

Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

1. Einführung und Grundlagen



<http://gymnasium-blomberg.de/wp-content/uploads/2013/05/Wegweiser.jpg>

Inhalt

1. Einführung und Grundlagen

- 1.1 Motivation
- 1.2 Einführung
- 1.3 Begriffe

Motivation: ESP in der Presse

Quelle: Tagesanzeiger (Schweiz) vom 12.11.1997

Auslieferungsstopp für A-Klasse

Mercedes unterbricht für drei Monate die Auslieferung des kompakten A-Klasse-Fahrzeugs. In dieser Zeit sollen erhebliche Änderungen vorgenommen werden.

Von Marcel Waeber

100 000 Vorbestellungen lagen vor, als vor gut zwei Wochen beim Mercedes der A-Klasse gravierende Sicherheitsmängel publik wurden. Das Fahrzeug war bei einem sogenannten Elchtest (einem abrupten Ausweichmanöver) gekippt, und eine Testperson war dabei verletzt worden (TA vom Montag). In der Folge wurden die ersten Verkaufsverträge rückgängig gemacht. Jetzt reagiert der Daimler-Benz-Konzern auf die Stornierungen.

Neue technische Lösungen

„Wir nehmen die öffentlich geäusserte Kritik und vor allem die Sorgen unserer Kunden sehr ernst und bedauern die gezeigte Schwäche der A-Klasse in einem Extremtest“, sagte am Dienstag der Vorstandsvorsitzende Jürgen E. Schrempp. Gleichzeitig kündigte er Lösungen für die technischen Probleme des „Baby-Benz“ an: neue Stabilisatoren, neue Feder- und Dämpferabstimmung an den Achsen, die Karosserie wird tiefergelegt, und die Reifen werden neu dimensioniert.

Wie Jürgen Hubbert, verantwortlich für den Bereich Personenwagen, ergänzend ausführte, soll die A-Klasse zusätzlich serienmäßig mit dem Electronic Stability Program (ESP) ausgerüstet werden. Rechnergesteuert greift das ESP gezielt in das Bremsystem ein und hält das Fahrzeug bei Ausweichmanövern und Kurvenfahrten stabil. Das Schleuderrisiko wird dadurch massiv reduziert. Hubbert: „Situationen bei Eis, Schnee und Nässe lassen sich mit dem ESP beherrschen.“

Copyright ©TA-Media AG

Motivation: ESP in der Presse

Quelle: Tagesanzeiger (Schweiz) vom 12.11.1997

Auslieferungsstopp für A-Klasse

Mercedes unterbricht für drei Monate die Auslieferung des kompakten A-Klasse-Fahrzeugs. In dieser Zeit sollen erhebliche Änderungen vorgenommen werden.

Von Marcel Waeber

100 000 Vorbestellungen lagen vor, als vor gut zwei Wochen beim Mercedes der A-Klasse gravierende Sicherheitsmängel publik wurden. Das Fahrzeug war bei einem sogenannten Elchtest (einem abrupten Ausweichmanöver) gekippt, und eine Testperson war dabei verletzt worden (TA vom Montag). In der Folge wurden die ersten Verkaufsverträge rückgängig gemacht. Jetzt reagiert der Daimler-Benz-Konzern auf die Stornierungen.

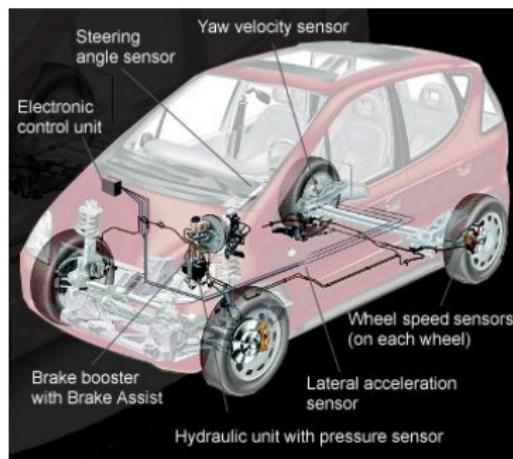
Neue technische Lösungen

„Wir nehmen die öffentlich geäusserte Kritik und vor allem die Sorgen unserer Kunden sehr ernst und bedauern die gezeigte Schwäche der A-Klasse in einem Extremtest“, sagte am Dienstag der Vorstandsvorsitzende Jürgen E. Schrempp. Gleichzeitig kündigte er Lösungen für die technischen Probleme des „Baby-Benz“ an: neue Stabilisatoren, neue Feder- und Dämpferabstimmung an den Achsen, die Karosserie wird tiefergelegt, und die Reifen werden neu dimensioniert.

Wie Jürgen Hubbert, verantwortlich für den Bereich Personenwagen, ergänzend ausführte, soll die A-Klasse zusätzlich serienmäßig mit dem Electronic Stability Program (ESP) ausgerüstet werden. Rechnergesteuert greift das ESP gezielt in das Bremsystem ein und hält das Fahrzeug bei Ausweichmanövern und Kurvenfahrten stabil. Das Schleuderrisiko wird dadurch massiv reduziert. Hubbert: „Situationen bei Eis, Schnee und Nässe lassen sich mit dem ESP beherrschen.“

Copyright ©TA-Media AG

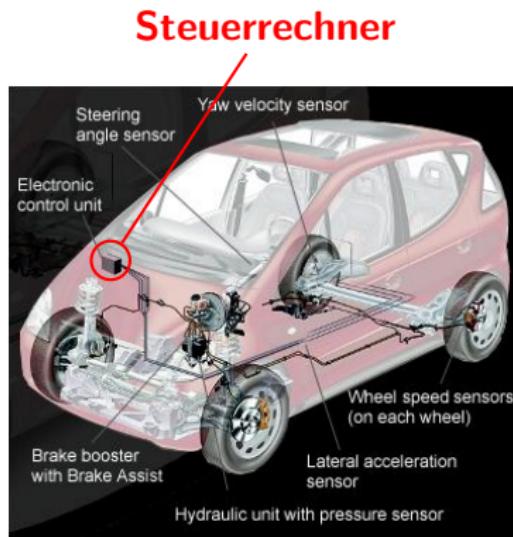
ESP-Hardware



Beispiel: ESP (Elektronisches Stabilitäts-Programm)

- Sensoren erfassen Beschleunigung und Winkelgeschwindigkeiten
- Einflussnahme auf Bremsen (einzelnen ansteuerbar)
- Steuerrechner implementiert Funktionalität

ESP-Hardware



Beispiel: ESP (Elektronisches Stabilitäts-Programm)

- Sensoren erfassen Beschleunigung und Winkelgeschwindigkeiten
- Einflussnahme auf Bremsen (einzelnen ansteuerbar)
- Steuerrechner implementiert Funktionalität

ESP-Software

Sensor-Überprüfung alle 20 Millisekunden

Das zentrale ESP-Steuergerät besteht aus zwei Rechnern mit einer Speicherkapazität von jeweils 56 kByte. Zum Vergleich: ABS benötigt nur ein Viertel dieser Rechnerleistung. ESP® nutzt diese großen Rechnerkapazitäten, um unter anderem laufend die einzelnen System-Komponenten zu überprüfen. So wird beispielsweise der wichtige Sensor zur Erfassung der Drehgeschwindigkeit des Fahrzeugs nach jedem Messvorgang zusätzlich kontrolliert - im Rhythmus von 20 Millisekunden.

©DaimlerChrysler, 1998

Beispiel: ESP (Elektronisches Stabilitäts-Programm)

- Digitale Regelung
- Zeitverhalten muss vorhersagbar („deterministisch“) sein.
- „Eingebettetes System“ (*embedded System*)
⇒ Mit klassischen PCs wohl kaum machbar.

ESP-Software

Sensor-Überprüfung alle 20 Millisekunden

Das zentrale ESP-Steuergerät besteht aus zwei Rechnern mit einer Speicherkapazität von jeweils 56 kByte. Zum Vergleich: ABS benötigt nur ein Viertel dieser Rechnerleistung. ESP® nutzt diese **großen Rechnerkapazitäten**, um unter anderem laufend die einzelnen System-Komponenten zu überprüfen. So wird beispielsweise der wichtige Sensor zur Erfassung der Drehgeschwindigkeit des Fahrzeugs nach jedem Messvorgang zusätzlich kontrolliert - **im Rhythmus von 20 Millisekunden**.

©DaimlerChrysler, 1998

Beispiel: ESP (Elektronisches Stabilitäts-Programm)

- Digitale Regelung
- Zeitverhalten muss vorhersagbar („deterministisch“) sein.
- „Eingebettetes System“ (*embedded System*)
⇒ Mit klassischen PCs wohl kaum machbar.

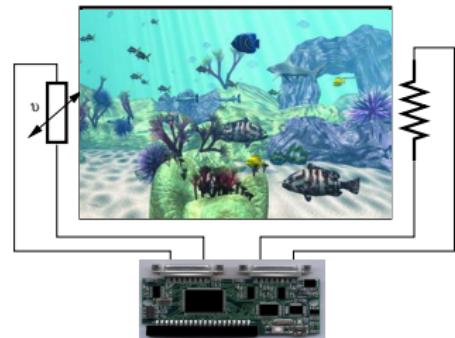
Einführung

Typische Anwendungsgebiete für Echtzeitsysteme:

- oft Problemstellungen der Mess- / Steuer- / Regelungstechnik
- Interaktion von Computern mit physikalischen/ technischen Systemen (mithilfe von Sensoren und Aktoren)
- phys./ techn. System gibt Zeitanforderungen vor
- Computer muss diesen Zeitanforderungen unter allen Umständen gerecht werden

Beispiel: Aquarium

- Technisches System: Fischtank
- Sensor: Temperaturfühler
- Aktor: Heizung
- Embedded System
(Regel-Algorithmus)



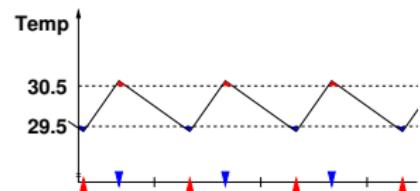
Beispiel: Aquarium

Einfacher Algorithmus

```

while TRUE do
    read(&WaterTemp);
    if WaterTemp < 29.5 then
        heater(TRUE);
    else
        if WaterTemp > 30.5 then
            heater(FALSE);
        end
    end
    wait(DELAY);
end

```



- In festen Zeitabständen prüfen:
 - ▶ Temp. zu hoch → Heizung aus
 - ▶ Temp. zu niedrig → Heizung an

Wichtig: Physikalisches System bestimmt Echtzeitanforderung

- Zeitabstände zu groß -> Fehler

„Zeitgesteuertes Echtzeitsystem“

Beispiel: Aquarium

Einfacher Algorithmus

```

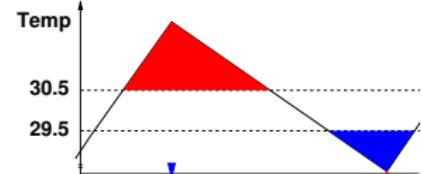
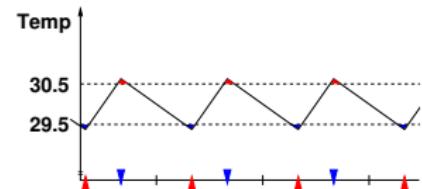
while TRUE do
    read(&WaterTemp);
    if WaterTemp < 29.5 then
        heater(TRUE);
    else
        if WaterTemp > 30.5 then
            heater(FALSE);
        end
    end
    wait(DELAY);
end
  
```

- In festen Zeitabständen prüfen:
 - ▶ Temp. zu hoch → Heizung aus
 - ▶ Temp. zu niedrig → Heizung an

Wichtig: Physikalisches System bestimmt Echtzeitanforderung

- Zeitabstände zu groß -> Fehler

„Zeitgesteuertes Echtzeitsystem“



Beispiel: Aquarium

Einfacher Algorithmus

```

while TRUE do
    read(&WaterTemp);
    if WaterTemp < 29.5 then
        heater(TRUE);
    else
        if WaterTemp > 30.5 then
            heater(FALSE);
        end
    end
    wait(DELAY);
end

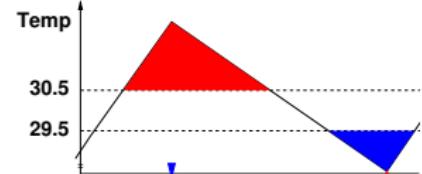
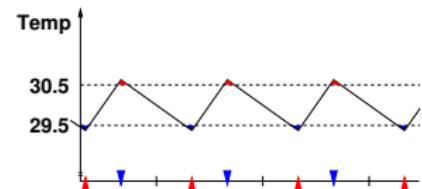
```

- In festen Zeitabständen prüfen:
 - ▶ Temp. zu hoch → Heizung aus
 - ▶ Temp. zu niedrig → Heizung an

Wichtig: Physikalisches System bestimmt Echtzeitanforderung

- Zeitabstände zu groß -> Fehler

„Zeitgesteuertes Echtzeitsystem“



„Echtzeit“ heißt nicht „echt schnell“!

- Zu beachten: „Echtzeitverhalten“ beinhaltet keine Aussage über absolute Geschwindigkeit

- Anwendung gibt Zeitschranken vor
- Beispiel: Airbag
 - ▶ Zeitschranken im Millisekunden-Bereich
 - ▶ „Ereignisgesteuert“ (Ereignis: Aufprall)
 - ▶ Sowohl zu frühes als auch zu spätes Auslösen kann fatal sein



<http://www.citroenet.org.uk/publicity-brochures/schiffer/schiffer.html>

- (Zugleich ein Beispiel für ein sicherheitskritisches System)

Software-Entwicklung

Sicherheitskritische Bereiche erfordern oft den Einsatz von Echtzeitsystemen

- Prozessleittechnik
 - ▶ Walzstraßen, Chemiewerke, (Kern-)Kraftwerke, ...
- Fahrzeugtechnik
 - ▶ ABS, ESP, Steer-by-wire, Motorsteuerung, ...
- Avionik
 - ▶ Autopilot, Fly-by-wire, ...
- Fehleranfälligkeit wächst mit steigender Komplexität der Hard- und Software
- Gefährdungspotenzial eines Systems ⇒ „safety“-Anforderungen

⇒ Systematisches Testen gefordert
(z.B. DO-178, Richtlinien der Flugzeugindustrie)

Software-Entwicklung

Sicherheitskritische Bereiche erfordern oft den Einsatz von Echtzeitsystemen

- Prozessleittechnik
 - ▶ Walzstraßen, Chemiewerke, (Kern-)Kraftwerke, ...
 - Fahrzeugtechnik
 - ▶ ABS, ESP, Steer-by-wire, Motorsteuerung, ...
 - Avionik
 - ▶ Autopilot, Fly-by-wire, ...
 - Fehleranfälligkeit wächst mit steigender Komplexität der Hard- und Software
 - Gefährdungspotenzial eines Systems ⇒ „safety“-Anforderungen
- ⇒ **Systematisches Testen gefordert**
(z.B. DO-178, Richtlinien der Flugzeugindustrie)

Beispiel: Sicherheitsstandard DO-178B

- Anforderungen der US-amerikanischen Bundesbehörde für Flugsicherheit (FAA) zur Zulassung von Systemen für den Einsatz in der zivilen Luftfahrt
- Definiert 5 „Integrity Levels“:

Level	Beschreibung	Anforderung
A	Failure would cause or contribute to a catastrophic failure of the aircraft	Level B + 100 % Modified Condition Decision Coverage
B	Failure would cause or contribute to a hazardous/severe failure condition	Level C + 100 % Decision Coverage
C	Failure would cause or contribute to a major failure condition	Level D + 100 % Statement (or line) Coverage
D	Failure would cause or contribute to a minor failure condition	Level D + 100 % Requirement Coverage
E	Failure would no effect on the aircraft or on pilot workload	No Coverage Requirements

⇒ Forderung nach systematischen Verfahren des Software Engineering entsprechender Systeme

Beispiel: Sicherheitsstandard DO-178B

- Anforderungen der US-amerikanischen Bundesbehörde für Flugsicherheit (FAA) zur Zulassung von Systemen für den Einsatz in der zivilen Luftfahrt
- Definiert 5 „Integrity Levels“:

Level	Beschreibung	Anforderung
A	Failure would cause or contribute to a catastrophic failure of the aircraft	Level B + 100 % Modified Condition Decision Coverage
B	Failure would cause or contribute to a hazardous/severe failure condition	Level C + 100 % Decision Coverage
C	Failure would cause or contribute to a major failure condition	Level D + 100 % Statement (or line) Coverage
D	Failure would cause or contribute to a minor failure condition	Level D + 100 % Requirement Coverage
E	Failure would no effect on the aircraft or on pilot workload	No Coverage Requirements

⇒ Forderung nach systematischen Verfahren des Software Engineering entsprechender Systeme

Definitionsversuch „Embedded System“



Was ist ein Embedded System ?

- Ein Rechner, der aufgrund seiner Programmierung die Funktion eines bestimmten Gerätes erfüllt (*embedded = eingebettet*)
- Rechner ist i.d.R physisch in das Gerät mit Sensoren und Aktoren integriert
- Beispiele: Telefon (Smartphone: Grenzfall), CD-Player, Mpeg/MP3-Player, Spielautomat, Scannerkasse, Motorsteuerung, ABS, Drohne, ...
- Falls User Interface vorhanden, auch aktueller Begriff „Cyber-Physical System“ genutzt



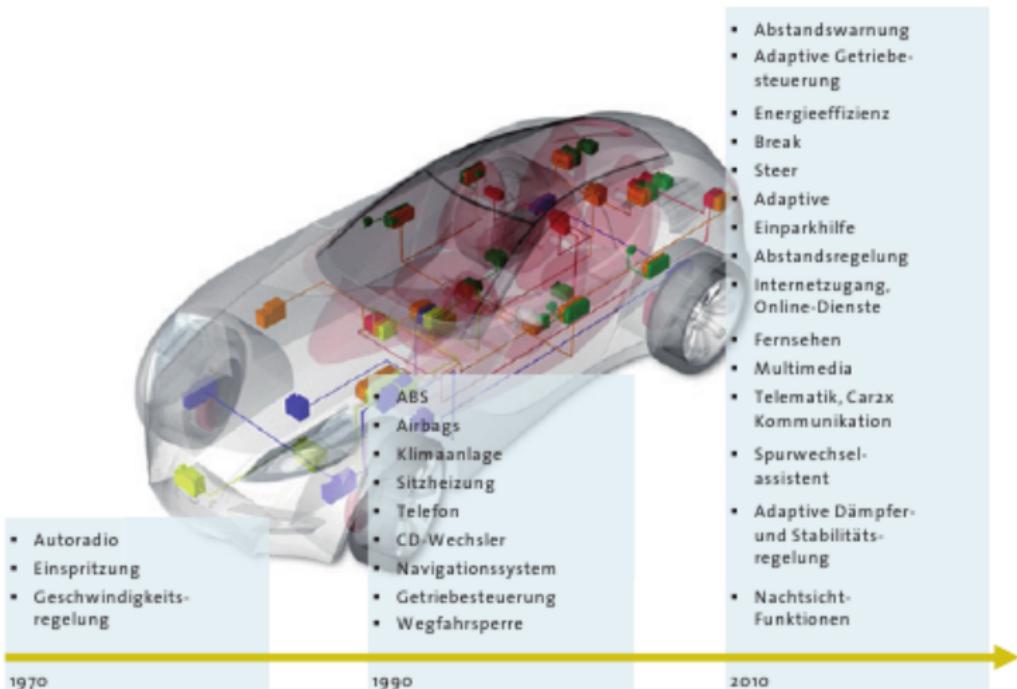
Definitionsversuch „Embedded System“ (2)

Anforderungen an Embedded Systems

- Häufig feste, statische Software („Firmware“)
- robust (Verhalten bei störenden Einflüssen)
- kompakt, portabel, häufig keine bewegten Teile
- häufig Massenprodukte → geringer Preis
- batteriebetrieben → geringer Energieverbrauch
- „24/7“-Betrieb → zuverlässig, fehlertolerant
- u.U. lange Lebensdauer (z.B. 30+ Jahre)
- u.U. Sicherheitskritische („safety-critical“) Anwendungen

Oftmals (aber nicht immer): echtzeitfähig

Immer mehr Anwendungen



Quelle: BITKOM – Eingebettete Systeme – Ein strategisches Wachstumsfeld für Deutschland (2010)

Embedded Systems als Wachsender Markt

Beispiel Automobilindustrie

- Teilweise mehr als 100 Prozessoren in einem Fahrzeug
- Haupt-Einsatzgebiet: Steuerung/Regelung
 - ▶ früher Oberklasse-Modellen vorbehalten
 - ▶ heute: z.B. ABS / ESP für die meisten Modelle erhältlich
- 25% der Gesamtkosten, bald bis zu 50%
- Erheblicher Teil der Kosten liegt in Software
 - ▶ z.B. ABS: 10.000 Zeilen C-Code
 - ▶ z.B. Klimaanlage: 25.000 Zeilen C-Code
 - ▶ Bereits > 100 MB Software auf allen Controllern

Klassifikation Embedded SW: Low-End

Low-End Embedded Control Anwendungen

- basierend auf 4/8-bit Prozessoren, μ Controller oder Festpunkt-Signalprozessoren
- geringe Anwendungskomplexität
- i.d.R. keine Notwendigkeit für Betriebssystem
- steigender Software-Umfang in Lösungen
- Über lange Zeit steigender Markt für Software
(1995: 4 Mrd\$, 2010: 749 Mrd\$, 2011: 713 Mrd\$)*
- Marktanteil der low-end Anwendungen sinkt gegenüber High-End Anwendungen

* Quelle: IDC-Studie „Intelligent Systems: The Next Big Opportunity“ (2011)

Klassifikation Embedded SW: High-End

High-End Embedded Control Anwendungen

- „Intelligente Systeme“:
- Ein oder mehrere 16/32/64-bit Prozessoren, μ Controller oder Gleitpunkt-Signalprozessoren
- (64-bit Anteil noch gering)
- mittlere bis hohe Anwendungskomplexität
- Einsatz von Echtzeitbetriebssystemen, Softwareentwicklungswerkzeugen, etc.
- Markt für Software-Zulieferer
- stark wachsender Markt (2010: 865 Mrd\$, 2011: 1075 Mrd\$)*
- Marktanteil der High-End Anwendungen steigt stark an (Migration aus dem low-end Bereich)

* Quelle: IDC-Studie „Intelligent Systems: The Next Big Opportunity“ (2011)

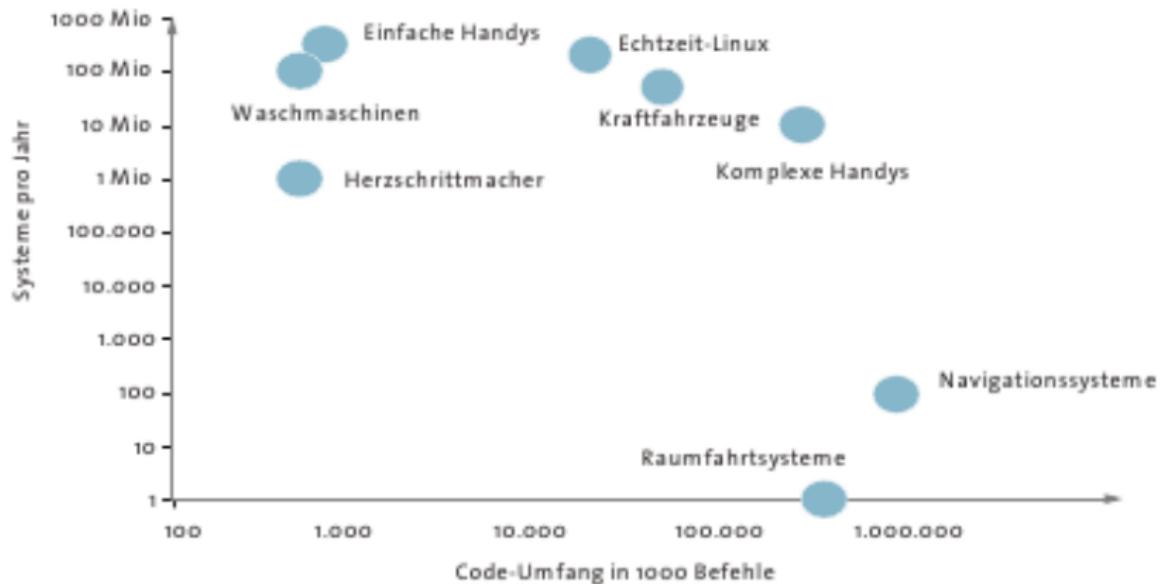
Klassifikation Embedded SW: High-End

High-End Embedded Control Anwendungen

- „Intelligente Systeme“:
- Ein oder mehrere 16/32/64-bit Prozessoren, μ Controller oder Gleitpunkt-Signalprozessoren
- (64-bit Anteil noch gering)
- mittlere bis hohe Anwendungskomplexität
- Einsatz von Echtzeitbetriebssystemen, Softwareentwicklungswerkzeugen, etc.
- Markt für Software-Zulieferer
- stark wachsender Markt (2010: 865 Mrd\$, 2011: 1075 Mrd\$)*
- Marktanteil der High-End Anwendungen steigt stark an (Migration aus dem low-end Bereich)

* Quelle: IDC-Studie „Intelligent Systems: The Next Big Opportunity“ (2011)

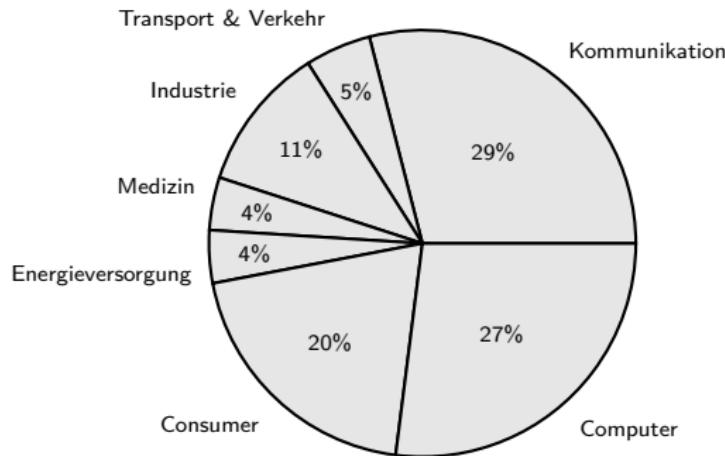
Komplexität von Embedded Software



* Quelle: BITKOM – Eingebettete Systeme – Ein strategisches Wachstumsfeld für Deutschland (2010)

Märkte der Embedded Systems

Anteile am Gesamterlös im Jahr 2011 (1.789 Mrd \$)*



* Quelle: IDC-Studie „Intelligent Systems: The Next Big Opportunity“ (2011)

Anwendungsgebiete

Hauptanwendungen im Segment Office Automation:

- Drucker (Laser, Tinte)
- Digitale Kopierer, Fax-Geräte, Dokumenten-Scanner
- Terminals (X, NCs, ThinClients)

Hauptanwendungen im Consumer-Segment:

- Unterhaltung (CD-/MP3-Player), Spielekonsolen
- Digitale Audio- und Video-Systeme, Interaktives TV (Set-Top-Boxen)
- Haushaltsgeräte (z.B. Mikrowelle)
- Haus-Management (Smart Home) (z.B. Heizungssteuerung, Alarmanlage)

Militärische Anwendungen:

- Große Investitionen, Förderung von Innovationen (ohne Wertung)

Anwendungsgebiete (2)

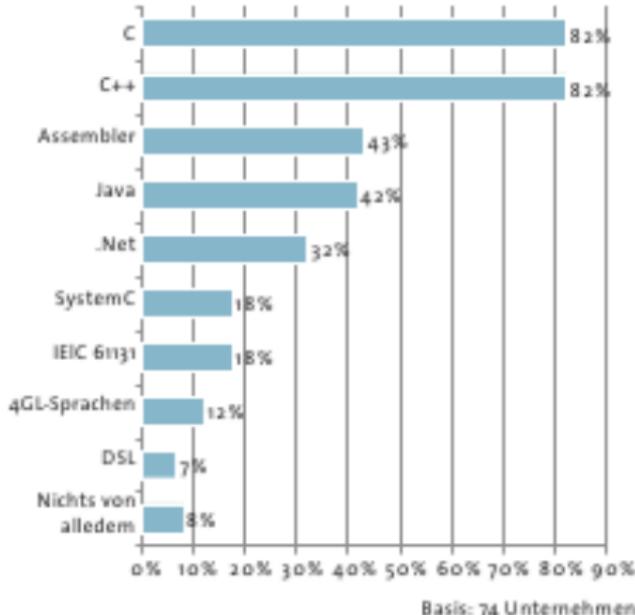
Hauptanwendungen im Transport-Segment:

- Luftfahrt (Flugzeug, Air Traffic Control)
- Automobil-Elektronik
- Schienenfahrzeuge (ICE)
- Verkehrssteuerungen (Schienenverkehr, Ampelanlagen, Toll Collect)

Hauptanwendungen im Kommunikations-Segment:

- Nebenstellenanlagen
- IP- und sonstige Telefone, digitale Anrufbeantworter
- Anruf-Verteilsysteme (Automatic Call Distribution)
- Voice Messaging
- Netzwerkgeräte zur Datenübertragung (z.B. Router)

Arbeitsmarkt



- Embedded Markt in DE:
 - > 19 Mrd € in 2010
 - ▶ 8% konstantes Wachstum
 - ▶ 85% des Umsatzes in EU
 - ▶ 69% des Umsatzes in DE
- Know-How wird in EU/DE gehalten
- Hardwarenahe Programmiersprachen gefordert

Quelle: BITKOM Studie zur Bedeutung des Sektor Embedded-Systeme in Deutschland

Begriffe

Echtzeitbetrieb (Def. nach DIN 44300, 1985):

- Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig bereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.
- ⇒ korrektes Systemverhalten erfordert damit auch, dass zeitliche Vorgaben eingehalten werden.

Deadline (Frist):

- Zeitpunkt, zu dem die Verarbeitungsergebnisse vorliegen müssen.

Begriffe

Echtzeitbetrieb (Def. nach DIN 44300, 1985):

- Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig bereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.
- ⇒ korrektes Systemverhalten erfordert damit auch, dass zeitliche Vorgaben eingehalten werden.

Deadline (Frist):

- Zeitpunkt, zu dem die Verarbeitungsergebnisse vorliegen müssen.

Begriffe: Deutsch / Englisch

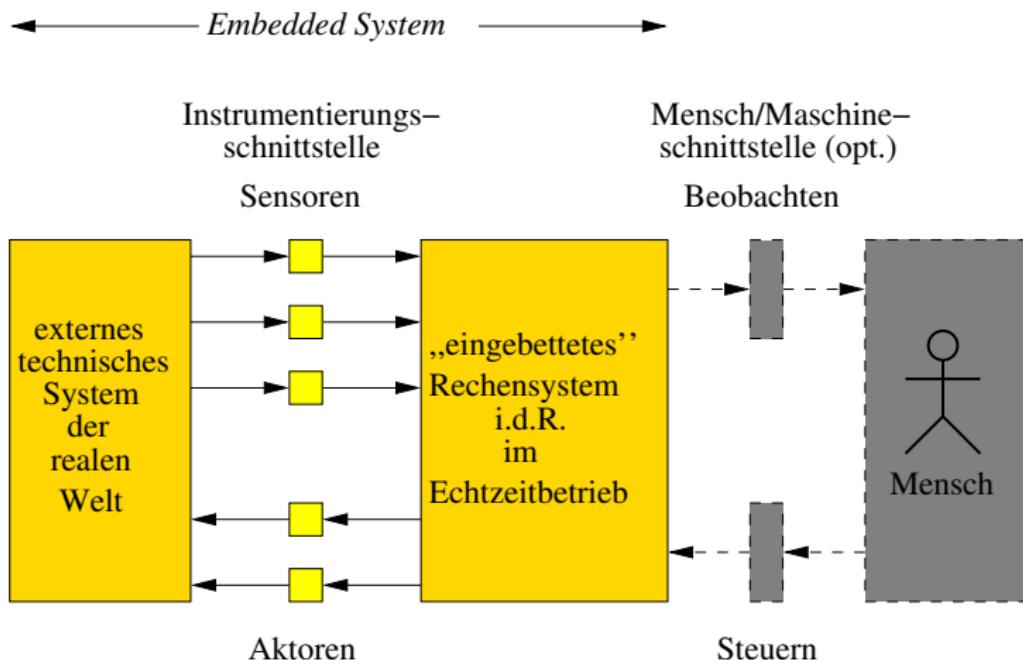
Deutsch:

Echtzeitsystem
Echtzeitbetriebssystem
eingebettetes Rechensystem
Frist
Rechtzeitigkeit
Ausführungsplanung, -plan
Sensoren
Aktoren
Mensch/Maschine-Schnittstelle
Steuerung

Englisch:

- real-time system (RTS)
- real-time operating system (RTOS)
- embedded (computer) system
- deadline
- timeliness
- scheduling, schedule
- sensors
- actuators, actors
- man/machine (operator) interface
- control

Grundmodell eines Embedded Systems



Spezialfälle

Monitoring:

- nur Sensoren, keine Aktoren

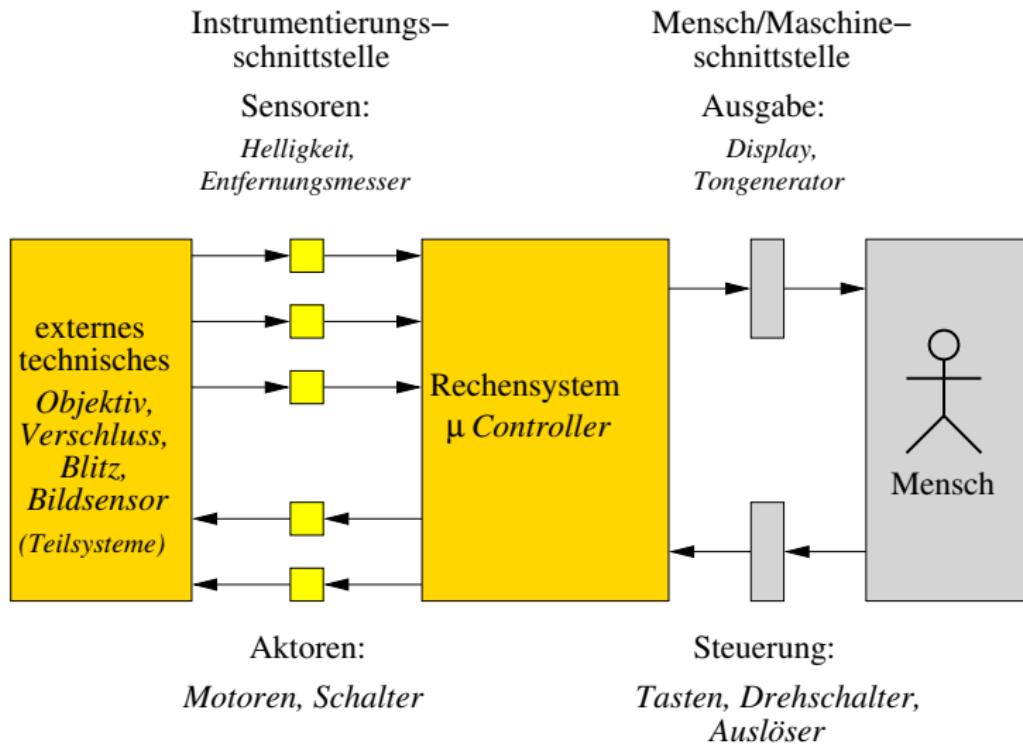
Open Loop:

- nur Aktoren, keine Sensoren

Feedback Control:

- Sensoren und Aktoren

Beispiel: Digital gesteuerte Kamera



Reaktives System

Das Verhalten des Systems ist ereignisgetrieben.

- Auf jedes Ereignis (Stimulus) der externen Welt, das an den Sensoren sichtbar wird, erfolgt eine spezifische Aktion (Reaktion).

Beispiele:

- Parkscheinautomat (reakтив, kein Echtzeitbetrieb)
- Airbag (reakтив, Echtzeitbetrieb)

Zeitgetriebenes System

Das Verhalten des Systems wird durch zeitliche Größen kontrolliert, z.B.

- Absolute Zeitpunkte (der realen Außenzeit UTC)
- Mission Time (relativ zum Beginn der Mission)
- Zeitintervalle, -dauern
- Periode (wiederkehrende Aktion)

Beispiele:

- öffentlicher Nahverkehr (zeitgetrieben, kein Echtzeitbetrieb)
- ABS (zeitgetrieben, Echtzeitbetrieb):
 - ▶ „Schau alle 5 ms, ob Bremse getreten. Wenn ja, bewirke einen Bremsvorgang innerhalb von 20 ms“.
 - ▶ „Schau alle 5 ms, ob Räder blockieren. Wenn ja, öffne die Bremse sofort für 100 ms“.
 - ▶ ...

Systemlast

Lastannahme:

- definiert die angenommene Spitzenbelastung (z.B. Anzahl Ereignisse je sec), die durch das externe System erzeugt wird.

Achtung:

- Mittelwerte sind unbrauchbar!
- Statistische Argumente bzgl. einer geringen Wahrscheinlichkeit des Auftretens unabhängiger Ereignisse sind unzulässig.
- In kritischen Situationen treten häufig sehr viele, zeitlich eng korrelierte Ereignisse auf (\rightarrow „Ereignissturm“).

Fehler

Fehlerannahme:

- definiert Art und Anzahl der angenommenen Fehler sowie welche Funktionalität das System unter diesen Annahmen aufrecht erhält.

Achtung:

- Das System muss im schlimmsten Fall (Worst Case) die maximale Anzahl von Fehlern bei Spitzenlast handhaben können.

Geltungsbereich der Annahmen (Assumption Coverage):

- Wahrscheinlichkeit, dass die gemachten Annahmen mit der Wirklichkeit übereinstimmen.

Vorhersagbarkeit

Vorhersagbarkeit (Predictability):

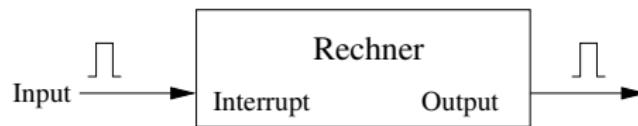
- bedeutet, dass das Systemverhalten bei gegebenen
 - ▶ Lastannahmen und
 - ▶ Fehlerannahmen
- in Hinblick auf
 - ▶ Funktionalität
 - ▶ zeitliches Verhalten (Rechtzeitigkeit) und
 - ▶ Verlässlichkeit (Dependability)
- für den worst case eingehalten wird.
→ Es wird damit "vorhersagbar".
- Garantiertes Systemverhalten muss nicht optimal sein.
- Häufig nimmt man z.B. eine etwas schlechtere Antwortzeit in Kauf, wenn man sicher ist, dass diese bestimmt nicht überschritten wird.

Zeitverhalten

Wesentlich: deterministisches Zeitverhalten

- zeitgesteuertes System: Zeitplan einhalten (kein „Jitter“)
- ereignisgesteuertes System: Antwortzeit einhalten

Gedankenexperiment zur Antwortzeit:



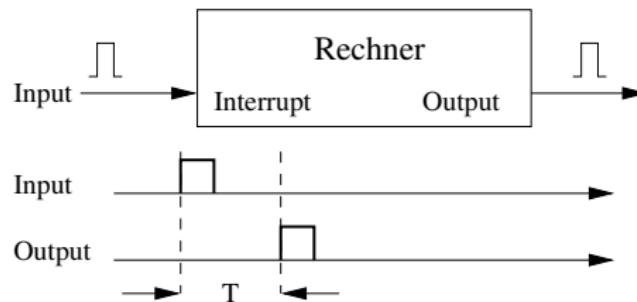
T Konstant?

Zeitverhalten

Wesentlich: deterministisches Zeitverhalten

- zeitgesteuertes System: Zeitplan einhalten (kein „Jitter“)
- ereignisgesteuertes System: Antwortzeit einhalten

Gedankenexperiment zur Antwortzeit:



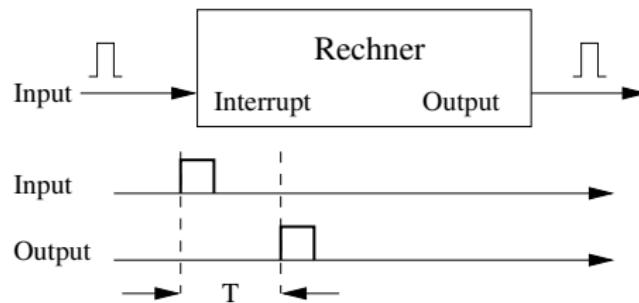
T Konstant?

Zeitverhalten

Wesentlich: deterministisches Zeitverhalten

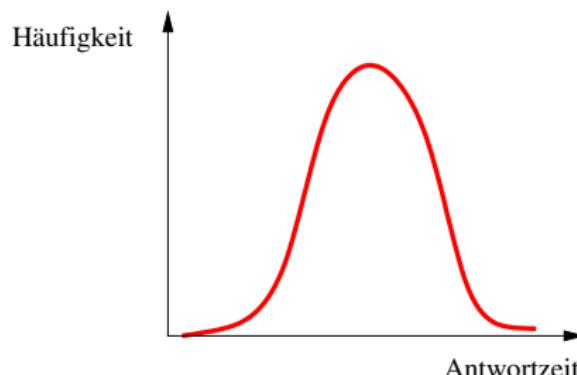
- zeitgesteuertes System: Zeitplan einhalten (kein „Jitter“)
- ereignisgesteuertes System: Antwortzeit einhalten

Gedankenexperiment zur Antwortzeit:



T Konstant? → Nein!

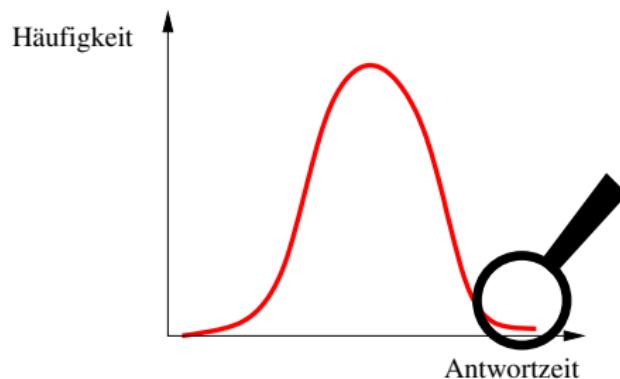
Antwortzeit-Histogramm



Wesentlich: deterministisches Zeitverhalten

- Gibt es eine maximale Antwortzeit, die niemals überschritten wird ?
- Ja → Echtzeitsystem

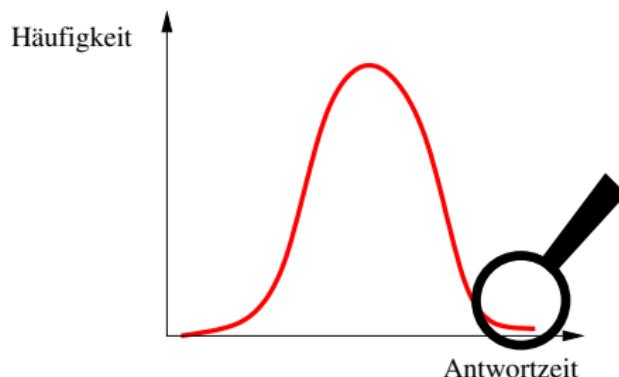
Antwortzeit-Histogramm



Wesentlich: deterministisches Zeitverhalten

- Gibt es eine maximale Antwortzeit, die niemals überschritten wird ?
- Ja → Echtzeitsystem

Antwortzeit-Histogramm

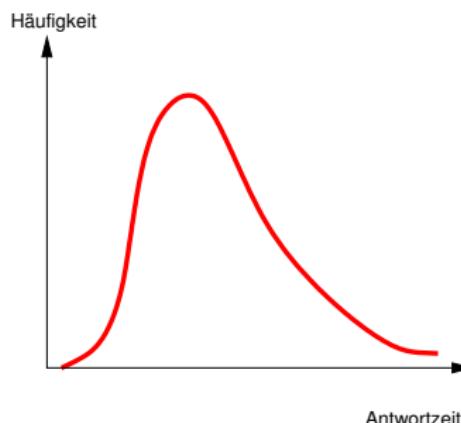


Wesentlich: deterministisches Zeitverhalten

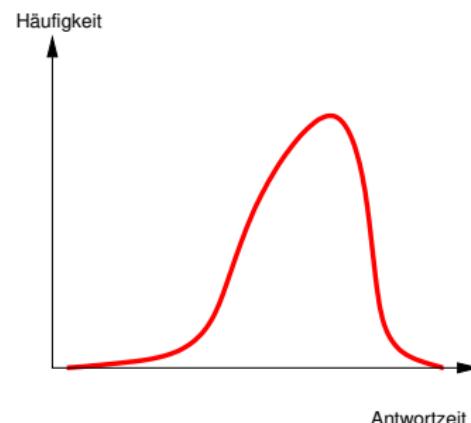
- Gibt es eine maximale Antwortzeit, die niemals überschritten wird ?
- Ja → Echtzeitsystem

Optimierungsziel: Worst Case

Nicht-Echtzeitsystem



Echtzeitsystem

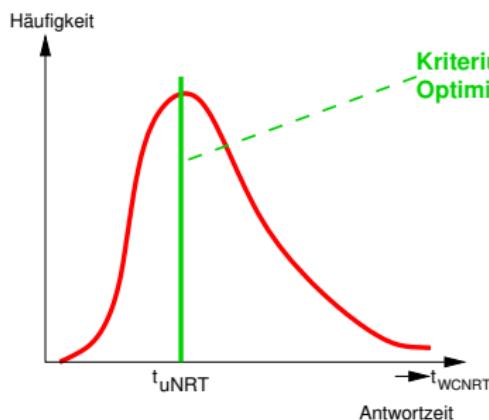


auf niedrige durchschnittliche Antwortzeit optimiert

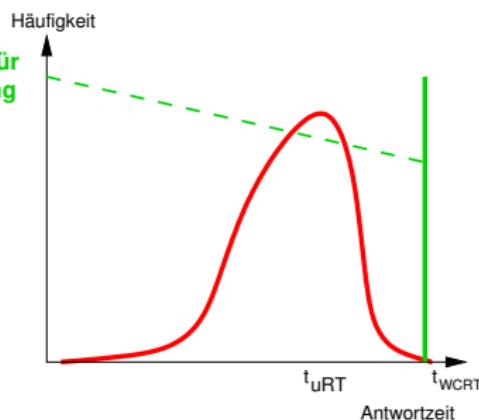
auf niedrige worst case Antwortzeit optimiert: evtl. höhere durchschnittliche Antwortzeit wird in Kauf genommen

Optimierungsziel: Worst Case

Nicht-Echtzeitsystem



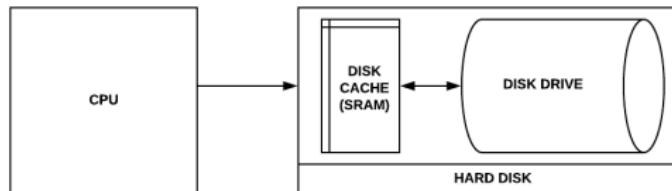
Echtzeitsystem



auf niedrige durchschnittliche Antwortzeit optimiert

auf niedrige worst case Antwortzeit optimiert: evtl. höhere durchschnittliche Antwortzeit wird in Kauf genommen

Optimierung auf worst-case: Beispiel



<https://medium.freecodecamp.org/the-hidden-components-of-web-caching-970854fe2c49>

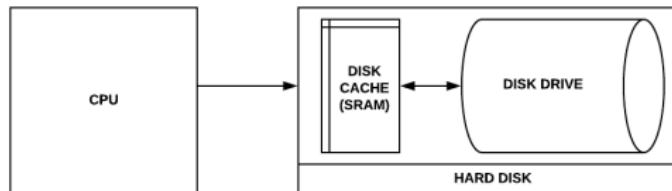
Disk-Cache: RAM-Puffer hält häufig benötigte Daten

- + Verbesserung der mittleren Zugriffszeit
- Aber: Verschlechterung der worst case Zugriffszeit (wg. Cache-Verwaltungsoverhead)

→ Für EZ-System (zumindest) fragwürdig

- Entsprechende Design-Entscheidungen sind auch auf unterster (Betriebssystem-) Ebene erforderlich → „Echtzeitbetriebssystem“
- „Querschneidendes“ Thema (*cross-cutting concern*)

Optimierung auf worst-case: Beispiel



<https://medium.freecodecamp.org/the-hidden-components-of-web-caching-970854fe2c49>

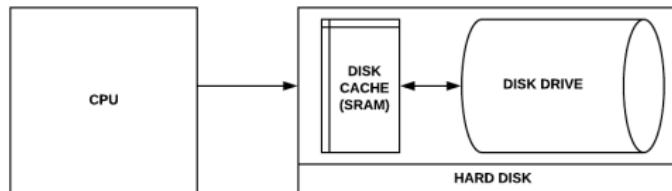
Disk-Cache: RAM-Puffer hält häufig benötigte Daten

- + Verbesserung der mittleren Zugriffszeit
- Aber: Verschlechterung der worst case Zugriffszeit (wg. Cache-Verwaltungsoverhead)

→ Für EZ-System (zumindest) fragwürdig

- Entsprechende Design-Entscheidungen sind auch auf unterster (Betriebssystem-) Ebene erforderlich → „Echtzeitbetriebssystem“
- „Querschneidendes“ Thema (*cross-cutting concern*)

Optimierung auf worst-case: Beispiel



<https://medium.freecodecamp.org/the-hidden-components-of-web-caching-970854fe2c49>

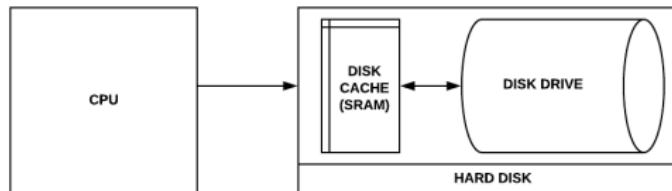
Disk-Cache: RAM-Puffer hält häufig benötigte Daten

- + Verbesserung der mittleren Zugriffszeit
- Aber: Verschlechterung der worst case Zugriffszeit (wg. Cache-Verwaltungsoverhead)

→ Für EZ-System (zumindest) fragwürdig

- Entsprechende Design-Entscheidungen sind auch auf unterster (Betriebssystem-) Ebene erforderlich → „Echtzeitbetriebssystem“
- „Querschneidendes“ Thema (*cross-cutting concern*)

Optimierung auf worst-case: Beispiel



<https://medium.freecodecamp.org/the-hidden-components-of-web-caching-970854fe2c49>

Disk-Cache: RAM-Puffer hält häufig benötigte Daten

- + Verbesserung der mittleren Zugriffszeit
- Aber: Verschlechterung der worst case Zugriffszeit (wg. Cache-Verwaltungsoverhead)

→ Für EZ-System (zumindest) fragwürdig

- Entsprechende Design-Entscheidungen sind auch auf unterster (Betriebssystem-) Ebene erforderlich → „Echtzeitbetriebssystem“
- „Querschneidendes“ Thema (*cross-cutting concern*)

Charakterisierung

Ein Echtzeitsystem hat
unter gegebenen Lastannahmen und
Fehlerannahmen
ein vorhersagbares Systemverhalten !

**Vorhersagbarkeit ist die Eigenschaft,
die ein Echtzeitsystem von anderen Systemen
unterscheidet.**

Klassifizierung bzgl. Rechtzeitigkeit

Harte Echtzeitsysteme (Hard Real-Time Systems):

- Mindestens eine zeitliche Anforderung (Deadline) an das Systemverhalten muss immer und unter allen Last- und Fehlersituationen eingehalten werden (s.o.).
- Es werden „Garantien“ gegeben, die z.B. durch formale Beweise oder analytische Modelle belegt werden.
- Z.B.: garantierte Antwortzeiten zwischen Eingabe vom Sensor und reaktionsbedingter Ausgabe an den Aktoren.
- Jedes Verhalten außerhalb der Garantien wird als Systemversagen eingestuft.
- Unterklasse *Feste Echtzeitsysteme*: Versagen macht zunächst nur die aktuelle Operation wertlos, Versagen bei Wiederholung

Weiche Echtzeitsysteme (Soft Real-Time Systems):

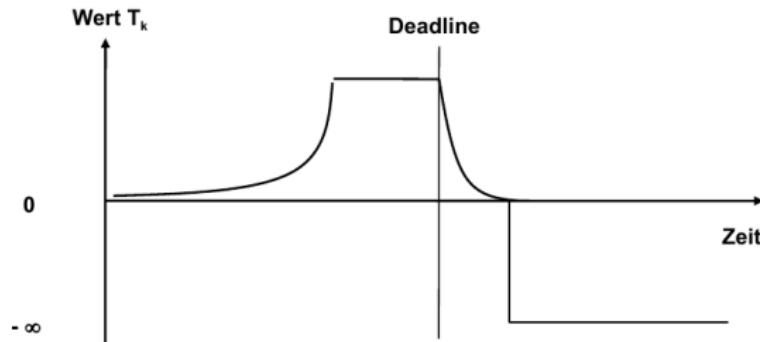
- Die zeitlichen Anforderungen werden in der Regel / statistisch eingehalten, gelegentliche Ausnahmen dürfen aber vorkommen.

Beispiele

Zeitforderungen	Zuverlässigkeit-forderungen	Beispiele
weiches Echtzeit-system	Hohe Verfügbarkeit Hohe Integrität	Telefon-Vermittlung Online Transaction Processing
festes Echtzeit-system	Hohe Verfügbarkeit Fail Safe Fehler-tolerant	Radardaten-Display
hartes Echtzeit-system	Fail Safe Fehler-tolerant	Eisenbahn-Signalsteuerung Fly-by-Wire

Beispiele

- Modellierung von Echtzeitsystemen nach D. Jensen, 1990
- Auch *Value Functions* oder *Utility Functions* genannt
- Der Wert der Berechnung eines Ergebnisses eines Systems wird als Funktion der Zeit definiert.

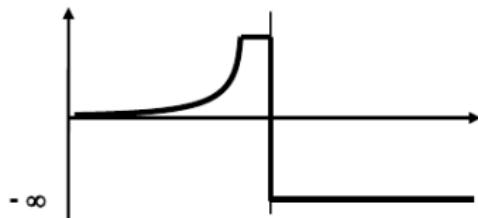


T_k : Wert der k-ten Task

Wert der gesamten Berechnung: $\sum_{k=1}^n T_k$

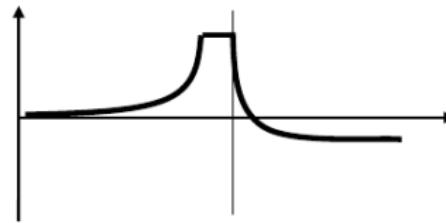
Beispiele

- Hard Real-Time: Verletzung der Zeitspezifikation kann katastrophale Folgen haben:



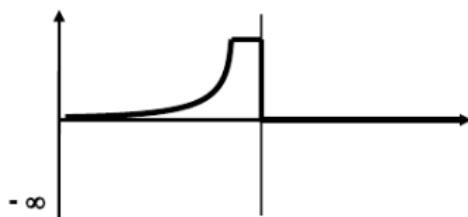
$$\sum_{k=1}^n T_k = -\infty$$

- Soft Real-Time: Zeitweise Verletzung der Zeitspezifikation kann mit gewissen Verlusten hingenommen werden:



Beispiele

- Firm Real-Time: Verletzung der Zeitspezifikation macht Ergebnis wertlos



- Abgrenzung von Hard Real-Time:
 - ▶ Firm Real-Time-Systeme erzeugen keinen unmittelbaren (Personen-)Schaden bei Verletzung der Zeitschranke
 - ▶ Firm Real-Time in Literatur nicht immer betrachtet

Fehleinschätzungen

Common Misconceptions in RT-Computing (Stankovic, 1987):

- RT-Computing ist äquivalent zu „Fast Computing“.
- Fortschritte beim Supercomputing werden die RT-Probleme lösen.
- RT-Programmierung ist Assembler- und Gerätetreiber-Programmierung.
- RT-Systeme sind statische Systeme, angepasst an eine statische Umgebung.
- Die Probleme von RT-Systemen sind in anderen Gebieten der Informatik bereits behandelt und gelöst worden.

„Necessary is a coherent treatment of correctness, timeliness, and fault-tolerance in large-scale distributed systems.“

Zukünftige Entwicklung

Häufige Mängel heutiger Echtzeit-Systeme:

- teuer
- lange Lebensdauer
- ad-hoc-Methoden zur Validierung von Zeitbedingungen
- nicht skalierbar, keine inkrementellen Erweiterungen möglich
- schlechtes dynamisches und adaptives Verhalten

Angestrebte Eigenschaften in den nächsten Generationen:

- komplex
- verteilt
- intelligente Sensoren und Aktoren
- adaptiv
- Verschärfung des Problems der Vorhersagbarkeit

→ **Responsive Systeme: echtzeitfähig, fehlertolerant, verteilt**

Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

2. Zeit und Ordnen



<https://www.unterstufe.ch/hinweise.php?id=25668>

Inhalt

2. Zeit und Ordnen

2.1 Einführung

2.2 Zeitbegriff und Zeitsysteme

2.3 Rechneruhren

2.4 Globale Zeitbasis und Synchronisationsprotokolle

2.5 Logische Zeitmarken

Einführung

- Zeit als physikalische Größe
- Ordnen von Ereignissen aufgrund zeitlicher Ordnung
- Verteilte Systeme mit mehreren Uhren
- Probleme mit nicht-synchronisierten Rechneruhren (Bsp.)
 - ▶ Zeitstempel von Dateien (make)
 - ▶ zeitgesteuertes Ausführen von Aufträgen (cron)
- Globaler Zeitbegriff
 - ▶ Etablierung von systemweiter Zeit in verteilten Systemen durch Synchronisation von Rechneruhren
 - ▶ Synchronität mit realer Außenzeit
- Anwendungsprogrammierung
 - ▶ Nutzung von Zeitdiensten
 - ▶ Zeitmessung

Einführung (2)

Anwendungen

- korrekte Funktion zeitbezogener lokaler und verteilter Anwendungen
- korrektes Ordnen von Ereignissen in verteilten Systemen, z.B. für Prozessvisualisierung
- effizientere verteilte Algorithmen durch Verringerung des Kommunikationsaufwands (vgl. [Liskov])
- Leistungsmessung in verteilten Systemen
- verteilte Echtzeitsysteme benötigen Zeitbegriff einschließlich Synchronität mit realer Außenzeit

Zeitbegriff und Zeitsysteme

Verschiedene Zeitbegriffe im Laufe der Geschichte

Astronomische Zeit

- basiert auf der gleichförmigen Bewegung von Himmelskörpern und deren Beobachtung
- Sonnenzeit
 - ▶ mittlere Dauer einer Erdumdrehung
 - ▶ Sonnentag: Zenit-Zenit (bis 1956)
 - ▶ $1 \text{ Sek} = \frac{1}{24 \cdot 60 \cdot 60} \text{ Sonnentag}$
 - ▶ wenig stabil (Abbremsung der Erdrotation, Schwankungen durch Massenverlagerungen)
- Sternzeit
 - ▶ mittlere Dauer der Umlaufzeit der Erde um die Sonne
 - ▶ $1 \text{ Sek} = \frac{1}{31.556.925,9747} \text{ Teil des trop. Jahres 1900 (ab 1957)}$

Physikalische Zeit

basiert auf (periodischen) physikalischen Prozessen klassische Beispiele:

- Kerzenuhr (Verbrennen von Wachs)
- Pendeluhr, Genauigkeit best: 10^{-7}
- Quarzuhr, Genauigkeit best: 10^{-9} , typisch: $10^{-5} \dots 10^{-6}$

Atomuhr

- Definition im SI-Einheitensystem (ab 1967):
„Die Sekunde ist das 9.192.631.770fache der Periodendauer der dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustandes von Atomen des Nuklids ^{133}Cs entsprechenden Strahlung.“
- Cäsium-133-Uhr, Genauigkeit best: 10^{-14} , typisch: 10^{-13} ($< 1\mu\text{s}$ pro Jahr)
- Caesium-Fontäne, Genauigkeit $< 10^{-15}$

Physikalisch-Technische Bundesanstalt (PTB)



- in Braunschweig
- Betrieb mehrerer Atomuhren (CS1-CS4, CSF1)
- Verantwortung für die gesetzliche Zeit in D (ab 1978)
- Betrieb von Verteildiensten



<https://www.meinberg.de/images/xatomuhr.jpg.pagespeed.ic.3l8wJGqj54.jpg>

Foto: PTB

Zeitsysteme

GMT: Greenwich Mean Time

- Lokale Ortszeitangaben (wahre und mittlere) üblich bis ca. 1880
- Probleme für Eisenbahn-Fahrpläne
- „Greenwich Mean Time“ gesetzliche Standardzeit in England ab 1880
- Ab 1.06.1891: deutsche und österreichisch/ungarische Eisenbahnverwaltungen führen die Zeit des 15. Längengrads als *mitteleuropäische Eisenbahn-Zeit (M. E. Z.)* ein.
- Deutsches Reich: gesetzliche Uhrzeit ab 1.04.1893 ist „die mittlere Sonnenzeit des fünfzehnten Längengrades östlich von Greenwich“
- Meridiankonferenz Washington 1884 definiert Greenwich als Null-Meridian und führt Zeitzonen ein →GMT Sonnenzeit
- ab 1.1.1925 Beginn des Tags um Mitternacht
(für Astronomen bis da hin mittags)

Zeitsysteme (2)

UT: Universal Time

- Weltzeit abgeleitet aus Sternzeit (ab 1957) am Null-Meridian
- UT1: Berücksichtigung der Polschwankungen

TAI: Temps Atomique International

- mittlere Atomzeit seit 1.1.1958
- Betrieb von ca. 250 Atomuhren weltweit
- weltweit koordiniert durch Bureau International de l'Heure (BIH)

UTC: Universal Time Coordinated

- heutiger Zeitstandard (ab 1972)
- basiert auf TAI, aber Anpassungen an UT1 durch „Schaltsekunde“ bei mehr als 900 ms Unterschied
- Abweichung: 1 Sek in 300.000 Jahren

Zeitverteildienste

Langwellen-Radiosender

- z.B. in D: DCF77 (77.5 kHz, Frankfurt/Mainflingen)
- basierend auf Atomuhr CS-2 der PTB
- Sekudentakt
- aufmodulierter voller BCD-Zeitcode (58 Bit) in jeder Minute
- Genauigkeit
 - ▶ $2 \cdot 10^{-13}$ gemittelt über 100 Tage
 - ▶ 1-10 msec je Sek. (atmosphärische Störungen)

GEOS Satellitensystem

- Geostationary Operational Environment Satellite
- Genauigkeit ca. 0.5 msec

Zeitverteildienste (2)

GPS-Satellitensystem als Basis

- Global Positioning System, primär militärisch
- 24 Satelliten, Umlaufzeit 12 h, mind. 4 jederzeit „sichtbar“
- Cäsium-Uhren an Bord
- Synchronisation gegenüber Uhren anderer Satelliten durch Bodenstation auf ± 5 ns genau
- Standortbestimmung durch Unterschiede in Signallaufzeiten ($5\text{ns} \cong 1.5\text{m}$ mil.; $1\mu\text{s} \cong 300\text{m}$ zivil)
- künstliche Ungenauigkeiten in Krisenzeiten
- Differentielles GPS nutzt zusätzlich Bodenstationen mit bekannten Standorten (Geodäsie)

GPS-basierte Uhr

- GPS-Signal als Referenz einer PLL-Schaltung
- hochgenaue Sekundenimpulse (pps pulse-per-second)
- typ. Genauigkeit: ca. $1\mu\text{s}$ (nach ca. $\frac{1}{2}$ h Betrieb, z.B. Meinberg)

Zeitverteildienste (3)

Galileo-Satellitensystem der EU/ESA (bis 201x)

- europäisches, zu GPS kompatibles System (GPS III)
- bis zu 30 Satelliten
 - ▶ mit je 2 Atomuhren
 - ▶ senden Zeitsignal und Positionsdaten
 - ▶ globale Abdeckung
- Dienste
 - ▶ Unterscheidung in globale, regionale, lokale Ebene
 - ▶ kostenloser Dienst für Ortung, Navigation, Zeitsynchronisation (Genauigkeit ca. 4 m horizontal, 8 m vertikal)
 - ▶ kommerzieller Dienst (Genauigkeit 1 m, Bewegungen 0.2 m/sec) (Vermessungswesen, Netzsynchronisation, Flottenmanagement)
 - ▶ Safety-of-Life-Dienst, (sicherheitskritische Anwendungen in Luft- und Schifffahrt, Bahnverkehr)
 - ▶ Dienst „von öffentlichem Interesse“, (Signal mit sehr hoher Genauigkeit, Qualität, Zuverlässigkeit und Integrität für hoheitliche Anwendungen)



Zeitverteildienste (4)

Galileo-Satellitensystem der EU/ESA (Stand)

- Sicht 2004:
Entwicklung bis 2006, Betrieb ab 2008 geplant
- Sicht 2006:
Konzessionsvergabe bis 2008, Errichtung 2009/10,
Betrieb ab 2010
- Sicht 2008:
Konsortium zerbrochen, Betrieb ab 2013 im Auftrag der EU,
Kosten ca. 3,6 Mrd. €
- Sicht 2011:
21.10.11: die ersten beiden Satelliten mit Sojus-Rakete in Orbit gebracht.
- Sicht 2012:
13.10.12: zwei weitere Satelliten, neue Kostenschätzung: 5,0 Mrd. €, Probetrieb ab
2013, Funktionsfähigkeit 2020 (?!?),
Boden testbetrieb Region Berchtesgaden (virtueller Satellitenbetrieb)
- Sicht 2016:
18 der vorgesehenen 30 Satelliten im Orbit. Letzte Satelliten sollen 2018 in ihre
Umlaufbahn geschossen werden. System ist seit 15. Dezember 2016 allgemein zugänglich.



Begriffe (nach Kopetz)

- Zeit modelliert als Zeitstrahl
 - ▶ Unendliche Menge \mathbb{T} von Zeitpunkten mit
 - ▶ \mathbb{T} ist geordnet: $p, q \in \mathbb{T} : p < q, p = q, p > q$
 - ▶ \mathbb{T} ist dicht:
 $p, q \in \mathbb{T}, p \neq q : \exists r \text{ zwischen } p \text{ und } q, \text{ d.h. f\"ur } p < q : p < r < q$
- Zeitdauer als Intervall auf Zeitstrahl
- Ereignis findet zu einem Zeitpunkt statt
(\rightarrow temporale Ordnung auf Ereignissen)
- Ereignisse finden gleichzeitig statt, wenn sie zu gleichem Zeitpunkt stattfinden
- Menge der Ereignisse nur partiell geordnet
 - ▶ wegen gleichzeitiger Ereignisse
 - ▶ Totale Ordnung konstruierbar mit zus\"atzlichem Kriterium (z.B. Knotennummer in verteilten Systemen)

Begriffe (nach Kopetz)

- Kausale Ordnung von Ereignissen

- ▶ Ursache/Wirkungs-Beziehung
- ▶ Aus kausaler Ordnung folgt temporale Ordnung der Ereignisse,
- ▶ Umkehrung gilt nicht
- ▶ Kausale Ordnung damit strenger als temporale Ordnung

- Einheitliche Empfangsordnung von Ereignissen (*Delivery Order*)

- ▶ Kommunikationssystem stellt einheitliche Ordnung aller Nachrichten bei allen Empfängern sicher
- ▶ Empfangsordnung muss nicht mit temporaler Ordnung oder Empfangsordnung übereinstimmen

Begriffe

Referenzzeit

- Approximation der wahren physikalischen Zeit
- Zähler der Referenzuhr zeigt immer korrekten Wert an

Abweichung, Genauigkeit (Accuracy)

- absolute oder relative Differenz zu einer Referenzzeit
- Offset als Zeitdifferenz zwischen zwei Uhren bzw. zur Referenzzeit

Auflösung/Granularität (Granularity)

- kleinste Zeitdauer zwischen zwei aufeinander folgenden anzeigbaren Zeitpunkten ($\frac{1}{f}$, wobei f Frequenz)

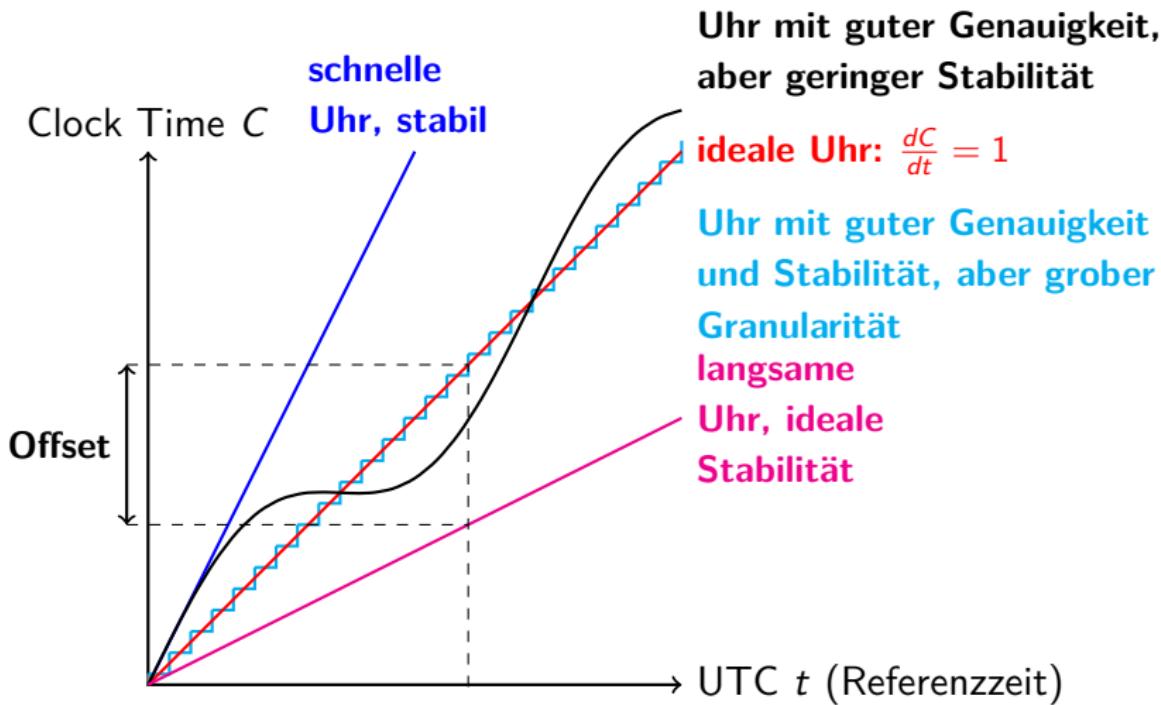
Stabilität (Stability)

- Frequenzschwankung einer Uhr
- Drift als Frequenzdifferenz zwischen zwei Uhren bzw. zur Referenzzeit

Präzision einer Menge von Uhren (Precision)

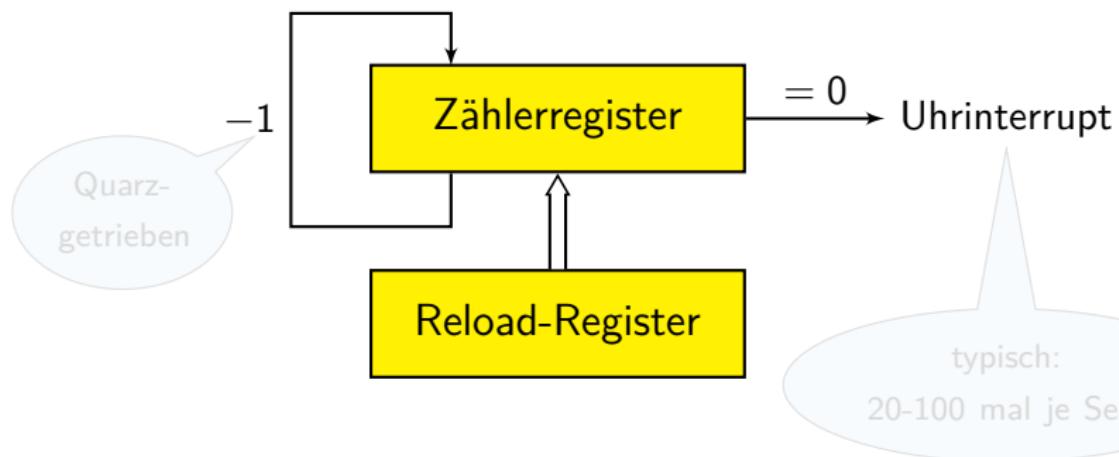
- Max. Offset zwischen irgend zwei Uhren der Menge

Veranschaulichung



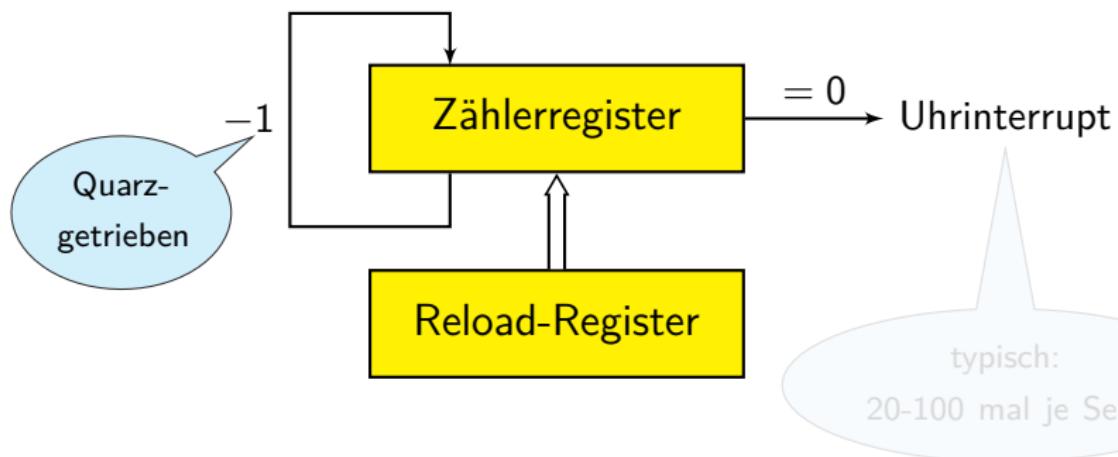
Rechneruhren

Hardware einer lokalen Rechensystemuhr



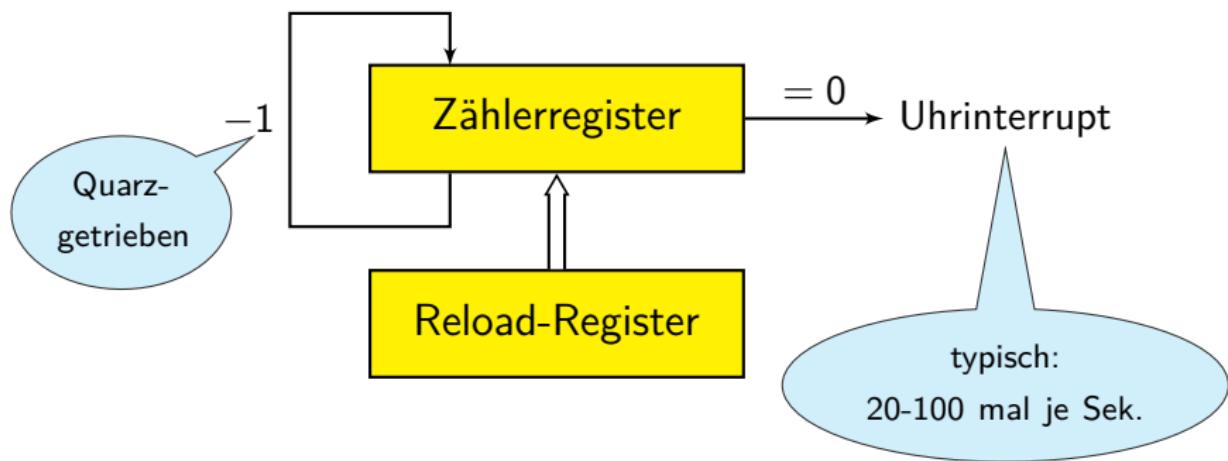
Rechneruhren

Hardware einer lokalen Rechensystemuhr



Rechneruhren

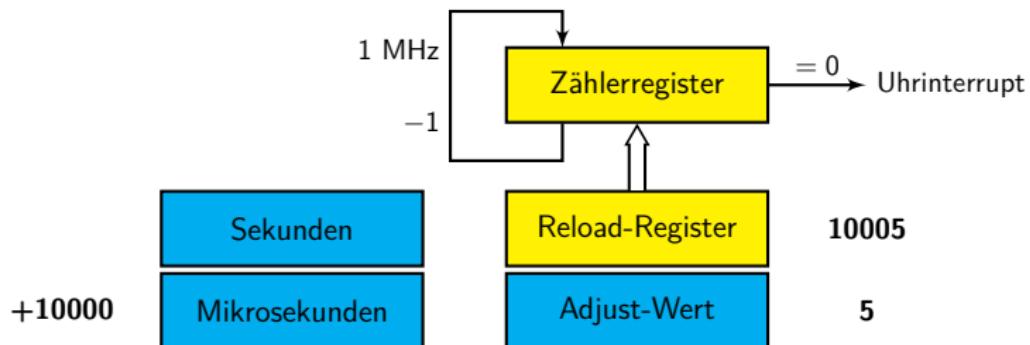
Hardware einer lokalen Rechensystemuhr



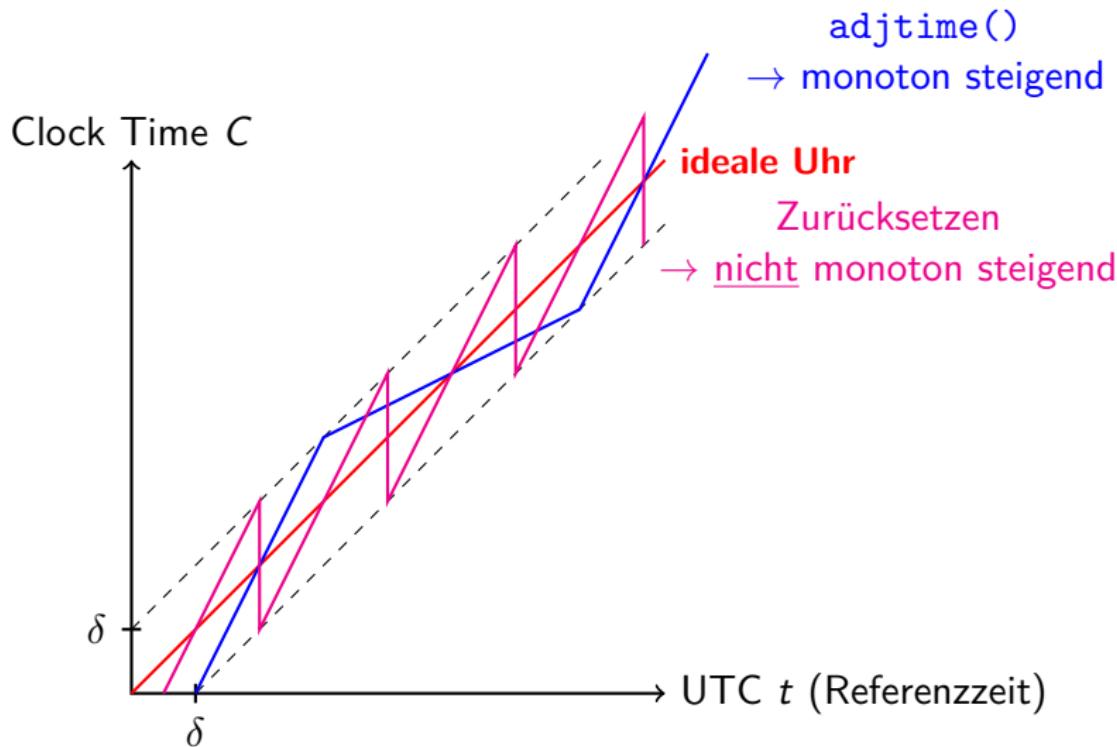
Betriebssystem-Uhren

Beispiel UNIX

- zwei 32-Bit (oder 64-Bit) Integer-Variablen
 - ▶ Anzahl Sekunden seit 1.1.1970
 - ▶ Anzahl μs (oder ns) in der aktuellen Sekunde
- typ. 100 Interrupts/s
- bei Interrupt werden Variablen um nominelle Anzahl μs erhöht
- Korrekturwert (Adjust-Wert) für Ausgleich der Drift des Quarzes
- Systemdienste `settimeofday`, `adjtime`



Prinzip der Korrektur



Referenzzeitquellen

DCF77-Uhr für einfache Anforderungen an Systemzeit

- Genauigkeit typisch: ± 2 msec

GPS-Uhr bei hohen Anforderungen (z.B. Messsystem)

- Genauigkeit typisch: ± 250 nsec

Atom-Uhr

- Rubidium / Caesium-Quellen
- Spezielle Zulassung erforderlich
- Montage in Rack
- z.T. ausschließlich für militärische Zwecke

Rechnerschnittstelle

- Erzeugung von Pulse-Per-Second (pps)-Signalen als Interrupts
- Kodierte Timecode-Signale,
z.B. IRIG-Standard (Inter Range Instrumentation Group)

Genaue lokale Betriebssystem-Uhren

Verwendung einer externen Referenzzeitquelle

Linux-Kern mit „Nano-Kernel-Patch“

- Erhöhung der Auflösung der Systemuhr auf 1 ns (statt μ s)
- Standard in neueren Linux-Kernen
- Nutzung der pps-Signale der Referenzzeitquelle als Interrupts
- Korrektur der Systemuhr entsprechend Referenzzeit der Hardware-Uhr
- Varianz der Interrupt-Latzenzeiten beeinflusst Genauigkeit
- mehrere externe Zeitquellen an einem Rechner möglich zur weiteren Erhöhung der Genauigkeit
- Genauigkeit: typisch $< 1 \mu$ s

Beispiel: David L. Mill's Uhren (Uni Delaware)



<http://doc.ntp.org/4.1.2/refclock.htm>

- Spectracom 8170 WWVB Receiver
- Spectracom 8183 GPS Receiver
- Spectracom 8170 WWVB Receiver
- Spectracom 8183 GPS Receiver
- Hewlett Packard 105A Quartz Frequency Standard
- Hewlett Packard 5061A Cesium Beam Frequency Standard
- NTP primary time server *rackety* and *pogo* (elsewhere)

Kommerzielle Time Server

Time Server

- Dedizierter LAN-Netzwerkknoten zur Zeitsynchronisation
- Interne oder externe Referenzzeitquelle
- Unterstützung für Standard-Protokolle (s.u.) (NTP, SNTP, PTP/IEEE 1588)

Produkte in vielen Varianten

- Meinberg (D)
- IPCAS (D)
- Galleon (UK)
- ELPROMA (NL)
- Time Tools (UK)

Globale Zeitbasis und Synchronisationsprotokolle

Globale Zeitbasis und Synchronisationsprotokolle

- Abstraktion
- Approximiert durch lokale Zeiten eines Ensembles synchronisierter lokaler Uhren

Plausibilitätsbedingung für globale Granularität g

- Die globale Zeit heißt plausibel für die Granularität g , wenn
 - ▶ die sie lokal implementierenden Uhren des Ensembles eine beschränkte Präzision Π besitzen (max. Unterschied je zweier Uhren)
 - ▶ $g > \Pi$
- Synchronisationsfehler ist kleiner als ein Tick der gedachten globalen Uhr
- für jedes Ereignis e unterscheiden sich die globalen Zeitmarken beliebiger lokaler Uhren j und k um maximal eine Einheit (Tick):
 $|t^j(e) - t^k(e)| \leq 1$
- Dies ist das optimal erreichbare Ergebnis

Folgerungen

- Wenn sich die globalen Zeitmarken zweier Ereignisse um max. einen Tick (Granularität g) unterscheiden, kann die korrekte temporale Ordnung nicht hergestellt werden.
- Ab Unterschied von zwei globalen Ticks ist korrekte temporale Ordnung möglich und durch die Zeitmarken gegeben
- Für die wahre Dauer d_{true} eines Zeitintervalls bei beobachteter Dauer d_{obs} gilt

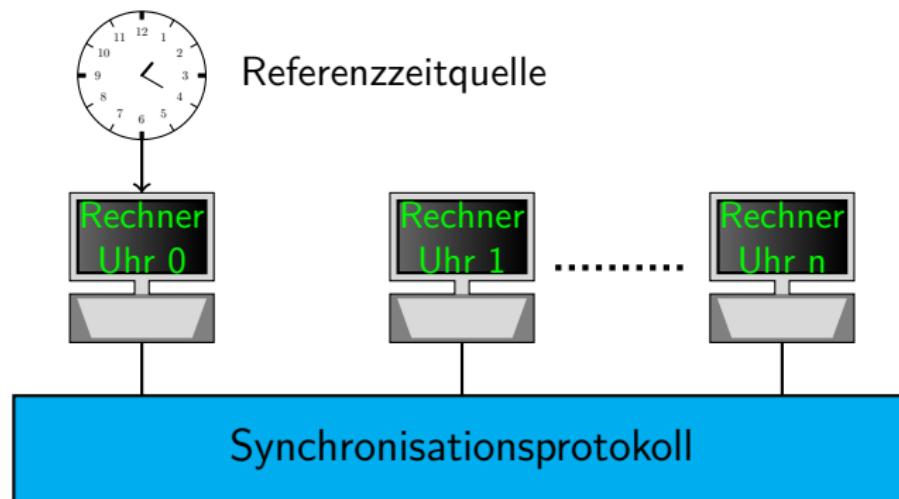
$$(d_{obs} - 2g) < d_{true} < (d_{obs} + 2g)$$

- Für eine konsistente globale temporale Ordnung zweier globaler Ereignisse durch zwei lokale Knoten ist ein Abstand der Ereignisse von $3g$ notwendig

Synchronisationsprotokolle

Konstruktion einer verteilten Zeitbasis für Rechensysteme

- UTC-basierte externe Referenzzeitquelle
- lokale Uhren in den Rechensystemen
- Synchronisationsprotokoll



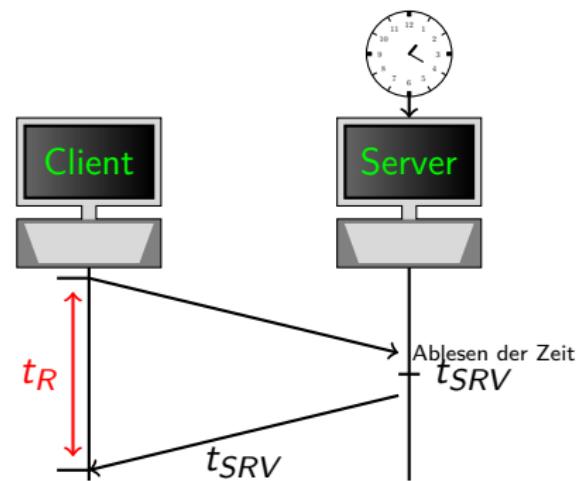
Probleme

- Nachrichtenverzögerung im Netzwerk nicht deterministisch
 - Bearbeitung der Protokollnachrichten zeitlich nicht deterministisch
- ⇒ **keine exakte Synchronisation möglich**

Algorithmus von Christian (1989)

- passiver Zeitserver (als Referenzzeitquelle)
- periodisches Abfragen der Zeit durch Klienten
- mittlere Roundtriptlaufzeit (incl. Verarbeitungszeit auf dem Server) messen und berücksichtigen
- Schwächen:
 - ▶ „Rückwärtsgehen“ einer Uhr ist möglich
 - ▶ Schwankungen in Nachrichtenlaufzeiten

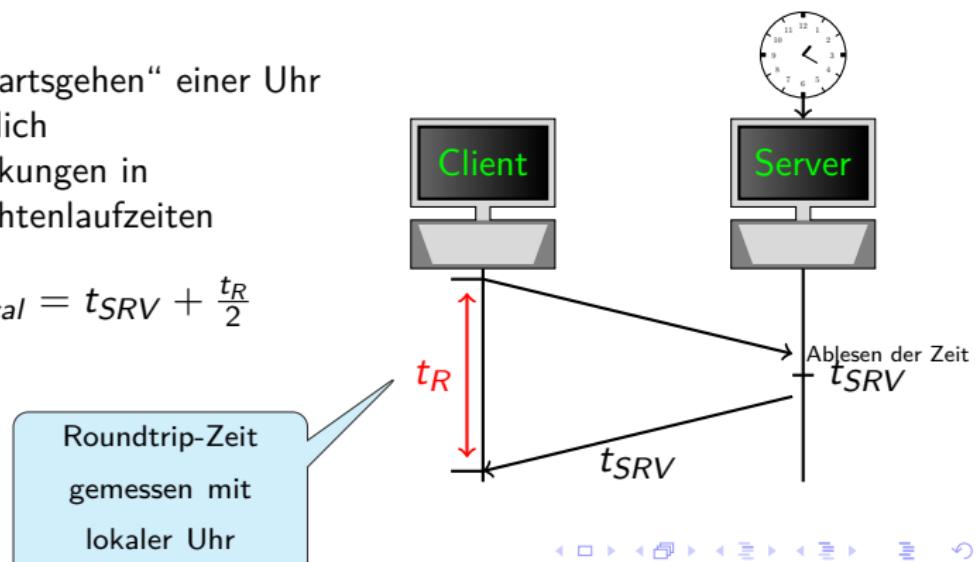
Setze: $t_{local} = t_{SRV} + \frac{t_R}{2}$



Algorithmus von Christian (1989)

- passiver Zeitserver (als Referenzzeitquelle)
- periodisches Abfragen der Zeit durch Klienten
- mittlere Roundtriptlaufzeit (incl. Verarbeitungszeit auf dem Server) messen und berücksichtigen
- Schwächen:
 - ▶ „Rückwärtsgehen“ einer Uhr ist möglich
 - ▶ Schwankungen in Nachrichtenlaufzeiten

Setze: $t_{local} = t_{SRV} + \frac{t_R}{2}$



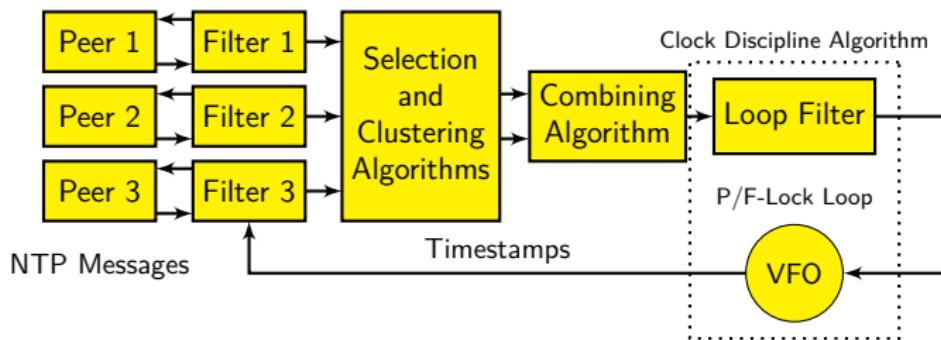
Time Synchronisation Protocol (TSP)

- Berkeley UNIX `timed`
- basiert auf ICMP/IP
- etabliert „mittlere Netzwerkzeit“ in allen Stellen
- Master/Slave-Algorithmus
 - ▶ aktiver Master: fragt aktuelle Zeiten aller Knoten ab, berechnet Mittel
 - ▶ verteilt Differenz (Offset) an jeden Client
- nutzt `settimeofday()` und `adjtime()` in den Knoten
- deutliche Schwächen
 - ▶ „Rückwärtsgehen“ einer Uhr ist möglich
 - ▶ keine Kompensation von Schwankungen in Nachrichtenlaufzeiten
 - ▶ keine Fehlerabschätzung
 - ▶ schlechte Skalierbarkeit
- Variante: Master mit ext. Referenzzeitquelle verteilt aktuelle Zeit statt berechnetem Mittelwert

Network Time Protocol (NTP)

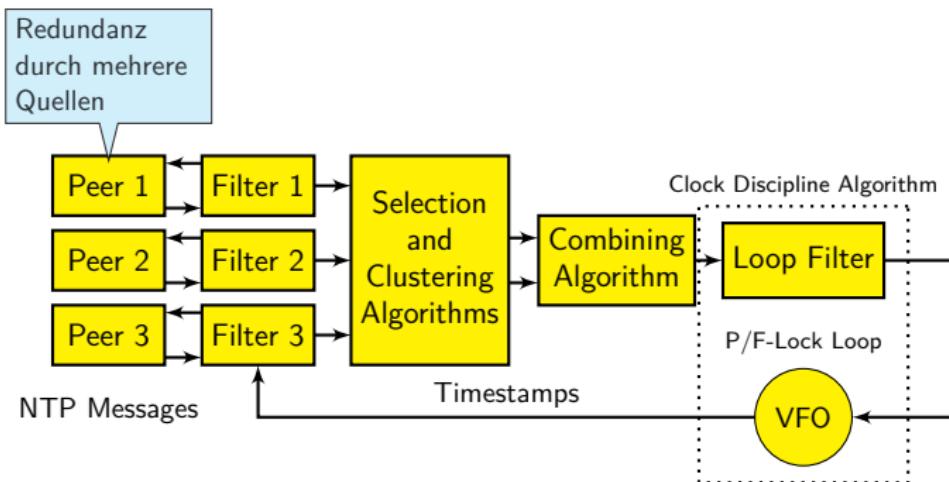
- Entwicklung primär durch D. Mills (Univ. of Delaware) getrieben
- <http://www.ntp.org>
- Ziele:
 - ▶ hohe Genauigkeit
 - ▶ Berücksichtigung schwankender Nachrichtenlaufzeiten
 - ▶ Berücksichtigung von Rechnerausfällen durch Bezug zu mehreren Zeitservern (Peers)
 - ▶ Aussortieren offensichtlich unbrauchbarer Zeitquellen (false ticker)
 - ▶ eingeschränkte Authentifizierung, Verschlüsselung
 - ▶ hohe Skalierbarkeit
- Heute Internet Standard
 - ▶ RFC 1305, 1992, frühere Version RFC 1129, RFC 958 (1985)
 - ▶ > 1.000.000 Rechner, Router, usw.
 - ▶ Nutzt UDP, Port 123
 - ▶ UNIX ntpd, xntpd (Clients aber auch für fast alle anderen Systeme)
 - ▶ Zeitserver der PTB: ptbtime1.ptb.de, ptbtime2.ptb.de
- Genauigkeit:
 - ▶ im LAN <1 ms, Internet < ca. 10 ms

NTP(2)-Arbeitsweise



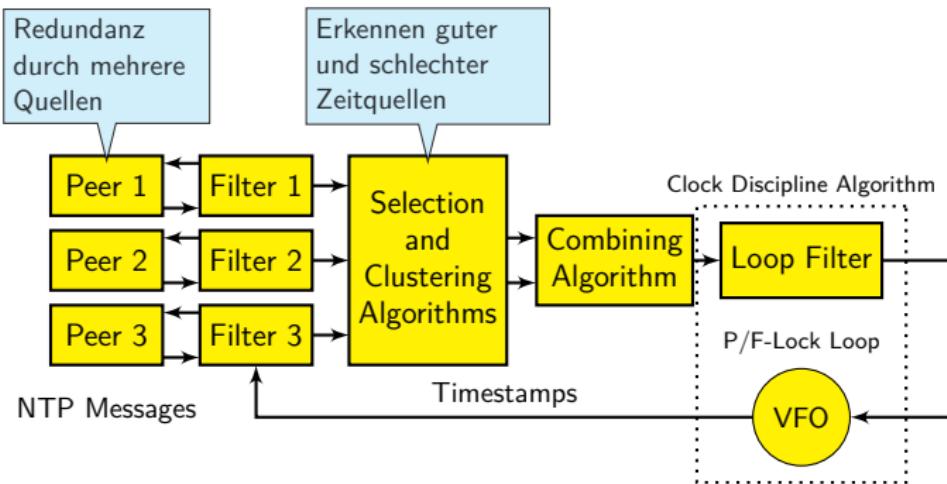
(Abbildung von Mills)

NTP(2)-Arbeitsweise



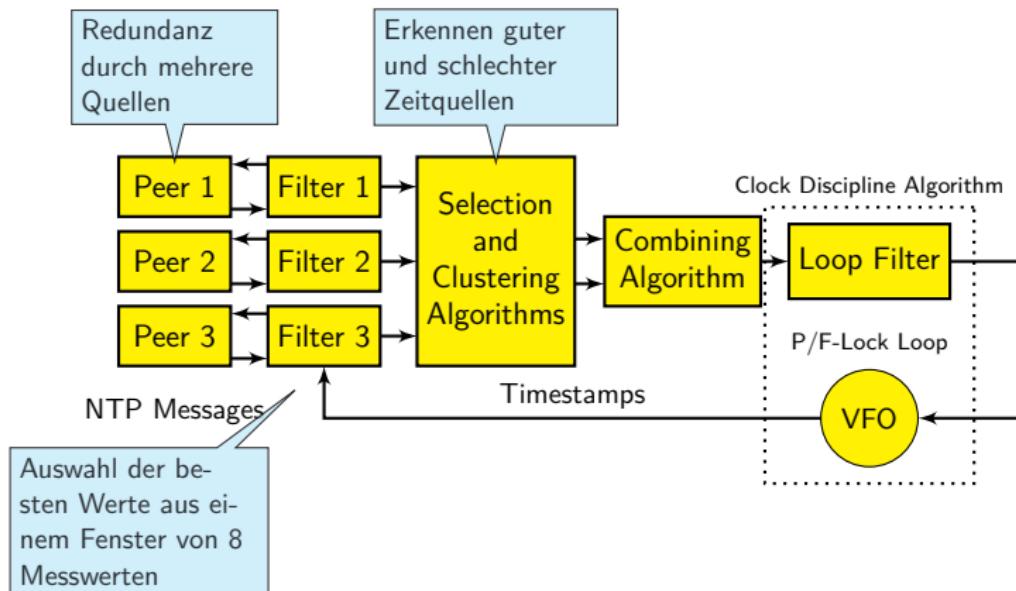
(Abbildung von Mills)

NTP(2)-Arbeitsweise



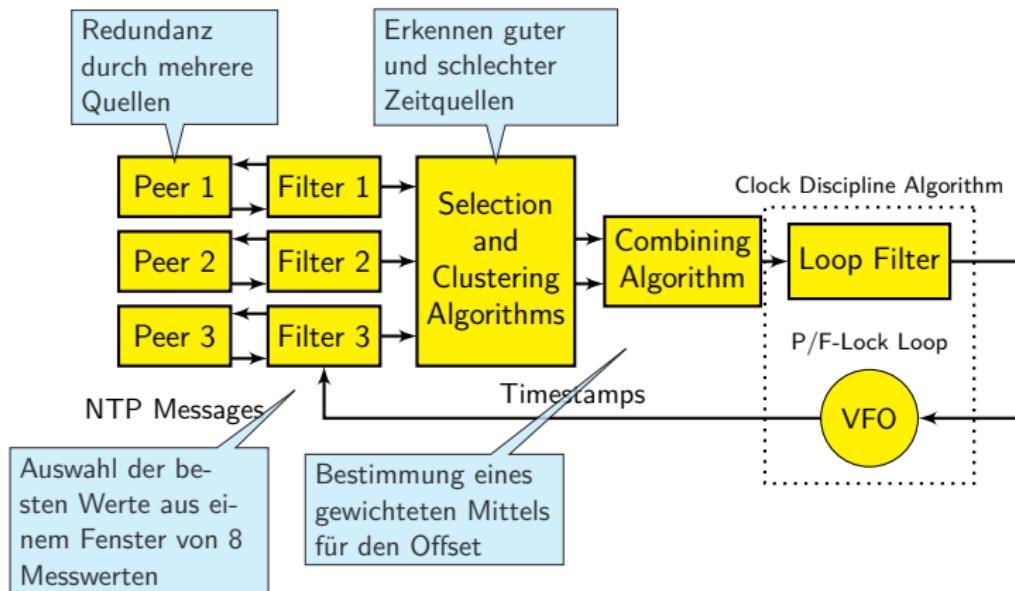
(Abbildung von Mills)

NTP(2)-Arbeitsweise



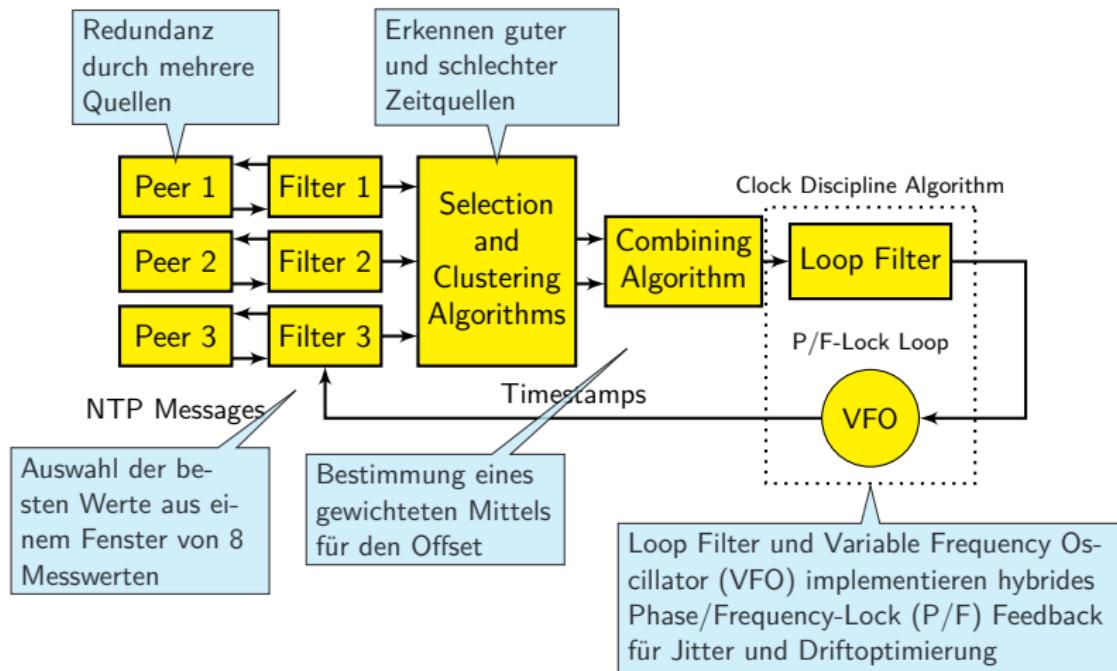
(Abbildung von Mills)

NTP(2)-Arbeitsweise



(Abbildung von Mills)

NTP(2)-Arbeitsweise



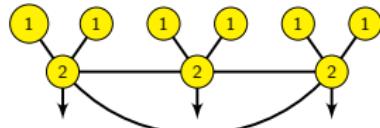
(Abbildung von Mills)

NTP(3)

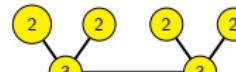
Server legen Zeit fest, Clients beziehen Zeit

Hierarchiebildung der Server durch „Stratum“-Level

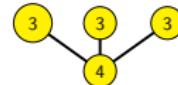
- Knoten mit externen Referenzzeitquellen bilden Stratum 1 -Server
(Genauigkeit: < 1 μ s möglich)
- Stratum n - Server synchronisieren sich mit Stratum n-1 - Servern, usw.
- im Internet (2015)
 - ▶ Jeweils ca. 300 aktive Stratum-1 und Stratum-2 Server
<http://support.ntp.org/bin/view/Servers/StratumOneTimeServers>
 - ▶ Praktisch: 4-stufige Hierarchie, Lastausgleich durch regionale NTP Pool Server
- Typische Strukturen:



Unternehmens-Zeitserver
(fehlertolerant)



Abteilungs-Zeitserver
(fehlertolerant)



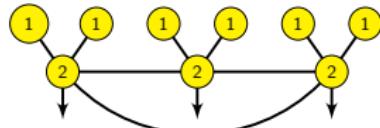
Workstation

NTP(3)

Server legen Zeit fest, Clients beziehen Zeit

Hierarchiebildung der Server durch „Stratum“-Level

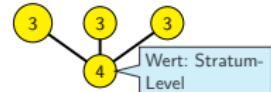
- Knoten mit externen Referenzzeitquellen bilden Stratum 1 -Server
(Genauigkeit: < 1 μ s möglich)
- Stratum n - Server synchronisieren sich mit Stratum n-1 - Servern, usw.
- im Internet (2015)
 - ▶ Jeweils ca. 300 aktive Stratum-1 und Stratum-2 Server
<http://support.ntp.org/bin/view/Servers/StratumOneTimeServers>
 - ▶ Praktisch: 4-stufige Hierarchie, Lastausgleich durch regionale NTP Pool Server
- Typische Strukturen:



Unternehmens-Zeitserver
(fehlertolerant)



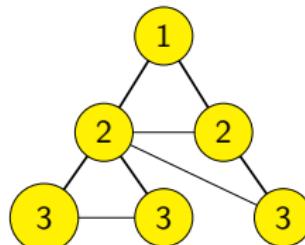
Abteilungs-Zeitserver
(fehlertolerant)



Workstation

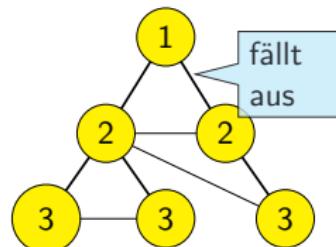
NTP(3)

- dynamisch festgelegte logische Verbindungsstruktur mit Backup-Verbindungen
 - ▶ spannende Bäume minimalen Gewichts basierend auf Server Level und Gesamtsynchronisationsverzögerung jedes Servers zu Primary Servern
- Beispiel Verbindungstopologie
- Nachrichtenaustausch zwischen Servern zwischen 64 sec und 1024 sec (17 min) je nach Qualität der Verbindung
- 64 Bit Zeitmarken
 - ▶ 32 Bit für Sekunden seit 1.1.1900 00:00:00
 - ▶ 32 Bit für Sekundenbruchteil
- Nutzung von `settimeofday()` und `adjtime()` zur Durchsetzung großer bzw. kleiner (<0.128 sec) Korrekturen.
- Kein Zurücksetzen der Uhr



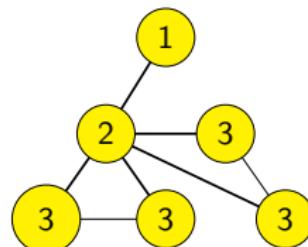
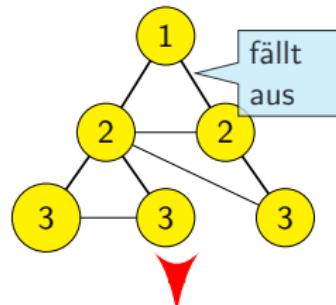
NTP(3)

- dynamisch festgelegte logische Verbindungsstruktur mit Backup-Verbindungen
 - ▶ spannende Bäume minimalen Gewichts basierend auf Server Level und Gesamtsynchronisationsverzögerung jedes Servers zu Primary Servern
- Beispiel Verbindungstopologie
- Nachrichtenaustausch zwischen Servern zwischen 64 sec und 1024 sec (17 min) je nach Qualität der Verbindung
- 64 Bit Zeitmarken
 - ▶ 32 Bit für Sekunden seit 1.1.1900 00:00:00
 - ▶ 32 Bit für Sekundenbruchteil
- Nutzung von `settimeofday()` und `adjtime()` zur Durchsetzung großer bzw. kleiner (<0.128 sec) Korrekturen.
- Kein Zurücksetzen der Uhr



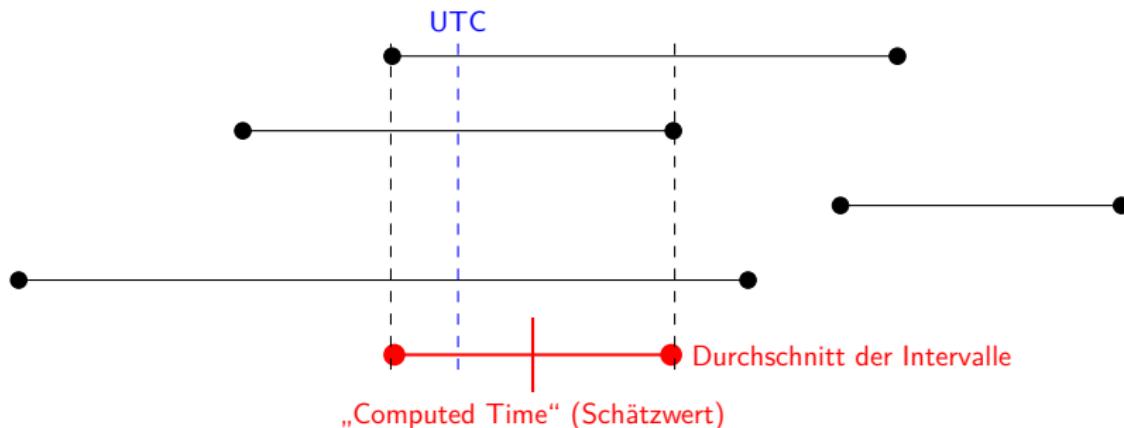
NTP(3)

- dynamisch festgelegte logische Verbindungsstruktur mit Backup-Verbindungen
 - ▶ spannende Bäume minimalen Gewichts basierend auf Server Level und Gesamtsynchronisationsverzögerung jedes Servers zu Primary Servern
- Beispiel Verbindungstopologie
- Nachrichtenaustausch zwischen Servern zwischen 64 sec und 1024 sec (17 min) je nach Qualität der Verbindung
- 64 Bit Zeitmarken
 - ▶ 32 Bit für Sekunden seit 1.1.1900 00:00:00
 - ▶ 32 Bit für Sekundenbruchteil
- Nutzung von `settimeofday()` und `adjtime()` zur Durchsetzung großer bzw. kleiner (<0.128 sec) Korrekturen.
- Kein Zurücksetzen der Uhr



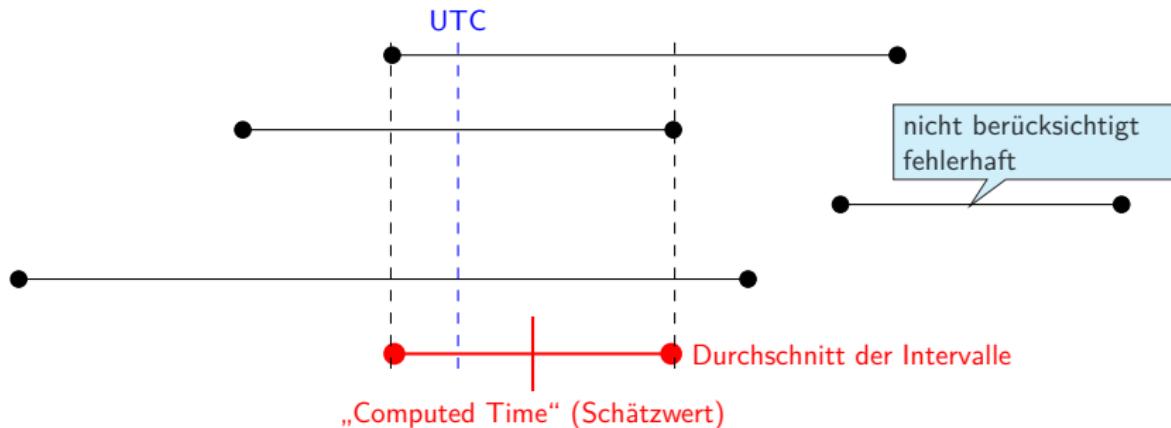
Distributed Time Service (DTS)

- Teil von OSF DCE, ursprünglich von Digital entwickelt
- Besonderheit: Etablieren eines Zeitintervalls, das UTC enthält und Ungenauigkeit minimiert
- adjust im Verhältnis 1:100



Distributed Time Service (DTS)

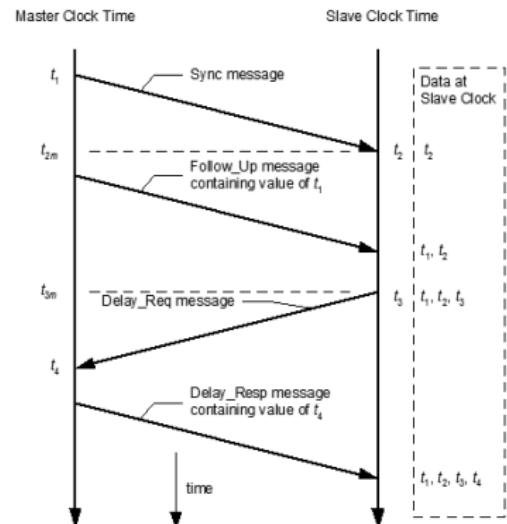
- Teil von OSF DCE, ursprünglich von Digital entwickelt
- Besonderheit: Etablieren eines Zeitintervalls, das UTC enthält und Ungenauigkeit minimiert
- adjust im Verhältnis 1:100



Precision Time Protocol (PTP, IEEE 1588)



- Hauptsächlich für mess- und Regelungstechnische Anwendungen
 - Erreicht höhere Genauigkeit als NTP für Netze mit räumlich begrenzter Ausdehnung
 - Master-Slave-Verfahren
 - Automatische Wahl der besten Uhr als Grandmaster-Clock
 - Primär auf Ethernet-Netzen angewendet
 - Timestamping-Unit kann als Teil des Netzwerk-Controllers (in Hardware) implementiert sein
- ⇒ Genauigkeit im ns-Bereich, in Software im μs -Bereich
- Ptpd als freie Implementierung
 - Verbesserte Version IEEE 1588-2008



http://www.real-time-systems.com/ieee_1588/index.php

$$t_2 - t_1 = \text{offset} + d$$

$$t_4 - t_3 = -\text{offset} + d$$

$$\text{offset} = \frac{(t_2 - t_1 - t_4 + t_3)}{2}$$

Logische Zeitmarken

Realzeit ist nicht immer notwendig

Beispiele:

- Ordnen von Ereignissen (vor - nach)
- zeitmarkenbasiertes Concurrency Control in Datenbanken

Lamport Zeitstempel

Relation happens-before

- Notation: $a \rightarrow b$ (a passiert-vor b)
- Ereignisse im selben Prozess sind linear geordnet
- Nachrichtenversand:
 - ▶ a sei Ereignis des Versendens einer Nachricht m
 - ▶ b sei Empfang der Nachricht m in einem anderen Prozess
 - ▶ dann gilt: $a \rightarrow b$
- Relation ist transitiv:
 - ▶ $a \rightarrow b, b \rightarrow c \Rightarrow a \rightarrow c$
- Nebenläufigkeit:
 - ▶ falls weder $a \rightarrow b$ noch $b \rightarrow a$ gilt, heißen a und b nebenläufig

Uhrenbedingung

- $C(a)$ bezeichne die (logische) Zeit, zu der das Ereignis a stattfinde.
- $a \rightarrow b \Rightarrow C(a) < C(b)$

Lamport-Uhren

Algorithmus für logische Uhren nach Lamport (1978)

Annahmen:

- Prozesse kommunizieren über Nachrichten (und nur über Nachrichten) miteinander
- jeder Prozess P hat eine logische Uhr C_P
- jedes Ereignis e des Prozesses P erhält logischen Zeitstempel $C_P(e)$
- zwei aufeinander folgende Ereignisse e_i und e_{i+1} eines Prozesses haben nie den gleichen Zeitstempel: $C_P(e_i) < C_P(e_i + 1)$

Lamport-Uhren (2)

Beispiel:

A	B	C
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

m1

m2

m3

m4

Uhren mit unterschiedlichen
Geschwindigkeiten ohne Korrektur

A	B	C
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

m1

m2

m3

m4

Uhren mit unterschiedlichen
Geschwindigkeiten **mit** Korrektur

Lamport-Uhren (3)

Algorithmus:

- Berücksichtigung von Kausalität im Nachrichtenversand!
- Sendeereignis s einer Nachricht m in Prozess A :
 - ▶ Zeitmarke $C_A(s)$
 - ▶ Versende Nachricht m zusammen mit aktuellem Zeitstempel des sendenden Prozesses $t = C_A(s)$
- Empfangsereignis e der Nachricht m in Prozess B :
 - ▶ sei $C_B(alt)$ die Zeitmarke des letzten Ereignisses in B
 - ▶ Setze $C_B(e) := \max\{C_B(alt), t\} + 1$
- Falls zwei Ereignisse in verschiedenen Prozessen die gleiche Zeitmarke haben sollten, ordne sie anhand der Prozessordnung
- Algorithmus erfüllt Uhrenbedingung
- Umkehrung gilt **nicht**:
 $C(a) < C(b) \Rightarrow a \rightarrow b$ ist falsch !
- Lamport-Uhren lösen nicht das Kausalitätsproblem

Vector Clocks

Vektor-Uhren, Mattern (Uni Kaiserslautern, 1989)

Vektor-Uhren lösen o.a. Kausalitätsproblem

Algorithmus:

- nachrichtenbasierte Kommunikation
- jeder Prozess P_i besitzt Uhr VC_i als Vektor von Zeitmarken
- lokales Ereignis in P_i :
 - ▶ $VC_i[i] := VC_i[i] + 1$, sonst unverändert
- Sendeereignis in P_i :
 - ▶ $VC_i[i] := VC_i[i] + 1$ (Erhöhe eigenen Ereigniszähler)
 - ▶ Versende Nachricht mit eigener Vektorzeit $vt = VC_i$
- Empfangsereignis in P_k :
 - ▶ $VC_k[j] := \max\{VC_k[j], vt[j]\}$ für alle j
 - ▶ $VC_k[k] := VC_k[k] + 1$ (Erhöhe eigenen Ereigniszähler)

Vector Clocks (2)

Vergleich von Zeitmarkenvektoren

- $S \leq T \Rightarrow S[i] \leq T[i]$ für alle i
- $S < T \Rightarrow S \leq T$ und $S \neq T$
- $S || T \Rightarrow \neg(S < T) \text{ und } \neg(T < S)$

Nebenläufigkeit

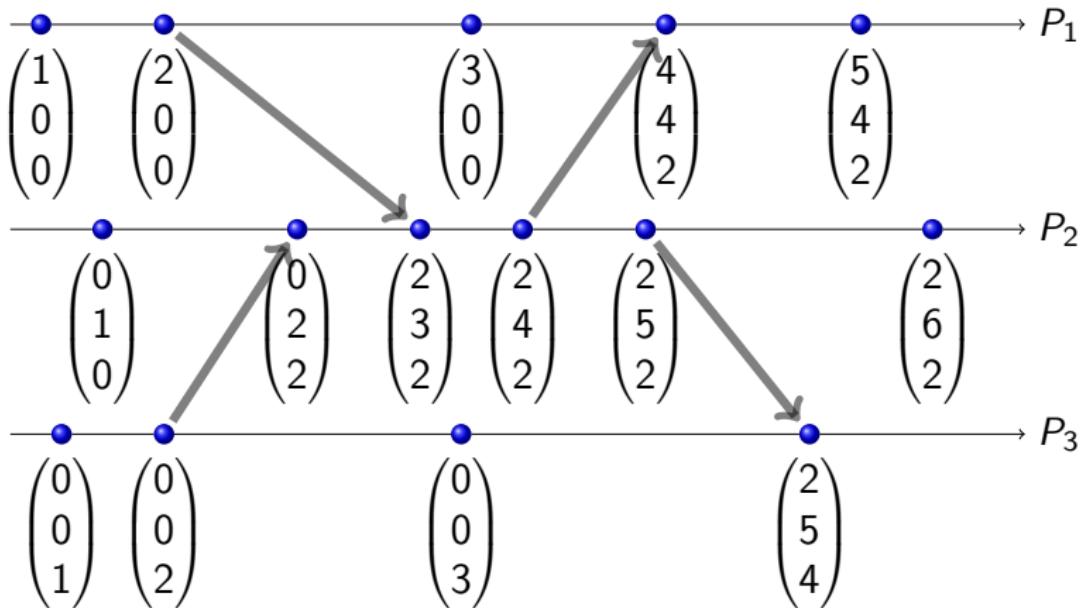
- Ereignisse a und b sind nebenläufig $\Leftrightarrow VC(a) || VC(b)$

Kausalität

- $a \rightarrow b \Leftrightarrow VC(a) < VC(b)$

Vector Clocks (3)

Beispiel



- kausal abhängige Ereignisse, z.B. $(0, 0, 1) \rightarrow (5, 4, 2)$, $(1, 0, 0) \rightarrow (2, 6, 2)$
- nebenläufige Ereignisse, z.B. $(0, 0, 3) \parallel (5, 4, 2)$

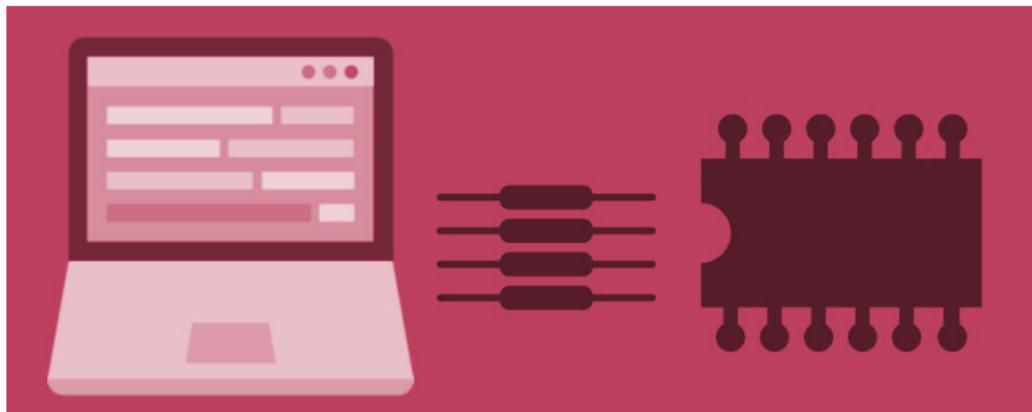
Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

3. C-Programmierung eingebetteter Systeme



<https://freevideolectures.com/course/3624/embedded-systems-programming>

Inhalt

3. C-Programmierung eingebetteter Systeme

3.1 Cross-Entwicklung

3.2 Hardwarenahes Programmieren in C

3.1 Cross-Entwicklung



https://blog.ergodirekt.de/index.php?aam_media=13965&size=full

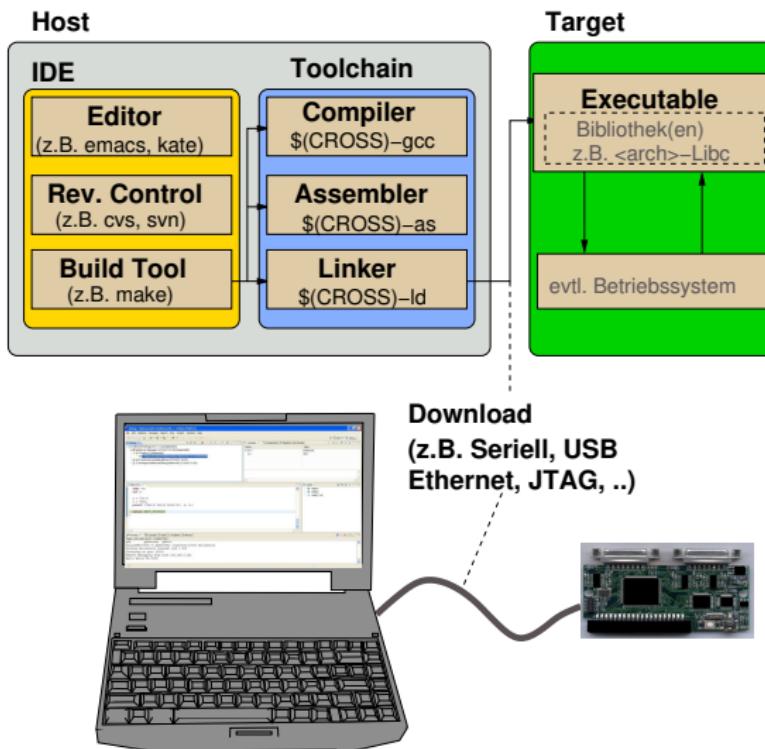
Cross-Entwicklung

- Bisher bekannt: *self-hosted*-Entwicklung: Programme werden auf dem Rechner entwickelt, auf dem sie ausgeführt werden
- Eingebettete Systeme verfügen i.d.R. nicht über geeignete / genügende Betriebsmittel für eine Entwicklungsumgebung:
 - ▶ Speicher
 - ▶ Dateisystem
 - ▶ Ein-/Ausgabe

→ *Cross-Entwicklung*:

- Entwicklungswerzeuge (IDE, Compiler, Linker) auf einem Entwicklungsrechner (*Host*)
- Generieren Code für ein Zielsystem (*Target*)
- Ausführbarer Code (*executable*) wird über geeignete Schnittstellen ins Zielsystem geladen und dort ausgeführt
- Host and Target können (müssen aber nicht) unterschiedliche Prozessorarchitekturen besitzen

Cross-Entwicklung



Cross-Entwicklung: Entwicklungsumgebung

- Entwicklungsumgebung: Sammlung von Programmen zum Erzeugen und Verwalten von Quellcode
 - ▶ Editor, Build-Tool, Revisionskontrolle, Handbuch, etc.
 - ▶ **Nicht** Target-spezifisch
 - ▶ Z.T. mit grafischer Benutzerschnittstelle
- Beispiele:
 - ▶ Editoren: emacs, vi, vim, kate, notepad, ...
 - ▶ Build-Tools: make, nmake, qmake, ...
 - ▶ Revisionskontrolle: Git, Mercurial, Subversion, CVS, RCS, SCCS,
- IDE: Integrierte Entwicklungsumgebung: (grafische) Benutzeroberfläche für die Programme der Entwicklungsumgebung
- Beispiele: Eclipse, Source Navigator, kscope, KDevelop, ...

Cross-Entwicklung: Toolchain

- **Cross-Toolchain:** Programme zum Erzeugen, Bearbeiten und Analysieren von ausführbarem Code:
 - ▶ Target-spezifischer Teil der Entwicklungsumgebung
 - ▶ I.d.R. Kommandozeilen-Programme
 - ▶ Aufruf aus bzw. Anpassung an IDE (s.o.) ggf. über „Plugins“
- Beispiel: GNU ARM Toolchain:

arm-linux-gnueabi-cpp	C-Präprozessor
arm-linux-gnueabi-gcc	C-Compiler
arm-linux-gnueabi-g++	C++-Compiler
arm-linux-gnueabi-as	Assembler
arm-linux-gnueabi-ld	Linker
arm-linux-gnueabi-ar	Archiver/Librarian
arm-linux-gnueabi-nm	Symbole anzeigen
arm-linux-gnueabi-objdump	Inhalte der Sektionen anzeigen
arm-linux-gnueabi-objcopy	Binärformate konvertieren
arm-linux-gnueabi-size	Größen von Sektionen anzeigen
arm-linux-gnueabi-strip	Symbolinformationen entfernen
arm-linux-gnueabi-gcov	Coverage-Analyse
...	...

Cross-Entwicklung: Plattformabhängigkeiten

- Bei GNU allgemein üblich: <Plattform>-<Programmname>
(z.B.: avr-gcc, ppc_60x-gcc, arm-elf-gcc, ...)

Tipp: Toolchain-Anpassung in Makefile z.B. so:

```
CROSS =  
CC = $(CROSS) gcc
```

- Bei Aufruf mit „make CROSS=arm-linux-gnueabi“ wird
arm-linux-gnueabi-gcc verwendet (sonst: gcc)

Maschinenspezifischer Code in C z.B. so:

```
#ifdef __ARMEL__  
    ... ARMEL-spezifischer Code ...  
#endif
```

- (Tipp: Anzeigen der vordefinierten Präprozessorkonstanten:
touch empty.c;<Plattform>-gcc -E -dM empty.c)

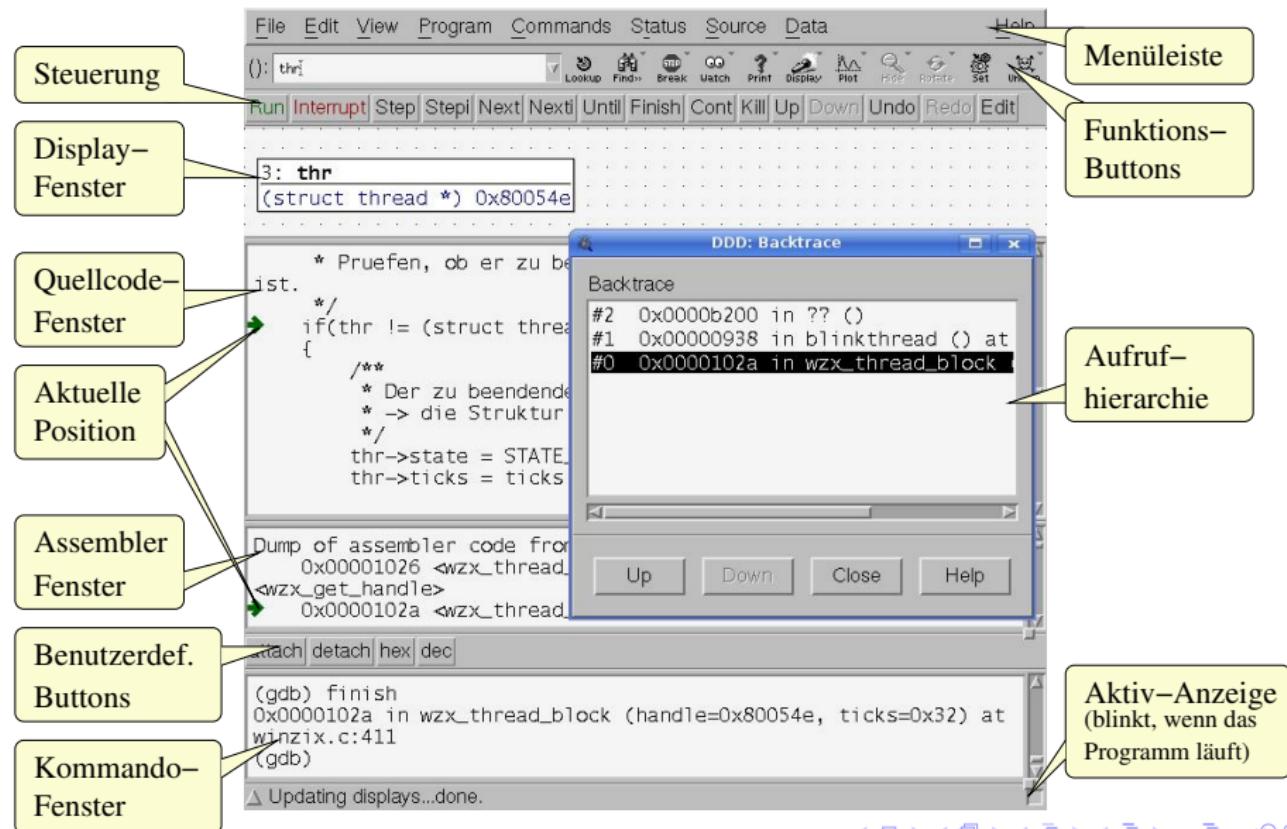
Cross-Entwicklung: Download

- Ziel: Ausführbares Programm (Executable) ins Target bringen
- Je nach Zielsystem unterschiedlichste Vorgehensweisen, z.B.:
 - ▶ Einspeichern in ein (E)(E)PROM mit Hilfe eines (externen) Programmiergerätes → Speicher muss zum Programmieren aus- und wiedereingebaut werden
 - ▶ Einspeichern in Flash-Speicher über in-System-Programmer- (ISP-) Schnittstelle → Schnittstellenspezifisches Programmier-Tool (z.B. stlink, avrdude) erforderlich
 - ▶ Laden des Codes über serielle Schnittstelle oder Netzwerk → erfordert ein *Bootloader*-Programm auf der Target-Seite
 - ▶ Laden des Programmcodes von einem Massenspeicher (z.B. CompactFlash oder USB-Stick) → erfordert ebenfalls einen *Bootloader* auf der Target-Seite
- Typische Dateiformate für Executables: ELF (.elf), Intel-Hex (.hex), S-Record (.srec, .sr), (Raw) Binary (.bin)
- Umwandeln zwischen diesen Formaten z.B. mit objcopy

Debugging

- Eigentlich: engl. *Bug* = Käfer, Insekt, Wanze, Laus
- In der Informationstechnologie: *Bug* = Programmierfehler
- *Debugging*: Finden und Entfernen von Fehlern
- *Debugger*: Hilfsprogramm, das die Fehlersuche ermöglicht bzw. erleichtert
- Ausführen des (evtl. fehlerhaften) Programmes unter der Kontrolle des Debuggers
 - ▶ Laden und Starten des Programmes
 - ▶ Ausführung –u.U. bedingt– anhalten (Breakpoint) und fortführen
 - ▶ Schrittweise Programm-Abarbeitung (Single-Step)
 - ▶ Variablenwerte und Registerinhalte beobachten und manipulieren
 - ▶ Aufrufhierarchie anzeigen
- GNU-Debugger `gdb`: Bedienung über Kommandozeile
- Verschiedene grafische Frontends: DDD, Insight, Kgdb, Eclipse, Nemiver

Beispiel: GDB-Frontend *DDD*



Cross-Debugging

- Funktionsweise eines Debuggers: „Fernsteuern“ der Programmausführung
- Dazu nötige Basisfunktionen:
 - ▶ Prozessor anhalten / weiterlaufen lassen
 - ▶ Register des (angehaltenen) Prozessors lesen / schreiben
 - ▶ Daten- **und** Programmspeicher Lesen **und** Schreiben
(Programmspeicher-Schreibzugriff ist zum Setzen von Breakpoints erforderlich)
 - ▶ Ausnahmebedingungen (z.B. Speicherzugriffsfehler, Nulldivision, Ungültiger Maschinenbefehl) abfangen und Prozessor anhalten
- *self-hosted* Debugger erreichen dies über spezielle Betriebssystemfunktionen (z.B. Linux: `ptrace(2)`)
- Ein *cross*-Debugger benötigt dazu einen (i.d.R. externen) „Debug-Server“

Debug-Server

- Allgemein: Client-Server Architektur:

- ▶ **Server:** Bietet Dienste (hier: Debug-Basisfunktionen)
- ▶ **Client:** Fordert Dienste an, wartet auf Ergebnis
- ▶ Nachrichtenbasierte Kommunikation: Übermitteln von Anforderungen / Ergebnissen in Form von Bytesequenzen

⇒ Zwischen Debugger und Debug-Server muss lediglich ein bidirektonaler, sequenzieller Kommunikationskanal existieren, z.B.:

- Seriell (RS-232)
- Netzwerk (UDP- oder TCP/IP-Socket)
- Logisch (UNIX Domain Socket, UNIX-Pipe, Pseudo-TTY)

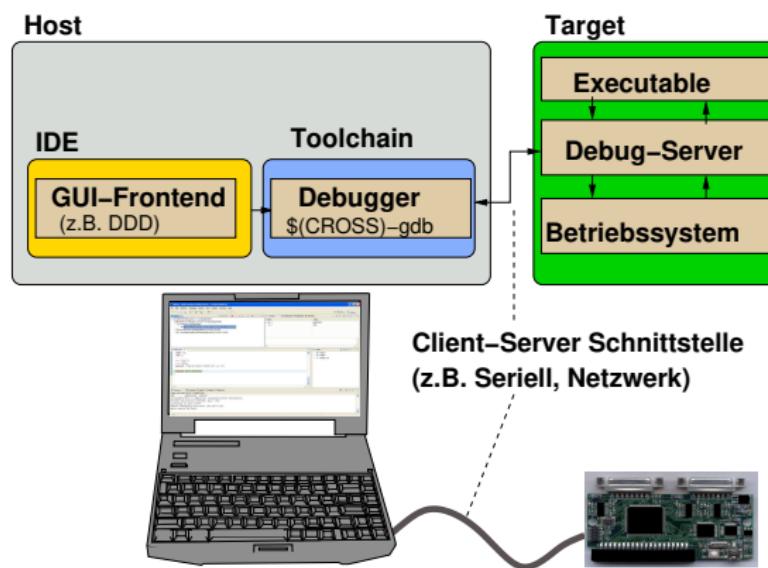
⇒ Debugger und Debug-Server können...

- auf getrennten Rechnern arbeiten, oder
- getrennte Tasks auf einem Rechner sein

- Verschiedene Konstellationen möglich...

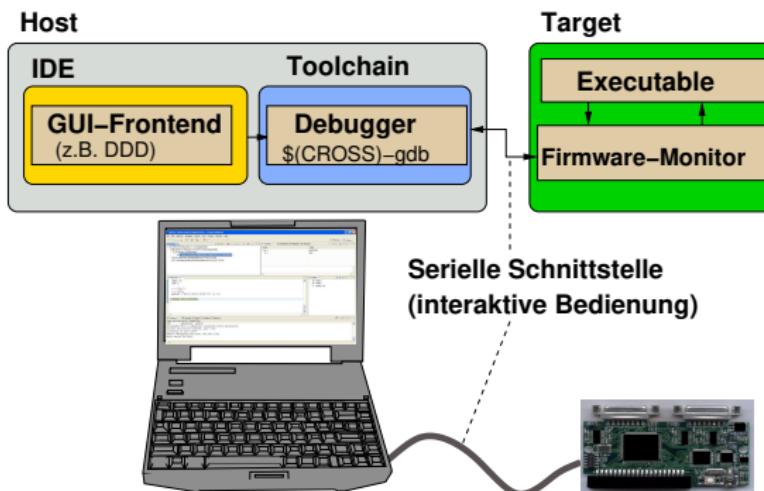
Debug-Server auf dem Target

- Server agiert als „Brückenkopf“ des Debuggers im Target
- In das Betriebssystem integriert oder eigene Task (z.B. gdbserver)
- Unterstützung durch das Target-Betriebssystem erforderlich (Netzwerk-Stack, etc.)



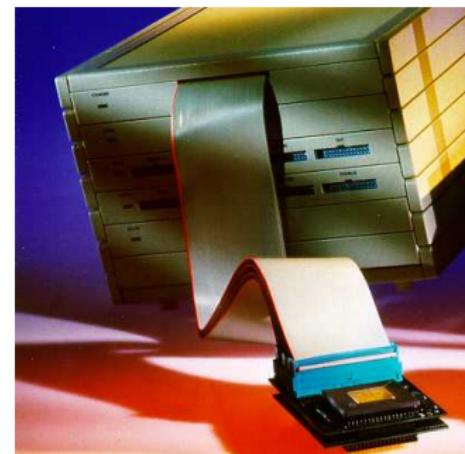
Nutzung der Target-Infrastruktur

- Eine ggf. vorhandene Monitor-Firmware dient als Debug-Server
- Kommunikation i.d.R. über serielle (RS-232) Schnittstelle
- Anpassung des Debuggers an das Monitor-Protokoll erforderlich



In-Circuit Emulator (ICE)

- Hardware, ersetzt den eigentlichen Prozessor bzw. Microcontroller
- Spezielle Version des zu emulierenden Chips mit erweiterten Funktionen zur Steuerung
 - ▶ Z.B. *Bond-out-Chip*: Interne Signale herausgeführt ...
 - ▶ ... oder Implementierung der Funktionalität als FPGA
- Erweiterte Möglichkeiten
 - ▶ Tracebuffer: „Mitschneiden“ des Echtzeitverhaltens
 - ▶ Watchpoints (bedingte Unterbrechungen)
- I.d.R. **sehr** teuer!



JTAG-Schnittstelle: Motivation

- Bausteine werden zunehmend komplexer → Anzahl der Pins steigt → Pins werden immer dichter gepackt
- Zugang zu Chip-internen Signalen wird (*Bond-out-Chip*) schwierig bis unmöglich (Ähnliche Situation auch bei Multilayer-Platinen)
- Das bisher übliche Testen von Chips und Platinen durch Anlegen von Testsignalen (*Stimuli*) und Messen der Antwortsignale (*Responses*) ist nicht mehr praktikabel
- Mitte der 80er Jahre entwickelte ein Firmenkonsortium (die *Joint Test Action Group – JTAG*) das *Boundary Scan* Verfahren
- Ziel: Ein- und Ausgangssignale Chip-intern abfragen und seriell¹ nach außen leiten

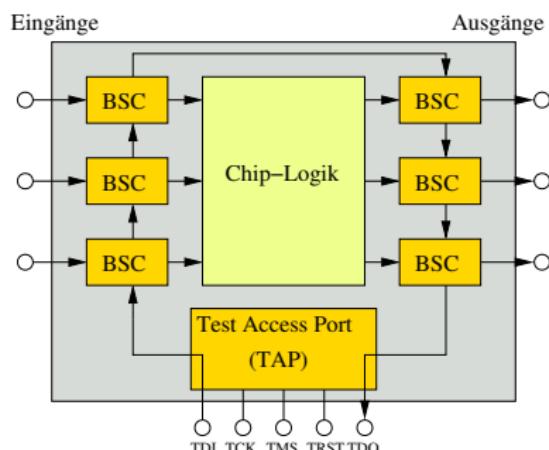
¹erfordert nur wenige Pins

JTAG-Schnittstelle: Aufbau (1)

- Ein- und Ausgangsleitungen werden über *Boundary Scan Cells* (BSCs) geführt
- Ein- und Ausgänge der BSCs werden zu einer Kette verbunden
- Ansteuerung der BSCs über einen *Test Access Port* (TAP)
- TAP-Signale:

- ① **TDI**: Test Data In
- ② **TDO**: Test Data Out
- ③ **TCK**: Test Clock
- ④ **TMS**: Mode Select
- ⑤ **TRST**: Reset (opt.)

- Ansteuerung / Abfrage **aller** Chip-Signale über nur 5 Pins

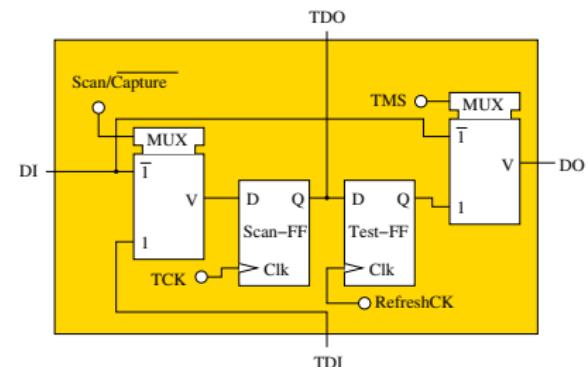


JTAG-Schnittstelle: Aufbau (2)

- Aufbau einer *Boundary Scan Cell* (BSC):

- Betriebsmodi:

- ① **Normal:** TMS = 0
- ② **Capture:** DI → Scan-FF
- ③ **Scan:** TDI → Scan-FF
- ④ **Refresh:** Scan → Test
- ⑤ **Test:** TMS = 1



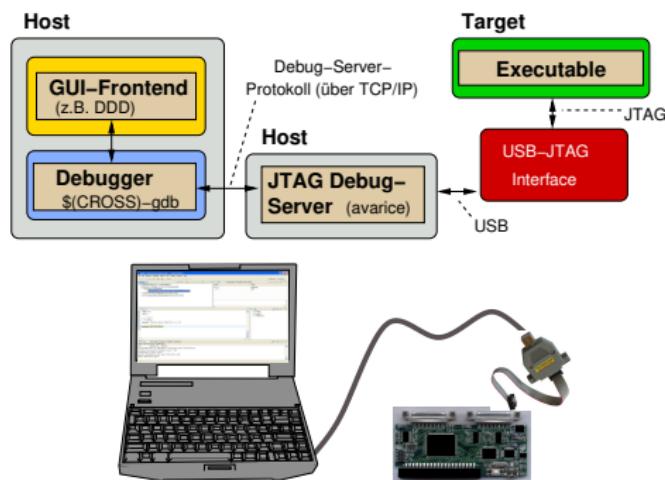
- Durch die „Kettenschaltung“ bilden die Scan-Flipflops aller BSCs im Scan-Modus ein großes Schieberegister
- Eingänge des Chip können mit beliebigen Stimuli beaufschlagt werden, Ausgänge können seriell ausgelesen werden

Cross-Debugging mit JTAG

- JTAG in Mikrocontrollern bildet eine Infrastruktur zum Test, aber auch zum Zugang zu Prozessor-internen Signalen
- Chip-Hersteller haben an dieser Infrastruktur Erweiterungen vorgenommen, um über JTAG ...
 - ▶ ... Programme in den Flash-Speicher zu übertragen
 - ▶ ... Programme schrittweise abzuarbeiten
 - ▶ ... Breakpoints und Watchpoints zu setzen
- Diese Erweiterungen des JTAG-Protokolls sind herstellerspezifisch
- Damit können (mit Hilfe eines relativ preiswerten JTAG-Adapters) ähnliche Funktionen wie mit einem ICE erreicht werden

Beispiel: AVR Cross-Debugging

- **avarice**: Debug-Server für AVR (\geq ATmega16) JTAG-Schnittstelle
 - Arbeitet auf Host
 - Kommuniziert mit Debugger über Netzwerk (*TCP/IP-Socket*)
- Kann auf demselben oder einem anderen Hostrechner arbeiten



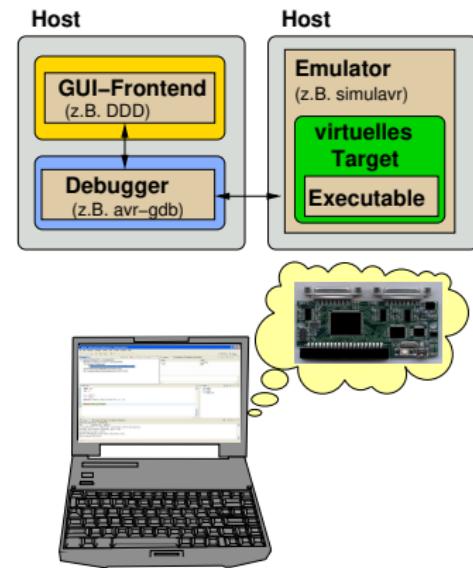
Virtuelle Target-Maschine

- Das Target wird durch ein Programm vollständig nachgebildet (*emuliert*)
- Dieser *Emulator* stellt damit eine *virtuelle Maschine* zur Verfügung, die –abgesehen von der Geschwindigkeit– exakt das Verhalten des Targets zeigt
- Er besitzt einen integrierten Debug-Server, über den ein Debugger die virtuelle Maschine steuern kann
- Beispiele: Bochs, Qemu (x86), simulavr (AVR), Android (Java)
 - + Softwareentwicklung ohne Target möglich
 - + Sehr weitgehende Kontrollmöglichkeiten (ähnl. ICE)
 - Nicht immer verfügbar
 - Zeitverhalten des Targets wird nicht korrekt nachgebildet (i.d.R. langsamer)
 - E/A-Bausteine werden u.U. nicht oder nur unvollständig nachgebildet

Beispiel (1): Cross-Debugging mit simulavr

- Target existiert als virtuelle Maschine auf dem Host

- **simulavr**: Emulator für AVR Microcontroller
- Emuliert nur den Prozessor, *keine E/A-Komponenten*
- Nur bedingt brauchbar
- Kommunikation mit Debugger über Netzwerk (*TCP/IP-Socket*)



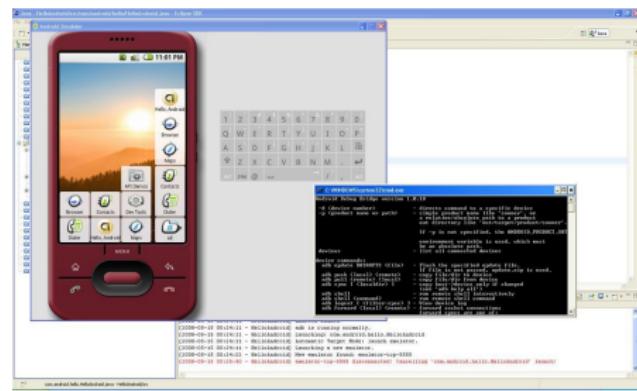
- Auch hier: Emulator kann auf demselben oder einem anderen Hostrechner arbeiten

Beispiel (2): Google Android

- Offene Entwicklungsumgebung für Smartphones²

- Hier:

- ▶ Programmierung in Java
- ▶ Peripherie wird durch die JVM emuliert
- ▶ Kein Echtzeit-Verhalten gefordert



- Anwendungsentwicklung war bereits vor Verfügbarkeit entsprechender Geräte möglich

²siehe <http://code.google.com/android/>

3.2 Hardwarenahes Programmieren in C



Motivation

- Vorteil der Assemblerprogrammierung: vollkommene Kontrolle über die Maschine
 - ▶ Effizientester Code (?)
 - ▶ Größte Freiheiten in der Wahl des Programmiermodells
 - ▶ Zugriff auf Besonderheiten der Architektur
(Register, Port-Mapped I/O, etc.)
- Nachteile:
 - ▶ Höchster Anspruch an Fähigkeit und Kenntnisse der ProgrammiererInnen
 - ▶ Programme (und Kenntnisse) sind nicht „portabel“
 - ▶ Komplexe Algorithmen kaum in Assembler beherrschbar
 - ▶ Fehlerträchtig: z.B. kein Typkonzept
- Heute wird nur noch in seltenen Ausnahmefällen in Assembler programmiert (i.d.R. um Dinge zu tun, die in Hochsprache „nicht gehen“, z.B. Interrupts maskieren, auf Register zugreifen, etc.)

Was ist „hardwarenahes Programmieren“?

- Z.B. Google-Suche liefert keine klare Definition des Begriffs
(Stattdessen aber sehr viele interessante Stellenangebote...)
- Vorstellung / Kenntnis der Vorgänge innerhalb des Rechensystems beim Ausführen von (Hochsprache-)Programmen
 - ▶ Speicherorganisation (Code / Daten / Stack / Heap)
 - ▶ Aufrufsschnittstelle (*Call by Value / Reference* und Konsequenzen)
 - ▶ Fallstricke (Pufferüberlauf, Stack-Überlauf, Nebenläufigkeit, Compileroptimierungen, Caches, virtuelle Adressierung...)
 - ▶ Tipps und Tricks: Effizienzoptimierung, Fehlersuche
- „Blick über den Tellerrand“ des Hochsprachen- Modells
- Jedes Rechensystem braucht zwingend hardwarenahe Programme
(Betriebssysteme / Gerätetreiber / E/A-Bibliotheken)
- Stark nachgefragte (aber selten anzutreffende) Fähigkeit

Warum C?

- C ist eine portable Programmiersprache, d.h. C-Programme können im Prinzip™ auf jeder Maschine laufen, für die es einen C-Compiler gibt
- Dennoch sind typische maschinenspezifische Konzepte wie Stack, Speicheradressen, etc. in C noch zugänglich
- Sprache ist unabhängig von Laufzeitbibliothek
- In der Regel benutzt jeder C-Compiler eine exakt definierte Schnittstelle (*Application Binary Interface*, ABI) zum Maschinencode
- Dadurch können aus C heraus problemlos Assembler-Unterprogramme aufgerufen werden (und umgekehrt)
- Mit `asm` können zudem bei den meisten C-Compilern Assemblerbefehle direkt in C-Code eingebettet werden (*inline assembler*)

Beispiel: inline-Assembler

Inline-Assembler-Routine (GNU C)

```
int set_stack_and_go(stackp, entry)
void *stackp;
int (*entry)();
{
#ifndef __i386__
asm("mov %%eax,%0 \
     mov %%esp,%1 \
     push %%eax \
     ret " : : "p" (entry), "p" (stackp) : "%eax", "%esp");
#endif
#ifndef __m68k__
asm("moveal %0,%a0 \
     moveal %1,%sp \
     jmp    %%a0@" : : "p" (entry), "p" (stackp) : "%a0", "%sp");
#endif
#ifndef __ppc__
.....
#endif
}
```

Speicherorganisation

- Ein Programm und seine Daten werden während der Ausführung vollständig im Hauptspeicher gehalten³
- Die Programm- und Datenobjekte lassen sich in verschiedene Klassen einteilen
- Der Compiler/Linker ordnet die statischen (d.h. zur Compilezeit bekannten) Objekte je nach Klasse bestimmten „Sektionen“ zu:

Klasse	Lesen	Schreiben	Ausführen	Initialisiert	Sektion
Programmcode	x	-	x	x	.text
initialisierte Daten	x	x	-	x	.data
uninitialisierte Daten	x	x	-	-	.bss
nur-lese Daten	x	-	-	x	.rodata

- Für jede Sektion wird ein entsprechend großes, zusammenhängendes Stück Speicher eingeteilt.

³Virtueller Speicher/Paging werden hier nicht betrachtet

Dynamische Daten

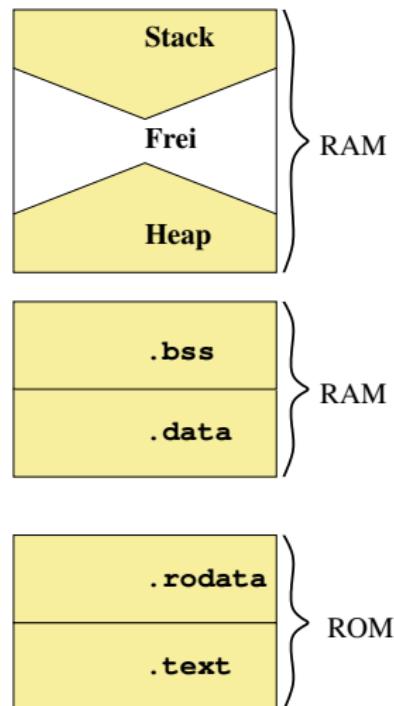
- Der restliche (freie) Speicher wird dynamisch, d.h. zur Laufzeit zugeteilt:
 - ▶ **Heap:** Dynamischer Speicher
 - ★ Wird in C mit `malloc()` zugeteilt und mit `free()` wieder freigegeben (In C++ mit `new` und `delete`)
 - ★ Wächst nach „oben“, d.h. zu höheren Adressen hin
 - ★ Anfang typischerweise direkt im Anschluss an statische Daten (.bss)
 - ▶ **Stack:** Lokale (dynamische) Variablen
 - ★ Sämtliche Automatic-Variablen in C (ausserdem `alloca()`-Funktion)
 - ★ Auch: Aufrufhierarchie (d.h. Rückkehradressen) und übergebene Parameter
 - ★ Zuteilung / Freigabe automatisch durch Inkrementieren bzw. Dekrementieren des Stackpointers (z.B. mit PUSH- und POP-Befehlen)
 - ★ Wächst (i.d.R.) nach „unten“, d.h. zu kleiner werdenden Adressen hin
 - ★ Anfang typischerweise am oberen Ende des RAM-Speichers
- Stack und Heap wachsen einander entgegen. Wenn sie „zusammenstoßen“ geschehen „merkwürdige Dinge“...

Beispiel

Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =      "Hallo";
int b;

main(int argc, char *argv[])
{
int i;
int l =      strlen(s);
char *p;
p =      malloc(l + 1);
for(i = 0; i < l; i++)
    p[i]= s[i];
}
```

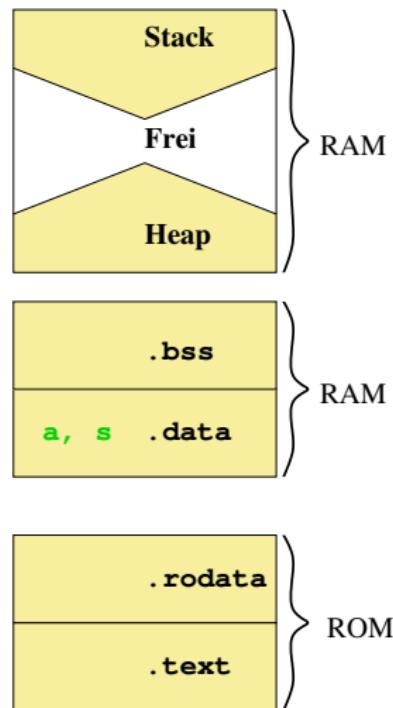


Beispiel

Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =    "Hallo";
int b;

main( int argc, char *argv[])
{
int i;
int l =      strlen(s);
char *p;
p =      malloc(l + 1);
for(i = 0; i < l; i++)
    p[i]= s[i];
}
```

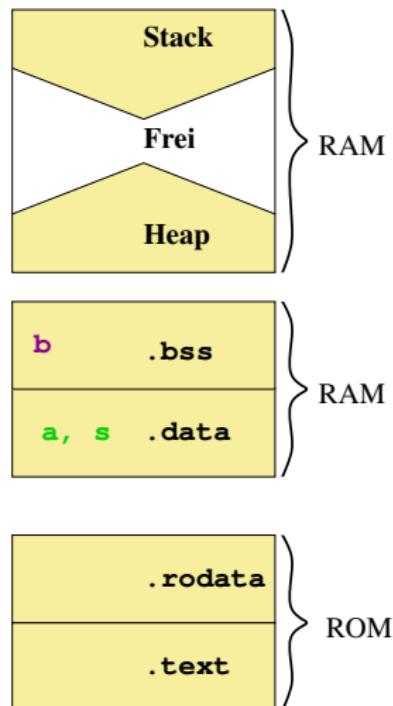


Beispiel

Speicherorganisation eines C-Programms

```
static int a = 5;
char *s = "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```

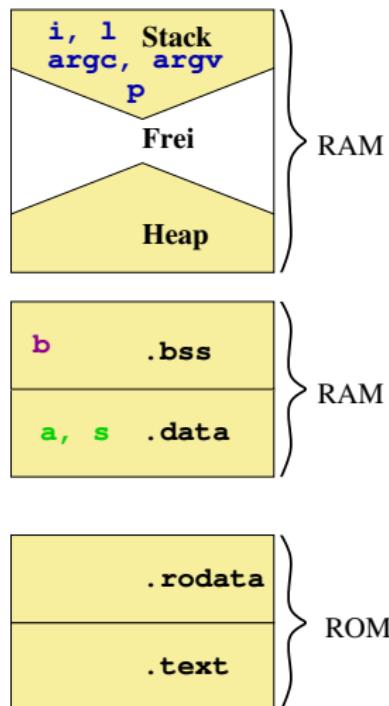


Beispiel

Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
int i;
int l =      strlen(s);
char *p;
p =      malloc(l + 1);
for(i = 0; i < l; i++)
    p[i]= s[i];
}
```



Beispiel

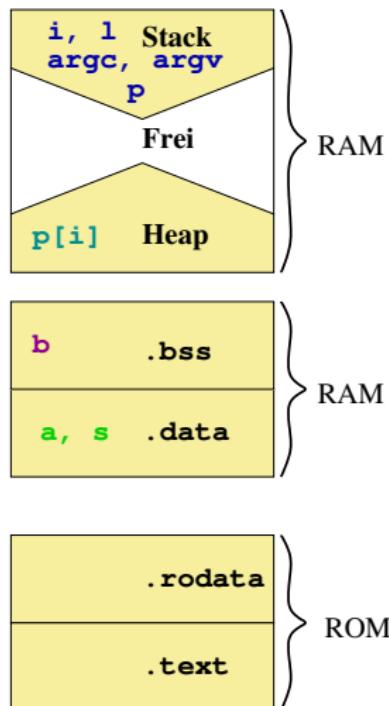
Speicherorganisation eines C-Programms

```

static int    a = 5;
char *s =      "Hallo";
int b;

main(int argc, char *argv[])
{
int i;
int l =      strlen(s);
char *p;
p =      malloc(l + 1);
for(i = 0; i < l; i++)
    p[i]= s[i];
}

```

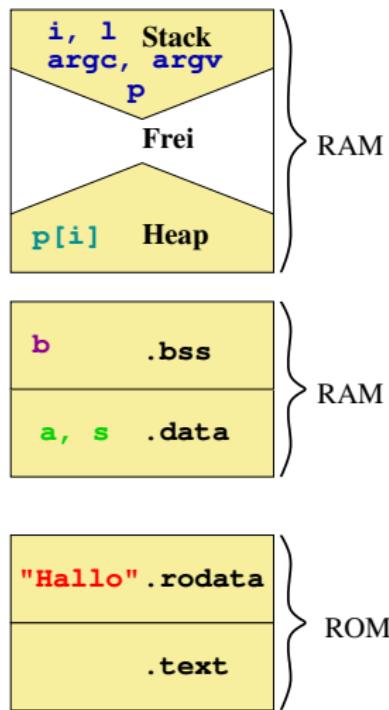


Beispiel

Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =      "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l =      strlen(s);
    char *p;
    p =      malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i]= s[i];
}
```



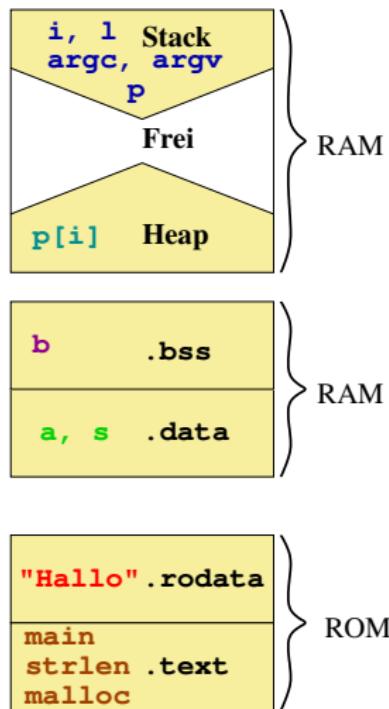
Beispiel

Speicherorganisation eines C-Programms

```

static int    a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
int i;
int l =      strlen(s);
char *p;
p =      malloc(l + 1);
for(i = 0; i < l; i++)
    p[i]= s[i];
}
  
```



Initialisierung – wie? (1)

- Die Sektionen .text, .rodata, .data und .bss müssen vor dem Start des Programms initialisiert (d.h. mit definiertem Inhalt versehen) werden
- Wird das Programm von einem Betriebssystem (z.B. Windows oder Linux) gestartet, so lädt dessen „Program Launcher“ die Speicherinhalte aus einer ausführbaren Datei (z.B. `program.exe` in Windows) ins RAM
 - Programmcode und nur-lese-Daten liegen dann im RAM und sind –technisch gesehen– jederzeit zur Laufzeit überschreibbar
 - Potenzielles Sicherheitsproblem, zugleich aber auch Möglichkeit des Debuggings mit Breakpoints, etc.

Initialisierung – wie? (2)

- Arbeitet das Programm ohne Betriebssystem, so liegen .text und .rodata z.B. im nicht-flüchtigen ROM-Speicher, d.h. sie sind beim Programmstart bereits initialisiert
 - Die Sektionen .data und .bss müssen zur Laufzeit schreibbar sein, d.h. sie können nur im (flüchtigen) RAM liegen
 - Der Compiler / Linker legt dazu eine „Schattenkopie“ mit den Initialisierungswerten der .data Sektion ins ROM
 - Vor dem Aufruf von `main()` wird eine (ebenfalls vom Linker automatisch hinzugefügte) Initialisierungsroutine aufgerufen, die die „Schattenkopie“ aus dem ROM ins RAM kopiert und die .bss Sektion mit Nullen füllt
- Initialisierte Daten verursachen Platzbedarf in ROM **und** RAM

C „Startup Code“

- Jede C-Programmierumgebung fügt einem kompilierten Programm implizit einen „Startup-Code“ hinzu
- Aufgabe: Umgebung für `main()` herstellen ...
 - ▶ .data Sektion initialisieren (macht ggf. auch das Betriebssystem)
 - ▶ .bss Sektion „nullen“ (dto.)
 - ▶ Ggf. Umgebungsvariablen und `argc` / `argv[]` aufbereiten
 - ▶ Bei C++-Programmen: Konstruktoren der statischen Klassen aufrufen
- ... dann `main()` aufrufen
 - ▶ Ggf. `argc` / `argv[]` übergeben
 - ▶ Falls `main()` zurückkehrt: `_exit()` aufrufen
- Startup-Code ist Maschinen-, Compiler- und Betriebssystemabhängig

Beispiel: Multiboot-Startup (1)

Label	Code	Kommentare
<code>_begin:</code>	<code>jmp multiboot_entry</code>	
	<code>.align 4</code>	Align 32 bits boundary
<code>multiboot_header:</code>		Multiboot header magic
	<code>.long MULTIBOOT_HEADER_MAGIC</code>	
	<code>.long MULTIBOOT_HEADER_FLAGS</code>	flags
	<code>.long -(MULTIBOOT_HEADER_MAGIC</code>	
	<code>+MULTIBOOT_HEADER_FLAGS)</code>	checksum
	<code>.long multiboot_header</code>	header_addr
	<code>.long _begin</code>	load_addr
	<code>.long _edata</code>	load_end_addr
	<code>.long _end</code>	bss_end_addr
	<code>.long multiboot_entry</code>	entry_addr
<code>multiboot_entry:</code>		
	<code>movl \$_end,%esp</code>	Initialize the stack pointer
	<code>addl \$STACK_SIZE,%esp</code>	
	<code>pushl \$0</code>	Reset EFLAGS
	<code>popf</code>	
	<code>pushl %ebx</code>	Push pointer toMultiboot struct
	<code>pushl %eax</code>	Push magic value
	<code>call __init_pc</code>	Now enter the C code

Beispiel: Multiboot-Startup (2)

```
__init_pc(int magic, multiboot_header *hdr)
{
    unsigned char *to;
    volatile int count;
    /* clear BSS section: */
    to = (unsigned char *)__bss_start;
    count = (unsigned char *)__end - to;

    if(count > 0) do
    {
        *to++ = 0;
    }
    while(--count);

    asm( /* reset stack, jump to main: */
        "movl  %0,%esp          \n"
        "addl  $STACK_SIZE, %%esp \n"
        "jmp   main              \n"
        :: "i" (__end));
}
```

Prozeduraufruf: Nötige Schritte

- Parameter **kopien** für aufgerufene Funktion hinterlegen
- Programmsteuerung an die Prozedur übergeben
- Arbeitsspeicher für Prozedur bereitstellen (lokale Variablen)
- Prozedur ausführen
- Ergebnis an den Aufrufer übergeben
- Arbeitsspeicher der Prozedur wieder freigeben
- zum Aufrufer zurückkehren

Beispiel: Prozeduraufruf

```
int funktion(int p1, int p2, ...)  
{  
    int lokal1, lokal2, ...;  
    int ergebnis;  
    ....  
    return(ergebnis);  
}  
  
main()  
{  
    int a, b, c, ...;  
    int resultat;  
    ....  
    resultat = funktion(a, b, ...);  
    ....  
}
```

Prozeduraufruf

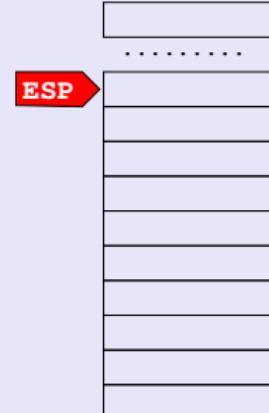
- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

Beispiel: Prozederaufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
EPC pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Prozeduraufruf

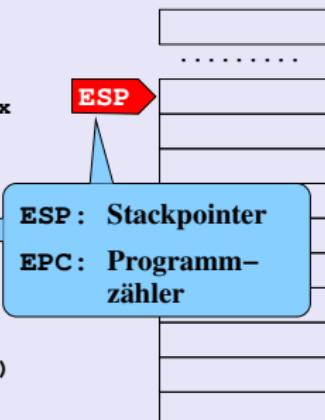
- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

Beispiel: Prozederaufruf

```
int proz(int a, int b)
{
    return(a+b);
}

.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(epb),eax
addl 8(epb),eax
popl ebp
ret
.....
EPC pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(epb)
add#8,esp
```



Prozeduraufruf

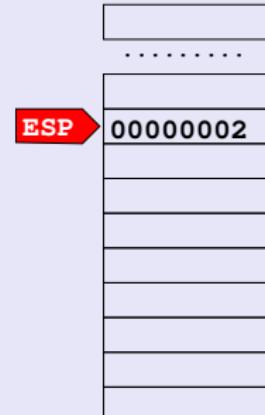
- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

Beispiel: Prozederaufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(epb),eax
addl 8(epb),eax
popl ebp
ret
.....
pushl #2
EPC pushl #1
call proz
rueckadr:
move eax,rw(epb)
add#8,esp
```



Prozeduraaufruf

- ① Argumentkopien auf den Stack
 - ② Call → Rücksprungadresse auf den Stack

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}

.....
rw = proz(1, 2);

```

Assembler:	C:
pushl ebp	push x * (--esp) = x
movl esp,	
movl 12(ep),eax	
addl 8(ep),eax	
popl ebp	
ret	
.....	
pushl #2	
pushl #1	
call proz	
rueckadr:	
move eax, rw(ep)	
add#8,esp	

Prozeduraufruf

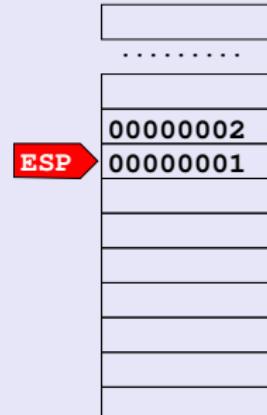
- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}

.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
EPCcall proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Prozeduraufruf

- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

Beispiel: Prozederaufruf

```
int proz(int a, int b)
{
    return (a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

Call bewirkt:

1. Rücksprungadresse auf Stack
2. Programmzähler = Ziel

```
proz:
pushl
movl
movl
addl
popl
ret
.....
```

(S=Größe des Call-Opcodes)

EPC	call proz	
	rueckadr:	
	move eax, rw(ebp)	
	add#8, esp	

Prozeduraufruf

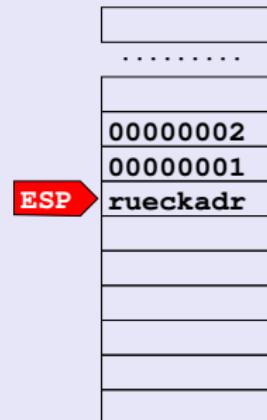
- ③ „Framepointer“ (EBP) retten
- ④ Neuen Framepointer laden

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
EPC pushl ebp
        movl esp,ebp
        movl 12(ebp),eax
        addl 8(ebp),eax
        popl ebp
        ret
        .....
        pushl #2
        pushl #1
        call proz
rueckadr:
        move eax,rw(ebp)
        add#8,esp
```



Prozeduraufruf

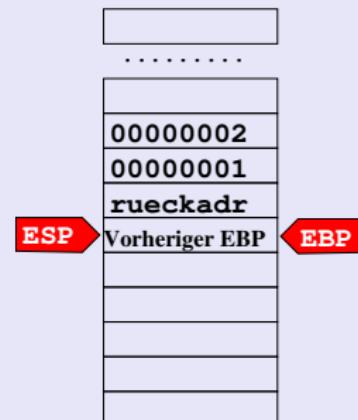
- ③ „Framepointer“ (EBP) retten
- ④ Neuen Framepointer laden

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return (a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    EPC   movl esp,ebp
          movl 12(epb),eax
          addl 8(epb),eax
          popl ebp
          ret
    .....
    pushl #2
    pushl #1
    call proz
    rueckadr:
    move eax,rw(epb)
    add#8,esp
```



Prozeduraufruf

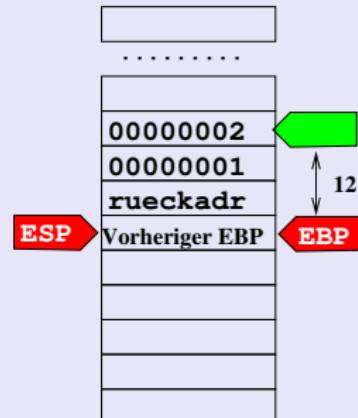
- ⑤ Parameter addieren
- ⑥ Returnwert in Register (hier: EAX)

Beispiel: Prozeduraufruf

```
int proz( int a, int b )
{
    return( a+b );
}

.....
rw = proz( 1, 2 );
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(epb),eax
addl 8(epb),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(epb)
add#8,esp
```



Prozeduraufruf

- ⑤ Parameter addieren
- ⑥ Returnwert in Register (hier: EAX)

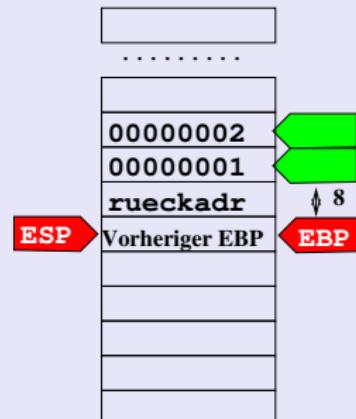
Beispiel: Prozeduraufruf

```
int proz( int a, int b )
{
    return (a+b);
}
```

.....

```
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(epb),eax
EPC addl 8(epb),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(epb)
add#8,esp
```



Prozeduraufruf

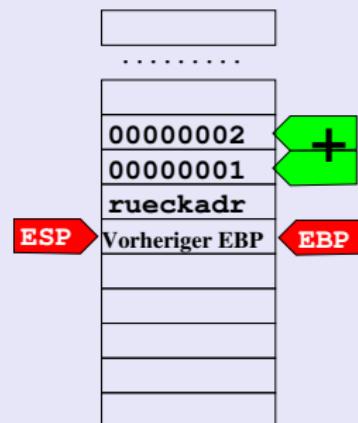
- ⑤ Parameter addieren
- ⑥ Returnwert in Register (hier: EAX)

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
EPC addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Prozeduraufruf

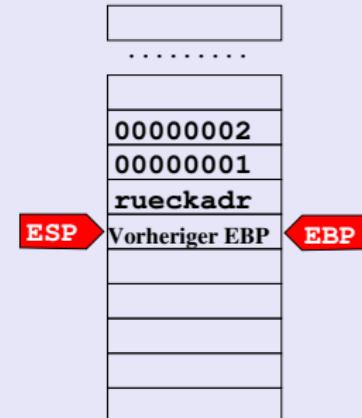
- 7 Alten Framepointer wiederherstellen
- 8 Rückkehr zum Aufrufer

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return (a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(epb),eax
addl 8(epb),eax
EPC popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(epb)
add#8,esp
```



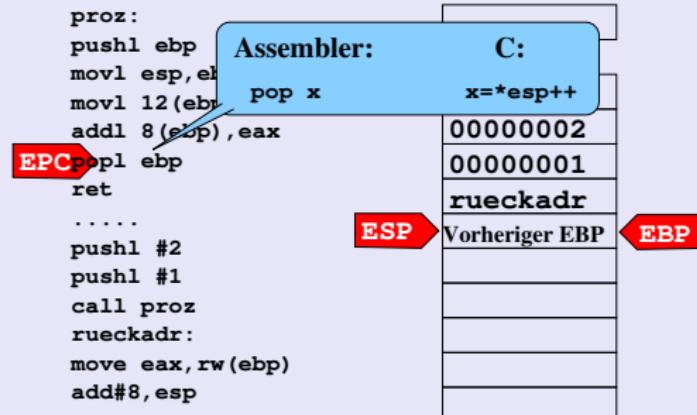
Prozeduraufruf

- ⑦ Alten Framepointer wiederherstellen
- ⑧ Rückkehr zum Aufrufer

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return (a+b);
}

.....
rw = proz(1, 2);
.....
```



Prozeduraufruf

- ⑦ Alten Framepointer wiederherstellen
- ⑧ Rückkehr zum Aufrufer

Beispiel: Prozeduraufruf

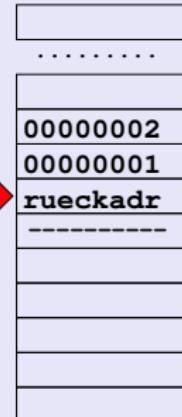
```
int proz(int a, int b)
{
    return (a+b);
}

.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```

EPC_{ret}

ESP



Prozeduraufruf

- ⑦ Alten Framepointer wiederherstellen
- ⑧ Rückkehr zum Aufrufer

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}

.....
rw = proz(1, 2);
.....
```

Assembler:	C:
proz:	epc=*esp++
pushl ebp	00000002
movl esp,ebp	00000001
ret	rueckadr
movl 12(ebp),eax	-----
addl 8(%bp),eax	
popl ebp	
.....	
pushl #2	
pushl #1	
call proz	
rueckadr:	
move eax,rw(%bp)	
addl #8,esp	

EPCret ESP

Prozeduraaufruf



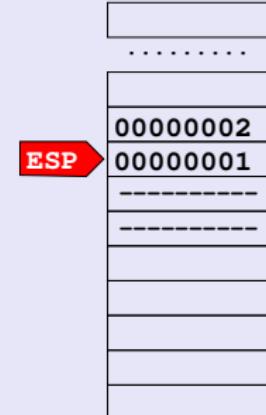
- ⑨ Ergebnis Speichern
 - ⑩ Stack abräumen

Beispiel: Prozeduraufruf

```
int proz( int a, int b)
{
    return(a+b);
}

.....
rw = proz(1, 2);
```

```
proz:  
pushl ebp  
movl esp,ebp  
movl 12(ebp),eax  
addl 8(ebp),eax  
popl ebp  
ret  
.....  
pushl #2  
pushl #1  
call proz  
rueckadr:  
EPCmove eax,rw(ebp  
add#8,esp
```



Prozeduraufruf

⑨ Ergebnis Speichern

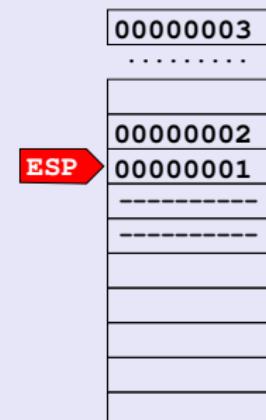
⑩ Stack abräumen

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return (a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
```



Prozeduraufruf

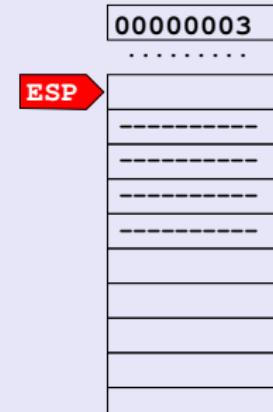
- ⑨ Ergebnis Speichern
- ⑩ Stack abräumen

Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
pushl ebp
movl esp,ebp
movl 12(ebp),eax
addl 8(ebp),eax
popl ebp
ret
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax,rw(ebp)
add#8,esp
.....
```



Prozeduraufruf – Fazit (1)

- Beträchtlicher Aufwand
- Nur sinnvoll, wenn „Nutzcode“ >> Code für Prozeduraufruf
(Im Beispiel: Eine Instruktion Nutzcode (Addition) vs. 10 Instruktionen für den Prozeduraufruf)
- In solchen Fällen effizienter:

① Präprozessor-Macro

```
#define proz(a, b) ((a)+(b));
```

② inline-Funktion

```
inline int proz(int a, int b)
{
    return (a+b);
}
```

Prozeduraufruf – Fazit (2)

- Wertparameter (*Call by value*) → Parameterkopien werden über den Stack übergeben
 - Einige (typischerweise RISC-)Architekturen verwenden alternativ auch Prozessorregister für die Übergabe (effizienter – warum?)
 - Kein Problem wenn die Größe der Objekte \leq Registergröße
- ? .. aber wie soll dann so etwas gehen:

Beispiel: Übergabeparameter \geq Registergröße

```
struct parameterlist {           main()
    int elem1;                   {
    int elem2;                   ...
    ...                           struct parameterlist x;
    int elemN;                   funktion(x);
};                                ...
}
```

Auswirkungen von *Call by Value*

- Größere Objekte „passen“ nicht in Register, können nur über den Stack kopiert werden
- Laufzeitaufwand und ggf. großer Stackbedarf
- Wird von älteren C-Compilern z.T. gar nicht unterstützt
- Meistens besser: Statt Objektkopie einen **Zeiger** auf das Objekt übergeben (Eigentlich ist das *Call by reference*)

Beispiel: Übergabe von Zeiger statt Objektkopie

```
struct parameterlist {           main()
    int elem1;
    int elem2;
    ...
    int elemN;
};

main()
{
    ...
    struct parameterlist x;
    funktion(&x);
    ...
}
```

Prozeduraufruf – Fazit (3)

- Rückgabewerte werden in einem Register zurückgeliefert

? Wie ist dann so etwas möglich:

Beispiel: Rückgabeobjekt \geq Registergröße

```
struct parameterlist {  
    int elem1;  
    int elem2;  
    ...  
    int elemN;  
};
```

```
struct parameterlist funktion()  
{  
    struct parameterlist x;  
    x.elem1 = ....;  
    ....  
    return(x);  
}
```

- Auch hier gilt:
 - ▶ Wird von älteren C-Compilern z.T. nicht unterstützt
 - ▶ Falls unterstützt: Aufwändiges Kopieren des Objekts (d.h. teuer)
- Besser *nicht* verwenden!

Große Objekte zurückliefern

- Besser: Aufrufer hält Platz für das Rückgabeobjekt vor
- Prozedur erhält Zeiger auf das Objekt, kann dieses manipulieren

Beispiel: Rückgabeobjekt \geq Registergröße

```
struct liste {
    int elem1;
    int elem2;
    ...
    int elemN;
};

void funktion(struct liste *px)
{
    px->elem1 = ....;
    return;
}

void aufrufer(..)
{
    struct liste x;
    ...
    funktion(&x);
    ...
}
```

- Entspricht funktional der von den meisten (aber nicht allen!) Compilern verwendeten Technik

Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

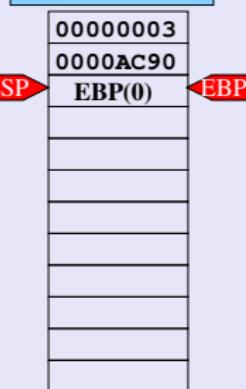
(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```
int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}
```

0100	fak:	pushl ebp
0104		
	movl esp,	ebp
1208		call fak
120C		move eax ,...
135E		ret
AC80		push #3
AC8E		call fak
AC90		move eax ,...

f3 = fak(3));

Tiefe = 1 (n = 3)



Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```

int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}

.....
f3 = fak(3));
.....

```

<pre> 0100 fak: pushl ebp 0104 <stop> movl esp, ebp 1208 call fak 120C move eax, 135E ret AC80 push #3 AC8E call fak AC90 move eax, ... </stop></pre>	Tiefe = 2 (n = 2) <table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>00000003</td></tr> <tr><td>0000AC90</td></tr> <tr><td>EBP(0)</td></tr> <tr><td>00000002</td></tr> <tr><td>0000120C</td></tr> <tr><td>EBP(1)</td></tr> <tr><td>.....</td></tr> <tr><td>.....</td></tr> </table>	00000003	0000AC90	EBP(0)	00000002	0000120C	EBP(1)
00000003									
0000AC90									
EBP(0)									
00000002									
0000120C									
EBP(1)									
.....									
.....									

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
.....
.....

ESP

EBP

Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```
int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}
.....
f3 = fak(3));
.....
```

0100	fak:	pushl	ebp
0104			
	movl	esp,	ebp
		
1208		call	fak
120C		move	eax ,...
135E		ret
AC80		push	#3
AC8E		call	fak
AC90		move	eax ,...

Tiefe = 3 (n = 1)

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
00000001
0000120C
EBP(2)

ESP

EBP

Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```
int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}
```

```
f3 = fak(3));
```

0100	fak:	pushl	ebp
0104			
	movl	esp,	ebp
1208			
120C	call	fak	
	move	eax ,...	
135E	ret		
AC80	push	#3	
AC8E	call	fak	
AC90	move	eax ,...	

Tiefe = 4 (n = 0)

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
00000001
0000120C
EBP(2)
00000000
0000120C
EBP(3)



Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```

int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}

f3 = fak(3));
.....

```

<pre> 0100 fak: pushl ebp 0104 <stop> movl esp, ebp</stop></pre>	<pre> 1208 call fak 120C move eax,</pre>
<pre> 135E ret</pre>	<pre> AC80 push #3 AC8E call fak AC90 move eax, ...</pre>

Tiefe = 3 wert=1

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
00000001
0000120C
EBP(2)

ESP

EBP

Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

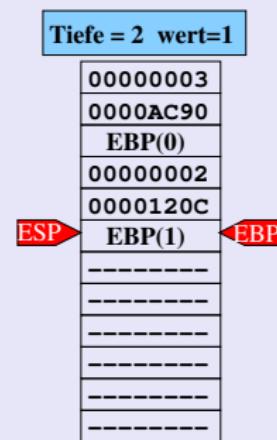
```

int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}

.....
f3 = fak(3));
.....

```

<pre> 0100 fak: pushl ebp 0104 <stop> movl esp, ebp 1208 call fak 120C move eax, 135E ret AC80 push #3 AC8E call fak AC90 move eax, ... </stop></pre>	<div style="border: 1px solid blue; padding: 5px; margin-bottom: 10px;"> Tiefe = 2 wert=1 </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px;">00000003</td></tr> <tr><td style="padding: 2px;">0000AC90</td></tr> <tr><td style="padding: 2px;">EBP(0)</td></tr> <tr><td style="padding: 2px;">00000002</td></tr> <tr><td style="padding: 2px;">0000120C</td></tr> <tr><td style="padding: 2px;">EBP(1)</td></tr> <tr><td style="padding: 2px;">-----</td></tr> </table>	00000003	0000AC90	EBP(0)	00000002	0000120C	EBP(1)	-----	-----	-----	-----	-----	-----
00000003													
0000AC90													
EBP(0)													
00000002													
0000120C													
EBP(1)													



Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$)))

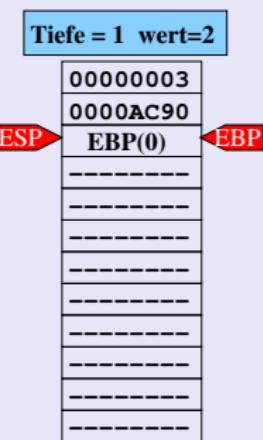
Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```

int fak(int n)          0100  fak: pushl  ebp
{                      0104
    if(0 == n)         STOP movl  esp,  ebp
        return(1);      1208  call   fak
    else               120C  move   eax, ...
        return(n * fak(n-1)); 135E  ret
}
.....
f3 = fak(3));          AC80  push   #3
.....
AC8E  call   fak
AC90  move   eax ...

```



Verschachtelte Prozeduren

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ($n!$))

Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```
int fak(int n)
{
    if(0 == n)
        return(1);
    else
        return(n * fak(n-1));
}

.....
f3 = fak(3));
```



<pre>0100 fak: pushl ebp 0104 movl esp, ebp 1208 call fak 120C move eax, 135E ret AC80 push #3 AC8E call fak AC90 move eax, ...</pre>	<div style="border: 1px solid blue; padding: 2px; display: inline-block;">Tiefe = 0 wert=6</div> <div style="border: 1px solid black; width: 150px; height: 200px; margin-top: 10px; position: relative;"> <div style="position: absolute; left: 0px; top: 0px; width: 100%; height: 100%; background-color: white; display: flex; align-items: center; justify-content: center;">-----</div> <div style="position: absolute; left: 0px; top: 10px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 20px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 30px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 40px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 50px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 60px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 70px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 80px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 90px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 100px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 110px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 120px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 130px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 140px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 150px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 160px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 170px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 180px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 190px; width: 100%; height: 10px; background-color: black;"></div> <div style="position: absolute; left: 0px; top: 200px; width: 100%; height: 10px; background-color: black;"></div> </div>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Rekursion – Fazit

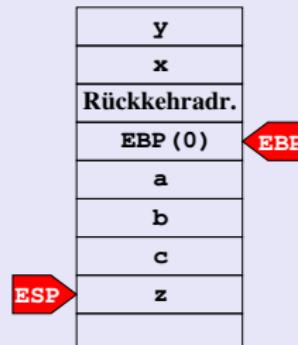
- Sowohl Parameterkopien als auch lokale Variablen erhalten mit jeder neuen Verschachtelung eine eigene Instanz
- Auf statische Variablen trifft das nicht zu!
- Rekursive (reentrant) Prozeduren sollten i.d.R. keine statischen Variablen benutzen

Lokale Variablen

- ① Bisher betrachtete Beispiele hatten keine lokalen Variablen
- ② Stackpointer (ESP) und Framepointer (EBP) waren stets gleich
(Ist der Framepointer damit redundant?)
- ③ Lokale Variablen liegen zwischen EBP und ESP:

Beispiel: Platz für lokale Variablen

```
fkt: pushl ebp
      movl esp,ebp
      subl#16,esp
      movl 8(epb),eax
      movl eax,-4(epb)
      movl 12(epb),eax
      movl eax,-8(epb)
      movl 12(epb),edx
      movl 8(epb),eax
      subl edx, eax
      .....
      ret
fkt(int x, int y)
{
    int a, b, c;
    int z;
    a = x;
    b = y;
    c = x - y;
    z = 2 * (a + b) - c;
}
```



Lokale Variablen

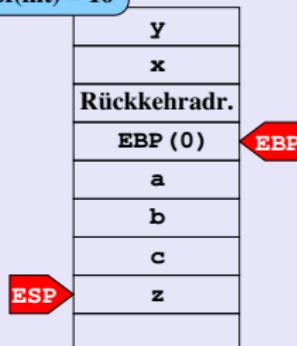
- ① Bisher betrachtete Beispiele hatten keine lokalen Variablen
- ② Stackpointer (ESP) und Framepointer (EBP) waren stets gleich
(Ist der Framepointer damit redundant?)
- ③ Lokale Variablen liegen zwischen EBP und ESP:

Beispiel: Platz für lokale Variablen

```
fkt(int x, int y)
{
    int a, b, c;
    int z;
    a = x;
    b = y;
    c = x - y;
    z = 2 * (a + b) - c;
}
```

```
fkt: pushl ebp
      movl esp,ebp
      subl#16,esp
      movl 8(epb),eax
      movl eax,-4(epb)
      movl 12(epb),eax
      movl eax,-8(epb)
      movl 12(epb),edx
      movl 8(epb),eax
      subl edx, eax
      ....
      ret
```

4 * sizeof(int) = 16



Dynamische lokale Variablen

- Im vorigen Beispiel: Gesamte Größe der lokalen Daten ist zur Compile-Zeit bekannt
 - Allokation durch Addition eines festen Betrages zum Stackpointer
 - Seit C99 Standard möglich: Lokale Felder *variabler* Größe
- Beispiel: variable Feldgröße**

Feld variabler Größe

```
fkt(int x, int y)
{
    int a[x];
    int i;

    for(i = 0; i < x; i++)
        a[i] = ....;
    ...
}
```

Vor C99: Funktion `alloca()`

```
fkt(int x, int y)
{
    int *a;
    int i;
    a = (int *)alloca(x * sizeof(*a));
    for(i = 0; i < x; i++)
        a[i] = ....;
    ...
}
```

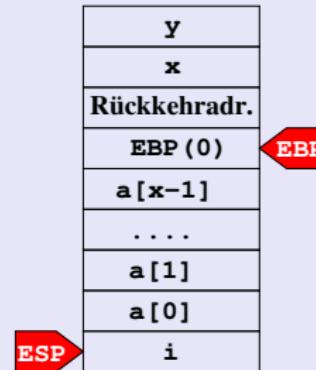
Dynamische lokale Variablen

- Feldgröße kann von Aufruf zu Aufruf variieren
- Adressabstand zwischen Framepointer und Variablen (hier z.B. i) ist *nicht* konstant
- Aufwändige Adressberechnung bei jedem Zugriff auf lokale Variablen
→ teuer

Beispiel: variable Feldgröße

```
fkt(int x, int y)
{
    int a[x];
    int i;

    for(i = 0; i < x; i++)
        a[i] = ....;
    ...
}
```



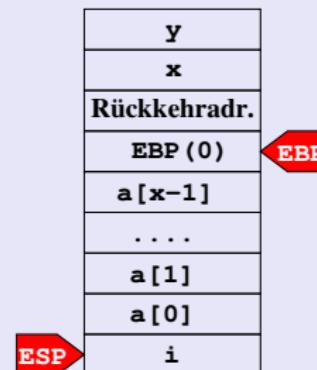
Pufferüberlauf-Attacke

- **Vorsicht:** $a[x+1]$ ist die Rückkehradresse
- Mit $a[x+1] = \text{Zieladresse}$ kann die Rückkehradresse verändert werden
- Statt zum Aufrufer zurückzukehren, kann ein beliebiges Stück (womöglich als Daten eingeschleusten) Schadcodes angesprungen werden

Beispiel: Gefahr von Puffer-Überlauf

```
fkt(int x, int y)
{
    int a[x];
    int i;

    for(i = 0; i < x; i++)
        a[i] = ....;
    ...
}
```

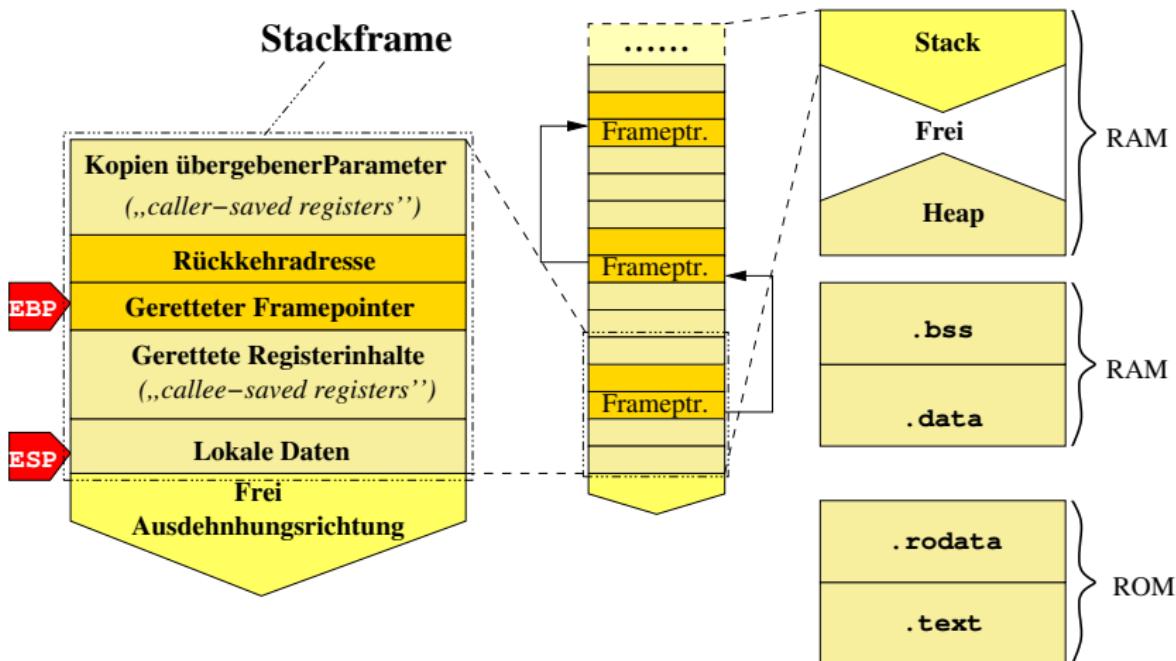


Frame pointer

- Lokale Variablen werden über Framepointer (EBP) referenziert:
 - ▶ Positives Displacement → Parameterkopie
 - ▶ Negatives Displacement → Lokale Variable
 - ▶ Displacement = 0 → Framepointer des Aufrufers
 - ▶ Displacement = 4 → Rückkehradresse
- Eigentlich könnten alle diese Objekte auch über den Stackpointer (ESP) referenziert werden → Prinzipiell wäre der Framepointer entbehrlich
- Einige Compiler / Architekturen können auch ohne Framepointer arbeiten Z.B. GNU Compiler (gcc) mit `-fomit-frame-pointer`
- Liefert geringfügig effizienteren Code
- Nachteil: Aufrufhierarchie (*Backtrace*) ist (z.B. bei der Fehlersuche) nicht mehr rekonstruierbar

Zusammenfassung – Prozeduren

- Allgemeine Struktur des *Stackframe* einer Prozedur



setjmp() / longjmp()

- **setjmp() / longjmp()** – Funktionen der Standard C-Bibliothek
 - ▶ **setjmp()**: Speichert („rettet“) den momentanen Prozessorzustand
 - ▶ **longjmp()**: Stellt einen zuvor gespeicherten Zustand wieder her
- Implementierung **immer** in Assembler, aus C heraus aufrufbar
- Verwendung typischerweise als „nicht-lokales goto“

Prototypen

```
#include <setjmp.h>
/* definiert u.a. jmp_buf */

int setjmp(jmp_buf zustand);

void longjmp(jmp_buf zustand, int retwert);
```

setjmp() – Arbeitsweise

- jmp_buf: Array zum Abspeichern des Prozessorzustandes
- setjmp() kopiert ..

- ▶ callee saved registers (einschl. Stackpointer)
- ▶ Rückkehradresse

Beispiel-Implementierung für Intel i386:

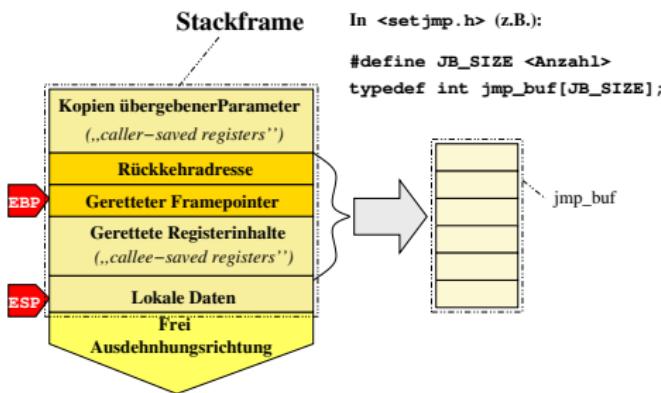
```
setjmp: movl 4(esp),eax ; jmp_buf Adresse in EAX
        movl ebx,0(eax) ; callee-saved Register..
        movl esi,4(eax) ; (hier: ebx, esi,edi, ebp)
        movl edi,8(eax) ; .. in jmp_buf speichern
        movl ebp,12(eax)
        movl 0(esp),ebx ; Rückkehradresse in EBX
        movl ebx,20(eax) ; in jmp_buf speichern
        movl 4(esp),ebx ; Stackpointer in ebx
        movl ebx,16(eax) ; in jmp_buf speichern
        movl 0(eax),ebx ; ebx wiederherstellen
        xorl eax,eax     ; Returnwert (eax) = 0
        ret
```

- .. in die bereitgestellte jmp_buf Datenstruktur
 - Liefert Null als Returnwert
- Macht einen „Schnappschuss des Prozessorzustandes“

setjmp() – Arbeitsweise

- jmp_buf: Array zum Abspeichern des Prozessorzustandes
- setjmp() kopiert ..

- ▶ *callee saved registers* (einschl. Stackpointer)
- ▶ Rückkehradresse



- .. in die bereitgestellte jmp_buf Datenstruktur
- Liefert Null als Returnwert
- Macht einen „Schnappschuss des Prozessorzustandes“

longjmp() – Arbeitsweise

- Lädt *callee saved registers* (einschl. Stackpointer) aus der übergebenen `jmp_buf` Datenstruktur
- Kehrt zur Rückkehradresse aus der `jmp_buf` Datenstruktur zurück
- Liefert vom Aufrufer übergebenen `retwert` als Returnwert
- Stellt den Prozessorzustand exakt wieder so her, wie er beim Aufruf von `setjmp()` gespeichert wurde
- Das bedeutet insbesondere auch, dass `longjmp()` zu der zugehörigen Aufrufposition von `setjmp()` zurückkehrt!
- Einzige Möglichkeit der Unterscheidung für den Aufrufer von `setjmp()`: der Returnwert
- Achtung: niemals 0 als zweites Argument an `longjmp()` übergeben

Typische Verwendung: „non-local goto“

Ohne setjmp()/longjmp()

„Durchreichen“ der Fehlerbedingung über alle Aufrufebenen:

```
if (FEHLER == funktion1(...)) {
    ... Fehlerabbruch ...
else
    .....

funktion1(...)
{
    if (FEHLER == funktion2(...))
        return (FEHLER);
    else
        .....
}
.....
funktionN(...)
{
    if (Fehlerbedingung)
        return (FEHLER);
}
```

Mit setjmp()/longjmp()

Direkte Rückkehr zur obersten Aufrufebene ohne Zwischenebenen zu involvieren

```
static jmp_buf buf;

if ((fcode = setjmp(buf)))
    ... Fehlerabbruch ...
else
    .....
funktion1(...)
{
    funktion2(...);
    .....
}
.....
funktionN(...)
{
    if (Fehlerbedingung)
        longjmp(buf, fcode);
}
```

Einschränkungen

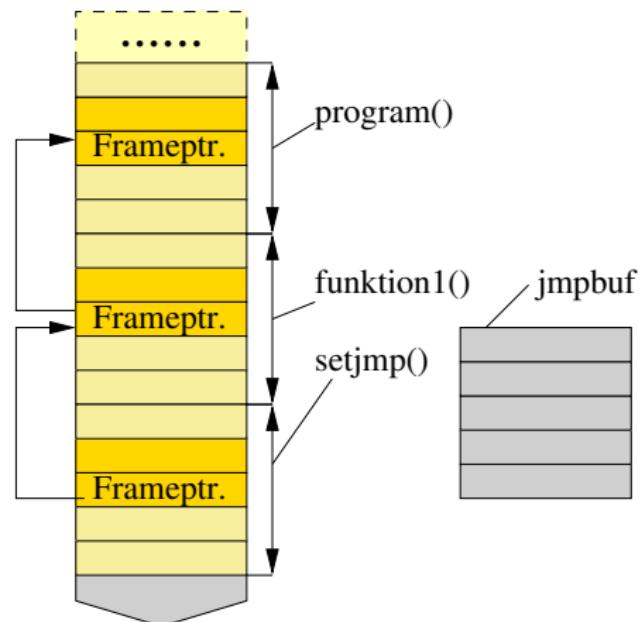
- Achtung: immer nur „von innen nach außen“ springen!

Beispiel: Unzulässig:

```
static jmp_buf buf;

programm()
{
    funktion1 ( .... );
    longjmp (buf);
}

funktion1 ( ... )
{
    if (setjmp (buf))
        ...
    else
        ...
}
```



Einschränkungen

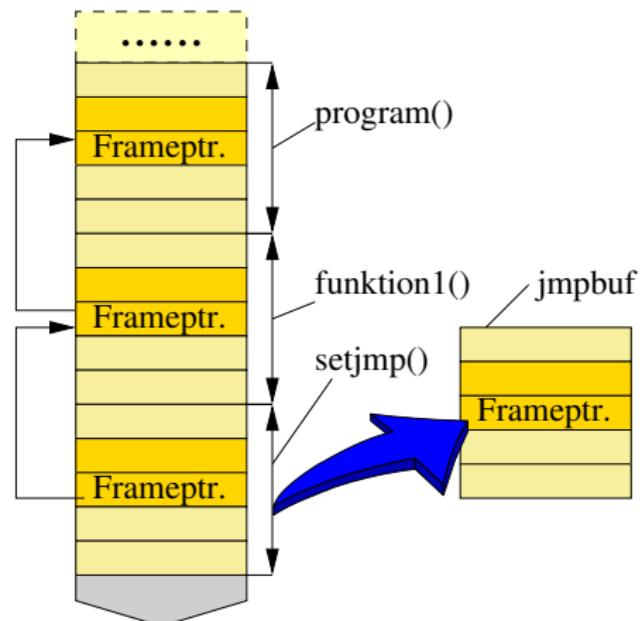
- Achtung: immer nur „von innen nach außen“ springen!

Beispiel: Unzulässig:

```
static jmp_buf buf;

programm()
{
    funktion1 ( .... );
    longjmp (buf);
}

funktion1 ( ... )
{
    if (setjmp (buf))
        ...
    else
        ...
}
```



Einschränkungen

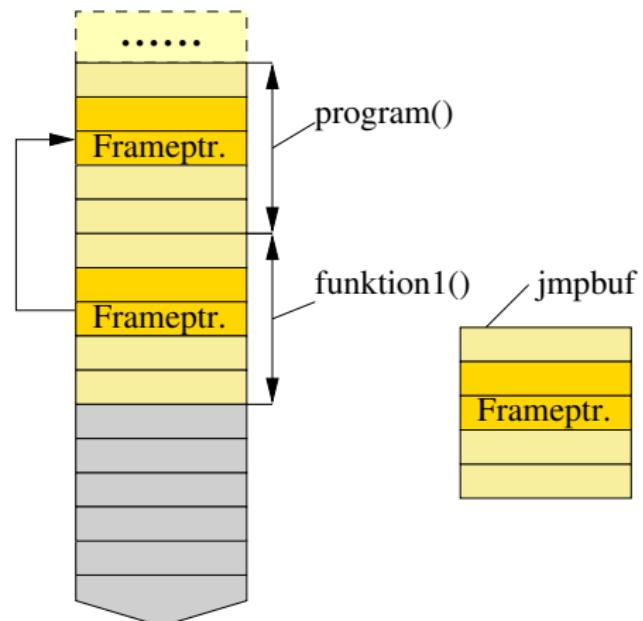
- Achtung: immer nur „von innen nach außen“ springen!

Beispiel: Unzulässig:

```
static jmp_buf buf;

programm()
{
    funktion1 ( .... );
    longjmp (buf);
}

funktion1 ( ... )
{
    if (setjmp (buf))
        ...
    else
        ...
}
```



Einschränkungen

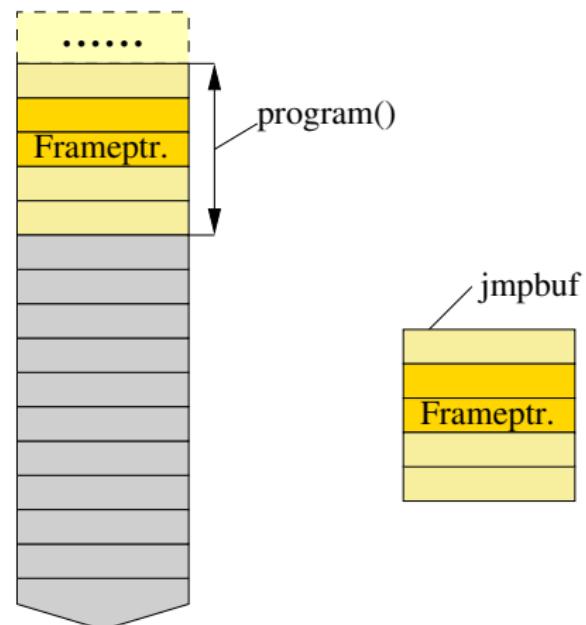
- Achtung: immer nur „von innen nach außen“ springen!

Beispiel: Unzulässig:

```
static jmp_buf buf;

programm()
{
    funktion1 ( .... );
    longjmp (buf);
}

funktion1 ( ... )
{
    if (setjmp (buf))
        ...
    else
        ...
}
```



Einschränkungen

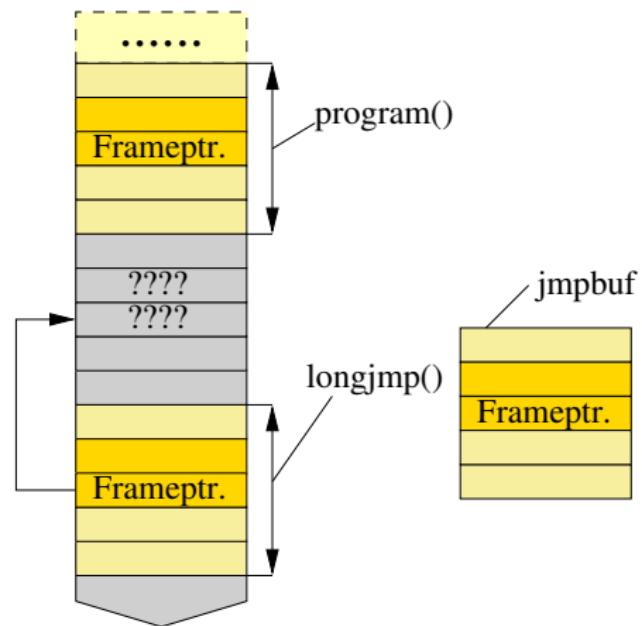
- Achtung: immer nur „von innen nach außen“ springen!

Beispiel: Unzulässig:

```
static jmp_buf buf;

programm()
{
    funktion1 ( .... );
    longjmp (buf);
}

funktion1 ( ... )
{
    if (setjmp (buf))
        ...
    else
        ...
}
```



Multitasking mit setjmp() / longjmp()



- „Missbräuchliche“ (?) Nutzung: Speichern / Wiederherstellen eines Prozesskontexts

Prozesswechsel mit setjmp() / longjmp()

```
struct tcb { /* Task Control Block */
    ...           /* Beschreibt Zustand eines Prozesses */
    jmp_buf zustand;
    ...
};

void task_wechsel(struct tcb *von, struct tcb *zu) {
    if (!setjmp(von->zustand)) {
        longjmp(zu->zustand);
        assert(0);
    }
}
```

- Vorstellung: N „virtuelle Prozessoren“, davon jeweils einer aktiv
- Umschalten mit task_wechsel()

Kooperatives Multitasking

- Laufender Prozess gibt Prozessor explizit frei

Kooperatives Multitasking (Prinzip)

```
struct tcb proztab[ANZ_PROZ]; /* Prozesstabelle */  
int lfd_proz; /* aktuell laufender Prozess */  
  
....  
if(nichts_zu_tun) {  
    task_wechsel(&proztab[lfd_proz],  
                 &proztab[(lfd_proz+1) % ANZ_PROZ]);  
}  
....
```

- + Einfach zu realisieren
- Keine Garantie, dass/wann ein Prozess „drankommt“
- Prozesse müssen „kooperativ“ sein

Preemptives Multitasking

- Taskwechsel in festen Zeitabständen (z.B. durch Timer-Interrupt)
- Laufender Prozess wird unterbrochen wenn seine „Zeitscheibe“ abgelaufen ist

Preemptives Multitasking (Prinzip)

```
struct tcb proztab[ANZ_PROZ]; /* Prozesstabelle */
int lfd_proz; /* aktuell laufender Prozess */
/* Timer 0 interrupt service routine: */
ISR(TIMER0_vect)
{
    /* wird (z.B.) alle 10ms aufgerufen */
    alt = lfd_proz;
    neu = lfd_proz = (lfd_proz+1) % ANZ_PROZ;
    task_wechselt(&proztab[alt],
                  &proztab[neu]);
}
```

- Prozesse arbeiten „quasi-parallel“
- Besondere Vorkehrungen nötig (Kritische Abschnitte, Reentranz)

Memory-mapped I/O

- E/A-Bausteine sind wie Speicher an den Bus angeschlossen
 - ▶ E/A-Register erscheinen wie Variablen im Speicher (unter spezieller, vorab bekannter Adresse)
 - ▶ Können ebenso wie solche –z.B. von C-Programmen– gelesen/manipuliert werden

Beispiel AVR Mikrocontroller: LEDs blinken lassen

```
#include <avr/io.h>
void blink(int led, int times) /* led = 1, 2, oder 1+2 */
{
    DDRD = (1<<6)|(1<<5); /* PortD6 .. PortD5 -> Ausgaenge */
    while(times--) { /* blinke <times> mal */
        PORTD = (led & 3) << 5; /* LEDs an */
        _delay_ms(250); /* warten ... */
        PORTD = 0; /* LEDs aus */
        _delay_ms(250) /* warten ... */
    }
}
```

Memory-mapped I/O Register ansprechen (1)



- Frage: Wie spricht man in C überhaupt ein E/A-Register „unter spezieller Adresse“ an ?

Beispiel: (Vereinfachter) Auszug aus <avr/io.h>

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *) (mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define PIND      _SFR_IO8(0x10)
#define DDRD      _SFR_IO8(0x11)
#define PORTD    _SFR_IO8(0x12)
```

→ (z.B.) DDRD wird durch den C-Preprozessor substituiert:

- ▶ DDRD → _SFR_IO8(0x11)
- ▶ _SFR_IO8(0x11) → _MMIO_BYTE((0x11)+0x20)
- ▶ _MMIO_BYTE((0x11)+0x20) → (*(volatile uint8_t*)((0x11)+0x20))

→ (*(volatile uint8_t*)0x31)

- Allgemein: (*(volatile <Typ> *)<Adresse>)

Memory-mapped I/O Register ansprechen (2)



- `(*(volatile <Typ> *)<Adresse>)`
- Allgemein: *Typecast*, bzw. *Cast*: „`(<Typ>)<Objekt>`“
 - ▶ Erzwingt, dass `<Objekt>` als vom Typ `<Typ>` aufgefasst wird
 - ▶ Beispiel: `(float)10` → Gleitkommazahl 10,0000
- Hier speziell: Zahl `<Adresse>` wird als Zeiger auf Objekte vom Typ `volatile <Typ>` aufgefasst, der ...
 - ... durch den vorangestellten „`*`“ unmittelbar de-referenziert wird
 - `<Typ>` dient zur Bestimmung der Zugriffsbreite:
 - ▶ `char`: 8 Bit (Byte)
 - ▶ `short`: i.d.R. 16 Bit („Wort“)
 - ▶ `int`: i.d.R. 16 oder 32 Bit
 - ▶ `long`: i.d.R. 32 Bit
 - ▶ `long long`: i.d.R. 64 Bit

Memory-mapped I/O Register ansprechen (2)



- `(*(volatile <Typ> *)<Adresse>)`
- Allgemein: *Typecast*, bzw. *Cast*: „`(<Typ>)<Objekt>`“
 - ▶ Erzwingt, dass `<Objekt>` als vom Typ `<Typ>` aufgefasst wird
 - ▶ Beispiel: `(float)10` → Gleitkommazahl 10,0000
- Hier speziell: Zahl `<Adresse>` wird als Zeiger auf Objekte vom Typ `volatile <Typ>` aufgefasst, der ...
- ... durch den vorangestellten „`*`“ unmittelbar de-referenziert wird
- `<Typ>` dient zur Bestimmung der Zugriffsbreite:
 - ▶ `char`: 8 Bit (Byte)
 - ▶ `short`: i.d.R. 16 Bit („Wort“)
 - ▶ `int`: i.d.R. 16 oder 32 Bit
 - ▶ `long`: i.d.R. 32 Bit
 - ▶ `long long`: i.d.R. 64 Bit

Memory-mapped I/O Register ansprechen (3)



- C-Standardtypen short, int, long, long long haben keine standardisierte Größe
- C-Programmierumgebungen definieren i.d.R. die passenden Typen mit `typedef`
- Z.B. AVR-Libc `<stdint.h>`:
 - ▶ `int8_t`: 8 Bit vorzeichenbehaftet
 - ▶ `uint8_t`: 8 Bit vorzeichenlos
 - ▶ `int16_t`: 16 Bit vorzeichenbehaftet
 - ▶ `uint16_t`: 16 Bit vorzeichenlos
 - ▶ `int32_t`: 32 Bit vorzeichenbehaftet
 - ▶ `uint32_t`: 32 Bit vorzeichenlos
 - ▶ `int64_t`: 64 Bit vorzeichenbehaftet
 - ▶ `uint64_t`: 64 Bit vorzeichenlos

Memory-mapped I/O Register ansprechen (4)



- Alternative („elegantere“?) Methode:

Register in Strukturen zusammenfassen

```
typedef struct {
    volatile uint8_t pin; /* Pegelregister      */
    volatile uint8_t ddr; /* Data direction reg. */
    volatile uint8_t port; /* Ausgaberegister   */
} atmega_port;

#define AVR_PORTA ((atmega_port*)0x39)
#define AVR_PORTB ((atmega_port*)0x36)
#define AVR_PORTC ((atmega_port*)0x33)
#define AVR_PORTD ((atmega_port*)0x30)
```

- Abbilden einer (logisch zusammengehörigen) Gruppe von Registern durch eine Datenstruktur
- Gleichartige E/A-Geräte (hier: Ports A ... D) werden durch die gleiche Datenstruktur mit jeweils anderer Adresse beschrieben

Memory-mapped I/O Register ansprechen (5)



- Damit: Blink-Routine (s.o.):

Register in Strukturen zusammenfassen

```
void blink(int led, int times)
{
    AVR_PORTE->ddr = (1<<6)|(1<<5);
    while(times--) {
        AVR_PORTE->port = (led & 3) << 5;
        _delay_ms(250);
        AVR_PORTE->port = 0;
        _delay_ms(250);
    }
}
```

- Vorsicht, wenn verschiedene Datentypen in der Struktur gemischt werden müssen
- Ggf. unbenutzte *padding*-Bytes einfügen, um Struktur an Hardware-gegebenheiten anzupassen

Fallstricke: Objektgröße

p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

→ (Unerwartete) Vorzeichenerweiterung!

Fallstricke: Objektgröße

p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

→ (Unerwartete) Vorzeichenerweiterung!

A5	00	00	00

Fallstricke: Objektgröße

p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

→ (Unerwartete) Vorzeichenerweiterung!

A5	00	00	00
5A	00	00	00

Fallstricke: Objektgröße

p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

→ (Unerwartete) Vorzeichenerweiterung!

A5	00	00	00
5A	00	00	00
A5	FF	FF	FF

Fallstricke: Objektgröße

p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

→ (Unerwartete) Vorzeichenerweiterung!

A5	00	00	00
5A	00	00	00
A5	FF	FF	FF

Fallstricke: Codeoptimierung (1)

- Codeoptimierung durch den Compiler:
- (An sich vernünftige) Annahme: „Variablenwerte ändern sich nur in Folge von Wertzuweisungen“

Auszug aus blink-Routine (s.o.)

```
4: while(times--) {  
5:     PORTD = (led & 3) << 5;  
6:     _delay_ms(250);  
7:     PORTD = 0;  
8:     _delay_ms(250);  
9: }
```

- Konsequenzen:
 - ▶ Zeile 5: Wertzuweisung an PORTD
 - ▶ Zeile 6: Keine Referenz auf Wert von PORTD
 - ▶ Zeile 7: Erneute Wertzuweisung an PORTD
- Compiler: „Zeile 5 ist überflüssig“ → wird „wegoptimiert“!

Fallstricke: Codeoptimierung (2)

- Codeoptimierung durch den Compiler:
- (An sich vernünftige) Annahme: „Variablenwerte ändern sich nur in Folge von Wertzuweisungen“

Auszug aus blink-Routine (s.o.)

```
4: while(times--) {  
5:  
6:     _delay_ms(250);  
7:     PORTD = 0;  
8:     _delay_ms(250);  
9: }
```

- Nächster Schritt
 - ▶ Zeile 7: Wertzuweisung an PORTD
 - ▶ Zeilen 4,5,6,8,9: Keine Referenz auf Wert von PORTD innerhalb der Schleife
- Compiler: „Zeile 7 kann aus der Schleife herausgezogen werden“!

Fallstricke: Codeoptimierung (2)

- Codeoptimierung durch den Compiler:
- (An sich vernünftige) Annahme: „Variablenwerte ändern sich nur in Folge von Wertzuweisungen“

Auszug aus blink-Routine (s.o.)

```
7: PORTD = 0;  
4: while(times--) {  
5:  
6:     _delay_ms(250);  
8:     _delay_ms(250);  
9: }
```

- Ergebnis
 - ▶ LEDs werden, bzw. bleiben ausgeschaltet
 - ▶ Es wird $times \cdot 500ms$ gewartet
- Wohl kaum das erwartete Ergebnis...

Fallstricke: Codeoptimierung (3)



- Anderes Beispiel: Memory-mapped Lesezugriff

C-Code

```
struct eageraet {
    uint8_t data;
    uint8_t __unused;
    uint8_t status;
};

struct eageraet *r = ....;

while((r->status & (1<<2)) != 0)
    ;
```

Assembler

```
.....
movl    r , eax
movzbl 2(eax) , eax
andl    #4, eax
.L3: testl  eax , eax
jne   .L3
.....
```

Lesen von r->status wurde aus der Schleife herausgenommen

- Auch hier: keine ersichtliche Änderung von r->status innerhalb der Schleife
- Änderung des Wertes erfolgt durch E/A-Gerät und wird hier nicht berücksichtigt

Fallstricke: Codeoptimierung (3)

- Anderes Beispiel: Memory-mapped Lesezugriff

C-Code

```
struct eageraet {
    uint8_t data;
    uint8_t __unused;
    uint8_t status;
};

struct eageraet *r = ....;

while((r->status & (1<<2)) != 0)
    ;
```

Assembler

Endlosschleife!!

```
.....
movl r , eax
movzbl 2(eax) , eax
andl #4,eax
.L3: testl eax , eax
jne .L3
.....
```

Lesen von `r->status` wurde aus der Schleife herausgenommen

- Auch hier: keine ersichtliche Änderung von `r->status` innerhalb der Schleife
- Änderung des Wertes erfolgt durch E/A-Gerät und wird hier nicht berücksichtigt

Fallstricke: Codeoptimierung (4)

- Weiteres Beispiel: Parallelle Programmausführung
- Hier: Interrupt Service Routine und Hauptprogramm

Interrupt Service

```
static int intcount = 0;  
  
ISR(TIMER0_vector)  
{  
    ++intcount;  
}
```

Hauptprogramm

```
main () {  
    .....  
    unsigned int old_count;  
    .....  
    old_count = intcount;  
    while (old_count == intcount)  
        ;  
    .....  
}
```

- Änderung von `intcount` erfolgt durch nebenläufigen Prozess (hier: Interrupt Service) und wird u.U. nicht berücksichtigt
- Wieder eine Endlossschleife!

Fallstricke: Codeoptimierung (5)

- Noch ein Beispiel: Verzögerungsschleife

Verzögerungsschleife ..

```
int i;  
for(i = 0; i < 10000; i++)  
    ;
```

...ist funktional gleich mit:

```
int i = 10000;
```

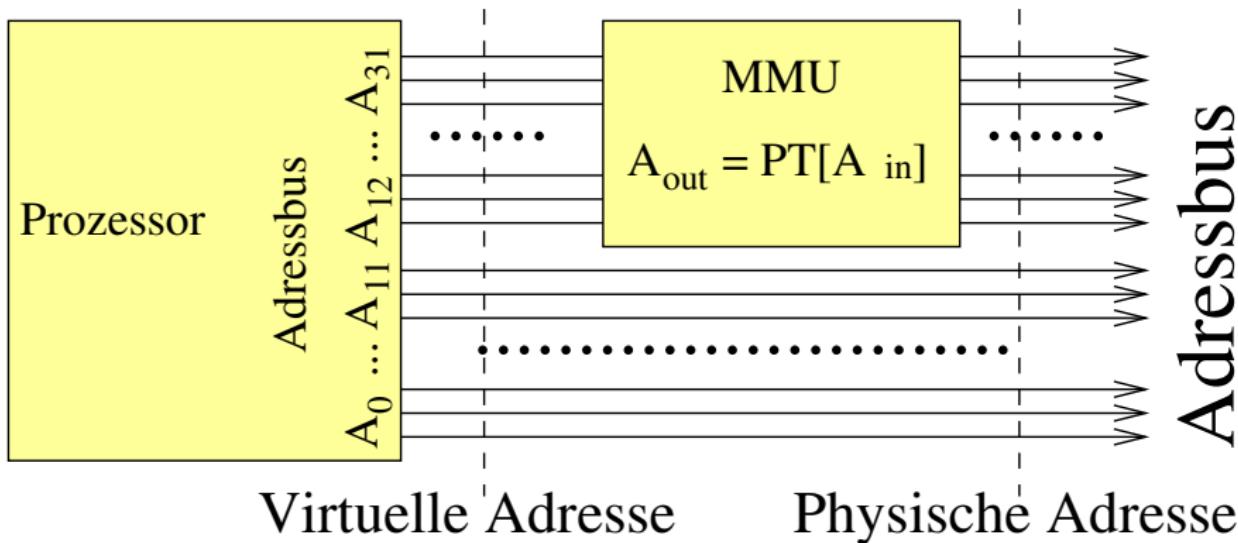
- (Hier) gewünschter Effekt der Verzögerung tritt nicht ein

Fallstricke: Codeoptimierung (6)

- Abhilfe in allen gezeigten Fällen: Register-Objekte, gemeinsame Daten nebenläufiger Programme oder Zählvariable von leeren Verzögerungsschleifen `volatile` („flüchtig“) erklären
- Zwingt den Compiler, keine Annahmen über den Inhalt der bezeichneten Variablen zu machen
 - Kein „Wegoptimieren“ von Wertzuweisungen oder Lesezugriffen
 - Zeiger auf Memory-mapped I/O Register immer `volatile` erklären (siehe auch obige Beispiele zu AVR)

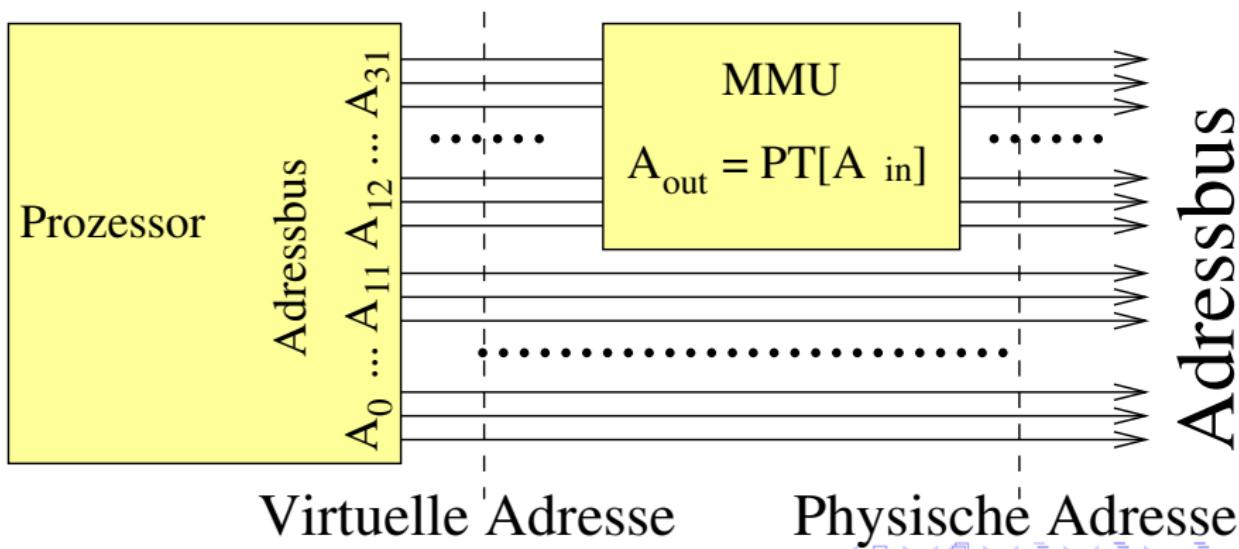
Fallstricke: Virtuelle Adressierung (1)

- Rechensysteme mit Speicherverwaltungseinheit (*Memory Management Unit – MMU*)
- Früher externe Hardware, heute ggf. im Prozessor integriert
- Verwendung: Speichervirtualisierung



Fallstricke: Virtuelle Adressierung (2)

- *Virtuelle Adressen*: Adressen, die vom Adresswerk des Prozessors generiert werden
- *Physische Adressen*: Adressen, die auf dem externen Adressbus anliegen
- MMU ordnet virtuellen Adressen physische Adressen zu



Fallstricke: Virtuelle Adressierung (3)

- Funktion der MMU (n -Bit Adressen):

- ▶ Obere $n - m$ Adressbits der virtuellen Adresse $A_m^{virt} \dots A_{n-1}^{virt}$ werden durch eine frei programmierbare Abbildung $PT[x]$ auf physische Adressen $A_m^{phys} \dots A_{n-1}^{phys}$ abgebildet:

$$A_m^{phys} \dots A_{n-1}^{phys} = PT[A_m^{virt} \dots A_{n-1}^{virt}]$$

- ▶ Untere m Adressbits der virtuellen Adresse $A_0^{virt} \dots A_{m-1}^{virt}$ werden nicht verändert:

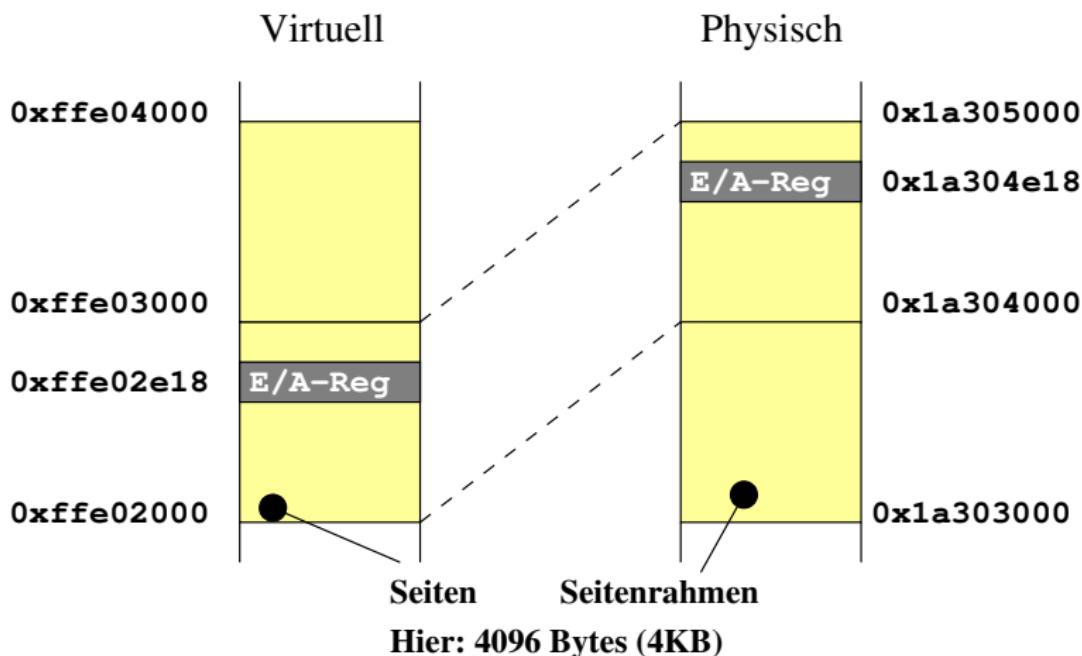
$$A_0^{phys} \dots A_{m-1}^{phys} = A_0^{virt} \dots A_{m-1}^{virt}$$

→ Speicher ist in „Seitenrahmen“ (*Page Frames*) fester Größe aufgeteilt, denen „Seiten“ (*Pages*) zugeordnet werden

- Gängige Seitengröße: $m = 12 \rightarrow 4096|_{10}$, bzw. $1000|_{16}$ Bytes

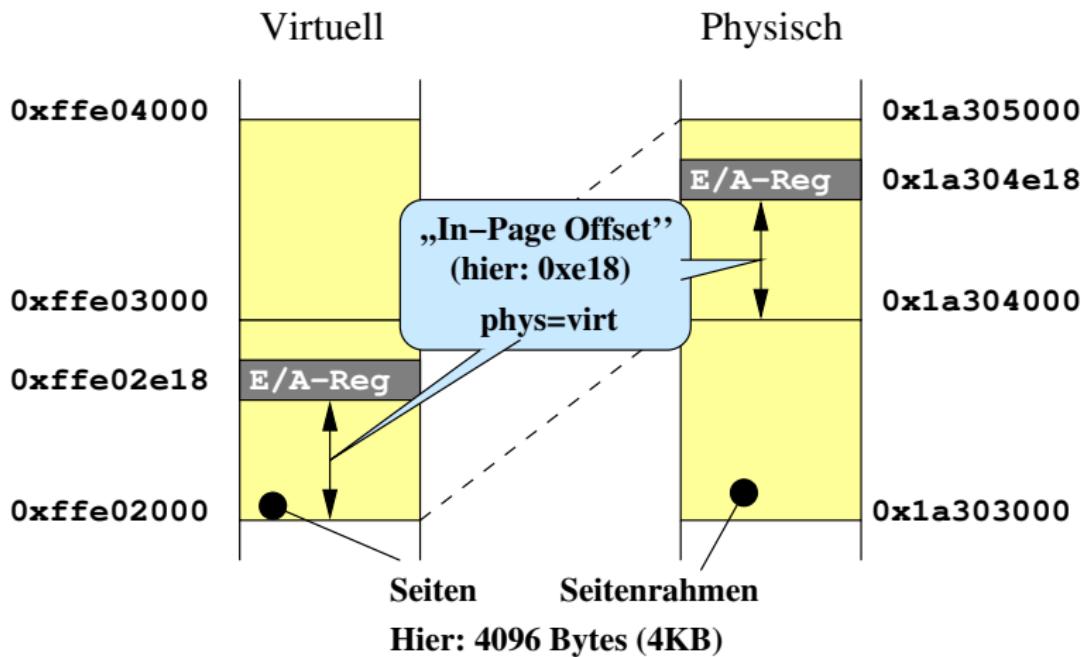
Fallstricke: Virtuelle Adressierung (4)

- Zugriff auf memory-mapped E/A an **physikalischer** Adresse muss über die entsprechende **virtuelle** Adresse erfolgen



Fallstricke: Virtuelle Adressierung (4)

- Zugriff auf memory-mapped E/A an **physikalischer** Adresse muss über die entsprechende **virtuelle** Adresse erfolgen



Fallstricke: Virtuelle Adressierung (5)



- Kenntnis bzw. Beeinflussung der Abbildung virtuelle → physische Adresse durch entsprechende Programmierung der MMU, i.d.R. mit Hilfe von Betriebssystemfunktionen

Beispiel: Zugriff aus Linux-Anwendung

```
struct eageraet *r;
int fd;
if ((fd = open("/dev/mem", O_RDWR)) < 0) {
    perror("open(/dev/mem)");
    exit(1);
}
r = (struct eageraet*)mmap((caddr_t)0, Size,
    PROT_READ|PROT_WRITE, MAP_SHARED, fd,
    (off_t)PHYS_ADDRESS);
close(fd);
if ((long)r == -1) {
    perror("mmap()");
    exit(1);
}
r->data = ....
```

Fallstricke: Virtuelle Adressierung (6)

- Kenntnis bzw. Beeinflussung der Abbildung virtuelle → physische Adresse durch entsprechende Programmierung der MMU, i.d.R. mit Hilfe von Betriebssystemfunktionen
- Cache muss für memory-mapped Adressregionen deaktiviert sein!

Beispiel: Zugriff aus Linux-Gerätetreiber

```
struct eageraet *r;  
  
r = (struct eageraet*)ioremap_nocache(  
    PHYS_ADDRESS, sizeof(*r));  
r->data = ....
```

Port-mapped I/O

- Port-mapped I/O:
 - ▶ E/A-Register liegen in einem eigenen Adressraum
 - ▶ Zugriffe erfordern spezielle Maschinenbefile (in/out)
- Nicht direkt in C/C++ implementierbar (i.d.R. stehen aber geeignete Macros zur Verfügung)

Beispiel: Port-mapped I/O

```
#define PP_DAT 0x378
#define PP_STAT (PP_DAT+2)
#define STAT_ACK 0x02
#define STAT_STROBE 0x01
void Print(char X)
{
    while((inb(PP_STAT) & STAT_ACK) != 0) /* warten... */ ;
    outb(X, PP_DAT);
    outb(inb(PP_STAT) | STAT_STROBE, PP_STAT);
    while((inb(PP_STAT) & STAT_ACK) == 0) /* warten... */ ;
    outb(inb(PP_STAT) & ~STAT_STROBE, PP_STAT);
}
```

Vergleich: Port-mapped / Memory-mapped I/O

- Port-mapped I/O ist i.W. nur bei Prozessoren der Intel ix86-Familie vorhanden
 - Nicht Portabel⁴
- Keine Adressübersetzung durch MMU → Keine Unterscheidung zwischen physikalischen und virtuellen Adressen
- Port-mapped I/O durchläuft **nicht** den Cache
- Port-Adressraum i.d.R. kleiner als Speicheradressraum
 - ▶ z.B. 16 Bit gegenüber 32 Bit
 - Geräte mit großem Adressraum (z.B. Grafikkarten) verwenden i.d.R. auch bei Intel-Prozessoren Memory-mapped I/O

⁴Linux bietet für nicht-Intel-Maschinen in/out-Funktionen an, die aber durch Memory-mapped I/O implementiert sind

Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

4. Echtzeitbetriebssysteme



<https://school-time.co/>

Inhalt

4. Echtzeitbetriebssysteme

4.1 Einführung

4.2 Architektur von Echtzeitbetriebssystemen

4.3 Der Markt für Echtzeitbetriebssysteme

4.4 Beispiele von Echtzeitbetriebssystemen

Einführung

Wiederholung Betriebssysteme

Definition (Betriebssystem)

Ein Betriebssystem ist ein Programm, das alle Betriebsmittel eines Rechensystems verwaltet und ihre Zuteilung kontrolliert und den Nutzern des Rechensystems eine virtuelle Maschine offeriert, die einfacher zu verstehen und zu programmieren ist als die unterlagerte Hardware.

- Betriebssystem als „Betriebsmittelverwalter“
- Betriebssystem als „virtuelle Maschine“
 - ▶ Hardware-Unabhängigkeit
 - ▶ Adäquate Abstraktionen
 - ▶ Langlebigkeit der Programmierschnittstelle

Aufgabenbereiche eines Betriebssystems

Wichtige Aufgabenbereiche

- Verwaltung der Schnittstelle zur unterliegenden Hardware
- Prozessverwaltung
- Speicherverwaltung
- Interprozesskommunikation
- Ein/Ausgabe
- Dateisysteme

Echtzeitbetriebssysteme ...

- ... bilden unterlagerte Software-Schicht für komplexe Echtzeit- / Embedded Control- Anwendungen
- ... haben prinzipiell die gleichen Aufgabenbereiche wie übliche Betriebssysteme

Aber: es kommen Randbedingungen hinzu ...

Aufgabenbereiche (2)

... nämlich:

- bessere Unterstützung für die Vorhersagbarkeit des Systemverhaltens
(→ Echtzeitcharakter) (s.u.)
- Konfigurierbarkeit / Skalierbarkeit
 - ▶ Auskommen mit minimalen Ressourcen
(Kostengründe in Massenprodukten)
 - ▶ nur die wirklich benötigten Komponenten sollten Bestandteil des aktuell verwendeten Betriebssystems sein
- Stark variierende Zielumgebungen
 - ▶ unterschiedlichste I/O-Konfigurationen
 - ▶ Boot aus ROM oder Betrieb aus ROM bei diskless targets (häufig!)
- Entwicklungsprozess
 - ▶ Häufig Cross-Entwicklungsumgebung notwendig (s. Kap. 3)
 - ▶ Debugging schwierig (auch wg. Verfälschung der Echtzeiteigenschaften)

Anforderungen bzgl. Vorhersagbarkeit

- Durchsetzen von Planungsentscheidungen (Scheduling):
 - ▶ Prioritäten-basierte Scheduler als Standardfall
 - ▶ Zuordnung von anwendungsspezifischen Prioritäten zu Prozessen
 - ▶ Verwendung spezifischer Scheduler
 - ▶ Formulierung und Überwachung von Zeitbedingungen
- Unterbrechungsbehandlung:
 - ▶ Auftreten externer Ereignisse muss gemäß der Dringlichkeit bearbeitet bzw. an das zugehörige Anwendungsprogramm weitergeleitet werden.
- Zeitdienste:
 - ▶ Operationen zum Realisieren von Verzögerungen, zeitgenauen Aktionen und das zeitgesteuerte Aktivieren und Deaktivieren.
- Unterbrechbarkeit des Betriebssystemkerns:
 - ▶ Die Bearbeitung eines Systemdienstes für einen Prozess darf die Bearbeitung einer zeitkritischen Aktion nicht behindern.
 - ▶ Problem: Unterbrechung der Bearbeitung eines system calls an beliebiger Stelle kann zu Inkonsistenzen führen

Anforderungen bzgl. Vorhersagbarkeit (2)

- Kontrolle über die Speicherverwaltung:
Virtuelle Speicherverwaltung muss ausgesetzt oder vorhersagbar sein.
 - ▶ wie lange braucht ein `malloc()`?
- Vorhersagbarkeit des Dateizugriffsverhaltens.
 - ▶ wie lange dauert das Anlegen einer Datei?
- Vorhersagbarkeit des Verhaltens der Kommunikationskanäle.
 - ▶ wann und für wie lange ist der Bus frei?

→ Designentscheidungen in allen Ebenen

Anforderungen bzgl. Vorhersagbarkeit (2)

- Kontrolle über die Speicherverwaltung:
Virtuelle Speicherverwaltung muss ausgesetzt oder vorhersagbar sein.
 - ▶ wie lange braucht ein `malloc()`?
- Vorhersagbarkeit des Dateizugriffsverhaltens.
 - ▶ wie lange dauert das Anlegen einer Datei?
- Vorhersagbarkeit des Verhaltens der Kommunikationskanäle.
 - ▶ wann und für wie lange ist der Bus frei?

→ Designentscheidungen in allen Ebenen

Klassifizierung

Klassifizierung von BSen nach dem Funktionsumfang:

- Elementare Runtime-Systeme

- ▶ einfaches Multitasking (eher Multithreading)
- ▶ Betriebsmittel-optimiert
- ▶ einfache (i.d.R. statische) Datenstrukturen (Bibliotheks-Charakter)
- ▶ Beispiele:
 - ★ Contiki (www.sics.se/contiki)
 - ★ μC/OS-II (Micrium)
 - ★ FreeRTOS (www.freertos.org)
 - ★ OSEK-OS

- Multitasking-Kerne

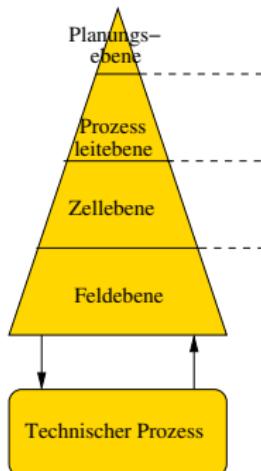
- ▶ Ein-Adressraum-Verwaltung (keine MMU-Nutzung)
- ▶ dynamische Datenstrukturen (Tasks, Speichersegmente, ...)
- ▶ moderater Betriebsmittelverbrauch
- ▶ Steuerbarkeit des Echtzeitverhaltens
- ▶ Beispiele:
 - ★ VxWorks (<https://www.windriver.com>)
 - ★ PXROS (HighTec, Saarbrücken)

Klassifizierung (2)

- Vollständige Betriebssysteme incl. MMU-Unterstützung
 - ▶ Bereitstellung/Nutzung mehrerer Adressräume
 - ▶ Umfangreiches Dienstangebot
 - ▶ Verschiedene Dateisysteme
 - ▶ Netzwerke, Protokollstacks
 - ▶ i.d.R. Mittlere bis hohe Betriebsmittelanforderungen
 - ▶ Beispiele:
 - ★ QNX-Neutrino
 - ★ PikeOS
 - ★ Linux (RTLinux, RTAI)
 - ★ LynxOS
 - ▶ Übergänge z.T. fließend durch Konfigurierbarkeit des Kerns bzw. Erweiterbarkeit um entsprechende Subsysteme.

Beispiel: Anforderungen

Automatisierungspyramide



Ebene	Charakteristische Anforderungen	Technologie
Planungsebene	Keine Echtzeitanforderungen, Mehrbenutzer-Umgebung, CAD, CAP	klassische EDV-Systeme
Prozessleitebene	Eingeschränkte Echtzeitfähigkeit, Graphische Bedienoberfläche, CAM, CAQ	Workstations, Visualisierungs-Software, Touchscreens
Zellebene	Harte Echtzeitbedingungen, kurze Reaktionszeiten, realisierung globaler konsistenter Zustände, Erkennung (globaler) kritischer Zustände, CNC, SPS, ...	Prozessrechner, Echtzeitbetriebs-systeme , komplexe verteilte Steuerungsaufgaben
Feldebene	Harte Echtzeitbedingungen, kurze bis sehr kurze Reaktionszeiten	μ Controller, Echtzeitkerne

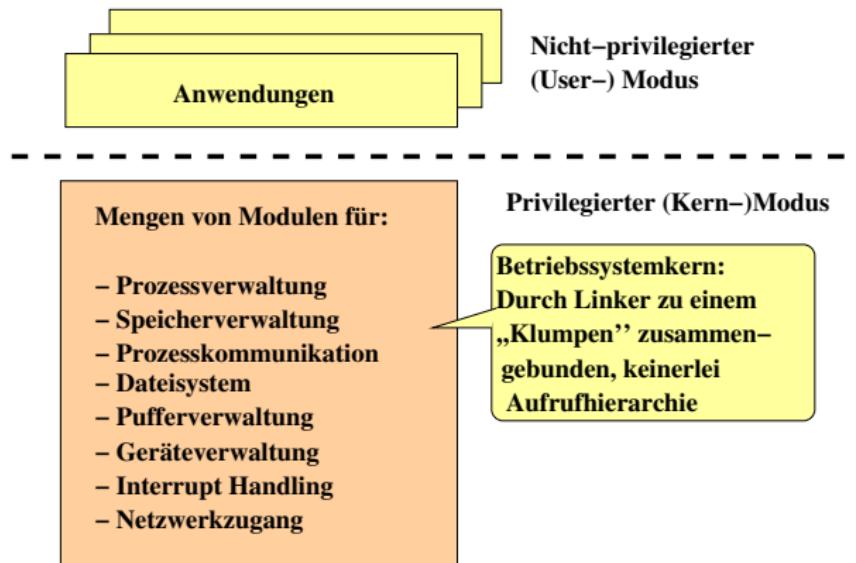
Architektur von Echtzeitbetriebssystemen

Organisationsformen von Betriebssystemen

- Klassifizierung der inneren Organisationsformen von Betriebssystemen
 - ① Monolithische Systeme
 - ② Geschichtete Systeme
 - ③ Kern-im-Kern Systeme
 - ④ Mikrokerne und virtuelle Maschinen

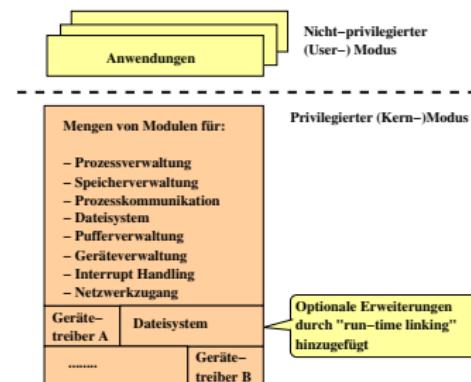
Monolithische Systeme

- Vorwiegende Struktur aller kommerziellen General Purpose Betriebssysteme: z.B. Windows, klassisches UNIX



Monolithische Systeme – Erweiterung

- Erweiterung eines monolithischen Kerns um ladbare Kernmodule
 - ▶ Ursprünglich in klassischen Betriebssystemen primär für Gerätetreiber gedacht
 - ▶ Sehr verbreitet bei eingebetteten Systemen, da so ein hohes Maß an Adaptierbarkeit an benötigte Kerndienste erreicht wird
 - ▶ Heute auch weit verbreitet bei üblichen Betriebssystemen (z.B. Linux)



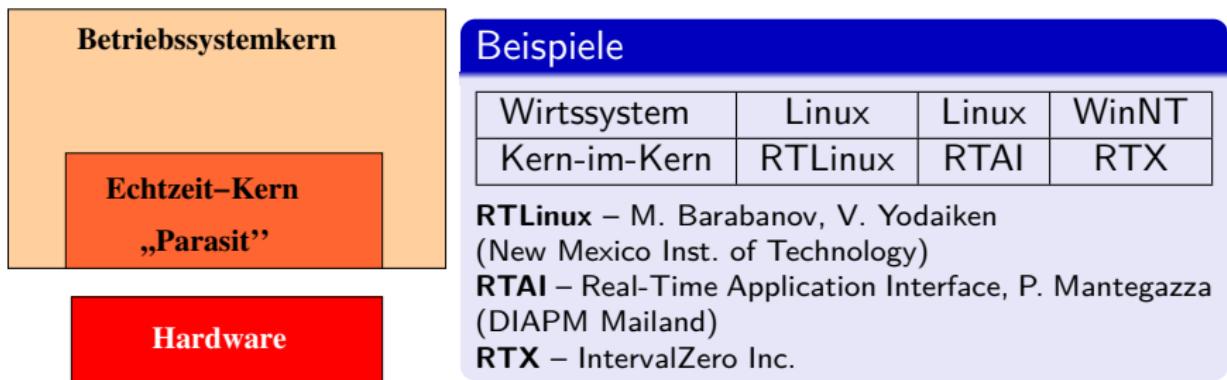
Geschichtete Systeme

Verallgemeinerung des monolithisches Ansatzes:

- Das BS wird als eine Hierarchie von Schichten (engl. layers) entworfen.
- Jede Schicht abstrahiert von gewissen Restriktionen der darunterliegenden Schicht. Die Implementierung einer Schicht benutzt die Dienste der darunterliegenden Schicht.
- Erstes System: THE (Techn. Hochschule Eindhoven, Dijkstra, 1968, einfaches Stapelverarbeitungssystem in Pascal).
- Weitere Verallgemeinerung in MULTICS: "konzentrische (Schutz-) Ringe", verbunden mit nach innen zunehmender Privilegierung, kontrollierter Aufruf zwischen den Ebenen zur Laufzeit.

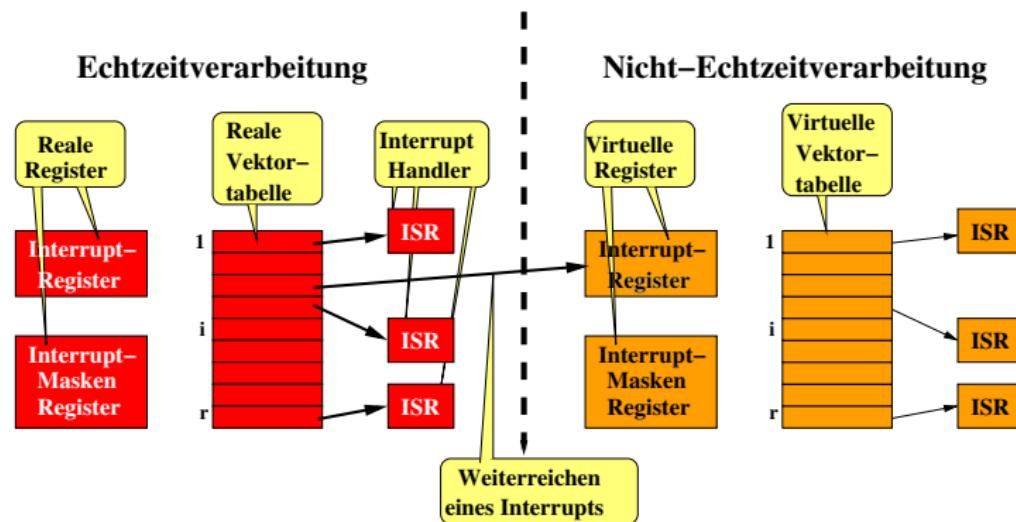
Kern-im-Kern Systeme

- Insbesondere zum „Nachrüsten“ von Echtzeit-Funktionalität in Nicht-Echtzeit Betriebssysteme genutzt
- „Virtualisierung“ des Interrupt-Systems
- Der Echtzeit-Kern setzt sich als Kernmodul des Wirtskerns unter diesen und kontrolliert das reale Interrupt-System
- Der Wirtskern wird zum „Idle“-Prozess des Echtzeit-Kerns

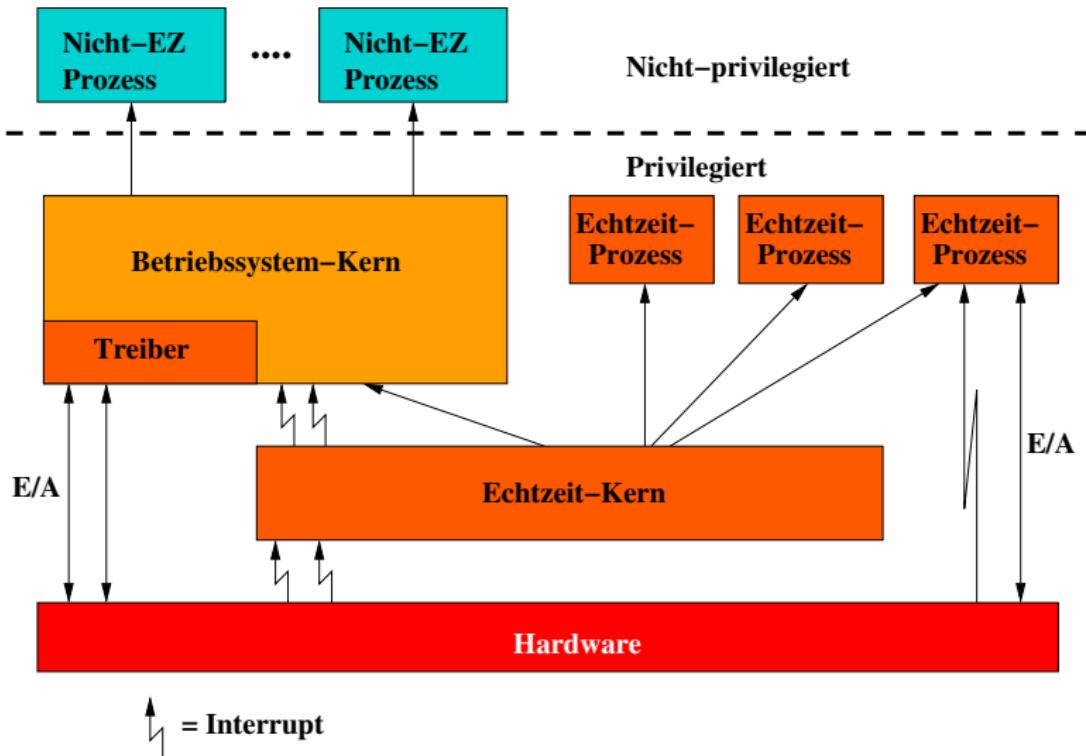


Virtualisierung des Interruptsystems

- Software-Emulation des Interrupt-Systems
- Minimale Änderungen des Wirts-Kerns: Ersetzen aller Interrupt-bezogenen Operationsaufrufe (`cli`, `sti`, `iret`) im Wirts-Kern durch entsprechende emulierende Makros



Gesamtarchitektur



Merkmale

- Echtzeit-Kern: Ziel: Nutzung der Hardware durch Echtzeit-Prozesse mit minimaler Latenzzeit
- Echtzeit-Kern mit allen Komponenten und Echtzeit-Anwendungen laufen im privilegierten Modus
 - Geringe Prozesswechselzeiten für Echtzeit-Prozesse
- aber: Fehler in Echtzeit-Anwendung → Absturz des Wirtskerns!
- aber: Keine Systemdienste des Wirtskerns verfügbar!
- aber: Echtzeit-Anwendungen müssen für jede neue Version des Wirts-Kerns neu kompiliert werden
- Scheduling und Echtzeiteigenschaften von Echtzeit-Prozessen können nicht durch den Wirtskern beeinflußt werden
- Wirtskern: Funktionalität nicht eingeschränkt
- Unmerklich schlechtere Rechenleistung wegen Indirektion der Interrupt-Verarbeitung

Der Echtzeit-Kern als Parasit

- Der Echtzeit-Kern nutzt den Wirtskern zu seiner eigenen Konfiguration:
- Installieren von Komponenten des Echtzeit-Kerns als ladbare Module des Wirtskerns
- Beispiel: Linux + RTAI (oder RTLinux):
 - ▶ Linux insmod und rmmod zum Laden der RTAI-Module
 - ▶ rtai – RTAI framework, interrupt dispatching, timer support
 - ▶ rtai_sched – preemptiver, Prioritäten-basierter Scheduler
 - ▶ rtai_fifo – FIFOs, Semaphoren
 - ▶ rtai_shm – shared memory
 - ▶ rtai_pthread – POSIX threads
- Der RT-Kern nutzt die Funktionen des Wirtskerns soweit irgend möglich
- Beispiel: Geräteinitialisierung (ist nicht echtzeitkritisch)

Mikrokerne und virtuelle Maschinen

- Bei den bisher vorgestellten Organisationsformen wird unnötig viel Funktionalität im privilegierten Modus realisiert
 - Für Funktionen wie Netzwerkprotokolle, Dateisysteme, ja sogar Gerätetreiber ist das keineswegs zwingend erforderlich
 - Da die Funktionen im Kern liegen, zählen sie zur „Trusted Code Base“
 - (s.o.) Dadurch wird die Trusted Code Base –selbst für einfachste Anwendungen– so groß, dass eine umfassende Validierung (geschweige denn eine Verifikation) praktisch unmöglich ist
- ⇒ Mikrokern-Ansatz¹: Funktionen nur dann im Kern, wenn sie ausserhalb nicht realisierbar sind
- ▶ Prozesse, Speicherschutz, Betriebsmittelzuteilung
 - ▶ Nur **ein** Dienst im Kern: Inter-Prozess-Kommunikation (IPC)

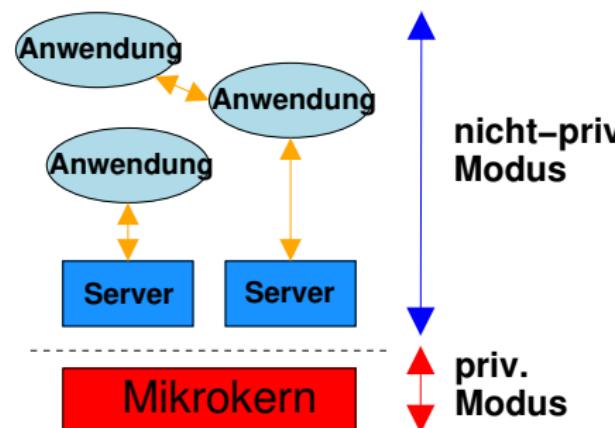
¹Siehe J. Liedtke: „On μ -kernel construction“

Mikrokern-Prinzip

- Ein Mikrokern ist nicht einfach nur ein kleinerer Kern
- Konstruktionsprinzip: Trennung von Methode und Mechanismus (*separation of policy and mechanism*)
- Beispiel: Speicherverwaltung:
 - ▶ Methode/Policy: „Welcher Prozess darf auf welchen Speicher zugreifen?“
 - ▶ Mechanismus/Mechanism: „Wie programmiert man die Memory Management Unit des Prozessors?“
- Prinzip: Mechanismen gehören in den Mikrokern, Methoden nicht!
 - ⇒ Kern wird klein, wenig komplex, kleine „Trusted Code Base“
 - ⇒ Validierung (sogar: Verifikation) wird machbar
- ?! „Aber wo werden dann die Betriebssystem-Dienste erbracht?“

Server/Client-Architektur

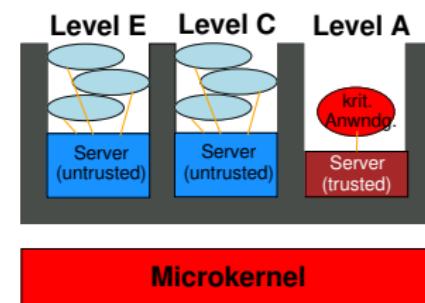
- Dienste werden durch *Server* erbracht
- Prozesse, die im nicht-privilegierten Modus in geschütztem Adressräumen arbeiten
- Interprozesskommunikation ersetzt den Kernauftrag
- Verschiedene Server können verschiedene, alternative Dienstmengen anbieten
- Mehrere Betriebssysteme gleichzeitig in einer Maschine
- Vorteil: Anwendungen müssen nur den Servern trauen, deren Dienste sie nutzen



Server/Client-Architektur

Codecomplexität und Sicherheitsklassifizierung gegeneinander abwägen

- MILS-Architektur²
- Sonderfall: Mikrokern- Prozesse als „Container“ für Betriebssysteme
- ⇒ Virtualisierung (vgl. Xen, VMware, Virtualbox)
- ⇒ Virtual Machine Monitore, Hypervisor: Spezielle Formen von Mikrokernen



²Multiple Independent Levels of Safety / Security

Interne Architektur von BS-Kernen

Aufgaben eines Echtzeit-Betriebssystems

- Verwaltung von Rechenprozessen und Betriebsmitteln unter Erfüllung der Forderungen nach Rechtzeitigkeit, Gleichzeitigkeit und Effizienz

Betriebssystemfunktionen:

- Organisation des Ablaufs der Rechenprozesse (Scheduling)
- Organisation der Interruptverwaltung
- Organisation der Speicherverwaltung
- Organisation der Ein-/Ausgabe
- Organisation des Ablaufs bei irregulären Betriebszuständen und des (Wieder-) Anlaufs

Rechenprozess-Verwaltung

Arten von Rechenprozessen

- Anwenderprozesse
- Systemprozesse:
 - ▶ zentrale Protokollierung
 - ▶ Verwaltung von Speichermedien
 - ▶ Netzwerk-Protokollabwicklung
 - ▶ Idle-Prozess

Aufgaben bei der Rechenprozess-Verwaltung

- Koordinierung des Ablaufs von Anwender- und Systemprozessen
- Parallelbetrieb möglichst vieler Betriebsmittel
- Abarbeitung von Warteschlangen bei Betriebsmitteln
- Synchronisierung von Anwender-Systemprozessen

Datenstrukturen zur Prozessverwaltung (1)

Prozesstabelle

0	PVB
1	PVB
2	PVB
3	PVB
.	PVB
.	PVB
.	PVB
n-1	PVB

- Liste der existierenden Rechenprozesse
- Elemente der Liste:
Prozessverwaltungsblock: PVB
- auch genannt:
 - ▶ PCB – *Process Control Block*
 - ▶ TCB – *Task Control Block/ Thread Control Block*

Datenstrukturen zur Prozessverwaltung (2)

Prozessverwaltungsblock – PVB

- Dient zum Speichern des gesamten Zustandes eines Prozesses

Prozessverwaltung	Speicherverwaltung	evtl. Dateisystem
Register	Zeiger auf Speichersegmente	Wurzelverzeichnis
Programmzähler	Belegtliste	aktuelles Verzeichnis
Programmstatuswort	Nachrichtenpuffer	offene Dateideskriptoren
Stack-Zeiger	Zugriffsrechte	Aufrufparameter
Prozesszustand	verschiedene Flags	verschiedene Flags
Prozessnummer		
Prozesserzeugungszeitpunkt		
Terminierungsstatus		
verbrauchte Prozessorzeit		
Alarm-Zeitpunkt		
Signalstatus		
Zeiger auf Nachrichten		
verschiedene Flags		

zusätzlich: Zeiger zur Verkettung in Warteschlangen

Datenstrukturen zur Prozessverwaltung (2)

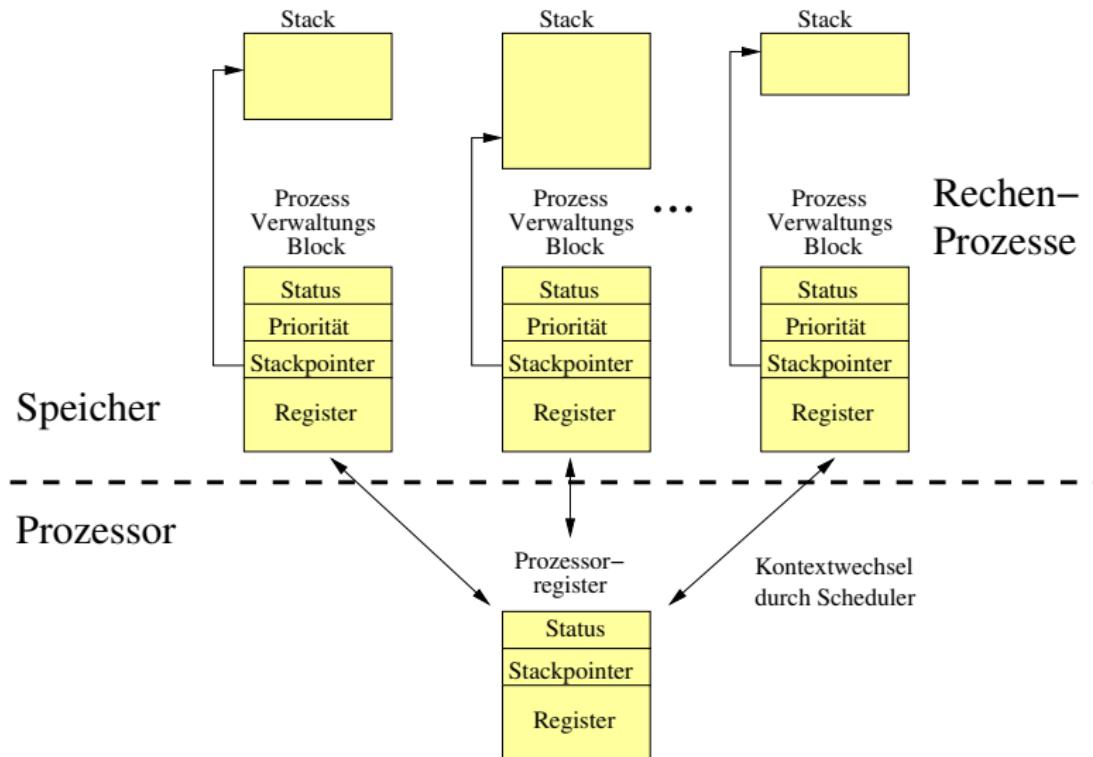
Prozessverwaltungsblock – PVB

- Dient zum Speichern des gesamten Zustandes eines Prozesses

Prozessverwaltung	Speicherverwaltung	evtl. Dateisystem
Register	Zeiger auf Speichersegmente	Wurzelverzeichnis
Programmzähler	Belegtliste	aktuelles Verzeichnis
Programmstatuswort	Nachrichtenpuffer	offene Dateideskriptoren
Stack-Zeiger	Zugriffsrechte	Aufrufparameter
Prozesszustand	verschiedene Flags	verschiedene Flags
Prozessnummer		
Prozesserzeugungszeitpunkt		
Terminierungsstatus		
verbrauchte Prozessorzeit		
Alarm-Zeitpunkt		
Signalstatus		
Zeiger auf Nachrichten		
verschiedene Flags		

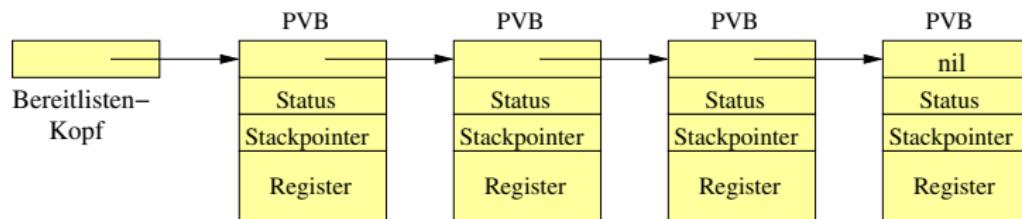
zusätzlich: Zeiger zur Verkettung in Warteschlangen

Mehrprozessbetrieb



Warteschlangenstruktur (1)

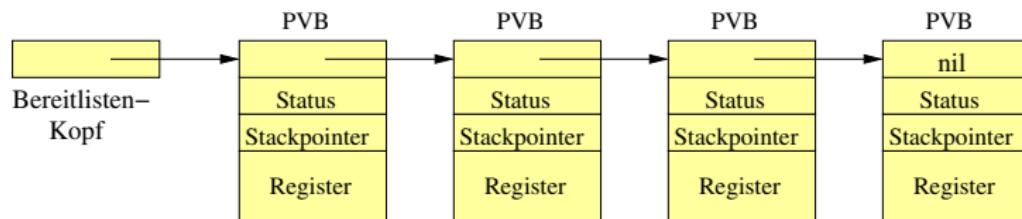
- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



- Scheduler entnimmt vordersten Rechenprozess und teilt ihm den Prozessor zu
 - Ankommende Rechenprozesse werden am Ende angefügt
- ⇒ FIFO-Scheduler

Warteschlangenstruktur (2)

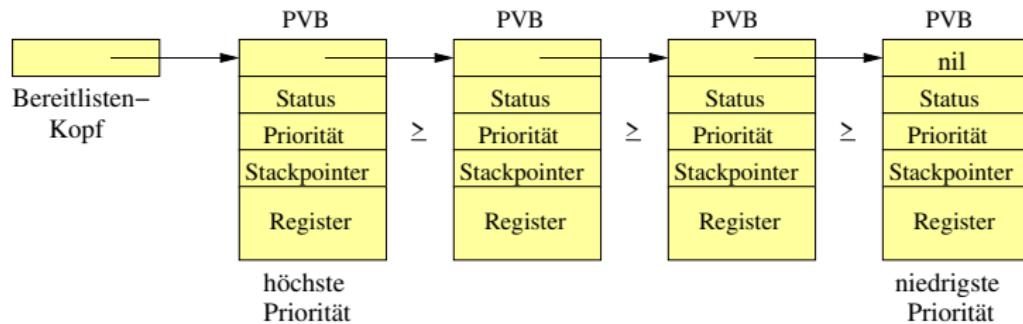
- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



- Nach Ablauf einer Zeitscheibe wird der laufende Rechenprozess unterbrochen und am Ende angefügt
 - Ansonsten unverändert
- ⇒ Round-Robin-Scheduler

Warteschlangenstruktur (3)

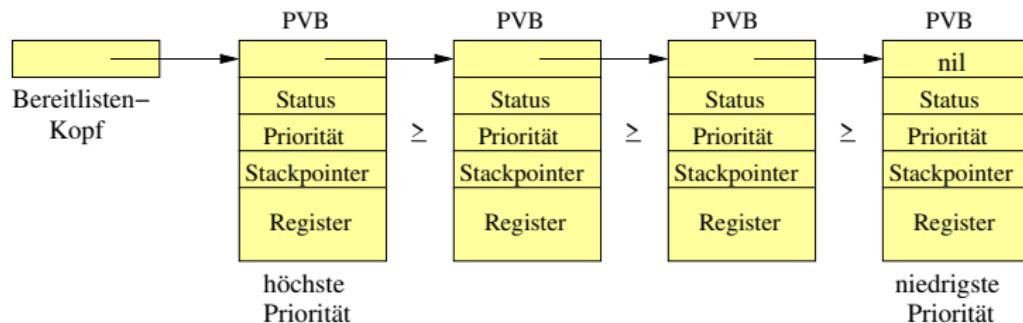
- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



- Ankommende Rechenprozesse werden anhand ihrer Priorität in die Liste eingefügt (Bei gleicher Priorität: FIFO-Reihenfolge)
- ⇒ Prioritäts-Scheduler
- Problem: Laufzeitaufwand beim einsortieren ist nicht konstant (hängt von der Länge der Liste ab) → Komplexität $O(n)$

Warteschlangenstruktur (3)

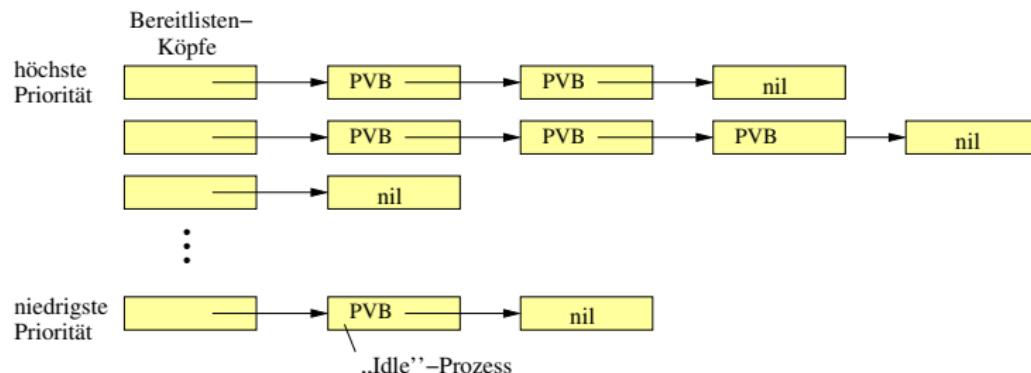
- Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder *Ready Queue*):



- Ankommende Rechenprozesse werden anhand ihrer Priorität in die Liste eingefügt (Bei gleicher Priorität: FIFO-Reihenfolge)
- ⇒ Prioritäts-Scheduler
- Problem: Laufzeitaufwand beim einsortieren ist nicht konstant (hängt von der Länge der Liste ab) → Komplexität $O(n)$

Warteschlangenstruktur (4)

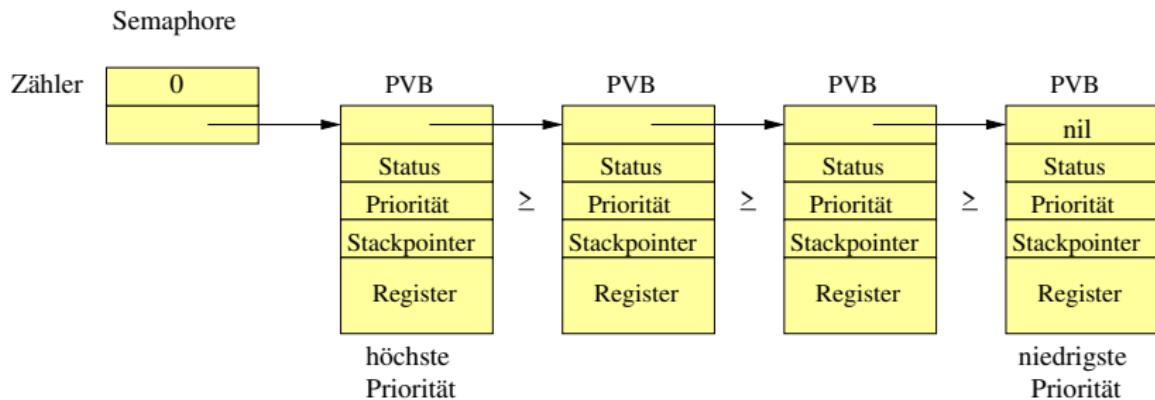
- (Für Echtzeitbetriebssysteme) typische Struktur der Bereit-Liste:



- Eine Bereitliste pro Prioritätsstufe
- Vorteil: Konstanter Laufzeitaufwand beim Einreihen („O(1)-Scheduler“)
- Nachteil: Feste Anzahl möglicher Prioritäten (typisch: 32-256)

Verwaltung blockierter Rechenprozesse (1)

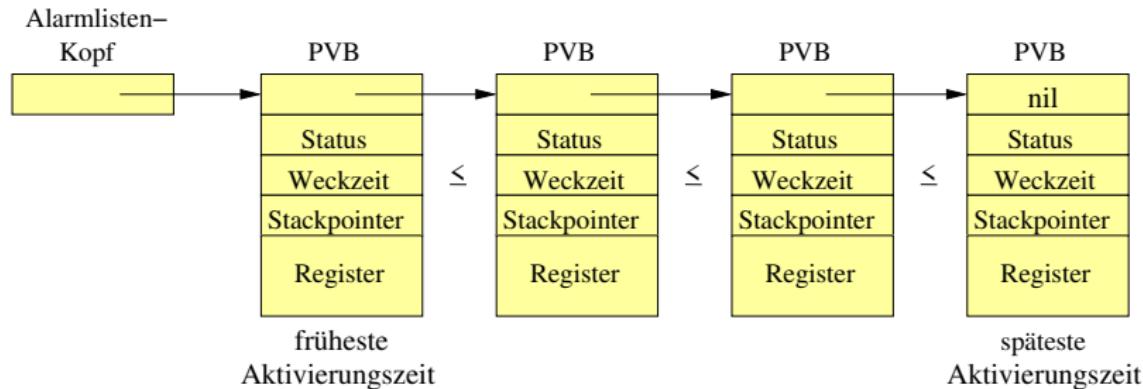
- z.B. Warten auf einen Semaphor
- Semaphor besteht aus Zähler und Wartelistenkopf



- Rechenprozesse sind entweder rechenwillig oder blockiert
- Können in gleicher Weise wie in Bereitliste verkettet werden
- Einreihen nach Priorität oder FIFO-Reihenfolge möglich

Verwaltung blockierter Rechenprozesse (2)

- z.B. Zeitbegrenztes Warten (*timed sleep*)



- Einreihen nach wachsender (absoluter) Weckzeit
- Interrupt-Service Routine der Hardware-Uhr muss stets nur den vordersten Eintrag prüfen ($O(1)!!$)

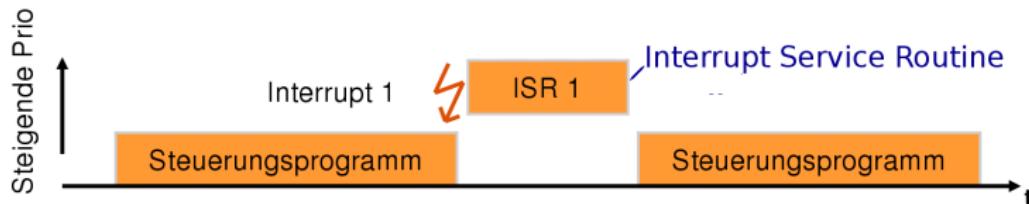
Interrupt-Verwaltung

- Unterbrechung des geplanten Programmablaufs
- Beauftragung einer Behandlungsroutine (ISR)

Geplanter Programmablauf: (ohne Interrupt)

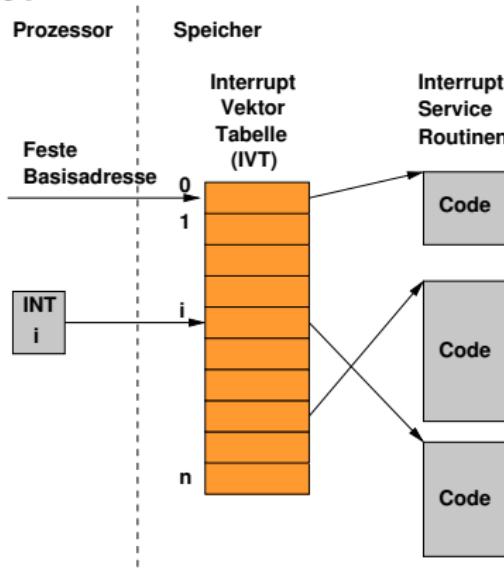


Tatsächlicher Ablauf: (mit Interrupt)



Bestimmung der ISR

Typisch:



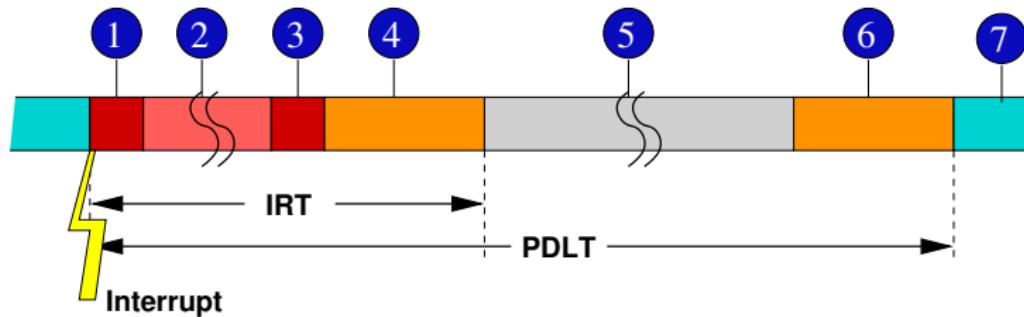
- **Interrupt Service Routine (ISR):** Gerätespezifische Routine zur Behandlung von Interrupts
- Zuordnung von Interrupt(-Nummer) zu ISR über *Interrupt-Vektor-Tabelle (IVT)*

Echtzeitbezogene Kenngrößen

Wichtige Kenngrößen zur Beurteilung der Echtzeitfähigkeit:

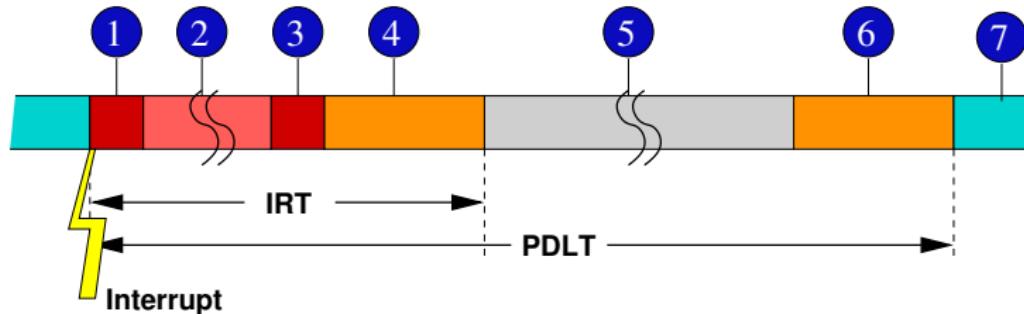
- Interrupt Latency Time
- Interrupt Response Time
- Interrupt Cycle Time
- Process Dispatch Latency Time

Detaillierter Ablauf der Interruptbehandlung (1) *



- ① Setzen der Interrupt-Anforderung durch die Hardware (HW-Verzögerung).
- ② Warten auf Freigabe einer ev. Interruptsperre, ev. Abwarten der Bearbeitung aller höher prioren Interrupts, Arbitrierung bei Anliegen mehrerer Interrupts.
- ③ Beenden der Ausführung des aktuellen Befehls.

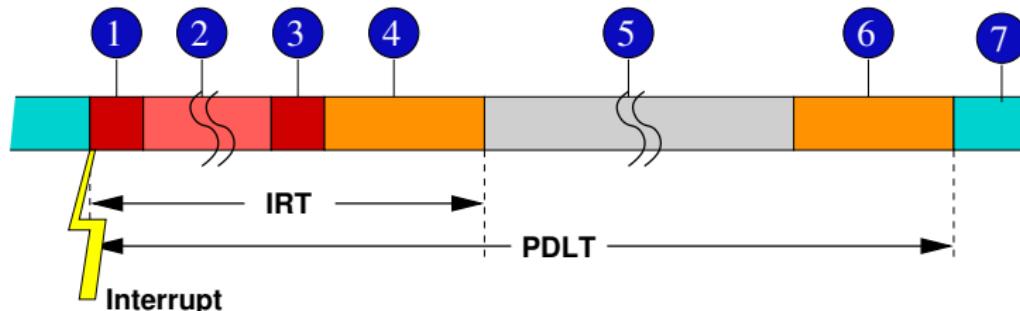
Detaillierter Ablauf der Interruptbehandlung (2) *



④ Kontextwechsel

- ▶ Retten Befehlszähler und Statusregister
- ▶ Bestimmung der zugehörigen Interrupt Service Routine (ISR).
- ▶ Verzweigen in die Interrupt Service Routine

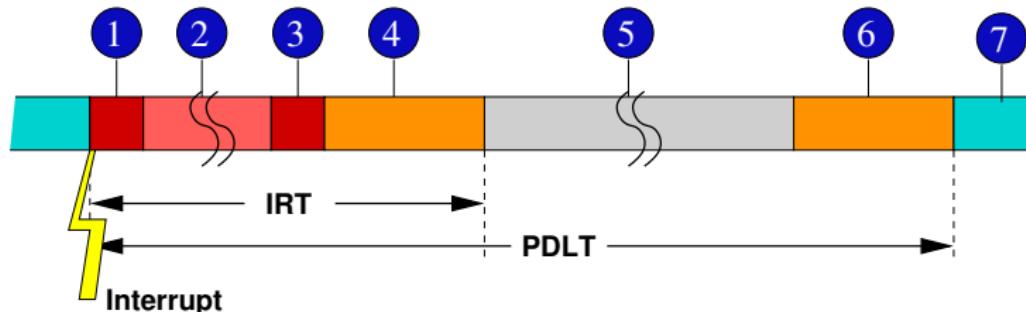
Detaillierter Ablauf der Interruptbehandlung (3)



⑤ Ausführen der Interrupt Service Routine (Interrupt Handler)

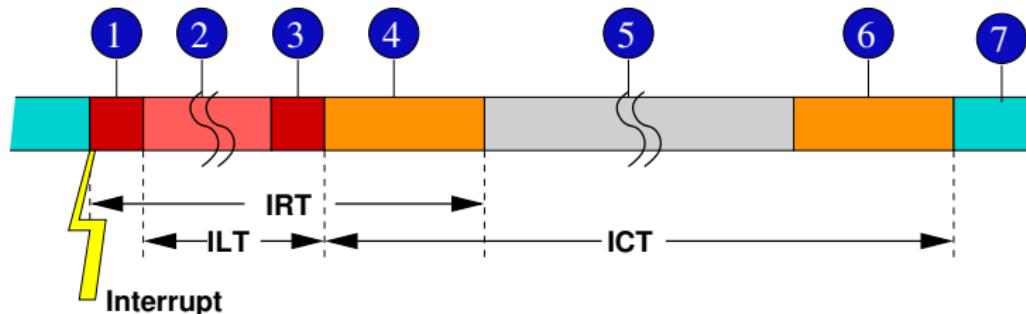
- ▶ Retten der Umgebung der unterbrochenen Prozesses, soweit dessen Betriebsmittel benötigt werden (häufig vollständiger Registersatz)
- ▶ ev. Aufbau einer für das Betriebssystem notwendigen Interrupt-Bearbeitungsumgebung
- ▶ Eigentlichen ISR-Code ausführen.
- ▶ Wiederherstellen der Umgebung des unterbrochenen Prozesses.

Detaillierter Ablauf der Interruptbehandlung (4) *



- ⑥ Rückkehr aus der Interruptbehandlung RETI
(Befehlszähler, PSW laden)
- ⑦ Fortsetzung des unterbrochenen Rechenprozesses

Kenngrößen bei der Interruptbehandlung



- **IRT (interrupt response time):** Zeit vom Eintreten des Interrupts bis zu Beginn der Ausführung der ISR
- **ILT (interrupt latency time):** Zeit für das Warten auf Interruptfreigabe (2) und Beenden des aktuellen Befehls (3)
- **ICT (interrupt cycle time):** Gesamtdauer der Bearbeitung des Interrupts

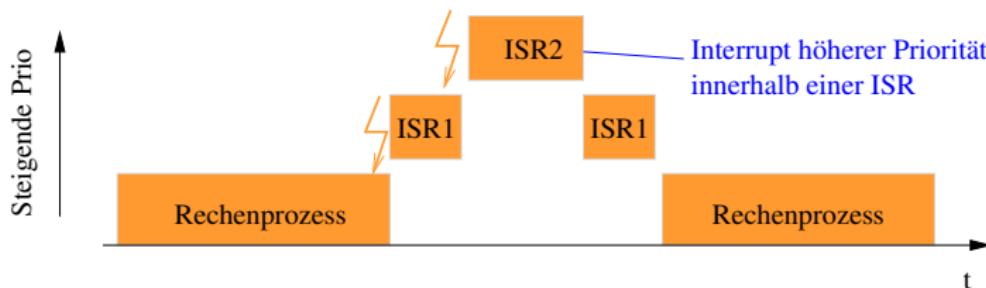
Interrupt Response Time (IRT)

- Typisches Merkmal zur Beschreibung der Reaktionsfähigkeit
- Typisch 1 – 100 μs ,
- hängt stark von Prozessor-Hardware ab
- Vgl. Tabelle Zusammenfassung Produkte

Priorisierte Interrupts

Interrupts sind unterschiedlich priorisiert

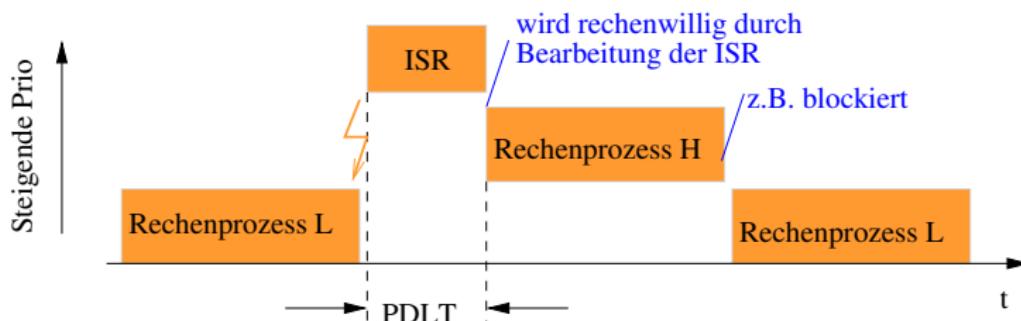
→ Möglicher Ablauf:



- Maximale Verschachtelungstiefe = Anzahl der Interrupts

Höherpriore Rechenprozesse

ISR setzt höherprioren, zuvor blockierten Prozess rechenwillig



- Rechenprozess wird unmittelbar nach der ISR gestartet
- Scheduler-Aufruf am Ende der ISR

Weitere wichtige Kenngröße:

- PDLT (*process dispatch latency time*): Zeit vom Eintreten des Interrupts bis zu Beginn des aktvierten Rechenprozesses

Auswirkung von Interrupt-Sperren

- Interrupt-Anforderungen können jederzeit auftreten.
- Interrupt-Sperren werden durch spezielle Maschinenbefehle realisiert.
- Interrupt-Sperren dienen der Vermeidung von Inkonsistenzen von Daten, die gemeinsam von unterbrochener Aktivität und Interrupt-Handler bearbeitet werden.
- Nachteile von Interrupt-Sperren:
 - ▶ Interrupt-Antwortzeit (IRT) wächst!
(worst case: um die Dauer der längsten Interrupt-Sperre).
 - ▶ Dadurch sinkt schnelle Reaktionsfähigkeit.
 - ▶ Interrupt-Sperren können periodische Aktivierung von Tasks bzw. Handlern verzögern (z.B. Störung von Regelalgorithmen).

Auslagern der Interrupt-Bearbeitung

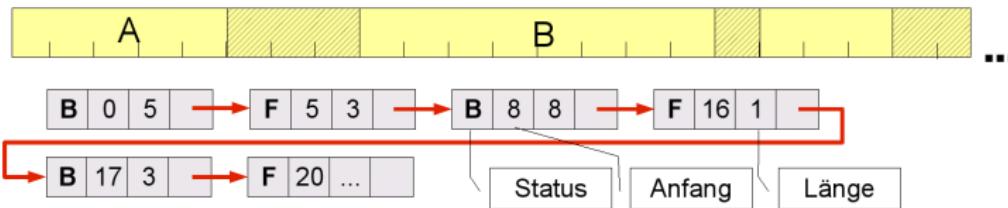
- Für die Dauer der Interrupt-Bearbeitung werden i.d.R. Interrupts gleicher oder niedrigerer Priorität nicht sichtbar (bleiben maskiert).
- Um entsprechende Ereignisse möglichst früh wahrnehmen zu können, ist es wünschenswert, die Interrupt-Bearbeitung durch Auslagern wesentlicher Aktivitäten zu verkürzen.
- Vorgehensweise:
 - In der ISR nur das unbedingt Notwendige tun.
 - ▶ Aktivität außerhalb der ISR einleiten
 - ▶ (z.B. Auftragerteilung an einen Thread (ISR Thread, Kernel Thread, „Deferred Procedure call“ (DPC)),
 - ▶ Event oder Message an Anwenderprozess zustellen, o.ä.).

Speicherverwaltung

Speicher: Je schneller desto teurer

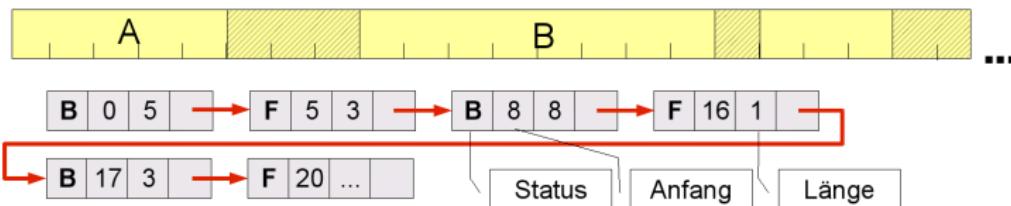
- Speicherhierarchie-Ebenen
 - ▶ Cache-Speicher (besonders schneller Halbleiterspeicher)
 - ▶ Arbeitsspeicher
 - ▶ Plattenspeicher
 - ▶ Backup-Speicher (z.B. Magnetband)
- Aufgaben der Speicherverwaltung
 - ▶ Optimale Ausnutzung der „schnellen“ Speicher
 - ▶ Koordinierung des gemeinsamen Zugriffs auf einen Speicherbereich
 - ▶ Schutz des Speicherbereichs verschiedener Rechenprozesse gegen Fehlzugriffe
 - ▶ Zuweisung von physikalischen Speicheradressen für die logischen Namen in Anwenderprogrammen

Einfache Speicherverwaltung – malloc() (1)



- Jedem belegten und jedem freien Speicherbereich wird ein Listenelement zugeordnet.
- Segmente dürfen variabel lang sein.
- Jedes Listenelement enthält Startadresse und Länge des Segments, sowie den Status (B=belegt, F=frei).
- Gefundenes freies Segment wird (falls zu groß) aufgespalten.
- Freigegebenes Segment wird ggf. mit ebenfalls freien Nachbarssegmenten „verschmolzen“

Einfache Speicherverwaltung – malloc() (2)



- Die freien Segmente können auch in separater Liste geführt werden („Freiliste“)
- Die Freiliste kann in sich selbst gehalten werden, d.h. in den verwalteten freien Bereichen (\rightarrow kein weiterer Speicher nötig).
- Die Segmentliste kann nach Anfangsadressen geordnet sein.
Vorteil: freiwerdendes Segment kann mit benachbartem freien Bereich zu einem freien Segment „verschmolzen“ werden.
- Freiliste kann alternativ nach der Größe des freien Bereichs geordnet sein.
Vorteil: Vereinfachung beim Suchen nach einem freien Bereich bestimmter Länge.

Ein-/Ausgabesteuerung

Verschiedenartige Typen von Ein-/Ausgabegeräten

- Unterscheidung in Geschwindigkeit
- Unterscheidung in Datenformaten

Realisierung der Ein-/Ausgabesteuerung

- Hardwareunabhängige Ebene für die Datenverwaltung und den Datentransport
- Hardwareabhängige Ebene, die alle gerätespezifischen Eigenschaften berücksichtigt (Treiber-Programme)

Behandlung irregulärer Betriebszustände (1)



Klassifizierung von Fehlern

- fehlerhafte Benutzereingaben: nicht zulässige Eingaben müssen mit Fehlerhinweisen abgelehnt werden.

Siehe „Sunk by Windows NT"³

„The source of the problem on the USS Yorktown was that bad data was fed into an application running on one of the 16 computers on the LAN. The data contained a zero where it shouldn't have, and when the software attempted to divide by zero, a buffer overrun occurred – crashing the entire network and causing the ship to lose control of its propulsion system“

- fehlerhafte Anwenderprogramme: Gewährleistung, dass ein fehlerhaftes Anwenderprogramm keine Auswirkungen auf andere Programme hat

³<http://www.wired.com/science/discoveries/news/1998/07/13987>

Behandlung irregulärer Betriebszustände (1)



Klassifizierung von Fehlern (Forts.)

- Hardwarefehler/-ausfälle:
 - ▶ Erkennung von Hardwarefehlern bzw. -ausfällen
 - ▶ Rekonfigurierung ohne die fehlerhaften Teile
 - ▶ Abschaltsequenzen bei Stromausfällen
- Deadlocks aufgrund dynamischer Konstellationen
 - ▶ sichere Vermeidung von Deadlocks ist nicht immer möglich

Der Markt für Echtzeitbetriebssysteme

Kriterien bei der Auswahl von Echtzeit-Betriebssystemen

- Entwicklungs- und Zielumgebung
 - ▶ Anzahl von Tasks
 - ▶ Prioritätsstufen
 - ▶ Taskwechselzeiten
 - ▶ Interruptlatenzzeit
- Anpassung an spezielle Zielumgebungen
 - ▶ z.B. Betrieb ohne Festplatte
- Allgemeine Eigenschaften
 - ▶ Schedulingverfahren
 - ▶ Interprozesskommunikation
 - ▶ Netzwerkkommunikation
 - ▶ Gestaltung Benutzeroberfläche

Welches Betriebssystem?

- Aufgrund der vielfältigen Anforderungen gibt es nicht **das** Echtzeitbetriebssystem
- Gerade „so viel Betriebssystem, wie nötig“
- Oft ist bereits bei der Konzeption bekannt, wieviel das konkret ist
 - In diesem Fall sollte das Betriebssystem statisch skalierbar sein
- Andererseits muss es auch kostengünstig sein (Auch Software-Lizenzen kosten Geld)
- So stellt u.U. ein eigentlich zu umfangreiches, nur bedingt echtzeitfähiges, dabei aber sehr kostengünstiges Betriebssystem (z.B. Linux) den wirtschaftlich besseren Kompromiss dar

Echtzeitunterstützung für Linux

Linux RT_PREEMPT patch: www.osadl.org

General road map of the main patch components

Architecture	x86	x86/64	powerpc	arm	mips	68knommu
Feature						
Deterministic Scheduler	●	●	●	●	●	●
Preemption Support	●	●	●	●	●	●
PI Mutexes	●	●	●	●	●	● ³
High-Resolution Timer	●	● ¹	● ¹	● ¹	● ¹	●
Preemptive Read-Copy Update	● ²					
IRQ Threads	● ⁴	● ^{3,4,5}				
Raw Spinlock Annotation	● ⁶					
Forced IRQ Threads	● ⁷					
R/W Semaphore Cleanup	● ⁷					
Full Realtime Preemption Support	● ³					

● Available in mainline Linux

● Available when Realtime Preempt patches applied

<https://www.osadl.org/Realtime-Linux/projects-realtime-linux.0.html>

Marktübersicht

breites Angebot am Markt für alle Klassen:

- es gibt nicht „den“ dominierenden Anbieter
(stark unterschiedliche Ziel-Hardware und Entwicklungsplattformen bieten außerdem zahlreiche Nischen)
- Auch: breites Preisspektrum
 - ▶ Unterscheidung Entwicklungsliczenzen
 - ▶ Runtime-Lizenzen für Target-Systeme
- Nach einer Umfrage aus 2006⁴ wurden 30% der eingebetteten Systeme ohne Betriebssystem realisiert

⁴<http://www.embedded.com/columns/showArticle.jhtml?articleID=187203732>

Echtzeit-BS Marktübersicht (iX 4/2012)



Anbieter von Echtzeitbetriebssystemen

Betriebssystem	Hersteller	Webseite	Echtzeitverhalten ¹	Lizenzenmodell
μC/OS-II; μC/OS-III; μC/OS-SEK; μC/OS-MMU	Micrium (Distributor: Embedded Office)	www.embedded-office.de , www.micrium.com	Hard-RT / 100 bis 120 µs	proprietär, Source-Code wird mitgeliefert
EB tresos	Elektronik	www.elektronik.com	k. A.	proprietär
eCosPro	eCosCentric	www.ecoscentric.com	Hard-RT / 0,67 µs wie andere Linux-Systeme	modifizierte GNU GPL
ElinOS	Sysgo AG	www.sysgo.com	Hard-RT / je nach Architektur	GNU GPL
embOS	Segger Microcontroller	www.segger.com	Hard-RT / 1 µs (ARM, 200 MHz)	proprietär, Source-Code erhältlich
Euros	Euros - Embedded Systems	www.euros-embedded.com	Hard-RT / 10 µs	proprietär
Integrity	Grenghills Software	www.ghs.com	Hard-RT / je nach Architektur	proprietär
lynxOS, lynxOS-178, lynxOS-SE	lynxWorks	www.linxworks.com	k. A.	proprietär
Microstar	Vector Informatik	www.vector.com	Hard-RT / k. A.	proprietär
Microware OS-9	Radisys (Distributor: Microsys)	www.radisys.com/germany	k. A.	proprietär, Source-Code erhältlich
Monta Vista Linux	Montavista	www.mvista.com	k. A.	GNU GPL
Neutrino	QNX Software Systems GmbH & Co. KG	www.qnx.com/company/germany	Hard-RT / 0,5 bis 2,6 µs z.B. ARM Cortex: 0,5 µs)	proprietär, Teile des Source-Codes sind offen
Nucleus	Mentor Graphics Deutschland GmbH	www.mentor.com/germany	Hard-RT / ja	proprietär
OSE, OSfek	Enea	www.enea.de	k. A.	proprietär
PikeOS	Sysgo AG	www.sysgo.com	Hard-RT / < 1 µs	proprietär
Realtime Linux (OSADL recommends 2.6.33.7.2+r30)	OSADL	www.osdl.org/Realtime-Linux.projects-realtime-linux.0.html	Hard-RT / max. 100 000 Taktzyklen (z. B. 1-GHz-CPU: 100 µs)	GNU GPL v2
Red Hat Enterprise MRG	Red Hat	www.redhat.com/mrg/	Soft-RT / 8 µs	GNU GPL
RMOS3	Siemens AG	www.siemens.de/rmos3	Hard-RT / 10 µs	proprietär
RTA-DECK3, RTA-OS3.0	ETAS	www.etas.com/de	Hard-RT / Autosar-4.0-konform	proprietär
RTOS-32	On Time Software	www.on-time.com	Hard-RT / < 5 µs	proprietär, Source-Code erhältlich
RTOS-UH	IEP	www.iep.de	Hard-RT / 1 µs (1 GHz Power-PC)	proprietär
SNX RTOS	Micro Digital (Distributor: Embedded Tools)	www.smntos.com , www.embedded-tools.de	Hard-RT / 78 Taktzyklen (z.B. 0,4 µs auf ARM 200 MHz)	proprietär
SUSE Linux Enterprise Real Time	SUSE Linux GmbH	www.suse.com/de/de/products/realtime/	Hard-RT / 10 bis 100 µs	GNU GPL
Symbio µnOS	Miray Software	www.miray.de , www.symbio.com	Hard-RT / ARM PXA-320: 0,2 ms	proprietär
Thread X	Express Logic	www.expresslogic.de , www.rtos.com	k. A.	proprietär, Source-Code wird mitgeliefert
VxWorks	Wind River Systems	www.windriver.com/de	Hard-RT / < 10µs	proprietär
Wind River Linux	Wind River Systems	www.windriver.com/de	wie andere Linux-Systeme	k. A.
Windows Embedded Compact 7	Microsoft	www.microsoft.com/windowsembedded	k. A.	proprietär

Alle Daten sind Herstellerangaben, ¹Definition Hard- und Soft-RT siehe Text, dahinter minimale garantierter Antwortzeiten. Die tatsächlich erreichbaren Werte hängen stark von der Kombination Hard- und Software ab.

Freie Produkte (iX 4/2012)

Auswahl freier Echtzeitprojekte

Projekt	Webseite	Lizenz
Atomthreads	atomthreads.com	BSD-Lizenz
ecos	ecos.sourceforge.org	modified GNU GPL
EmboX	code.google.com/p/embox/	BSD
FreeOSEK	opensem.sourceforge.net	GPLv3
freertos	www.freertos.org	modified GNU GPL
Realtime for Debian	debian.pengutronix.de	GNU GPL
RTLinuxFree	www.rtlinuxfree.com	GNU GPL
TinyOS	www.tinyos.net	BSD
Ubuntu RealTime-Erweiterung	wiki.ubuntu.com/RealTime	GNU GPL
Xenomai	www.xenomai.org	GPLv2

Beispiele von Echtzeitbetriebssystemen

Gliederung

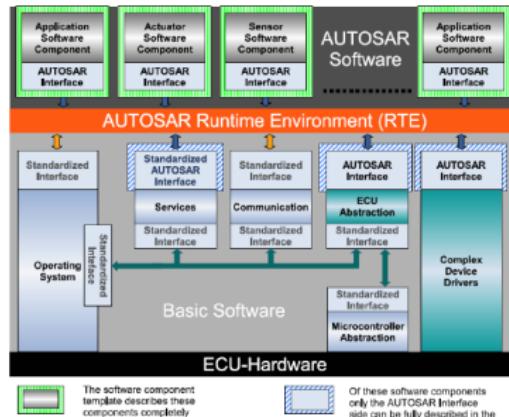
- ① AUTOSAR/OSEK
- ② POSIX

AUTOSAR

- AUTOSAR: AUTomotive Open System ARchitecture
- Weltweiter Zusammenschluss von Automobilherstellern und -zulieferern
- Ziel: Standardisierung einer Software-Architektur für Automotive-Systeme
- (Mittlerweile) zwei Plattformen:
 - ▶ *Classic*: Statische Laufzeitumgebung (Runtime Environment, RTE) auf Basis OSEK (s.u.) für „klassische“ Steuerungs- und Regelungsanwendungen
 - ▶ *Adaptive*: Dynamische Laufzeitumgebung auf Basis des POSIX-Standards (s.u.) für „neuere“ Anwendungen (z.B. autonomes Fahren)
- Konsortium beschliesst Standards, Softwarehersteller sind aufgefordert, diese zu implementieren

AUTOSAR Architektur

- AUTOSAR RTE („Run Time Environment“): Schnittstelle für (ggf. verteilte) Applikationen
- Ortstransparenz durch Sicht als: „Virtual Function Bus“

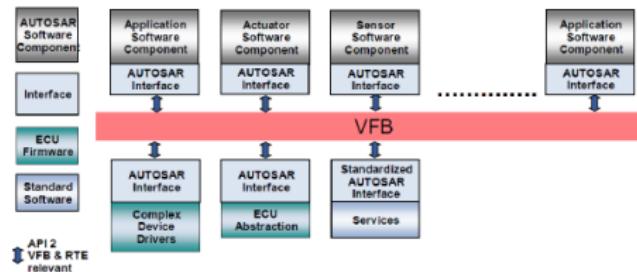


Quelle: Nico Naumann: „AUTOSAR Runtime Environment and Virtual Function Bus“

https://hpi.de/fileadmin/hpi/FG...AUTOSAR0809/NicoNaumann_RTE_VFB.pdf

AUTOSAR Architektur

- AUTOSAR RTE („Run Time Environment“): Schnittstelle für (ggf. verteilte) Applikationen
- Ortstransparenz durch Sicht als: „Virtual Function Bus“



Quelle: Nico Naumann: „AUTOSAR Runtime Environment and Virtual Function Bus“

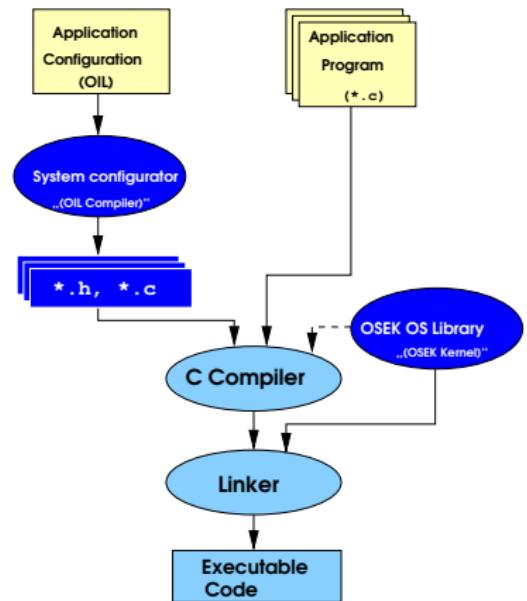
https://hpi.de/fileadmin/hpi/FG...AUTOSAR0809/NicoNaumann_RTE_VFB.pdf

OSEK-OS

- OSEK: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
- Ursprung: Franz. VDX-Initiative 1988, verschmolzen mit deutschem OSEK-Konsortium in 1994
- OSEK OS: Spezifikation für Echtzeitbetriebssysteme, i.w. für Bereich Automotive
- Offener Standard seit 1997, Internationaler Standard ISO 17356-3 seit 2005
- Es existiert eine Vielzahl an Produkten und auch freien Implementierungen nach OSEK-Standard:
 - ▶ Arctic Core <https://www.arccore.com/>
 - ▶ Erika Enterprise <http://erika.tuxfamily.org/drupal/>
 - ▶ FreeOSEK <https://github.com/ciaa/Firmware/>
 - ▶ nxtOSEK (für Lego Mindstorms)
<https://en.wikipedia.org/wiki/NxtOSEK>
 - ▶
- OSEK-Spezifikation 2003 vom AUTOSAR-Konsortium übernommen

Grundidee: statisches System

- Dynamisches Erzeugen/Verwerfen von Objekten ist nicht deterministisch
- **Keine** Funktionen zum Allokieren bzw. Verwerfen von Objekten zur Laufzeit
- Dynamisches Ändern von Taskprioritäten in Verbindung mit Resource Locking kann zu Deadlocks führen
- **Kein** Ändern von Prioritäten zur Laufzeit
- Alle Objekte (Tasks, Events, Timers, etc.) und deren Parameter müssen zur Konfigurationszeit deklariert werden
- Spezielle Sprache dazu: *OIL*⁵)



⁵OSEK Implementation Language

Skalierbarkeit (1)

- OSEK muss auf einem weiten Bereich von Plattformen einsetzbar sein (8-bit - 64-bit)
 - Der Standard definiert vier „Konformanzklassen“ (Untermengen der Scheduler-Funktionalität)
- Validierung von Funktionsargumenten zur Laufzeit kostet Rechenleistung, ist aber während Entwicklung/Test unverzichtbar
 - Der Standard definiert zwei „Error Checking Levels“

Einige OSEK Implementierungen erreichen weniger als 1kB Codegröße.

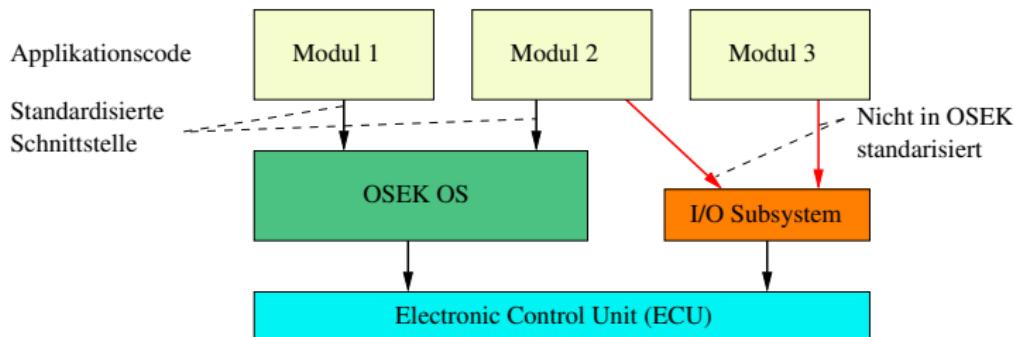
Skalierbarkeit (1)

- OSEK muss auf einem weiten Bereich von Plattformen einsetzbar sein (8-bit - 64-bit)
 - Der Standard definiert vier „Konformanzklassen“ (Untermengen der Scheduler-Funktionalität)
- Validierung von Funktionsargumenten zur Laufzeit kostet Rechenleistung, ist aber während Entwicklung/Test unverzichtbar
 - Der Standard definiert zwei „Error Checking Levels“

Einige OSEK Implementierungen erreichen weniger als 1kB Codegröße.

Skalierbarkeit (2)

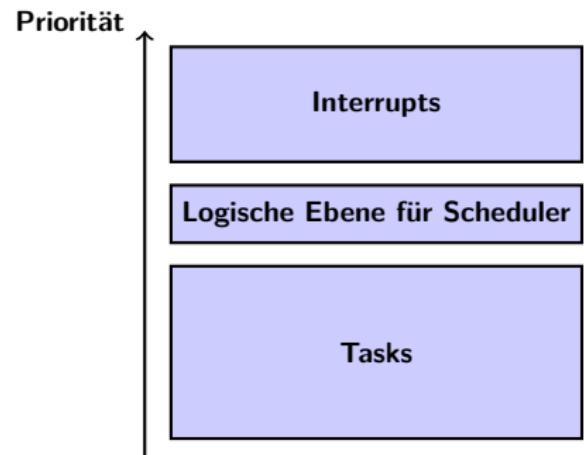
- Plattformspezifische Dinge (z.B.) I/O sind nicht im OSEK OS Standard enthalten
(Classic AUTOSAR definiert „Complex Device Drivers“)



OSEK OS Processing Levels

OSEK definiert drei Ebenen:

- ① Interrupt
- ② Scheduler
- ③ Tasks



OSEK OS Funktionsgruppen

Taskverwaltung

- Aktivierung/Terminierung von Tasks
- Taskstatus Verwaltung, Taskumschaltung

Synchronisation

- Ressourcenverwaltung
- Eventsteuerung

Interrupts

- Dienste zur Interruptverwaltung

Alarme

- relative und absolute Alarme

Intra-Prozessor Messages

- Datenaustausch zwischen Tasks
- (Inter-Prozessor Messages: OSEK COM)
- In AUTOSAR RTE enthalten

Fehlerbehandlung

- „Hook“-Funktionen

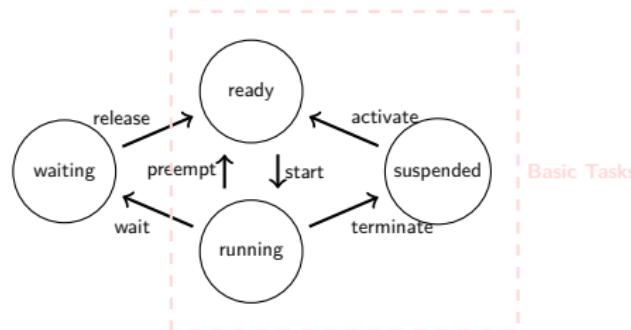
OSEK OS Task Konzept (1)

Basic Tasks: geben die Kontrolle nur ab, wenn

- sie terminieren
- eine höherpriore Task rechenwillig wird
- ein Interrupt auftritt (→ Interrupthandler wird aktiviert)

Extended Tasks:

- kennen zusätzlich den Zustand „warten“ (auf ein Event)



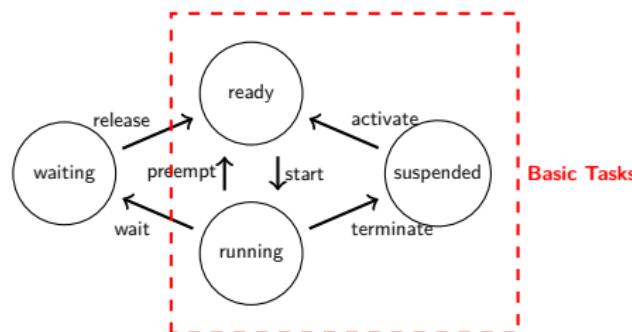
OSEK OS Task Konzept (1)

Basic Tasks: geben die Kontrolle nur ab, wenn

- sie terminieren
- eine höherpriore Task rechenwillig wird
- ein Interrupt auftritt (→ Interrupthandler wird aktiviert)

Extended Tasks:

- kennen zusätzlich den Zustand „warten“ (auf ein Event)



OSEK OS Task Konzept (2)

Aktivierung einer Task mit ActivateTask() oder ChainTask()

- Je nach Konformanzklasse sind Mehrfach-Aktivierungen zulässig (werden nach Priorität bearbeitet)

Der Scheduler wird als Ressource betrachtet

- Tasks können durch Reservieren der Scheduler-Ressource verhindern, dass sie von anderen Tasks verdrängt werden

Prioritätenbasiertes Scheduling

- 0 = niedrigste Priorität
- Je nach Konformanzklasse eine oder mehrere Tasks je Prioritätsstufe

OSEK OS Task Konzept (3)

Non-preemptive Scheduling: Taskwechsel nur möglich, wenn:

- eine Task beendet wird (mit TerminateTask())
- eine Task beendet wird und eine Nachfolgetask aktiviert (mit ChainTask())
- der Scheduler explizit aufgerufen wird (mit Schedule())
- in den Wartezustand übergegangen wird (mit WaitEvent())

Full-preemptive Scheduling: Verdrängung einer Task durch eine andere (höherpriore) ist jederzeit möglich

- Ausnahme: Task hält die Scheduler-Ressource

Mixed-preemptive Scheduling: Koexistenz von Non-preemptive und Full-preemptive

OSEK OS Interruptbearbeitung

Drei Interrupt-Kategorien:

- Kategorie 1: Interrupthandler verwendet keine OSEK OS Systemdienste → geringster Overhead
- Kategorie 2: Interrupthandler verwendet eine Teilmenge der OSEK OS Systemdienste
- Kategorie 3 (Implementierung optional): wie Kategorie 1, jedoch können nach Aufruf von EnterISR() auch OSEK OS Systemdienste verwendet werden wie in Kategorie 2. Dann muss der Handler mit LeaveISR() beendet werden.

Sperren/Erlauben von Interrupts:

- bezogen auf die Interruptquelle
- global

OSEK OS Eventmechanismus

Ein „Event“ ist

- ein Mittel zur Task-Synchronisation
- fest einer Extended Task zugeordnet („Eigentümer“)
- bewirkt den Übergang des Eigentümers in den oder aus dem Wartezustand
- eine Extended Task kann Eigentümer mehrerer Events sein
- Basic Tasks kennen keinen Wartezustand → Basic Tasks können nicht Eigentümer eines Events sein

OSEK OS Ressourcenverwaltung (1)

Koordination gleichzeitiger Zugriffe mehrerer Tasks auf gemeinsame Ressourcen

- gegenseitiger Ausschluss: eine Ressource kann immer nur von einer Task belegt sein
- OSEK-Ressourcen verhindern Prioritätsinversion und Deadlocks
- Zugriff auf Ressourcen führt niemals zu einem Wartezustand
- Der Scheduler wird in OSEK als Ressource behandelt: durch Belegen der Scheduler-Ressource kann eine Task ihre Verdrängung (Preemption) durch andere Tasks verhindern

OSEK OS Ressourcenverwaltung (2)

Priority Ceiling Protokoll

- Bei der Systemkonfiguration ist die Gesamtheit der Tasks, die auf diese Ressource zugreifen (und deren Priorität), bekannt
- ⇒ *Ceiling-Priorität:*
 - ▶ \geq höchste aller Prioritäten der Tasks, die auf die Ressource zugreifen
 - ▶ $<$ niedrigste aller Prioritäten der Tasks, die nicht auf die Ressource zugreifen, und deren Priorität höher liegt, als die Höchstpriorität der Tasks, die darauf zugreifen.
- Wenn eine Task eine Ressource beansprucht, und ihre Priorität unter der Ceiling-Priorität liegt, so wird ihre Priorität vorübergehend auf die Ceiling-Priorität angehoben
- Wenn eine Task die Ressource freigibt, fällt ihre Priorität auf den ursprünglichen Wert zurück

OSEK OS Alarme

Alarme dienen zur Behandlung wiederkehrender Ereignisse

- z.B. periodische Timer-Interrupts, Winkelgeber an Kurbelwellen oder Nockenwellen, etc.
- Kopplung der Ereignisquellen an Zähler (Counters) wird vorausgesetzt (nicht im OSEK-Standard spezifiziert)
- In jeder Implementierung existiert mindestens ein Counter (OS_Counter), alle weiteren Counter sind implementierungsspezifisch
- Mehrere Alarne können einem gemeinsamen Counter zugeordnet werden
- Jedem Alarm wird statisch (im OIL-File) ein Counter und eine Task zugewiesen

OSEK OS Alarme

Alarme dienen zur Behandlung wiederkehrender Ereignisse

- OSEK OS bietet Dienste zum Aktivieren von Tasks, wenn ein Alarm abläuft, (d.h. wenn ein vorgegebener Zählerwert erreicht wird)
- Es gibt relative und absolute Alarme
- Es gibt einzelne oder zyklische Alarme
- Der Ablauf eines Alarms bewirkt wahlweise das Setzen eines Events oder die Aktivierung einer Task

OSEK OS Messages

Nur lokale (Intra-Prozessor) Messages

- nicht-lokale Messages sind in OSEK COM spezifiziert (AUTOSAR: RTE beinhaltet OSEK OS und OSEK COM)
- *Static length Messages*: Größe in der Systemkonfiguration („OIL-File“) festgelegt
- *Dynamic length Messages*: Größe wird zur Laufzeit festgelegt, Maximalgröße in der Systemkonfiguration
- *Unqueued Messages*: neue Nachrichten überschreiben alte
- *Queued Messages*: Nachrichten werden in FIFO-Reihenfolge abgearbeitet; bei Überlauf geht die zuletzt geschriebene Nachricht verloren

OSEK OS Fehlerbehandlung und Debugging (1)

„Hook“-Routinen

- Durch den Anwender spezifizierbare Routinen, die vom System bei bestimmten Ereignissen aufgerufen werden
- Ausführung erfolgt als Teil des Betriebssystems, mit höherer Priorität als alle Tasks, nicht unterbrechbar durch Interrupts der Kategorien 2 und 3
- Aufrufsyntax und -parameter standardisiert, nicht jedoch die Funktion (→ i.A. nicht portabel)
- Nur eine Teilmenge der OSEK OS Funktionen darf aus einer Hook-Funktion aufgerufen werden

OSEK OS Fehlerbehandlung und Debugging (2)

„Hook“-Routinen

- StartupHook(): Wird beim Start des Systems aufgerufen
- ShutdownHook(): Wird beim „Herunterfahren“ des Systems aufgerufen
- ErrorHook(): Wird bei Fehlern aufgerufen,
Unterscheidung zwischen:
 - ▶ Applikationsfehler: Angeforderter Systemdienst konnte nicht ausgeführt werden, System ist intakt. Die Hook-Routine muss entscheiden, was zu tun ist (z.B. Shutdown oder Weitermachen)
 - ▶ Fatale Fehler: System ist nicht intakt (→ Shutdown)
- PreTaskHook(), PostTaskHook(): Werden vor bzw. nach jedem Taskwechsel aufgerufen

POSIX: Portable Operating System Interface (for unI^X)

- Familie internationaler Standards ISO/IEC 9945 ursprünglich spezifiziert durch IEEE Computer Society als IEEE 1003
- Zusammenfassung vieler Teile ab 2008
- Aktuell POSIX.1-2008 = IEEE Std 1003.1-2008, Issue 7, 2016 Edition. <http://pubs.opengroup.org/onlinepubs/9699919799/>
- Üblicherweise API-Spezifikationen für „C“
- Kompatibilität auf Quellcode-Ebene (kein ABI)
- Funktionalität: POSIX Base Definitions, System Interfaces, and Commands and Utilities (which include POSIX.1, extensions for POSIX.1, Real-time Services, Threads Interface, Real-time Extensions, Security Interface, Network File Access and Network Process-to-Process Communications, User Portability Extensions, Corrections and Extensions, Protection and Control Utilities and Batch System Utilities.

Portables Programmieren mit POSIX (1)

- Headerdateien definieren Funktions-Prototypen, Konstanten und Makros
- Maschinenabhängigkeiten verstecken durch konsequente Verwendung der POSIX-Headerdateien
- Beispiel: Prozess IDs:

älteres UNIX

```
short int pid;  
pid = getpid();
```

neueres UNIX

```
long int pid;  
pid = getpid();
```

POSIX

```
#include <sys/types.h>  
#include <unistd.h>  
  
pid_t pid;  
pid = getpid();
```

Portables Programmieren mit POSIX (2)

- Headerdateien definieren Funktions-Prototypen, Konstanten und Makros
- Maschinenabhängigkeiten verstecken durch konsequente Verwendung der POSIX-Headerdateien
- Beispiel: Manipulation der Signalmaske:

UNIX

```
#include <signal.h>

int mask;

mask = 0;
mask |= 1 << (SIGALRM-1)
```

POSIX

```
#include <signal.h>

sigset_t mask;

sigemptyset(&mask);
sigaddset(&mask, SIGALRM);
```

POSIX 1003.1b - Echtzeiterweiterungen

Funktionsgruppen

- ① Prioritätengesteuertes Scheduling
- ② Echtzeit-Signale
- ③ Clocks und Timer
- ④ Semaphore
- ⑤ Messages
- ⑥ Memory Mapped Files und Shared Memory
- ⑦ Asynchrone Ein/Ausgabe
- ⑧ Synchrone Ein/Ausgabe
- ⑨ Memory Locking

POSIX 1003.1b - Scheduling

Scheduling Parameter

- Scheduling policy
 - ▶ SCED_FIFO: Prioritätenbasiert, Preemptiv
 - ▶ SCED_RR: wie SCED_FIFO, jedoch mit Quantum
 - ▶ SCED_OTHER: nicht näher spezifiziert
- Priorität
 - ▶ Prioritäten sind fix
 - ▶ Eine Ready-Liste pro Prioritätsstufe
 - ▶ Die älteste Task auf der höchsten Prioritätsstufe wird jeweils ausgeführt
 - ▶ Zustandsänderung „blockiert“ → „rechenwillig“: Task an das Ende der Ready-Liste (ihrer Prioritätsstufe)
 - ▶ Preemption: Task an den Anfang der Ready-Liste
- Funktionen:

<code>sched_setscheduler()</code>	Scheduling Policy / Parameter setzen
<code>sched_getscheduler()</code>	Scheduling Policy ermitteln
<code>sched_setparam()</code>	Scheduling Parameter setzen
<code>sched_getparam()</code>	Scheduling Parameter ermitteln
<code>sched_yield()</code>	Prozess ans Ende der Ready-Liste
<code>sched_get_priority_min()</code>	Minimale Priorität ermitteln
<code>sched_get_priority_max()</code>	Maximale Priorität ermitteln

POSIX 1003.1b - Signale

- Signale: vergleichbar mit Interrupts:

- ▶ Es gibt eine begrenzte Anzahl von Signalen
- ▶ Eine Task kann einen Handler für ein Signal installieren
- ▶ Eine Task kann Signale selektiv maskieren
- ▶ Falls kein Handler installiert ist, gibt es einen Default-Handler für jedes Signal
- ▶ Signale werden i. A. nicht gepuffert

- POSIX 1003.1b Erweiterung: Echtzeitsignale

- ▶ Rückwärtskompatibel mit POSIX 1003.1
- ▶ Mindestens 8 neue Signale (SIGRTMIN ... SIGRTMAX)
- ▶ EZ-Signale können gepuffert werden

- Funktionen:

<code>sigaction()</code>	Signal Handler setzen
<code>sigprocmask()</code>	Signale maskieren / freigeben
<code>sigxxxset()</code>	Signalmaske manipulieren (<code>xxx=add, del, fill</code>)
<code>sigsuspend()</code>	Blockieren, bis Signal eintritt
<code>sigwaitinfo()</code>	Warten auf Signal (ohne Handleraufruf)
<code>sigtimedwait()</code>	Dito, mit Timeout

POSIX 1003.1b - Clocks und Timer

• Clocks

- ▶ Verschiedene „Uhren“ (Taktquellen) möglich
- ▶ Mindestens eine Uhr (`CLOCK_REALTIME`)
- ▶ API erlaubt Auflösung bis zu einer Nanosekunde

• Timer

- ▶ Senden eines Signals an eine Task nach Ablauf eines vorgegebenen Zeitintervalls
- ▶ Zeitquelle wählbar (`CLOCK_REALTIME`, ..)
- ▶ Dynamisch zu erzeugen: Bis zu 32 Timer pro Prozess
- ▶ Absolute und Relative Verzögerung möglich
- ▶ Overrun-Erkennung

• Funktionen:

<code>clock_settime()</code>	Uhr setzen
<code>clock_gettime()</code>	Uhrzeit ermitteln
<code>clock_getres()</code>	Uhrenauflösung ermitteln
<code>timer_create()</code>	Timer erzeugen
<code>timer_settime()</code>	Ablaufzeit / -Intervall für Timer setzen
<code>timer_gettime()</code>	Zeit bis Timer-Ablauf ermitteln
<code>timer_getoverrun()</code>	Anzahl verpasster Timer-Zyklen ermitteln
<code>timer_delete()</code>	Timer löschen
<code>nanosleep()</code>	Hochauflösendes Blockieren

POSIX 1003.1b - Semaphore

- Zählsemaphore

- ▶ Queueing nach Prozesspriorität
- ▶ Name-based: Identifikation über (Datei-)Namen
- ▶ Memory-based: Identifikation über Speicheradresse

- Funktionen:

<code>sem_init()</code>	Memory-based Semaphore erzeugen
<code>sem_destroy()</code>	Memory-based Semaphore verwerten
<code>sem_open()</code>	Name-based Semaphore erzeugen
<code>sem_close()</code>	Name-based Semaphore verwerten
<code>sem_unlink()</code>	Name-based Semaphore löschen
<code>sem_wait()</code>	Semaphore dekrementieren („P-Operation“)
<code>sem_trywait()</code>	Nichtblockierendes dekrementieren
<code>sem_post()</code>	Semaphore inkrementieren („V-Operation“)
<code>sem_getvalue()</code>	Zählerstand ermitteln

POSIX 1003.1b - Message Queues

- Medium für ungerichtete Inter-Prozess-Kommunikation
 - ▶ Queueing nach Prozesspriorität
 - ▶ Name-based: Identifikation über (Datei-)Namen
- Funktionen:

<code>mq_open()</code>	Message Queue erzeugen
<code>mq_close()</code>	Message Queue verwerfen
<code>mq_unlink()</code>	Message Queue löschen
<code>mq_send()</code>	Nachricht senden
<code>mq_receive()</code>	Nachricht empfangen
<code>mq_getattr()</code>	Message Queue Attribute ermitteln
<code>mq_setattr()</code>	(Teilmenge der) Message Queue Attribute setzen
<code>mq_notify()</code>	Signal senden, wenn Nachricht eintrifft

POSIX 1003.1b - Mapped Files und Shared Memory

- Memory Mapped Files
 - ▶ Abbildung von Dateien in den Adressraum (mit `mmap()`)
- Shared Memory
 - ▶ Implementiert als Spezialfall von Memory Mapped Files
 - ▶ Identifier für Objekte sind Dateinamen
 - ▶ Trotzdem ohne Dateisystem implementierbar
(→ Restriktionen bei der Namenswahl um Portabilität zu sichern)
 - ▶ Die Dateioperationen `ftruncate()` und `close()` sind auch auf Shared Memory Objekte anwendbar
- Funktionen:

<code>mmap()</code>	Shared Memory oder Datei in Adressraum abbilden
<code>munmap()</code>	Adressraum-Abbildung verwerfen
<code>shm_open()</code>	File Descriptor für shared Memory erzeugen
<code>shm_close()</code>	File Descriptor für shared Memory verwerfen
<code>shm_unlink()</code>	Shared Memory Segment löschen
<code>ftruncate()</code>	Größe eines shared Memory Segments festlegen
<code>mprotect()</code>	Zugriffsattribute für shared Memory Segment ändern

POSIX 1003.1b - Asynchrone Ein/Ausgabe

- Daten Lesen/Schreiben „im Hintergrund“:
 - ▶ Standard (POSIX 1003.1) Funktionen `read()` und `write()` blockieren
 - ▶ Es ist nicht sichergestellt, dass der I/O Vorgang nach Rückkehr von `read()`, bzw. `write()` tatsächlich beendet ist
 - ▶ `aio_read()` und `aio_write()` blockieren nicht.
 - ▶ Jeder `aio_xxx()`-Auftrag beinhaltet:
 - ★ Datei-Position
 - ★ Signal, das bei Beendigung geschickt wird
 - ★ Priorität (relativ zu anderen `aio_xxx()`-Anforderungen)
- Funktionen:

<code>aio_read()</code>	Asynchron lesen
<code>aio_write()</code>	Asynchron schreiben
<code>aio_listio()</code>	Mehrere I/O-Anforderungen absetzen
<code>aio_cancel()</code>	Asynchronen I/O-Vorgang abbrechen
<code>aio_suspend()</code>	Warten auf Beendigung des asynchronen I/O
<code>aio_return()</code>	Ergebniswert von asynchronem I/O ermitteln
<code>aio_error()</code>	Fehlercode von asynchronem I/O ermitteln

POSIX 1003.1b - Synchrone Ein/Ausgabe

- Sicherstellen, dass Daten und Dateiinhalt übereinstimmen
 - ▶ Nach (POSIX 1003.1) stellen `read()` und `write()` nicht sicher, dass die Daten wirklich gelesen / geschrieben wurden (Buffer Cache!)
 - ▶ Entweder explizites Synchronisieren über Funktionsaufrufe, oder optionale Flags bei `open()` angeben:
 - ★ `O_DSYNC` - Nur Daten synchronisieren
 - ★ `O_SYNC` - Daten und Metadaten synchronisieren
 - ★ `O_RSYNC` - Auch Lesezugriffe synchronisieren
- Funktionen:

<code>fsync()</code>	Daten und Metadaten synchronisieren
<code>fdatasync()</code>	Nur Daten synchronisieren

POSIX 1003.1b - Memory Locking

- Temporäres Auslagern von Speicherseiten („paging“) führt bei Echtzeitprogrammen zu unvorhersagbaren Verzögerungen
 - ▶ „Festnageln“ der für die Echtzeitverarbeitung erforderlichen Ressourcen im physikalischen Speicher
- Funktionen:

<code>mlock()</code>	Adressbereich „festnageln“
<code>munlock()</code>	Adressbereich wieder auslagerbar machen
<code>mlockall()</code>	Alle Ressourcen eines Prozesses „festnageln“
<code>munlockall()</code>	Ressourcen eines Prozesses wieder auslagerbar machen

POSIX 1003.1c - Threads

Funktionsgruppen

- ① Thread Erzeugen/Löschen
- ② Thread Attribute
- ③ Thread Synchronisation
- ④ Signalbehandlung

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param);

main(int argc, char *argv[])
{
    pthread_t thread;
    int arg = atoi(argv[1]);

    pthread_create(&thread,
                  NULL,
                  (void*)my_thread,
                  (void*)&arg);
    .....

    pthread_join(thread, NULL);
    return 0;
}
```

```
void my_thread(int* pcount)
{
    int i;
    for(i = 0; i < *pcount; i++)
        do_whatever();
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param);

main(int argc, char *argv[])
{
    Thread erzeugen und starten
    pthread_t thread;
    int arg = atoi(argv[1]);

    pthread_create(&thread,
                  NULL,
                  (void*)my_thread,
                  (void*)&arg);
    .....

    pthread_join(thread, NULL);
    return 0;
}
```

```
void my_thread(int* pcount)
{
    int i;
    for(i = 0; i < *pcount; i++)
        do_whatever();
}
```

- Funktionen:

<code>pthread_create()</code>	Thread erzeugen
<code>pthread_cancel()</code>	Anderen Thread beenden
<code>pthread_exit()</code>	Eigenen Thread beenden
<code>pthread_join()</code>	Auf Beendigung von Thread warten
<code>pthread_setcancelstate()</code>	Thread cancel verhindern / zulassen
<code>pthread_setcanceltype()</code>	Cancel zu jeder Zeit verhindern / zulassen
<code>pthread_testcancel()</code>	Anstehende Cancels ausliefern
<code>pthread_equal()</code>	Vergleich zweier Thread Handles
<code>pthread_self()</code>	Eigenes Thread Handle ermitteln
<code>pthread_detach()</code>	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param);

main(int argc, char *argv[])
{
    Thread erzeugen
    und starten
    {
        pthread_t thread;
        int arg = atoi(argv[1]);
        pthread_create(&thread,
                      NULL,
                      (void*)my_thread,
                      (void*)&arg);
        .....
        pthread_join(thread, NULL);
        return 0;
    }

    Attribute, default
    falls NULL
    void my_thread(int* pcount)
    {
        int i;
        for(i = 0; i < *pcount; i++)
            do_whatever();
    }
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param);
main(int argc, char *argv[])
{
    Thread erzeugen
    und starten
    pthread_t thread;
    int arg = atoi(argv[1]);
    pthread_create(&thread,
                  NULL,
                  (void*)my_thread,
                  (void*)&arg);
    .....
    pthread_join(thread, NULL);
    return 0;
}

int my_thread(int* pcount)
{
    Attribute, default
    falls NULL
    Zeiger auf
    Thread-Code
    for(i = 0; i < *pcount; i++)
        do_whatever();
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param);

main(int argc, char *argv[])
{
    Thread erzeugen
    und starten
    {
        pthread_t thread;
        int arg = atoi(argv[1]);
        pthread_create(&thread,
                      NULL,
                      (void*)my_thread,
                      (void*)&arg);
        .....
        pthread_join(thread, NULL);
        return 0;
    }
}

Zeiger auf
Argument
(beliebiger Zeiger)
Attribute, default
falls NULL
Thread-Code
int i
for(i = 0; i < *pcount; i++)
    do_whatever();
}
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param);
main(int argc, char *argv[])
{
    Thread erzeugen
    und starten
    pthread_t thread;
    int arg = atoi(argv[1]);
    Attribute, default
    falls NULL
    Argument
    (beliebiger Zeiger)
    Warten auf Ende
    NULL
    des Thread
    pthread_create(&thread, NULL, my_thread, (void*)&arg);
    ....
    pthread_join(thread, NULL);
    return 0;
}
```

The code snippet demonstrates the creation and joining of a thread. It includes annotations explaining the purpose of various parts:

- Thread erzeugen und starten**: The creation and starting of a thread.
- Attribute, default**: Default thread attributes.
- falls NULL**: If the argument is NULL.
- Argument (beliebiger Zeiger)**: The argument passed to the thread function.
- Warten auf Ende des Thread**: Waiting for the end of the thread.
- pthread_create**: The function used to create the thread.
- pthread_join**: The function used to join the thread.
- my_thread**: The thread function definition.

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (1) *

- Thread Erzeugen/Löschen: Beispiel

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

void my_thread(int *param)
{
    Thread erzeugen
    und starten
    main(int argc, char *argv[])
    {
        pthread_t thread;
        int arg = atoi(argv[1]);
        pthread_create(&thread, NULL, my_thread, (void*)&arg);
        ....
        pthread_join(thread, NULL);
        return 0;
    }
}

Zeiger auf Speicher
für Returnwert
(beliebiger Zeiger)
falls NULL Argument
übergeben wird
(NULL falls
nicht erwünscht)

Attribute default
```

- Funktionen:

pthread_create()	Thread erzeugen
pthread_cancel()	Anderen Thread beenden
pthread_exit()	Eigenen Thread beenden
pthread_join()	Auf Beendigung von Thread warten
pthread_setcancelstate()	Thread cancel verhindern / zulassen
pthread_setcanceltype()	Cancel zu jeder Zeit verhindern / zulassen
pthread_testcancel()	Anstehende Cancels ausliefern
pthread_equal()	Vergleich zweier Thread Handles
pthread_self()	Eigenes Thread Handle ermitteln
pthread_detach()	Thread „abhangen“

POSIX 1003.1c Threads Erzeugen/Löschen (2) *

Cleanup Stack

- Liste von Routinen (dynamisch erstellt), die bei Terminierung eines Thread aufgerufen werden
- Funktionen:

<code>pthread_cleanup_push()</code>	Neuer Eintrag auf Cleanup Stack
<code>pthread_cleanup_pop()</code>	Obersten Eintrag vom Cleanup Stack entfernen

POSIX 1003.1c Thread Attribute (1)

Thread Attribute

- Das System gibt sinnvolle Default Attribute vor
- Detached/Joinable
 - ▶ Detached: Thread läuft eigenständig (weniger Ressourcen)
 - ▶ Joinable: Andere Threads können `pthread_join()` aufrufen
- Scheduling Parameter
 - ▶ Vgl. POSIX 1003.1b
- Vererbbarkeit von Scheduling-Parametern
- Scheduling scope
- Stackposition und -Größe
 - ▶ **Vorsicht:** Stackmanipulationen sind nicht portabel!

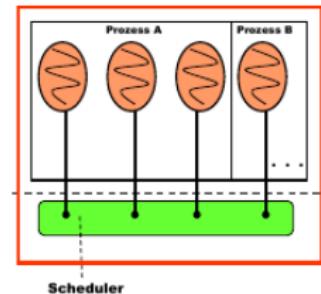
POSIX 1003.1c Thread Attribute (2)

• Scheduling Scope

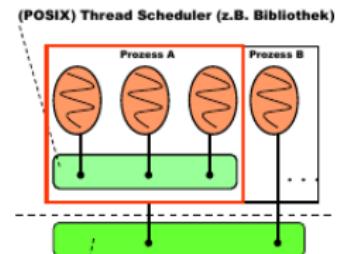
- ▶ *System scope*: Threads konkurrieren systemweit mit anderen Threads/Prozessen
 - „Threads sind Objekte des System-Schedulers“
- ▶ *Process scope*: Threads konkurrieren nur mit anderen Threads desselben Prozesses
 - ★ z.B. Thread-Bibliothek
- ▶ Vielfach nur Untermenge implementiert
 - ★ z.B. Linux: nur System Scope

• Funktionen:

<code>pthread_attr_init()</code>	Attribute default-initialisieren
<code>pthread_attr_destroy()</code>	Attributstruktur verwerfen
<code>pthread_attr_getxxx()</code>	Diverse Attribute in Struktur ermitteln
<code>pthread_attr_setxxx()</code>	Diverse Attribute in Struktur setzen
<code>pthread_getschedparam()</code>	Scheduling Parameter ermitteln
<code>pthread_setschedparam()</code>	Scheduling Parameter setzen



System Scope



Prozess (Task) Scheduler

POSIX 1003.1c Thread Synchronisation (1)

Mutex (Mutual Exclusion = Wechselseitiger Ausschluss)

- Attribut: Prioritätsprotokoll:
 - ▶ PTHREAD_PRIO_NONE: keines
 - ▶ PTHREAD_PRIO_PROTECT: priority ceiling
 - ▶ PTHREAD_PRIO_INHERIT: priority inheritance
- Zuteilung nach Priorität geordnet
- Funktionen:

<code>pthread_mutexattr_init()</code>	Mutex-Attribute default-initialisieren
<code>pthread_mutexattr_destroy()</code>	Mutex-Attribute verwerfen
<code>pthread_mutexattr_getxxx()</code>	Diverse Mutex-Attribute ermitteln
<code>pthread_mutexattr_setxxx()</code>	Diverse Mutex-Attribute setzen
<code>pthread_mutex_init()</code>	Mutex initialisieren
<code>pthread_mutex_destroy()</code>	Mutex verwerfen
<code>pthread_mutex_lock()</code>	Mutex acquirieren
<code>pthread_mutex_trylock()</code>	Mutex acquirieren ohne blockieren
<code>pthread_mutex_unlock()</code>	Mutex freigeben

`pthread_once()`-Funktion

- Sicherstellen, dass gegebene Funktion genau einmal ausgeführt wird
- z.B. Anlegen einer von mehreren Threads benötigten Ressource

POSIX 1003.1c Thread Synchronisation (2)

Condition Variablen

- ähnlich Zählsemaphore (signal- und wait-Operationen)
- signalisieren/erwarten eines Zustandes
- immer im Zusammenhang mit einem Mutex

Beispiel:

```
void inc_count(void)
{
    /* z.B. mehrfach-Aufruf
     * aus verschiedenen Threads
     */
    int i;
    for(i = 0; i < TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        ++count;
        if(count == WATCH_COUNT)
            pthread_cond_signal(&count_cond);
        pthread_mutex_unlock(&count_mutex);
    }
}
```

```
int count = 0;
pthread_mutex_t count_mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond =
    PTHREAD_COND_INITIALIZER;

void watch_count(void)
{
    pthread_mutex_lock(&count_mutex)
    while(count <= WATCH_COUNT) {
        pthread_cond_wait(&count_cond,
                          &count_mutex);
    }
    pthread_mutex_unlock(&count_mutex);
    printf("watch_count_reached\n");
}
```

POSIX 1003.1c Thread Synchronisation (3)

„Process shared“ Attribut:

- Bedeutung: Objekt ist über Prozessgrenzen hinweg nutzbar
- Anwendbar auf Mutexe & Condition Variablen
- Objekt (Mutex bzw. Condition Variable) muss in einem shared Memory Segment liegen
(ist nicht automatisch gegeben!)
- Nur in Verbindung mit „System“ Scheduling scope
- Alternativ: POSIX 1003.1b Semaphore (`sem_xxx()`, s.o.)

Condition Funktionen

<code>pthread_condattr_init()</code>	Condition Attribute default-initialisieren
<code>pthread_condattr_destroy()</code>	Condition Attribute verwerfen
<code>pthread_condattr_getpshared()</code>	Condition Attribut „process shared“ ermitteln
<code>pthread_condattr_setpshared()</code>	Condition Attribut „process shared“ setzen
<code>pthread_cond_init()</code>	Condition initialisieren
<code>pthread_cond_destroy()</code>	Condition verwerfenden
<code>pthread_cond_signal()</code>	Condition signalisieren
<code>pthread_cond_broadcast()</code>	Condition an alle signalisieren
<code>pthread_cond_wait()</code>	Auf Condition warten
<code>pthread_cond_timedwait()</code>	Auf Condition warten mit timeout

POSIX 1003.1c Thread Spezifische Daten (1)



Threadbezogene statische Daten

- z.B. errno

Beispiel:

<pre>static int Path; int OpenFile(void) { int x; x = open(FILENAME, O_RDONLY); if(x >= 0) { Path = x; return(OK); } else return(ERROR); } int ReadFile(int count) { return(read(Path, count)); }</pre>	<pre>static pthread_key_t Path; int OpenFile(void) { int *x = (int*)malloc(sizeof(int)); pthread_key_create(&Path, NULL); *x = open(FILENAME, O_RDONLY); if(*x >= 0) { pthread_setspecific(Path, (void*)x); return(OK); } else return(ERROR); } int ReadFile(int count) { int *x = (int*)pthread_getspecific(Path); return(read(*x, count)); }</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Funktionen

<code>pthread_key_create()</code>	Key erzeugen
<code>pthread_key_delete()</code>	Key verwerfen
<code>pthread_getspecific()</code>	Threadspezifisches Datum ermitteln
<code>pthread_setspecific()</code>	Threadspezifisches Datum setzen

POSIX 1003.1c Thread Spezifische Daten (1)



Threadbezogene statische Daten

- z.B. errno

Beispiel Nicht Thread-safe

```

static int Path;

int OpenFile(void)
{
    int x;
    x = open(FILENAME, O_RDONLY);
    if(x >= 0) {
        Path = x;
        return(OK);
    }
    else
        return(ERROR);
}

int ReadFile(int count)
{
    return(read(Path, count));
}

static pthread_key_t Path;

int OpenFile(void)
{
    int *x = (int*)malloc(sizeof(int));
    pthread_key_create(&Path, NULL);
    *x = open(FILENAME, O_RDONLY);
    if(*x >= 0) {
        pthread_setspecific(Path, (void*)x);
        return(OK);
    }
    else
        return(ERROR);
}

int ReadFile(int count)
{
    int *x = (int*)pthread_getspecific(Path);
    return(read(*x, count));
}

```

Funktionen

<code>pthread_key_create()</code>	Key erzeugen
<code>pthread_key_delete()</code>	Key verwerfen
<code>pthread_getspecific()</code>	Threadspezifisches Datum ermitteln
<code>pthread_setspecific()</code>	Threadspezifisches Datum setzen

POSIX 1003.1c Thread Spezifische Daten (1)



Threadbezogene statische Daten

- z.B. errno

Beispiel

Nicht Thread-safe

```
static int Path;
int OpenFile(void)
{
    int x;
    x = open(FILENAME, O_RDONLY);
    if(x >= 0) {
        Path = x;
        return(OK);
    }
    else
        return(ERROR);
}

int ReadFile(int count)
{
    return(read(Path, count));
}
```

Thread-safe

```
static pthread_key_t Path;

int OpenFile(void)
{
    int *x = (int*)malloc(sizeof(int));
    pthread_key_create(&Path, NULL);
    *x = open(FILENAME, O_RDONLY);
    if(*x >= 0) {
        pthread_setspecific(Path, (void*)x);
        return(OK);
    }
    else
        return(ERROR);
}

int ReadFile(int count)
{
    int *x = (int*)pthread_getspecific(Path);
    return(read(*x, count));
}
```

Funktionen

<code>pthread_key_create()</code>	Key erzeugen
<code>pthread_key_delete()</code>	Key verwerfen
<code>pthread_getspecific()</code>	Threadspezifisches Datum ermitteln
<code>pthread_setspecific()</code>	Threadspezifisches Datum setzen

POSIX 1003.1c Thread Signalbehandlung (1)



Auswahl des Thread, der ein Signal erhält

Signalart	Ursache	Ziel des Signals	Auswahl
Synchron	Exception (z.B) Division durch Null	Ein bestimmter Thread	Verursacher-Thread
Synchron	Anderer Thread ruft <code>pthread_kill()</code>	Ein bestimmter Thread	Ziel-Thread
Asynchron	Externer Prozess ruft <code>kill()</code>	Ganzer Prozess	Jeder Thread des Prozesses

POSIX 1003.1c Thread Signalbehandlung (1)



Auswahl des Thread, der ein Signal erhält

Signalart	Ursache	Ziel des Signals	Auswahl
Synchron	Exception (z.B) Division durch Null	Ein bestimmter Thread	Steuerbar durch individuelle (Per-Thread) Ziel-Thread Signalmaske
Synchron	Anderer Thread ruft <code>pthread_kill()</code>	Ein bestimmter Thread	
Asynchron	Externer Prozess ruft <code>kill()</code>	Ganzer Prozess	Jeder Thread des Prozesses

POSIX 1003.1c Thread Signalbehandlung (2)



Zustellung asynchroner Signale

- Threads haben individuelle Signalmasken
- Möglichkeit der Zuordnung bestimmter Signale an bestimmte Threads
- Falls ein Signal bei mehreren Threads nicht maskiert ist: Zustellung an (irgend)einen der Threads (!)

Signalbehandlung

- Signal Handler Tabelle ist gemeinsam für alle Threads
- Nur eine Untermenge der POSIX Funktionsaufrufe sind zulässig (insbesondere keine Pthread Synchronisationsfunktionen)
- Aber: POSIX 1003.1b Funktionen (`sem_xx`) sind zulässig

Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

5. Basiswissen Regelungstechnik



<https://www.stadtreporter.de/hannover/news/wirtschaft/robben-am-ball-ein-tierisch-sportlicher-geburtstag>

Inhalt

5. Basiswissen Regelungstechnik

5.1 Einführung

5.2 Grundlagen

5.3 Systemtheorie

5.4 PID-Regler

5.5 Stabilität

Was ist Regelungstechnik?

Wikipedia:

„Regelungstechnik ist ein Teilgebiet der Automatisierungstechnik und eine Ingenieurwissenschaft, die die in der Technik vorkommenden Regelungsvorgänge behandelt. Ein technischer Regelvorgang ist eine gezielte Beeinflussung von physikalischen, chemischen oder anderen Größen in technischen Systemen.“

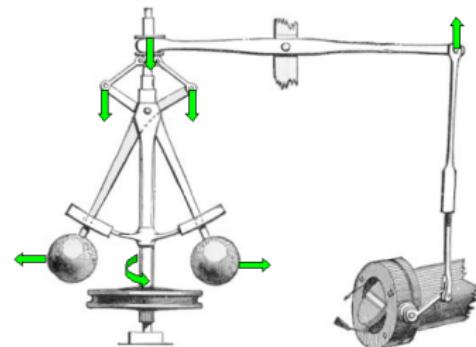
Beeinflussung von dynamischen Systemen

- Wie komme ich vom IST-Wert zum SOLL-Wert?
- Regelung kann mechanisch, elektrisch, biologisch, rechnergestützt erfolgen
- Sensoren, Aktoren, Regelungsalgorithmus

Wie alles begann ...

1788: James Watt soll eine Dampfmaschine reparieren

- Gewünscht: Gleichmäßige Geschwindigkeit
- Problem: Mehr Kohle → mehr Dampf → mehr Geschwindigkeit
- Lösung: „Zentrifugalregulator“
- Ventilöffnung abhängig von der Drehgeschwindigkeit
- Erster „Regler“
- Parameter über Experimente/„Probieren“ bestimmt
- (N.B.: Verbleibendes Problem: unsicherer Fehlerzustand: Keilriemen gerissen → Volldampf ...)

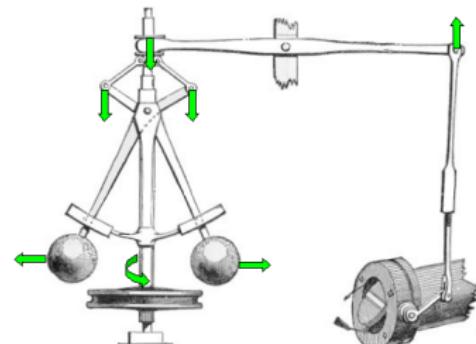


<https://commons.wikimedia.org/wiki/File:Fleihkraffregler.PNG>

Wie alles begann ...

1788: James Watt soll eine Dampfmaschine reparieren

- Gewünscht: Gleichmäßige Geschwindigkeit
- Problem: Mehr Kohle → mehr Dampf → mehr Geschwindigkeit
- Lösung: „Zentrifugalregulator“
- Ventilöffnung abhängig von der Drehgeschwindigkeit
- Erster „Regler“
- Parameter über Experimente/„Probieren“ bestimmt
- (N.B.: Verbleibendes Problem: unsicherer Fehlerzustand: Keilriemen gerissen → Volldampf ...)

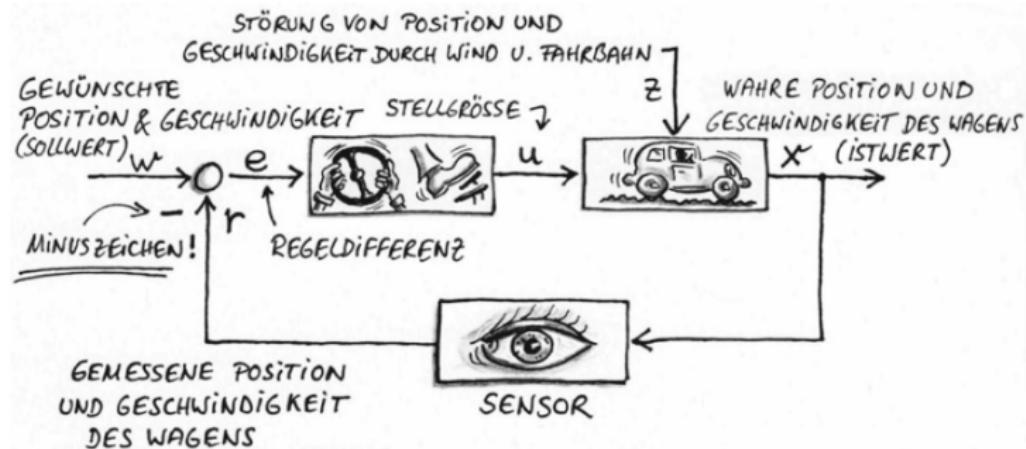


<https://commons.wikimedia.org/wiki/File:Fleihkraffregler.PNG>

Wo wird geregelt

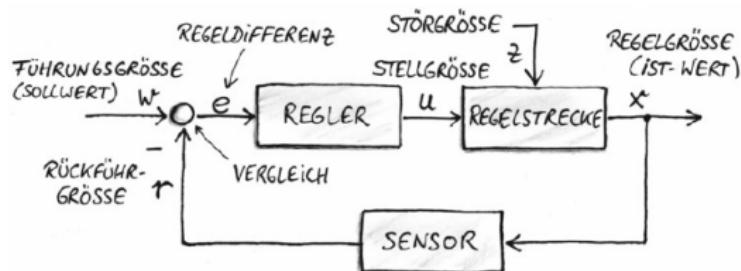
- Gebäudeautomation (Heizung, Licht)
- Medizintechnik (Herzschrittmacher)
- Prozessautomation (Chemische Prozesse)
- Robotersteuerungen
- Automotive (ABS, ESP, Motorsteuerung)
- Avionik (Starten, Landen, Autopilot, ...)
- Haushalt (Backofen, Waschmaschine, ...)
- Energienetze, Kraftwerke
- Politik (Steuern, Subventionen, ...)

Der Regelkreis



Aus Tietze, Romberg, Keine Panik vor Regelungstechnik, 2011

Begriffe



Aus Tieste, Romberg, Keine Panik vor Regelungstechnik, 2011

- Regler
- Regelkreis
- Regelgröße (Istwert) $x(t)$
- Rückführgröße (Messgr.) $r(t)$
- Führungsgröße (Sollwert) $w(t)$
- Regeldifferenz $e(t) = w(t) - r(t)$, Ziel: $e \rightarrow 0$
- Stellgröße $u(t)$
- Störgröße $z(t)$

Steuerung vs. Regelung und Stabilität

Ohne Rückkopplung, bzw. Regler \Rightarrow Steuerung

- Offener „Regelkreis“
- Bsp: Wecker, zeitgesteuertes Bewässerungssystem
- Einfach aber keine Reaktion auf Änderungen und Störgrößen!

Stabilität

- Stabile Systeme sind notwendig
- Systeme können träge sein
- „Aufschaukeln“ von Aktionen ist möglich
 - ▶ Ursache und Wirkung verstärken sich (Positive Rückkopplung¹)
 - ▶ Eingang w , oder Störung z enthalten Frequenzen, die der Eigenfrequenz des Systems entsprechen \Rightarrow Resonanz

¹ „Bei positiver Rückkopplung rechtzeitig in Deckung gehen!“ [Tiste, Romberg 2011]

Modellbildung

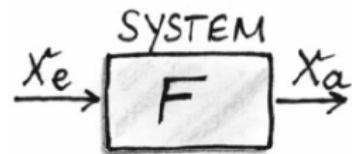
Wie Regelungen entwickeln?

Primär Aufgabe für Ingenieure, E-Techniker

- Aber auch ein Informatiker sollte die Basics kennen!

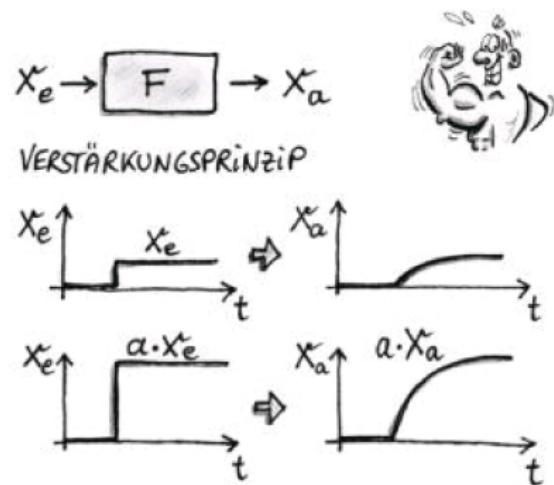
Am Anfang steht das Modell!

- Abstraktion, Blöcke, etc. pp.
- Eingangs- und Ausgangssignal
- Gewünscht: Lineare Systeme
„Mehr Eingang → mehr Ausgang“
- Zeitliche Verzögerung möglich
- Verhalten wird über Differentialgleichung beschrieben
- Auch mehrere Eingänge und oder Ausgänge möglich



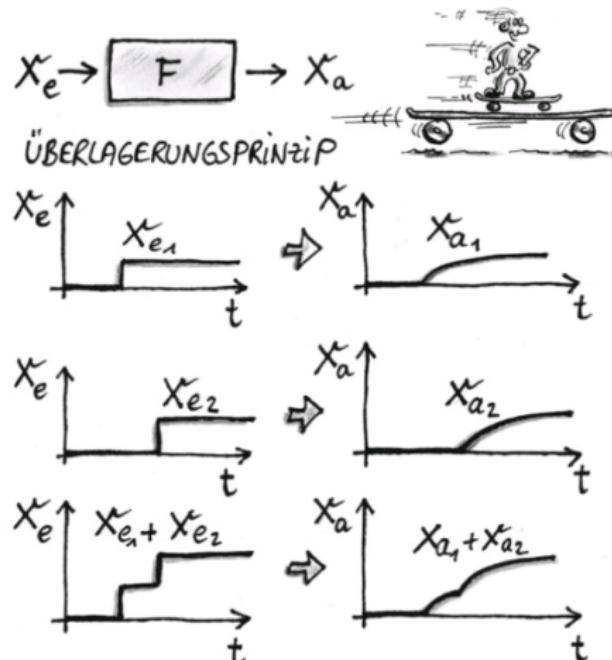
Aus Tieste, Romberg, Keine Panik vor Regelungstechnik, 2011

Verstärkungsprinzip (Folge der Linearität)



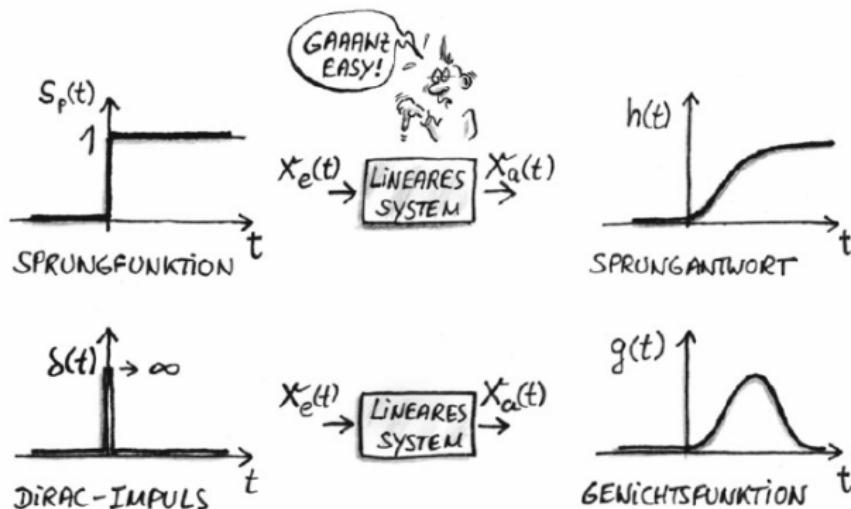
Aus Tietze, Romberg, Keine Panik vor Regelungstechnik, 2011

Überlagerungsprinzip (Folge der Linearität)



Aus Tiste, Romberg, Keine Panik vor Regelungstechnik, 2011

Übertragungsfunktion



Aus Tiste, Romberg, Keine Panik vor Regelungstechnik, 2011

Laplace-Transformation

Lösen von Differentialgleichungen ist schwierig

Math. Hilfsmittel: Laplace-Transformation

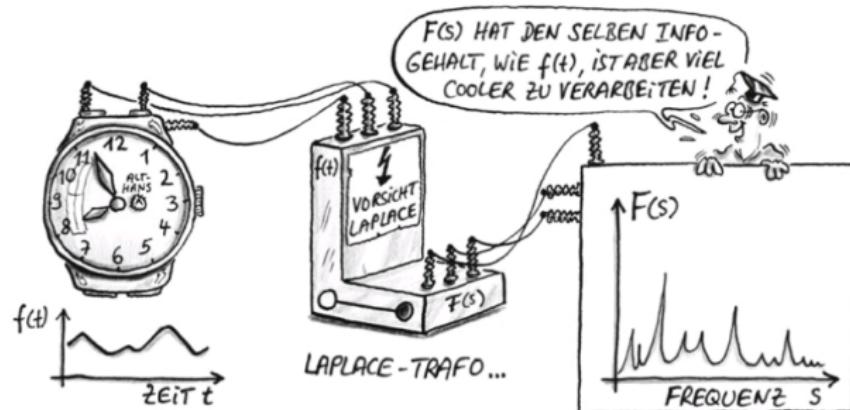
- Transformation einer Funktion $f(t)$ vom Zeitbereich in eine Funktion $F(s)$ im Bild- bzw. Frequenzbereich (s ist eine komplexe Zahl)
- Es gibt eine inverse Transformation
- DGLs werden zu “normalen Polynomen” transformiert
⇒ Im Bildbereich lösen, dann rücktransformieren
- Abbildung i.d.R. über Korrespondenztabellen

Oft ist eine Rücktransformation nicht notwendig

Wichtig: besondere Rechenregeln im Bildbereich!

Wird hier nicht weiter vertieft

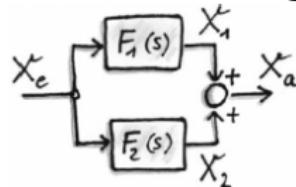
Laplace-Transformation (2)



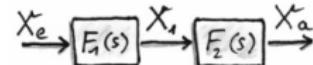
Aus Tietze, Romberg, Keine Panik vor Regelungstechnik, 2011

Rechenregeln für Blockschaltbilder

Parallelschaltung



Reihenschaltung



Aus Tietze, Romberg, Keine Panik vor Regelungstechnik, 2011

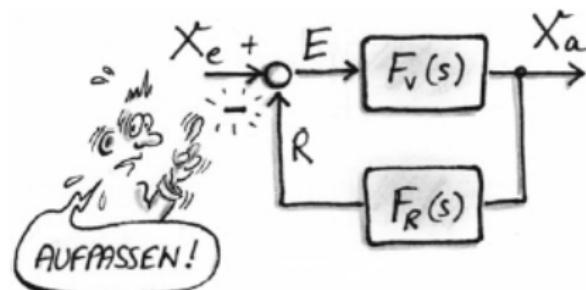
Aus Tietze, Romberg, Keine Panik vor Regelungstechnik, 2011

$$X_a(s) = (F_1(s)F_2(s))X_e(s)$$

$$X_a(s) = (F_1(s) + F_2(s))X_e(s)$$

Rechenregeln für Blockschaltbilder (2)

System mit Rückkopplung



Aus Tietze, Romberg, Keine Panik vor Regelungstechnik, 2011

$$E(s) = X_e(s) - R(s)$$

$$X_a(s) = F_v(s) \cdot E(s)$$

$$\frac{X_a(s)}{X_e(s)} = \frac{F_v(s)}{1 + F_v(s)F_r(s)}$$

$$X_a(s) = X_e(s) \cdot \frac{F_v(s)}{1 + F_v(s)F_r(s)}$$

Übertragungsfunktionen (P-System)

- „P“ für Proportional → Verstärkung
- DGL: $x_a(t) = K_p \cdot x_e(t) \Rightarrow$ Übertragungsfunktion: $F(s) = K_p$
- K_p = Verstärkungsfaktor
- Im Regler bspw. Verstärkung der Regeldifferenz
- Reiner P-Regler kann nie den Sollwert erreichen
 - ▶ Überschwingen
 - ▶ Problem der Trägheit/Verzögerungen → Instabilität

Übertragungsfunktionen (I-System)

- „I“ für Integral → Gedächtnis
- DGL: $x_a(t) = K_i \cdot \int x_e(t)dt \Rightarrow$ Übertragungsfunktion: $F(s) = \frac{K_i}{s}$
- K_i = Gewichtungsfaktor
- I-Regler
 - ▶ langsame, aber genaue Annäherung an Soll-Wert
 - ▶ Dämpfung von starken (sprunghaften) Änderungen

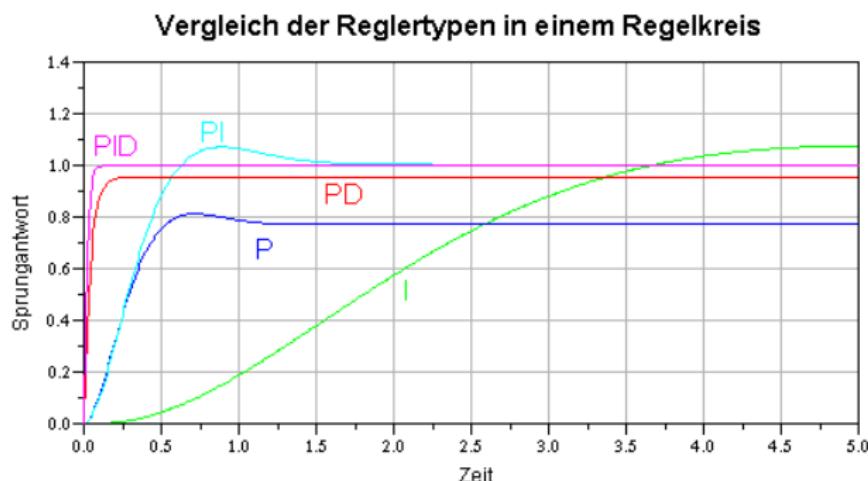
Übertragungsfunktionen (D-System)

- „D“ für Differenzierer → schnelle Reaktion auf Änderungen
- DGL: $x_a(t) = K_d \cdot \frac{d}{dt}x_e(t) \Rightarrow$ Übertragungsfunktion: $F(s) = K_s \cdot s$
- K_d = Anstiegskonstante
- Reaktion auf die Änderungsgeschwindigkeit
 - ▶ Je steiler die Änderung, desto größer der Wert
 - ▶ Kein eigenständiger Regler
 - ▶ Immer zusammen mit P- oder I-Anteil
 - ▶ Gut geeignet um „früh“ gegenzuregeln
- Oft mit zusätzlicher Verzögerung $\frac{1}{(1+T_p)}$

PID-Regler

- Kombinationen von P, I und D-Anteil
 - ▶ Parametrisierung über die Konstanten
 - ▶ Unterschiedliche Vor- und Nachteile
- PD-Regler
 - ▶ Schnelle Reaktion auf Ankündigungen von Veränderungen
 - ▶ Unruhig, bleibende Regelabweichung
- PI-Regler
 - ▶ Kombination schnelle Reaktion und exakte Ausregelung
 - ▶ Genau und mittelschnell
- PID-Regler
 - ▶ Universalregler
 - ▶ Genau und schnell

Vergleich Reglertypen



<http://rn-wissen.de/wiki/index.php/Datei:Reglervergleich.gif>

Stabilität

Regler sollen stabil sein

- Reaktionen auf Störungen
- Überreaktion auf Regelabweichungen

Instabilitäten treten auf, wenn

- sich Ursache und Wirkung verstärken (Pos. Rückkopplung)
- Eingang, Störung oder Ausgang mit der Eigenfrequenz des Systems übereinstimmen

Stabilität lässt sich aus der Pol-Nullstellenverteilung der Übertragungsfunktionen ablesen

- Polstellen mit negativen Realteil sind gut
- Polstellen mit positiven Realteil sind problematisch

Hurwitz-Kriterium:

- Abschätzung ob Polstellen negativen Realteil besitzen
- Bereichsabschätzung von Parametern, für die der Regler stabil ist

Einstellregeln für PID-Regler

Wie Parameter für PID-Regler ermitteln?

Methode nach Ziegler und Nichols

- Start mit P-Regler und kleinem K_p
- K_p erhöhen, bis ungedämpfte Schwingungen ausgeführt werden (Kritische Verstärkung K_{pkr})
- Schwingungsdauer T_{kr} bei krit. Verstärkung messen

Dann Regelparameter:

P-Regler $K_p = 0,5K_{pkr}$

PI-Regler $K_p = 0,45K_{pkr}$ $T_N :=$ Nachstellzeit
 $T_n = 0,85T_{kr}$ $T_V :=$ Vorstellzeit

PID-Regler $K_p = 0,6K_{pkr}$ $K_i = \frac{K_p}{T_N}$
 $T_n = 0,5T_{kr}$ $K_d = \frac{K_p}{T_V}$
 $T_v = 0,12T_{kr}$

Digitale Regler

Von der Differentialgleichung zur Differenzengleichung

- Zyklische Ausführung $\rightarrow \frac{dx}{dt} \rightarrow \frac{\Delta x}{\Delta t} \rightarrow \frac{(x_n - x_{n-1})}{\Delta t}$
- Wenn Δt nicht Konstant ist, dann $\Delta t \rightarrow (t_n - t_{n-1})$
- $\Delta t \rightarrow T_a$ Abtastzeit

$e = w - x;$ // Abweichung

$esum = esum + e;$ // Integration I-Anteil

$y = K_p * e +$ // P-Anteil

$K_i * T_a * esum +$ // I-Anteil

$K_d * (e - e_{alt})/T_a;$ // D-Anteil

$e_{alt} = e;$ // x_{n-1} fuer D-Anteil

Quellen

- <http://rn-wissen.de/wiki/index.php/Regelungstechnik>
- Tiste, Romberg, Keine Panik vor Regelungstechnik, 2011
<https://link.springer.com/book/10.1007%2F978-3-8348-81>
- PDV-Vorlesung Prof. Dr. Linn

Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

6. Echtzeitplanung



<https://school-time.co/>

Inhalt

6. Echtzeitplanung

- 6.1 Modellbildung
- 6.2 Planen durch Suchen
- 6.3 Planen nach Fristen
- 6.4 Planen nach Spielräumen
- 6.5 Planen nach monotonen Raten
- 6.6 Planen nach Zeitscheiben
- 6.7 Planen nach garantierten Zeitanteilen
- 6.8 Bewertung
- 6.9 Prozesssynchronisation
- 6.10 Planen und Synchronisation
- 6.11 WCET-Analyse

Modellbildung

- Ausgangspunkt: Modellierung eines realen Anwendungssystems
 - ▶ Zuordnung von Rechensystem-Prozessen zu technischen Prozessen
 - ▶ Eingangsgrößen und Zielgrößen
 - ▶ Informationsfluss zwischen Prozessen
 - ▶ Zeitbedingungen
 - ▶ Betriebsmittelbedarf

Prozesse in der Echtzeitplanung

- Prozess: Ausführung eines sequenziellen Programms auf einem Prozessor (s.o.)
- Ausführung **endet** nach endlich vielen Schritten
- Prozess entspricht endlicher Ausführungsfolge von Maschinenbefehlen
- Aufgabe der *Echtzeitplanung* (engl. *real-time scheduling*): Den Prozessen den Prozessor so zuzuteilen, dass alle von der Anwendung herrührenden Zeitbedingungen eingehalten werden
- Dabei ist der Prozess die kleinste einplanbare Einheit

Unterbrechbarkeit und Periodizität

- Bezogen auf den Startzeitpunkt spricht man von einem:
 - ▶ *periodischen* Prozess, falls er jeweils nach Ablauf einer festen Zeitspanne (der *Periode*) erneut gestartet werden soll,
 - ▶ *aperiodischen* oder *sporadischen* Prozess, sonst.
- Abhängig von der Anwendung ist zu unterscheiden zwischen Prozessen,
 - ▶ deren Ausführung zwischen Start und Beendigung nicht unterbrochen werden darf (nicht-unterbrechbar, engl. *non-preemptive*),
 - ▶ deren Ausführung i.d.R. nach jeder Anweisung unterbrochen werden kann (unterbrechbar, engl. *preemptive*)

Mehrfachinstanziierung

- Prozess als „Typ“
- Prozessinstanzen als konkrete Prozesse, die mit privaten Eingangsvariablen und Zielvariablen ausgestattet sind

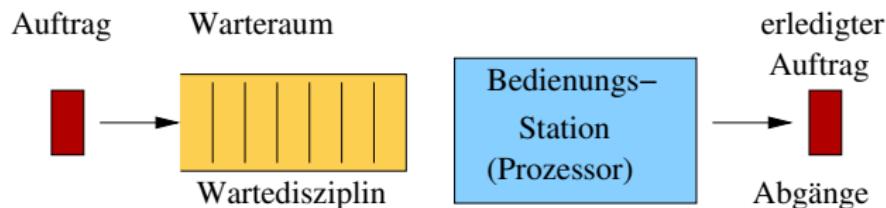
Prozessumschaltung

- Prozessumschaltung (engl. *context switch*): Änderung in der Zuordnung des gerade ausgeführten Prozesses
- Start eines Prozesses und Prozessumschaltung können vom Rechensystem oder vom technischen System ausgehen
- Dauer der Prozessumschaltung wird in den Planungsverfahren i.d.R. nicht berücksichtigt oder als Konstante den Ausführungszeiten zugeschlagen

Dauer der Prozessausführung

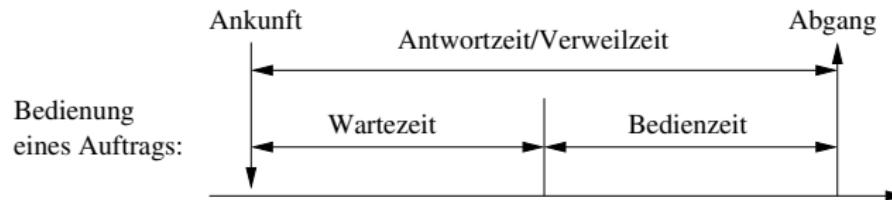
- abhängig von:
 - ▶ Leistungsfähigkeit des Prozessors
 - ▶ Peripherie des Prozessors
 - ▶ Eingangsdaten
 - ▶ Verfügbarkeit der Betriebsmittel
 - ▶ Verzögerung durch Erledigung wichtigerer Aufgaben
(z.B. Interrupt-Bearbeitung)

Begriffe aus der Bedientheorie (1)



- **Auftrag:** Einheit zur Bearbeitung (hier: Prozessinstanz)
- **Bedienzeit:** Zeitdauer für die reine Bearbeitung eines Auftrags durch die Bedienstation (hier: den Prozessor)
- **Wartedisziplin:** z.B. (einfache):
 - ▶ FCFS (*First-Come-First-Served*) oder FIFO (*First-In-First-Out*)
 - ▶ LIFO (*Last-In-First-Out*)
 - ▶ Random (zufällige Auswahl)
- Ankünfte und Bedienungszeiten werden bei Bedienungsmodellen häufig durch stochastische Prozesse modelliert
- komplexere Bedienmodelle können mehrere Bedienstationen (Multiprozessorsystem), mehrere Warteräume, mehrphasige Bedienung sowie die Rückführung teilweise bearbeiteter Aufträge enthalten.

Begriffe aus der Bedientheorie (2)



- **Antwortzeit:** Zeitdauer vom Eintreffen eines Auftrags bis zur Fertigstellung
Bei Dialogaufträgen Zeitdauer von der Eingabe eines Benutzers (z.B. Drücken der Return-Taste) bis zur Erzeugung einer zugehörigen Ausgabe (z.B. auf dem Bildschirm)
Auch **Verweilzeit** genannt
- **Wartezeit:** Antwortzeit - Bedienzeit
- **Durchsatz:** Anzahl erledigter Aufträge pro Zeiteinheit
- **Auslastung:** Anteil der Zeit im Zustand „belegt“
- **Fairness:** „Gerechte“ Behandlung aller Aufträge, z.B. alle rechenwilligen Prozesse haben gleichen Anteil an der zur Verfügung stehenden Rechenzeit

Relevante Größen

- Für eine Prozessausführung P_i *sporadischer* Prozesse sind folgende Zeitpunkte bzw. Zeitspannen relevant:
 - ▶ **Bereitzeit** (*ready time*) r_i : frühester Zeitpunkt, zu dem der Prozessor an P_i zugeteilt werden darf.
 - ▶ **Ausführungszeit / Bedienzeit** (*execution time*) Δe_i : reine Rechenzeit auf dem Prozessor; zur Planung wird meistens die für alle möglichen Eingangsdaten längste Zeitdauer zugrunde gelegt (schwierig zu bestimmen !!) (*WCET = Worst Case Execution Time*)
 - ▶ **Frist** (*deadline*) d_i : zu diesem Zeitpunkt muss die Ausführung von P_i abgeschlossen sein.
 - ▶ **Startzeit** (*starting time*) s_i : Beginn der Ausführung von P_i .
 - ▶ **Abschlusszeit / Abgangszeit** (*completion time*) c_i : Beendigung der Ausführung von P_i

Zur Veranschaulichung

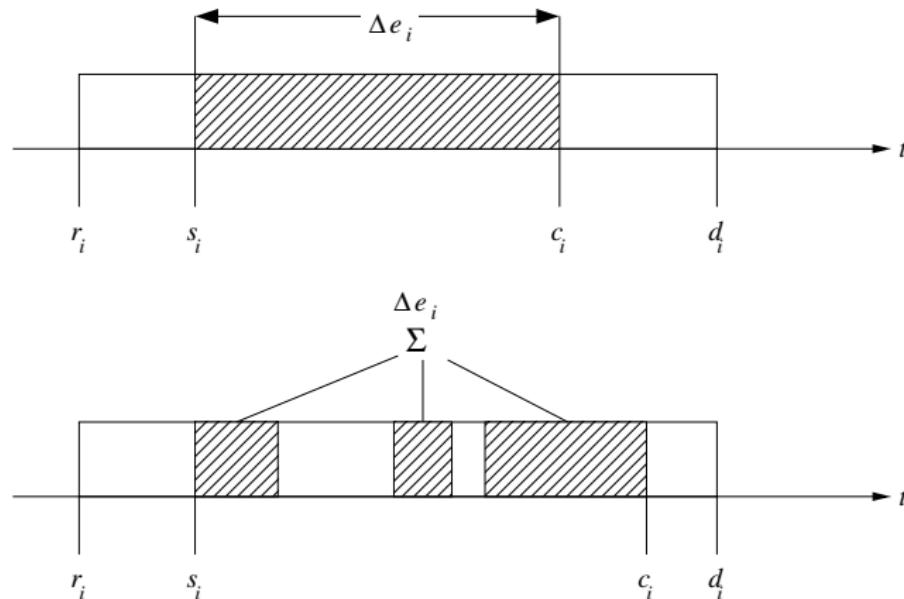


Abbildung: Zeitdiagramm einer ununterbrochenen (oben) und einer unterbrochenen (unten) Ausführung eines Prozesses P_i

Relevante Größen bei periodischen Prozessen

- Ein periodischer Prozess P_i wird mit einer bestimmten Frequenz f_i regelmäßig bereit gesetzt.
- Die Periode $\Delta p_i = \frac{1}{f_i}$ definiert den Rahmen seiner j -ten Ausführung P_i^j
- Ausgehend von der ersten Bereitzeit r_i^1 sind alle folgenden Bereitzeiten festgelegt durch

$$r_i^j = (j - 1) \cdot \Delta p_i + r_i^1 \quad \forall j \geq 1$$

- Das Ende einer Periode ist gleichzeitig eine Frist für die Prozessausführung:

$$d_i^j = j \cdot \Delta p_i + r_i^1 = r_i^{j+1} \quad \forall j \geq 1$$

Zur Veranschaulichung

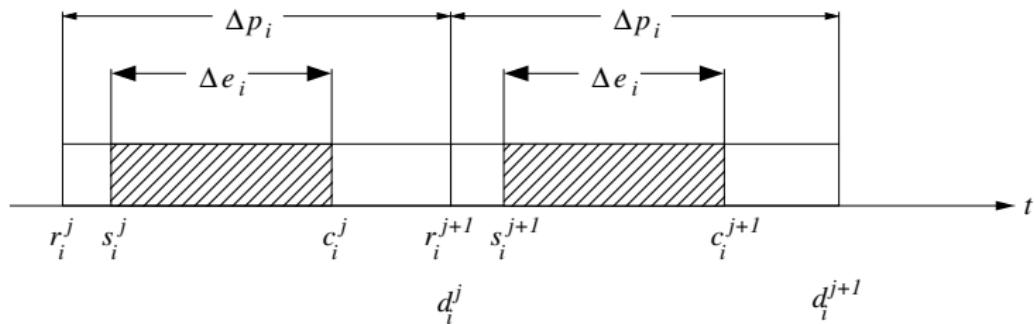


Abbildung: j -te und $(j + 1)$ -te ununterbrochene Ausführung von P_i^j und P_i^{j+1} .

Aufgaben der Echtzeitplanung

- Die Anwendung gibt Bereitzeiten, Ausführungszeiten, Fristen und Perioden vor
- Die Echtzeitplanung legt Start- und Abschlusszeiten und bei unterbrechbaren Prozessen die Folge von Unterbrechungen fest
- Im folgenden wird ausschließlich Prozessorzeit als Betriebsmittel betrachtet. Ein analoges Vorgehen ist auch für Speicher- und Kommunikations-Betriebsmittel (z.B. Busbelegung oder Netzwerkbelegung) möglich

Varianten der Echtzeitplanung

- *Statisches Scheduling*: Alle Daten für die Planung sind vorab bekannt, die Planung erfolgt durch eine *Offline-Analyse*
- *Dynamisches Scheduling*: Daten für die Planung fallen zur Laufzeit an und müssen zur Laufzeit verarbeitet werden
- *Explizite Planung*: Dem Rechensystem wird ein vollständiger Ausführungsplan (*Schedule*) übergeben, der zur Laufzeit befolgt wird (Umfang kann extrem groß werden)
- *Implizite Planung*: Dem Rechensystem werden nur die Planungsregeln übergeben

Phasen der Echtzeitplanung

- ① Einplanbarkeitsanalyse (*schedulability analysis, S-Test*)
- ② Planerstellung (*schedule construction*)
- ③ Prozessorzuteilung (*dispatching*)

Phase 1: Einplanbarkeitsanalyse

- Unter Berücksichtigung der Planungsgrößen wird geprüft, ob überhaupt ein brauchbarer Ausführungsplan existiert
- Prozess P_i charakterisiert durch $(r_i, \Delta e_i, f_i$ bzw. $\Delta p_i, d_i)$ mit f_i als Frequenz bei periodischen Prozessen, bzw. als obere Schranke bei sporadischen Prozessen
- Notwendige Bedingungen:
 - ▶ $\forall i : \Delta e_i < d_i - r_i < \frac{1}{f_i}$
 - ▶ $\sum f_i \Delta e_i \leq$ verfügbare Ressourcen

Brauchbarkeit eines Plans

- Ein Plan (*Schedule*) heißt **brauchbar** (*feasible*) für eine Prozessmenge $\{P_1, P_2, \dots, P_n\}$, falls bei gegebenen $(r_i, \Delta e_i, f_i, d_i)$ die Startzeit s_i und die Abschlusszeit c_i so gewählt sind, dass

- ▶ alle Zeitbedingungen für jeden einzelnen Prozess eingehalten werden:

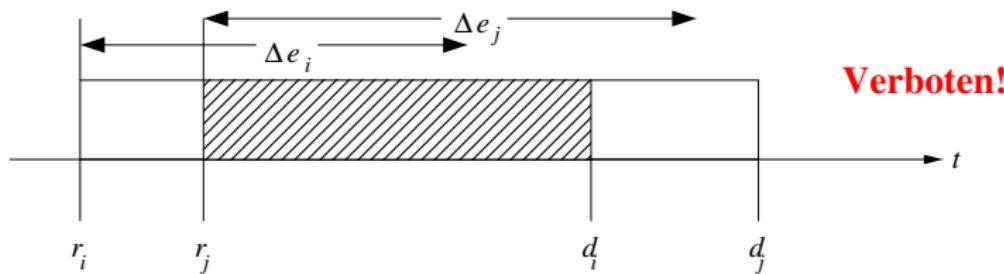
$$\forall i : \Delta e_i < d_i - r_i < \frac{1}{f_i}$$

- ▶ genügend Ressourcen vorhanden sind:

$$\sum f_i \Delta e_i \leq \text{verfügbare Prozessorzeit}$$

- ▶ keine überlappendenden Ausführungszeiten entstehen:

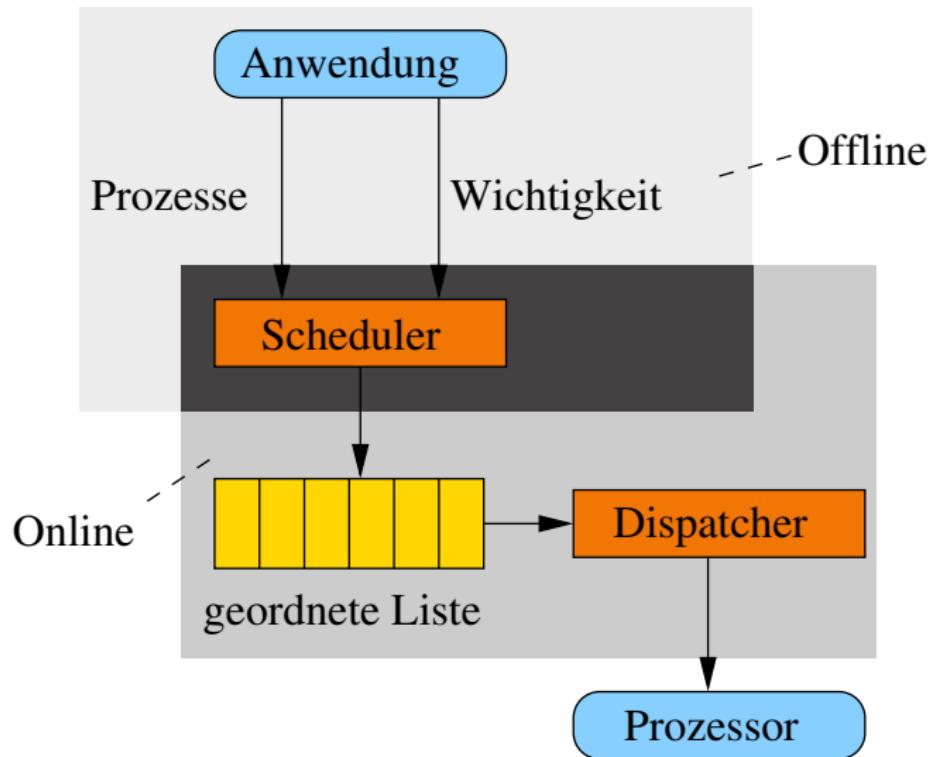
$$\forall i, j \text{ } d_i < d_j : d_i \leq d_j - \Delta e_j$$



Phase 2: Planerstellung

- Der Plan umfasst alle Informationen, die zur Laufzeit notwendig sind, um die eingeplanten Prozesse einem Prozessor zuzuordnen
- Eigenschaften statischer bzw. dynamischer Planung
- statisch:
 - + statische, nicht-unterbrechbare Prozesse
 - + Offline Einplanbarkeitsanalyse
 - + Einplanbarkeit bewiesen
 - + Plan z.B. in Form einer Prozesstabellen mit nach Startzeiten geordneten Einträgen
 - + Minimaler Aufwand zur Laufzeit
 - Bestimmung WCET
 - Komplexität des Verfahrens
- dynamisch:
 - + Mix aus periodischen und sporadischen Prozessen
 - + Prozesse unterbrechbar
 - + harte und weiche Zeitbedingungen
 - + Plan liefert Wichtigkeit und Prioritäten
 - + Online Analyse und Planerstellung
 - Aufwand zur Laufzeit
 - Einplanen des Zusatzaufwandes

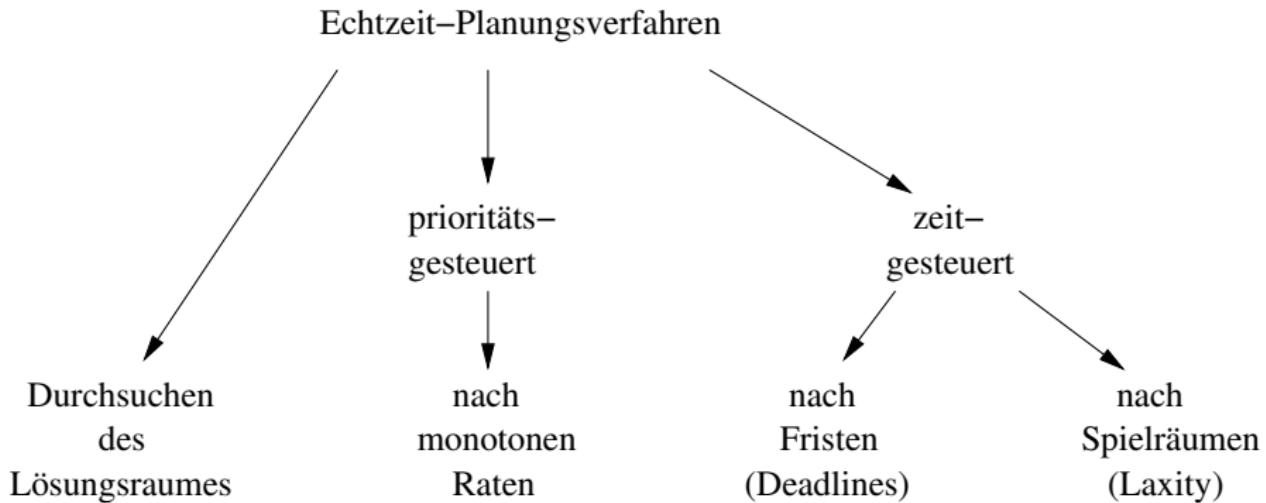
Phase 3: Prozessorzuteilung (Dispatching)



Gütekriterien für Planungsverfahren

- Ein *statisches* Planungsverfahren heißt *optimal*, falls es zu jeder beliebigen Prozessmenge einen Plan findet, sofern ein solcher existiert
- Ein *dynamisches* Planungsverfahren heißt *optimal*, falls es zu jeder beliebigen Prozessmenge einen brauchbaren Plan liefert, wenn ein statisches Verfahren in Kenntnis der gesamten Eingabedaten auch einen brauchbaren Plan geliefert hätte

Klassifizierung der Echtzeit-Planungsverfahren



Planen durch Suchen

- Durchsuchen des Lösungsraumes ohne Beachtung von Strategien oder Heuristiken stellt das „einfachste“ Planungsverfahren dar (*Branch-and-Bound-Verfahren*)
- (A) Es werden zunächst alle möglichen Kombinationen von nacheinander ausgeführten Prozessen ohne Berücksichtigung von Bereitzeiten und Fristen generiert
- Für n Prozesse existieren $n!$ Möglichkeiten
Darstellung als Baum der Tiefe n
 - Aufwand des Planungsverfahrens: $O(n!)$

Einbeziehen von Bereitzeiten und Fristen

- Nicht alle nach (A) erzeugten Pläne sind brauchbar, wenn Bereitzeiten und Fristen berücksichtigt werden
- (B) Heuristik: Einplanen eines Prozesses so früh wie möglich, d.h.

$$s - r \rightarrow \min$$

- ▶ Äste des Lösungsbaumes werden nicht weiter geführt, wenn sich eine Fristüberschreitung ergeben würde
- ▶ **Satz:** Wenn es brauchbare Pläne für eine Prozessmenge gibt, so wird ein solcher durch (B) gefunden

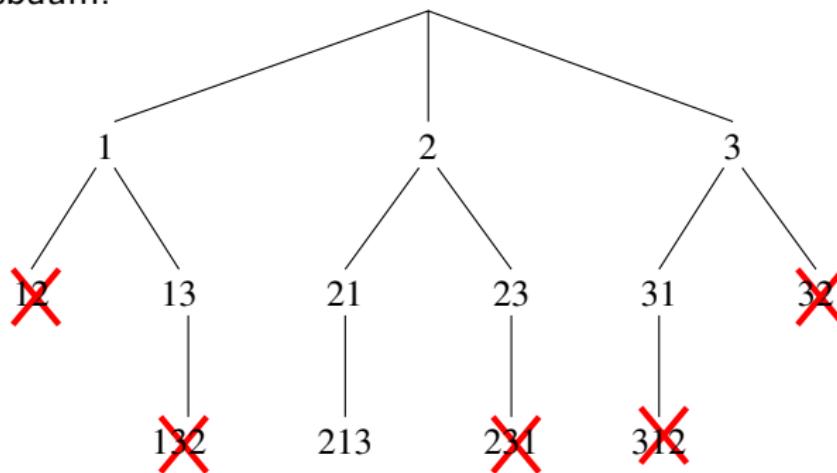
Beispiel

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i
1	13	19
2	3	8
3	6	24

Alle Aufträge seien zur Zeit Null bekannt

- Lösungsbaum:



Planen nach Fristen

- Strategie: *Earliest Deadline First* (EDF)
 - ▶ Der Prozessor wird demjenigen Prozess i zugewiesen, dessen Frist d_i den kleinsten Wert hat (am nächsten ist)
 - ▶ Wenn es keinen rechenbereiten Prozess gibt, bleibt der Prozessor untätig (d.h. „idle“)
- EDF ist eine der verbreitetsten zeitbasierten Strategien
- EDF ist anwendbar auf nicht-unterbrechbare und unterbrechbare Prozesse
- EDF ist anwendbar in statischen und dynamischen Planungsverfahren
- EDF ist bei nicht-unterbrechbaren Prozessen optimal, wenn alle Bereitzeiten gleich sind
- Falls EDF in diesem Fall keinen brauchbaren Plan liefert, gibt es keinen

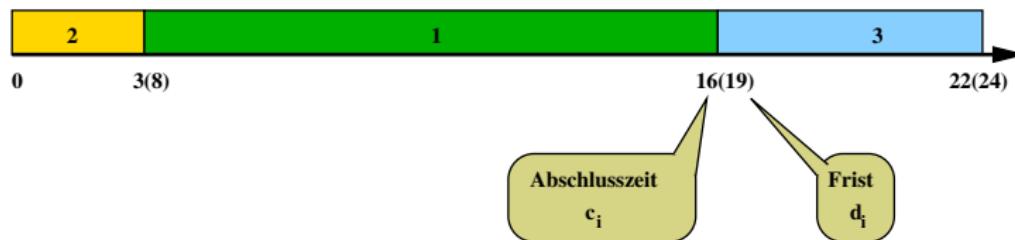
Beispiel (EDF nicht-unterbrechbar)

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i
1	13	19
2	3	8
3	6	24

Alle Aufträge seien zur Zeit Null bekannt

- Aus EDF resultierender Schedule:



EDF bei unterbrechbaren Prozessen

- Motivation: nicht alle Bereitzeiten sind identisch
- Ansatz: Zerlegen der Ausführungszeit eines Prozesses in mehrere Teilintervalle
- Variante EDF_p :
 - ▶ *Ready(t)* sei zum Zeitpunkt t die nach Fristen geordnete Liste aller noch nicht fertig abgearbeiteten Prozesse
 - ▶ Bestimme zum Zeitpunkt t den Prozess, der die nächste Deadline besitzt
 - ▶ Ordne den Prozessor zu für das Minimum aus
 - ★ Restbedienzeit des Prozesses
 - ★ kleinster Zeitpunkt, zu dem einer der bisher nicht betrachteten Prozesse bereit wird
 - ▶ Plane zu diesem Zeitpunkt neu
- EDF ist auch für unterbrechbare Prozesse optimal

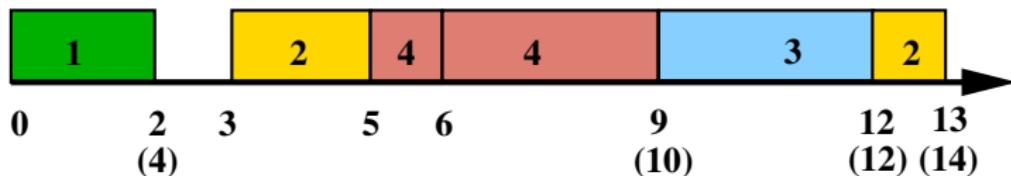
Beispiel (EDF unterbrechbar)

Gegeben: Prozessmenge mit 4 Prozessen

Prozess i	Bedienzeit Δe_i	Bereitzeit r_i	Frist d_i
1	2	0	4
2	3	3	14
3	3	6	12
4	4	5	10

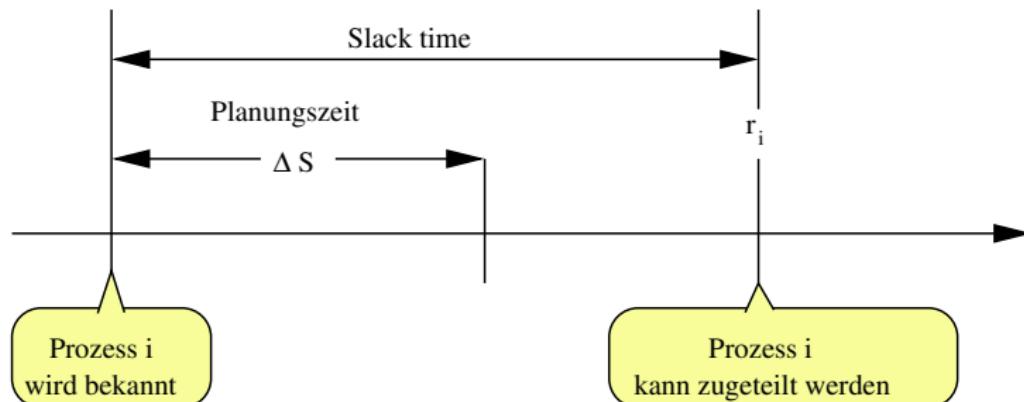
Alle Aufträge seien
zur Zeit Null
bekannt

- Aus EDF_p resultierender Schedule:



EDF bei dynamischem Scheduling(1)

- Voraussetzung:
 - ▶ Alle Prozessgrößen (Bereitzeit, Ausführungszeit, Frist) werden hinreichend früh bekannt, so dass der Prozess noch eingeplant werden kann
- Oft wird unterstellt, dass der Zeitaufwand für die Planung und Prozessorzuteilung vernachlässigt werden kann
- Bei Berücksichtigung der Planungszeit:



EDF bei dynamischem Scheduling(2)

- Vorgehensweise ansonsten wie bei EDF_p
- Dynamische EDF-Planung ist ebenfalls optimal
- Dynamische EDF-Planung ist sogar dann optimal, wenn die Ausführungszeiten nicht bekannt sind, d.h. wenn es überhaupt einen brauchbaren Plan gibt, wird auch einer durch EDF gefunden

EDF bei periodischen Prozessen

- Voraussetzung:
 - ▶ Perioden aller Prozesse bekannt
- Bei periodischen Prozessen $P = \{P_1, P_2, \dots, P_n\}$ bezeichnet die Auslastung U (Utilisation) die Summe der Zeitanteile, die die Ausführungszeit an der Periode des jeweiligen Prozesses hat:

$$U = \sum_{i \in P} \frac{\Delta e_i}{\Delta p_i} = \sum_{i \in P} \Delta e_i \cdot f_i$$

- Einplanbarkeit ist nur gegeben, wenn $0 \leq U \leq 1$
- Es gilt auch: Wenn $U \leq 1$, wird durch EDF *immer* ein brauchbarer Plan für periodische Prozesse gefunden

Zusammenfassung der Vorteile von EDF¹ (1)



- EDF ist anwendungsorientiert
- Erweiterungen, Modifikationen und Berücksichtigung zusätzlicher Prozesse möglich
- Einfacher Einplanbarkeitstest
- Antwortzeiten können dynamisch garantiert werden
- Überlast kann frühzeitig erkannt werden
- EDF-Planungsalgorithmus hat lineare Komplexität
- EDF-Planungsalgorithmus hat beschränkte Ausführungszeit
- EDF-Planungsalgorithmus ist leicht implementierbar
- EDF-Planungsalgorithmus erlaubt Behandlung periodischer, sporadischer und untereinander in Vorrang-Relation stehender Prozesse durch gemeinsames Verfahren
- EDF erreicht beste Prozessorauslastung unter Beachtung der Planbarkeit

¹nach W. Halang

Vorteile von EDF (2)

- EDF ist i.w. „non-preemptive“, d.h. Unterbrechung der Bedienung ist nur notwendig, wenn sich die Situation ändert (z.B. neuer Prozess kommt, blockierter Prozess wird rechenwillig, ...)
- Wenn ein neuer Prozess hinzukommt, bleibt die Reihenfolge unter den anderen unverändert
- EDF hat i.W. eine sequentielle Ausführung der Prozesse zur Folge
- EDF vermeidet Ressourcen-Konflikte und Deadlocks
- EDF erlaubt die Planung unterbrechbarer und nicht-unterbrechbarer Prozesse

Probleme bei EDF

- Um EDF auf Basis prioritätsbasierter Scheduler zu implementieren, muss das Betriebssystem das dynamische Ändern der Prioritätszuordnung ermöglichen (daher auch *dynamic priority scheduling*)
Einige Betriebssysteme (z.B. OSEK) unterstützen dies nicht
- Überlastsituationen ($U > 1$) führen unweigerlich zum Verpassen von Fristen. Alle Tasks („kritische“ wie „unkritische“) sind davon gleichermaßen betroffen.
Keine „graceful degradation“

Planen nach Spielräumen

- Das Planen nach Fristen ist für Mehrprozessorsysteme bei so früh wie möglicher Vergabe an rechenwillige nicht unterbrechbare Prozesse selbst bei gleichen Bereitzeiten nicht optimal!

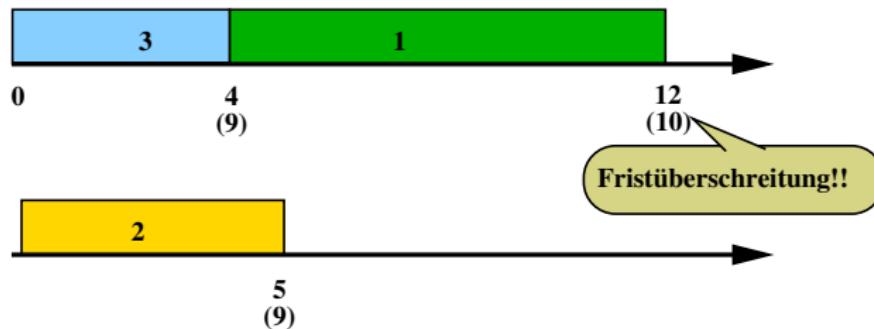
Beispiel: 2 Prozessoren

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i
1	8	10
2	5	9
3	4	9

Alle Aufträge seien zur Zeit Null bekannt

- Aus EDF resultierender Schedule:



- Das Planen nach Fristen berücksichtigt nicht, bis zu welchem Zeitpunkt man mit der Ausführung eines Auftrags spätestens begonnen haben muss

Least Laxity First (LLF)-Algorithmus

- Def.: *Spielraum* (engl. *Laxity*)

$$\Delta lax_i = (d_i - r_i) - \Delta e_i$$

- Notwendige Bedingung für die Einplanbarkeit:

$$r_i \leq s_i \leq r_i + \Delta lax_i$$

- Strategie: Least Laxity First (LLF):

- ▶ Gegeben sei eine Menge nicht unterbrechbarer Prozesse $P = \{P_1, P_2, \dots, P_n\}$ mit gleicher Bereitzeit für M Prozessoren
- ▶ LLF wählt denjenigen Prozess i aus, dessen Spielraum Δlax_i den kleinsten Wert hat (am kürzesten ist)

- LLF ist für Mehrprozessorsysteme optimal bei nicht-unterbrechbaren Prozessen mit gleicher Bereitzeit

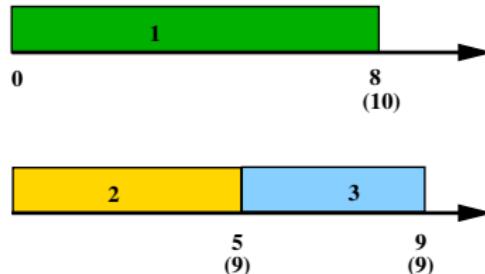
Beispiel: 2 Prozessoren

Gegeben: Prozessmenge mit 3 Prozessen

Prozess i	Bedienzeit Δe_i	Frist d_i	Spielraum lax_i
1	8	10	2
2	5	9	4
3	4	9	5

Alle Aufträge seien
zur Zeit Null
bekannt

- Aus LLF resultierender, brauchbarer Schedule:



- Das Problem, nicht unterbrechbare Prozesse brauchbar auf einem Mehrprozessorsystem einzuplanen, ist NP-vollständig. (Gilt sogar bei nur zwei Prozessoren und Gleichheit aller Bereitzeiten und Fristen)

LLF bei beliebigen Bereitzeiten

- Das Planen nach Spielräumen bei beliebigen Bereitzeiten ist nicht optimal!
- Genauer:
Satz: Kein Planungsverfahren für mindestens 2 Prozessoren kann optimal sein, ohne die Bereitzeiten der einzuplanenden Prozesse zu kennen

Planen nach monotonen Raten

Vorbemerkung

- Planen nach monotonen Raten erzeugt keinen expliziten Plan, sondern es entsteht ein impliziter Plan durch **feste** Vergabe von Prioritäten an Prozesse:

$$prio : P \rightarrow \mathbb{Z}$$

- Daher auch *fixed priority scheduling*
- Die meisten Echtzeit-Betriebssysteme unterstützen prioritätsbasiertes, unterbrechendes Scheduling
- Daher sind die Voraussetzungen für die Anwendung unmittelbar gegeben

Rate Monotonic Scheduling (RMS)

- Es werden ausschließlich unterbrechbare, periodische Prozesse betrachtet
- Die *Rate* eines Prozesses i bezeichnet die Anzahl der Perioden in einem Betrachtungszeitraum bzw. die Frequenz bei unbegrenzter Zeit
- Strategie: Rate Monotonic Scheduling (RMS)
 - ▶ Entsprechend den Raten werden den Prozessen Prioritäten durch die Prioritätszuordnungsfunktion $rms(i)$ mit folgenden Eigenschaften zugeordnet:

$$rms(i) < rms(j) \iff \frac{1}{\Delta p_i} < \frac{1}{\Delta p_j}$$

- ▶ Im Weiteren wird angenommen, dass die Prozesse entsprechend ihrer Priorität nummeriert sind:

$$\forall i, j : i < j \iff rms(i) < rms(j)$$

Kritischer Zeitpunkt, kritisches Intervall

- Def.: *Kritischer Zeitpunkt* eines Prozesses i ist diejenige Bereitzeit r_i^k , die abhängig von den Bereitzeiten aller übrigen Prozesse die Abschlusszeit c_i^k maximal werden lässt. Entsprechend heißt $[r_i^k, c_i^k]$ *kritisches Intervall*
- Satz: Für einen Prozess $i \in P$ ist ein kritischer Zeitpunkt r_i^k immer dann gegeben, wenn die Bereitzeiten aller höher priorisierten Prozesse auf r_i^k fallen.
(Wenn alle Bereitzeiten höher priorisierter Prozesse mit der Bereitzeit r_i^k zusammenfallen, erhalten diese Prozesse – deren Perioden alle kürzer sind als die vom Prozess i – die maximale Rechenzeit im Intervall $[r_i^k, c_i^k]$)

Eigenschaften von RMS

- Das Planen nach monotonen Raten nimmt keine Rücksicht auf die „Wichtigkeit“ von Prozessen. Das widerspricht der intuitiven Neigung, den wichtigen Prozessen Vorrang zu geben
- Dennoch ist die Prioritätenvergabe nach monotonen Raten die beste Methode zur Erzeugung brauchbarer Pläne bei periodischen Prozessen

Beispiel

Gegeben: Prozessmenge mit 2 Prozessen

Prozess <i>i</i>	Bedienzeit Δe_i	Periode Δp_i	\Rightarrow krit. Intervall
1	3	10	Prio: [0, 3]; RMS: [0, 4]
2	1	4	Prio: [0, 4]; RMS: [0, 1]

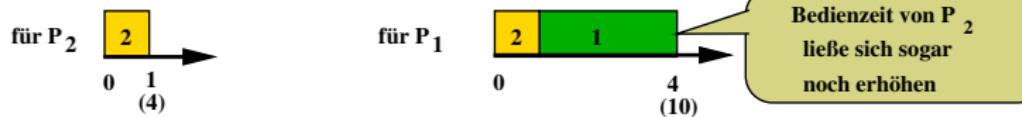
← „wichtiger“

- Aus vorgegebener Priorisierung resultierender Schedule:



- Aus RMS resultierender Schedule:

(wg. $\frac{1}{10} < \frac{1}{4}$ bekommt P_2 höhere Priorität)



Beispiel (Forts)

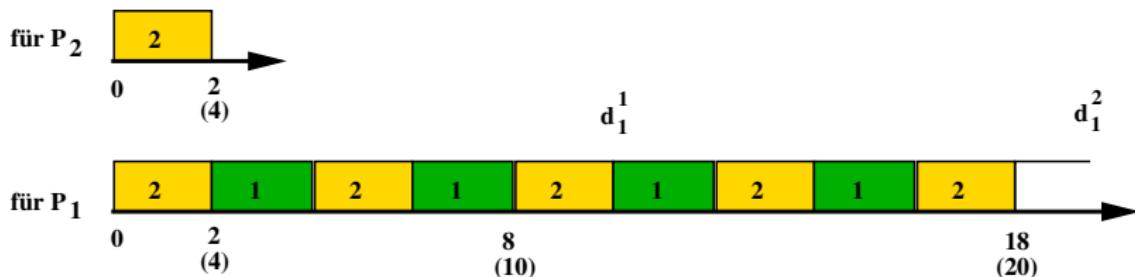
Gegeben: Prozessmenge mit 2 Prozessen

Prozess <i>i</i>	Bedienzeit Δe_i	Periode Δp_i	\Rightarrow krit. Intervall
1	4	10	[0, 8] RMS
2	2	4	[0, 2] RMS

← „wichtiger“

↑ Jeweils um 1 erhöhte Bedienzeiten

- Aus RMS resultierender Schedule:



Eigenschaften von RMS (2)

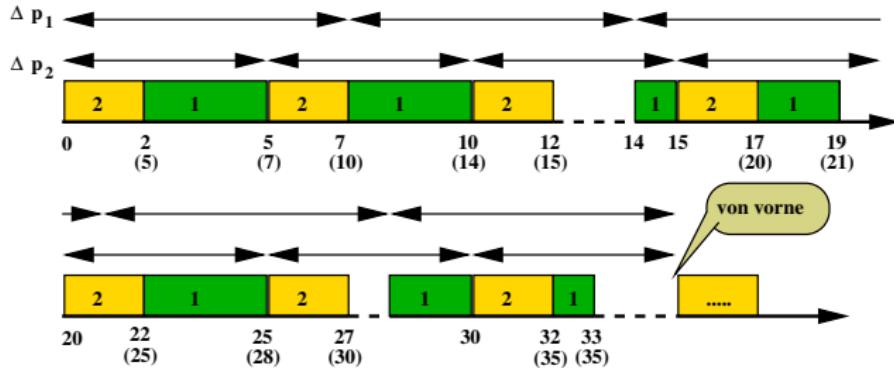
- Das Planen nach monotonen Raten ist nicht optimal
- Bevorzugt wird der zum aktuellen Zeitpunkt höchspriore Prozess, obwohl durch seine Verzögerung die näherliegenden Fristen anderer Prozesse noch gehalten werden könnten

Beispiel

Gegeben: Prozessmenge mit 2 Prozessen

Prozess <i>i</i>	Bedienzeit Δe_i	Periode Δp_i	\Rightarrow krit. Intervall
1	3	7	[0, 5] RMS
2	2	5	[0, 2] RMS

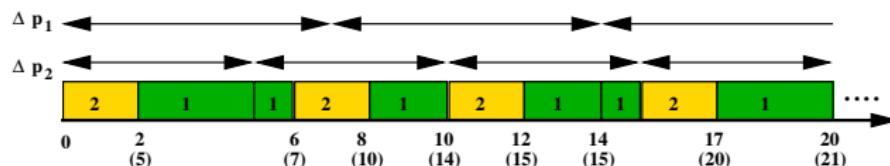
- brauchbarer Plan existiert, da krit. Intervalle in Periode passen
- Aus RMS resultierender Schedule²:



$$^2(\text{wg. } \frac{1}{7} < \frac{1}{5} \text{ bekommt } P_2 \text{ höhere Priorität})$$

Beispiel (Forts.)

- Die Auslastung U ist im Beispiel: $U = \frac{3}{7} + \frac{2}{5} = 0,82857\dots$
- Jede weitere Erhöhung der Bedienzeit führt jedoch zu einer Fristüberschreitung (z.B. für $\Delta e_1 = 4$)
- In diesem Fall ist $U = \frac{4}{7} + \frac{2}{5} = 0,9714\dots < 1$!
- EDF würde wegen $U \leq 1$ auch in diesem Fall einen Schedule finden:



Eigenschaften von RMS (3)

- Man kann den Prozessor nicht so gut auslasten wie im Fall, wenn man auch Fristen berücksichtigt
- Es gibt für RMS eine Grenze für die Prozessorauslastung U , für die gesichert ein Plan gefunden wird
- U_{min} bezeichnet diese Auslastung, für die RMS optimal ist, d.h. immer einen Plan findet, sofern einer existiert
- Für n Prozesse gilt:

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

- $n = 2 \Rightarrow U_{min} = 0.828427\dots$
- $n \rightarrow \infty \Rightarrow U_{min} = \ln(2) = 0,693\dots$
- Im obigen Beispiel war $U_{min} = 0.82857\dots$, und RMS hatte trotzdem noch einen brauchbaren Schedule gefunden

Eigenschaften von RMS (4)

- Von Überlastsituationen ($U > 1$) sind zuerst nur die niederprioren Tasks betroffen
- robuster als EDF

Planen nach Zeitscheiben

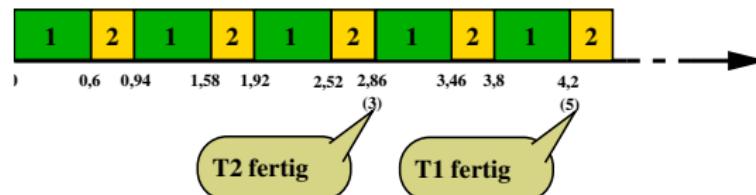
- Wie bei RMS werden ausschließlich unterbrechbare, periodische Prozesse betrachtet.
- Bei bekannter Prozessorauslastung durch einzelne Tasks können individuell angepasste Zeitscheiben als Planungsgrundlage verwendet werden (*Time-Slice-Scheduling*, TSS)
- Jeder Task wird eine Zeitscheibe zugeordnet. Das Verhältnis der Dauern der Zeitscheiben zueinander entspricht dem Verhältnis der Prozessorauslastungen durch die jeweilige Task.
- Zeitscheiben werden ohne weitere Priorisierung im RR-Verfahren bearbeitet.
- Mischung aus RR und periodischem FCFS mit Unterbrechung

Beispiel

Gegeben: Prozessmenge mit 2 Prozessen

Prozess <i>i</i>	Bedienzeit Δe_i	Periode Δp_i	Auslastung U_i	$\sum_i U_i = 94\% \leq 100\%$
1	3	5	$U_1 = \frac{3}{5} = 60\%$	
2	1	3	$U_2 = \frac{1}{3} = 33,3\% \approx 34\%$	

- Annahme: Zeiteinheiten in ms, Zeitscheiben auf Basis $1\% = 0,01$ ZE
- Resultierende Zeitscheiben: $TS1 = 0,6$; $TS2 = 0,34$



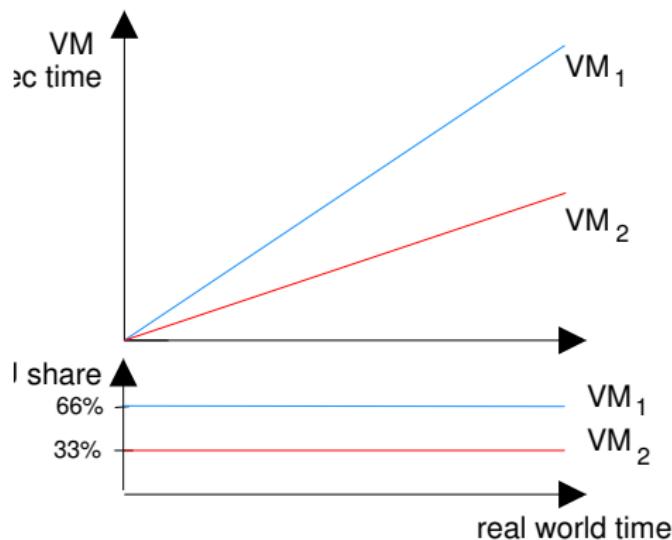
- Bei zu groben Zeitscheiben nicht optimal, da keine exakte Abbildung der Zeitverhältnisse der Tasks
- Bei beliebig feinen Zeitscheiben TSS → optimal, dann aber Effizienz → 0

Planen nach garantierten Zeitanteilen (1)

- *Guaranteed Percentage (GP)* , (auch „proportional share“ oder „Pfair“) Scheduling: eng verwandt mit TSS, Grundlage ist auch hier die bekannte Prozessorauslastung durch Tasks
- Task erhält einen prozentualen Anteil der Gesamtprozessorleistung, die dem Anteil ihrer Auslastung an der Gesamtauslastung entspricht.

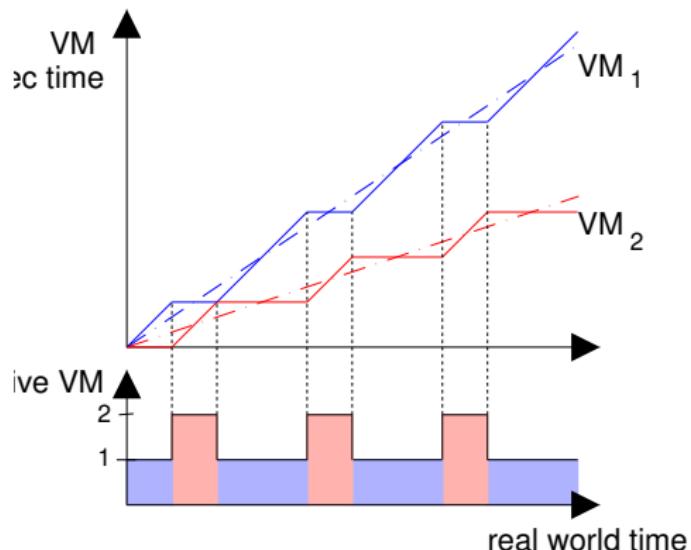
Planen nach garantierten Zeitanteilen (2)

- Mögliche Sichtweise: Jede Task erhält einen virtuellen Prozessor, der genau ihre Leistungsanforderungen abdeckt



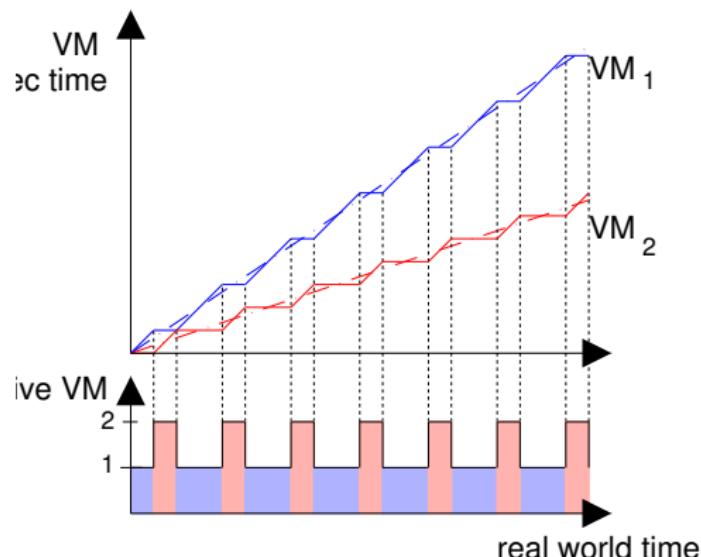
Planen nach garantierten Zeitanteilen (3)

- Verwendung von homogenen Zeitscheiben ggf. verbunden mit paralleler Ausführung in Hardware.



Planen nach garantierten Zeitanteilen (4)

- Erfordert für Optimalität noch feinere Zeitscheiben als TSS, daher Overhead noch größer



Vergleich der eingeführten Verfahren

Verfahren	Ein-Proz.-Systeme	Mehr-Proz.-Systeme	unterbrechbare Prozesse	nicht-unterbrechbare Prozesse
Planen durch Suchen	x	x		optimal, aber nicht von prakt. Relevanz. NP-vollständig
Planen nach Fristen (EDF)	x		optimal für spor.u. period. Prozesse in stat.u.dyn. Verfahren	optimal bei Prozessen mit gleichen Bereitzeiten
Planen nach Spielräumen (LLF)		x	optimal in stat. Verfahren; nicht optimal in dyn. Verfahren	NP-vollständig
Planen nach monotonen Raten (RMS)	x		nur für period. Prozesse; optimal bis Auslastungs-grenze U_{min} ; einfach realisierbar	
Planen nach Zeitscheiben (TSS)	x		nur period. Prozesse; einfach realisierbar, aber Overhead	

Bewertung

- Scheduling-„Theorie“ lässt Problembereiche außer Acht:
 - ▶ genaue Bestimmung der Ausführungszeit
 - ▶ Unabhängigkeit der Ausführungszeit von den Prozessen selbst
 - ★ mittelbare: indirekte Abhängigkeiten über Betriebssysteme, ...
 - ★ unmittelbare:
 - Vorrangbeziehung: Anstoß eines Prozesses durch anderen
 - Fluss der Daten von einem Prozess zu anderem
 - Wechselseitiger Ausschluss durch Nutzung von Betriebsmitteln, die z.B. durch Semaphore geschützt sind
 - ▶ Trennung in nur sporadische oder nur periodische Prozesse nicht praxisgerecht
- Die Realität kümmert sich daher kaum um eine korrekte Analyse, auch nicht in sicherheitskritischen Anwendungen (wenig beruhigend !)

Prozesssynchronisation

● Problemstellung

- ▶ Gegeben: Nebenläufige (konkurrente) Prozesse
- ▶ Ziel: Vermeidung ungewollter gegenseitiger Beeinflussung
- ▶ Ziel: Unterstützung gewollter Kooperation zwischen Prozessen:
 - ① Gemeinsame Benutzung von Betriebsmitteln (Sharing)
 - ② Übermittlung von Signalen
 - ③ Austausch von Nachrichten

→ Fazit: Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig

Beispiel (1)

- Gemeinsame Benutzung Speicherbereiches
(Hier: Datum „Kontostand“)
- Ungewollte gegenseitige Beeinflussung

Prozess 1

```
/* Gehaltsueberweisung */  
z = lies_kontostand();  
z = z + 1000;  
schreibe_kontostand(z);
```

Prozess 2

```
/* Dauerauftrag Miete */  
x = lies_kontostand();  
x = x - 800;  
schreibe_kontostand(x);
```

- (s.o.) Prozesse arbeiten „quasi-gleichzeitig“ → können an belieber Stelle unterbrochen und später fortgeführt werden

Beispiel (2)

Mögliche Ausführungsreihenfolge der Anweisungen

```
/* Gehaltsüberweisung */ /* Dauerauftrag Miete */
z = lies_kontostand();
x = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
x = x - 800;
schreibe_kontostand(x);
```

- Pech, die Gehaltsüberweisung ist „verloren gegangen“
- Bei anderen Reihenfolgen werden die beiden Berechnungen „richtig“ ausgeführt, oder es geht der Dauerauftrag verloren

Definitionen(1)

- Annahme: Prozesse mit Lese/Schreib-Operationen auf Betriebsmitteln
- **Def.:** Zwei nebenläufige Prozesse heißen
im Konflikt zueinander stehend oder überlappend, wenn es ein Betriebsmittel gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie unabhängig oder disjunkt
(N.B.: Die Scheduling Theorie, wie sie bisher vorgestellt wurde, geht von unabhängigen Prozessen aus)
- **Def.:** Folgen von Lese / Schreib-Operationen der verschiedenen Prozesse heißen zeitkritische Abläufe (engl. *race conditions*), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese / Schreib-Operationen sind

Definitionen (2)

- **Def.:** Verfahren zum wechselseitigen Ausschluss (engl. *mutual exclusion*): Ein Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess auf ein Betriebsmittel zugreift
- **Def.:** Kritischer Abschnitt oder kritischer Bereich (engl. *critical section* oder *critical region*): der Teil eines Programms, in dem auf gemeinsam benutzte Betriebsmittel (z.B. Datenbereiche) zugegriffen wird
- Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe
- Kritische Abschnitte realisieren sogenannte komplexe unteilbare oder atomare Operationen

(Nicht-)atomare Operationen (1)

Nicht atomar

```
/* Gehaltsueberweisung */  
z = lies_kontostand();  
z = z + 1000;  
schreibe_kontostand(z);
```

Jetzt atomar??

```
/* Gehaltsueberweisung 2.0 */  
int kontostand;  
  
kontostand += 1000;
```

- Auch Operationen, die in Hochsprache (hier: C) atomar zu sein scheinen, sind es auf Maschinenebene u.U. nicht!
- Architekturabhängig

⇒ Maschinencode:

```
lw r2 , kontostand  
addu r3 , r2 , #1000  
sw r3 , kontostand
```

→ 3 Instruktionen: **unterbrechbar!!**

(Nicht-)atomare Operationen (2)

- Sogar einzelne Maschinenbefehle können u.U. nicht-atomar sein
- Beispiel

```
la r2 ,0x1001  
sw 0x12345678 , kontostand
```

0x1000:		0x12	0x34	0x56
	0x78			
0x1004:				

- Zugriff auf ungerade Adresse erfordert mehrere Buszyklen
→ Unterbrechbar!

„Guter“ Algorithmus zum wechselseitigen Ausschluss

Hochschule RheinMain

- Anforderungen:

- ① Zu jedem Zeitpunkt darf sich nur ein Prozess in seinem kritischen Abschnitt befinden (Korrektheit, Basisforderung)
- ② Es dürfen keine Annahmen über die Ausführungsgeschwindigkeiten oder die Anzahl der Prozessoren gemacht werden
- ③ Kein Prozess, der sich nicht in seinem kritischen Abschnitt befindet, darf andere Prozesse blockieren (Fortschritt)
- ④ Alle Prozesse werden gleich behandelt (Fairness)
- ⑤ Kein Prozess darf unendlich lange warten müssen, bis er in seinen kritischen Abschnitt eintreten kann

Wechselseitiger Ausschluss mit aktivem Warten *

Generelles Vorgehen

```
enter_crit();      /* Prolog */
<statement>;
...
           /* critical section */
<statement>;
leave_crit();     /* Epilog */
```

- Prolog/Epilog-Paar
- Aktives Warten auf Eintritt in den kritischen Abschnitt (engl. *busy waiting*) (Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit)
- Alle Prozesse müssen sich an das Vorgehen halten (nicht forcierbar!)

Mögliche Lösungsansätze

- Ansätze zum wechselseitigen Ausschluss:
 - ① Sperren aller Unterbrechungen
 - ② Sperrvariablen
 - ③ Striktes Alternieren
 - ④ Peterson-Algorithmus
 - ⑤ Test-and-set-Instruktion
- Nur die letzten beiden sind brauchbar ...

Sperren aller Unterbrechungen

- Der Prozessor kann nur dann zu einem anderen Prozess wechseln, wenn eine Unterbrechung auftritt
- Einfachste Lösung:
 - ▶ Jeder Prozess sperrt vor Eintritt in seinen kritischen Abschnitt alle Unterbrechungen (*disable interrupts*)
 - ▶ Jeder Prozess lässt die Unterbrechungen am Ende seines kritischen Abschnitts wieder zu (*enable interrupts*)
- Korrektheitsforderung wird erfüllt
- Dennoch unbrauchbar für allgemeine Benutzerprozesse: keine Garantie, dass Unterbrechungen Jemals wieder zugelassen werden
- Wird aber häufig *innerhalb* von Betriebssystemen verwendet (*trusted code*)
- Generell nicht brauchbar bei Mehrprozessorsystemen

Sperrvariablen (1)

- Jedem kritischen Abschnitt wird eine Sperrvariable zugeordnet
- Wert 0 bedeutet „frei“, Wert 1 bedeutet „belegt“
- Jeder Prozess prüft die Sperrvariable vor Eintritt in den kritischen Abschnitt
- Ist sie 0, so setzt er sie auf 1 und tritt in den krit. Abschnitt ein
- Ist sie 1, so wartet er, bis sie den Wert 0 annimmt
- Am Ende seines kritischen Abschnitts setzt der Prozess den Wert der Sperrvariablen auf 0 zurück

Prozess 1

```
while(sperrvar) {}  
sperrvar = 1;  
/* kritischer Bereich */  
....  
sperrvar = 0;
```

Prozess 2

```
while(sperrvar) {}  
sperrvar = 1;  
/* kritischer Bereich */  
....  
sperrvar = 0;
```

Sperrvariablen (2)

Mögliche Ausführungsreihenfolge der Anweisungen

```
while(sperrvar) {}           while(sperrvar) {}  
sperrvar = 1;                 sperrvar = 1;  
/* kritischer Bereich */     /* kritischer Bereich */  
....                          ....  
sperrvar = 0;                 sperrvar = 0;
```

- Gleicher Fehler wie beim Konto-Beispiel: Zwischen Abfrage der Sperrvariablen und folgendem Setzen kann der Prozess unterbrochen werden
- Damit ist es möglich, dass sich beide Prozesse im kritischen Abschnitt befinden (Korrektheitsbedingung verletzt !!)

Striktes Alternieren

Prozess 1

```
while (1) {  
    while (dran != 1) { }  
    /* kritischer Bereich */  
    dran = 2;  
    /* unkritischer Ber. */  
}
```

Prozess 2

```
while (1) {  
    while (dran != 2) { }  
    /* kritischer Bereich */  
    dran = 1;  
    /* unkritischer Ber. */  
}
```

- Gemeinsame Variable „dran“ gibt an, welcher Prozess den kritischen Bereich betreten darf (Anfangswert z.B. 1)
- Prozesse wechseln sich ab: 2,1,2,1,...
- Lösung erfüllt Korrektheitsbedingung, aber
- Fortschrittsbedingung (3) kann verletzt sein, wenn ein Prozess wesentlich langsamer als der andere ist
- Fazit: keine ernsthafte Lösung.

Lösung von Peterson

- Historische Vorläufer:
 - ▶ Anfang der 60er Jahre viele Lösungsansätze, nur wenige erfüllten alle genannten Bedingungen
 - ▶ Erste korrekte Software-Lösung für 2 Prozesse:
Algorithmus von Dekker
- Neue Lösung zum wechselseitigen Ausschluss:
 - ▶ Lösung von Peterson (1981) (im folgenden betrachtet)
 - ▶ Basiert (ebenfalls) auf unteilbaren Speicheroperationen und aktivem Warten, ist aber einfacher
 - ▶ Prolog: `enter_crit()`, Epilog: `leave_crit()`
- Weitere Lösungen für mehr als zwei Prozesse von:
 - ▶ Dijkstra, Peterson, Knuth, Eisenberg/McGuire, Lamport (hier nicht weiter diskutiert)

Peterson-Algoritmus (1)

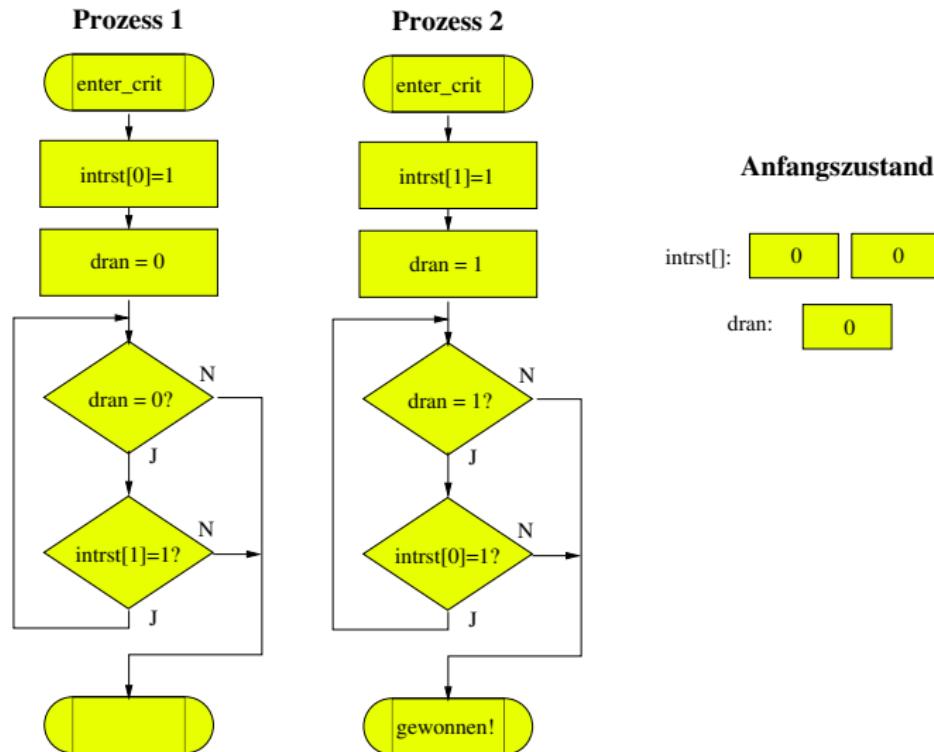
```
#define N      2           /* Anzahl der Prozesse      */

/* gemeinsame Variablen          */
volatile int dran;             /* Wer kommt dran?          */
volatile bool interested[N];   /* Wer will, anfangs alle false */

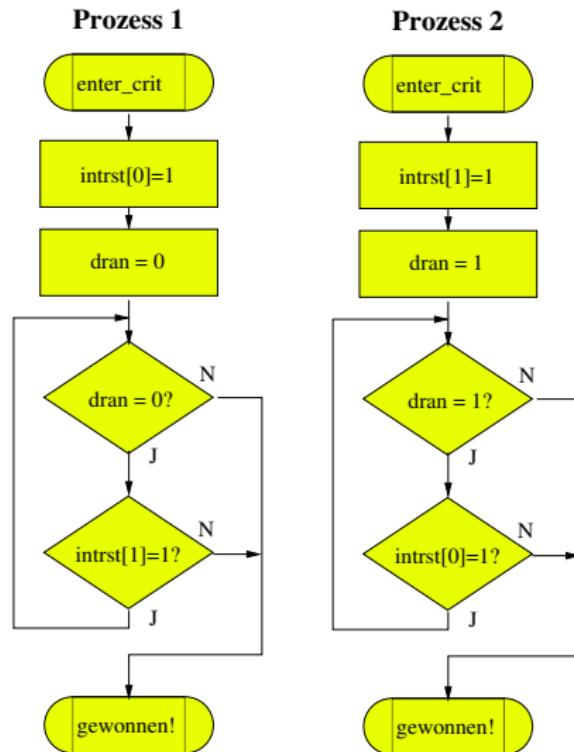
/* Prolog-Operation, vor Eintritt in den
   kritischen Bereich ausfuehren */
void enter_crit(int process) { /* process: wer tritt ein: 0,1 */
    int other;                /* Nummer des anderen Prozesses */
    other = 1 - process;
    interested[process] = true; /* zeige eigenes Interesse */
    dran = process;           /* setze Marke, unteilbar! */
    while(dran==process && interested[other]);
                                /* ev. Aktives Warten !!! */
}

/* Epilog-Operation, nach Austritt aus dem
   krit. Bereich ausfuehren */
void leave_crit(int process) { /* process: wer verlaesst: 0,1 */
    interested[process] = false; /* Verlasse krit. Bereich */
}
```

Peterson-Algorithmus (2)



Peterson-Algorithmus (3)

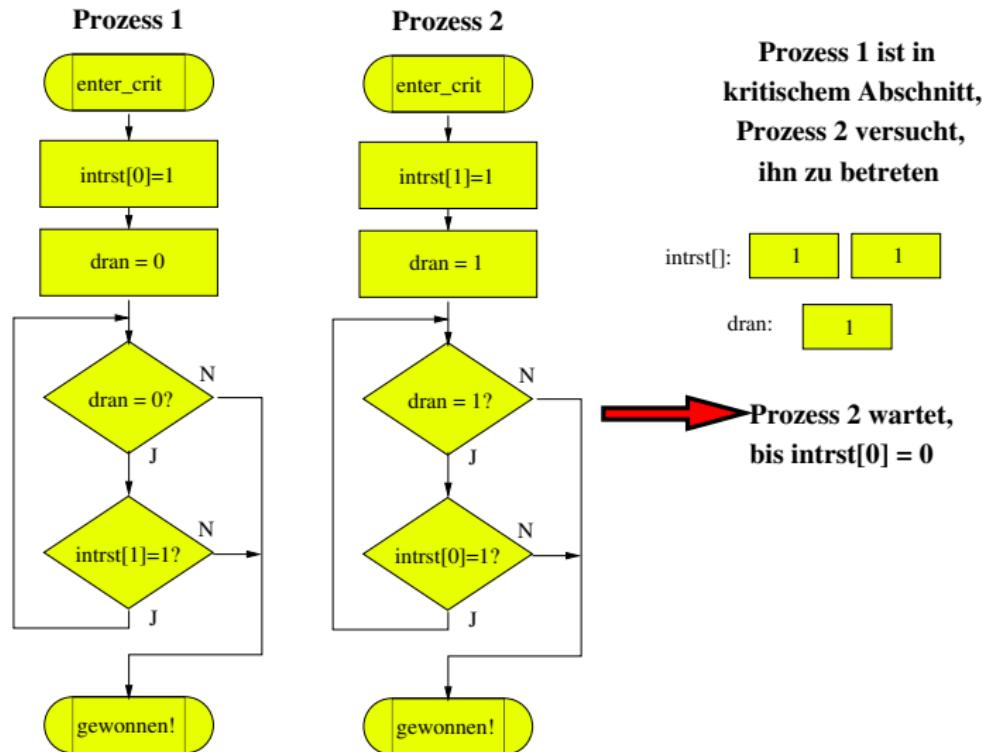


**Prozess 1 betritt
kritischen Abschnitt,
Prozess 2 nicht**

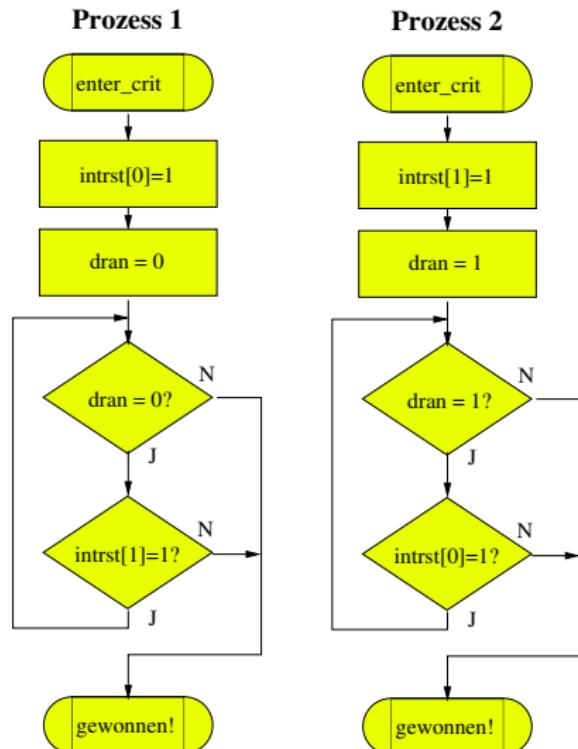
intrst[]: 

dran: 

Peterson-Algorithmus (4)



Peterson-Algorithmus (5)



Kritischer Fall:
**beide versuchen gleichzeitig,
 den kritischen Abschnitt
 zu betreten**

intrst[]:  

dran: 

Schreibzugriff auf „dran“ ist
 atomar. Nur einer von beiden
 kann „dran“ erfolgreich setzen.

Wer Erfolg hat, muss hier warten!

Test-and-Set-Befehl (1)

- Algorithmen, die nur auf atomaren Speicherzugriffen basieren sind
 - ▶ Komplex → fehlerträchtig
 - ▶ Auf eine feste Teilnehmerzahl beschränkt
- Zudem besteht bei einigen Verfahren die Gefahr des „Verhungerns“ (engl. *starvation*)
- Notwendigkeit einer effizienten Lösung
- Lösung durch Hardware-Unterstützung:
- Einführung eines Maschinenbefehls „Test-and-Set“: unteilbares Lesen einer Speicherzelle mit anschließendem Schreiben eines Wertes in die Zelle (Speicherbus wird zwischendurch **nicht** freigegeben)

Test-and-Set-Befehl (2)

- Jedem kritischen Abschnitt wird eine Sperrvariable „flag“ zugeordnet
- Wert 0 bedeutet „frei“, Wert 1 bedeutet „belegt“
- Auf Test-and-Set basierende Sperrvariablen mit aktivem Warten heißen auch *spin locks*
- Große Bedeutung in Multiprozessor-Betriebssystemen
- Typische Assembler-Routinen für Prolog und Epilog mit Test-and-Set-Instruktion „TAS“:

```

enter_crit:
    tas reg,flag      | kopiere flag in reg, setze flag = 1
    cmp reg,#0        | war flag vorher 0?
    jnz enter_crit   | nein -> Sperre war belegt -> warten
    ret               | ja -> darf einreten, zurueck zum Aufrufer

leave_crit:
    mov flag,#0       | loesche flag -> setze Sperre zurueck
    ret               | zurueck zum Aufrufer
  
```

- (Im Gegensatz zur Darstellung hier erfolgt die Realisierung i.d.R. über Makros, da Prozeduraufrufe zur Laufzeit zu großen Overhead darstellen)

Wechselseitiger Ausschluss mit passivem Warten

- Bisher:
 - ▶ Prolog-Operationen zum Betreten des kritischen Abschnitts führen zum aktiven Warten, bis der betreffende Prozess in den kritischen Abschnitt eintreten kann
 - ▶ Unerwünscht: Lediglich auf Multiprozessor-Systemen kann kurzzeitiges aktives Warten zur Vermeidung eines Prozesswechsels sinnvoll sein (spin locks)
- Ziel: Vermeidung von verschwendeter Prozessorzeit
- Vorgehensweise:
 - ▶ Prozesse blockieren, wenn sie nicht in ihren kritischen Abschnitt eintreten können
 - ▶ Ein solcher blockierter Prozess wird durch den aus seinem kritischen Abschnitt austretenden Prozess entblockiert

Einfache Primitive

- Einfachste Primitive: SLEEP() und WAKEUP():
 - ▶ SLEEP() blockiert den ausführenden Prozess, bis er von einem anderen Prozess geweckt wird
 - ▶ WAKEUP(process) weckt den Prozess process. Der ausführende Prozess wird dabei nie blockiert
- Häufig wird als Parameter von SLEEP und WAKEUP ein Ereignis (Speicheradresse einer beschreibenden Struktur) verwendet, um die Zuordnung treffen zu können
- Diese Primitive können auch der allgemeinen ereignisorientierten Kommunikation dienen

Semaphoren

- 1965 von Edsger W. Dijkstra eingeführt
- Supermarkt-Analogie:



- ▶ Kunde darf den Laden nur mit einem Einkaufswagen betreten
- ▶ Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
- ▶ Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird



- Semaphor besteht aus:
 - einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
 - einer **Warteschlange** für (passiv) wartende Prozesse

Semaphoren

- 1965 von Edsger W. Dijkstra eingeführt
- Supermarkt-Analogie:



- ▶ Kunde darf den Laden nur mit einem Einkaufswagen betreten
- ▶ Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
- ▶ Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird



- Semaphor besteht aus:
 - einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
 - einer **Warteschlange** für (passiv) wartende Prozesse

Operationen auf Semaphoren

• Initialisierung

- ▶ Zähler auf initialen Wert setzen
- ▶ „Anzahl der freien Einkaufswagen“



• Operation P(): Passier(-Wunsch)

- ▶ Zähler = 0 → Prozess in Warteschlange setzen, blockieren
- ▶ Zähler > 0: Prozess kann passieren
- ▶ In beiden Fällen wird der Zähler (ggf. nach dem Ende der Blockierung) erniedrigt
- ▶ P steht für „proberen“ (niederländisch für „testen“)

• Operation V(): Freigeben

- ▶ Zähler wird erhöht
- ▶ Falls es Prozesse in der Warteschlange gibt, wird einer de-blockiert (und erniedrigt den Zähler dann wieder, s.o.)
- ▶ V steht für „verhogen“ (niederländisch für „erhöhen“)

Operationen auf Semaphoren

- P() und V() sind atomare Operationen:
 - ▶ Das Überprüfen, Dekrementieren und Sich-Schlafen-Legen des Aufrufers bei P() sowie das inkrementieren und ggf. Wecken eines Prozesses bei V() ist jeweils eine Transaktion, die nur vollständig „in einem Zuge“ ausgeführt werden kann
- Kein Prozess wird bei der Ausführung der V()-Operation blockiert

Implementierung von Semaphoren

- I.d.R. als Systemaufrufe
- Intern Nutzung für Sich-Schlafen-Legen und Aufwecken
- Wesentlich ist die Unteilbarkeit von P() und V():
 - ▶ Auf Einprozessorsystemen: durch sperren aller Unterbrechungen während der Ausführung von P() und V(). Zulässig, da nur wenige Maschineninstuktionen zur Implementierung nötig sind.
 - ▶ Auf Multiprozessorsystemen: Jedem Semaphor wird eine Test-and-Set-Sperrvariable mit aktivem Warten vorgeschaltet. So kann stets höchstens ein Prozessor den Semaphor manipulieren
 - ▶ Man beachte: Unterscheide zwischen aktivem Warten auf den Zugang zum Semaphor, der einen kritischen Abschnitt schützt (einige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt)

Binärsemaphore und Zählsemaphore

- Semaphore, die nur die Werte 0 und 1 annehmen, heißen *binäre Semaphore*
- ansonsten heißen sie *Zählsemaphore*
- Binäre Semaphore dienen zur Realisierung des wechselseitigen Ausschlusses
- Ein mit $n > 1$ initialisierter Zählsemaphor kann zur Kontrolle der Benutzung eines in n Exemplaren vorhandenen Typs von sequentiell wiederverwendbaren Betriebsmitteln verwendet werden
- Semaphore sind weit verbreitet, innerhalb von Betriebssystemen und zur Programmierung nebenläufiger Anwendungen
- Programmierung mit Semaphoren ist oft fehleranfällig, wenn mehrere Semaphore benutzt werden müssen (Deadlocks)

Anwendung: Wechselseitiger Ausschluss

Wechselseitiger Ausschluss

```
semaphore sem = 1; /* Init. mit 1      */
...
P(sem);           /* Prolog          */
    <statement>; /* krit. Abschnitt */
...
    <statement>; /* krit. Abschnitt */
V(sem);           /* Epilog          */
...
...
```

- Der mit 1 initialisierte Semaphor `sem` wird von allen beteiligten Prozessen benutzt
- Jeder Prozess klammert seinen kritischen Abschnitt mit `P(sem)` zum Eintreten und `V(sem)` zum Verlassen

Anwendung: Betriebsmittelvergabe

- Es seien n Exemplare vom Betriebsmitteltyp bm vorhanden
- Jedes BM dieses Typs sei sequentiell benutzbar

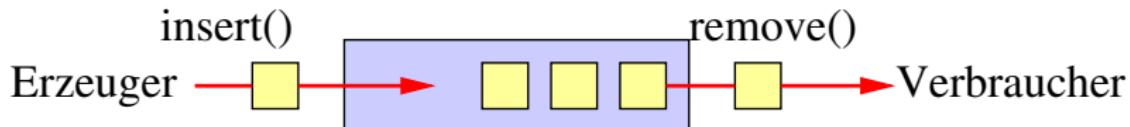
Betriebsmittelvergabe

```
semaphore bm = n;  
  
P(bm);           /* Beantragen */  
  
    Benutze das Betriebsmittel  
  
V(bm);           /* Freigeben */
```

- Dem BM-Typ bm wird ein Semaphor bm , zugeordnet
- Beantragen eines BM durch $P(bm)$, evtl. blockieren, bis ein BM verfügbar wird
- Nach der Benutzung freigeben des BM durch $V(bm)$
(Es kann damit von einem weiteren Prozess benutzt werden)

Beispiel: Erzeuger/Verbraucher-Problem

- Gegeben:
 - ▶ Datenobjekte (z.B. Nachrichten) fester Größe
 - ▶ Pufferspeicher von begrenzter Größe (n Objekte)
- Aufgabe
 - ▶ Verschiedene Prozesse legen Daten mit Hilfe einer Funktion `insert()` im Puffer ab („Erzeuger“), bzw. entnehmen sie mit einer Funktion `remove()` („Verbraucher“)
 - ▶ `remove()` soll warten, wenn keine Objekte im Puffer sind
 - ▶ `insert()` soll warten, wenn kein Platz mehr im Puffer ist



Lösung mit Semaphoren

Erzeuger/Verbraucher

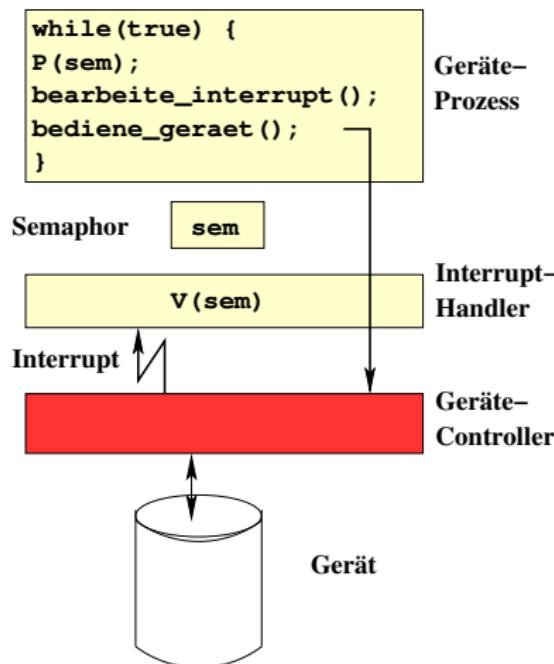
```
/* Initialisierung der Semaphore:
 * empty = Puffergroesse ,
 * full=0, mutex=1
 */

void insert(int item) {
    P(&empty);
    P(&mutex);
    /* item in Puffer stellen */
    V(&mutex);
    V(&full );
}

int remove(void) {
    P(&full );
    P(&mutex);
    /* vorderstes Item aus
    Puffer holen */
    V(&mutex);
    V(&empty );
}
```

- **full**:zählt belegte Einträge im Puffer, verhindert Entnahme aus leerem Puffer
- **empty**:verwaltet freie Plätze im Puffer, verhindert Einfügen in vollen Puffer
- **mutex**:schützt den kritischen Bereich vor gleichzeitigem Betreten (binär-Semaphor: nimmt nur Werte 1/0 an)

Anwendung: Verstecken von Interrupts



- Einem E/A-Gerät wird ein Geräteprozess und ein mit 0 initialisierter Semaphor zugeordnet
- Nach dem Start des Geräteprozesses führt dieser eine P-Operation auf dem Semaphor aus (und blockiert)
- Im Falle eines Interrupts des Gerätes führt der Interrupt-Handler eine V-Operation auf dem Semaphor aus
- Der Geräteprozess wird entblockiert (und dadurch rechenwillig) und kann das Gerät bedienen

P() und V() in Echtzeitsystemen

- Die meisten (alle?) (Echtzeit-)Betriebssysteme bieten Dienste zum Umgang mit Semaphoren
- In der Regel heißen diese aber *nicht* P() und V()
- Beispiele:
 - ▶ POSIX Threads Interface (pthreads): Mutex-Funktionen `phtread_mutex_xxx()` implementieren binäre Semaphoren
 - ▶ UNIX System V Release 4 (SVR4): Funktionen `semget()`, `semop()`, `semctl()` arbeiten auf *Mengen* von semaphoren
 - ▶ Windows NT: Zählsemaphore als Systemobjekte, Funktionen: `CreateSemaphore()`, `ReleaseSemaphore()`, `OpenSemaphore()`, `WaitForSingleObject()`

Planen und Synchronisation

Überblick

- Berücksichtigung von Abhängigkeiten zwischen Prozessen, die durch Synchronisation entstehen
- ① Problem der Prioritätsinversion (*Priority Inversion Problem*)
- ② Prioritätsvererbungsprotokoll (*Priority Inheritance Protocol*)
- ③ Prioritätsanhebungsprotokoll (*Priority Ceiling Protocol*)

Problem der Prioritätsinversion

- Problem der Prioritätsinversion tritt bei prioritätsbasiertem Scheduling unterbrechbarer Prozesse in Zusammenhang mit kritischen Abschnitten auf
- Szenario:
 - ▶ Prozess H habe hohe Priorität, Prozess L habe niedere Priorität.
Scheduling-Regel: wenn H rechenwillig ist, soll H ausgeführt werden
 - ▶ Es werde H rechenwillig, während L sich in seinem kritischen Abschnitt befindet. H wolle ebenfalls in seinen kritischen Abschnitt eintreten
 - ▶ Der höher priore Prozess H kann nicht weiter ausgeführt werden, da L den kritischen Abschnitt noch nicht freigegeben hat
 - ▶ Die Dauer der Verzögerung hängt nicht nur von L ab, sondern von allen Prozessen, die aufgrund ihrer Priorität vor H ausgeführt werden
- Es muss also der niedrigere priore Prozess ausgeführt werden!
Diese Situation heißt Prioritätsinversionsproblem.

Praktische Relevanz....

- Siehe Mike Jones: „What Happened on Mars?“
(www.cs.cmu.edu/afs/cs/user/raj/www/mars.html)

„The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. ... But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. ...

Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Pathfinder contained an "information bus", which you can think of as a shared memory area ... Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus .. When publishing its data, it would acquire a mutex ... If an interrupt caused the information bus thread to be scheduled while this mutex was held, ..., this would cause it to block on the mutex, waiting until the meteorological thread released the mutex ... The spacecraft also contained a communications task that ran with medium priority.

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running... After some time had passed, a watchdog timer would go off ... and initiate a total system reset.

This scenario is a classic case of priority inversion."

Prioritätsvererbungsprotokoll

- *Priority Inheritance Protocol*

- ▶ Jeder Prozess läuft mit seiner ihm zugewiesenen Priorität, es sei denn, er befindet sich in einem kritischen Abschnitt und blockiert einen höher prioren Prozess
- ▶ Wenn ein Prozess in einem kritischen Abschnitt einen höher prioren Prozess blockiert, erbt er die höchste Priorität aller durch ihn blockierten Prozesse
- ▶ Verlässt ein Prozess einen kritischen Abschnitt, innerhalb dessen er eine höhere Priorität ererbt hat, so wird er auf seine Priorität zum Zeitpunkt des Eintritts in den krit. Abschnitt zurückgesetzt

- Prioritätsvererbung ist transitiv und kann mehrfach innerhalb eines krit. Abschnitts erfolgen
- Das Prioritätsvererbungsprotokoll kann Deadlocks nicht ausschließen und löst daher nicht alle Probleme

Prioritätsanhebungsprotokoll (1)

• Priority Ceiling Protocol

- ▶ Jedem kritischen Abschnitt s wird eine Prioritätsgrenze $ceil(s)$ zugeordnet. Dieser Wert entspricht der höchsten Priorität aller Prozesse, die diesen kritischen Abschnitt jemals betreten wollen (**Achtung**: muss damit vorab bestimmbar sein!)
- ▶ Ein Prozess darf seinen kritischen Abschnitt nur betreten, wenn er von keinem anderen Prozess, der sich bereits in kritischen Abschnitten befindet, verzögert werden kann. Maßgeblich ist die aktuelle Prioritätsgrenze, die von allen übrigen Prozessen, die kritische Abschnitte gesperrt haben, bestimmt wird:

$$actceil(i) = \max\{ceil(s) | s \in \text{locked_crit_sect}(i)\}$$

- ▶ Entsprechend darf ein Prozess i einen kritischen Abschnitt nur betreten, wenn seine aktuelle Priorität (d.h. anfängliche oder ererbte) so groß ist, dass gilt:

$$actprio(i) > actceil(i)$$

Prioritätsanhebungsprotokoll (2)

- Protokoll:
 - ▶ Wenn ein Prozess einen krit. Abschnitt beansprucht und seine Priorität unter der Ceiling-Priorität liegt, so wird sie vorübergehend auf die Ceiling-Priorität angehoben
 - ▶ Wenn ein Prozess den krit. Abschnitt freigibt, fällt seine Priorität auf den ursprünglichen Wert zurück
- Das Prioritätsanhebungsprotokoll vermeidet Deadlocks
- Die Verzögerungszeit des jeweils höchstpriorisierten Prozesses ist durch den längsten krit. Abschnitt bestimmt, den er mit irgendeinem niedriger priorisierten Prozess gemeinsam hat

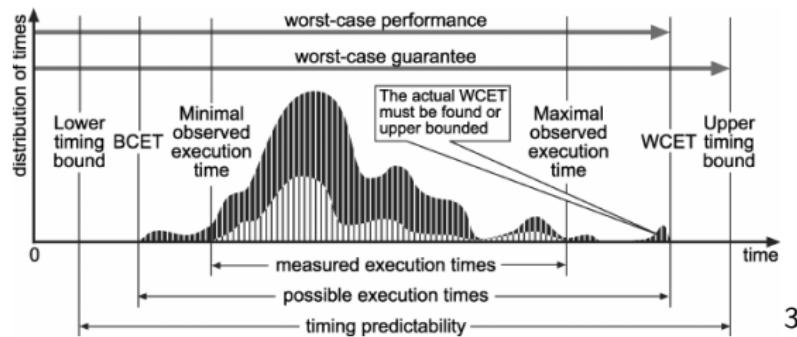
WCET-Analyse

- In der Echtzeitplanung wird die Kenntnis der (als konstant angenommenen) Ausführungszeit eines Prozesses vorausgesetzt
- Tatsächlich hängt diese ab von
 - ▶ Programmcode des Prozesses
 - ▶ Zustand des Prozesses zum Startzeitpunkt
 - ▶ Geschwindigkeit des Prozessors
 - ▶ Zustand des Prozessors
(Pipeline, Cache(s), Tag-RAM für Sprungvorhersage, ...)
- Bestimmung der Ausführungszeit ist komplex
- Ausführungszeit ist nicht konstant: variiert zwischen verschiedenen Prozessinstanzen

Ad-hoc Methode

In der Vergangenheit (und leider auch heute noch) anzutreffen:
End-to-End Messungen:

- Laufzeit wird experimentell bestimmt
- „Sicherheitsaufschlag“ hinzugaddiert
- + Einfach
 - keine Garantie, dass der worst case wirklich beobachtet wurde
- Werte sind i.A. nicht zufällig verteilt



³[Wilhelm et al, 2008]

Statische WCET-Analyse

- Forderung nach einer oberen Schranke der Ausführungszeit führt auf das Halteproblem (Turing)
 - Generell nicht für alle Programme lösbar
- Prinzipiell könnte mit einem Modell der Hardware (des Prozessors) der gesamte Zustandsraum eines Programmes „durchexerziert“ werden
- Der längste dabei auftretende Pfad liefert die WCET
- Probleme:
 - ▶ Explosion des Zustandsraums
 - ▶ Komplexität des Modells
 - ▶ Nur off-line möglich
 - ▶ Berücksichtigen von Unterbrechungen?
 - ▶ Ggf. zu pessimistisch
- Bei einigen Sicherheitsstandards (z.B. EASA) verbindlich

Dynamische WCET-Analyse

- Argumentation: Moderne Hardware zu komplex für Modellierung
 - ▶ Fehlerhaftes Modell wird bei statischer Analyse nicht erkannt, liefert aber ggf. falsche Ergebnisse
- Experimenteller Ansatz
 - ▶ Messen der WCET von Basic Blocks
 - ▶ Anhand des Kontrollflussgraphen alle möglichen Reihenfolgen der Basic Blocks (Ausführungspfade) konstruieren
 - ▶ Längster Pfad liefert WCET
- Probleme:
 - ▶ Berücksichtigen von Unterbrechungen?
 - ▶ Cache(s)?

Cache: empirische Sicht (1)

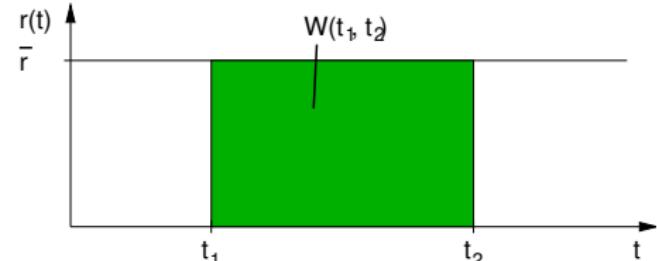
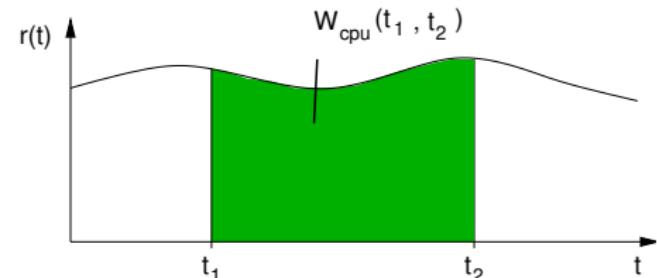
Prozessor führt mit einer Fortschrittsrate $r(t)$ Programmcode aus

- Rechenleistung, oder
Fortschrittsrate: $r(t)$
- Verrichtete Arbeit: $W_{cpu}(t)$

$$W_{cpu}(t_1, t_2) = \int_{t_1}^{t_2} r(\tau) d\tau$$

- Mittlere Rechenleistung
(d.h. $r(t) = \bar{r}$):

$$W(t_1, t_2) = (t_2 - t_1) \cdot \bar{r}$$



Umschaltverluste

- Durch Prozesswechsel entstehen Verluste:

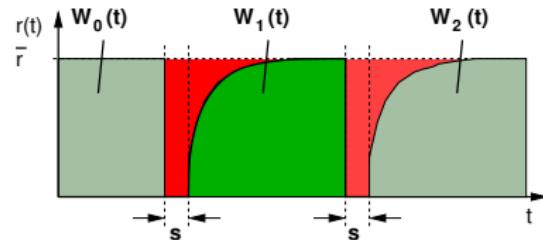
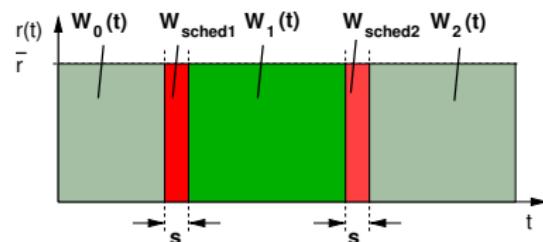
- Zeitscheiben anderer Prozesse (kein Verlust)
- Laufzeit des Schedulers (Verlust)

- Annahme: Scheduler-Laufzeit konstant (= s)

⇒ Verlust pro Scheduler-Ausführung:

$$W_s = s \cdot \bar{r}$$

- Bei Prozesswechsel⁴: weitere Verluste
- Verluste können Prozessen zugeordnet werden



⁴(N.B.: Prozesswechsel ≠ Scheduler-Aufruf)

Cache-bedingte Umschaltkosten

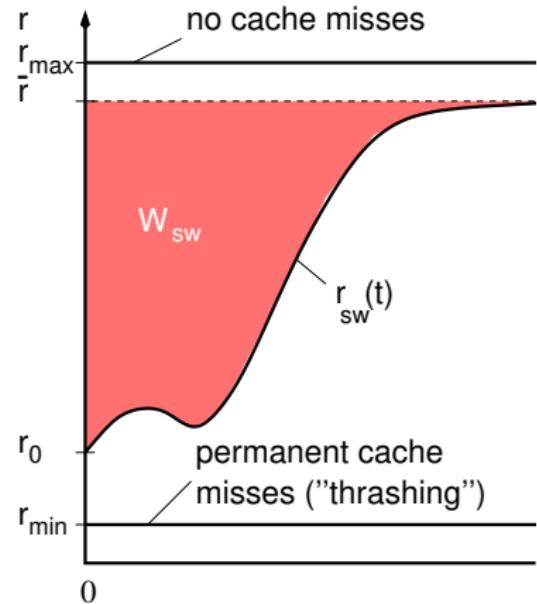
Bedingt durch Cache/TLB-Misses wird der Prozessor vorübergehend „langsamer“

⇒ Kosten je Prozesswechsel:

$$W_{sw}(t) = t \cdot \bar{r} - \int_0^t r_{sw}(\tau) d\tau$$

- Relativer Verlust:

$$O_{sw}(t) = 1 - \frac{1}{t} \int_0^t \frac{r_{sw}(\tau)}{\bar{r}} d\tau$$



Prozesswechsel bei $t = 0$

Nutzlastanteil

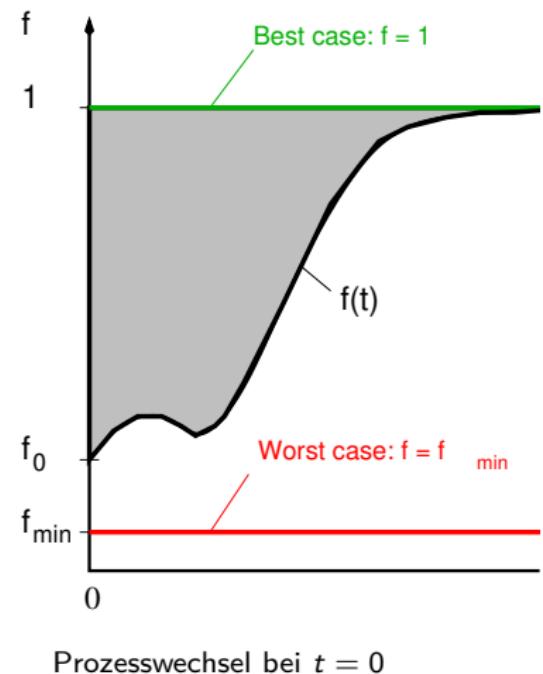
- Verhältnis von genutzter zu gesamter Rechenleistung:

$$f(t) := \frac{r_{sw}(t)}{\bar{r}}$$

- Damit:

$$O_{sw}(t) = 1 - \frac{1}{t} \int_0^t f(\tau) d\tau$$

- $f(t)$ ist i.A. unbekannt, aber
 - best case: $f(t) = 1$
 - worst case: $f(t) = f_{min} > 0$
 - Real: dazwischen ..

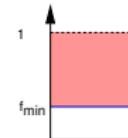


Nutzlastanteil approximieren

- Annahme eines durch eine Funktion $f(t)$ beschreibbaren Verlaufes des Nutzlastanteils, z.B.:

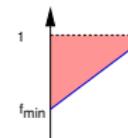
 - "cache flooding" (worst case) ...

$$f_{\text{flood}}(t) = \begin{cases} f_{\min}, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases}$$



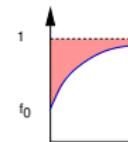
 - ... oder linear ...

$$f_{\text{lin}}(t) = \begin{cases} f_{\min} + \frac{1-f_{\min}}{t_s} \cdot t, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases}$$



 - ... oder Exponentialfunktion ...

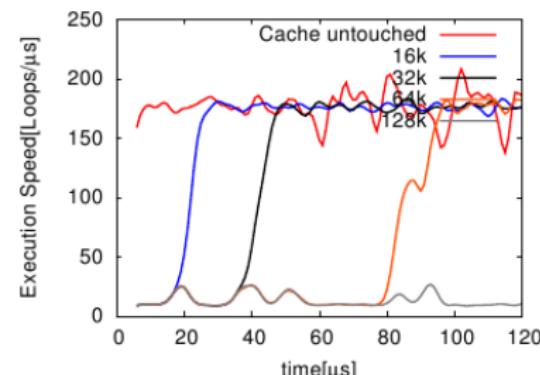
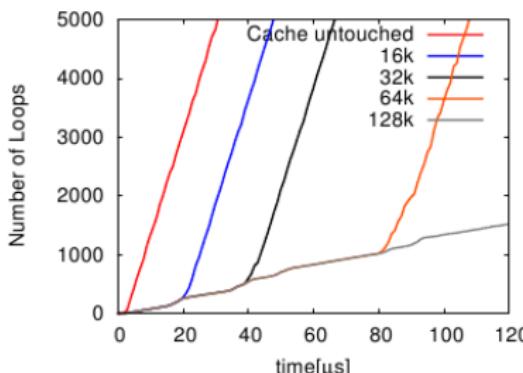
$$f_{\text{exp}}(t) = 1 + (f_0 - 1) \cdot e^{-kt}$$



- Task-bezogene Schätzung der Verluste möglich
- Wahl von $f(t)$ entsprechend Anforderungen an die Task, z.B.
 - „harte“ Echtzeit → wähle $f_{\text{flood}}(t)$
 - „weiche“ Echtzeit → wähle $f_{\text{lin}}(t)$ or $f_{\text{exp}}(t)$

„Kalibrieren“ der Nutzlastfunktion

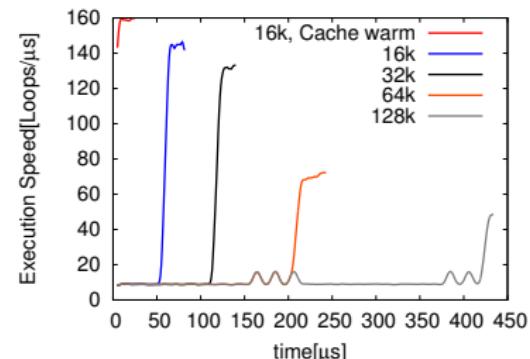
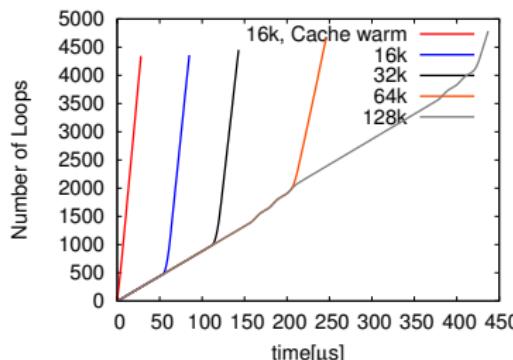
- Ermitteln konkreter Werte für (t_s , f_{min} , etc.)
 - ▶ Caches in definierten Zustand bringen (invalidate / read-fill / write-fill)
 - ▶ Intensive Lese- oder Schreibzugriffe auf variable großen, zuvor ungewachsene Speicherbereich („working set“)
 - ▶ Zeit für vorgegebene (variable) Anzahl Zugriffe messen
- Plattform: Intel Celeron 2,5 GHz



- Gemessen: Arbeit
- Daraus Abgeleitet: Leistung

„Kalibrieren“ der Nutzlastfunktion

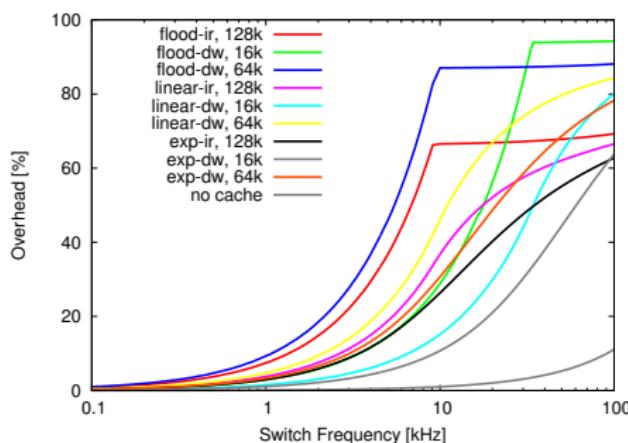
- Ermitteln konkreter Werte für (t_s , f_{min} , etc.)
 - ▶ Caches in definierten Zustand bringen (invalidate / read-fill / write-fill)
 - ▶ Intensive Lese- oder Schreibzugriffe auf variable großen, zuvor ungewachsene Speicherbereich („working set“)
 - ▶ Zeit für vorgegebene (variable) Anzahl Zugriffe messen
- Plattform: i.MX6 (SabreLight)



- Gemessen: Arbeit
- Daraus Abgeleitet: Leistung

Anwendung der Ergebnisse

- Aus den Messungen konkrete Werte für f_{min} und t_s ableitbar
- Damit: Simulation eines Zeitscheiben Schedulers (TSS)
- Cachesimulationen: linear, exponential und flood (worst case)
- Parametrisiert mit f_{min} , t_s wie bei i.MX6 SabreLight gemessen



- ⇒ Zeigt Beziehung zwischen Zeitscheibe ↔ Umschaltkosten
- ⇒ Bei SabreLight: Umschaltfrequenzen über $\approx 2\text{-}3 \text{ kHz}$ führen zu inakzeptablem Overhead!
- ⇒ Celeron: schon ab $\approx 1 \text{ kHz}$ inakzeptabel

- 2 Tasks, 50% Zeitscheiben, Angenommene Scheduler Ausführungszeit: $1\mu\text{s}$

Kap. 7:

Entwicklung von Anwendungen

(a) Programmiersprachen

7. Entwicklung von Anwendungen

Konzentration auf

- Sicherheitskritische Anwendungen
- Echtzeitanwendungen

Gliederung

(a)

1. Einführung
2. IEC EN 61508
3. Programmiersprachen (MISRA C/C++, Ada)

(b)

4. Modellierung (SysML, UML MARTE, ...)

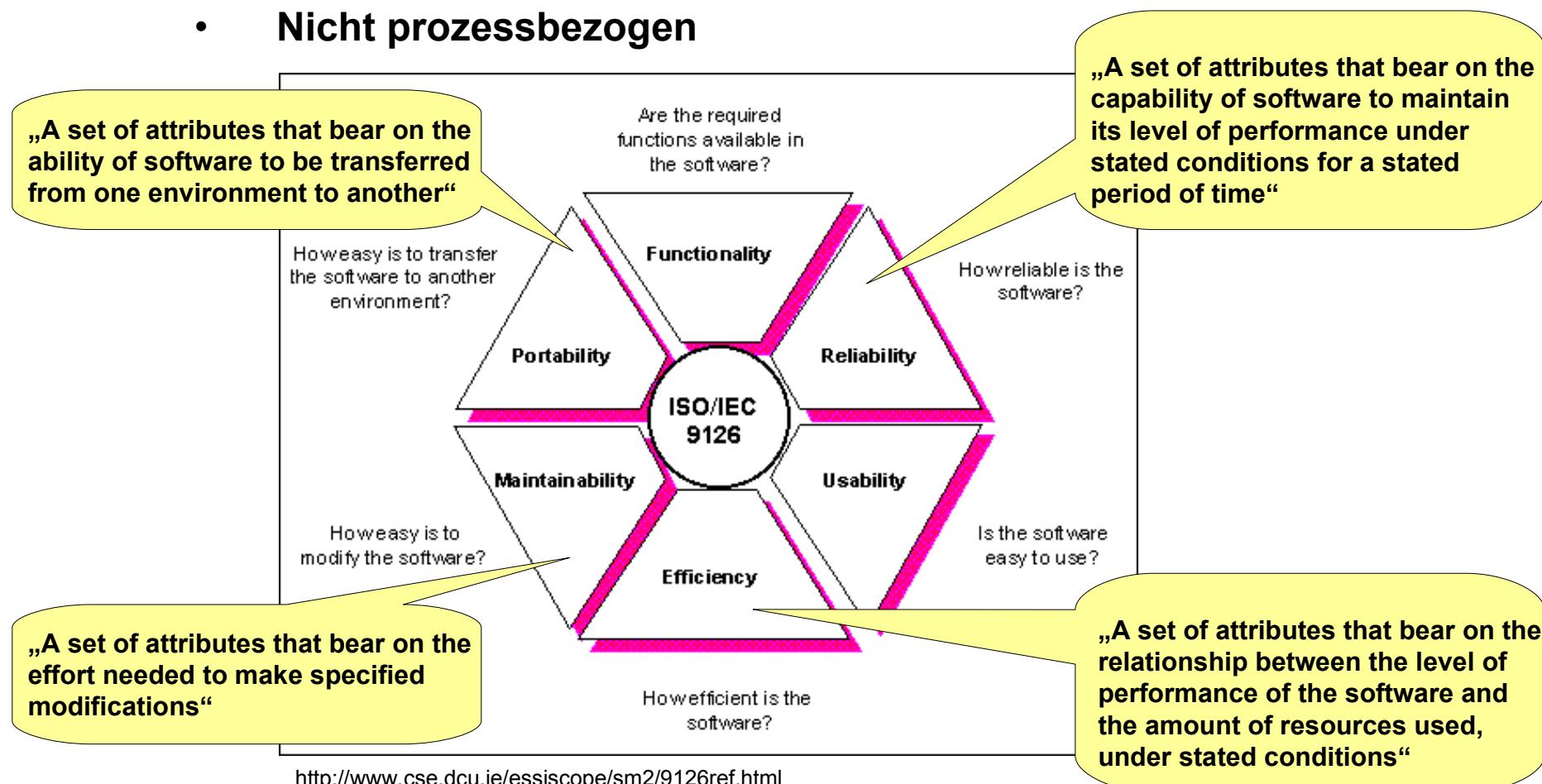
(c)

5. Validierung
6. Systematisches Testen

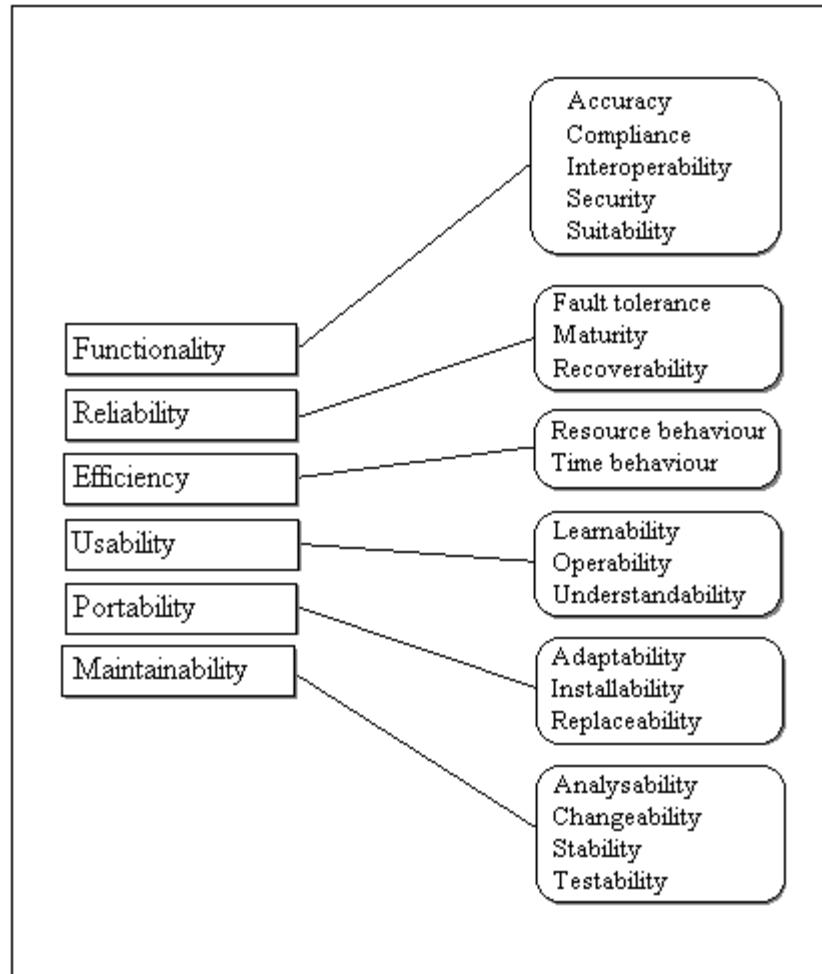
7.1. Einführung

ISO/IEC 9126:1991 → ISO/IEC 25010 (ab 2011)

- Kriterien zur Evaluation von Softwarequalität
- Nicht prozessbezogen

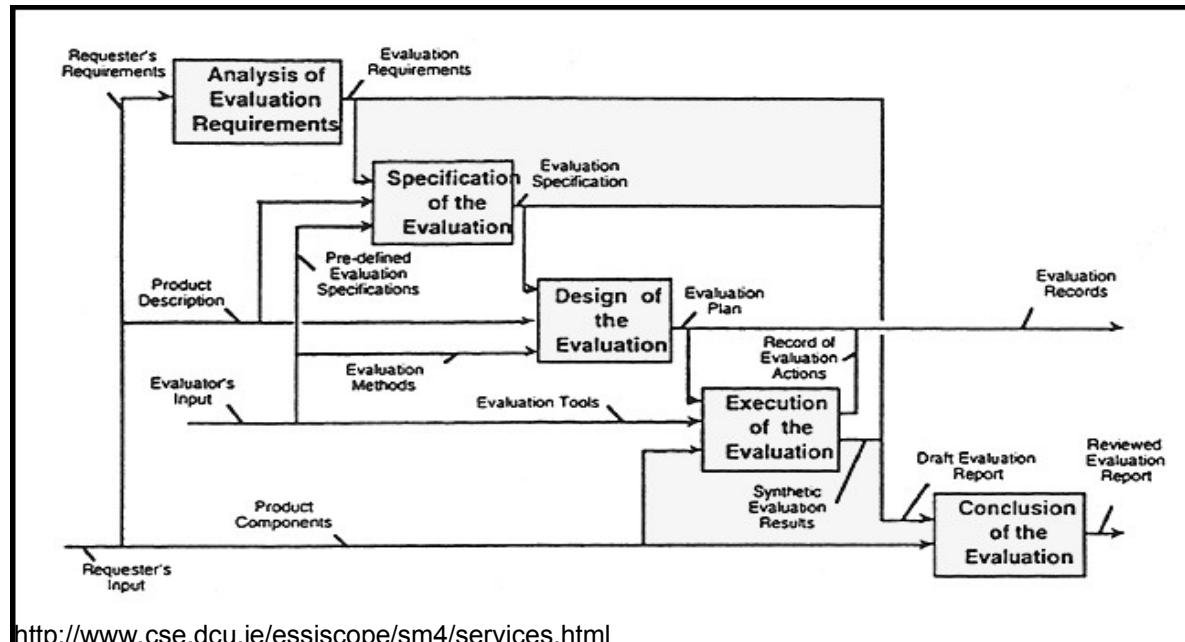


ISO/IEC 9126 Subattribute



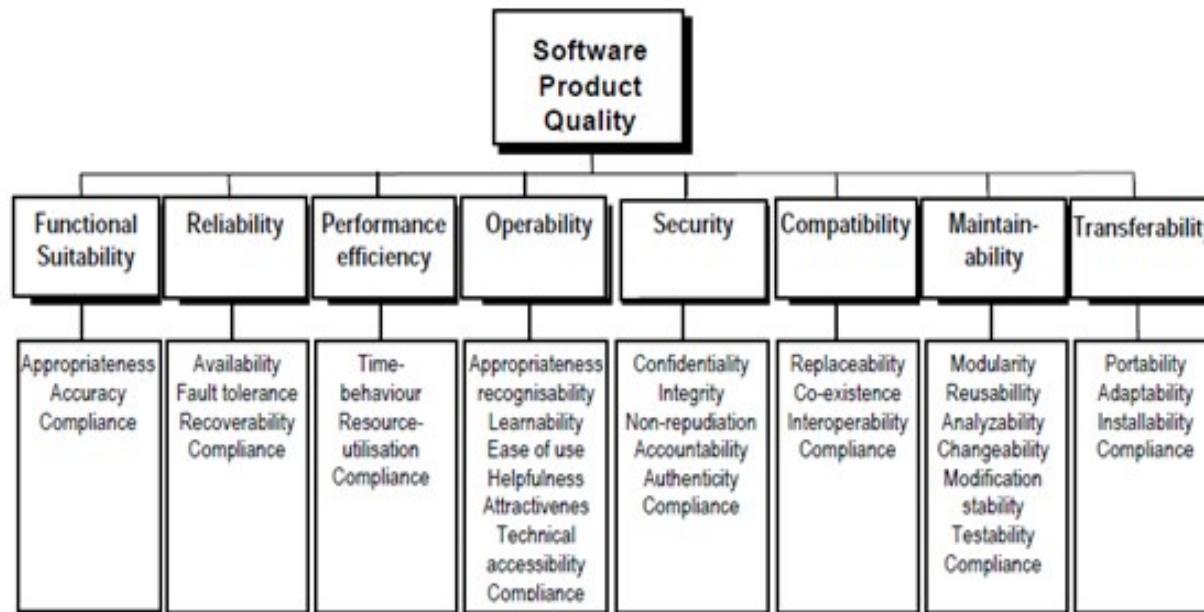
ISO/IEC 14598 (aktuell ISO/IEC 25040:2011)

- Betrachtet Prozess zur SW-Produkt-Bewertung
- Charakteristiken:
 - Repeatability
 - Reproducibility
 - Impartiality, Objectivity



ISO/IEC 25010 (ab 2011)

- Weiterentwicklung und Ersatz für 9126 und 14598
- ISO/IEC 25010:2011 Software product Quality Requirements and Evaluation (SQuaRE)
- 8 Qualitätskriterien (statt 6) mit 31 Subkategorien
- Ziel: Spec für nicht-funktionale Eigenschaften



<http://www.xbosoft.com/english/definition-software-quality/>

7.2. IEC EN 61508

- **Titel: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme**
 - Brachenübergreifende generische Richtlinie für sicherheitsgerichtete Systeme
 - Definiert Sicherheits-Integritätslevel
 - Beurteilung elektrischer/elektronischer/programmierbar elektronischer (E/E/PE)-Systeme in Bezug auf die Zuverlässigkeit von Sicherheitsfunktionen
 - SIL 1-4: "Vier wohlunterschiedene Stufen zur Spezifizierung der Anforderung für die Sicherheitsintegrität von Sicherheitsfunktionen, die dem E/E/PE-sicherheitsbezogenen System zugeordnet werden, wobei der Sicherheits-Integritätslevel 4 die höchste Stufe der Sicherheitsintegrität und der Sicherheits-Integritätslevel 1 die niedrigste darstellt."
- **Ziel: Schutz der Gesundheit von Menschen, der Umwelt und von Gütern**

- Die Sicherheitsanforderungsstufe stellt ein Maß für die Zuverlässigkeit des Systems in Abhängigkeit von der Gefährdung dar.
- Höherer Aufwand je höher die Gefährdung
- Beispiel: Antiblockiersystem
- Für Informatiker relevant: Sicherheitsfunktionen, die durch ein Programm erbracht werden einschl. der für die Erstellung verwendeten Werkzeuge (z.B. Compiler)

- Anwendung noch freiwillig, aber Druck durch Produkthaftungsgesetz (ProdHaftG)
- Speziell in Deutschland gilt nach §4 ProdHaftG, dass der Hersteller des Endproduktes sowohl bei der Haftung (wie auch bei eventuellen Imageschäden) selbst dann in Mithaftung genommen wird, wenn der Verursacher ausschließlich ein Unterlieferant war.
- IEC 61508 ist als „Sicherheits-Grundnorm“
 - IEC 61511: Funktionale Sicherheit — Sicherheitstechnische Systeme für die Prozessindustrie
 - IEC 61513: Kernkraftwerke — Leittechnik für Systeme mit sicherheitstechnischer Bedeutung – Allgemeine Systemanforderungen
 - EN 50128: Bahnanwendungen — Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Sicherheitsrelevante elektronische Systeme für Signaltechnik
 - IEC 62061: Sicherheit von Maschinen — Funktionale Sicherheit sicherheitsbezogener elektrischer, elektronischer und programmierbarer elektronischer Steuerungssysteme
 - IEC 60601: Medizinische elektrische Geräte - Allgemeine Festlegungen für die Sicherheit
 - ISO 26262: Road vehicles – Functional safety

- **Risikoklassen I-IV:**

- Kombination aus Kombination der Variablen Schwere des Unfalls und wahrscheinlicher Häufigkeit.
- I: Inakzeptables Risiko
- II: Unerwünscht, bei nicht reduzierbarem Risiko oder Unverhältnismäßigkeit des Aufwands akzeptabel
- III: Tolerierbares Risiko
- IV: Unbedeutendes Risiko

	Bedeutung/ Schwere				
Häufigkeit	Katastrophal	Kritisch	Gering	Unbedeutend	
Häufig	I	I	I	II	
Wahrscheinlich	I	I	II	III	
Gelegentlich	I	II	III	IV	
Zukünftig	II	III	III	IV	
Unwahrscheinl.	III	III	IV	IV	
Unglaublich	IV	IV	IV	IV	

- **Korrekte Ausführung der Aktionen, um den sicheren Zustand einer Einrichtung zu erreichen oder zu erhalten**
- **Gewährleistung aller Maßnahmen zur Vermeidung zufälliger oder systematischer Fehler**
- **Unterscheidung bei der Ermittlung der Ausfallwahrscheinlichkeiten nach Betriebsarten:**
 - Niedrige Anforderungsrate → Ausfallwahrscheinlichkeit PFD
(Probability of Failure on Demand)
 - Hohe oder kontinuierliche Anforderung → Ausfallwahrscheinlichkeit PFH
(Probability of Dangerous Failure per Hour)

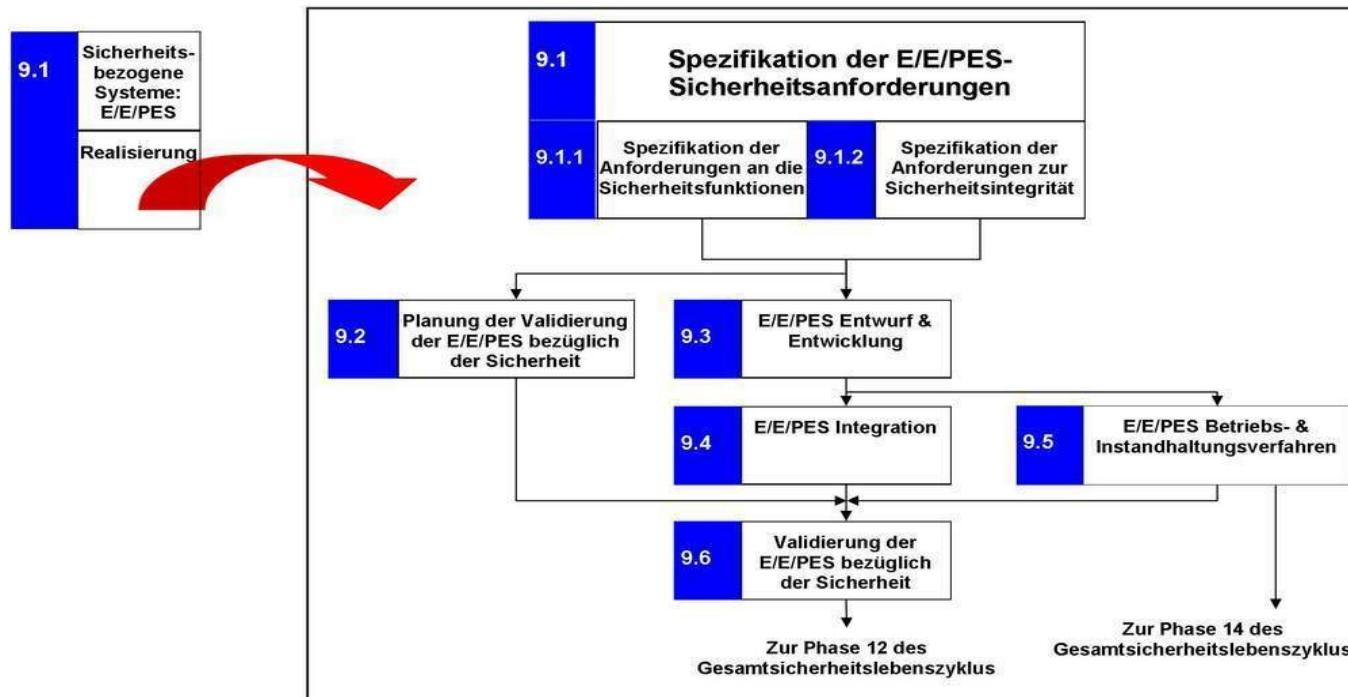
- Niedrige Anforderungsrate
 - PFD (Prob. Of Failure on Demand,
Wahrscheinlichkeit des Versagens bei Anforderung)
 - RRF (Risk Reduction Factor)

SIL	PFD	PFD (power)	RRF
➤ 1	0.1-0.01	$10^{-1} - 10^{-2}$	10-100
➤ 2	0.01-0.001	$10^{-2} - 10^{-3}$	100-1000
➤ 3	0.001-0.0001	$10^{-3} - 10^{-4}$	1000-10.000
➤ 4	0.0001-0.00001	$10^{-4} - 10^{-5}$	10.000-100.000

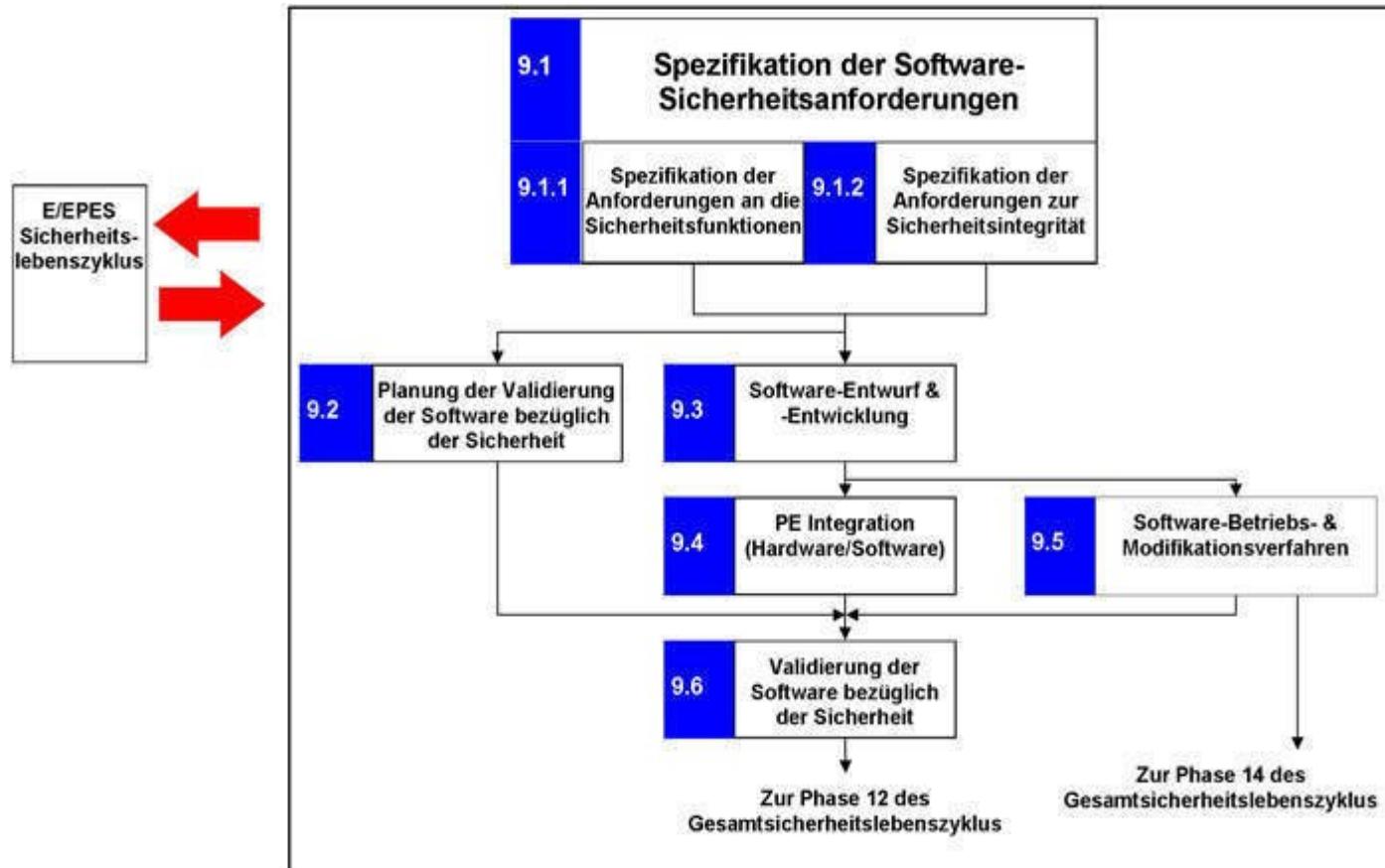
- **Continuous Operation (Dauerbetrieb)**
 - PFH (Prob. of Failure per Hour,
Wahrscheinlichkeit des Versagens pro Stunde)
 - RRF (Risk Reduction Factor)
- **SIL PFD (power) RRF**

➤ 1	10^{-5} - 10^{-6}	100.000-1.000.000
➤ 2	10^{-6} - 10^{-7}	1.000.000-10.000.000
➤ 3	10^{-7} - 10^{-8}	10.000.000-100.000.000
➤ 4	10^{-8} - 10^{-9}	100.000.000-1.000.000.000

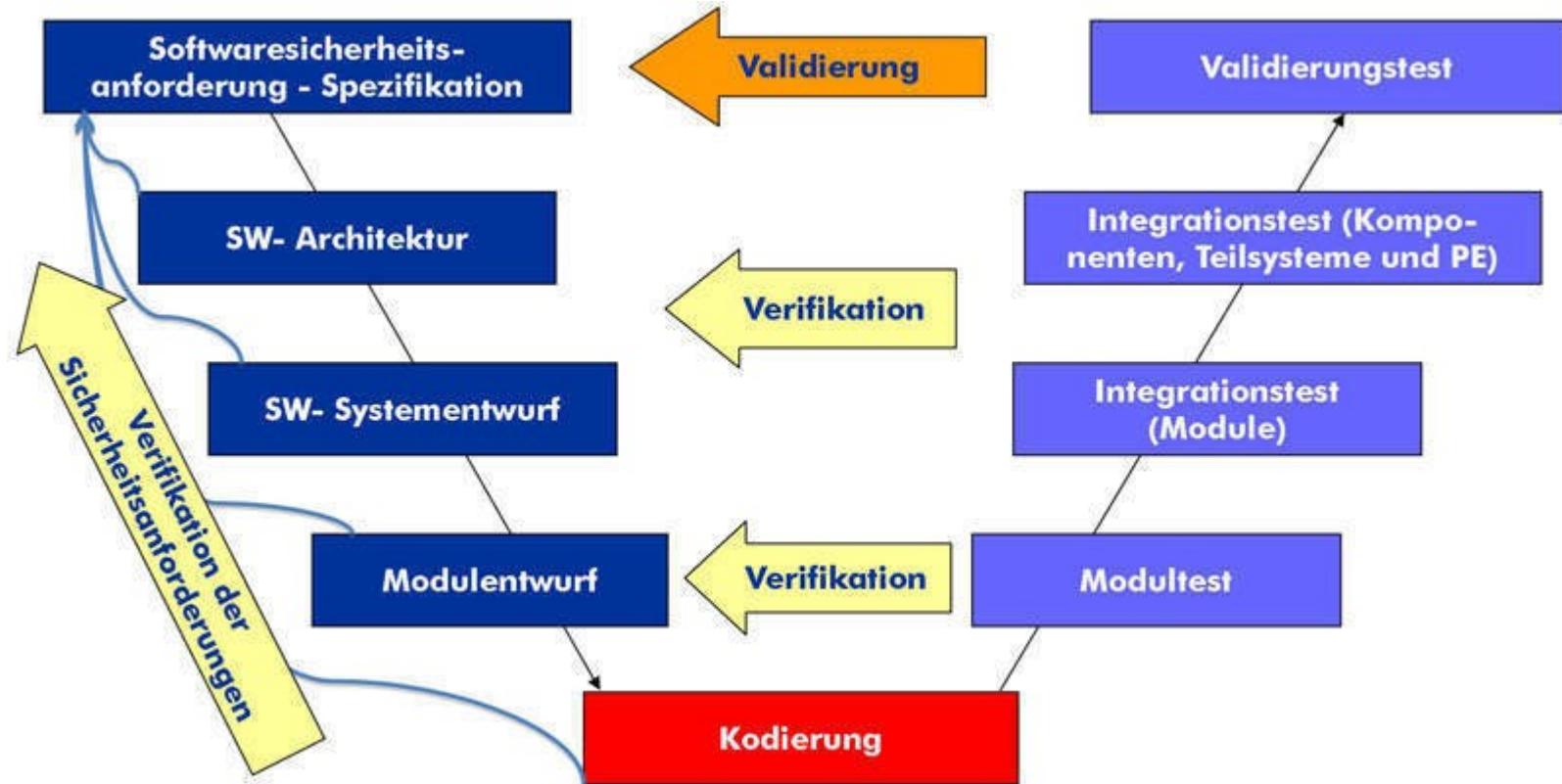
- **Gesamtlebenszyklus**
 - In allen Phasen des E/E/PES-Lebenszyklus müssen Maßnahmen zum Management der funktionalen Sicherheit, der Verifikation und der Beurteilung der funktionalen Sicherheit geplant, durchgeführt und dokumentiert werden.



- **Lebenszyklus von Software (61508 Teil 3)**



- **V-Modell für SW-Sicherheitsfunktionen (61508 Teil 3)**



- **Entwurfs- und Implementierungsfehler
(systematische Fehler)**
- **Sicherheitskritische Software sind Programmteile, die unmittelbar oder mittelbar die Sicherheit eines Systems durch Fehlfunktionen beeinflussen können.**
- **Restfehler in Software:**

Applikation	Umfang der SW	Zahl der Fehler	Zahl der Restfehler	Schwerwiegende Restfehler
Autopilot einer Rakete	30 000	1 500	60	6
Navigationssystem des Space Shuttle	500 000	25 000	1 000	100
Flugkontrollsoftware für Europa und USA	1 000 000	50 000	2 000	200
SW zur Steuerung eines Kernkraftwerkes	1 500 000	75 000	3 000	300

Anforderungen an SW-Entwicklungsprozess 7.2

- **Definition von Techniken und Verfahren, wie sicherheitsgerichtete SW entworfen, implementiert, getestet und dokumentiert werden soll**
- **Je nach SIL sind dabei bestimmte Methoden der Beschreibung gefordert (z.B. semi-formale oder formale Methoden)**
- **IEC 61508 Teil-7 enthält Beschreibung der vorgeschlagenen Methoden, äquivalente Methoden und Verfahren sind erlaubt – z.B. UML 2.0**
- **Entwurf**
 - **Trennung zwischen sicherheitsgerichteten und anderen Programmteilen**
 - **Einsatz semi-formeller Methoden: Klassendiagramme, Block-Diagramme, Sequenz-Diagramme, Programmablaufpläne und/oder Zustandsautomaten**

Anforderungen an SW-Entwicklungsprozess 7.2

- **Implementierung**
 - Nutzung bestimmter Programmiersprachen
 - Code Reviews, Walkthroughs
 - Statische Code-Analyse: z.B. Lint

Anforderungen an SW-Entwicklungsprozess 7.2

Table C.1 — Recommendations for specific programming languages

Programming language	SIL1	SIL2	SIL3	SIL4
1 Ada	HR	HR	R	R
2 Ada with subset	HR	HR	HR	HR
3 MODULA-2	HR	HR	R	R
4 MODULA-2 with subset	HR	HR	HR	HR
5 PASCAL	HR	HR	R	R
6 PASCAL with subset	HR	HR	HR	HR
7 FORTRAN 77	R	R	R	R
8 FORTRAN 77 with subset	HR	HR	HR	HR
9 C	R	---	NR	NR
10 C with subset and coding standard, and use of static analysis tools	HR	HR	HR	HR
11 PL/M	R	---	NR	NR
12 PLM with subset and coding standard	HR	R	R	R
13 Assembler	R	R	---	---
14 Assembler with subset and coding standard	R	R	R	R
15 Ladder Diagrams	R	R	R	R
16 Ladder Diagram with defined subset of language	HR	HR	HR	HR
17 Functional Block Diagram	R	R	R	R
18 Function Block Diagram with defined subset of language	HR	HR	HR	HR
19 Structured Text	R	R	R	R
20 Structured Text with defined subset of language	HR	HR	HR	HR
21 Sequential Function Chart	R	R	R	R
22 Sequential Function Chart with defined subset of language	HR	HR	HR	HR
23 Instruction List	R	---	NR	NR
24 Instruction List with defined subset of language	HR	R	R	R

HR Highly recommended
if this technique or measure is not used, the reasons for not using it should be detailed during safety planning.

R Recommended
as a lower measure to an HR recommendation.

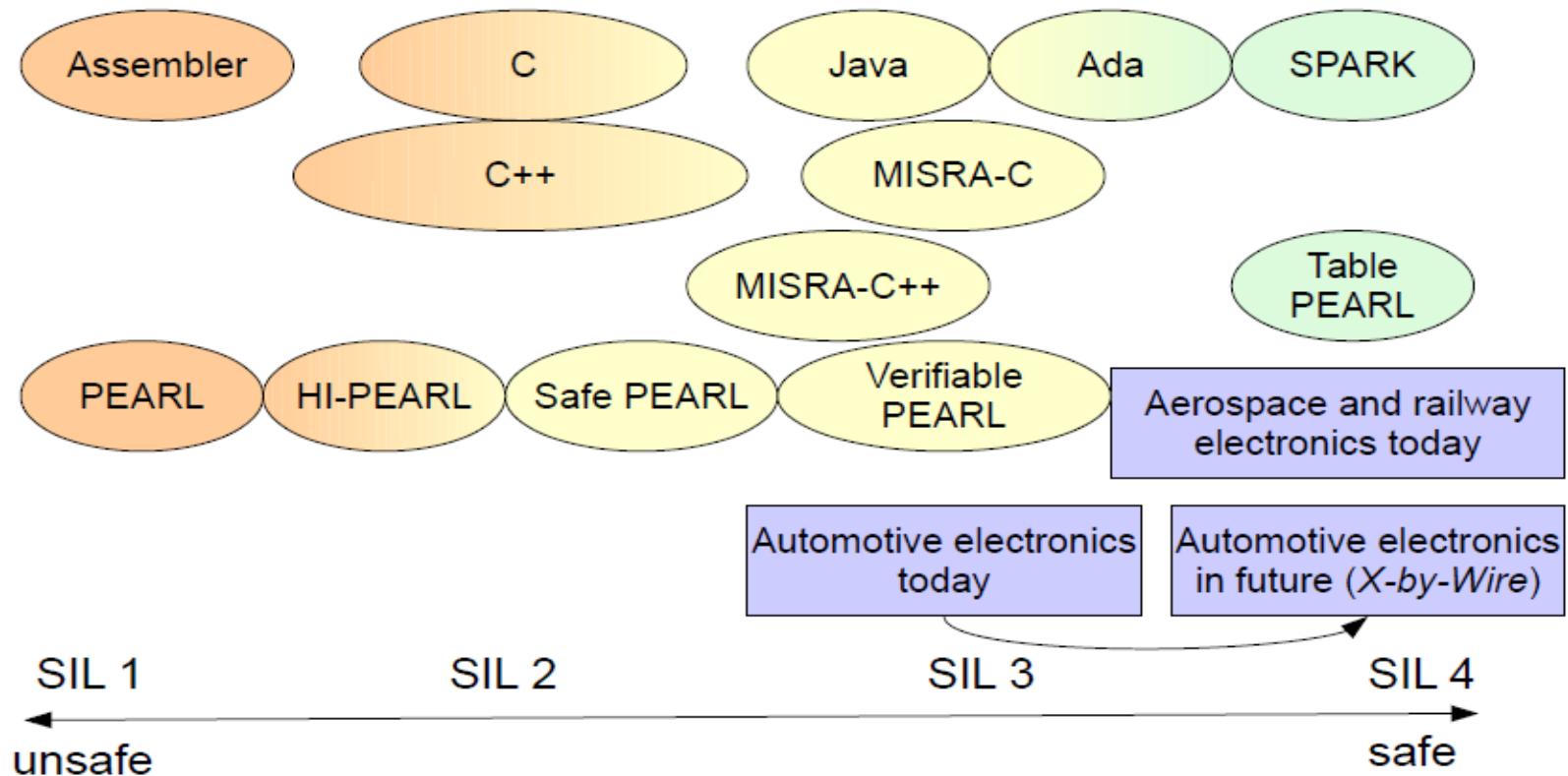
--- The technique has **no recommendation for or against** its use.

NR The technique or measure is positively **not recommended** for this safety integrity level – if it is used, the reasons for using it should be detailed during safety planning.

source: "IEC 61508-7: Functional safety of electrical/electronic/programmable electronic safety-related systems Part 7: Overview of techniques and measures"
Aus Folien Prof. Müller (Chemnitz)

Anforderungen an SW-Entwicklungsprozess 7.2

- Zusammenfassung aus Folien Prof. Müller (Chemnitz)



Anforderungen an SW-Entwicklungsprozess 7.2

- **Zusammenfassung**
 - C und C++ für sicherheitskritische Software nur als wohldefinierte Teilmengen mit Regeln und Guidelines und nur bis SIL 3
 - Dominant in der Automobilindustrie sind MISRA-C (2004) und MISRA-C++ (2008) mit SIL 3
 - Alternative Ada nur auf bestimmte Anwendungsgebiete beschränkt
 - Ansätze mit formaler Nachweisführung notwendig für SIL 4

Anforderungen an SW-Entwicklungsprozess 7.2

- **Test**
 - Modultests
 - Interface- und Datenfluss-Tests
 - Integrationstests mit vordefinierten Testfällen/-daten (Funktions-, Black Box-, White Box-Test)
 - Stress-, Reaktions-, Performance-Test
 - Alle Tests werden entsprechend Testplan durchgeführt
- **Regressionstests nach Modifikationen**
 - Beurteilung der Beeinflussung der Sicherheitsfunktion
 - Evtl. vollständiger neuer Nachweis
- **Validierung und Verifikation**
 - Testplanung und Testfälle aus Spezifikation abgeleitet
 - Simulation
- **Dokumentation, Versionierung und Rückverfolgbarkeit der Ergebnisse**

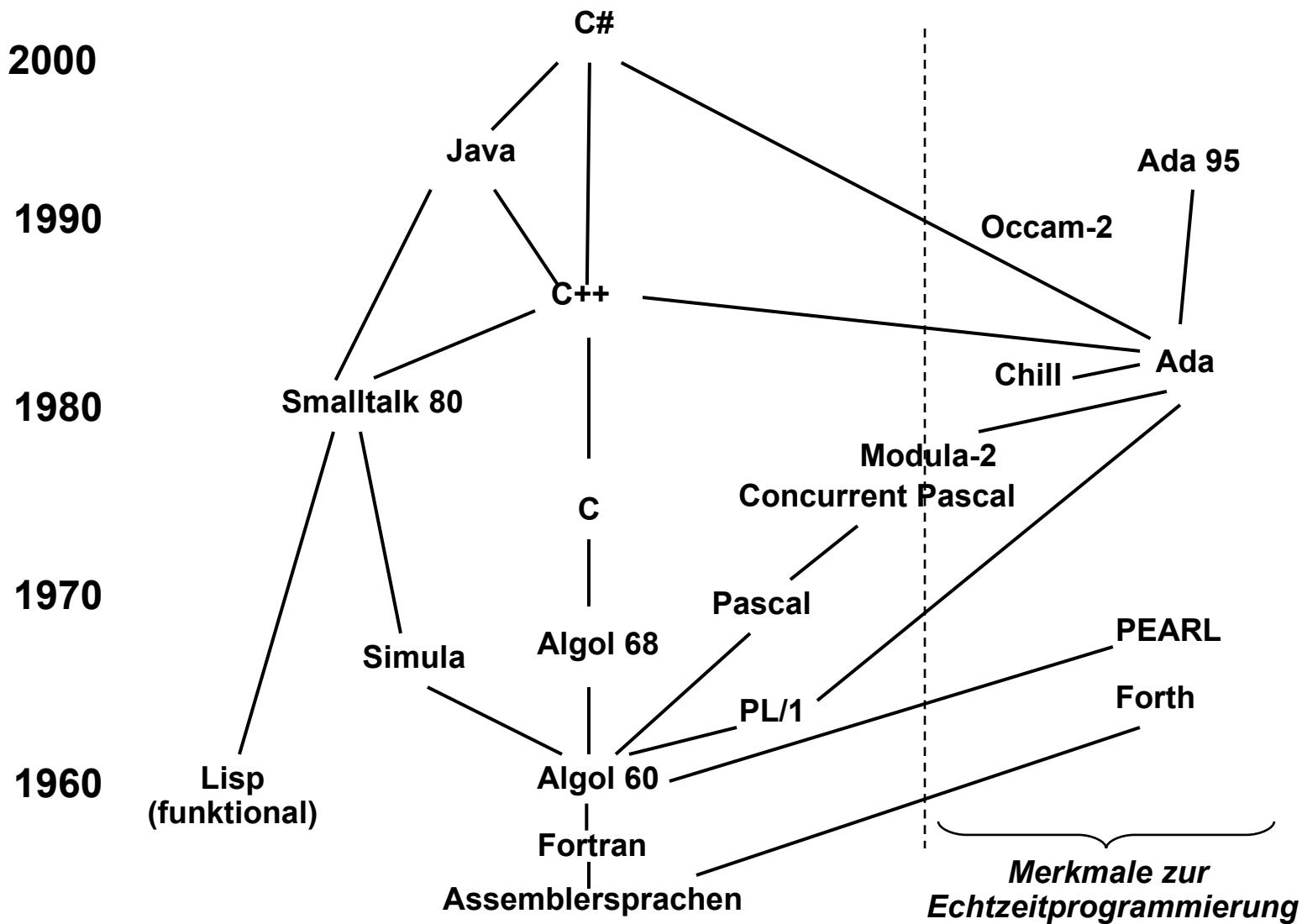
7.3 Programmiersprachen

Gliederung

1. Einführung
2. MISRA-C / C++
3. ADA

7.3.1. Einführung

- **Imperative Programmierung im Laufe der Zeit**
 - '60er: **prozedurale Abstraktion**
Algol 60
 - '70er: **strukturierte Programmierung**
Simula (67), Algol 68, C, Pascal
 - '80er: **Modularisierung**
Modula, Ada
 - '90er: **Objektorientierung**
C++, Java
 - '00er: **Web- und Plattformorientierung**
PHP, Ruby, C#, VB.NET
(alle eigentlich schon späte '90er)



- **Zeitbedingungen**
 - Formulierbarkeit
 - Überwachung der Einhaltung
- **Parallelität und Synchronisation**
 - Definition und Erzeugung nebenläufiger Prozesse (sporadisch, periodisch)
 - Mechanismen zur Interprozesskommunikation und Synchronisation
 - Einwirkung auf das Scheduling (z.B. durch Prioritäten)
- **Anpassungsfähigkeit an spezielle E/A**
- **Prüfung auf Plausibilität**
 - strenge Typprüfung
 - umfassende Fehlererkennung
- **Unterstützung des Programmierens im Großen**

- **bis Mitte '70er: Meinung: Echtzeitanwendungen müssen in Assembler implementiert werden !**
- **Dann: Zwei Entwicklungstendenzen:**
 - **Verbreitete Wirtssprache + neue Konstrukte für Echtzeitprogrammierung**
 - » Beispiele: Real-Time Fortran, Concurrent Pascal
 - » Problem: Wirtssprache behindert Entfaltung der echtzeitbezogenen Sprachmittel
 - **Neue Programmiersprache für die Bedürfnisse der Echtzeitprogrammierung**
 - » Beispiele: Forth, PEARL
 - » Problem: Nische = keine breite Akzeptanz
- **Idee: Neue Universalssprache mit Merkmalen der Echtzeitprogrammierung in den Grundkonzepten
⇒ Ada**

- **Aber: Starke Hinwendung zu C/C++ auch für Echtzeitanwendungen / Kritische Anwendungen**
 - obwohl keine Unterstützung dafür in der Sprache!
 - Gründe:
 - » komfortable Entwicklungsumgebungen
 - » erfahrene Entwickler verfügbar
 - » Hinwendung zu offenen Systemen
- **Ada wird nur dort angewendet, wo dies vom Auftraggeber vorgeschrieben wird:**
 - Militärische und zivile Luft- und Raumfahrt
- **Selbst dort C++ möglich:**
 - Joint Strike Fighter Air Vehicle (Lockheed Martin)
 - Vgl. JSF C++ Coding Standards (2005)

- C und C++ gehören zu den derzeit gebräuchlichsten Programmiersprachen
- C (ab 1972)
 - Imperativ, strukturiert
 - Geringe Anzahl an Schlüsselwörtern
 - Portabel, effizient (Bedeutung für Embedded Systems)
 - Für Systemprogrammierung sehr gut geeignet
- Probleme
 - Wenig geeignet für sicherheitskritische Anwendungen
 - z.B. kann fremder Speicher überschrieben werden
 - Viele Anfängerfehler (== / =, && / &, Setzen ;, Pointer, Pointer-Arithmetik)
 - Nicht behoben in C90 oder C99

- ***Nicht-spezifiziertes Verhalten***
 - Auswertungsfolge in Ausdrücken
 - Wandlung von int in enumeration
 - Bereichsverletzung
- ***Undefiniertes Verhalten***
 - Aktueller Parameter entspricht nicht vorgesehenem Typ
- ***Implementierungsabhängiges Verhalten***
 - Probleme bei Portierung
 - Länge des signifikanten Teils eines Bezeichners
 - Ist char signed oder unsigned?
- ***Internationalisierung***
 - Länderabhängige Spezifika (z.B. Zeichensatz)

- **Pros:**
 - Höhere Abstraktionsebene und konstruktive Mächtigkeit als C
 - Objektorientierung und Generizität (Templates)
 - Portierbarkeit (breite Verfügbarkeit von guten Compilern)
 - Trotzdem hohe Effizienz des Codes
 - Ausgebildete Entwickler, Werkzeugunterstützung, Solide Erfahrungen
- **Cons:**
 - Komplex, hoher Lernaufwand
 - Kompromisse durch Kompatibilität zu C
 - Viele ähnliche Probleme

- **Java (1995)**
 - Bessere Portabilität durch JVM
 - Vermeidung von Zeigerarithmetik
 - Garbage Collection
 - » Vereinfacht Programmierung
 - » Nicht Echtzeit-fähig da nicht deterministisch
- **Embedded C++ / EC++ (1996, Rev.3 1999)**
 - Einschränkungen für C++
 - » Keine Mehrfachvererbung, keine Exceptions, keine virtuellen Basisklassen, keine dynamischen Typen, keine *_cast-Operatoren, keine Namensräume, keine Templates, ...
 - Geringe Verbreitung, primär Japan
 - Starke Kritik von Stroustrup
(besser: Coding Standards verwenden!)

- **C# (2001, Rev. 4.0 in 2010, ISO/IEC 23270)**
 - Microsoft-orientiert (Lock-In-Gefahr)
 - Konzepte aus vielen anderen Sprachen
 - Objektorientierung
 - Starke statische Typisierung
 - Zeiger nur in „unsicherem Code“ erlaubt
 - Auswertung von Metadaten (z.B. für Methoden) zur Laufzeit
 - Trennung von .NET z.T. schwierig

- D (2007, www.dlang.org)
 - Erfinder: Walter Bright
 - Einflüsse aus C, C++, Java, C#, Python
 - Imperativ, objektorientiert, funktional (innerhalb imperativer Sprache), parallel, generisch, modular
 - Starke statische Typisierung
 - Zeiger möglichst vermieden, aber Nutzung möglich
 - Vermeidung von Zeiger-Arithmetik
 - Überladbare Operatoren
 - Module, Design by contract
 - Templates
 - Garbage Collection, aber global oder klassenbezogen deaktivierbar und durch eigene Speicherverwaltung ersetzbar
 - Maschinennahe Programmierung möglich

- **Verbreitung von C/C++ förderte andere Sichtweise für Programmierung von sicherheitskritischen und Echtzeit-Anwendungen:**
 - Versuche das Problem auf der Engineering-Ebene zu lösen, nicht auf der Programmiersprachen-Ebene
 - Guter Programmierstil durch Beschränkungen statt neuer Sprache
- **Ziele:**
 - Effizienz
 - Geringer Speicherbedarf
 - Portabilität
 - Echtzeitfähigkeit
 - Weiterverwendung existierender Werkzeuge wie Compiler, ...
 - Überprüfbarkeit der Regelmengen (z.B. lint)

7.3.2. MISRA-C / MISRA-C++



The Motor Industry Software Reliability Association

<http://www.misra.org.uk/>

- MISRA-C am weitesten verbreiteter Standard für C in sicherheitskritischen Anwendungen
- Erste Version 1998 veröffentlicht, ursprünglich auf Automobilindustrie beschränkt
 - „Guidelines for the use of the C language in Vehicle Based Software“
- MISRA-C: 2012:
 - Basis C99
 - „Guidelines for the use of the C language in critical systems“, (MIRA Limited)
 - „safe“ subset von C mit 143 verpflichtenden und 16 empfohlenen Regeln
 - Dokumentierte Ausnahmen vom Standard möglich

- Einfluss von MISRA-C auf JSF C++
- MISRA C++ (2008):
 - Guidelines for the use of the C++ language in critical systems

- 1 All code shall conform to ISO 9899 standard C, with no extensions permitted
- 9 Comments shall not be nested
- 10 Sections of code should not be commented out
- 11 Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers
- 13 The basic type of char, int, short, long, float and double should not be used, but specific length equivalents should be typedef'd for the specific compiler, and these type names used in the code
- 14 The type char shall always be declared as unsigned char or signed char
- 19 Octal constants (other than zero) shall not be used
- 20 All object and function identifiers shall be declared before use
- 21 Identifiers in an inner scope shall not use the same name as an identifier in an outer scope and therefore hide that identifier
- 30 All automatic variables shall have been assigned a value before being used
- 45 Type casting from any type to or from pointers shall not be used
- 50 Floating point variables shall not be tested for exact equality or inequality
- 52 There shall be no unreachable code

- 53 All non-null statements shall have a side-effect**
- 54 A null statement shall only occur on a line by itself, and shall not have any other text on the same line**
- 55 Labels should not be used, except in switch statements**
- 56 The goto statement shall not be used**
- 57 The continue statement shall not be used**
- 58 The break statement shall not be used (except to terminate the case of a switch statement)**
- 59 The statements forming the body of an if, else if, else, while, do... while or for statement shall always be enclosed in braces**
- 60 All if, else if constructs should contain a final else clause**
- 61 Every non-empty case clause in a switch statement shall be terminated with a break statement**
- 62 All switch statements should contain a final default clause**
- 63 A switch expression should not represent a Boolean value**
- 64 Every switch statement shall have at least one case**
- 65 Floating point variables shall not be used as loop counters**
- 69 Functions with variable numbers of arguments shall not be used**
- 70 Functions shall not call themselves, either directly or indirectly**

- **MISRA -C++**
 - Echte Teilmenge von C++, als geeignet für sicherheitskritische Anwendungen
 - 228 Regeln, kaum Beschränkung der charakteristischen Leistungsmerkmale von C++ (Exceptions, Templates erlaubt)
 - Nutzt viele Regeln von MISRA-C
- **Beispiele**
 - Präprozessor
 - » Nur für include-guards
 - » Defines für Konstanten oder Makros verboten (→ const, enumerations, inline functions)
 - Variablen müssen initialisiert sein
 - Keine unbenutzten Variablen

- **JSF C++**
 - Nutzt z.T. Regeln von MISRA-C
 - Keine Nutzung von Exceptions
 - Nutzung von Software-Metriken zur statischen Code-Beurteilung
 - ...
- **Ref: Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Lockheed Martin Corporation, Dec. 2005**

- MISRA-C, MISRA-C++ bilden Ansatz, durch Richtlinien die verbreiteten Sprachen C und C++ sicherer zu machen
- MISRA-C sehr etabliert, MISRA-C++ noch nicht
- MISRA-C, MISRA-C++ erlauben die vertretbare Nutzung von C/C++ bis SIL Level 3
(für SIL 4 formalerer Ansatz notwendig)

7.3.3. Ada

- Historie
 - Auftraggeber: US DoD (Department of Defense)
 - Anforderungsbeschreibungen: Strawman (1975), Woodenman (1975), Tinman (1976), Ironman (1977), Steelman(1978)
 - Sprachvorschläge verschiedener Institutionen
 - Auswahl von 4 Vorschlägen und deren vollständige parallele Entwicklung (nach Farben benannt)
 - Auswahl des Vorschlag „Green“ (CII Honeywell Bull, Frankreich, Leitung J. Ichbiah), 1979
 - Preliminary Language Reference Manual (PreLRM), 1980
 - Standard Language Reference Manual (LRM), 1983, Basis des Ada83-Standards (ANSI/MIL-STD 1815A-1983)

- **Mehrjähriges Ada9X-Projekt führt 1994/95 zu Ada95 (ISO865)**
 - Erweiterungen für Objektorientierung
 - Distributed Programming
- **Version Ada 2005**
 - macht Anleihen aus Java:
Interfaces (mit Mehrfachvererbung)
 - Positive Erfahrung durch Lockheed Martin zu
Flugsicherungssystem vor Liefertermin fertig und
unter geplantem Budget
- **Aktuelle Version 2012**

- **Ada Ravenscar Tasking Profile**
 - `pragma profile(Ravenscar)`
 - **Sichere Nebenläufigkeit von Tasks**
 - **Garantie für**
 - » Determinismus
 - » Möglichkeit zur statischen Analyse
 - » Task-Einplanbarkeit
 - » Speicherbeschränktheit
 - **Verbietet u.a. Rendezvous-Nutzung für Tasks**
 - **Zertifizierung bis SIL Level 4 möglich**

- **SPARK Ada**
 - Strikte Untermenge von Ada mit Annotationen
 - Versionen für Ada 83 und Ada 95
 - Math. Korrektheitsbeweise von Programmen durch statische Analyse mit entsprechenden Werkzeugen möglich
 - RavenSPARK enthält Ravenscar-Profil
 - Von vielen Autoren als sehr gut geeignet für SIL-4-Anwendungen angesehen
- **Aktuell SPARK 2014, basiert auf Ada 2012**
 - Schwerpunkt Korrektheitsbeweise
 - Sprachvereinfachungen zur besseren Lesbarkeit
 - Verbesserungen bzgl. Concurrency für Multicore-Prozessoren
 - Vgl. Ada Information Clearinghouse: www.adaic.org

- **Namensursprung**
 - Ada Augusta Byron, Countess of Lovelace (1816-56)
 - Mitarbeiterin von Charles Babbage
 - "the world's first programmer"
- **Sprache ist frei verfügbar**
 - z.B. im Gegensatz zu Java
 - Jeder Compiler, der sich Ada Compiler nennen will, muss umfangreiche Validierung bestehen
 - Beispiel: GNU Ada Translator (GNAT) frei verfügbar
- **Ada Programming Support Environment (APSE)**
 - angepasste Entwicklungswerkzeuge
 - Ada Runtime Environment
 - Bibliotheken



Aus
Wikipedia

- **Objektbasiertheit (abstrakte Datentypen):**
 - abgegrenzte Einheiten mit wohldefinierten äußeren und inneren Eigenschaften
 - Wertemenge, Operationenmenge
 - keine Vererbung
- **Typ-Konzept und -Erweiterungskonzept**
- **Trennung von Spezifikationsteilen und zugehörigen Rümpfen (information hiding, Impl.-Unabhängigkeit)**
- **Exception Handling**
- **Unterstützung für Programmierung im Großen**
 - Module
 - getrennt übersetzbare Einheiten
 - generische Module, Unterprogramme
(ähnlich Templates)

```
with Basis_EA;                      -- Benutzung einer EA-Bibliothek
use  Basis_EA;
procedure GGT is
    subtype MyInteger is Integer range -100 .. 100;
    x,y : MyInteger;
begin
    Schreibe("Bitte den 1.Paramater eingeben: ");
    Lies(x);
    -- Eingabe y
    while x /= y loop
        if x > y then x := x-y;
        else y := y-x;
        end if;
    end loop;
    -- Ausgabe
exception
    when Constraint_Error =>
        -- Fehlerbehandlung für Bereichsverletzung
    when others =>
        -- Fehlerbehandlung für sonstige Fehler
end GGT;
```

- **Zeitbedingungen**
 - Zugriff auf Echtzeituhr
 - Zeitdauern: Typ `duration`
 - Zeitpunkte: Typ `time`
 - Zeitfunktionen: package `calendar` einschließlich Rechnen + und - mit diesen Typen
 - Verzögerungs-Anweisung
 - » `delay <zeitraum>`
 - » `delay until <zeitpunkt>` (in Ada 95 zusätzlich)
 - insgesamt nur schwache Festlegungen in Ada 83
- **Prüfung auf Plausibilität**
 - Sehr strenge Typprüfung
 - möglichst vollst. Prüfung zur Compile-Zeit

- **Parallelität und Synchronisation**
 - nebenläufige Prozesse durch hierarchisches task-Konzept auf Sprachebene mit Verankerung im Ada Runtime-System
 - Unterstützung für statische Task-Prioritäten auf Ebene von pragmas als Subtyp von integer
 - gemeinsamer Mechanismus zur Interprozesskommunikation und -Synchronisation durch Rendezvous-Konzept

```
with Basis_EA;
use Basis_EA;
procedure Produzenten is
    task type Producer (ausgabe : character;
                           anzahl   : natural := 10);

    task_A : Producer('A');      -- Mit default Anzahl 10
    task_B : Producer('B', 5);  -- Mit Anzahl 5

    task body Producer is
        begin
            for i in 1 .. anzahl loop
                Schreibe(ausgabe);
            end loop;
        end Producer;

begin
    -- ausfuehrbarer Teil der umgebenden Prozedur
end Produzenten;
```



Prozesse starten
mit ihrer Vereinbarung

Beispiel: dynamische Prozesserzeugung

7.3.3

```
procedure system is
    -- Deklaration des Prozesstyps Producer

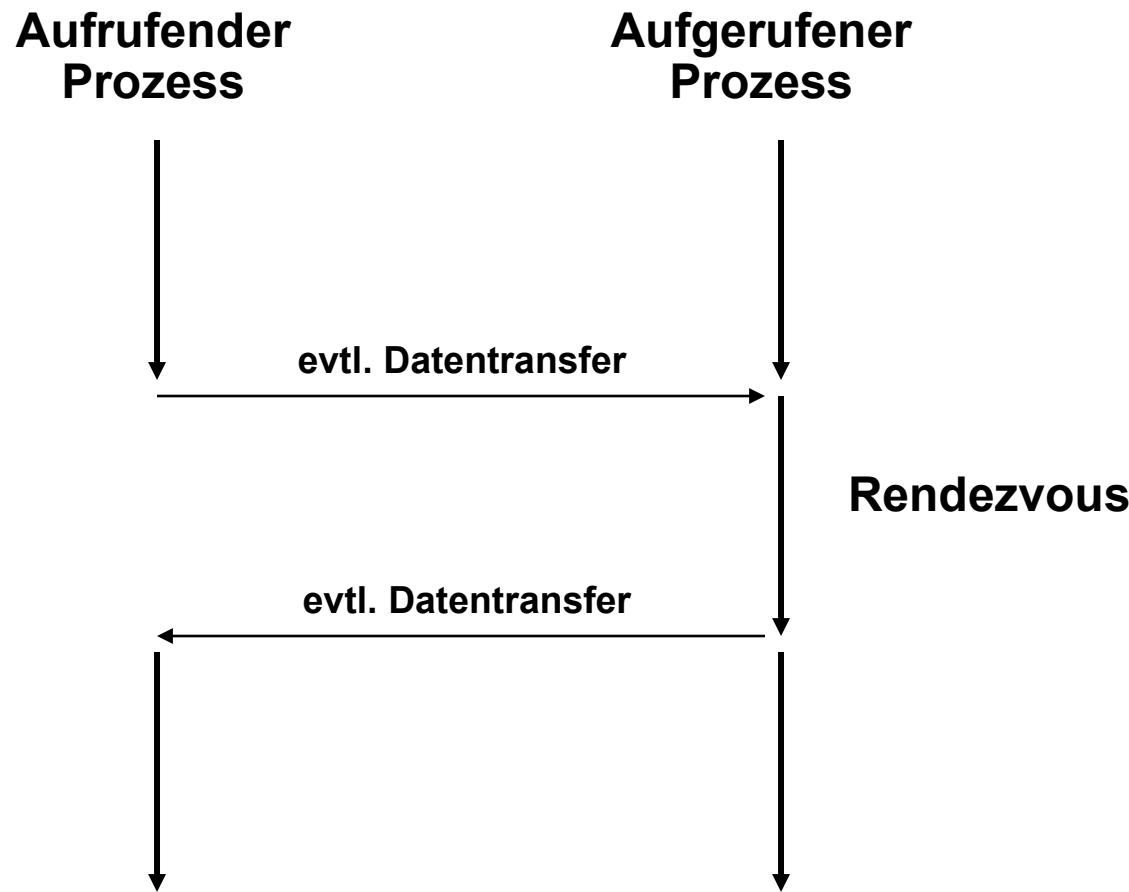
    type ProducerRef is access Producer; -- typisierter Zeiger-Typ

    myTaskRefs : array (1..10) of ProducerRef;

begin
    ...
    myTaskRefs(4) := new Producer('C', 100);
        -- task startet unmittelbar jetzt
        -- Lebensdauer nicht an umgebenden Block gebunden

end system;
```

- **synchron**
 - keine Pufferung
 - wechselseitiges Warten auf den Partner
 - Austausch von typisierten Parametern
- **asymmetrisch**
 - Aufrufer setzt Entry-Call auf einen bestimmten Eingang des Aufgerufenen/Empfängers ab
 - Aufgerufener signalisiert Bereitschaft zum Rendezvous durch Annahme oder Auswahl-Anweisung (accept bzw. select-Statement)
- **einseitig-anonym**
 - Aufgerufener weiß nicht, wer ihn aufruft
- **evtl. Anhebung der Priorität des niederen prioren Prozesses eines Rendezvous**



```
task type buffer is
    entry inbuffer(c : in character);
    entry outbuffer(c : out character);
end buffer;
```

Vereinbarung
von Entry-Punkten

```
task body buffer is
  N : constant integer := 1000;
  b : integer range 0..N;
  i,j : integer range 1..N;
  speicher : array (1..N) of character;
begin
  b := 0;  i := j := 1;
loop
  select
    when b<0 => -- Speicher nicht voll
      accept inbuffer(c : in character) do
        speicher(i):=c;
      end inbuffer;
      b := b+1;
      i := (i mod N) + 1;
    or
    when b>0 => -- Speicher nicht leer
      accept outbuffer(c : out character) do
        c:=speicher(j);
      end outbuffer;
      b := b-1;
      j := (j mod N) + 1;
    end select;
  end loop;
end buffer;
```

} Rendezvous
} folgende Verarbeitung

Beispiel: Task Entry-Punkte (2)

7.3.3

```
task body buffer is
  N : constant integer := 1000;
  b : integer range 0..N;
  i,j : integer range 1..N;
  speicher : array (1..N) of character;
begin
  b := 0;  i := j := 1;
loop
  select
    when b<0 => -- Speicher nicht voll
      accept inbuffer(c : in character) do
        speicher(i):=c;
      end inbuffer;
      b := b+1;
      i := (i mod N) + 1;
    or
    when b>0 => -- Speicher nicht leer
      accept outbuffer(c : out character) do
        c:=speicher(j);
      end outbuffer;
      b := b-1;
      j := (j mod N) + 1;
    end select;
  end loop;
end buffer;
```

} Rendezvous
} folgende Verarbeitung

- Objektorientierung durch *tagged types*, von denen (einfach) abgeleitet werden kann und die dabei erweitert werden können.
- Typklassen (Vereinigung aus einem Typ T und allen Typen, die aus T direkt oder indirekt abgeleitet sind) mit polymorphen Typen und impliziten Konvertierungen
- Abstrakte Unterprogramme, die in abgeleiteten Klassen definiert werden müssen

- **Interrupt-Behandlung auf Basis von geschützten Prozeduren**
- **Pakete zum Identifizieren von Tasks und zur Verwaltung von Task-Attributen**
- **Unterbrechbares Task-Scheduling mit Basis-Prioritäten und aktiven Prioritäten**
- **Prioritäts-gesteuerte Warteschlangen mit der Möglichkeit system- oder benutzerdefinierter Alternativ-Strategien**
- **Priority Inheritance Protocol bei allen geschützten Operationen**
- **Dynamische Prioritäten durch spezielles Paket**

- **Protected Types zum koordinierten Zugriff auf gemeinsam benutzte Daten**
 - spezielle Zugriffsfunktionen (protected operations)
 - nur lesender Zugriff ohne Einschränkungen
 - beliebiger Zugriff mit wechselseitigem Ausschluss
 - Entry-Calls eines Rendezvous mit Blockierung, bis eine Bedingung (Condition) wahr wird
- **Funktionalität war auch in Ada 83 mit dem vorhandenen Rendezvous-Konzept erreichbar, jetzt natürlicher und komfortabler.**

- **Weiterer Standardisierungsprozess (bis 2007)**
 - Java-ähnliche Interfaces (inheritance of interfaces) statt echte Mehrfachvererbung wie in C++
 - Real-Time and High-Integrity Support
 - » task termination handlers,
 - » timing events, execution-time clocks,
 - » alternative task dispatching policies,
 - » standardization of the Ravenscar restricted tasking profile
 - ...

Kap. 7:

Entwicklung von Anwendungen

(b) Modellierung

7. Entwicklung von Anwendungen

Konzentration auf

- Sicherheitskritische Anwendungen
- Echtzeitanwendungen

Gliederung

(a)

1. Einführung
2. IEC EN 61508
3. Programmiersprachen (MISRA C/C++, Ada)

(b)

4. Modellierung (SysML, UML MARTE, ...)

(c)

5. Validierung
6. Systematisches Testen

7.4 Modellierung

Gliederung

7.4.1 Einführung

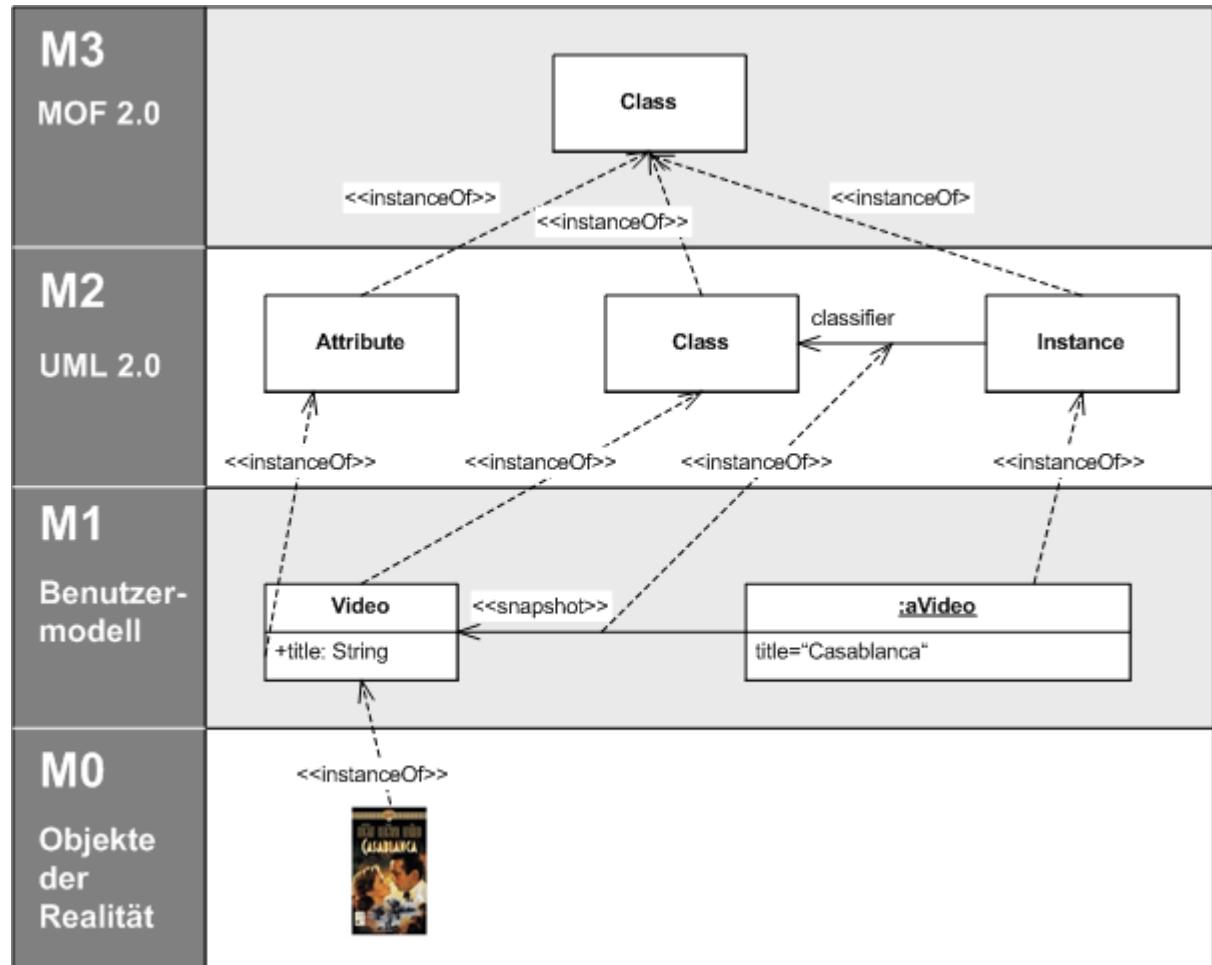
7.4.2 SysML

7.4.3 MARTE

7.4.1 Einführung

- **OMG Object Management Group**
- **UML und ihre Verwandten - Standards zu Modellierung**
 - Bisher primär UML (Unified Modeling Language) kennengelernt (vgl. LV Softwaretechnik)
 - Katalog: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF
 - hier nur Ausschnitt
 - SysML (Systems Modeling Language, für komplexe Systeme)
 - MOF Meta Object Facility (extensible model driven integration framework, u.a. Modellierung von UML)
 - OCL Object Constraint Language (Festlegung von Prädikaten/Restriktionen für UML-Modelle und Metamodelle)
 - XMI XML Metadata Interchange Specification
OMG MOF 2 XMI Mapping Specification
(MOF → XML Mapping, Exportieren/Importieren von Modellen aus/in Modellierungswerkzeugen)

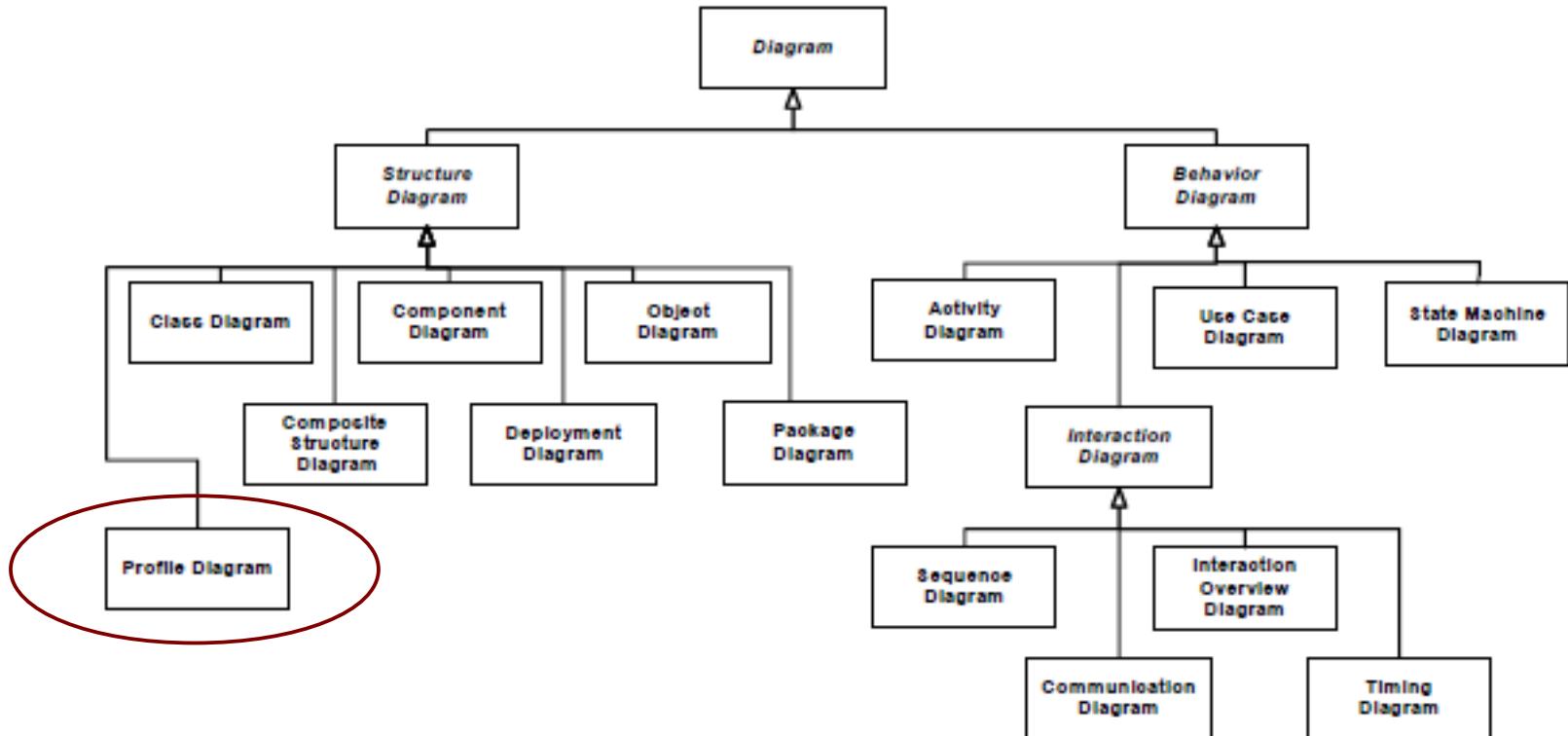
- **MOF: OMG Meta Object Facility**
- **M3 hat sich selbst als Metamodell**



<http://upload.wikimedia.org/wikipedia/commons/9/93/M0-m3.png>

- Erweiterung des UML 2 Metamodells durch einen leichtgewichtigen Erweiterungsmechanismus
- „A profile defines limited extensionsto a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain“ (OMG 10-05-05)
- UML Metamodell bleibt unverändert
- Profile Diagram als neuer Diagrammtyp für die Anwendung von Profilen
- Erweiterung basiert auf
 - Stereotypes mit Tagged Values
 - Constraints (Kommentare oder in formaler OMG Object Constraint Language (OCL))
- Analogie zu Einschränkungen auf Quellcode-Ebene wie z.B. in MISRA C

- UML 2.3



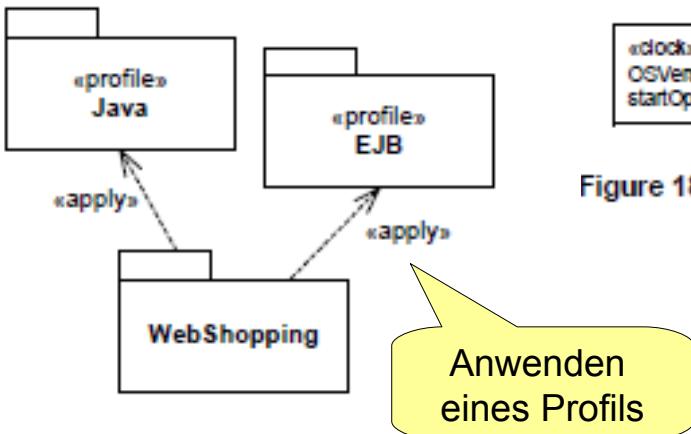
UML 2.3 OMG 10-05-05.pdf

- A UML profile is a specification that does one or more of the following:
 - Identifies a subset of the UML metamodel.
 - Specifies “well-formedness rules” beyond those specified by the identified subset of the UML metamodel.
 - “Well-formedness rule” is a term used in the normative UML metamodel specification to describe a set of constraints written in UML’s Object Constraint Language (OCL) that contributes to the definition of a metamodel element.
 - Specifies “standard elements” beyond those specified by the identified subset of the UML metamodel.
 - “Standard element” is a term used in the UML metamodel specification to describe a standard instance of a UML stereotype, tagged value or constraint.
 - Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML metamodel.
 - Specifies common model elements, expressed in terms of the profile.

Figure 18.5 - The notation for an Extension



Figure 18.17 - Using a stereotype



<metaclass>
Class

<stereotype>
Clock

OSVersion:String
startOperation:Operation
POSIXCompliant:Boolean

Figure 18.13 - Defining a stereotype



Figure 18.18 - Showing values of stereotypes and a simple instance specification

Constraints
als Kommentar

Figure 18.12 - Profiles applied to a package

UML 2.3 OMG 10-05-05.pdf

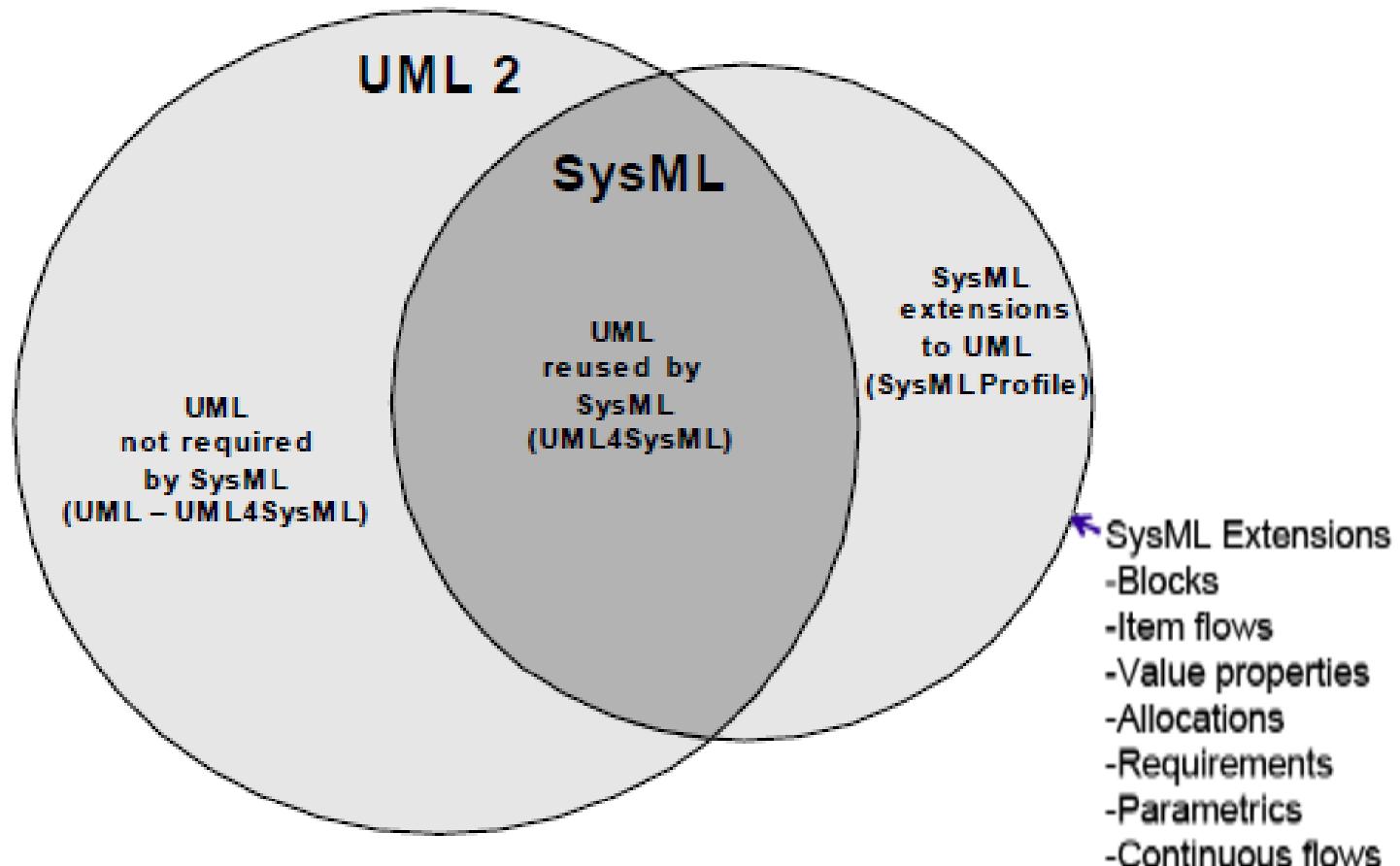
7.4.2 SysML

- **OMG SysML: Systems Modeling Language**
- **UML 2 Profil für Systems Engineering**
- **OMG Standard formal/2012-06-01 (Version 1.3 beta)**
- The OMG Systems Modeling Language (OMG SysML™) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities. In particular, the language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which is used to integrate with other engineering analysis models. SysML represents a subset of UML 2.0 with extensions needed to satisfy the requirements of the UML™ for Systems Engineering RFP. SysML uses the OMG XML Metadata Interchange (XMI®) to exchange modeling data between tools.
- **Erweiterung durch UML Stereotypes, UML Diagramm-Erweiterungen, Modell-Bibliotheken**
- **Engineering-Prozess: Von Dokumenten zu Modellen**

SysML Tools

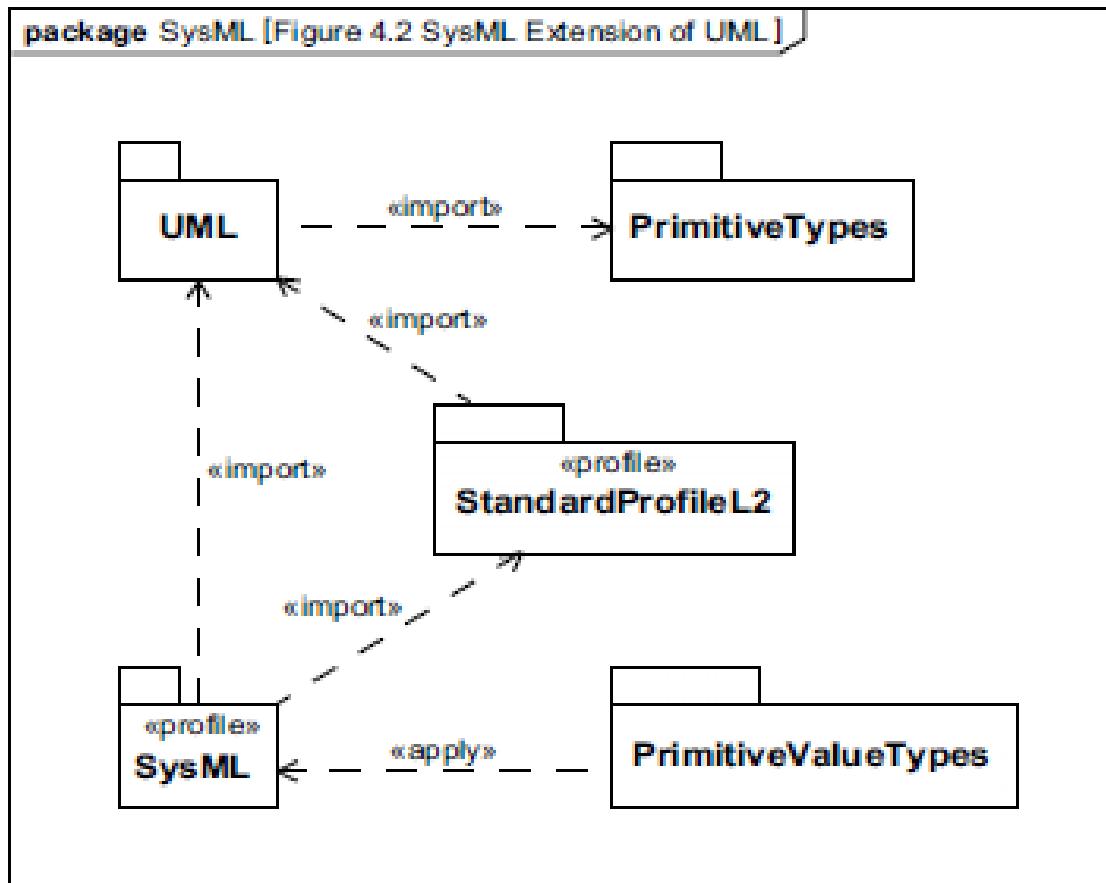
- **Basieren i.w. auf UML-Modellierungswerkzeugen**
- **Kommerziell**
 - Artisan (Studio)
 - EmbeddedPlus (SysML Toolkit)
 - No Magic (Magic Draw)
 - SparxSystems (Enterprise Architect)
 - IBM / Telelogic (Tau and Rhapsody)
- **Open Source basierend auf Eclipse**
 - TopCased
 - Papyrus

- Überblick zum Zusammenhang

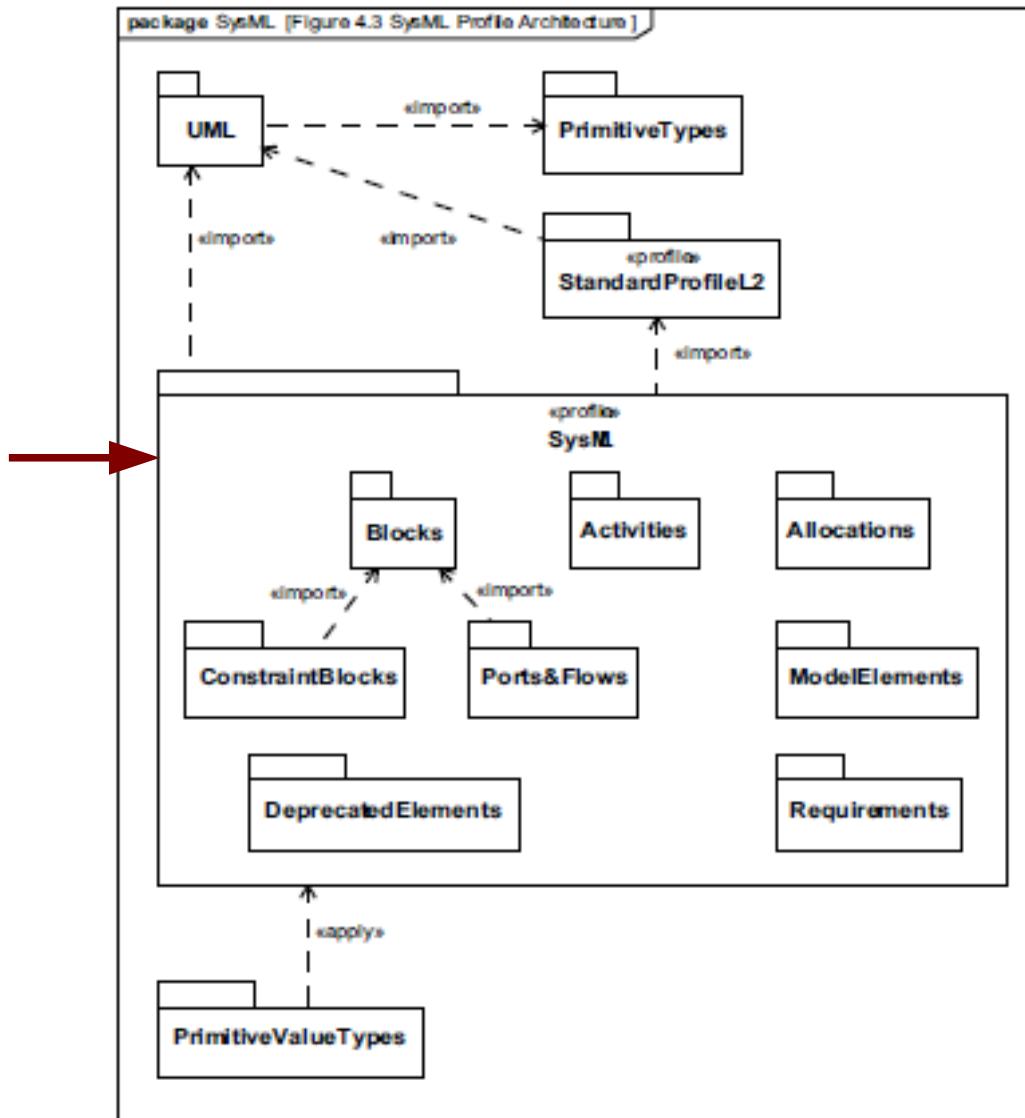


SysML 1.3 OMG 12-06-01.pdf

- Erweiterungen

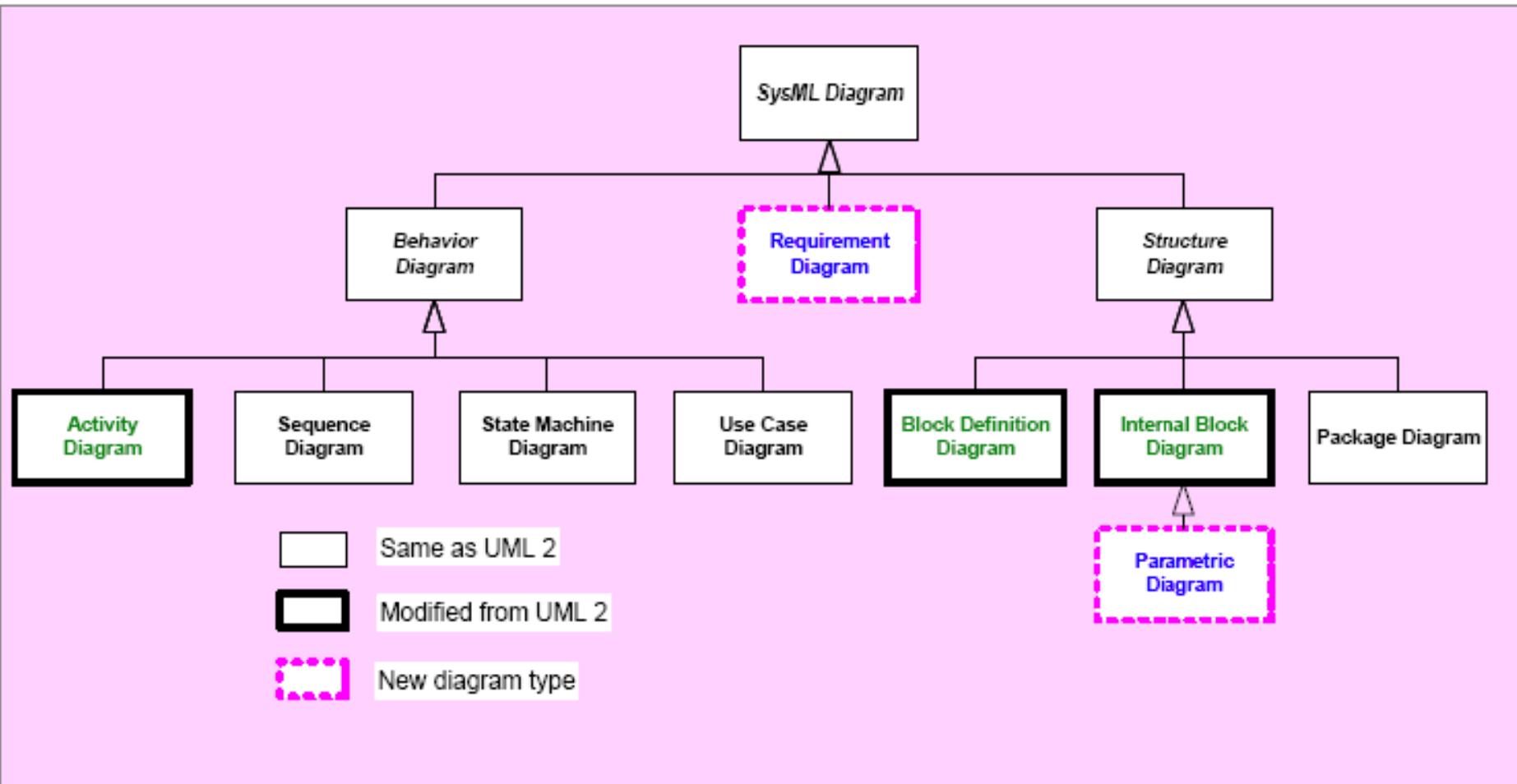


SysML 1.3 OMG 12-06-01.pdf



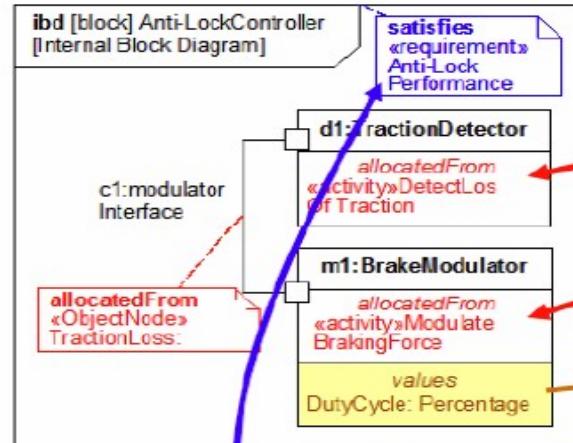
SysML 1.3 OMG 12-06-01.pdf

Überblick



- The SysML packages extend UML as follows:
 - **SysML::Model Elements** refactors and extends the UML kernel portion of UML classes.
 - **SysML::Blocks** reuses structured classes from composite structures.
 - **SysML::ConstraintBlocks** extends Blocks to support parametric modeling.
 - **SysML::Ports and Flows** extends UML ports, UML information flows, and SysML Blocks.
 - **SysML::Activities** extends UML activities.
 - **SysML::Allocations** extends UML dependencies.
 - **SysML::Requirements** extends UML classes and dependencies.
 - **SysML::DeprecatedElements** extends UML ports, UML interfaces, and SysML Item Flows.

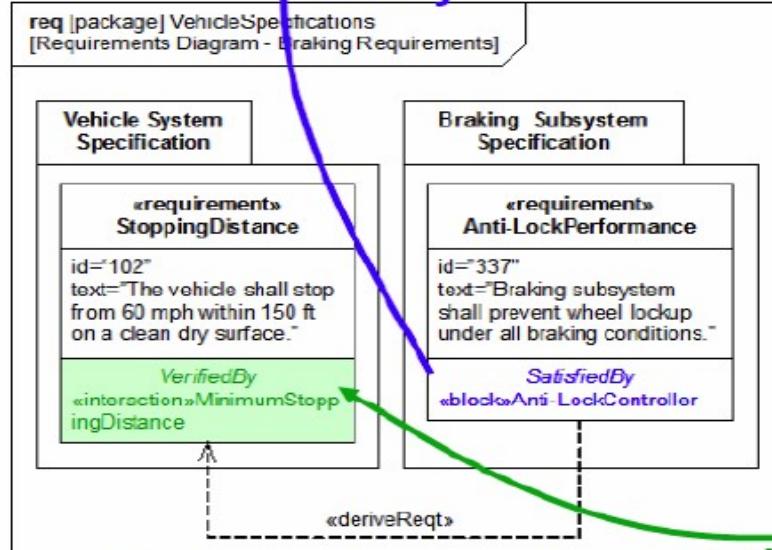
1. Structure



allocate

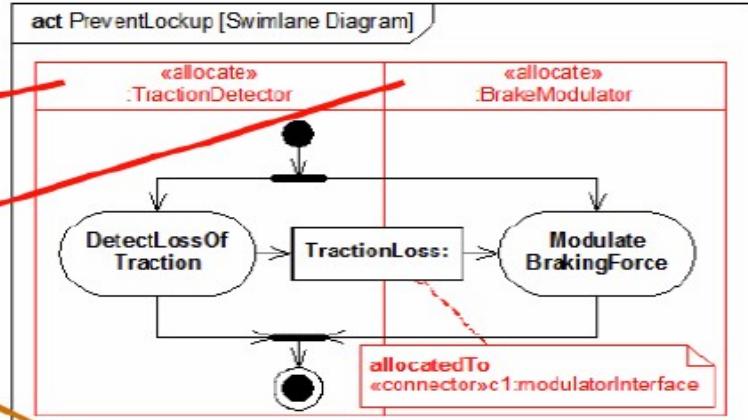
value binding

satisfy

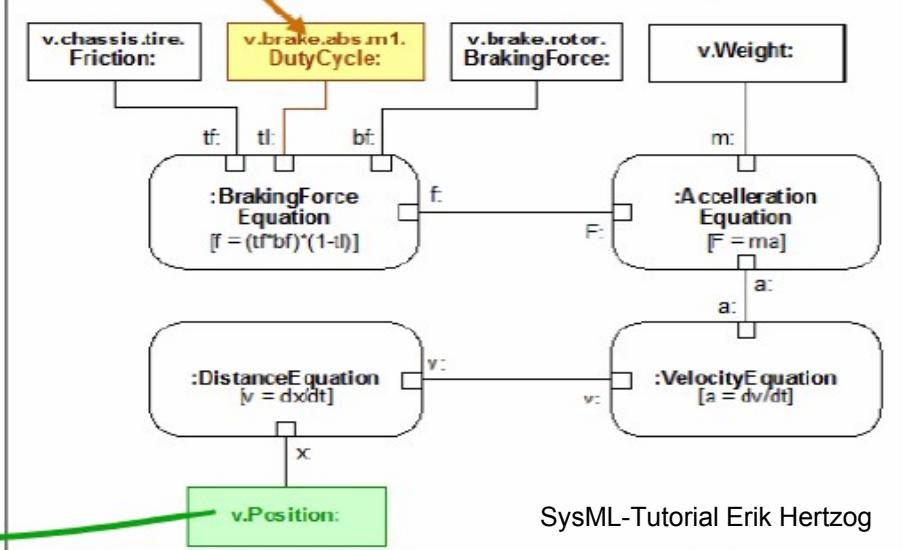


verify

2. Behavior



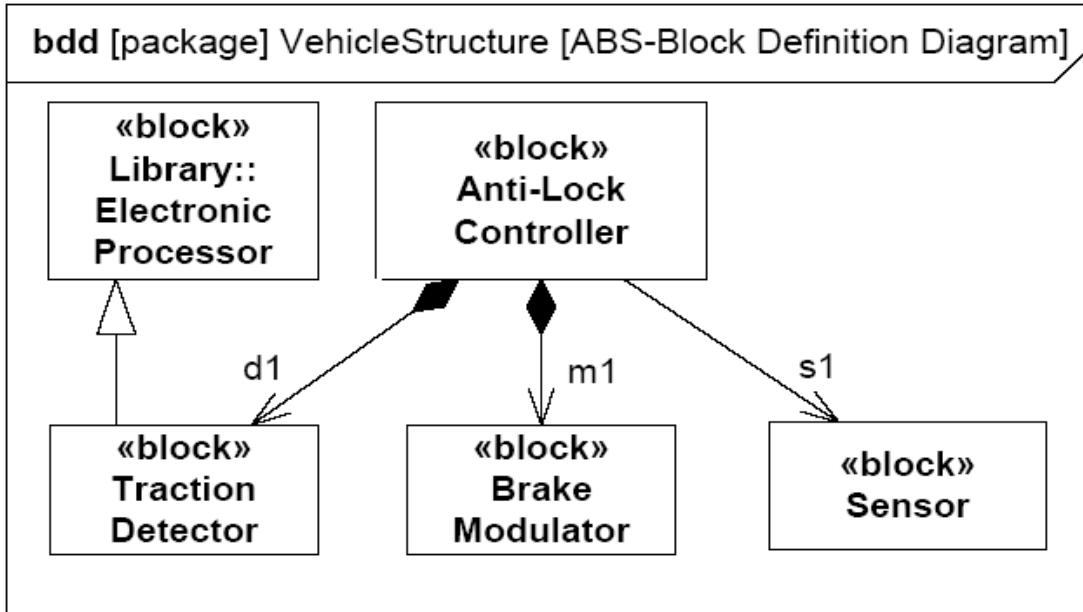
3. Requirements



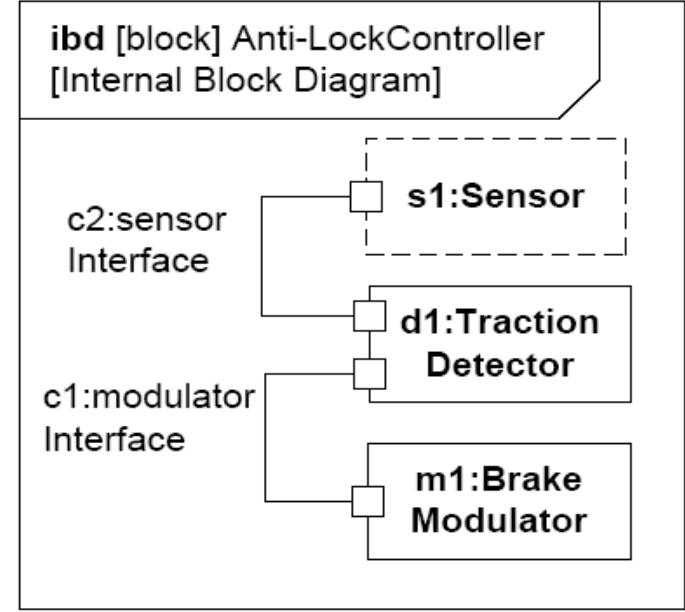
SysML-Tutorial Erik Hertzog

4. Parametrics

Block Definition Diagram



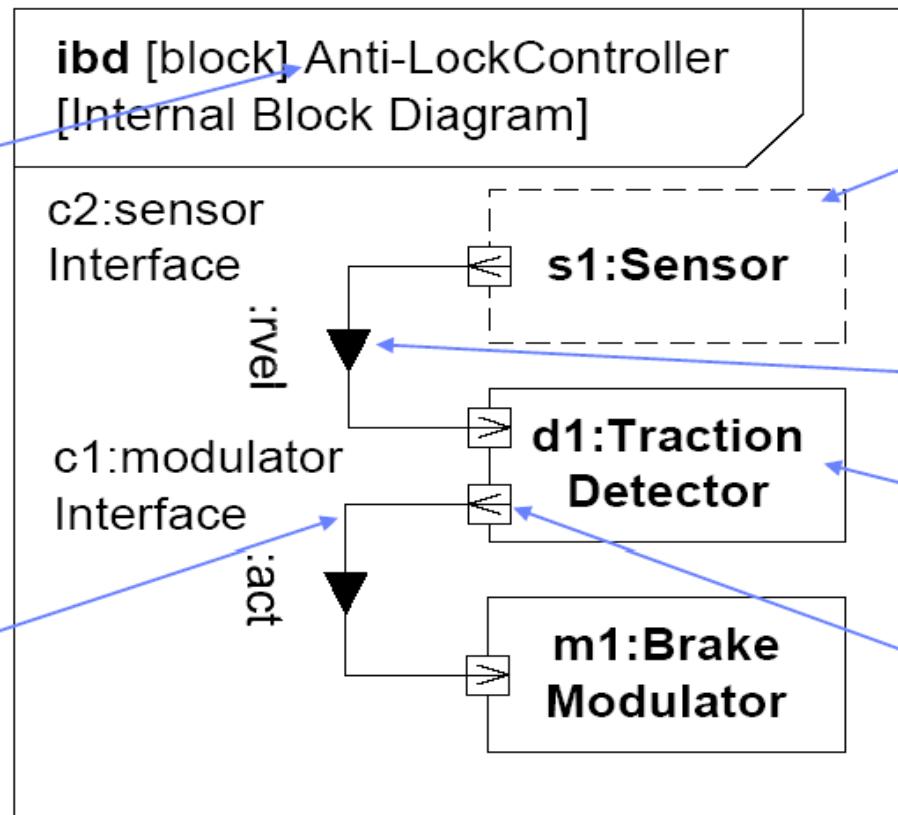
Internal Block Diagram



- **Definition von Bausteinen**
- **Festlegung von Eigenschaften**
- **Beziehungen zwischen Blöcken**
("part" = Benutzung eines Blocks)
- **Sichtbare Schnittstellen**

SysML-Tutorial Erik Hertzog

Enclosing Block



Reference Property
(in, but not of)

Item Flow

Part

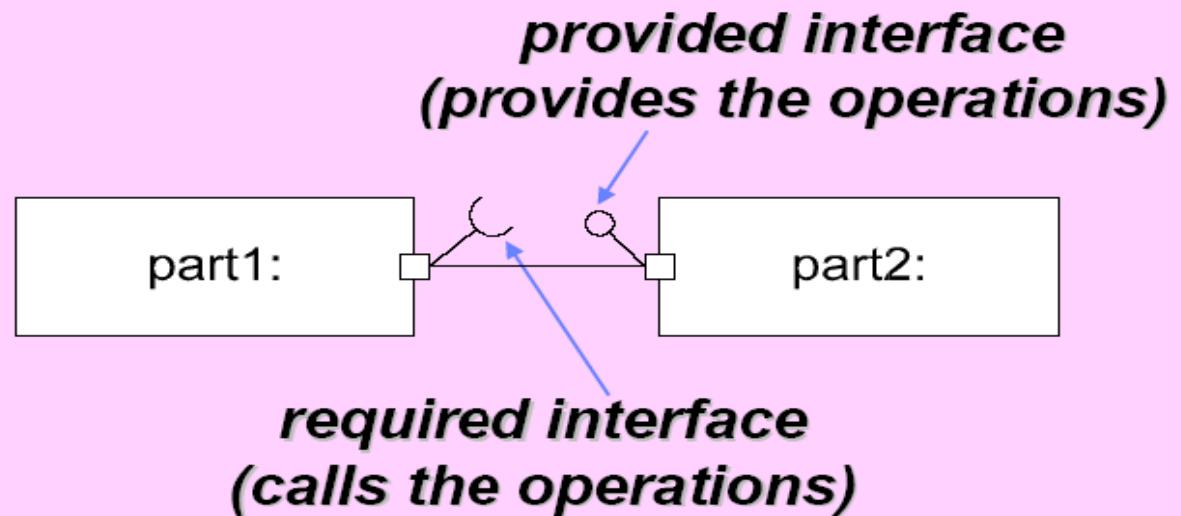
Port

Connector

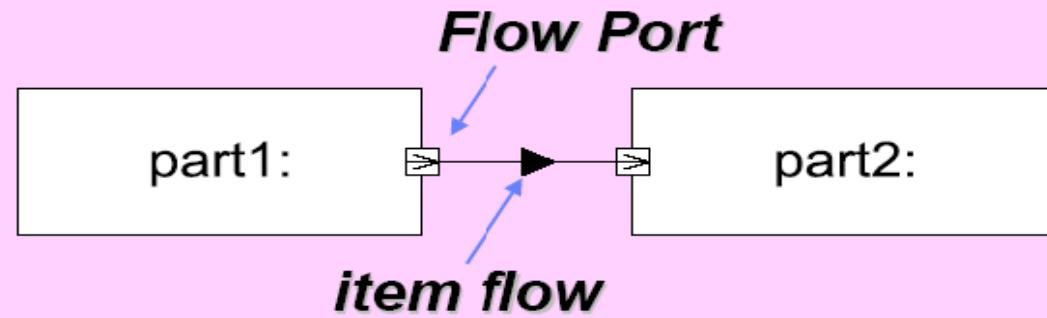
SysML-Tutorial Erik Hertzog

- **Standard (UML) port**
 - The port indicate the existence of a service interface which external blocks may call (as in software)
 - Interaction is as defined for the individual operation made available through the interface
- **Flow ports**
 - Specifies what can flow in or out of a component
 - Has a specified direction and content
 - May be bi-directional

Standard Port

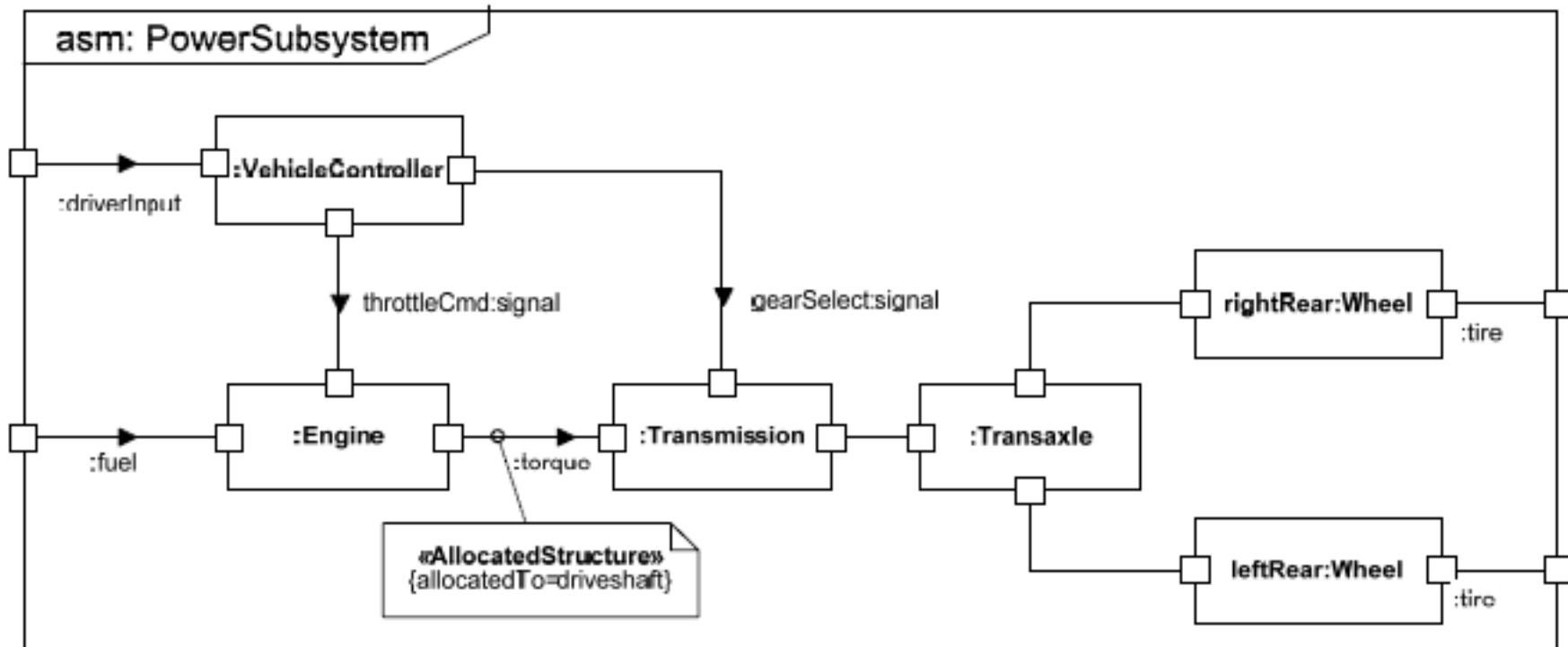


Flow Port



Beispiel Internes Blockdiagramm

7.4.2

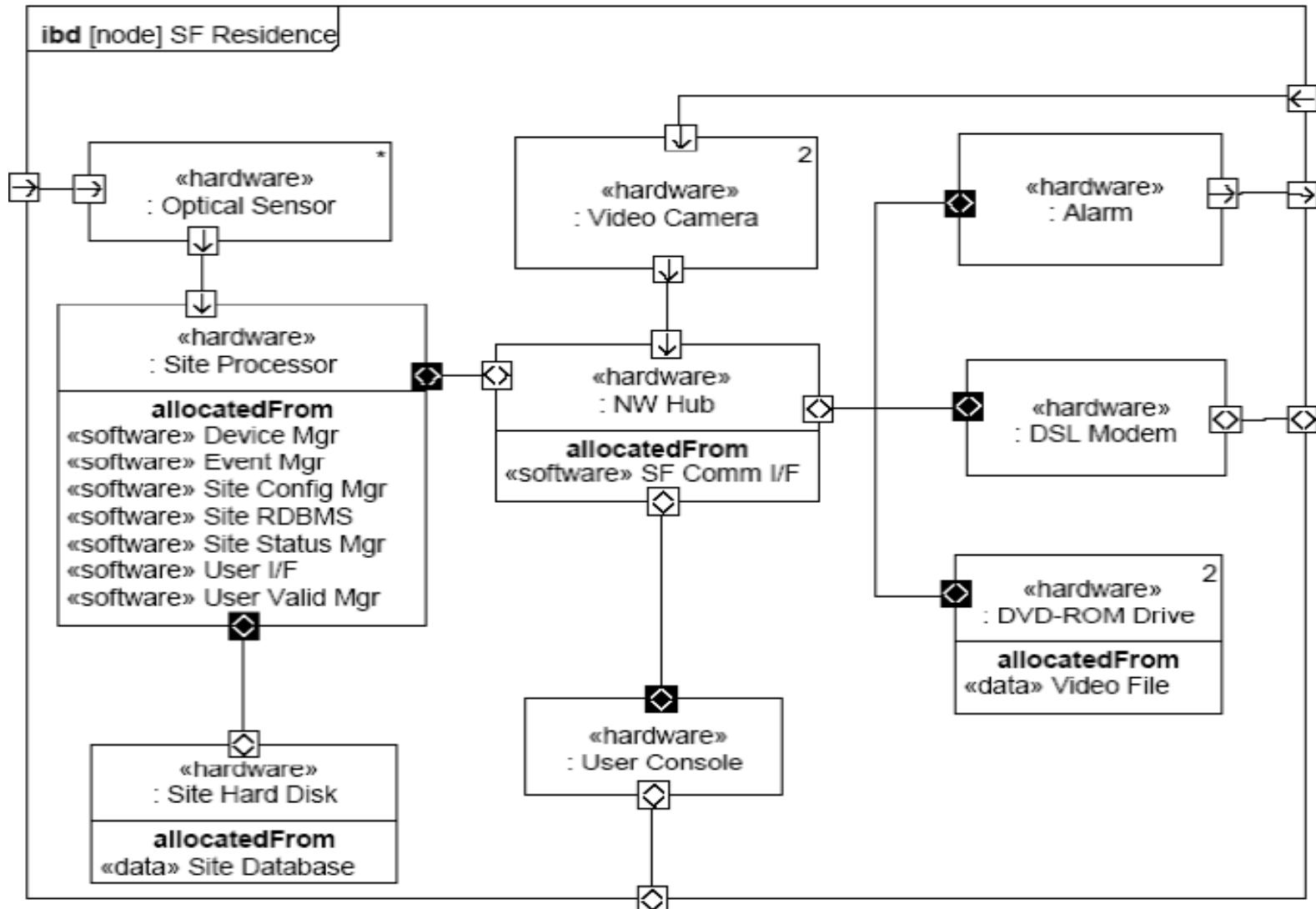


SysML 1.3 OMG 12-06-01.pdf

- **SysML provides 3 mechanisms for representing the allocation of functional or physical elements to other physical elements**
 - **Via Swimlanes in activity diagrams**
 - » **Elegant**
 - **Via the addition of a separate compartment in the block structure**
 - **Via relationships directly on diagrams**

Beispiel Allocation

7.4.2

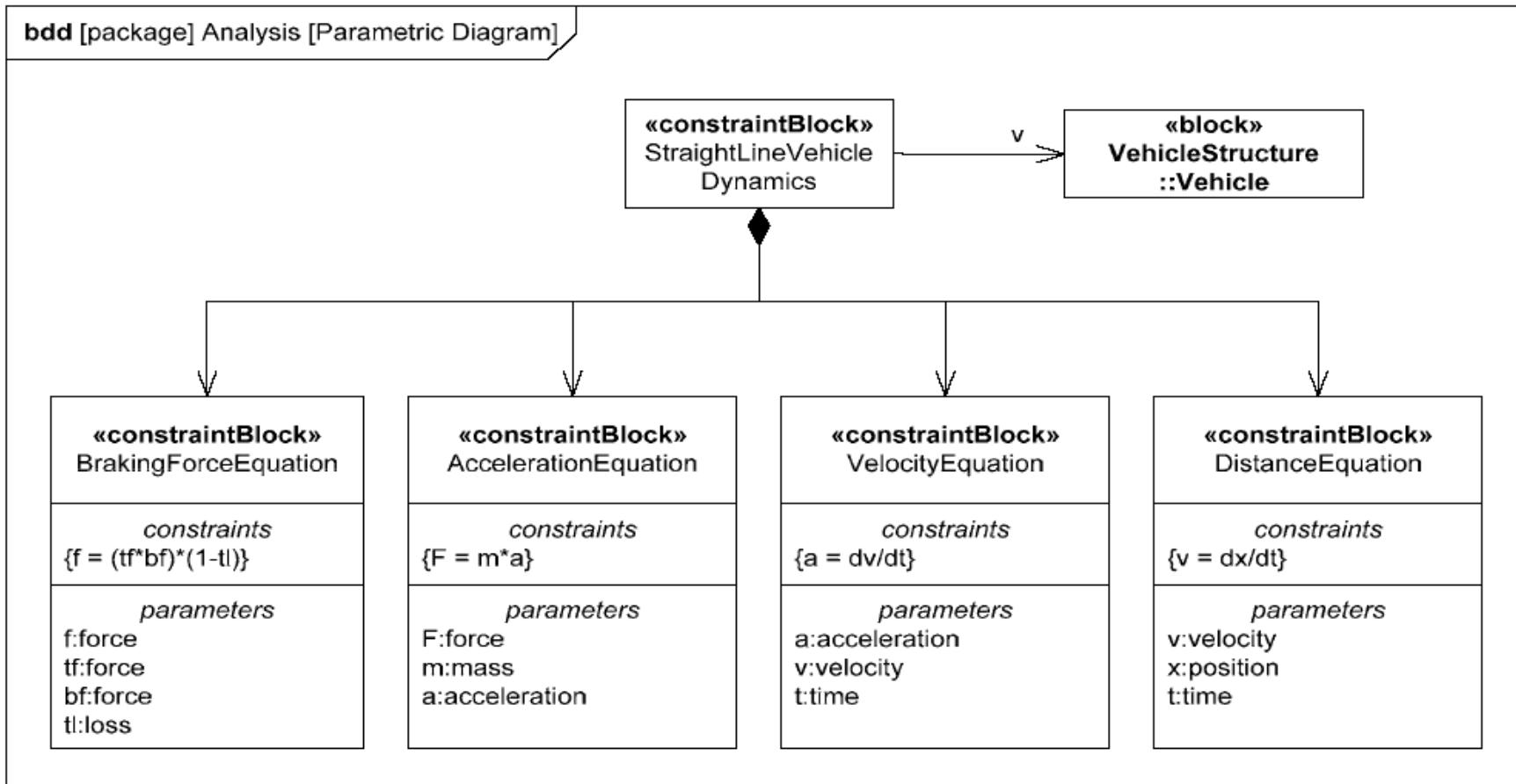


SysML-Tutorial Erik Hertzog

- **Genutzt für Constraints zwischen quantifizierbaren Eigenschaften**
- **Definiert als Stereotype**
 - **Expression = Text String**
 - **Sprache für Expressions offen (informell, OCL, MathML)**
 - **Auswertung durch externe Tools (nicht durch SysML)**

Beispiel Parametrische Constraints

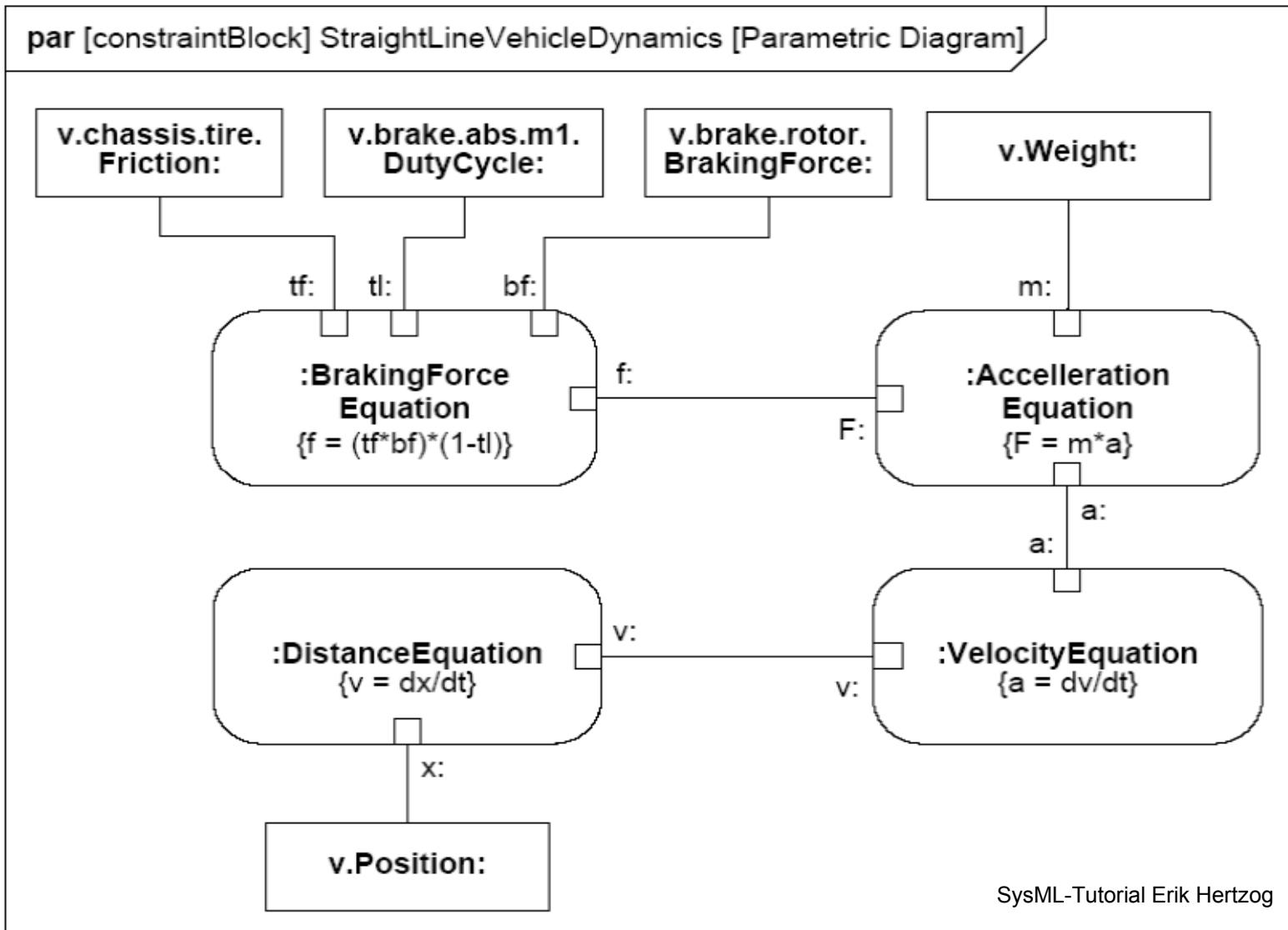
7.4.2



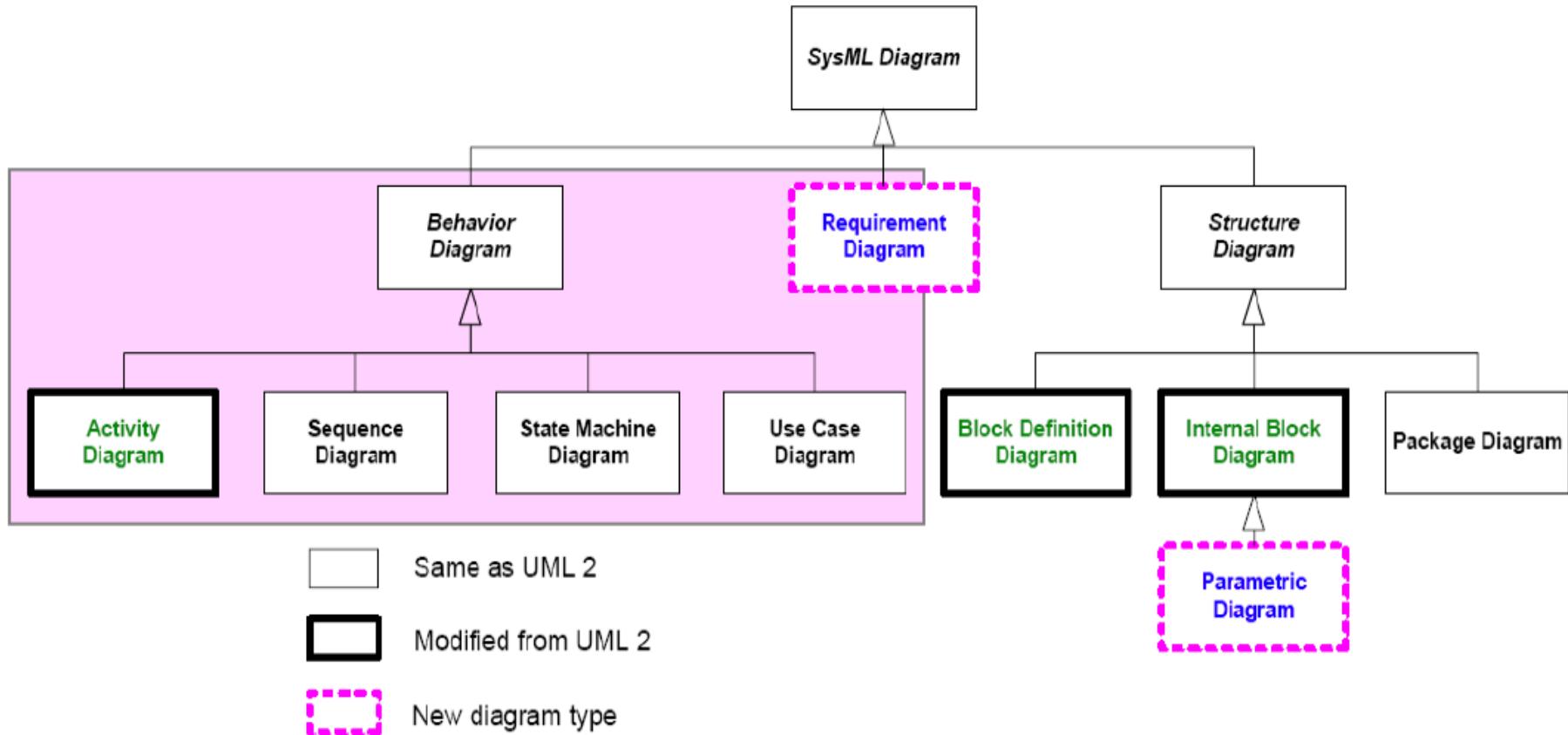
SysML-Tutorial Erik Hertzog

Beispiel Variablen-Bindung

7.4.2

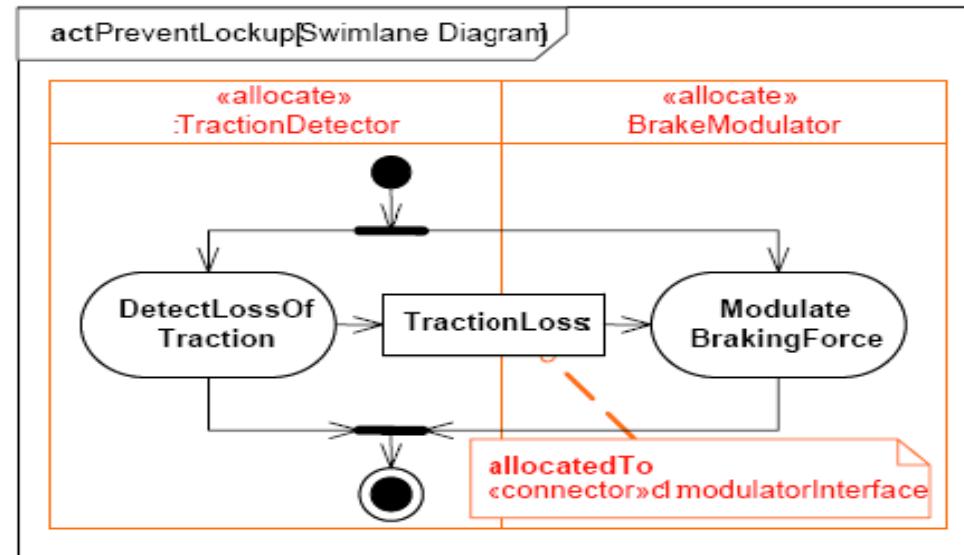
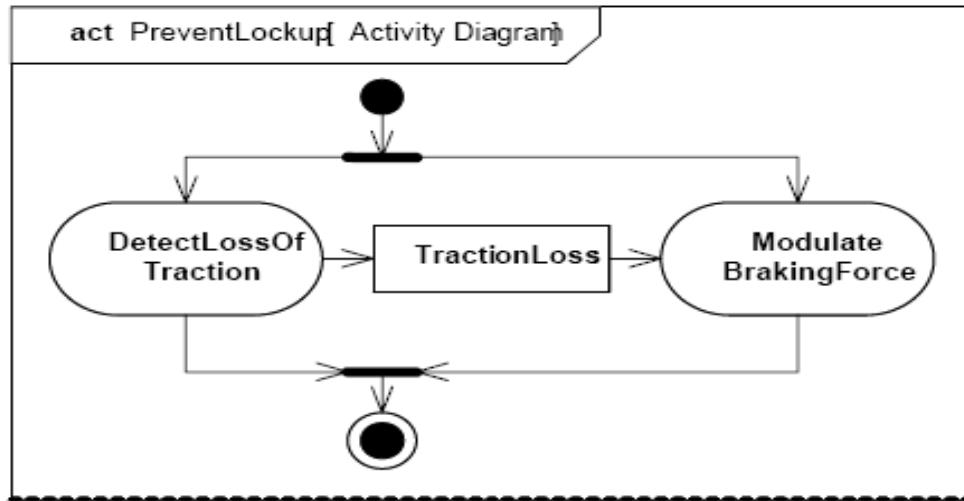


SysML-Tutorial Erik Hertzog



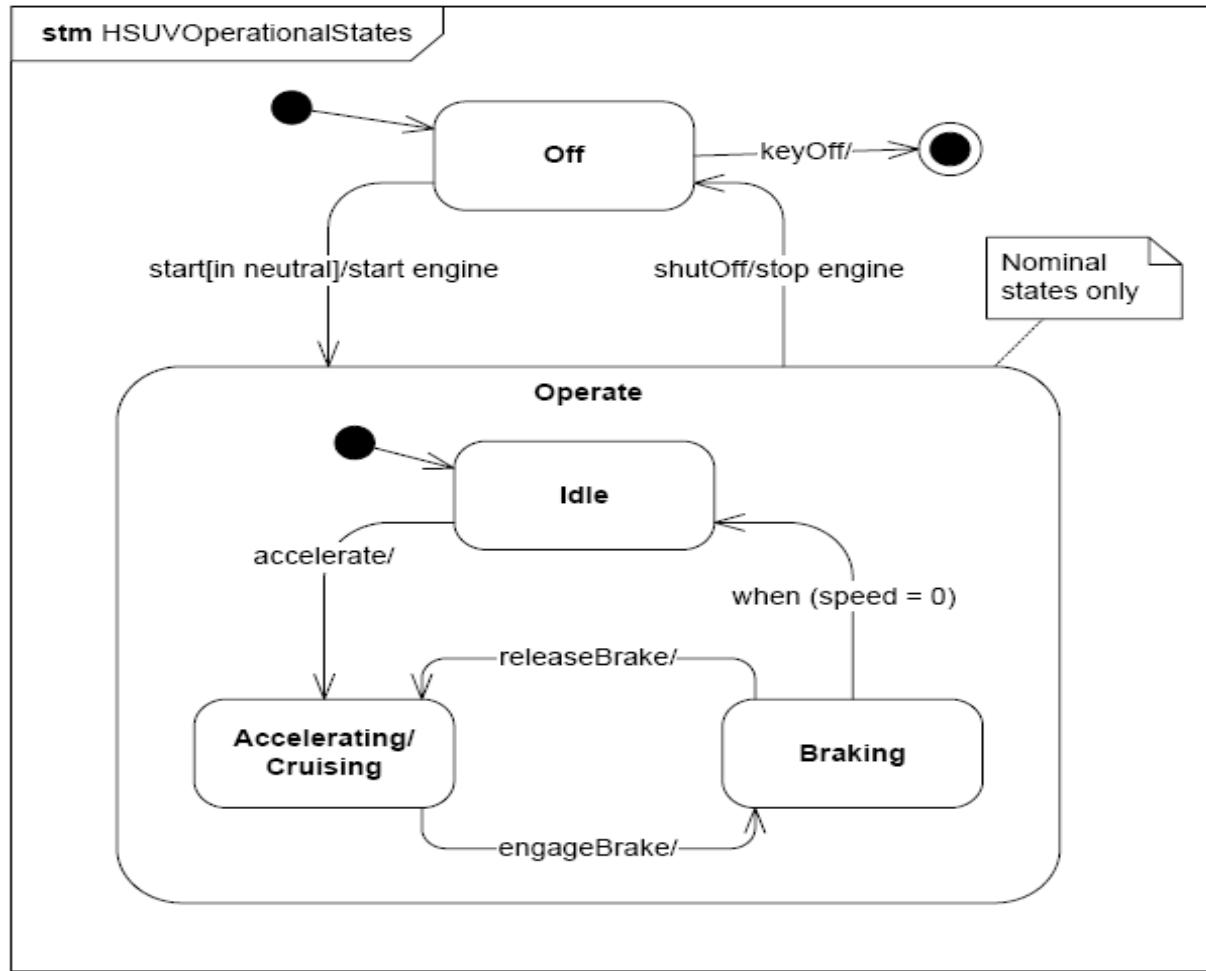
Aktivitätsdiagramme

7.4.2



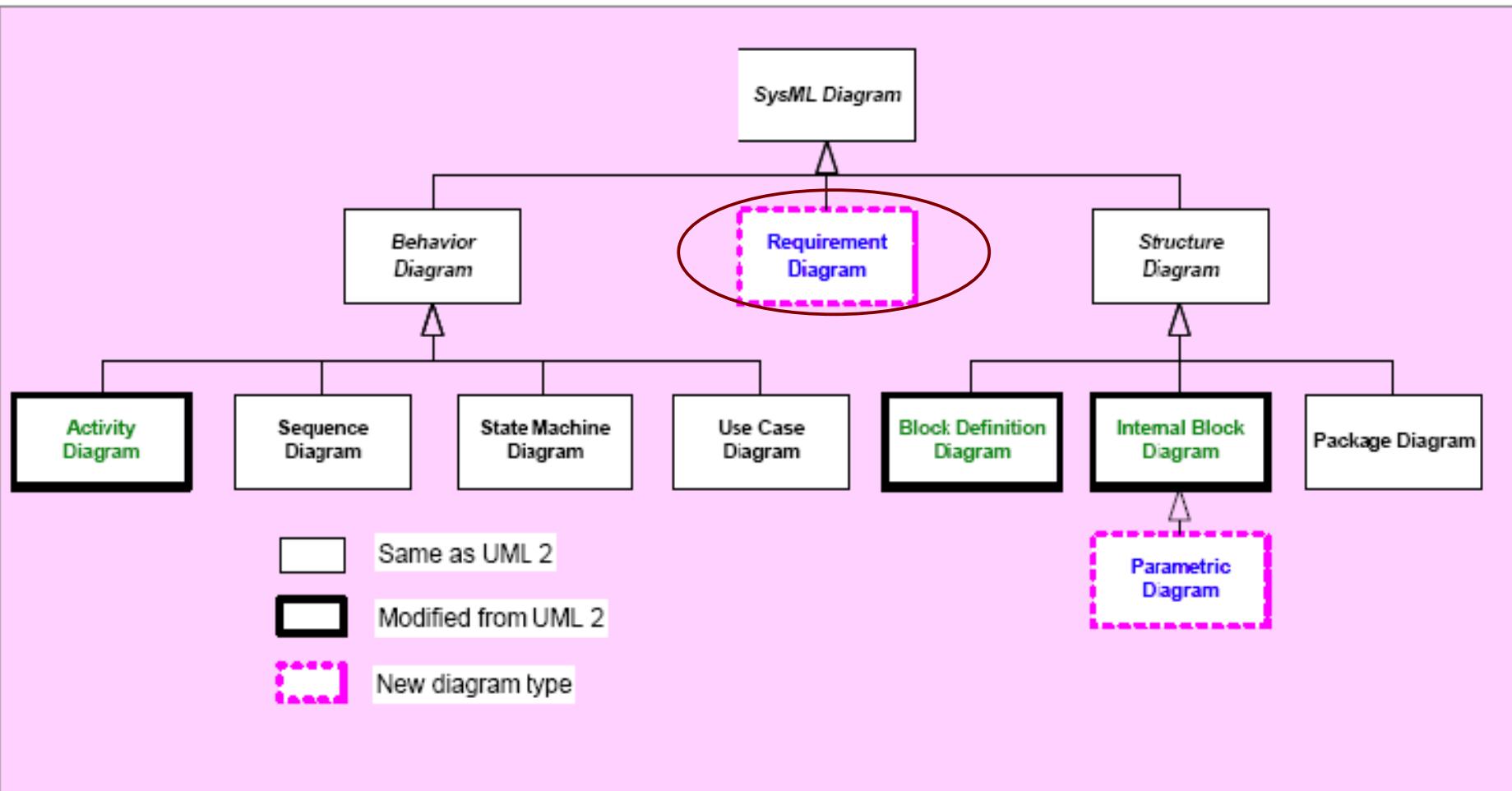
SysML-Tutorial Erik Herzog

- Beispiel

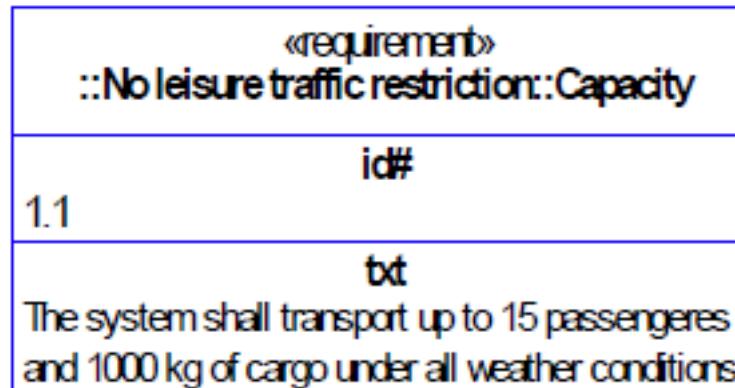


Requirements-Diagramm

7.4.2



- In SysML ist Requirement eine Zeichenkette (String)
- Keine Annahmen darüber, wann Requirement Elemente im Entwurfsprozess zu verwenden sind
- Notation:



- **SysML provides the following features**
 - Representation of requirements
 - » Representation of individual *requirements*
 - » Requirement composition
 - » Requirements can be sub-classed using specialization
 - Requirement relationships
 - » derive relationship between derived and source requirements
 - » satisfy relationship between design models and requirements
 - » verify relationship between requirements and test cases
 - » generalized trace relationship between requirements and other model elements
 - » rationale for requirements traceability, satisfaction, etc
 - Alternative graphical, tabular and tree representations
 - » Supported by the standard, but currently not implemented in any tools

- Überblick

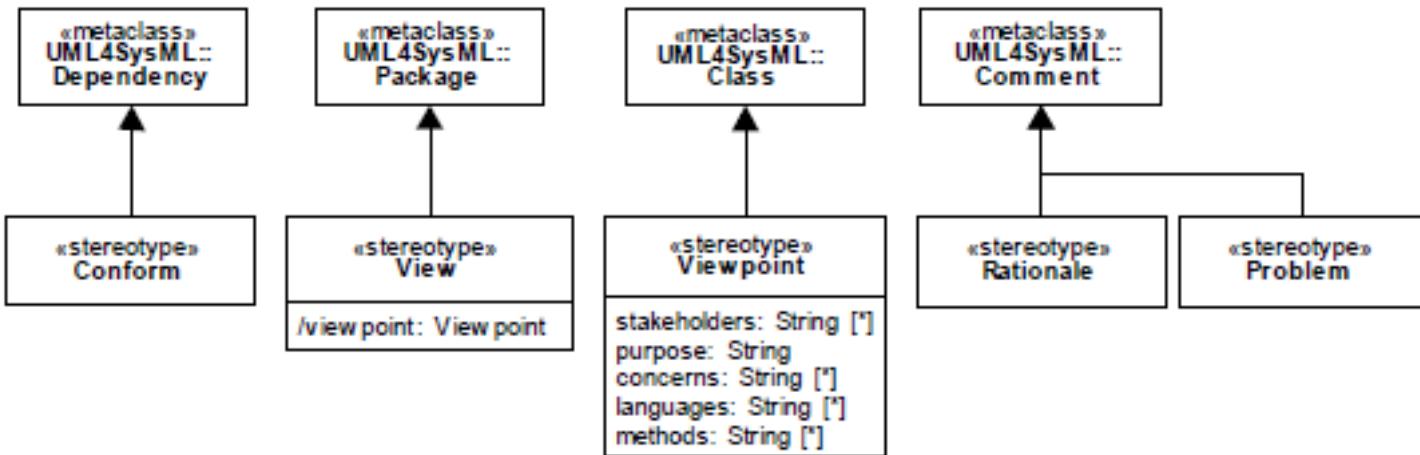


Figure 7.1 - Stereotypes defined in package ModelElements

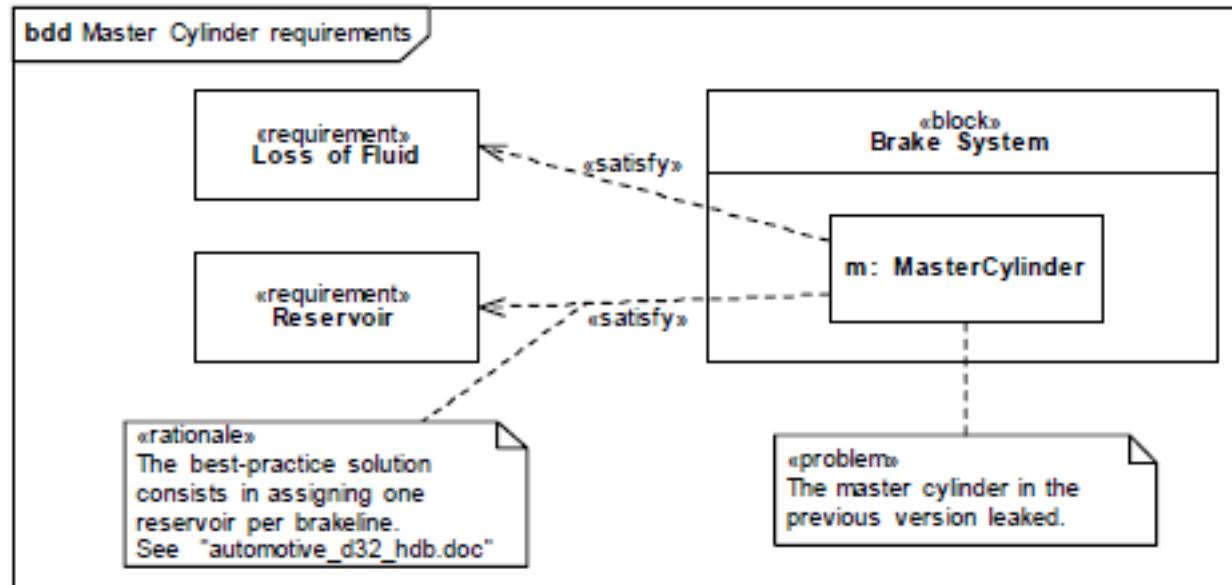


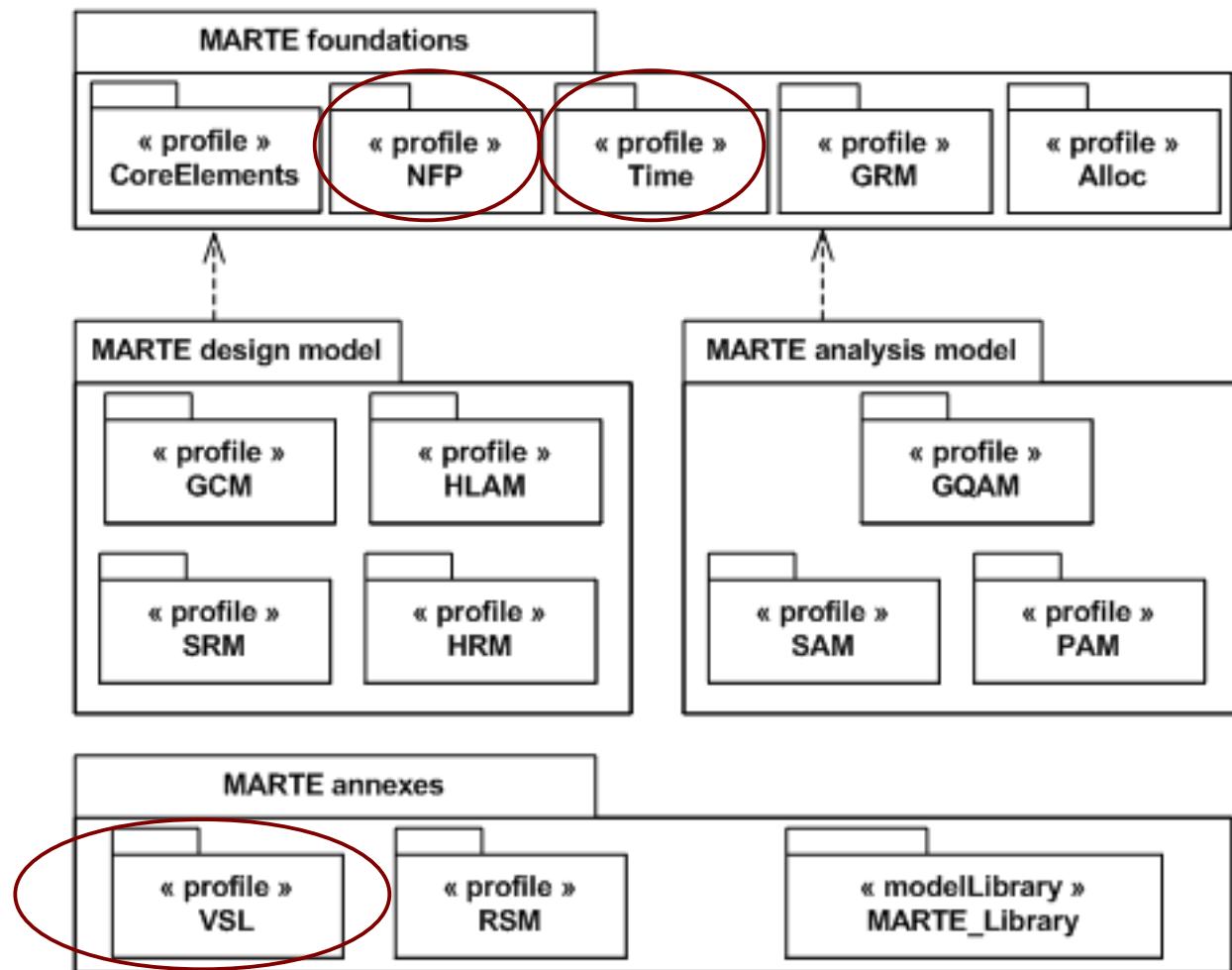
Figure 7.2 - Rationale and Problem examples

7.4.3 MARTE

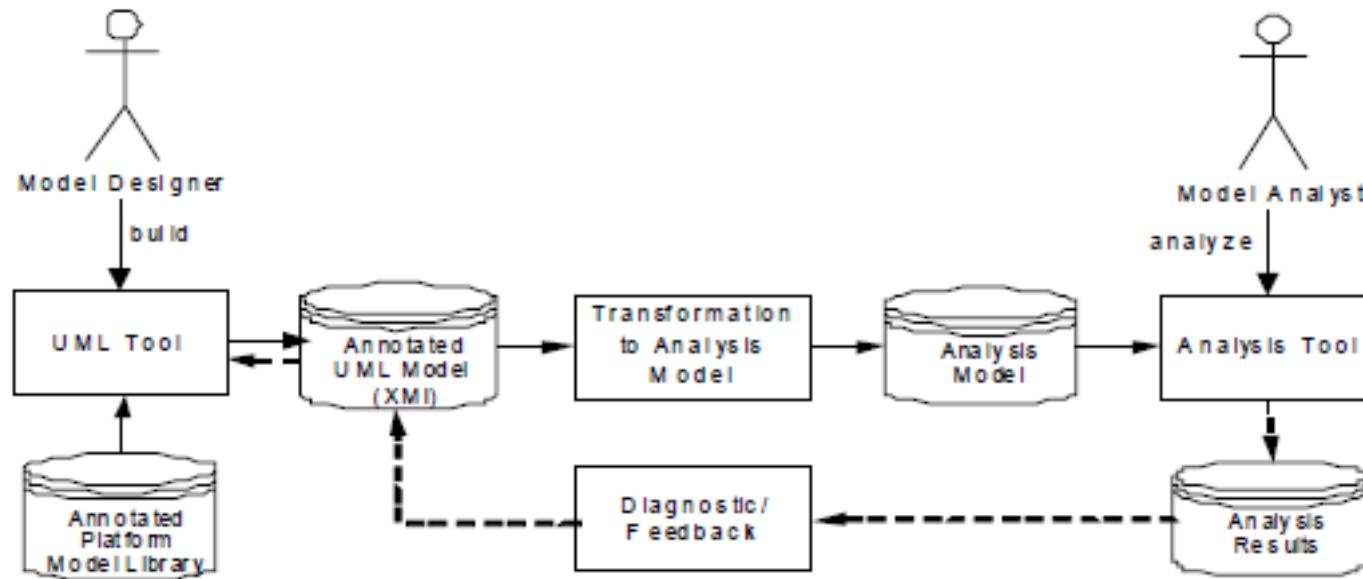
- **MARTE: Modeling and Analysis of Real Time and Embedded systems**
- **OMG Standard formal/2011-06-02 (Version 1.1)**
- **UML 2 Profil**
- **Unterstützung für Modellierung, Simulation und statische Analyse**
- **Zahlreiche MARTE Modell-Bibliotheken**
 - Primitive Typen
 - Erweiterte Datentypen
 - Maßeinheiten
 - Vordefinierte NFP-Typen
 - Time
 - Echtzeitbetriebssysteme (OSEK/VDX, Arinc-653 (Avionik))
 - ...

- **Core Framework zur Definition der Basiskonzepte**
 - Core Elements
 - Non-Functional Properties (NFP) Modeling
 - Time Modeling (Time)
 - Generic Resource Modeling (GRM)
 - Allocation Modelling (Alloc)
- **Refinement 1: Reine Applikationsmodellierung
(z.B. Hardware and Software Plattform Modellierung)**
- **Refinement 2: Unterstützung für quantitative Analyse
of UML2-Modellen, insb. Schedulability und
Performance-Analyse**
- **Anhänge**
 - insb. Value Specification Language (VSL)
 - Sprache zum Zusammenfügen der Modellbibliotheken

Überblick



UML MARTE OMG 11-06-02.pdf



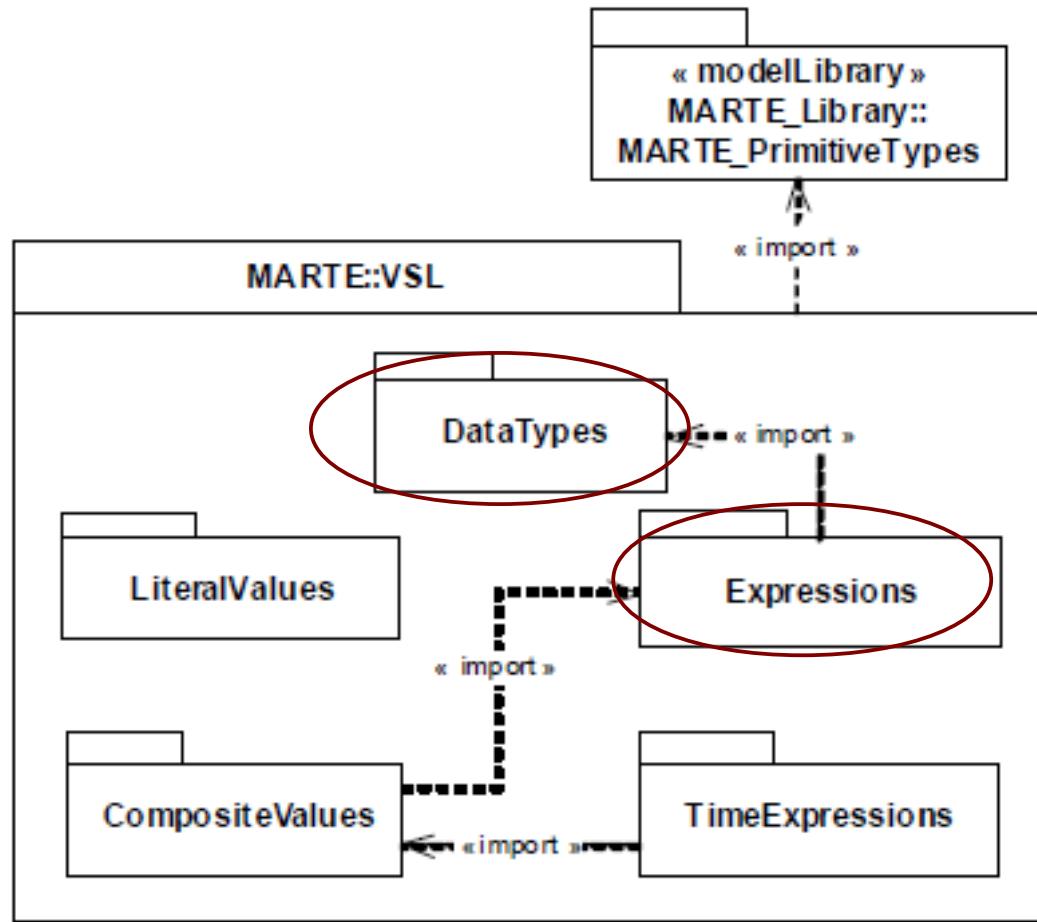


Figure B.1 - Structure of the VSL framework

UML MARTE OMG 11-06-02.pdf

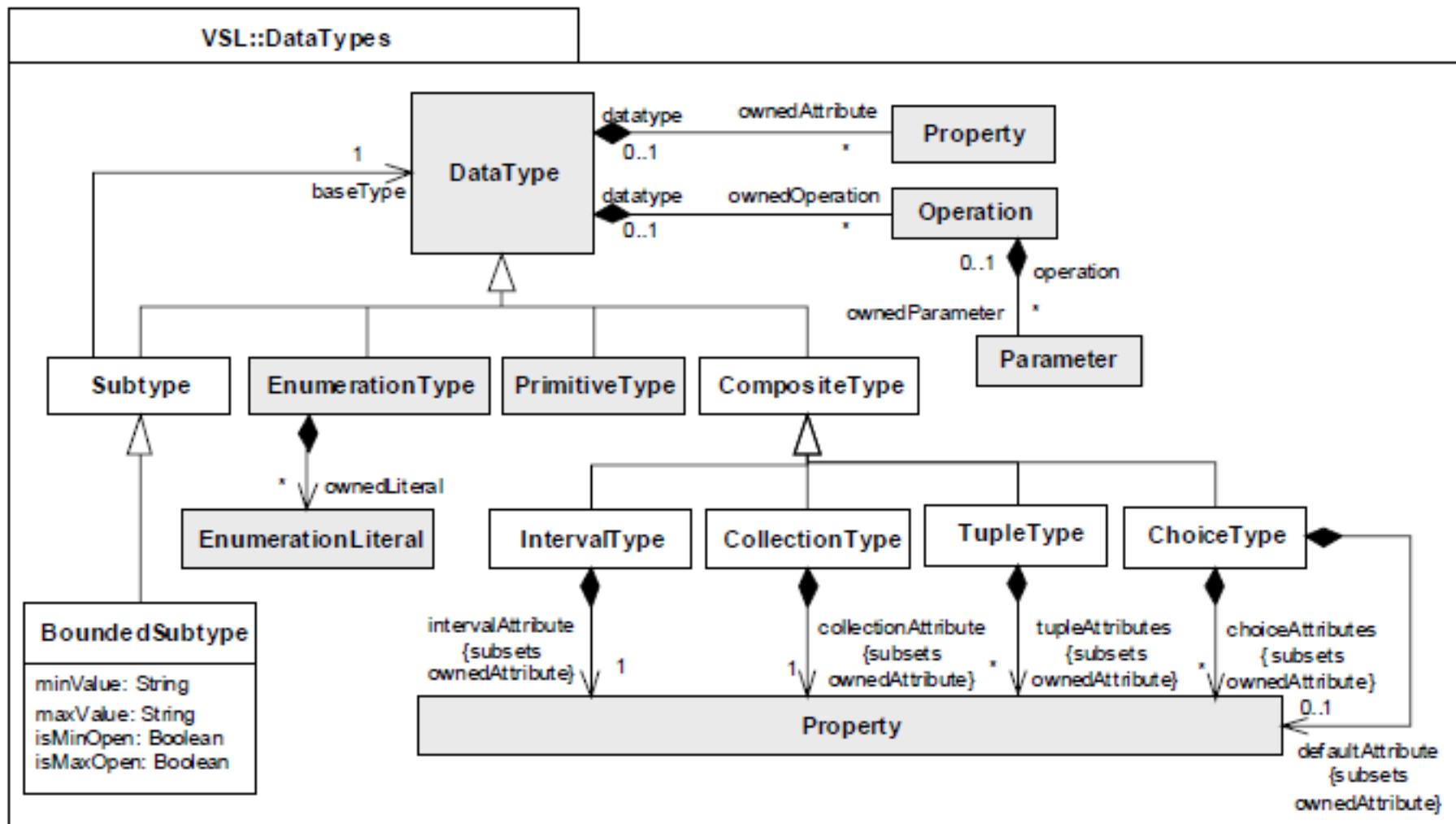


Figure B.2 - VSL::DataTypes package

UML MARTE OMG 11-06-02.pdf

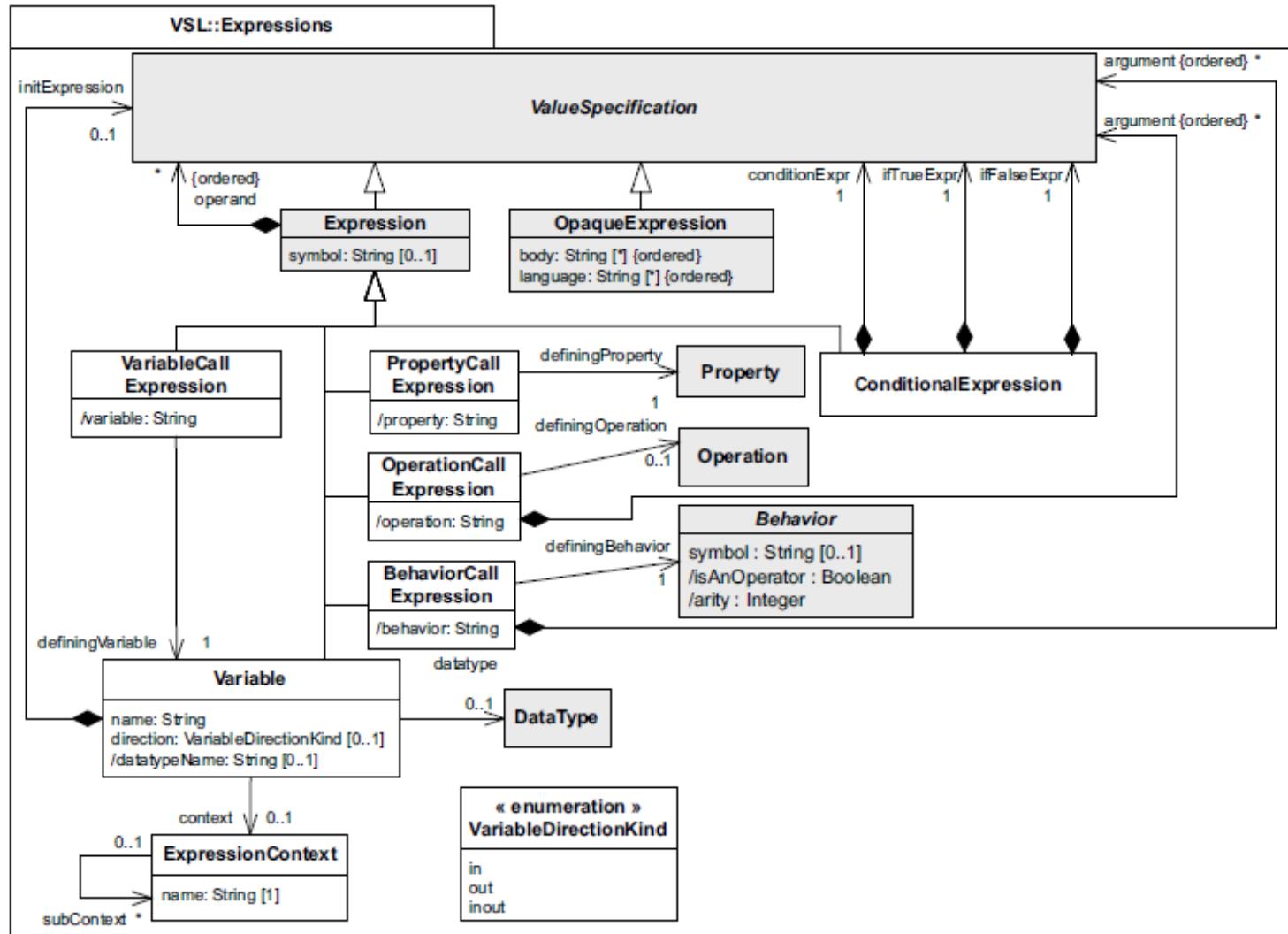


Figure B.4 - VSL::Expressions package

Non-Functional Properties (NFC) Modelling 7.4.3

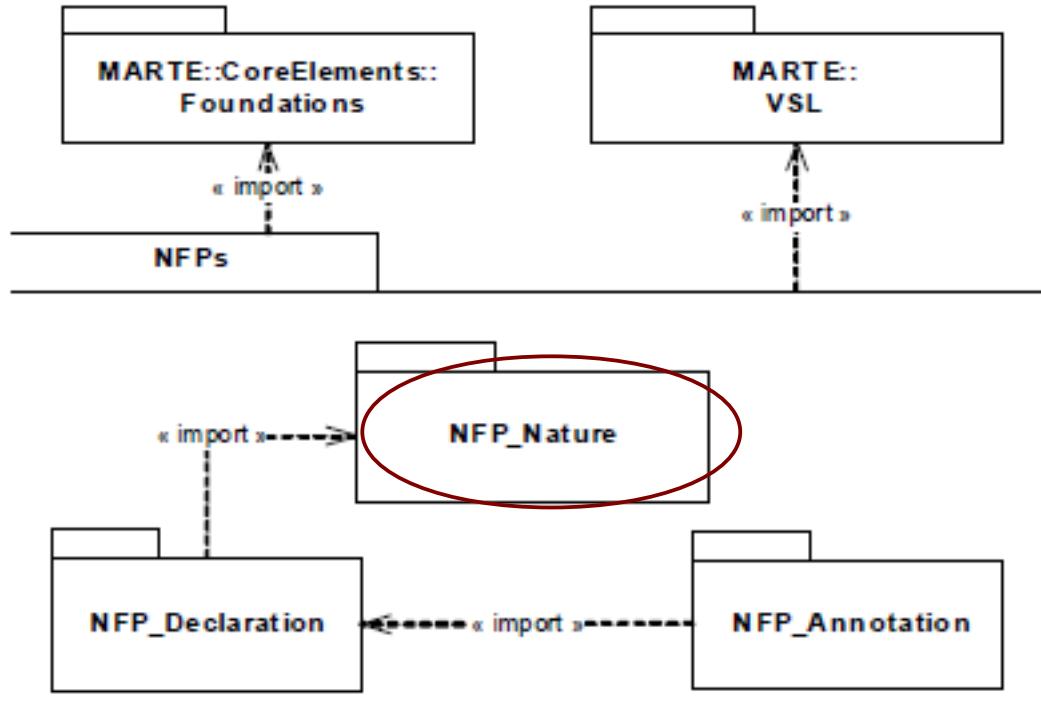


Figure 8.1 - Structure and dependencies of the NFPs modeling package

Non-Functional Properties (NFC) Modelling (2) 7.4.3

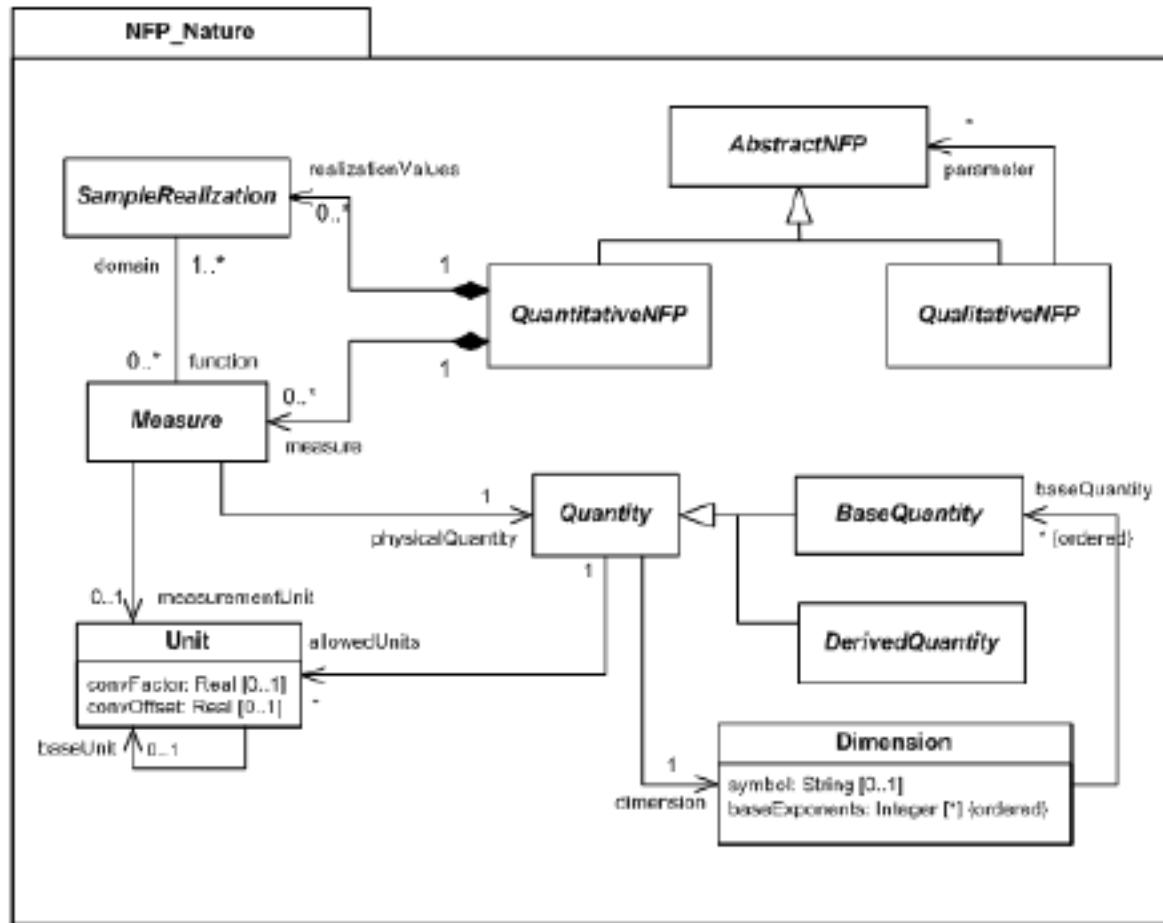


Figure 8.2 - Domain Model for NFP Nature

UML MARTE OMG 11-06-02.pdf

- Überblick

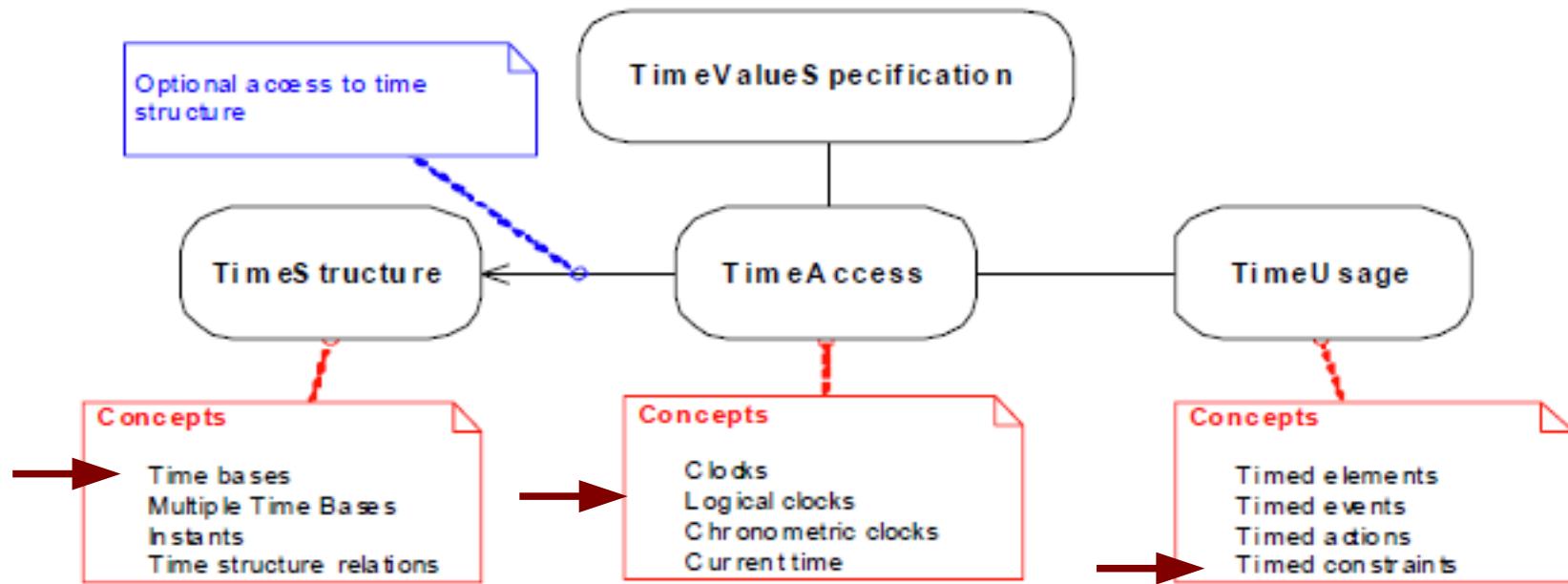


Figure 9.1 - Overview of the time model concerns

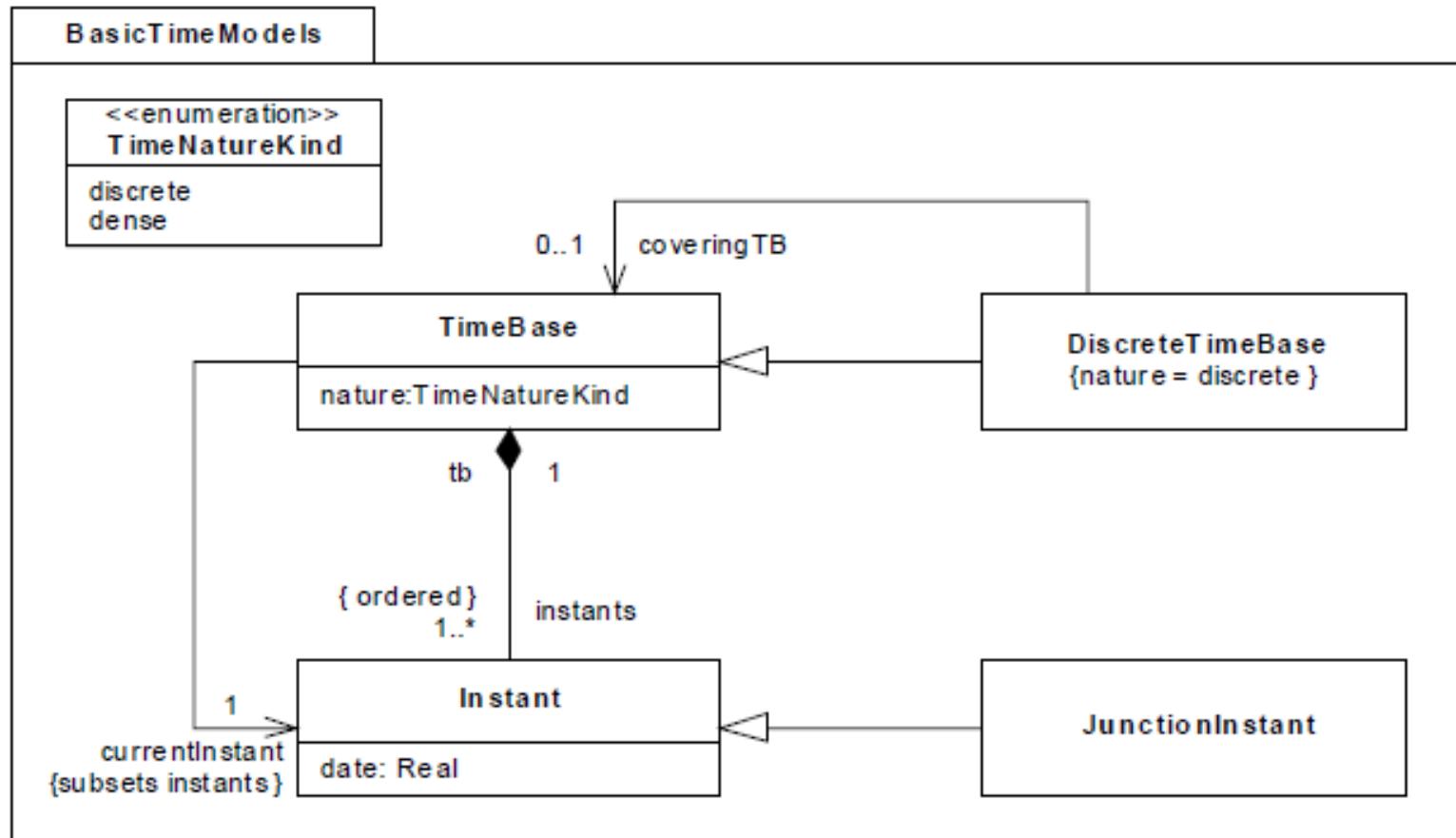


Figure 9.3 - Basic time diagram of the time model

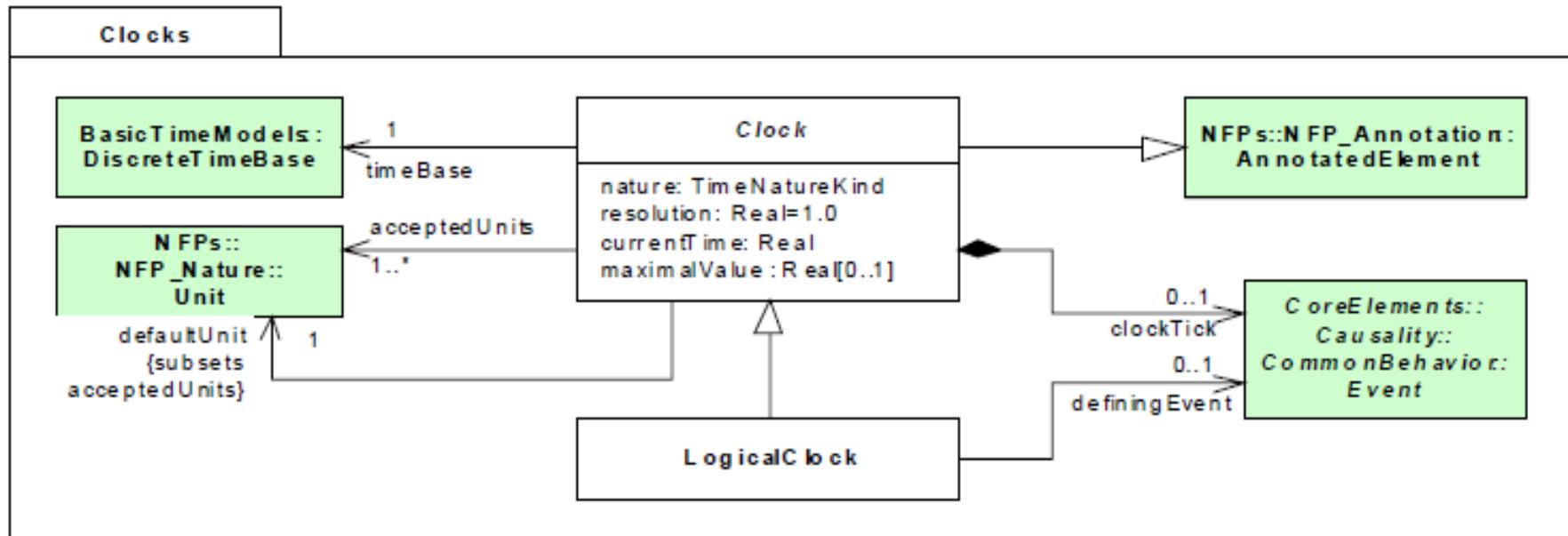


Figure 9.9 - Clocks diagram of the time model

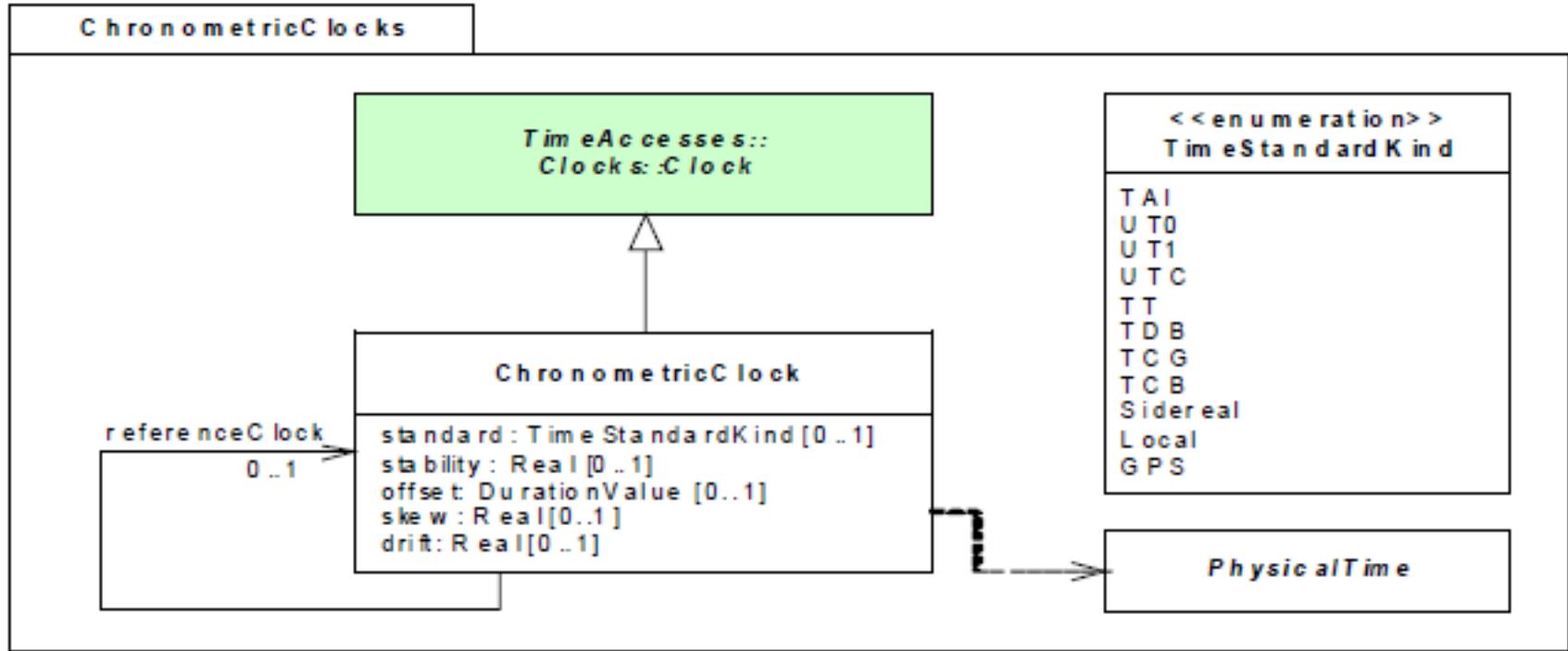


Figure 9.12 - ChronometricClocks diagram of the time model

- **Beispiel**

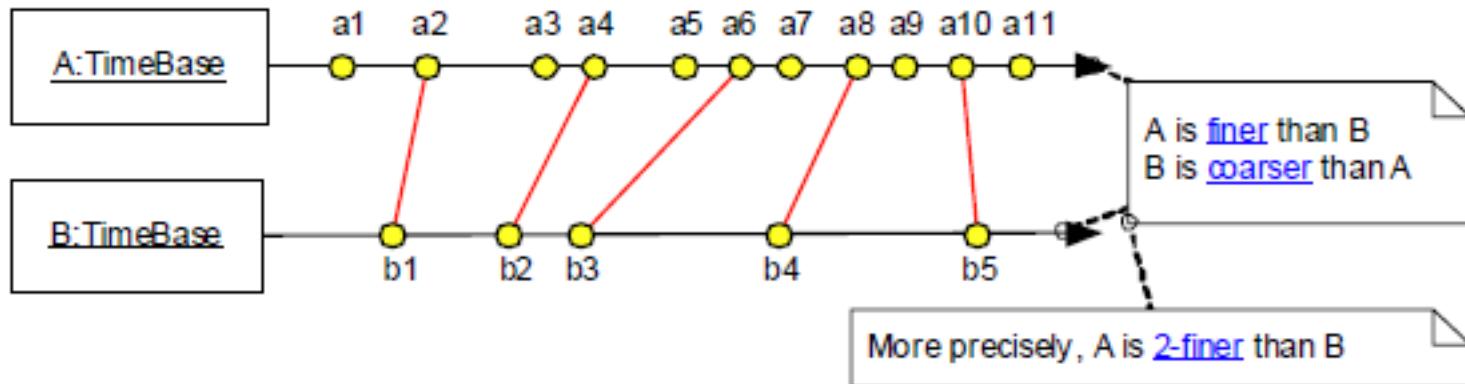


Figure 9.7 - Example of time relations between two time bases

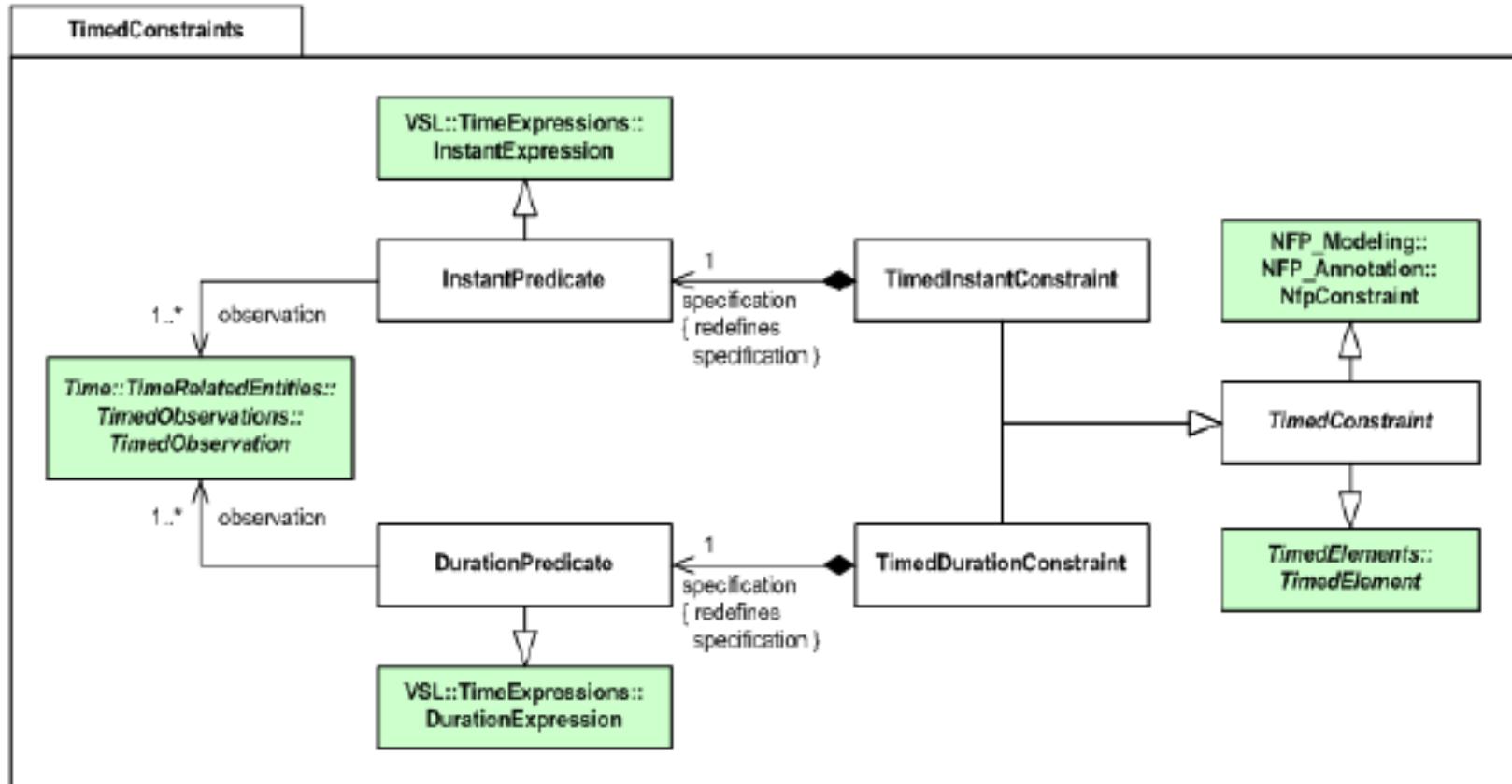


Figure 9.18 - TimedConstraints diagram of the time model

- Zusammenfassendes Beispiel

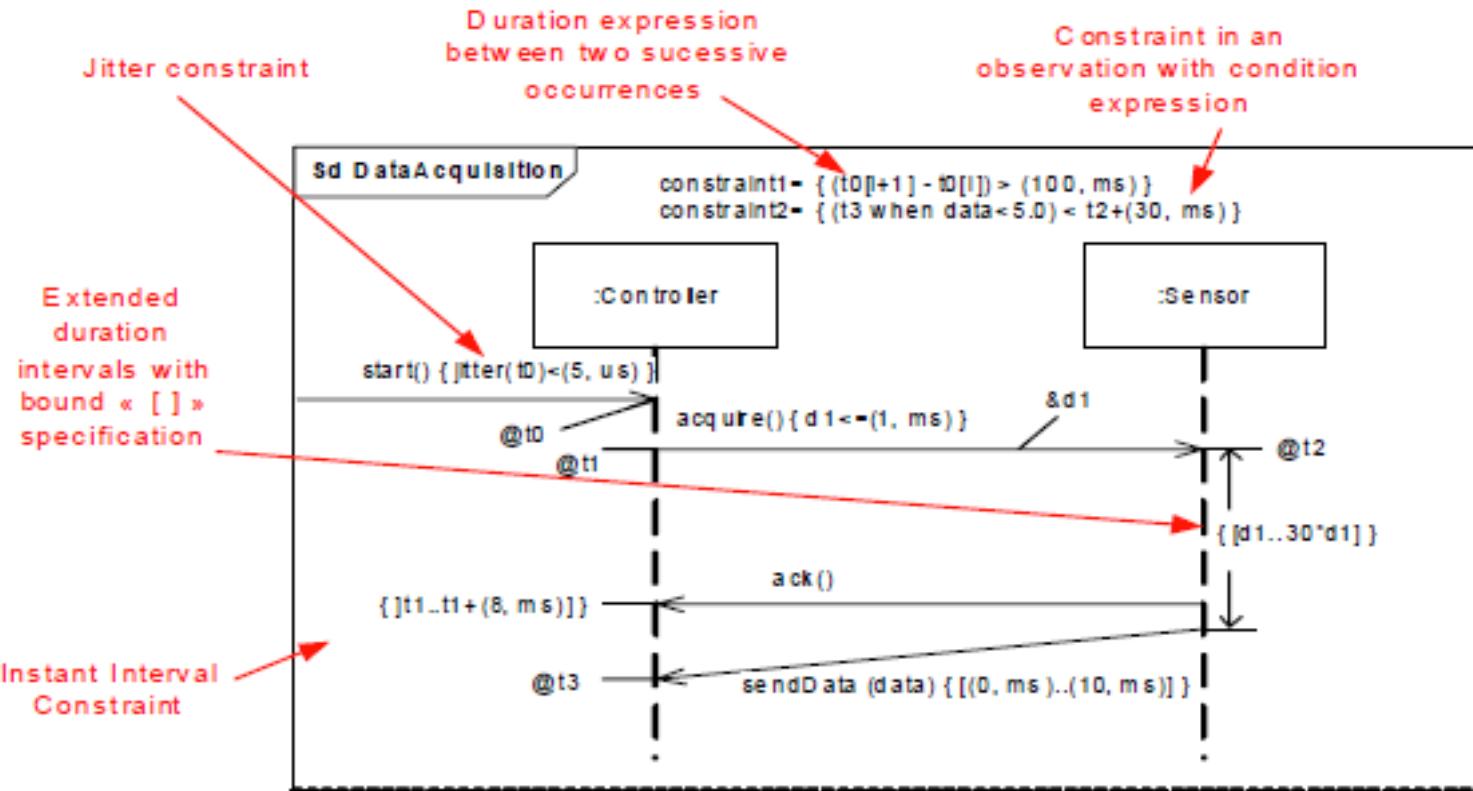


Figure B.14 - Time Expressions and Constraints in Sequence Diagrams with VSL

Kap. 7:

Entwicklung von Anwendungen

(c) Validierung & Test

7. Entwicklung von Anwendungen

Konzentration auf

- Sicherheitskritische Anwendungen
- Echtzeitanwendungen

Gliederung

(a)

1. Einführung
2. IEC EN 61508
3. Programmiersprachen (MISRA C/C++, Ada)

(b)

4. Modellierung (SysML, UML MARTE, ...)

(c)

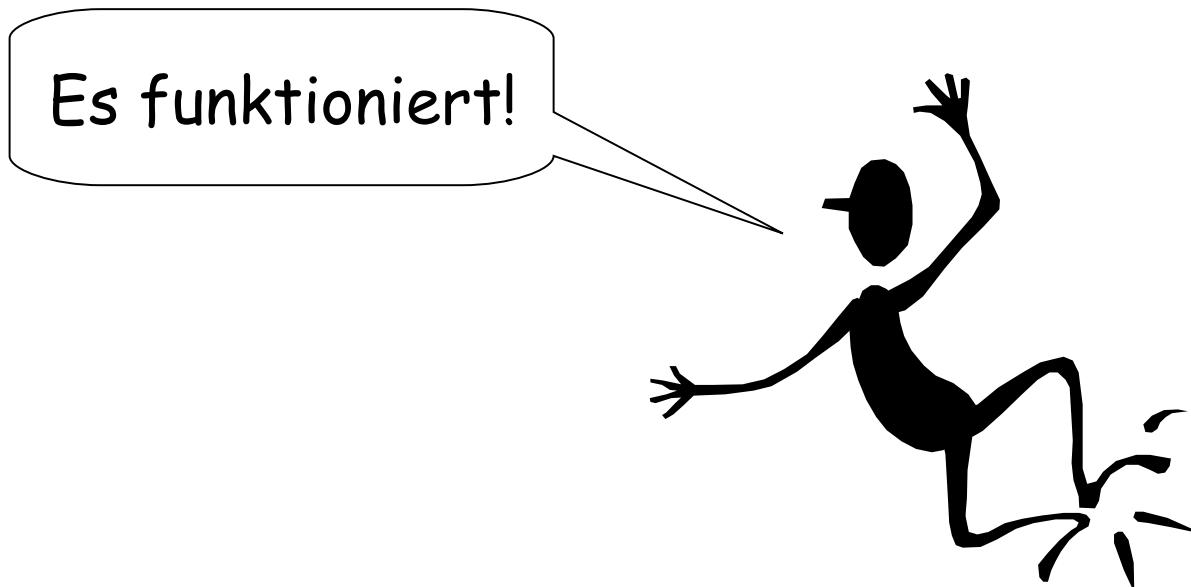
5. Validierung
6. Systematisches Testen

7.5 Validierung

- Häufiger Einsatz von Echtzeitsoftware in sicherheitskritischen Bereichen (z.B. ABS, Avionik, ..)
- Je nach Einsatzgebiet existieren Vorschriften für die Zertifizierung von Software
- z.B.:
 - DO-178A (FAA)
 - IEC 61508

7.6 Systematischen Testen

- **Ad-hoc Methoden...**



- ... sind unzureichend!
 - Eher „Debugging/Inbetriebnahme“ als „Test“
 - Qualität ist nicht nachvollziehbar
 - Entwickler sind schlechte Tester (ihrer eigenen Programme)

<http://www.fbe.hs-bremen.de/spillner/systtest>

- **Ziel von Tests (bzw. der Validierung):**
 - Nachweis der Existenz von Fehlern in einem Programm.
 - Erfolgserlebnis des Testers: „Fehler gefunden“.
- **Ziel der Verifikation:**
 - Nachweis der nicht-Existenz von Fehlern in einem Programm.
 - Erfolgserlebnis des Verifizierens: „Beweis der Funktion“.
- **Problem bei beiden: Codekomplexität**
 - Hoher Aufwand bei Validierung
 - Verifikation komplexer Gewerke (heute noch) unmöglich

- **Geordnete Abfolge von Testfällen**
- **muss jederzeit reproduzierbare Ergebnisse liefern**
- **automatisierter Ablauf (keine Benutzerinteraktion)**
- **modular aufgebaut -> erweiterbar**
- **z.B. shell-Skript, spezielles Anwenderprogramm,
vordefinierte Abfolge von Benutzereingaben,**
- **Vorzugsweise realisiert mit Hilfe eines „Test Harness“
(z.B. DejaGNU)**

- **Testen auf Rückschritte (engl. regression)**
- **Es kommt immer wieder zu Rückschritten im Zuge von Weiterentwicklung/Pflege:**
 - Fehler bei Hinzufügen neuer Funktionalität
 - „verschlimmbessern“
 - Fehler im Umgang mit Revisionskontrollwerkzeugen
 - Beheben von Symptomen anstelle der Ursachen
 - Schlechte Kommentierpraxis im Team
- **Testsuite als Bestandteil des Produkts (Weiterpflege):**
 - Neue Funktionen im Produkt: neue Testfälle
 - Fehler im Produkt:
 - 1. Neuer Testfall, der den Fehler reproduziert
 - 2. Fehler beheben
 - 3. Testfall verbleibt in der Testsuite

- **Unit Test**
 - Schwierigkeit/Unmöglichkeit, bei komplexen Gewerken Programmteile in tieferen Schichten über die Programmierschnittstelle gezielt anzusprechen
 - Aufteilen des Gewerkes in kleinere Module (*Units*)
 - evtl. emulieren der externen Schnittstellen durch „stubs“
 - Test der einzelnen Units
- **Integrationstest**
 - Gemeinsames Testen der Units im Zusammenspiel
 - Haupt-Augenmerk dabei auf Schnittstellen
- **Systemtest**
 - Testen des gesamten Systems

- **Schrittweises Entwickeln von Testfällen ausschließlich anhand der Funktionsbeschreibung („Black-Box“-Test)**
- **Eigenschaften werden Punkt für Punkt (in Form einzelner Testfälle) überprüft**

```
READ(2)                      UNIX Programmer's Manual
READ(2)

NAME
    read - read from a file descriptor

SYNOPSIS
    #include <unistd.h>

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

DESCRIPTION
    read() attempts to read up to count bytes from file descriptor fd into
    the buffer starting at buf. (...)

RETURN VALUE
    On success, the number of bytes read is returned (....)
    On error, -1 is returned, and errno is set appropriately. (...)

ERRORS
    EBADF  fd  is  not  a valid file descriptor or is not open for
    reading.

   EFAULT buf is outside your accessible address space.
```

- Vorgehensweise:
 - Dokumentierter, umkehrbar eindeutiger Zusammenhang von Eigenschaften und Testfällen

```
READ(2)      UNIX Programmer's Manual      READ(2)

NAME
read - read from a file descriptor

SYNOPSIS
#include <unistd.h>

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

DESCRIPTION
read() attempts to read up to count bytes
from file descriptor fd into the buffer
starting at buf. (...)

RETURN VALUE
On success, the number of bytes read is
returned (...).
On error, -1 is returned, and errno is set
appropriately. (...)

ERRORS
EBADF fd is not a valid file descriptor or
is not open for reading.

EFAULT buf is outside your accessible address
space.

test_read_EBADF()
{
    int badfd = 1234567;
    char buf[SIZE];
    if(read(badfd, buf, SIZE) != -1) {
        printf("Testcase failed (1)\n");
        return ERROR;
    }
    else if(errno != EBADF) {
        printf("Testcase failed (2)\n");
        return ERROR;
    }
    return OK;
}

test_read_EFAULT()
{
    int fd = ....;
    if(read(fd, NULL, 1) != -1) {
        printf("Testcase failed (3)\n");
        return ERROR;
    }
    else if(errno != EFAULT) {
        printf("Testcase failed (4)\n");
        return ERROR;
    }
    return OK;
}
```

- **Stärken**
 - Führt bei vollständiger Spezifikation zum vollständigen Test (theoretisch)
 - Keine Abhängigkeit von der Implementierung
 - Daher anwendbar auf alle Implementierungen der gleichen Schnittstelle
- **Schwächen**
 - Qualität des Ergebnisses steht und fällt mit der Vollständigkeit der Funktionsbeschreibung
 - Liefert keine Hinweise auf Lücken in der Spezifikation
 - „Toter“ Code wird weder erkannt noch getestet
 - Vollständiger Test i.A. nicht praktikabel

- **vollständiger Test i.A. nicht praktikabel**

```
int function(int x)
{
    switch(x) {
        case 1:
        ....
        case 135:
        ....
    }
}
```

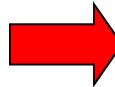


2^{32} Testfälle!

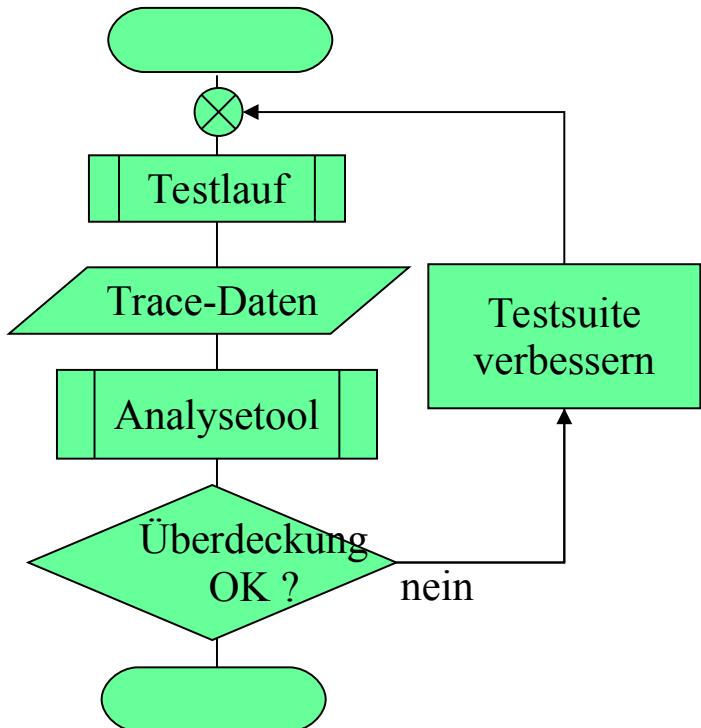
- I.A. ist der bei weitem größte Teil der Testfälle redundant
(=> „Äquivalenzklassen“)
- Reduktion auf relevante Testfälle erfordert Einbeziehung des Quellcodes.

- auch: „White-Box“ / „Glass-Box“-Test)
 - Ausgangsbasis: Quellcode des Prüflings
 - Ziel: Alle Programmteile des Prüflings sollen berührt (*überdeckt*) werden

<pre> unsigned char toupper(unsigned char c) /* Note: this also converts international character codes (0xc0 .. 0xfe) */ if((c >= 'a' && c <= 'z') (c >= 0xc0 && c <= 0xfe)) { return(c - 'a' + 'A'); } else { return c; } </pre>	<pre> struct testcase { const unsigned char in, expect; }; static struct testcase tc[4] = { {'a', 'A', }, {0xe0, 0xc0, }, {'A', 'A', }, {0xff, 0xff, } }; test_toupper() { int i; for(i = 0; i <= 4; i++) { if(tc[i].expect != toupper(tc[i].in)) { printf("Test %d failed\n", i); } } } </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

 **4 Testfälle
(statt 256)**

- **Vorgehensweise**



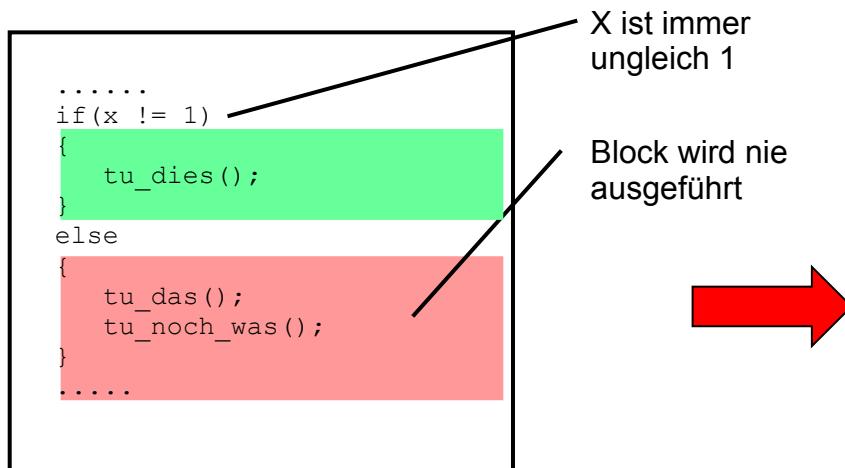
- Instrumentierung mit Hilfe eines Überdeckungswerkzeuges (Coverage Tool)
- Instrumentiertes Programm verhält sich (idealerweise) wie original-Programm, erzeugt aber zusätzlich Trace-Daten
- Analysewerkzeug ermittelt anhand der Trace-Daten Überdeckungsmaße und „weiße Stellen“
- Erweiterung der Testsuite zur Verbesserung der Überdeckung
- Iterativ optimieren, bis geforderte Überdeckungsmaße erreicht werden

- **Stärken**
 - Erkennt redundante Testfälle
 - Erkennt „toten Code“
 - Quantitative Erfassung der Testtiefe (?)
- **Schwächen**
 - Test ist spezifisch für die Implementierung
 - Keine Hinweise auf Unterlassungsfehler:

```
int divide(int x, int y)
{
    return (x / y);
}
```

Fehler:
Prüfung auf $y \neq 0$ fehlt!

- **Eintrittsüberdeckung (function/entry coverage)**
 - Wurden alle Funktionen aufgerufen ?
- **Anweisungsüberdeckung (statement coverage)**
 - Wurde sämtlicher Code des Prüflings mindestens einmal ausgeführt ?
- **Blocküberdeckung (block coverage)**
 - Wurden sämtliche basic blocks ausgeführt ?



Testsuite um einen entsprechenden Test ($x=1$) erweitern!

- Zweigüberdeckung(decision/branch coverage)
 - Sind alle möglichen Alternativen aller Bedingungen aufgetreten ?

```
if(bedingung)
{
    tu_dies();
}

....
```



```
switch(x)
{
    case 'A':
    case 'B':
        tu_was();
        break;
}
```

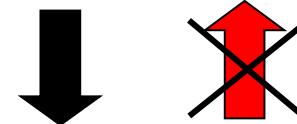


```
....
```

Bedingung war niemals „FALSE“
Block wird abgedeckt, dennoch kein vollständiger Test

X war niemals 'A'
Block wird abgedeckt, dennoch kein vollständiger Test

100% Zweigüberdeckung



100% Anweisungsüberdeckung

- **Mehrfach-Bedingungsüberdeckung (multiple condition coverage)**
 - Sind alle möglichen Kombinationen aller Bedingungen aufgetreten ?

```
if(a && (b || c))  
{  
    tu_was();  
}
```

	a	b	c	Ergebnis
Komb. 1	FALSE	FALSE	FALSE	FALSE
Komb. 2	FALSE	FALSE	TRUE	FALSE
Komb. 3	FALSE	TRUE	FALSE	FALSE
Komb. 4	FALSE	TRUE	TRUE	FALSE
Komb. 5	TRUE	FALSE	FALSE	FALSE
Komb. 6	TRUE	FALSE	TRUE	TRUE
Komb. 7	TRUE	TRUE	FALSE	TRUE
Komb. 8	TRUE	TRUE	TRUE	TRUE

N Bedingungen -> 2^N Kombinationen

Bestimmte Kombinationen aufgrund von Shortcuts u.U.
nicht erreichbar

- **Modifizierte Bedingungsüberdeckung (modified condition/decision coverage, MC/DC)**
 - Reduktion auf diejenigen Kombinationen, bei denen die Änderung einer Bedingung das Ergebnis beeinflußt

```
if(a && (b || c))  
{  
    tu_was();  
}
```

	a	b	c	Ergebnis
Komb. 1	FALSE	FALSE	FALSE	FALSE
Komb. 2	FALSE	FALSE	TRUE	FALSE
Komb. 3	FALSE	TRUE	FALSE	FALSE
Komb. 4	FALSE	TRUE	TRUE	FALSE
Komb. 5	TRUE	FALSE	FALSE	FALSE
Komb. 6	TRUE	FALSE	TRUE	TRUE
Komb. 7	TRUE	TRUE	FALSE	TRUE
Komb. 8	TRUE	TRUE	TRUE	TRUE

N Bedingungen -> N+1 <= Anzahl Kombinationen <= 2*N

Relevant bei Anwendungen in der Avionik (DO-178B)

- **Pfadüberdeckung („path coverage“)**
 - Sind alle möglichen Pfade ausgeführt worden ?

```
if(bedingung1)
    tu_dies();
if(bedingung2)
    tu_das();
if(bedingung3)
    tu_nochwas();
```

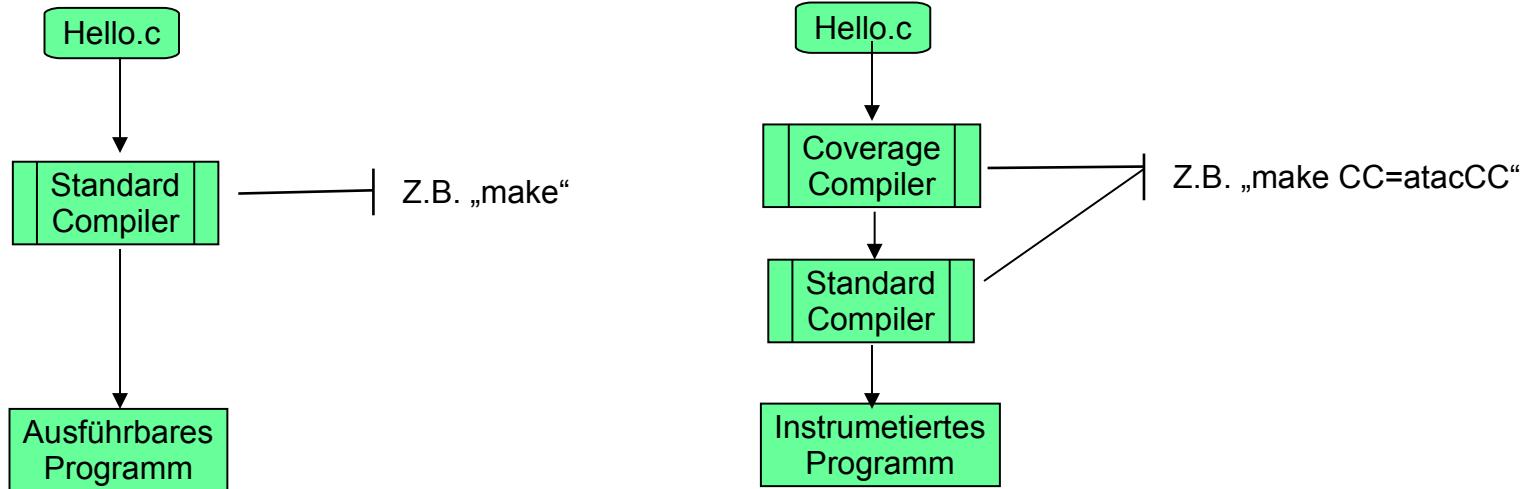
	Bedingung1	Bedingung2	Bedingung3
Pfad1	FALSE	FALSE	FALSE
Pfad2	FALSE	FALSE	TRUE
Pfad3	FALSE	TRUE	FALSE
Pfad4	FALSE	TRUE	TRUE
Pfad5	TRUE	FALSE	FALSE
Pfad6	TRUE	FALSE	TRUE
Pfad7	TRUE	TRUE	FALSE
Pfad8	TRUE	TRUE	TRUE

Aufwand steigt exponentiell mit der Anzahl der Bedingungen
-> i.A. zu Komplex !

- Defs/Uses Kriterium (DU-Path)
 - Reduzierte Variante der Pfadüberdeckung
 - Es wird nur die Untermenge der Pfade zwischen der Wertzuweisung einer Variablen und der Referenz auf den Wert der Variablen betrachtet
 - Computational use coverage („C-use“): Referenz auf die Variable ist nicht Teil einer Bedingung
 - Predicate use coverage(„P-use“): Referenz ist Teil einer Bedingung -> alle Verzweigungen aufgrund des Wertes der Bedingung werden zu der Liste betrachteten Pfade gezählt

- **Vorgehensweise:**

- Das Coverage Tool liest Quellcode (z.B. „C“) und erzeugt „instrumentierten“ Quellcode,
- Der instrumentierte Code wird mit Hilfe des Standard-Compilers in Binärkode übersetzt
- Oft transparenter Vorgang (Substitution des Standard-Compilers).



- Beispiel: „ATAC“ (rein softwarebasiert)
 - „Automatic Test Analysis for C Programs“ (Horgan/London `91)
 - Es gibt auch hardwaregestützte Werkzeuge
 - -> geringere Verfälschung des Laufzeitverhaltens

Originalcode:

```
main()
{
    printf("Hello World\n");
}
```

Funktion aus Laufzeitbibliothek

Instrumentierter Code:

```
main()
{
    struct Ztables *ZTptr = &ZTmain;
    void *ZBfr = alloca(
        sizeof(struct Zcontext) +
        ZTptr->max_puse * sizeof(struct Zpdef)
        + ZTptr->nvar * sizeof(short));
    int Z=aTaC(0, (int)&ZTmain, ZTptr, ZBfr);
    {
        aTaC(Z, 1, ZTptr, ZBfr);
        printf("Hello World\n");
    }
}
```

- **Vorgehensweise**
 - Das instrumentierte Programm verhält sich (idealerweise) genauso wie das nicht-instrumentierte Programm
 - Es erzeugt zusätzlich zur Laufzeit „trace“-Daten
 - Auswertung der trace-Daten mit Analysetools liefert z.B.
 - » Block Coverage (Prozent/absolut)
 - » Decision Coverage (Prozent/absolut)
 - » C-uses/P-uses (Prozent/absolut)
 - » Quellcode-Listing mit Markierung der nicht-abgedeckten Fälle

- **Besonderheiten bei embedded Systems**
 - höherer Ressourcenbedarf durch Instrumentierung
 - Probleme durch Verfälschung des Zeitverhaltens
 - evtl. kein Massenspeicher für Trace-Daten
- **Mögliche Abhilfen**
 - Ausweichen auf Testrechner mit ausreichend Ressourcen
 - Hardwaregestützte Werkzeuge

- **Instrumentieren von Betriebssystemcode**
 - Keine Standard-C Bibliothek
 - BS-Code verwendet häufig non-Standard Konstrukte (assembler-inline, pragma, ..), die vom Instrumentierer (=eigenständiger Compiler) u.U. nicht unterstützt werden
 - Fehlender Dateisystemzugriff für Trace-Daten
- **Mögliche Abhilfen**
 - Implementierung der fehlenden Funktionen als Pseudotreiber (erfordert Quellcode)
 - Hardwaregestützte Werkzeuge
 - Evtl. Instrumentierung lokal unterbinden

Vergleich: Funktionaler vs. Struktureller Test 7.6

- **Rein funktionales Testen kann sinnvoll sein**
 - Unabhängigkeit der Testsuite von der Implementierung des Prüflings
 - z.B. Test auf Standard API Konformität
- **Rein strukturelles Testen ist niemals sinnvoll**
 - 100%ige Überdeckung trotz schwerster (Unterlassungs-) Fehler möglich
 - » („A fool with a tool is still a fool“)
- **Struktureller Test als Ergänzung zum funktionalen Test**
 - 1) Entwickeln einer funktionalen Testsuite
 - 2) Optimierung der Testsuite anhand strukturellem Test

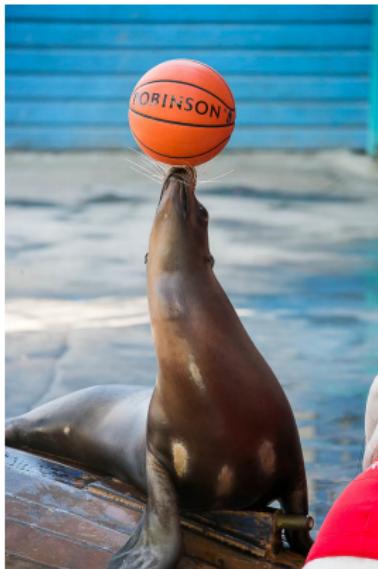
Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>
EMail: robert.kaiser@hs-rm.de)

Sommersemester 2021

8. Regelungstechnische Grundlagen



<https://www.stadtreporter.de/hannover/news/wirtschaft/robben-am-ball-ein-tierisch-sportlicher-geburtstag>

Inhalt

8. Regelungstechnische Grundlagen

8.1 Ziele

8.2 Grundbegriffe

8.3 Grundlagen der Systemtheorie

8.4 Entwurf zeitkontinuierlicher Regler

8.5 Unstetige Regelung

8.6 Fuzzy-Regler

Regelungstechnik-Grundlagen

Grundlegende Einführung in die Regelungstechnik

- Grundbegriffe
 - ▶ Signal
 - ▶ System
 - ▶ Steuerung und Regelung
- Grundlagen der Systemtheorie
 - ▶ Differentialgleichung und Übergangsfunktion
 - ▶ Laplace-Transformation und Übertragungsfunktion
- Entwurf zeitkontinuierlicher Regler
 - ▶ Regelkreis
 - ▶ Klassische Regelalgorithmen
 - ▶ Stabilität
- Unstetige Regelung
- Fuzzy-Regler

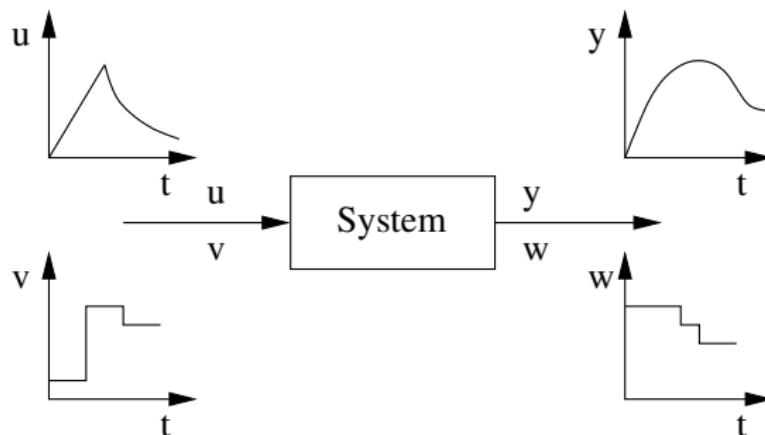
Grundbegriffe: Signal

Definition: Signal

Ein *Signal* ist eine sich zeitlich ändernde Größe, durch die eine Information ausgedrückt wird.

- Darstellung als Funktion der Zeit: $s(t)$
- Formen:
 - ▶ Zeit- und wertkontinuierlich: $t, s(t) \in \mathbb{R}$
Beispiel: $s(t) = \sin(2\pi ft)$
 - ▶ Wertkontinuierlich, zeitdiskret: $s(n) \in \mathbb{R}, n \in \mathbb{N}$
 - ▶ Zeit- und werdiskret: $n, s(n) \in \mathbb{N}$
Beispiel: $s = (1, 3, 4, 4, \dots) \Rightarrow s(0) = 1, s(1) = 3, s(2) = 4, \dots$
→ „Folge von Abtastwerten“
→ Verwendung in digitalen Regelungen

Grundbegriffe: System



- Zuordnung von Eingangs- zu Ausgangssignal
z.B. $u(t) \rightarrow y(t)$
- **Rückwirkungsfrei** → Kein Einfluss von Ausgang auf Eingang

Statisches System

- Der Wert der Ausgangsgröße zur Zeit t wird nur durch den Wert der Eingangsgröße zur selben Zeit t bestimmt:

$$y(t) = S(u(t)) \quad \forall t \in \mathbb{R}$$

→ System „ohne Gedächtnis“

- Beispiele:
 - ▶ Ohmscher Widerstand ($U = R \cdot I$)
 - ▶ Tabelle ($y = \text{Tabelle}[n]$;)
- Math. Beschreibung durch Funktion: $y = S(u)$

Dynamisches System (2)

- Der Wert der Ausgangsgröße zur Zeit t wird nur durch den bisherigen Verlauf der Eingangsgröße bis zur Zeit t bestimmt:

$$y(t) = S(u([-\infty, t]))$$

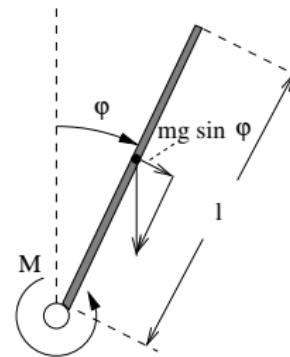
→ „Vorgeschichte“ geht ein
 (Aber *kausal*: Zukunft geht nicht ein)

- Mathematische Beschreibung durch Differentialgleichung

- Beispiel: Inverses Pendel
- Momentengleichgewicht

$$\Rightarrow M = \frac{I}{2} \cdot m \cdot g \cdot \sin\phi - J \cdot \ddot{\phi}$$

- hier: $M(t) \hat{=} u(t)$, $\phi(t) \hat{=} y(t)$



Dynamisches System (2)

Weitere Eigenschaften

- **Linear** → „Verzerrungsfrei“

wenn: $y_1(t) = S(u_1([-\infty, t]))$

und: $y_2(t) = S(u_2([-\infty, t]))$

$$\Rightarrow y_1(t) + y_2(t) = S(u_1([-\infty, t]) + u_2([-\infty, t]))$$

(N.B.: daraus folgt auch: $k \cdot y(t) = S(k \cdot u([-\infty, t])) \forall k \in \mathbb{R}$)

- **Zeitinvariant** → „unabhängig von absoluter Zeit“

wenn: $y(t) = S(u([-\infty, t]))$

dann: $y(t + t_0) = S(u([-\infty, t + t_0]))$

- **Kausal** (s.o.) → „Keine Kenntnis über die Zukunft“

„**LTI-System**“ (*linear time invariant*)

Der Weg zur Differentialgleichung

Vorgehen Allgemein	Am Beispiel inv. Pendel
1. System in Komponenten zerlegen	Gewicht, Trägheit, Drehmoment
2. Physikalische Gesetze für die Komponenten zusammenstellen	$F_g = m \cdot g \cdot \sin \phi$ $\Rightarrow M_g = F_g \cdot r = \frac{m \cdot g \cdot l}{2} \cdot \sin \phi$ $M_t = J \cdot \ddot{\omega} = J \cdot \ddot{\phi}$
3. Beziehungen zwischen den Komponenten aufstellen	Summe aller Momente ist Null: $M - M_g + M_t = 0 \Rightarrow M = M_g - M_t$
4. Gleichungen zu einer DGL zusammenfassen	$M = \frac{m \cdot g \cdot l}{2} \cdot \sin \phi - J \cdot \ddot{\phi}$
5. ggf. DGL linearisieren	für kleine ϕ gilt: $\phi \approx \sin \phi$ $\Rightarrow M \approx \frac{m \cdot g \cdot l}{2} \cdot \phi - J \cdot \ddot{\phi}$

Differentialgleichung: allgemeine Form

Allgemeine Form der Differentialgleichung:

$$\dots + a_3 \cdot \ddot{y} + a_2 \cdot \ddot{y} + a_1 \cdot \dot{y} + a_0 \cdot y = \dots + b_3 \cdot \ddot{u} + b_2 \cdot \ddot{u} + b_1 \cdot \dot{u} + b_0 \cdot u$$

- Grad der höchsten Ableitung: *Ordnung* des Systems
- Division durch a_0 und $T_i^n := \frac{a_i}{a_0}$, und $K_i^n := \frac{b_i}{a_0}$ ergibt:

$$\Rightarrow \dots + T_3^3 \cdot \ddot{y} + T_2^2 \cdot \ddot{y} + T_1 \cdot \dot{y} + y = \dots + K_3^3 \cdot \ddot{u} + K_2^2 \cdot \ddot{u} + K_1 \cdot \dot{u} + K_0 \cdot u$$

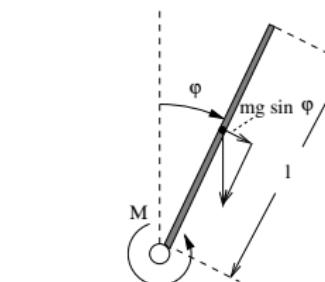
→ Die T_i und K_i haben die Dimension „Zeit“ → **Zeitkonstanten**

Steuerung (1)

Steuerung: offene Wirkungskette, keine Rückkopplung

- Ziel: Eingangssignal $u(t)$ so, dass das gewünschte Ausgangssignal $y(t)$ erzeugt wird
- Beispiel: invertiertes Pendel aufrichten
 - d.h.: $\phi \stackrel{!}{=} 0, \dot{\phi} \stackrel{!}{=} 0$
- Dabei Vorgaben:
 - Möglichst schnell
 - $M(t) \leq M_{max}$
- Gesucht: $M(t)$

Prinzipielles Vorgehen:



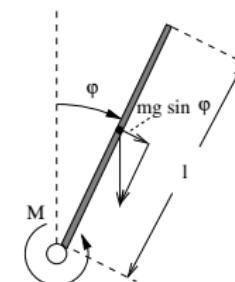
$$M = \frac{l}{2} \cdot m \cdot g \cdot \sin\phi - J \cdot \ddot{\phi}$$

- $M(t)$ vorgeben
- Lösung der Differentialgleichung (s.o.) suchen
- ... nichtlineare DGL 2. Ordnung → heute nicht!

Steuerung (1)

Steuerung: offene Wirkungskette, keine Rückkopplung

- Ziel: Eingangssignal $u(t)$ so, dass das gewünschte Ausgangssignal $y(t)$ erzeugt wird
- Beispiel: invertiertes Pendel aufrichten
 - d.h.: $\phi \stackrel{!}{=} 0, \dot{\phi} \stackrel{!}{=} 0$
- Dabei Vorgaben:
 - Möglichst schnell
 - $M(t) \leq M_{max}$
- Gesucht: $M(t)$



Prinzipielles Vorgehen:

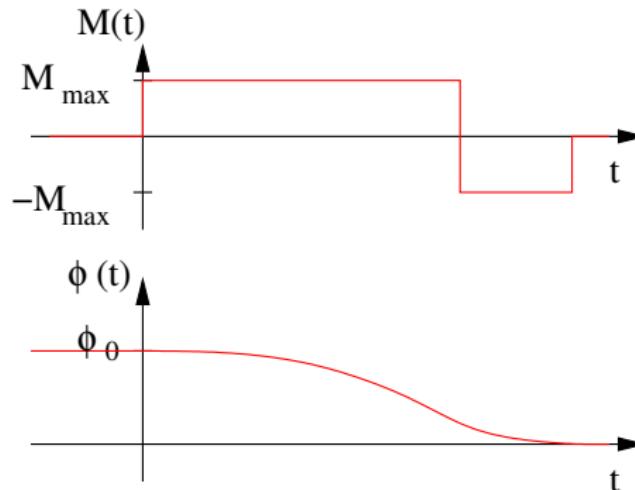
$$M = \frac{l}{2} \cdot m \cdot g \cdot \sin\phi - J \cdot \ddot{\phi}$$

- $M(t)$ vorgeben
- Lösung der Differentialgleichung (s.o.) suchen
- ... nichtlineare DGL 2. Ordnung → heute nicht!

Steuerung (2)

Intuitive Lösung:

- ① Beschleunigen mit $M = M_{max}$
- ② Rechtzeitig abbremsen mit $M = -M_{max}$



• Wann ist „rechtzeitig“?

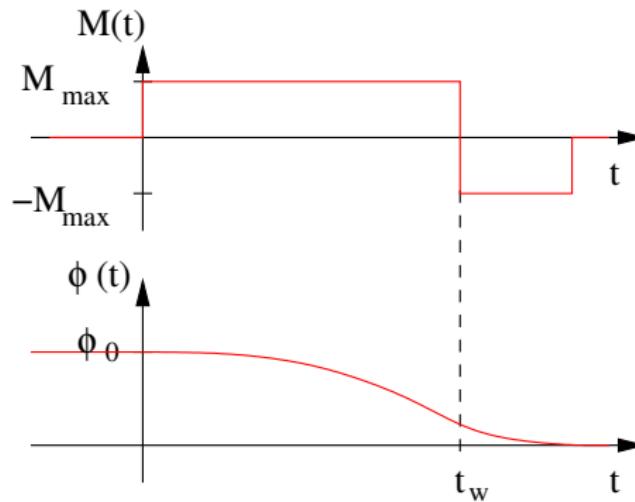
Energieerhaltungssatz:

$$\phi_w = \frac{\phi_0}{2} - \frac{mgl}{4M_{max}} \cdot (1 - \cos\phi_0)$$

Steuerung (2)

Intuitive Lösung:

- ① Beschleunigen mit $M = M_{max}$
- ② Rechtzeitig abbremsen mit $M = -M_{max}$



- Wann ist „rechtzeitig“?

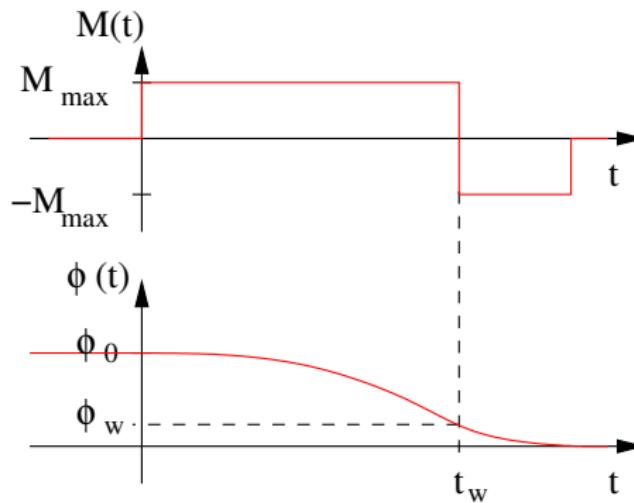
Energieerhaltungssatz:

$$\phi_w = \frac{\phi_0}{2} - \frac{mgl}{4M_{max}} \cdot (1 - \cos\phi_0)$$

Steuerung (2)

Intuitive Lösung:

- ① Beschleunigen mit $M = M_{max}$
- ② Rechtzeitig abbremsen mit $M = -M_{max}$



- Wann ist „rechtzeitig“?

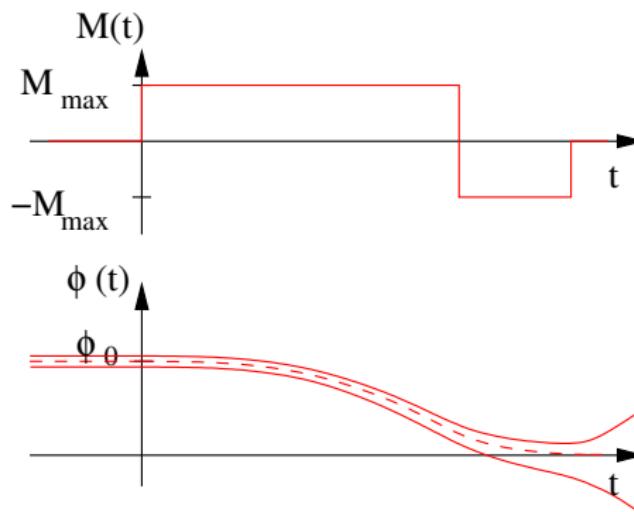
Energieerhaltungssatz:

$$\phi_w = \frac{\phi_0}{2} - \frac{mgl}{4M_{max}} \cdot (1 - \cos\phi_0)$$

Steuerung (3)

Bei Steuerung: Keine Rückkopplung

- Reagiert äußerst empfindlich auf kleinste Störungen bzw. Fehler:



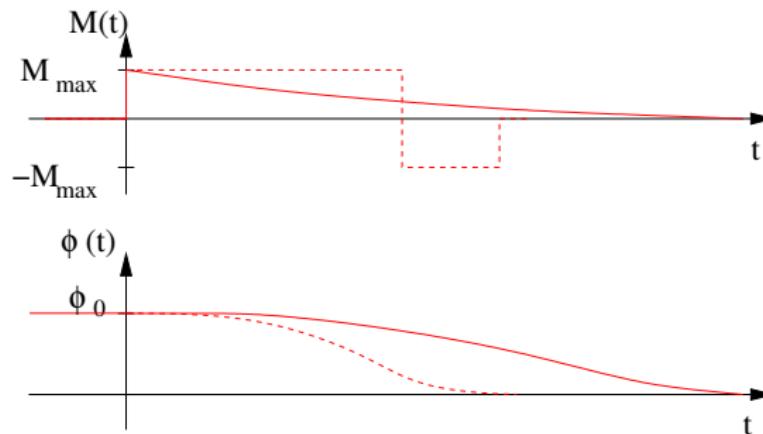
- Beispiel
 - Minimaler Fehler bei ϕ_0
- Aufrichten misslingt
(Instabiles System)

Regelung

Stellgröße $M(t)$ aus $\phi(t)$ berechnen:

$$M(t) = a \cdot \phi(t) + b \cdot \dot{\phi}(t), \text{ wobei: } a \cdot \phi(0) + b \cdot \dot{\phi}(0) = M_{max}$$

- Je nach Wahl von a und b unterschiedliches Verhalten
- Gleichgewichtszustand wird erreicht, da
 $M(t) = 0$, nur wenn $\phi(t)$ und $\dot{\phi}(t) = 0$

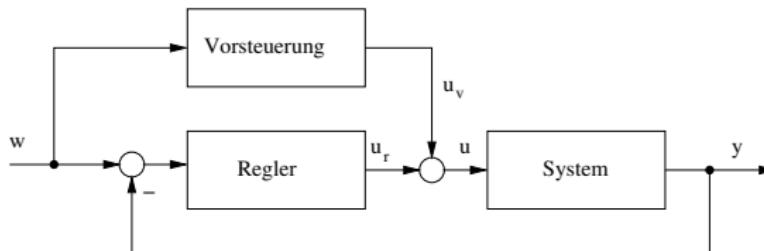


- Nachteil: Vorgang dauert länger

Kombination: Steuern + Regeln

„Vorsteuerung“:

- $u(t) = u_v(t) + u_r(t)$
- Vorsteuerung kann schnell das gewünschte Signal $y(t)$ einstellen
- (kleine) Fehler dabei werden durch die Regelung kompensiert

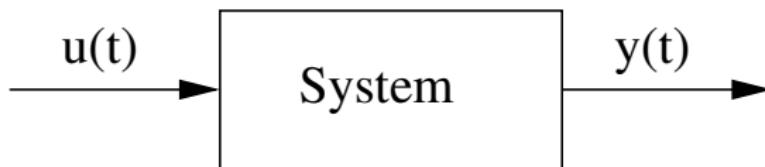


Kombiniert Vorteile von Steuerung und Regelung:

- schnell
- eigenstabil

Grundlagen der Systemtheorie

Allgemein: System



- Eingangssignal $u(t)$
- Ausgangssignal $y(t)$
- Linear, zeitinvariant, kausal, rückwirkungsfrei (s.o)
- Kann durch eine Differentialgleichung beschrieben werden (s.o)

Lösen der Differentialgleichung (1)

Prinzipielles Vorgehen

- gegeben Differentialgleichung (s.o):

$$\dots + T_3^3 \cdot \ddot{y} + T_2^2 \cdot \ddot{y} + T_1 \cdot \dot{y} + y = \dots + K_3^3 \cdot \ddot{u} + K_2^2 \cdot \ddot{u} + K_1 \cdot \dot{u} + K_0 \cdot u$$

- Rechte Seite = 0 setzen → *homogene DGL*

$$\dots + T_3^3 \cdot \ddot{y} + T_2^2 \cdot \ddot{y} + T_1 \cdot \dot{y} + y = 0$$

- Ansatz: $y = A \cdot e^{s \cdot t}$ ($\Rightarrow \dot{y} = A \cdot s \cdot e^{s \cdot t}, \ddot{y} = A \cdot s^2 \cdot e^{s \cdot t}, \dots$)

→ *Charakteristische Gleichung:*

$$\dots + T_3^3 \cdot s^3 + T_2^2 \cdot s^2 + T_1 \cdot s + 1 = 0$$

- Hieraus kann s ermittelt werden → Lösung der homogenen DGL:

$$\Rightarrow y_{hom}(t) = A \cdot e^{s \cdot t}$$

Lösen der Differentialgleichung (2)

- Spezielle Lösung der inhomogenen DGL:

$$\dots + T_3^3 \cdot \ddot{y} + T_2^2 \cdot \ddot{y} + T_1 \cdot \dot{y} + y = \dots + K_3^3 \cdot \ddot{u} + K_2^2 \cdot \ddot{u} + K_1 \cdot \dot{u} + K_0 \cdot u$$

- Eingangsfunktion $u(t)$ muss bekannt sein
- Anfangsbedingungen (Werte für bestimmte Zeitpunkte) müssen bekannt sein
- Wähle Ansatz für $y_{inh}(t)$ entsprechend der rechten Seite der DGL.
- Gesamtlösung ist dann:

$$y(t) = y_{hom}(t) + y_{inh}(t)$$

- Konstanten können aus Anfangsbedingungen bestimmt werden
- *Keine Panik! Beispiel folgt...*

Lösen der Differentialgleichung (2)

- Spezielle Lösung der inhomogenen DGL:

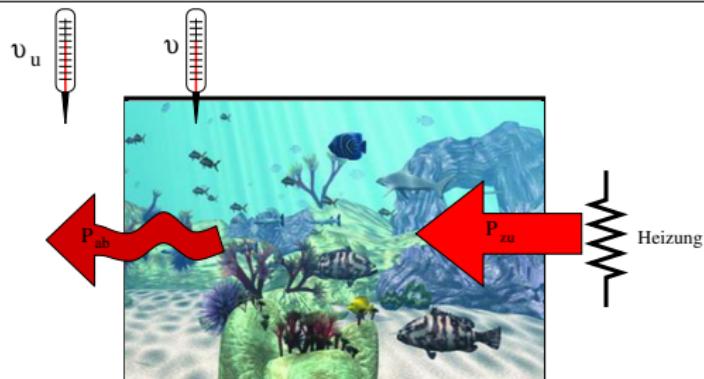
$$\dots + T_3^3 \cdot \ddot{y} + T_2^2 \cdot \ddot{y} + T_1 \cdot \dot{y} + y = \dots + K_3^3 \cdot \ddot{u} + K_2^2 \cdot \ddot{u} + K_1 \cdot \dot{u} + K_0 \cdot u$$

- Eingangsfunktion $u(t)$ muss bekannt sein
- Anfangsbedingungen (Werte für bestimmte Zeitpunkte) müssen bekannt sein
- Wähle Ansatz für $y_{inh}(t)$ entsprechend der rechten Seite der DGL.
- Gesamtlösung ist dann:

$$y(t) = y_{hom}(t) + y_{inh}(t)$$

- Konstanten können aus Anfangsbedingungen bestimmt werden
- *Keine Panik! Beispiel folgt....*

Beispiel (1)



$$\begin{aligned}
 P_{Heiz} &= P_{zu} - P_{ab} \\
 \dot{\theta} &= \frac{P_{Heiz}}{C} = \frac{P_{Heiz}}{c_{H_2O} \cdot m} \\
 P_{ab} &= R_{th} \cdot (\theta - \theta_u) \\
 &= A \cdot K_{Glas} \cdot (\theta - \theta_u)
 \end{aligned}$$

Gegeben ein Aquarium mit:

- Volumen: $250\text{ l} \rightarrow \text{Masse } m = 250\text{ kg}$
- Oberfläche: $A = 1.5\text{ m}^2$
- Temperatur für $t = 0$: $\theta(0) = \text{Umgebungstemperatur } \theta_U = 20^\circ\text{C}$
- Zugeführte Heizleistung für $t \geq 0$: $P_{zu} = 100\text{ W}$

Gesucht: Temperaturverlauf $\theta(t)$ für $t \geq 0$

Beispiel (2)

Materialparameter (aus Wikipedia):

- Spezifische Wärmekapazität von Wasser: $c_{H_2O} = 4,187 \cdot 10^3 \frac{J}{kg \cdot K}$
- K-Wert (Wärmedurchgangskoeffizient) von Glas: $K_{Glas} = 5,9 \frac{W}{m^2 \cdot K}$

Aufstellen der DGL:

$$\begin{aligned}\dot{\theta} &= \frac{P_{zu} - P_{ab}}{c_{H_2O} \cdot m} \\ &= \frac{1}{c_{H_2O} \cdot m} \cdot (P_{zu} - A \cdot K_{Glas} \cdot (\theta - \theta_u)) \\ \Rightarrow \frac{c_{H_2O} \cdot m}{A \cdot K_{Glas}} \cdot \dot{\theta} + \theta &= \frac{P_{zu}}{A \cdot K_{Glas}} + \theta_u \\ \Rightarrow \text{DGL 1. Ordnung, } T_1 &= \frac{c_{H_2O} \cdot m}{A \cdot K_{Glas}} = 118022,6 \text{ s}\end{aligned}$$

Beispiel (3)

- Rechte Seite = 0 setzen → homogene DGL

$$T_1 \cdot \dot{\theta} + \theta = 0$$

- Ansatz: $\theta = \Theta_0 \cdot e^{s \cdot t}$ ($\Rightarrow \dot{\theta} = \Theta_0 \cdot s \cdot e^{s \cdot t}$)
→ Charakteristische Gleichung:

$$\begin{aligned} T_1 \cdot \Theta_0 \cdot s \cdot e^{s \cdot t} + \Theta_0 \cdot e^{s \cdot t} &= 0 \mid : \Theta_0 \cdot e^{s \cdot t} \\ \Rightarrow T_1 \cdot s + 1 &= 0 \\ \Rightarrow s &= -\frac{1}{T_1} \end{aligned}$$

→ Lösung der homogenen DGL:

$$\theta_{hom} = \Theta_0 \cdot e^{-\frac{t}{T_1}}$$

Beispiel (4)

- Spezielle Lösung der inhomogenen DGL:

$$T_1 \cdot \dot{\theta} + \theta = \theta_u + \begin{cases} \frac{P_{zu}}{A \cdot K_{Glas}} & \text{für } t \geq 0 \\ 0 & \text{für } t < 0 \end{cases}$$

$$\theta_{inh} = \theta_u + \begin{cases} \frac{P_{zu}}{A \cdot K_{Glas}} & \text{für } t \geq 0 \\ 0 & \text{für } t < 0 \end{cases}$$

- Gesamtlösung:

$$\begin{aligned} \theta &= \theta_{hom} + \theta_{inh} \\ &= \Theta_0 \cdot e^{-\frac{t}{T_1}} + \theta_u + \begin{cases} \frac{P_{zu}}{A \cdot K_{Glas}} & \text{für } t \geq 0 \\ 0 & \text{für } t < 0 \end{cases} \end{aligned}$$

Beispiel (5)

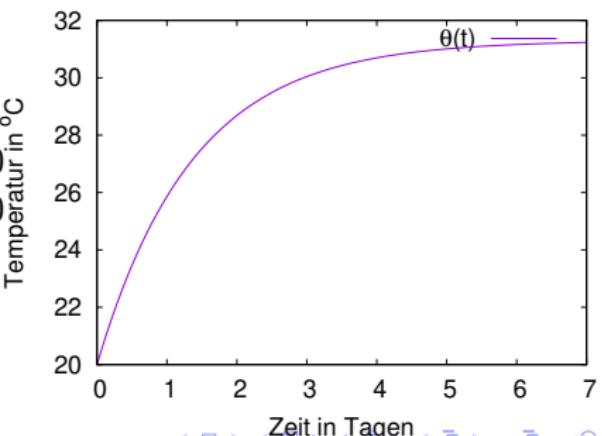
- Anfangsbedingungen: für $t \leq 0$ gilt: $\theta(t) = \theta_u$

$$\Rightarrow 0 = \Theta_0 + \begin{cases} \frac{P_{zu}}{A \cdot K_{Glas}} & \text{für } t \geq 0 \\ 0 & \text{für } t < 0 \end{cases}$$

$$\Rightarrow \Theta_0 = \begin{cases} -\frac{P_{zu}}{A \cdot K_{Glas}} & \text{für } t \geq 0 \\ 0 & \text{für } t < 0 \end{cases}$$

- Ergebnis („Sprungantwort“):

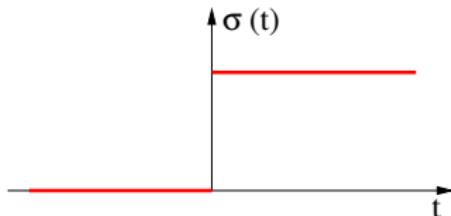
$$\theta = \theta_u + \left(1 - e^{-\frac{t}{T_1}}\right) \cdot \begin{cases} \frac{P_{zu}}{A \cdot K_{Glas}} & \text{für } t \geq 0 \\ 0 & \text{für } t < 0 \end{cases}$$



Spezielle Eingangsfunktionen

- Anregung des Systems wird durch eine Eingangsfunktion (auch: „Störfunktion“) beschrieben
- Im vorangegangenen Beispiel: *Sprungfunktion* $\sigma(t)$

$$\sigma(t) = \begin{cases} 0 & \text{für } t < 0 \\ 1 & \text{für } t > 0 \end{cases}$$



(N.B.: für $t = 0$ ist $\sigma(t)$ streng genommen undefiniert)

- Damit: Störfunktion ...

$$\theta_{inh} = \theta_u + \frac{P_{zu}}{A \cdot K_{Glas}} \cdot \sigma(t)$$

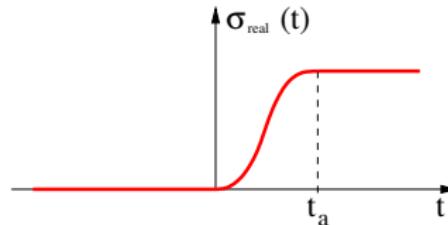
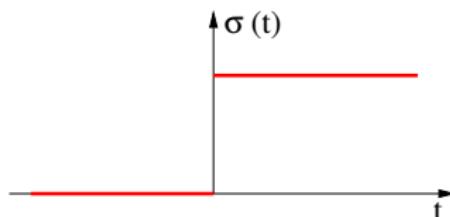
- ... und Sprungantwort des Beispielsystems:

$$\theta = \theta_u + \frac{P_{zu}}{A \cdot K_{Glas}} \cdot \left(1 - e^{-\frac{t}{\tau_1}}\right) \cdot \sigma(t)$$

Sprungfunktion

Technische Realisierbarkeit

- Unendlich schnelle Werteänderung nicht technisch realisierbar
(Reale Signale sind **immer** stetig)



- Realistische Anstiegszeiten t_a
 - ▶ Elektronik: < 1 ns
 - ▶ Mechanik, Thermodynamik, etc. z.T. wesentlich größer

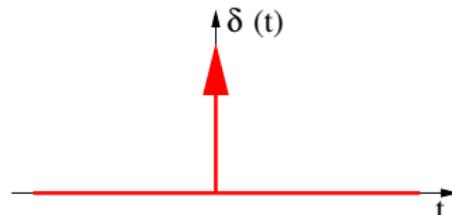
Bedeutung

- Sprungantwort eines Systems kann (innerhalb gegebener Grenzen) experimentell ermittelt werden
- Charakterisiert das dynamische Verhalten eines Systems

Impulsfunktion

Impulsfunktion $\delta(t)$ (auch: „Dirac-Impuls“ oder „Dirac-Stoß“)

- $\delta(t) = 0 \forall t \neq 0$
- für $t = 0$: unendlich hoher, unendlich schmaler Impuls der Fläche 1, d.h.: $\int_{-\infty}^{+\infty} \delta(t) dt = 1$
- 1. Ableitung der Sprungfunktion: $\delta = \dot{\sigma}$



Bedeutung

- Eher theoretisches Konstrukt, technisch nicht realisierbar
- Enthält alle Frequenzen gleichermaßen
- Die „Eins der Laplace-Transformation“ (s.u.)

Laplace-Transformation

Definition

$$\mathcal{L}(y(t))(s) = Y(s) = \int_0^{\infty} y(t) \cdot e^{-st} dt$$

Schreibweise: $Y(s)$ •—○ $y(t)$ (\rightarrow „ $Y(s)$ korrespondiert zu $y(t)$ “)

- Umkehrbar eindeutige Abbildung von Funktionen im Zeitbereich in den Bildbereich („Laplace-Raum“) mit s als neuer unabhängiger Variablen (s ist i.A. komplex: $s \in \mathbb{C}$, Dimension: Frequenz (Hz))
- Anwendbar auf Zeitfunktionen, die für $t < 0$ Null sind, d.h.
 $y(t) = y(t) \cdot \sigma(t) \quad \forall t \in \mathbb{R}$
- Berechnung des Integrals ist i.d.R. nicht erforderlich, da es Korrespondenztabellen¹ gibt

Eigenschaften der \mathcal{L} -Transformation

- Linearität, Skalierung und Verschiebung:

Überlagerungssatz: $a \cdot u(t) + b \cdot v(t)$ $\circ \rightarrow \bullet$ $a \cdot U(s) + b \cdot V(s)$

Ähnlichkeitssatz: $u(a \cdot t)$ $\circ \rightarrow \bullet$ $\frac{1}{a} \cdot U\left(\frac{s}{a}\right)$

Verschiebesatz: $u(t - a)$ $\circ \rightarrow \bullet$ $e^{-as} \cdot U(s)$

Integration: $\int_0^t u(\tau) d\tau$ $\circ \rightarrow \bullet$ $\frac{1}{s} \cdot U(s)$

- Differentiation:

$$\dot{u}(t) \quad \circ \rightarrow \bullet \quad s \cdot U(s) - u(0)$$

$$\ddot{u}(t) \quad \circ \rightarrow \bullet \quad s^2 \cdot U(s) - s \cdot u(0) - \dot{u}(0)$$

$$\dddot{u}(t) \quad \circ \rightarrow \bullet \quad s^3 \cdot U(s) - s^2 \cdot u(0) - s \cdot \dot{u}(0) - \ddot{u}(0)$$

... $\circ \rightarrow \bullet$...

\mathcal{L} -Transformation zum Lösen einer DGL (1)



- Durch die Regel zur Differentiation werden Differentialgleichungen im Zeitbereich zu einfachen Gleichungen im Bildbereich:

$$\begin{aligned} T_1 \cdot \dot{y} + y &= K_0 \cdot u & \circ - \bullet & \quad T_1 \cdot s \cdot Y(s) - y(0) + Y(s) = K_0 \cdot U(s) \\ && \Rightarrow & \quad Y(s) \cdot (T_1 s + 1) - y(0) = K_0 \cdot U(s) \end{aligned}$$

- bzw. für $y(0) = 0$:

$$\begin{aligned} Y(s) \cdot (T_1 s + 1) &= K_0 \cdot U(s) \\ \Rightarrow \frac{Y(s)}{U(s)} &= \frac{K_0}{T_1 s + 1} =: G(s) \end{aligned}$$

- $G(s)$ heißt *Übertragungsfunktion* des Systems
- $G(s)$ liefert eine vollständige Beschreibung des Systems
(Mit Ausnahme der Anfangszustände)

\mathcal{L} -Transformation zum Lösen einer DGL (2)

- Aquariumsbeispiel (s.o.)

$$T_1 \cdot \dot{\theta} + \theta = K_0 \cdot \sigma(t) + \theta_u$$

(wobei (s.o) $T_1 = \frac{c_{H_2O} \cdot m}{A \cdot K_{Glas}}$ und $K_0 = \frac{P_{zu}}{A \cdot K_{Glas}}$)

- Substitution: $y(t) := \theta(t) - \theta_u$ ($\Rightarrow \dot{y}(t) = \dot{\theta}(t)$)
- Damit DGL: $T_1 \cdot \dot{y} + y = K_0 \cdot \sigma(t)$

$$\mathcal{L}\text{-Transformation: } \Rightarrow \quad T_1(sY - y(0)) + Y = K_0 \cdot \frac{1}{s}$$

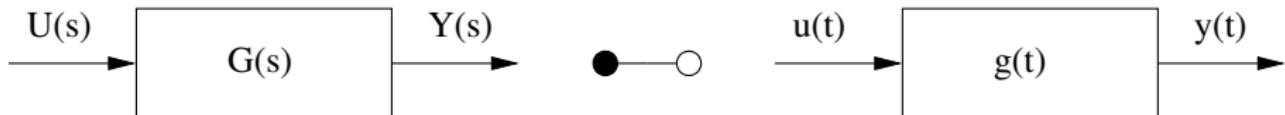
$$y(0) = 0 \Rightarrow \quad Y = \frac{K_0}{T_1} \cdot \frac{1}{s(s + \frac{1}{T_1})}$$

$$\mathcal{L}\text{-Rücktransformation: } \Rightarrow \quad y(t) = \frac{K_0}{T_1} \cdot T_1 \cdot \left(1 - e^{-\frac{1}{T_1}}\right) \cdot \sigma(t)$$

- Rück-Substitution liefert gleiches Ergebnis wie oben:

$$\theta(t) = \theta_u + K_0 \cdot \left(1 - e^{-\frac{1}{T_1}}\right) \cdot \sigma(t)$$

Übertragungsfunktion (1)



- Übertragungsverhalten eines Systems entspricht im Bildbereich einer Multiplikation mit der Übertragungsfunktion

$$(s.o.) \quad G(s) = \frac{Y(s)}{U(s)} \quad \Rightarrow \quad Y(s) = U(s) \cdot G(s)$$

- Die Impulsfunktion $\delta(t)$ ist das „1-Element“ der Laplace-Transformation, d.h. es gilt: $\delta(t) \circ \bullet 1$ (vgl. Korrespondenztabelle)
- Bei Anregung mit einer Impulsfunktion antwortet das System mit seiner Übertragungsfunktion:

Für $u(t) = \delta(t)$ gilt: $Y(s) = 1 \cdot G(s) \bullet \circ y(t) = g(t)$

Übertragungsfunktion (2)

- Für die Sprungantwort eines Systems gilt:

$$\sigma(t) \quad \text{---} \bullet \quad \frac{1}{s}$$

\Rightarrow für $u(t) = \sigma(t)$ gilt:

$$Y(s) = \frac{1}{s} \cdot G(s) \quad \bullet \text{---} \circ \quad y(t) = \int_0^t g(\tau) d\tau$$

- Die Sprungantwort ist das Integral über die Impulsantwort
- Rückschluss auf die Übertragungsfunktion auch ohne „echten“ Dirac-Impuls möglich

Übertragungsfunktion (3)

- Die Übertragungsfunktion eines LTI-Systems lässt sich als Quotient zweier Polynome darstellen:

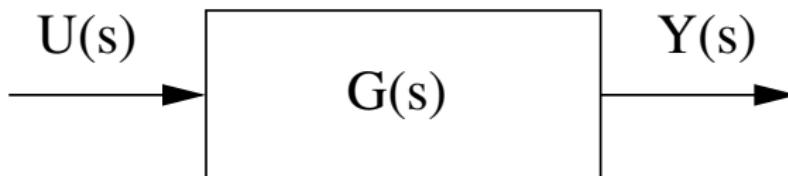
$$\begin{aligned} \dots + T_2^2 \ddot{y} + T_1 \dot{y} + y &= \dots + K_2^2 \ddot{u} + K_1 \dot{u} + K_0 u \\ \mathcal{L} \Rightarrow Y(s) \cdot (\dots + T_2^2 s^2 + T_1 s + 1) &= U(s) \cdot (\dots + K_2^2 s^2 + K_1 s + K_0) \\ \Rightarrow \frac{Y(s)}{U(s)} = G(s) &= \frac{\dots + K_2^2 s^2 + K_1 s + K_0}{\dots + T_2^2 s^2 + T_1 s + 1} \end{aligned}$$

- Dies lässt sich auch in *Nullstellenform* bringen::

$$G(s) = \frac{\dots (s - \mu_2) \cdot (s - \mu_1) \cdot (s - \mu_0)}{\dots (s - \lambda_2) \cdot (s - \lambda_1) \cdot (s - \lambda_0)}$$

- Die λ_i sind die *Pole*, die μ_i die *Nullstellen* der Übertragungsfunktion
- LTI-System kann auch durch Pole und Nullstellen der Übertragungsfunktion charakterisiert werden

Aggregation von Systemen

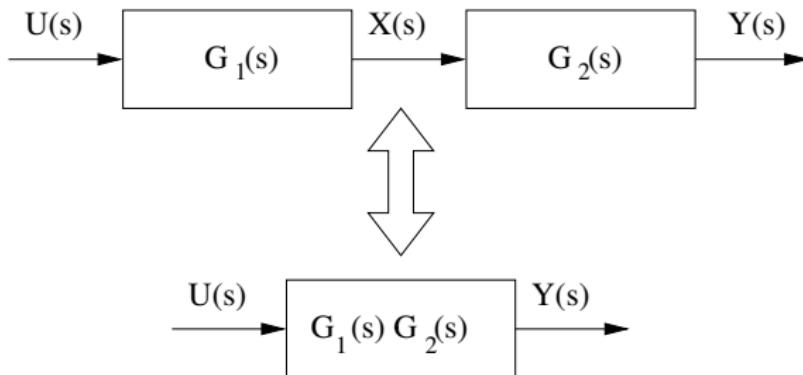


- (s.o.) Verhalten eines LTI-System kann durch seine Übertragungsfunktion $G(s)$ beschreiben werden

$$Y(s) = U(s) \cdot G(s)$$

- LTI-Systeme können als Blöcke kombiniert werden..

Serienschaltung



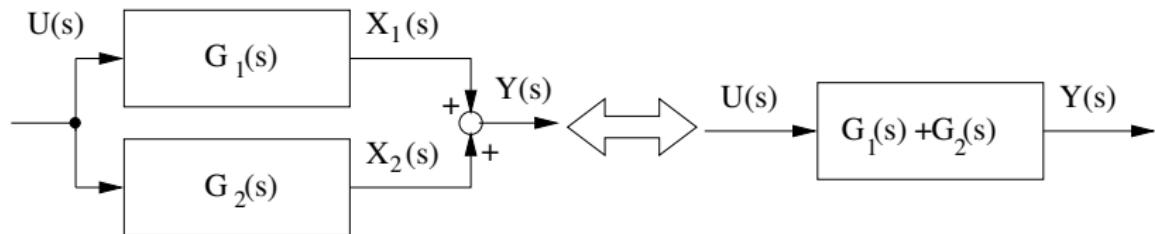
- Wegen Rückwirkungsfreiheit gilt:

$$X(s) = U(s) \cdot G_1(s)$$

$$Y(s) = X(s) \cdot G_2(s)$$

$$\Rightarrow Y(s) = U(s) \cdot G_1(s) \cdot G_2(s)$$

Parallelschaltung



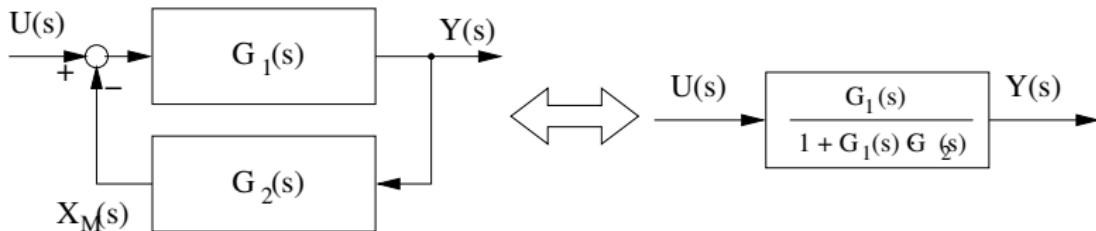
- Addierschaltung:

$$X_1(s) = U(s) \cdot G_1(s)$$

$$X_2(s) = U(s) \cdot G_2(s)$$

$$\Rightarrow Y(s) = U(s) \cdot (G_1(s) + G_2(s))$$

Rückkopplung (Gegenkopplung)



- Grundschaltung für Regelungen:

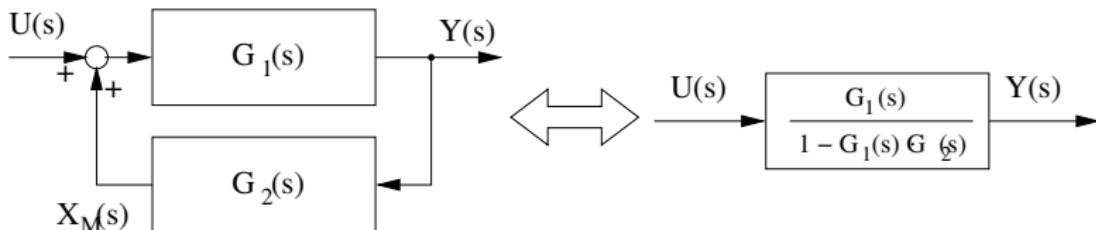
$$X(s) = E(s) \cdot G_1(s)$$

$$E(s) = U(s) - X_M(s)$$

$$X_M(s) = X(s) \cdot G_2(s)$$

$$\Rightarrow Y(s) = U(s) \cdot \frac{G_1(s)}{1 + G_1(s) \cdot G_2(s)}$$

Rückkopplung (Mitkopplung)



- Analog zu Gegenkopplung:

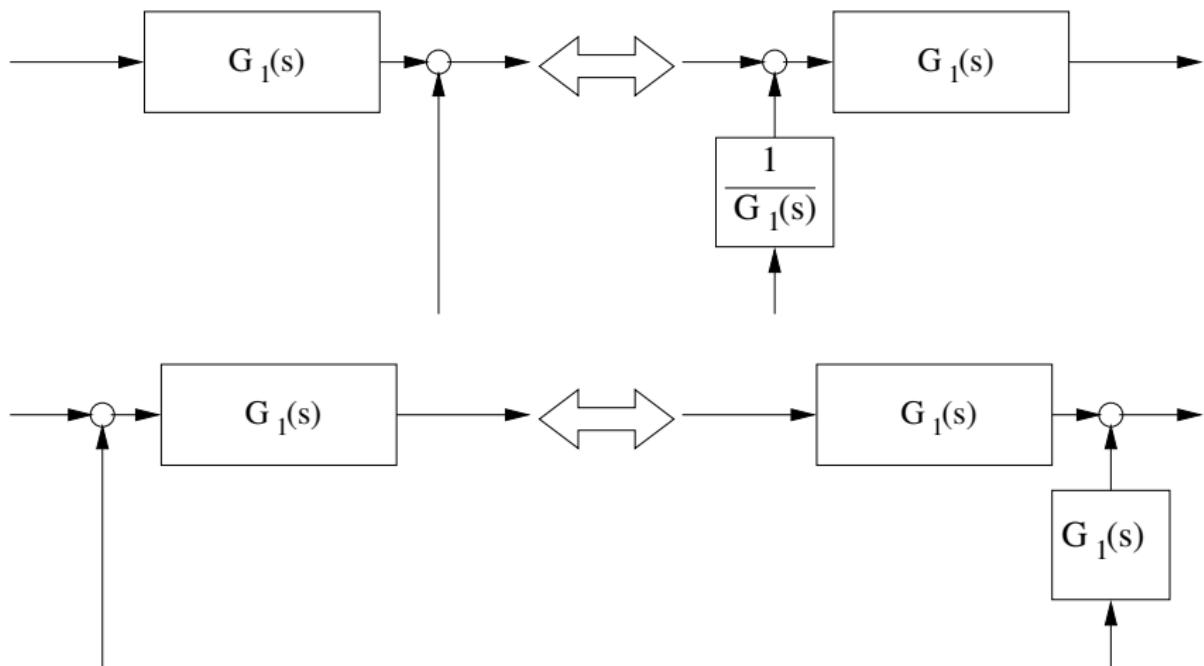
$$X(s) = E(s) \cdot G_1(s)$$

$$E(s) = U(s) + X_M(s)$$

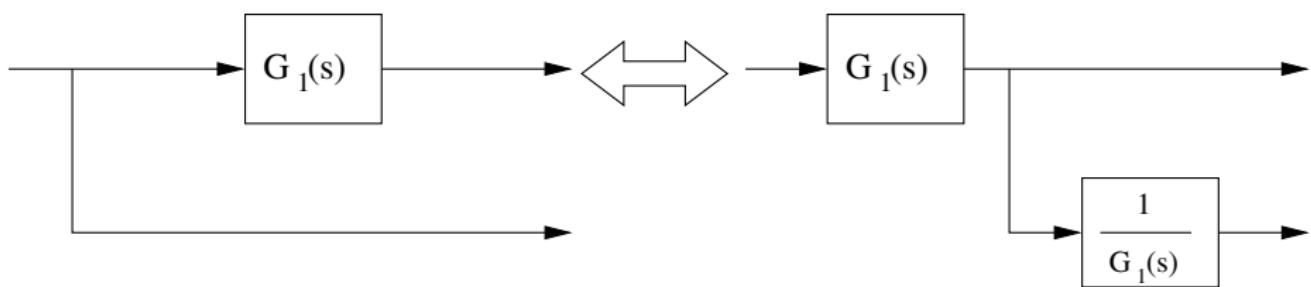
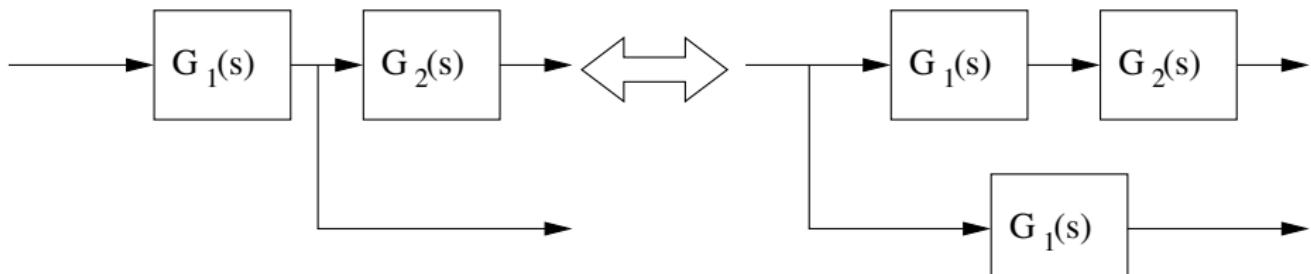
$$X_M(s) = X(s) \cdot G_2(s)$$

$$\Rightarrow Y(s) = U(s) \cdot \frac{G_1(s)}{1 - G_1(s) \cdot G_2(s)}$$

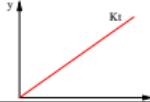
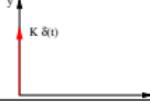
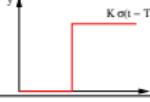
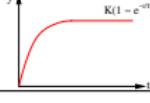
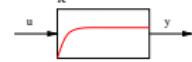
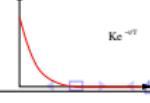
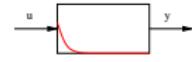
Verschieben von Summationsstellen



Verschieben von Verzweigungsstellen

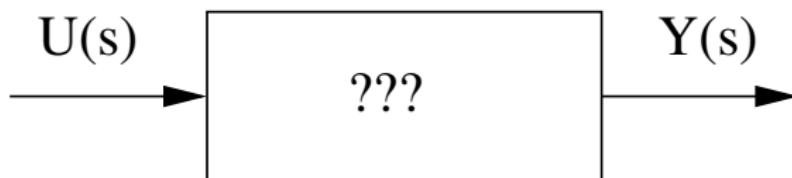


Wichtige Übertragungsfunktionen

Name	DGL	$G(s)$	SprungAW	Symbol
P-Glied	$y = K \cdot u$	K		
I-Glied	$y = K \cdot \int_0^{\infty} u(\tau) d\tau$	$\frac{K}{s}$		
D-Glied	$y = K \cdot \dot{u}$	$K \cdot s$		
TZ-Glied (T_t -Glied)	$y = K \cdot u(t - T)$	$K \cdot e^{-T \cdot s}$		
VZ ₁ -Glied (PT ₁ -Glied)	$T \dot{y} + y = K \cdot u$	$\frac{K}{1+Ts}$		
VD ₁ -Glied (DT ₁ -Glied)	$T \dot{y} + y = KT \dot{u}$	$\frac{K Ts}{1+Ts}$		

Experimentelle Bestimmung der Ü-Fkt. (1)

Genaue physikalische Zusammenhänge sind mitunter nicht bekannt



- Es kann kein math. Modell des Systems aufgestellt werden
- Vorgehen:
 - ▶ Annahme eines (näherungsweise) passenden, parametrisierten Modells
 - ▶ Ermitteln der Parameter durch Messungen im Zeitbereich

Experimentelle Bestimmung der Ü-Fkt. (2)

- Viele praktische Systeme können durch ein Übertragungsglied 2. Ordnung (PT₂-Glied) approximiert werden.
- Normierte Übertragungsfunktion eines PT₂-Gliedes:

$$G_n(s) = \frac{1}{\left(\frac{s}{\omega_0}\right)^2 + 2 \cdot \zeta \cdot \left(\frac{s}{\omega_0}\right) + 1} = \frac{\omega_0^2}{s^2 + 2 \cdot \zeta \cdot \omega_0 \cdot s + \omega_0^2}$$

darin sind:

ω_0 : Kennkreisfrequenz (Resonanzfrequenz) des Systems

ζ : Dämpfung des Systems

- Polstellen von $G_n(s)$...:

$$\lambda_{1,2} = -\zeta\omega_0 \pm \omega_0\sqrt{\zeta^2 - 1}$$

... bestimmen das Zeitverhalten

Experimentelle Bestimmung der Ü-Fkt. (3)

- Normierte Sprungantwort

→ Anregung mit $u(t) = \sigma(t) \bullet \circ U(s) = \frac{1}{s}$:

$$Y_n(s) = G_n(s) \cdot \frac{1}{s} = \frac{\omega_0^2}{s^2 + 2 \cdot \zeta \cdot \omega_0 \cdot s + \omega_0^2} \cdot \frac{1}{s}$$

- Laplace-Rücktransformation (längere Rechnung...)

für $\zeta \leq 1$:

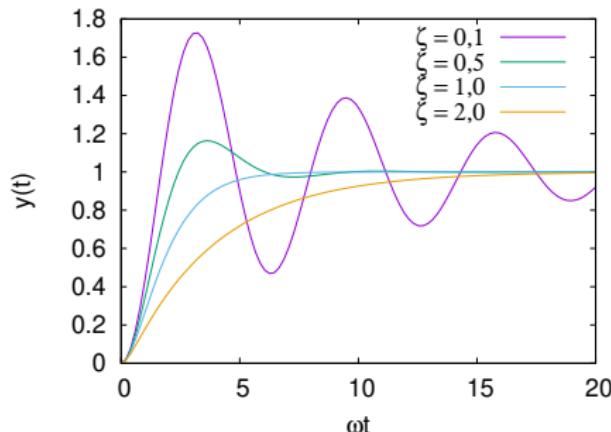
$$y_n(t) = 1 - e^{-\zeta \omega_0 t} \cdot \left(\cos(\sqrt{1-\zeta^2} \omega_0 t) + \frac{\zeta}{\sqrt{1-\zeta^2}} \sin(\sqrt{1-\zeta^2} \omega_0 t) \right)$$

für $\zeta \geq 1$:

$$y_n(t) = 1 - e^{-\zeta \omega_0 t \frac{1}{2}} \cdot \left(\left(1 - \frac{\zeta}{\sqrt{\zeta^2-1}}\right) \cdot e^{-\sqrt{\zeta^2-1} \omega_0 t} + \left(1 + \frac{\zeta}{\sqrt{\zeta^2-1}}\right) \cdot e^{\sqrt{\zeta^2-1} \omega_0 t} \right)$$

Experimentelle Bestimmung der Ü-Fkt. (4)

Sprungantwort eines Systems ist i.d.R. gut messbar



Mögliche Fälle:

- $\zeta < 0$: Instabil (System „explodiert“)
- $\zeta = 0$: System schwingt ungedämpft (Mit Frequenz $F = \frac{\omega_0}{2\pi}$)
- $0 < \zeta < 1$: Abklingende Schwingung
- $\zeta = 1$: „Aperiodischer Grenzfall“
- $\zeta > 1$: Strebt gegen Endwert, keine Schwingung

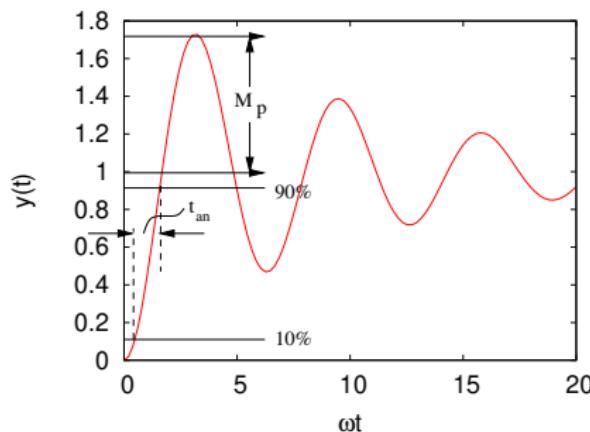
Vorgehen:

- Messen der Sprungantwort
- Ermitteln der Parameter (ζ , ω_0 , Skalierfaktor)

Oft genügen schon ungenaue Werte als Anhaltspunkte

Experimentelle Bestimmung der Ü-Fkt. (5)

Faustregeln ...



Faustformeln:

$(t_s = \text{Ausregelzeit für } \epsilon = 0.05)$

- $\zeta \approx \sin \left(\arctan \left(\frac{1}{\pi} \ln \frac{1}{M_p} \right) \right)$

- $\omega_0 \approx \frac{3}{\zeta \cdot t_s}$

- Wert für $t \rightarrow \infty$: $e_\infty \Rightarrow$
Proportionalfaktor:

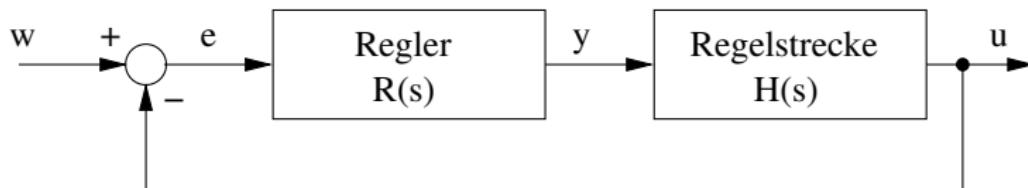
$$K = \frac{1}{e_\infty} - 1$$

Damit nicht-normierte Übertragungsfunktion:

$$G(s) = \frac{K}{1 + \frac{2\zeta}{\omega_0}s + \frac{1}{\omega_0^2}s^2}$$

Entwurf zeitkontinuierlicher Regler

Allgemeine Reglerstruktur:



Übertragungsfunktion:

$$G(s) = \frac{R(s)H(s)}{1 + R(s)H(s)}$$

Idealziel:

- Entwurf eines Reglers so dass gilt: $e(t) = 0 \forall t$
- Kein Einfluss von Störungen auf Regel- und Messgrößen

Theoretisch erreichbar durch $|R(s)| \rightarrow \infty \Rightarrow G(s) \rightarrow 1$

Gütekriterien



Beschränkung auf erreichbare Forderungen

- Der Regelkreis soll stabil bleiben und nicht schwingen
- Der Regelkreis soll der Führungsgröße $w(t)$ unabhängig von äußeren Störeinflüssen möglichst schnell und genau folgen
(→ er soll gutes „*Führungsverhalten*“ zeigen)

Gütekriterien:

- ① Stabilität
- ② Schnelligkeit
- ③ Genauigkeit

Gütekriterien

Beschränkung auf erreichbare Forderungen

- Der Regelkreis soll stabil bleiben und nicht schwingen
- Der Regelkreis soll der Führungsgröße $w(t)$ unabhängig von äußeren Störeinflüssen möglichst schnell und genau folgen
(→ er soll gutes „*Führungsverhalten*“ zeigen)

Gütekriterien:

- ① Stabilität
- ② Schnelligkeit
- ③ Genauigkeit

Im Folgenden näher betrachtet

Stabilität

Definition: E/A-Stabilität (Auch: „BIBO^(*)-Stabilität“)

Ein LTI-System heißt E/A-stabil, wenn für verschwindende Anfangswerte $x^{(i)} = 0$ und ein beschränktes Eingangssignal:

$$|w(t)| < w_{max} \quad \forall t > 0$$

das Ausgangssignal beschränkt bleibt:

$$|u(t)| < u_{max} \quad \forall t > 0$$

Anders ausgedrückt:

- Ein System ist stabil, wenn zu einer beschränkten Eingangsgröße eine beschränkte Ausgangsgröße gehört

(*) (BIBO: Bounded Input Bounded Output)

Asymptotische Stabilität (1)

Definition: Asymptotische Stabilität

Ein LTI-System heißt asymptotisch stabil, wenn seine Ausgangsvariable $u(t)$ mit der Zeit eindeutig nach Null strebt bei einer ebenfalls nach Null strebenden Eingangsvariable $w(t)$:

$$\lim_{t \rightarrow \infty} u(t) = 0 \quad \text{wenn} \quad \lim_{t \rightarrow \infty} w(t) = 0$$

- D.h. ein asymptotisch stabiles System kehrt nach Abklingen einer Störung in seinen Ruhezustand zurück

Bedingung:

- Ein System ist asymptotisch stabil wenn die Pole (=Nullstellen des Nenners der Übertragungsfunktion) einen negativen Realteil haben.

Asymptotische Stabilität (2)

Beispiel PT₂-Glied (s.o.)

- Normierte Übertragungsfunktion:

$$G_n(s) = \frac{\omega_0^2}{s^2 + 2 \cdot \zeta \cdot \omega_0 \cdot s + \omega_0^2} = \frac{\omega_0^2}{(s - \lambda_1)(s - \lambda_2)}$$

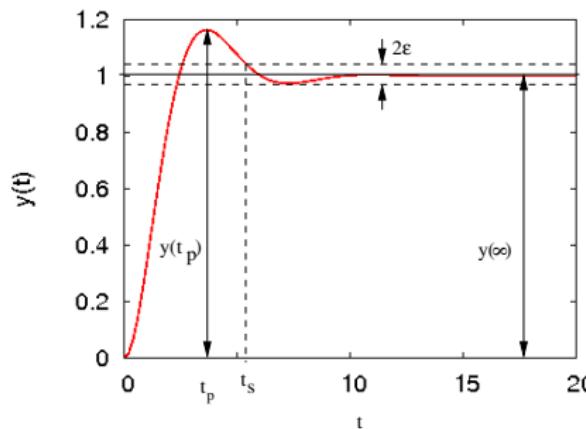
Wobei: $\lambda_{1,2} = -\zeta\omega_0 \pm \omega_0\sqrt{\zeta^2 - 1}$ (Polstellen von $G_n(s)$)

- Instabil für $\zeta < 0$
- Stabil für $0 < \zeta < 1 \Rightarrow \Re\{\lambda_{1,2}\} = -\zeta\omega_0 < 0$
- Stabil für $\zeta > 1$ da $\sqrt{\zeta^2 - 1} < \zeta$

Weitere Kriterien (Hurwitz-, Routh-, Nyquist,- Phasenrand-)
hier nicht behandelt →[Lunze],[Wörn/Brinkschulte]

Dynamisches Verhalten (1)

- Zur Spezifikation von **Stabilität** und **Schnelligkeit**: Betrachte Sprungantwort
- „relative Stabilität“ oder Stabilitätsgüte: Sprungantwort soll möglichst schnell auf stationären Wert gehen
- wichtige Maße: Überschwingweite M_p , Ausregelzeit t_s



- $M_p = \frac{y(t_p)}{y(\infty)}$
- Überschwingen (*overshoot*):
 $M_o = M_p - 1$, $\ddot{u}(\%) = 100 \cdot M_o$

Dynamisches Verhalten (2)

Genauigkeit: Integralkriterien, z.B.:

- „ISE“-Kriterium (*integral of squared error*)

$$Q_1 = \int_0^{\infty} (1 - y(t))^2 dt$$

- „IAE“-Kriterium (*integral of absolute error*)

$$Q_2 = \int_0^{\infty} |1 - y(t)| dt$$

Neben den hier vorgestellten, anwendungsunabhängigen Verfahren werden in der Praxis auch anwendungsabhängige Kriterien verwendet (z.B. „Benchmarkbahnen“ bei Robotern)

PID-Regler

- In der Praxis i.d.R. kein Entwurf neuer Reglertypen
- Stattdessen Anpassen eines Standardreglers, so dass Gütekriterien (s.o.) erfüllt sind
- Am häufigsten verwendet: *PID*-Regler:
 - ▶ Propotional-Anteil
 - ▶ Integral-Anteil
 - ▶ Differenzial-Anteil

PID-Regler: P-Anteil

- P-Anteil: Stellgröße $y(t)$ Proportional zur Regeldifferenz $e(t)$
 - Faktor (Verstärkung): K_p
- ⇒ Übertragungsfunktion:

$$G(s) = \frac{R(s)H(s)}{1 + R(s)H(s)} = \frac{K_p H(s)}{1 + K_p H(s)}$$

- Da für reale Regelstrecke i.A. gilt: $K_p H(s) > 0$ ist $|G(s)| < 1$, also $U(s) \neq W(s)$
- Problem: Bleibende Regelabweichung
- Regler **muss** um einen I-Anteil erweitert werden.

PID-Regler: PI-Anteil

- Übertragungsfunktion eines PI-Reglers mit Nachstellzeit T_N :

$$R(s) = K_P \cdot \left(1 + \frac{1}{T_N s} \right)$$

- Keine bleibende Regelabweichung
- **Aber:** Regelung reagiert langsam
- Abhilfe: Änderungen der Regeldifferenz $e(t)$ stärker betonen
→ Regler um einen D-Anteil erweitern.

PID-Regler und PD-Regler

- Allgemeine Übertragungsfunktion eines PID-Reglers:

$$R(s) = K_P \cdot \left(1 + \frac{1}{T_N s} + T_V s \right)$$

- Falls höhere Verstärkung erforderlich und einfaches Erhöhen von K_p zur Instabilität führt: **PD**-Regler:

$$R(s) = K_P \cdot (1 + T_V s)$$

P & I Qualitativ

P-Glied

- Verändert Stellsignal proportional zu Regeldifferenz
("Je größer die Abweichung desto größer die Stellgröße")
- Verstärkungsfaktor K_p bestimmt Regelgeschwindigkeit
("Je höher desto schneller")
- Zu hoher Verstärkungsfaktor führt zu Instabilität
- Es bleibt eine dauerhafte Regeldifferenz

I-Glied

- Integriert die Regeldifferenz
("Solange Regelabweichung → Stellgröße verändern")
- Regeldifferenz wird immer ausgeregelt
- Kann zu Instabilität führen

D Qualitativ

D-Glied

- Differenziert die Regeldifferenz
„Je stärker die Änderung der Regelabweichung desto stärker muss Stellgröße verändert werden“)
- Verbessert i.d.R. Regelgeschwindigkeit und dynamische Regelabweichung
- Verstärkt besonders hochfrequente Anteile (Rauschen)
→ Neigung zum Schwingen

Einstellregeln für PID-Regler (1)

Methode nach Ziegler und Nichols

- ① Zunächst reiner P-Regler mit minimalem K_p
- ② K_p erhöhen, bis ungedämpfte Schwingungen ausgeführt werden
(→ Stabilitätsgrenze und krit. Verstärkung K_{pkr} erreicht)
- ③ Schwingungsdauer T_{kr} bei krit. Verstärkung messen

Dann Regelparameter:

P-Regler	$K_p = 0,5K_{pkr}$
PI-Regler	$K_p = 0,45K_{pkr}$ $T_N = 0,85 T_{kr}$
PID-Regler	$K_p = 0,6K_{pkr}$ $T_N = 0,5 T_{kr}$ $T_V = 0,12 T_{kr}$

Einstellregeln für PID-Regler (2)



Weitere Verfahren

- Kompensationsreglerentwurf
- T-Summen Einstellverfahren
- Einstellregel von Chien, Hrones und Reswick (CHR)

Hier nicht weiter vertieft→[Wörn/Brinkschulte]

Unstetige Regelung

Statische Kennlinie eines Reglers

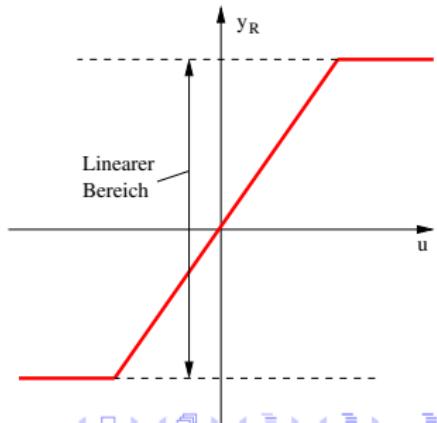
- Beschreibt das Übertragungsverhalten für statische (d.h. konstante) Signale
- Entsteht aus DGL durch Nullsetzen aller Ableitungen:

$$\dots T_2^2 \cdot \ddot{y} + T_1 \cdot \dot{y} + y = \dots + K_2^2 \cdot \ddot{u} + K_1 \cdot \dot{u} + K_0 \cdot u$$

$$\Rightarrow y_R = K_0 \cdot u$$

Stetige Regler

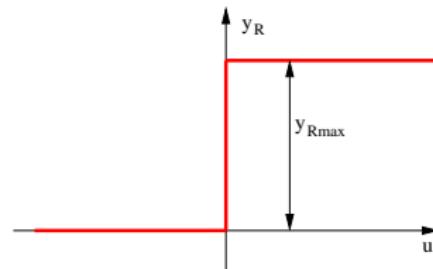
- Bisher betrachtete Regler:
Statische Kennlinie ist eine stetige Funktion
- Voraussetzung für LTI-System:
statische Kennlinie ist zudem (zumindest stückweise) linear



Unstetige Regelung

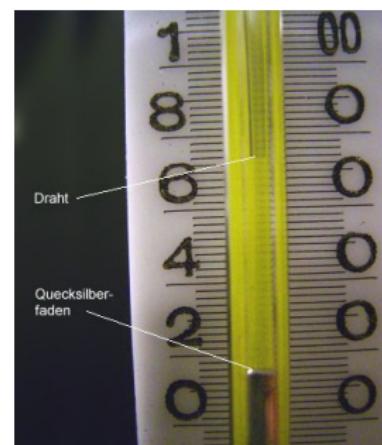
Dagegen unstetiger Regler

- Einfachste Form: *Zweipunktregler*
- Nur zwei diskrete Zustände
 - ① Regeldifferenz $> 0 \rightarrow$ Ein
 - ② Regeldifferenz $\leq 0 \rightarrow$ Aus



Technische Realisierung...

- ... ist denkbar einfach ...
(z.B. Bimetallschalter,
Kontaktthermometer, Relais, ...)
- ... und damit preiswert.



Zweipunktregler

Einfachster unstetiger Regler

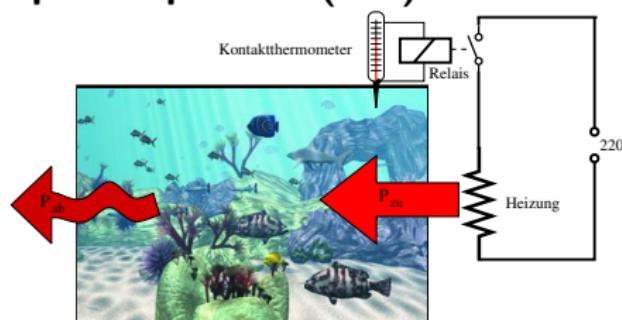
- Wohl der am weitesten verbreitete Regler (da einfach und billig)
- Zahlreiche Anwendungsbeispiele
Kühlschrank, Bügeleisen, Heizkörperventil,
Lichtmaschinen-Reglerschalter, ...

Hauptnachteil:

- Istwert pendelt ständig um den Sollwert
 - Bei einfachen Systemen kein Problem
 - Durch geeignete Maßnahmen (Meßgenauigkeit) lässt sich die Pendelamplitude sehr weit reduzieren
- Dann auch für komplexere Anwendungen nutzbar
- Dann ist aber auch der Preisunterschied zu stetigen Reglern nicht mehr groß

Zweipunktregler ohne Hysterese

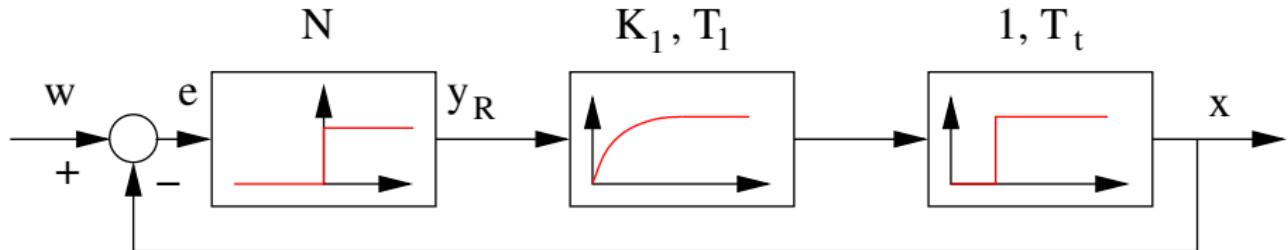
Beispiel: Aquarium (s.o.)



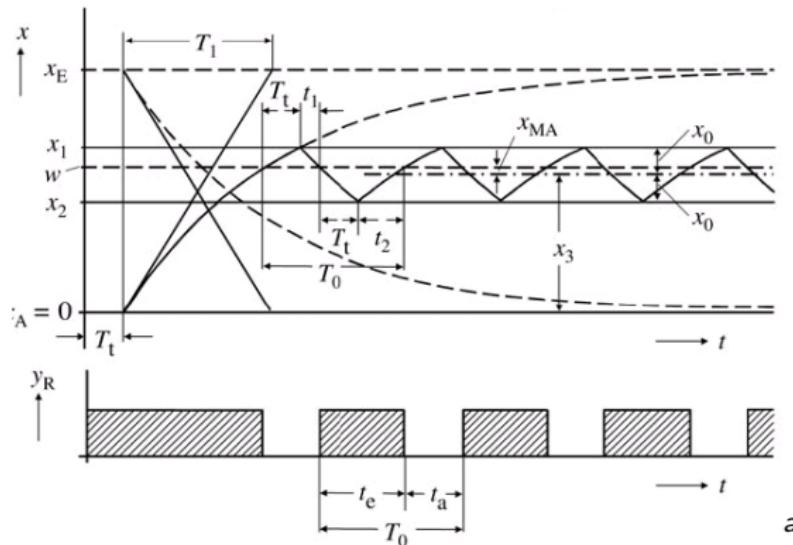
$$x = K_1 \cdot \left(1 - e^{-\frac{t}{T_1}}\right) \cdot \sigma(t)$$

- Modellierung der Thermometer-Trägheit durch Laufzeit T_t
- Annahme: Thermometer-Schaltpunkt in beide Richtungen gleich

Modell:



Temperaturverlauf (1)



- Inbetriebnahme bei $t = 0$
- Solltemperatur w
- Anfangstemperatur x_A
- Endtemperatur x_E

^aAus [Zacher/Reuter]

- Periodische Temperaturschwankung um Mittelwert x_3 ($\neq w!$)
- Amplitude der Schwankung x_0 , Mittelwertabweichung x_{MA}

Temperaturverlauf (2)

Schwankungsamplitude x_0

- x_1, x_2 : oberer/unterer Grenzwert

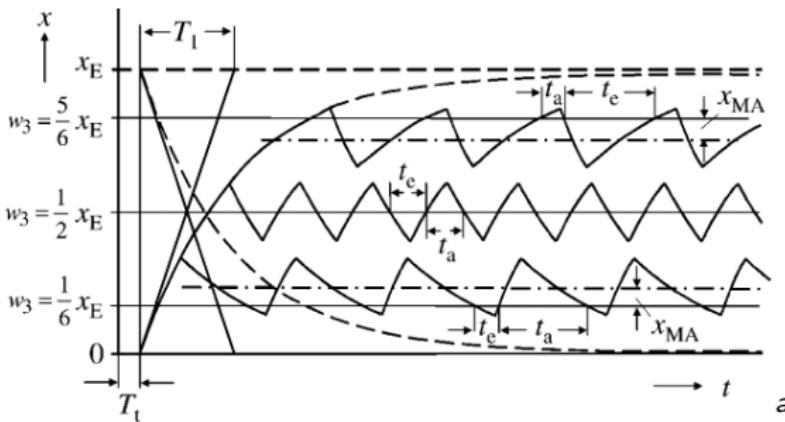
$$\begin{aligned}x_1 &= w + (x_E - w) \cdot (1 - e^{-\frac{T_t}{T_1}}) \quad , \quad x_2 = w \cdot e^{-\frac{T_t}{T_1}} \\&\Rightarrow 2 \cdot x_0 = x_1 - x_2 = x_E \cdot (1 - e^{-\frac{T_t}{T_1}})\end{aligned}$$

N.B: hängt nicht von w ab

Mittelwert x_3 und Mittelwertabweichung x_{MA}

$$\begin{aligned}x_3 &= \frac{x_1 + x_2}{2} = \frac{1}{2} \cdot \left(x_E \cdot (1 - e^{-\frac{T_t}{T_1}}) + 2w \cdot e^{-\frac{T_t}{T_1}} \right) \\x_{MA} &= w - x_3 = \left(w - \frac{x_E}{2} \right) \cdot \left(1 - e^{-\frac{T_t}{T_1}} \right)\end{aligned}$$

Temperaturverlauf (3)

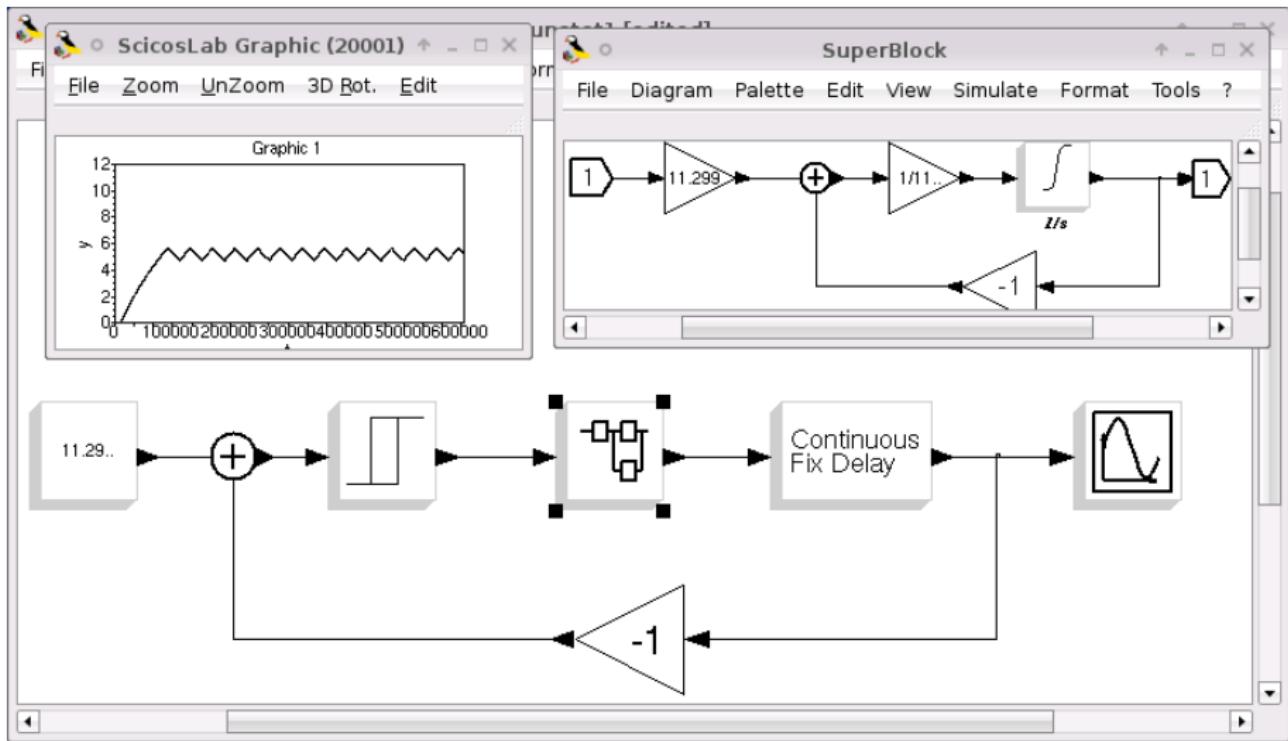


- Kurvenform ist sollwertabhängig
- Für $w = \frac{x_E}{2} \rightarrow x_{MA} = 0$
Symmetrischer Betrieb
- Dann: $t_e = t_a = T_t$

^aAus [Zacher/Reuter]

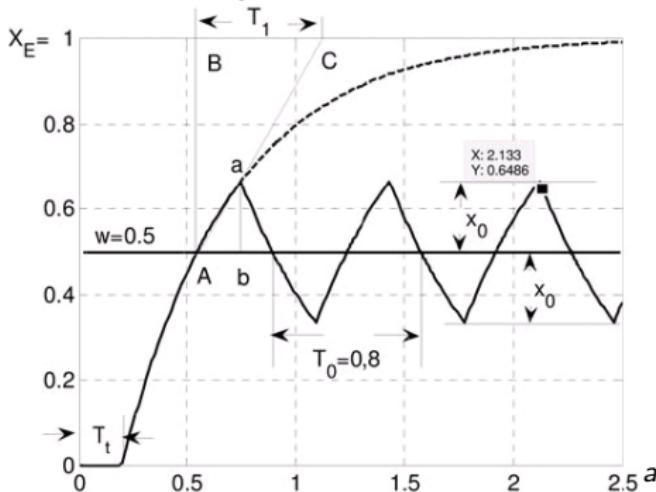
→ Schwingungsdauer im symmetrischen Betrieb:
 $T_0 = 2t_e + 2t_a = 4T_t$

Modell in SciCosLab



Linearisierung

Für kleine x_0 : Annähernd lineare Verläufe



- Dreiecke ABC und Aab sind ähnlich

$$\Rightarrow \frac{\overline{AB}}{\overline{BC}} = \frac{\overline{ab}}{\overline{Ab}} \Rightarrow \frac{\frac{x_E}{2}}{\overline{T_1}} = \frac{x_0}{\overline{T_t}}$$

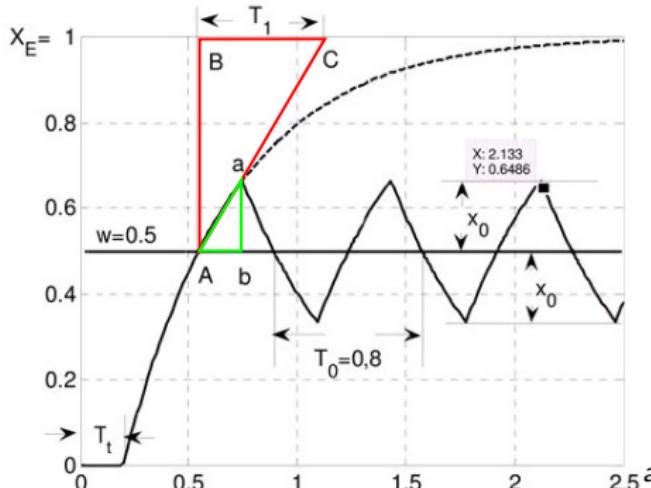
⇒ Faustformel:

$$x_0 \approx \frac{x_e}{2} \cdot \frac{T_t}{T_1}$$

^aAus [Zacher/Reuter]

Linearisierung

Für kleine x_0 : Annähernd lineare Verläufe



- Dreiecke ABC und Aab sind ähnlich

$$\Rightarrow \frac{\overline{AB}}{\overline{BC}} = \frac{\overline{ab}}{\overline{Ab}} \Rightarrow \frac{x_E}{T_1} = \frac{x_0}{T_t}$$

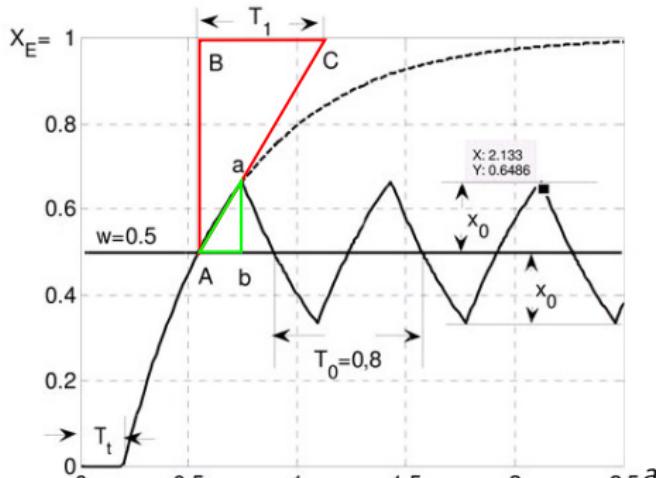
⇒ Faustformel:

$$x_0 \approx \frac{x_e}{2} \cdot \frac{T_t}{T_1}$$

^aAus [Zacher/Reuter]

Linearisierung

Für kleine x_0 : Annähernd lineare Verläufe



- Dreiecke ABC und Aab sind ähnlich

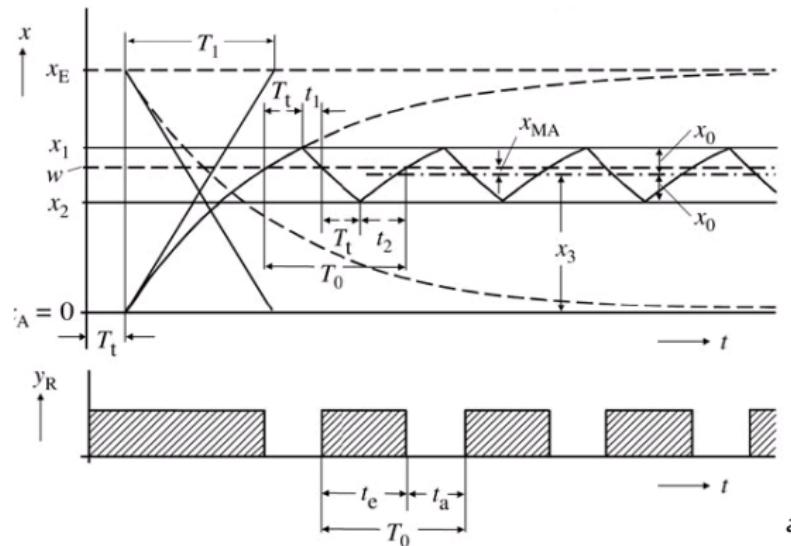
$$\Rightarrow \frac{\overline{AB}}{\overline{BC}} = \frac{\overline{ab}}{\overline{Ab}} \Rightarrow \frac{x_E}{2} = \frac{x_0}{T_1}$$

⇒ Faustformel:

$$x_0 \approx \frac{x_e}{2} \cdot \frac{T_t}{T_1}$$

^aAus [Zacher/Reuter]

Schaltfrequenz und Schwingdauer (1)



^aAus [Zacher/Reuter]

- Schwingdauer
 $T_0 = 2T_t + t_1 + t_2$
- Ausserdem $w = x_1 \cdot e^{-\frac{t}{T_1}}$
- und (s.o.):
 $x_1 =$
 $w + (x_E - w) \cdot (1 - e^{-\frac{T_t}{T_1}})$

→ ... (längere Rechnung): $t_1 = T_1 \cdot \ln \left(\frac{x_E}{w} + \left(1 - \frac{x_E}{w}\right) \cdot e^{-\frac{T_t}{T_1}} \right)$

Schaltfrequenz und Schwingdauer (2)



- Analog für t_2 :

$$\begin{aligned} w - x_2 &= (x_e - x_2)(1 - e^{-\frac{t_2}{T_1}}) \\ e^{-\frac{t_2}{T_1}} &= 1 - \frac{w - x_2}{x_e - x_2} = \frac{x_e - w}{x_e - x_2} \\ \Rightarrow t_2 &= R_1 \cdot \ln \frac{x_e - x_2}{x_e - w} \end{aligned}$$

- mit (s.o. $x_2 = w \cdot e^{-\frac{T_t}{T_1}}$):

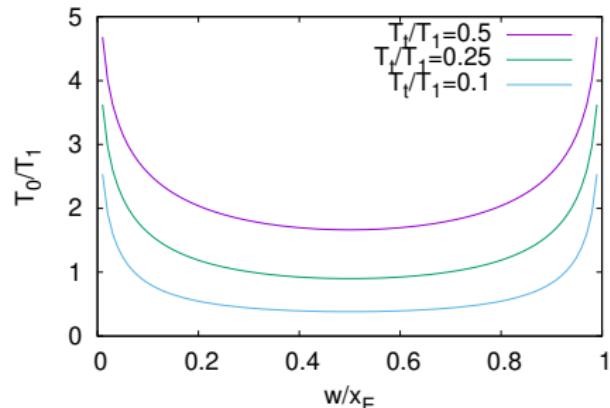
$$t_2 = T_1 \cdot \ln \frac{x_e - w \cdot e^{-\frac{T_t}{T_1}}}{x_e - w}$$

$$\Rightarrow T_0 = 2T_t + T_1 \cdot \ln \left[\left(\frac{1}{1 - \frac{w}{x_e}} - e^{-\frac{T_t}{T_1}} \right) \cdot \left(\frac{x_e}{w} - e^{-\frac{T_t}{T_1}} \right) \right]$$

Schaltfrequenz und Schwingdauer (3)

Abhängigkeit der Schwingdauer vom Sollwert:

- Für $w = \frac{x_E}{2}$ ist unabhängig von T_t Schwingdauer minimal, d.h. die Schaltfrequenz maximal
- Vernünftiger Regelbereich etwa $0,2x_E < w < 0,8x_E$

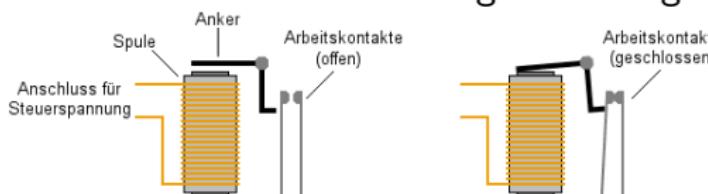
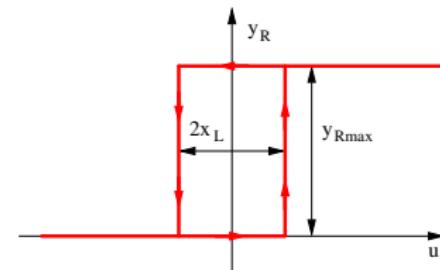


- Für $T_t \rightarrow 0$ wird die Schwankungsbreite 0
 \Rightarrow Totzeit sollte minimiert werden
- Allerdings wird dann auch $T_0 = 0$,
 d.h. die Schaltfrequenz wird unendlich
- Mechanische Kontakte stoßen hier schnell an Grenzen

Zweipunktregler mit Hysterese

Hysterese

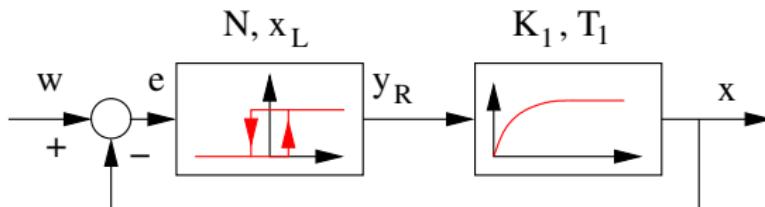
- Reale Zweipunktregler sind stets Hysterese-behaftet
- Im Gegensatz zum Idealfall ohne Hysterese **zwei** Schaltschwellen:
 - ① Einschalten bei $x = +x_L$
 - ② Abschalten bei $x = -x_L$
- Entsteht z.B. durch Restmagnetisierung bei Relais:



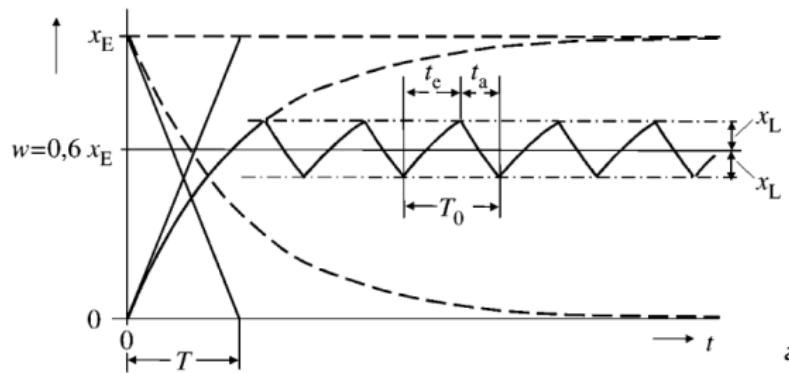
- Einschaltstrom ist höher als Abschaltstrom

Beispiel: Aquarium (s.o.)

Modell:



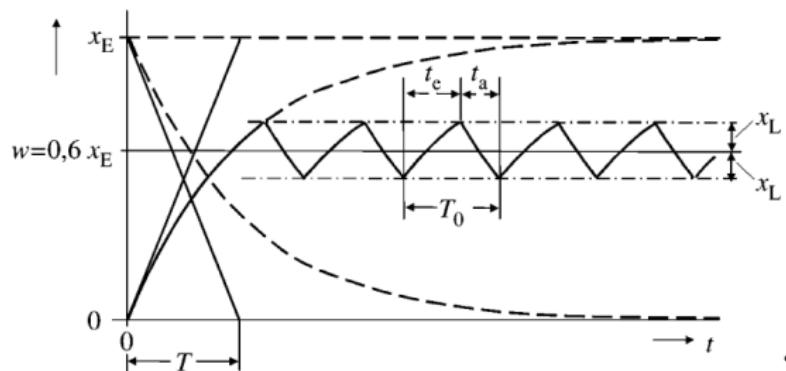
Temperaturverlauf:



- Inbetriebnahme bei $t = 0$
- Anfangstemperatur 0
- Endtemperatur x_E
- Solltemperatur $0 \leq w \leq x_E$

^aAus [Zacher/Reuter]

Temperaturverlauf



- Periodische Schwankung
- **Symmetrisch** um Sollwert w
- Amplitude: $2x_L$

^aAus [Zacher/Reuter]

- Schwingdauer und Frequenz:

$$2 \cdot x_L = (x_E - w + x_L) \cdot (1 - e^{-\frac{t_e}{T}})$$

$$\Rightarrow t_e = T \cdot \ln \frac{x_E - w + x_L}{x_E - w - x_L}$$

$$w - x_L = (w + x_L) \cdot e^{-\frac{t_a}{T}}$$

$$\Rightarrow t_a = T \cdot \ln \frac{w + x_L}{w - x_L}$$

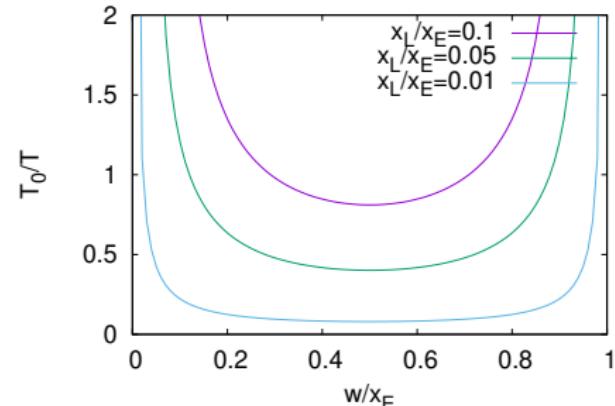
⇒ t_e, t_a proportional zu T , für $w = \frac{x_E}{2}$ ist $t_a = t_e$

Schwingdauer und Frequenz (1)

- Schwingdauer $T_0 = t_e + t_a$

$$\Rightarrow T_0 = T \cdot \left(\ln \frac{x_E - w + x_L}{x_E - w - x_L} + \ln \frac{w + x_L}{w - x_L} \right)$$

- Für $w = \frac{x_E}{2}$ ist die Schwingdauer minimal, d.h. die Schaltfrequenz maximal
- Vernünftiger Regelbereich etwa $0,2x_E < w < 0,8x_E$



Schwingdauer und Frequenz (2)

- Ziel für möglichst kleine Schwingungsbreite: kleine Hysterese
- Allerdings wird für $x_L \rightarrow 0$ die Schwingungsdauer $T_0 \rightarrow 0$, d.h. die Schwingungsfrequenz $f_0 = \frac{1}{T_0} \rightarrow \infty$
- Die maximale Frequenz ergibt sich bei $w = \frac{x_E}{2}$, dann gilt

$$T_{0min} = T \cdot 2 \cdot \ln \frac{0,5x_E + x_L}{0,5x_E - x_L} = 2T \ln \frac{1 + 2\frac{x_L}{x_E}}{1 - 2\frac{x_L}{x_E}}$$

- Näherung (Reihenentwicklung) für $\frac{2x_L}{x_E} \ll 1$:

$$T_{0min} \approx 2 \cdot T \cdot 2 \cdot 2 \cdot \frac{x_L}{x_E} = 8T \cdot \frac{x_L}{x_E}$$

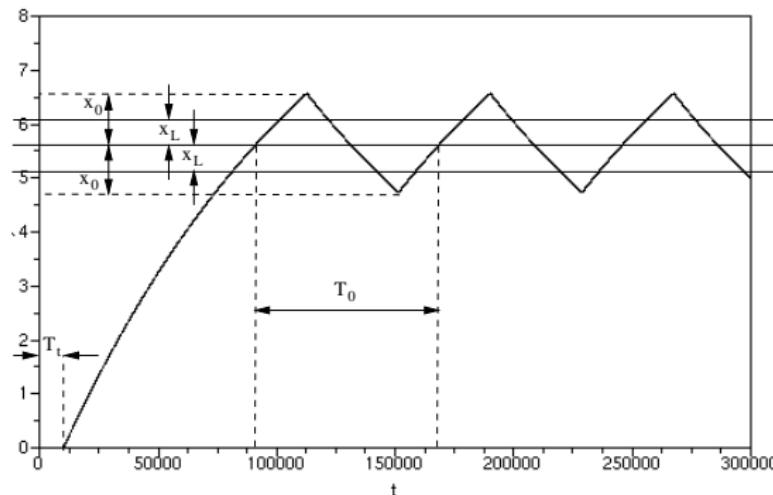
- Maximale Schaltfrequenz:

$$f_{0max} \approx \frac{x_E}{8Tx_L}$$

- Exakte Werte durch Simulation ...

Simulation

- Simulation in Scicos: Aquarium mit Hysterese ($x_L = 0.5 \cdot x_E$) und Totzeit $T_t = 10000$

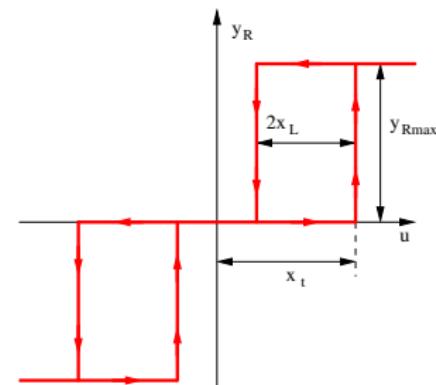


- Daraus abgelesen:
 - ▶ $T_0 = 168.100 - 91.490 = 76.600\text{s}$
 - ▶ $2x_0 = 6,6^\circ\text{C} - 4,7^\circ\text{C} = 1,9^\circ\text{C}$

DreiPunktregler

- Zweipunktregler sind für Stellglieder mit Motorantrieb nicht gut geeignet (keine Möglichkeit zur Richtungsumkehr)

- Daher: *DreiPunktregler*
- Weiterer Vorteile:
- Ruhezustand ohne Dauerschwingen ist möglich
- „Abgestufte“ Reaktion



DreiPunktregler: Statische Kennlinie

Motivation

Vorgehensweise bisher:

- Eigenschaften der Regelstrecke bestimmen
(z.B. experimentell aus Sprungantwort oder durch Modellierung)
- Wahl eines Reglers (z.B. PID), Festlegung seiner Anforderungen
(Überschwingen, Ausregelzeit)
- Simulation, Optimierung, Test

Probleme dabei:

- Eigenschaften der Regelstrecke sind nicht immer konstant und nicht einfach zu berücksichtigen
- „Experten“ können das System dennoch beherrschen

→ **Frage:** Wie kann man Expertenwissen auf Regler übertragen?

Regelbasis

Beispiel:

- Der Bremsweg eines Zuges hängt ab von Beladung, Wetter, Steigung
...
- Trotzdem bekommt ein erfahrener Lokführer i.d.R eine Zielbremsung hin (.. und das ohne große Physik-Kenntnisse)
- Würde man den Lokführer fragen, wie er denn den Bremsweg „regelt“, so wäre die Antwort in etwa:

„*WENN es nicht regnet UND es warmes Wetter ist, UND viele Passagiere an Bord sind, DANN ...*“

- Ziel bei der Entwicklung eines „Expertenreglers“:
eine möglichst vollständige Liste solcher Regeln zusammenstellen, in der alle vorstellbaren Fälle berücksichtigt sind
- Diese Liste ist die **Regelbasis**

Unscharfe Begriffe

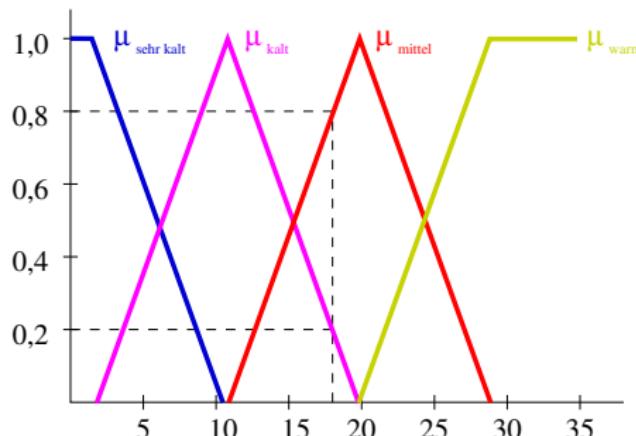
- **Fragen:**

- ▶ was genau ist „warmes Wetter“ (wieviele $^{\circ}\text{C}$...)
- ▶ wieviel sind „viele Passagiere“ (genaue Anzahl ...)

- Umgangssprachliche Begriffe wie „warm“, „kalt“, „schnell“ sind zwar verständlich, aber dennoch unscharf (engl.: *fuzzy*)
- Gesucht: math. Methoden zur Verarbeitung unscharfer Begriffe
- Erster Schritt: *Fuzzyfizierung*:
 - ▶ Umwandlung exakter Eingangsgrößen in *linguistische Variablen*
 - ▶ Ermitteln eines Maßes der „Zugehörigkeit“ des Wertes zu unscharfen Begriffen anhand von Zugehörigkeitsfunktionen μ .

Fuzzyfizierung

Beispiel: Temperatur $\theta = 18^\circ C$:



→ „zu 20% kalt und zu 80% mittel“

- $\mu_{\mathbb{A}}$: Zugehörigkeitsgrad zu einer *unscharfen Menge*² \mathbb{A}
- Wird gewöhnlich aus Trapez-/Dreieckformen (s. Beispiel) oder Gaußfunktionen zusammengesetzt

Inferenz

Nächster Schritt: Verarbeitung *Inferenz* der linguistischen Variablen

- Anwenden der Regelbasis auf die „fuzzifizierten“ Eingangsgrößen
- Aus mehr oder weniger erfüllten Prämissen der Regeln („WENNUND ...“) ergeben sich Folgerungen (Konklusionen) („....DANN.....“)
- Diese können sich teilweise widersprechen
- Auch hier ergeben sich unscharfe Größen:
 - ▶ Folgerungen sind meist nicht zu 100% erfüllt
 - ▶ „Erfülltheitsgrad“ als Maß der Wirksamkeit
- Inferenz umfasst drei Schritte:
 - ① *Aggregation*: Bestimmen des Erfülltheitsgrades der Prämisse
 - ② *Implikation*: Bestimmen des Erfülltheitsgrades der Konklusion
 - ③ *Akkumulation*: Zusammenfassen der Ergebnisse aus allen Regeln

Aggregation

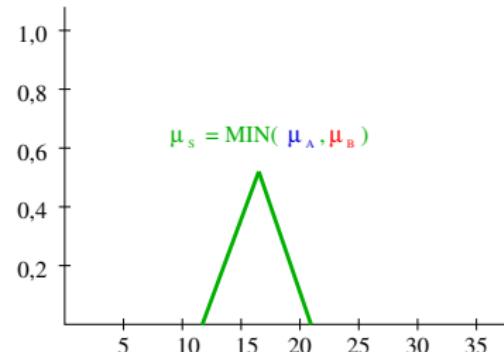
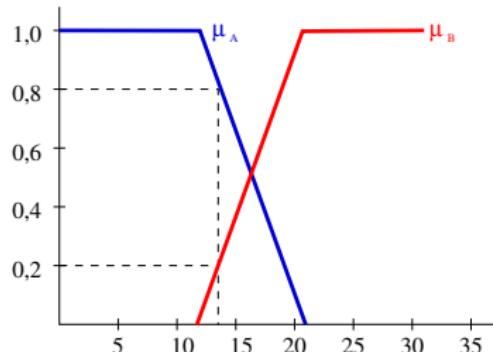
- Die Regelbasis besteht aus Prämissen und zugeordneten Konklusionen:

Regel Nr.	Prämissen	Konklusion
1	WENN ... UND ...	DANN
2	WENN ... ODER ...	DANN
3	WENN NICHT ...	DANN
...

- Aggregation: Anwenden der Prämissen auf linguistische Variablen
- Fragen:
 - Wie bildet man UND- und ODER-Verknüpfungen linguistischer Variablen?
 - Was ist das „Gegenteil“ (NICHT) einer linguistischen Variablen?

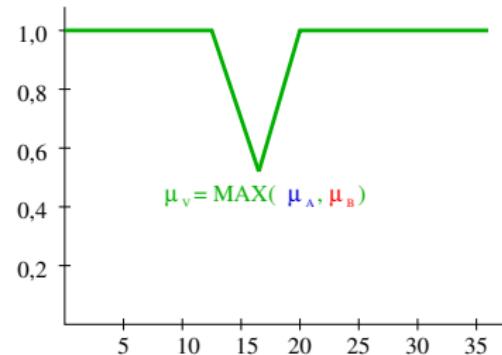
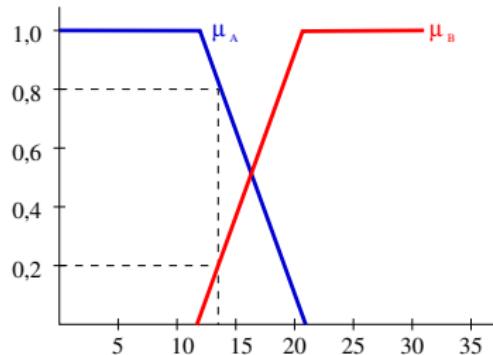
UND: Schnittmenge

- Die Schnittmenge μ_S zweier unscharfer Mengen³ μ_A und μ_B entspricht der logischen UND-Verknüpfung
- wird als *t-Norm* ($\mu_A \text{ UND } \mu_B$) bezeichnet
- Entspricht dem Minimum-Operator: $\mu_S = \text{MIN}(\mu_A, \mu_B)$:



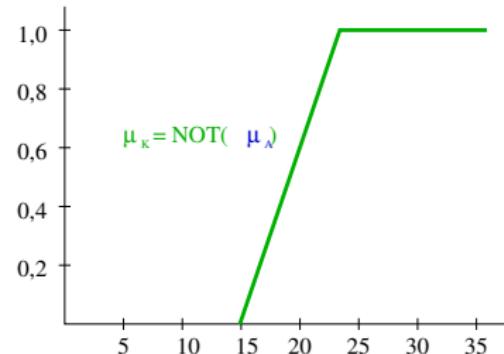
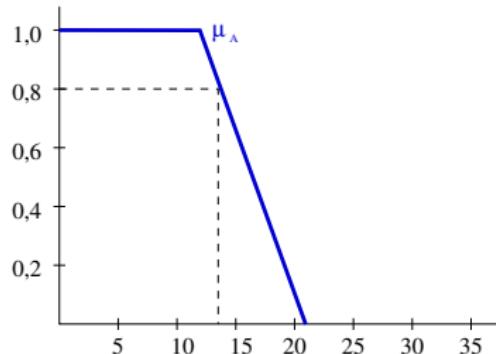
ODER: Vereinigungsmenge

- Die Vereinigungsmenge μ_V zweier unscharfer Mengen μ_A und μ_B entspricht der logischen ODER-Verknüpfung
- wird als *t-CoNorm* ($\mu_A \text{ ODER } \mu_B$) bezeichnet
- Entspricht dem Maximum-Operator: $\mu_V = \text{MAX}(\mu_A, \mu_B)$:



NICHT: Komplementärmenge

- Die Komplementärmenge $\mu_{\bar{A}}$ einer unscharfen Menge μ_A entspricht dem logischen NICHT-Operator
- Entspricht der Subtraktion von 1: $\mu_{\bar{A}} = NOT(\mu_A) = 1 - \mu_A$:



Implikation

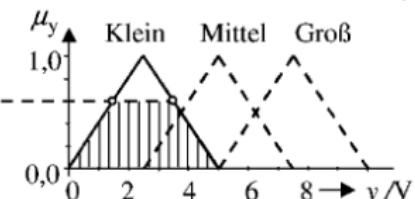
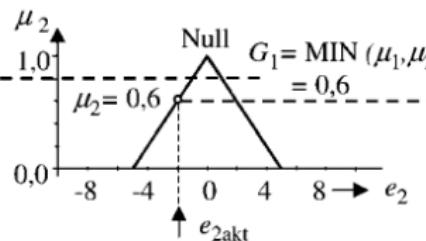
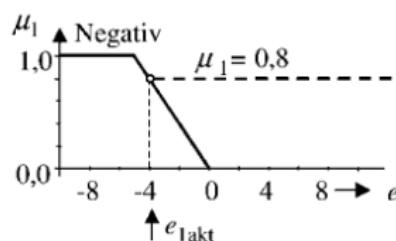
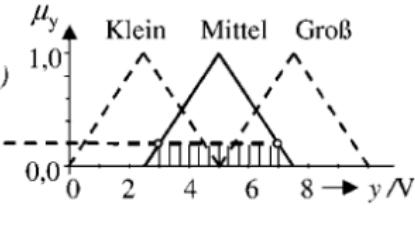
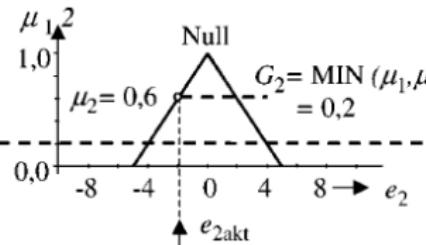
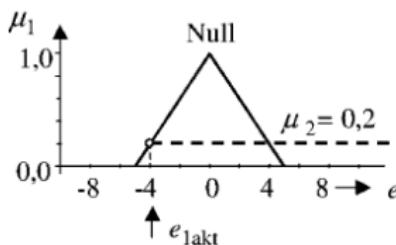
- Regeln der Regelbasis liefern für jeder Prämisse wiederum linguistische Variablen als Ergebnis
- Diese werden mit dem Erfülltheitsgrad der Prämisse gewichtet
- Zu betrachten: alle *aktiven Regeln*: Alle Regeln mit einem Erfülltheitsgrad > 0
- Beispiel: Regelbasis:

Regel Nr.	Prämisse	Konklusion
1	WENN e_1 ist negativ UND e_2 ist null	DANN y ist klein
2	WENN e_1 ist null UND e_2 ist null	DANN y ist mittel

Implikation: Beispiel

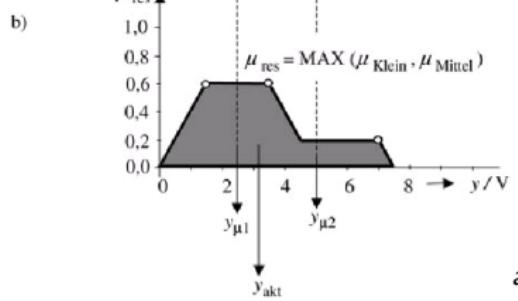
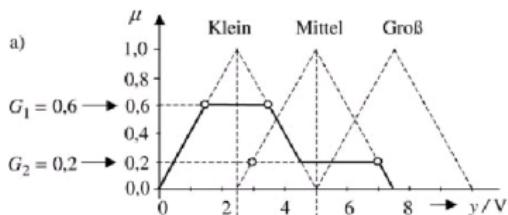
WENN-Teil

DANN-Teil

Regel 1:**Regel 2:**

Akkumulation

a) Jede aktive Regel liefert eine unscharfe Ergebnismenge



b) Ergebnismengen werden durch ODER-Verknüpfung zu einer unscharfen Vereinigungsmenge zusammengefaßt

^aAus [Zacher/Reuter]

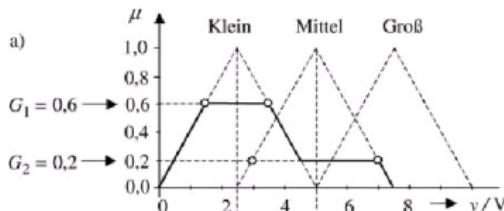
Defuzzyfizierung (1)

- Ergebnis der Inferenz: unscharfe Menge („Fläche im Diagramm“)
- Defuzzyfizierung: daraus einen scharfen Wert ermitteln
- Verschiedene Methoden sind möglich, meist wird mit der Flächenschwerpunkt- oder COG⁵-Methode gearbeitet
- Flächenschwerpunkt der unscharfen Menge $\mu_{res}(y)$:

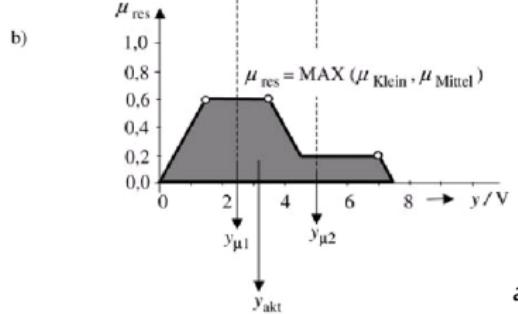
$$y_{akt} = \frac{\int_a^b y \cdot \mu_{res}(y) dy}{\int_a^b \mu_{res}(y) dy}$$

Defuzzyfizierung (2)

- Im Beispiel: Fläche besteht aus Rechtecken und Dreiecken



→ einfach zu berechnen:

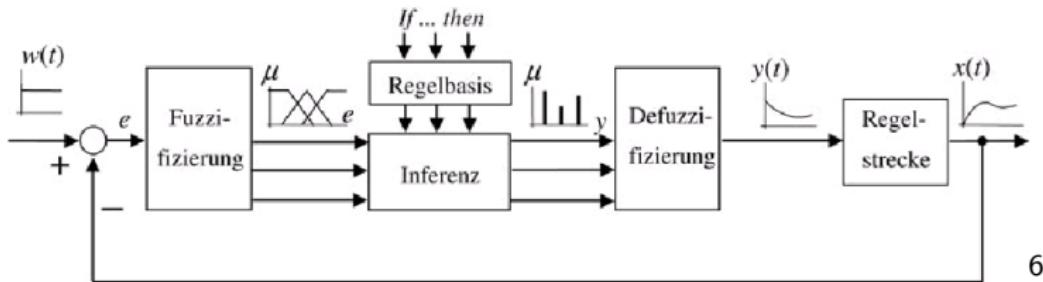


$$\begin{aligned}
 y_{akt} &= \frac{G_1 \cdot y_{\mu 1} + G_2 \cdot y_{\mu 2}}{G_1 + G_2} \\
 &= \frac{0,6 \cdot 2,5 + 0,2 \cdot 5}{0,6 + 0,2} \\
 &= 3,125
 \end{aligned}$$

^aAus [Zacher/Reuter]

Test des Reglers

Aufbau eines Regelkreises mit Fuzzy-Regler:



6

- Ein Fuzzy-Regler muss in der Regel getestet, evtl. auch iterativ nachgebessert werden
- Erst hier zeigt sich, ob die Regelbasis brauchbar ist
- Hierzu können Simulationen in MATLAB/ScicosLab hilfreich sein

Eigenschaften von Fuzzy-Reglern

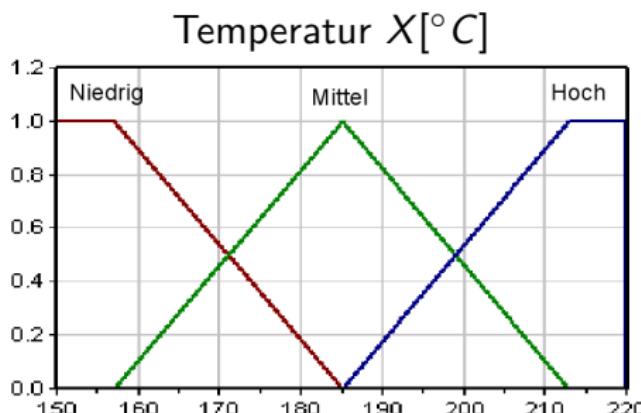
- Im Vergleich zu „klassischen“ Reglern sind Fuzzy-Regler ...
 - ▶ .. robust: behalten stabiles Verhalten, auch bei Änderungen der Regelstreckenparameter
 - ▶ .. mit weniger Aufwand zu entwickeln
- Typische Einsatzgebiete:
 - ▶ Haushaltsgeräte
 - ▶ Kraftfahrzeuge
 - ▶ Medizingeräte

Fuzzy-Beispiel: Temperaturregelung

- Ein befragter Experte beschreibt die Temperatur X eines Ofens mit den Begriffen „niedrig“, „mittel“ und „hoch“
- Der interessierende Temperaturbereich ist: $150^{\circ}\text{C} \leq X \leq 220^{\circ}\text{C}$
- Für eine effizientere Regelung soll auch die Temperaturänderung \dot{X} erfasst werden
- Für diesen „Temperaturtrend“ werden die Begriffe „fallend“, „gleichbleibend“ und „steigend“ definiert
- Der interessierende Bereich ist dabei: $-10\frac{{}^{\circ}\text{C}}{\text{s}} \leq \dot{X} \leq +10\frac{{}^{\circ}\text{C}}{\text{s}}$
- Nach Befragung des Experten werden für die Heizleistung Y die Begriffe „sehr niedrig“, „niedrig“, „mittel“, „hoch“ und „sehr hoch“ definiert.

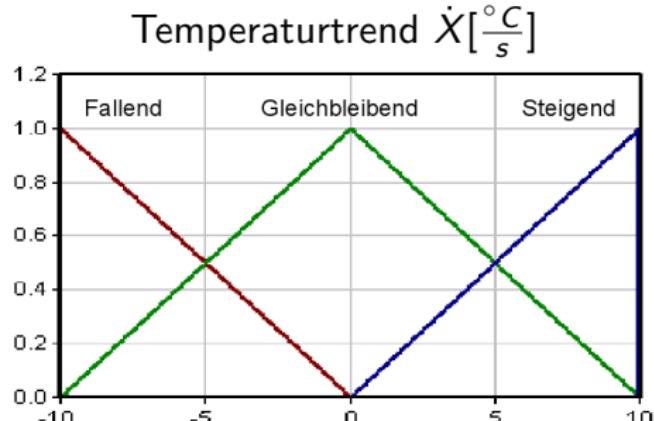
Fuzzy-Beispiel: Fuzzyfizierung (1)

Unscharfe Mengen für:



- z.B.: $X = 180^{\circ}\text{C}$

$$\begin{aligned} \rightarrow \mu_{\text{niedrig}}(X) &= 0,2 \\ \rightarrow \mu_{\text{mittel}}(X) &= 0,8 \\ \rightarrow \mu_{\text{hoch}}(X) &= 0 \end{aligned}$$

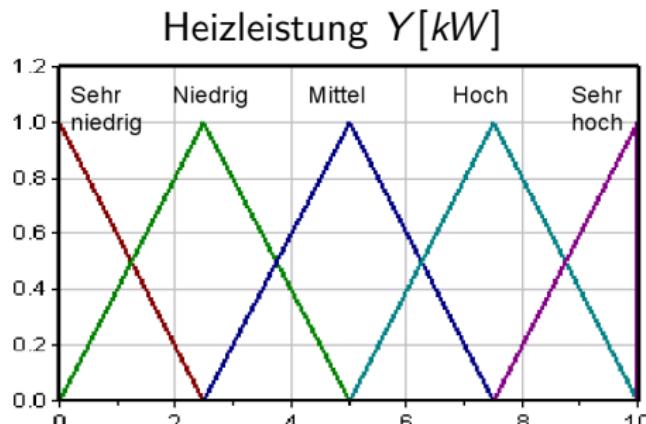


- z.B.: $\dot{X} = -5 \frac{\text{ }^{\circ}\text{C}}{\text{s}}$

$$\begin{aligned} \rightarrow \mu_{\text{fallend}}(\dot{X}) &= 0,5 \\ \rightarrow \mu_{\text{gleichbleibend}}(\dot{X}) &= 0,5 \\ \rightarrow \mu_{\text{steigend}}(\dot{X}) &= 0 \end{aligned}$$

Fuzzy-Beispiel: Fuzzyfizierung (2)

Unscharfe Menge für:



- z.B.: $Y = 6\text{ kW}$
 - $\mu_{sehrniedrig}(Y) = 0$
 - $\mu_{niedrig}(Y) = 0$
 - $\mu_{mittel}(Y) = 0,6$
 - $\mu_{hoch}(Y) = 0,4$
 - $\mu_{sehrhoch}(Y) = 0$

Als nächstes: Regelbasis aufstellen

- Durch Expertenbefragung
- Je nachdem, welchen Experten man fragt können die Regeln unterschiedlich ausfallen
- Erst ein Praxistest zeigt, ob die Regelbasis etwas taugt

Fuzzy-Beispiel: Regelbasis

Regel Nr.	Prämissen	Konklusion
1	WENN X niedrig UND \dot{X} fallend	DANN Y sehr hoch
2	WENN X niedrig UND \dot{X} gleich	DANN Y hoch
3	WENN X niedrig UND \dot{X} steigend	DANN Y mittel
4	WENN X mittel UND \dot{X} fallend	DANN Y mittel
5	WENN X mittel UND \dot{X} gleich	DANN Y mittel
6	WENN X mittel UND \dot{X} steigend	DANN Y mittel
7	WENN X hoch UND \dot{X} fallend	DANN Y mittel
8	WENN X hoch UND \dot{X} gleich	DANN Y niedrig
9	WENN X hoch UND \dot{X} steigend	DANN Y sehr niedrig

- Beispiel: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$

→ Aktive Regeln: 1,2,4,5

Fuzzy-Beispiel: Inferenz (1)

Für alle aktiven Regeln: Schnittmenge bilden

Beispiel wieder: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$

Regel Nr.	Prämissen	Konklusion
1	WENN X niedrig UND \dot{X} fallend	DANN Y sehr hoch
2	WENN X niedrig UND \dot{X} gleich	DANN Y hoch
4	WENN X mittel UND \dot{X} fallend	DANN Y mittel
5	WENN X mittel UND \dot{X} gleich	DANN Y mittel

Fuzzy-Beispiel: Inferenz (1)

Für alle aktiven Regeln: Schnittmenge bilden

Beispiel wieder: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$

Regel Nr.	Prämissen	Konklusion
------------------	------------------	-------------------

1	$MIN(0.2, 0.5) = 0.2$	DANN Y sehr hoch
---	-----------------------	--------------------

2	$MIN(0.2, 0.5) = 0.2$	DANN Y hoch
---	-----------------------	---------------

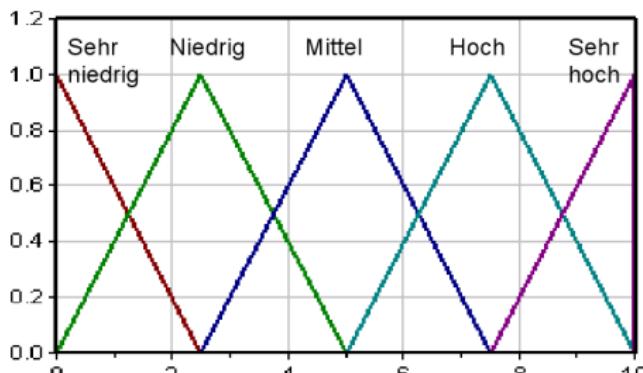
4	$MIN(0.8, 0.5) = 0.5$	DANN Y mittel
---	-----------------------	-----------------

5	$MIN(0.8, 0.5) = 0.5$	DANN Y mittel
---	-----------------------	-----------------

Fuzzy-Beispiel: Inferenz (2)

Durchführen der Inferenz

(Beispiel immer noch: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$)



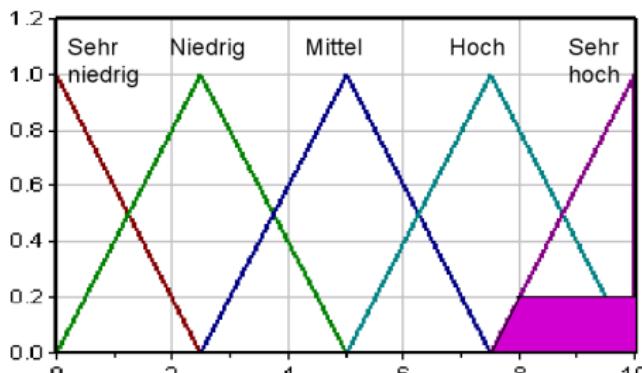
• z.B.: $Y = 6kW$

- $\mu_{sehrhoch}(Y) = 0,2$
- $\mu_{hoch}(Y) = 0,2$
- $\mu_{mittel}(Y) = 0,5$
- $\mu_{mittel}(Y) = 0,5$

Fuzzy-Beispiel: Inferenz (2)

Durchführen der Inferenz

(Beispiel immer noch: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$)

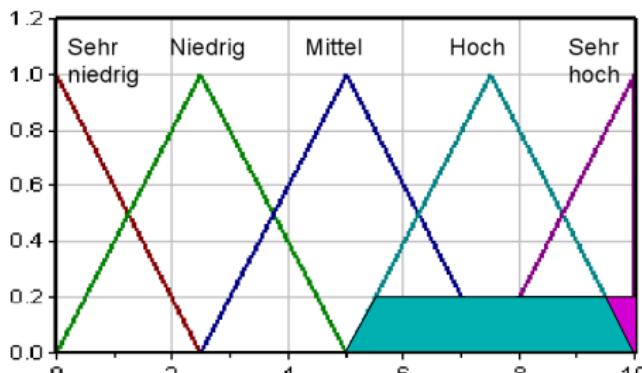


- z.B.: $Y = 6kW$
 - $\mu_{sehrhoch}(Y) = 0,2$
 - $\mu_{hoch}(Y) = 0,2$
 - $\mu_{mittel}(Y) = 0,5$
 - $\mu_{mittel}(Y) = 0,5$

Fuzzy-Beispiel: Inferenz (2)

Durchführen der Inferenz

(Beispiel immer noch: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$)

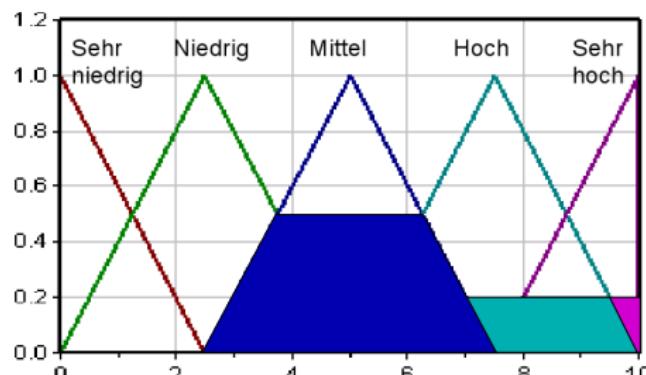


- z.B.: $Y = 6kW$
 - $\mu_{sehrhoch}(Y) = 0,2$
 - $\mu_{hoch}(Y) = 0,2$
 - $\mu_{mittel}(Y) = 0,5$
 - $\mu_{mittel}(Y) = 0,5$

Fuzzy-Beispiel: Inferenz (2)

Durchführen der Inferenz

(Beispiel immer noch: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$)

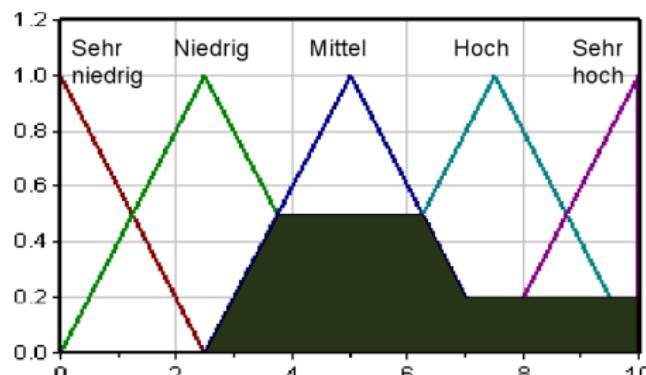


- z.B.: $Y = 6kW$
 - $\mu_{sehrhoch}(Y) = 0,2$
 - $\mu_{hoch}(Y) = 0,2$
 - $\mu_{mittel}(Y) = 0,5$
 - $\mu_{mittel}(Y) = 0,5$

Fuzzy-Beispiel: Inferenz (2)

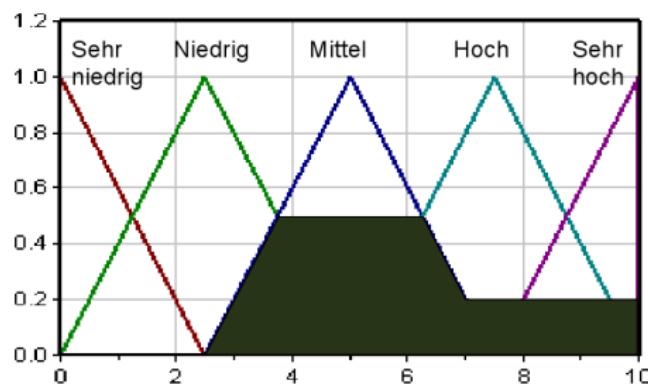
Durchführen der Inferenz

(Beispiel immer noch: $X = 180^\circ C$, $\dot{X} = -5 \frac{^\circ C}{s}$)



- z.B.: $Y = 6kW$
 - $\rightarrow \mu_{sehrhoch}(Y) = 0,2$
 - $\rightarrow \mu_{hoch}(Y) = 0,2$
 - $\rightarrow \mu_{mittel}(Y) = 0,5$
 - $\rightarrow \mu_{mittel}(Y) = 0,5$

Fuzzy-Beispiel: Defuzzifizierung



- Hier: $Y_{akt} \approx 5,75$

- Flächenschwerpunkt nach:

$$Y_{akt} = \frac{\int_a^b Y \cdot \mu_{res}(Y) dY}{\int_a^b \mu_{res}(Y) dY}$$

Fuzzy-Beispiel: Vergleich mit PID-Regler

