Was ist ein eingebettetes System?

Geschichte:

1927: ABS in Flugzeugen, Mechanisch (später auch für Autos)

1936: Konrad Zuse, Rechner (mit Relais)

1940: Problem der Raketensteuerung (Timersteuerung ungenau, Funk (durch Röhren) hat versagt)

1947: Entdeckung des Transistors

1950: Verbauung von Röhren und Transistoren in Rechner (erster: ENIAC)
Intercontinental-Raketen (minute man project), Computer so klein, dass er in Rakete passt (mit Transistoren) Funkschatten

1960: Apollo-Raumschiff, integrierte Schaltungen (IC) zu AGC Funkschatten Verzögerung

später: AGC in Kampfflugzeugen (Schussfestigkeit)

eingebettetes System:

wesentlicher Faktor: technisches System $\xrightarrow{\text{Sensoren}}$ / $\xleftarrow{\text{Aktoren}}$ Computer

zusammenfassend: Computer, der steuert und regelt und in einen technischen Kontext eingebunden ist, den der Benutzer gar nicht sieht (ist nicht zugänglich)

Techniken: Hardware Abstraktion
Prioritäten gesteuertes Betriebssystem } gehen auf AGC zurück

VHDL

Einführung:

IC wird benötigt → Schaltung auf Rechner simulieren

Hardwarebeschreibungssprache: Art Programmiersprache, mit der man Hardware bauen kann (VHDL, ...)

1980: Standardisierung von VHDL vom amerikanischen Verteidigungsministerium

Idee: Verhaltensbeschreibung der Schaltung, mit der man versteht was sie macht → Zusammensetzung → Simulierung

textuelle Beschreibung (nicht graphisch)

Ziel: Verhaltensbeschreibung in VHDL bauen

synthesefähige Beschreibungen, die man auf FPGA ausprobieren kann

Motivation:

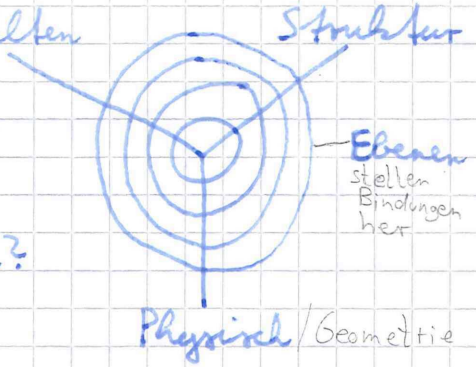
GdRa: Wahrheitstabelle → Funktion der Schaltalgebra → Umsetzung mit logischgattern (Basis: NAND, NOR)

↳ Verhaltensbeschreibung

↳ Strukturbeschreibung · graphisch skaliert nicht

Beschreibungs-ebene	Verhaltens-beschreibung	Struktur-beschreibung	Physische Realisierung	Geometrie
Transistorebene	DGL	Transistor-Schaltplan	Layout	Transistorlayout
logische Ebene	Boolsche Gleichung	Gatter-Schaltung	IC	Zellen
Registertransfer Ebene (RTL)	Datenfluss	Operationen, Register	Board	Chips
↓ Algorithmische Ebene	Algorithmen	Speicher, Prozessor	System	Boards

→ Y-Diagramm:



- Welche Entwurfsschritte muss man durchführen?
- Wie kann man diese automatisieren?

- unterschiedliche Sichten beschreiben und darstellen

VHDL: Very High Speed Hardware Description Language

Beispiel:

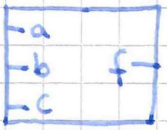
```
signal a,b,c;
f <= ((not a) and b and c) or
     ( a and (not b) and c) or
     ( a and b and c);
```

} Verhaltensbeschreibung, in VHDL: Datenflussbeschreibung

Beispiel:

```
entity DECO is
    port (a,b,c : in bit;
          f     : out bit);
end DECO;
```

} Schnittstelle



→ Beginn der Implementierung

```
architecture BEHVIOR of DECO is
begin
    signal x,y,z;   f <= ...siehe oben
end BEHAVIOR
```

```
architecture STRUCTURE of DEMO is
    component AND
      port (a,b,c : in bit;
            f : out bit);
    end component
    component OR
      port (a,b,c : in bit;
            f : out bit);
    end component;
```

Schnittstelle zur Bibliothek

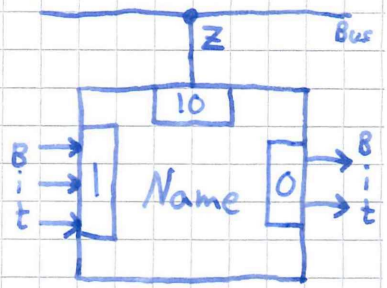
```
begin
    signal ia,ib:bit;  signal f01, f02, f03 : bit;
    I1: INV  port map (a,ia)
    I2: INV  port map (b,ib)
    A1: AND  port map (ia,b,c, f01);
    A2: AND  port map (a,ib,c, f02);
    A3: AND  port map (a,b,c, f03);
    O: OR  port map (f01,f02,f03);
end
```

Struktur-beschreibung einer Architektur

<u>Komponenten:</u>

graphisch: Blöcke (diese werden zusammengefügt)

VHDL: Module / entities



Syntax:
```
entity Name is
    port (I1, I2, I3 : in bit;
          O1, O2 : out bit
          IO1 : in out bit);
end;
```

Schnittstelle der Komponente

Unter-Komponenten aus Bibliothek: LIBRARY IEEE;
↳ vor „Syntax"
USE ieee.std_logic_1164.all

Inhalt der Komponente spezifizieren: architecture behavior of Name is
↳ nach „Syntax"
Begin

... (Verhalten: if|... then... /

End;

architecture structure of Name is

Begin

... (Struktur:)

End;

Idee hinter verschiedenen Architekturen: Entity (Device under Test)

- Verhalten → Spezifikation
- Struktur → Implementierung

Entity (Test Bench): Verhalten

Syntax: configuration name of Entity is

... (Spezifikation: wählt Verhalten oder Struktur)

end configuration;

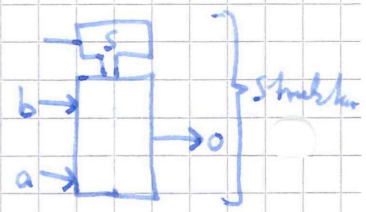
Verhalten

algorithmische Ebene:

```
MX: process (a,b,s)
  Begin
    if s='1' then
      o<=a;
    else
      o<=b;
    endif;
  end process;
```

Sensitivitätsliste

Verhalten
- for
- while



In einem Prozess spezifiziert man beliebige Algorithmen

Registertransfer Ebene:

```
architecture ...
  begin
    o<=a when s='1' else b;
  end...
```

Datenflussbeschreibung

Struktur

Elemente aus Bibliotheken und spezifizieren wie wir die Komponente aus der

Bibliothek verstehen

Beispiel: component NAND
port (a,b : in bit;
c : out bit);
end component;

VHDL Compiler prüft diese Spezifikation gegen die Spezifikation in der Bibliothek
↳ Sicherstellung das Benutzung der Komponente richtig verstanden

Aufruf der Komponente

Beispiel: nand1: NAND port map (t, s, o) Reihenfolge abc muss nicht eingehalten werden
(a => t, b => s, c => o)

Datentypen

numerische Datentypen: Integer und Real
Boolean (true / false)
Character
String

Time (fps oder s,h)
Bit ('1'/'0')
Bit_Vector ("10010")

→ Diese Standardtypen reichen nicht aus um digitale Schaltungen zu beschreiben: sehr gut geeignet für Simulation oder Verhaltensbeschreibung oder Dokumentation
nicht geeignet für Synthese oder detaillierte Simulation

IEEE: std_logic (U → noch nicht gesetzt, X → noch nicht bekannt, 0, 1, Z → auf hoch-ohmig geschaltet, L → weak low, H → weak high, W → weak unknown, - → don't care)

std_ulogic

std_logic_vector

eigene Datentypen: Bsp: typ int0 is range 0 to 100;
typ int1 is range -100 to 100;
} erlaubt Laufzeitüberprüfung ↳ Compiler

Bsp: subtype int5 is Integer range -5 to 5

Bsp: type state is (locked, unlocked, ...)
type BUS is array (0 to 3) of bit;

→

eigene Strukturen: Bsp: type INTERFACE is record

```
    bus: BUS;
    ctrl: BIT;
    data: BUS;
end record;
```

Simulationssemantik

Verzögerungszeit: Bsp: x <= Not In after 10ns
($\hat{=}$ R·C)

↳ lang: x <= Reject 10ns Inertial not In after 10ns

→ inertiales-Modell

Bsp: x <= Transport not In after 10ns

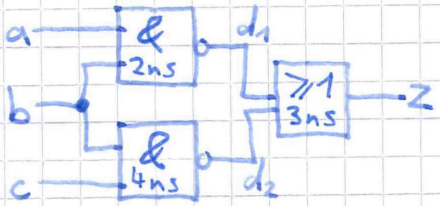
↳ lang: x <= Reject 0ns Inertial not In after 10ns

→ transport-Modell

→ after-Konstrukt darf nicht verwendet werden, wenn eine Schaltung synthetisiert werden soll

after-Konstrukt nur für Modellierung und Beschreibung einer Schaltung in ihrem Verhalten

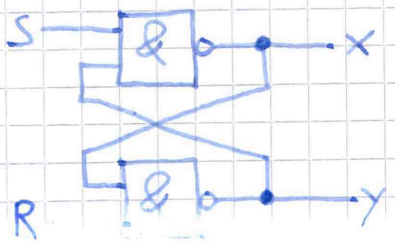
Transaktions-/Ereignisliste: Bsp:
↳ Signalwechsel = Transaktion



t[ns]	a	b	c	d_1	d_2	z
0	0	0	0	1	1	1
10	0	1	0	1	1	1
12	0	1	0	1	1	1
14	0	1	0	1	1	1
17	0	1	0	1	1	1
20	1	1	0	1	1	1
22	1	1	0	0	1	1
24	1	1	0	0	1	1
27	1	1	0	0	1	1

→ kombinatorische Schaltung simulieren
(Signalbegriff, Ereignisliste)

Bsp:



t[ns]	S	R	x	y
0	1	1	1	0
20	↓	1	1	0
20+Δ	0	1	↓	0
20+Δ	0	1	0	↑
20+Δ	0	1	0	1

→ sequentielle rückgekoppelte Schaltungen simulieren (Δ: Delta-Zyklus)

Zusammenfassung

Grundkomponente: Entity

- Schnittstelle: Port (ist immer gleich)
- Implementierung: Architectures
  - textuelle Beschreibung einer Struktur
  - streng sequentielles Verhalten
    ↳ Process mit Sensitivitätsliste, läuft sequentiell in einem Δ-Zyklus ab → 0ns
    wait/wait until hält Process an
  - sequentielles Verhalten
    ↳ Dataflow, Signalzuweisungen, läuft nach Einträgen der Transaktionsliste ab

Spezifikation: besteht aus mehreren Entities

jede Entity kann in einem Package gespeichert werden

Zugriff auf Package: Library Zugriff auf Entity: use

Auswahl einer der Architectures: configuration

VHDL ist ein mächtiges Werkzeug

- Spezifikation von Verhalten und Struktur (auch mischbar)
- Abbildung des Entwurfsprozesses

Architektur der Komponenten

eingebettete Hardware/Software Systeme

mechanisch → elektronisch: Verschleiß

analoge Elektronik → Mikrocontroller: Hitzeempfindlich, Rauschempfindlich, Veränderbarkeit → Software

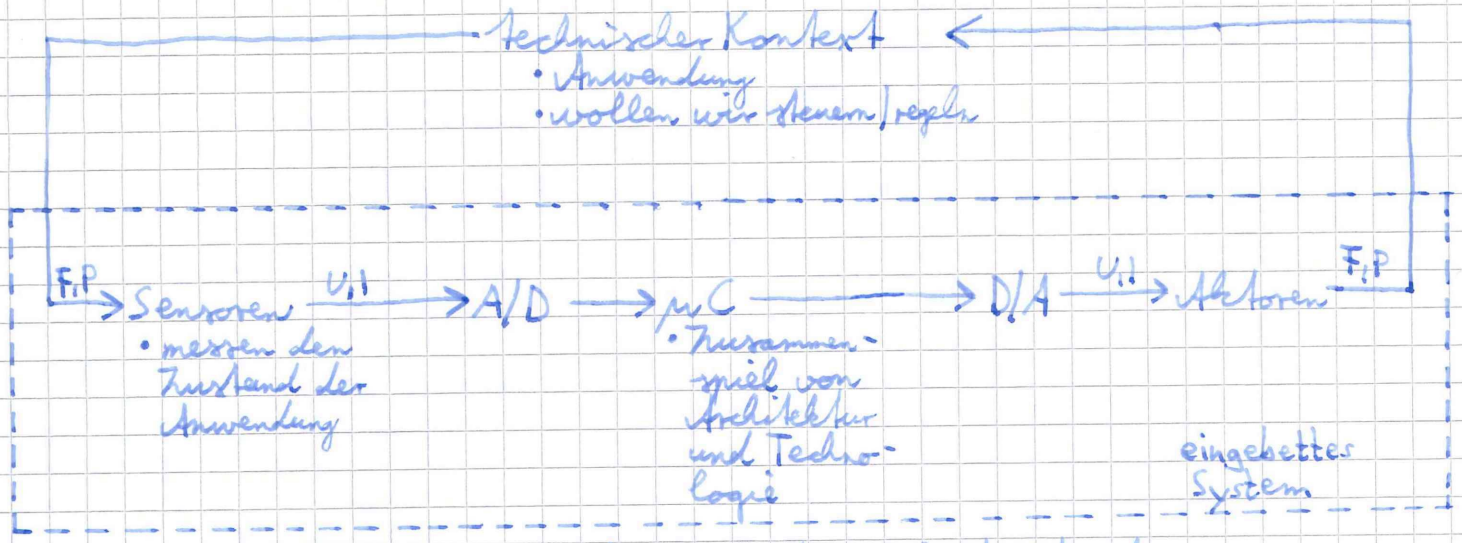
wichtige Punkte: Gewicht, Energie (Batterie), Kosten, Heizung

häufige Lösung: Kombination eines günstigen Mikrocontrollers mit Hardware, die zu erfüllende Aufgabe besonders schnell kann

Kombination aus Hardware und Software bringt Mehrwert

- günstiges System, dass größeres System steuert und regelt

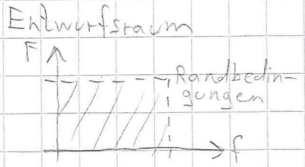
Was ist ein eingebettes System?



spezielle Anforderungen: zeitliche Fristen → Echtzeitsystem

Kosten

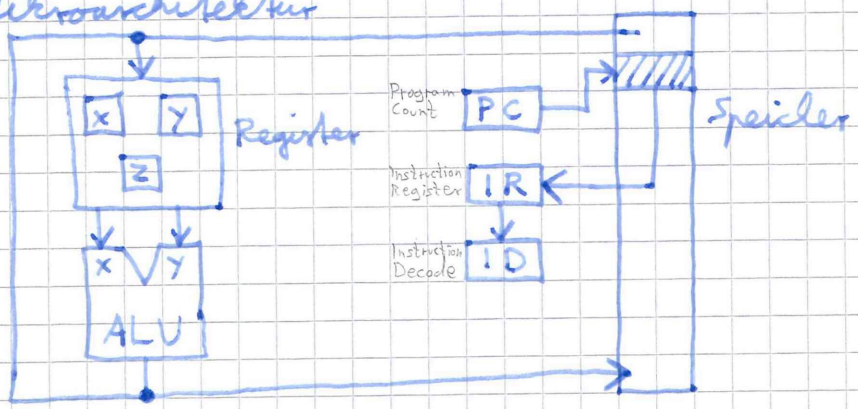
Fläche vs. Geschwindigkeit ($f = \frac{1}{T}$) (je größer desto schlechter)



↳ Abtastrate muss Nyquist-Kriterium erfüllen

Architektur: Instruction Set Processor (ISP)

Struktur: Mikroarchitektur



Verhalten: Architektur

3-Adress-Architektur add, load, store

Befehlszyklus: 1) Befehl holen (IF) Instruction Fetch

2) Befehl decodieren (ID) Instruction Decode

3) Operanden holen (OF) Operand Fetch

4) Ausführen (EX) Execute

5) Speicher schreiben (MEM) Memory

6) Ergebnis speichern (OS) Operand Save

Ergebnis: Ressourcen sehr gut wiederverwendbar

zeitlich gestreckte Verwendung der Ressource

↳ dauert bis wiederverwendbar

Komplexe Berechnung: Unterteilung in mehrere 3-Adress-Operationen

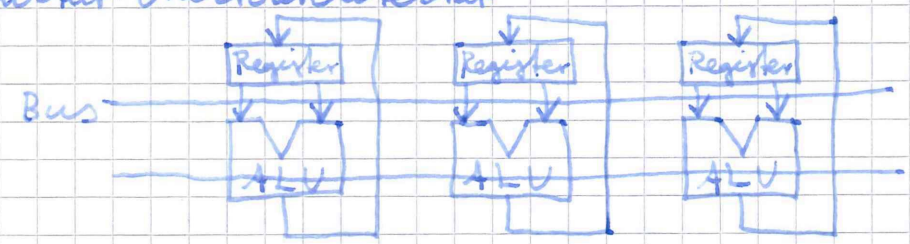
↳ Ablauf benötigt mehrere Befehlszyklen

sequentielle Ausführung

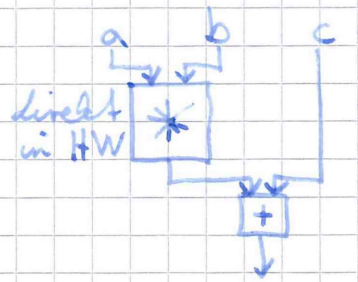
- Δe: execution time
- c^+: maximal mögliches Δe (→ Echtzeitsysteme)

<u>Architektur: Application Specific Instruction Set Processor (ASIP)</u>

Struktur: Mikroarchitektur



MAC: Multiply and Accumulate (statt ALU)



Verhalten: Architektur

Vektor-Rechner / Single Instruction Multiple Data (SIMD)

Beispiel: lineare Algebra (Vektoren, Matrizen)

↳ Filter, digitale Signalverarbeitung → digital signal processor (DSP)

↳ Grafik Prozessoren

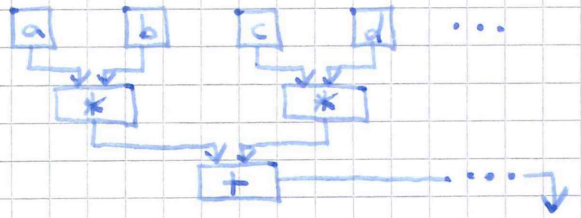
Ergebnis: strikte sequentielle Abarbeitung parallelisiert

↳ weniger Ausführungsschritte

Fläche gegen Zeit getauscht

Architektur: Application Specific Processor (ASP)

Struktur: Mikroarchitektur



Verhalten: Architektur
arbeitet keine sequentiellen Befehle ab
spezifiziert durch Funktion

Ergebnis: nur für eine ganz bestimmte Operation gebaut
Fläche gegen Zeit getauscht

Beispiel: Fourier Trafo (FFT, IFT)
Burst Mode Controller (Datenübertragung, z.B. Telefon)

Architekturen und ihr Einfluss auf den Entwurf

ISP: eine allgemeine Recheneinheit
sequentieller Befehlszyklus

ASIP: mehrere spezielle Recheneinheiten und Bus-Netzwerk
SIMD

ASP: vorgegebene Funktion/Aufgabe/Task fest implementiert

Fläche ↓ Zeit ↑

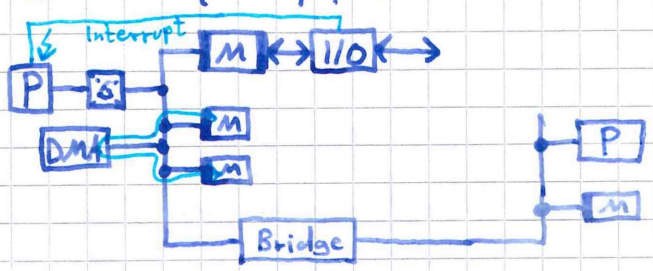
Kombination: ASP für zeitkritische Tasks
ISP für nicht zeitkritische Software

Verschaltung von Komponenten

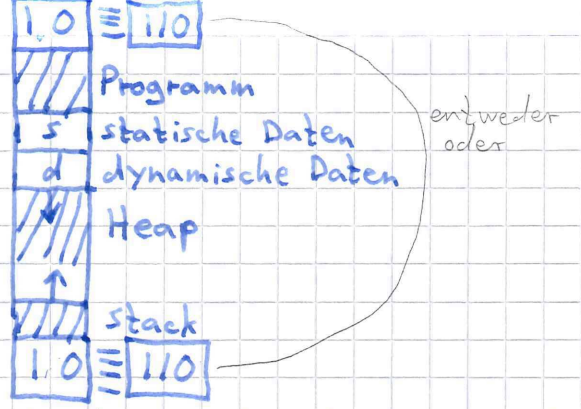
Processing Elements: ISP, ASIP, ASP
Direct Memory Access (DMA)
Bridge (DMA – Dual Port RAM – DMA)

Memory Elements: Eintorspeicher (M), Mehrtorspeicher (M)
↳ große Datenströme
Cache (C), Daten- und Befehlscaches

Hardware Struktur:



Speicher-Layout:



Architektur eines eingebetten Systems (am Beispiel von DECT)

DECT: Digital Enhanced Cordless Telephone

Mobile Part (Telefone), Fixed Part (Funkmasten)

120 Kanäle, 10 Funkkanäle, ~1,8 bis 1,9 GHz

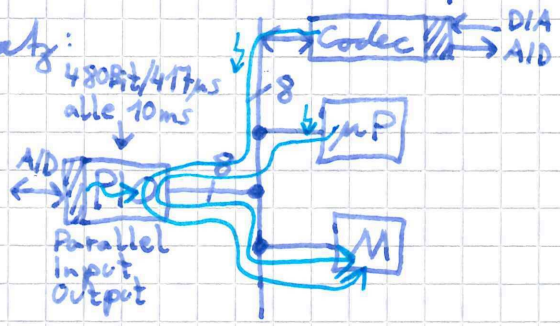
~12 Zeitschlitze ⇒ 24 für bidirektionale Kommunikation

- Frames von 10ms mit Slots 0...11 für MP und 12...23 für FP

  Slot dauert 417 µs und MP, FP wählen „gleichen" Slot $\frac{5}{4}, \frac{5}{16}$

  Slot hat 480 Bit: Sync, A-Feld, B-Feld, CRC

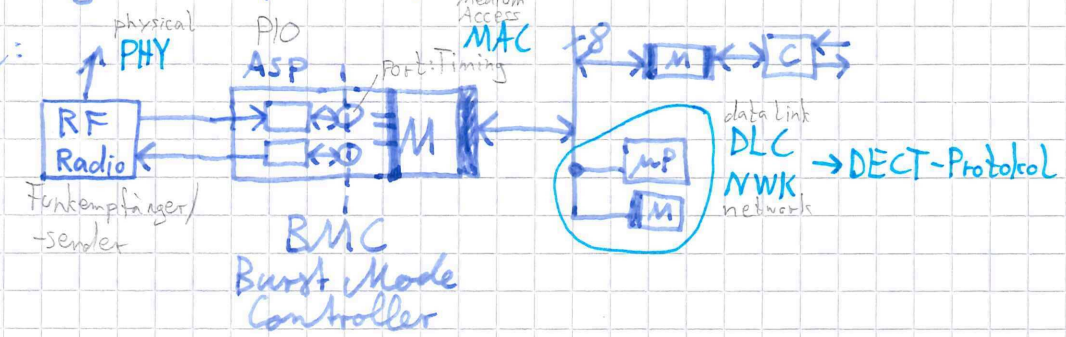
erster Ansatz:



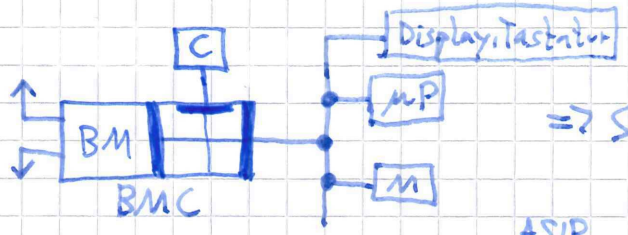
P	Programm-/Textsegment
s	statische Daten
d	dynamische Daten
	Heap
s	Stack
A_i	Audio in
A_o	Audio out

→ zu teuer, Datenfluss ↯

zweiter Ansatz:

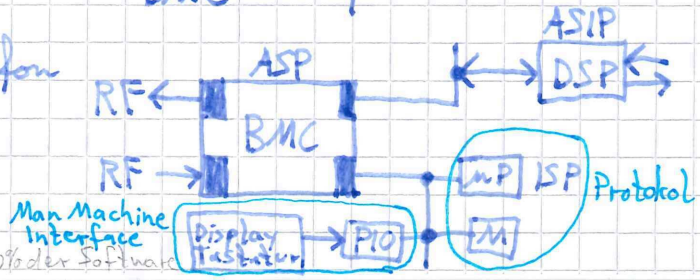


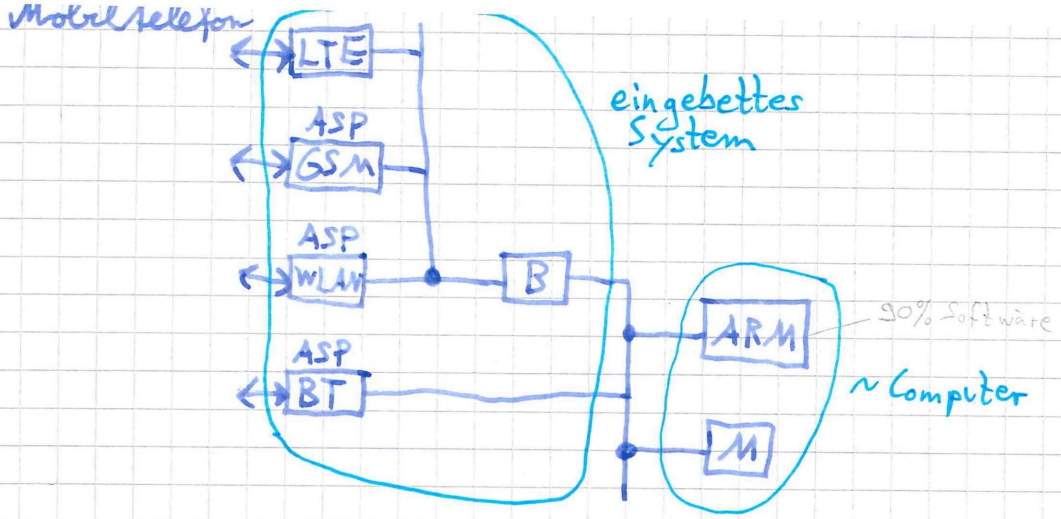
Verfeinerung:



⇒ System on a Chip (auf einem IC)

Ergebnis: Telefon





Logic Fabrics: Technologie & Entwurf

Einführung

Mikrocontroller für bestimmte Applikationen existieren

erste Prototypen meist mit programmierbarer Logik (sehr umfangreich)

für große Stückzahlen muss IC entworfen werden als ASIC

Entwurf → technologische Realisierung → Kosten (Aufwand, Geld)

Entwurfsablauf

einzelne Komponenten können unabhängig voneinander entwickelt werden

- Algorithmus spezifiziert Verhalten
  Spezifikation ist ausführbar → Überprüfung möglich
- gemeinsames Verständnis einer Komponente: port, architecture
  ist validiert und im Gesamtsystem geprüft
- unabhängiges Team kann Testbench für Komponente bauen
  Chipentwicklungs-Team baut weitere architecture (auf anderer Ebene)
  Design-Team kann neue architecture gegen Testbench und System validieren

→ Schrittweise Verfeinerung (deshalb mehrere architectures und configure in VHDL)

Entwurf und Synthese digitaler Schaltungen

automatische Synthese: Verhalten (RTL)

RTL-Synthese

Verhalten (Boolsche Gleichung)

Logiksynthese

Struktur (Gatternetzliste)

Logiksynthese: → logische Komponenten (Logikminimierung)
↓
Gatter (Technologie) →

Entwurfansätze: Simulations-/Validationsgestützt

- Verhalten → Simulation ⇒ Verhalten durch implementierte Struktur austauschen
  Testbench ↗

Correct by Construction

- Verhalten ↔ Simulation
  ↓ ↑
  Synthese Testbench
  ↓ ↑
  Netzliste ↔
- Arbeitet Synthesetool korrekt und verwendete Bibliotheken sind korrekte, dann erhält man eine korrekte Schaltung

mögliche Synthesen: RTL → Logikebene

Silicon-Synthesis/Silicon-Compilation

- Layout der Schaltung wird erzeugt

Architektur-/Verhaltenssynthese

- RTL Schaltung wird erzeugt

Ergebnis: Konstrukteur eines eingebetten Systems muss nicht unbedingt Chipentwickler sein

- Synthesetools kennen und richtig einsetzen
  ↳ automatische Synthese

<u>Logic Fabric: Full Custom Entwurf</u>

Layout und Schaltung werden explizit entworfen und hergestellt
↳ alles ist genau so wie man es haben will

- geometrisches Layout ist entworfen worden

<u>Logic Fabric: Semi-Custom Entwurf</u> (ASIC)

Standardzellen: vorgefertigte Layouts (Geometrie- und/oder Verhaltensbeschreibung) in Bibliothek ↳ Simulation

Gate Array: Array of Logic (Abstand zwischen Arrays)

→

Sea of Gates: Gate Array ohne Abstand zwischen Arrays

Ergebnis: Layout einzelner Komponenten ist vorgefertigt

Verdrahtung zwischen Komponenten muss vorgenommen werden

↳ Layout-Synthese

Vergleich: Standardzellen — Sea of Gates / Gate Array

← Komplexität

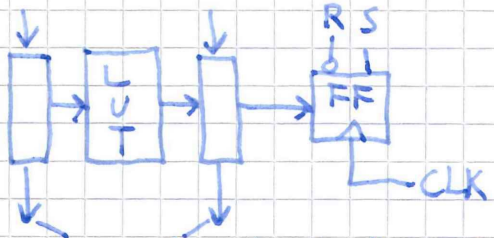
→ Fläche

→ Automatisierung

Logic Fabric: Programmierbare Logik

FPGA: Field Programmable Gate Array

Array von programmierbaren Schaltungen

- über Speicherzellen programmierbar
  ↳ LUT (Look up Table)
- 

Addierer!

Daisy Chain Logik: Durchleitung von Daten von anderen Arrays

Architekturen und Technologie

Architekturen: ISP, ASIP, ASP

Technologien: Full-Custom — Semi-Custom — Programmable

← Komplexität

→ Fläche

→ Automatisierung

← Entwicklung Kosten

→ Chip Kosten

← Stückzahl

Kombination: Jede Architektur kann mit jeder Technologie realisiert werden zu einem IC

Chip Fabrication

Schritte: Circuit Diagrams

Manifacturing: wafer, photolithography
measure electric properties, copper wires

Packaging: cut chip off wafer

Sensoren und Aktoren I

Einführung

Einbettung des Computers: Sensoren, Aktoren, Umsetzer D/A A/D

elektrische Messwerte aufnehmen: Messfühler

Werte → elektrische Signale → vorverarbeiten → zeit- und wertdiskrete Werte

Sensoren und Aktoren in eingebetteten Systemen

Sensor: lat. sensire, fühlen → Messfühler

Gyroscop: Information über Lage des Objektes im Raum

Staurohr: Geschwindigkeit (relativ)

Aufgabe des Sensors: physikalische Größe in elektrische Größe umwandeln

Messprinzip: resistiv

- $R = \sigma \frac{l}{A}$ $\Delta l \sim \Delta R \sim \Delta U$ $R = \frac{U}{I}$

kapazitiv

- $C = \varepsilon_0 \varepsilon_r \frac{A}{d}$ $\Delta d \sim \Delta C \sim \Delta U$ $i = C \frac{du}{dt}$

induktiv (Mikrophon)

Magnetfeld

(Hall-Sensor)

- $\Delta B \sim \Delta U$

Piezoelektrischer Effekt

- $\Delta d \sim \Delta U$

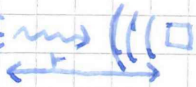
Temperatur

- $\Delta T \sim 1/\Delta R$

optoelektronische Sensoren

- $\Delta q \sim \Delta U$

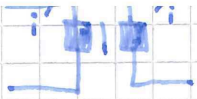
CCD-Kamera

Radar 

↳ Funkwelle

Sonar
↳ akustisches Signal

- Entfernungsmesser

Messwandler: Transformator 

Messumformer, Messverstärker: Operationsverstärker ⇒

A/D-Umsetzer: Messumsetzer: analog → digital

Aktor: Ansteuerung, z.B. Motor

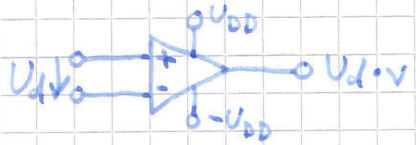
Wie kriegen wir Sensorwerte in Programm, das auf Mikrokontroller läuft?

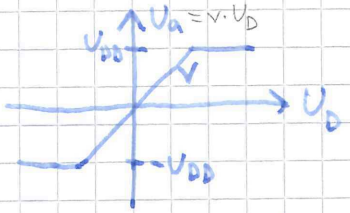
Operationsverstärker und Komparatoren

Operationsverstärker: Verstärker mit besonderen Eigenschaften
↳ OP

sehr gut geeignet um mathematische Operationen als analoge Schaltung zu realisieren

- Messverstärker
- zeitliche Operationen auf Signalen im Analog-Wandler

Schaltsymbol: 

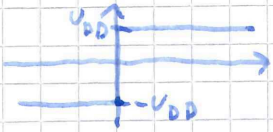
Kennlinie: 

ideal: Verstärkung $v = \infty$

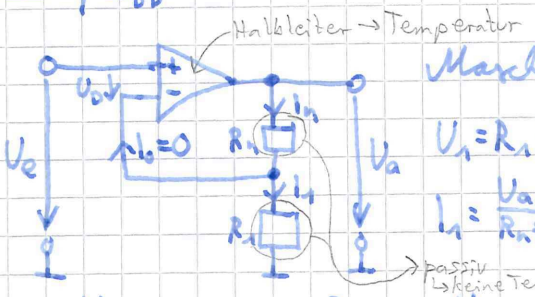
Eingangswiderstand $R_D = \infty$

Eingangsstrom $i_D = 0$

Komparator: Schaltsymbol:

Kennlinie: 

rückgekoppelte OP-Schaltung:
↳ nicht invertierender Verstärker



Masche: $U_e - U_1 - U_D = 0$

$U_1 = R_1 i_1 \rightarrow U_e - R_1 i_1 - U_D = 0$

$i_1 = \frac{U_a}{R_n + R_1} \rightarrow U_e - R_1 \frac{U_a}{R_n + R_1} - U_D = 0$

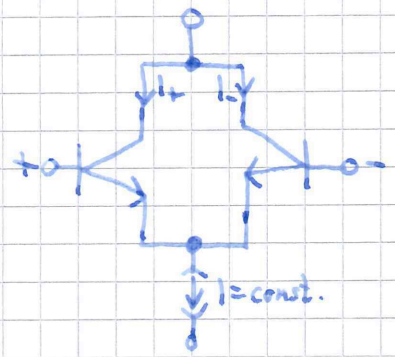
$U_D = \frac{U_a}{v} \rightarrow U_e - U_a \frac{R_1}{R_1 + R_n} - \frac{U_a}{v} = 0$

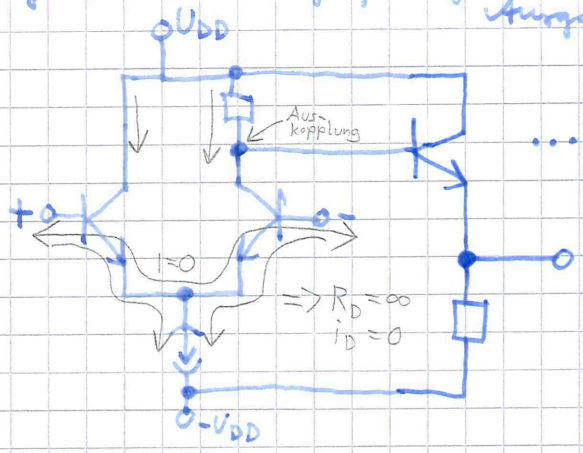
$\Leftrightarrow U_e - U_a \left(\frac{R_n}{R_1 + R_n} - \frac{1}{v} \right) = 0$

$\overset{1/v \to 0}{\Leftrightarrow} U_e / U_a = \frac{R_1}{R_1 + R_n} := k \Rightarrow \frac{U_a}{U_e} = \frac{1}{k} = 1 + \frac{R_n}{R_1}$

Aufbau OP: Differenzverstärker

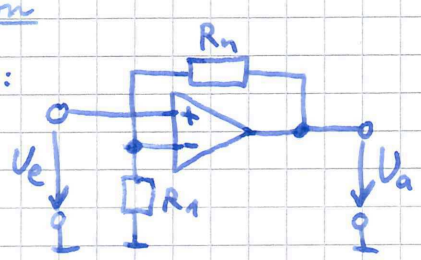
symmetrischer Eingang → asymmetrischer Ausgang





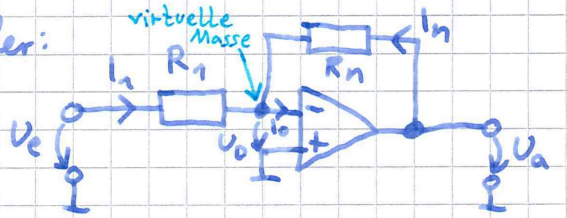
Rechnen mit Operationsverstärkern

nicht invertierender Verstärker:



$$\rightarrow U_a = U_e\left(1 + R_n/R_1\right)$$

invertierender Verstärker:



$$\rightarrow U_a = U_e\left(-R_n/R_1\right)$$

Rechnung: $I_1 + I_n = I_0 = 0 \Rightarrow I_1 = -I_n = I$

$$U_e - U_1 - U_D = 0 \Leftrightarrow U_e - I_1 R_1 + \cancel{U_D}^{\,0} = 0$$

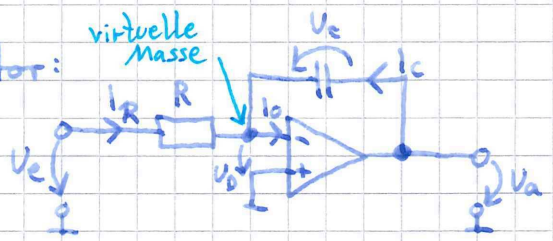
$$\Leftrightarrow U_e - U_1 = 0 \Rightarrow U_e = U_1$$

$$\Rightarrow I = \frac{U_1}{R_1} = \frac{U_e}{R_1}$$

$$-\cancel{U_e + U_1}^{\,0} - U_n + U_a = 0$$

$$\Leftrightarrow U_a = U_n = I_n R_n = -I R_n = -U_e \frac{R_n}{R_1}$$

Integrator:



$$\rightarrow U_a = -\frac{1}{RC}\int_0^t U_e(t')\,dt'$$

Rechnung: von oben wissen wir $I_R = -I_c = I$, $U_e = U_R$, $I = \frac{U_1}{R} = \frac{U_e}{R}$

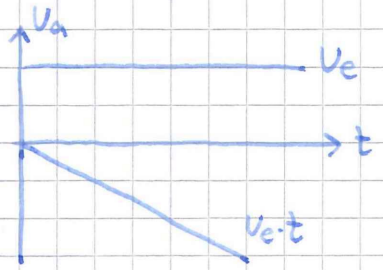
$$U_a = U_c \Rightarrow I_c = C\frac{dU_c}{dt} = C\frac{dU_a}{dt} = -I$$

$$U_e = U_R = I \cdot R = -RC\frac{dU_a}{dt} \Rightarrow dU_a = -\frac{1}{RC}U_e\,dt$$

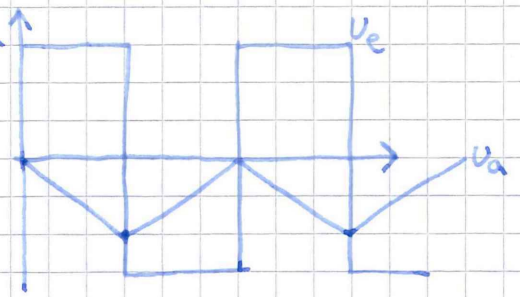
$$\Rightarrow U_a = -\frac{1}{RC}\int_0^t U_e(t')\,dt'$$

Fall $U_e(t) = \text{const.}$: $U_a = -\frac{1}{RC} U_e \cdot t$

graphisch:

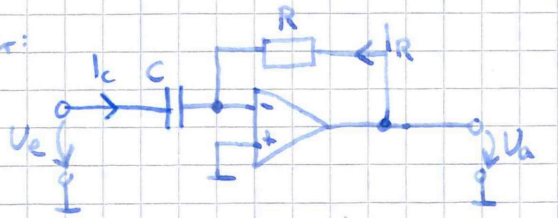


Fall $U_e(t)$ period. rect: U_a



→ Sägezahn Spannung erzeugt

Differenzierer:



$$\rightarrow U_a = -RC \frac{dU_e}{dt}$$

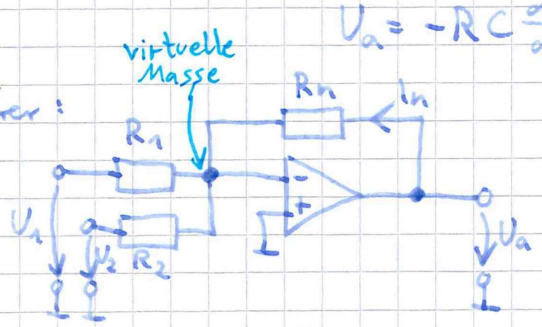
Rechnung: virtuelle Masse → $U_a = U_R$, $U_e = U_C$, $I_R = -I_C$

$$U_a = U_R = I_R R = -I_C R$$

$$I_C = C \frac{dU_C}{dt} = C \frac{dU_e}{dt}$$

$$U_a = -RC \frac{dU_e}{dt}$$

Addierer:

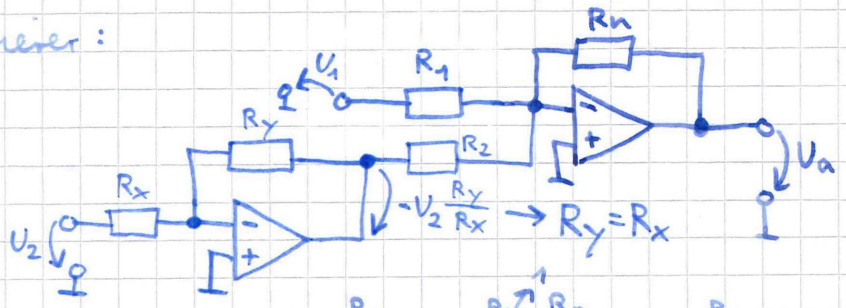


$$\rightarrow U_a = U_1 \frac{R_n}{R_1} + U_2 \frac{R_n}{R_2}$$

Rechnung: $U_1 = R_1 \cdot I_1$ $\quad U_2 = R_2 \cdot I_2$ $\quad I_n = I_1 + I_2 = \frac{U_1}{R_1} + \frac{U_2}{R_2} = -I$

$$U_a = U_n = -I R_n = \left(\frac{U_1}{R_1} + \frac{U_2}{R_2}\right) R_n = U_1 \frac{R_n}{R_1} + U_2 \frac{R_n}{R_2}$$

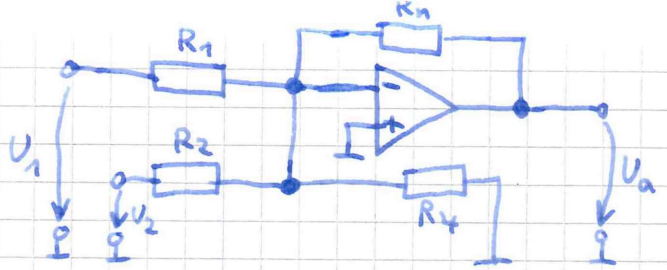
Subtrahierer:



$$-U_2 \frac{R_y}{R_x} \rightarrow R_y = R_x$$

$$\rightarrow U_a = U_1 \frac{R_n}{R_1} - U_2 \cancel{\frac{R_y}{R_x}}^{\,1} \frac{R_n}{R_2} = U_1 \frac{R_n}{R_1} - U_2 \frac{R_n}{R_2}$$

alternative Schaltung mit nur einem OP



allgemein: Regeln: Knoten-, Maschen-

idealer OP: $v \to \infty$, $i_d = 0$, $R_d = \infty$

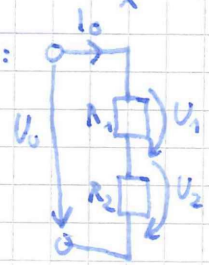
virtuelle Masse $\Rightarrow U_e = U_v$, $U_a = U_0$

Messbrücken

resistiver Messfühler: $R = \sigma \frac{l}{A} \to \Delta R = \sigma \frac{\Delta l}{A}$

$R_x = R + \Delta R$

mit Spannungsteiler:



Maschenregel: $U_0 - U_2 - U_1 = 0$

Knotenregel: $I_0 = I_1 = I_2 = \frac{U_1}{R_1} = \frac{U_2}{R_2}$

$U_1 = U_0 - U_2 = U_0 - I_0 R_2 = U_0 - \frac{U_1}{R_1} R_2$

$U_0 = U_1 + U_2 = U_1 \left(1 + \frac{R_2}{R_1}\right) = U_1 \left(\frac{R_1 + R_2}{R_1}\right)$

$U_1 = \frac{R_1}{R_1 + R_2} U_0$, $U_2 = \frac{R_2}{R_1 + R_2} U_0$, $R_0 = R_1 + R_2$, $I_0 = \frac{U_0}{R_0}$

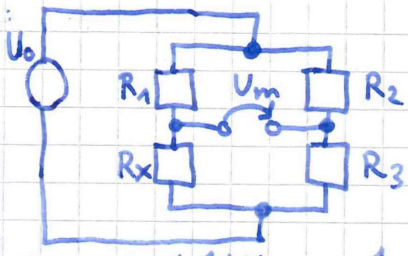
Anwendung auf Messfühler

- $R_2 = R + \Delta R \to$ messen U_2

$$U_2 = \frac{R + \Delta R}{R_1 + R + \Delta R} U_0 = \frac{1 + \frac{\Delta R}{R}}{R_1 + 1 + \frac{\Delta R}{R}} U_0$$

- für $\Delta R / R = 10^{-3}$ und $U_0 = 10V$ muss $1mV$ gemessen werden können

mit Wheatstone Brückenschaltung:



$U_m = U_x - U_3$

$= U_0 \left(\frac{R_x}{R_1 + R_x} - \frac{R_3}{R_2 + R_3}\right)$

Längenänderung

$\hookrightarrow R_x = R + \Delta R$

$$U_m = U_0 \left(\frac{1+v}{1+v+k} - \frac{1}{k+1}\right)$$

$$= \frac{vk}{(1+v+k)(k+1)} U_0 \qquad v = \Delta R / R, \; k = \frac{R_3}{R_2}$$

Bedingung: $v \ll k+1 \to v \ll 2$

$\hookrightarrow U_m = \frac{vk}{(k+1)^2} U_0$

Für welches k wird U_m maximal?

$$\frac{d}{dk} \frac{k}{(k+1)^2} = \frac{(k+1)^2 - 2(k+1)k}{(k+1)^2} \overset{!}{=} 0 \to k = 1$$

$\Rightarrow U_m = \frac{1}{4} \cdot U_0 = \frac{1}{4} \frac{\Delta R}{R} U_0$

für $\Delta R/R = 10^{-3}$ und $U_0 = 10V$ beträgt $U_m = 250mV$

Bedingung $I_m = 0$ beim Messen von U_m

↳ OP: $I_0 = 0$ → kein Haftstrom

Sensoren und Aktoren II

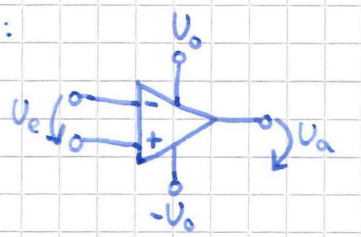
Einführung

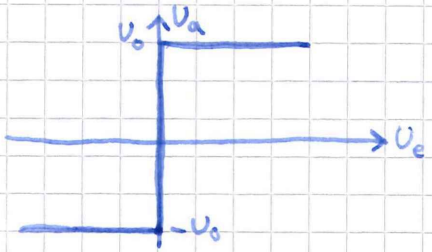
Sensor: von „Physik" zur Spannung

Komparatoren

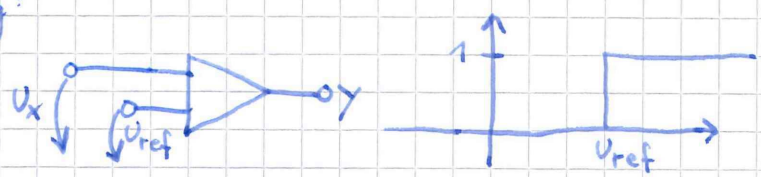
Zum Vergleich von Spannungen.

allgemein:

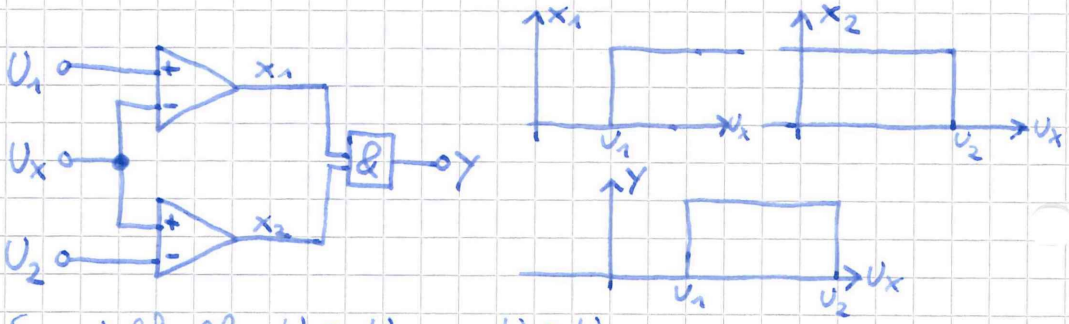




Logikpegel am Ausgang:

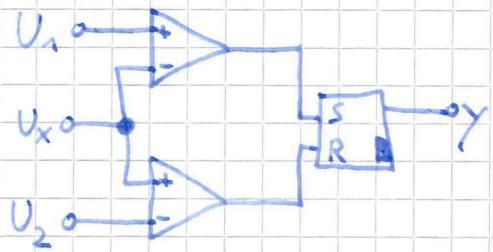


Fensterkomparator:

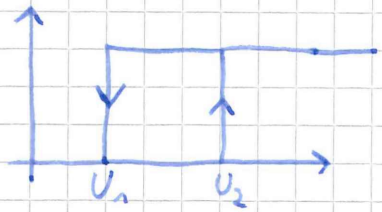


Spezialfall: $U_1 = -U_{ref}$, $U_2 = U_{ref}$

Schmitt-Trigger:

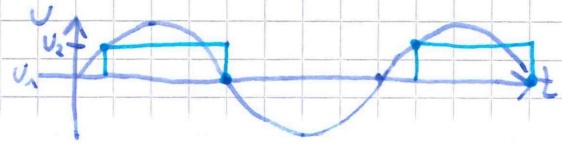


Hysterese



Anwendung: Kompensation von jitternden Eingangsmesswerten / Sensoren

→ Diskriminator (Pulsformung)

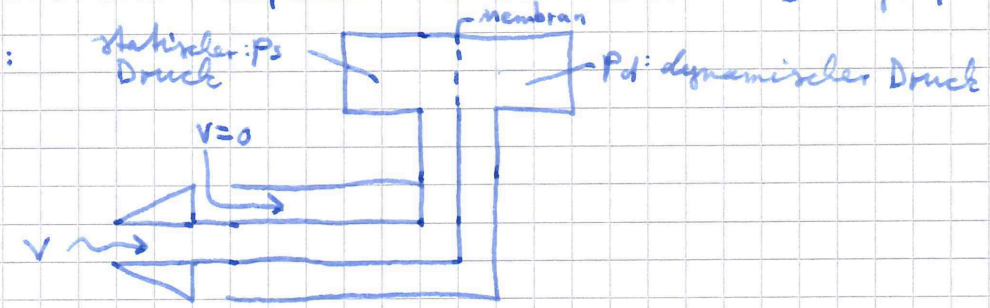


bei U_2 an
bei U_1 aus

Beispiel: Sensoren zur Geschwindigkeitsmessung

Geschwindigkeit durch Luft mit Sensoren messen: v_L sorgt für Auftrieb

Prandtlsonde:

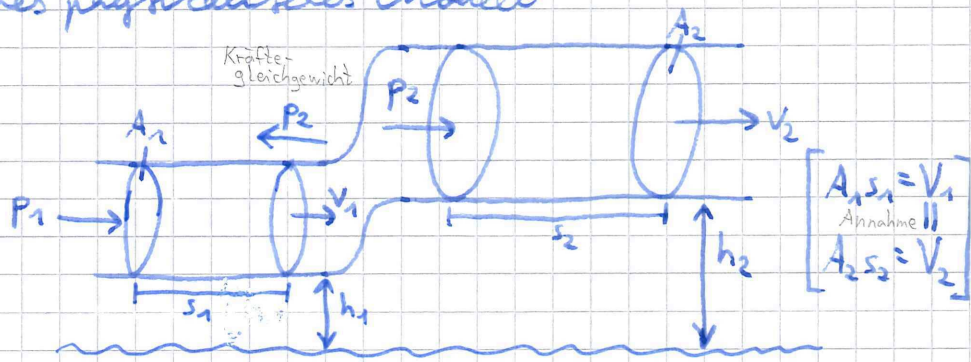


im Flugzeug:
- $p_d \longrightarrow v_L$

  $p_s \longrightarrow$ Höhenmesser (Barometrische Höhenformel)

  $\longrightarrow$ Steigungsrate
- v_L berechnen

  allgemeines physikalisches Modell



$$W = Fs = (p_2 - p_1) A_1 s_1 = (p_2 - p_1) V_1 \qquad \left[p = \frac{F}{A}\right]$$

$$W_{pot} = mgh = mg\,\Delta h = mg\,(h_2 - h_1) = \rho g V (h_2 - h_1) \qquad [m = \rho V]$$

$$W_{kin} = \frac{1}{2} m v^2 = \frac{1}{2} \rho V (v_2^2 - v_1^2)$$

Energieerhaltung: $W = W_{pot} + W_{kin}$

$$\Leftrightarrow p_2 - \rho g h_2 - \frac{1}{2}\rho v_2^2 = p_1 - \rho g h_1 + \frac{1}{2}\rho v_1^2$$

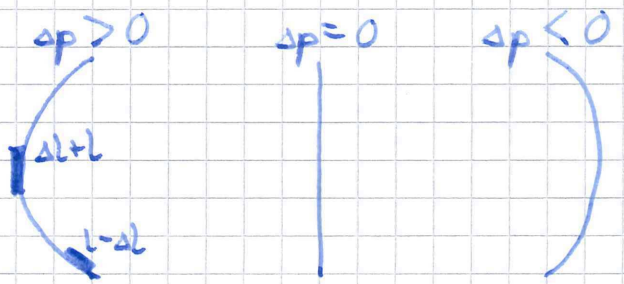
$\hookrightarrow$ Bernoulli Gleichung

hier: $h = h_1 = h_2$, $v_s = 0$ $\rightarrow$ Einsetzen: $p_2 = p_d$, $p_1 = p_s$, $v_2 = v_s$, $v_1 =$

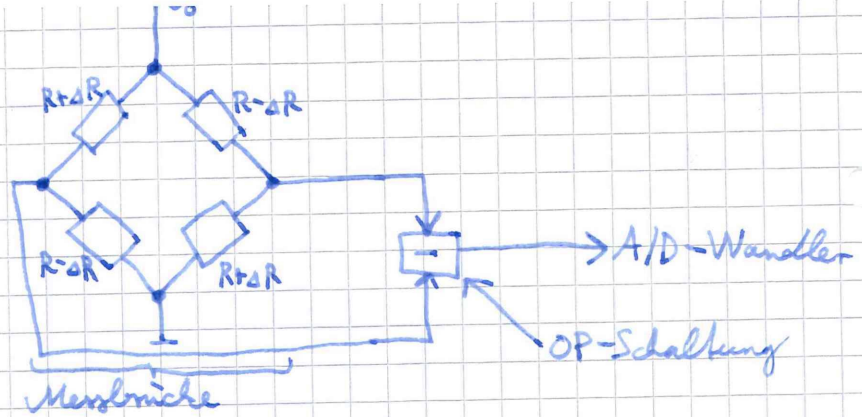
$$p_d - \rho g h - \frac{1}{2}\rho v_s^2 = p_s - \rho g h + \frac{1}{2}\rho v_d^2$$

$$\Leftrightarrow p_d = p_s + \frac{1}{2}\rho v_d^2 \Rightarrow v_d = \sqrt{\frac{2}{\rho}\Delta p} = \sqrt{\frac{2}{\rho}(p_d - p_s)}$$

- Sensor für v_L: Dehnungsmessstreifen an Membran



Schaltung:



+ Zusatzschaltung zum Kalibrieren der Messbrücke

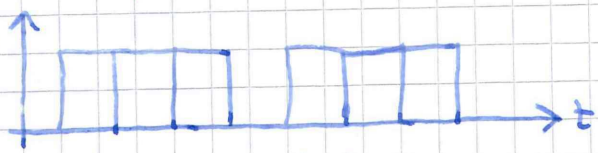
Sensoren zur Umdrehung- und Abstandsmessung

Drehratenmessung: Drehwinkelsensor

optisch: • Glasscheibe mit winkeltreuen glasdunkeln Abschnitten

Leuchtdiode (zwei) zum Messen

Lichtquelle

• 

• Drehgeschwindigkeit

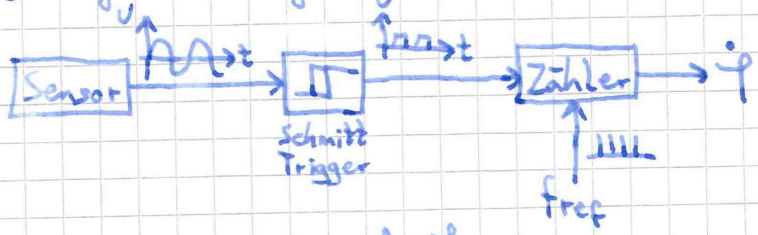
Richtung der Drehung (zwei Dioden)

magnetisch: • Zahnrad mit n Zähnen

Hall-Sonde

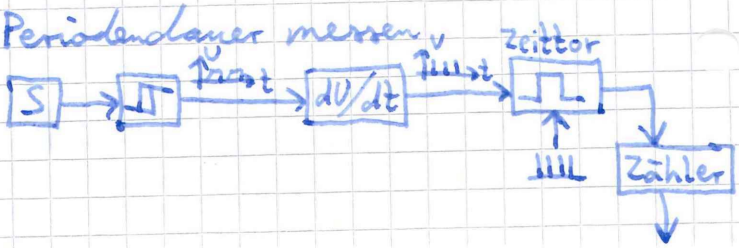
• lässt man ein Zahn weg im Rad → 0 Lage

• Drehgeschwindigkeit messen



$\varphi_0 = 2\pi/n$ $\quad \dot{\varphi} = \frac{\varphi_2 - \varphi_1}{t_2 - t_1}$

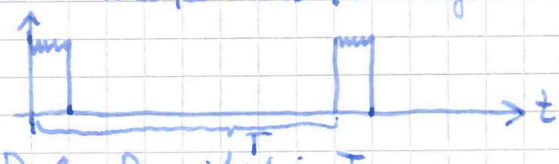
• Periodendauer messen



Abstandsmessung: Radar (Radio Direction and Ranging)

- Richtungs- und Entfernungsbestimmung mit Funk(wellen)
- Prinzip: Funkwelle wird ausgesendet
  Objekt mit genug metallischer Fläche reflektiert Welle

gepulstes Radar: Modulation des Funksignals



Pulse Repitition Frequency
↳ $PRF = 1/T$

- ist eingebettes System: Funksender und -empfänger
  Auswertungselektronik
- Entfernung: Laufzeitmessung $L = \frac{ct}{2}$ (c: Lichtgeschwindigkeit)
  Richtung: drehende Antenne → gerichtet
  Empfänger-Array → Laufzeitunterschied

Sonar (Sound Direction and Ranging)

- Richtungs- und Entfernungsbestimmung mit Schallwellen
- mechanisch: Glocke mit Hammer und Hörgerät
  piezoelektrisch: Piezokristall
  ↳ Ausdehnung bei Spannung
  Druckunterschiede erzeugen Schallwellen
- Entfernung: Laufzeitmessung $L = \frac{vt}{2}$ (v: Schallgeschwindigkeit)
  Richtung: Empfänger-Array → Laufzeitunterschiede

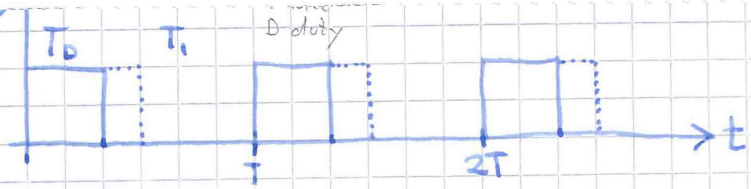
<u>Aktoren: Ansteuern von Motoren mit Pulsweiten-Modulation</u>

früher: Hydraulik, Bowdenzüge

1980: Fly by Wire → elektrische Leitungen ECU → A → Mechanik

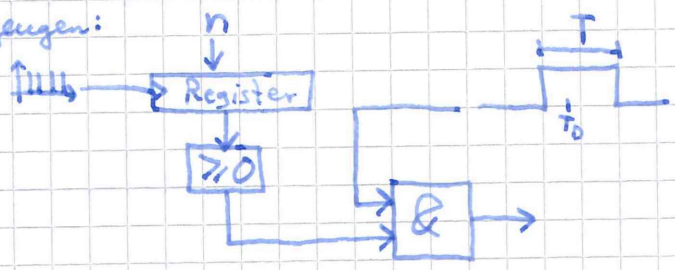
analog: ECU $\xrightarrow[\text{Signal}]{\text{PWM}}$ PWM-Demodulator → Motor → Mechanik

Puls-Weiten-Modulation (PWM):

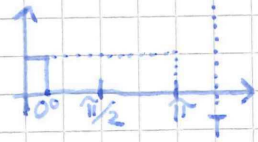


$$U_{eff} = \int_0^T U(t)\,dt = \int_0^{T_D} U(t)\,dt = \frac{T_D}{T} U_0 = U_D$$

• erzeugen:

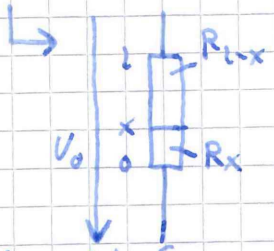


Verhältnis PWM-Drehwinkel: Winkeleinstellung des Aktors in PWM codieren



Motor ansteuern: • PWM decodieren mit Potentiometer $P(\alpha)$ am Motor

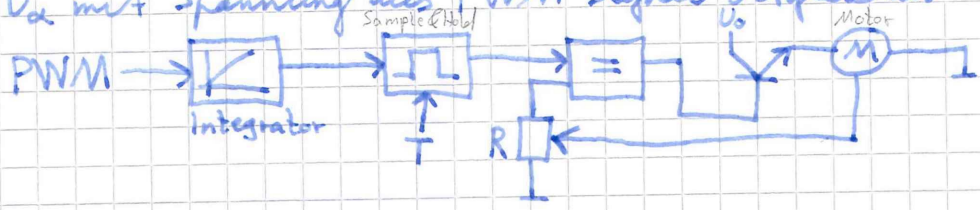
Potentiometer liefert $U_\alpha = \frac{r\alpha}{2\pi} U_0$ (Drehpotentiometer)



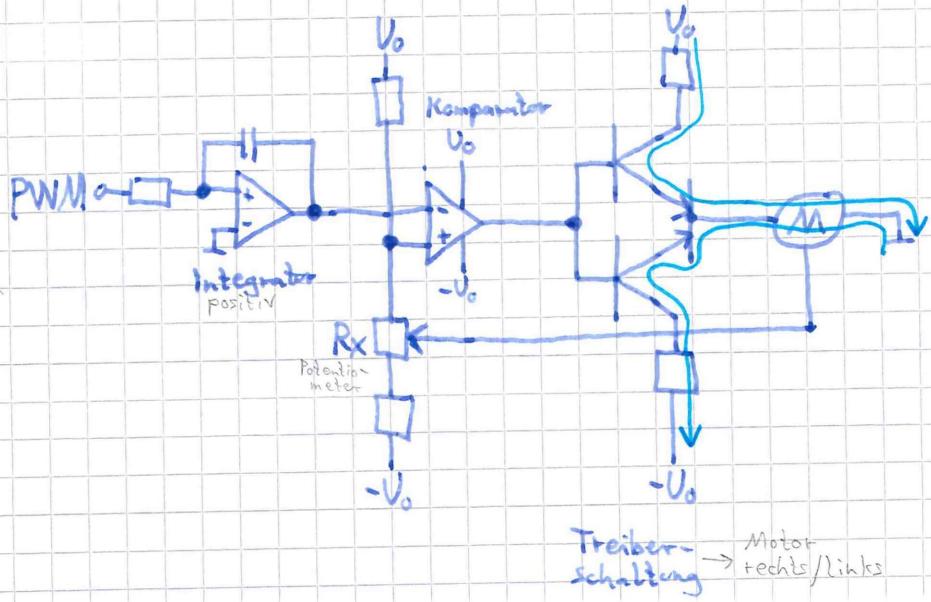
$R = \sigma \frac{L}{A}$

$$U_x = \frac{R_x}{R_x + R_{L-x}} U_0 = \frac{x}{x + L - x} U_0 = \frac{x}{L} U_0$$

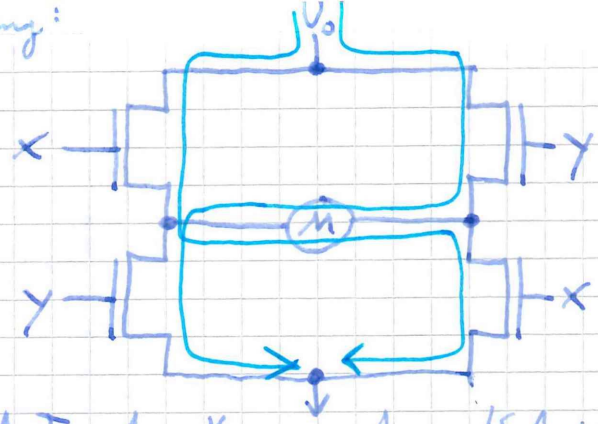
• U_α mit Spannung aus PWM Signal vergleichen



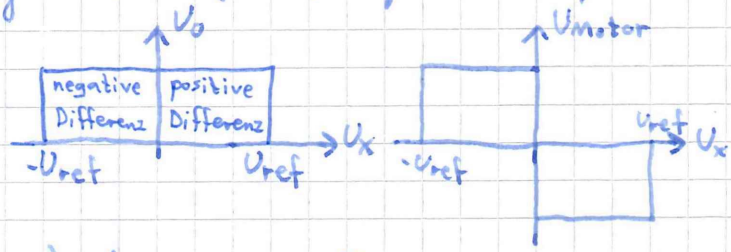
Vergleicher muss Unterschied zwischen positiver und negativer Differenz erkennen können

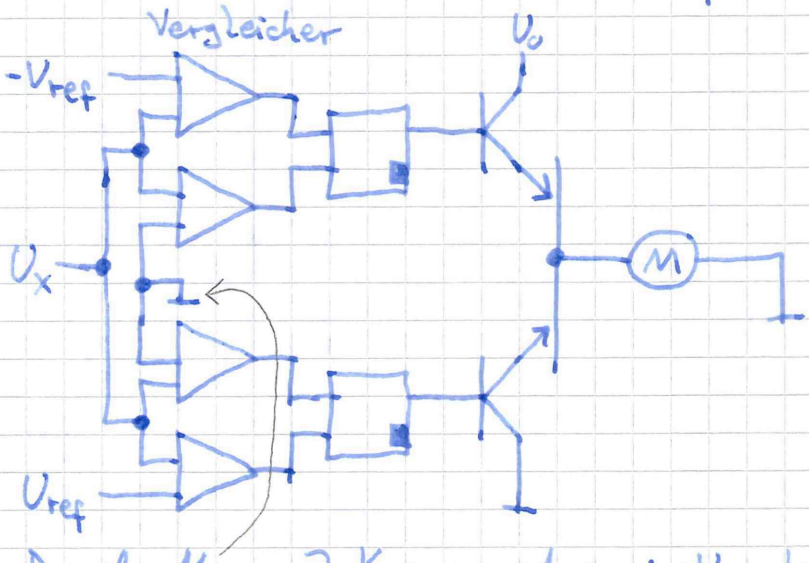


Treiberschaltung:



Realisierung mit Fenster-Komparatoren / Schmitt-Triggern



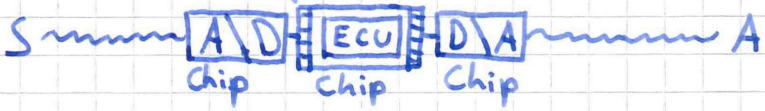


Durch Masse 2 Komparatoren: $-U_{ref}$ bis 0
0 bis U_{ref}

AVIONIC: linearer Motor auf Magnetfeld mit Gestänge zur Physik

<u>Verbinden der Sensoren mit einem eingebetteten Computer</u>

Modelle: klassisch

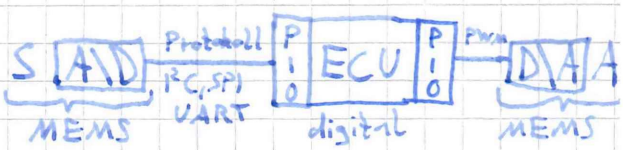


- jeder Standard µC verwendbar: Logik-Halbleiter (CMOS)

integrierter

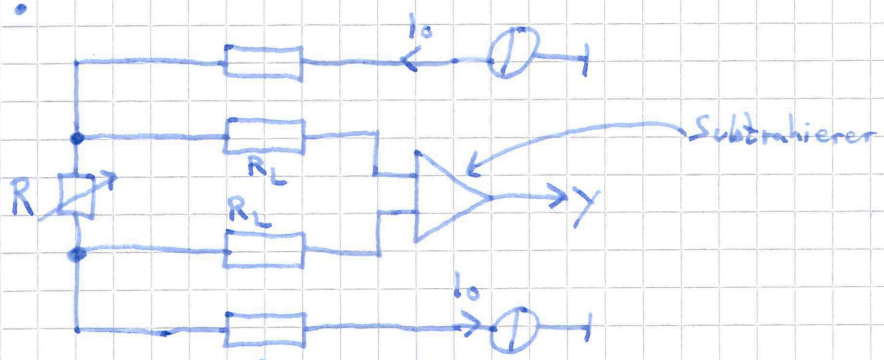
- Logik und Analog → zusätzliche Prozessschritte
  → teuer

Trennung nach analog/digital



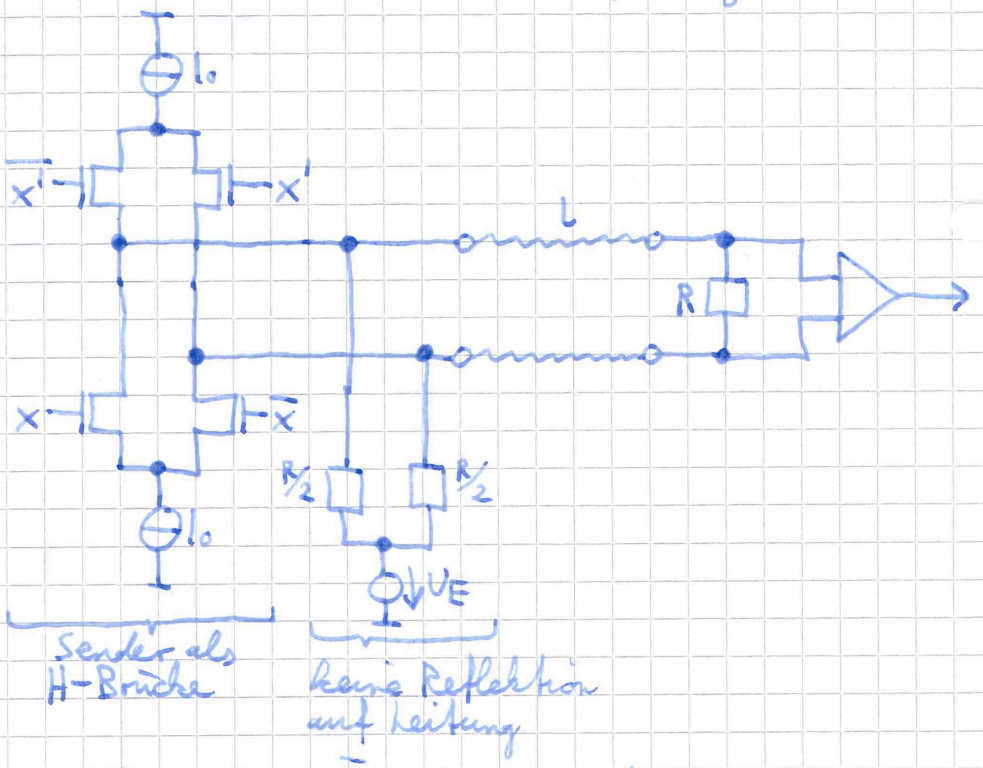
- MEMS: micro electro mechanic system (analog)
  → kostengünstig

analoge Übertragung: Leitung anfällig, Widerstand der Leitung

- 

über R_L fällt keine Spannung ab

- Low Voltage Differential Signaling (LVDS)



Differentielle Übertragung des Signals

↳ L aus Verzwirbelten-Leitungen

äußere Einflüsse mitteln sich durch Subtrahierer am Widerstand R heraus

Abtastung von Signalen und Digital/Analog-Umsetzer

Einführung

analoge Spannung ↔ digitale Zahlenwerte

Parallelumsetzer: sehr hoher Hardwareaufwand, schnell

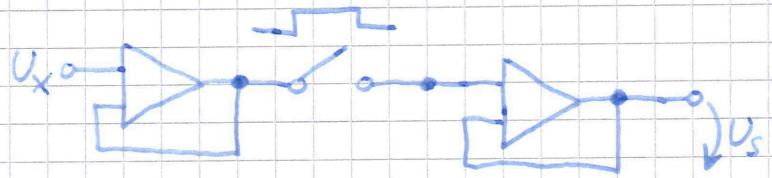
Zählverfahren: kostengünstig in der Herstellung (wenig Hardware), langsam

Realisierung durch analoge Schaltungen mit OPs

AD-Umsetzer: Theoretische Grundlagen

analoges Signal: zeit- und wertekontinuierlich
digitales Signal: zeit- und wertediskret
Abtastung

Abtast-Halte-Glied:
↳ AHG



Schalter: • offen → U_s lesen
geschlossen → nächstes U_x aufnehmen

• Signaltheorie

Heaviside-Funktion: $H(t)$, $\Theta(t)$

$$H(t) = \begin{cases} 1, & t > 0 \\ 0, & t < 0 \end{cases}$$

Rechteck-Funktion: $\mathrm{rect}(t)$

$$\mathrm{rect}(t/T) = H(t + T/2) \cdot H(T/2 - t)$$

Dirac-Impuls: $\delta(t)$

$$\delta(t) = \lim_{\varepsilon \to 0} \int_{-\infty}^{\infty} \frac{1}{2\varepsilon} H(t+\varepsilon) H(\varepsilon - t)$$

$$\int_{-\infty}^{\infty} \delta(t) = 1 \wedge \delta(t) = 0 \quad \forall t \neq 0$$

Ausblendeigenschaft

$$\int_{-\infty}^{\infty} f(t)\,\delta(t - t_0)\,dt = f(t_0)$$

Dirac-Kamm: $\sqcup\!\sqcup(t)$

$$\sqcup\!\sqcup(t) = \sum_{n \in \mathbb{Z}} \delta(t - nT)$$

Signaltheorie: komplexe Fourierreihe Darstellung für jedes periodisches Signal

• $f(t) = \sum_{n \in \mathbb{Z}} c_k e^{int}$, $c_k = \frac{1}{p} \int_{-p/2}^{p/2} f(t)\, e^{i \frac{n}{p} t}\, dt$, $p > 0$

Fouriertransformation Wechsel in Frequenzbereich

• $F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t}\, dt$

$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{-i\omega t}\, d\omega$

• $\sum_{k \in \mathbb{Z}} f(k) = \sum_{k \in \mathbb{Z}} \int_{-\infty}^{\infty} f(t) e^{-i2\pi k t}\, dt$

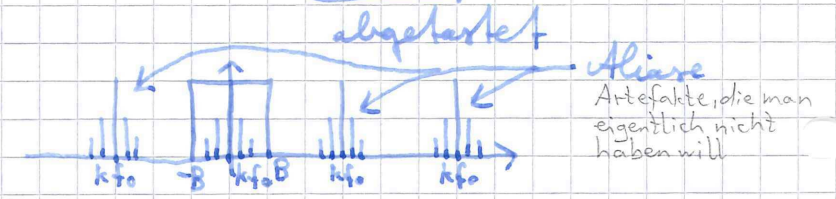
$= \int_{-\infty}^{\infty} f(t) \sum_{k \in \mathbb{Z}} \delta(t-k)\, dt$

$= \sum_{k \in \mathbb{Z}} \int_{-\infty}^{\infty} f(t)\, \delta(t-k)\, dt = \sum_{k \in \mathbb{Z}} f(k)$

Poissonsche Summenformel: $\sqcup\!\sqcup_T \circ\!\!-\!\!\bullet \frac{1}{T} \sqcup\!\sqcup_{1/T}$

↳ neutrales Element der Trafo

Abtastung (im Frequenzbereich): • $x(t) = \int_{-\infty}^{\infty} X(\omega) \underbrace{\sum_{k\in\mathbb{Z}} t_0 (t-kt_0)}_{\text{abgetastet}} e^{i\omega t} d\omega$, $t_0 = \frac{1}{f}$



$$x(t) = \int_{-\infty}^{\infty} X(\omega) \sum_{k\in\mathbb{Z}} \omega_0 (\omega - k\omega_0) e^{i\omega t} d\omega$$
$$\overset{!}{=} \int_{-\infty}^{\infty} X(\omega_0)\, \omega_0 e^{i\omega t} d\omega$$

→ Rechteckfilter $M(-B,B) = \mathrm{rect}\left(\frac{\omega-\omega_0}{2B}\right)$

$$x(t) = \sum_k \int_{-\infty}^{\infty} X(\omega)\, \mathrm{rect}\left(\frac{\omega-\omega_0 k}{2B}\right) \omega_0 e^{i\omega t} d\omega$$
$$= \sum_k \int_{-\infty}^{\infty} X(\omega)\, \omega_0 H(\omega-(k\omega_0-B))\, H(k\omega_0+B-\omega) e^{i\omega t} d\omega$$
$$= \sum_k \int_{-k\omega_0-B}^{k\omega_0+B} \omega_0 X(\omega_0) e^{i\omega t} d\omega$$
$$x(t) = \frac{1}{T} \int_{-B}^{B} \sum X(\omega_0)\, e^{i2\pi \frac{n}{T}}$$

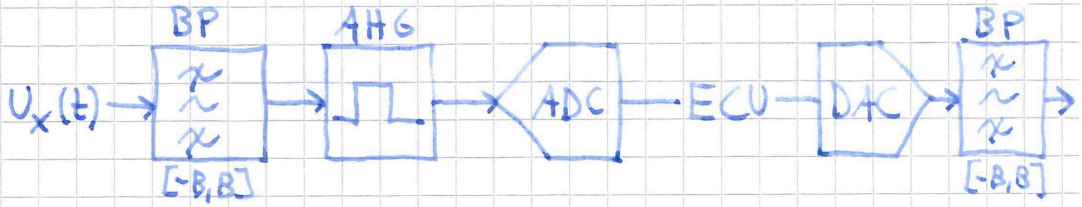
↳ Fourier-Reihe ⇒ $B = T/2$

• Abtasttheorem: Damit eine korrekte Rekonstruktion des Signals möglich ist muss mit mindestens der doppelten Frequenz des das Eingangssignal als Bandbreite hat abgetastet werden → $f_s \geq 2 \cdot f^{max}$

ansonsten: Überlappung mit Aliasen

↳ Schwebungen

Zusammenfassung:

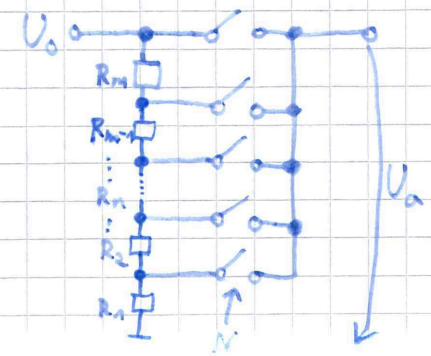


Abtastfrequenz $f = 2B$

DA-Umsetzer (DAC): Prinzipien und Verfahren

Aufgabe: binäre Zahl → dazu proportionale Spannung

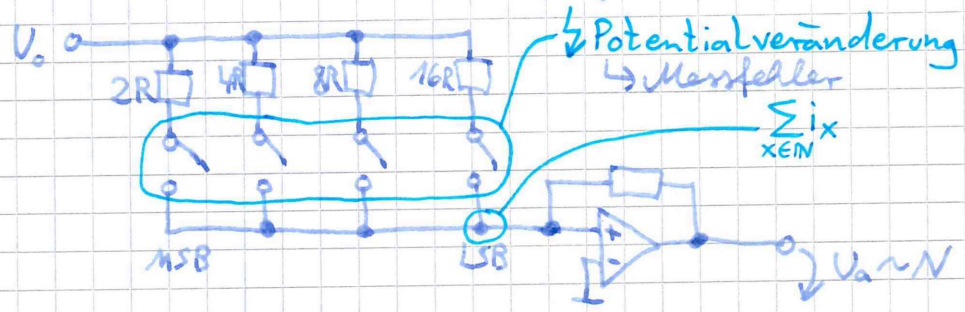
Parallelverfahren:



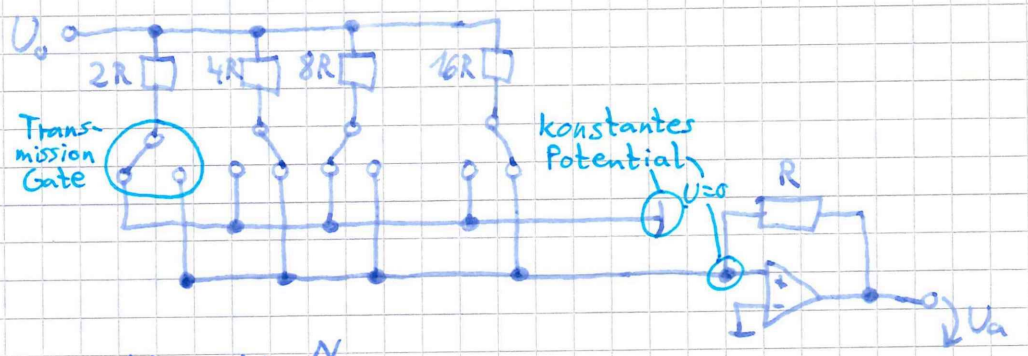
$$U_a = \sum_{i=1}^{n} R_i \Big/ \sum_{i=1}^{m} R_i \cdot U_0$$

binäre Zahl N muss one-hot codiert werden, sodass immer genau ein Schalter geschlossen ist

Wägeverfahren: Variante 1 (Widerstandsschaltung)

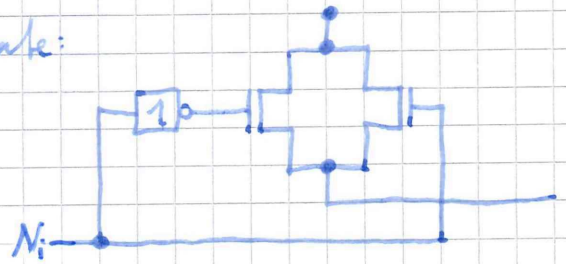


Variante 2

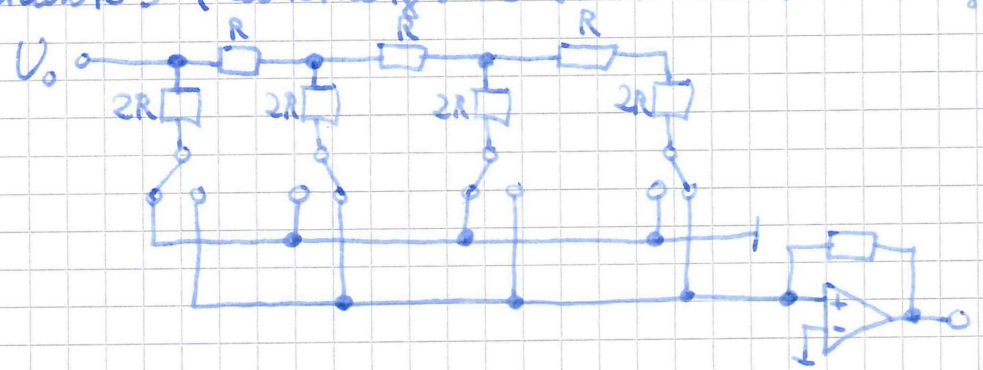


$$U_a = -U_0 \frac{N}{16}$$

Transmission Gate:



Variante 3 (Leiternetzwerk statt Widerstandsnetzwerk)



unabhängig von Schwankungen von R!

Zählverfahren

Analog-Digital Umsetzer (ADC)

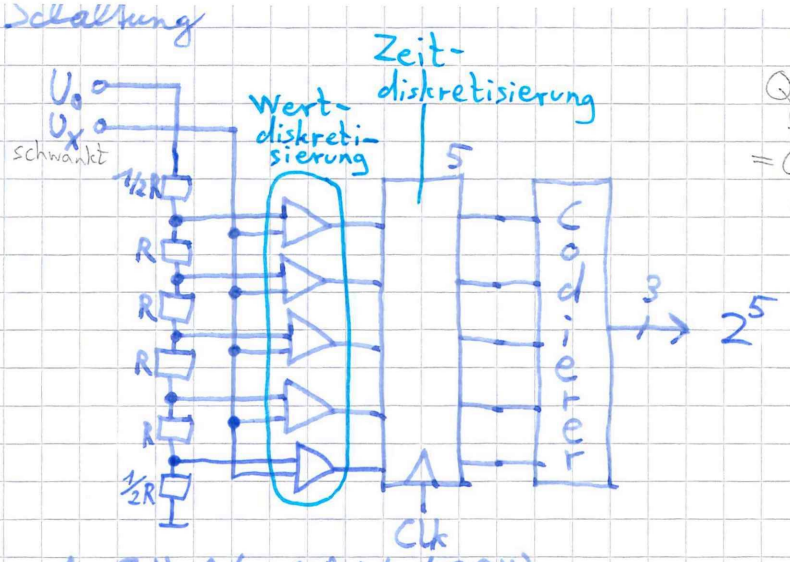
Vom Parallelverfahren zum Wägeverfahren

Parallelverfahren: word at a time

Schablonenumsetzer: Nachteil

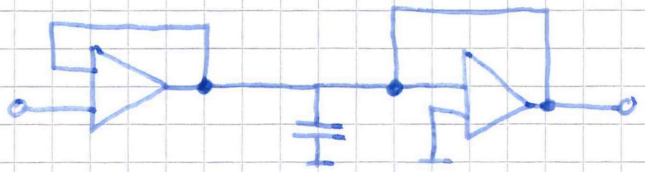
- Zur Darstellung der Zahl 2^n werden n Komparatoren benötigt

Schaltung

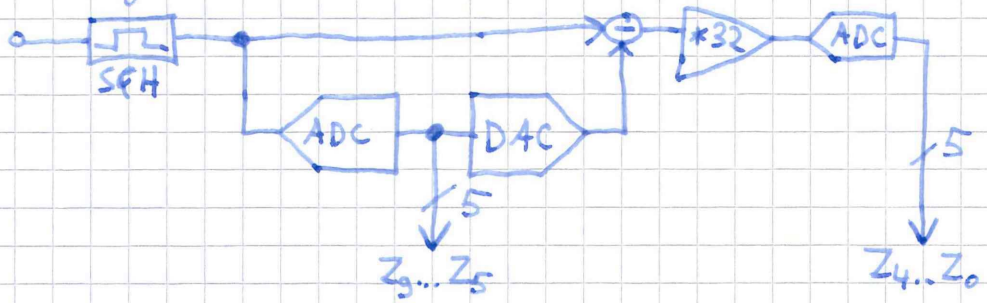


Quantisierungsfehler
= Quantisierungsrauschen

Kaskadenverfahren: Sample & Hold-Glied (S&H)



Schaltung

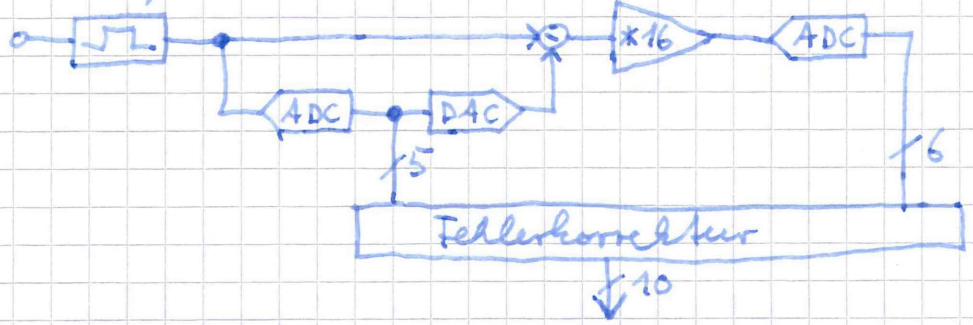


ADC ist Schablonenumsetzer

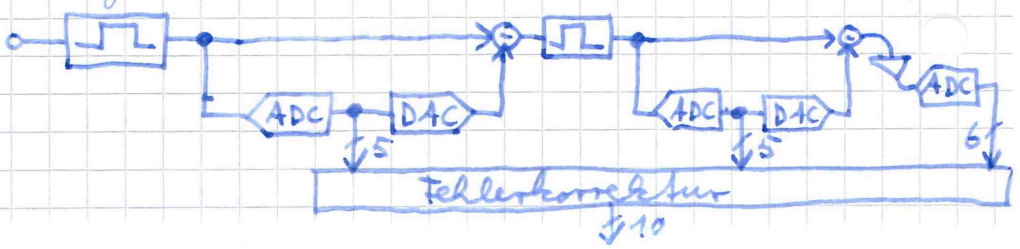
Vorteil

- kostengünstiger (weniger Hardware)
- fast gleich schnell

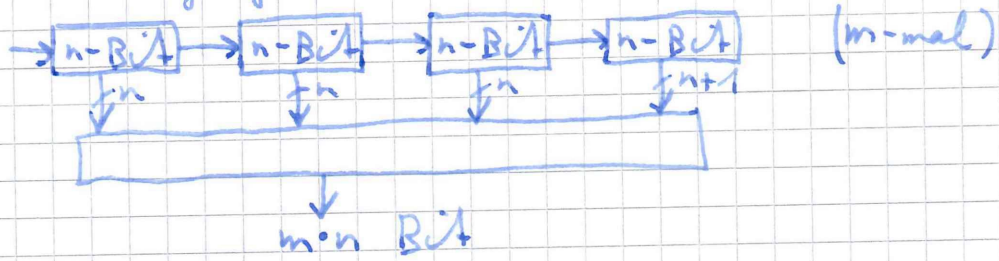
Schaltung mit Fehlerkorrektur



Pipelineverfahren: Schaltung



Schaltung (grob)

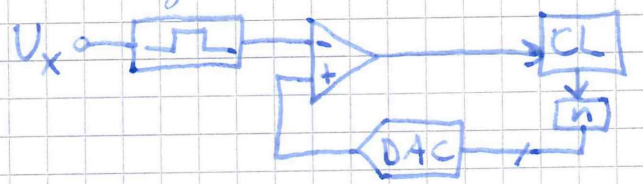


Vorteil

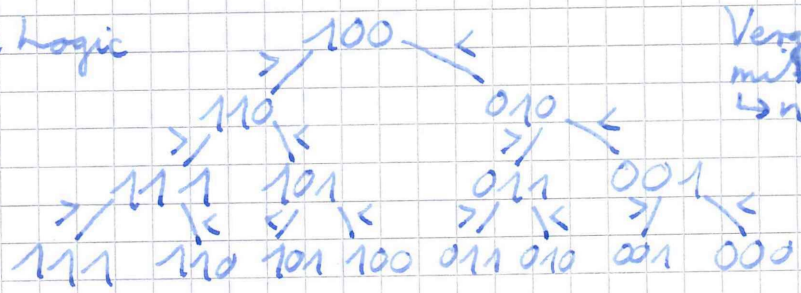
- erhöhter Durchsatz — fertige Komponente kann bereits nächsten Input bearbeiten
- jede Stufe bringt zusätzliche Quantisierungsstufe

Wägeverfahren: digit at a time

Schaltung



Compare Logic



Vergleich mit U_x
↳ $n \gtrless U_x$

Vorteil

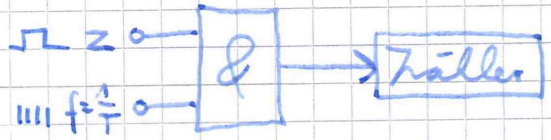
- CL kann in Software realisiert werden

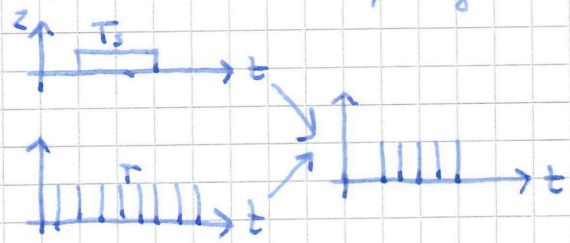
Zählverfahren

allgemein: level at a time

Genauigkeit erhöhen, indem man Zeit oder Frequenz misst

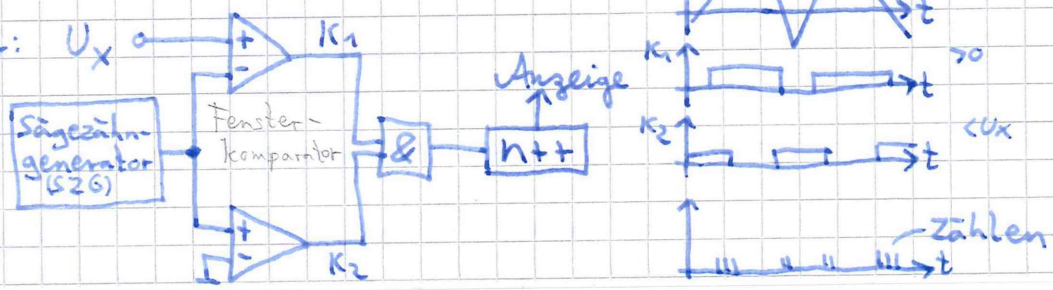
Schaltung



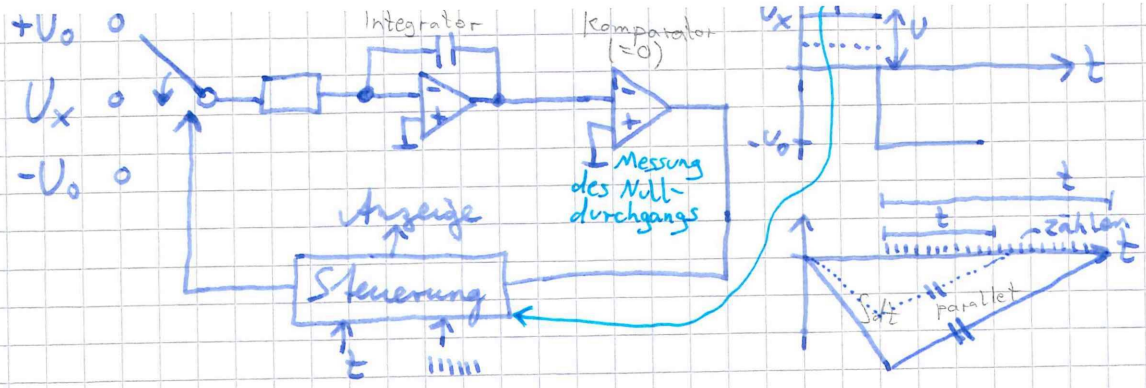


$T_s \sim U_x$

Sägezahnumsetzer:

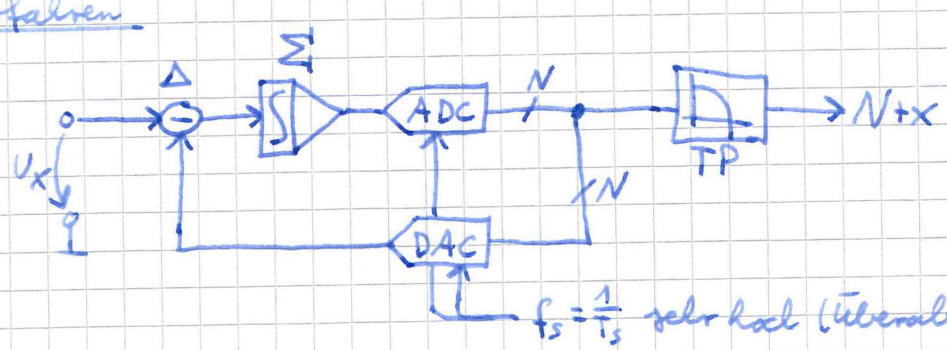


Dual Slope:



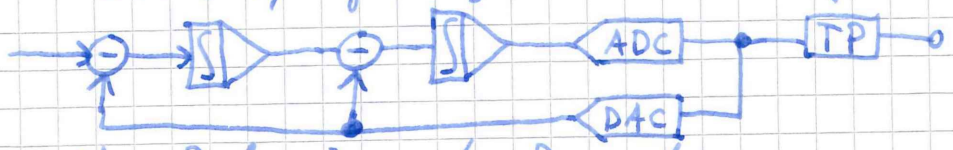
Sigma-Delta-Verfahren

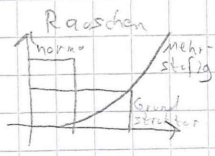
Grundstruktur:



Oversampling reduziert Quantisierungsrauschen

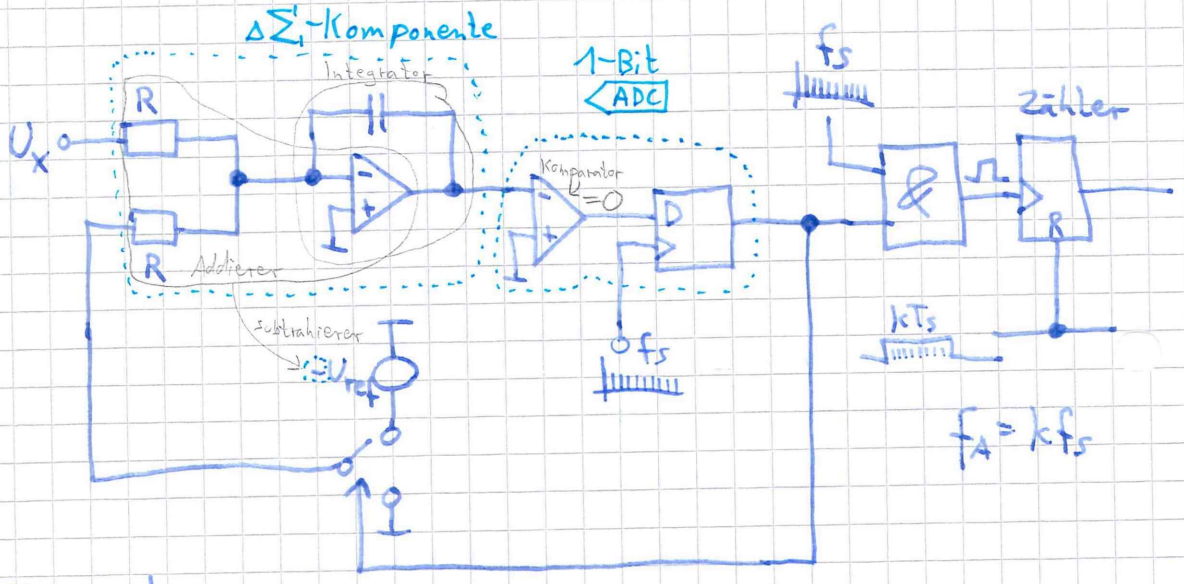
Mehrstufig:

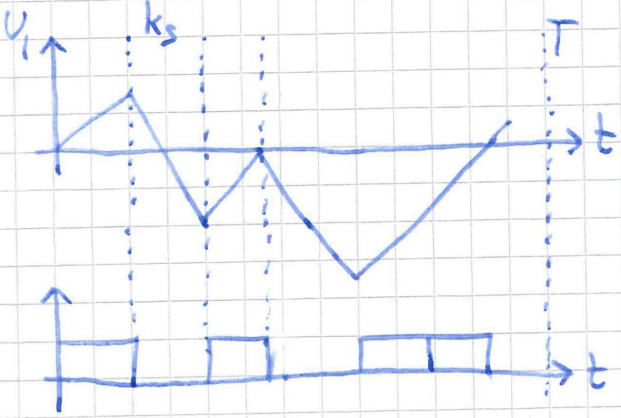




weitere Reduzierung des Rauschens

detailliertere:





Anstieg: Schalter auf GND
$0 + U_x$ → Addierer

Abfall: Schalter auf $-U_{ref}$
$-U_{ref} + U_x$
→ Subtrahierer

Durch das Oversampling wird der Wert über die Zeit abgetastet, es wird also bitweise und seriell gelesen.

Zusammenfassung

Kaskaden- und Parallelumsetzer: schnell
teuer (schwer herzustellen → aufwendig)
Einsatz in teuren und schnellen Messgeräten (Digital-Oszilloskope)

Zeitumsetzer (Dual-Slope): günstig
langsam
Einsatz in Voltmetern

Wägeverfahren: Hardware kann relativ einfach aufgebaut werden
• einfacher DA-Umsetzer
Messalgorithmus kann in Software realisiert werden
Einsatz im Mondflugprogramm (Apollo-Guidance Computer)

Delta-Sigma Verfahren: schnelle Schaltungen mit heutigen Transistoren möglich
vielfaches Oversampling ermöglicht rauscharme Umsetzung
gutes Verfahren um es dicht mit Sensoren auf einem Chip zu integrieren
• schnell mit wenig Hardwareaufwand

Programmierung eingebetteter Systeme

Einführung

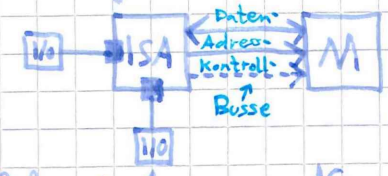
Anbindung der Hardware (Sensoren, Aktoren) an Software
Software für Informationsverarbeitung: leicht änderbar
sehr viele Funktionen möglich
→ flexibel

Programmierung in C oder Ada: Ada speziell designed für eingebette Systeme
↳ sicherer wegen zusätzlicher Typbegrenzung

→

Eingebettete Programmierung in C: Tolkel Loops

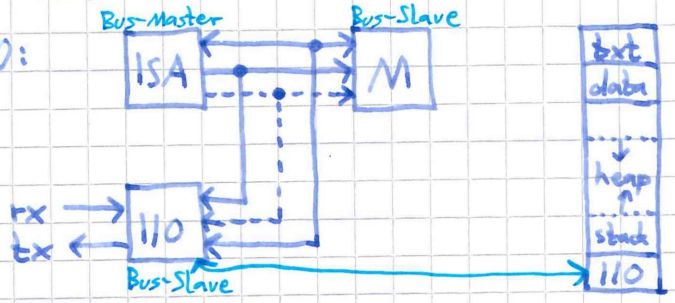
Port-mapped I/O:



spezielle Ein- und Ausgabe Befehle

klassische von Neumann-Architektur

Memory-mapped I/O:



fiktiver Bus-Slave

- STATUS Register: Number of Packets tx rx

  CTRL Register: Threshold tx rx, enable, CLK
  ↳ 00 empty
  01 1/4
  10 1/2
  11 full → "interrupt"

- rx Data

  tx Data

Programmierung in C

- Header:

```
sSlaveDev struct {
    UINT8 status;
    UINT8 ctrl;
    UINT8 tx;
    UINT8 rx;
};

#define NOP_TX   0xF0;
#define NOP_RX   0x0F;
#define FULL_TX  0xF0;
#define EMPTY_TX 0x00;
...
#define SUCCESS  0;
#define FAIL     1;
```

- main:

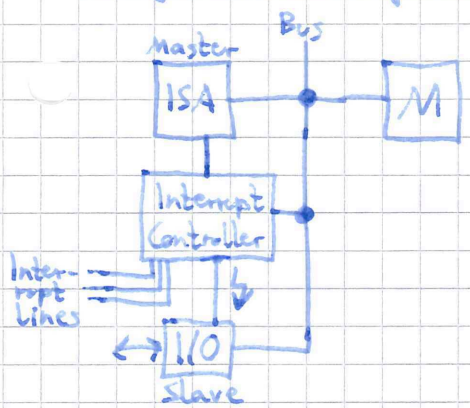
```
const int size = 100;
UINT8 rxBuf[size]; UINT8 txBuf[size];
sSlaveDev *devPtr = 0x8000;

int put (sSlaveDev *dev, UINT8 *txBuf){
  if(dev->status < FULL_TX){
    dev->tx = txBuf;
    return SUCCESS;
  } return FAIL;
}
```

```
int get (sSlaveDev *dev, UINT8 *rxBuf){
  if(dev->status > EMPTY_RX) {
    rxBuf = dev->rx;
    return SUCCESS;
  } return FAIL;
};

int write (sSlaveDev *dev, UINT8 *txBuf, int size){
  while (size > 0) {
    if (put (dev, txBuf)){
      return FAIL;
    } txBuff++;
      size--;
  } return SUCCESS;
};
int read (sSlaveDef *dev, UINT8 *rxBuf, int size){
  while (size > 0) {
    if (get(dev, rxBuf)){
      return FAIL;
    } rxBuf++;
      size--;
  } return SUCCESS;
};

void main () {
  while (true) {
    write();
    ...
    read();
    ...
    delay;
  }
}
```

never ending code

polled Loop
↳ rechenaufwändig

Eingebettete Programmierung in C: Unterbrechungen



Interrupt Controller: Tabelle mit mapping von Linie auf Adresse eines interrupt-Handlers, der ISR (interrupt service routine) ausführt

Programmierung in C: main:

```
void isr0 (){
  interruptDisable();
  //do something → foreground
  interruptEnable();
};

void main () {
  initInterruptHandler (*isr0, 0);
  for(;;){
    //background routine
  }
};
```

const UINT8 *isrPtr = &isr0
→ isrPtr

Eingebettete Programmierung in ADA: Gerätetreiber

Register:
```
type rtStatus is
   record
      rxNoP at 0 range 0..3;
      txNoP at 0 range 4..7;
   end record;
```

```
type rtControl is
   record
      clk at 0 range 0..2;
      rxth at 0 range 3..4;
      enable at 0 range 5..5;
      txth at 0 range 6..7;
   end record;
```

Slave Device:
```
type atBuffer is array (Integer range <>) of UINT;
subtype UINT is Integer range 0..255;

type rtSlaveDev is
   record
      status: rtStatus;
      ctrl: rtControl;
      rx: atBuffer;
      tx: atBuffer;
   end record;
```

I/O:
```
procedure get (dev: IN rtSlaveDev, rxValue: OUT UINT) is
   if dev.status.rxNoP > 0 then
      rxValue := dev.rx
   else exception;
end procedure;

function put (dev: IN rtSlaveDev, txValue: IN UINT) is
   if dev.status.txNoP > sizeBuf then
      dev.tx := txValue
   else exception;
end function;

procedure read( dev: in rtSlaveDev, rxBuf: out atBuffer, nopRead: out Integer) is
   for i in 0.. dev.status.rxNoP Loop
      put (dev, rxBuf(i));
   end Loop;
end procedure;
```

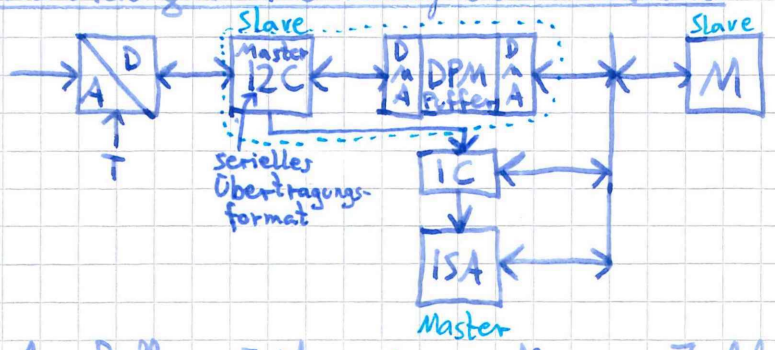
nopRead := dev.status.rxNoP

nopRead

Interrupt:
```
protected type
   ptDevInterface (IntID: ADA.interrupts.Interrupt_ID) is
      procedure interruptServiceRoutine;
      pragma Attach_Handler (interruptServiceRoutine, IntID);
   end ptDevInterface;

dev1: ptDevInterface (1)
```

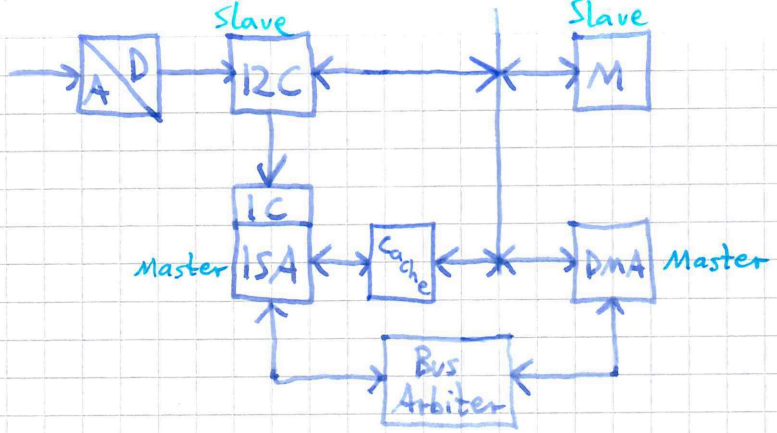
Architekturvarianten zur Anbindung von Sensoren

Single-Bus:

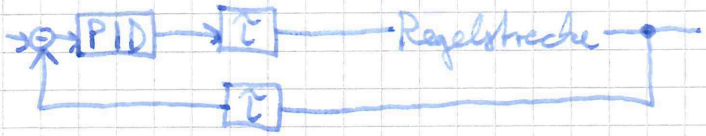


ohne Puffer müsste ein von Neumann Zyklus zeitlich < T sein

Multi-Bus:



LVDS:

Regler und Zeit: 

τ möglichst klein, damit Regler nicht instabil wird

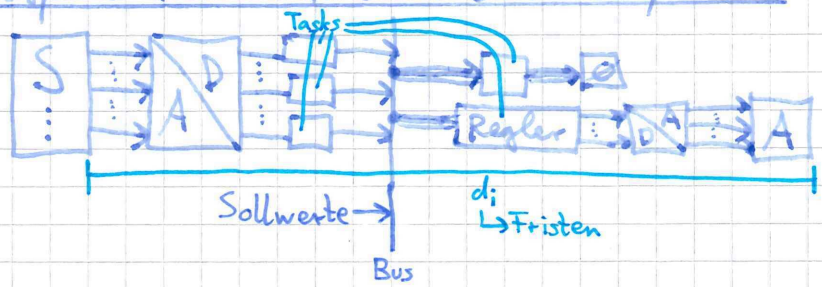
Echtzeitsysteme

Einführung

ständig müssen Werte von Sensoren gelesen werden und an Aktoren gegeben werden

Rechner muss immer auf externe Ereignisse reagieren

Beispiel: Funktion eines Bordcomputers



Dynamik hängt vom Zeitverhalten des Rechners ab

- schnellste verfügbare Rechner
- so schnell wie nötig → Fläche und Kosten minimieren

Zeit und Echtzeitsysteme

Zeit: logische Zeit

- Halbordnung von Anweisungen und daraus resultierenden Ereignissen: Ursache → Wirkung
- z.B. sequentiell ablaufendes Programm

Realzeit / Echtzeit

- unabhängig von Vorgängen im Rechner

abgetastete Zeit

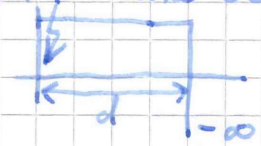
- Abtastung / Taktung der "Realzeit"

Ereignis: zeitloser Zustandswechsel zu einem Zeitpunkt

Echtzeitsystem: Ein eingebetteter Rechner, der durch externe Ereignisse ausgelöste Berechnungen rechtzeitig ausführt.

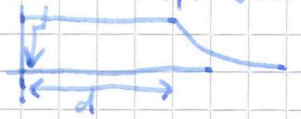
Das beinhaltet das Bereitstellen von Ergebnissen zu fest vorgegebenen Zeitpunkten, als auch die rechtzeitige Fertigstellung von Berechnungen.

hart: keine Echtzeitverletzung → katastrophale Folgen



z.B. ABS, Atomkraftwerk, Flugregler

weich: Zeitfristüberschreitung manchmal → unbequem



z.B. Streaming

Nebenläufigkeit in Echtzeitsystemen

time triggered System: alle T findet Abarbeitung statt

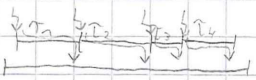
Verwendung: Datenstrom, statischer

Ereignis-gesteuertes System: wenn Ereignis eintritt findet Abarbeitung statt

Verwendung: sporadische Ereignisse, dynamisch

Nebenläufigkeit: sequentielle Abarbeitung

↳ keine Nebenläufigkeit

scheinbare Nebenläufigkeit 

↳ (zeitlich) unabhängige Aufgaben / Tasks, die scheinbar gleichzeitig ausgeführt werden

wahre Nebenläufigkeit

↳ Aufgaben können gleichzeitig ausgeführt werden

scheinbare Nebenläufigkeit: Ablaufplan legt Ausführungsreihenfolge der Aufgaben fest

→

statisches Echtzeitsystem: jeder programmierte Ablaufplan

dynamisches System: geregelter Ablaufplan, der flexibel auf Ereignisse reagieren kann

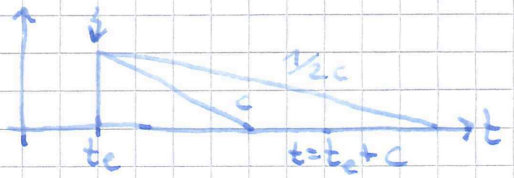
Ereignisse: unterbrechbar oder nicht unterbrechbar

Zum Festlegen der Ablaufreihenfolge erhalten Ereignisse Prioritäten: statisch (RMS/DMS,..

dynamisch (EDF/EDD,..

Ausführungsmodelle

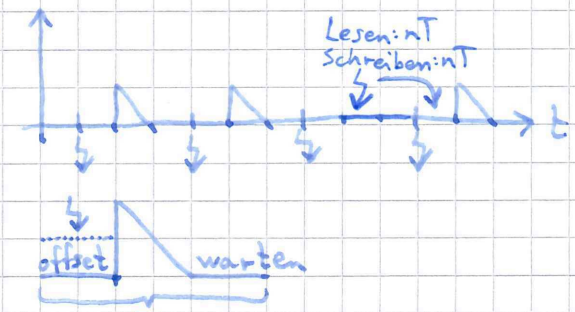
Physical execution time programming:



$t - t_e - c = 0$

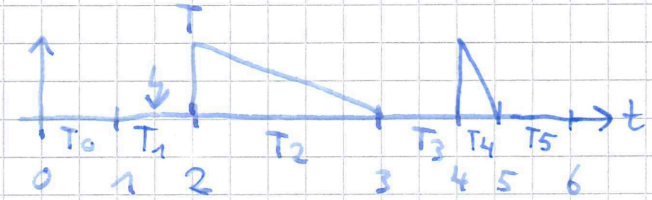
Realzeit — physikalische Zeit

logical execution time programming:

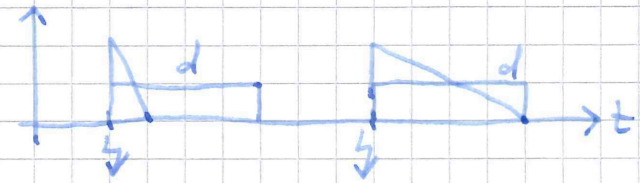


statische Ablaufpläne möglich

Zero execution time programming:



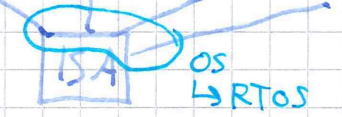
Bounded execution time programming:



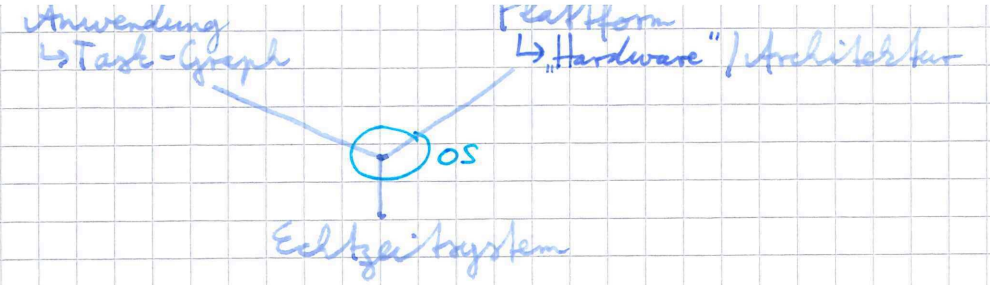
Ausführungsplan des scheinbar nebenläufigen Systems muss vorhersbar sein

Kombiniert portabilität mit PET

Echtzeitsystem: PET LET ZET BET



Nebenläufigkeitsmodelle werden durch Scheduler implementiert



Echtzeit-Betriebssystem (RTOS): geringe Interruptlatenz
vorhersehbar
→ Regel muss zu deterministischen Ablaufplan führen

Echtzeit-Betriebssysteme

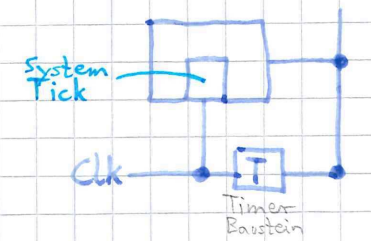
Einführung

Ada: Tasks, IPC, gegenseitiger Ausschluss
wesentliche Datenstruktur: Task-Kontroll-Block

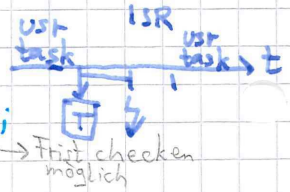
Aufgaben eines Echtzeitbetriebssystems

Kernel Implementierung eines RTOS
Nebenläufigkeit: unabhängiger Programmfluss → task
- process: program und eigener Speicher
- thread: program und globaler Speicher

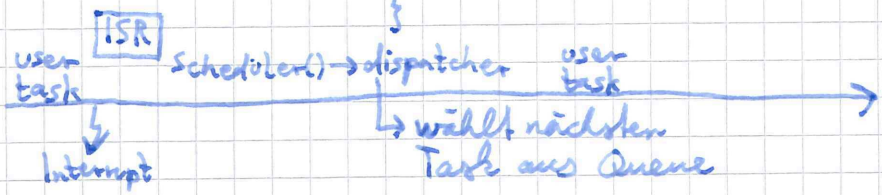
Prozessor Ressourcen verwalten: Zeit




```
isrTime(){
  disable();
  iSystemTick++;
  enable();
}
```

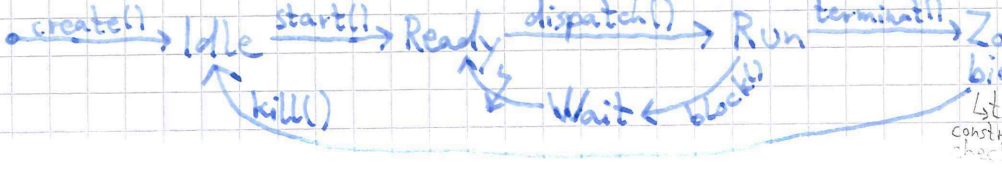



```
isrEvent(){
  disable();
  event := isr→status
  enable();
}
```



Ziel: kurze ISRs → geschachtelte Unterbrechungen / nested interrupts

Zustände eines Tasks:



Echtzeitkernel

Datenstruktur zur Repräsentation:

```
structure sTcb {                    TaskControlBlock
  integer: pid;
  sCtx: ctx;
  integer: state;
  ⋮ memory, priority, deadline
  pointer to nextTbc
  pointer to prevTbc
}

structure sCtx {                    ProzessorKontext
  char sReg[];
  char pc;
  char sp;
  ⋮
}
```

Organisation als Task Set: Array von 0 nach iMaxTask-1

jeder Eintrag ist Zeiger auf sTbc mit entsprechender pid

create() liefert pid zurück (=Position)

signal(pid, sig) über Array[pid] an sTbc

Statuslisten: pointer auf ersten sTbc im Array mit diesem Status

ready-liste bearbeiten

↳ Round-Robin

↳ Earliest Deadline First → D (nicht d)

Methoden / Funktionen

- create(), schedule(), dispatch(), suspend()
- read(), write(), send() → IPC

Organisation mit Priority level: Array von 0 nach iMaxPrio-1

jeder Eintrag ist Zeiger auf sTbc liste mit entsprechender Priorität

an Stelle iMaxPrio-1 befindet sich der Idle-Task

listen selber Priorität bearbeiten

↳ Round Robin

↳ Earliest Deadline First

aktuellen Task mit höchster Prio finden

↳ Array der Länge (iMaxPrio - 1)/w

0 → kein Task mit dieser Prio

1 → Task mit dieser Prio

→ führende 0-en zählen

Signalverwaltung (Interrupt, IPC)

- Organisation als Ringpuffer

  Verwaltung als Mailbox

  ↳ Mailbox → pid

  zweite Tabelle

  ↳ pid → Mailbox

- Tabelle: isrNr → pid

  Tabelle: pid → send (pid)

Schreiben und lesen eines Ringpuffers

Struktur im Speicher: Array von min nach max mit Pointer auf Head und Tail Position

Anforderungen: atomare Funktionen für lesen und schreiben

kein Interrupts während lesen / schreiben

Sperre:

```
boolean Lock (boolean semaphore) {
    int v;
    disableInterrupt();
    if (semaphore)  v:=LOCKED;
    else {semaphore:=true; v:=PASS;}
    enableInterrupt();
    return v;
}
```

```
boolean Lock(boolean semaphore){
    disableInterrupt();
    semaphore:=false;
    enableInterrupt();
}
```

I/O:

```
int write (UINT data, UINT *buffer){
    if (Lock(buffer->semaphore))
        return LOCKED;
    if (++buffer->head == buffer->tail)
        buffer->head--;
        return FULL;
    if (buffer->head > buffer->max)
        buffer->head = buffer->min
    &buffer->head = data;
    unlock (buffer->semaphore);
    return WRITE;
}
```

```
int read (UINT data, UINT* buffer) {
    if (buffer->tail == buffer->head)
        return EMPTY;
    data := &buffer->tail++;
    if (buffer->tail == buffer->
        buffer->tail = buffer->
    return READ;
}
```

↳ kein Lock, da beim lesen normal nichts schief geht, aber könnte man noch machen

Tasks in AD 1

Datentypen:
```
package datatype is
   type sampledValue is new Natural range 0..255;
   type surfaceCtrl is new Integer range -10..10;
   type aVector is array (1 to 3) of Integer;
   type inertial record is
      acceleration: aVector;
      speed: aVector;
      distance: aVector;
      position: aVector;
      angularAcceleration: aVector;
      angularSpeed: aVector;
      angularW: aVector;
   end inertial;
   type sFlightPosition record is
      angleOfAttack: Integer range -30..30;
      airSpeed: Integer range 0..1000;
   end sFlightPosition;
   type sSurfaceCtrl record is
      right: surfaceCtrlAngle;
      left: surfaceCtrlAngle;
   end sSurfaceCtrl;
   type sActor record is
      throttle: sThrottle;
      rudder: SurfaceCtrlAngle;
      evaluator: sSurfaceCtrl;
      ailivion: sSurfaceCtrl;
      flaps: sSurfaceCtrl;
      stats: sSurfaceCtrl;
   end sActors;
   type airSpeed is new Natural;
   type position is new aVector;
end dataTypes;
```

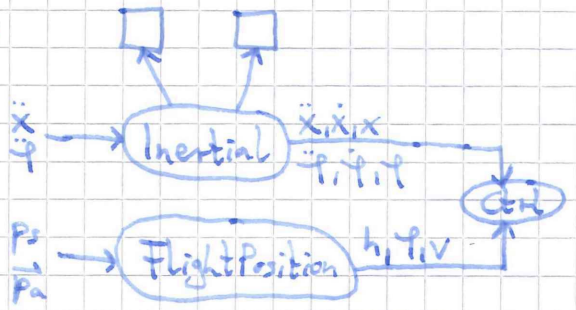
I/O:
```
use dataTypes use dataTypes;
package IO is
   read(accelaration: out aVector);
   read(angleAccelaration: out aVector);
   read(staticPreassure: sampledValue);
   read(dynamicPressure: sampledValue);
end IO;
protected type boundedBuffer is  -> IPC
   entry send(Item1: in sInertial);
   entry send(ItemF: in sFlightPosition);
   entry send(Item1: in sInertial, ItemF: in sFlightPosition);
   entry receive(..)
   :
end boundedBuffer;
protected type pSamples is
   entry put(sample: in sampledValue);
   entry set(sampleTime: Natural);
   entry integrate(sum: out Integer, intervall: in Integer);
   entry derive(d: out Integer, intervall: in Integer);
   private
      typesampleBuffer is new sampledValue range 0..127;
end pSample
```

→

Controller:

```
with... use...
package PlaneCtrl is
   task Inertial;
   task Navigation;
   task FlightPosition;
   task FlightCtrl;
end planeCtrl;
package body planeCtrl is
   inertial: sInertial;
   flightPosition: sFlightPosition;
   aBuffer: sampleBuffer;
   bBuffer: sampleBuffer;
task body Inertial is
   loop
      read (inertial.acceleration);
      read (inertial.angularAcceleration);
      aBuffer.put (acceleration);
      bBuffer.put (angularAcceleration);
      aBuffer.integrate (inertial.Speed, inertial.acceleration, T);
      vBuffer.integrate (inertial.distance, inertial.speed);
      send (ctrl, inertial);
   end loop;
end Inertial;
task body FlightPosition is
   read (staticPressure, dynamicPressure);
   angleOfAttack();
   height();
   airspeed();
   send (ctrl, FlightPosition);
end FlightPosition;
```

Zusammenfassung:



FlightControll:

```
task body FlightCtrl is
   loop
      receive (inertial, FlightPosition);
      actor = control(inertial, Flightposition);
      write (actor);
   end loop;
end FlightCtrl;
```

Ergebnis: Ada ist Programmiersprache → kein RTOS notwendig