

Zusammenfassung Echtzeitsysteme

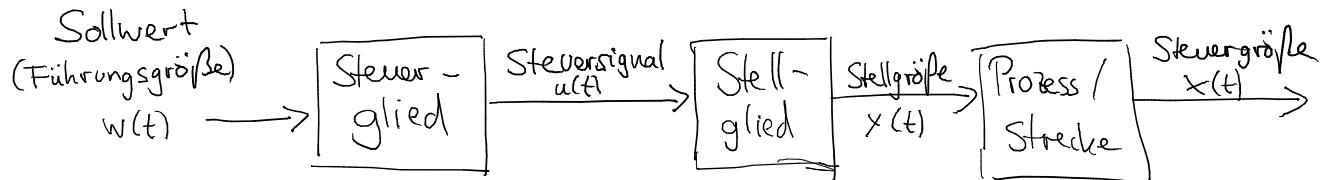
Gehört am Karlsruher Institut
für Technologie im Sommersemester 2016

Orientiert an Vorlesung, Übung und den
Altklausuren ab Sommersemester 2011

von Sarah Lutteropp

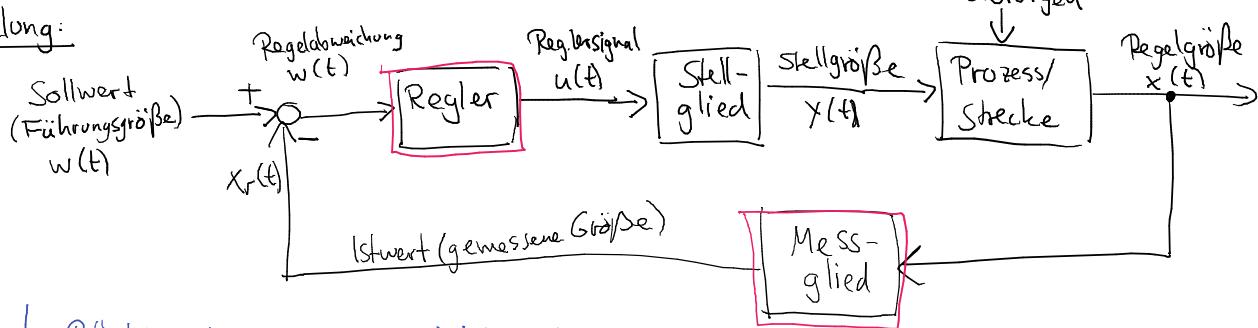
Aufgabe 1.1 : Zeitkontinuierliche Regelung

Steuerung:



↳ Es muss kein Ist-Wert erfasst werden → kein Sensor benötigt

Regelung:

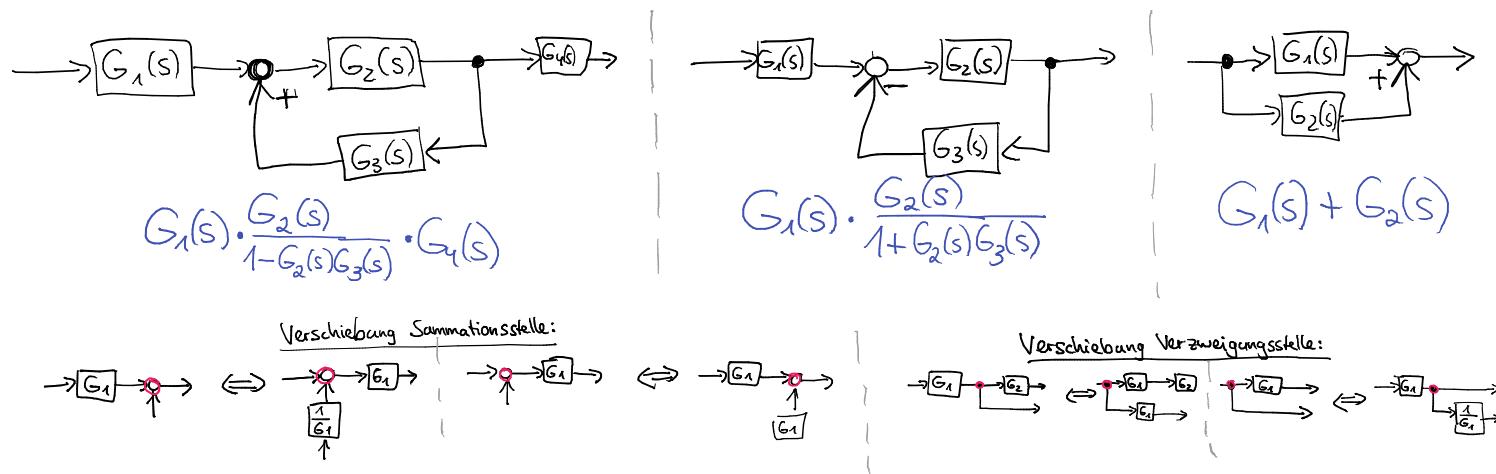


↳ Rückkopplung, erfasst Ist-Wert, braucht Sensor

Gütekriterien Regelungssystem: Stabilität, Schnelligkeit, Genauigkeit

LTI-System: Linear, zeitinvariant. Sprungantwort (Ausgangsgröße des Systems bei sprungförmiger Änderung der Eingangsgröße) genügt, um ein LTI-System vollständig zu charakterisieren

Übertragungsfkt. des Gesamtsystems im Laplace-Bereich:



Übertragungsfunktion mit Laplace-Transformations-tabelle aus Gleichungen:

$x(t) \mapsto X(s)$ $w(t) \mapsto W(s)$, Konstante Faktoren bleiben Beispiel: $5\ddot{x}(t) = 5s^2X(s)$ (falls $\dot{x}(t) = x(t) = 0$)

① Gleichung Laplace-Transformieren

② $W(s)$ auf eine Seite bringen

③ $G(s) = \frac{X(s)}{W(s)}$ Das $X(s)$ kürzt sich weg.

Beispiel: $\ddot{x}(t) = 2w(t) + w(t)$, $x(0) = 0$, $\dot{x}(0) = 0$, $w(0) = 0$

$$\Leftrightarrow s^2X(s) = 2sW(s) + W(s) \rightarrow (2s+1)W(s) = s^2X(s) \rightarrow W(s) = \frac{s^2}{2s+1} \cdot X(s)$$

$$G(s) = \frac{X(s)}{W(s)} = \frac{\cancel{s^2}X(s)}{\cancel{s^2}W(s)} = \frac{2s+1}{s^2}$$

Ausgangssignal $x(t)$ im Zeitbereich:

$$u(t) \rightarrow g_1(t) \rightarrow g_2(t) \rightarrow x(t)$$

$$x(t) = \int_0^t h(t-\tau) \cdot u(\tau) d\tau,$$

$$\text{wobei } h(t) = \int_0^t g_1(t-\tau) \cdot g_2(\tau) d\tau \text{ ist.}$$

Stabilitätskriterien Übertragungsfunktion (kontinuierliches LTI-System)

① Hurwitz-Kriterium: Alle Nennerpolynom-Koeffizienten > 0 (bei 3. Ableitung zusätzlich $a_2 \cdot a_1 - a_0 > 0$ testen)

Gegeben LTI-System, durch DGL $a_n x^{(n)} + a_{n-1} x^{(n-1)} + \dots + a_1 x + a_0 = 0$ beschrieben.

Hurwitz-Matrix:

$$H = \begin{pmatrix} a_{n-1} & a_{n-2} & \dots & 0 \\ a_n & a_{n-1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_0 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

Das System ist asymptotisch stabil, wenn

- $a_i > 0$ für $i=0 \dots n$
- alle Hauptabschnittsdeterminanten von H positiv

Beispiel: $G(s) = \frac{s-1}{s^2-2s+2}$ nicht asymptotisch stabil, da einer der Nennerpolynom-Koeffizienten ≤ 0 .

② Polstellenkriterium: Das System ist asymptotisch stabil, wenn der Realteil aller Pole der Übertragungsfunktion negativ ist.

Beispiel: $G(s) = \frac{s-1}{(s-2)(s+0,5)}$

→ Polstellen bei $+ \frac{2}{3}$ und $-0,5$

→ nicht asymptotisch stabil, da mindestens eine Polstelle ≥ 0 .

Bode-Diagramm:

Stellt den Frequenzgang $G(i\omega)$ eines Systems graphisch dar. Dabei wird in einem Graph der Amplitudengang $|G(i\omega)|$ über der Kreisfrequenz ω aufgetragen. (beide Achsen logarithmisch skaliert) In einem zweiten Graph wird der Phasengang $\arg(G(i\omega))$ über der Kreisfrequenz ω aufgetragen (nur ω -Achse logarithmisch skaliert)

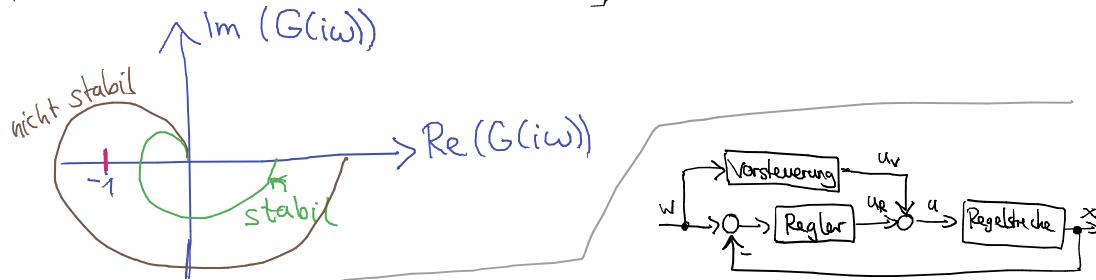
Stabilität Bode-Diagramm: Bei der Frequenz, an der die Betragskennlinie 10^0 schneidet, ist die Phase größer als -180° (Phasenreserve) \rightarrow stabil

Stabilität Ortskurve: (Nachteil: macht Frequenzabhängigkeit nicht sichtbar)

Nyquist-Kriterium: Stabil, falls die Kurve den Punkt $(-1, 0i)$ nicht umfährt.

(\Leftrightarrow Stabil, wenn -1 links von der Ortskurve)

Beispiel:



Vorsteuerung: Bei einer Vorsteuerung wird ein Referenzwert anhand eines Modells generiert, welches keine Messfehler einbezieht. Dieser Referenzwert wird von einer Regelung überlagert um die Führungsgröße, trotz Wirkung von Störungen und Vorhandensein von Messfehlern, möglichst genau einzubehalten.

Adaptiver Regler: Adaptive Regelsysteme sind Systeme, bei denen ein adaptiver Regler seine Parameter → oder auch seine Reglerstruktur (z.B. P- zu D-Regler) der sich ändernden Regelstrecke entsprechend anpasst (adaptiert).

PID-Regler

P

- verändert das Steuersignal proportional zur Regelabweichung
- Strategie: Je größer die Regelabweichung, umso größer muss die Stellgröße sein
- durch den Verstärkungsfaktor K_p kann die Regelgeschwindigkeit eingestellt werden (je höher, desto schneller)
- ein hoher Verstärkungsfaktor kann zu Instabilität des Regelkreises (Schwingungen) führen.
- ein P-Glied allein kann die Regelabweichung nicht vollständig auf 0 ausregeln.

- integriert die Regelabweichung, sodass bei konstanter Regelabweichung das Ausgangssignal des Reglers stetig ansteigt.
- Strategie: Solange eine Regelabweichung auftritt, muss die Stellgröße verändert werden.
- Bei einem I-Glied wird die Regelabweichung immer ausgeregelt.
- I-Glieder führen bei Regelkreisen leicht zu Instabilitäten.

I

- differenziert die Regelabweichung
- durch die Betrachtung der Änderung des Signals wird ein zukünftiger Trend berücksichtigt.
- Strategie: Je stärker die Änderung der Regelabweichung ist, desto stärker muss das Stellsignal verändert werden.
- D-Glieder verbessern gewöhnlich die Regelgeschwindigkeit d. die dynamische Regelabweichung.
- D-Glieder verstärken besonders hochfrequente (verrausche) Anteile des Eingangssignals. Dies erhöht die Neigung zu Schwingungen.

D

Das geregelte System reagiert sehr langsam: P-Anteil erhöhen, D-Anteil erhöhen

Das geregelte System ist instabil: P-Anteil reduzieren, I-Anteil reduzieren, D-Anteil erhöhen

Das geregelte System reagiert anfangs schnell, vor Erreichen der Sollgröße bremsst es ab und kriecht nur noch langsam ans Ziel: I-Anteil erhöhen, D-Anteil reduzieren

Das geregelte System schwingt stark in der Sprungantwort: D-Anteil erhöhen

Das geregelte System schwingt stark bei verrostetem Eingangssignal: D-Anteil reduzieren

Aufgabe 1.2: Zeitdiskrete Regelung

Abtasttheorem: Ein Signal muss mit mindestens der doppelten Maximalfrequenz des Signals abgetastet werden, damit dieses verlustfrei erfasst wird. $f_{\text{Abtast}} \geq 2 \cdot f_{\max}$

↳ Abtastperiode $T_A = \frac{1}{f_{\text{Abtast}}}$

$$\begin{aligned} ms &= 10^{-3} s \\ \mu s &= 10^{-6} s \\ \eta s &= 10^{-9} s \end{aligned}$$

$$\begin{aligned} kHz &= 10^3 \text{ Hz} \\ MHz &= 10^6 \text{ Hz} \\ GHz &= 10^9 \text{ Hz} \end{aligned}$$

↳ falls Eingangssignal zu hochfrequent: Filtern des Eingangssignals nötig.

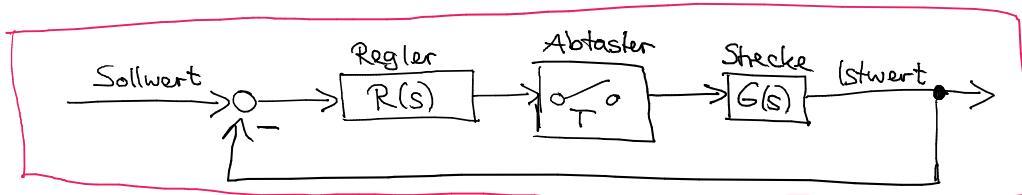
Aliasing-Effekt: Entsteht, wenn ein Signal höhere Frequenzen als $\frac{1}{2} \cdot$ Abtastfrequenz enthält. Dann kann das ursprüngliche Signal nicht aus den Abtastwerten korrekt rekonstruiert werden, die höheren Frequenzen werden als niedrigere interpretiert. Kann während der Messung nicht festgestellt werden → vorher überprüfen!

Digitaler Regler vs. analoger (kontinuierlicher) Regler:

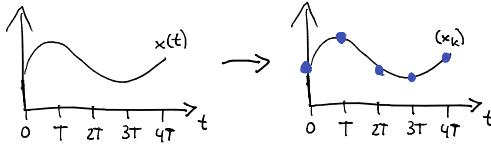
↳ zusätzliche Elemente: A/D-Wandler, D/A-Wandler (ggf. mit Halteglied)

- ↳ Vorteile: 1) Mit der gleichen Hardware ist es möglich unterschiedliche Algorithmen zu rechnen
- 2) Hohe Flexibilität bei Änderungen

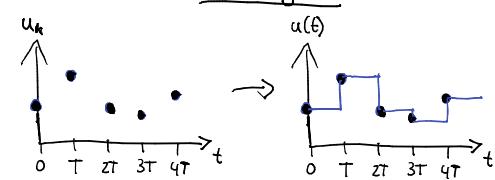
Zeitdiskreter Regelkreis (Blockschaltbild):



Abtastglied:



Halteglied:



Z-transformierte Übertragungsfunktion $G(z)$ der zeitdiskreten Regelstrecke aus zeitkontinuierlicher Regelstrecke $G_s(s)$ bestimmen: (mit Abtastperiode T_A gegeben)

Hinweis aus Aufgabenstellung benutzen, und wissen: $\mathcal{Z}[s] = \frac{z-1}{z}$

$$\mathcal{Z}[s \cdot f_{\text{abt}}] = \mathcal{Z}[s] \cdot \mathcal{Z}[f_{\text{abt}}]$$

Oft nützlich: Binomische Formeln
 $(a+b)^2 = a^2 + 2ab + b^2$
 $(a-b)^2 = a^2 - 2ab + b^2$
 $(a+b)(a-b) = a^2 - b^2$

Differentialgleichung in Differenzengleichung umwandeln: (Abtastperiode ist T_A)

$$x(t) \mapsto x_k, w(t) \mapsto w_k, t \mapsto k \cdot T_A$$

$$\dot{x}(t) \mapsto \frac{x_k - x_{k-1}}{T_A}, \ddot{x}(t) \mapsto \frac{x_k - 2x_{k-1} + x_{k-2}}{T_A^2}$$

Beispiel: $\ddot{x}(t) + \dot{x}(t) - t \cdot x(t) = w(t)$

$$\Leftrightarrow \frac{x_k - 2x_{k-1} + x_{k-2}}{T_A^2} + \frac{x_k - x_{k-1}}{T_A} - k \cdot T_A \cdot x_k = w_k$$

Z-transformierte Übertragungsfunktion aus Differenzengleichung ermitteln: (Abtastperiode ist T_A)

$$x_k \mapsto X(z), w_k \mapsto W(z), X_{k-n} \mapsto z^{-n} \cdot X(z)$$

(Rechtsverschiebung)

Hinweise aus Aufgabenstellung benutzen, $G(z) = \frac{X(z)}{W(z)}$

Stabilitätskriterium für zeitdiskrete Systeme: Zeitdiskretes System ist asymptotisch stabil, wenn die Beträge aller seiner Polstellen kleiner als 1 sind. $|p_i| < 1 \quad \forall \text{Polstellen } p_i$

Übertragungsfunktion aus Pol-Nullstellen-Diagramm:

Beispiel: Nullstelle bei $-1,5$; Polstellen bei $2; -0,5 \pm 0,5j$

Pole heißt: Nennerpolynom wird Null.

$$\rightarrow G(z) = \frac{(z+1,5)}{(z-2) \cdot (z^2 + p_2 z + q)}$$

p-q-Formel:

$$x^2 + px + q$$

$$\Rightarrow \text{Nullstellen bei } x_{1/2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

$$(0,5)^2 = \frac{1}{4} \rightarrow \left(\frac{1}{2}\right)^2 - q = \frac{1}{4} \rightarrow q = \frac{1}{4}$$

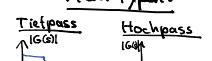
$$\hookrightarrow \text{hier: } p=1, q=0,5$$



2-transformierte Übertragungsfunktion PID-Regler:
 $R_{\text{PID}}(z) = R_p(z) + R_i(z) + R_d(z)$

$$R_p(z) = K_p \frac{z - 1}{z - 1}, \quad R_i(z) = K_p \frac{T_A}{z - 1}, \quad R_d(z) = K_p \frac{T_A}{T} \frac{z - 1}{z - 1}$$

Filtertypen:



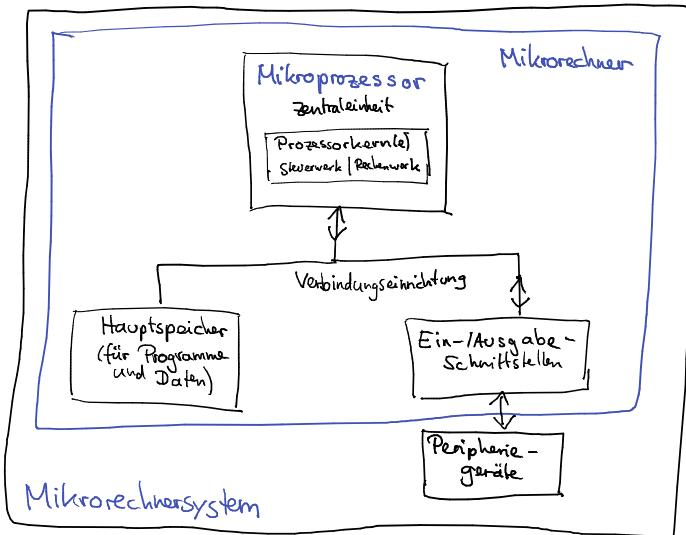
Aufgabe 2.1: Rechnerarchitekturen, Busse

Mikroprozessor: Zentraleinheit (CPU) eines Mikrorechners

Mikrocontroller: Einfacher Ein-Chip-Mikrorechner mit speziell für Steuerungsaufgaben zugeschnittener Peripherie

Mikrorechner: Mikroprozessor + Hauptspeicher (für Programme und Daten) + Verbindungseinrichtung + I/O-Schnittstelle

- Beschleunigungstechniken:
- 1) Pipelining erhöht Durchsatz
 - 2) Sprungvoraussage
 - 3) Speicherhierarchie



Eigenschaften von Signalprozessoren:

- konsequente Harvard-Architektur
- Hochleistungsarithmetik, optimiert für aufeinanderfolgende Multiplikationen und Additionen
- hohe, benutzerkontrollierbare Parallelität

Echtzeit-Ausgabeeinheit:

- Beseitigt unerwünschte Eigenschaft einer herkömmlichen Ausgabeeinheit: **Jitter** (Zeitschwankung zwischen den Ausgaben)
- zusätzliche Komponenten:
 - Pufferregister** (Laden Ausgangswert in Register, Inhalt/Zeitpunkt softwarebestimmt)
 - Zeitgeber** (Ausgabe der Daten, Zeitpunkt durch Hardwaretimer)

Watchdog: 🐺 Spezieller Zeitgeber (Timer) zur Programmüberwachung

- ↳ Erkennen von Software-/Systemfehlern: Absturz, Verklemmung
- ↳ Ausführung von Gegenmaßnahmen: Reset des Systems/Prozessors (=Programmneustart)

↳ Programm muss regelmäßig ein Lebenszeichen (Zähler des Watchdogs auf Startwert zurücksetzen) an den Watchdog senden, sonst wird es zurückgesetzt

Nulldurchgangszeit: $T = \text{Startwert} \cdot \text{Zykluszeit} = \text{Startwert} \cdot \frac{1}{\text{Refraktärzeit}}$

Wahl des Startwerts kritisch: System-Reset-Zeit zu kurz: Programme müssen ständig den Timer/Startwert zurücksetzen
System-Reset-Zeit zu lang: System zu lange inaktiv

Pipelining: Steigert den Befehlsdurchsatz

Ansatz: ① Unterteilung der Befehlsverarbeitung in Sequenz von Pipelinestufen

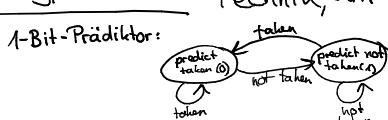
② Pipelinestufen arbeiten zeitlich parallel → mehrere Befehle können gleichzeitig in der Pipeline sein
Parallele Ausführung unterschiedlicher Bearbeitungsschritte

Skalärer Prozessor: Prozessor, welcher einen Befehl pro Takt verarbeiten kann

Superskalärer Prozessor: Prozessor, welcher mehr als einen Befehl pro Takt verarbeiten kann

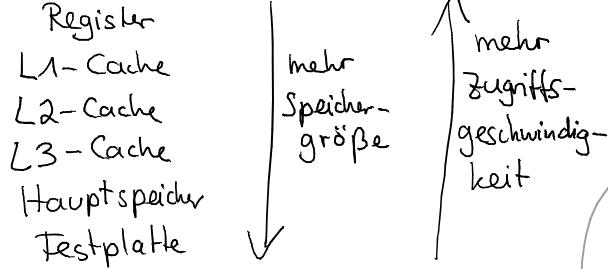
↳ Ausnutzung räumlicher Parallelität: Vervielfältigung von Verarbeitungseinheiten der Pipeline
Parallele Ausführung des selben Bearbeitungsschritts

Sprungprädiktion: Technik, um Hemmnisse in Bezug auf Sprungbedingungen aufzulösen

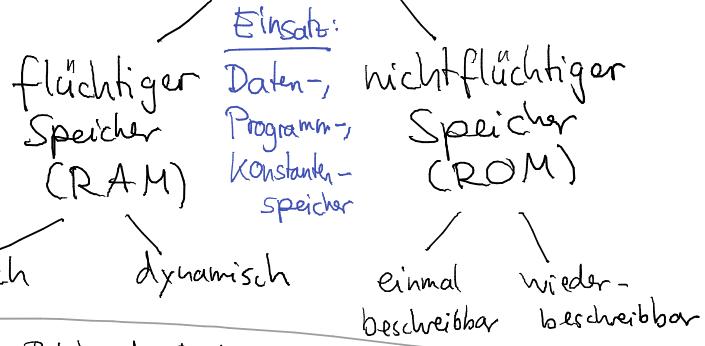


2-Bit-Prädiktor: Nach 3 Sprüngen sagt ein Zwei-Bit-Prädiktor auf jeden Fall einen Sprung voraus.

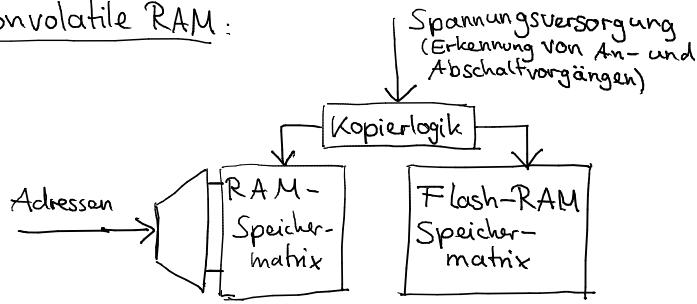
Speicherhierarchie:



Speicherxpen für Mikrocontroller



Nonvolatile RAM:



- Normaler Betriebsmodus: RAM-Speichermatrix wird benutzt.
 - Bei Abschalten der Spannungsversorgung: Kopieren der RAM-Speichermatrix in die Flash-RAM-Speichermatrix
 - Bei Wiedereinschalten der Spannungsversorgung: Kopieren der Flash-RAM-Speichermatrix in die RAM-Speichermatrix
- Vorteile:
- 1) Verhält sich im Betrieb wie ein normales RAM
 - 2) robust gegen Stromausfälle
 - 3) Anzahl von Schreib-/Lesezyklen auf das Flash-RAM wird reduziert → erhöhte Lebensdauer

Nonvolatile RAM ist flüchtiger Speicher, welcher seinen Speicherinhalt in einen nichtflüchtigen Speicher kopiert, damit dieser z.B. nach einem Stromausfall noch vorhanden ist.

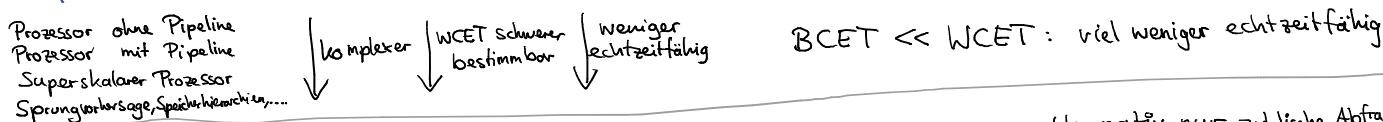
DMA (Direct Memory Access): Daten direkt ohne Beteiligung des Prozessorkerns zwischen Peripherie und Speicher transportieren

Wichtige Parameter für Echtzeitverhalten:

- ↳ Worst-Case Execution Time (WCET): Maximale Ausführungszeit eines Programms (wichtigster Parameter für Echtzeitssysteme, harke Zeitbedingungen)
- ↳ Best-Case Execution Time (BCET): Minimale Ausführungszeit eines Programms
- ↳ WCET/BCET: Maß für die Schwankung der Ausführungszeit eines Programms

→ Je komplexer der Prozessor, desto schwieriger ist es die maximale Ausführungszeit zu bestimmen
Keine oder geringe Beschleunigung des Echtzeitssystems durch komplexe Ansätze (WCET muss oft angenommen werden, WCET gleich lang wie in einfachen Prozessoren ohne Pipeline)

→ Je komplexer der Prozessor, desto weniger geeignet



Unterbrechungen: System kann auf (unvorhergesehene) Ereignisse reagieren, alternativ nur zyklische Abfrage (Polling) möglich
↳ zyklisches Abfragen ob etwas passiert ist ⇔ einmaliges Berichten, dass etwas passiert ist

Unterbrechungsarten:

- ↳ Intern: Exception (Fehlersituationen im Prozessorkern)
- ↳ Software Interrupt (Unterbrechungswunsch des laufenden Programms)
- ↳ Extern: Hardware Interrupt (Unterbrechungswunsch einer externen Systemkomponente)

Reaktionsmöglichkeiten auf eine Unterbrechung:

Taskabbruch (Auftreten eines schwerwiegenden Fehlers im Programmablauf → Ausführung Fehlerbehandlungsprogramm)

Taskwechsel (Unterbrechungsbehandlung mit höherer Priorität verdrängt das Programm.
Nach Behandlung der Unterbrechung wird das Programm fortgesetzt.)

Kein Taskwechsel (Aktuelles Programm hat höhere Priorität als die Unterbrechungsbehandlung (ISR).
→ Unterbrechungsbehandlung wird zurückgestellt)

Eigenschaften einer Unterbrechungsbehandlung:

Vektorisierung (Unterbrechungsquelle übermittelt Vektornummer zur eindeutigen Identifizierung)

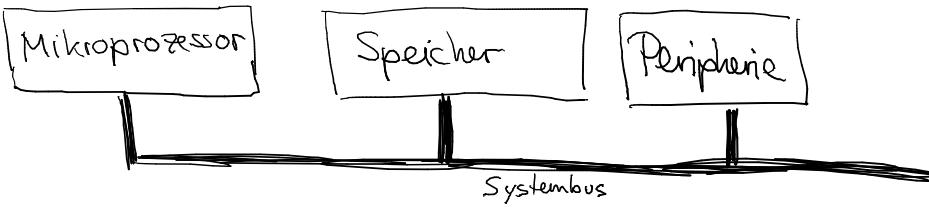
Maskierbarkeit (Sperrung einer bestimmten Unterbrechungsquelle. Dort auftretende Ereignisse werden zwar entgegengenommen, aber für die Dauer der Maskierung zurückgestellt)

Priorität (Zur Behandlung von Ereignissen unterschiedlicher Dringlichkeit stehen Prioritäten zur Verfügung. Unterbrechungen mit höherer Priorität können solche mit niedrigerer vorrangig.)

Externe dezentrale Prioritätssteuerung (Daisy Chain):
④: einfacher Aufbau; prinzipiell beliebig viele Teilnehmer
⑤: starre Prioritätsvorgabe; „Auslagerung“ der Quellen am Kettende; Zeitbedarf bis zur Ablösung der Unterbrechung wächst mit der Kettenlänge

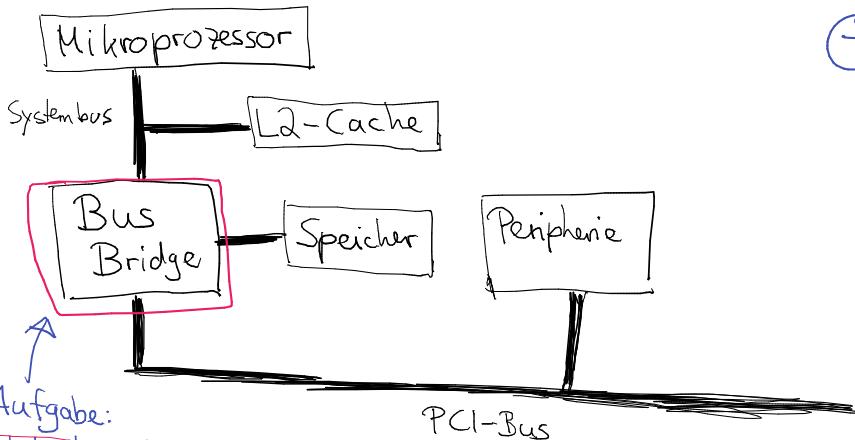
Externe zentrale Prioritätssteuerung (externer Interrupt-Controller):
⑥: Freie Vergabe von Prioritäten; hohe Flexibilität;
dynamisch konfigurierbar (zur Laufzeit); einfacher Aufbau der Unterbrechungsquellen: Interrupt-Vektor wird nicht von der Quelle übermittelt

Prozessor abhängiges Bussystem



- ⊕: kostengünstig, schnell
- ⊖: unflexibel, da auf einen Prozessortyp festgelegt

Prozessor unabhängiges Bussystem



- ⊕: hohe Flexibilität
 - ↳ System kann modular aufgebaut werden → Steckkartenkonzepte
 - ↳ bei einem Prozessorwechsel muss nur die Bus-Brücke ausgetauscht werden
- ⊖: erhöhter Hardwareaufwand

Aufgabe:

Protokollumsetzung

↳ Prozessor muss Busprotokoll nicht sprechen und funktioniert somit universell auf mehreren Systemen mit unterschiedlichen Busstandards

Signalleitungen eines synchronen Busses:

In Adressleitungen, werden vom Master gesetzt

in Datenleitungen, beim Schreiben vom Master gesetzt, beim Lesen vom Slave gesetzt

R/W : Read/Write: wird vom Master gesetzt (Signal auf (HIGH) bei Lesezugriff, Signal auf (LOW) bei Schreibzugriff)

READY : wird vom Slave gesetzt (Signal auf (LOW) → fertig. Bei Schreibzugriff: Daten wurden vom Slave übernommen
Bei Lesezugriff: Daten wurden vom Slave auf den Bus gelegt)

Signalleitungen eines asynchronen Busses:

Adressleitungen, Datenleitungen, R/W : wie beim synchronen Bus

AS : Adress strobe, wird vom Master gesetzt. (Signal auf (LOW) → Adresse ist gültig)

↳ beim synchronen Bus wird die Adresse an fest vorgeschriebenen Taktanfällen übernommen. Dieses Signal wird daher dort nicht benötigt.

DTACK : Data acknowledge: wird vom Slave gesetzt (Signal auf (LOW) → fertig. Bei Schreibzugriff: Daten wurden vom Slave übernommen)

↳ beim synchronen Bus werden die Daten, ebenso wie die Adresse an fest vorgegebenen Taktanfällen übernommen.
Bei Lesezugriff: Daten wurden vom Slave auf den Bus gelegt
Ein Zugriff kann dort allerdings verzögert werden, indem READY auf (HIGH) gesetzt wird. Insofern sind beide Signale „ähnlich“.

Bei einem asynchronen Bus gibt es keine Wartezyklen, da es keinen gemeinsamen Takt gibt.
Hier wird der Abschluss der Transaktion durch Handshake-Signale asynchron angezeigt (z.B. DTACK).

Bus-Arbiter: Verfahren zur Zuteilung des Systembusses an jeweils einen Master

Externer Bus-Arbiter:

Zentraler Bus-Arbiter: Jeder Master wird mit dem selben Arbiter verbunden

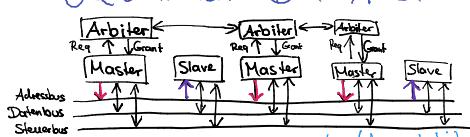
⊕: einfacher Systemaufbau, beliebige und schnelle Priorisierung

⊖: hoher Leitungsaufwand

Dezentraler Bus-Arbiter: Jeder Master hat einen eigenen Arbiter, welche miteinander kommunizieren

⊕: geringe Leitungszahl

⊖: starke Priorisierung (Master in der Kette wird u.U. ausgeschlossen), Zuteilungszeit wächst mit der Länge der Daisy-Chain
↳ Hilfe: überlappende Arbitrierung (gemeinsame Bus-Busy-Leitung, mit der ein aktiver Master seine Bus-Belieferung anzeigt → Arbitrierung des nächsten Zyklus kann bereits während des gerade aktiven Zyklus erfolgen)



Identifikationsbus (dezentral, ohne Daisy-Chain): Jeder lokale Arbiter erhält eine ID-Nummer, welche seine Priorität bestimmt. Bei Bus-Anforderung geben alle aktiven Arbiter ihre ID-Nummer auf den Identifikationsbus, in einem konkurrenzfreien Verfahren gewinnt höchste Nummer.

Echtzeitaspekte von Systembussen

Einfaches System, synchroner Bus, keine Wartezyklen: vollständig deterministisches Zeitverhalten

Einfaches System, synchroner Bus, Wartezyklen: ebenfalls deterministisch, wenn Wartezyklen bekannt.
Bei variablen Wartezyklen Oberschranke, Abbruch durch Busmaster wenn überschritten

Mehrere Busmaster: Das Echtzeitverhalten wird vom Verhalten des Busses im Konfliktfall bestimmt. Erforderliche Maßnahmen zur Wahrung von Echtzeitfähigkeit:

- Prioritäten (Jeder Busmaster muss eindeutig eine Priorität zugewiesen sein)
- Preemption (Verlangt einen Busmaster mit höherer Priorität den Bus, wird der mit niedrigerer Priorität unterbrochen)
- Unterbrechbarkeit von Blocktransfers (Zur Vermeidung von Prioritäteninversion müssen Blocktransfers d.h. Datentransfers in langen Blöcken, unterbrochen werden können)
- Busmonitor (Dient der Überwachung der Buszuweisungsregeln. Überschreitet ein Master das ihm zugewiesene Zeitintervall, so muss der Busmonitor dies feststellen und Gegenmaßnahmen (z.B. Bus Error) einleiten.)

PCI-Bus: echtzeitfähig, gemultiplexter Adress- und Datenbus, synchron, erlaubt Burst-Transfers

PCI-Bus-Bridge wird wie ein normales PCI-Device behandelt
⇒ einfache hierarchische Kaskadierung

Burst-Transfer: Im Gegensatz zum Standardtransfer werden größere Datenblöcke als ununterbrochenes Bündel kleinerer Dateneinheiten übertragen, ohne dass gewisse Initialisierungen für jede dieser Untereinheiten neu erfolgen müssten.

Anzahl der benötigten Takte bei der Übertragung von n Worten:

Standard Read: $3 \cdot n$ Takte (je 1 Takt für Adresse, Warten und Daten)

Standard Write: $2 \cdot n$ Takte (n. je 1 Takt für Adresse und Daten)

Burst Read: $2 + n$ Takte (1 Adresse, 1 Warten, n · Daten)

Burst Write: $1 + n$ Takte (1 Adresse, n · Daten)

Dauer Buszyklus: $\frac{1}{Bustaktfrequenz}$, Anzahl Buszyklen in 10 s: $\frac{10s}{Dauer Buszyklus}$

$$\text{Übertragungsrate} = \frac{\text{Anzahl Bytes}}{\text{Taktykluszeit} \cdot \text{Anzahl Taktzyklen}}$$

Kodierung und Overhead bei PCIe:

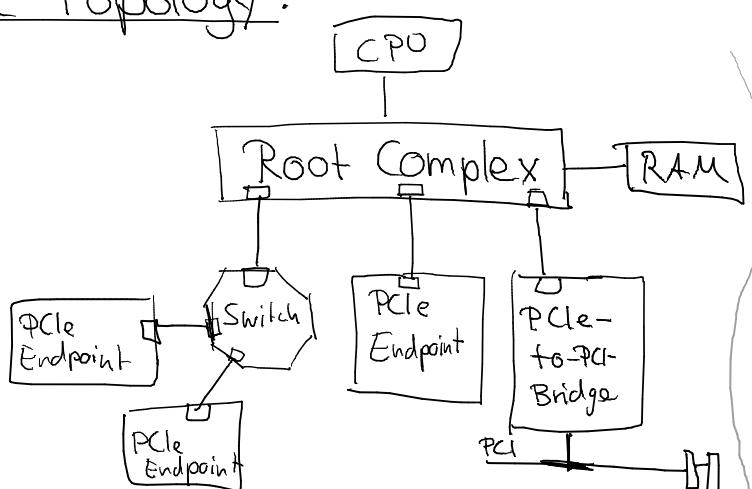
Standard	Kodierung	Overhead
PCIe 1.x, 2.x	8b/10b	0.2=20%
PCIe 3.x	128b/130b	0.015≈1,5%

$$\text{Overhead} = \frac{\text{Anzahl Bits Gesamt} - \text{Anzahl Bits Daten}}{\text{Anzahl Bits Gesamt}}$$

Voll duplex-Übertragungsgeschwindigkeit eines PCIe Links

$$x = 2 \cdot [\text{#Lanes}] \cdot [\text{Gigatransfer/s}] \cdot 1 \cdot \frac{1 - [\text{Overhead}]}{8} \quad \left[\frac{\text{Bytes}}{\text{s}} \right]$$

PCIe Topology:



Root Complex:

- Device welches den Host (CPU) und den Hauptspeicher mit der PCIe - Hierarchie verbindet.
- Agiert als Slave auf dem Host-Bus und als Master auf dem Systemspeicherbus
- tritt als Requester und Completer auf

Requester: Memory Read/Write Transaktion
 $\text{CPU} \rightarrow \text{PCIe Endpoint}$

Completer: Memory Read/Write Transaktion
 $\text{PCIe Endpoint} \rightarrow \text{RAM}$

Port: Schnittstelle zwischen PCIe Komponente und Link

↳ Ingress Port: Empfängt Pakete

↳ Egress Port: Sendet Pakete

Requester: Startet eine Read-/Write Transaktion in der PCIe Hierarchie
 (RC und EPs sind Requester)

Completer: Wird von einem Requester bei einer Read/Write Transaktion adressiert

↳ Ein Requester liest/schreibt von einem Completer
 (RC und EPs sind Completer)

PCIe-to-PCI-Bridge:

Verbindet eine PCIe und eine PCI Hierarchie

(ist eine Endpoint Komponente)

Switch: Besteht aus mehreren logischen PCI-to-PCI Bridges, welche jeweils einem Switch Port zugewandt sind.
 ↳ verbündet mehrere Sub-Hierarchien miteinander, und diese mit der übergeordneten Hierarchie

↳ realisiert das Routing von Paketen in der gesamten Hierarchie: Adress-Routing, ID-Routing, Implicit-Routing

Aufgabe 2-2: Operationsverstärker, A/D-Wandler

Operationsverstärker:

Annahmen idealer

Operationsverstärker:

Eingangswiderstand ∞

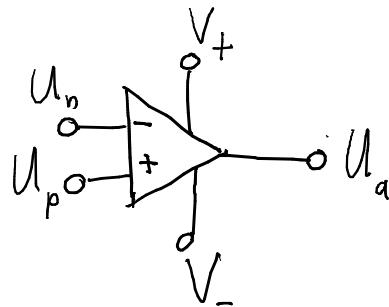
Ausgangswiderstand 0

Verstärkungsfaktor ∞

Lineare Verstärkung

Keine Eigendynamik

Eingangsstrom 0



U_n : invertierender Eingang

U_p : nichtinvertierender Eingang

V_+ : positive Versorgungsspannung

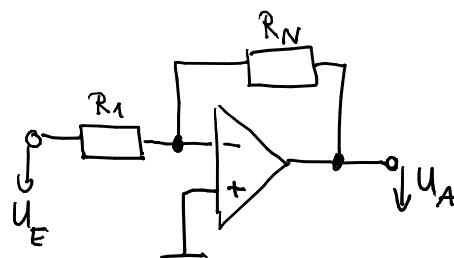
V_- : negative Versorgungsspannung

U_a : Ausgang $U_a = y \cdot (U_p - U_n)$

Clipping beim realen Operationsverstärker:

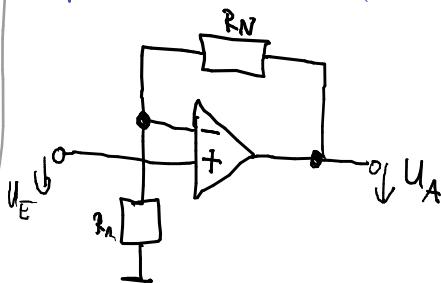
Der Ausgangswert ist durch die positive und negative Eingangsspannung begrenzt, weil der Operationsverstärker nur maximal bis zu seiner Versorgungsspannung verstärken kann.

Invertierender Operationsverstärker:



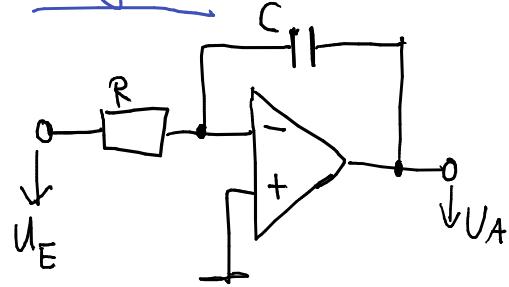
$$\frac{U_A}{U_E} = -\frac{R_N}{R_1} = y$$

Nichtinvertierender Operationsverstärker:



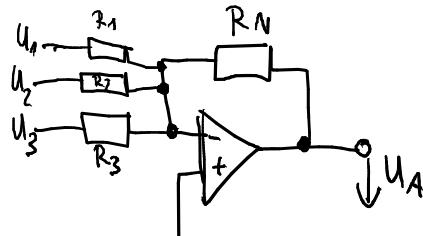
$$\frac{U_A}{U_E} = \frac{R_1 + R_N}{R_1} = y$$

Integrierer:



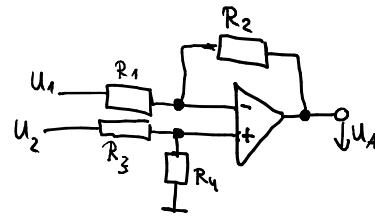
$$U_A = -\frac{1}{R \cdot C} \int U_E dt$$

Invertierender Addiervier:



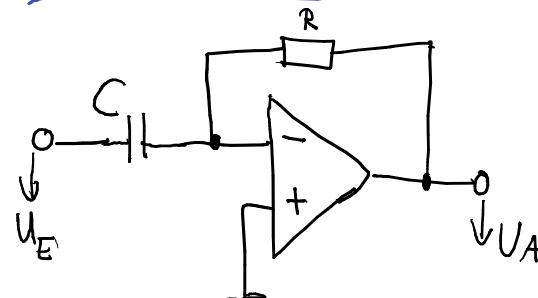
$$U_A = -\left(\frac{U_1}{R_1} + \frac{U_2}{R_2} + \frac{U_3}{R_3}\right) \cdot R_N$$

Subtrahierer:



$$U_A = \frac{(R_1 + R_2) \cdot R_4}{(R_3 + R_4) \cdot R_1} \cdot U_2 - \frac{R_2}{R_1} \cdot U_1$$

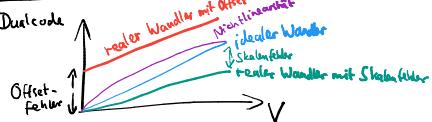
Differenzierer:



$$U_A = -C \cdot R \cdot \frac{dU_E}{dt}$$

A/D-Wandler: \oplus : digitale Signale weniger störanfällig, \ominus : Informationsverlust durch die Wandlung

Kennzahlen: Relative Genauigkeit, Skalenfehler, Offsetfehler



$$U_{LSB} = \frac{U_{max} - U_{min}}{2^n} \quad (\text{bei } n \text{ Bit, kleiner Spannungsschritt})$$

$$\text{Dualzahl} \rightarrow \text{Spannung: } U = (Z \cdot U_{LSB}) + U_{min}$$

$$\text{Spannung} \rightarrow \text{Dualzahl: } Z = \frac{U - U_{min}}{U_{LSB}}$$

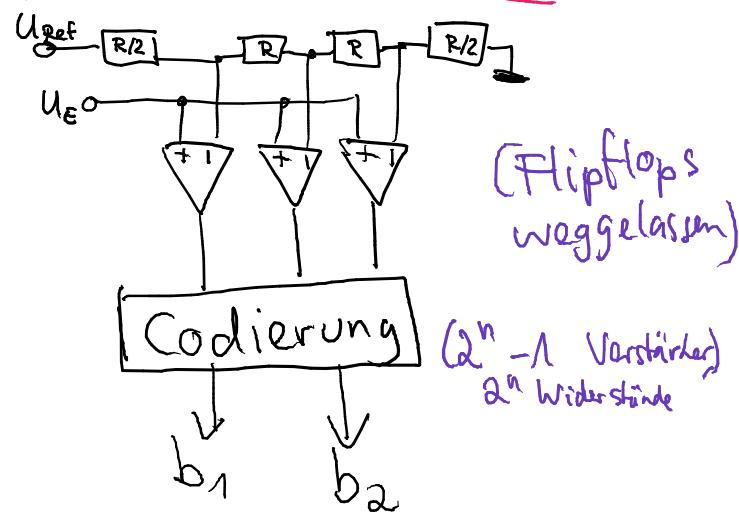
Integrationsverfahren: Funktioniert über den Vergleich der Zeit, die zum Laden und Entladen eines Kondensators benötigt wird.

Kompensationsverfahren/Zählverfahren: Die Eingangsspannung wird mit der maximalen Wandlungsdauer, durch einen DA-Wandler gewandelten Spannung verglichen.
 2^{n-1} Takte
Ein Vorwärts-Rückwärts-Zähler enthält den zu wandelnden Wert.

Wägeverfahren: Wie Kompensationsverfahren, nur mit Register statt
hohe Umsatzrate:
Vorwärts-Rückwärts-Zähler \rightarrow Prinzip binäre Suche
n Takte bei n Bitwort
 \rightarrow für ES geeignet

Parallelverfahren: Die Eingangsspannung wird gleichzeitig mit $2^n - 1$ Referenzspannungen über einen Spannungsteiler durch Komparatoren verglichen. Der Ausgang der Komparatoren wird durch D-Flipflops gespeichert. Die Ausgänge der Flipflops sind an einen Codierer \rightarrow für ES geeignet angeschlossen, welcher die entsprechende binäre Zahl ausgibt.

2-Bit-Parallel-Wandler:



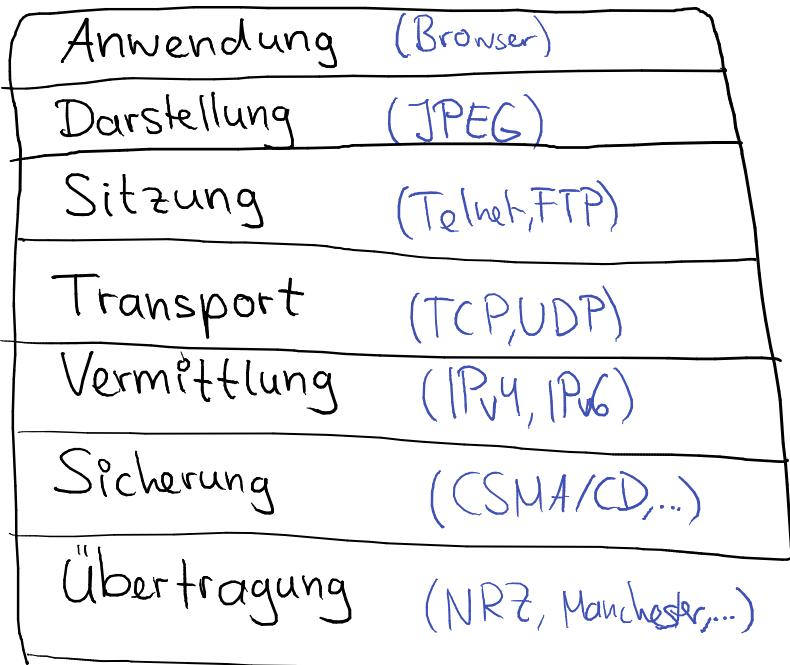
Differentielle Datenübertragung:

(doppelte Leitungszahl, die jeweils zweite Leitung sendet das invertierte Signal)

\oplus : höhere Störsicherheit und damit höhere mögliche Datenübertragungsraten möglich

Aufgabe 3.1: Echtzeitkommunikation

ISO / OSI Schichtenmodell:



„Aber die Sarah tut Verdran sicher übel.“

Vorteile Schichtenmodell:

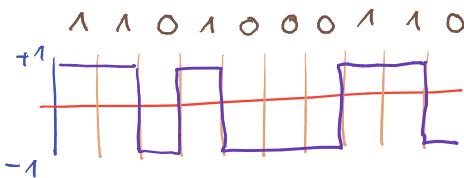
- Schichten einzeln implementierbar
- nicht alle Schichten müssen für eine erfolgreiche Kommunikation implementiert werden
- Einsatz verschiedener Protokolle auf einer Schicht möglich
- Datenintegrität kann in verschiedenen Schichten geprüft und sichergestellt werden

Verringerung der Nutzdatenrate durch:

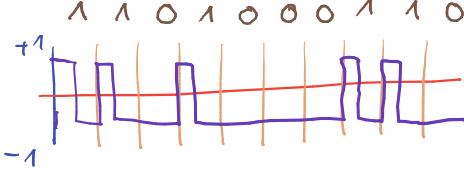
- Protokoll-Overhead (Header+Trailer)
- Übertragungsfehler
- Kollisionen
- Paket-Wiederholungen (Re-Transmissions)

Signalkodierung:

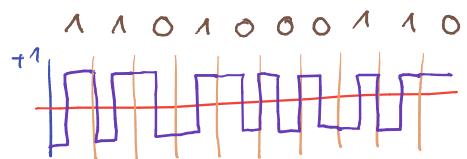
No Return to Zero (NRZ):



Return to zero (RZ):



Manchester (NRZ XOR Takt):



(Wenn lang, dann Wechselt)

- ⊕ benötigt zur Dekodierung kein externes Taktsignal zwischen Sender und Empfänger

→ Takt kann aus dem Signalverlauf abgeleitet werden, da Manchester-Kodierung in der Mitte jedes Taks den Spannungspiegel wechselt.

- ⊖: Signal benötigt doppelte Übertragungsbandbreite im Vergleich zu NRZ-Kodierung, da zur Übertragung eines Nutzdaten-Bits zwei Signalzustände (Symbole) benötigt werden, die innerhalb einer Taktperiode übertragen werden müssen.

Manchester-Kodierung selbstsynchronisierend, weil kein zusätzlicher Takt zur Synchronisierung benötigt wird, da das kodierte Signal an sich den Takt enthält.

Übertragungsfehler:

- falsche Wiederholung von Telegrammen
- falsche Einfügung von Telegrammen
- falsche Abfolge von Telegrammen
- Verzögerung von Telegrammen
- Verlust von Telegrammen

Zugriffsverfahren auf Echtzeit-Kommunikationssysteme:

Polling (zyklisches Abfragen):

- ↪ (+): einfach implementierbar, Worst-Case-Alarmerkennungszeit ($= 2 \cdot \text{Zykluszeit}$, proportional zu Anzahl Teilnehmer) kann berechnet werden → echtzeitfähig
- ↪ (-): geringere Übertragungskapazität und verlängerte Gesamtreaktionszeiten durch „unnötige Abfragen“ bei Teilnehmern mit hoher Priorität und kurzer Antwortzeit

CSMA/CD (Carrier Sense Multiple Access/Collision Detection):

- ↪ wird bei Ethernet eingesetzt
- ↪ (+): ermöglicht Multi-Masterbetrieb, Übertragungszeiten nur für die eigentliche Nachrichtenübertragung gebraucht (falls Kollisionen vernachlässigbar)
- ↪ (-): Antwortzeitverhalten nicht exakt vorhersehbar, falls Kollisionen auftreten
→ nicht echtzeitfähig

$$T_{\text{Paket}} \geq 2 \cdot T_{\text{Prop}} \Rightarrow \frac{T_{\text{Paket}}}{BW} \geq 2 \cdot \frac{l}{V_R}$$

Annotations:
 T_{Paket} : Übertragungsdauer eines Paketes
 T_{Prop} : Signallaufzeit von einem Ende des Busses zum anderen
 BW : Übertragungsrate
 l : Paketlänge
 V_R : Ausbreitungsgeschwindigkeit
maximale Breitbandbreite

Standard-Ethernet ist nicht echtzeitfähig, da es keine prioritätenbasierte Arbitrierung besitzt, welche für Echtzeitfähigkeit eine wichtige Voraussetzung darstellt.
(echtzeitfähiger Feldbus, der auf Ethernet basiert: EtherCAT)

CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance):

- ↪ wird vom CAN-Bus verwendet
- ↪ Unterschied zu CSMA/CD: Übertragungskanal wird nicht während, sondern vor dem Senden abgehört (geschriebene „1“ kann von fremder „0“ überschrieben werden, das wird erkannt)
→ die Nachricht des „überlegenen“ Senders wird bei Kollision nicht zerstört (anders als bei CSMA/CD)
- ↪ implizite Priorisierung der Nachrichten gegeben

↪ echtzeitfähig

Token-Passing:

- ↪ deterministischer Buszugriff, Antwortzeitverhalten vorhersagbar
→ echtzeitfähig

↪ wird von Ring-Netzwerk benutzt

$$T_{\text{Paket}} = \frac{\text{Paketlänge in Bit}}{\text{Übertragungsrate}}$$

$$\text{MMAT} = T_{\text{Paket}} + N \cdot T_{\text{Token}} + T_{\text{Prop}} + (N-1) \cdot T_{\text{THT}}$$

Annotations:
 T_{Paket} : Übertragungsdauer eines Pakets
 N : Anzahl Netznoten
 T_{Token} : Übertragungsdauer des Tokens
 T_{Prop} : Zeit, die ein Signal benötigt, um einmal um den Ring zu wandern
 T_{THT} : Token-Haltezeit

TDMA (Time Division Multiple Access):

- ↪ jeder Busteilnehmer bekommt feste Zeitscheibe → echtzeitfähig
- ↪ (-): Synchronisierung der Teilnehmer notwendig

Anforderungen an Feldbusse:

- Echtzeitfähigkeit: Einhaltung harter Zeitbedingungen
- effiziente Übertragung kleiner Datenmengen
- geringe Latenz
- Robustheit (Material, Fehlererkennung und -behebung, digital statt analog)
- Netzausdehnung bis 1000m, Knotenzahl bis 254
- Interoperabilität (Austauschbarkeit) von Geräten verschiedener Hersteller

CAN Dataframe

TCP/IP

Profibus:

- ↳ Polling + Token-Passing zwischen Mästern, NRZ und MBP Kodierung, Linie mit Stichleitungen
- ↳ multimasterfähig, große Buslänge → gut für Gebäude automation

CAN-Bus:

- ↳ Polling- oder ereignisgesteuerter Betrieb möglich (CSMA/CA: bitweise, nicht zerstörende Arbitrierung), NRZ-Kodierung mit dominanter 0 und rezessiver 1, Linie mit Stichleitungen
- ↳ hohe Übertragungssicherheit → gut für Datenaustausch zwischen den elektronischen Steuengeräten eines Kraftfahrzeugs

Interbus:

- ↳ Feste Zeitslots (TDMA), NRZ-Kodierung, aktiver Ring, Monomaster
- ↳ feste MMAT, geringe Latenz → gut für Automation in der Produktion

Sercos:

- ↳ kurze Zykluszeiten, nur wenige Daten, Synchronisierung → gut für Regelung eines Industrieroboters

Bei der Übertragung von Daten können Schwankungen der Übertragungsrate durch das Zwischenpuffern von Daten ausgeglichen werden.

Paketpuffer: (Paketlänge P bits, Übertragungsrate $R \frac{\text{bit}}{\text{s}}$)

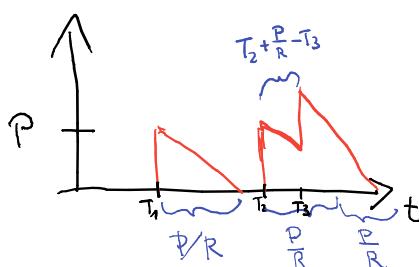
- bei jedem Ablegen eines neuen Pakets im Puffer springt dessen Füllstand um P bit und nimmt dann mit $R \frac{\text{bit}}{\text{s}}$ ab, Puffer aufangs leer.

- Verzögerungszeit: Summe der Übertragungsdauer des Pakets sowie eine eventuelle Wartezeit

- Es gibt zwischen Zeitpunkt T_i und T_{i+1} keine Wartezeiten,

$$\text{falls } T_{i+1} \geq T_i + \frac{P}{R}$$

Füllstand des Puffers über der Zeit:



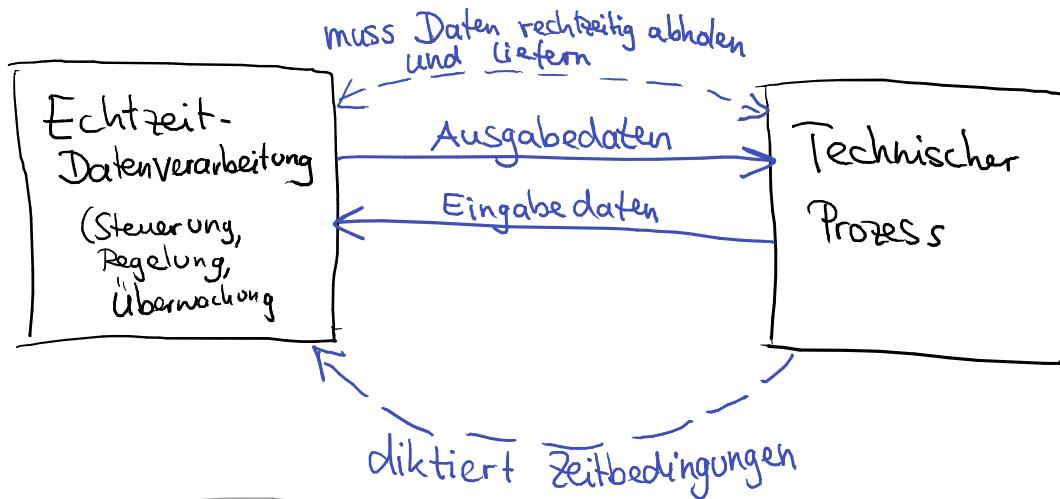
Aufgabe 3.2: Echtzeitprogrammierung

Nicht-Echtzeit System: logische Korrektheit

Echtzeit System: logische Korrektheit + zeitliche Korrektheit

→ benötigt Verfahren, die eine automatische Berücksichtigung ermöglichen (Scheduling, Prioritätsverteilung)

Zusammenspiel Echtzeit-Datenverarbeitung und technischer Prozess:



Zeitbedingungen: (genauer/frühester/spätester Zeitpunkt, Zeitintervall)

(periodische Zeitbedingungen: zyklische (periodische) Tasks, aperiodische Zeitbedingungen: asynchrone Tasks)

↳ **Welche Echtzeitbedingungen** (können in gewissen Grenzen überschritten werden, ohne zu fatalen Systemzuständen zu führen)

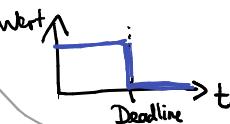
z.B.: Video-Streaming, periodische Abfrage Temperatursensor



↳ **feste Echtzeitbedingungen** (bewirken bei Überschreiten einen Abbruch der Aktion)

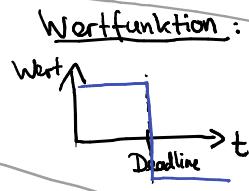
z.B.: Positionsbestimmung

Wertfunktion:



↳ **harte Echtzeitbedingungen** (müssen auf jeden Fall eingehalten werden, andernfalls droht Schaden)

z.B.: Anhalten vor Ampel bei autonomen Autos



Anforderungen für Echtzeitsysteme:

↳ **Rechtzeitigkeit** (Die Ausgabedaten müssen rechtzeitig berechnet werden und zur Verfügung stehen)
↳ kann durch dynamische Speicherallokation nicht gewährleistet werden, da die Laufzeit und erfolgreiche Speicherallokation nicht gewährleistet ist)

- ① Eingabedaten rechtzeitig holen
- ② Ausgabedaten rechtzeitig berechnen und zur Verfügung stellen

↳ **Gleichzeitigkeit** (Rechtzeitigkeit für mehrere Aktionen)

↳ Erfüllung durch

- vollständige Parallelverarbeitung in Mehrprozessor-System
- jeder Prozessor macht eine Aufgabe

Quasi-Parallelverarbeitung
(Mehr- oder Einprozessor-System)

Überlappend Echtzeit-Scheduler

↳ **Verfügbarkeit** (Läuft 24/7) → erfordert keine Unterbrechungen des Betriebs

↳ kann durch Garbage Collection nicht garantiert werden, da zu lange Rüben in Programm ausführen durch Speicherover-

für Reorganisationsphasen gression im Block)

Echtzeitprogrammierverfahren

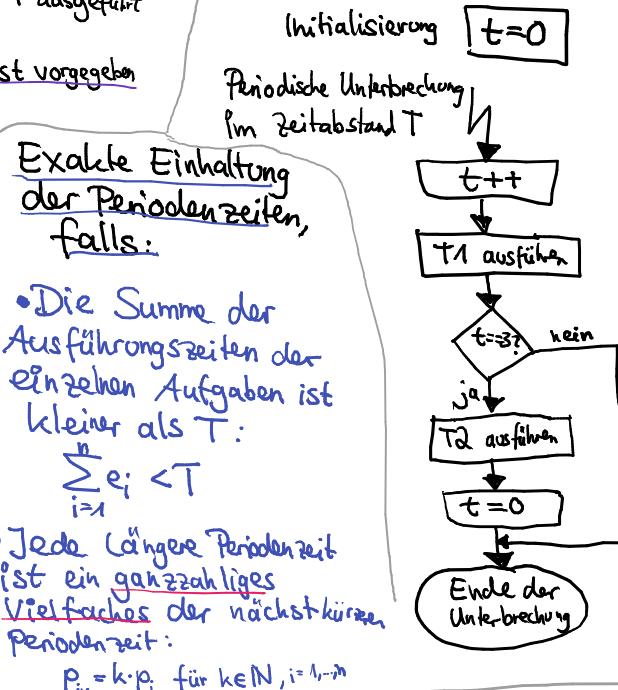
Synchrone Programmierung (Konstruktion zeitgesteuerter Systeme)

- ↳ das zeitliche Verhalten periodischer Aktionen wird vor deren Ausführung geplant
- ↳ die periodischen Aktionen werden in ein Zeitraster T synchronisiert
- ↳ Aktionen werden immer zu ganzzahligen Vielfachen von T ausgeführt
- ↳ das Zeitraster wird durch einen Zeitgeber gewonnen
- ↳ die Reihenfolge des Ablaufs der Teilprogramme wird fest vorgegeben

- (+): Festes, vorhersagbares Zeitverhalten
- (+): Einfache Tests und Analyse des Systems
- (+): Rechtzeitigkeit und Gleichzeitigkeit können leicht garantiert werden
- (-): Geringe Flexibilität bei Änderung der Aufgabenstellung
- (-): Keine Behandlung aperiodischer Ereignisse vorgesehen (nur mittels periodischer Überprüfung möglich)

Abweichung der Periodendauer = Summe der Laufzeiten der vorher ablaufenden Tasks

Ablaufplanung:



- Die Summe der Ausführungszeiten der einzelnen Aufgaben ist kleiner als T:
- $$\sum_{i=1}^n e_i < T$$

- Jede längere Periodenzzeit ist ein ganzzahliges Vielfaches der nächstkürzeren Periodenzzeit:
- $$P_m = k \cdot p_i \text{ für } k \in \mathbb{N}, i=1, \dots, n$$

asynchrone Programmierung (Konstruktion ereignisgesteuerter Systeme)

- ↳ Ablaufplanung der Aktionen zur Laufzeit durch ein Organisationsprogramm (Echtzeitbetriebssystem)

Aufgabe des Organisationsprogramms: Echtzeitscheduling

- ↳ Ereignisse überwachen
- ↳ Informationen über einzuhaltende Zeitbedingungen entgegennehmen
- ↳ Ablaufreihenfolge von Teilprogrammen zur Bearbeitung der eingetretenen Ereignisse ermitteln
- ↳ Teilprogramme gemäß der ermittelten Ablaufreihenfolge aktivieren

- (+): flexibler Programmablauf und flexible Programmstruktur

- (+): Reaktion auf periodische und aperiodische Ereignisse

- (-): Rechtzeitigkeit nicht in jedem Fall im Voraus garantierbar

(Je nach verwandelter Echtzeitschedulingstrategie ist jedoch bei Kenntnis bestimmter Parameter (kleinste Periodendauer, größte Ausführungszeit der Aktivitäten) eine Voranalyse zur Rechtzeitigkeit möglich (Feasibility Analysis))

- (-): Systemtest und Systemanalyse schwieriger als bei der synchronen Programmierung, denn je niedriger die Priorität einer Aktivität, desto größer die möglichen Zeitschwankungen

Arten von Ablaufsteuerung:

- zyklisch { Prozessor ständig aktiv, geeignet für synchrone Programmierung
- zeitgesteuert

- unterbrechungssteuert { energieeffizient, geeignet für synchrone Programmierung (Unterbrechung durch Zeitablauf) und asynchrone Programmierung (Unterbrechung durch Ereignis)

Kaskadenregler: Regler bekommt Sollwert, vergleicht mit Istwert und erzeugt Sollwert für nächsten Reg

Echtzeit Scheduling (Aufgabe: Aufteilung des Prozessors zwischen allen ablaufwilligen Tasks derart, dass - soweit möglich - alle Zeitbedingungen eingehalten werden)

Unterteilungskriterien Scheduling-Verfahren:

- statisch vs. dynamisch
- preemptive vs. nicht-preemptive
- keine / statische/dynamische Prioritäten

Ein Schedulingverfahren heißt optimal, wenn es einen Schedule findet, sofern dieser existiert.

Prozessorauslastung: $H = \frac{\text{Benzige Prozessorzeit}}{\text{Verfügbare Prozessorzeit}}$

Auf einem Einprozessorsystem für n periodische Tasks mit Zeitschranke = Periode:

$$H = \sum_{i=1}^n \frac{e_i}{P_i}$$

Ausführungszeit Task i
Periodendauer von Task i

$H > 100\%$: kein Schedule

$H \leq 100\%$: Schedule existiert

FIFO-Scheduling (First In First Out)

„Wer zuerst kommt, bekommt den Prozessor“

↳ dynamisch, nicht-preemptiv, ohne Prioritäten

↳ wenig geeignet für Echtzeit, da keine Zeitbedingungen eingehalten

Fixed-Priority-Scheduling

„jeder Task erhält feste Priorität“

↳ dynamisch, statische Prioritäten

↳ 2 Varianten: FPP (Fixed Priority Preemptive)

FPN (Fixed Priority Non Preemptive)

↳ Zuordnung der Prioritäten zu den Tasks ist entscheidend für das Echtzeitverhalten

RMS (Rate Monotonic Scheduling): Prioritätenzuordnung für periodische Tasks

Prioritäten umgekehrt proportional zu Periodendauern: $PR_i \sim \frac{1}{P_i}$

„je geringer die Periodendauer, desto höher die Priorität“

↳ preemptiv ↳ Tasks voneinander unabhängig

↳ liefert bei festen Prioritäten und Reemption für periodische Tasks, bei denen die Zeitschranke identisch zur Periode ist, eine optimale Prioritätenverteilung

↳ Feste Prioritäten mit Reemption kein optimales Schedulingverfahren:

Obergrenze der Prozessorauslastung, bis zu der RMS immer einen Schedule findet:

$$H_{\max} = n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

bei n Tasks
die Formel auch bei FPP benutzen!

$$H_n \xrightarrow{n \rightarrow \infty} \ln(2)$$

RMS ist bei harmonischen Periodendauern (Periodendauern sind Vielfache von einander) und Auslastung ≤ 1 optimal.

EDF (Earliest Deadline First): „derjenige Task bekommt den Prozessor, der am nächsten an seiner Zeitschranke (Deadline) ist“

↳ Prioritäten ergeben sich automatisch und dynamisch aus den Zeitschränken

↳ dynamisch, dynamische Prioritäten, preemptiv oder nicht-preemptiv

↳ Preemptives EDF-Scheduling optimal auf Einprozessorsystemen

LLF (Least Laxity First): „derjenige Task bekommt den Prozessor, der den geringsten Spielraum (Laxity) hat.“

↳ dynamisch, dynamische Prioritäten, preemptiv oder nicht-preemptiv

⊖: drastisch höhere Anzahl an Taskwechseln als EDF

⊕: besseres Verhalten bei Überlast als EDF

↳ Preemptives LLF-Scheduling optimal auf Einprozessorsystemen

Time-Slice-Scheduling (Zeitscheibenverfahren): „ordnet jedem Task eine feste Zeitscheibe zu“

↳ Reihenfolge der Taskausführung entspricht der Reihenfolge der ablaufwilligen Tasks in der Taskverwaltungsliste des Betriebssystems

↳ Dauer der Zeitscheibe jedes Tasks individuell festlegbar

↳ preemptiv, dynamisch, ohne Prioritäten

↳ einfaches Verfahren, bei feinkörnigen Zeitscheiben nahe an der Optimalität, aber Periodendauer abhängig von der Anzahl der Tasks

GP (Guaranteed Percentage):

„ordnet jedem Task einen festen Prozentsatz der verfügbaren Prozessorleistung innerhalb einer Periode zu (virtueller Prozessor pro Task)“

↳ preemptiv, dynamisch, ohne Prioritäten

↳ Periodendauer unabhängig von der Anzahl der Tasks

↳ viele Taskwechsel wie bei LLF

↳ für mehrfädige Prozessoren geeignet

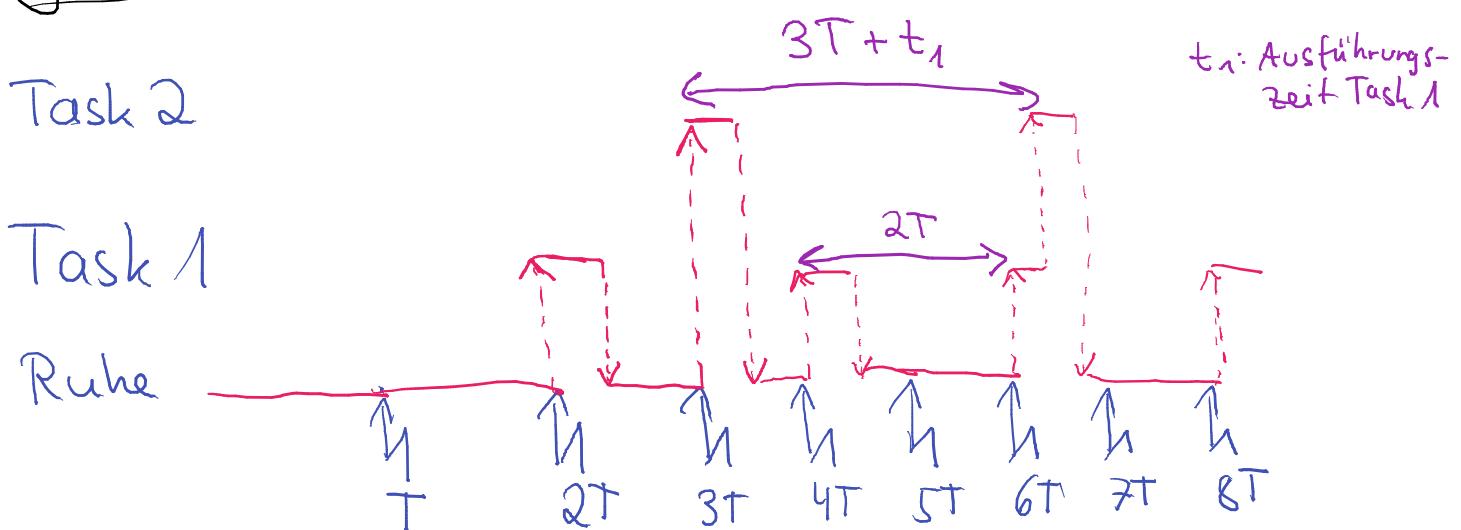
↳ auf Einprozessorsystemen optimales Schedulingverfahren

$$\text{Spielraum } l_i = d_i - (t + e_r)$$

↑ Zeitschranke (Deadline) ↑ aktueller Zeitpunkt ↑ Restausführungszeit

- Existiert gültiger Schedule?
 - ↳ Ja, wenn Prozessorauslastung $H = \sum_{i=1}^n \frac{e_i}{p_i} \leq 100\%$.
- Findet (spezifisches) Verfahren einen gültigen Schedule?
 - ↳ optimale Verfahren (EDF, LLF, GP): Ja, wenn $H \leq 100\%$.
 - ↳ FPP, RMS:
 - ↳ $H \leq n \cdot (2^{\frac{1}{n}} - 1)$?
 - ↳ ja
 - ↳ sonst: Prioritäten manuell vergeben und ausprobieren
- Schedule gültig?
 - ↳ prüfen ob Verfahren optimal
 - ↳ sonst: Zeitverteilung finden, in Busy-Period-Analysis
(Zeitverhalten für begrenzte Zeit analysieren)

Beispiel Periodenabweichung, wenn Periode P1 kein ganzzahliges Vielfaches von Periode P2:



Angestrebte Perioden dauer: Task 1: $\frac{2T}{3T}$
Task 2: $\frac{3T}{3T + t_1}$

Aufgabe 4.1: Echtzeitbetriebssysteme

Aufgaben eines Standardbetriebssystems:

- Taskverwaltung (Zuteilung des Prozessors an die Tasks)
- Betriebsmittelverwaltung (Speicherverwaltung, I/O-Verwaltung)
- Interprozesskommunikation (Kommunikation zwischen Tasks)
- Synchronisation (Zeitliche Koordination der Tasks)
 - ↳ Verwaltung des Zugriffs von gemeinsam genutzten Betriebsmitteln
- Schutzmaßnahmen (Schutz der Betriebsmittel vor unberechtigten Zugriffen der Tasks)

ist Unterkategorie der Kommunikation kann durch Kommunikation von realisiert werden
 ↳ Verschiedene Methoden der Nachrichtenweiterleitung
 ↳ Asynchrone Antwort mit Wartezeit
 ↳ gemeinsamer Speicher (Scalpium)

Zusätzliche Aufgaben eines Echtzeitbetriebssystems:

- Wahrung der Rechtzeitigkeit und Gleichzeitigkeit
 - ↳ Daten müssen rechtzeitig geholt, verarbeitet und ausgegeben werden
 - ↳ Rechtzeitigkeit muss für mehrere Aktionen gleichzeitig gewährleistet sein
- Wahrung der Verfügbarkeit
 - ↳ keine unvorhergesehenen und/oder unberechenbaren Unterbrechungen des Betriebs

Monolithischer Aufbau eines Betriebssystems:

- ↳ Alle Funktionalität in einem einheitlichen, nicht weiter unterteilten Block realisiert
 - ⊖ Schlechte Wartbarkeit und Anpassbarkeit
 - ⊖ hohe Fehleranfälligkeit
- ↳ heutige Betriebssysteme folgen stattdessen hierarchischen Schichtenmodellen (Schichten abstrahieren jeweils von der darunterliegenden Schicht und erweitern deren Funktionalität)

Kern eines Betriebssystems: Teil des Betriebssystems, der im Kernelmode ausgeführt wird (Zugriff auf privilegierte Befehle, Speicherbereiche und Ressourcen).
 (Gegenteil ist der Usermode: Zugriff auf Ressourcen nur über den Kern möglich → Schutzmechanismus)

Makrokern-Betriebssystem: Viele Schichten laufen im Kernelmode

- ↳ Gerätetreiber, Ressourcenverwaltung, Taskverwaltung sind Teil des Kerns
- ↳ Wird auch als „monolithischer Kern“ bezeichnet
- ↳ Beispiel: Linux, Windows XP

Beispiel Betriebssystemaufsatz: RT Linux
 Beispiel Echtzeitbetriebssystem: Windows CE

Mikrokern-Betriebssystem: Kern sehr schlank → enthält nur die in allen Fällen benötigten Bestandteile (Interprozesskommunikation, Synchronisation, elementare Taskverwaltung (einrichten, aktivieren, blockieren, beenden) Adressraumverwaltung (sonst wäre Spezialschutz nicht realisierbar)). Alle anderen Funktionen werden im Usermode erledigt

- ⊕: Anpassbarkeit, Skalierbarkeit an Aufgabe
- ⊕: Portierbarkeit
- ⊕: Zeitliche Vorhersagbarkeit

kurze, kritische Bereiche
 Kern fast immer unterbrechbar (preemptiver Kern)
 Schnelle Reaktion auf Ereignisse

- ⊖: häufiger Wechsel zwischen Kernel- und User mode

- ⊖: Schutz verringert

Unterschied zwischen Task und Thread:

Aus Sicht der Betriebszustände, der Zeitparameter, des Schedulings und der Synchronisation sind Thread und Task gleich.

Task (schwergewichtiger Prozess):

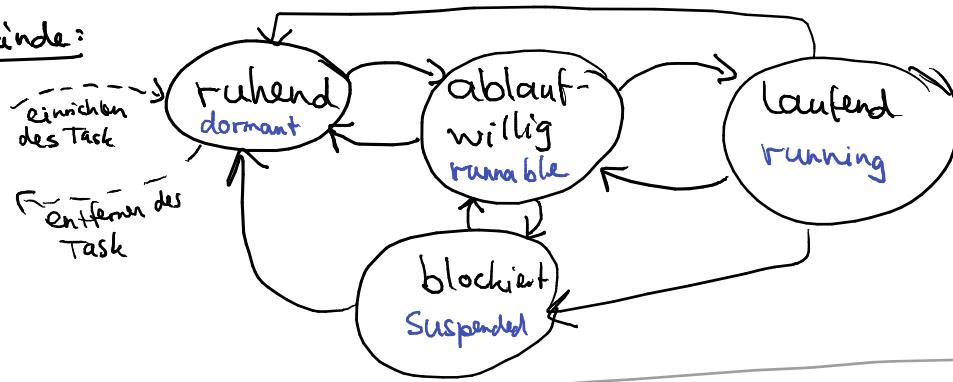
- ↳ eigene Umgebung, Betriebsmittel, Adressraum
- ↳ abgeschirmt von anderen Tasks
- ↳ Kommunikation nur über Interprozesskommunikation
- ↳ Umschaltung zwischen Tasks teuer

Thread (leichtgewichtiger Prozess):

- ↳ existiert innerhalb eines Tasks
- ↳ Betriebsmittel, Adressraum, Variablen geteilt mit den anderen Threads dieses Tasks
- ↳ schneller Kontextwechsel, effiziente Kommunikation
- ↳ kaum Schutz zwischen Threads eines Tasks

Prozesskontext: Programmzähler, Statusregister, Steuerregister, Stapelzeiger, Registerblock

Prozesszustände:



Arten von Synchronisation:

Sperrsynchronisation (Mutex):

- ↳ Wechselseitiger Ausschluss
- ↳ die geschützte Ressource kann zu jeder Zeit nur von einem Task verwendet werden

Reihenfolgensynchronisation (Kooperation):

- ↳ Reihenfolge der Zugriffe auf eine Ressource wird exakt definiert

Realisierung von Synchronisation durch Semaphoren:

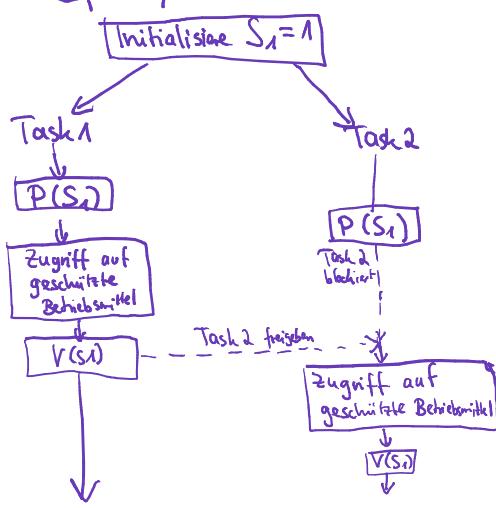
↳ atomare Operationen „Passieren“(P) und „Verlassen“(V)

↳ P verringert, V erhöht eine Zählervariable

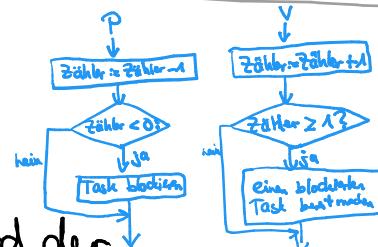
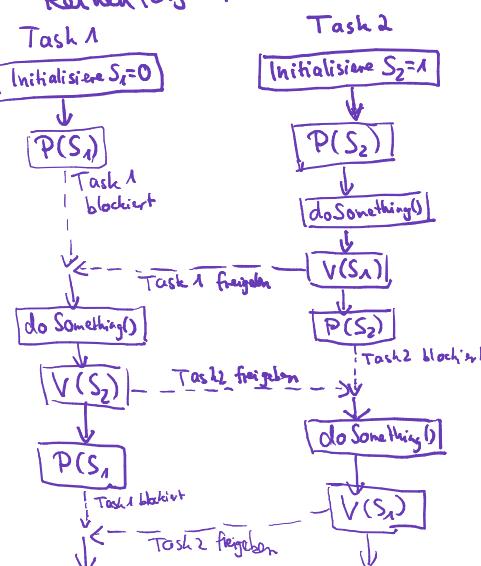
↳ erreicht die Zählervariable einen Wert kleiner 0, wird der aufrufende Task blockiert

↳ Freigabe darf auch durch anderen Task erfolgen!

Sperrsynchronisation



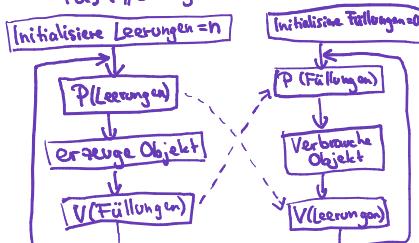
Reihenfolgensynchronisation



$$P(S) \triangleq S--;$$

$$V(S) \triangleq S++;$$

Erzeuger/Verbraucher-Synchronisation
Task „Erzeuger“ Task „Verbraucher“



Leerungen: Freie Plätze für Objekte des Erzeugers
Füllungen: erzeugte Objekte für Verbraucher

Typen von Verklemmungen:

Deadlock (mehrere Tasks warten auf die Freigabe von Betriebsmitteln, die sich gegenseitig blockieren)

↳ Beispiel: Kreuzweise Betriebsmittelbelegung führt zu gegenseitiger Blockierung

↳ kann nicht mehr aufgelöst werden

↳ Vermeidung:

• Alle Tasks, die mehrere gemeinsam benutzte Betriebsmittel gleichzeitig belegen wollen, müssen dies in der gleichen Reihenfolge tun! → Vergabe der Betriebsmittel in gleicher Reihenfolge

• Monitore können fehlerhafte Benutzung von Semaphoren verhindern

• Abhängigkeitsanalyse

• Timeout

Livelock / Starvation (ein Task wird durch andere Tasks ständig an der Ausführung gehindert)

↳ Unterschied zu Deadlock: Task verharrt nicht in einem Zustand, sondern wechselt ständig zwischen mehreren, aus denen er nicht mehr entkommen kann

↳ Beispiel: Niederpriore Tasks werden durch höherpriore ständig an der Ausführung gehindert

↳ Durch Prioritäteninversion kann jedoch auch ein niederpriorer Task einen höherprioren aushungern!

Prioritäteninversion

↳ niederpriorer Task belegt Betriebsmittel, das durch höherprioren Task benötigt wird

↳ letzterer muss daher warten

↳ wird nun der niederpriore Task durch andere, höherpriore Tasks an der Ausführung gehindert, kommt es zur Prioritäteninversion

↳ Dieser blockiert nun auch den ursprünglichen, hochprioren Task!

Kurzform: Der niederpriore Task hat eine Ressource, auf die der hochpriorere Task zugreifen möchte. Der niederpriore Task wird nun von einem mittelprioren Task unterbrochen, dadurch wird der hochpriorere Task noch länger blockiert.

Vermeidung durch Prioritätenvererbung:

↳ niederpriorer Task erhält vorübergehend die Priorität des hochprioren

Varianten der Task-Kommunikation

1) Gemeinsamer Speicher (Synchronisation per Semaphore; schnell, daher bevorzugt in Echtzeitsystemen; nicht möglich in verteilten Systemen)

2) Nachrichten (Daten austausch über das Verschicken von Nachrichten)

Prioritätenbasierte Kommunikation in Echtzeitsystemen:

↳ hochpriorisierte Nachrichten können niederpriore überholen → vermeidet Prioritäteninversion

↳ „Wahrung der End-zu-End-Prioritäten“

• zeitliche Koordination: synchron vs. asynchron

↳ Briefkasten: Zwischen Speicher für Nachrichten, ist keinem Prozess zugeordnet („lose Kopplung“)
Port: Anschluss zur Weiterleitung/Zustellung von Nachrichten, der einem bestimmten Prozess zugeordnet ist („enge Kopplung“)

• Kanal: temporär vs. stationär

Speicherverwaltung: Voraussetzungen für Mehrprogrammbetrieb

- **Verschiebbarkeit (Relozierbarkeit):** Tasks können an beliebigen Stellen im Arbeitsspeicher abgelegt werden
- **Auslagerung:** Tasks (oder Teile davon) müssen vom Arbeitsspeicher auf einen Massenspeicher abgelegt werden können.
- **Schutzmaßnahmen:** Tasks dürfen nicht in den Arbeitsspeicherbereich anderer Tasks schreiben

Speicherverwaltung ist selbst ein Task, benötigt daher die Taskverwaltung. Anfragen müssen synchronisiert werden. Informationen werden mit anderen Tasks ausgetauscht, also wird Interprozesskommunikation benötigt.

Speicherverwaltung: Adressierung

Reelle Adressierung: Ein kleiner logischer Adressraum wird auf einen größeren oder gleichgroßen Adressraum abgebildet. Das bedeutet, der gesamte logische Adressraum für alle Tasks kann auf dem physikalischen Speicher abgebildet werden

Virtuelle Adressierung: Ein größerer Adressraum wird auf einen kleineren physikalischen Adressraum abgebildet. Dies heißt, es müssen Verdrängungen stattfinden, um den Speicherbedarf aller Tasks zu befriedigen.

⊕: größerer Adressraum

⊕: leichteres Abfangen von Zugriffsverletzungen ("Zugriffsfags")

⊖: aufwändige Realisierung

⊖: Verdrängungsstrategien und Nachschubstrategien benötigt

⊖: Adressen müssen aufgelöst werden

⊖: Zugriffszeit auf Speicherseiten nicht vorhersagbar

(Vermeidung: Fest reservierter physikalischer Speicher für Echtzeitthreads)

Speicherverwaltung: Adressbildung

Lineare Adressbildung: Die Speicherabbildungsfunktion M bildet einen Block sequentieller Speicheradressen geschlossen wieder auf einen Block ab. Benachbarte logische Adressen innerhalb solch einen Block ab. Benachbarte logische Adressen innerhalb eines solchen Blocks bleiben auch im physikalischen Adressraum benachbart. (Kompletter logischer Speicher eines Tasks wird als einziger Block in den realen Speicher gelegt)

Streuende Adressbildung: Die Speicherabbildungsfunktion M kann einen Block sequentieller Speicheradressen in eine beliebige Reihenfolge überführen, logisch benachbarte Adressen müssen im physikalischen Adressraum nicht mehr benachbart sein. (Logischer Speicher wird in Seiten unterteilt und diese auf Kacheln im realen Speicher verteilt)

⊕: einfache Zuweisung, keine Zuteilungsstrategie

⊕: dynamische Taskgrößenänderung einfacher und zeitlich besser vorhersagbar

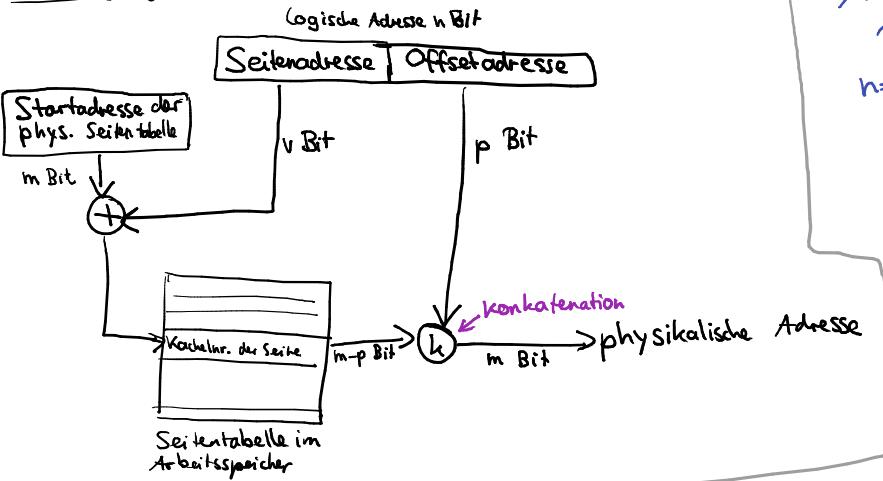
⊕: keine Speicherbereinigung nötig (da keine externe Fragmentierung)

⊕: virtuelle Adressierung benötigt nur Verdrängungs- und Nachschubsstrategie

⊖: interne Fragmentierung

⊖: hoher Verwaltungsaufwand

Abbildung logische Adresse auf physikalische Adresse:



Adressaufteilung bei streuender Adressbildung, Beispiel:

$n=3$ 2-Bit Prozessor, 2 GB physikalischer Speicher, Kachelgröße 4 kB

Logische Adresse: n Bit \rightarrow hier: 32 Bit
Physikalische Adresse: \log_2 (Größe physikal. Speicher in Byte)

\hookrightarrow hier: 31 bit, da $2GB = 2 \cdot 2^{30}$ Bit = 2^{31} Byte

Offsetadresse: \log_2 (Kachelgröße in Byte)
 \hookrightarrow hier: 12 bit, da $4KB = 2^2 \cdot 2^{10} = 2^{12}$ Byte

Seitenadresse: $n - \text{Offset}$
 \hookrightarrow hier: $32 - 12 = 20$ Bit

Basisadresse: physikalisch-offset
 \hookrightarrow hier: $31 - 12 = 19$ Bit

Speicherverwaltung: Vor- und Nachteile einer kleinen Kachelgröße

⊕: feinere Granularität

⊕: geringere interne Fragmentierung

⊖: hoher Verwaltungsaufwand

⊖: Seitenliste deutlich größer

Abhilfe: mehrstufige, hierarchische Seitenverwaltung

Synchronisationsmechanismen der I/O-Verwaltung:

- Polling (Warteschleife bis Gerät Bereitschaft signalisiert)
 - \hookrightarrow einfach, aber rechenzeitintensiv
- Busy-Waiting (wie Polling, aber Wartezeiten werden genutzt)
 - \hookrightarrow nur kurze Aktivitäten möglich, schwierige Realisierung
- Interrupts
 - \hookrightarrow schnelle Reaktion, kein unnötiger Rechenzeitverbrauch
 - \hookrightarrow etwas Overhead durch Aktivierung der Interrupt-Service-Routine
- Handshaking (Synchronisation durch Ready- und Ack-Signale)
 - \hookrightarrow sinnvoll, falls Gerät schneller als der Task ist

Anforderungen an Echtzeit-Middleware:

- Wahrung der End-zu-End Prioritäten
- Definition von Zeitbedingungen bzw. Zeitschränken
- echtzeitfähige Funktionsaufrufe und Ereignisbehandlung
- Unterstützung des Echtzeit-Schedulings

Middleware für verteilte Systeme

- Software Schicht über der Betriebssystemebene, welche die Entwicklung verteilter Systeme unterstützt
- \hookrightarrow soll von den einzelnen heterogenen Systemen abstrahieren und der Anwendung eine einheitliche Schicht zur Verfügung stellen

Aufgaben Middleware:

- Identifikation (Verteilte Komponenten müssen systemweit eindeutig identifizierbar sein)
- Lokalisierung (mit Hilfe der eindeutigen Identität kann die Middleware den Ort jeder Komponente im verteilten System auffinden)
- Interaktion (Mechanismen zur Interaktion von Komponenten, z.B. Übertragung von Nachrichten, Erteilung von Aufträgen, Aufruf entfernter Methoden)
- Erzeugung und Vernichtung (neue Komponenten können erzeugt bzw. bestehende vernichtet werden)
- Fehlerbehandlung (z.B. Exception-Management)

\hookrightarrow Die Anwendung muss sich hierdurch nicht mehr um die Aspekte der Verteilung kümmern!

⊕ Transparenz

⊕ Interoperabilität
(da Unterschiede der einzelnen Systeme, Programmiersprachen etc. verborgen werden)

⊕ Vereinfachung der Anwendungsentwicklung
⊕ Wartbarkeit, Testbarkeit, Portierbarkeit

⊕ Flexibilität, Anpassung des Systems an aktuelle Gegebenheiten durch Verteilung von Aufgaben

⊖ zusätzlicher Overhead

Middleware: Transparenz

- ↳ Verbergen spezifischer Eigenschaften von Hardware, Betriebssystem und Kommunikationssystem sowie der physikalischen Verteilung vor der Anwendung
- ⇒ Die Anwendung wird unabhängig von der Struktur des verteilten Systems
- ⇒ Man kann mit dem verteilten System unabhängig von Zeit und Ort auf einheitliche, konsistente Weise interagieren

Arten von Transparenz

- Access (Zugriffstransparenz)
- Concurrency (Nebenläufigkeitstransparenz)
- Location (Ortstransparenz)
- Migration (Verlagerungstransparenz)
- Replication (Replikationstransparenz)
- Failure (Fehler- und Ausfalltransparenz)

Anforderungen an Echtzeit-Middleware:

- Wahrung der End-zu-End-Prioritäten
 - ↳ Prioritätskette darf nicht unterbrochen werden
 - ↳ Vermeidung von Prioritäteninversion
- Definition von Zeitbedingungen / -schranken
 - ↳ Zeitanforderungen können spezifiziert werden und werden von der Middleware eingehalten
- Echtzeitfähige Funktionsaufrufe
 - ↳ Die Ausführungszeit (WCET) für Funktionsaufrufe der Middleware muss bekannt und beschränkt sein
- Echtzeitfähige Ereignisbehandlung
 - ↳ Auf Ereignisse muss in definierter Zeit reagiert werden
- Unterstützung des Echtzeitschedulers

Vorteile der Mikrokernarchitektur einer Middleware wie OSA+

- Anpassbarkeit, Skalierbarkeit
 - ↳ auf kleinen Systemen müssen nur wirklich benötigte Dienste geladen werden
 - ↳ zusätzliche Ressourcen größerer Systeme können bei Bedarf genutzt werden
- Erweiterbarkeit
 - ↳ zusätzliche Dienste erweitern die Funktionalität der Middleware
- Portierbarkeit
 - ↳ der Kern ist hardwareunabhängig

Aufgabe 4.2: Speicherprogrammierbare Steuerung

SPS-Programmiersprachen

graphisch

KOP
(Kontaktplan)

FUP
(Funktionsplan)

textuell

AWL
(Anweisungsliste)

ST
(Strukturierter Text)

Kontaktplan (KOP)



Halteglied

Schließer: Schließt Strompfad bei E=1 und öffnet Strompfad bei E=0

Öffner: Öffnet Strompfad bei E=1 und schließt Strompfad bei E=0

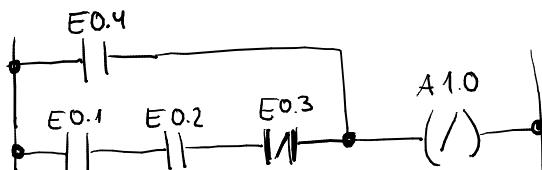
Zuweisung: Weist bei Stromfluss A=1 zu und weist ohne Stromfluss A=0 zu

Zuweisung: Weist bei Stromfluss A=0 zu und weist ohne Stromfluss A=1 zu

Zuweisung: Weist bei Stromfluss M=1 zu (Marker setzen)

Zuweisung: Weist bei Stromfluss M=0 zu (Marker löschen)

Seierschaltung entspricht logischer UND-Verknüpfung
Parallelschaltung entspricht logischer ODER-Verknüpfung



$$A1.0 := \neg(E0.4 \vee (E0.1 \wedge E0.2 \wedge \neg E0.3))$$

Strukturierter Text (ST):

NOT

OR

&

$$A1.0 := (\text{NOT } E0.4 \& \text{NOT } E0.3) \text{ OR } (E0.1 \& \text{NOT } E0.2);$$

$$A1.0 := (\text{NOT } E0.4 \& \text{NOT } E0.3) \text{ OR } (E0.1 \& \text{NOT } E0.2);$$

Beispiel: Das ist alles dasselbe:

U E0.1

U A0.1

= M1.0

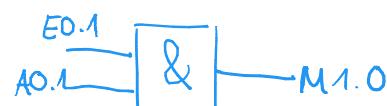
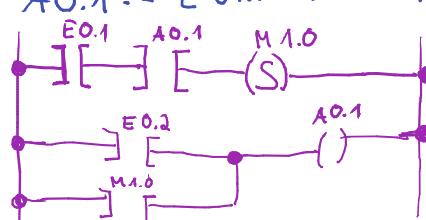
O E0.2

O M1.0

= A0.1

M1.0 := E0.1 & A0.1;

A0.1 := E0.2 OR M1.0;



Abkürzungen:

SPS (Speicherprogrammierbare Steuerung)

NC (Numerical Control)

PLC (Programmable Logic Controller)

RC (Robot Control)

Vorteile Kaskadenregelung:

↳ Vereinfachung des Reglerentwurfs durch mehrere Hilfsregelkreise, die unabhängig voneinander betrachtet werden können

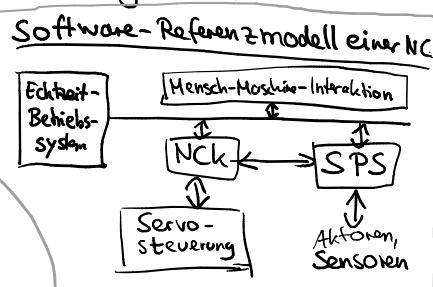
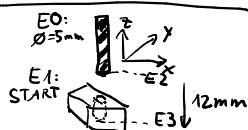
↳ Störgrößen am Eingang der Regelstrecke werden durch den schnellen Hilfsregelkreis ausgeregelt, bevor sich diese auf den Hauptregelkreis auswirken können

↳ Verzögerungen werden verkleinert, wodurch die Regelfähigkeit des Systems verbessert wird

Aufgaben und Zusammenhang SPS und NC:

Sowohl SPS als auch NC werden zur Steuerung von Werkzeugmaschinen verwendet, wobei NC auf diese beschränkt ist und mit einer SPS ebenfalls andere Maschinen und Prozesse gesteuert werden können. Als Zusammenhang könnte man NC als Vorfürer der SPS sehen. Häufig wird bei einer NC eine SPS für erweiterte Funktionalitäten wie z.B. Werkzeugwechsel genutzt.

Standbohrmaschine:



Verarbeitungsschritte SPS zyklischer Programmablauf:

- ① Einlesen der Eingänge
- ② Ausführen des Automatisierungsprogramms
- ③ Bereitstellen der Ergebnisse an den Ausgängen

Graphische vs. textuelle Programmiersprachen:

SPS: Herstellerspezifisch
(Entwicklung und Produktion)

soft-SPS: Emuliert auf PC

- ⊕ intuitiver zu bedienen, evtl. auch von Endbenutzern
- ⊕ anschaulicher und übersichtlicher (höheres Abstraktionsniveau)
- ⊖ Probleme mit der Darstellung von Kontrollstrukturen wie Schleifen oder Fallunterscheidungen
⇒ begrenzte Ausdrucksfähigkeit
- ⊖ Übersichtlichkeit großer/komplexer Programme ist nicht gegeben