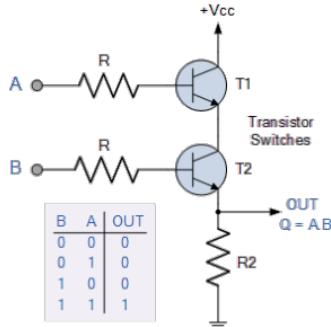
Beispiel: AND und OR Gatter

AND Gatter

- Schaltbaustein

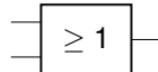


- Beispielschaltung

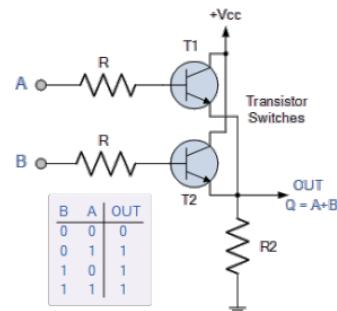


OR Gatter

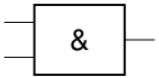
- Schaltbaustein



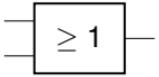
- Beispielschaltung



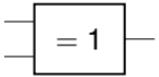
- AND Gatter



- OR Gatter



- XOR Gatter



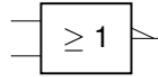
- Buffer Gatter



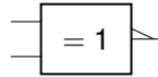
- NAND Gatter



- NOR Gatter



- XNOR Gatter

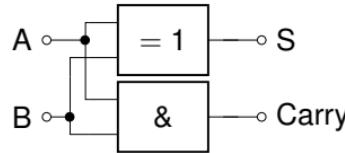


- NOT Gatter



1-Bit Addierer

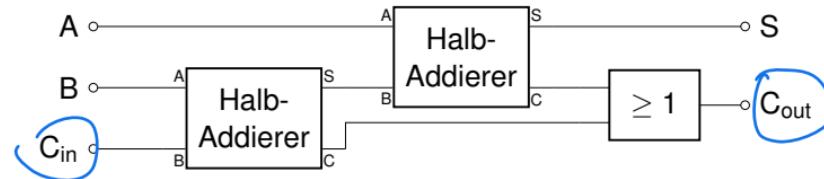
- 1-Bit Halbaddierer



$$1 + 1 = 1 \text{ } \textcircled{0}$$

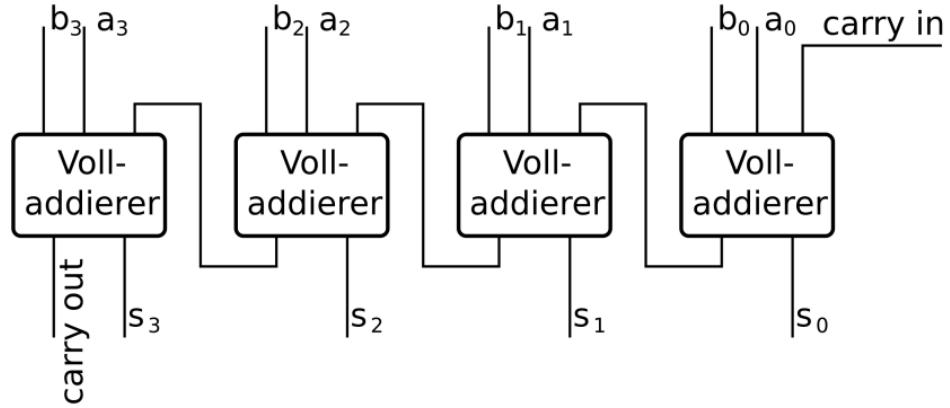
↓ in 2s (aus) → carry bit

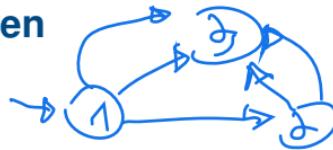
- 1-Bit Volladdierer



Logische Schaltungen

4-Bit Carry-Ripple-Addierer



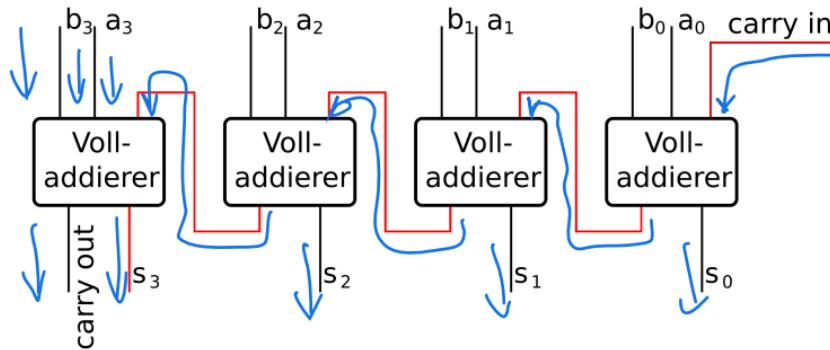


Taktsignal sagt mir wann ich von 1 nach 2 wechseln kann der nach 3 usw.

- Signalausbreitung erfolgt mit endlicher Geschwindigkeit.
 - Unterschiedliche Leitungslängen führen zu unterschiedlichen Laufzeiten.
⇒ Signale müssen zeitlich synchronisiert werden.
- Für die Synchronisierung kommt ein Taktsignal zum Einsatz.
 - Elektrische Schaltung als Zustandsautomat: Das Taktsignal steuert die Zustandswechsel.
 - Computer als Maschine: Das Taktsignal gibt die Arbeitsgeschwindigkeit vor (vgl. Taktsignal im I2C-Bus als Angabe der Bitlänge).

Die Rolle des Taktsignals

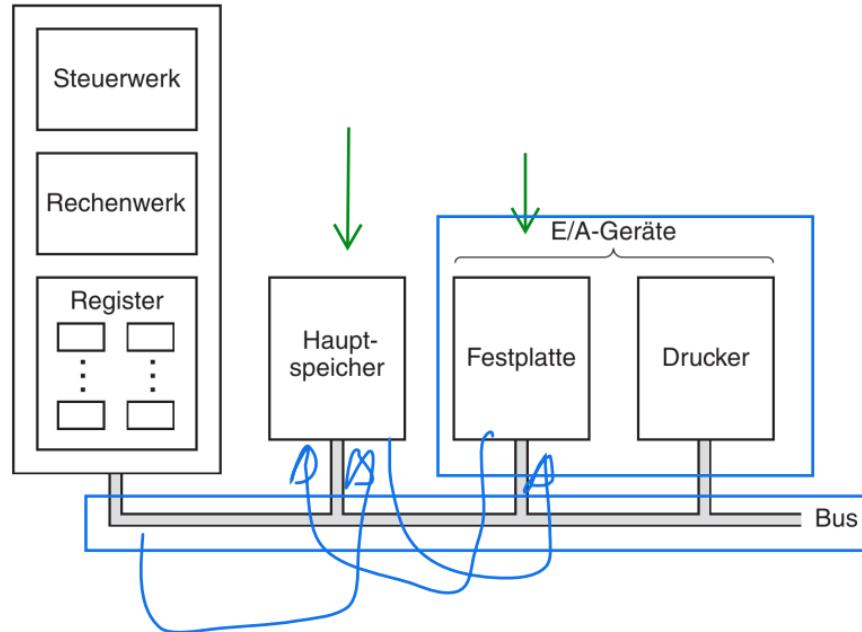
Beispiel: 4-Bit Carry-Ripple-Addierer



- Überlauf-Bits müssen erst durch die Volladdierer durchwandern und kommen später an als direkte Eingabe-Bits a_x und b_x .
- Taktsignal synchronisiert Eingänge und gibt vor, wann gültige Signale anliegen.

Aufbau von Computersystemen

Zentrale Recheneinheit (CPU)



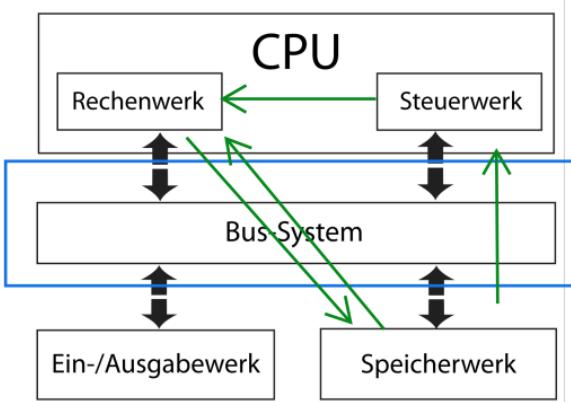
Definition (Von-Neumann-Architektur, laut Wikipedia)

Die **Von-Neumann-Architektur** ist ein **Schaltungskonzept** zur Realisierung **universeller Rechner** (Von-Neumann-Rechner, VNR). Sie realisiert alle Komponenten einer **Turingmaschine**.

- Bildet die Grundlage der allermeisten modernen Computer.
 - Teilweise mit leichten Abweichungen.
- Implementiert eine Turingmaschine.
 - Alles was eine Turingmaschine berechnen kann, kann (theoretisch) auch ein Computer berechnen.

Von-Neumann-Architektur

Komponenten



- **Rechenwerk**
 - Führt logische und arithmetische Operationen durch.
- **Steuerwerk**
 - Interpretiert Befehle und steuert restliche Komponenten.
- **Bus-System**
 - Verbindet Komponenten und ermöglicht Kommunikation.
- **Speicherwerk (Arbeitsspeicher)**
 - Speichert Befehle und Daten.
- **Eingabe-/Ausgabewerk**
 - Datenverbindung zu anderen Komponenten.

- Harvard-Architektur

- Logisch und physisch getrennter Befehls- und Datenspeicher
 - Meist unterschiedliche Addressräume für Befehle und Daten
 - Oft unterschiedliche Befehle zum Laden/Speichern von Daten bzw. Befehlen.
- Kann Befehle und Daten gleichzeitig laden, Zugriffsrechtetrennung und Speicherschutz einfach realisierbar
- z.B. die meisten Mikrocontrollern

Wenn ich mehr Arbeitsspeicher brauche bei Harvard Architektur geht nicht.
Und bei Neumann geht, weil nur 1 Speicher, gemeinsame.

- Von-Neumann-Architektur

- Gemeinsamer Speicher für Befehle und Daten.
- Flexibler als Harvard-Architektur
- z.B. x86 Computer

Aufbau von Computersystemen

CPU: Aufbau



Die Central Processing Unit (CPU) ist aus Milliarden von Transistoren aufgebaut, kann mehrere Verarbeitungskerne aufweisen und wird üblicherweise als „Gehirn“ des Computers bezeichnet.

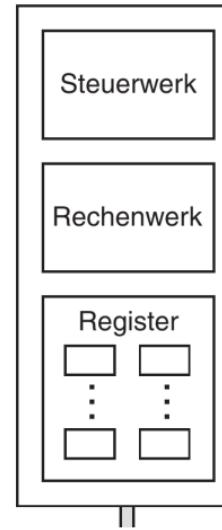
implementiert Befehle

- **Steuerwerk**

- Steuert den Ablauf der Befehlsverarbeitung
- Sendet und empfängt Steuersignale an alle anderen Funktionseinheiten.
- Besteht aus Programmzähler, Befehlsregister, Befehlsdekoder und den Steuerleitungen.

- **Rechenwerk** ausführt Befehle

- Nennt man auch **Arithmetic Logic Unit (ALU)**
- Führt die Maschinenbefehle aus.
- Besteht oft aus einzelnen spezialisierten Funktionseinheiten (z.B. Addierwerk, Multiplizierwerk, Sprungeinheit, ...)

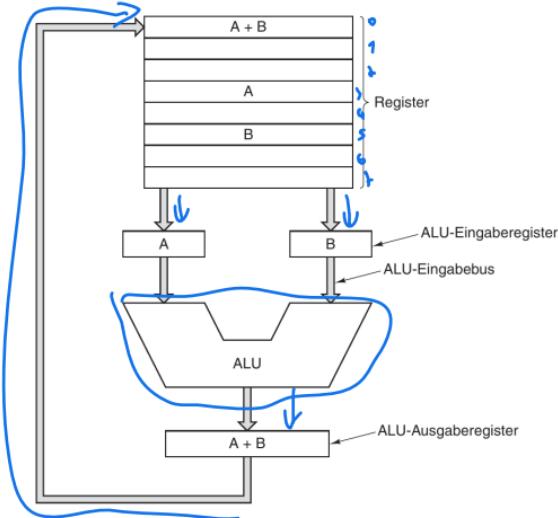


- Register sind Speicherstellen, auf die sehr schnell zugegriffen werden kann
- Dienen dem Zwischenspeichern von Daten
- Befinden sich auf der Prozessor-Die (Die nennt man den Siliziumchip in einer CPU) in unmittelbarer Nähe der Rechenwerke
- Wieso braucht es Register?
 - Hauptspeicherzugriffe sind vergleichsweise langsam
 - z.B. DDR4 Zugriffszeit: ≈ 15 ns
 - z.B. Intel Core i7-9700K @ 4.9GHz: ≈ 0.2 ns Zykluszeit
 - ⇒ Speicherzugriff braucht mind. 75 Taktzyklen
 - Auf Register kann innerhalb eines Taktzyklus zugegriffen werden
 - ⇒ Daten sollten daher möglichst lange in Registern gehalten werden

Das Steuerwerk besteht aus:

- **Programmzähler**
 - Enthält Speicheradresse des nächsten auszuführenden Befehls
- **Befehlsregister**
 - Der nächste auszuführende Befehl wird in dieses Register geschrieben, bevor er vom Befehlsdekoder weiter verarbeitet wird.
- **Befehlsdekoder**
 - Befehle sind Bitfolgen, die den auszuführenden Befehl und seine Operanden (also in welchem Register sie gespeichert sind) kodieren.
 - Der Befehlsdekoder dekodiert die Befehle und weist die anderen Funktionseinheiten über Steuerleitungen an, was zu tun ist.

Beispiel: Addition zweier Zahlen A und B



Einschub: volatile Variablenzusatz

$$\begin{aligned}c &= a + b \\d &= a * c\end{aligned}$$

- Aus Performancegründen versucht der Compiler, **Variablenwerte möglichst lange in Registern zu halten.**
- Beispiel einer problematischen Programmausführung:
 - 1 Programm führt Anweisung `a = 3` aus (wird in Register R1 gehalten)
 - 2 Interrupt wird ausgelöst, Interrupt Service Routine wird ausgeführt
 - 1 ISR: Sichere Registerwerte des gerade laufenden Programms (u.a. Register R1)
 - 2 ISR: Führt Anweisung `a = 42` aus
 - 3 ISR: Stelle gesicherte Registerwerte wieder her (u.a. Register R1)
 - 3 Programm rechnet mit falschen Wert für `a` weiter

⇒ Wenn das Programm den aktuellen Variablenwert nur im Register hat, und Interrupt Service Routine diesen ändert, dann geht Änderung wieder verloren.
- Wenn Variable als `volatile` gekennzeichnet wird, dann wird der Compiler den aktuellen Variablenwert immer im Hauptspeicher halten bzw. von dort holen (Deklaration: `volatile int a;`).
- `volatile` verhindert auch andere Compileroptimierungen.

- Die **Befehlssatzarchitektur** (englisch: **Instruction Set Architecture (ISA)**) bezeichnet die nach außen sichtbare Schnittstelle zwischen Hardware und Software, die die verfügbaren Maschinenbefehle, Datentypen, Register, Adressierungsmodi und Speicherzugriffsmechanismen definiert.
- Eine wichtige Unterscheidung von unterschiedlichen ISA ist, woher die Operanden von Befehlen kommen können.
- Drei grundlegende Möglichkeiten:
 - **Load-Store Architekturen**
 - **Register-Memory Architekturen**
 - **Stack Architekturen**

- Load-Store Architektur

- Operanden von Befehlen können nur in Registern sein.
- Müssen vorher aus dem Hauptspeicher in ein Register geladen werden (via Load-Befehl)
- Und danach wieder vom Register in den Hauptspeicher zurückgeschrieben werden (via Store-Befehl).
- z.B. PowerPC, ARM, MIPS Befehlssätze

- Register-Memory Architektur

- Operanden von Befehlen können sowohl in Registern sowie direkt im Hauptspeicher sein.
- z.B. x86 Befehlssatz

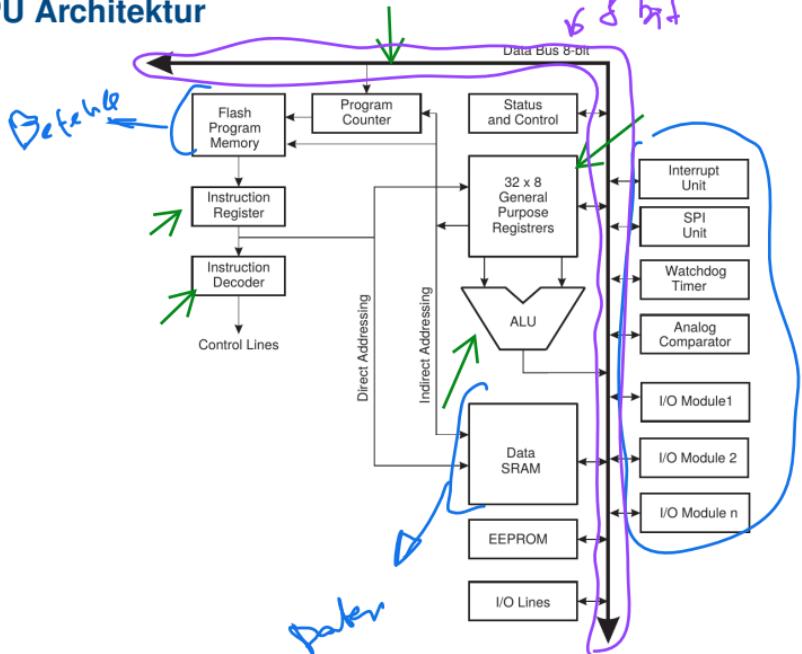
- Stack Architektur

- Operanden von Befehlen können nur in Registern sein, die in Form eines Stacks organisiert sind.
- Hauptsächlich bei virtuellen CPU-Architekturen zu finden.
- Vorteil sind unendlich viele Register (zumindest in der Theorie).
- z.B. Java Virtual Maschine

Aufbau von Computersystemen

AVR CPU Architektur

Harvard Arch



- ① Die **Bittigkeit** eines Prozessors gibt an, wie viele Bits gleichzeitig verarbeitet, adressiert oder übertragen werden können.
 - Gibt Breite der Register, der Datenbusse und der Adressbusse vor.

- ② z.B.: Addition zweier 64-Bit Zahlen
 - 8-Bit CPU: 8 Verarbeitungsschritte (Verarbeitung erfolgt in 8-Bit Häppchen)
 - 32-Bit CPU: 2 Verarbeitungsschritte (Verarbeitung erfolgt in 32-Bit Häppchen)
 - 64-Bit CPU: 1 Verarbeitungsschritt (Verarbeitung erfolgt in 64-Bit Häppchen)

① Fetch

- Speichere den nächsten Befehl aus dem Speicher in das Befehlsregister.
- Ändere den Programmzähler, damit er auf den nächsten Befehl zeigt.

② Decode

- Den Typ und die Operanden des Befehls im Befehlsregister bestimmen, und die Steuerleitungen dementsprechend setzen.

③ Load (Optional)

- Falls notwendig, Daten aus Speicher laden.

④ Execute

- Befehl ausführen.

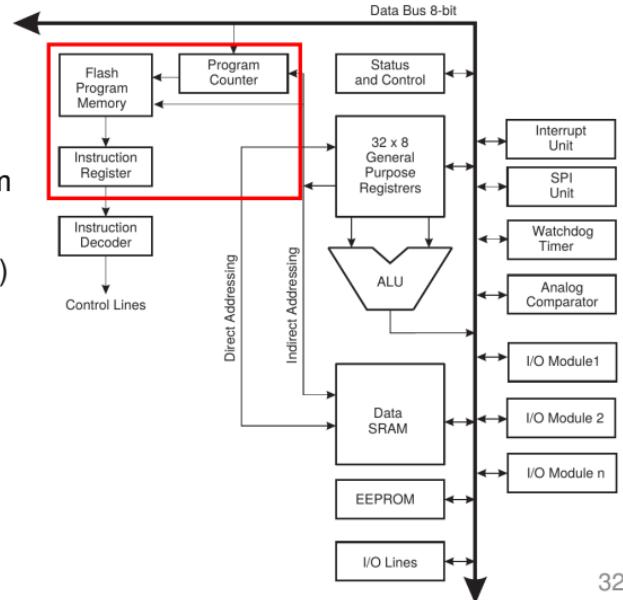
⑤ Store/Writeback (Optional)

- Falls notwendig, Daten in Speicher zurückschreiben

- Die genannten 5 **Phasen** (auch **Stages** genannt) der Befehlsausführung sind ein **theoretisches Modell**.
- Und selbst da gibt es **Unterschiede je nach Lehrbuch**.
 - Die optionalen Load/Store-Phasen werden manchmal weggelassen,
 - oder haben eine leicht unterschiedliche Bedeutung (z.B. Load: Laden aus Arbeitsspeicher vs. Laden aus Registerspeicher)
 - Auslagern mancher Arbeitsschritte in eigene Phasen (z.B. Anpassen des Programmzählers als eigene Phase)
- In der Praxis können die Phasen auch noch **weiter unterteilt werden**.
 - Z.B. Die Intel Pentium 4 Prescott Architektur hat 31 Phasen.

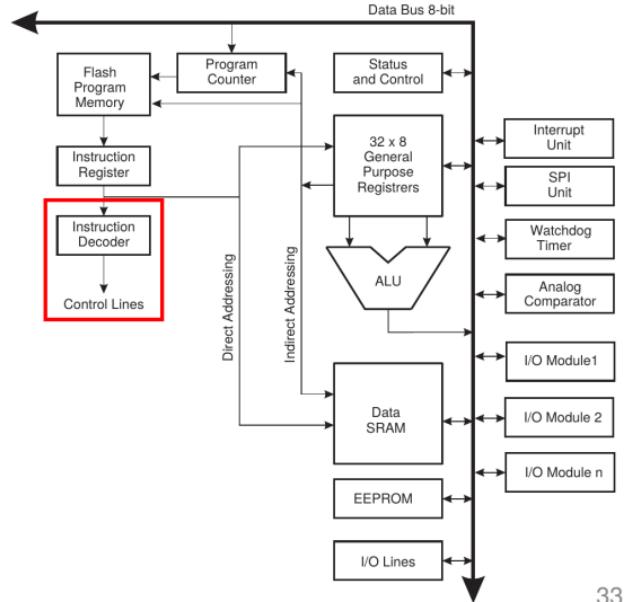
1 Fetch:

- Ladet Befehl aus Speicher in Befehlsregister (Adresse steht im Programmzähler (PC))
- $PC = PC + 1$ (Befehl, nicht Byte!)



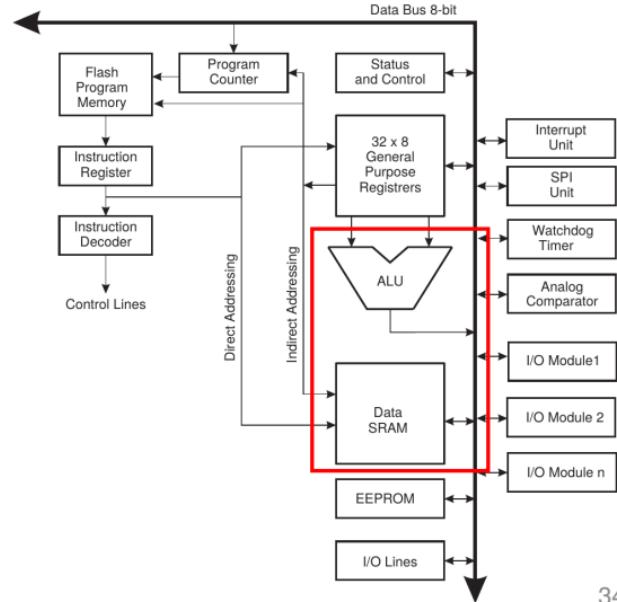
② Decode:

- Befehlsdecoder setzt Steuerleitungen.
- Dadurch wissen die anderen Komponenten, was zu tun ist.



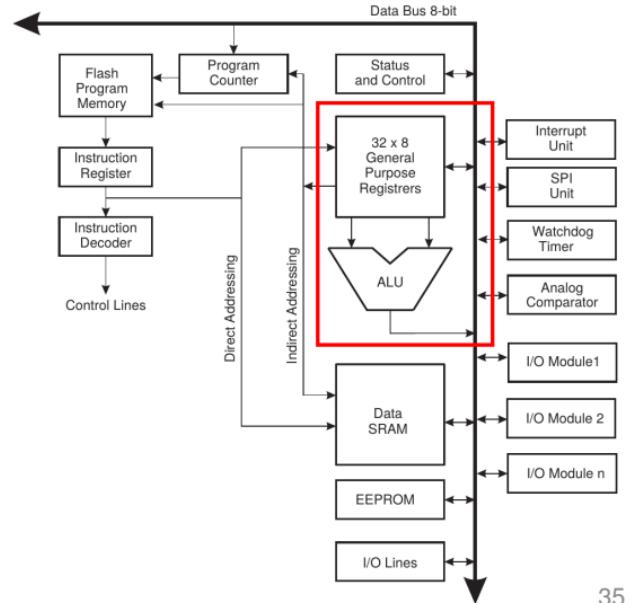
③ Load:

- Lade Operanden aus Speicher.



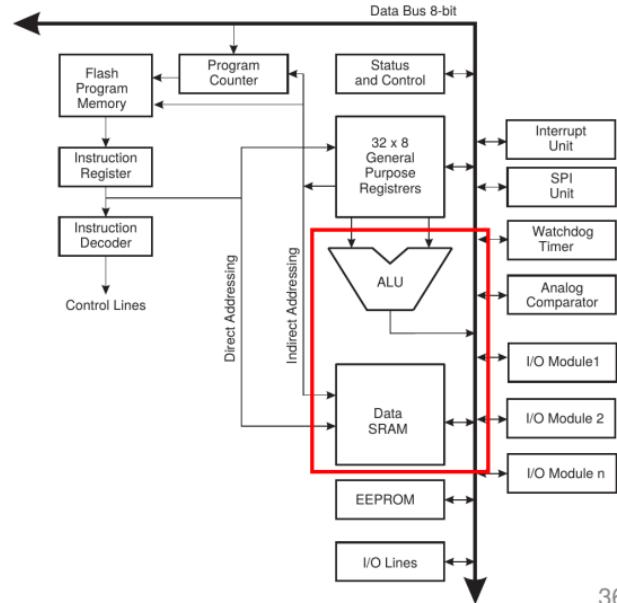
④ Execute:

- Führe Befehl aus.



5 Store:

- Speichere Ergebnis zurück in den Speicher.



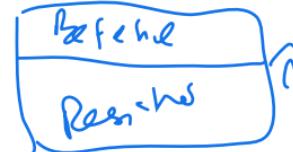


- Complex Instruction Set Computer (CISC)

- **Umfangreicher Befehlssatz** mit z.T. sehr komplexen Befehlen
 - Fokus liegt auf möglichst umfangreicher Abdeckung aller Einsatzszenarien
 - Es stehen vergleichsweise wenig Register zur Verfügung
 - Einzelne Befehle brauchen lange zum Verarbeiten
 - Die meisten Programme brauchen nur einen Bruchteil des Befehlsumfang

- Reduced Instruction Set Computer (RISC)

- **Reduzierter Befehlssatz** mit nur wenigen elementaren Befehlen
 - Fokus liegt auf möglichst schnelle Abarbeitung der einzelnen Befehle
 - Komplexe Befehle werden durch mehrere simple Befehle bzw. Miniprogramme ersetzt
 - Es stehen vergleichsweise viele Register zur Verfügung



z.B. multiplikation durch Addition

- **CISC/RISC Hybrid Rechner**

- RISC hat sich in den meisten Szenarien als das bessere Konzept erwiesen.
- Aus Gründen der Abwärtskompatibilität wird aber immer noch an CISC-Befehlssätzen festgehalten (z.B. x86-Befehlssatz).
 - ⇒ Man verwendet einen RISC Prozessorkern mit einer CISC-Befehlsdekodiereinheit.
 - ⇒ Auf der MikroarchitekturEbene (Ebene 1) werden CISC-Befehle durch ein oder mehrere RISC-Befehle (Mikrocode Programme) ersetzt

CPI: Cycles per Instructions

- Wieviele Taktzyklen pro Befehl braucht der Prozessor durchschnittlich über alle Befehle

je kleiner CPI desto schneller CPU

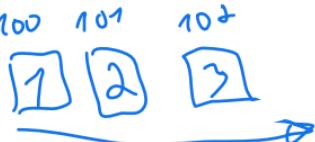
IPC: Instructions per Cycle

- Wieviele Befehle pro Taktzyklus kann ein Prozessor abarbeiten
- Bei einfachen Prozessoren: $IPC = \frac{1}{CPI}$
- Bei komplexeren Prozessoren kein unmittelbarer Zusammenhang zwischen IPC und CPI

IPS: Instructions per Seconds

- Wieviele Befehle pro Sekunde kann ein Prozessor abarbeiten
- Kann mit Vorsilben versehen werden, z.B. MIPS: Million Instructions per Seconds

- Ist der Teil des Computers, der die Programme und Daten aufnimmt.
- Der Prozessor kann direkt darauf zugreifen.
- Wir daher auch **Primärspeicher** genannt.
- Besteht aus einer Reihe von **Speicherzellen** bzw. -stellen.
 - Speichern mehrere Bits (i.d.R. 8 Bit)
 - Werden durch Adressen angesprochen.
 - Bilden die kleinste adressierbare Dateneinheit (wird **Byte** genannt).
- Größe des Adressbusses bestimmt maximal adressierbaren Speicher.
 - 32-Bit Adressbus: max 4 GiB Speicher adressierbar
 - 64-Bit Adressbus: max 4 EiB Speicher adressierbar



Zwei unterschiedliche Konventionen, wie mehrere Byte große Datentypen adressiert werden:

- Big Endian

- Adressierung beginnt am "großen" (höherwertigen) Ende (**Most Significant Byte**)
- Adressen von links nach rechts aufsteigend.
- z.B.: Sparc-, PowerPC-Prozessoren



- Little Endian

- Adressierung beginnt am "kleinen" (niederwertigen) Ende (**Least Significant Byte**)
- Adressen von rechts nach links aufsteigend.
- z.B.: x86-, Atmel-AVR-Prozessoren

Beispiel: Die Zahl $305\ 419\ 896_{10} = 12\ 34\ 56\ 78_{16}$

- Big Endian

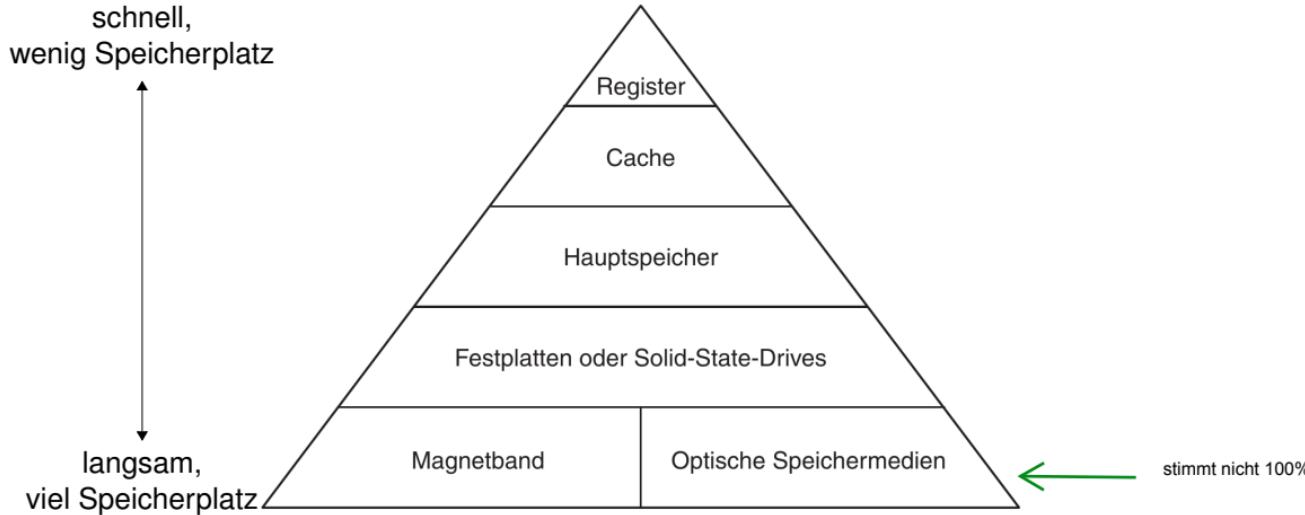
Adresse:	0	1	2	3
Zelleninhalt:	12	34	56	78

- Little Endian

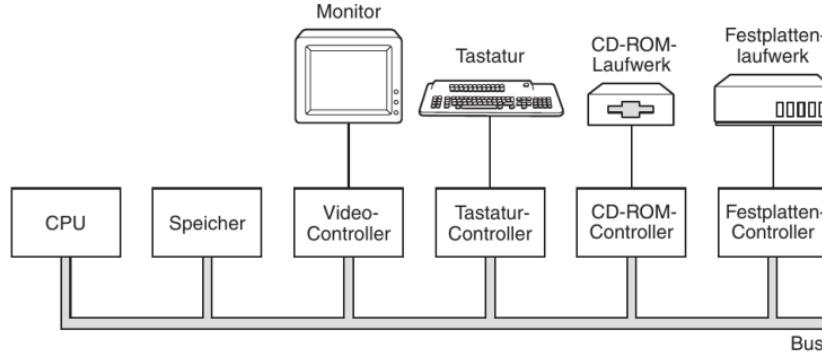
Adresse:	0	1	2	3
Zelleninhalt:	78	56	34	12

- Für Menschen aus dem westlichen Kulturkreis ungewohnt.

- Hauptspeicher kann nicht alle Daten aufnehmen.
 - Hauptspeicher speichert Daten nicht persistent.
- ⇒ Für die **persistente** Speicherung von **großen** Datenmengen werden sog. **Sekundärspeicher** verwendet.
- z.B.: Festplatten, SSDs, DVDs, Blu-rays, Magnetbänder, ...
 - Prozessor kann Daten von Sekundärspeichern nicht direkt verarbeiten.
- ⇒ Daten müssen zur Verarbeitung vorher in den Hauptspeicher geladen werden.

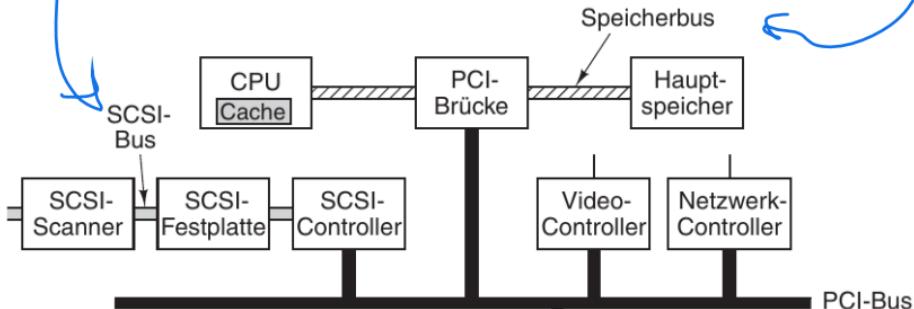


- Damit der Prozessor zu seinen Daten kommt, und man sich das Ergebnis anschauen kann, braucht es auch noch Ein- und Ausgabegeräte.
- Diese Ein- und Ausgabegeräte werden i.d.R. über ein oder mehrere **Bussysteme** angeschlossen.



- Ein- und Ausgabegeräte bestehen i.d.R. aus zwei Teilen:
 - Dem Controller
 - Dem eigentlichen Eingabe/Ausgabe-Gerät
- Der Controller steuert das E/A-Gerät und realisiert den Buszugriff.
- Manche Controller können ohne Umweg über den Prozessor direkt auf dem Hauptspeicher zugreifen.
 - Wird [Direct Memory Access \(DMA\)](#) genannt.
 - Muss vom Bussystem unterstützt werden.
 - Prozessor stoßt Übertragungsvorgang an, Controller führt selbstständig Übertragungsvorgang durch und informiert am Ende den Prozessor i.d.R. über einen Interrupt.
 - Entlastet den Prozessor, hat aber [Sicherheitsimplikationen](#).

- Bussysteme sind oft hierarchisch aufgebaut.
- Schnelle Bussysteme verbinden CPU mit wichtigsten Komponenten
- Langsamere Bussysteme verbinden die restlichen Komponenten



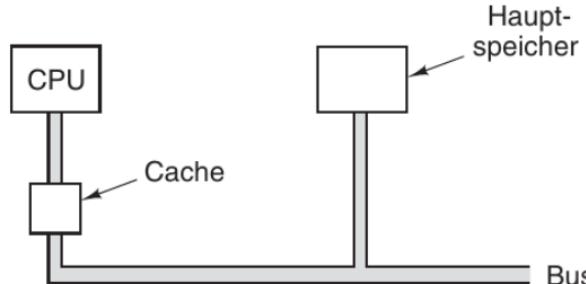
Einführung

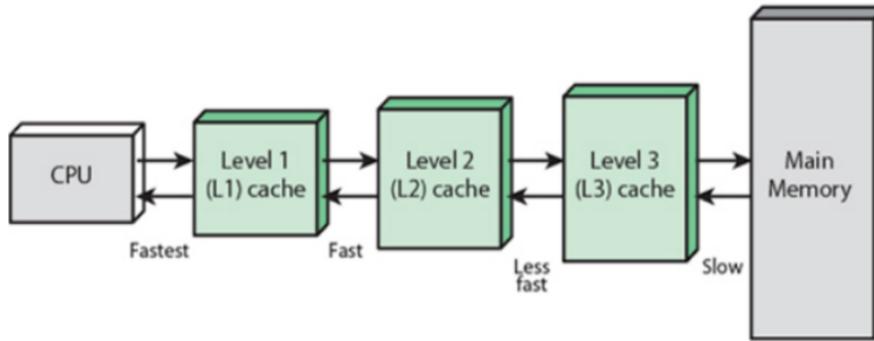
Aufbau von Computersystemen

Methoden zur Leistungssteigerung

- Moderne Prozessoren verwenden technisch ausgefeilte Methoden zur Leistungssteigerung.
 - Cache-Systeme
 - Pipelining
 - Superskalarität
 - Out-of-Order Execution

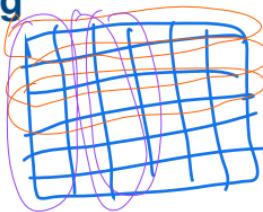
- Zugriff auf Hauptspeicher ist vergleichsweise langsam.
- Diskrepanz zwischen Prozessor- und Hauptspeichergeschwindigkeit wächst immer mehr.
- Zur Leistungssteigerung wird zwischen Prozessor und Hauptspeicher ein schneller Zwischenspeicher platziert (den sog. Cache).
 - Befindet sich i.d.R. direkt auf dem Prozessor-Die.





- Cache-Systeme bestehen oft aus mehreren hierarchisch organisierten Cache-Speichern (z.B.: L1-, L2- und L3-Cache).
 - L1: Schnellster aber kleinster Cache
 - L3: Größter aber langsamster Cache

- Caches nutzen das [Lokalitätsprinzip](#) aus.
 - **Zeitliche Lokalität:** Die Wahrscheinlichkeit ist sehr hoch, dass gerade benutzte Daten auch in naher Zukunft wieder benötigt werden (z.B. Variablen, auf die in kurzer Zeit immer wieder zugegriffen werden).
 - **Räumliche Lokalität:** Die Wahrscheinlichkeit ist sehr hoch, dass benötigte Daten im Speicher sehr nahe beieinander liegen (z.B. Arrayelemente, auf die in einer Schleife zugegriffen wird).
- Um die zeitliche Lokalität ausnutzen zu können, werden Daten, die aus dem Hauptspeicher geladen werden, auch immer im Cache abgelegt.
- Um die räumliche Lokalität ausnutzen zu können, werden nicht nur die benötigten Daten geladen, sondern auch alle benachbarten (die sog. [Cache-Line](#) als kleinste Dateneinheit im Cache).
 - Viele Architekturen verwenden 64-Byte Cache-Lines.



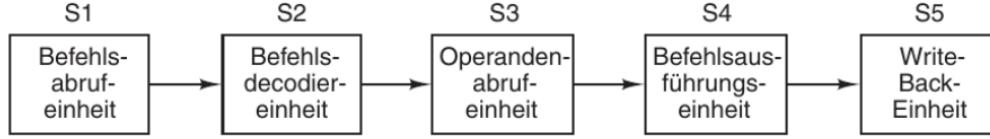
- Beim Programmieren sollte darauf Wert gelegt werden, das Lokalitätsprinzip bestmöglichst zu unterstützen.
 - z.B. in einem C/C++ Programm werden mehrdimensionale Arrays zeilenweise im Hauptspeicher abgelegt.
 - Daher sollten in einem C/C++ Programm mehrdimensionale Arrays auch bevorzugt zeilenweise durchlaufen werden.

- Caches sind bedeutend kleiner als der Hauptspeicher.
- Wenn ein neuer Wert im Cache abgelegt werden soll, muss oft ein alter Wert gelöscht werden, um Platz zu machen.
- Dafür gibt es mehrere Verdrängungsstrategien:
 - First In First Out (FIFO): Der jeweils älteste Wert wird verdrängt.
 - Least Recently Used (LRU): Der Eintrag, auf den am längsten nicht zugegriffen wurde, wird verdrängt.
 - Least Frequently Used (LFU): Der am seltensten gelesene Eintrag wird verdrängt.
 - ...

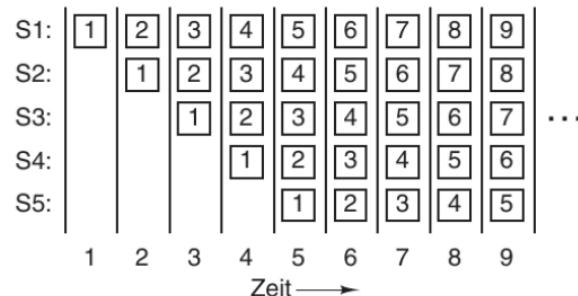
- Die Befehlsausführung ist in mehrere Phasen gegliedert.
- Die unterschiedlichen Phasen können gleichzeitig für verschiedene Befehle ausgeführt werden.
- Dieses Konzept nennt man **Pipelining**.
- Die dafür benötigte Hardware nennt man **Pipeline**.
bevor ein Befehl fertig ist wird andere angefangen
- Kann man sich wie Fließbandarbeit vorstellen.
 - Mehrere hintereinandergeschaltete Arbeitsstationen für genau spezifizierte Arbeitsschritte.
 - Produkt (Befehl) wandert von Station zu Station
- Erhöht den Durchsatz (ermöglicht IPC von 1)

Methoden zur Leistungssteigerung

Pipelining: Visualisierung



(a)



(b)

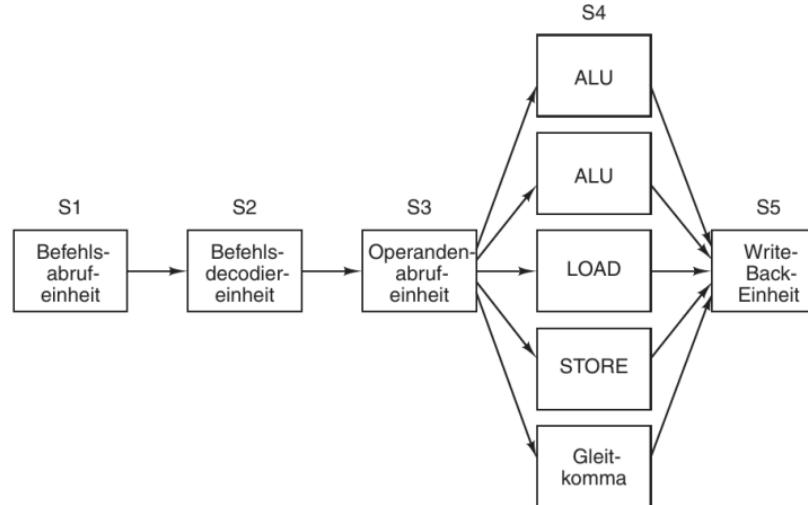
- Problem: Bei Verzweigungen ist nicht klar, welcher Befehl als nächstes kommt.
 - Lösung: **Sprungvorhersage** und **spekulative Ausführung**
 - Es wird versucht, Sprünge vorherzusagen und Befehle dementsprechend “auf gut Glück” auszuführen.
 - Wenn Sprung falsch vorhergesagt wurde, dann muss das Ergebnis der spekulativen Berechnungen verworfen werden.
- ⇒ Sprungvorhersagealgorithmus hat Auswirkungen auf Performance.
- ⇒ Eine der Ursachen der “Spectre” und “Meltdown” Sicherheitslücken.

- Superskalarität bedeutet, dass mehrere Befehle gleichzeitig von mehreren parallel arbeitenden Funktionseinheiten verarbeitet werden.
- Ein *n*-fach superskalarer Prozessor besitzt *n* parallel arbeitende Funktionseinheiten.
- Frühe Implementierungen arbeiteten mit mehreren parallelen Pipelines.
- Moderne Implementierungen arbeiten mit einer Pipeline, die in manchen Stages mehrere Befehle gleichzeitig verarbeiten können.
- Erhöht den Durchsatz (ermöglicht IPC größer als 1).
- Basis für Hyperthreading (virtuelle Cores nutzen brachliegende Funktionseinheiten).

Methoden zur Leistungssteigerung

Superskalarität: Visualisierung

Moderne 5-fach superskalare Pipeline:



- In-Order Execution
 - Befehle werden in genau der Reihenfolge abgearbeitet, in der sie im Programmcode stehen.
- Out-of-Order Execution
 - Befehle können in einer anderen Reihenfolge abgearbeitet werden, als sie im Programmcode stehen
- Die Idee ist, die Auslastung der Pipeline und der Funktionseinheiten bei superskalaren Prozessoren zu verbessern.
- Von den vorigen Befehlen unabhängige Befehle können vorgezogen werden.
- Korrektheit des Programmes muss gewährleistet sein.
 - Das Ergebnis muss dasselbe sein wie bei In-Order-Execution.

Beispielcode

1: ~~a = b + c~~
2: ~~d = e + f~~
3: g = a * d
4: x = y + z



- Befehl 3 ist von Befehl 1 und 2 abhängig.
 - Muss in der Pipeline verzögert werden bis Ergebnisse von 1 und 2 vorliegen.
 - Kann bei super-skalaren Prozessoren nicht parallel zu Befehlen 1 und 2 ausgeführt werden.
- Befehl 4 ist von den vorhergehenden Befehlen unabhängig.
 - Vorziehen macht Verzögerung von Befehl 3 in der Pipeline überflüssig.
 - Kann bei super-skalaren Prozessoren parallel zu Befehlen 1 und 2 ausgeführt werden.

- Die CPU Performance ist nicht nur von der Taktfrequenz verfügbar.
- Viele andere Kriterien beeinflussen auch CPU Performance
 - RISC vs CISC
 - Länge der Pipeline (kürzer ist i.d.R. besser)
 - Güte der Sprungvorhersage
 - Superskalarität
 - In-Order vs Out-of-Order Execution
 - Größe der Caches
 - ...

5 RECHNERARCHITEKTUR

▼ Logikgatter



Was sind Logikgatter?

Logikgatter (auch *logische Gatter* oder *Schaltgatter*) sind die **Grundbausteine digitaler Schaltungen**.

Sie verarbeiten **binäre Eingangssignale (0 oder 1)** und liefern **ein Ausgangssignal** basierend auf einer **logischen Funktion** (z. B. UND, ODER, NICHT).



Grundlegende Logikgatter

Gatter	Symbol	Beschreibung	Wahrheitstabelle
NOT	$\neg A$	Invertiert das Eingangssignal	$A \rightarrow Y: 0 \rightarrow 1, 1 \rightarrow 0$
AND	$A \wedge B$	Ergebnis ist 1, wenn beide 1 sind	$0 \wedge 0 = 0, 0 \wedge 1 = 0, 1 \wedge 0 = 0, 1 \wedge 1 = 1$
OR	$A \vee B$	Ergebnis ist 1, wenn mindestens eins 1 ist	$0 \vee 0 = 0, 0 \vee 1 = 1, 1 \vee 0 = 1, 1 \vee 1 = 1$
NAND	$\neg(A \wedge B)$	Gegenteil von AND	Nur 1, wenn nicht beide 1 sind
NOR	$\neg(A \vee B)$	Gegenteil von OR	Nur 1, wenn beide 0 sind
XOR	$A \oplus B$	Ergebnis ist 1, wenn genau eines 1 ist	$0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0$
XNOR	$\neg(A \oplus B)$	Ergebnis ist 1, wenn beide gleich sind	$0 \rightarrow 1, 1 \rightarrow 1, \text{ sonst } \rightarrow 0$



Anwendungen von Logikgattern

- In jedem **Mikroprozessor und Mikrocontroller** verbaut
- Grundlage für:
 - Addierer, Multiplexer, Register, ALUs
 - Steuerlogik in FSMs (Endlichen Automaten)
- Bildung komplexer logischer Ausdrücke (z. B. $(A \&& !B) || C$)



Zusammenhang mit bitweisen Operatoren

Logikgatter \triangleq bitweise Operatoren in C/Arduino:

Gatter	C-Operator	Beispiel	Ergebnis
AND	&	<code>0b1010 & 0b1100</code>	<code>0b1000</code>
OR	<code>'</code>	<code>'0b1010</code>	<code>'0b1010</code>

Gatter	C-Operator	Beispiel	Ergebnis
XOR	\wedge	0b1010 \wedge 0b1100	0b0110
NOT	\sim	\sim 0b1010	0b0101...*

- abhängig von Bitbreite (z. B. 8 Bit = \sim 0b1010 = 0b11110101)



Grafische Symbole (nach IEC)

- Rechteck mit Gattertyp: z. B. AND, OR, NOT
- US-Symbole: bekannte "Blasenformen", z. B. Spitze für OR, Halbkreis für AND



Kombination von Gattern – Beispiel: Lichtsteuerung

Beispiel: Eine Lampe soll **nur** leuchten, wenn:

- der **Schalter A ein** ist,
- **und nicht gleichzeitig** Schalter B.

Logik: A AND NOT B \Rightarrow Schaltung: $A \wedge \neg B$



Quellen

1. [Wikipedia: Logikgatter](#)
2. [Studyflix: Logikgatter](#)

✓ Zusammenfassung

Zweck	Logikgatter (Beispiele)
Invertieren	NOT ($\neg A$)
Beide Bedingungen erfüllt	AND ($A \wedge B$)
Eine Bedingung reicht	OR ($A \vee B$)
Nur eins darf wahr sein	XOR ($A \oplus B$)
Gegenteil von AND/OR	NAND/NOR

▼ Cache-Systeme



Was ist ein Cache?

Ein **Cache** ist ein **kleiner, sehr schneller Speicher**, der zwischen der CPU und einem langsameren Hauptspeicher (RAM) liegt.

Ziel: **Zugriffszeit auf Daten verkürzen**, indem häufig benutzte Daten **zwischengepuffert** werden.



Warum ist ein Cache notwendig?

- CPUs sind **viel schneller** als RAM

- Ohne Cache müsste die CPU **ständig auf langsame Speicher warten**
 - Cache-Systeme überbrücken diesen Geschwindigkeitsunterschied
-



Typische Cache-Hierarchie

Ebene	Bezeichnung	Größe	Geschwindigkeit	Zugriff durch
L1-Cache	Level 1	~16–64 KB	Sehr schnell	Nur CPU direkt
L2-Cache	Level 2	~256 KB–1 MB	Schnell	CPU
L3-Cache	Level 3	~4–64 MB	Langsamer	Geteilt von mehreren Cores
RAM	Hauptspeicher	~8–64 GB	Deutlich langsamer	Gesamtes System



Wie funktioniert ein Cache?

1. CPU benötigt Daten (z. B. Variable, Anweisung)
 2. **Cache wird zuerst überprüft**
 3. Wenn Daten **vorhanden (Cache-Hit)** → sofortige Nutzung
 4. Wenn **nicht vorhanden (Cache-Miss)** → Daten werden aus RAM geholt und im Cache gespeichert
-



Arten von Cache-Zugriffsstrategien

Strategie	Beschreibung
Write-Through	Schreibvorgang erfolgt gleichzeitig in Cache und Hauptspeicher
Write-Back	Daten werden zuerst nur im Cache geändert, später in den RAM geschrieben
Write-No-Allocate	Bei einem Schreib-Miss wird nicht in den Cache geladen



Cache-Organisation: Mapping-Verfahren

◆ Direktes Mapping

- Jede Speicherstelle im RAM hat genau **einen festen Platz im Cache**
- Einfach, aber es kommt häufig zu **Kollisionen** (Miss-Risiko hoch)

◆ Vollassoziativ

- Ein beliebiger Speicherblock kann an **jeder Cache-Stelle** abgelegt werden
- Aufwändiger, aber **höhere Flexibilität**

◆ Set-assoziativ

- Kompromiss: Cache ist in **Sets unterteilt**, jedes Set enthält mehrere Plätze
- Speicheradresse wird einem Set zugeordnet, innerhalb des Sets ist Auswahl frei



Begriffserklärungen

- **Cache Hit:** Gesuchter Datensatz ist bereits im Cache vorhanden → sehr schnell
- **Cache Miss:** Datensatz fehlt → muss aus langsamem RAM geholt werden
- **Hit Rate:** Anteil der Zugriffe, die erfolgreich im Cache gefunden werden



Verhalten aus Programmierersicht (z. B. bei Mikrocontrollern)

- Viele Mikrocontroller (z. B. ATmega-Serie) **haben keinen Cache** → deterministisches Zeitverhalten
- In **leistungsstarken Embedded-Prozessoren (z. B. ARM Cortex-A)** spielt Cache eine große Rolle!
- Entwickler können durch **Cache-optimisierten Code** viel Performance gewinnen:
 - Daten lokal halten
 - Schleifen optimieren
 - Cache-freundliche Datenstrukturen nutzen



Typische Probleme / Herausforderungen

- **Cache Coherency:** Bei Multicore-CPUs müssen Caches synchron bleiben
- **False Sharing:** Zwei Cores greifen scheinbar auf denselben Cacheblock zu
- **Determinismus:** Bei sicherheitskritischen Echtzeitsystemen ist Caching teils problematisch



Quellen

1. [Wikipedia – Cache](#)
2. [Elektronik-Kompendium – Cache](#)
3. [Wikibooks – Computerhardware: Cache](#)



Zusammenfassung

Merkmal	Bedeutung
Zweck	Schnelleren Zugriff auf oft benötigte Daten
Typen	L1, L2, L3 (hierarchisch)
Strategien	Write-Through, Write-Back etc.
Mapping	Direkt, Assoziativ, Set-assoziativ
Performance-Faktor	Hit-Rate entscheidend

▼ Pipelining und spekulativer Ausführung



1. Was ist Pipelining?

Pipelining ist eine Technik zur **Beschleunigung von Prozessoren**, indem man die **Abarbeitung einzelner Instruktionsschritte parallelisiert**.

⌚ Ziel:

Mehrere Befehle gleichzeitig verarbeiten – **jeder in einer anderen Phase**.



Beispiel: 5-stufige Pipeline

Stufe	Bedeutung
IF	Instruction Fetch
ID	Instruction Decode
EX	Execute (z. B. ALU)
MEM	Memory Access
WB	Write Back (Ergebnis speichern)

➡ In jedem Taktzyklus wird **eine neue Instruktion gestartet**, während vorherige noch in späteren Stufen arbeiten.



Vorteil: Höherer Durchsatz

Ohne Pipelining:

- 1 Befehl dauert z. B. 5 Takte
→ 5 Befehle = 25 Takte

Mit Pipelining:

- 1 Befehl dauert ebenfalls 5 Takte
- Aber: Danach **1 Befehl pro Takt**
→ 5 Befehle = **9 Takte**



Probleme im Pipelining

Problemtyp	Beschreibung	Beispiel
Datenkonflikt	Ein Befehl braucht ein Ergebnis, das noch nicht bereit ist	add, store
Kontrollkonflikt	Bei Sprüngen ist unklar, welcher Befehl folgt	if, jmp
Strukturkonflikt	Zwei Befehle benötigen dieselbe Hardware gleichzeitig	ALU + Speicher

➡ Lösung: **Stall, Forwarding, Branch Prediction**



2. Spekulative Ausführung

Definition:

Die CPU **rät** (spekuliert), **welcher Pfad eines Sprunges** genommen wird, und **führt schon mal Befehle aus**, bevor sicher ist, ob der Pfad stimmt.

Vorteil:

- **Minimiert Stalls** bei Verzweigungen
- Kombiniert mit **Branch Prediction** (Vorhersage des nächsten Sprungs)

Beispiel:

```
if (a < b) {  
    x = 1;  
} else {  
    x = 2;  
}
```

➡ CPU beginnt **beide Pfade vorzubereiten** oder rät einen



Risiko: Spekulative Ausführung + Caching

Meltdown & Spectre zeigten:

- Auch **spekulativ ausgeführte, eigentlich ungültige Befehle** können Auswirkungen auf **den Cache** haben
 - Diese Cache-Veränderungen kann man später **ausnutzen, um geheime Daten auszulesen**
- ➡ **Timing-Angriffe** auf spekulative Pfade
➡ Moderne CPUs benötigen **Mitigations** (z. B. Cache-Flushing, Retpolines)



Reale Beispiele:

- **Meltdown**: Zugriff auf Kernel-Speicher durch spekulatives Lesen
- **Spectre**: Zugriff auf fremde Speicherbereiche durch manipulierte Sprungvorhersage

Präsentation von Michael Schwarz (GLT 2018):

- 👉 https://media.ccc.de/v/GLT18_-330-de-g_ap147_005-201804281005-x-factor_das_unfassbare-die_geschichte_von_meltdown_und_spectre_-michael_schwarz
👉 [Foliensätze \(PDF\)](#)



Zusammenfassung: Pipelining & Spekulation

Technik	Zweck	Risiko / Herausforderung
Pipelining	Höherer Durchsatz	Daten- & Steuerkonflikte
Spekulative Ausführung	Minimiert Verzögerung durch Sprünge	Sicherheitslücken (Spectre)
Branch Prediction	Erhöht Erfolg von Spekulation	Falsch vorhergesagt → Rollback



Quellen

1. [Wikipedia – Pipeline \(Prozessor\)](#)
2. [Elektronik-Kompendium – Pipelining](#)
3. [GLT18 – Vortrag + Slides zu Spectre & Meltdown](#)

▼ Superskalarität



Was ist Superskalarität?

Ein **superskalarer Prozessor** kann **mehrere Instruktionen pro Taktzyklus** ausführen – im Gegensatz zu skalarer Architektur, wo nur **eine Instruktion pro Takt** verarbeitet wird.



Ziel:
Erhöhung des Durchsatzes (**Instructions per Cycle, IPC**) durch **Parallelausführung auf Hardwareebene** – ohne explizite Programmierung in Threads!



Grundidee:

Superskalare CPUs besitzen **mehrere Ausführungseinheiten** (Execution Units):

Komponente	Zweck
Integer-ALU	Ganzzahloperationen
FPU	Gleitkommazahlen
Load/Store Unit	Speicherzugriffe
Branch Unit	Sprungbefehle

→ Mehrere Einheiten = parallele Verarbeitung mehrerer Instruktionen



Vergleich: Skalar vs. Superskalar

Architektur	Instruktionen pro Takt (ideal)	Beispiel
Skalar	1	Klassische RISC-CPU
Superskalar	>1 (z. B. 2, 4, 8...)	Intel Core, AMD Ryzen



Funktionsweise einer superskalaren CPU

1. **Instruction Fetch:** Mehrere Instruktionen gleichzeitig holen
2. **Instruction Decode:** Parallel dekodieren
3. **Instruction Dispatch:** An verschiedene Ausführungseinheiten verteilen
4. **Out-of-Order Execution (optional):** Reihenfolge darf abweichen
5. **Commit:** Ergebnisse werden in der ursprünglichen Reihenfolge zurückgeschrieben



Herausforderungen bei Superskalarität

Problem	Erklärung
Datenabhängigkeiten	Eine Instruktion braucht das Ergebnis der vorherigen
Strukturkonflikte	Zwei Instruktionen wollen dieselbe Einheit nutzen
Kontrollkonflikte	Verzweigungen führen zu spekulativen Fehlern

➡ Lösungen:

- Register-Umbenennung (Register Renaming)
- Out-of-Order Execution
- Spekulative Ausführung + Branch Prediction



Beispiel – Dual-Issue Prozessor:

CPU kann pro Takt:

- **1 Add** in der ALU
- **1 Load** aus dem Speicher

```
ADD R1, R2, R3 ; Ganzzahloperation  
LOAD R4, [R5] ; Speicherzugriff
```

➡ Beide Instruktionen gleichzeitig → besserer Durchsatz



Technik für höhere Leistung (IPC)

Superskalärität ist Teil vieler moderner Architekturen:

- x86 (Intel Core i7, AMD Ryzen)
- ARM Cortex-A Serien
- PowerPC, MIPS

➡ Superskalärität ist **unsichtbar für den Programmierer**, aber wichtig für **Compileroptimierungen**

📚 Quelle:

- [Wikipedia – Superscalar processor](#)

✓ Zusammenfassung

Merkmal	Beschreibung
Definition	Mehrere Instruktionen pro Taktzyklus
Ziel	Höherer Durchsatz (IPC ↑)
Voraussetzung	Parallele, unabhängige Instruktionen
Grenzen	Datenkonflikte, Ressourcenmangel, Branches
Lösungstechniken	Out-of-Order, Register-Renaming, Prediction

▼ Out-of-Order Execution



Was ist Out-of-Order Execution (OoOE)?

Out-of-Order Execution ist eine Technik moderner Prozessoren, bei der **Befehle nicht streng in Programmreihenfolge ausgeführt werden**, sondern **möglichst früh, sobald ihre Operanden bereit sind**.

Ziel:

- Leerlauf vermeiden
- Rechenwerke auslasten
- Durchsatz (IPC) erhöhen



Warum braucht man OoOE?

Stellen wir uns folgende Befehlsfolge vor:

1. LOAD R1, [A] ; dauert lange (z. B. RAM-Zugriff)
2. ADD R2, R3, R4 ; sofort ausführbar
3. MUL R5, R1, R6 ; abhängig von 1.

Ein **in-order-Prozessor** würde:

- bei 1. warten, bis fertig,
- dann 2. und 3. ausführen

Ein **Out-of-Order-Prozessor**:

- führt 2. **sofort aus**, obwohl 1. noch nicht fertig ist,
- wartet nur mit 3., weil es R1 braucht.

 Nutzt Rechenzeit besser aus 



Wie funktioniert das intern?

Komponente	Aufgabe
Instruction Queue	speichert viele geladene, aber noch nicht ausgeführte Befehle
Register Renaming	verhindert Registerkonflikte (WAR, WAW) durch Umbenennen
Reservation Stations	warten auf Befehle + deren Operanden
Reorder Buffer (ROB)	sorgt dafür, dass die Ergebnisse in der Originalreihenfolge zurückgeschrieben werden
Scheduler	entscheidet, welche Einheit was ausführen darf



Voraussetzungen für OoOE

1. **Hardwareunterstützung** für:
 - Umbenennung von Registern
 - Mehrere parallele Ausführungseinheiten
 - Ergebnis-Zurückhaltung in Buffern
2. **Unabhängige Befehle**, die nicht aufeinander warten müssen



Probleme bei OoOE

Problem	Beschreibung
Datenabhängigkeiten	z. B. RAW (Read After Write), WAR (Write After Read), WAW (Write After Write)
Seitenkanalangriffe	OoOE kann Speicherbereiche spekulativ lesen , was Angriffe wie Spectre ermöglicht
Komplexität	Erfordert sehr komplexe Logik → große Siliziumfläche, mehr Energieverbrauch



Beispiel: Abhängigkeiten erkennen

- 1. $A = B + C$; unabhängig
- 2. $D = A + E$; abhängig von 1
- 3. $F = G + H$; unabhängig

Ausführungsreihenfolge (OoOE):

- - 1. → 3. → 2.
- (2 muss warten, 3 kann sofort laufen)



Sicherheitsaspekt (Spectre)

- OoOE führt **spekulative Anweisungen** aus, bevor klar ist, ob sie gebraucht werden
- Diese **spekulativen Speicherzugriffe** können **Nebeneffekte im Cache** hinterlassen
- Diese Nebeneffekte → **per Timing-Messung ausnutzbar** (Side-Channel-Angriff)

Quellen:

1. [Wikipedia – Out-of-Order Execution](#)
2. [Cadence Blog – How OoO processors work](#)

Zusammenfassung

Merkmal	Beschreibung
Ziel	Maximale Auslastung, weniger Leerlauf
Voraussetzung	Unabhängige Befehle, Register-Renaming
Ergebnis	Höherer Durchsatz (mehr IPC)
Komplexität	Erhöhte Hardwarekosten
Sicherheitsrisiko	Grundlage für Angriffe wie Spectre



ASSEMBLERPROGRAMMIERUNG

RECHNERSTRUKTUREN & EMBEDDED SYSTEMS

MATTHIAS JANETSCHEK
SS 2025

Inhaltsverzeichnis



Einführung

AVR Assembler

“Strukturierte Programmierung” in Assembler

Q&A Session

Inhaltsverzeichnis



Einführung

AVR Assembler

"Strukturierte Programmierung" in Assembler

Q&A Session

Das Wort “Assembler” hat zwei Bedeutungen:

- Als **Assembler** wird die **Maschinensprache** bezeichnet (die sog. **Assemblersprache**)
 - Strenggenommen bezeichnet Assemblersprache nur die **vom Menschen lesbare** Version der Maschinensprache.
 - Die Bitfolgen der eigentlichen Maschinensprache werden auch als **Opcode** bezeichnet.
 - Assemblersprache: `add r1, r2`
 - Opcode `0000 1100 0001 0010`
- Als **Assembler** wird auch das **Computerprogramm** bezeichnet, welches die Befehle der Assemblersprache durch den jeweiligen Opcode ersetzen.

- Eine Assemblersprache ist eine Programmiersprache, die auf eine **bestimmte Prozessorarchitektur** ausgerichtet ist.
 - Im Unterschied zu Hochsprachen.
- Zählt zu den Programmiersprachen der 2. Generation.
 - Statt direkt mit Bitfolgen zu programmieren, werden leichter verständliche Symbole verwendet (z.B. add xx, xx statt 0000 11xx xxxx xxxx).
 - Diese Symbole werden auch als **Mnemonics** (deutsch: Gedächtnishilfe) bezeichnet.
- Folgt **nicht** der strukturierten Programmierung.
 - Kennt keine Schleifen oder Verzweigungen.
 - Stattdessen werden bedingte bzw unbedingte Sprünge verwendet.

Wann kommt ein:e Programmierer:in mit Assembler in Berührung:

- Hochoptimierter Code für Platformen mit begrenzten Ressourcen.
 - z.B.: Verschlüsselungsroutinen oder Interrupt Service Routinen auf Mikrocontrollern.
- Systemnahe Programmierung
 - z.B.: Boot- bzw. Kernelroutinen für Betriebssysteme.
- Reverse Engineering
 - z.B.: Analyse von Schadsoftware
- Debugging
 - z.B.: Compiler-Bugs finden und analysieren

Einführung

Wie Assemblercode erhalten



Wie erhält man zu einem bestehenden Programm Assemblercode:

- Kompilieren mit -S:
 - Beim kompilieren mit gcc/g++ -S als Kommandozeilenargument übergeben.
 - Dann wird für jede .c/.cpp Datei eine .s Datei mit dem Assemblercode erzeugt.
- Nachträglich mit objdump:
 - objdump wandelt die Opcodes in einem Binary wieder in Assembler um.
 - Diese Art von Tools nennt man [Disassembler](#), und den Vorgang nennt man [disassemblieren](#).
- Es gibt ganze Tool Suites, die neben dissasemblieren auch den Assemblercode analysieren können.
 - z.B.: Ghidra (Open-Source), ursprünglich von der NSA entwickelt
 - z.B.: Interactive Disassembler (IDA) von Hex-Rays

- Ein Dekompilierer (engl. **Decompiler**) geht noch einen Schritt weiter.
 - Versucht aus Assemblercode wieder Quelltext einer Hochsprache zu erzeugen.
- Funktioniert ganz gut bei Bytecode-Sprachen (z.B. Java und C#)
 - Bytecode enthält i.d.R. mehr Informationen über Hochsprachenkonstrukte als Assemblercode.
 - Bytecode benötigt daher zusätzliche Maßnahmen, um Reverse Engineering zu erschweren (z.B. Code-Obfuscation, Code-Verschlüsselung).
- Funktioniert weniger gut bei kompilierten Sprachen (z.B.: C/C++)
 - Bei der Übersetzung von der Hochsprache in den Assemblercode geht zu viel Informationen verloren.

- Compiler wenden sehr viele Tricks zur Optimierung von Code an
 - Assembler-Anweisungen werden oft "zweckentfremdet", weil sie mehr Features bieten oder schneller sind als die eigentlich dafür gedachten Anweisungen.
- Z.B.: Wie schreibt man in x86 am effizientesten die Zahl 0 in ein Register?
 - Mit `mov %Register, 0` (Ladet die Konstante 0 in das Register)?
 - Nein, mit `xor %Register, %Register` (xor-Verknüpfung mit sich selber)!
- Z.B.: Verwendung von `lea` (Speicheradressenberechnung in x86) anstatt von `add` (`lea` unterstützt mehr als 2 Operanden und kann Ergebnis in beliebiges Zielregister schreiben).
⇒ Um Assemblercode korrekt zu interpretieren, muss man diese Tricks kennen!¹

¹Siehe <https://www.agner.org/optimize/#manuals> für weitere Beispiele

Inhaltsverzeichnis



Einführung

AVR Assembler

"Strukturierte Programmierung" in Assembler

Q&A Session

- AVR Assembler wird auf den Atmel AVR Mikrocontrollern verwendet.
 - Weitgehende Kompatibilität zwischen den einzelnen Mikrocontrollern.
- Bei der AVR Familie handelt es sich um **8-Bit Mikrocontroller**.
 - Datenbus und Register sind 8-Bit groß.
 - Die meisten Anweisungen verarbeiten Daten in 8-Bit Häppchen.
 - Mehrere Byte große Datentypen werden in mehreren Schritten verarbeitet.
- Der ATMega32u4 besitzen einen **16-Bit Adressbus**.
 - Normalerweise sind Daten- und Adressbus gleich groß, es gibt aber Ausnahmen (z.b. alle größeren 8-Bit Mikrocontroller).
 - Maximal adressierbarer Speicher: 2^{16} Byte = 65 KB
 - Speicherbedarf eines Pointers: 16 Bit = 2 Byte

- AVR Assembler ist ein **RISC Befehlssatz**.
 - Manche Anweisungen fehlen (z.B. keine Division, keine Gleitkomma-Befehle).
 - Fehlende Anweisungen müssen durch **entsprechende Unterprogramme substituiert werden** (siehe Hausübungsbeispiel).
- Die meisten Anweisungen sind 16-Bit groß.
- Einige wenige Anweisungen sind 32-Bit groß.
 - Z.B. call, jmp, lds und sts;
 - Kodieren Speicheradressen, die in einer 16-Bit Anweisung nicht Platz haben.

- AVR Mikrocontroller implementieren eine Load-Store Architektur
 - Daten müssen zuerst vom Hauptspeicher in Register geladen werden, bevor sie von der CPU verarbeitet werden können.
- AVR folgt der Harvard-Architektur (getrennter Befehls- und Arbeitsspeicher)
 - Jeweils unterschiedliche Load/Store-Befehle für Befehls- und Arbeitsspeicher.
- Die **AVR Instruction Set Architecture (ISA)** besteht aus:
 - **Bis zu 124 Befehle** (abhängig vom konkreten Mikrocontroller)
 - **1x 8-Bit Status Register (SREG)**
 - **32x 8-Bit General Purpose Register (R0 - R21)**
R0 - R31 ist richtig
 - **1x 16-Bit Stackpointer Register (SPL/SPH)** (Benötigt für Stackverwaltung)

AVR Assembler

Status Register (SREG)



Bit	7	6	5	4	3	2	1	0	SREG
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

- Das Status Register enthält **Statusinformationen der zuletzt ausgeführten arithmetischen Anweisung**.
 - Wird für bedingte Sprünge oder zusammengesetzte Anweisungen verwendet.
 - Wird nach jeder arithmetischen Anweisung aktualisiert, d.h. Statusinformation muss sofort verarbeitet werden.

AVR Assembler

Status Register (SREG)



- Bit 7 – I: Global Interrupt Enable
 - Schaltet alle Interrupts entweder ein oder aus.
- Bit 6 – T: Bit Copy Storage
 - Wird von der `bld` und `bst` Anweisung verwendet.
- Bit 5 – H: Half Carry Flag
 - Zeigt Überlauf in den 4 least significant Bits an (wird für BCD-Rechnungen verwendet).
- Bit 4 – S: Sign Bit
 - $S = N \text{ xor } V$
 - Wird für Vorzeichentests verwendet.

- Bit 3 – V: Two's Complement Overflow Flag
 - Zeigt Überlauf bei einer Zweier-Komplement Operation an.
- Bit 2 – N: Negative Flag
 - Zeigt negative Ergebnisse bei arithmetischen Operationen an.
- Bit 1 – Z: Zero Flag
 - Zeigt an, dass die Zahl 0 das Ergebnis der letzten Operation war.
- Bit 0 – C: Carry Flag
 - Zeigt an, dass es zu einem Überlauf gekommen ist.

AVR Assembler

General Purpose Registers

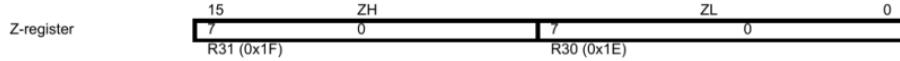
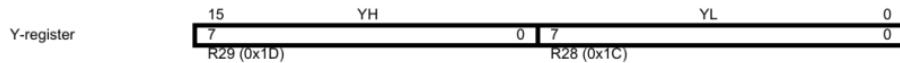
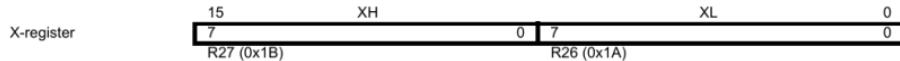


- 32x 8-bit Register zur freien Verwendung
- Mehrere Byte große Datentypen werden auf mehrere Register aufgeteilt.
- Die meisten Anweisungen können alle Register verwenden.
- Manche Anweisungen (z.B. ldi) können nur Register R16 – R31 verwenden.

7	0	Addr.
R0		0x00
R1		0x01
R2		0x02
...		
R13		0x0D
R14		0x0E
R15		0x0F
R16		0x10
R17		0x11
...		
R26		0x1A X-register Low Byte
R27		0x1B X-register High Byte
R28		0x1C Y-register Low Byte
R29		0x1D Y-register High Byte
R30		0x1E Z-register Low Byte
R31		0x1F Z-register High Byte

AVR Assembler

X-, Y- und Z-Register



- Register R26 – R31 werden als 16-bit Spezialregister (X-, Y-, Z-Register) für indirekte Speicheradressierung verwendet.

AVR Assembler

Stackpointer



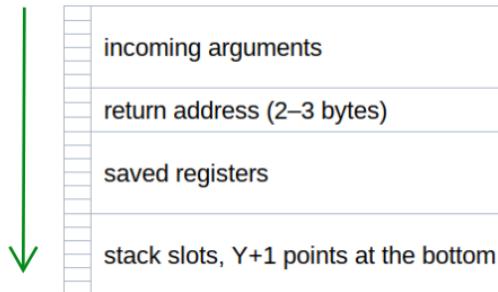
Bit	15	14	13	12	11	10	9	8	SPH	SPL
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8		
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0		
	7	6	5	4	3	2	1	0		
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W		
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W		
Initial Value	0	0	1	0	0	0	0	0		
	1	1	1	1	1	1	1	1		

- Ein eigenes 16-bit Stackpointer-Register (realisiert durch 2x 8-bit Register **SPH** und **SPL**) verwaltet den Stack.
- Wird i.d.R. automatisch durch die entsprechenden Anweisungen verwaltet (z.B. push, pop, call, ret, ...)

- Der avr-gcc Compiler hat eine **eigene Konvention für die Verwendung der General Purpose Register**.
- Diese Konvention muss beachtet werden, wenn man Code schreiben will, der mit vom avr-gcc erzeugten Code kompatibel ist.
- Ansonsten kann man seine eigene Konvention verwenden.
- So eine Konvention ist Teil des **Application Binary Interface (ABI)**.
 - Siehe <https://gcc.gnu.org/wiki/avr-gcc>

- R0: Scratch Register
 - Dieses Register kann für temporäre Zwischenergebnisse verwendet werden.
- R1: Zero Register
 - Dieses Register enthält immer die Zahl 0.
- R0, R18 – R27, R30, R31: Call-Saved Registers
 - Dürfen durch einen Funktionsaufruf nicht verändert werden.
- R1 – R17, R28, R29: Call-Used bzw. Call-Clobbered Registers
 - Können durch einen Funktionsaufruf verändert werden.
- Y-Register (R28:R29): Framepointer
 - Enthält den Framepointer
- ...

- Im ABI ist auch das [Frame Layout](#) festgelegt.
- Das Frame Layout bestimmt, wie ein zu einem Funktionsaufruf gehörender Frame am Stack aufgebaut ist.
 - Speichert Eingabeargumente, die Rücksprungadresse, zwischengespeicherte Register und lokale Variablen.



- Das ABI schreibt auch die sog. **Calling Convention** vor.
- Die Calling Convention regelt, wie Funktionsaufrufe stattfinden.
 - Wo werden die Eingabeparameter gespeichert.
 - Wie wird der Rückgabewert zurückgegeben.
- Für mehr Informationen über die Calling Convention siehe
https://gcc.gnu.org/wiki/avr-gcc#Calling_Convention

Die AVR Assembler Anweisungen können in fünf Kategorien eingeteilt werden:

- **Arithmetische und Logische Anweisungen**
 - Addition, Subtraktion, logisches UND, ...
- **Bedingte und unbedingte Sprunganweisungen**
 - Anweisungen, die den Programmzähler verändern.
- **Datentransferanweisungen**
 - Anweisungen, die Daten zwischen Register oder vom bzw. zum Daten- oder Programmspeicher transferieren.
- **Bit und Bit-Test Anweisungen**
 - Anweisungen, die einzelne Bits verändern bzw. testen.
- **Mikrocontroller Steueranweisungen**
 - Anweisungen, die spezielle Mikrocontroller-Funktionen steuern (z.B. Watchdog)

- Die meisten Anweisungen haben ein bis zwei Operanden.
 - Operanden können Register oder (eingeschränkt) Konstanten sein.
 - z.B.: add Rd,Rr; andi Rd,K; neg Rd
 - Rd, Rr ... Register
 - K ... Konstante
 - Befehle die mit *i* (steht für “Immediate”) enden, haben meist eine Konstante als zweiten Operanden.
- Ergebnis wird meist im ersten angegebenen Register gespeichert.
 - z.B.: add Rd,Rr: $Rd = Rd + Rr$

- Werte in Register sind nur **Bitfolgen ohne Datentyp**
 - Es gibt keine dazugehörende Typinformation!
- Beteiligte Datentypen sind in der Anweisung kodiert.
 - z.B.: mult Rd,Rr ... Multiplikation zweier vorzeichenloser Ganzzahlen
 - z.B.: mults Rd,Rr ... Multiplikation zweier vorzeichenbehafteter Ganzzahlen
 - z.B.: fmul Rd,Rr ... Multiplikation zweier vorzeichenloser Festkommazahlen
 - ...

- Der AVR Mikroprozessor hat keine Hardware-Gleitkommaeinheit!
 - Operationen mit Gleitkommazahlen in Software umgesetzt.
 - ⇒ Langsam und ineffizient.
 - ⇒ Gleitkommazahlen sollten daher möglichst vermieden werden!
- Es gibt auch keine Hardware-Divisionseinheit!
 - Division wird auch in Software gemacht!

- Mehrere Byte große Datentypen werden über mehrere Register verteilt
 - AVR verwendet Little Endian!
- Operationen werden byteweise angewendet.

Beispiel: UND-Verknüpfung der 16-bit Zahlen 0x1234 und 0x5678

```
1: ldi r16, 0x34 {Lade Low-Byte der ersten Zahl in Register r16}
2: ldi r17, 0x12 {Lade High-Byte der ersten Zahl in Register r17}
3: ldi r18, 0x78 {Lade Low-Byte der zweiten Zahl in Register r18}
4: ldi r19, 0x56 {Lade High-Byte der zweiten Zahl in Register r19}
5: and r16, r18 {Und-Verknüpfe Low-Bytes}
6: and r17, r19 {Und-Verknüpfe High-Bytes}
```

AND

1 0 1 0		1 1 0 0
0 1 1 1		0 0 1 1
0 0 1 0		0 0 0 0

$$\begin{array}{r} 1 \ 2 \ 3 \ 9 \\ 5 \ 8 \ 7 \ 8 \\ \hline 6 \ 9 \ 1 \ 2 \end{array}$$

$$\begin{array}{r} 1 \ 2 \ 3 \ 9 \\ 5 \ 8 \ 7 \ 8 \\ \hline 6 \ 9 \ 1 \ 2 \end{array}$$

Diagram illustrating the addition of two 4-byte values:

- The first byte (least significant) is $1 + 8 = 9$.
- The second byte is $2 + 5 = 7$, with a carry of 1.
- The third byte is $3 + 7 + 1 = 11$, resulting in $1 + 1 = 2$ and a carry of 1.
- The fourth byte (most significant) is $9 + 8 + 1 = 18$, resulting in $8 + 1 = 9$ and a carry of 1.

Labels below the result indicate the bytes: $ad\ c$ and add .

- Für die Berücksichtigung von Überläufen gibt es spezielle Anweisungen
 - Überläufe werden im Status-Register SREG im Bit C gespeichert.
- z.B.: Addition:
 - add Rd, Rr: Addition ohne Überlaufberücksichtigung
 - adc Rd, Rr: Addition mit Überlaufberücksichtigung ($Rd = Rd + Rr + C$)

Beispiel: Addition der 16-bit Zahlen 0x1234 und 0x5678

```
1: ldi ... {Siehe Beispiel Und-Verknüpfung}  
→ 2: add r16, r18 {Addiere Low-Bytes}  
→ 3: adc r17, r19 {Addiere High-Bytes und Überlauf-Bit}
```

- Sprünge verändern den Programmzähler.
- bedingte oder unbedingte Sprünge.
 - Es wird abhängig von einer Bedingung oder immer gesprungen.
- absolute oder relative Sprünge:
 - Sprungziel ist eine absolute Adresse.
 - Sprungziel ist ein Offset relativ zur aktuellen Position.
- near oder far Jumps:
 - Unterscheidung, wie "weit" gesprungen werden kann.
 - Unterscheiden sich auch in Größe der Anweisung.
- direkte oder indirekte Sprünge:
 - Bei direkten Sprüngen ist das Sprungziel direkt im Opcode kodiert.
 - Bei indirekten Sprüngen steht das Sprungziel in einem Register.

- Für Funktionen bzw. Subroutinen gibt es spezielle Sprungbefehle.
 - call: Sichere vor dem Sprung den alten Programmzählerwert auf den Stack.
 - ret: Lese Sprungadresse vom Stack und springe dort hin.

Bedingte Sprünge werden in zwei Schritten ausgeführt:

① Auswertung der Bedingung.

- Ergebnis der Auswertung wird im Statusregister gespeichert.
- z.B.: `cp r3, r4`: Vergleicht die zwei Zahlen in den Registern r3 und r4.

② Durchführung des bedingten Sprung.

- Abhängig davon, ob bestimmte Bits im Statusregister 0 oder 1 sind, wird der Sprung durchgeführt.
- z.B.: `brcc k`: Springt um k Bytes, wenn das Carry Flag (Überlauf) im Statusregister gesetzt ist (d.h. wenn $r3 < r4$).
- z.B.: `brcc k`: Springt um k Bytes, wenn das Carry Flag (Überlauf) im Statusregister nicht gesetzt ist (d.h. wenn $r3 \geq r4$).

- Da man beim Programmieren noch nicht die Adressen kennt, werden symbolische Labels für Sprünge verwendet.
- Linker ersetzt Sprunglabels durch konkrete Adressen.

Beispiel: Bedingter Sprung mit Sprunglabel

```
1: ...
2: cp r26, r27 {Vergleiche r26 und r27}
3: breq mylabel {Springe wenn r26 und r27 gleich ist (Bit Z in SREG ist gesetzt)}
4: ...
5: mylabel: {Springe an diese Stelle}
6: ...
```

- **RJMP:** Relative Jump
 - Verwendet 16-Bit Opcode
 - Sprungoffset wird mit 12-Bits kodiert (vorzeichenbehaftet).
 - Maximale Sprungweite (in Words): $PC - 2K + 1$ bzw. $PC + 2K$
 - ⇒ Wenn Programmspeicher größer als 8KB ist, dann kann nicht der gesamte Speicher addressiert werden.
 - ⇒ Ein Sprung mit Reichweiteneinschränkung nennt man [Near Jump](#).
- **JMP:** Jump
 - Verwendet 32-Bit Opcode
 - Absolute Sprungadresse wird 22-Bits kodiert.
 - ⇒ Kann gesamten Programmspeicher addressieren.
 - ⇒ Ein Sprung ohne Einschränkungen (bzw. mit weniger Einschränkungen) nennt man [Far Jump](#).

Inhaltsverzeichnis



Einführung

AVR Assembler

“Strukturierte Programmierung” in Assembler

Q&A Session

- Assembler kennt keine Verzweigungen und Schleifen.
- Stattdessen werden Sprünge verwendet, um Verzweigungen und Schleifen abzubilden

Verzweigung in einer Hochsprache

```
1: if Bedingung1 then
2:   Befehl1
3: else if Bedingung2 then
4:   Befehl2
5: else
6:   Befehl3
7: end if
```

Verzweigung in Assembler

```
1: cp ... {Überprüfe Bedingung1}
2: brxx label1 {Springe, wenn Bedingung1 nicht erfüllt}
→ 3: Befehl1
→ 4: jmp end {Unbedingter Sprung zum Ende der Verzweigung}
5: label1: {Sprunglabel für else-if-Zweig}
6: cp ... {Überprüfe Bedingung2}
7: brxx label2 {Springe, wenn Bedingung2 nicht erfüllt}
→ 8: Befehl2
9: jmp end {Unbedingter Sprung zum Ende der Verzweigung}
10: label2: {Sprunglabel für else-Zweig}
11: Befehl3
→ 12: end: {Sprunglabel am Ende der Verzweigung}
13: ...
```

- Im folgenden wird nur die While-Schleife gezeigt.
- Um die anderen Schleifenarten in Assembler umsetzen zu können, müssen diese zuvor in eine While-Schleife umgewandelt werden.

While-Schleife in einer Hochsprache

```
1: while Bedingung1 do  
2:   Befehl1  
3: end while
```

While-Schleife in Assembler

- 1: loop: {Sprunglabel am Beginn der Schleife}
- 2: cp ... {Überprüfe Bedingung1}
- 3: brxx end {Springe, wenn Bedingung1 **nicht** erfüllt}
- 4: Befehl1
- 5: jmp loop {Unbedingter Sprung zum Kopf der Schleife}
- 6: end: {Sprunglabel am Ende der Schleife}
- 7: ...

Inhaltsverzeichnis



Einführung

AVR Assembler

"Strukturierte Programmierung" in Assembler

Q&A Session



Q&A Session

6 ASSEMBLERPROGRAMMIERUNG

▼ VR Assembler & Inline-Assembler



Was ist AVR Assembler?

AVR Assembler ist die **Maschinensprache** für Mikrocontroller der AVR-Familie (z. B. ATmega32u4, wie auf dem EduArdu-Board).

Assembler ermöglicht **direkten Zugriff auf Register, Ports und Hardware**, ohne Compilerabstraktionen.



Grundstruktur eines Assemblerprogramms

```
ldi r16, 0xFF ; Lade 0xFF in Register 16  
out DDRB, r16 ; Setze alle Pins von Port B als Ausgang
```



Wichtige Assemblerbefehle (AVR)

Befehl	Bedeutung	Beispiel
ldi	Lade unmittelbaren Wert in Register	ldi r16, 0xFF
out	Schreibe Wert in I/O-Register	out PORTB, r16
in	Lese Wert aus I/O-Register	in r16, PINB
sbi	Setze Bit in I/O-Register	sbi PORTB, 3
cbi	Lösche Bit in I/O-Register	cbi PORTB, 3
rjmp	Unbedingter Sprung	rjmp main
cp	Vergleiche zwei Register	cp r16, r17
breq	Springe, wenn gleich (Zero Flag)	breq gleich



Registerübersicht

- **Allgemeine Register:** r0 bis r31
 - r16–r31: für Konstanten mit ldi
- **Spezialregister:**
 - r26–r31: **Zeigerregister** (X, Y, Z)
 - X = r27:r26, Y = r29:r28, Z = r31:r30



Inline-Assembler in C (avr-gcc)

Syntax:

```
asm volatile ("Befehl" : Ausgang : Eingang : veränderte Register);
```

Beispiel:

```
uint8_t x = 5;
asm volatile (
    "inc %[val]"      "\n\t"
    : [val] "+r" (x)  // Ausgangsoperanden (lesend + schreibend)
    :                 // keine Eingänge
    :                 // keine zerstörten Register
);
```

 Danach ist x = 6;



Beispiele aus der Praxis

◆ LED blitzen (PORTB.0):

```
ldi r16, (1<<PB0)
out DDRB, r16    ; PB0 als Ausgang
loop:
    out PORTB, r16 ; LED an
    rcall wait
    out PORTB, r1  ; LED aus
    rcall wait
    rjmp loop
```



Assembler vs. C: Wann einsetzen?

Vorteil (Assembler)	Nachteil
Extrem effizient und schnell	Weniger portabel
Direktzugriff auf Hardware	Schwer wartbar
Kontrolle über Taktzyklen	Fehleranfällig

Empfehlung: Nur für zeitkritische / hardwarenahe Aufgaben!



Typische Anwendungen von Inline-Assembler

- Bitmanipulation auf Ports
- Timing-kritische ISR-Optimierung
- Speicherzugriff ohne Overhead

- SPI/UART-Bit-Banging
 - Delay-Schleifen
-

Quellen

1.  [AVR Assembler "Beginner" PDF](#)
 2.  [avr-libc Inline-Assembler Guide](#)
 3.  [RN-Wissen Inline-Assembler](#)
 4.  [AVR-ASM-Tutorial \(PDF\)](#)
-

Zusammenfassung

Thema	Wichtige Punkte
AVR Assembler	Direkt, effizient, hardwarenah
Befehle	ldi, out, in, sbi, rjmp, cp...
Inline-Assembler	Mit asm volatile in C eingebettet
Anwendungen	Timing, Registerzugriffe, Bitsteuerung
Risiko	Wenig Fehlertoleranz, schwer lesbar

Fragen

1. Erklären Sie die Unterschiede zwischen einer RISC- und einer CISC-Architektur.

Antwort:

RISC (Reduced Instruction Set Computer) verwendet einen reduzierten, einfachen Befehlssatz mit dem Ziel, jeden Befehl in einem Taktzyklus auszuführen – dadurch sind RISC-Prozessoren oft schneller und energieeffizienter.

CISC (Complex Instruction Set Computer) hingegen besitzt einen umfangreichen, komplexen Befehlssatz, bei dem einzelne Befehle mehrere Taktzyklen dauern und direkt komplexe Operationen ausführen können.

RISC setzt stark auf viele Register und einfache Lade-/Speicherbefehle, während CISC auch direkte Speicherzugriffe innerhalb komplexer Befehle erlaubt.

Moderne CISC-Prozessoren (z. B. x86) nutzen intern oft RISC-Kerne mit einer CISC-Dekodiereinheit.

2. Erklären Sie, welche zwei Paradigmen zur Implementierung von Echtzeitsystemen in der Vorlesung behandelt wurden.

Zeitgesteuerte Systeme arbeiten nach einem festen Zeitplan, in dem jede Aufgabe einen definierten Zeitslot erhält. Sie sind gut planbar und vermeiden Überlastung, reagieren aber nicht exakt zum Zeitpunkt eines Ereignisses.

Ereignisgesteuerte Systeme reagieren sofort auf eintretende Ereignisse, meist über Interrupts. Sie sind sehr reaktionsschnell und deterministisch, aber schwieriger planbar und anfällig für Überlastung bei vielen gleichzeitigen Ereignissen.

3. Inwiefern unterscheidet sich die Softwareentwicklung für eingebettete Systeme von der „normalen“ Softwareentwicklung?

- Bei sicherheitskritischen Systemen kommen meist konservative Vorgehensmodelle zum Einsatz."
- Eine Codezeile ist i.d.R. billiger als bei der ‚normalen‘ Softwareentwicklung.
- Beim Bauen von Software für Mikrocontroller werden dieselben Optimierungsrichtlinien wie beim Bauen ‚normaler‘ Software angewendet.
- Features von C++ wie Polymorphismus, dynamische Bindung oder Vererbung können ohne Bedenken exzessiv angewendet werden.

4. Erklären Sie, was das 'Nyquist-Shannon-Theorem' besagt.

Das Nyquist-Shannon-Theorem besagt, dass ein Signal exakt rekonstruiert werden kann, wenn es mit einer Frequenz abgetastet wird, die mehr als doppelt so hoch ist wie die höchste vorkommende Frequenz im Signal ($f_a > 2 \cdot f_{\max}$).

Ist die Abtastfrequenz zu niedrig, entstehen Fehler im Signal (Aliasing).

Deshalb muss man beim Messen oder Digitalisieren auf eine ausreichend hohe Abtastfrequenz achten.

5. Welche Aussagen treffen auf die Tätigkeit des „Messens“ zu?

- Zufällige Fehler können am besten durch eine sorgfältige Untersuchung des Messaufbaus kompensiert werden.
- Jede Messung ist fehlerbehaftet.
- Unter Messen versteht man die Gewinnung von Informationen über eine physikalische Größe.

- Systematische Messfehler können am besten durch wiederholte Messungen und statistische Analysen kompensiert werden.

6. Erklären Sie die Besonderheiten von Software, welche speziell für eingebettete Systeme entwickelt wird.

- Ressourcensparsamkeit: Die Software muss mit sehr wenig Speicher, Rechenleistung und Energie auskommen, da Mikrocontroller oft stark begrenzt sind.
- Echtzeitfähigkeit: Viele eingebettete Systeme müssen innerhalb fester Zeitvorgaben reagieren, sonst droht ein Funktionsausfall.
- Zuverlässigkeit und Sicherheit: Fehler können gefährlich sein, deshalb wird meist konservativ entwickelt, mit strengen Tests und einfachen, robusten Code-Strukturen.

7. Welche Aussagen treffen auf die 'Pulsweitenmodulation (PWM)' zu?

- Ein PWM-Signal ist ein digitales Signal.
- Ein PWM-Signal ist ein analoges Signal.
- Ein PWM-Signal kann aufgrund des Trägheitseffekts die gleiche Wirkung wie ein analoges Signal erwirken.
- PWM-Signale gelten als veraltet und kommen kaum mehr zum Einsatz.

8. Welche Aussagen treffen auf einen 'Analog-Digital-Wandler' zu?

- Es gibt Analog-Digital-Wandler, die jedes wertkontinuierliche Signal ohne Genauigkeitsverlust in einen numerischen Wert umwandeln können.
- Quantisierungsfehler treten ausnahmslos bei jeder einzelnen Umwandlung auf.
- Das Ergebnis einer Umwandlung ist ein numerischer Wert, der das Verhältnis des Eingangssignals zu einem Referenzsignal angibt.
- Die Auflösung eines Analog-Digital-Wandlers gibt an, wie nahe im Durchschnitt ein Umwandlungsergebnis an den echten Signalwert herankommt.

9. Welche Aussagen treffen auf einen 'Open-Loop Controller' zu?

- Ein Open-Loop Controller kann Störungen nicht erkennen und daher auch nicht beheben.
- Ein Open-Loop Controller ist eine Vorrichtung zur Steuerung eines Systems.
- Ein Open-Loop Controller kann Störungen erkennen und beheben.
- Ein Open-Loop Controller dient der Messung einer physikalischen Größe nach dem Open-Loop Prinzip.

10. Welche Aussagen treffen auf 'Plattformsoftware' zu?

- Die Plattformsoftware implementiert die eigentliche Anwendung.
- Mikrocontrollerprogramme werden zur Unterscheidung von 'normalen' Programmen als Plattformsoftware bezeichnet.
- Die Plattformsoftware kapselt hardwarespezifische Details.
- Programme, die hauptsächlich auf Ereignissesteuerung setzen, werden Plattformsoftware genannt.

10. Erklären Sie den Unterschied zwischen 'Polling' und 'Ereignis-' bzw. 'Interrupt-Steuerung'

Beim Polling fragt der Mikrocontroller in regelmäßigen Abständen aktiv ab, ob ein Ereignis eingetreten ist. Bei der Interrupt-Steuerung reagiert der Mikrocontroller automatisch, sobald ein Ereignis auftritt, ohne ständig abzufragen.

Beispiel: Wenn man auf einen Freund wartet und jede Minute zur Tür geht, ist das Polling – wenn man wartet, bis es klingelt, ist das ein Interrupt.

Polling ist weniger effizient, da es ständig prüft, während Interrupts ressourcenschonender und schneller reagieren.

10. Welche Aussagen treffen auf die 'Assemblersprache' zu?

- Die einzelnen Befehle der Assemblersprache nennt man Opcode.
- Die Assemblersprache ist abhängig von der jeweiligen Prozessorarchitektur.
- Die Assemblersprache ist die vom Menschen lesbare Version der Maschinensprache.
- Die Assemblersprache ist eine Hochsprache, die die strukturierte Programmierung umsetzt.

10. Erklären Sie, wieso bei eingebetteten Systemen bevorzugt Bus-Systeme zum Einsatz kommen.

Bus-Systeme ermöglichen es mehreren Komponenten, sich eine gemeinsame Datenleitung zu teilen, was Platz und Kosten spart.

Gerade in eingebetteten Systemen sind diese Faktoren entscheidend, da dort oft wenig Raum und begrenzte Ressourcen vorhanden sind.

Zudem lassen sich über Bus-Systeme viele Peripheriegeräte einfach und flexibel anbinden.
Typische Beispiele sind I²C, SPI oder CAN-Bus.

10. Was ist der Unterschied zwischen einem 'Mikrocontroller' und einem 'Mikroprozessor'?

- Ein Mikrocontroller enthält schon die meisten für den Betrieb eines Computers notwendigen Komponenten, ein Mikroprozessor nicht.
- Prozessoren mit wenig Betriebsressourcen nennt man Mikrocontroller, Prozessoren mit vielen nennt man Mikroprozessoren.
- Beides sind nur unterschiedliche Bezeichnungen, die aber dieselbe Bedeutung haben.
- Ein Prozessor, der in einem technischen System eingebunden ist, nennt man Mikrocontroller.

10. Welche Aussagen treffen auf 'echtzeitfähige Systeme' zu?

- Echtzeitfähigkeit ist eine reine Eigenschaft der Hardware.
- Praktisch jedes System mit einem Prozessor, der nur schnell genug ist, ist weich-echtzeitfähig.
- oder ?? Praktisch jedes System mit einem Prozessor, der nur schnell genug ist, ist hart-echtzeitfähig.
- Echtzeitfähigkeit ist eine essentielle Voraussetzung von sicherheitskritischen Systemen.

10. Was ist ein 'Logikgatter'?

- Logikgatter sind zumeist elektrische Schaltungen, die boolesche Funktionen implementiert.
- Logikgatter bilden den kleinsten Baustein eines Prozessors.
- Logikgatter übersetzen Maschinenbefehle in interne Mikrobefehle.
- Logikgatter bilden die Schnittstelle zwischen Hardware und Software.

10. Welche Aussagen treffen auf 'Interrupts' zu?

- Ein Interrupt ist eine kurzfristige Unterbrechung des normalen Programmflusses.
- Interrupts sind für den normalen Betrieb eines Prozessors essentiell.
- Interrupts benötigen i.d.R. Unterstützung durch die Prozessorhardware.
- Interrupts sind nur eine andere Bezeichnung für Exceptions.

10. Beschreiben Sie die fünf Phasen der Befehlsausführung in einer CPU.

Fetch: Ladet Befehl aus Speicher in Befehlsregister

Decode: Befehlsdecoder setzt Steuerleitungen.

Load (Optional): Lade Operanden aus Speicher.

Execute: Führe Befehl aus.

Store (Optional): Speichere Ergebnis zurück in den Speicher.

10. Erklären Sie, was ein 'PID-Regler' ist und wofür dieser verwendet wird.

Ein PID-Regler ist ein Regelalgorithmus, der aus drei Anteilen besteht: Proportional (P), Integral (I) und Differential (D).

Er vergleicht den Ist-Wert mit dem Soll-Wert und berechnet daraus ein Steuersignal, um die Abweichung zu minimieren.

PID-Regler werden häufig in technischen Systemen eingesetzt, z. B. zur Temperatur-, Drehzahl- oder Positionsregelung.

Durch die Kombination der drei Anteile reagiert der Regler sowohl auf aktuelle, vergangene als auch auf zukünftige Abweichungen.

10. Was versteht man unter 'bidirektional'?

- Es kann gleichzeitig gesendet und empfangen werden.
- Es kann in beide Richtungen kommuniziert werden.
- Es kann nur in eine Richtung kommuniziert werden.
- Es kann nicht gleichzeitig gesendet und empfangen werden.

10. Welche Aussagen treffen auf die Tätigkeit des 'Steuerns' zu?

- Die Steuerung eines technischen Systems durch einen Mikrocontroller wird immer als digitale Steuerung umgesetzt.
- Externe Störeinflüsse können in allen Fällen erkannt und kompensiert werden.
- Unter Steuern versteht man die gezielte Beeinflussung des Verhaltens von technischen Systemen.
- Steuerungsabweichungen aufgrund von äußeren Störfaktoren können nicht erkannt werden.

10. Was ist der Unterschied zwischen 'Punkt-zu-Punkt Verbindungen' und 'Bussystemen'? Was sind die jeweiligen Vor- und Nachteile? Was kommt in eingebetteten Systemen bevorzugt zum Einsatz und wieso?

Bei Punkt-zu-Punkt-Verbindungen sind zwei Komponenten direkt miteinander verbunden – das ist schnell und störsicher, aber aufwendig bei vielen Teilnehmern.

Bussysteme ermöglichen es mehreren Komponenten, eine gemeinsame Leitung zu nutzen, was Platz und Kosten spart, aber durch Kollisionen oder Adressierung komplexer ist.

In eingebetteten Systemen werden meist Bussysteme eingesetzt, da sie kompakter und günstiger sind und den Ressourcenanforderungen solcher Systeme besser entsprechen.

oder

Punkt zu Punkt → exklusive Direktverbindung.

Vorteile: Echtzeitfähig, keine Störungen durch andere, effizient.

Nachteile: Viel Verkabelung, schlechte Ressourcennutzung.

Bus System → Der Kommunikationskanal kann von mehreren Komponenten genutzt werden.

Vorteile: Verkabelung kann von vielen genutzt werden, gemeinsamer Pin am Microcontroller.

Nachteile: Nicht unbedingt echtzeitfähig, Störung durch andere möglich, nicht immer effiziente Übertragungsleistung.

10. Erklären Sie den Unterschied zwischen 'harter' und 'weicher' Echtzeit.

Harte Echtzeit: Garantierte Reaktionszeit wird niemals überschritten. Sicherheitskritische Systeme verwenden meist harte Echtzeit.

Weiche Echtzeit: Die Reaktionszeit wird nur im Durchschnitt erreicht. Nicht sicherheitskritische Systeme benötigen oft nicht mehr als weiche Echtzeit.

10. Welche Aussagen treffen auf 'batteriebetriebene Systeme' zu?

- Software hat kaum eine Auswirkung auf den Energieverbrauch des Systems.
- Mikrocontroller können schlafen gelegt werden, d. h. sie stellen die Programmausführung vorübergehend ein, um Energie zu sparen.
- Software kann eine enorme Auswirkung auf den Energieverbrauch haben.
- Die Stromverbräuche der einzelnen Schlafmoduse unterscheiden sich nur minimal.

10. Was ist ein 'intelligenter Sensor'?

- Ein Sensor, der neben der eigentlichen Messung auch die Signalaufbereitung und -auswertung übernimmt.
- Ein Sensor, der das für die jeweilige Situation am besten passende Messprinzip selbstständig auswählt.
- Ein Sensor, der sich selbstständig schlafen legt, um Strom zu sparen.
- Ein Sensor, der selbstständig Messwerte an den Mikrocontroller meldet.