Echtzeitsysteme
Lehrstuhl Informatik VI - Robotics and Embedded Systems

Echtzeitsysteme
Wintersemester 2010/2011

Prof. Dr. Alois Knoll, Dr. Christian Buckl

TU München

Lehrstuhl VI Robotics and Embedded Systems

Echtzeitsysteme: Organisation

Team

Prof. Dr. Alois Knoll

Dr. Christian Buckl

Übungen: Simon Barner, Dominik Sojer, Stephan Sommer

Homepage der Vorlesung mit Folien, Übungsaufgaben und weiterem Material:
http://www6.informatik.tu-muenchen.de/Main/TeachingWs2010Echtzeitsysteme

Bestandteile der Vorlesung

- Vorlesung:
  - Dienstag 10:15-11:45 Uhr MI HS 2
  - Mittwoch 12:15-13:00 Uhr MI HS 2
  - 6 ECTS Punkte
  - Wahlpflichtvorlesung im Gebiet Echtzeitsysteme (Technische Informatik)
  - Wahlpflichtvorlesung für Studenten der Elektro- und Informationstechnik
  - Pflichtvorlesung für Studenten des Maschinenbau Richtung Mechatronik
- Übung:
  - zweistündige Tutorübung, im Raum 03.05.012
  - Montags 14:00 – 15:30 Uhr
  - Dienstags 8:30 – 10:00 Uhr
  - Mittwochs 10:15 – 11:45 Uhr und 14:30 – 15:30 Uhr
  - Weitere Termine bei Bedarf nach Vereinbarung
  - Beginn: voraussichtlich ab 08.11.2010, Anmeldung ab sofort über TUMonline
- Prüfung:
  - Schriftliche Klausur am Ende des Wintersemesters, falls ein Schein benötigt wird.

Informationen zur Übung

- Ziel: Praktisches Einüben von Vorlesungsinhalten
- Übungsaufgaben werden in Gruppen zu zweit am Computer gelöst
- Platz ist begrenzt (8 Computer ⇔ 16 Studenten) ⇒ **Anmeldung erforderlich**
- Übungsaufgaben sind auch auf der Vorlesungsseite verfügbar
- Es werden diverse Aufgaben aus dem Bereich der Echtzeitprogrammierung angeboten, wie z.B. Aufgaben zu Threads, Semaphore, Kommunikation
- Programmiersprache ist überwiegend C, zu Beginn der Übung wird eine kurze Einführung in C angeboten
- Die Anmeldung erfolgt über **TUMonline** (Tutorübungen zu Echtzeitsysteme (IN2060))
- Falls Bedarf an weiteren Terminen besteht, senden Sie bitte eine Mail an sommerst@in.tum.de.
- Die Übungsinhalte sind nicht direkt prüfungsrelevant, **tragen aber stark zum Verständnis bei.**

Mögliche Übungsinhalte

- Modellierungssprachen/-werkzeuge (Esterel Studio, SCADE, Ptolemy, EasyLab, FTOS)
- Programmierung von Echtzeitsystemen (Programmiersprache C, Betriebssysteme VxWorks, PikeOS)
  - Nebenläufigkeit (Threads, Prozesse)
  - Interprozesskommunikation / Synchronisation: Semaphore, Signale, Nachrichtenwarteschlangen
  - Unterbrechungsbehandlung
- Kommunikation (Ethernet, CAN)
- Programmierung von Adhoc-Sensornetzwerken (TinyOS, ZigBee)
- Umsetzung von diversen Demonstratoren (Aufzug, Kugelfall, Murmelbahn)

- **Ihre Rückmeldung ist wichtig, denn sie bestimmt über die Inhalte!**

Klausur

- Für Studenten, die einen Schein benötigen, wird am Ende der Vorlesung eine schriftliche Klausur angeboten.
- Stoff der Klausur sind die Inhalte der Vorlesung.
- Die Inhalte der Übung sind nicht direkt prüfungsrelevant, tragen allerdings zum Verständnis des Prüfungsstoffes bei.
- Voraussichtlicher Termin: letzte Vorlesungswoche (Rückmeldung mit Prüfungsamt steht noch aus)
- Voraussichtlich erlaubte Hilfsmittel: keine

Grundsätzliches Konzept

- Themen werden aus verschiedenen Blickrichtungen beleuchtet:
  - Stand der Technik in der Industrie
  - Stand der Technik in der Wissenschaft
  - Existierende Werkzeuge
  - Wichtig: nicht die detaillierte Umsetzung, sondern die Konzepte sollen verstanden werden
- Zur Verdeutlichung theoretischer Inhalte wird versucht, Analogien zum Alltag herzustellen. Wichtig: Praktische Aufgaben in der Vorlesung und der Übung
- In jedem Kapitel werden die relevanten Literaturhinweise referenziert
- Zur Erfolgskontrolle werden Klausuraufgaben der letzten Jahre am Ende eines Kapitels diskutiert
- **Wir freuen uns jederzeit über Fragen, Verbesserungsvorschläge und konstruktive Kommentare!**

Weitere Angebote des Lehrstuhls

- Weitere Vorlesungen: Robotik, Digitale Signalverarbeitung, Maschinelles Lernen und bioinspirierte Optimierung I&II, Sensor- und kamerageführte Roboter
- Praktika: Echtzeitsysteme, Roboterfußball, Industrieroboter, Neuronale Netze und Maschinelles Lernen, Bildverarbeitung, Signalverarbeitung
- Seminare: Sensornetzwerke, Modellierungswerkzeuge, Busprotokolle, Objekterkennung und Lernen, Neurocomputing,
- Diplomarbeiten / Masterarbeiten
- Systementwicklungsprojekte / Bachelorarbeiten
- Guided Research, Stud. Hilfskräfte
- Unser gesamtes Angebot finden Sie unter http://wwwknoll.in.tum.de

Leitprojekte des Lehrstuhls/fortiss im Bereich ES

- Am Lehrstuhl werden eine große Zahl von Projekten im Bereich embedded systems bzw. cyber-physical systems durchgeführt
- Für die Hörer dieser Vorlesung bieten sich zum Einstieg zwei Projekte besonders an: die Leitprojekte "**E-Fahrzeug 2.0**" und "**InnoTruck**"
- "E-Fahrzeug 2.0" ist ein Projekt des **Transferinstituts fortiss** eGmbH (www.fortiss.org)
- Der "InnoTruck" entsteht im Projekt "Diesel Reloaded" des Rudolf-Diesel-Fellows Prof. G. Spiegelberg **des TUM Institute for Advanced Studies** (http://www.ias.tum.de/)

E-Fahrzeug 2.0

- Entwicklung eines Versuchsträgers für innova
  eines rein elektrischen Fahrzeugs
- Basisdaten:
  - Vier Radnabenmotoren in “e-corners”, Lenkw
  - Keine Bremsen, rein elektrische Verzögerung
  - Sidestick-Steuerung
  - Kommunikation basierend auf einer Echtzeit-

Petronius Electric Car Software Architecture

- The central concept of *Petronius* is **data flowing** from the sensors of the vehicle to its actuators, based on events and/or time predicates → *data-centric architecture with clear data structuring*
- **Avoid replication** of data and its acquisition → centralised logical data base or "*data cloud*" for decoupling modules
- **Modular design** for easy extensibility, testability, independent yet safe development → provision for specifying *abstract data dependencies*
- Simple **network structure** → logical or *virtual networking*
- **Scalability and portability** across vehicle classes → *flexible mapping* from logical architecture to target hardware through suitable tools
- **Fault tolerance** is mandatory → *fault recognition and redundancy management*

Experimental Platform

Key Facts (per wheel):

- Drive motor: 2…8 kW
- *RPM: 520 → 48 km/h*
- *Maximum torque: 160 Nm*
- *No Brakes*
- *X-by-Wire*
- *Sidestick control*

Studentenarbeiten im Rahmen des eCar-Projektes (1)

- **Verbesserung eines internen Kommunikationsnetzes (HIWI)**
  Im Rahmen dieser Arbeit soll ein auf Echtzeit-Ethernet basierendes Kommunikationsnetz weiterentwickelt werden.
- **Integration eines Winkelsensors zur Messung des aktuellen Radwinkels (SEP, Bachelor, HIWI)**
  Es soll mit Hilfe eines Winkelsensors die absolute Radposition ermittelt werden.
- **Erweiterung der Mensch-Maschine Schnittstelle (HIWI)**
  Ziel ist es die MMI zu erweitern und besser auf die Bedürfnisse des Fahrers abzustimmen.
- **Implementierung einer Fahrdynamikregelung (SEP, Bachelor, HIWI)**
  Die Aufgabe besteht in der Portierung einer Matlab/Simulink Regelung auf das eCar.

Studentenarbeiten im Rahmen des eCar-Projektes (2)

- **Entwicklung eines modellgetriebenen Tools für die Funktionsentwicklung des eCar (SEP, Bachelor, Master, Diplom, HIWI)**
  Um die Funktionsentwicklung für das eCar zu erleichtern, soll ein modellgetriebenes Tool entwickelt werden, das das Hinzufügen von weiteren Funktionen unterstützt.
- **Ermittlung des aktuellen Verbrauchs (SEP, Bachelor, HIWI)**
  Die Aufgabe besteht in bedarfsgerechter Ermittlung der Energiemenge. Diese Information soll in einem ersten Schritt dem Fahrer zur Verfügung gestellt werden.
- **Diverse weitere Themen**
  Außerdem gibt es noch unzählige weitere Themen die im Rahmen des eCar-Projektes bearbeitet werden können.

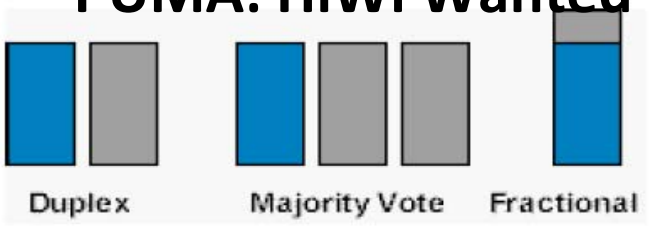
Kontakt:

ecar@fortiss.org

Nähere Informationen:

http://www.fortiss.org/en/research/cyber-physical-systems/projects/ecar.html

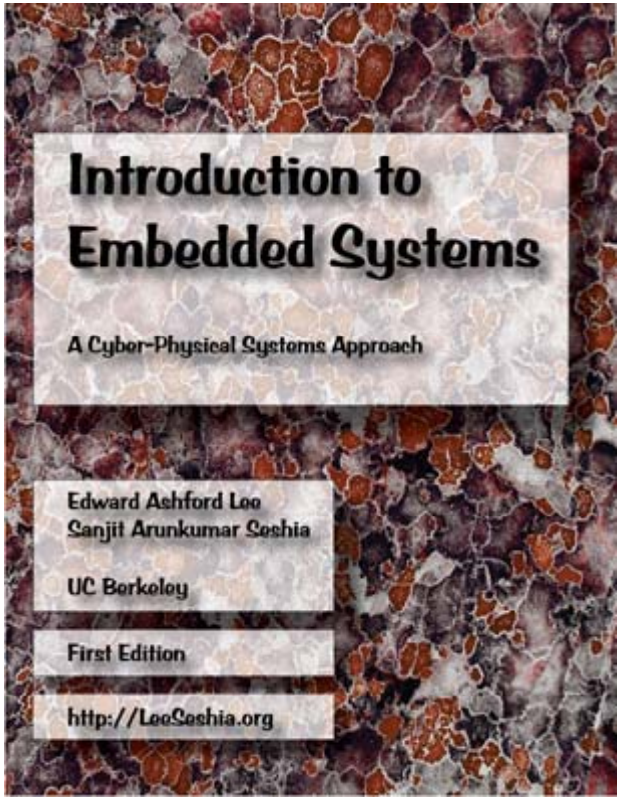
PUMA: HiWi Wanted



- The PUMA doctoral school is looking for long-term (> 4 months) student assistants to work with us on the implementation (extension) of **Gecko**, a novel framework combining game solving and fault-tolerant embedded systems.
- Requirements (applicants should fulfill one of them):
  - 1. Knowledge in verification or automata theory
    - You can work on extensions in model translation or engines
  - 2. Knowledge in embedded/real-time systems
    - You can work on extensions which establish links between concrete examples (production line system based on Festo MPS) and our tool
- For further information (job packages), please contact
- Chih-Hong Cheng (chengch@in.tum.de).

PUMA Graduiertenkolleg

Literatur



- Lee, Seshia: Introduction to Embedded Systems
  - Buch deckt die meisten Kapitel der Vorlesung (bis auf Kommunikation) ab
  - Kostenlos online verfügbar unter http://leeseshia.org/
  - Vorlesung wird in Zukunft an dieses Buch angepasst (Buch ist gerade erst erschienen)

Weitere Literatur

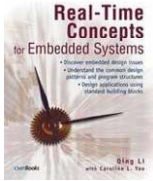
Weitere Literaturangaben befinden sich in den jeweiligen Abschnitten.

Hermann Kopetz: Real-Time Systems (Überblick)

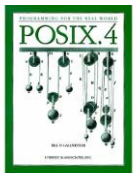
Jane W. S. Liu: Real-Time Systems
(Überblick, Schwerpunkt Scheduling)

Stuart Bennet: Real-Time Computer Control:
An Introduction (Überblick, Hardware)

Alan Burns, Andy Wellings: Real-Time Systems and Programming
Languages (Schwerpunkt: Programmiersprachen)



Qing Li, Caroline Yao: Real-Time Concepts for
Embedded Systems (Schwerpunkt: Programmierung)



Bill O. Gallmeister: Programming for the Real-World: POSIX.4
(Schwerpunkt: Posix)

Vorlesungsinhalte

1. Einführung Echtzeitsysteme
2. Modellierung und Werkzeuge
3. Nebenläufigkeit
4. Scheduling
5. Echtzeitbetriebssysteme
6. Programmiersprachen
7. Uhren
8. Kommunikation
9. Fehlertolerante Systeme
10. Spezielle Hardware
11. Regelungstechnik

Weitere Themen können bei Interesse aufgenommen werden. Melden Sie sich einfach nach der Vorlesung oder per Email.

Inhalt I

- Kapitel Einführung (ca. 1 Vorlesungswoche)
  - Definition Echtzeitsysteme
  - Klassifikation
  - Echtzeitsysteme im täglichen Einsatz
  - Beispielanwendungen am Lehrstuhl
- Kapitel Modellierung/Werkzeuge (ca. 3 Vorlesungswochen)
  - Allgemeine Einführung
  - Grundsätzlicher Aufbau, Models of Computation, Ptolemy
  - Synchrone Sprachen (Esterel, Lustre), SCADE, EasyLab
  - Zeitgesteuerte Systeme: Giotto, FTOS, TTA
  - Exkurs: Formale Methoden

Inhalt II

- Kapitel Nebenläufigkeit (2 Vorlesungswochen)
  - Prozesse, Threads
  - Interprozesskommunikation
- Kapitel Scheduling (2 Vorlesungswochen)
  - Kriterien
  - Planung Einrechner-System, Mehrrechnersysteme
  - EDF, Least Slack Time
  - Scheduling mit Prioritäten (FIFO, Round Robin)
  - Scheduling periodischer Prozesse
  - Scheduling Probleme

Inhalt III

- Kapitel Echtzeitbetriebssysteme (1 Vorlesungswoche)
  - QNX, VxWorks, PikeOS
  - RTLinux, RTAI, Linux Kernel 2.6
  - TinyOS, eCos
  - OSEK
- Kapitel Programmiersprachen (1 Vorlesungswoche)
  - Ada
  - Erlang
  - C mit POSIX.4
  - Real-time Java
- Kapitel Uhren (1 Vorlesungswoche)
  - Uhren
  - Synchronisation von verteilten Uhren

Inhalt IV

- Kapitel Echtzeitfähige Kommunikation (1 Vorlesungswoche)
  - Token-Ring
  - CAN-Bus
  - TTP, FlexRay
  - Real-Time Ethernet
- Kapitel Fehlertoleranz (ca. 2 Vorlesungswochen)
  - Bekannte Softwarefehler
  - Definitionen
  - Fehlerarten
  - Fehlerhypothesen
  - Fehlervermeidung
  - Fehlertoleranzmechanismen

Potentielle zusätzliche Inhalte

- Kapitel: Spezielle Hardware (1 Vorlesungswoche)
  - Digital-Analog-Converter (DAC)
  - Analog-Digital-Converter (ADC)
  - Speicherprogrammierbare Steuerung (SPS)
- Kapitel: Regelungstechnik (ca. 2 Vorlesungswochen)
  - Definitionen
  - P-Regler
  - PI-Regler
  - PID-Regler
  - Fuzzy-Logic

Kapitel 1

Einführung Echtzeitsysteme

Inhalt

- Definition Echtzeitsysteme
- Klassifikation von Echtzeitsystemen
- Echtzeitsysteme im täglichen Leben
- Beispielanwendungen am Lehrstuhl

Definition Echtzeitsystem

Ein Echtzeit-Computersystem ist ein Computersystem, in dem die Korrektheit des Systems nicht nur vom logischen Ergebnis der Berechnung abhängt, sondern auch vom physikalischen Moment, in dem das Ergebnis produziert wird.

Ein Echtzeit-Computer-System ist immer nur ein Teil eines größeren Systems, dieses größere System wird Echtzeit-System genannt.

Hermann Kopetz

TU Wien

Definition Eingebettetes System

Technisches System, das durch ein integriertes, von Software gesteuertes Rechensystem gesteuert wird. Das Rechensystem selbst ist meist nicht sichtbar und kann in der Regel nicht frei programmiert werden. Um die Steuerung zu ermöglichen ist zumeist eine Vielzahl von sehr speziellen Schnittstellen notwendig.

In der Regel werden leistungsärmere Mikroprozessoren mit starken Einschränkung in Bezug auf die Rechenleistung und Speicherfähigkeit eingesetzt.

Resultierende Eigenschaften

⇒ zeitliche Anforderungen

- Zeitliche Genauigkeit (nicht zu früh, nicht zu spät)
- Garantierte Antwortzeiten
- Synchronisation von Ereignissen / Daten
- **Aber nicht:** Allgemeine Geschwindigkeit

⇒ Eigenschaften aufgrund der Einbettung

- Echtzeitsysteme sind typischerweise sehr Eingabe/Ausgabe (E/A)-lastig
- Echtzeitsysteme müssen fehlertolerant sein, da sie die Umgebung beeinflussen
- Echtzeitsysteme sind häufig verteilt

Zeitlicher Determinismus vs. Leistung

- Konsequenz der Forderung nach deterministischer Ausführungszeit: Mechanismen, die die allgemeine Performance steigern, aber einen negativen, nicht exakt vorhersehbaren Effekt auf einzelne Prozesse haben können, werden in der Regel nicht verwendet:
  - Virtual Memory
  - Garbage Collection
  - Asynchrone IO-Zugriffe
  - rekursive Funktionsaufrufe

Klassifikation von Echtzeitsystemen

- Echtzeitsysteme können in verschiedene Klassen unterteilt werden:
  - Nach den Konsequenzen bei der Überschreitung von Fristen: harte vs. weiche Echtzeitsysteme
  - Nach dem Ausführungsmodell: zeitgesteuert (zyklisch, periodisch) vs. ereignisbasiert (aperiodisch)

Harte bzw. weiche Echtzeitsysteme

- **Weiche Echtzeitsysteme:**
  Die Berechnungen haben eine zeitliche Ausführungsfrist, eine Überschreitung dieser Fristen hat jedoch keine katastrophale Folgen. Eventuell können die Ergebnisse nc werden, insgesamt kommt es durch die Fristverletzung evtl. zu einer Diens
  Beispiel für ein weiches Echtzeitsystem: Video
  Konsequenz von Fristverletzungen: einzelne Videoframes gehen verloren,

- **Harte Echtzeitsysteme:**
  Eine Verletzung der Berechnungsfristen kann sofort zu fatalen Folgen (hohe Sachschäden oder sogar Gefährdung von Menschenleben) führen. Die Einhaltung der Fristen ist absolut notwendig.
  Beispiel für ein hartes Echtzeitsystem: Raketensteuerung
  Konsequenz von Fristverletzung: Absturz bzw. Selbstzerstörung der Rakete

Unterteilung nach Ausführungsmodell

- Zeitgesteuerte Applikationen:
  - Der gesamte zeitliche Systemablauf wird zur Übersetzungszeit festgelegt
  - Notwendigkeit einer präzisen, globalen Uhr $\Rightarrow$ Uhrensynchronisation notwendig
  - Für die einzelnen Berechnungen ist jeweils ein Zeitslot reserviert $\Rightarrow$ Abschätzung der maximalen Laufzeiten (**worst case execution times - WCET**) notwendig
  - **Vorteil**: Statisches Scheduling möglich und damit ein vorhersagbares (**deterministisches**) Verhalten
- Ereignisgesteuerte Applikationen:
  - Alle Ausführungen werden durch das Eintreten von Ereignissen angestoßen
  - Wichtig sind bei ereignisgesteuerten Anwendungen garantierte Antwortzeiten
  - Das Scheduling erfolgt dynamisch, da zur Übersetzungszeit keine Aussage über den zeitlichen Ablauf getroffen werden kann.

Einführung Echtzeitsysteme

Echtzeitsysteme im Alltag

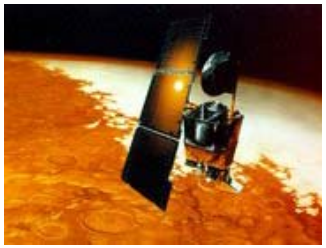
Echtzeitsysteme sind allgegenwärtig!











Beispiel: Kuka Robocoaster




http://www.robocoaster.com

Einleitung Echtzeitsysteme

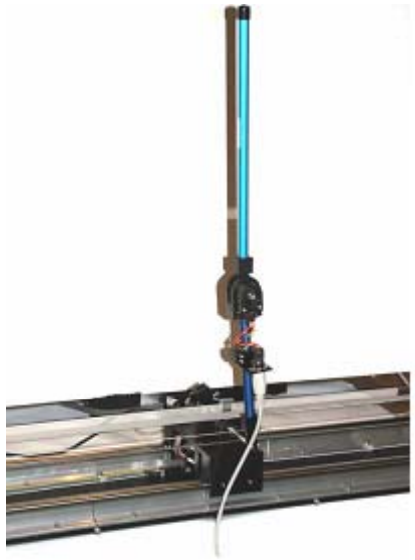
Anwendungen am Lehrstuhl / fortiss

Steuerungsaufgaben (Praktika+Studienarbeiten)

Regelungsaufgaben (Praktika+Studienarbeiten)

Schwebender Stab

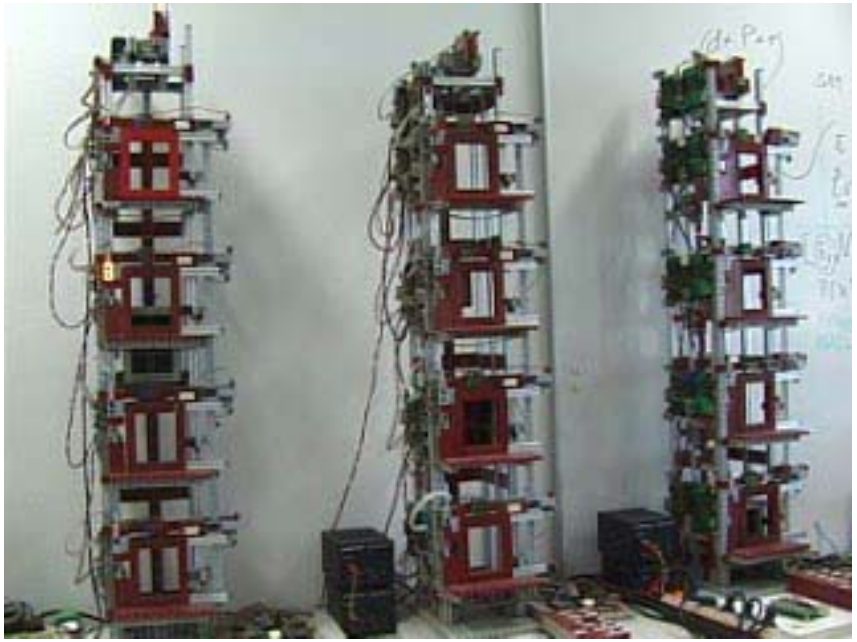
Produktionstechnik



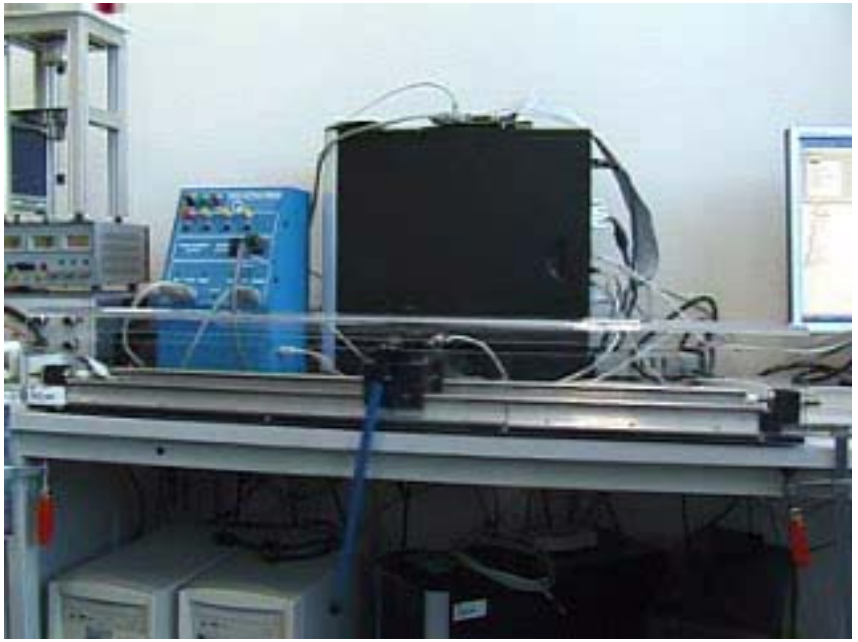
Invertiertes Pendel

Elektroauto

Videos



Software entwickelt mit FTOS

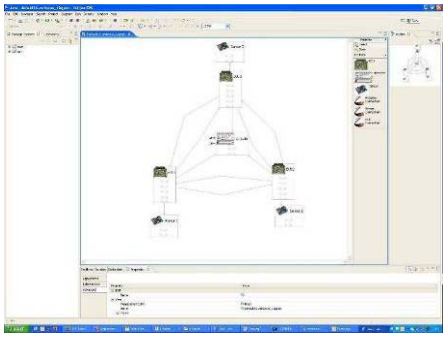


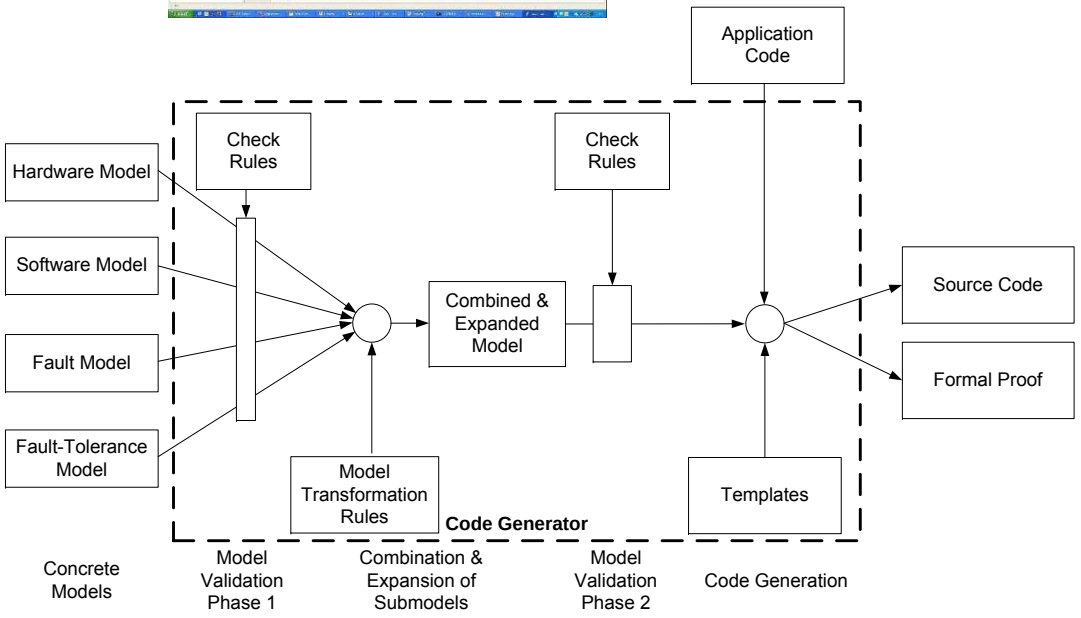
Software entwickelt mit EasyLab

FTOS: Modellbasierte Entwicklung fehlertoleranter Echtzeitsysteme

Ziele:

- Umfangreiche Generierung von Code auf Systemebene:
  - Fehlertoleranzmechanismen
  - Prozessmanagement, Scheduling
  - Kommunikation
- Erweiterbarkeit der Codegenerierung durch Verwendung eines vorlagen-basierten Codegenerators
- Zertifizierung des Codegenerators

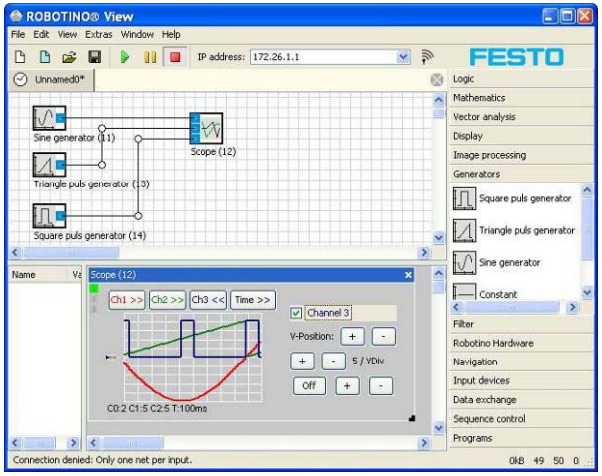




GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Modellbasierte Software-Entwicklung für mechatronische Systeme

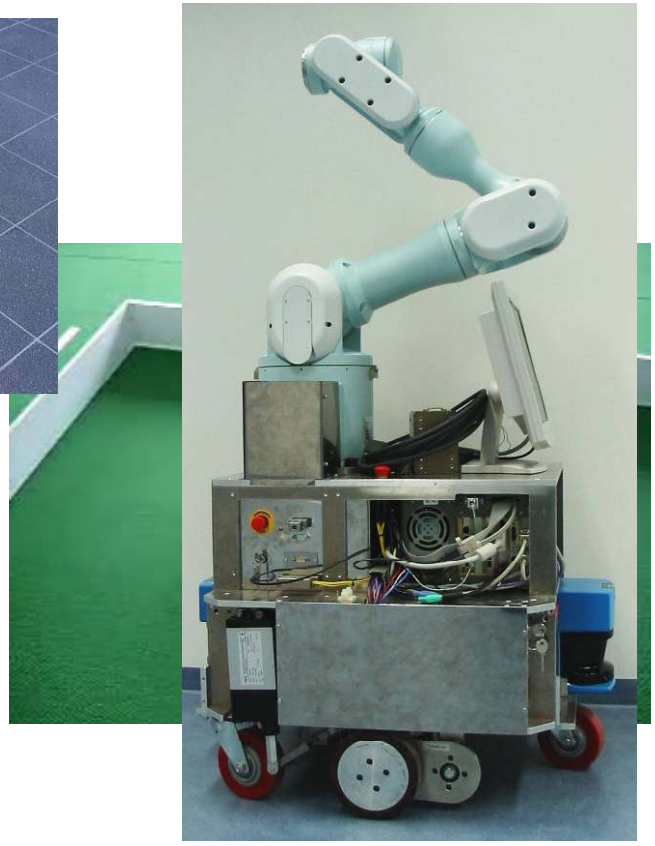
- Entwicklung von komponentenbasierten Architekturen für mechatronische Systeme (Mechanik, Elektronik, Software)
- Ziele:
  - Reduzierung der Entwicklungszeiten
  - Vereinfachung des Entwicklungsprozesses
- Komponenten:
  - Hardwaremodule
  - Softwaremodule
  - Werkzeugkette:
    - Codegenerierung
    - Graphische Benutzerschnittstelle
    - Debugging-Werkzeug



GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

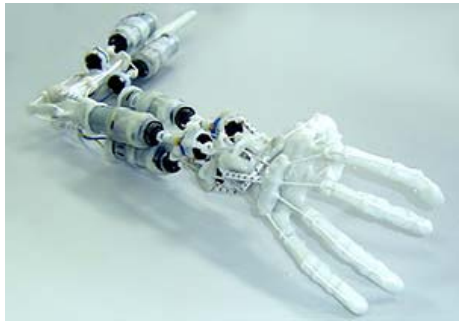
Robotersteuerung

Robotino



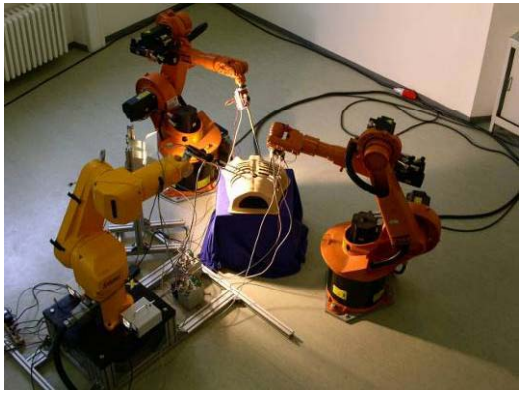
Leonardo

Stäubli



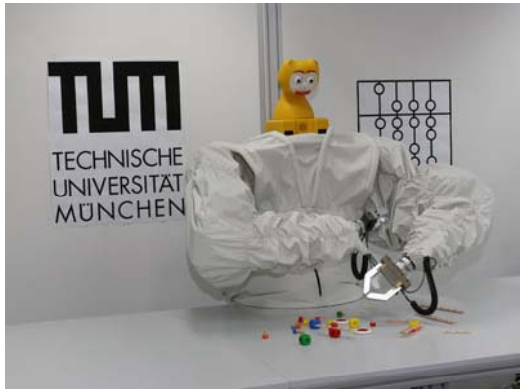
Tumanoid

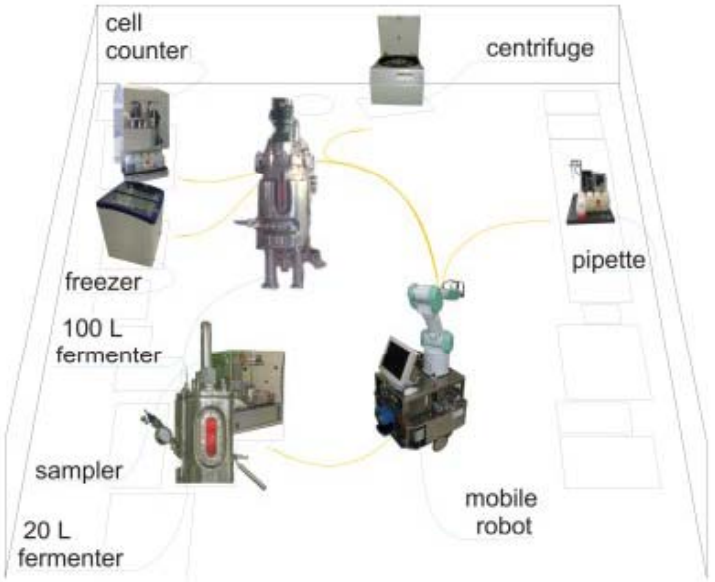
Anwendungen der Robotik



Telemedizin

Jast





Automatisiertes biotechnisches Labor

Automatisiertes Labor

Kapitel 2

Modellierung von Echtzeitsystemen und Werkzeuge

Inhalt

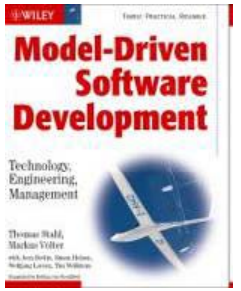
- Motivation
- Grundsätzlicher Aufbau, „Modelle“ und „Models of Computation“
  - Werkzeug Ptolemy
- Synchrone Sprachen (Esterel, Lustre)
  - Reaktive Systeme: Werkzeuge Esterel Studio, SCADE
  - Synchroner Datenfluss: EasyLab
- Zeitgesteuerte Systeme
  - Werkzeug Giotto
- Domänenspezifische Codegeneratoren
  - Werkzeug FTOS
- Verifikation durch den Einsatz formaler Methoden

Fokus dieses Kapitels

- Konzepte und Werkzeuge zur Modellierung **und** Generierung von Code für Echtzeit- und eingebettete Systeme
- Voraussetzungen an Werkzeuge für ganzheitlichen Ansatz:
  - Explizite Modellierung des zeitlichen Verhaltens (z.B. Fristen)
  - Modellierung von Hardware und Software
  - Eindeutige Semantik der Modelle
  - Berücksichtigung von nicht-funktionalen Aspekten (z.B. Zeit*, Zuverlässigkeit, Sicherheit)
- Ansatz zur Realisierung:
  - Schaffung von domänenspezifischen Werkzeugen (Matlab/Simulink, Labview, SCADE werden überwiegend von spezifischen Entwicklergruppen benutzt)
  - Einfache Erweiterbarkeit der Codegeneratoren oder Verwendung von virtuellen Maschinen / Middleware-Ansätzen

* Zeit wird zumeist als nicht-funktionale Eigenschaft betrachtet, in Echtzeitsystemen ist Zeit jedoch als funktionale Eigenschaft anzusehen (siehe z.B. Edward Lee: Time is a Resource, and Other Stories, May 2008) http://chess.eecs.berkeley.edu/pubs/426/Lee_TimeIsNotAResource.pdf

Literatur



- Sastry et al: Scanning the issue – special issue on modeling and design of embedded software, Proceedings of the IEEE, vol.91, no.1, pp. 3-10, Jan 2003
- Thomas Stahl, Markus Völter: Model-Driven Software Development, Wiley, 2006
- Ptolemy: Software und Dokumentation http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm
- Benveniste et al.: The Synchronous Languages, 12 Years Later, Proceedings of the IEEE, vol.91, no.1, pp. 64-83, Jan 2003
- Diverse Texte zu Esterel, Lustre, Safe State Machines: http://www.esterel-technologies.com/technology/scientific-papers/
- David Harrel, Statecharts: A Visual Formalism For Complex Systems, 1987
- Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003

Hinweis: Veröffentlichungen von IEEE, Springer, ACM können Sie kostenfrei herunterladen, wenn Sie den Proxy der TUM Informatik benutzen (proxy.in.tum.de)

Begriff: Modell

- Brockhaus:
  Ein **Abbild** der Natur unter der Hervorhebung für **wesentlich** erachteter **Eigenschaften** und Außerachtlassen als nebensächlich angesehener Aspekte. Ein M. in diesem Sinne ist ein Mittel zur Beschreibung der erfahrenen Realität, zur Bildung von Begriffen der Wirklichkeit und Grundlage von Voraussagen über künftiges Verhalten des erfassten Erfahrungsbereichs. Es ist um so realistischer oder wirklichkeitsnäher, je konsistenter es den von ihm umfassten Erfahrungsbereich zu deuten gestattet und je genauer seine Vorhersagen zutreffen; es ist um so **mächtiger**, je **größer** der von ihm beschriebene **Erfahrungsbereich** ist.
- Wikipedia:
  Von einem **Modell** spricht man oftmals als Gegenstand wissenschaftlicher Methodik und meint damit, dass eine zu untersuchende Realität durch bestimmte Erklärungsgrößen im Rahmen einer wissenschaftlich handhabbaren Theorie abgebildet wird.

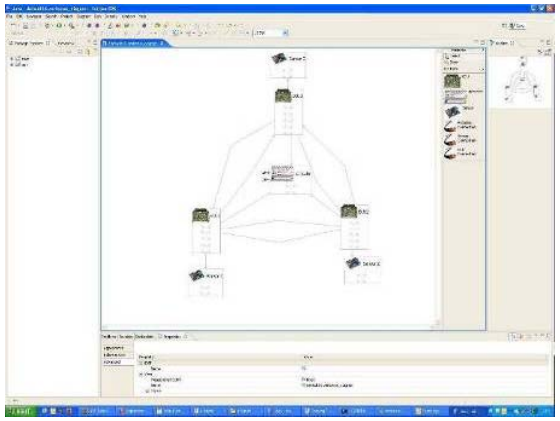
Modellbasierte Entwicklung

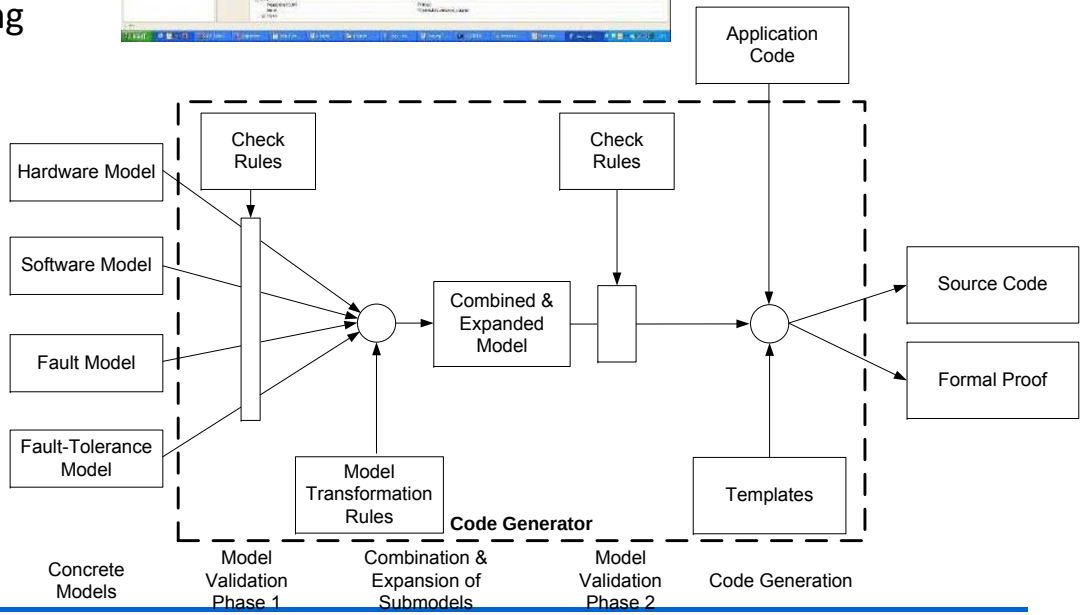
- Wir beobachten im Bereich eingebettete Systeme seit längerem einen Übergang von der klassischen Programmentwicklung zur einer Vorgehensweise, bei der Modelle (für physi-kalische Prozesse, für die Hardware eines Systems, für Verhalten eines Kommunikations-netzes, usw.) eine zentrale Rolle spielen
- Modelle werden typischerweise durch verknüpfte grafische Notationselemente dargestellt (bzw. definiert)
- **Funktions-Modelle** sind dabei (meist Rechner-ausführbare) Beschreibungen der zu reali-sierenden Algorithmen
- In unterschiedlichen Phasen der Systementwicklung kann ein Modell unterschiedliche Rollen annehmen – und etwa zum Funktionsdesign, zur Simulation, zur Codegenerierung verwendet werden
- Im günstigsten Fall kann ein System grafisch notiert („zusammengeklickt“) werden und unmittelbar Code erzeugt werden, der dann auf einem Zielsystem zur Ausführung gebracht wird (Beispiel: EasyKit)
- Diese Entwicklung steht im Einklang mit der generellen Geschichte der Informatik, die immer mächtigere (Programmier-) Werkzeuge auf immer abstrakterem Niveau geschaffen hat

Beispiel FTOS: Modellbasierte Entwicklung fehlertoleranter Echtzeitsysteme

Ziele:

- Umfangreiche Generierung von Code auf Systemebene:
  - Fehlertoleranzmechanismen
  - Prozessmanagement, Scheduling
  - Kommunikation
- Erweiterbarkeit der Codegenerierung durch Verwendung eines vorlagen-basierten Codegenerators
- Zertifizierung des Codegenerators





GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Vorteile Modellbasierter Entwicklung

- Für die modellbasierte Entwicklung sprechen diverse Gründe:
  - Modelle sind häufig einfacher zu verstehen als der Programmcode (graphische Darstellung, Erhöhung des Abstraktionslevels)
  - Vorwissen ist zum Verständnis der Modelle häufig nicht notwendig:
    - Experten unterschiedlicher Disziplinen können sich verständigen
  - Systeme können vorab simuliert werden. Hierdurch können Designentscheidungen vorab evaluiert werden und späte Systemänderungen minimiert werden.
  - Es existieren Werkzeuge um Code automatisch aus Modellen zu generieren:
    - Programmierung wird stark erleichtert
    - Ziel: umfassende Codegenerierung (Entwicklung konzentriert sich ausschließlich auf Modelle)
  - Mittels formaler Methoden kann
    - die Umsetzung der Modelle in Code getestet werden
    - das Modell auf gewisse Eigenschaften hin überprüft werden

Modellbasierte Ansätze sind erfolgreich …

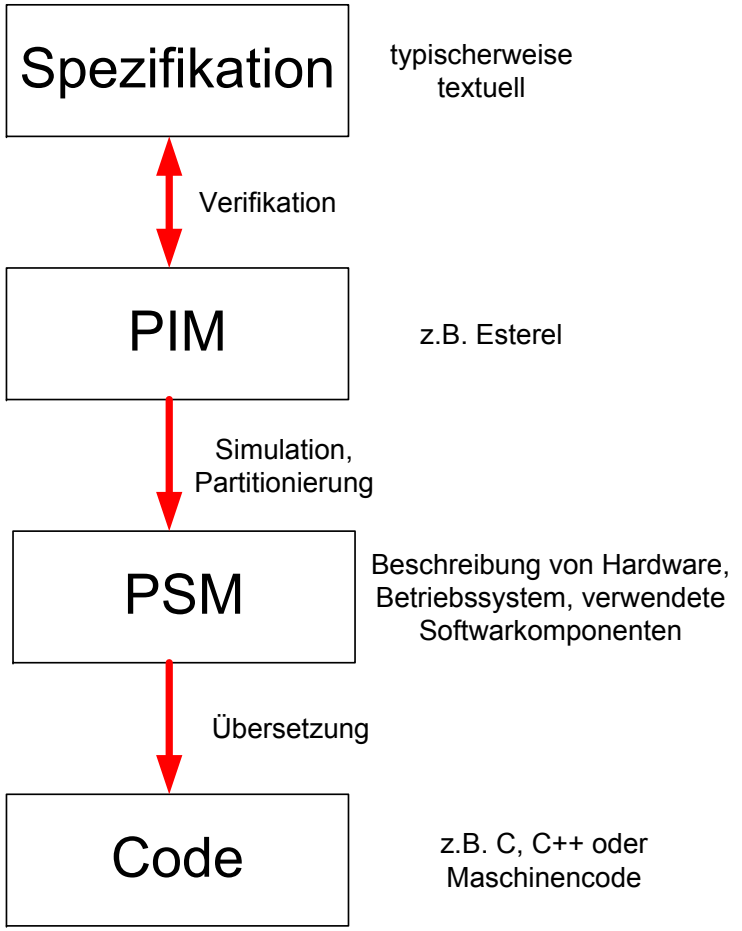
… sie haben unter anderem zu folgendem beigetragen:

- Ablösen des klassischen „Komponenten-Kopie-Ansatzes“ durch das Prinzips der **Klasse-Instanz-Beziehung**
- Definition von **domänenspezifische Architekturmodellen**: Schichtenmodelle, Bushierarchien
- Konstruktion von **hybriden Umgebungsmodellen**: Abstraktere und erweiterte Modelle der Umgebung
- Definition von Entwurfsebenen: Funktionsarchitektur, logische Architektur, technische Architektur (analog zur Strukturierung der Hardware-Entwicklungsschritte: Architektur – Implementierung – Realisierung, (Gene Amdahl, IBM, 1964), siehe VL-homepage)
- Vermittlung von Modellen und Techniken zur Unterstützung der Anforderungsanalyse und -definition
- Vermittlung von Modellen und Techniken zur Beschreibung von Varianten und Produktlinien

(siehe http://www4.informatik.tu-muenchen.de/~schaetz/MBEES/index.html)

Beispiel OMG: Model-Driven Architecture (MDA)

- Die Entwicklung des Systems erfolgt in diversen Schritten:
  - textuelle Spezifikation
  - PIM: platform independent model
  - PSM: platform specific model
  - Code: Maschinencode bzw. Quellcode
- Aus der Spezifikation erstellt der Entwickler das plattformunabhängige Modell
- Hoffung: weitgehende Automatisierung der Transformationen PIM → PSM → Code (Entwickler muss nur noch notwendige Informationen in Bezug auf die Plattform geben)
- http://www.omg.org/mda



MDA im Kontext von Echtzeitsystemen

- In Echtzeitsystemen / eingebetteten Systemen ist bei einem umfassenden Ansatz ein Hardwaremodell (z.B. Rechner im verteilten Systen, Topologie) schon in frühen Phasen (PIM) notwendig
- Das plattformspezifische Modell (PSM) erweitert das Hardware- & Softwaremodell um Implementierungskonzepte, z.B.
  - Implementierung als Funktion/Thread/Prozess
  - Prozesssynchronisation

Anwendungsbeispiele

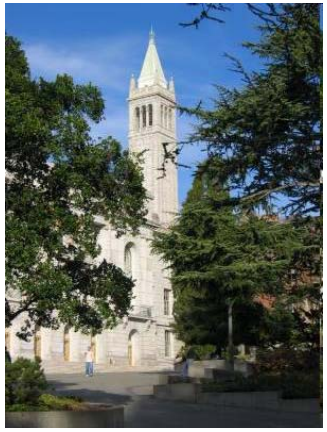
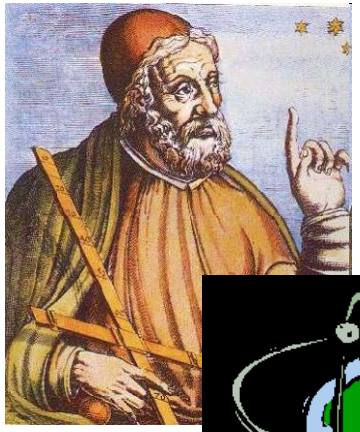
- Zur Illustration diverser Konzepte werden in der Vorlesung / Übung mehrere Beispiele verwendet
- Beispiel 1: Aufzugssteuerung
  - Verteiltes System:
    - Steuerungsrechner (einfach oder redundant ausgelegt)
    - Microcontroller zur Steuerung der Sensorik / Aktorik
    - CAN-Bus zur Kommunikation
- Beispiel 2: Ampel

Modellierung von Echtzeitsystemen

Aktoren, Ausführungsmodelle

Werkzeuge: Ptolemy

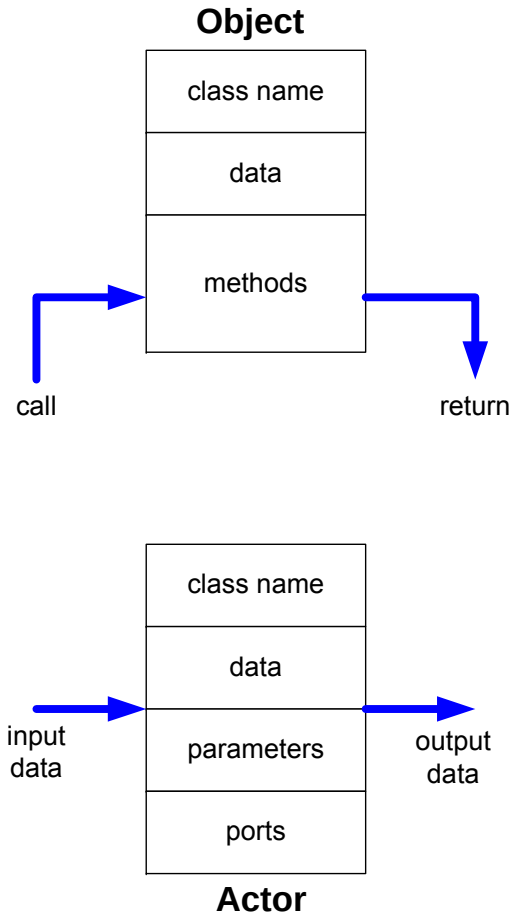
Ptolemy


- Das Ptolemy*-Projekt an der UC Berkeley untersucht verschiedene Modellierungs-methodiken für eingebettete Systeme mit einem Fokus auf verschiedene Ausführungs-modelle (Models of Computation)
- Ptolemy unterstützt
  - Modellierung
  - Simulation
  - Codegenerierung
  - Formale Verifikation (teilweise)
- Weitere Informationen unter: http://ptolemy.eecs.berkeley.edu/

***Claudius Ptolemaeus**, (* um 100, vermutlich in Ptolemais Hermii, Ägypten; † um 175, vermutlich in Alexandria), war ein griechischer Mathematiker, Geograph, Astronom, Astrologe, Musiktheoretiker und Philosoph. Ptolemäus schrieb die *Mathematike Syntaxis* („mathematische Zusammen-stellung“), später *Megiste Syntaxis* („größte Zusammenstellung“), heute *Almagest* (abgeleitet vom Arabischen *al-Majisṭī*) genannte Abhandlung zur Mathematik und Astronomie in 13 Büchern. Sie war bis zum Ende des Mittelalters ein Standardwerk der Astronomie und enthielt neben einem ausführlichen Sternenkatalog eine Verfeinerung des von Hipparchos von Nicäa vorgeschlagenen geozentrischen Weltbildes, das später nach ihm *Ptolemäisches Weltbild* genannt wurde. (Wikipedia)

Ptolemy: Aktororientiertes Design



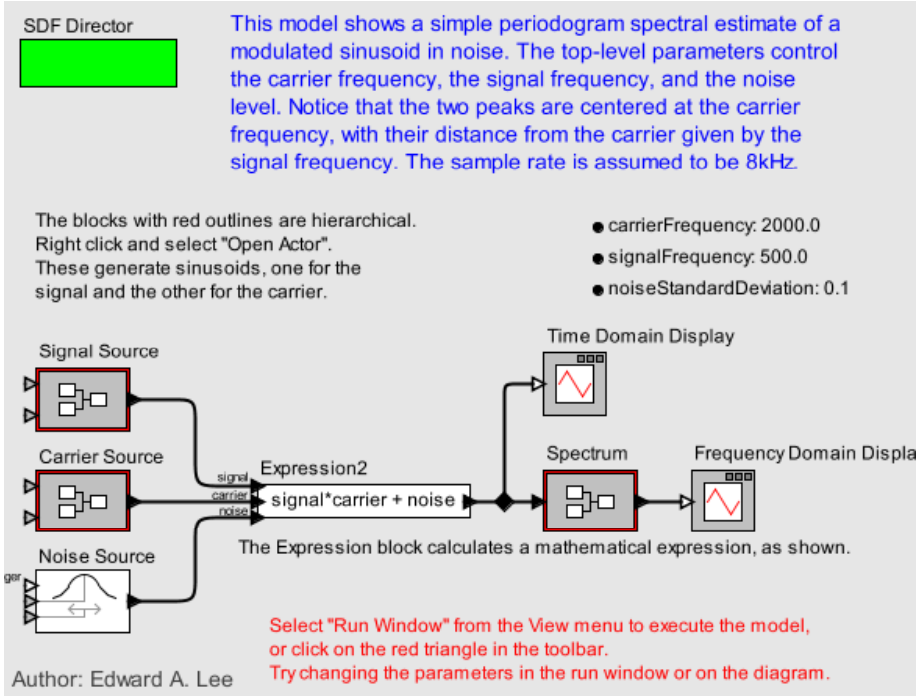
- Ptolemy-Modelle basieren auf Aktoren anstelle von Objekten
- Objekte:
  - Fokus liegt auf Kontrollfluss
  - Objekte werden manipuliert
- Aktoren
  - Fokus liegt auf Datenfluss
  - Aktoren manipulieren das System
- Vorteil beider Ansätze: erhöhte Wiederverwendbarkeit
- Vorteil von Aktoren: leichtere Darstellung von Parallelität

Ptolemy: Aktororientiertes Design

- Ausführungsmodelle (models of computation) bestimmen die Interaktion von Komponenten/Aktoren
- Die Eignung eines Ausführungsmodells hängt von der Anwendungs-domäne, aber auch der verwendeten Hardware, ab
- In Ptolemy wird durch die Einführung von „Dirigenten“ (director) die funktionale Ausführung (Verschaltung der Aktoren) von der zeitlichen Ausführung (Abbildung im Direktor) getrennt.
- Aktoren können unter verschiedenen Ausführungsmodellen verwendet werden (z.B. synchron, asynchron)
- Verschiedene Ausführungsmodelle können hierarchisch geschachtelt werden (modal models).
  - Typisches Beispiel: Synchroner Datenfluss und Zustandsautomaten

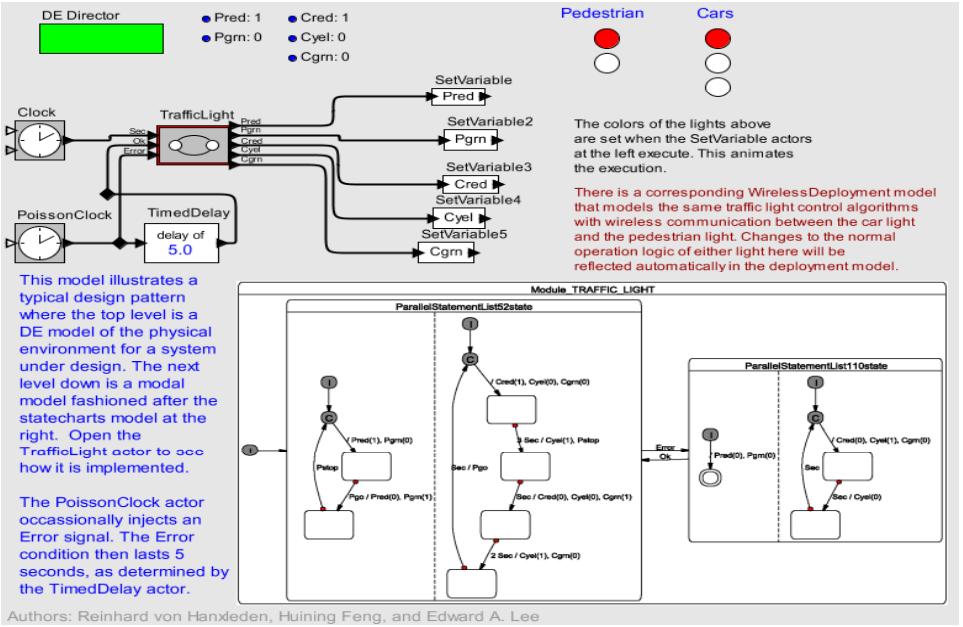
Example Ptolemy Model of Computation: Synchronuous Dataflow

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Daten werden zyklisch verarbeitet
  - Pro Runde wird genau einmal der Datenfluss ausgeführt
- Vorteile:
  - Statische Speicherallokation
  - Statischer Schedule berechenbar
  - Verklemmungen detektierbar
  - Laufzeit kann bestimmt werden
- Werkzeuge:
  - Matlab
  - Labview
  - **EasyLab**



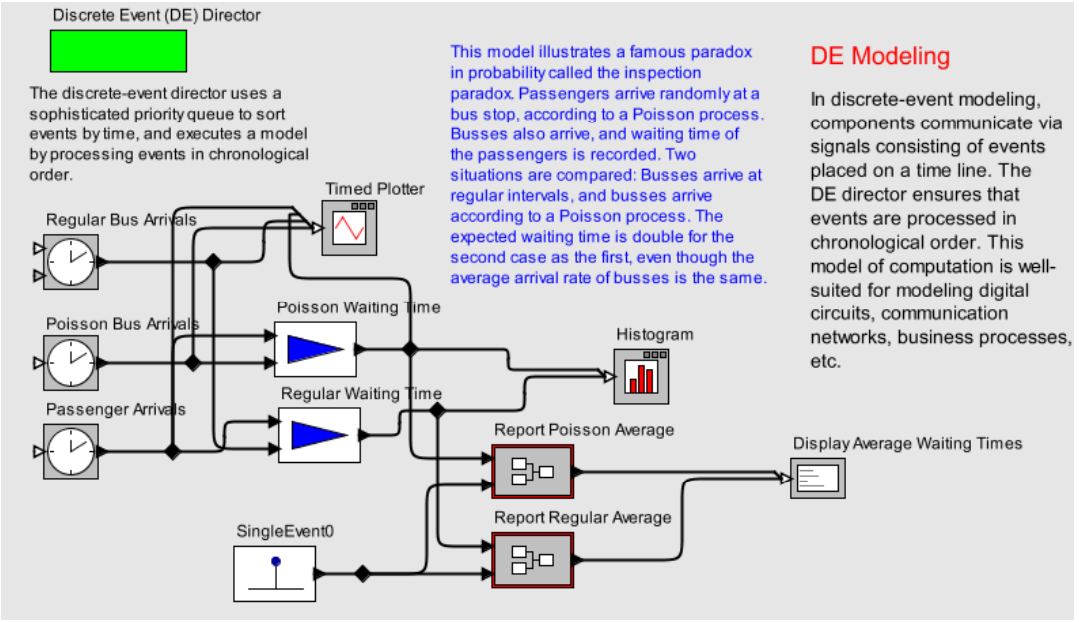
Example Ptolemy Model of Computation: Synchronous Reactive

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Diskrete Ereignisse (DE) werden zyklisch verarbeitet (Ereignisse müssen nicht jede Runde eintreffen)
  - Pro Runde wird genau eine Reaktion berechnet
  - Häufig verwendet in Zusammenhang mit Finite State Machines
- Vorteile:
  - einfache formale Verifikation
- Werkzeuge:
  - Esterel Studio
  - Scade



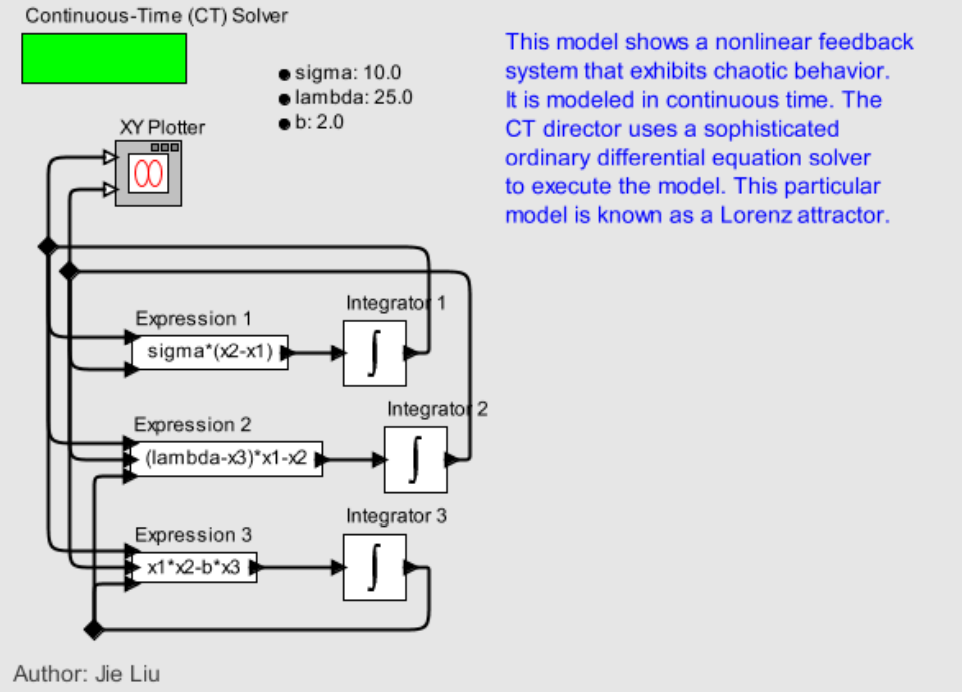
Example Ptolemy Model of Computation: Discrete Event

- Prinzip:
  - Kommunikation über Ereignisse
  - Jedes Ereignis trägt einen Wert und einen Zeitstempel
- Anwendungsgebiet:
  - Digitale Hardware
  - Telekommunikation
- Werkzeuge:
  - VHDL
  - Verilog
- Varianten:
  - Distributed Discrete Events



Example Ptolemy Model of Computation: Continuous Time

- Prinzip:
  - Verwendung kontinuierlicher Signale (bestimmt gemäß Differential-gleichungen)
- Anwendungsgebiet:
  - Simulation
- Werkzeuge:
  - Simulink
  - Labview



Weitere Models of Computation

- Component Interaction:
  - Mischung von daten- und anfragegetriebener Ausführung
  - Beispiel: Web Server
- Discrete Time:
  - Erweiterung des synchronen Datenflussmodels um Zeit zwischen Ausführungen zur Unterstützung von Multiratensystemen
- Time-Triggered Execution
  - Die Ausführung wird zeitlich geplant
  - Anwendungsgebiet: kritische Regelungssysteme
  - Werkzeug: Giotto, FTOS
- Process Networks
  - Prozess senden zur Kommunikation Nachrichten über Kanäle
  - Kanäle können Nachrichten speichern: asynchrone Nachrichten
  - Anwendungsgebiet: verteilte Systeme
- Rendezvous
  - synchrone Kommunikation verteilter Prozesse (Prozesse warten am Kommunikationspunkt, bis Sender und Empfänger bereit sind)
  - Beispiele: CSP, CCS, Ada

Modellierung von Echtzeitsystemen

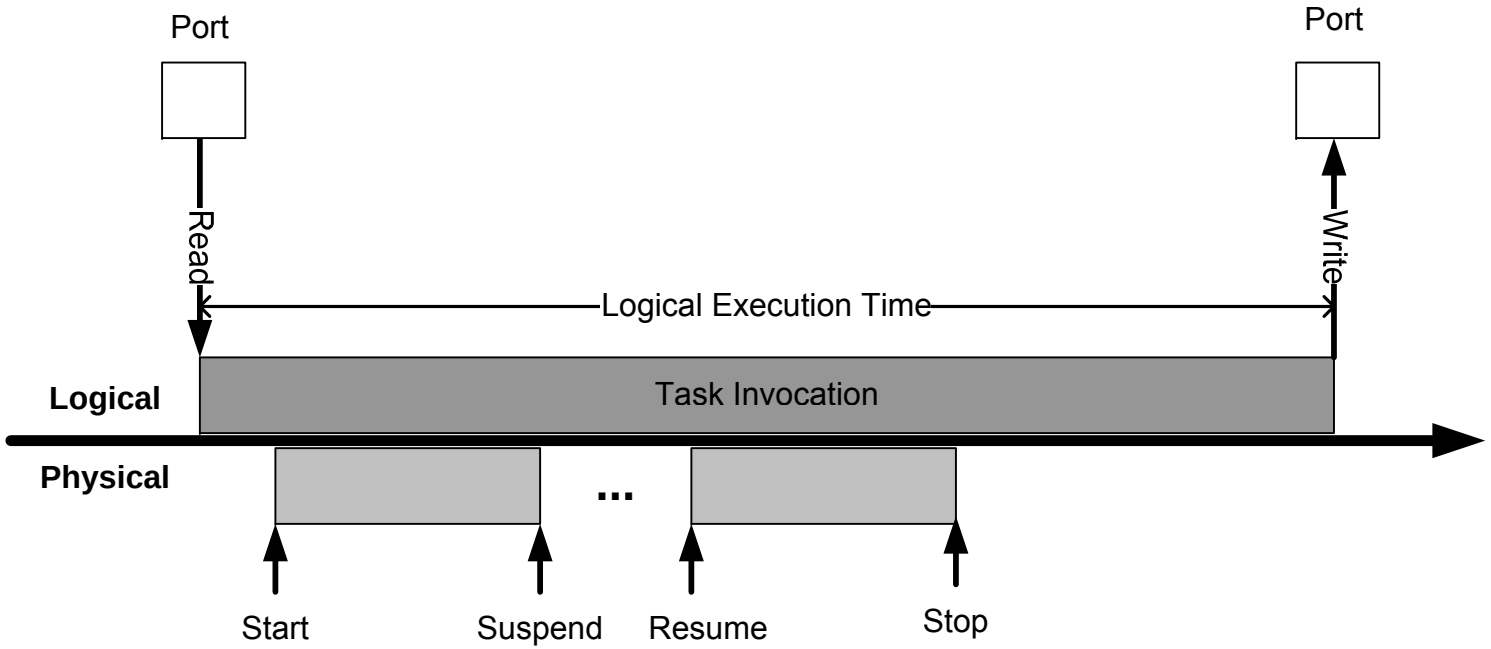
Zeitgesteuerte Systeme

Werkzeug: Giotto

Giotto: Hintergrund

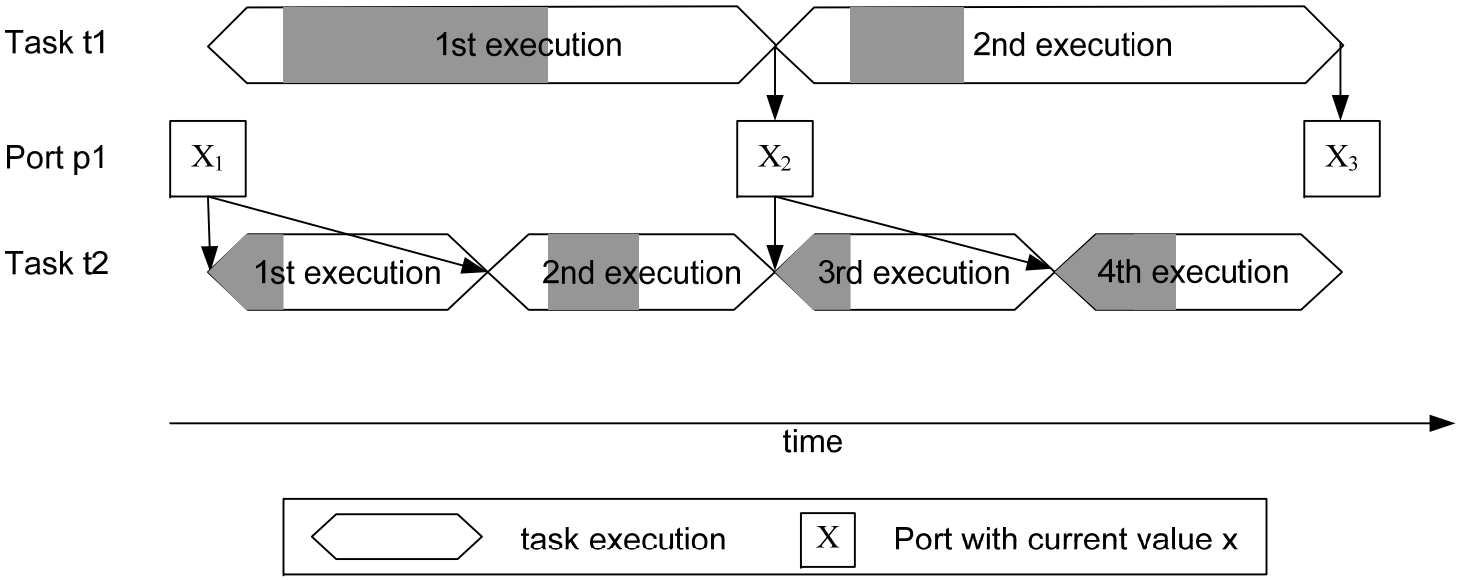
- Programmierumgebung für eingebettete Systeme (evtl. ausgeführt im verteilten System)
- Ziel:
  - strikte Trennung von plattformunabhängiger Funktionalität und plattformabhängigen Scheduling und Kommunikation
  - temporaler Determinismus
- Hauptkonzept: Logische Ausführungszeiten
- Aktoren:
  - Tasks
    - Programmblock aus sequentiellen Code
    - keine Synchronisationspunkte, blockende Operationen erlaubt
    - Schnittstellen: Ports
  - Drivers: realisieren die Kommunikation zwischen Ports
  - Flexibilität durch Modes/Guards
- Ausführung durch virtuelle Maschinen:
  - Embedded Machine: Reaktion der Tasks auf physikalische Ereignisse
  - Scheduling Machine: physikalisches Scheduling
- http://embedded.eecs.berkeley.edu/giotto/

Logische Ausführungszeit



Motivation siehe http://www.cs.uic.edu/~shatz/SEES/henzinger.slides.ppt

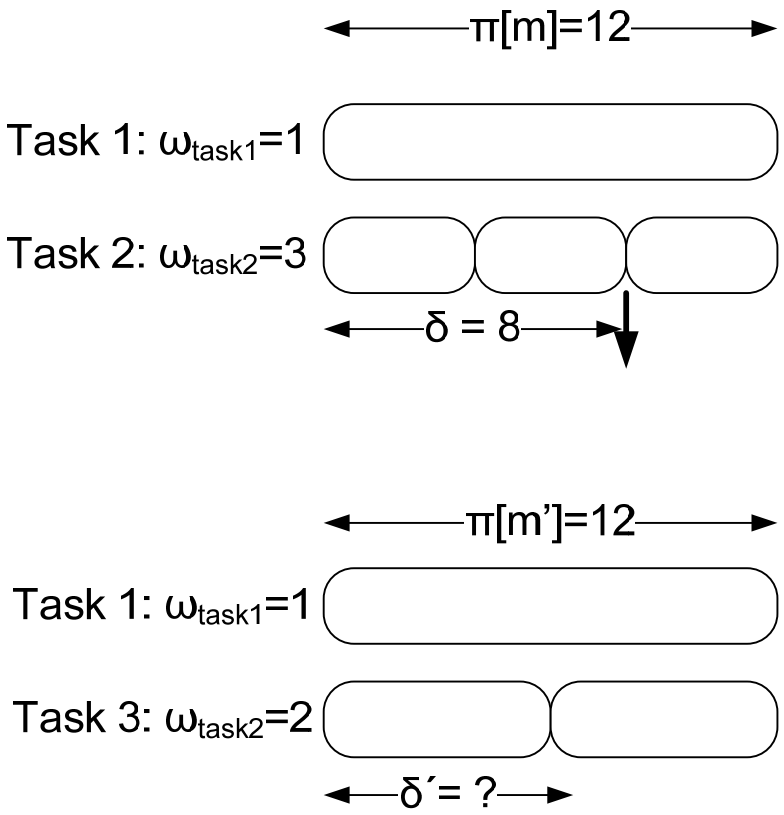
Kommunikation zwischen Tasks



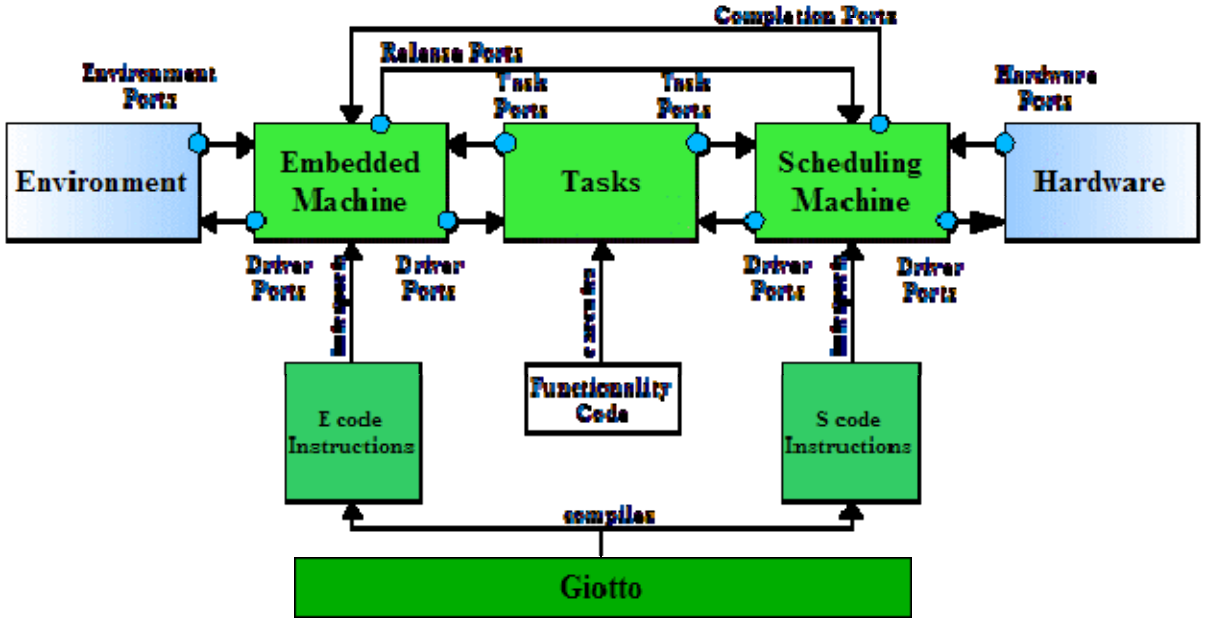
Modes / Guards

- Um die Ausführung flexibel zu gestalten, bietet Giotto Modes und Guards an
  - Guards: Boolsche Funktion, die über die Ausführung eines Tasks entscheidet (wird vor Start des Tasks aufgerufen)
  - Mode: Menge von Tasks und Drivers die zeitgleich ausgeführ werden, es kann immer nur ein Mode aktiv sein.
- Nicht-harmonischer Moduswechsel (Unterbrechung eines laufenden Modes):
  - Voraussetzung: $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$
    m: Quellmodus, m': Zielmodus, $\pi[m]$: Modusdauer m, ω_{task}: Taskfrequenz $\Rightarrow$ Logische Ausführungszeit muss gleich sein
  - Wechselmechanismus:

    γ = LCM $\{\pi[m]/\omega_{task} | \{\omega_{task}, t,) \in$ Invokes[m],
    $\delta' = \pi[m']-(\varepsilon-\delta)$ mit $\varepsilon = n^* \gamma \geq \delta$
    LCM: least common multiple, δ: aktuelle Rundenzeit, δ': neue Rundenzeit in m', $\varepsilon - \delta$: Zeit bis zum nächsten gleichzeitigen Beendigungspunkt



Ausführungsumgebung



Zusammenfassung

- Das Konzept der logischen Ausführungszeiten erlaubt eine Abstrahierung von der physikalischen Ausführungszeit und somit die Trennung von plattformunabhängigem Verhalten (Funktionalität und zeitl. Verhalten) und plattformabhängiger Realisierung (Scheduling, Kommunikation)
- Die Ausführung erfolgt über zwei virtuelle Maschinen:
  - E-Machine: Interaktion mit der Umgebung (reaktiv)
  - S-Machine: Interaktion mit der ausführenden Plattform (proaktiv), Vorteil: Schedule kann vorab berechnet werden
- Weitere Literaturhinweise:
  - Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003
  - Henzinger et al.: Schedule-Carrying Code, Proceedings of the Third International Conference on Embedded Software (EMSOFT), 2003

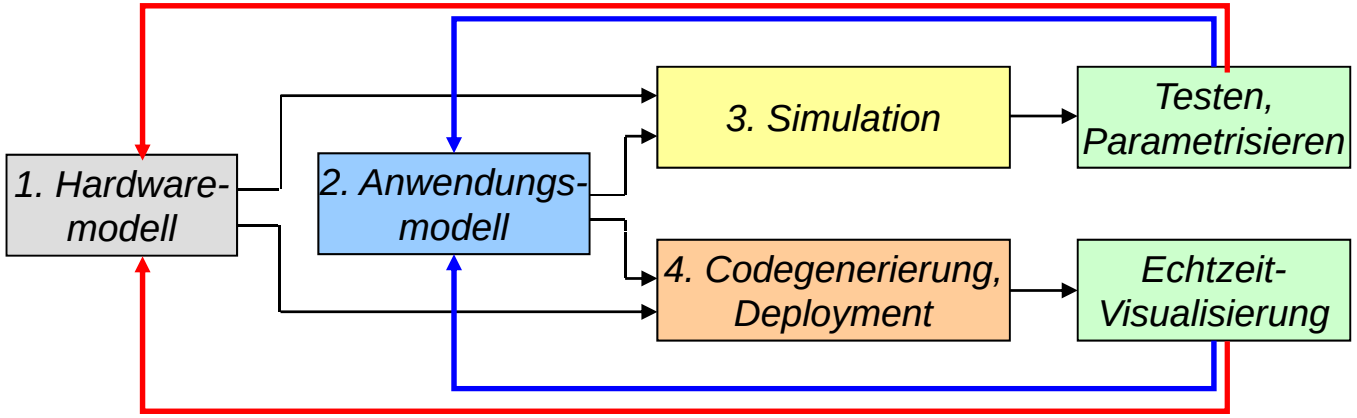
Modellierung von Echtzeitsystemen

Synchroner Datenfluss

Werkzeug: EasyLab

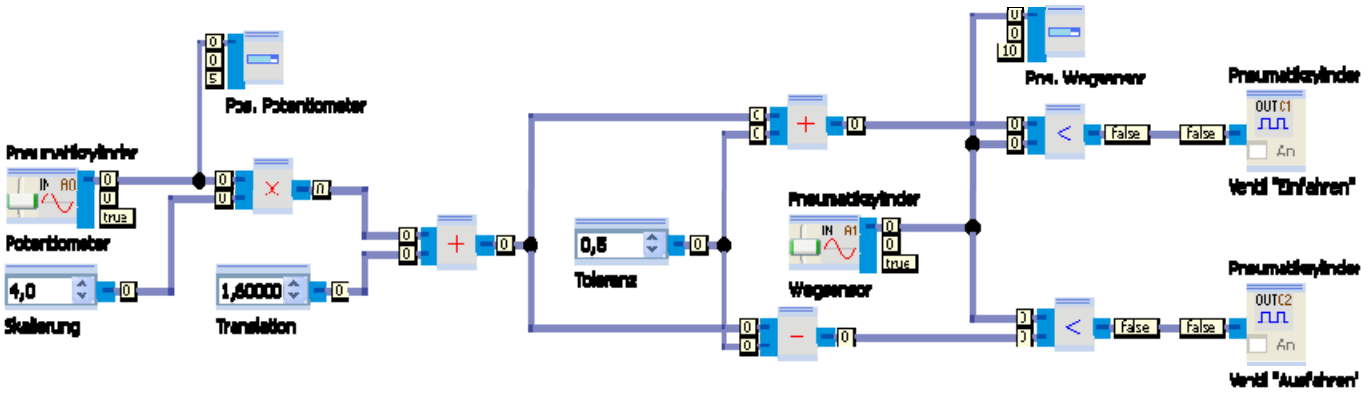
Entwicklungsprozess in EasyLab

1. Spezifikation der Zielhardware
2. Modellierung der Zustandslogik sowie der abzuarbeitenden Aufgabe je Zustand
3. Simulation des Programms zum Testen und zur Erkennung von Fehlern
4. Codegenerierung und Echtzeit-Visualisierung des Zustands der Zielhardware



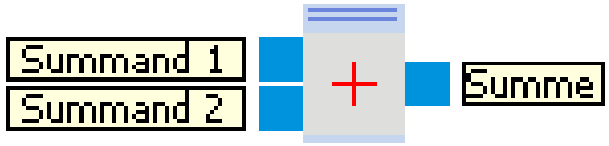
Synchroner Datenfluss

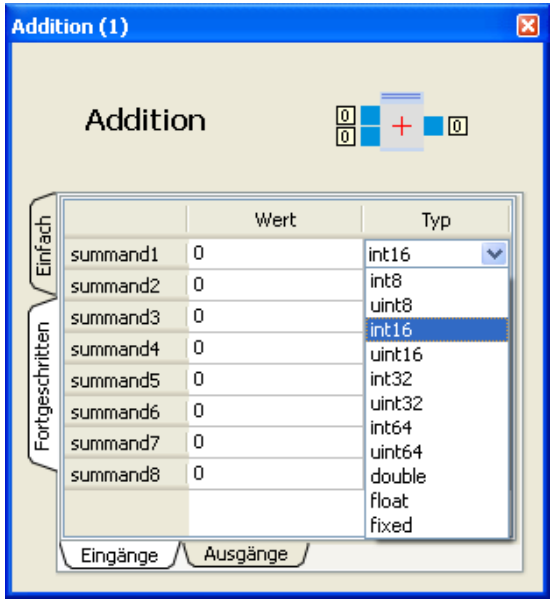
- Funktionsblöcke
  - Eingänge und Ausgänge
  - Interner Zustand
  - Zugeordnete Aktionen
- Komposition von Funktionsblöcken
  - Typisierte Verbindungen
  - In Bearbeitung: Hierarchische Funktionsblöcke
- Grundlagen
  - Synchronitätshypothese
  - „Black boxes“-Sicht
  - Effizienz und Zuverlässigkeit
    - Berechnung statischer Schedules
    - Deterministisches Laufzeitverhalten
    - Statische Speicherallokation (RAM)
    - Erzeugung kompakten Codes (ROM)



Überladen von Aktoren

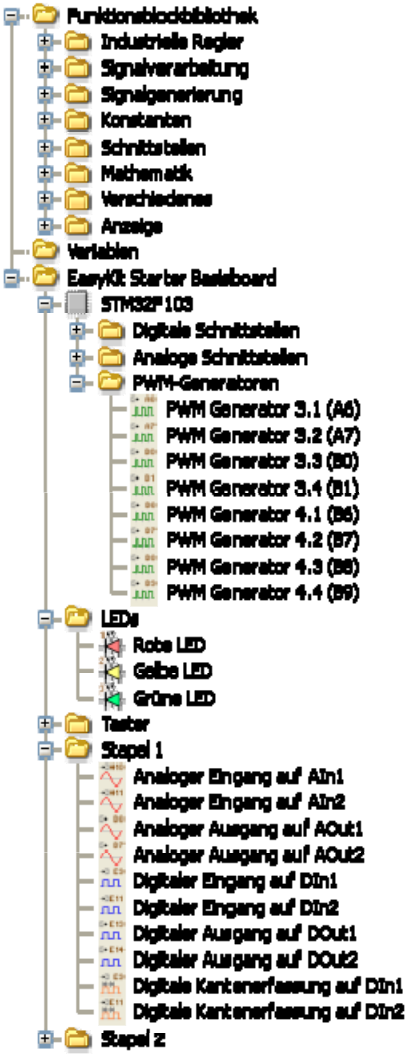
- Aktoren können typunabhängig angeboten werden
- Der Typ eines Aktors ist solange frei wählbar, bis sich durch Anschlussbelegung der Typ automatisch ergibt



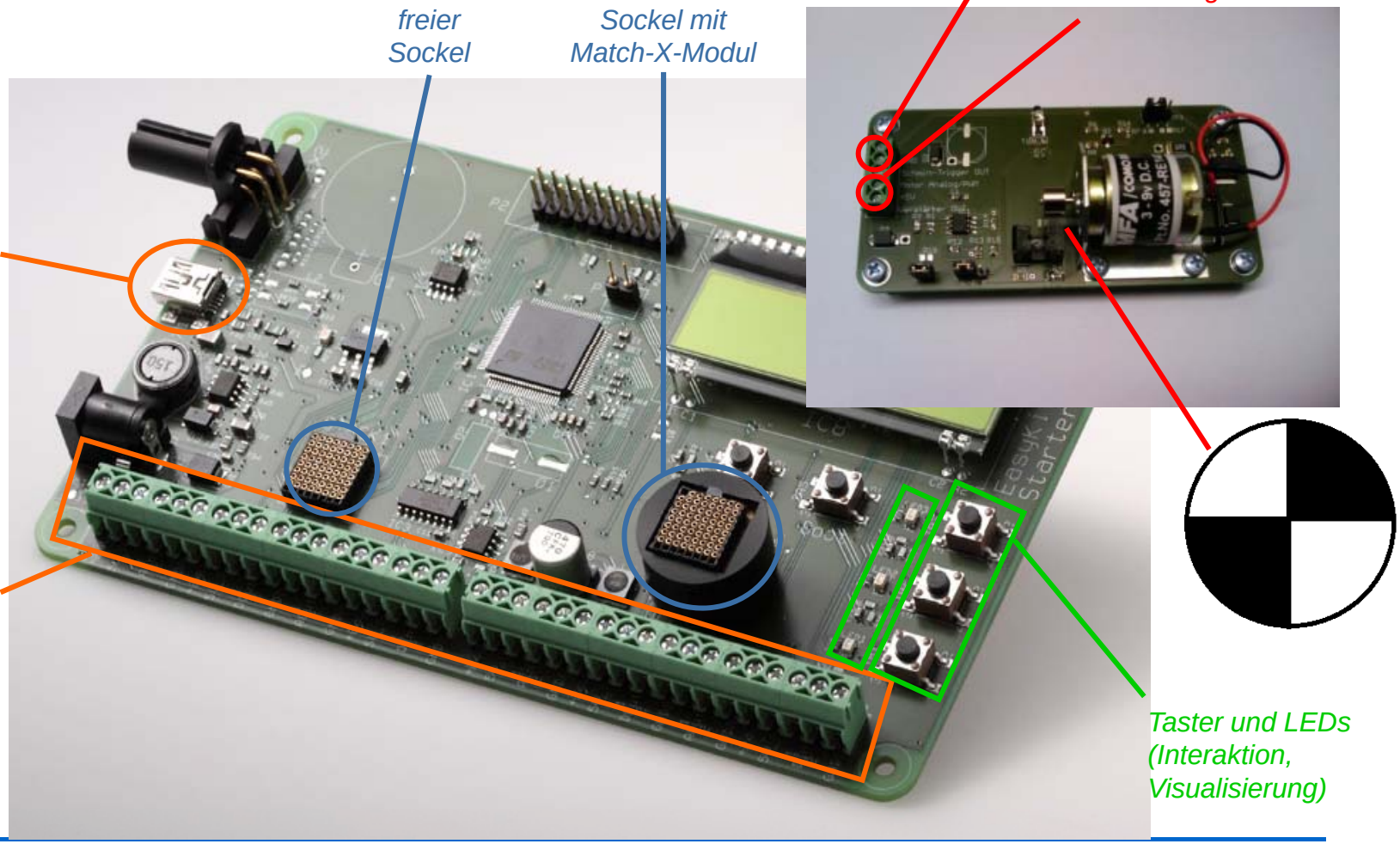


Geräte

- Hardware
  - Mikrocontroller
  - Sensoren und Aktoren
- Geräte
  - Hierarchische Beschreibung der Hardware
  - Ressourcenmanagement
    - Modellierung der Hardwarefunktionalität
    - Beispiele: I/O-Pins, ADCs, Timer,
  - Schnittstelle zur Anwendungslogik
    - Hardwarezugriff
    - Geräte bieten eine Menge von Funktionsblöcken an

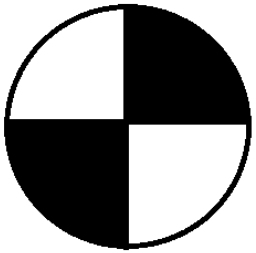


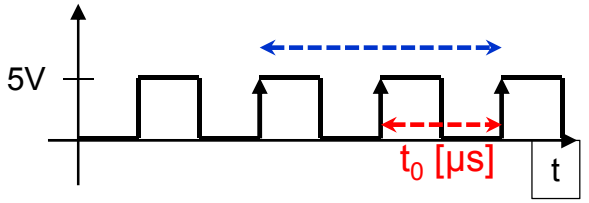
Anwendung – EasyKit Starter



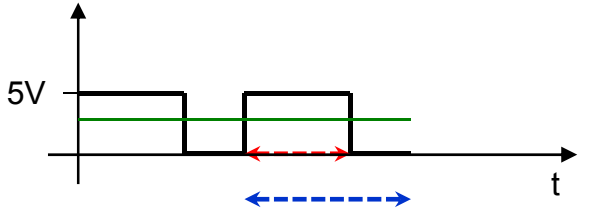
Aufgabe – Drehzahlregelung

- Eingabe
  - Sollwert
  - Ist-Wert: Lichtschranke;
    Messung der Zeit t_0 zwischen
    zwei steigenden Flanken des Signals

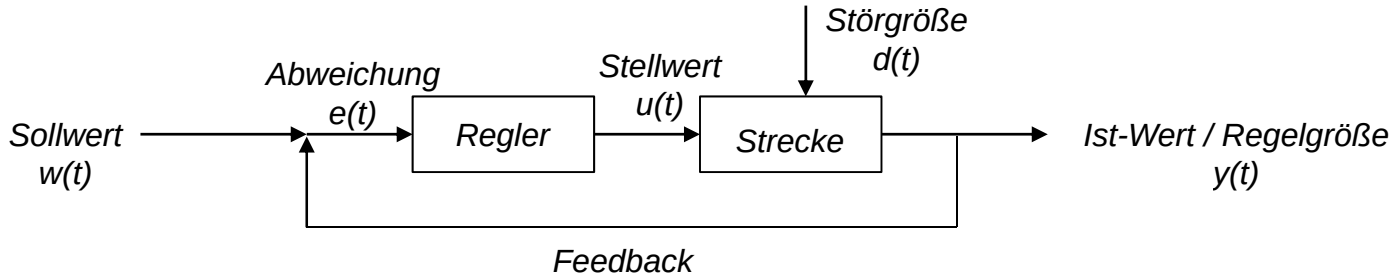




- Ausgabe
  - Stellwert: Motorspannung 0..5 V
    als Tastverhältnis (in Prozent)



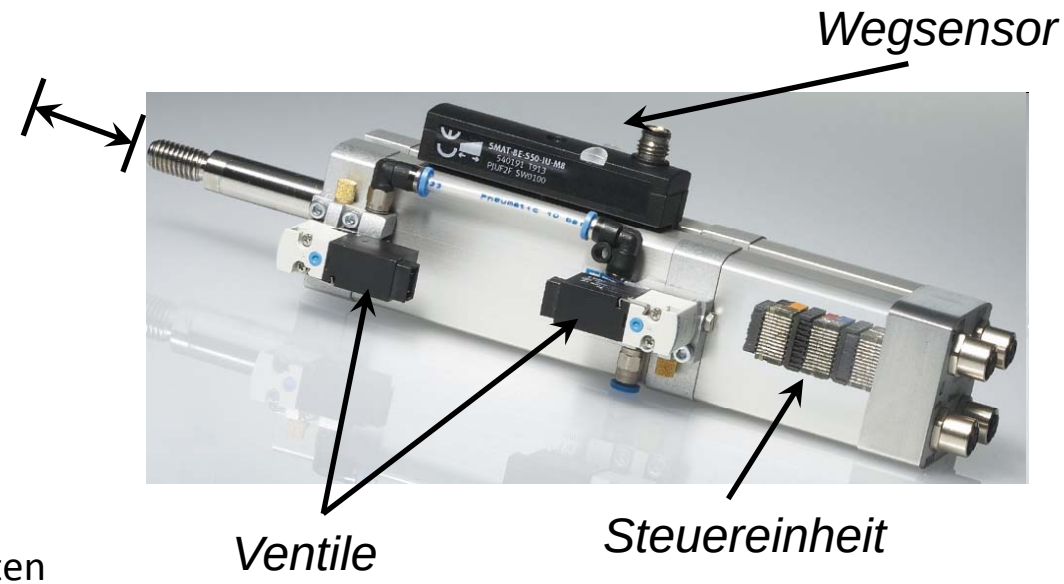
Aufgabe – Drehzahlregelung

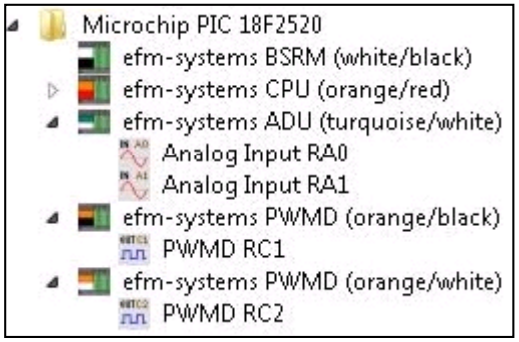


- P-Regler $u(t) = K_p \cdot e(t)$
- I-Regler $u(t) = \frac{1}{T_N} \int_0^t (\tau) d\tau$ T_N: Nachstellzeit
- PI-Regler $u(t) = K_p \cdot [e(t) + \frac{1}{T_N} \int_0^t (\tau) d\tau]$

Anwendung – Pneumatischer Zylinder

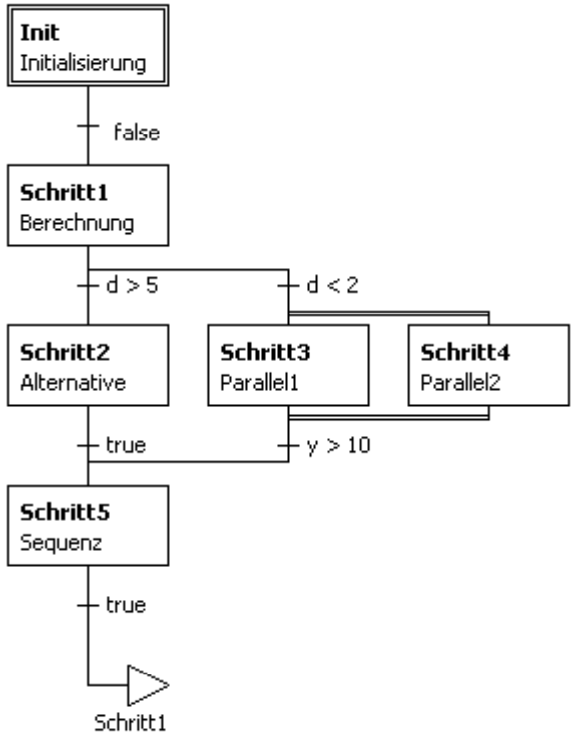
- Hardware
  - Zylinder
    - Positionssensor (Kolben)
    - Endlagenschalter
    - Zwei Magnetventile
  - Steuerungseinheit
    - Mikrocontroller
    - Analog-Digital-Wandler
    - Treiber für induktive Lasten
- Ziel: Positionssteuerung des Kolbens
- Umsetzung
  - Hardware-Modell aus Bibliothek für Match-X
  - Anwendungsmodell
    - Kleines Datenflussdiagramm
    - Integration der Hardwarefunktionalität





EasyLab: Zustandsfluss-Diagramme

- Zustandsfluss-Diagramme
  - An IEC-61131-3 angelehnt
  - Zustand: Referenz auf SDF-Modell
  - Transitionsbedingungen:
    Boolesche Ausdrücke mit Variablen
- Komposition von Zuständen
  - Zustandsfolgen
  - Alternativ- und Parallelzweige
  - Sprünge
- Vorteile
  - Modellierung von Zuständen
    (vgl. Automaten)
  - Diagrammart weit verbreitet in
    Anwendungsdomäne



Modellierung von Echtzeitsystemen

Reaktive Systeme

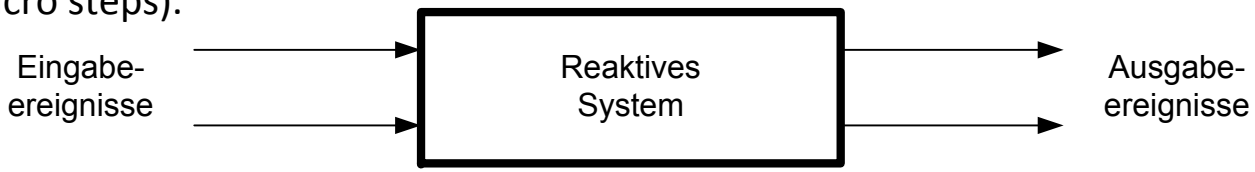
Werkzeuge: SCADE, Esterel Studio

Esterel

- Esterel ist im klassischen Sinne eher eine Programmiersprache, als eine Modellierungs-sprache
- Esterel wurde von Jean-Paul Marmorat und Jean-Paul Rigault entwickelt um die Anforderungen von Echtzeitsystemen gezielt zu unterstützen:
  - direkte Möglichkeit zum Umgang mit Zeit
  - Parallelismus direkt in der Programmiersprache
- G. Berry entwickelt die formale Semantik für Esterel
- Es existieren Codegeneratoren zur Generierung von u.a. **sequentiellen** C, C++ Code:
  - In Esterel werden (parallele) Programme in **einen** endlichen Automaten umgewandelt
  - Aus dem endlichen Automaten wird ein Programm mit **einem** Berechnungsstrang erzeugt ⇒ deterministische Ausführung trotz paralleler Modellierung.
- SCADE (ein kommerzielles Tool, dass u.a. die Esterel-Sprache verwendet) wurde bei der Entwicklung von Komponenten für den Airbus A380 eingesetzt.
- Ein frei verfügbarer Esterel-Compiler kann unter http://www-sop.inria.fr/esterel.org/files/ bezogen werden (siehe Links auf der Vorlesungs-Homepage).

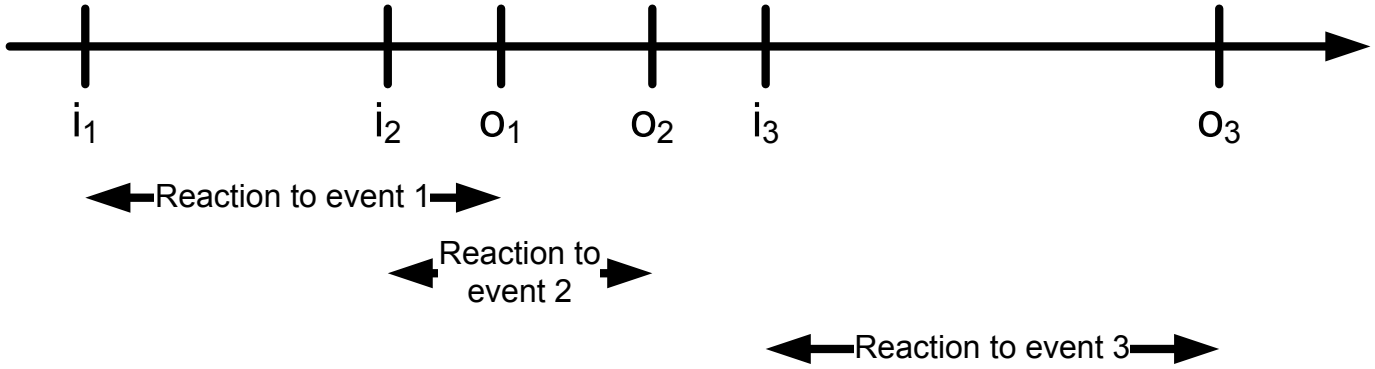
Einführung in Esterel

- Esterel gehört zu der Familie der **synchronen** Sprachen. Dies sind Programmiersprachen, die optimiert sind, um **reaktive Systeme** zu programmieren. Weitere Vertreter: Lustre, Signal, Statecharts
- Bei **reaktiven Systemen** erfolgen Reaktionen direkt auf **Eingabeereignisse**
- Synchrone Sprachen zeichnen sich vor allem dadurch aus, dass
  - Interaktionen (Reaktionen) des Systems mit der Umgebung die Basisschritte des Systems darstellen (**reaktives System**).
  - Anstelle von physikalischer Zeit die **logische Zeit** (die Anzahl der Interaktionen) verwendet wird.
  - Interaktionen, oft auch **macro steps** genannt, bestehen aus einzelnen Teilschritten (micro steps).



Reaktive Systeme

- In reaktiven Systemen (reactive / reflex systems) werden für Eingabeereignisse Ausgaben unter Einhaltung zeitlicher Rahmenbedingungen erzeugt.
- Reaktive Systeme finden u.a. Anwendung in der Industrie zur Prozesssteuerung und zur Steuerung / Regelung in Automobilen und Flugzeugen.
- Schwerpunkte bei der Umsetzung von reaktiven Systemen sind Sicherheit und Determinismus.
- Bearbeitung der Ereignisse kann sich überlappen (i input, o output)



Synchronitätshypothese

- Die Synchronitätshypothese (synchrony hypothesis) nimmt an, dass die zugrunde liegende physikalische Maschine des Systems unendlich schnell ist.

  → Die Reaktion des Systems auf ein Eingabeereignis erfolgt augenblicklich (ohne erkennbare Zeitverzögerung). Reaktionsintervalle reduzieren sich zu Reaktionsmomenten (reaction instants).

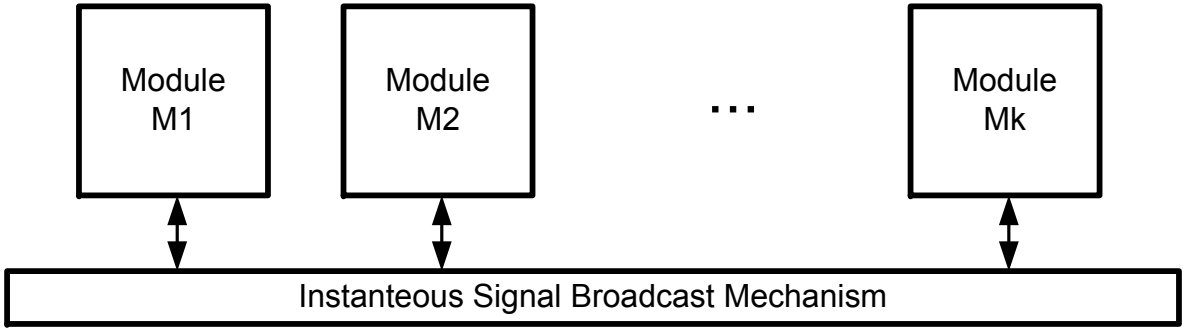
- **Rechtfertigung**: Diese Annahme ist korrekt, wenn die Wahrscheinlichkeit des Eintreffens eines zweiten Ereignisses, während der initialen Reaktion auf das vorangegangene Ereignis, sehr klein ist.
- Esterel erlaubt das gleichzeitige Auftreten von mehreren Eingabeereignissen. Die Reaktion ist in Esterel dann vollständig, wenn das System auf alle Ereignisse reagiert hat.

Determinismus

- Esterel ist deterministisch: auf eine Sequenz von Ereignissen (auch gleichzeitigen) muss immer dieselbe Sequenz von Ausgabe Ereignissen folgen.
- Alle Esterel-Anweisungen und -Konstrukte sind garantiert deterministisch. Die Forderung nach Determinismus wird durch den Esterel Compiler überprüft.
- Durch den Determinismus wird die Verifikation von Anwendungen wesentlich vereinfacht, allerdings birgt er auch die Gefahr, dass Ereignisse „vergessen“ werden, falls sie exakt zeitgleich mit höher priorisierten Ereignissen eintreffen.

Grundlagen der Esterel Sprache: Signale / Sensoren

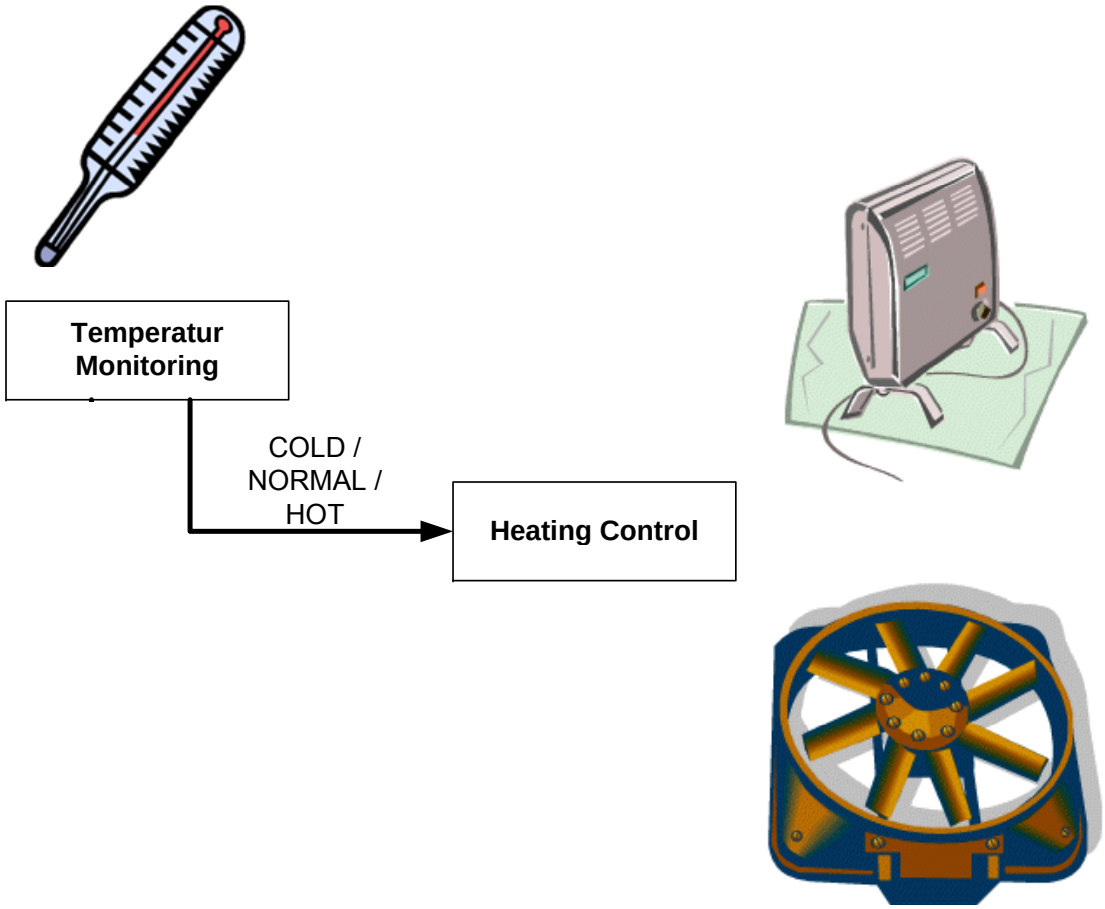
- Die Kommunikation im System erfolgt durch Signale und Sensoren
  - Sensoren stellen Messwerte zur Verfügung; Es muss immer ein Wert anliegen.
  - Signale können zur Ein- und zur Ausgabe verwendet werden. Es muss nicht immer ein Wert anliegen.
- Man unterscheidet dabei zwei Arten von Signalen:
  - Wertbehaftete Signale (Signal liegt mit einem Wert an oder liegt nicht an)
  - *pure* Signale (Signal liegt an oder nicht)
- Esterel Programme können in mehrere Module aufgeteilt werden
- Kommunikation der Module erfolgt über einen Broadcast-Mechanismus



Grundlagen der Esterel Sprache: Programmsteuerung

- Das Verhalten des Programms wird durch Signale beeinflusst:
  - *emit*: Ein Signal wird ausgesandt.
  - *await*: Auf das Auftreten eines Signals wird gewartet.
  - *present*: Prüft, ob ein Signal anliegt.
- Die Ausführung eines Moduls kann mittels `abort` abgebrochen werden:
  - Syntax: `abort Body when Exit_Condition`
- Ein periodisches Ausführen von Anweisungen kann mit `every` umgesetzt werden.
  - Syntax: `every Occurrence do Body end every`
- Komposition von Anweisungen:
  - Blöcke von Befehlen (Module) können **nacheinander** oder **gleichzeitig** ausgeführt werden.
  - Blöcke von Befehlen (Module) können **wiederholt** ausgeführt werden.
  - Blöcke von Befehlen (Module) können **unterbrochen** werden.

Beispiel: Einfache Temperaturregelung



Beschreibung Beispiel

- Ziel: Regelung der Temperatur (Betriebstemperatur 5-40 Grad Celsius) mittels eines sehr einfachen Reglers.
- Ansatz:
  - Nähert sich die Temperatur einem der Grenzwerte, so wird der Lüfter bzw. die Heizung (Normalstufe) eingeschaltet.
  - Verbleibt der Wert dennoch im Grenzbereich, so wird auf die höchste Stufe geschaltet.
  - Ist der Wert wieder im Normalbereich, so wird (zur Vereinfachung) der Lüfter bzw. die Heizung wieder ausgeschaltet.
  - Wird die Betriebstemperatur über- bzw. unterschritten, so wird ein Abbruchssignal geschickt.

Esterel Code für Temperatur-Regelung (Auszug)

```
module TemperatureControler:
input TEMP: integer, SAMPLE_TIME, DELTA_T;
output HEATER_ON, HEATER_ON_STRONG,
       HEATER_OFF,VENTILATOR_ON, VENTILATOR_OFF,
       VENTILATOR_ON_STRONG, ABORT;

relation SAMPLE_TIME => TEMP;

  signal COLD, NORMAL, HOT in
    every SAMPLE_TIME do
      await immediate TEMP;
      if  ?TEMP<5 or ?TEMP>40 then emit ABORT
      elsif ?TEMP>=35 then emit HOT
      elseif ?TEMP<=10 then emit COLD
      else emit NORMAL
      end if
    end every
    ||
    loop
      await
        case COLD do
          emit HEATER_ON;
          abort
            await NORMAL;
            emit HEATER_OFF;
          when DELTA_T do
            emit HEATER_ON_STRONG;
            await NORMAL;
            emit HEATER_OFF;
          end abort
        case HOT do
%...
      end await
    end loop
  end signal
end module
```

Esterel-Konstrukt: Module

- **Module** definieren in Esterel (wiederverwendbaren) Code. Module haben ähnlich wie Unterprogramme ihre eigenen Daten und ihr eigenes Verhalten.
- Allerdings werden Module nicht aufgerufen, vielmehr findet eine Ersetzung des Aufrufs durch den Modulcode zur Übersetzungszeit statt (Inlining).
- Globale Variable werden nicht unterstützt. Ebenso sind rekursive Moduldefinitionen nicht erlaubt.
- Syntax:

```
%this is a line comment
module module-name:
declarations and compiler directives
%signals, local variables etc.
body
end module % end of module body
```

Esterel-Konstrukt: Parallele Komposition

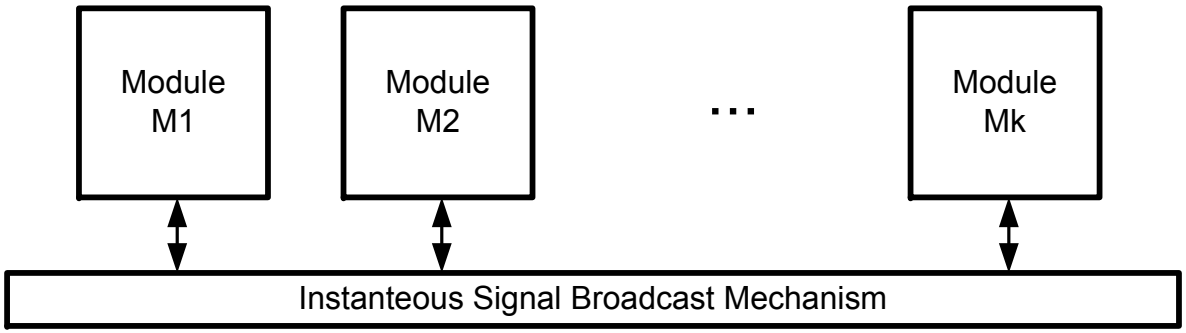
- Zur parallelen Komposition stellt Esterel den Operator || zur Verfügung. Sind P1 und P2 zwei Esterel-Programme, so ist auch P1||P2 ein Esterel-Programm mit folgenden Eigenschaften:
  - Alle Eingabeereignisse stehen sowohl P1 als auch P2 zur Verfügung.
  - Jede Ausgabe von P1 (oder P2) ist im gleichen Moment für P2 (oder P1) sichtbar.
  - Sowohl P1 als auch P2 werden parallel ausgeführt und die Anweisung P1||P2 endet erst, wenn beide Programme beendet sind.
  - Es können keine Daten oder Variablen von P1 und P2 gemeinsam genutzt werden.
- Zur graphischen Modellierung stehen parallele Teilautomaten zur Verfügung.

Signale

- Zur Kommunikation zwischen Komponenten (Modulen) werden Signale eingeführt. Signale sind eine logische Einheit zum Informationsaustausch und zur Interaktion.
- Die *Deklaration* eines Signals erfolgt am Beginn des Moduls. Der Signalname wird dabei typischerweise in Großbuchstaben geschrieben. Zudem muss der Signaltyp festgelegt werden.
- Esterel stellt verschiedene Signale zur Verfügung. Die Klassifikation erfolgt nach:
  - Sichtbarkeit: Schnittstellen (interface) Signale vs. lokale Signale
  - Enthaltener Information: pure Signale vs. wertbehaftete Signale (typisiert)
  - Zugreifbarkeit der Schnittstellensignale: Eingabe (input), Ausgabe(output), Ein- und Ausgabe (inputoutput), Sensor (Signal, das immer verfügbar ist und das nur über den Wert zugreifbar ist)

Esterel-Konstrukt: Broadcast-Mechanismus

- **Versand**: Der Versand von Signalen durch die `emit` Anweisung (terminiert sofort) erfolgt über einen Broadcast-Mechanismus, d.h. Signale sind immer sofort für alle anderen Module verfügbar.
  Die `sustain` Anweisung erzeugt in jeder Runde das entsprechende Signal und terminiert nicht.
- **Zugriff**: Prozesse können per `await` auf Signale warten oder prüfen, ob ein Signal momentan vorhanden ist (`if`). Auf den Wert eines wertbehafteten Signals kann mittels des Zugriffsoperator ? zugegriffen werden.



Esterel-Konstrukt: Ereignisse (Events)

- **Ereignisse** setzen sich zu einem bestimmten Zeitpunkt (**instant**) aus den Eingabesignalen aus der Umwelt und den Signalen, die durch das System als Reaktion ausgesendet werden, zusammen.
- Esterel-Programme können nicht direkt auf das ehemalige oder zukünftige Auftreten von Signalen zurückgreifen. Auch kann nicht auf einen ehemaligen oder zukünftigen Moment zugegriffen werden.
- Einzige Ausnahme ist der Zugriff auf den letzten Moment. Durch den Operator `pre` kann das Auftreten in der vorherigen Runde überprüft werden.

Beziehungen (relations)

- Der Esterel-Compiler erzeugt aus der Esterel-Datei einen endlichen Automaten. Hierzu müssen für jeden Zustand (Block) sämtliche Signalkombinationen getestet werden.
- Um bei der automatischen Generierung des endlichen Automaten des Systems die Größe zu reduzieren, können über die `relation` Anweisung Einschränkungen in Bezug auf die Signale spezifiziert werden:
- `relation Master-signal-name => Slave-signal-name;`

  Bei jedem Auftreten des Mastersignals muss auch das Slave-Signal verfügbar sein.
- `relation Signal-name-1 # Signal-name-2 # ... # Signal-name-n;`

  In jedem Moment darf maximal eines der spezifizierten Signale Signal-name-1, Signal-name-2 ,..., Signal-name-n präsent sein.

Zeitdauer

- Die Zeitachse wird in Esterel in diskrete Momente (**instants**) aufgeteilt. Über die Granularität wird dabei in Esterel keine Aussage getroffen.
- Zur deterministischen Vorhersage des zeitlichen Ablaufes von Programmen wird jede Anweisung in Esterel mit einer genauen Definition der Ausführungszeitdauer verknüpft.
- So terminiert beispielsweise `emit` sofort, während `await` so viel Zeit benötigt, bis das assoziierte Signal verfügbar ist.
- Auf den folgenden Folien werden die wichtigsten Konstrukte erläutert.

Esterel-Konstrukt: await Anweisung

```
await
    case Occurence-1 do Body-1
    case Occurence-2 do Body-2
    ...
    case Occurence-n do Body-n
end await;
```

- Mit Hilfe dieser Anweisung wird auf das Eintreten einer Bedingung gewartet. Im Falle eines Auftretens wird der assoziierte Code gestartet. Werden in einem Moment mehrere Bedingungen wahr, entscheidet die textuelle Reihenfolge. So kann eine deterministische Ausführung garantiert werden.

Esterel-Konstrukt: Unendliche Schleife (infinite loop)

```
loop Body end loop;
```

- Mit Hilfe dieser Anweisung wird ein Stück Code Body endlos ausgeführt. Sobald eine Ausführung des Codes beendet wird, wird der Code wieder neu gestartet.
- **Bedingung**: die Ausführung des Codes darf nicht im gleichen Moment, indem sie gestartet wurde, terminieren.

Esterel-Konstrukt: abort

- Zur einfacheren Modellierung können Abbruchbedingungen nicht nur durch Zustandsübergänge, sondern auch direkt mit Makrozuständen verbunden werden.
- Dabei wird zwischen zwei Arten des Abbruches unterschieden:
  - weak abort: die in der Runde vorhandenen Signale werden noch verarbeitet, danach jedoch der Abbruch vollzogen
  - strong abort: der Abbruch wird sofort vollzogen, eventuell vorhandene Signale ignoriert.
- In der Sprache Esterel wird eine Abbruchbedingung durch das Konstrukt
  `abort Body when Exit_Condition`
  bzw.
  `abort Body when immediate Exit_Condition`
  ausgedrückt.

Esterel-Konstrukt: Lokale und wertbehaftete Signale

```
signal Signal-decl-1,Signal-decl-
2,...,Signal-decl-n in
   Body
end;
```

- Durch diese Anweisung werden lokale Signale erzeugt, die nur innerhalb des mit Body bezeichneten Code verfügbar sind.

```
Signal-name: Signal-type
```

- Der Typ eines wertbehaften Signals kann durch diese Konstruktion spezifiziert werden.

Esterel-Konstrukt: every Anweisung

- Mit Hilfe der `every` Anweisung kann ein periodisches Wiederstarten implementiert werden.
- Syntax:

```
every Occurence do
    Body
end every
```

- Semantik: Jedes Mal falls die Bedingung `Occurence` erfüllt ist, wird der Code Body gestartet. Falls die nächste Bedingung `Occurence` vor der Beendigung der Ausführung von Body auftritt, wird die aktuelle Ausführung sofort beendet und eine neue Ausführung gestartet.
- Es ist auch möglich eine Aktion in jedem Moment zu starten:

```
every Tick do
    Body
end every;
```

Esterel-Konstrukt: if Anweisung in Bezug auf Signale

- Durch Verwendung der if- Anweisung kann auch die Existenz eines Signals geprüft werden.
- **Syntax**:

```
if Signal-Name then
    Body-1
else
    Body-2
```

- **Semantik**: Bei Start dieser Anweisung wird geprüft, ob das Signal `Signal-Name` verfügbar ist. Ist es verfügbar, so wird der Code von `Body-1` ausgeführt, anderenfalls von Body-2. Innerhalb der Anweisung `if` kann auch entweder der `then Body-1` oder der `else Body-2`-Teil weggelassen werden.

Automaten zur Modellierung von reaktiven Systemen

Automaten im Kontext von reaktiven Systemen

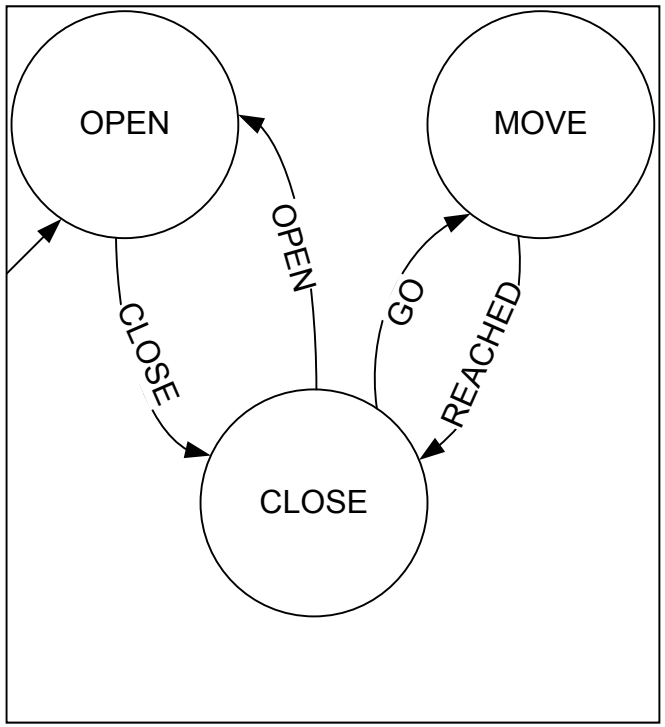
- Durch graphische Darstellung (z.B. Automaten) kann die Verständlichkeit des Codes stark verbessert werden.

- Ein reaktives System kann durch die zyklische Ausführung folgender Schritte beschrieben werden:
  1. Lesen der Eingangssignale
  2. Berechnung der Reaktionen
  3. Auslösen der Ausgangssignale

- Die zyklische Ausführung kann im Automatenmodell wie folgt interpretiert werden:
  1. Lesen der Eingabe im aktuellen Zustand
  2. Berechnen der Zustandsübergangsfunktion und ggfs. Zustandswechsel
  3. Erzeugung von Ausgangssignalen (abhängig von alten Zustand und gelesenem Eingangssignal)

- Die Synchronitätshypothese bedeutet im Bezug auf den Automaten, dass im Vergleich zur Zeit für Änderungen der Umgebung eine vernachlässigbare Zeit für die Berechnung der Zustandsübergange und Ausgabefunktion benötigt wird.

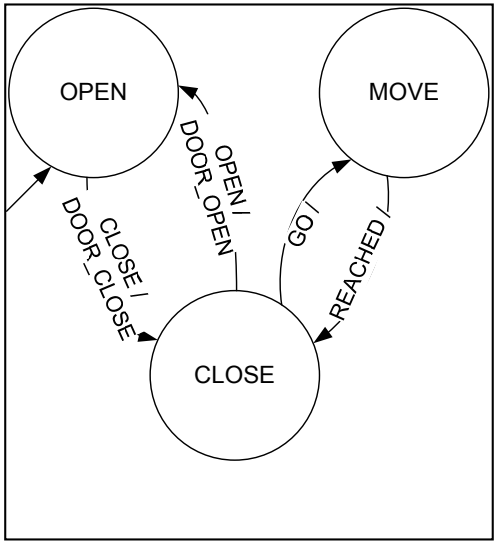
Endliche Automaten

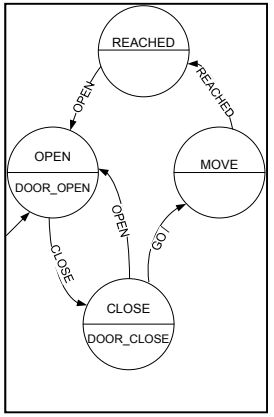


- Ein klassischer endlicher Automat (Q, $\sum$, δ, S, F) ist eine endliche Menge von Zuständen und Zustandsübergängen mit:
  - endlicher Menge von Zuständen Q
  - endliches Eingabealphabet $\sum$
  - Übergangsfunktion δ : Q x $\sum \rightarrow$ Q
  - Endlicher Menge von Startzuständen S
  - Menge von Endzuständen F $\subseteq$ Q
- Um die Verständlichkeit zu verbessern, werden nur deterministische Automaten modelliert, also |S|=1 und $\delta(q,\alpha)=q' \wedge \delta(q,\alpha)=q'' \Rightarrow q'=q''$
- Problem bei der klassischen Definition von Automaten: Ausgaben können nicht modelliert werden.

Automaten mit Ausgaben

- Mealy- und Moore-Automaten unterstützen die Ausgabe von Signalen
- Die Ausgaben von Mealy-Automaten ($Q,\Sigma,\Omega, \delta, \lambda, S,F$) sind dabei an die Übergänge gebunden mit
  - Q, Σ, δ, S, F wie bei klassischem Automat
  - Ausgabealphabet Ω
  - Ausgabefunktion $\lambda: Q \times \Sigma \rightarrow \Omega$
- Bei Moore-Automaten ($Q,\Sigma,\Omega, \delta, \lambda, S,F$) ist die Ausgabe dagegen von den Zuständen abhängig:
  - $Q, \Sigma, \Omega, \delta, S, F$ wie bei Mealy-Automat
  - Ausgabefunktion $\lambda: Q \rightarrow \Omega$
- Moore- und Mealy-Automat sind gleich mächtig: sie können ineinander konvertiert werden.





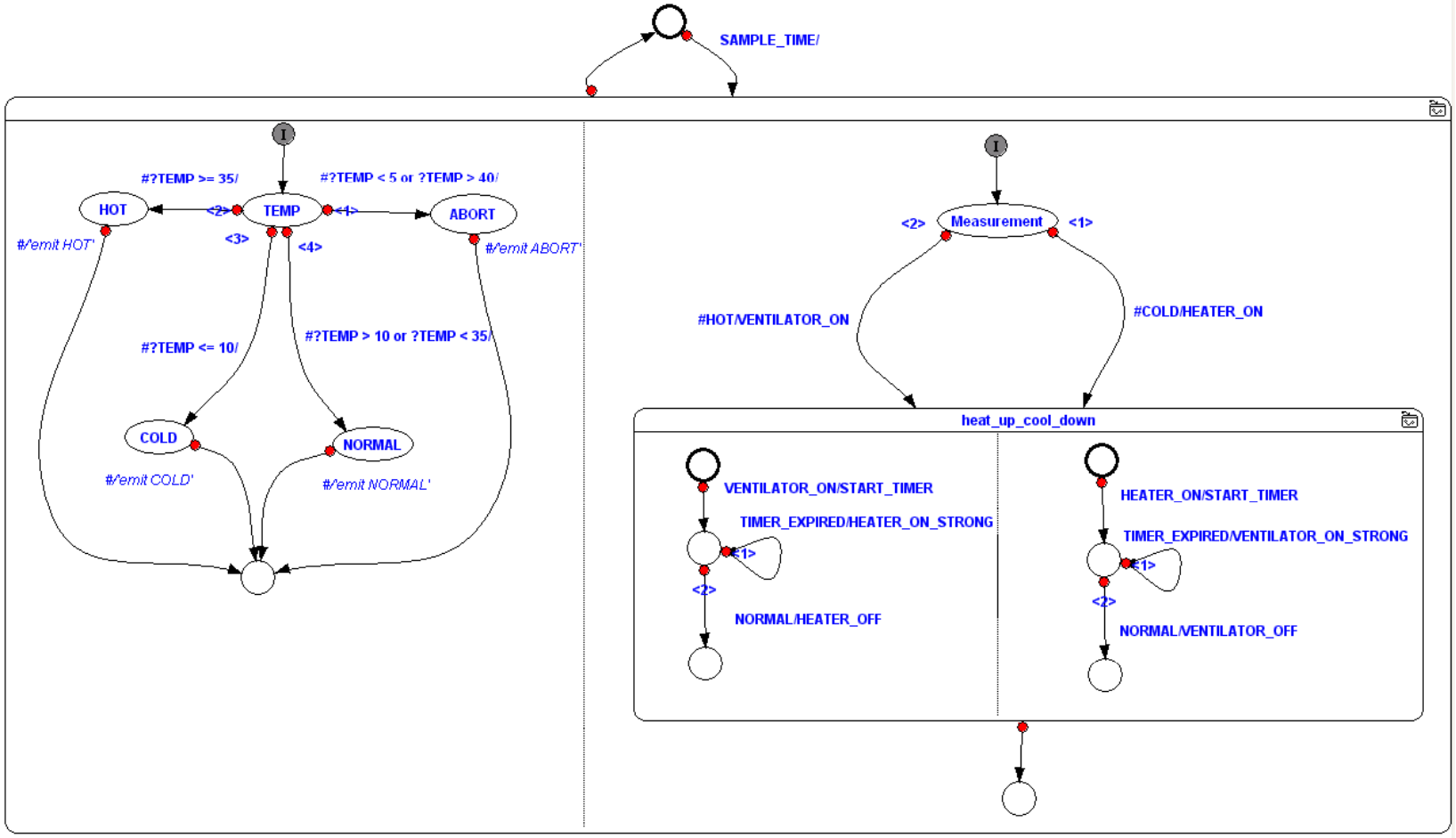
Harel-Automaten / Statecharts

- Heutiger Standard zur Beschreibung von reaktiven Systemen sind die von David Harel 1987 vorgeschlagenen Statecharts.
- Statecharts zeichnen sich durch folgende Eigenschaften aus:
  - Vereinigung der Eigenschaften von Mealy- und Moore-Automaten
    - Ausgaben in Abhängigkeit von Zustandsübergängen möglich
    - Durchführung von Ausgaben beim Erreichen eines Zustands (**`onEntry`**) oder Verlassen eines Zustandes (**`onExit`**)
  - Zur Erhöhung der Lesbarkeit: Hierarchische Strukturierung von Teilautomaten möglich inkl. Gedächtnisfunktionalität (History)
  - Darstellung paralleler Abläufe durch parallele Teilautomaten.
  - Verknüpfung von Zuständen mit Aktionen: Befehle **`do`** (zeitlich begrenzte Aktivität), **`throughout`** (zeitlich unbegrenzte Aktivität)
  - Einführung spontaner bzw. überwachter (guarded) Übergänge

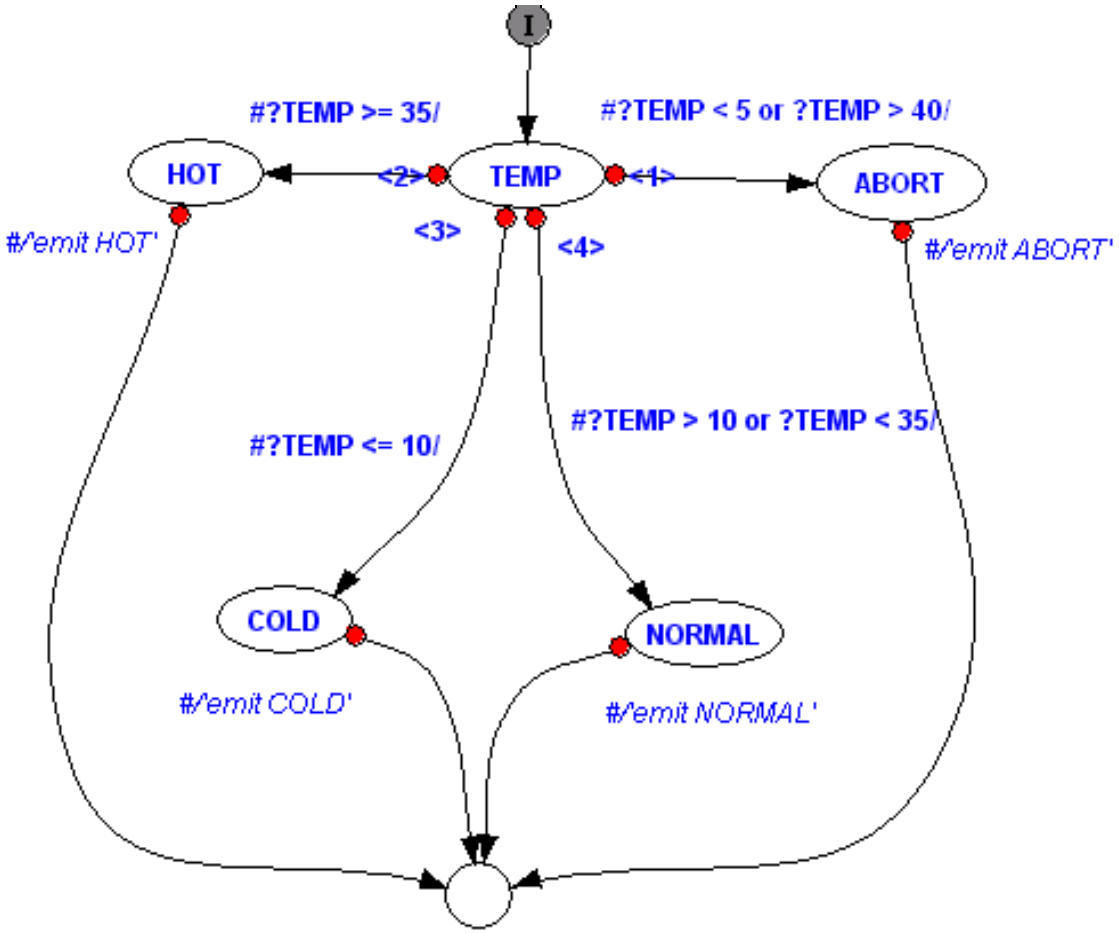
Safe State Machine

- Esterel benutzt eine eigene Klasse von Automaten, die den Statecharts sehr ähnlich sind:
  - Vereinigung der Eigenschaften von Mealy- und Moore-Automaten
    - Ausgaben in Abhängigkeit von Zustandsübergängen möglich
    - Durchführung von Ausgaben beim Erreichen eines Zustands (**`onExit`**) oder Verlassen eines Zustandes (**`onEntry`**)
  - Zur Erhöhung der Lesbarkeit: Hierarchische Strukturierung von Teilautomaten möglich inkl. Gedächtnisfunktionalität
  - Darstellung paralleler Abläufe durch parallele Teilautomaten.
  - ~~Verknüpfung von Zuständen mit Aktionen: Befehle **`do`** (zeitlich begrenzte Aktivität), **`throughout`** (zeitlich unbegrenzte Aktivität)~~
  - Einführung ~~spontaner~~ bzw. überwachter (guarded) Übergänge
  - Zusätzliche Esterel abhängige Konstrukte (z.B. `pre` Operator)

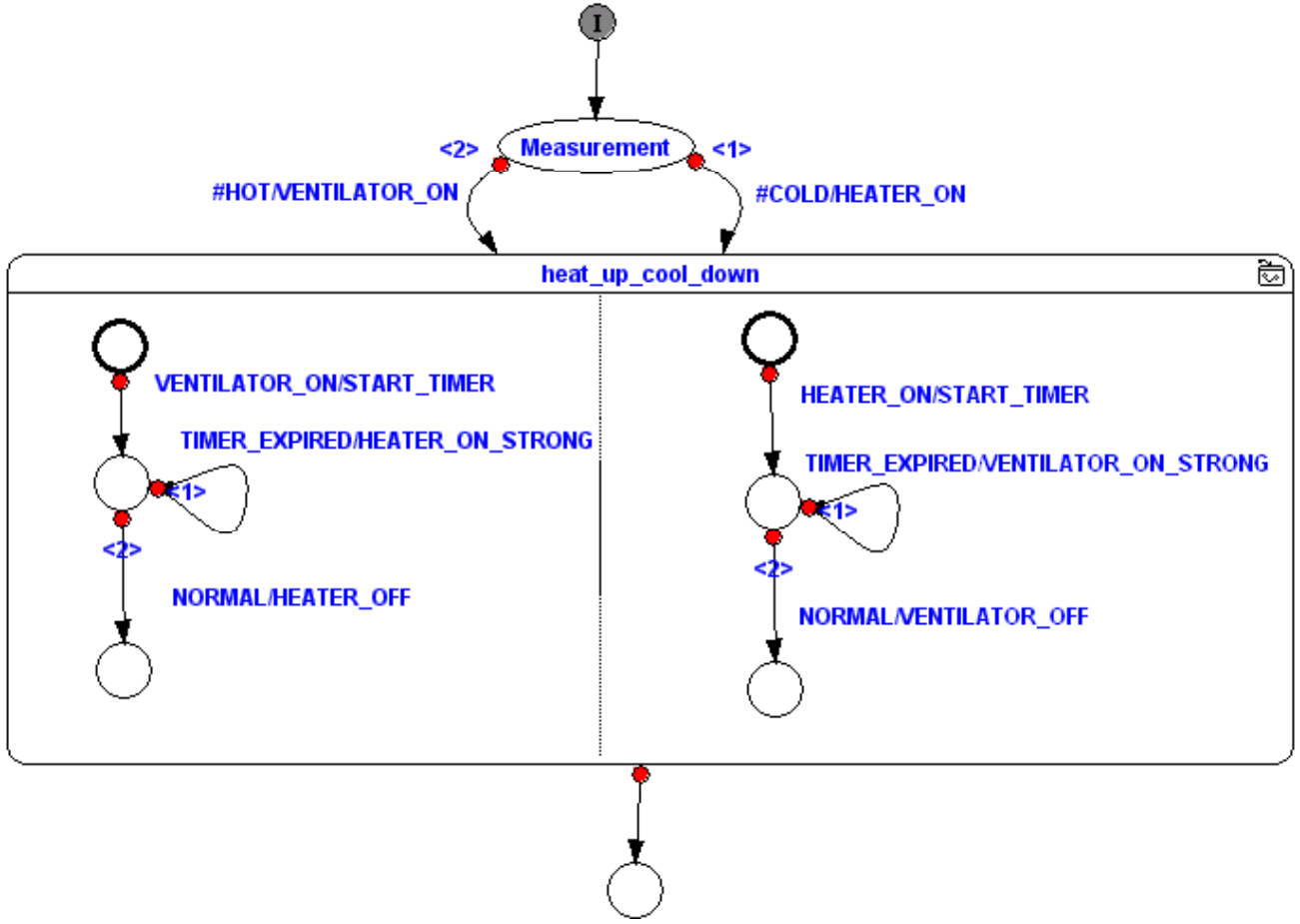
Beispiel als Automat



Beispiel als Automat – Teil 1



Beispiel als Automat – Teil 2



Modellierung von Echtzeitsystemen

Verifikation von Echtzeitsystemen - Einsatz von Formalen Methoden

Problemstellung

„As soon as we started programming, we found to our surprise that it wasn‘t as easy to get programs right as we had thought.
Debugging had to be discovered.
I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.“

Maurice Wilkes.
(Turing Award 1967)

Verifikation & Validierung

- Verifikation: Um die Korrektheit von Programmen in Bezug auf die Spezifikation zu garantieren, wird eine formale Verifikation benutzt. Dazu werden mathematische Korrektheitsbeweise durchgeführt.
- Validierung: Durch eine Validierung kann überprüft werden, dass das System als Modell hinreichend genau nachgebildet wird.
  Techniken:
  - Inspektion
  - Plausibilitätsprüfung
  - Vergleich unabhängig entwickelter Modelle
  - Vergleichsmessung an einem Referenzobjekt

Übersicht über (formale) Methoden

- Deduktive (SW-)Verifikation
  - Beweissysteme, Theorembeweiser
- Model Checking
  - für Systeme mit endlichem Zustandsraum
  - Anforderungsspezifikation mit temporaler Logik
- Testen
  - spielt in der Praxis eine große Rolle
  - sollte systematisch erfolgen $\rightarrow$ ausgereifte Methodik
  - ... stets unvollständig

Verifikation in der Realität

- In der Industrie wird der Begriff Verifikation häufig im Zusammenhang mit nicht formalen Methoden verwendet:
  - Testen, Strategien:
    - 100% Befehlsabdeckung (Statement Coverage)
    - 100% Zweigüberdeckung (Branch Coverage)
    - 100% Pfadüberdeckung (Path Coverage)
    - Siehe auch http://www.software-kompetenz.de/?10764
  - Code-Reviews
  - Verfolgbarkeitsanalysen

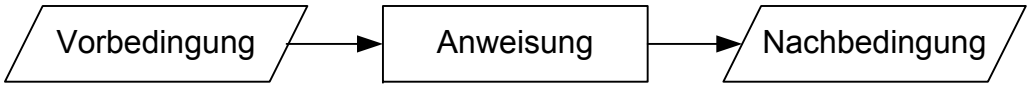
Testen

Mit Testen ist es möglich die Existenz von Fehlern nachzuweisen, nicht jedoch deren Abwesenheit.

- Testen ist von Natur aus unvollständig (non-exhaustive)
- Es werden nur ausgewählte Testfälle / Szenarien getestet, aber niemals alle möglichen.

Deduktive Methoden

- Nachweis der Korrektheit eines Programms durch math.-logisches Schließen
- Anfangsbelegung des Datenraums ⟶ Endbelegung
- Induktionsbeweise, Invarianten
  - klass. Bsp: Prädikatenkalkül von Floyd und Hoare, Betrachten von Einzelanweisungen eines Programms:



- Programmbeweise sind aufwändig, erfordern Experten
- i.A. nur kleine Programme verifizierbar
- Noch nicht vollautomatisch, aber es gibt schon leistungsfähige Werkzeuge

Temporale Logik

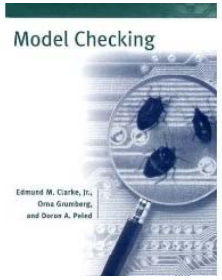
- Mittels Verifikation soll überprüft werden, dass:
  - Fehlerzustände nie erreicht werden
    - Der Aufzug soll nie mit offener Tür fahren.
  - ein System irgendwann einen bestimmten Zustand erreicht (und evtl. dort verbleibt)
    - Nach einer endlichen Initialisierungsphase, geht der Aufzug in den Betriebsmodus über.
  - Zustand x immer nach Eintreten des Zustandes y auftritt.
    - Nach Drücken des Tasters im Stockwerk wird der Aufzug in einem späteren Zustand auch dieses Stockwerk erreichen.
- Um solche Aussagen auch für Rechner lesbar auszudrücken, kann temporale Logik, z.B. in Form von LTL (linear time temporal logic), verwendet werden.
- In LTL werden Zustandsübergänge und damit auch die Zeit als diskrete Folge von Zuständen interpretiert.

Kripke-Struktur

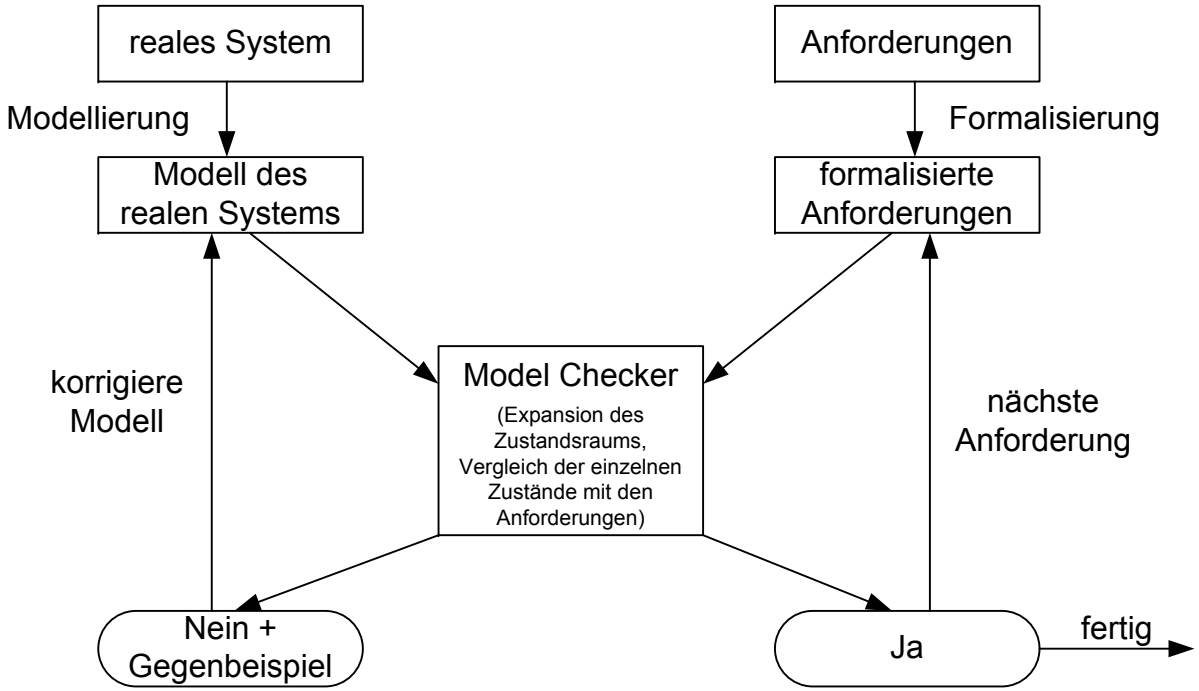
- Zur Darstellung eines Systems werden Kripke-Strukturen K = (V,I,R,B) und eine endliche Menge P von atomaren logischen Aussagen verwendet.
  - V: Menge binärer Variablen (z.B. Tür offen, Aufzug fährt)
  - Die Zustandsmenge S ergibt sich aus allen möglichen Kombinationen über V, somit gilt $S = 2^V$
  - Menge der möglichen Anfangszustände $I \sqsubseteq S$
  - R: Transitionsstruktur $R \sqsubseteq S \times S$
  - B: Bewertungsfunktion S x P $\rightarrow$ {true,false} zur Feststellung, ob ein Zustand eine Eigenschaft auf P erfüllt
- Mittels Model-Checking muss nun nachgewiesen werden, dass eine gewisse Eigenschaft P ausgehend von den Anfangszuständen
  - immer gilt
  - schließlich erfüllt wird
  - …

Explizites Model Checking: Verfahren

- Ausgehend von den Startzuständen exploriert der Model Checker mögliche Nachbarzustände:
  - Auswahl eines noch nicht evaluierten Zustandes
  - Prüfung aller möglichen Zustandsübergange:
    - bereits bekannter Zustand: verwerfen
    - unbekannter Zustand, Eigenschaft prüfen
      - falls Eigenschaft nicht erfüllt, Abbruch und Präsentation eines Gegenbeispiels
      - falls erfüllt, zur Menge der nicht evaluierten Zustände hinzufügen
  - Abbruchbedingung: alle erreichbaren Zustände wurden überprüft
- Problem: Zustandsexplosion
- Literaturhinweis: Edmund M. Clarke, Orna Grumberg, Doron A. Peled, *Model Checking*, 1999, MIT Press



Umgang mit Model Checking



Probleme mit formalen Methoden

- Entwickler empfinden formale Methoden häufig als zu kryptisch
- Beispiel TLA:

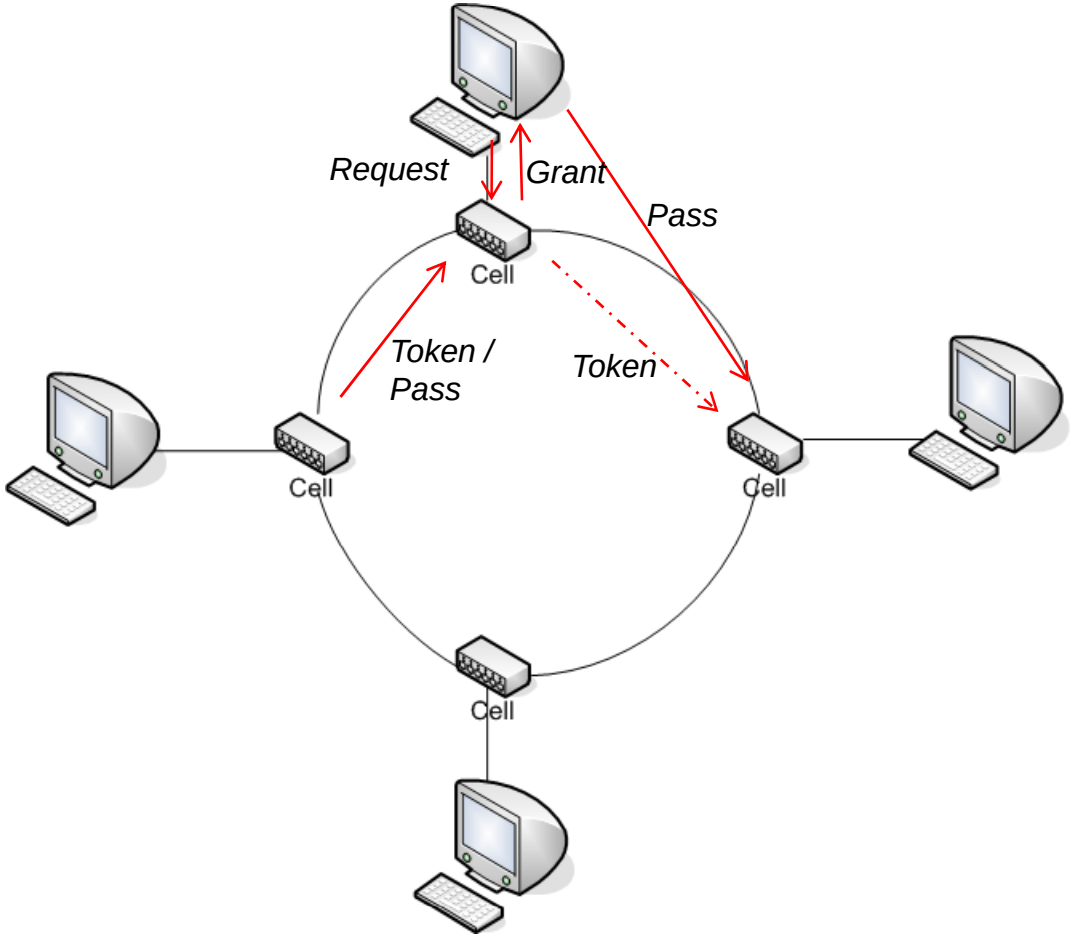
$$
\begin{aligned}
HCini \triangleq &\quad \wedge \quad hr \in \{0, \ldots, 23\} \\
HCnxt \triangleq &\quad \wedge \quad hr' = IF\ hr \neq 23\, THEN\ hr + 1\ ELSE\ 0 \\
HC \triangleq &\quad \wedge \quad HCini \\
&\quad \wedge \quad \Box HCnxt
\end{aligned}
$$

- Neue Ansätze: Erweiterung der Programmier / Modellierungssprachen, automatische Übersetzung

Beispiel: Verifikation von Esterel-Programmen

- Esterel-Verifier `xeve`
  - Einfach: Verifikation der sicheren (Nicht-)Auftretens von Signalen
  - Komplexe Bedingungen: Codierung als Esterel-Programm (Observer), dessen Verhalten mit dem des Ausgangsprogrammes geschnitten wird
- Grundsätzliche Vorgehensweise:
  - Finden von Fehlern in den Annahmen / Modellen mit begrenztem Model Checker
  - Nachweis der Korrektheit des verbesserten Modells in Bezug auf die korrigierten Eigenschaften mit unbegrenztem Model Checking / symbolischen Model Checking
- Esterel Studio
  - Eingebaute Verifikationsfunktionalität zur einfachen Verifikation von Programmen
  - Zur Modellierung der verschiedenen Eigenschaften kann das Schlüsselwort `assert` verwendet werden.
  - Im Verifikationsmodus können die Eigenschaften dann getestet werden, dabei stehen Methoden zum unbegrenzten / in der Testtiefe begrenzten Modell Checking, sowie zum symbolischen Model Checking zur Verfügung.
- Details siehe Demonstration: Bus-Arbiter (`xeve`)

Bus-Arbiter Beispiel



Verifikation ist immer nur auf Modellebene

```
int16_t number;
Signal e,end;
```

```
float share;
```

Prozess 1:

```
number:=1;
while (true)
{
    wait(e);
    number++;
}
```

Prozess 2:

```
while (true)
{
    wait(end);
    share:=1/number;
    print(share);
    number:=1;
}
```

Was kann hier schiefgehen (unabhängig von inhaltlicher Korrektheit)?
Problem Überlauf: Annahme Ereignisse zwischen zwei Endsignalen begrenzt (<1000)?
Problem: Atomare Erhöhung, korrekt auf 16-Bit Controller, falsch auf 8-Bit Controller.

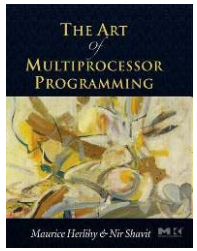
Kapitel 3

Nebenläufigkeit

Inhalt

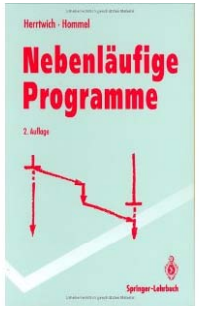
- Motivation
- Unterbrechungen (Interrupts)
- (Software-) Prozesse
- Threads
- Interprozesskommunikation (IPC)

Literatur



Maurice Herlihy, Nir Shavit,
The Art of Multiprocessor
Programming, 2008

A.S.Tanenbaum, Moderne
Betriebssysteme, 2002



R.G.Herrtwich, G.Hommel,
Nebenläufige Programme
1998

- Edward Lee: The Problem with Threads:
  http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- http://www.beyondlogic.org/interrupts/interupt.htm
- http://www.llnl.gov/computing/tutorials/pthreads/

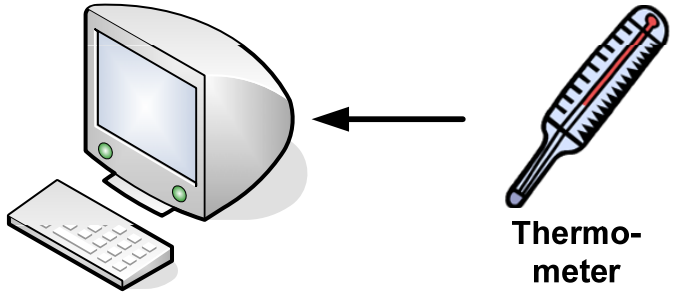
Definition von Nebenläufigkeit

- **Allgemeine Bedeutung**: Nebenläufige Ereignisse sind nicht kausal abhängig. Ereignisse (bzw. Ereignisfolgen) sind dann nebenläufig, wenn keines eine Ursache im anderen hat.
- **Bedeutung in der Informatik**: Nebenläufig bezeichnet hier die Eigenschaft von Programmcodes, nicht linear hintereinander ausgeführt werden zu müssen, sondern zeitlich parallel zueinander ausführbar zu sein.
- Aktionen (Programmschritte) können parallel (gleichzeitig oder quasi gleichzeitig) ausgeführt werden, wenn keine das Resultat der anderen benötigt. Die parallele Ausführung von mehreren unabhängigen *Prozessen* (siehe später) auf einem oder mehreren Prozessoren bezeichnet man als *Multitasking*. Die parallele Ausführung von Teilsequenzen innerhalb eines Prozesses heißt *Multithreading*.

Motivation

- Gründe für nebenläufige Ausführung von Programmen in Echtzeitsystemen:
  - Echtzeitsysteme sind häufig verteilte Systeme (Systeme mit mehrere Prozessoren).
  - Zumeist werden zeitkritische und zeitunkritische Aufgaben parallel berechnet.
  - Bei reaktiven Systemen ist die maximale Antwortzeit häufig limitiert.
  - Abbildung der parallelen Abläufe im technischen Prozeß
- Aber: kleinere (Monoprozessor-)Echtzeit-Systeme verzichten häufig auf die parallele Ausführung von Code, weil der Aufwand für die Prozeßverwaltung zu hoch ist.
  Dennoch auch hier: typischerweise Parallelverarbeitung in „Hauptprogramm“ und „Unterbrechungsbehandler“ (interrupt service routine, interrupt handler)

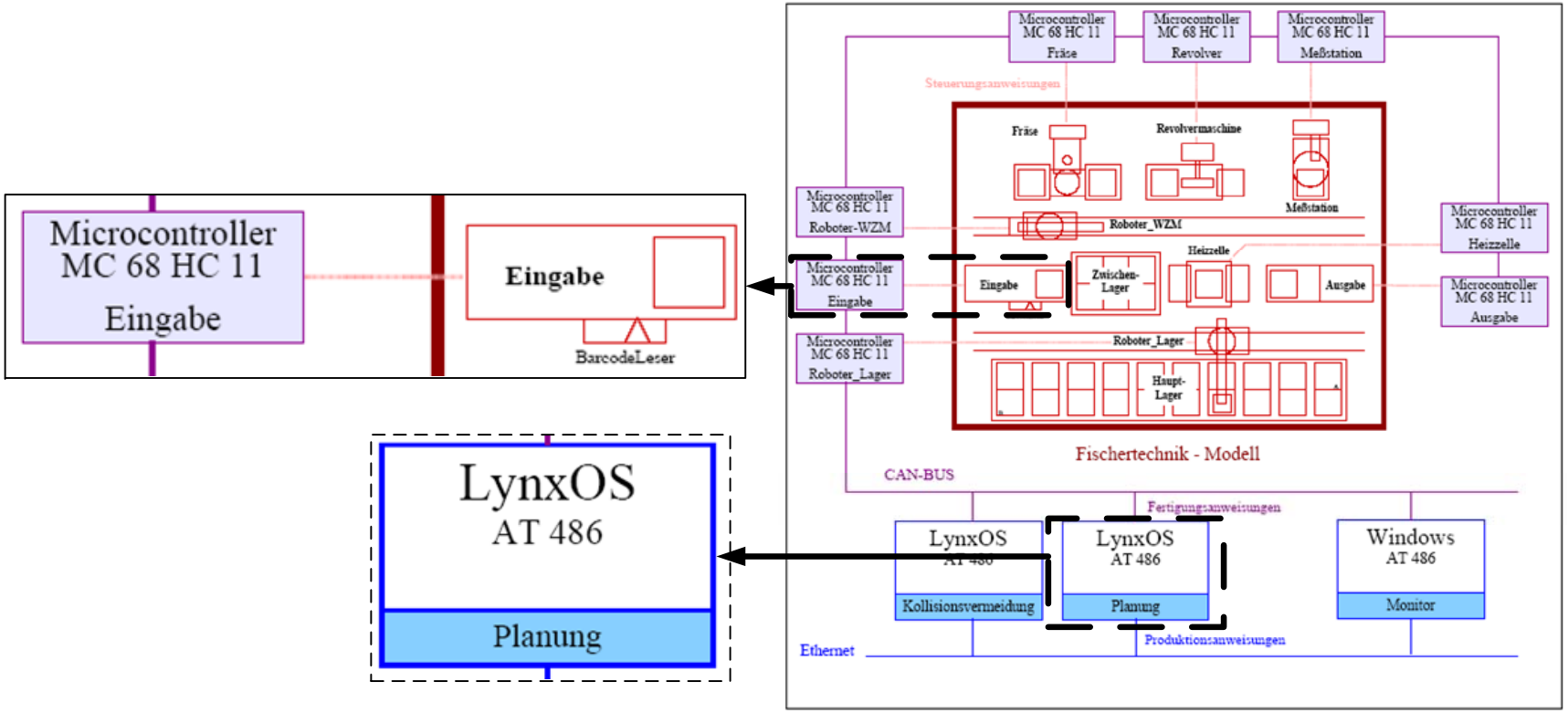
Anwendungsfälle für Nebenläufigkeit (Unterbrechungen)



Signal falls Temperaturwert überschritten wird
⇨ **Unterbrechungen (interrupts)**

Allgemeines Anwendungsgebiet: hauptsächlich zur Anbindung von externer Hardware

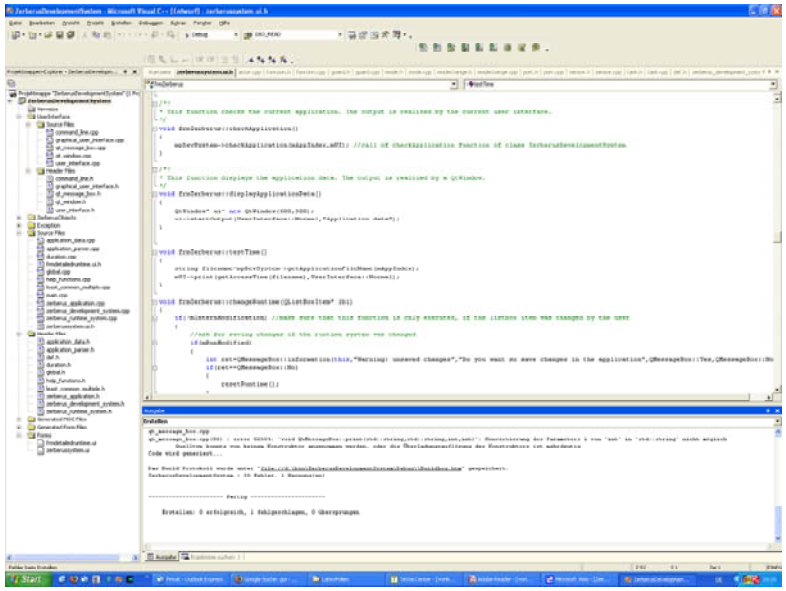
Anwendungsfälle für Nebenläufigkeit (Prozesse)



Verteiltes System zur Steuerung der Industrieanlage ⇨ **Prozesse (tasks)**

Allgemeine Anwendungsgebiete: verteilte Systeme, unterschiedlichen Anwendungen auf einem Prozessor

Anwendungsfälle für Nebenläufigkeit (Threads)



Reaktion auf Nutzereingaben trotz Berechnungen (z.B. Übersetzen eines Programms)

⇨ **leichtgewichtige Prozesse (Threads)**

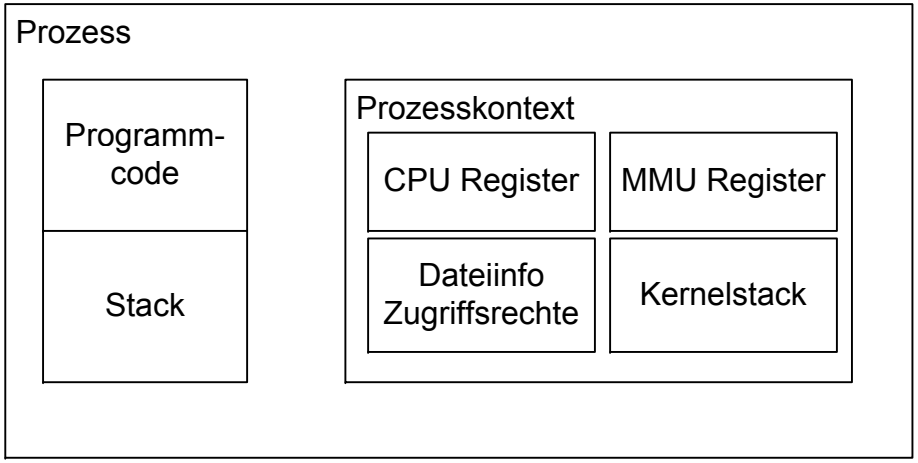
Allgemeines Anwendungsgebiet: unterschiedliche Berechnungen im gleichen Anwendungskontext

Nebenläufigkeit

Prozesse

Definition

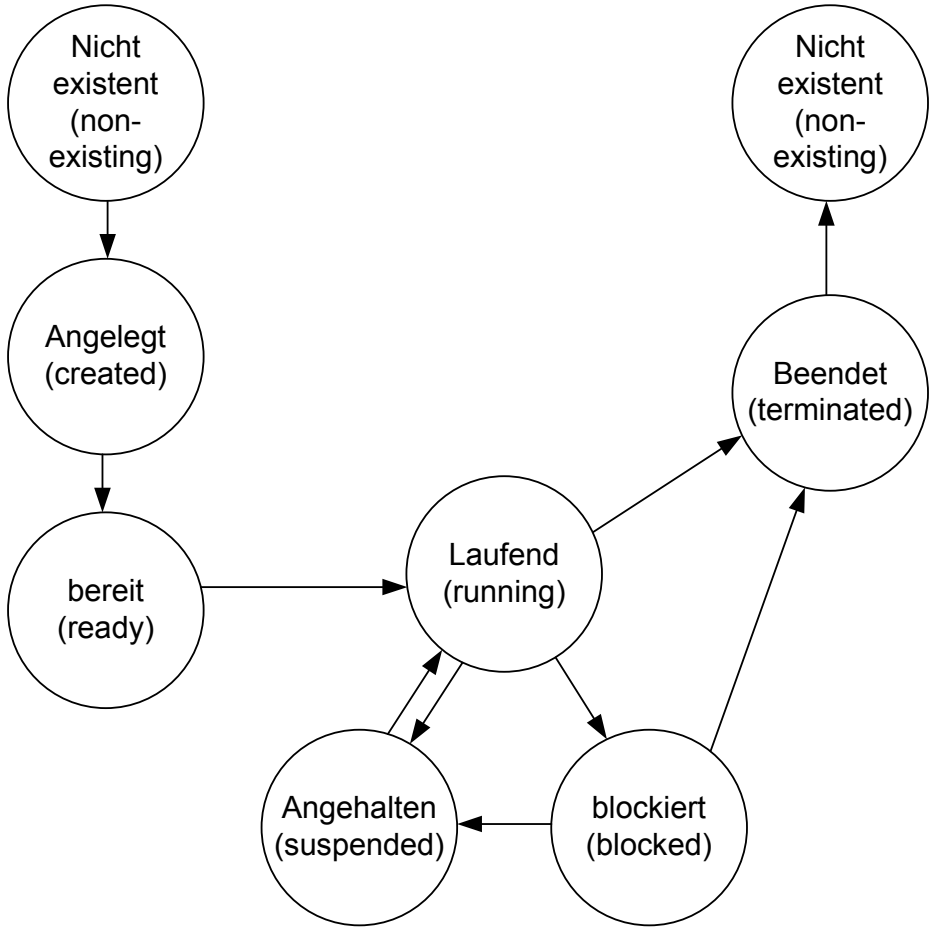
- **Prozess**: Abstraktion eines sich in Ausführung befindlichen Programms
- Die gesamte Zustandsinformation der Betriebsmittel für ein Programm wird als eine Einheit angesehen und als Prozess bezeichnet.
- Prozesse können weitere Prozesse erzeugen ⇨ Vater-,Kinderprozesse.



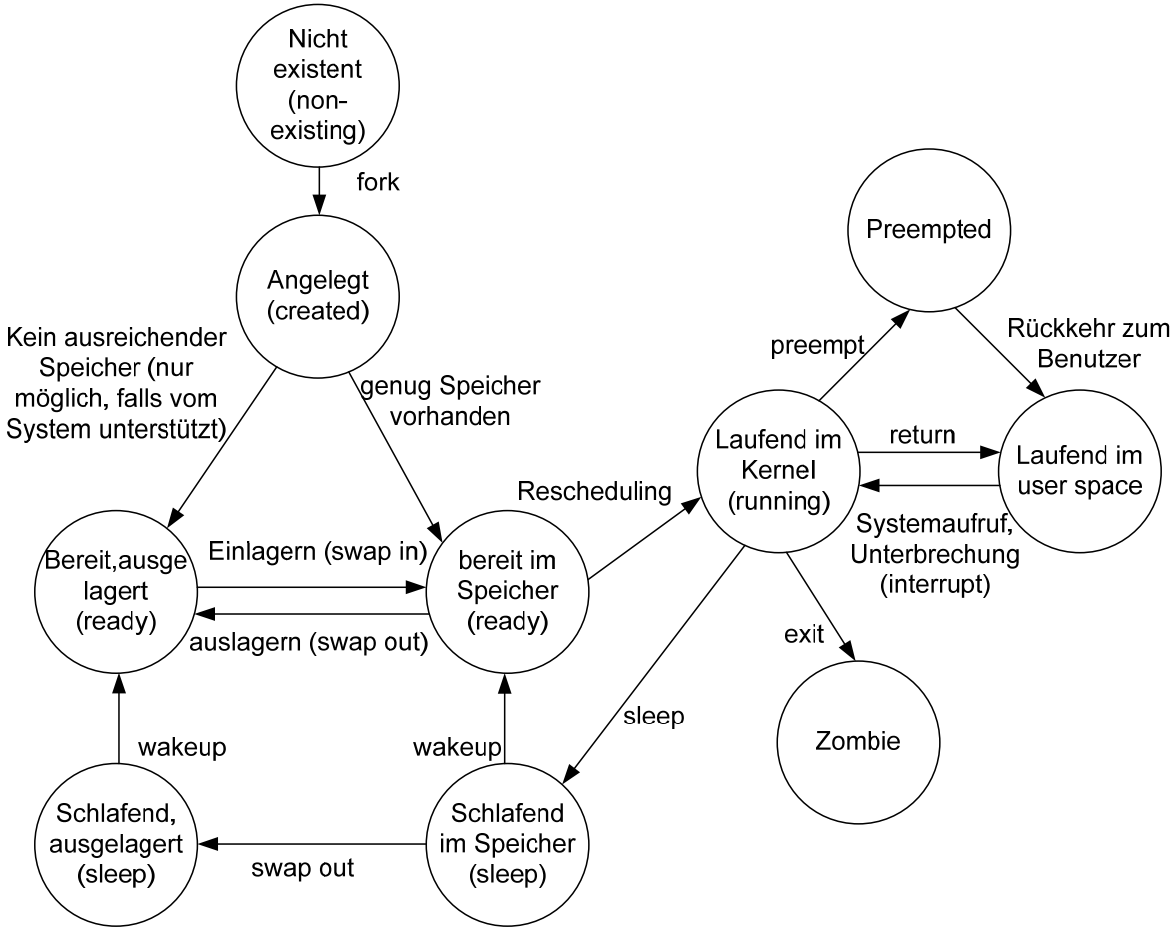
Prozessausführung

- Zur Prozessausführung werden diverse Resourcen benötigt, u.a.:
  - Prozessorzeit
  - Speicher
  - sonstige Betriebsmittel (z.B. spezielle Hardware)
- Die Ausführungszeit ist neben dem Programm abhängig von:
  - Leistungsfähigkeit des Prozessors
  - Verfügbarkeit der Betriebsmittel
  - Eingabeparametern
  - Verzögerungen durch andere (wichtigere) Aufgaben

Prozesszustände (allgemein)



Prozeßzustände in Unix



Fragen bei der Implementierung

- Welche Betriebsmittel sind notwendig?
- Welche Ausführungszeiten besitzen einzelne Prozesse?
- Wie können Prozesse kommunizieren?
- Wann soll welcher Prozess ausgeführt werden?
- Wie können Prozesse synchronisiert werden?

Klassifikation von Prozessen

- periodisch vs. aperiodisch
- statisch vs. dynamisch
- Wichtigkeit der Prozesse (kritisch, notwendig, nicht notwendig)
- speicherresident vs. verdrängbar
- Prozesse können auf
  - einem Rechner (Pseudoparallelismus)
  - einem Multiprozessorsystem mit Zugriff auf gemeinsamen Speicher
  - oder auf einem Multiprozessorsystem ohne gemeinsamen Speicher

  ausgeführt werden.

Nebenläufigkeit

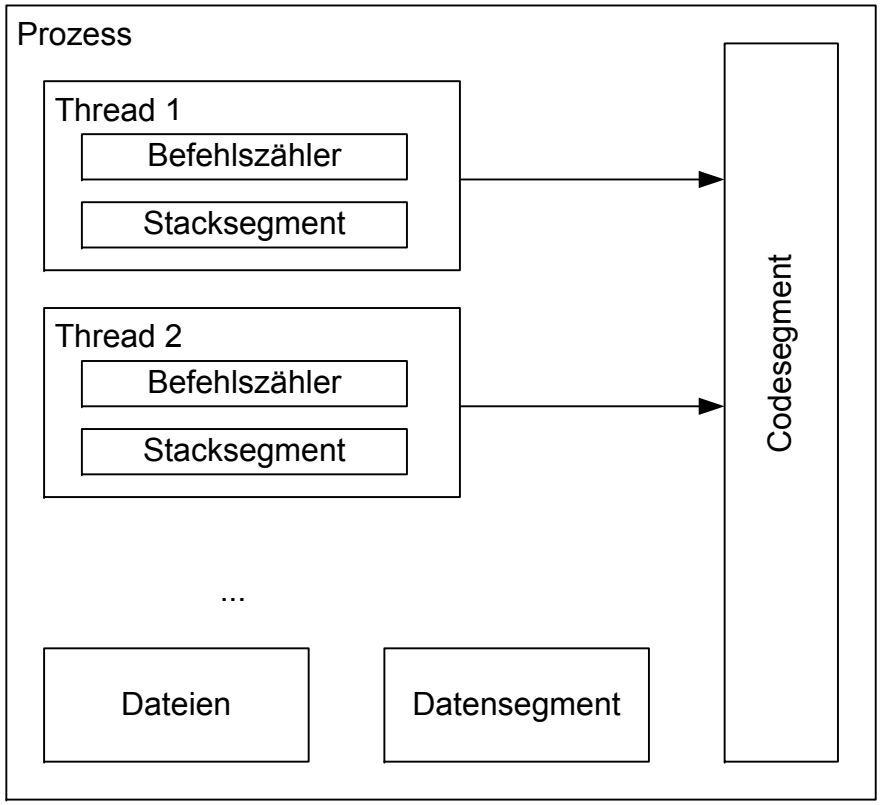
Threads

Leichtgewichtige Prozesse (Threads)

- Der Speicherbedarf von Prozessen ist in der Regel groß (CPU-Daten, Statusinformationen, Angaben zu Dateien und EA-Geräten...).
- Bei Prozesswechsel müssen die Prozessdaten ausgetauscht werden $\Rightarrow$ hohe Systemlast, zeitaufwendig.
- Viele Systeme erfordern keine komplett neuen Prozesse.
- Vielmehr sind Programmabläufe nötig, die auf den gleichen Prozessdaten arbeiten.

⇨ Einführung von Threads

Threads


Prozess
Thread 1
Befehlszähler
Stacksegment
Thread 2
Befehlszähler
Stacksegment
...
Dateien
Datensegment
Codesegment

Prozesse vs. Threads

- Verwaltungsaufwand von Threads ist deutlich geringer
- Effizienzvorteil: bei einem Wechsel von Threads im gleichen Prozessraum ist kein vollständiger Austausch des Prozesskontextes notwendig.
- Kommunikation zwischen Threads des gleichen Prozesses kann über gemeinsamen Speicher erfolgen.
- Zugriffe auf den Speicherbereich anderer Prozesse führen zu Fehlern.
- Probleme bei Threads: durch die gemeinsame Nutzung von Daten kann es zu Konflikten kommen.

Nebenläufigkeit

Unterbrechungen

Binding Rechnersystem-Umwelt

- Es muss ein Mechanismus gefunden werden, der es erlaubt, Änderungen der Umgebung (z.B. Druck einer Taste) zu registrieren.
- **1. Ansatz**: Abfrage (Polling)
  Es werden die E/A-Register reihum nach Änderungen abgefragt und bei Änderungen spezielle Antwortprogramme ausgeführt.
  - Vorteile:
    - bei wenigen EA-Registern sehr kurze Latenzzeiten
    - bei einer unerwarteten Ereignisflut wird das Zeitverhalten des Programms nicht übermäßig beeinflusst
    - Kommunikation erfolgt synchron mit der Programmausführung
  - Nachteile:
    - die meisten Anfragen sind unnötig
    - hohe Prozessorbelastung
    - Reaktionszeit steigt mit der Anzahl an Ereignisquellen

Lösung: Einführung des Begriffs der Unterbrechung

- **2. Ansatz**: Unterbrechung (Interrupt)
- Eine Unterbrechung stoppt die Verarbeitung des laufenden Programms. Die Wichtigkeit des Ereignisses, welches die Unterbrechung ausgelöst hat, wird überprüft. Darauf basierend erfolgt die Entscheidung, welche Reaktion erfolgt.
- Vorteile:
  - Prozessorressourcen werden nur dann beansprucht, wenn es nötig ist
- Nachteile:
  - Nicht-Determinismus: Unterbrechungen asynchron zum Programmlauf (und zum Prozessorzustand) eintreffen.

Unterbrechungen

- **Unterbrechungen**: Stopp des Hauptprogrammablaufs, Aufnahme der Programmausführung eines „Unterbrechungsbehandlers (UBB)“ an einer anderen Stelle; nach Beendigung des UBB (zumeist) Rückkehr an die Stelle des Auftritts der Unterbrechung im Hauptprogramm.
- **Synchrone** Unterbrechungen: treten, falls sie auftreten, immer an *derselben Stelle* im Programmcode auf. Man bezeichnet sie auch als *Traps* oder *Exceptions* bzw. „Software-Interrupts“
- **Asynchrone** Unterbrechungen: Auftrittszeitpunkt ist unbestimmt; es kann nicht gesagt werden, an welcher Stelle der Hauptprogrammausführung der Prozessor zum Zeitpunkt der Unterbrechung ist. Asynchrone Unter-brechungen werden auch als Interrupts bezeichnet; weil sie von der Hardware-Peripherie erzeugt werden, auch als *Hardware-Interrupts*. Sie üben „Brückenfunktion“ zwischen Hardware und Software aus.

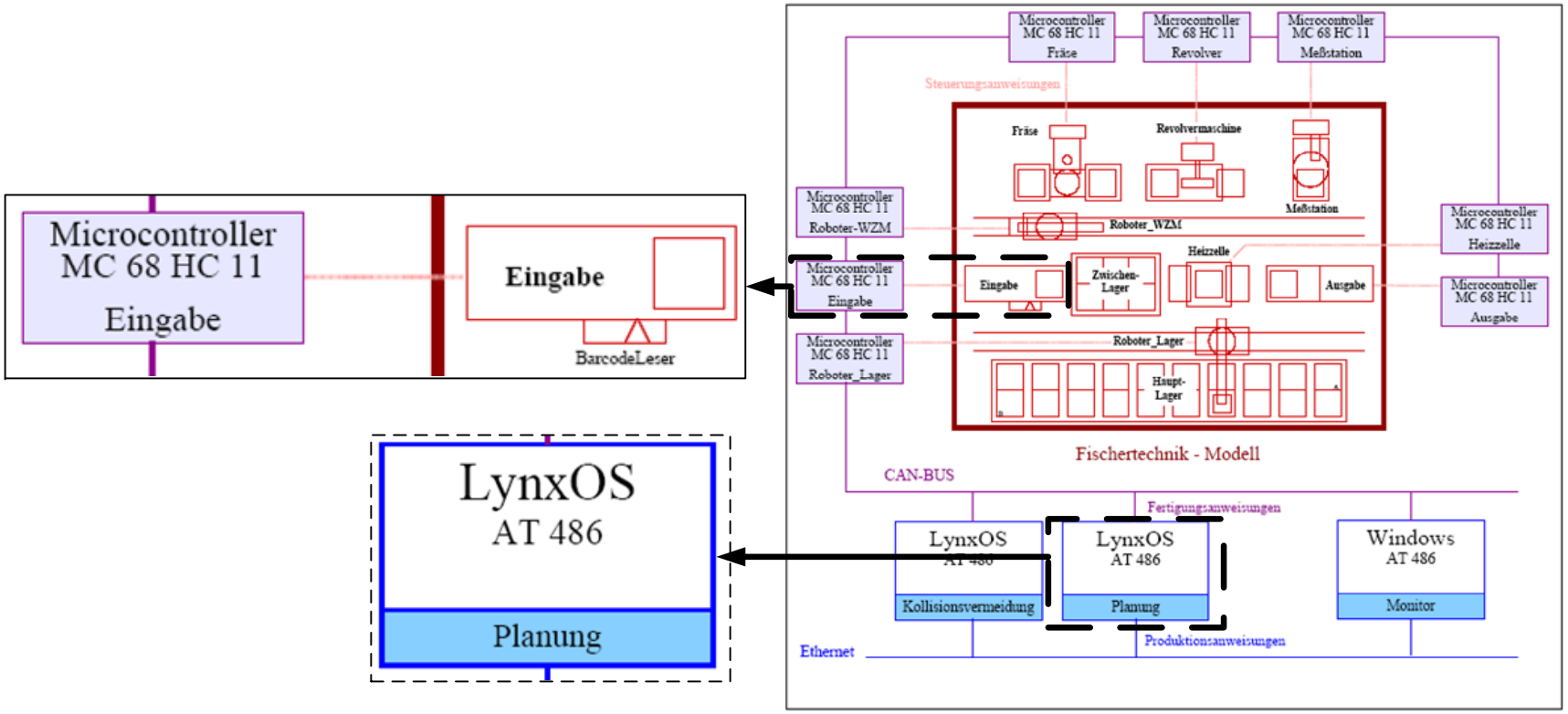
Synchrone Unterbrechungen (Traps/Exceptions)

- Werden durch das Programm selbst ausgelöst, d.h. dasselbe Programm, ausgeführt mit denselben Parametern wird in der Regel an derselben Stelle dieselbe Unterbrechung auslösen (vorhersagbar in dieselbe „Falle“ laufen)
- **Auslösung bei Fehler** – Ausnahme/Exception, Beispiele:
  - Arithmetikfehler (Division by zero, overflow, not-a-number NaN, …)
  - Speicherfehler (Page Fault, segment Fault, memory full, …)
  - Befehlsfehler (Illegal instruction, privileged instruction, bus error, …)
  - Peripheriefehler (End-of-file EOF, channel blocked, unknown device, …)
- Bei Exceptions **nur dann** Rückkehr an den Auftrittspunkt, wenn die Fehlerbedingung im Ausnahmebehandler beseitigt werden kann, andernfalls Abbruch (resumption vs. termination)
- **Auslösung durch spezifische Instruktion**: Breakpoint, SWI, TRAP, INT, … entweder zum Zwecke des „Debuggings“ oder zum Aufruf von Betriebssystem-Diensten (z.B. MS-DOS „INT 21h“, siehe z.B. http://en.wikipedia.org/wiki/MS-DOS_API)
- Traps können auch benutzt werden, um einen Hardware-Interrupthandler zu testen.

Asynchrone Unterbrechungen (Interrupts)

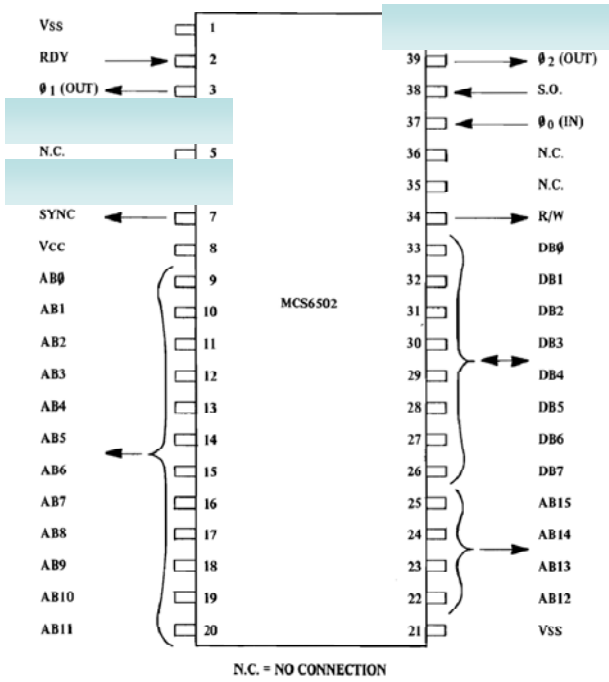
- Werden durch externe Prozesse ausgelöst, d.h. sind bezüglich des genau-en Auftrittszeitpunkts unvorhersagbar und zumeist nicht reproduzierbar
- Beispiele:
  - Signalisierung „normaler“ externer Ereignisse durch periphere Einheiten (Timer, Schalter, Grenzwertüber-/unterschreitung, …)
  - Warnsignale der Hardware (Energiemangel, „Watchdog-timer“ abgelaufen, …)
  - Beendigung einer Ein-/Ausgabeoperation (Wort von serieller Schnittstelle komplett empfangen oder komplett gesendet, Operation von Coprozessor (DMA, FPU) komplett, …)
- Die Unterbrechungsbehandlung muß *nebeneffektfrei* verlaufen, d.h. das Hauptprogramm darf nach Abschluß der Behandlung keinen veränderten Ausführungs-Kontext vorfinden.
- Aber: typischerweise wird der Behandler zum Zwecke der Kommunikation über globale Variable mit dem Hauptprogramm kommunizieren.

Beispiel: Modellfabrik (Praktikum) mit Prozessoren und INT-Quellen



Technische Realisierung von Interrupts

- Zur Realisierung besitzen Prozessoren einen oder mehrere spezielle Interrupt-Eingänge (typ. IRQ oder INT-Anschluß). Wird ein Interrupt aktiviert, so führt dies zur Ausführung einer Unterbrechungs-behandlungsroutine.
- Das Auslösen der Unterbrechungsroutine entspricht einem Unterprogrammaufruf. Der Programmablauf wird an einer anderen Stelle fortgesetzt und nach Beendigung der Routine (normalerweise) an der unterbrochenen Stelle fortgefahren.



Pin-Belegung des MOS 6502-Prozessors (z.B. Commodore 64).
Quelle: MCS 6500 HW-Manual, MOS Technology, Jan. 1976

Durchführung einer einfachen INT-Behandlung

1. IRQ-Anschluß wird durch peripheres Gerät aktiviert
2. Wenn Interrupts momentan zugelassen sind: Beendigung der Abarbeitung der gerade noch laufenden Instruktion
3. Sicherung der Register des Prozessors (Prozessorkontext) auf dem Stapelspeicher (Stack) – insbesondere Sicherung des Programmzählers; dafür spezielle Instruktionen verfügbar
4. Sprung an den Behandler (entspricht Laden des Programmzählers mit der Programmstartadresse des Behandlers) – dies kann auf verschiedene Arten erfolgen, siehe unten
5. Ausführung des Codes des Behandlers, an dessen Ende steht ein „Return from Interrupt“-Befehl; dabei Signalisierung an Peripherie, daß Behandlung abgeschlossen
6. Zurückladen des gesicherten Prozessorregistersatzes, Rücksprung
   (= Laden des Programmzählers mit der Adresse der Instruktion, die auf diejenige folgt, an der die Unterbrechung auftrat).

Beispiel INT beim 6502

- Bei Vorliegen eines IRQ wird der 16-Bit Programmzähler mit dem Inhalt der 8-Bit Adressen FFFE und FFFF geladen.
- FFFE/F enthält Adresse des (= *Vektor* auf) IRQ-Behandler
- FFFC/D enthält Adresse des Reset-Behandlers
- FFFA/B enthält Adresse des NMI-Behandlers. NMI: Non-Maskable-Interrupt, kann nicht abgeschaltet werden (also auch nicht durch fehlerhaftes Programm)

Speicher-Aufteilung für 6502 (in Hex-Adressen)

0000-00FF - RAM for Zero-Page & Indirect-Memory Addressing

0100-01FF - RAM for Stack Space & Absolute Addressing

0200-3FFF - RAM for programmer use

4000-7FFF - Memory mapped I/O

8000-FFF9 - ROM for programmer usage

FFFA - Vector address for NMI (low byte)

FFFB - Vector address for NMI (high byte)

FFFC - Vector address for RESET (low byte)

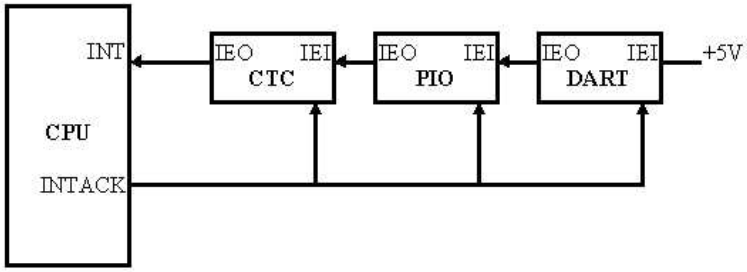
FFFD - Vector address for RESET (high byte)

FFFE - Vector address for IRQ & BRK (low byte)

FFFF - Vector address for IRQ & BRK (high byte)

Behandlung mehrerer Quellen von Interrupts

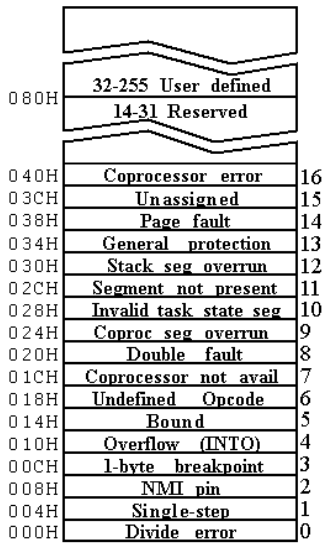
- Beim 6502 nur *ein* IRQ-Eingang und nur *ein* Vektor auf *einen* Behandler. Bei mehreren Quellen für Interrupts (z.B. serielle Schnittstelle, Parallele Schnittstelle, Timer) werden daher die Interrupt-Ausgänge dieser Einheiten „verodert“ und an den IRQ-Eingang gelegt. Alle Quellen sind damit gleichberechtigt.
- Damit ist keine automatische HW-Priorisierung (nach Wichtigkeit der eintreffenden Interrupts) möglich.
- Der Behandler muß nach dem Auftritt des Interrupts Quelle für Quelle abfragen, welche den Interrupt verursacht hat (implizite Priorisierung je nach Abfragereihenfolge)
- Einfache Möglichkeit der HW-Priorisierung: Daisy-Chain (Prioritätskette). Die Einheit, die am nächsten an der CPU liegt, hat die höchste Priorität (siehe Bild rechts) und sperrt Interrupts anderer Quellen solange aus, bis der eigene Interrupt abgearbeitet ist (INTACK durch Prozessor signalisiert).



Daisy-Chaining beim Z80-Prozessor
CTC: Counter-Timer-Circuit
PIO: Parallel In-/Out
DART: Dual Asynchronous Receiver Transmitter

Vektorisierte Interrupt-Behandlung

- Verfügt ein Rechnersystem über viele Interrupt-Quellen (wie typischerweise im Bereich eingebetteter Systeme), ist es zweckmäßig, für jedes Gerät (mindestens) einen Behandler vorzusehen und diesen nach dem Ereignisauftritt auch direkt ausführen zu können.
- Dazu Einführung von vektorbasierten Interrupt-Systemen.
- Prinzip: Gerät erzeugt nicht nur einen Interrupt, sondern liefert dem Prozessor parallel auch eine eigene Kennung (z.B. 8-Bit Wert A). Dieser Wert A verweist auf einen Tabelleneintrag, an dem sich die Einsprung-Adresse des Behandlers befindet.
- Exceptions und Interrupts folgen dem gleichen Schema, d.h. Tabelleneintrag = Einsprungadresse für Exception/Interrupt-Handler



The interrupt vector table is located in the first 1024 bytes of memory at addresses 000000H through 0003FFH.

There are 256 4-byte entries (segment and offset in real mode).

Seg high	Seg low	Offset high	Offset low
Byte 3	Byte 2	Byte 1	Byte 0

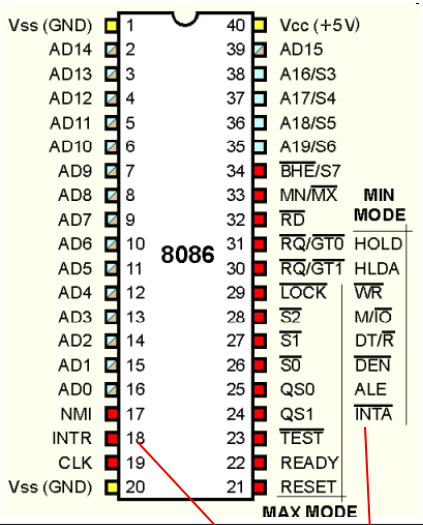
- Typische Interrupt-Vektor-Tabelle eines 8086-Systems im „Real Mode“
- 32-Bit-Adresse (4 Byte) pro Eintrag
- 0 bis 31 sind Prozessor-interne Ausnahmen
- Hardware-Interrupts können über Interrupt-Controller auf beliebige Tabellenplätze gelegt werden

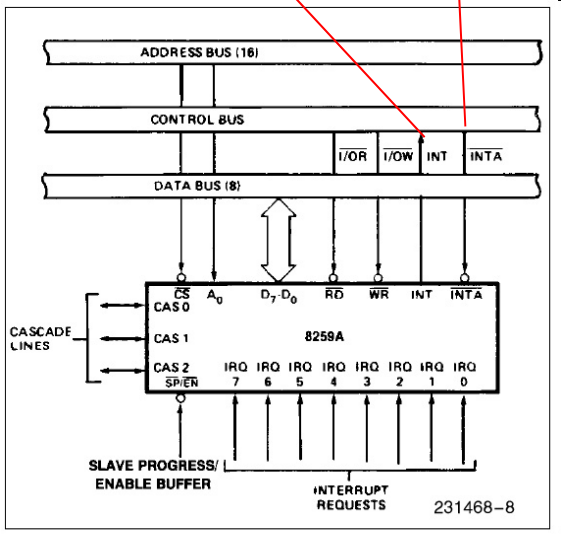
Interrupt-(Priority)-Controller

- Typischerweise wird mit der vektorbasierten Verwaltung auch eine Prioritätsverwaltung eingeführt. Klassisches Beispiel: der Interrupt-Controller 8259A des IBM-PC.
- Verwaltet 8 Hardware-Interrupts und kann diese mit Prioritäten belegen. Ist kaskadierbar, der PC/AT hatte zwei 8259A.
- Typen der Interrupts beim PC/AT:

00	Systemtaktgeber	08	Echtzeitsystemuhr
01	Tastatur	09	Frei
02	Programmierbarer Interrupt-Controller	10	Frei
03	Serielle Schnittstelle COM2 (E/A-Bereich 02F8)	11	Frei
04	Serielle Schnittstelle COM1 (E/A-Bereich 03F8)	12	PS/2-Mausanschluss
05	Frei, oft Soundkarte oder LPT2	13	Koprozessor (ob separat oder in CPU integriert)
06	Diskettenlaufwerk	14	Primärer IDE-Kanal
07	Parallel (Drucker-) Schnittstelle LPT1 (E/A-Bereich 0378)	15	Sekundärer IDE-Kanal

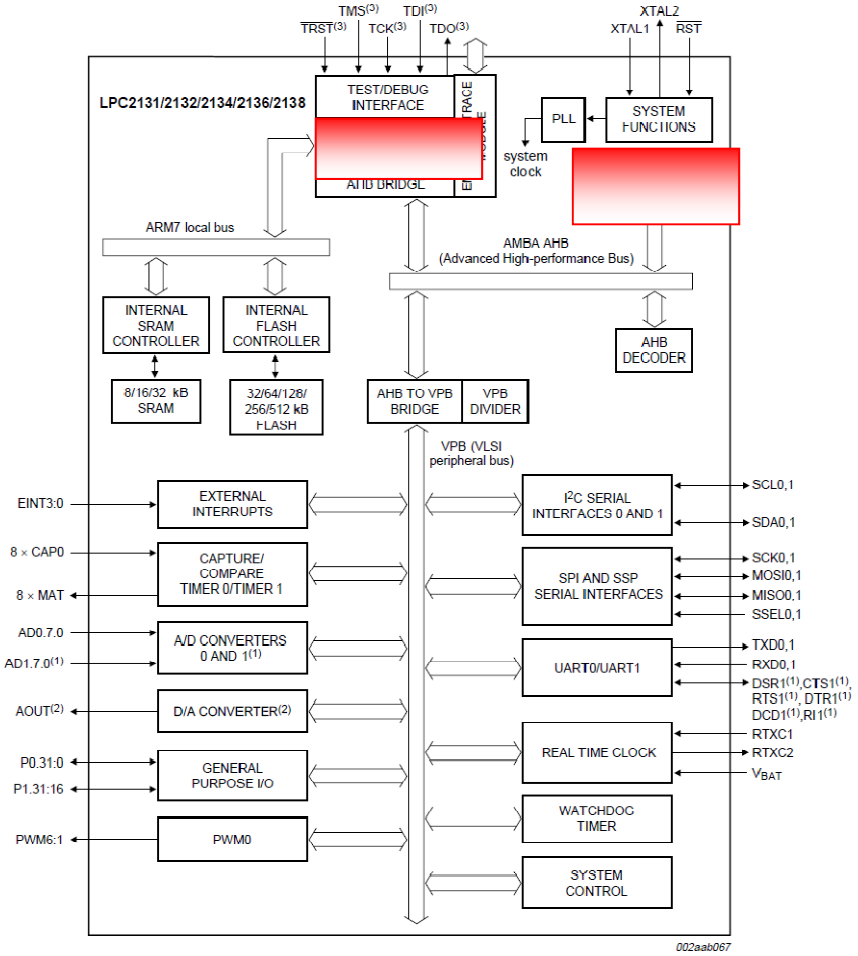
- Heute Standard (seit ca. 2002): Nachfolgekonzept, Intel APCI 82093AA, sehr viel höhere Anzahl von Interrupts realisiert





Interrupts an einem Micro-Controller

- Häufig eingesetztes System der 32-Bit-Klasse: ARM (Acorn Risc Machine)
- Hier betrachtet LPC 2138 von NXP, 32-Bit ein-Chip Controller mit Echtzeit-Uhr (RTC), UART, I2C-Bus, USB, A/D, D/A, etc.
- Alle peripheren einheiten auf dem Chip können Interrupts auslösen
- Alle Interrupts sind priorisierbar
- Dazu ist ein VIC (Vectored Interrupt Controller) vorgesehen mit 32 Interrupts, davon 16 vektorisiert



Acorn founders
Hermann Hauser and Chris Curry
(http://atterer.net/acorn.html)

LPC-Interrupt-Controller VIC: Register

Table 33: VIC register map

Name	Description	Access	Reset value[1]	Address
VICIRQStatus	IRQ Status Register. This register reads out the state of those interrupt requests that are enabled and classified as IRQ.	RO	0	0xFFFF F000
VICFIQStatus	FIQ Status Requests. This register reads out the state of those interrupt requests that are enabled and classified as FIQ.	RO	0	0xFFFF F004
VICRawIntr	Raw Interrupt Status Register. This register reads out the state of the 32 interrupt requests / software interrupts, regardless of enabling or classification.	RO	0	0xFFFF F008
VICIntSelect	Interrupt Select Register. This register classifies each of the 32 interrupt requests as contributing to FIQ or IRQ.	R/W	0	0xFFFF F00C
VICIntEnable	Interrupt Enable Register. This register controls which of the 32 interrupt requests and software interrupts are enabled to contribute to FIQ or IRQ.	R/W	0	0xFFFF F010
VICIntEnClr	Interrupt Enable Clear Register. This register allows software to clear one or more bits in the Interrupt Enable register.	WO	0	0xFFFF F014
VICSoftInt	Software Interrupt Register. The contents of this register are ORed with the 32 interrupt requests from various peripheral functions.	R/W	0	0xFFFF F018
VICSoftIntClear	Software Interrupt Clear Register. This register allows software to clear one or more bits in the Software Interrupt register.	WO	0	0xFFFF F01C
VICProtection	Protection enable register. This register allows limiting access to the VIC registers by software running in privileged mode.	R/W	0	0xFFFF F020
VICVectAddr	Vector Address Register. When an IRQ interrupt occurs, the IRQ service routine can read this register and jump to the value read.	R/W	0	0xFFFF F030
VICDefVectAddr	Default Vector Address Register. This register holds the address of the Interrupt Service routine (ISR) for non-vectored IRQs.	R/W	0	0xFFFF F034
VICVectAddr0	Vector address 0 register. Vector Address Registers 0-15 hold the addresses of the Interrupt Service routines (ISRs) for the 16 vectored IRQ slots.	R/W	0	0xFFFF F100
VICVectAddr1	Vector address 1 register.	R/W	0	0xFFFF F104
VICVectAddr2	Vector address 2 register.	R/W	0	0xFFFF F108
VICVectAddr3	Vector address 3 register.	R/W	0	0xFFFF F10C
VICVectAddr4	Vector address 4 register.	R/W	0	0xFFFF F110
VICVectAddr5	Vector address 5 register.	R/W	0	0xFFFF F114
VICVectAddr6	Vector address 6 register.	R/W	0	0xFFFF F118
VICVectAddr7	Vector address 7 register.	R/W	0	0xFFFF F11C
VICVectAddr8	Vector address 8 register.	R/W	0	0xFFFF F12[illegible]
VICVectAddr9	Vector address 9 register.	R/W	0	0xFFFF F124
VICVectAddr10	Vector address 10 register.	R/W	0	0xFFFF F128
VICVectAddr11	Vector address 11 register.	R/W	0	0xFFFF F12C

Table 33: VIC register map

Name	Description	Access	Reset value[1]	Address
VICVectAddr12	Vector address 12 register.	R/W	0	0xFFFF F130
VICVectAddr13	Vector address 13 register.	R/W	0	0xFFFF F134
VICVectAddr14	Vector address 14 register.	R/W	0	0xFFFF F138
VICVectAddr15	Vector address 15 register.	R/W	0	0xFFFF F13C
VICVectCntl0	Vector control 0 register. Vector Control Registers 0-15 each control one of the 16 vectored IRQ slots. Slot 0 has the highest priority and slot 15 the lowest.	R/W	0	0xFFFF F200
VICVectCntl1	Vector control 1 register.	R/W	0	0xFFFF F204
VICVectCntl2	Vector control 2 register.	R/W	0	0xFFFF F208
VICVectCntl3	Vector control 3 register.	R/W	0	0xFFFF F20C
VICVectCntl4	Vector control 4 register.	R/W	0	0xFFFF F210
VICVectCntl5	Vector control 5 register.	R/W	0	0xFFFF F214
VICVectCntl6	Vector control 6 register.	R/W	0	0xFFFF F218
VICVectCntl7	Vector control 7 register.	R/W	0	0xFFFF F21C
VICVectCntl8	Vector control 8 register.	R/W	0	0xFFFF F220
VICVectCntl9	Vector control 9 register.	R/W	0	0xFFFF F224
VICVectCntl10	Vector control 10 register.	R/W	0	0xFFFF F228
VICVectCntl11	Vector control 11 register.	R/W	0	0xFFFF F22C
VICVectCntl12	Vector control 12 register.	R/W	0	0xFFFF F230
VICVectCntl13	Vector control 13 register.	R/W	0	0xFFFF F234
VICVectCntl14	Vector control 14 register.	R/W	0	0xFFFF F238
VICVectCntl15	Vector control 15 register.	R/W	0	0xFFFF F23C

LPC-2138-Interrupt-Quellen

5.4.3 Raw Interrupt status register (VICRawIntr - 0xFFFF F008)

This is a read only register. This register reads out the state of the 32 interrupt requests and software interrupts, regardless of enabling or classification.

Table 38: Raw Interrupt status register (VICRawIntr - address 0xFFFF F008) bit allocation
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	RO	RO	RO	RO	RO	RO	RO	RO
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	-	-	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	RO	RO	RO	RO	RO	RO	RO	RO
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	RO	RO	RO	RO	RO	RO	RO	RO
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	RO	RO	RO	RO	RO	RO	RO	RO

LPC-2138-Interrupt-Programmierung

- Beispiel:
  Regelmäßiges Auslösen eines Interrupts durch Timer 0 (z.B. alle 10ms) und Hochzählen einer Variable num_calls
- **1.Schritt:** Definition des Behandlers (ISR)

```
int volatile num_calls;

void IRQ_Timer0(void)  __attribute__((naked)); // Erklärung naked siehe nächste Folie

void IRQ_Timer0 (void)
{
        ISR_ENTRY();            // Eingangssequenz-Makro zur Sicherung des Prozessor-Status
        num_calls++;            //  Zähle die Anzahl der Aufrufe
        T0IR = 0x01;            //  Clear Interrupt Flag
        VICVectAddr = 0x00;     // Update priority hardware
        ISR_EXIT();             // Abschluß-Makro zur Wiedergherstellung des Prozessor-Status
}
```

- Alternativ kann in gcc auch ohne die Makros verfahren werden, wenn Prozedurkopf lautet:
  `void __attribute__ ((interrupt)) IRQ_Timer0 (void)`
  aber nicht für jeden Prozessor verfügbar!

LPC-2138-Interrupt-Programmierung

- void IRQ_Timer0(void) __attribute__((naked));

  Gcc-Manual (http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc.pdf) Seite 289:

  ((naked))

  Use this attribute on the ARM, AVR, IP2K and SPU ports to indicate that the specified function does not need prologue/epilogue sequences generated by the compiler. It is up to the programmer to provide these sequences. The only statements that can be safely included in naked functions are asm statements that do not have operands. All other statements, including declarations of local variables, if statements, and so forth, should be avoided. Naked functions should be used to implement the body of an assembly function, while allowing the compiler to construct the requisite function declaration for the assembler.

- Die Macros ISR_ENTRY() und ISR_EXIT() sind auf der Webseite verfügbar (armVIC.h).

LPC-2138-Interrupt-Programmierung

- **2.Schritt:** Programmierung des VIC

```
void sysInit (void)
{
    VICIntEnClear = 0xFFFFFFFF;          // clear all interrupts
    VICIntSelect = 0x00000000;           // clear all FIQ selections
    VICDefVectAddr = (uint32_t)reset;    // point unvectored IRQs to reset()
    VICVectAddr2 = (uint32_t) IRQ_Timer0; // Entry-Point IRQ-Timer0 (= Address of ISR); Prio=2
    VICVectCntl2 = 0x20 | 0x04;          // For IRQ slot 2 set „enable slot“ Bit 5 AND activate
    for                                  // Timer0 interrupts
    VICIntEnable = 1 << 0x04;            // Enable Tomier 0 Interrupts
}
```

LPC-2138-Interrupt-Programmierung

- **3.Schritt:** Programmierung des Timers 0

```
void timerInit (void)
{
    T0TC = 0;           // Timer-Value (to be incremented by clock)
    T0PR = 0;           // Prescaler = 0 (Vorteiler für Timer-Eingangstakt)
    T0MR0 = 240000;     // Match-Register. Generate Interrupt when Timer has reached 240000
    T0MCR = 0x03;       //  Generate interrupt on match and reset to zero
    T0TCR = 0x01;       // Timer Control Register: enable counting
    T0IR = 0x01;        // Enable Timer 0 Interrupts
}
```

Nebenläufigkeit

Probleme

Probleme

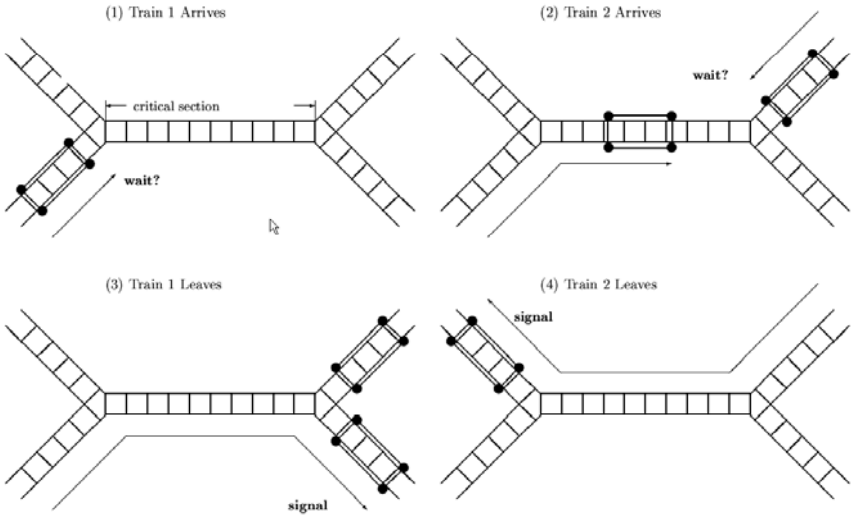
- **Race Conditions**:
  - Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
  - Lösung: Einführung von **kritischen Bereichen** und **wechselseitiger Ausschluss**.
- **Starvation** (**Aussperrung**):
  - Situation, in der ein Prozess unendlich lange auf ein Betriebsmittel wartet. Wichtig: sinnvolle Realisierung von Warteschlangen bei der Betriebsmittelvergabe, z.B. Prioritätenbasierte Warteschlangen
- **Priority Inversion** (**Prioritätsinversion**):
  - Wichtige Prozesse können durch unwichtigere Prozesse, die Betriebsmittel belegt haben verzögert werden, genaue Problemstellung siehe Kapitel Scheduling

Bedingungen an Lösung für wechselseitigen Ausschluss

- An eine gute Lösung für den wechselseitigen Ausschluss (WA) können insgesamt vier Bedingungen gestellt werden:
  1. Es dürfen niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
  2. Es dürfen keine Annahmen über die Geschwindigkeit oder Anzahl der Prozessoren gemacht werden.
  3. Kein Prozess darf außerhalb von kritischen Regionen andere Prozesse blockieren.
  4. Kein Prozess soll unendlich auf das Eintreten in den kritischen Bereich warten müssen.

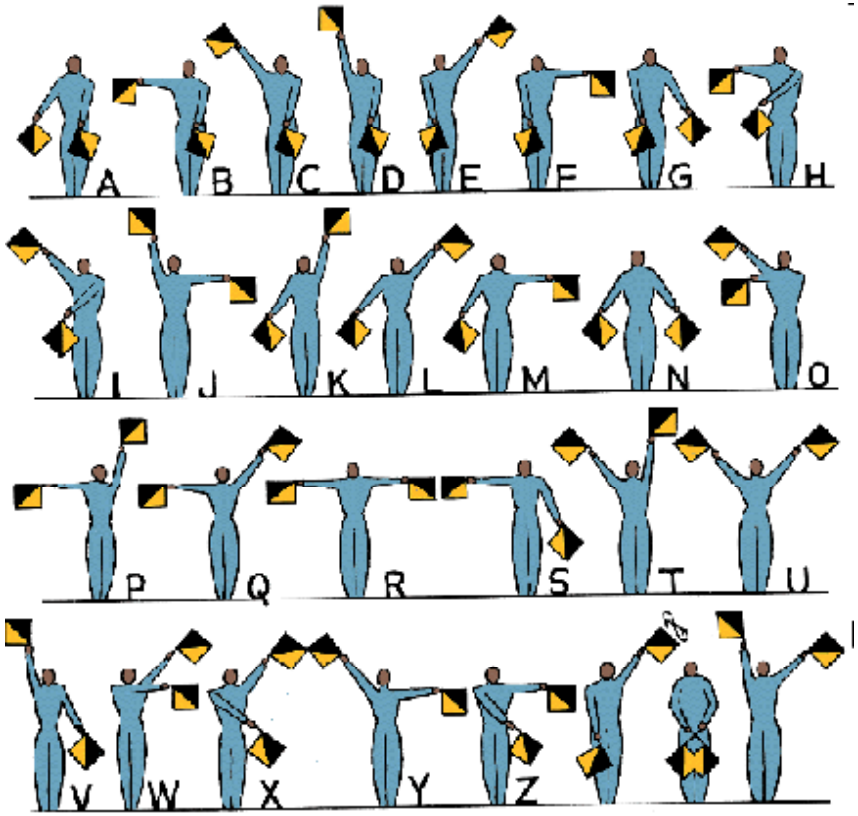
Kritische Bereiche

- Um einen kritischen Bereich zu schützen, sind Mechanismen erforderlich, die ein gleichzeitiges Betreten verschiedener Prozesse bzw. Prozeßklassen dieser Bereiche verhindern.
  - Darf maximal nur ein Prozess gleichzeitig auf den kritischen Bereich zugreifen, so spricht man vom **wechselseitigen Ausschluss**.
  - Wird verhindert, daß mehrere (unterschiedlich viele) Instanzen unterschiedlicher Prozeßklassen auf den Bereich zugreifen, so entspricht dies dem **Leser-Schreiber-Problem** (so dürfen beispielsweise mehrere Instanzen der Klasse Leser auf den Bereich gleichzeitig zugreifen, Instanzen der Klasse Schreiber benötigen den exklusiven Zugriff).



Programmtechnik zum Schutz kritischer Bereiche

- Aus dem Alltag sind diverse Mechanismen zum Schutz kritischer Bereiche bekannt:
  - *Signale* im Bahnverkehr
  - *Ampeln* zum Schutz der Kreuzung
  - *Schlösser* für einzelne Räume
  - *Vergabe* von Tickets
- Erstes Mittel zum Schutz kritischer Bereiche in der Softwaretechnik: der/das Semaphor, griech. für „Zeichenträger“ (siehe später und rechts)



Falsche Lösung: Verwendung einer globalen Variable

Prozeß A

```
bool block = false; //global
variable
```

Prozeß B

```
...
while(block){}; //busy wait
block=true;
... critical section A ...
block=false;
...
```

```
...
while(block){}; //busy wait
block=true;
... critical section A ...
block=false;
...
```

- Die obige Implementierung ist nicht korrekt,
  - da der Prozess direkt nach dem while-Abschnitt unterbrochen werden könnte und evtl. dann fortgesetzt wird, wenn block bereits durch einen anderen Prozess belegt ist.
  - Zudem ist die Lösung ineffizient (busy wait)

1. Möglichkeit: Peterson 1981 (Lösung für zwei Prozesse)

Deklaration globale Variablen

```
int turn=0;
boolean ready[2];
ready[0]=false;
ready[1]=false;
```

Prozess 0

```
...
ready[0]=true;
turn = 1;
while(ready[1]
    && turn==1){};//busy waiting
... critical section ...
ready[0]=false;
...
```

Prozess 1

```
...
ready[1]=true;
turn = 0;
while(ready[0]
    && turn==0){}; //busy waiting
... critical section ...
ready[1]=false;
...
```

- Ausschluß ist garantiert, aber „busy waiting“ verschwendet immer noch Rechenzeit
- Die Realisierung für N Prozesse ist als „Lamport‘s Bakery Algorithmus“ bekannt: http://en.wikipedia.org/wiki/Lamport‘s_bakery_algorithm

2. Möglichkeit: Ausschalten von Unterbrechungen zum WA

- Prozesswechsel beruhen immer auf dem Eintreffen einer Unterbrechung (z.B. neues Ereignis, Ablauf einer Zeitdauer)
- Die einfachste Möglichkeit einen Kontextwechsel zu verhindern ist das Ausschalten von Unterbrechungen bevor ein Prozess in den kritischen Bereich geht.
- Vorteile:
  - einfach zu implementieren, keine weiteren Konzepte sind nötig
  - schnelle Ausführung, Schreiben von Bits in Register
- Nachteile:
  - Für Multiprozessorsysteme ungeeignet
  - Keine Gerätebehandlung während der Sperre
  - Lange Sperren kritisch bei Echtzeitanwendungen

5.4.5 Interrupt Enable Clear register (VICIntEnClear - 0xFFFF F014)

This is a write only register. This register allows software to clear one or more bits in the Interrupt Enable register (see Section 5.4.4 "Interrupt Enable register (VICIntEnable - 0xFFFF F010)" on page 52), without having to first read it.

Table 42: Software Interrupt Clear register (VICIntEnClear - address 0xFFFF F014) bit allocation
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Symbol	-	-	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

3. Möglichkeit: Semaphor

- Semaphor (griechisch von Zeichenträger, Signalmast) wurden von Edsger W. Dijkstra im Jahr 1965 eingeführt.
- Ein Semaphor ist eine Datenstruktur, bestehend aus einer Zählvariable `s`, sowie den Funktionen `down()`oder `wait()` (bzw. `P()`, von probeer te verlagen) und `up()` oder `signal()` (bzw. `V()`, von verhogen).

```
Init(Semaphor s, Int v)
{
  s = v;
}

V(Semaphor s)
{
  s = s+1;
}

P(Semaphor s)
{
  while (s <= 0) {} ;  // Blockade, unterschiedliche Implementierungen
  s = s-1 ;            // sobald s>0 belege eine Ressource
}
```

- Bevor ein Prozess in den kritischen Bereich eintritt, muss er den Semaphor mit der Funktion `down()` anfordern. Nach Verlassen wird der Bereich durch die Funktion `up()` wieder freigegeben.
- **Wichtige Annahme**: die Ausführung der Funktionen von `up` und `down` darf nicht unterbrochen werden (atomare Ausführung), siehe Realisierung
- Solange der Bereich belegt ist (Wert des Semaphors <=0), wird der aufrufende Prozess blockiert.

Beispiel: Bankkonto

- Durch Verwendung eines gemeinsamen Semaphors `semAccount` kann das Bankkonto `account` auch beim schreibenden Zugriff von zwei Prozessen konsistent gehalten werden:

Prozess A	Prozess B
`P(semAccount);`	`P(semAccount);`
`x=readAccount(account);`	`y=readAccount(account);`
`x=x+500;`	`y=y-200;`
`writeAccount(x,account);`	`writeAccount(y,account);`
`V(semAccount);`	`V(semAccount);`

- Zur Realisierung des wechselseitigen Ausschlusses wird ein binärer Semaphor mit zwei Zuständen: 0 (belegt), 1 (frei) benötigt. Binäre Semaphore werden auch *Mutex* (von *mutal exclusion*) genannt.

Erweiterung: zählender Semaphore

- Nimmt ein Wert auch einen Wert größer eins an, so wird ein solch ein Semaphor auch als **zählender Semaphor** (counting semaphore) bezeichnet.
- Beispiel für den Einsatz von zählenden Semaphoren: In einem **Leser-Schreiber-Problem** kann die Anzahl der Leser aus Leistungsgründen z.B. auf 100 gleichzeitige Lesezugriffe beschränkt werden:

```
semaphore sem_reader_count;
init(sem_reader_count,100);
```

- Jeder Leseprozess führt dann folgenden Code aus:

```
P(sem_reader_count);
read();
V(sem_reader_count);
```

- Leser-Schreiber-Probleme sind vielfältig modifizierbar, je nach Priorität der prozesse. LS-Problem: Keine Prioritäten. Erstes LS-Problem: Leserpriorität. Zweites LS-Problem: Schreiber-Priorität.

Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.
  $\Rightarrow$ Die Funktionen `up()` und `down()` dürfen nicht unterbrochen werden, da sonst der Semaphor selbst inkonsistent werden kann.
- Funktionen die nicht unterbrechbar sind, werden **atomar** genannt.
- Realisierungsmöglichkeiten:
  1. Kurzfristige Blockade der Prozeßwechsel während der Bearbeitung der Funktionen `up()` und `down()`. Implementierung durch Verwendung einer Interrupt-Sperre, denn sämtliche Prozesswechsel werden durch **Unterbrechungen** (**Interrupts**) ausgelöst.
  2. **Spinlock**: Programmiertechnik auf der Basis von Busy Waiting. Vorteil: Unabhängig vom Betriebssystem und auch in Mehrprozessorsystemen zu implementieren, jedoch massive Verschwendung von Rechenzeit. Im Gegensatz dazu können die Lösungen von 1 und 2 mit Hilfe von Warteschlangen sehr effizient realisiert werden.
  3. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den Inhalt (typ. 0 für frei, 1 für belegt) eines Speicherwortes in ein Register und schreibt ununterbrechbar einen Wert (typ. $\neq 0$, z.B. 1 für belegt) in das Speicherwort.

Realisierungen von Semaphoren

Test&Set-Maschinenbefehl bei Mehrprozessorsystemen

- Problem: gemeinsamer Zugriff von mehreren Prozessoren auf einen Speicherbereich
- Für die Test&Set Operation muss für eine CPU der exklusive Zugriff auf den Speicherbereicht garantiert sein.

  → Bus Locking

- Mechanismen im Intel Pentium II für den atomaren Zugriff auf Speicherbereiche: Multiple Processor Management Abschnitt 7.1: http://download.intel.com/design/PentiumII/manuals/24319202.pdf

Verwendung des Test&Set Maschinenbefehls

```
enter_region: ; A "jump to" tag; function entry point.

tsl reg, flag       ; Test and Set Lock; flag is the
                    ; shared variable; it is copied
                    ; into the register reg and flag
                    ; then atomically set to 1.
cmp reg, #0         ; Was flag zero on entry?
jnz enter_region    ; Jump to enter_region if
                    ; reg is non-zero; i.e.,
                    ; flag was non-zero on entry.
ret                 ; Exit; i.e., flag was zero on
                    ; entry. If we get here, tsl
                    ; will have set it non-zero; thus,
                    ; we have claimed the resource as-
                    ; sociated with flag.
```

Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung P und Freigabe V des kritischen Bereiches durch den Programmierer
- Vergißt der Entwickler z.B. die Freigabe V des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
  - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozess zugreifen kann.
  - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
  - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
  - Die Signalisierung ist innerhalb des Monitors festgelegt, der Programmierer muss sie nicht selbstständig implementieren.

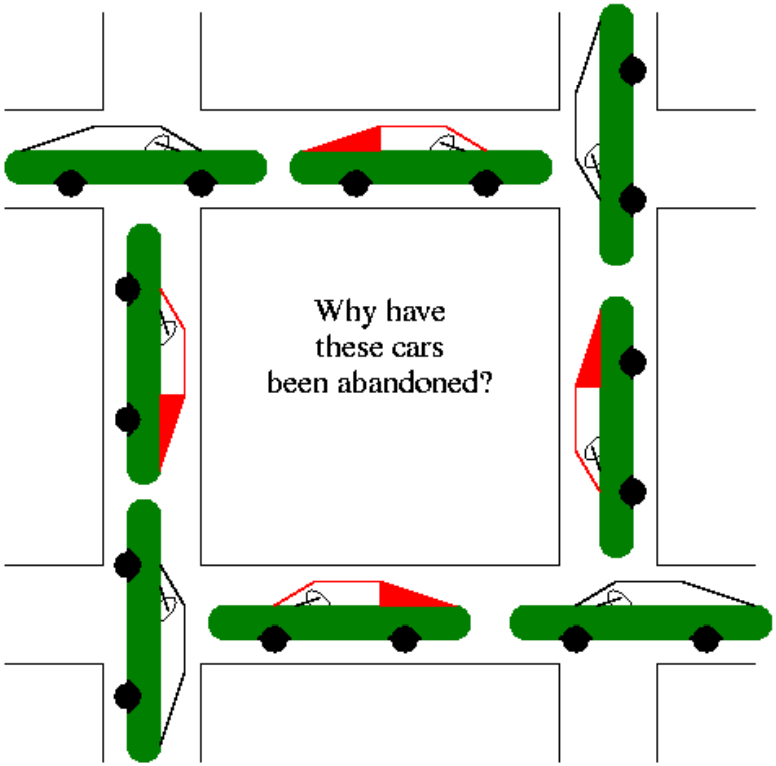
Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung**: normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphor implementiert.
  Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

```
public class Semaphore {
        private int value;

        public Semaphore (int initial) {
                value = initial;
        }

        synchronized public void up() {
                value++;
                if(value==1) notify();
        }

        synchronized public void down() {
                while(value==0) wait();
                value- -;
        }
}
```

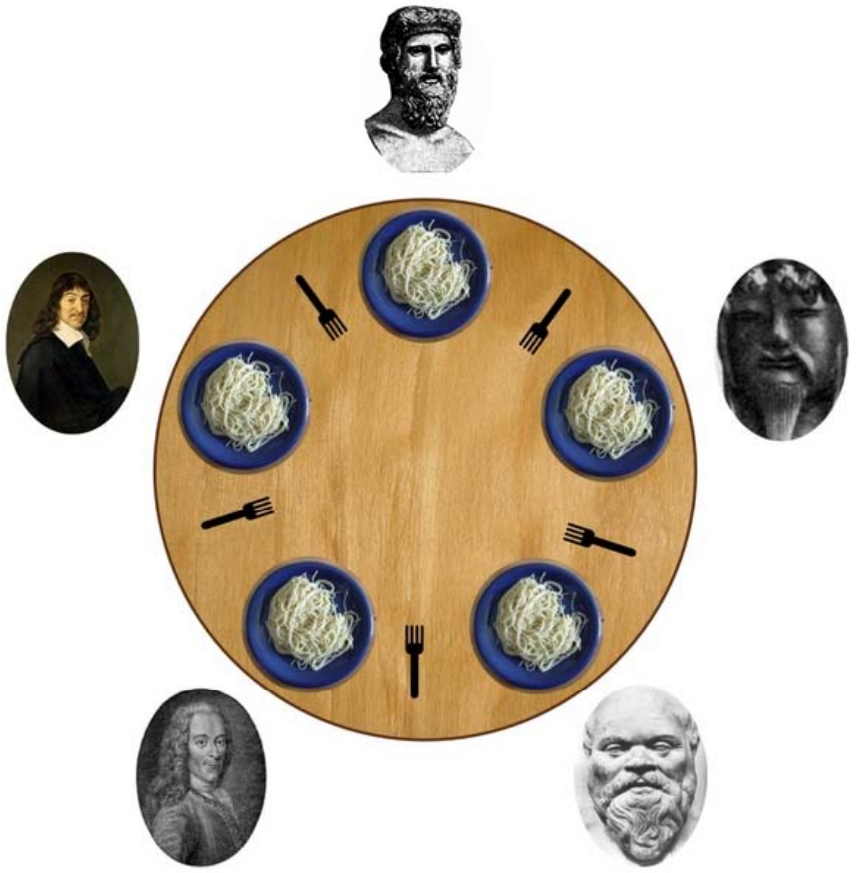
Bemerkung zu Verklemmungen (Deadlocks)

- Auch bei der korrekten Verwendung von Semaphoren und Monitoren kann es zu Deadlocks kommen, siehe Beispiel auf der folgenden Seite.
- Coffman, Elphick und Shoshani haben 1971 die vier konjunktiv notwendigen Voraussetzungen für einen Deadlock formuliert:
  1. Wechselseitiger Ausschluss: Es gibt eine Menge von exklusiven Ressourcen R_{exkl}, die entweder frei sind oder genau einem Prozess zugeordnet sind.
  2. Hold-and-wait-Bedingung: Prozesse, die bereits im Besitz von Ressourcen aus R_{exkl} sind, fordern weitere Ressourcen aus R_{exkl} an.
  3. Ununterbrechbarkeit: Die Ressourcen R_{exkl} können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
  4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.
- Umgekehrt (und positiv) formuliert: ist eine der Bedingungen nicht erfüllt, so sind Verklemmungen ausgeschlossen.



Klassisches Beispiel: Speisende Philosophen

- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber nur 5 Gabeln verfügbar.
- Die Philosophen denken und diskutieren. Ist einer hungrig, so greift er zunächst zur linken und dann zur rechten Gabel. Ist eine Gabel nicht an ihrem Platz, so wartet er bis die Gabel wieder verfügbar ist (ohne eine evtl. in der Hand befindliche Gabel zurückzulegen). Nach dem Essen legt er die Gabeln zurück.
- Problem: sind alle Philosophen gleichzeitig hungrig, so nehmen sie alle ihre linke Gabel und gleichzeitig ihrem Nachbarn die rechte Gabel weg. Alle Philosophen warten auf die rechte Gabel und es entsteht eine Verklemmung (deadlock).
- Gibt ein Philosoph seine Gabel nicht mehr zurück, so stirbt der entsprechende Nachbar den **Hungertod (starvation)**.



Fragestellung: Invers zählender Semaphor

- Aufgabenstellung: Implementierung des Leser-Schreiber-Problems mit Schreiber-Priorität
- Erläuterung:
  - Auf einen Datensatz können mehrere Leser gleichzeitig oder aber ein Schreiber zugreifen.
  - Sobald ein Schreiber den Schreibwunsch äußert, soll kein weiterer Leser (oder Schreiber) mehr auf den Datensatz zugreifen können. Zum Zeitpunkt der Signalisierung bestehende Lesevorgänge können regulär beendet werden, erst danach darf der Schreiber auf die Daten zugreifen.
- Problem: Häufig wird versucht das Problem mit einem *„invers zählenden Semaphor“* zu lösen, also einem Semaphor, der bei 0 freigibt und sonst blockiert.
- Wie geht es richtig?

Nebenläufigkeit

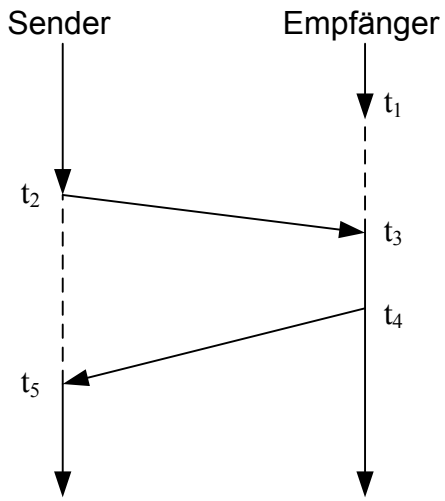
Interprozesskommunikation (IPC)

Interprozesskommunikation

- Notwendigkeit der Interprozesskommunikation
  - Prozesse arbeiten in unterschiedlichen Prozessräumen oder sogar auf unterschiedlichen Prozessoren.
  - Prozesse benötigen evtl. Ergebnisse von anderen Prozessen.
  - Zur Realisierung von wechselseitigen Ausschlüssen werden Mechanismen zur Signalisierung benötigt.
- Klassifikation der Kommunikation
  - synchrone vs. asynchrone Kommunikation
  - pure Ereignisse vs. wertbehaftete Nachrichten

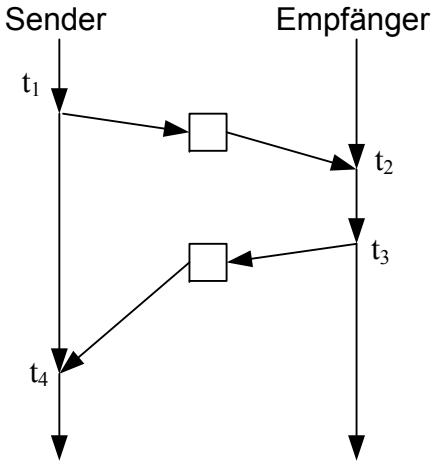
Synchron vs. Asynchron

Synchrone Kommunikation



t_1 : Empfänger wartet auf Nachricht
t_2 : Sender schickt Nachricht und blockiert
t_3 : Empfänger bekommt Nachricht, die Verarbeitung startet
t_4 : Verarbeitung beendet, Antwort wird gesendet
t_5 : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



t_1 : Sender schickt Nachricht an Zwischenspeicher und arbeitet weiter
t_2 : Empfänger liest Nachricht
t_3 : Empfänger schreibt Ergebnis in Zwischenspeicher
t_4 : Sender liest Ergebnis aus Zwischenspeicher

(Nicht eingezeichnet: zusätzliche Abfragen des Zwischenspeichers und evtl. Warten)

IPC-Mechanismen

- Übermittlung von Datenströmen:
  - direkter Datenaustausch
  - Pipes
  - Nachrichtenwarteschlangen (Message Queues)
- Signalisierung von Ereignissen:
  - Signale
  - Semaphore
- Synchrone Kommunikation
  - Barrieren/Rendezvous
  - Kanäle wie z.B. Occam
- Funktionsaufrufe:
  - RPC
  - Corba

Nebenläufigkeit

IPC: Kommunikation durch Datenströme

Direkter Datenaustausch

- Mit Semaphoren und Monitoren geschützte Datenstrukturen eignen sich sehr gut für den Austausch von Daten:
  - schnelle Kommunikation, da auf den Speicher direkt zugegriffen werden kann.
- Allerdings kann die Kommunikation nur lokal erfolgen und zudem müssen die Prozesse eng miteinander verknüpft sein.
- Programmiersprachen, Betriebssysteme, sowie Middlewareansätze bieten komfortablere Methoden zum Datenaustausch.
- Grundsätzlich erfolgt der Austausch über das Ausführen von Funktionen `send(receiver address, &message)` und `receive(sender address, &message)`.

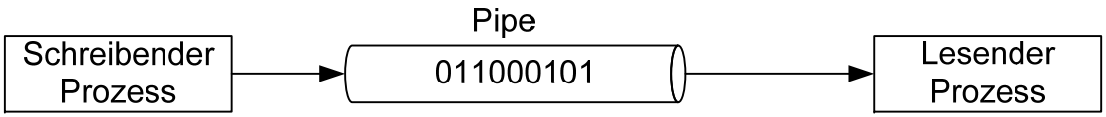
Fragestellungen beim Datenaustausch

- Nachrichtenbasiert oder Datenstrom?
- Lokale oder verteilte Kommunikation?
- Kommunikationsparameter:
  - mit/ohne Bestätigung
  - Nachrichtenverluste
  - Zeitintervalle
  - Reihenfolge der Nachrichten
- Adressierung
- Authentifizierung
- Performance
- Sicherheit (Verschlüsselung)

Heute: vor allem lokale Kommunikation, echtzeitfähige Protokolle zur Kommunikation in eigenem Kapitel

Pipes

- Die Pipe bezeichnet eine gepufferte, unidirektionale Datenverbindung zwischen zwei Prozessen nach dem **First-In-First-Out-** (**FIFO**-)Prinzip.
- Über den Namen der Pipe (ähnlich einem Dateinamen) können Prozesse unterschiedlichen Ursprungs auf eine Pipe lesend oder schreibend zugreifen. Zur Kommunikation zwischen Prozessen gleichen Ursprungs (z.B. Vater-, Kindprozess) können auch anonyme Pipes verwendet werden. Die Kommunikation erfolgt immer asynchron.



Pipes in Posix

- POSIX (Portable Operating System Interface) versucht durch Standardisierung der Systemaufrufe die Portierung von Programmen zwischen verschiedenen Betriebssystemen zu erleichtern.
- POSIX.1 definiert folgende Funktionen für Pipes:

```
int mkfifo(char* name, int mode);       /*Erzeugen einer benannten Pipe*/
int unlink ( char *name );              /*Loeschen einer benannten Pipe*/
int open ( char *name, int flags);      /*Oeffnen einer benannten Pipe*/
int close ( int fd );                   /*Schliessen des Lese- oder Schreibendes einer
                                          Pipe*/
int read ( int fd, char *outbuf, unsigned bytes ); /*Lesen von einer Pipe*/
int write ( int fd, char *outbuf,unsigned bytes ); /*Schreiben an eine Pipe*/
int pipe ( int fd[2] );                 /*Erzeugen eine unbenannte Pipe*/
```

Nachteile von Pipes

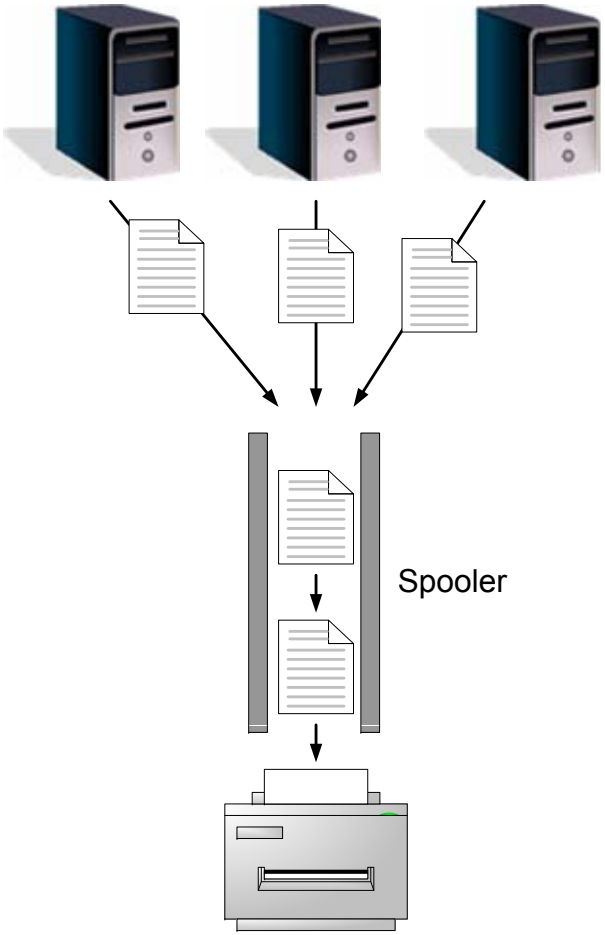
- Pipes bringen einige Nachteile mit sich:
  - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
  - Daten sind nicht priorisierbar.
  - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung:
  - Es können keine Daten aufgehoben werden.
  - Beim Öffnen blockiert der Funktionsaufruf, bis auch das zweite Zugriffsende geöffnet wird (Verhinderung durch O_NDELAY Flag).
- Lösung: Nachrichtenwartschlangen

Nachrichtenwarteschlangen (message queues)

- Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenwarteschlangen, wie in POSIX 1003.1b (Echtzeiterweiterung von POSIX) definiert, betrachtet.
- Eigenschaften der POSIX MessageQueues:
  - Beim Anlegen einer MessageQueue wird der benötigte Speicher reserviert. ⇒ Speicher muss nicht erst beim Schreibzugriff angelegt werden.
  - Die Kommunikation erfolgt nachrichtenorientiert. Die Anzahl der vorhandenen Nachrichten kann dadurch abgefragt werden.
  - Nachrichten sind priorisierbar ⟶ Es können leichter Zeitgarantien gegeben werden.

Nachrichtenwarteschlangen

- Schreibzugriff in Standardsystemen: Der schreibende/sendende Prozess wird nur dann blockiert werden, falls der Speicher der Datenstruktur bereits voll ist. **Alternative in Echtzeitsystemen: Fehlermeldung ohne Blockade**.
- Lesezugriff in Standardsystemen: Beim lesenden/empfangenden Zugriff auf einen leeren Nachrichtenspeicher wird der aufrufende Prozess blockiert bis eine neue Nachricht eintrifft. **Alternative: Fehlermeldung ohne Blockade**.
- Ein anschauliches Beispiel für den Einsatzbereich ist der Spooler eines Druckers: dieser nimmt die Druckaufträge der verschiedenen Prozesse an und leitet diese der Reihe nach an den Drucker weiter.



Message Queues in POSIX

- POSIX definiert folgende Funktionen für Nachrichtenwartschlangen:

```
mqd_t mq_open(const char *name, int oflag, ...);  /*Oeffnen einer Message Queue*/
int mq_close(mqd_t mqdes);                        /*Schliessen einer Message Queue*/
int mq_unlink(const char *name);                  /*Loeschen einer
                                                    Nachrichtenwarteschlange*/
int mq_send(mqd_t mqdes, const char *msg_ptr,
    size_t msg_len, unsigned int msg_prio);       /*Senden einer Nachricht*/
size_t mq_receive(mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned int *msg_prio);      /*Empfangen einer Nachricht*/
int mq_setattr(mqd_t mqdes, const struct
    mq_attr *mqstat, struct mq_attr *mqstat);     /*Aendern der Attribute*/
int mq_getattr(mqd_t mqdes,
    struct mq_attr *mqstat);                      /*Abrufen der aktuellen
                                                    Eigenschaften*/
int mq_notify(mqd_t mqdes,
    const struct sigevent *notification);         /*Anforderung eines Signals bei
                                                    Nachrichtenankunft*/
```

Nebenläufigkeit

IPC: Kommunikation durch Ereignisse

Signale

- **Signale** werden in Betriebssystemen typischerweise zur Signalisierung von Ereignissen an Prozessen verwendet.
- Signale können verschiedene Ursachen haben:
  - Ausnahmen, z.B. Division durch Null (SIGFPE) oder ein Speicherzugriffsfehler (SIGSEGV)
  - Reaktion auf Benutzereingaben (z.B. Ctrl / C)
  - Signal von anderem Prozess zur Kommunikation
  - Signalisierung von Ereignissen durch das Betriebssystem, z.B. Ablauf einer Uhr, Beendigung einer asynchronen I/O-Funktion, Nachrichtankunft an leerer Nachrichtenwarteschlange (siehe mq_notify())

Prozessreaktionen auf Signale

- Der Prozess hat drei Möglichkeiten auf Signale zu reagieren:
  1. Ignorierung der Signale
  2. Ausführen einer Signalbehandlungsfunktion
  3. Verzögerung des Signals, bis Prozess bereit für Reaktion ist
- Zudem besteht die Möglichkeit mit der Standardreaktion auf das bestimmte Signal zu reagieren. Da aber typischerweise die Reaktion auf Signale die Beendigung des Empfängerprozesses ist, sollte ein Programm über eine vernünftige Signalbehandlung verfügen, sobald ein Auftreten von Signalen wahrscheinlich wird.

Semaphore zur Vermittlung von Ereignissen

- Semaphore können neben der Anwendung des wechselseitigen Ausschlusses auch zur Signalisierung von Ereignissen verwendet werden.
- Es ist zulässig, dass Prozesse (Erzeuger) Semaphore andauernd freigeben und andere Prozesse (Verbraucher) Semaphore dauern konsumieren.
- Es können auch benannte Semaphoren erzeugt werden, die dann über Prozessgrenzen hinweg verwendet werden können.
- Notwendige Funktionen sind dann:
  - sem_open(): zum Erzeugen und / oder Öffnen eines benannten Semaphors
  - sem_unlink(): zum Löschen eines benannten Semaphors

Signalisierung durch Semaphore: Beispiel

- Beispiel: ein Prozeß **Worker** wartet auf einen Auftrag (abgespeichert z.B. in einem char-Array job) durch einen Prozess **Contractor**, bearbeitet diesen und wartet im Anschluß auf den nächsten Auftrag:

Worker*:

```
while(true)
{
        down(sem); /*wait for
                   next job*/
        execute(job);
}
```

Contractor*:

```
...
job=...  /*create new job and save
         address in global variable*/
up(sem); /*signal new job*/
...
```

** sehr stark vereinfachte Lösung, da zu einem Zeitpunkt nur ein Job verfügbar sein darf*

Probleme

- Problematisch an der Implementierung des Beispiels auf der letzten Folie ist, dass der Zeiger auf den Auftrag `job` nicht geschützt ist und es so zu fehlerhaften Ausführungen kommen kann.
- Durch Verwendung eines zusätzlichen Semaphors kann dieses Problem behoben werden.
- Ist die Zeit zwischen zwei Aufträgen zu kurz um die rechtzeitige Bearbeitung sicherzustellen, so kann es zu weiteren Problemen kommen:
  - Problem 1: Der Prozess **Contractor** muss warten, weil der Prozeß **Worker** den letzten Auftrag noch bearbeitet.
  - Problem 2: Der letzte Auftrag wird überschrieben, falls dieser noch gar nicht bearbeitet wurde. Abhängig von der Implementierung des Semaphors könnte dann der neue Auftrag zudem zweifach ausgeführt werden.

→ mit Semaphoren sind nur einfache Signalisierungsprobleme (ohne Datentransfer) zu lösen, ansonsten sollten Warteschlangen verwendet werden

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Vorherige Lösung:

Reader:

```
…
        down(semWriter);
        down(semCounter);
        rcounter++;
        up(semCounter);
        up(semWriter);

        read();

        down(semCounter);
        rcounter--;
        up(semCounter);
…
```

Writer:

```
…
        down(semWriter);

        while(true)
        {
                down(semCounter);
                if(rcounter==0)
                        break;
                up(semCounter);
        }
        up(semCounter);

        write();

        up(semWriter);
…
```

Problem: Busy Waiting

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Lösung mit Signalisierung:

Reader:

```
…
            down(semWriter);
            down(semCounter);
            rcounter++;
            if(rcounter==1)
                        down(semReader);
            up(semCounter);
            up(semWriter);

            read();

            down(semCounter);
            rcounter--;
            if(rcounter==0)
                        up(semReader);
            up(semCounter);
…
```

Writer:

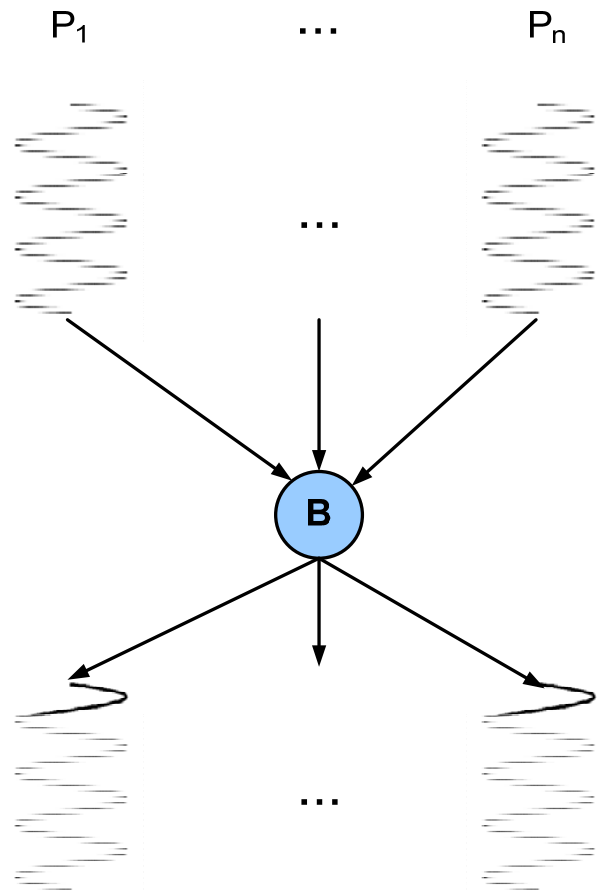
```
…
            down(semWriter);
            down(semReader);
            up(semReader);

            write();


            up(semWriter);
…
```

Nebenläufigkeit

Synchrone Kommunikation: Barrieren

Synchrone Kommunikation: Barrieren

- **Definition**: Eine Barriere für eine Menge *M* von Prozessen ist ein Punkt, den alle Prozesse $P_i \in M$ erreichen müssen, bevor irgendein Prozess aus *M* die Berechnung über diesen Punkt hinaus fortfahren kann.
- Der Spezialfall für $|M|=2$ wird als Rendezvous, siehe auch Ada, bezeichnet.
- Barrieren können mit Hilfe von Semaphoren implementiert werden.



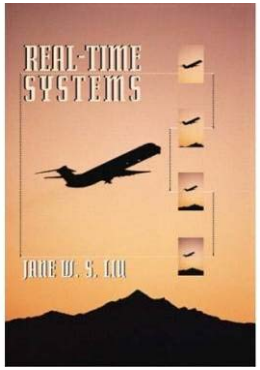
Kapitel 4

Scheduling

Inhalt

- Definitionen
- Kriterien zur Auswahl des Scheduling-Verfahrens
- Scheduling-Verfahren
- Prioritätsinversion
- Exkurs: Worst Case Execution Times

Literatur



Jane W. S. Liu, Real-Time Systems, 2000

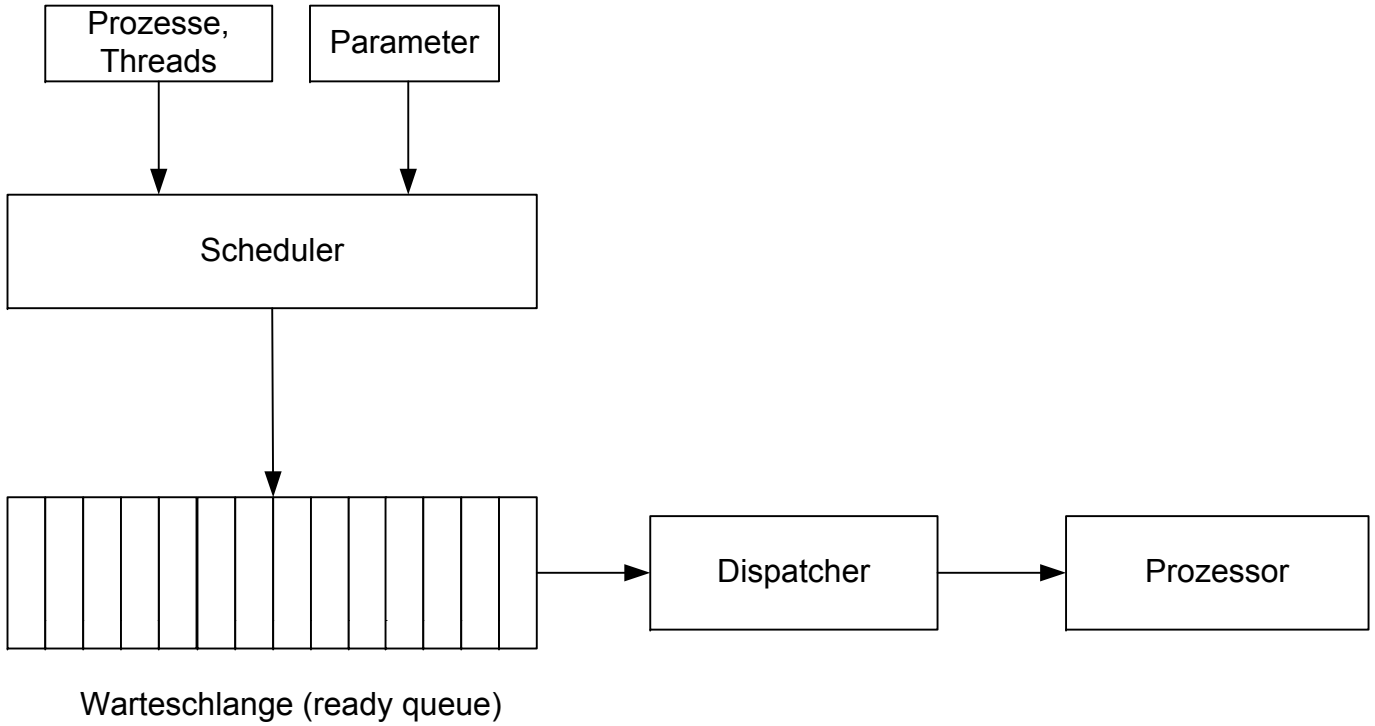
Fridolin Hofmann: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991

- Journals:
  - John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo: Implications of classical scheduling results for real-time systems. IEEE Computer, Special Issue on Scheduling and Real-Time Systems, 28(6):16–25, June 2005.
  - Giorgio C. Buttazzo: Rate Monotonic vs. EDF: Judgement Day (http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf)
  - Puschner, Peter; Burns, Alan: A review of Worst-Case Execution-Time Analysis, Journal of Real-Time Systems 18 (2000), S.115-128

Scheduling

Definitionen

Scheduler und Dispatcher

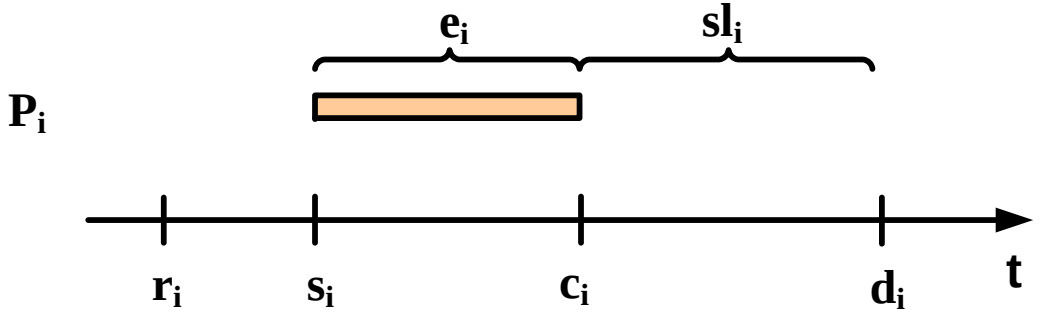


Scheduler und Dispatcher

- **Scheduler**: Modul eines Betriebssystems, das die Rechenzeit an die unterschiedlichen Prozesse verteilt. Der ausgeführte Algorithmus wird als Scheduling-Algorithmus bezeichnet. Aufgabe des Schedulers ist also die langfristige Planung (Vergleich: Erstellung eines Zugfahrplans).
- **Dispatcher**: Übersetzung: Einsatzleiter, Koordinator, Zuteiler (v.a. im Bereich der Bahn gebräuchlich). Im Rahmen der Prozessverwaltung eines Betriebssystems dient der Dispatcher dazu, bei einem Prozesswechsel dem derzeit aktiven Prozess die CPU zu entziehen und anschließend dem nächsten Prozess die CPU zuzuteilen. Die Entscheidung, welcher Prozess der nächste ist, wird vom Scheduler im Rahmen der Warteschlangenorganisation getroffen.

Zeitliche Bedingungen

- Folgende Größen sind charakteristisch für die Ausführung von Prozessen:
  1. P_i bezeichnet den i. **Prozess** (bzw. Thread)
  2. r_i: **Bereitzeit** (**ready time**) des Prozesses P_i und damit der früheste Zeitpunkt an dem der Prozess dem Prozessor zugeteilt werden kann.
  3. s_i: **Startzeit**: der Prozessor beginnt P_i auszuführen.
  4. e_i: **Ausführungszeit** (**execution time**): Zeit die der Prozess P_i zur reinen Ausführung auf dem Prozessor benötigt.
  5. c_i: **Abschlußzeit** (**completion time**): Zeitpunkt zu dem die Ausführung des Prozesses P_i beendet wird.
  6. d_i: **Frist** (**deadline**): Zeitpunkt zu dem die Ausführung des Prozesses P_i in jeden Fall beendet sein muss.
  7. sl_i: **Slack**: Deadline-(aktuelle Zeit + verbleibende Berechnungszeit)



Spielraum (slack time)

- Mit dem Spielraum (slack time) sl_i eines Prozesses P_i wird Zeitraum bezeichnet, um den ein Prozess noch maximal verzögert werden darf:
  - Die Differenz zwischen der verbleibenden Zeit bis zum Ablauf der Frist und der noch benötigten Ausführungszeit zur Beendigung des Prozesses Pi .
- Der Spielraum eines Prozesses, der aktuell durch den Prozessor ausgeführt wird, bleibt konstant, während sich die Spielräume aller nicht ausgeführten Prozesse verringern.

Faktoren bei der Planung

- Für die Planung des Schedulings müssen folgende Faktoren berücksichtigt werden:
  - Art der Prozesse (periodisch, nicht periodisch, sporadisch)
  - Gemeinsame Nutzung von Ressourcen (**shared ressources**)
  - Fristen
  - Vorrangrelationen (**precedence constraints**: Prozess P_i muss vor P_j ausgeführt werden)

Arten der Planung

- Es kann zwischen unterschiedlichen Arten zum Planen unterschieden werden:
  - offline vs. online Planung
  - statische vs. dynamische Planung
  - präemptives vs. nicht-präemptives Scheduling

Offline Planung

- Mit der offline Planung wird die Erstellung eines Ausführungsplanes zur Übersetzungszeit bezeichnet. Zur Ausführungszeit arbeitet der Dispatcher den Ausführungsplan dann ab.
- **Vorteile**:
  - deterministisches Verhalten des Systems
  - wechselseitiger Ausschluss in kritischen Bereichen wird direkt im Scheduling realisiert
- **Nachteile**:
  - Bereitzeiten, Ausführungszeiten und Abhängigkeit der einzelnen Prozesse müssen schon im Voraus bekannt sein.
  - Die Suche nach einem Ausführungsplan ist im Allgemeinen ein NP-hartes Problem. Es werden jedoch keine optimalen Pläne gesucht, vielmehr ist ein gute Lösung (Einhaltung aller Fristen) ausreichend.

Online Scheduling

- Alle Schedulingentscheidungen werden online, d.h. auf der Basis der Menge der aktuell lauffähigen Prozesse und ihrer Parameter getroffen.
- Im Gegensatz zur offline Planung muss wechselseitiger Ausschluss nun über den expliziten Ausschluss (z.B. Semaphoren) erfolgen.
- Vorteile:
  - Flexibilität
  - Bessere Auslastung der Ressourcen
- Nachteile:
  - Es müssen zur Laufzeit Berechungen zum Scheduling durchgeführt werden $\Rightarrow$ Rechenzeit geht verloren.
  - Garantien zur Einhaltung von Fristen sind schwieriger zu geben.
  - Problematik von Race Conditions

Statische vs. dynamische Planung

- Bei der statischen Planung basieren alle Entscheidungen auf Parametern, die vor der Laufzeit festgelegt werden.
- Zur statischen Planung wird Wissen über:
  - die Prozessmenge
  - ihre Prioritäten
  - das Ausführungsverhalten

  benötigt.
- Bei der dynamischen Planung können sich die Scheduling-Parameter (z.B. die Prioritäten) zur Laufzeit ändern.
- **Wichtig**: Statische Planung und Online-Planung schließen sich nicht aus: z.B. Scheduling mit festen Prioritäten.

Präemption

- Präemptives (bevorrechtigt, entziehend) Scheduling: Bei jedem Auftreten eines relevanten Ereignisses wird die aktuelle Ausführung eines Prozesses unterbrochen und eine neue Schedulingentscheidung getroffen.
- Präemptives (unterbrechbares) Abarbeiten:
  - Aktionen (Prozesse) werden nach bestimmten Kriterien geordnet (z.B. Prioritäten, Frist,…).
  - Diese Kriterien sind statisch festgelegt oder werden dynamisch berechnet.
  - Ausführung einer Aktion wird sofort unterbrochen, sobald Aktion mit höherer Priorität eintrifft.
  - Die unterbrochene Aktion wird an der Unterbrechungsstelle fortgesetzt, sobald keine Aktion höherer Priorität ansteht.
  - Typisch für Echtzeitaufgaben (mit Ausnahme von Programmteilen, die zur Sicherung der Datenkonsistenz nicht unterbrochen werden dürfen).
  - Nachteil: häufiges Umschalten reduziert Leistung.

Ununterbrechbares Scheduling

- Ein Prozess, der den Prozessor zugewiesen bekommt, wird solange ausgeführt, bis der Prozess beendet wird oder er aber den Prozess freigibt.
- Scheduling-Entscheidungen werden nur nach der Prozessbeendigung oder dem Übergang des ausgeführten Prozesses in den blockierten Zustand vorgenommen.
- Eine begonnene Aktion wird beendet, selbst wenn während der Ausführung Aktionen höherer Dringlichkeit eintreffen
  $\Rightarrow$ Nachteil: evtl. Versagen (zu lange Reaktionszeit) des Systems beim Eintreffen unvorhergesehener Anforderungen
- Anmerkung: Betriebssysteme unterstützen allgemein präemptives Scheduling solange ein Prozess im Userspace ausgeführt, Kernelprozesse werden häufig nicht oder selten unterbrochen.

  $\Rightarrow$ Echtzeitbetriebssysteme zeichnen sich in Bezug auf das Scheduling dadurch aus, dass nur wenige Prozesse nicht unterbrechbar sind und diese wiederum sehr kurze Berechnungszeiten haben.

Schedulingkriterien

- Kriterien in Standardsystemen sind:
  - Fairness: gerechte Verteilung der Prozessorzeit
  - Effizienz: vollständige Auslastung der CPU
  - Antwortzeit: interaktive Prozesse sollen schnell reagieren
  - Verweilzeit: Aufgaben im Batchbetrieb (sequentielle Abarbeitung von Aufträgen) sollen möglichst schnell ein Ergebnis liefern
  - Durchsatz: Maximierung der Anzahl der Aufträge, die innerhalb einer bestimmten Zeitspanne ausgeführt werden
- In Echtzeitsystemen:
  - Einhaltung der Fristen: d.h. $\forall i\ c_i<d_i$ unter Berücksichtigung von Kausalzusammenhängen (Synchronisation, Vorranggraphen, Präzedenzsystemen)
  - Zusätzliche Kriterien können anwendungsabhängig hinzugenommen werden, solange sie der Einhaltung der Fristen untergeordnet sind.

Scheduling

Verfahren

Allgemeines Verfahren

- Gesucht: Plan mit aktueller Start und Endzeit für jeden Prozess P_i.
- Darstellung zum Beispiel als nach der Zeit geordnete Liste von Tupeln (P_i, s_i, c_i)
- Falls Prozesse unterbrochen werden können, so kann jedem Prozess P_i auch eine Menge von Tupeln zugeordnet werden.
- Phasen der Planung:
  - Test auf Einplanbarkeit (feasibility check)
  - Planberechnung (schedule construction)
  - Umsetzung auf Zuteilung im Betriebssystem (dispatching)
- Bei Online-Verfahren können die einzelnen Phasen überlappend zur Laufzeit ausgeführt werden.
- Zum Vergleich von Scheduling-Verfahren können einzelne Szenarien durchgespielt werden.

Definitionen

- **Zulässiger Plan**: Ein Plan ist zulässig, falls alle Prozesse einer Prozessmenge eingeplant sind und dabei keine Präzedenzrestriktionen und keine Zeitanforderungen verletzt werden.
- **Optimales Planungsverfahren**: Ein Verfahren ist optimal, falls es für jede Prozessmenge unter gegebenen Randbedingung einen zulässigen Plan findet, falls ein solcher existiert.

Test auf Einplanbarkeit

- Zum Test auf Einplanbarkeit können zwei Bedingungen angegeben werden, die für die Existenz eines zulässigen Plans notwendig sind (Achtung: häufig nicht ausreichend):
  1. $r_i + e_i \leq d_i$, d.h. jeder Prozess muss in dem Intervall zwischen Bereitzeit und Frist ausgeführt werden können.
  2. Für jeden Zeitraum $[t_i, t_j]$ muss die Summe der Ausführungszeiten e_x der Prozesse P_x mit $r_x \geq t_i \wedge d_x \leq$ tj kleiner als der Zeitraum sein.
- Durch weitere Rahmenbedingungen (z.B. Abhängigkeiten der einzelnen Prozesse) können weitere Bedingungen hinzukommen.

Schedulingverfahren

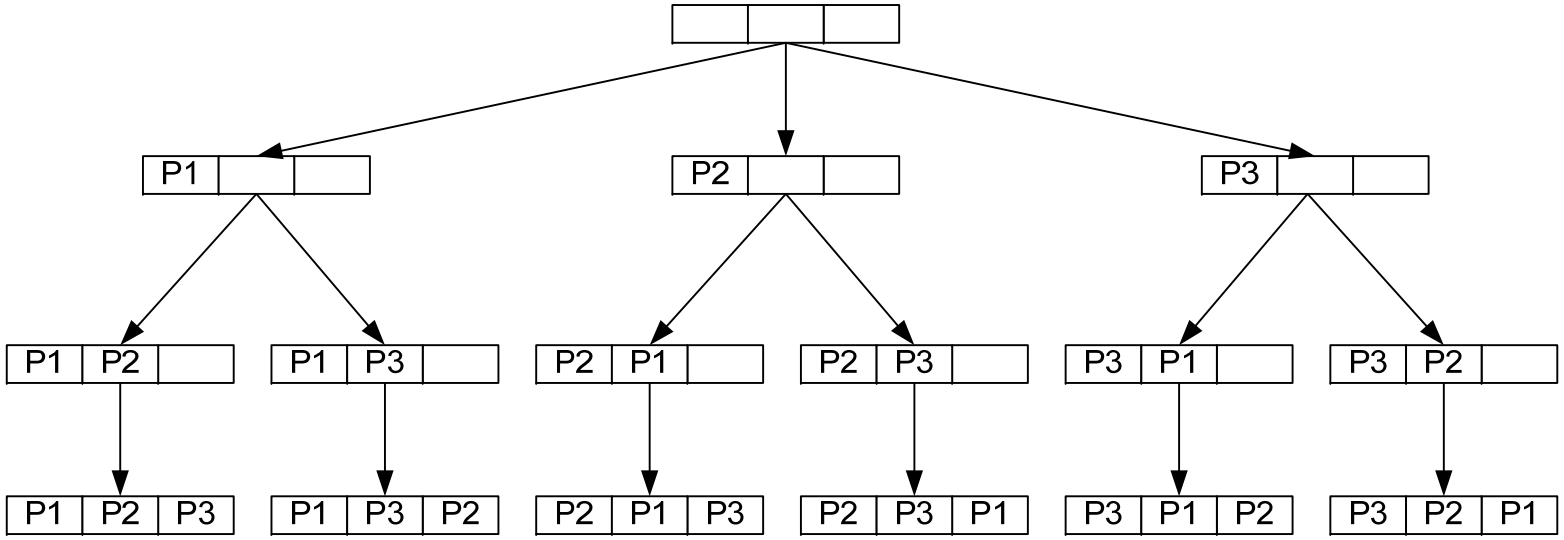
- Planen aperiodischer Prozesse
  - Planen durch Suchen
  - Planen nach Fristen
  - Planen nach Spielräumen
- Planen periodischer Prozesse
  - Planen nach Fristen
  - Planen nach Raten
- Planen abhängiger Prozesse

Scheduling

Scheduling-Verfahren für 1-Prozessor-Systeme

Planen durch Suchen

- Betrachtung: ununterbrechbare Aktionen/Prozesse vorausgesetzt
- Lösungsansatz: exakte Planung durch Durchsuchen des Lösungsraums
- Beispiel:
  - n=3 Prozesse P_1,P_2,P_3 und 1 Prozessor
  - Suchbaum:



Problem: Komplexität

- n! Permutationen müssen bewertet werden, bei Mehrprozessorsystemen ist das Problem der Planung NP-vollständig
- Durch präemptives Scheduling bzw. durch unterschiedliche Bereitzeiten kann das Problem weiter verkompliziert werden.
- Die Komplexität kann durch verschiedene Maßnahmen leicht verbessert werden:
  - Abbrechen von Pfaden bei Verletzung von Fristen
  - Verwendung von Heuristiken: z.B. Sortierung nach Bereitstellzeiten r_i
- Prinzipiell gilt jedoch: **Bei komplexen Systemen ist Planen durch Suchen nicht möglich**.

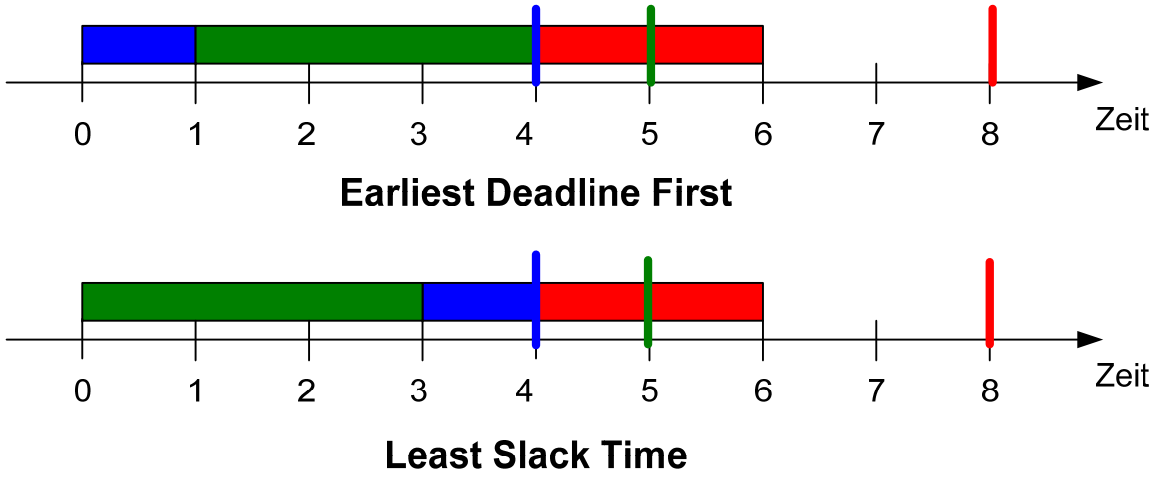
Scheduling-Strategien (online, nicht-präemptiv) für Einprozessorsysteme

1. EDF: Einplanen nach Fristen (Earliest Deadline First): Der Prozess, dessen Frist als nächstes endet, erhält den Prozessor.
2. LST: Planen nach Spielraum (Least Slack Time): Der Prozess mit dem kleinsten Spielraum erhält den Prozessor.
   - Der Spielraum berechnet sich wie folgt:
     Deadline-(aktuelle Zeit + verbleibende Berechnungszeit)
   - Der Spielraum für den aktuell ausgeführten Prozess ist konstant.
   - Die Spielräume aller anderen Prozesse nehmen ab.

- Vorteil und Nachteile:
  - LST erkennt Fristverletzungen früher als EDF.
  - Für LST müssen die Ausführungszeiten der Prozesse bekannt sein.

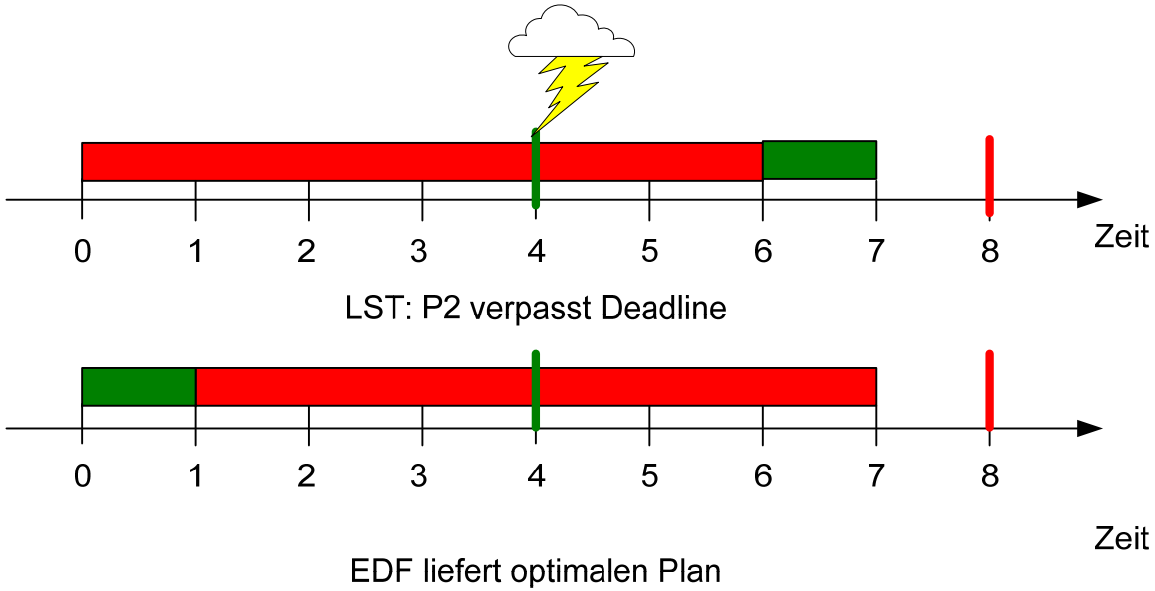
Beispiel

- 3 Prozesse:
  P_1: r_1=0; e_1=2; d_1=8;
  P_2: r_2=0; e_2=3; d_2=5;
  P_3: r_3=0; e_3=1; d_3=4;



Versagen von LST

- LST kann selbst bei gleichen Bereitzeiten im nicht-präemptiven Fall versagen.
- 2 Prozesse:
  P_1: r_1=0; e_1=6; d_1=8;
  P_2: r_2=0; e_2=1; d_2=4;



- Anmerkung: Aus diesem Grund wird LST nur in präemptiven Systemen eingesetzt. Bei Prozessen mit gleichen Spielräumen wird einem Prozess Δ eine Mindestausführungszeit garantiert.

Optimalität von EDF

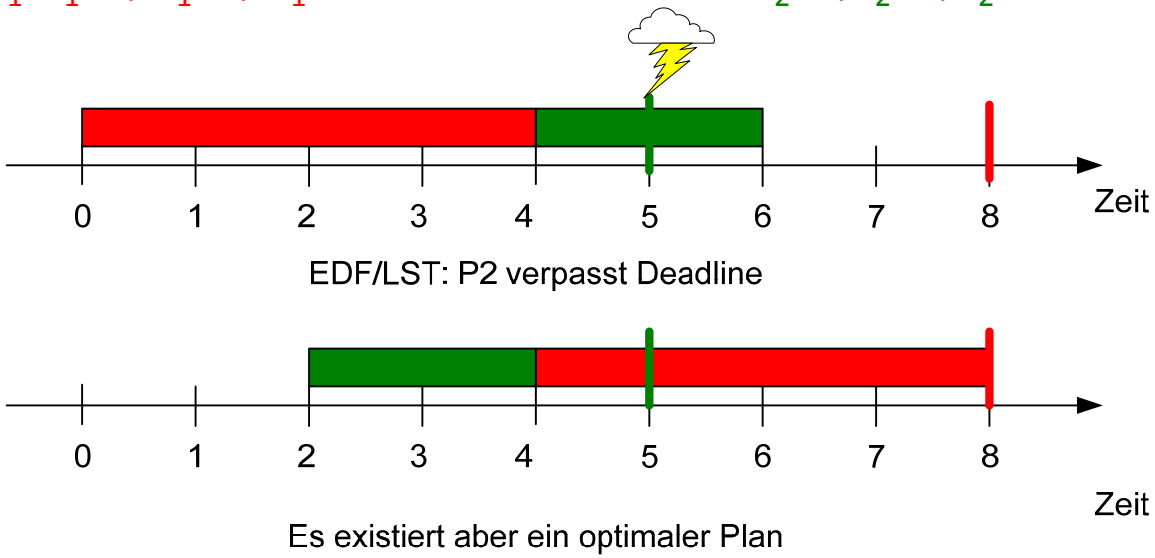
- Unter der Voraussetzung, dass alle Prozesse P_i eine Bereitzeit $r_i=0$ besitzen und das ausführende System ein Einprozessorsystem ist, ist EDF optimal, d.h. ein zulässiger Plan wird gefunden, falls ein solcher existiert.
- Beweisidee für EDF: Tausch in existierendem Plan
  - Sie $Plan_x$ ein zulässiger Plan.
  - Sei $Plan_{EDF}$ der Plan, der durch die EDF-Strategie erstellt wurde.
  - Ohne Einschränkung der Allgemeinheit: die Prozessmenge sei nach Fristen sortiert, d.h. $d_i \leq d_j$ für i<j.
  - Idee: Schrittweise Überführung des Planes $Plan_x$ in $Plan_{EDF}$
    - $P(Plan_x,t)$ sei der Prozess, der von $Plan_x$ zum Zeitpunkt t ausgeführt wird.
    - $Plan_x(t)$ ist der bis zum Zeitpunkt t in $Plan_{EDF}$ überführte Plan ($\Rightarrow Plan_x(0)=Plan_x$).

Fortsetzung des Beweises

- Wir betrachten ein Zeitinterval Δ_t.
- Zum Zeitpunkt t gilt:
  $i=P(Plan_{EDF},t)$
  $j=P(Plan_x,t)$
- Nur der Fall j>i ist interessant. Es gilt:
  - $d_i \leq d_j$
  - $t+\Delta_t \leq d_i$ (ansonsten wäre der $Plan_x$ nicht zulässig)
  - Da die Pläne bis zum Zeitpunkt t identisch sind und P_i im $Plan_{EDF}$ zum Zeitpunkt t ausgeführt sind, kann der Prozess P_i im $Plan_x$ noch nicht beendet sein.
    $\Rightarrow \exists\, t'>t+\Delta_t: (i=P(Plan_x,t')=P(Plan_x,t'+\Delta_t) \wedge t'+\Delta_t \leq d_i \leq d_j$
    $\Rightarrow$ Die Aktivitätsphase von P_i im Zeitintervall $t'+\Delta_t$ und P_j im Zeitintervall $t+\Delta_t$ können ohne Verletzung der Zeitbedingungen getauscht werden $\Rightarrow$ Übergang von $Plan_x(t)$ zu $Plan_x(t+\Delta_t)$

Versagen von EDF bei unterschiedlichen Bereitzeiten

- Haben die Prozesse unterschiedliche Bereitzeiten, so kann EDF versagen.
- Beispiel: P_1: $r_1=0$; $e_1=4$; $d_1=8$ P2:$r_2=2$;$e_2=2$;$d_2=5$



- **Anmerkung**: Jedes prioritätsgesteuerte, **nicht präemptive** Verfahren versagt bei diesem Beispiel, da ein solches Verfahren nie eine Zuweisung des Prozessors an einen laufbereiten Prozess , falls ein solcher vorhanden ist, unterlässt.

Modifikationen

- Die Optimalität der Verfahren kann durch folgende Änderungen sichergestellt werden:
  - Präemptive Strategie
  - Neuplanung beim Erreichen einer neuen Bereitzeit
  - Einplanung nur derjenigen Prozesse, deren Bereitzeit erreicht ist

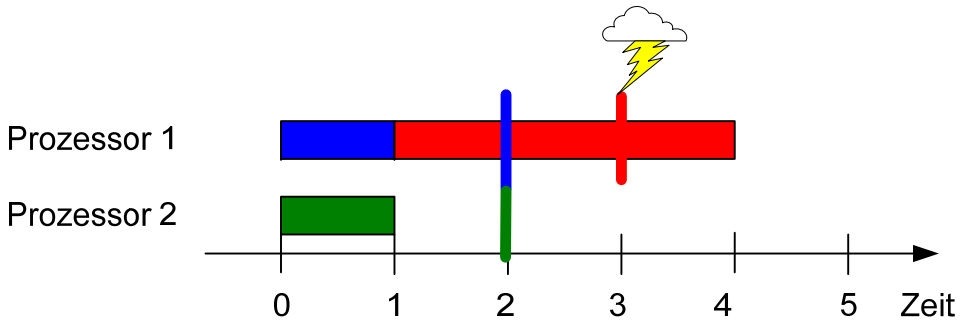
  → Entspricht einer Neuplanung, falls ein Prozess aktiv wird.
- Bei Least Slack Time müssen zusätzlich Zeitscheiben für Prozesse mit gleichem Spielraum eingeführt werden, um ein ständiges Hin- und Her Schalten zwischen Prozessen zu verhindern.
- Generell kann gezeigt werden, dass die Verwendung von EDF die Anzahl der Kontextwechsel in Bezug auf Online-Scheduling-Verfahren minimiert (siehe Paper von Buttazzo)

Zeitplanung auf Mehrprozessorsystemen

- Fakten zum Scheduling auf Mehrprozessorsystemen (Beispiele folgen):
  - EDF nicht optimal, egal ob präemptiv oder nicht präemptive Strategie
  - LST ist nur dann optimal, falls alle Bereitzeitpunkte r_i gleich
  - korrekte Zuteilungsalgorithmen erfordern das Abarbeiten von Suchbäumen mit NP-Aufwand oder geeignete Heuristiken
  - Beweisidee zur Optimalität von LST bei gleichen Bereitzeitpunkten: Der Prozessor wird immer dem Prozess mit geringstem Spielraum zugewiesen , d.h. wenn bei LST eine Zeitüberschreitung auftritt, dann auch, falls die CPU einem Prozess mit größerem Spielraum zugewiesen worden wäre.

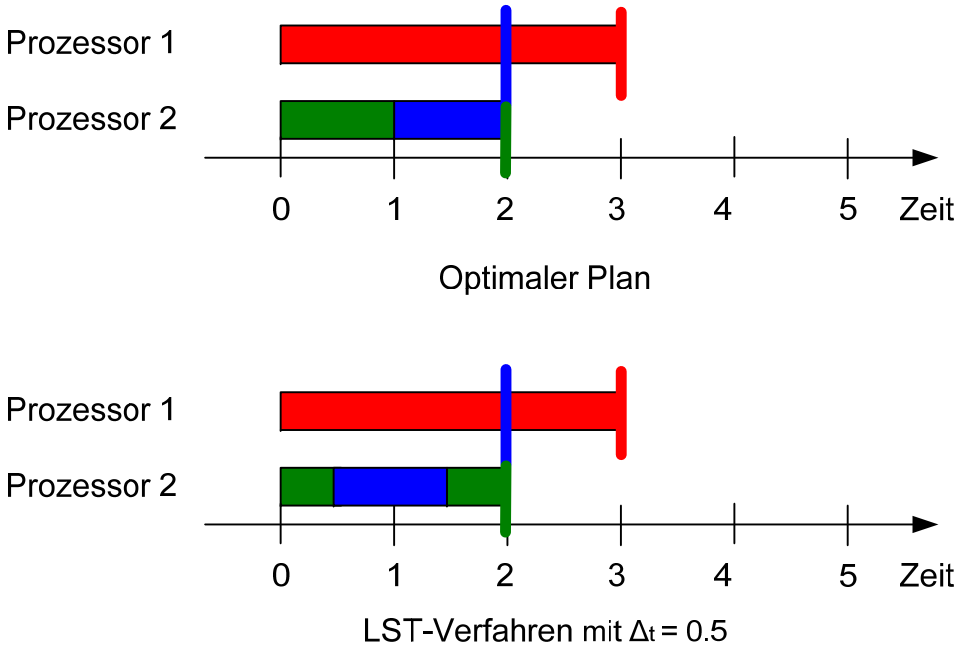
Beispiel: Versagen von EDF

- 2 Prozessoren, 3 Prozesse:
  P_1: r_1=0; e_1=3; d_1=3;
  P_2: r_2=0; e_2=1; d_2=2;
  P_3: r_3=0; e_3=1; d_3=2;



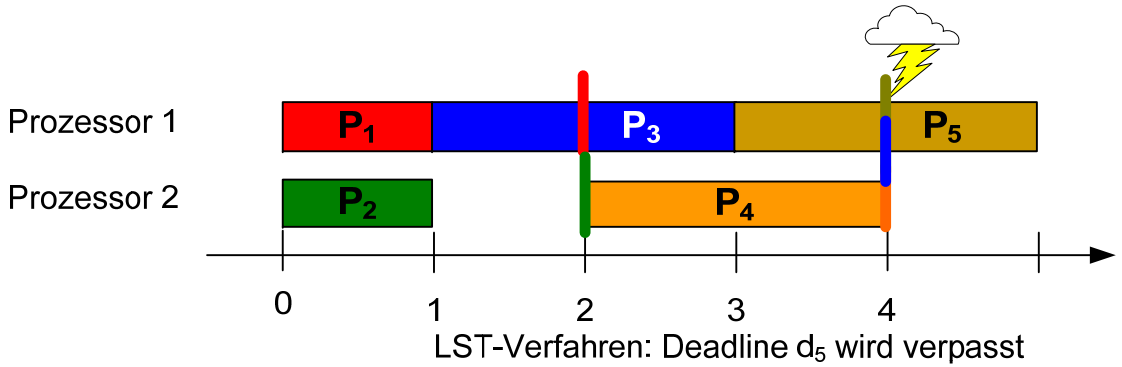
EDF-Verfahren: Deadline d1 wird verpasst

Beispiel: Optimaler Plan und LST-Verfahren

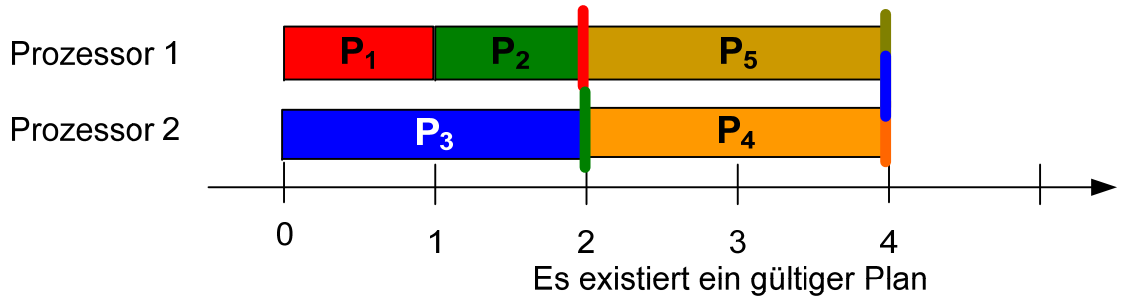


Beispiel: Versagen von LST

- 2 Prozessoren, 5 Prozesse, Δ_t=0,5:
  P_1: r_1=0; e_1=1; d_1=2;
  P_2: r_2=0; e_2=1; d_2=2;
  P_3: r_3=0; e_3=2; d_3=4;
  P_4: r_4=2; e_4=2;d_4=4;
  P_5: r_5=2; e_5=2;d_5=4;



LST-Verfahren: Deadline d_5 wird verpasst



Es existiert ein gültiger Plan

Versagen von präemptiven Schedulingverfahren

- Jeder präemptiver Algorithmus versagt, wenn die Bereitstellzeiten unterschiedlich sind und nicht im Voraus bekannt sind.
  Beweis:
  - n CPUs und n-2 Prozesse ohne Spielraum (n-2 Prozesse müssen sofort auf n-2 Prozessoren ausgeführt werden) $\Rightarrow$ Reduzierung des Problems auf 2-Prozessor-Problem
  - Drei weitere Prozesse sind vorhanden und müssen eingeplant werden.
  - Die Reihenfolge der Abarbeitung ist von der Strategie abhängig, in jedem Fall kann aber folgender Fall konstruiert werden, so dass:
    - es zu einer Fristverletzung kommt,
    - aber ein gültiger Plan existiert.

Fortsetzung Beweis

- Szenario:
  P_1: r_1=0; e_1=1; d_1=1;
  P_2: r_2=0; e_2=2; d_2=4;
  P_3: r_3=0; e_3=1; d_3=2;

  ⟶ Prozess P_1 (kein Spielraum) muss sofort auf CPU1 ausgeführt werden.

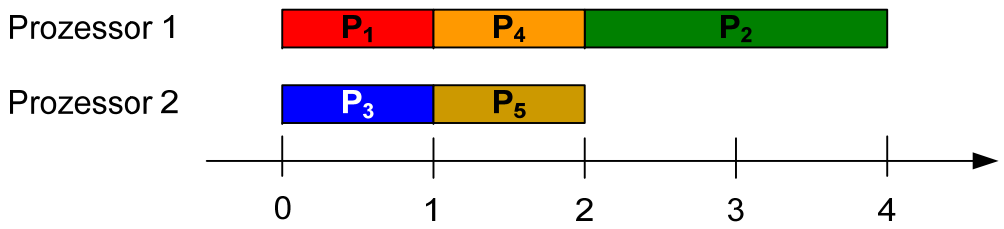
  ⟶ Es gibt je nach Strategie zwei Fälle zu betrachten: P2 oder P3 wird zunächst auf CPU2 ausgeführt.

1. Fall

- P_2 wird zum Zeitpunkt 0 auf CPU2 ausgeführt.
  - Zum Zeitpunkt 1 muss dann P_3 (ohne Spielraum) ausgeführt werden.
  - Zum Zeitpunkt 1 treffen aber zwei weitere Prozesse P_4 und P_5 mit Frist 2 und Ausführungsdauer 1 ein.

  → Es gibt drei Prozesse ohne Spielraum, aber nur zwei Prozessoren.

- Aber es gibt einen gültigen Ausführungsplan:

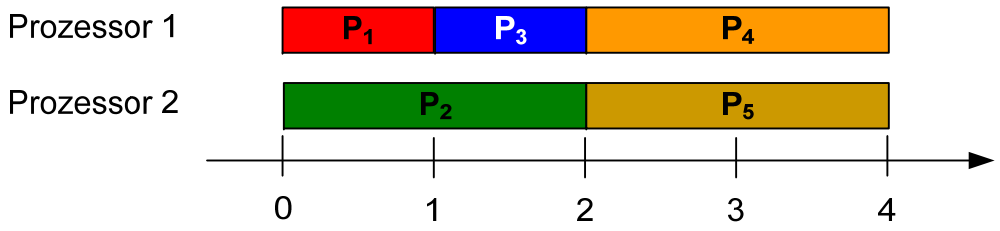


2. Fall

- P_3 wird zum Zeitpunkt 0 auf CPU2 ausgeführt.
  - Zum Zeitpunkt 1 sind P_1 und P_3 beendet.
  - Zum Zeitpunkt 1 beginnt P_2 seine Ausführung.
  - Zum Zeitpunkt 2 treffen aber zwei weitere Prozesse P_4 und P_5 mit Deadline 4 und Ausführungsdauer 2 ein.

  $\Rightarrow$ Anstelle der zum Zeitpunkt 2 noch notwendigen 5 Ausführungseinheiten sind nur 4 vorhanden.

- Aber es gibt einen gültigen Ausführungsplan:



Strategien in der Praxis

- Die Strategien EDF und LST werden in der Praxis selten angewandt. Gründe:
  - In der Realität sind keine abgeschlossenen Systeme vorhanden (Alarme, Unterbrechungen erfordern eine dynamische Planung)
  - Bereitzeiten sind nur bei zyklischen Prozessen oder Terminprozessen bekannt.
  - Die Abschätzung der Laufzeit sehr schwierig ist (siehe Exkurs).
  - Synchronisation, Kommunikation und gemeinsame Betriebsmittel verletzen die Forderung nach Unabhängigkeit der Prozesse.

Ansatz in der Praxis

- Zumeist basiert das Scheduling auf der Zuweisung von statischen Prioritäten.
- Prioritäten werden zumeist durch natürliche Zahlen zwischen 0 und 255 ausgedrückt. Die höchste Priorität kann dabei sowohl 0 (z.B. in VxWorks) als auch 255 (z.B. in POSIX) sein.
- Die Priorität ergibt sich aus der Wichtigkeit des technischen Prozesses und der Abschätzung der Laufzeiten und Spielräume. Die Festlegung erfolgt dabei durch den Entwickler.
- Bei gleicher Priorität wird zumeist eine FIFO-Strategie (d.h. ein Prozess läuft solange, bis er entweder beendet ist oder aber ein Prozess höherer Priorität eintrifft) angewandt.
  Alternative Round Robin: Alle laufbereiten Prozesse mit der höchsten Priorität erhalten jeweils für eine im Voraus festgelegte Zeitdauer die CPU.

Scheduling

Zeitplanen periodischer Prozesse

Zeitplanung periodischer Prozesse

- Annahmen für präemptives Scheduling
  - Alle Prozesse treten periodisch mit einer Frequenz f_i auf.
  - Die Frist eines Prozesses entspricht dem nächsten Startpunkt.
  - Sind die maximalen Ausführungszeiten e_i bekannt, so kann leicht errechnet werden, ob ein ausführbarer Plan existiert.
  - Die für einen Prozesswechsel benötigten Zeiten sind vernachlässigbar.
  - Alle Prozesse sind unabhängig.
- Eine sehr gute Zusammenfassung zu dem Thema Zeitplanung periodischer Prozesse liefert Giorgio C. Buttazzo in seinem Paper „Rate Monotonic vs. EDF: Judgement Day“ (http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf).

Einplanbarkeit

- Eine notwendige Bedingung zur Einplanbarkeit ist die Last:
  - Last eines einzelnen Prozesses: $\rho_i = e_i * f_i$
  - Gesamte Auslastung bei n Prozessen:

$$\rho = \sum_{i=0}^{n} \rho_i$$

  - Bei m Prozessoren ist $\rho < m$ eine notwendige aber nicht ausreichende Bedingung.

Zeitplanen nach Fristen

- **Ausgangspunkt**: Wir betrachten Systeme mit einem Prozessor und Fristen der Prozesse, die relativ zum Bereitzeitpunkt deren Perioden entsprechen, also $d_i=1/f_i$.
- **Aussage**: Die Einplanung nach Fristen ist optimal.
- **Beweisidee**: Vor dem Verletzen einer Frist ist die CPU nie unbeschäftigt $\Rightarrow$ die maximale Auslastung liegt bei 100%.
- Leider wird aufgrund von diversen Vorurteilen EDF selten benutzt.
- Betriebssysteme unterstützen selten ein EDF-Scheduling
  $\Rightarrow$ Die Implementierung eines EDF-Scheduler auf der Basis von einem prioritätsbasierten Scheduler ist nicht effizient zu implementieren (Ausnahme: zeitgesteuerte Systeme)

Zeitplanung nach Raten

- Rate Monotonic bezeichnet ein Scheduling-Verfahren mit festen Prioritäten Prio(i), die sich proportional zu den Frequenzen verhalten.

  → Prozesse mit hohen Raten werden bevorzugt. Das Verfahren ist optimal, falls eine Lösung mit statischen Prioritäten existiert. Verfahren mit dynamischen Prioritäten können allerdings eventuell bessere Ergebnisse liefern.

- Liu und Layland haben 1973 in einer Worst-Case-Analyse gezeigt, dass Ratenplanung sicher erfolgreich ist, falls bei n Prozessen auf einem Prozessor gilt:

$$\rho \leq \rho_{max} = n*(2^{1/n} - 1)$$

$$\lim_{n \to \infty} \rho_{max} = \ln 2 \approx 0{,}69$$

- Derzeit zumeist verwendetes Scheduling-Verfahren im Bereich von periodischen Prozessen.

Scheduling

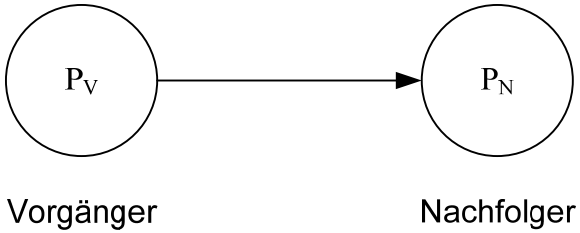
Planen abhängiger Prozesse

Allgemeines zum Scheduling in Echtzeitsystemen

- Grundsätzlich kann der Prozessor neu vergeben werden, falls:
  - ein Prozess endet,
  - ein Prozess in den blockierten Zustand (z.B. wegen Anforderung eines blockierten Betriebsmittels) wechselt,
  - eine neuer Prozess gestartet wird,
  - ein Prozess vom blockierten Zustand in den Wartezustand wechselt (z.B. durch die Freigabe eines angeforderten Betriebmittels durch einen anderen Prozess)
  - oder nach dem Ablauf eines Zeitintervals, siehe z.B. Round Robin.
- Hochpriorisierte Prozesse dürfen in Echtzeitsystemen nicht durch unwichtigere Prozesse behindert werden $\Rightarrow$ Die Prioritätsreihenfolge muss bei allen Betriebsmitteln (CPU, Semaphore, Netzkommunikation, Puffer, Peripherie) eingehalten werden, d.h. Vordrängen in allen Warteschlangen.

Präzedenzsysteme

- Zur Vereinfachung werden zunächst Systeme betrachtet, bei denen die Bereitzeiten der Prozesse auch abhängig von der Beendigung anderer Prozesse sein können.
- Mit Hilfe von Präzedenzsystemen können solche Folgen von voneinander abhängigen Prozessen beschrieben werden.
- Zur Beschreibung werden typischerweise Graphen verwendet:



- Der Nachfolgerprozess kann also frühestens beim Erreichen der eigenen Bereitzeit **und** der Beendigung der Ausführung des Vorgängerprozesses ausgeführt werden.

Probleme bei Präzedenzsystemen

- Bei der Planung mit Präzedenzsystemen muss auch berücksichtigt werden, dass die Folgeprozesse noch rechtzeitig beendet werden können.
- Beispiel:
  P_V: $r_V=0$; $e_V=1$; $d_V=3$;
  P_N: $r_N=0$; $e_N=3$; $d_N=5$;
- Falls die Frist von P_V voll ausgenutzt wird, kann der Prozess P_N nicht mehr rechtzeitig beendet werden.

  ⟶ Die Fristen müssen entsprechend den Prozessabhängigkeiten neu berechnet werden (Normalisierung von Präzedenzsystemen).

Normalisierung von Präzedenzsystemen

- Anstelle des ursprünglichen Präzedenzsystems PS wird ein normalisiertes Präzedenzsystem PS' mit folgenden Eigenschaften:
  - $\forall\rangle$: $e'_i = e_i$
  - $$\forall i : d'_i = \begin{cases} d_i, falls N_i = \emptyset \\ min(d_i, min(d'_q - e'_q | q \in N_i) \end{cases}$$

    wobei N_i die Menge der Nachfolger im Präzedenzgraph bezeichnet und d'_i rekursiv beginnend bei Prozessen ohne Nachfolger berechnet wird.
  - Falls die Bereitzeiten von externen Ereignissen abhängig sind, gilt $r'_i = r_i$. Sind die Bereitzeiten dagegen abhängig von der Beendigung der Prozesse, so ergeben sie sich aus dem konkreten Scheduling.

  eingeführt.

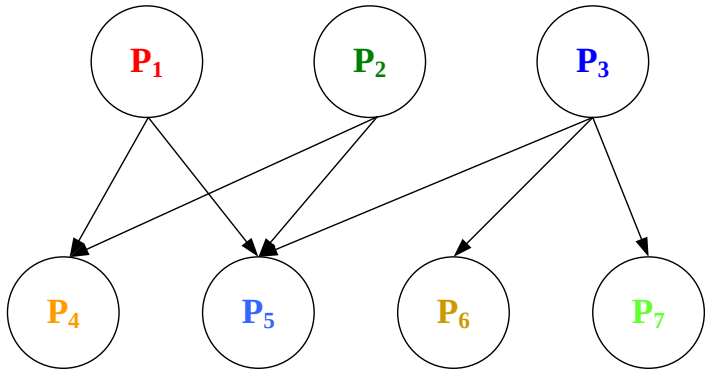
  $\rightarrow$ Ein Präzedenzsystem ist nur dann planbar, falls das zugehörige normalisierte Präzedenzsystem planbar ist.

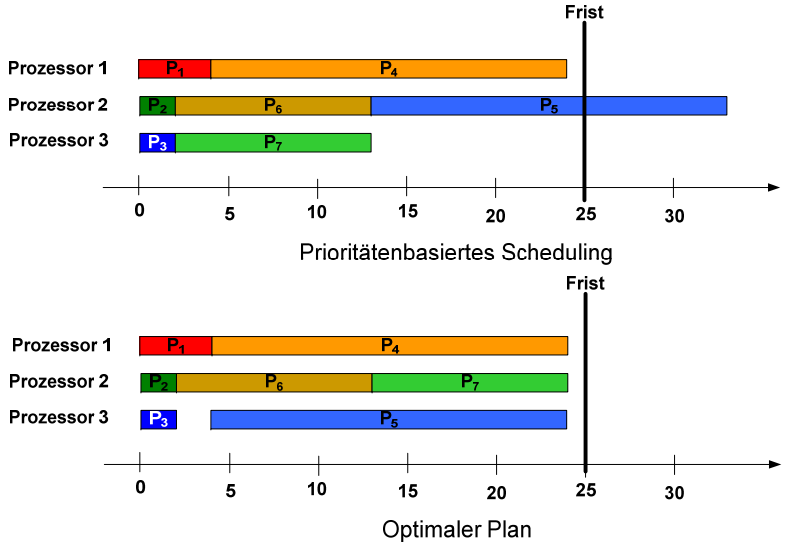
Anomalien bei nicht präemptiven Scheduling

- Wird zum Scheduling von Präzedenzsystemen ein nicht präemptives prioritätenbasiertes Verfahren (z.B. EDF, LST) verwendet, so können Anomalien auftreten:
  - Durch Hinzufügen eines Prozessors kann sich die gesamte Ausführungszeit verlängern.
  - Durch freiwilliges Warten kann die gesamte Ausführungszeit verkürzt werden.

Beispiel: Verkürzung durch freiwilliges Warten

- Beispiel: 3 Prozessoren, 7 Prozesse (r_i=0, e_1=4; e_2=2; e_3=2; e_4=20; e_5=20; e_6=11; e_7=11, d_i=25), Präzedenzgraph:

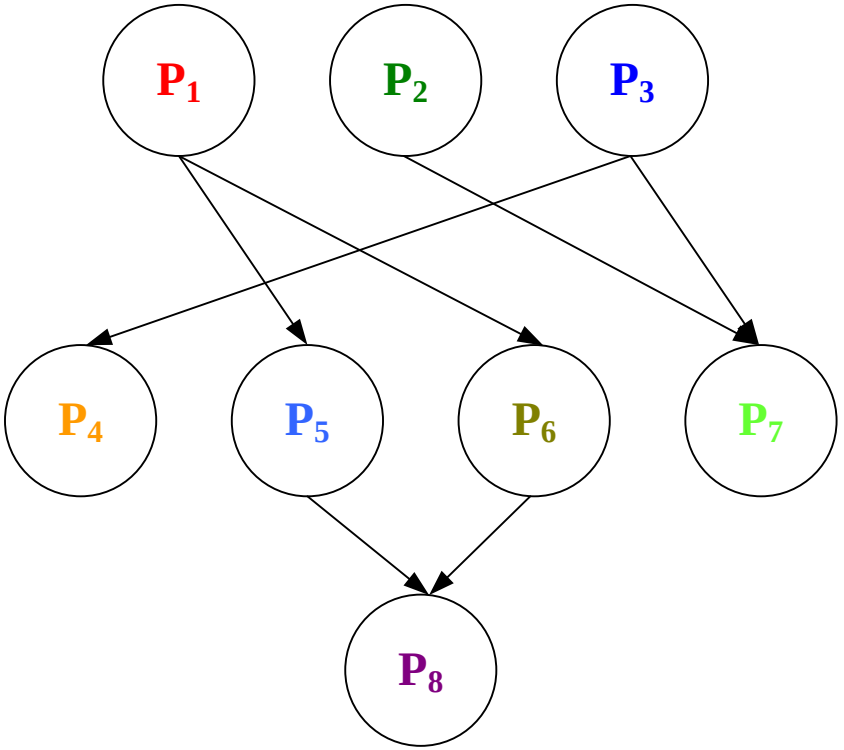




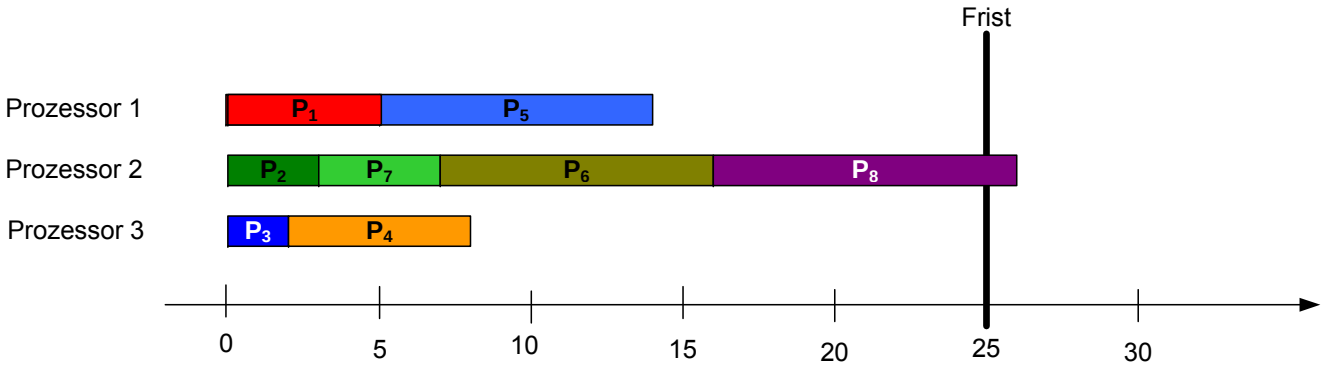
Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II

- Beispiel:
  - 2 bzw. 3 Prozessoren
  - 8 Prozesse:
    - Startzeiten $r_i=0$
    - Ausführungszeiten
      $e_1=5$;
      $e_2=3$;
      $e_3=2$;
      $e_4=6$;
      $e_5=9$;
      $e_6=9$;
      $e_7=4$,
      $e_8=10$
    - Frist: $d_i=25$

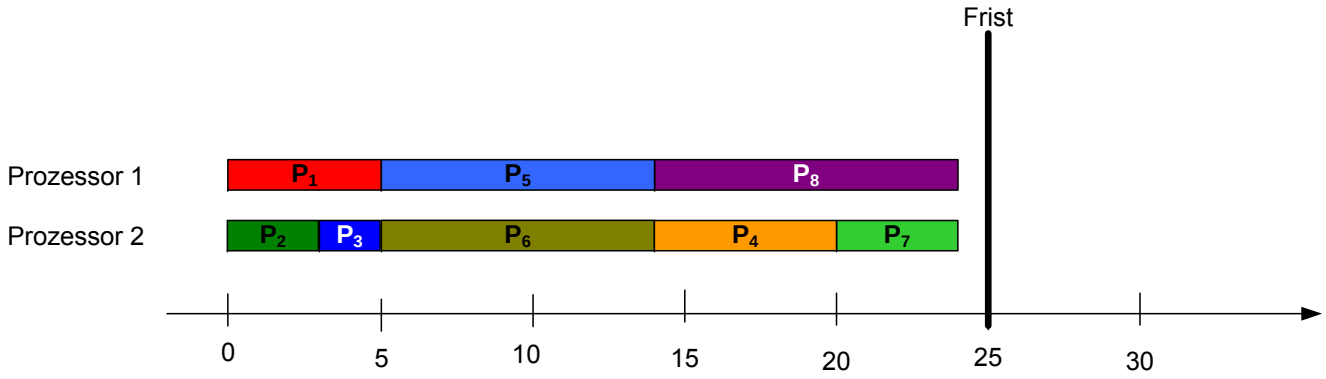
- Präzedenzgraph:



Beispiel: Laufzeitverlängerung durch zusätzlichen Prozessor II



Prioritätenbasiertes Scheduling (LST) auf 3 Prozessoren



Prioritätenbasiertes Scheduling (LST) auf 2 Prozessoren

Scheduling

Problem: Prioritätsinversion

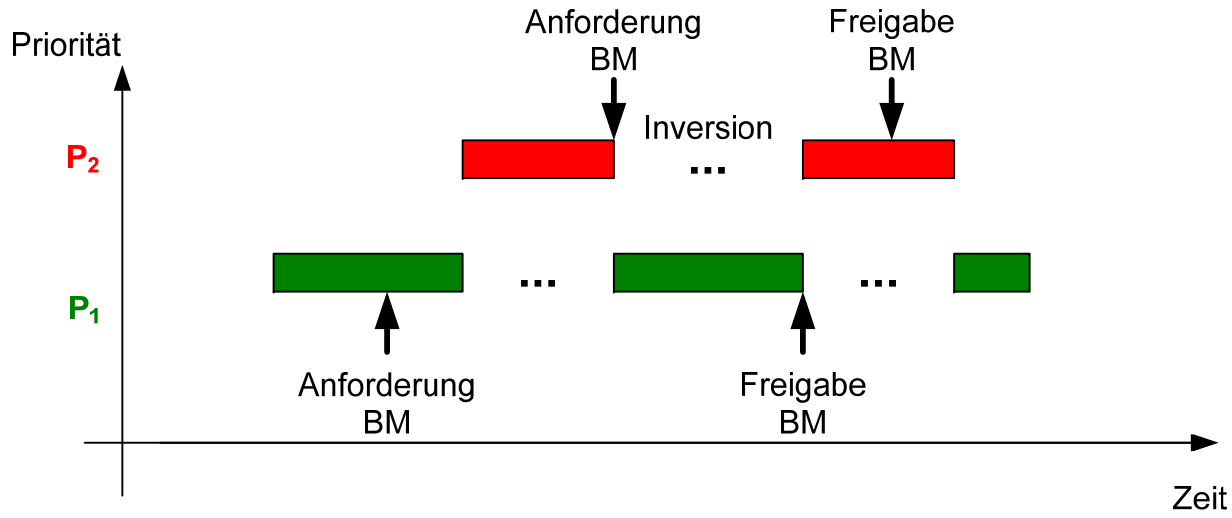
Motivation des Problems

- Selbst auf einem Einprozessoren-System mit präemptiven Scheduling gibt es Probleme bei voneinander abhängigen Prozessen.
- Abhängigkeiten können diverse Gründe haben:
  - Prozesse benötigen Ergebnisse eines anderen Prozesses
  - Betriebsmittel werden geteilt
  - Es existieren kritische Bereiche, die durch Semaphoren oder Monitoren geschützt sind.
- Gerade aus den letzten zwei Punkten entstehen einige Probleme:
  - Die Prozesse werden unter Umständen unabhängig voneinander implementiert $\Rightarrow$ das Verhalten des anderen Prozesses ist nicht bekannt.
  - Bisher haben wir noch keinen Mechanismus zum Umgang mit blockierten Betriebsmitteln kennengelernt, falls hochpriore Prozesse diese Betriebsmittel anfordern.

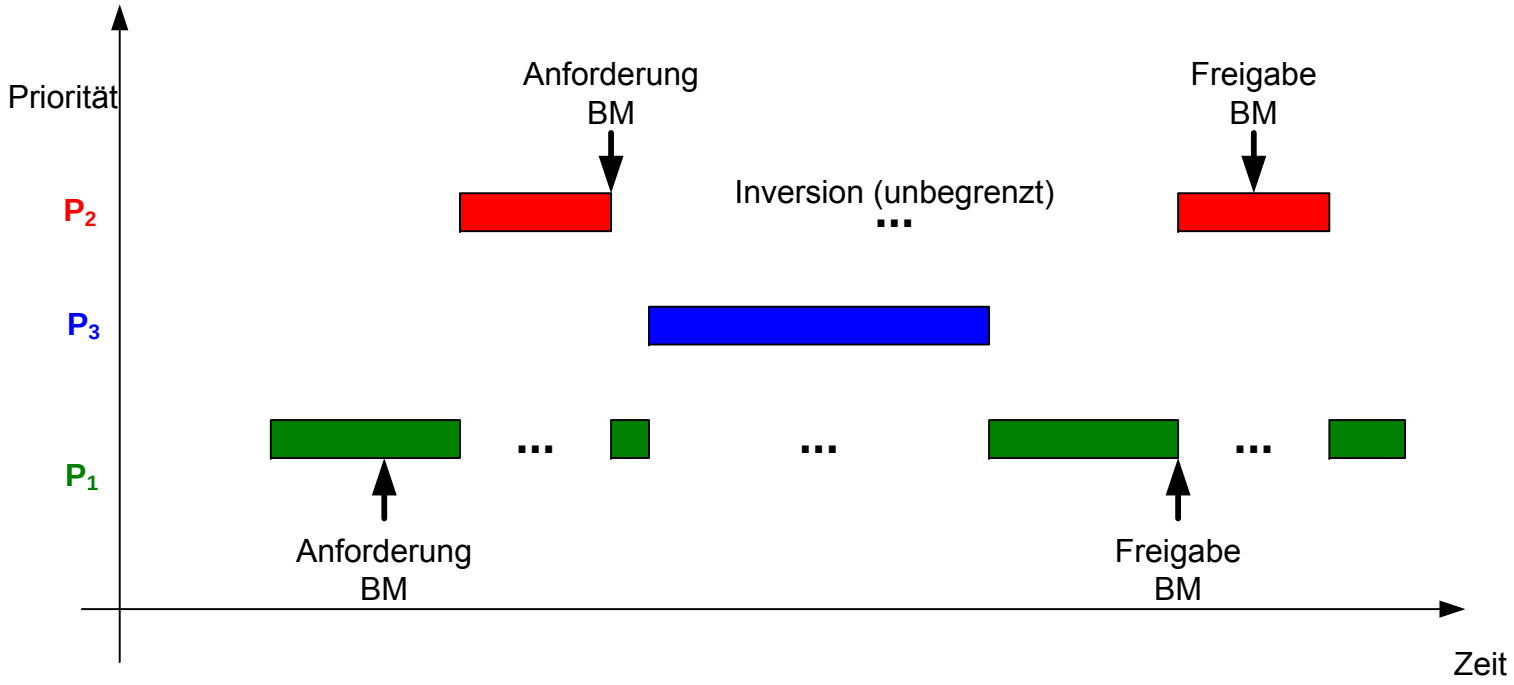
Prioritätsinversion

- **Definition**: Das Problem der **Prioritätsinversion** bezeichnet Situationen, in denen ein Prozess mit niedriger Priorität einen höherpriorisierten Prozess blockiert.
- Dabei unterscheidet man zwei Arten der Prioritätsinversion:
  - **begrenzte** (bounded) Prioritätsinversion: die Inversion ist durch die Dauer des kritischen Bereichs beschränkt.
  - **unbegrenzte** (unbounded) Prioritätsinversion: durch weitere Prozesse kann der hochpriorisierte Prozess auf unbestimmte Dauer blockiert werden.
- Während das Problem der begrenzten Prioritätsinversion aufgrund der begrenzten Zeitdauer akzeptiert werden kann (muss), ist die unbegrenzte Prioritätsinversion in Echtzeitsystemen unbedingt zu vermeiden.

Begrenzte Inversion



Unbegrenzte Inversion



Reales Beispiel: Mars Pathfinder

- **System**: Der Mars Pathfinder hatte zur Speicherung der Daten einen Informationsbus (vergleichbar mit Shared Memory). Der Informationsbus war durch einen binären Semaphore geschützt. Ein Bus Management Prozess verwaltete den Bus mit hoher Priorität. Ein weiterer Prozess war für die Sammlung von geologischen Daten eingeplant. Dieser Task lief mit einer niedrigen Priorität. Zusätzlich gab es noch einen Kommunikationsprozess mittlerer Priorität.
- **Symptome**: Das System führte in unregelmäßigen Abständen einen Neustart durch. Daten gingen dadurch verloren.
- **Ursache**: Der binäre Semaphore war nicht mit dem Merkmal zur Unterstützung von Prioritätsvererbung (siehe später) erzeugt worden. Dadurch kam es zur Prioritätsinversion. Ein Watchdog (Timer) erkannte eine unzulässige Verzögerung des Bus Management Prozesses und führte aufgrund eines gravierenden Fehlers einen Neustart durch.

Ansätze zur Lösung der Prioritätsinversion

- Es existieren verschiedene Ansätze um das Problem der unbegrenzten Prioritätsinversion zu begrenzen:
  - Prioritätsvererbung (priority inheritance)
  - Prioritätsobergrenzen (priority ceiling)
  - Unmittelbare Prioritätsobergrenzen (immediate priority ceiling)
- Anforderungen an Lösungen:
  - leicht zu implementieren
  - Anwendungsunabhängige Implementierung
  - Eventuell Ausschluss von Verklemmungen

Prioritätsvererbung (priority inheritance)

- Sobald ein Prozess höherer Priorität ein Betriebsmittel anfordert, das ein Prozess mit niedrigerer Priorität besitzt, erbt der Prozess mit niedrigerer Priorität die höhere Priorität. Nachdem das Betriebsmittel freigegeben wurde, fällt die Priorität wieder auf die ursprüngliche Priorität zurück.

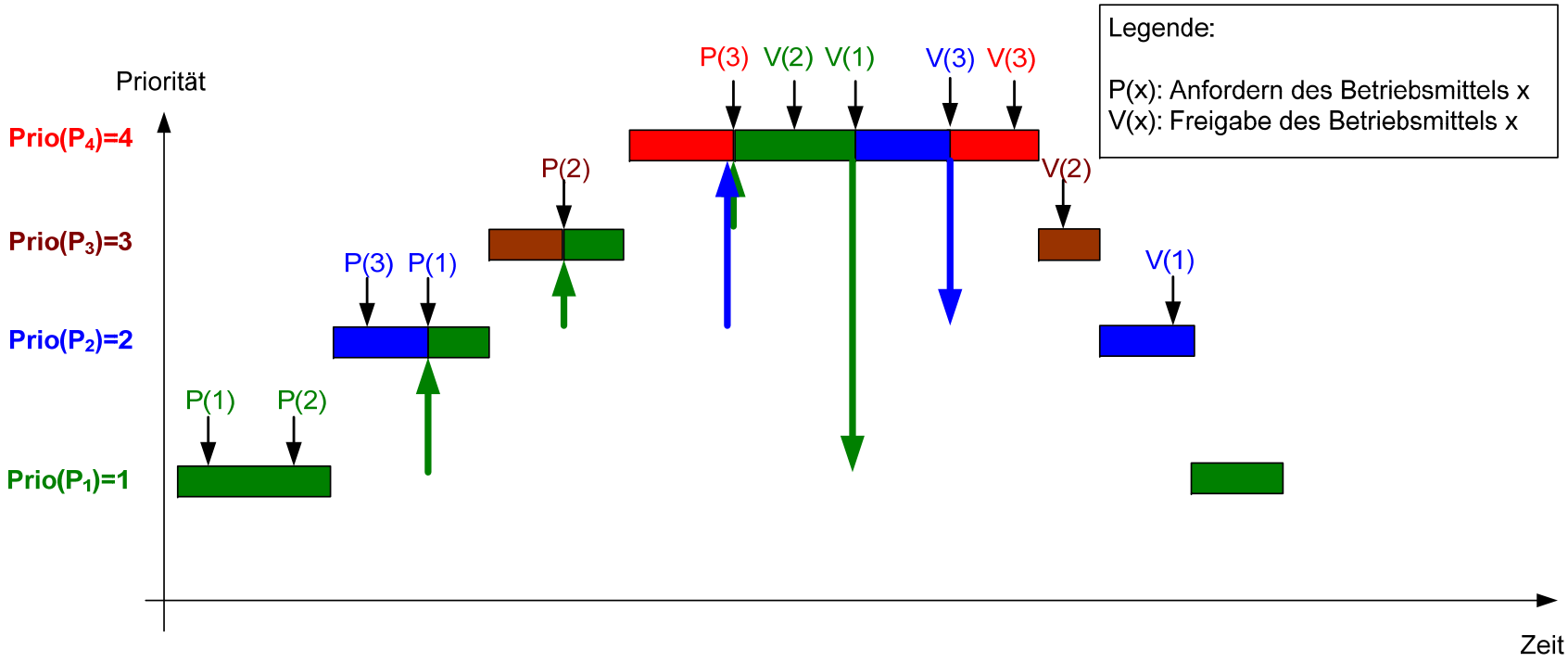
  $\rightarrow$ Unbegrenzte Prioritätsinversion wird verhindert.

  $\rightarrow$ Die Dauer der Blockade wird durch die Dauer des kritischen Abschnittes beschränkt.

  $\rightarrow$ Blockierungen werden hintereinander gereiht (Blockierungsketten).

  $\rightarrow$ Verklemmungen durch Programmierfehler werden nicht verhindert.

Beispiel: Prioritätsvererbung



Prioritätsobergrenzen (priority ceiling)

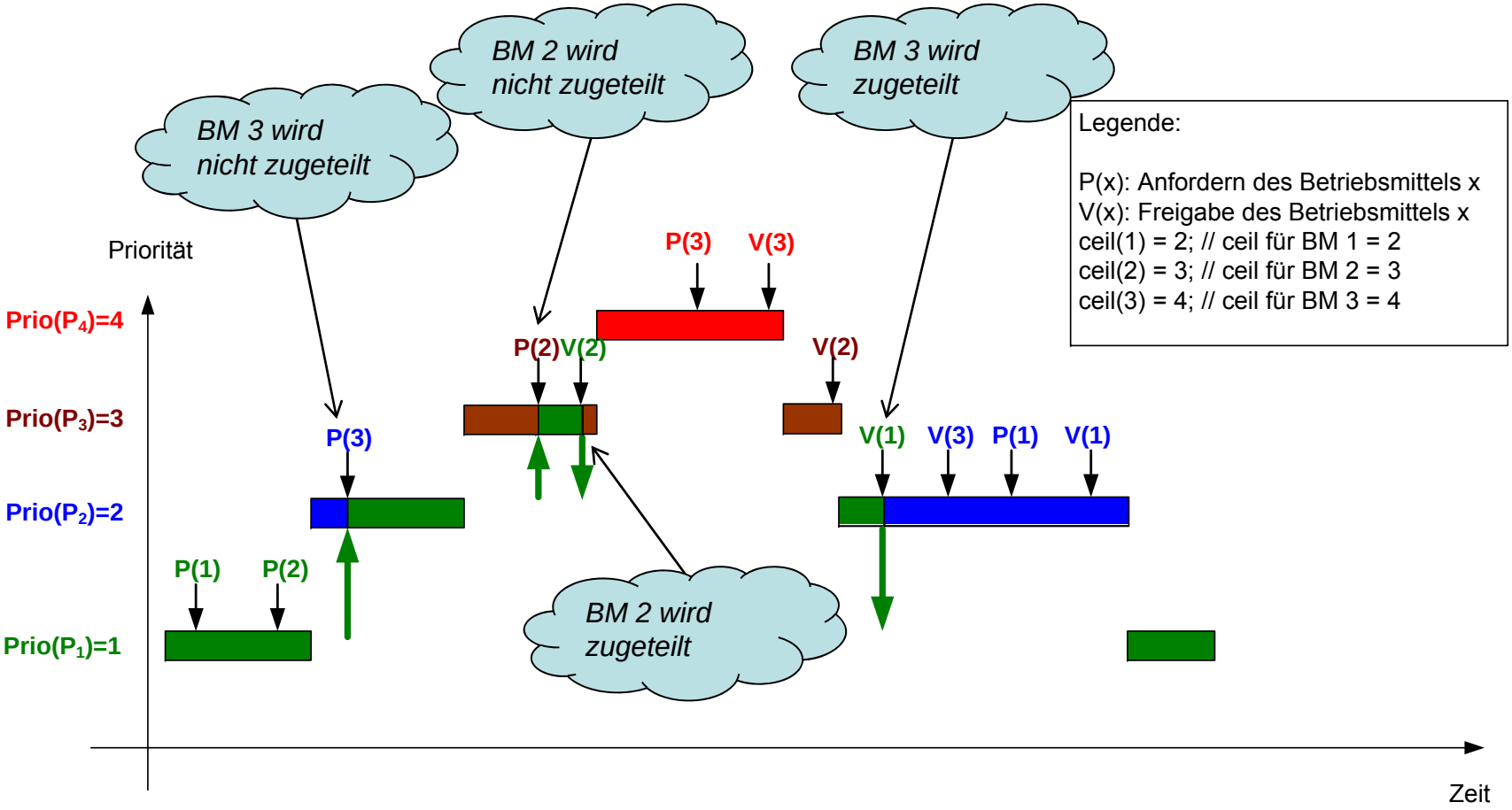
- Jedem Betriebsmittel (z.B. Semaphor) s wird eine Prioritätsgrenze ceil(s) zugewiesen, diese entspricht der maximalen Priorität der Prozesse, die auf s zugreifen.
  - Ein Prozess p darf ein BM nur blockieren, wenn er von keinem anderen Prozess, der andere BM besitzt, verzögert werden kann.
  - Die aktuelle Prioritätsgrenze für Prozess p ist
    aktceil (p)= max{ ceil(s) | s ∈ locked} mit locked = Menge aller von **anderen Prozessen** blockierten BM
  - Prozess p darf Betriebsmittel s benutzen, wenn für seine aktuelle Priorität aktprio gilt: aktprio(p) > aktceil(p)
  - Andernfalls gibt es genau einen Prozess, der s besitzt. Die Priorität dieses Prozesses wird auf aktprio gesetzt.

  → Blockierung nur für die Dauer eines kritischen Abschnitts

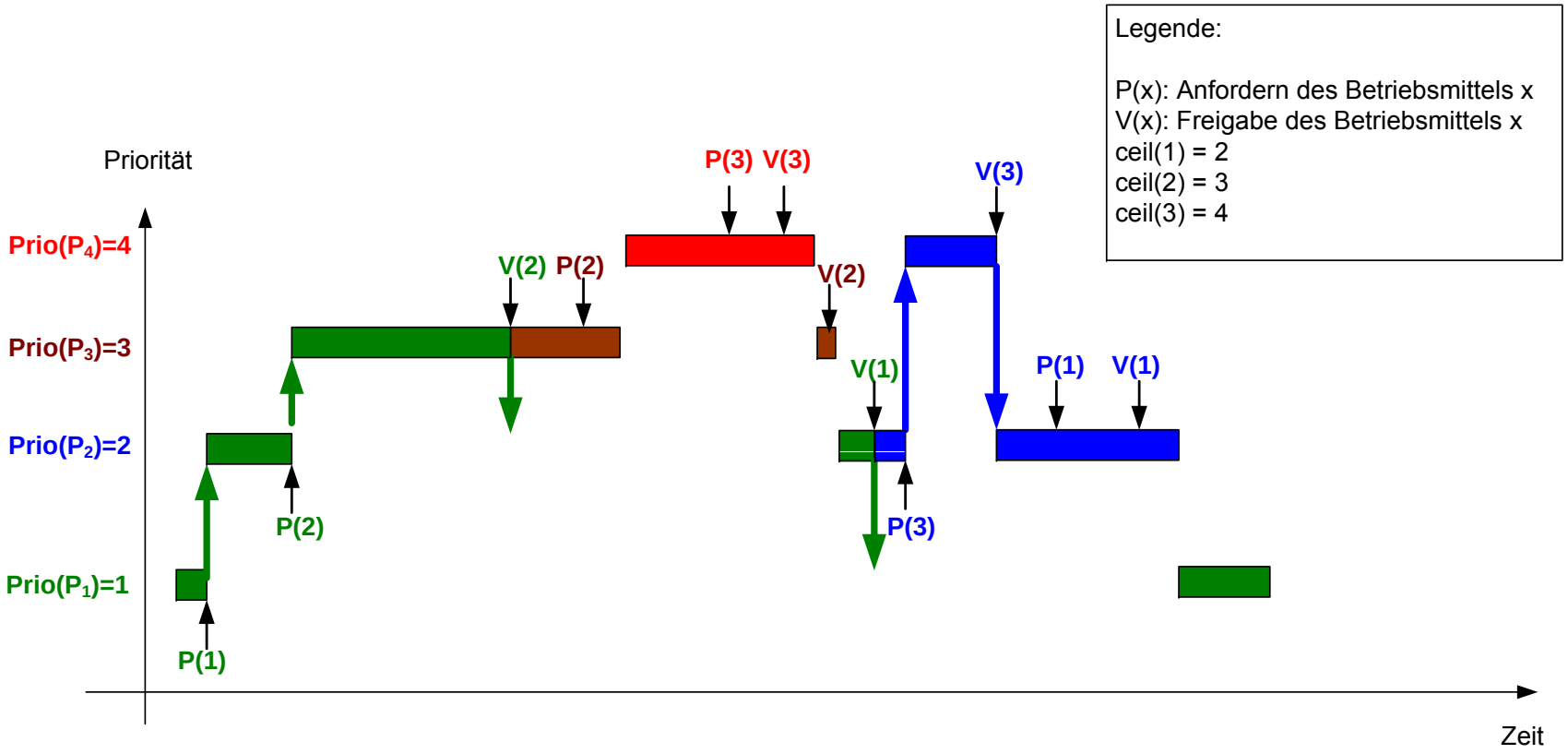
  → Verhindert Verklemmungen

  → schwieriger zu realisieren, zusätzlicher Prozesszustand
- Vereinfachtes Protokoll: **Immediate priority ceiling**: Prozesse, die ein Betriebsmittel s belegen, bekommen sofort die Priorität ceil(s) zugewiesen.

Beispiel: Prioritätsobergrenzen



Beispiel: Immediate Priority Ceiling

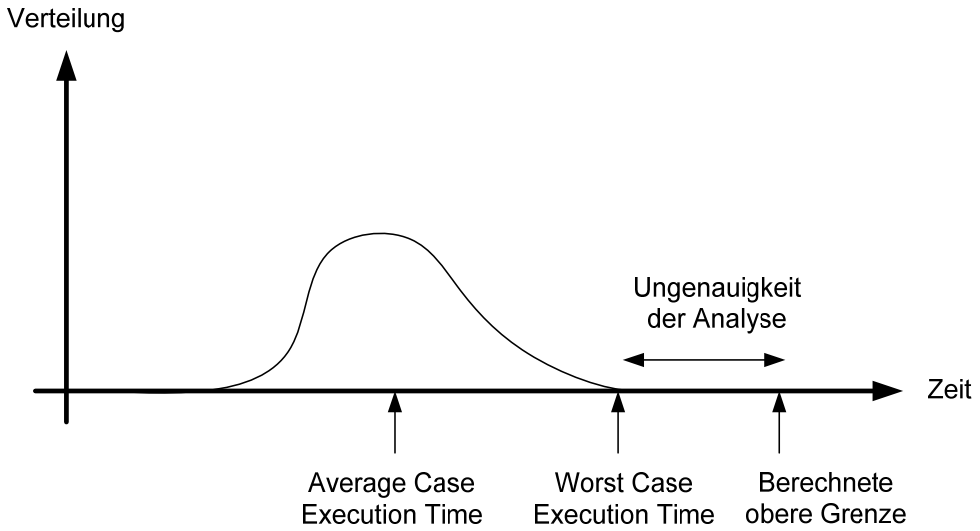


Scheduling

Exkurs: WCET (Worst Case Execution Time) - Analyse

WCET Analyse

- Ziel der Worst Case Execution Time Analyse ist die Abschätzung der maximalen Ausführungszeit einer Funktion



- Die Laufzeit ist abhängig von den Eingabedaten, dem Prozessorzustand, der Hardware,…

Probleme bei der WCET Analyse

- Bei der Abschätzung der maximalen Ausführungszeiten stößt man auf einige Probleme:
  - Es müssen unter anderem die Auswirkungen der Hardwarearchitektur, des Compilers und des Betriebssystems untersucht werden. Dadurch erschwert sich eine Vorhersage.
  - Zudem dienen viele Eigenschaften der Beschleunigung des allgemeinen Verhaltens, jedoch nicht des Verhaltens im schlechtesten Fall, z.B.:
    - Caches, Pipelines, Virtual Memory
    - Interruptbehandlung, Präemption
    - Compileroptimierungen
    - Rekursion

	Zugriffszeit	Größe
Register	0.25 ns	500 bytes
Cache	1 ns	64 KB
Hauptspeicher	100 ns	512 MB
Festplatte	5 ms	100 GB

Zugriffszeiten für verschiedene Speicherarten

  - Noch schwieriger wird die Abschätzung falls der betrachtete Prozess von der Umgebung abhängig ist.

Unterscheidungen bei der WCET-Analyse

- Die Analyse muss auf unterschiedlichen Ebenen erfolgen:
  - Was macht das Programm?
  - Was passiert im Prozessor?

- Bei der Analyse werden zwei Methoden unterschieden:
  - **statische** WCET Analyse: Untersuchung des Programmcodes
  - **dynamische** Analyse: Bestimmung der Ausführungszeit anhand von verschiedenen repräsentativen Durchläufen

Statische Analyse

- Aufgaben:
  - Bestimmung von Ausführungspfaden in der Hochsprache
  - Transformation der Pfade in Maschinencode
  - Bestimmung der Laufzeit einzelner Maschinencodesequenzen
- Probleme:
  - Ausführungspfade lassen sich oft schlecht vollautomatisch ableiten (zu pessimistisch, zu komplex)
  - Ausführungspfade häufig abhängig von Eingabedaten
- Lösungsansatz: Annotierung der Pfade mit Beschränkungen (wie z.B. maximale Schleifendurchläufe)

Dynamische Analyse

- Statische Analysen können zumeist die folgenden Wechselwirkungen nicht berücksichtigen:
  - Wechselwirkungen mit anderen Programmen (siehe z.B. wechselseitiger Ausschluss)
  - Wechselwirkungen mit dem Betriebssystem (siehe z.B. Caches)
  - Wechselwirkungen mit der Umgebung (siehe z.B. Interrupts)
  - Wechselwirkungen mit anderen Rechnern (siehe z.B. Synchronisation)
- Durch dynamische Analysen können diese Wechselwirkungen abgeschätzt werden.
- Problem: Wie können die Testläufe sinnvoll ausgewählt werden.

Dimensionierung der Rechenleistungen

- Aufstellen der Worst-Case Analyse:
  - Rechenaufwand für bekannte periodische Anforderungen
  - Rechenaufwand für erwartete sporadische Anforderungen
  - Zuschlag von 100% oder mehr zum Abfangen von Lastspitzen
- Unterschied zu konventionellen Systemen:
  - keine maximale Auslastung des Prozessors
  - keine Durchsatzoptimierung
  - Abläufe sollen determiniert abschätzbar sein

Scheduling

Zusammenfassung

Zusammenfassung

- Kenntniss der Schedulingkriterien (Einhalten von Fristen, Fairness,…) , sowie der verschiedenen Prozessparameter (Startzeit, Laufzeit, Deadline, Priorität).
- Klassische Verfahren (EDF, LST, RM) und Anforderungen an die Optimalität dieser Verfahren
- Problem der Prioritätsinversion, sowie Lösungsverfahren
- Problematik der WCET-Analyse

Kapitel 8 (vorgezogen)

Echtzeitfähige Kommunikation

Inhalt

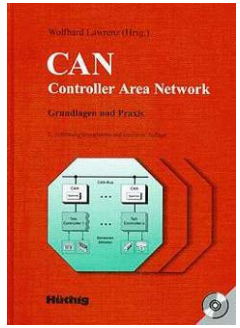
- Grundlagen
- Medienzugriffsverfahren und Vertreter
  - CSMA-CD: Ethernet
  - CSMA-CA: CAN-Bus
  - Tokenbasierte Protokolle: Token Ring, FDDI
  - Zeitgesteuerte Protokolle: TTP
  - Real-Time Ethernet

Literatur

- Spezifikationen:
  - TTTech Computertechnik AG, Time Triggered Protocol TTP/C High-Level Specification Document, 2003 (http://www.vmars.tuwien.ac.at/projects/ttp/)
  - http://www.can-cia.org/
  - http://standards.ieee.org/getieee802/portfolio.html

Andrew S. Tanenbaum, Computernetzwerke, 2005

Wolfhard Lawrenz: CAN Controller Area Network. Grundlagen und Praxis, 2000

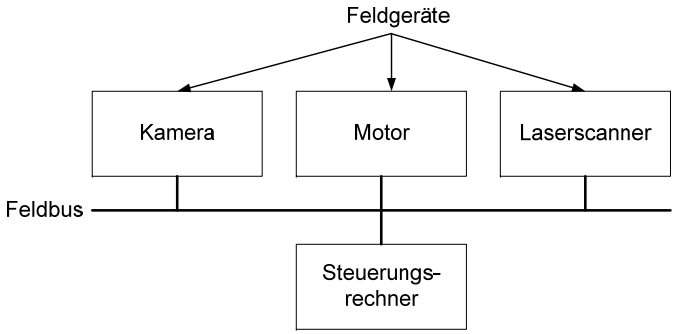


Anforderungen

- Echtzeitsysteme unterscheiden sich in ihren Anforderungen an die Kommunikation von Standardsystemen.
- Anforderungen speziell von Echtzeitsystemen:
  - vorhersagbare maximale Übertragungszeiten
  - kleiner Nachrichtenjitter
  - garantierte Bandbreiten
  - effiziente Protokolle: kurze Latenzzeiten
  - teilweise Fehlertoleranz
- Kriterien bei der Auswahl:
  - maximale Übertragungsrate
  - maximale Netzwerkgröße (Knotenanzahl, Länge)
  - Materialeigenschaften (z.B. für Installation)
  - Störungsempfindlichkeit (auch unter extremen Bedingungen)
  - Kosten, Marktproduktpalette

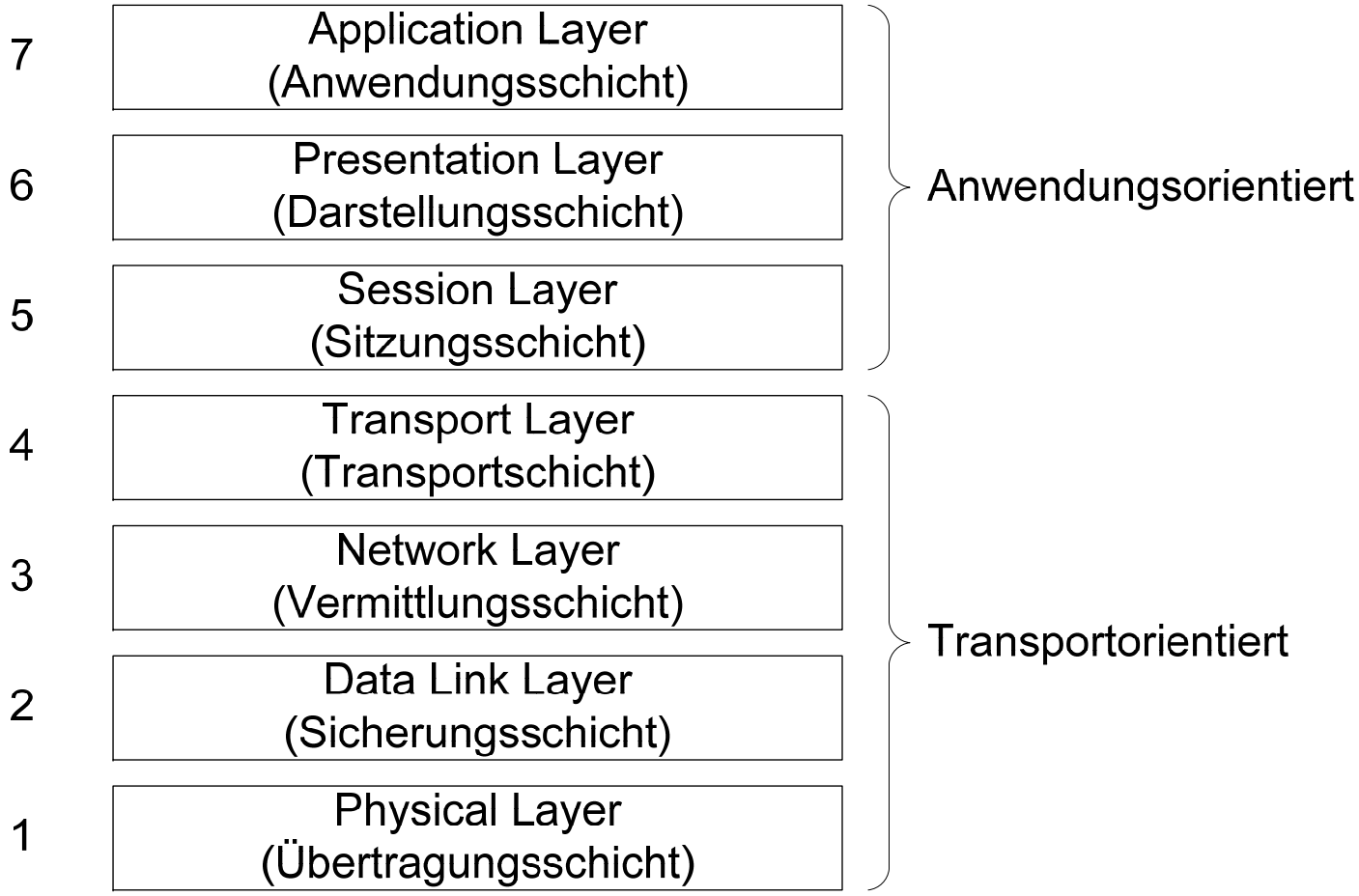
Definition Feldbus

- Die Kommunikation in Echtzeitsystemen erfolgt häufig über **Feldbusse**:



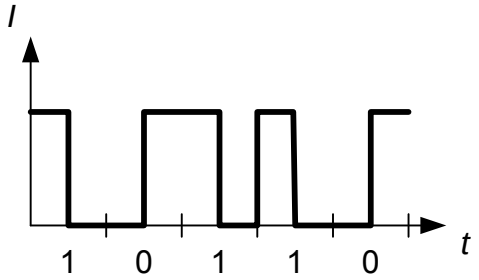
- Feldgeräte sind dabei Sensoren/Aktoren, sowie Geräte zur Vorverarbeitung der Daten.
- Der Feldbus verbindet die Feldgeräte mit dem Steuerungsgerät.
- Beobachtung: echtzeitkritische Nachrichten sind in der Regel kürzer als unkritische Nachrichten.
- Es existiert eine Vielzahl von Feldbus-Entwicklungen: MAP (USA - General Motors), FIP (Frankreich), PROFIBUS (Deutschland), CAN (Deutschland – Bosch), …

Schichtenmodell: ISO/OSI-Modell

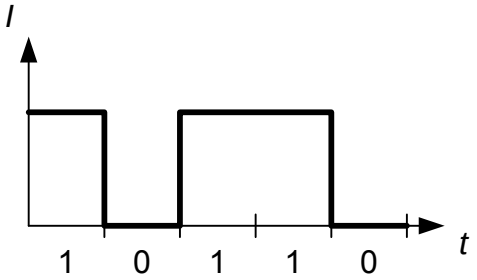


Beschreibung der einzelnen Schichten: Übertragungsschicht

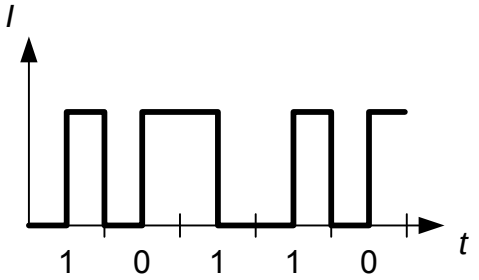
- Aufgaben:
  - Bitübertragung auf physikalischen Medium
  - Festlegung der Medien
    - elektrische, optische Signale, Funk
    - Normung von Steckern
  - Festlegung der Übertragungsverfahren/Codierung
    - Interpretation der Pegel
    - Festlegung der Datenrate



Manchester-Code



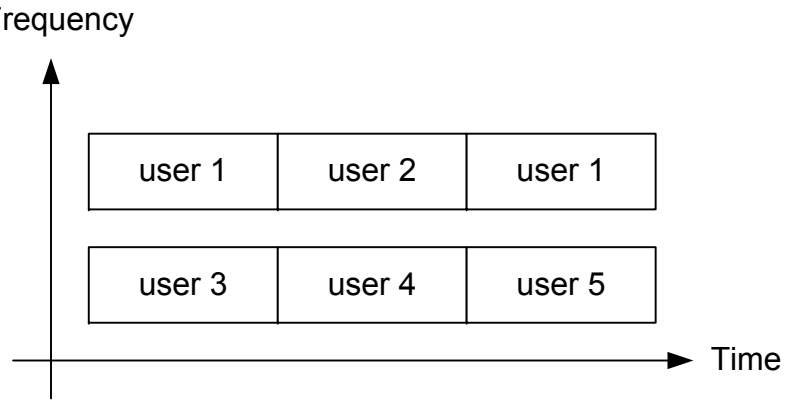
Non-return-to-zero Code



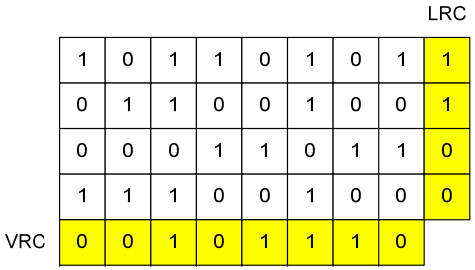
Differentieller Manchester-Code

Beschreibung der einzelnen Schichten: Sicherungsschicht

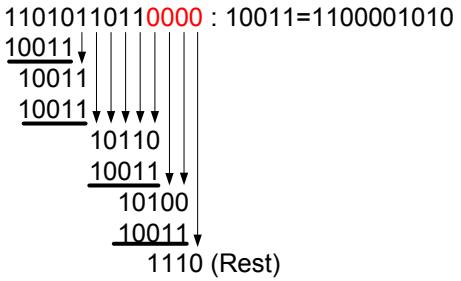
- Aufgaben:
  - Fehlererkennung
    - Prüfsummen
    - Paritätsbits
  - Aufteilung der Nachricht in Datenpakete
  - Regelung des Medienzugriffs
  - Flusskontrolle



TDMA+FDMA



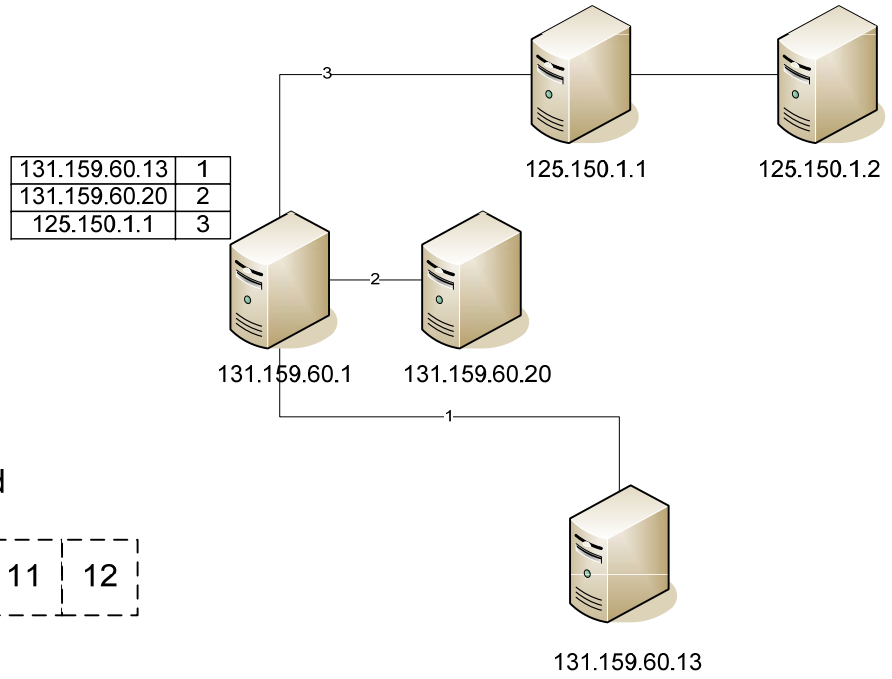
Paritätsbits

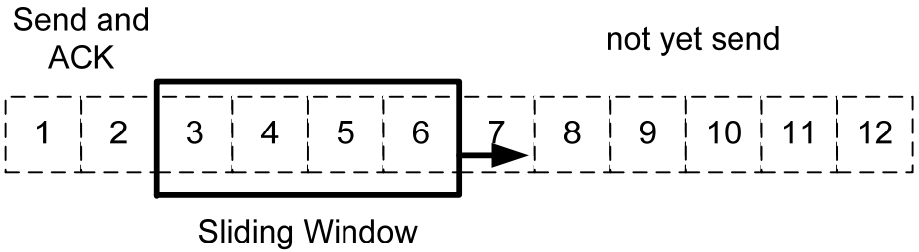


CRC-Verfahren

Beschreibung der einzelnen Schichten: Vermittlungsschicht

- Aufgaben:
  - Aufbau von Verbindungen
  - Weiterleitung von Datenpaketen
    - Routingtabellen
    - Flusskontrolle
    - Netzwerkadressen





Sliding Window Protokoll

Weitere Schichten

- Transportschicht:
  - Transport zwischen Sender und Empfänger (End-zu-End-Kontrolle)
  - Segmentierung von Datenpaketen
  - Staukontrolle (congestion control)
- Sitzungsschicht:
  - Auf- und Abbau von Verbindungen auf Anwendungsebene
  - Einrichten von Check points zum Schutz gegen Verbindungsverlust
  - Dienste zur Organisation und Synchronisation des Datenaustauschs
  - Spezifikation von Mechanismen zum Erreichen von Sicherheit (z.B. Passwörter)
- Darstellungsschicht:
  - Konvertierung der systemabhängigen Daten in unabhängige Form
  - Datenkompression
  - Verschlüsselung
- Anwendungsschicht:
  - Bereitstellung anwendungsspezifischer Übertragungs- und Kommunikationsdienste
  - Beispiele:
    - Datenübertragung
    - E-Mail
    - Virtual Terminal
    - Remote Login
    - Video-On-Demand
    - Voice-over-IP

Schichten in Echtzeitsystemen

- Die Nachrichtenübertragungszeit setzt sich aus folgenden Komponenten zusammen:
  - Umsetzung der Protokolle der einzelnen Schichten durch den Sender
  - Wartezeit auf Medienzugang
  - Übertragungszeit auf Medium
  - Entpacken der Nachricht in den einzelnen Schichten durch den Empfänger

⇒ Jede zu durchlaufende Schicht verlängert die Übertragungszeit und vergrößert die zu sendenden Daten.

⇒ in Echtzeitsystemen wir die Anzahl der Schichten zumeist reduziert auf:

  - Anwendungsschicht
  - Sicherungsschicht
  - Physikalische Schicht

Echtzeitfähige Kommunikation

Medienzugriffsverfahren

Problemstellung

- Zugriffsverfahren regeln die Vergabe des Kommunikationsmediums an die einzelnen Einheiten.
- Das Kommunikationsmedium kann in den meisten Fällen nur exklusiv genutzt werden, Kollisionen müssen zumindest erkannt werden um Verfälschungen zu verhindern.
- Zugriffsverfahren können dabei in unterschiedliche Klassen aufgeteilt werden:
  - Erkennen von Kollisionen, Beispiel: CSMA/CD
  - Vermeiden von Kollisionen, Beispiel: CSMA/CA
  - Ausschluss von Kollisionen, Beispiel: token-basiert, TDMA

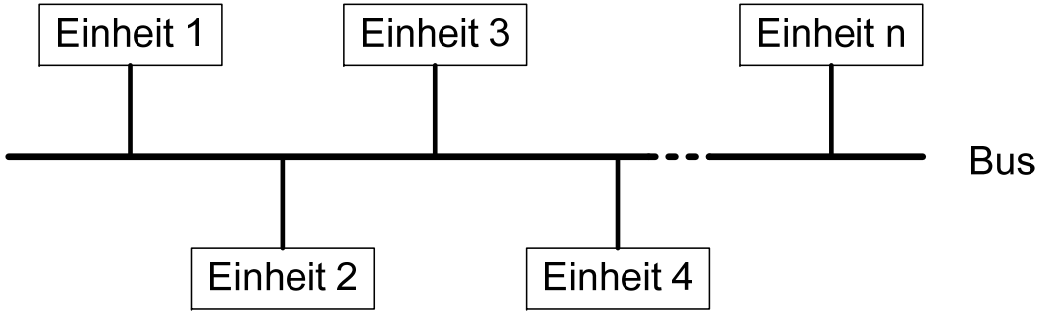
Echtzeitfähige Kommunikation

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

Vertreter: Ethernet (nicht echtzeitfähig!)

CSMA/CD

- CSMA/CD: Carrier Sense Multiple Access - Collision Detection
  - alle am Bus angeschlossenen Einheiten können die aktuell versendeten Daten lesen (**Carrier Sense**).
  - mehrere Einheiten dürfen Daten auf den Bus schreiben (**Multiple Access**).



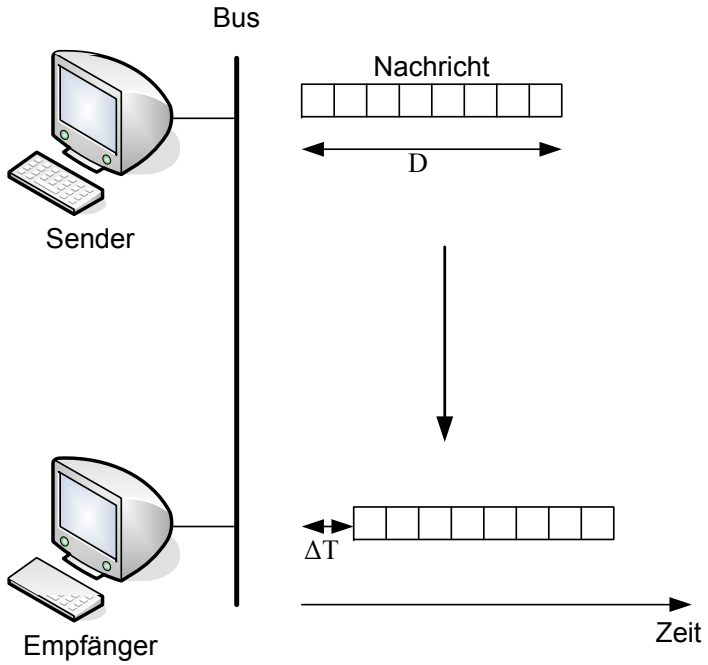
  - Während der Übertragung überprüft der sendende Knoten gleichzeitig das Resultat auf dem Bus, ergibt sich eine Abweichung, so wird eine Kollision angenommen (**Collision Detection**)

CSMA/CD: Ablauf

- Beschrieben wird im Folgenden das 1-persistente CSMA/CD- Verfahren (Spezifikation in der Norm IEEE 802.3)
- Ablauf zum Senden eines Paketes:
  1. Test, ob Leitung frei ist (**carrier sense**)
  2. Falls Leitung für die Zeitdauer eines IFS (**inter frame spacing**) frei ist, wird die Übertragung gestartet, ansonsten Fortfahren mit Schritt 5.
  3. Übertragung der Daten inklusive Überwachung der Leitung. Im Fall einer Kollision: senden eines **JAM**-Signals, fortfahren mit Schritt 5.
  4. Übertragung erfolgreich beendet: Benachrichtige höhere Schicht, Beendigung
  5. Warten bis Leitung frei ist
  6. Sobald Leitung frei: weitere zufälliges Warten (z.B. **Backoff-Verfahren**) und Neustarten mit Schritt 1, falls maximale Sendeversuchsanzahl noch nicht erreicht.
  7. Maximale Anzahl an Sendeversuchen erreicht: Fehlermeldung an höhere Schicht.

Kollisionen

- Um Kollisionen rechtzeitig zu erkennen muss die Signallaufzeit ΔT deutlich kleiner als die Nachrichtenübertragungsdauer D sein.
- Das Störsignal (JAM) wird geschickt um alle anderen Nachrichten auf die Kollision aufmerksam zu machen $\Rightarrow$ Verkürzung der Zeit zur Kollisionserkennung
- Würden die Rechner nach einer Kollision nicht eine zufällige Zeit warten, käme es sofort zu einer erneuten Kollision.
- Lösung im Ethernet: Die Sender wählen eine zufällige Zahl d aus dem Interval $[0...2^i]$, mit i = Anzahl der bisherigen Kollisionen (Backoff-Verfahren).
  $\Rightarrow$ Mit ansteigendem i wird eine Kollision immer unwahrscheinlicher.
  $\Rightarrow$ Bei i = 16 wird die Übertragung abgebrochen und ein Systemfehler vermutet.



TCP vs. UDP

- TCP (Transmission Control Protocol) ist ein zuverlässiges, verbindungsorientiertes Transportprotokoll:
  - Vor der Übertragung der Daten wird zunächst eine Verbindung zwischen Sender und Empfänger aufgebaut (Handshake).
  - Datenverluste werden erkannt und automatisch behoben durch Neuversenden des entsprechenden Datenpakets.

  ⇒ Aufgrund von unvorhersehbaren Verzögerungen (Backoff-Verfahren) und hohem Overhead ist TCP nicht für den Einsatz in Echtzeitsystemen geeignet.
  - Weiteres Problem: Slow Start der Congestion Control Strategie von TCP/IP ⇒ zu Beginn der Übertragung wird nicht die volle Bandbreite ausgenutzt
- UDP (User Datagram Protocol) ist ein minimales, verbindungsloses Netzprotokoll:
  - Verwendung vor allem bei Anwendungen mit kleinen Datenpaketen (Overhead zum Verbindungsaufbau entfällt)
  - UDP ist nicht-zuverlässig: Pakete können verloren gehen und in unterschiedlicher Reihenfolge beim Empfänger ankommen.

  ⇒ Einsatz in weichen Echtzeitsystemen, in denen der Verlust einzelner Nachrichten toleriert werden kann (z.B. Multimedia-Protokollen wie z.B. VoIP, VoD) möglich.

RTP, RTSP: Motivation

- Problem von UDP/IP in Multimediasystemen:
  - keine Möglichkeit zur Synchronisation
  - verschiedene Multimediaströme können kollidieren (z.B. in VoD)
  - Qualitätskontrolle ist wünschenswert

  $\Rightarrow$ in Multimediasystemen werden zusätzliche Protokolle (RTP, RTCP) verwendet.
- Multimediaverbindung mit RTP/RTCP
  - Zur Übertragung der **Steuerungsnachrichten** (in der Regel nicht zeitkritisch) werden zuverlässige Protokolle eingesetzt (z.B. TCP/IP)
  - Zur **Datenübertragung** wird ein **RTP** (**Real-Time Transport Protocol**)-Kanal eingesetzt.
  - Jeder RTP-Kanal wird mit einem **RTCP** (**Real-Time Control Protocol**)-Kanal zur Überwachung der Qualität verknüpft.
  - RTP/RTCP setzen in der Regel auf UDP/IP auf und sind End-zu-End-Protokolle

RTP, RTCP

- RTP:
  - Multicasting
  - Bestimmung des Datenformats (PT)
  - Zeitgebend durch Zeitstempel, die Berechnung des Jitters wird dadurch möglich
  - Möglichkeit zur Ordnung der Pakete und zum Erkennen von verlorenen Paketen durch Sequenznummer

Byte 0								Byte 1								Byte 2								Byte 3							
Bit 0	1	2	3	4	5	6	7	Bit 0	1	2	3	4	5	6	7	Bit 0	1	2	3	4	5	6	7	Bit 0	1	2	3	4	5	6	7
V=2			P	X	CC			M		PT						sequence number															
timestamp (in sample rate units)																															
sychronization source (SSRC) identifier																															
contributing source (CSRC) identifiers (optional)																															
Header Extension (optional)																															

RTP Header

- RTCP:
  - Überwachung der Qualität der Datenkanäl: versandte Daten/Pakete, verlorene Pakete, Jitter, Round trip delay
  - Unterschiedliche Pakete stehen zur Verfügung: Sender report, receiver report, source description und anwendungsspezifische Pakete

Zusammenfassung Ethernet

- Ethernet ist aufgrund des CSMA/CD Zugriffsverfahrens für harte Echtzeitsysteme nicht geeignet:
  - unbestimmte Verzögerungen durch Backoff-Verfahren
  - keine Priorisierung von Nachrichten möglich
- Aufgrund der starken Verbreitung ($\Rightarrow$ niedrige Kosten, gute Unterstützung) wird Ethernet dennoch häufig in Echtzeitsystemen eingesetzt:
  - Durch Verwendung von echtzeitfähigen Protokollen in weichen Echtzeitsystemen (z.B. Multimediakontrolle).
  - Durch Verringerung der Kollisionswahrscheinlichkeit durch Aufteilung des Netzes in verschiedene Kollisionsdomänen (z.B. switched ethernet).
- Mittlerweile werden auch diverse Implementierungen von Real-Time Ethernet eingesetzt, allerdings gibt es noch keinen allgemein anerkannten Standard (siehe Zusammenfassung/Trends).

Echtzeitfähige Kommunikation

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

Vertreter: CAN

Teilweise wird die hier vorgestellte Methode auch CSMA/CR (Collision Resolution) genannt.

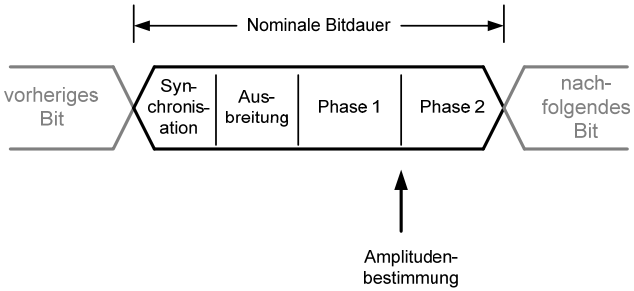
CAN-Protokoll

- Grundidee von Collision Avoidance:
  - Kollisionen werden rechtzeitig erkannt, bevor Nachrichten unbrauchbar werden
  - Wichtigere Nachrichten werden bevorzugt $\Rightarrow$ Priorisierung der Nachrichten
- Daten:
  - CAN (Controller Area Network) wurde 1981 von Intel und Bosch entwickelt.
  - Einsatzbereich: vor allem Automobilbereich, Automatisierungstechnik
  - Datenübertragungsraten von bis zu 1Mbit/s, Reichweite 1km
  - Implementierung der Schichten 1,2 und 7 des ISO/OSI-Modells

CAN: Schicht 1

- Busmedium:
  - Kupfer oder Glasfaser
  - Empfehlung Twisted Pair: Möglichkeit zur differentiellen Übertragung (robuster gegenüber Störungen)
- Codierung: NRZ-L (Non-Return-to-Zero-Level)
  - Problem mit NRZ-L: lange Sequenzen monotone Sequenzen von 0 oder 1 können zu Problemen bei der Synchronisation führen, in CAN wird deshalb nach fünf gleichen Bits ein inverses Bit eingefügt (**Bitstuffing**)
- Daten werden **bitsynchron** übertragen:

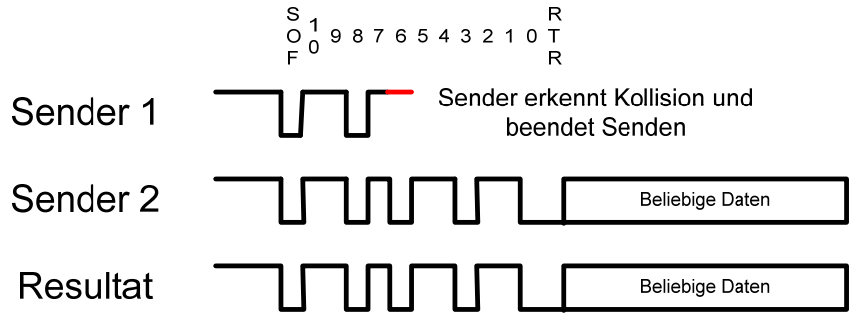
  ⇒ Datenübertragungsrate und maximale Kabellänge sind miteinander verknüpft.
  - Konfigurationsmöglichkeiten:
    - 1 MBit/s, maximale Länge: 40m
    - 500 kBit/s, maximale Länge: 100m
    - 125 kBit/s, maximale Länge: 500m
  - Maximale Teilnehmerzahl: 32-128



http://www.port.de/pdf/CAN_Bit_Timing.pdf

CAN: Schicht 2

- Realisierung eines CSMA/CA-Verfahrens:
  - Bei der Übertragung wirken Bits je nach Wert entweder **dominant** (typischerweise 0) oder **rezessiv** (1).
  - Dominante Bits überschreiben rezessive Bits, falls sie gleichzeitig gesendet werden.
  - Jedem Nachrichtentyp (z.B. Sensorwert, Kontrollnachricht) wird ein Identifikator zugewiesen, der die Wichtigkeit des Typs festlegt.
  - Jeder Identifikator sollte nur einem Sender zugewiesen werden.
  - Wie bei Ethernet wartet der Sender bis der Kanal frei ist und startet dann die Versendung der Nachricht.



  - Beim gleichzeitigen Senden zweier Nachrichten, dominiert der Identifikator des wichtigeren Nachrichtentyps, den Sender der unwichtigeren Nachricht beendet das Senden.

⇒ Verzögerung von hochprioren Nachrichten auf die maximale Nachrichtenlänge begrenzt (in Übertragung befindliche Nachrichten werden nicht unterbrochen)

CAN: Framearten

- Datenframe:
  - Versand von maximal 64bit Daten
- Remoteframe:
  - Verwendung zur Anforderung von Daten
  - Wie Datenframe, nur RTR-Feld auf 1 gesetzt
- Fehlerframe:
  - Signalisierung von erkannten Fehlerbedingungen
- Überlastframe:
  - Zwangspause zwischen Remoteframe und Datenframe

Länge in Bit	1	11	1	1	1	4	0...64	15	1	1	1	7	3
Zweck	Start of frame	Identifier (Extended CAN 27bit)	Remote Transmission Bit	Identifier Extension Bit	reserviert	Datenlängenfeld	Datenfeld	CRC-Prüfsumme	CRC Delimeter	Bestätigungsslot	Bestätigungsdelimeter	End of Frame	Intermission

CAN: Schicht 7

- Im Gegensatz zu Schicht 1 und 2 ist die Schicht 7 nicht in einer internationalen Norm spezifiziert.
- Es existieren jedoch diverse Implementierungen (z.B. CANOpen) für Dienste der Schichten 3-7 zur Realisierung von:
  - Flusskontrolle
  - Geräteadressierung
  - Übertragung größerer Datenmengen
  - Grunddienste für Anwendungen (Request, Indication, Response, Confirmation)
- Zudem gibt es Versuche eine Norm CAL (CAN Application Layer) einzuführen.
- Ziele:
  - Einheitliche Sprache zur Entwicklung von verteilten Anwendungen
  - Ermöglichung der Interaktion von CAN-Modulen unterschiedlicher Hersteller

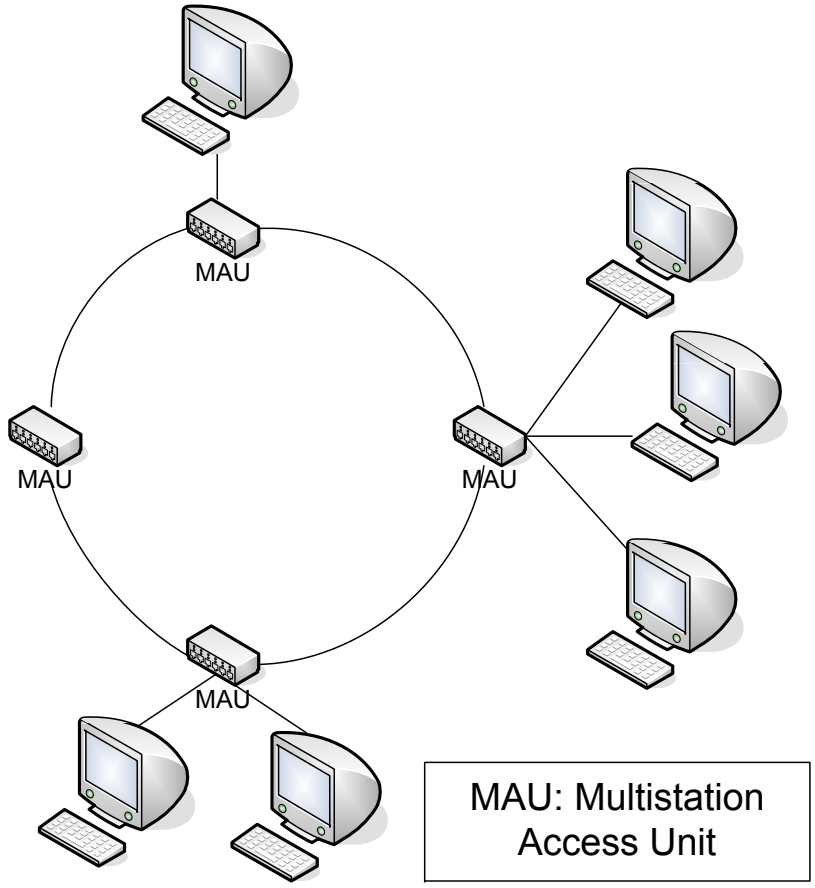
Echtzeitfähige Kommunikation

Tokenbasierte Verfahren

Vertreter: Token Ring

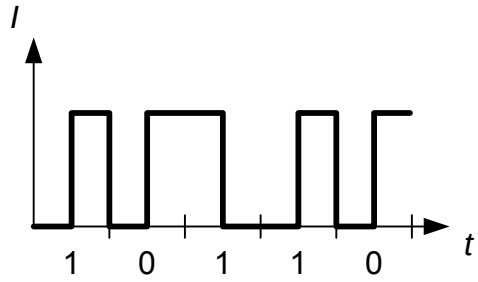
Tokenbasierte Verfahren

- Nachteil von CSMA/CA: Begrenzung der Datenrate und der Netzlänge durch Bitsynchronität
- Tokenbasierter Ansatz: Eine Einheit darf nur dann senden, wenn sie eine Berechtigung (Token) besitzt.
- Die Berechtigung wird zumeist zyklisch weitergegeben ⇒ Token Ring.
- Die Berechtigung / das Token ist dabei eine spezielle Bitsequenz.



Token Ring: Schicht 1

- Token Ring wird im Standard IEEE 802.5 spezifiziert.
- Erreichbare Geschwindigkeiten: 4 bzw. 16 MBit/s
  $\Rightarrow$ aufgrund der Kollisionsfreiheit mit den effektiven Datenübertragungsraten von 10 bzw. 100 MBit/s Ethernet vergleichbar
- Codierung:
  - differentieller Manchester-Code
  - somit selbstsynchronisierend
- Topologie:
  - Ring
  - aufgrund der möglichen Verwendung von MAUs auch sternförmige Verkabelung möglich



Differentieller Manchester-Code

Token Ring: Zugriffsverfahren

1. Die Station, die das Token besitzt, darf Daten versenden.
2. Das Datenpaket wird von Station zu Station übertragen.
3. Die einzelnen Stationen empfangen die Daten und regenerieren sie zur Weitersendung an den nächsten Nachbarn.
4. Der Empfänger einer Nachricht kopiert die Nachricht und leitet die Nachricht mit dem gesetzten C-Bit (siehe Nachrichtenaufbau) zur Empfangsbestätigung weiter.
5. Empfängt der Sender seine eigene Nachricht, so entfernt er diese aus dem Netz.
6. Nach Ende der Übertragung wird auch das Token weitergesendet (maximale Token-Wartezeit wird vorher definiert, Standardwert: 10ms)
7. Im 16 MBit/s Modus wird das Token direkt im Anschluß an das Nachrichtenpaket versendet (**early release**) $\Rightarrow$ es können sich gleichzeitig mehrere Token im Netz befinden

Token Ring: Prioritäten

- Token Ring unterstützt Prioritäten:
  - Insgesamt gibt es 8 Prioritätsstufen (3 Bit)
  - Jeder Station wird eine Priorität zugewiesen.
  - Der Datenrahmen besitzt ebenfalls einen Speicherplatz für die Priorität.
  - Eine Station kann in die Priorität in dem Prioritätsfeld von Nachrichten vormerken, allerdings darf die Priorität nur erhöht werden.
  - Stationen dürfen Tokens nur dann annehmen, wenn ihre Priorität mindestens so hoch ist, wie die Priorität des Tokens.
  - Applet zum Ablauf: http://www.nt.fh-koeln.de/vogt/mm/tokenring/tokenring.html

Token Ring: Token Paket

- Das Token besteht aus:
  - Startsequenz (1 Byte, JK0JK000)
    - J, K: Codeverletzungen entsprechend Manchester-Code (kein Übergang in Taktmitte)
  - Zugriffskontrolle (1 Byte, PPPTMRRR)
    - P: Zugriffspriorität
    - T: Tokenbit (0: freies Token, 1:Daten)
    - M: Monitorbit
    - R: Reservierungspriorität
  - Endsequenz (1 Byte, JK1JK1IE)
    - I: Zwischenrahmenbit (0: letztes Paket, 1: weitere Pakete folgen)
    - E: Fehlerbit (0: fehlerfrei, 1: Fehler entdeckt)

Token Ring: Tokenrahmen

- Der Datenrahmen besteht aus:
  - Startsequenz wie Token
  - Zugriffskontrolle wie Token
  - Rahmenkontrolle (1 Byte, FFrrZZZZ)
    - FF: Paketart (00: Protokollsteuerpaket, 01: Paket mit Anwenderdaten)
    - rr: reserviert für zukünftige Anwendungen
    - ZZZZ: Informationen zur Paketpufferung
  - Zieladresse (6 Byte): Adresse eines spezifischen Geräts oder Multicast-Adresse
  - Quelladresse (6 Byte)
  - Routing Informationen (0-30 Bytes): optional
  - Daten
  - Prüfsumme FCS (4 Byte): Berechung auf Basis der Daten zwischen Start- und Endsequenz
  - Endsequenz wie Token
  - Paketstatus (1 Byte ACrrACrr)
    - A: Paket wurde vom Empfänger als an in adressiert erkannt
    - C: Paket wurde vom Empfänger erfolgreich empfangen

Token Ring: Monitor

- Für den fehlerfreien Ablauf des Protokolls existiert im Token Ring ein Monitor.
- Aufgaben:
  - Entfernung von fehlerhaften Rahmen
  - Neugenerierung eines Tokens bei Verlust des Tokens (nach Ablauf einer Kontrollzeit)
  - Entfernung endlos kreisender Nachrichten bei Ausfall der Senderstation (Markierung der Nachricht beim Passieren des Monitors, Löschen der Nachricht beim 2. Passieren)
  - Signalisierung der Existenz des Monitors (durch Active Monitor Present Nachricht)

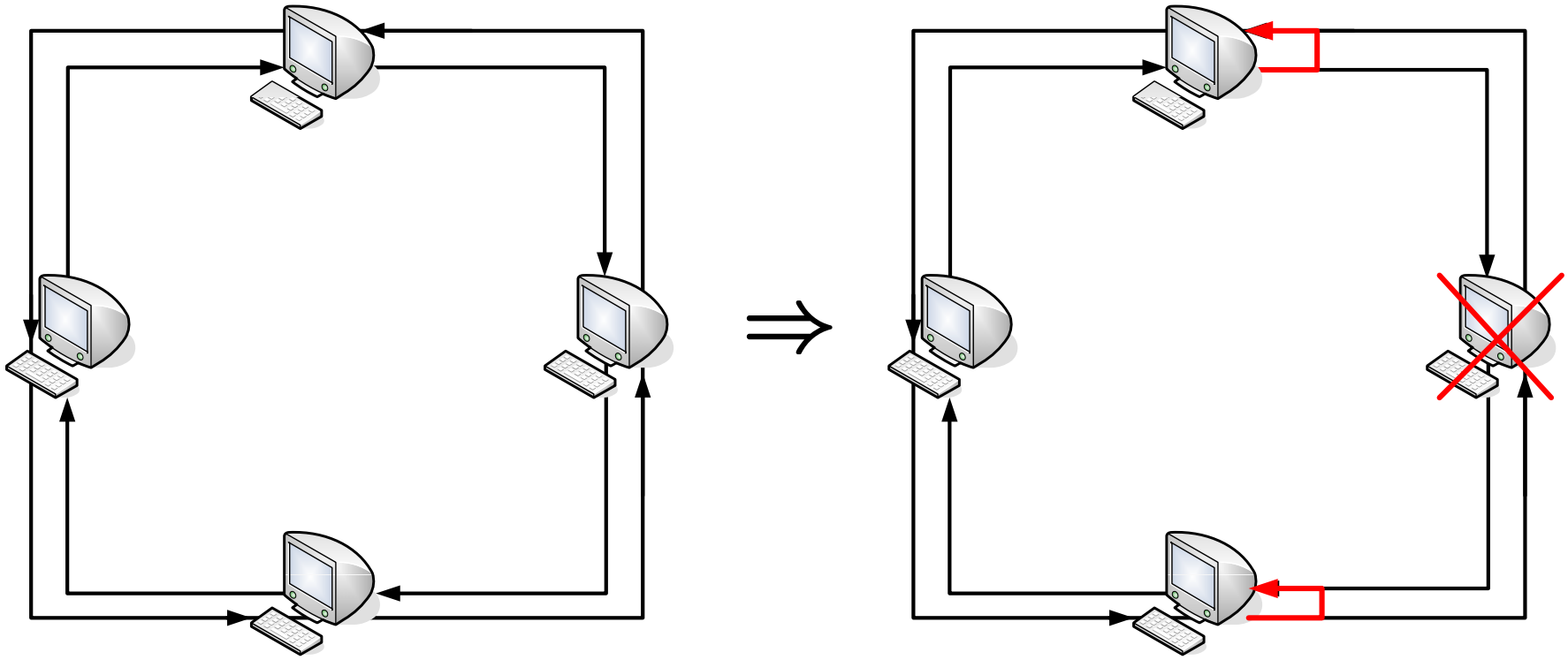
Token Ring: Initialisierung / Rekonfigurierung

- Bei der Initialisierung bzw. dem Ablauf des Standby Monitor Timer (Mechanismus zur Tolerierung des Ausfalls des Monitors)
  1. Senden eines Claim Token Paketes
  2. Überprüfung, ob weitere Pakete die Station passieren
  3. Falls nein $\Rightarrow$ Station wird zum Monitor
  4. Generierung eines Tokens
  5. Jede Station überprüft mittels des Duplicate Adress Test Paketes, ob die eigene Adresse bereits im Netzwerk vorhanden ist.
- Der Ausfall einer Station kann durch das Netzwerk erkannt werden und evtl. durch Überbrückung kompensiert werden.

FDDI

- Fiber Distributed Data Interface (FDDI) ist eine Weiterentwicklung von Token Ring
- Medium: Glasfaserkabel
- doppelter gegenläufiger Ring (aktiver Ring, Reservering) mit Token-Mechanismus
- Datenrate: 100 MBit/s, 1000 MBit/s
- Codierung: 4B5B (wie in FastEthernet)
- maximal 1000 Einheiten
- Ringlänge: max. 200 km
- Maximaler Abstand zwischen zwei Einheiten: 2 km
- Fehlertoleranz (maximal eine Station)
- Nachrichten können hintereinander gelegt werden (early release)
- Weitere Entwicklungen FDDI-2

Fehlerkonfiguration in FDDI



MAP / Token Bus

- **MAP**: **Manufacturing Automation Protocol** (Entwicklung ab 1982 von General Motors)
- Einsatz hauptsächlich im Produktionsbereich
- Schicht 1: anstelle von Ring-Topologie nun beliebige Topologie durch den Einsatz von Bridges, Gateways und Routern
- Medienzugriffsverfahren:
  - Token Bus, spezifiziert in IEEE 802.4
  - ähnlich Token-Ring, die benachbarte Station zur Weiterleitung des Tokens wird anhand einer Adresse bestimmt.
- In MAP werden zudem alle sieben Schichten des ISO/OSI-Modells spezifiziert.
- Aufgrund des Umfangs und der Komplexität konnte sich MAP nicht durchsetzen.
- Maximale Übertragungsrate: 10 MBit/s

Echtzeitfähige Kommunikation

Zeitgesteuerte Verfahren

Vertreter: TTP

Zugriffsverfahren: TDMA

- **TDMA** (**Time Division Multiple Access**) bezeichnet ein Verfahren, bei dem der Zugriff auf das Medium in Zeitscheiben (slots) eingeteilt wird.
- Die Zeitscheiben werden für jeweils einen Sender zur Verfügung gestellt.
- Vorteile:
  - Kollisionen sind per Design ausgeschlossen
  - Einzelnen Sendern kann eine Bandbreite garantiert werden.
  - Das zeitliche Verhalten ist vollkommen deterministisch.
  - Synchronisationsalgorithmen können direkt im Protokoll spezifiziert und durch Hardware implementiert werden.
- Nachteil:
  - keine dynamische Zuteilung bei reinem TDMA-Verfahren möglich
- Bekannte Vertreter: TTP, Flexray (kombiniert zeitgesteuert und dynamische Kommunikation)

Einführung TTP

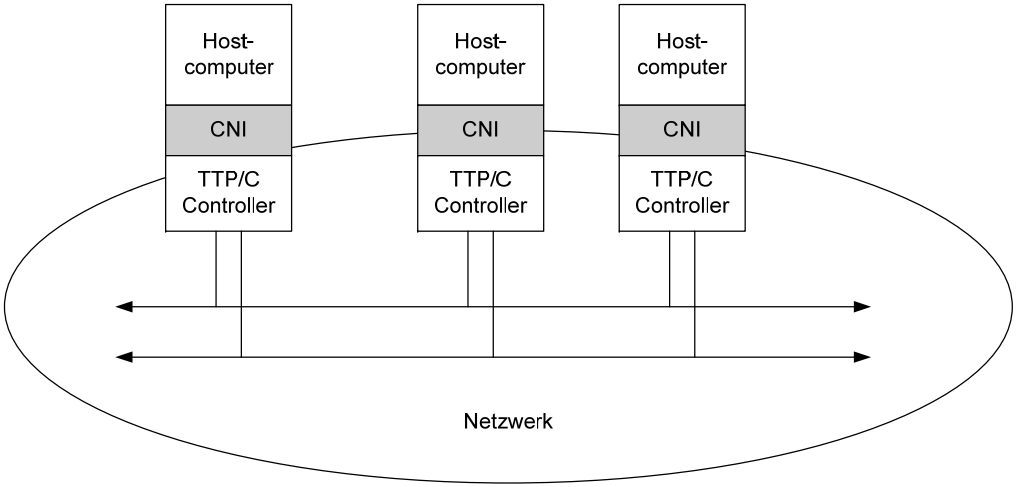
- Entstanden an der TU Wien (SpinOff TTTech)
- TTP steht für Time Triggered Protocol
- TTP ist geeignet für harte Echtzeitsysteme:
  - verteilter, fehlertoleranter Uhrensynchronisationsalgorithmus (Einheit: 1 µs), toleriert beliebige Einzelfehler.
  - Zwei redundante Kommunikationskanäle $\Rightarrow$ Fehlersicherheit
  - Einheiten werden durch Guards geschützt (Vermeidung eines babbling idiots).
  - Kommunikationsschema wird in Form einer **MEDL** (**Message Descriptor List**) a priori festgelegt und auf die Einheiten heruntergeladen.
- Einsatz unter anderem im Airbus A380

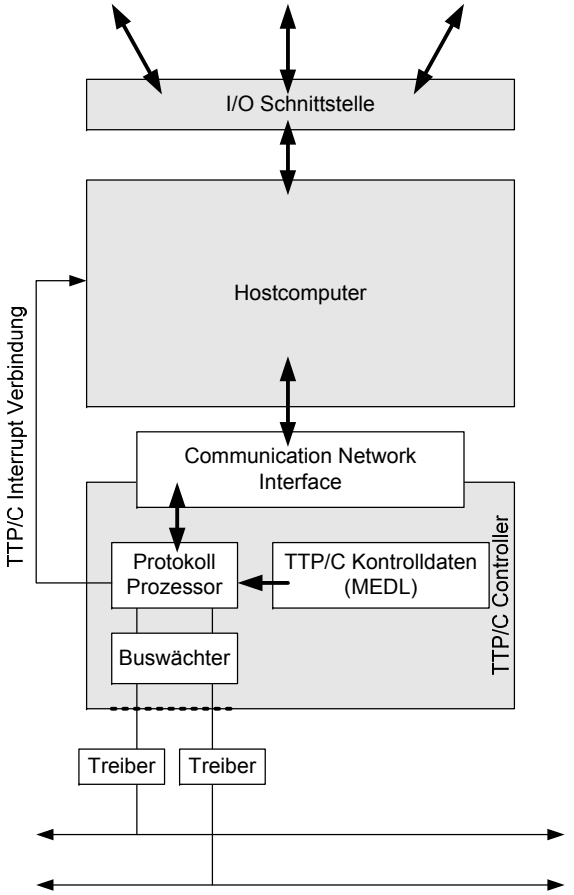
Echtzeitfähige Kommunikation

Zeitgesteuerte Verfahren

Vertreter: TTP

TTP-Architektur

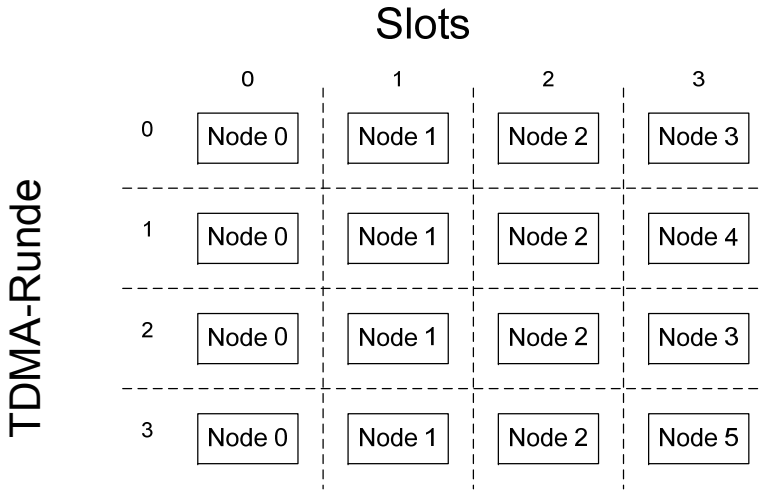




- Erläuterung:
  - Hostcomputer: Ausführung der eigentlichen Anwendung
  - CNI: Gemeinsamer Speicherbereich von Hostcomputer und TTP/C-Kontroller
  - Unterbrechungsverbindung: zur Übermittlung von Ticks der globalen Uhr und außergewöhnlicher Ereignisse an den Hostcomputer
  - MEDL: Speicherplatz für Kontrolldaten

TTP: Arbeitsprinzip

- Die Controller arbeiten autonom vom Hostcomputer (notwendige Daten sind in MEDL enthalten)
  - für jede zu empfangende und sendende Nachricht: Zeitpunkt und Speicherort in der CNI
  - zusätzliche Informationen zur Ausführung des Protokolls
- In jeder TDMA-Runde sendet ein Knoten genau einmal
  - Unterscheidung zwischen
    - reellen Knoten: Knoten mit eigenem Sendeschlitz
    - virtuelle Knoten: mehrere Knoten teilen sich einen Sendeschlitz
    - Die Länge der Sendeschlitze kann sich dabei unterscheiden, für einen Knoten ist die Länge immer gleich
      ⇒ TDMA-Runde dauert immer gleich lang



Protokolldienste

- Das Protokoll bietet:
  - Vorhersagbare und kleine, nach oben begrenzte Verzögerungen aller Nachrichten
  - Zeitliche Kapselung der Subsysteme
  - Schnelle Fehlerentdeckung beim Senden und Empfangen
  - Implizite Nachrichtenbestätigung durch Gruppenkommunikation
  - Unterstützung von Redundanz (Knoten, Kanäle) für fehlertolerante Systeme
  - Unterstützung von Clustermoduswechseln
  - Fehlertoleranter, verteilter Uhrensynchronisationsalgorithmus ohne zusätzliche Kosten
  - Hohe Effizienz wegen kleinem Protokollaufwand
  - Passive Knoten können mithören, aber keine Daten versenden.
  - Schattenknoten sind passive redundante Knoten, die im Fehlerfall eine fehlerhafte Komponente ersetzen können.

Fehlerhypothese

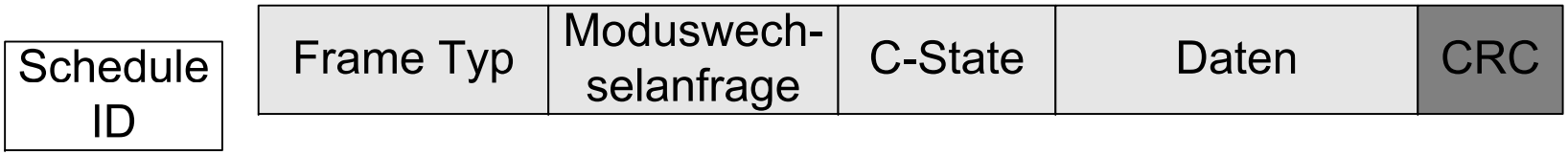
- Interne physikalische Fehler:
  - Erkennung einerseits durch das Protokoll, sowie Verhinderung eine babbling idiots durch Guards.
- Externe physikalische Fehler:
  - Durch redundante Kanäle können diese Fehler toleriert werden.
- Designfehler des TTP/C Kontrollers:
  - Es wird von einem fehlerfreien Design ausgegangen.
- Designfehler Hostcomputer:
  - Protokollablauf kann nicht beeinflusst werden, allerdings können inkorrekte Daten erzeugt werden.
- Permanente Slightly-Off-Specification-Fehler:
  - können durch erweiterte Guards toleriert werden.
- Regionale Fehler (Zerstören der Netzwerkverbindungen eines Knotens):
  - Folgen können durch Ring- und Sternarchitektur minimiert werden.

Zustandsüberwachung

- Das Protokoll bietet Möglichkeiten, dass Netzwerk zu analysieren und fehlerbehaftete Knoten zu erkennen.
- Der Zustand des Netzwerkes wird dabei im Kontrollerzustand (C-State) gespeichert.
- Der C-State enhält:
  - die globale Zeit der nächsten Übertragung
  - das aktuelle Fenster im Clusterzyklus
  - den aktuellen, aktiven Clustermodus
  - einen eventuell ausstehenden Moduswechsel
  - den Status aller Knoten im Cluster
- Das Protokoll bietet einen Votierungsalgorithmus zur Überprüfung des eigenen Zustands an.
- Ein Knoten ist korrekt, wenn er in seinem Fenster eine korrekte Nachricht versendet hat.
- Knoten können sich durch die Übernahme der Zeit und der Schedulingposition integrieren, sobald ein integrierender Rechner eine korrekte Nachricht sendet, erkennen in die anderen Knoten an.

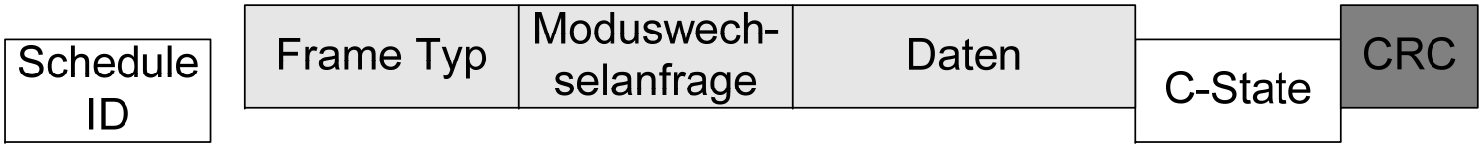
Datenpakete in TTP

- Paket mit explizitem C-State



- Kaltstartpaket

- Paket mit implizitem C-State



TTP: Clusterstart

- Der Start erfolgt in drei Schritten:
  1. Initialisierung des Hostcomputers und des Controllers
  2. Suche nach Frame mit expliziten C-State und Integration
  3. a) Falls kein Frame empfangen wird, werden die Bedingungen für einen Kaltstart geprüft:
     - Host hat sein Lebenszeichen aktualisiert
     - Das Kaltstart Flag in der MEDL ist gesetzt
     - die maximale Anzahl der erlaubten Kaltstarts wurde noch nicht erreicht

     Sind die Bedingungen erfüllt, sendet der Knoten ein Kaltstartframe.
  3. b) Falls Frame empfangen wird: Versuch zur Integration

TTP: Sicherheitsdienste / Synchronisation

- Sicherheitdienste:
  - Korrektheit: Alle Knoten werden über die Korrektheit der anderen Knoten mit einer Verzögerung von etwa einer Runde informiert.
  - Cliquenentdeckung: Es werden die Anzahl der übereinstimmenden und entgegen gesetzten Knoten gezählt. Falls mehr entgegen gesetzte Knoten gezählt werden, so wird ein Cliquenfehler angenommen.
  - Host/Kontroller Lebenszeichen: der Hostcomputer muss seine Lebendigkeit dem Kontroller regelmäßig zeigen. Sonst wechselt der Kontroller in den passiven Zustand.
- Synchronisation:
  - In regelmäßigen Abständen wird die Uhrensynchronisation durchgeführt.
  - Es werden die Unterschiede der lokalen Uhr zu ausgewählten (stabilen) Uhren (mind.4) anderer Rechner anhand den Sendezeiten gemessen.
  - Die beiden extremen Werte werden gestrichen und vom Rest der Mittelwert gebildet.
  - Die Rechner einigen sich auf einen Zeitpunkt für die Uhrenkorrektur.

Echtzeitfähige Kommunikation

Zusammenfassung

Zusammenfassung

- Die Eignung eines Kommunikationsmediums für die Anwendung in Echtzeitsystemen ist vor allem durch das Medienzugriffsverfahren bestimmt.
- Die maximale Wartezeit ist bei
  - CSMA/CD: unbegrenzt und nicht deterministisch ($\Rightarrow$ keine Eignung für Echtzeitsysteme)
  - CSMA/CA,tokenbasierten Verfahren: begrenzt, aber nicht deterministisch (abhängig von anderen Nachrichten)
  - zeitgesteuerten Verfahren: begrenzt und deterministisch.
- Die Priorisierung der Nachrichten wird von CSMA/CA und tokenbasierten Verfahren unterstützt.
- Nachteil der zeitgesteuerten Verfahren ist die mangelnde Flexibilität (keine dynamischen Nachrichten möglich).
- Trotz diverser Nachteile geht der Trend hin zum Ethernet.

Trends: Real-Time Ethernet

- Es existieren verschiedene Ansätze
  - Beispiel: Ethercat von Beckhoff
    - Die Nachrichten entsprechen dem Standardnachrichtenformat von Ethernet
    - Pakete werden von einem Master initiiert und werden von den Teilnehmern jeweils weitergeleitet.
    - Jeder Knoten entnimmt die für ihn bestimmten Daten und kann eigene Daten anfügen.
    - Die Bearbeitung erfolgt on-the-fly, dadurch kann die Verzögerung minimiert werden.
  - Beispiel: Profinet von Siemens
    - Drei verschiedene Protokollstufen (TCP/IP – Reaktionszeit im Bereich von 100ms, Real-time Protocol - bis 10ms, Isochronous Real-Time - unter 1ms)
    - Profinet IRT benutzt vorher bekannte, reservierte Zeitschlitze zur Übertragung von echtzeitkritischen Daten, in der übrigen Zeit wird das Standard-Ethernet Protokoll ausgeführt

Klausurfragen

- Klausur Wintersemester 07/08 (4 Punkte = 4min)
  - Erläutern Sie kurz die wesentlichen Unterschiede zwischen TokenRing, TokenBus und Ethercat in Bezug auf Topologie und Mediumszugriffverfahren.
- Wiederholungsfragen:
  1. Was ist der Unterschied zwischen dominanten und rezessiven Bits.
  2. Nennen Sie zwei Mechanismen zur Bitsynchronisierung und erklären Sie diese.
  3. Was ist der Unterschied zwischen CSMA/CD und CSMA/CA?
  4. Erläutern Sie zwei verschiedene Ansätze um Ethernet echtzeitfähig zu machen.
  5. Beurteilen Sie die Kommunikationsprotokolle Ethernet, CAN, TTP nach Ihrer Echtzeitfähigkeit und gehen Sie vor allem auf die Möglichkeit zur Vorhersage der maximalen Nachrichtenlatenz ein.

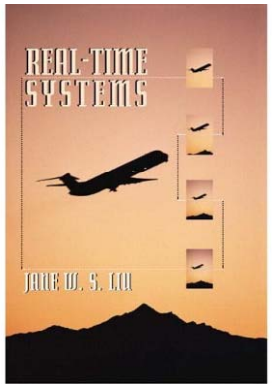
Kapitel 5

Echtzeitbetriebssysteme

Inhalt

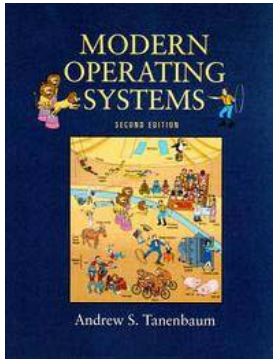
- Grundlagen
- Betrachtung diverser Betriebssysteme:
  - Domänenspezifische Betriebssysteme:
    - OSEK
    - TinyOS
  - Klassische Echtzeitbetriebssysteme
    - QNX
    - VxWorks
    - PikeOS
  - Linux- / Windows-Echtzeitvarianten
    - RTLinux/RTAI
    - Linux Kernel 2.6
    - Windows CE

Literatur



Jane W. S. Liu, Real-Time Systems, 2000

Dieter Zöbel, Wolfgang Albrecht: Echtzeitsysteme: Grundlagen und Techniken, 1995



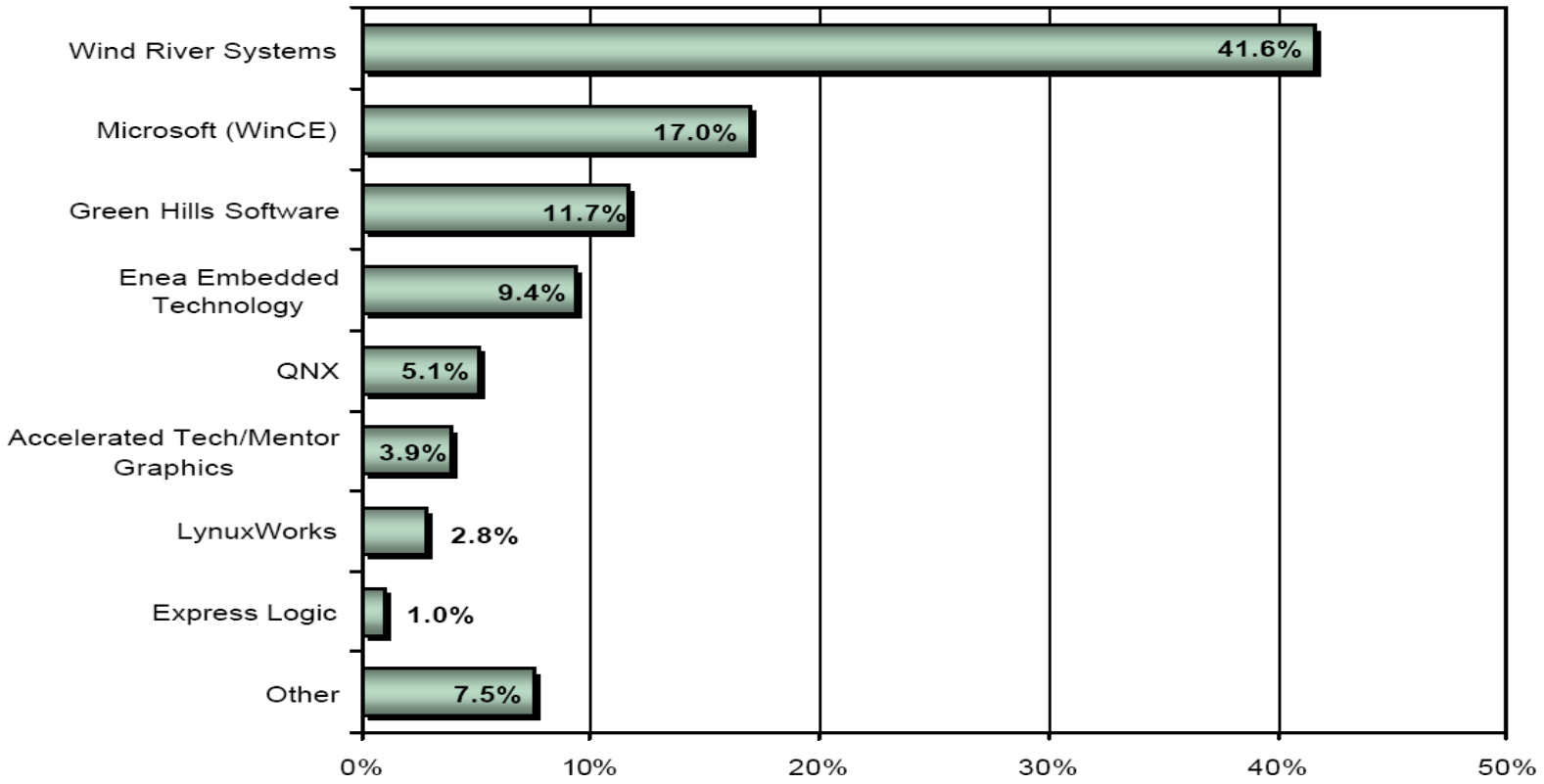
Andrew S. Tanenbaum: Modern Operating Systems, 2001

Arnd Heursch et al.: Time-critical tasks in Linux 2.6, 2004
http://inf3-www.informatik.unibw-muenchen.de/research/linux/hannover/automation_conf04.pdf

Interessante Links

- http://www.mnis.fr/en/support/doc/rtos/
- http://aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf
- http://www.osek-vdx.org/
- http://www.qnx.com/
- http://www.windriver.de
- http://www.fsmlabs.com
- http://www.rtai.org
- http://www.tinyos.net/

Marktaufteilung (Stand 2004)



Marktanteil am Umsatz, Gesamtvolumen 493 Mio. Dollar, Quelle: The Embedded Software Strategic Market Intelligence Program 2005

Anforderungen an Echtzeitbetriebssysteme

- Echtzeitbetriebssysteme unterliegen anderen Anforderungen als Standardbetriebssysteme:
  - stabiler Betrieb rund um die Uhr
  - definierte Reaktionszeiten
  - parallele Prozesse
  - Unterstützung von Mehrprozessorsystemen
  - schneller Prozesswechsel (geringer Prozesskontext)
  - echtzeitfähige Unterbrechensbehandlung
  - echtzeitfähiges Scheduling
  - echtzeitfähige Prozesskommunikation
  - umfangreiche Zeitdienste (absolute, relative Uhren, Weckdienste)
  - einfaches Speichermanagement

Fortsetzung

- Unterstützung bei der Ein- und Ausgabe
  - vielfältigste Peripherie
  - direkter Zugriff auf Hardware-Adressen und -Register durch den Benutzer
  - Treiber in Benutzerprozessen möglichst schnell und einfach zu implementieren
  - dynamisches Binden an den Systemkern
  - direkte Nutzung DMA
  - keine mehrfachen Puffer: direkt vom Benutzerpuffer auf das Gerät
- Einfachste Dateistrukturen
- Protokoll für Feldbus oder LAN-Bus, möglichst hardwareunterstützt
- Aufteilung der Betriebssystemfunktionalität in optionale Komponenten (Skalierbarkeit)

Echtzeitbetriebssysteme

Kriterien zur Beurteilung

Beurteilung von Echtzeitbetriebssystemen

- Folgende Aspekte werden wir genauer betrachten:
  - Schedulingverfahren
  - Prozessmanagement
  - Speicherbedarf (Footprint)
  - Garantierte Reaktionszeiten

Schedulingverfahren

- Fragestellung:
  - Welche Konzepte sind für das Scheduling von Prozessen verfügbar?
  - Gibt es Verfahren für periodische Prozesse?
  - Wie wird dem Problem der Prioritätsinversion begegnet?
  - Wann kann eine Ausführung unterbrochen werden?

Arten von Betriebssystemen

- Betriebsysteme werden in drei Klassen unterteilt:
  - Betriebssysteme mit **kooperativen Scheduling**: es können verschiedene Prozesse parallel ausgeführt werden. Der Dispatcher kann aber einem Prozess den Prozessor nicht entziehen, vielmehr ist das Betriebssystem auf die Kooperation der Prozesse angewiesen (z.B. Windows 95/98/ME)
  - Betriebssysteme mit **präemptiven Scheduling**: einem laufenden Prozess kann der Prozessor entzogen werden, falls sich der Prozess im Userspace befindet. (z.B. Linux, Windows 2000/XP)
  - **Präemptible Betriebssysteme**: der Prozessor kann dem laufenden Prozess jederzeit entzogen werden, auch wenn sich dieser im Kernelkontext ausgeführt wird.

$\Rightarrow$ Echtzeitsysteme müssen präemptibel sein.