Beispiel FTOS: Hintergrund

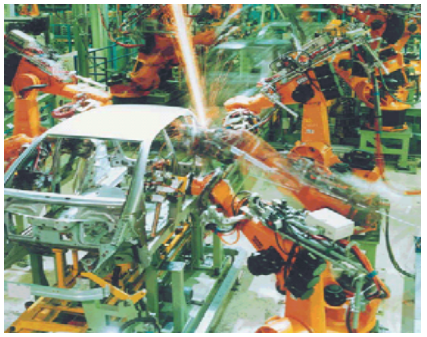
- Programmierumgebung für fehlertolerante, verteilte Echtzeitsysteme
- Komplette Werkzeugkette von der Modellierung bis zur Codegenerierung für verschiedenste Plattformen
- Fokus liegt auf der Generierung von nicht-funktionalen Aspekten
- Ausgelegt auf Anwendungen, die traditionell nicht oder nur minimal fehlertolerant entwickelt wurden.
- Es wird kein eingeschränktes Fehlermodell zugrunde gelegt.

The operating system must provide basic support for guaranteeing real-time constraints, supporting fault tolerance and distribution, and integrating time-constrained resource allocations and scheduling across a spectrum of resource types, including sensor processing, communications, CPU, memory, and other forms of I/O.

John A. Stankovic, Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems, 1988

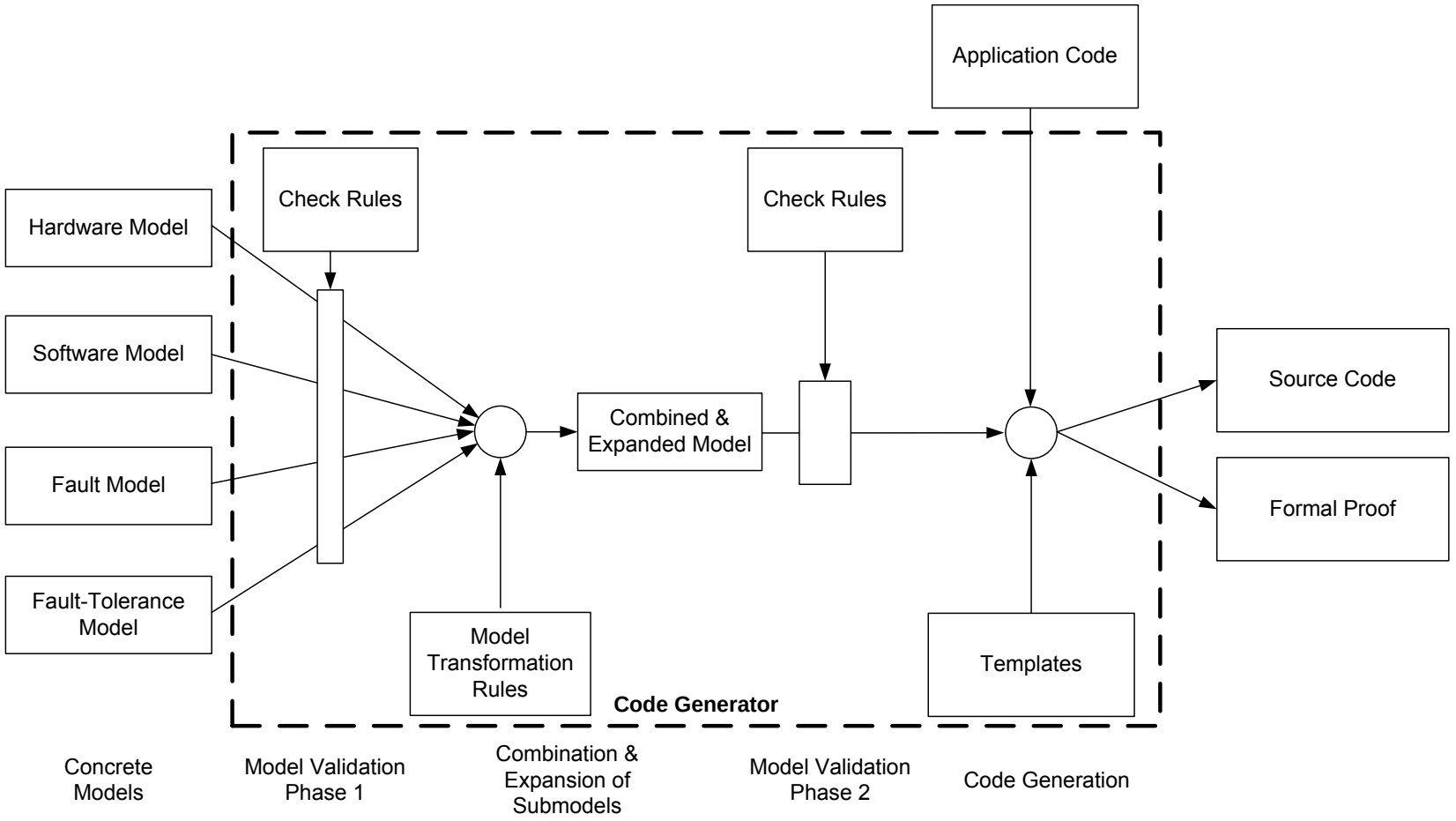
Stromerzeugung

Medical

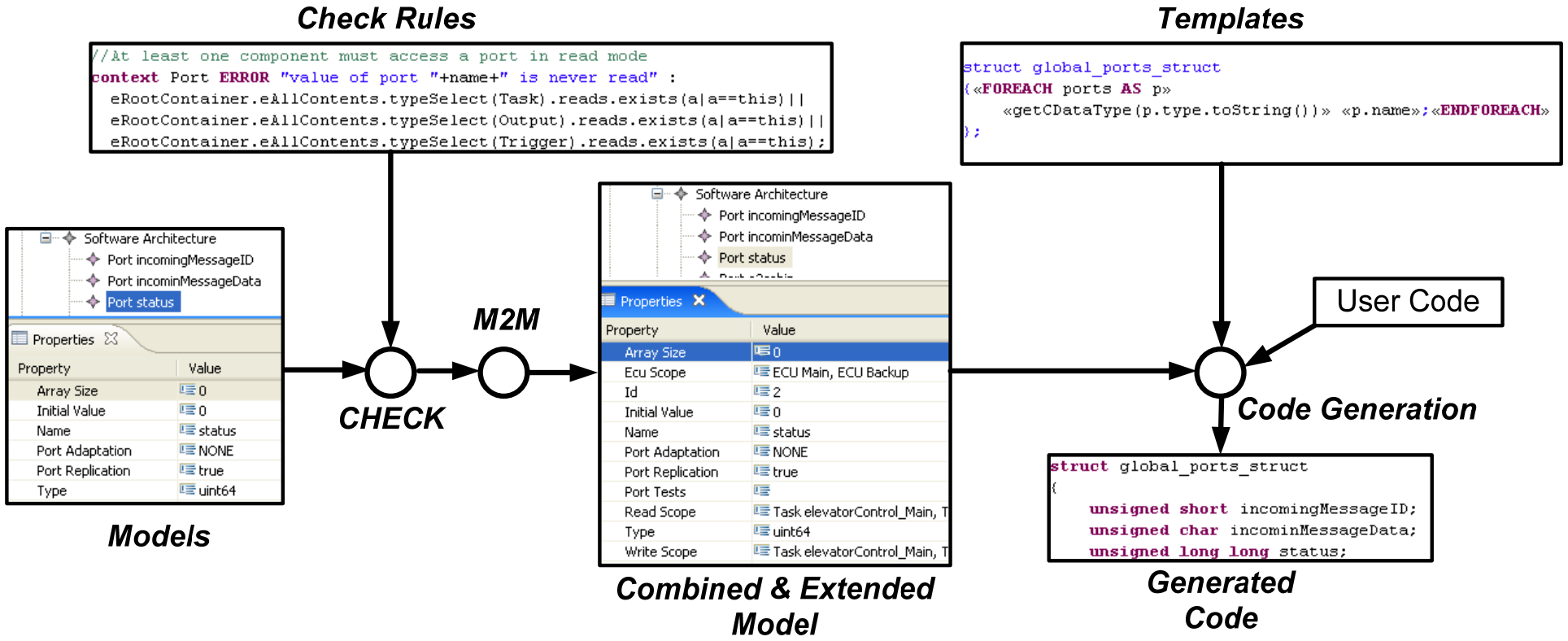


Automatisierung

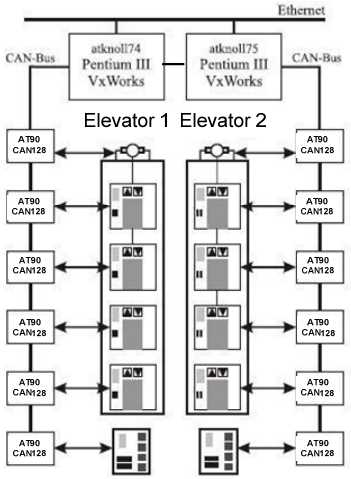
FTOS: Codegenerator-Architektur



Ein konkretes Beispiel



Demonstratoren



Schwebender Stab (auf 2-aus-3-Rechnersystem)

→ Regelungsrate 2,5 ms

→ nur 24 Zeilen zur Programmierung des PID-Reglers notwendig

Aufzugssteuerung

→ Codegenerierung für 8-Bit Atmel-Controller bzw. Standarddesktop-rechner

→ In Kombination mit EasyLab 100% modellbasierte Entwicklung möglich

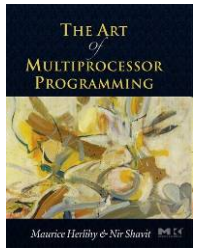
Kapitel 3

Nebenläufigkeit

Inhalt

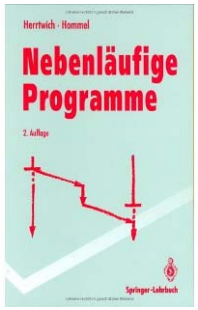
- Motivation
- Unterbrechungen (Interrupts)
- (Software-) Prozesse
- Threads
- Interprozesskommunikation (IPC)

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

- Links:
  - http://www.beyondlogic.org/interrupts/interupt.htm
  - http://www.llnl.gov/computing/tutorials/pthreads/

Definition von Nebenläufigkeit

- **Allgemeine Bedeutung**: Nebenläufigkeit bezeichnet das Verhältnis von Ereignissen, die nicht kausal abhängig sind, die sich also nicht beeinflussen. Ereignisse sind nebenläufig, wenn keines eine Ursache des anderen ist. Oder anders ausgedrückt: Aktionen können nebenläufig ausgeführt werden (sie sind parallelisierbar), wenn keine das Resultat der anderen benötigt.
- **Bedeutung in der Programmierung**: Nebenläufigkeit bezeichnet hier die Eigenschaft von Programmcode nicht linear hintereinander ausgeführt zu werden, sondern parallel ausführbar zu sein.

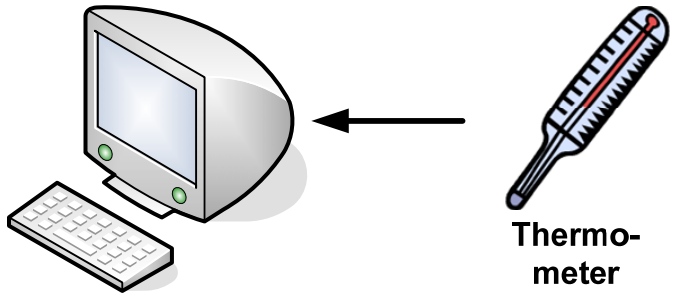
  Die Nebenläufigkeit von mehreren unabhängigen Prozessen bezeichnet man als *Multitasking*;

  Nebenläufigkeit innerhalb eines Prozesses als *Multithreading*.

Motivation

- Gründe für Nebenläufigkeit 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

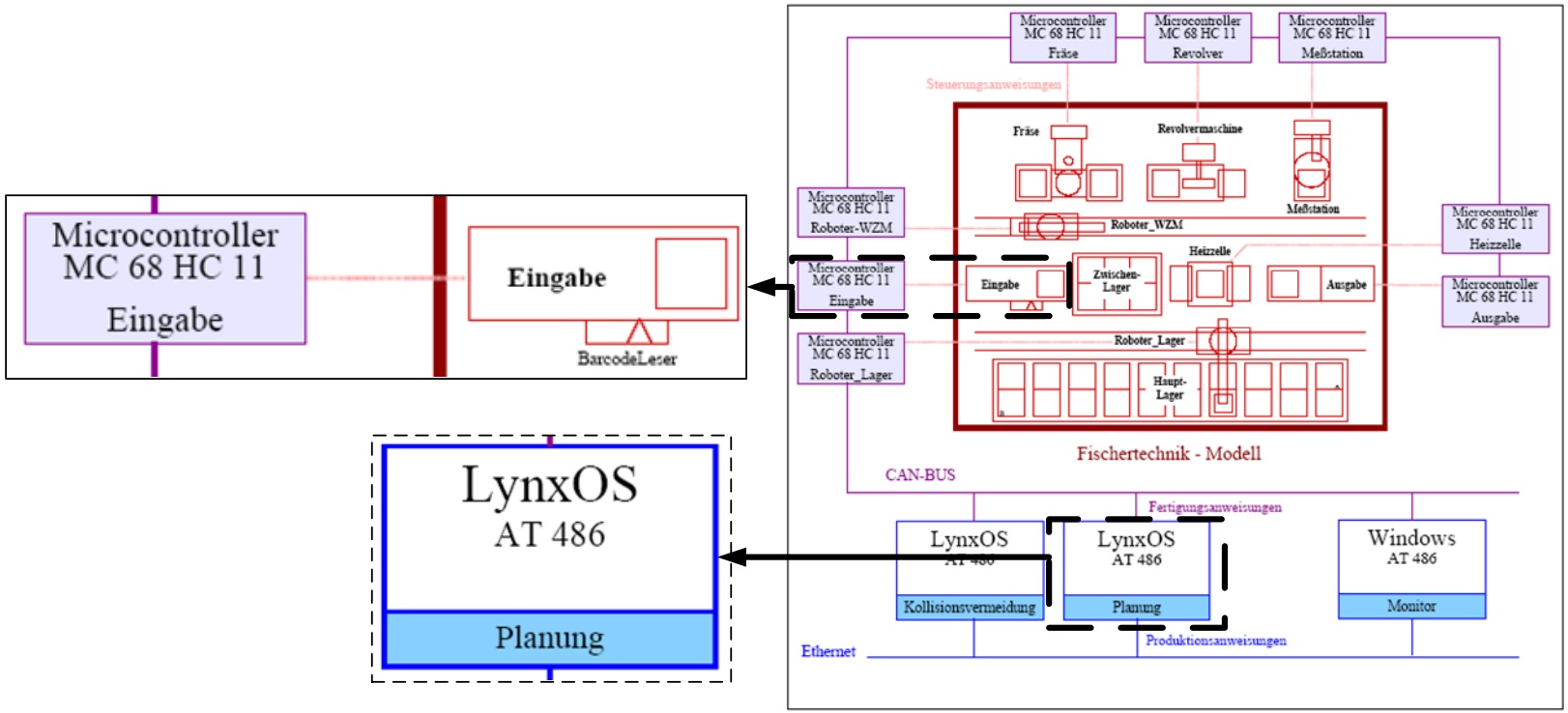
Anwendungsfälle für Nebenläufigkeit (Unterbrechungen)



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

Allgemeines Anwendungsgebiet: hauptsächlich zur Anbindung von externer Hardware

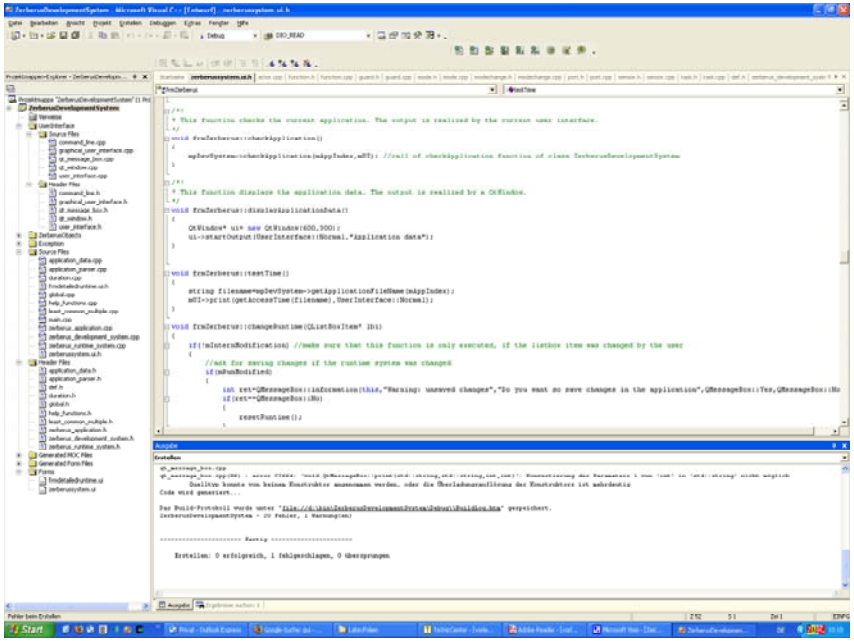
Anwendungsfälle für Nebenläufigkeit (Prozesse)



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

Unterbrechungen

Anbindung an die Umwelt

- Es muss ein Mechanismus gefunden werden, der es erlaubt Änderungen der Umgebung (z.B. Tastatur) zu registrieren.
- 1. Ansatz: **Polling**
  Es werden die IO-Register reihum nach Änderungen abgefragt und bei Änderungen spezielle Antwortprogramme ausgeführt.
  - Vorteile:
    - bei wenigen IO-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: Interruptkonzept

- **Interrupt**: Ein Interrupt ist ein durch ein Ereignis ausgelöster, automatisch ablaufender Mechanismus, der die Verarbeitung des laufenden Programms unterbricht und die Wichtigkeit des Ereignisses überprüft. Darauf basierend erfolgt die Entscheidung, ob das bisherige Programm weiter bearbeitet wird oder eine andere Aktivität gestartet wird.
- Vorteile:
  - sehr geringe Extrabelastung der CPU
  - Prozessor wird nur dann beansprucht, wenn es nötig ist
- Nachteile:
  - Nicht-Determinismus: Unterbrechungen können zu einem beliebigen Zeitpunkt eintreffen.

Technische Realisierung

- Zur Realisierung besitzen Rechner einen oder mehrere spezielle Interrupt-Eingänge. Wird ein Interrupt aktiviert, so führt dies zur Ausführung der entsprechenden Unterbrechungsbehandlungs-routine (**interrupt handler, interrupt service routine (ISR)**).
- Das Auslösen der Unterbrechungsroutine ähnelt einem Unterprogrammaufruf. Der Programmablauf wird an einer anderen Stelle fortgesetzt und nach Beendigung der Routine an der unterbrochenen Stelle fortgefahren. Allerdings tritt die Unterbrechungsroutine im Gegensatz zum Unterprogrammsaufruf asynchron, also an beliebigen Zeitpunkten, auf.

Sperren von Interrupts

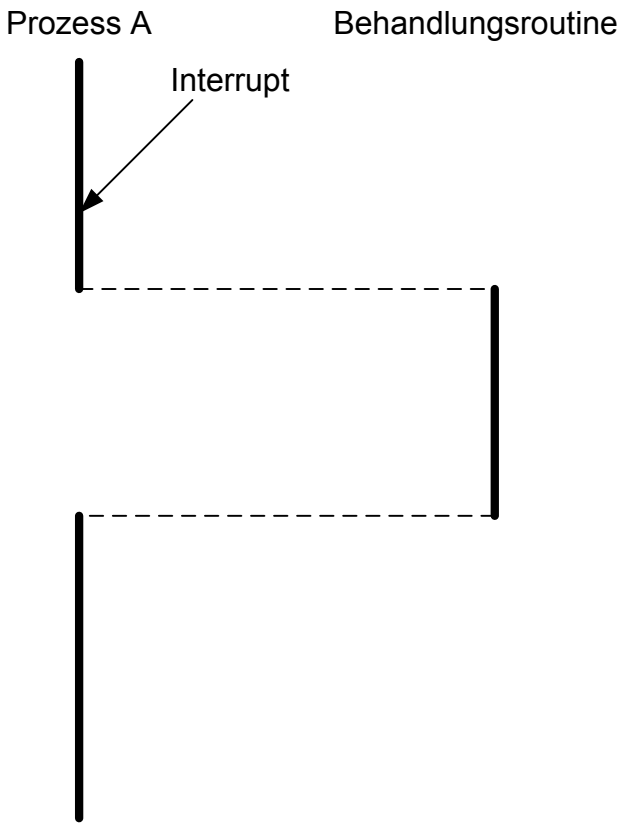
- Durch die Eigenschaft der Asynchronität kann eine deterministische Ausführung nicht gewährleistet werden. Aus diesem Grund kann eine kurzfristige Sperrung von Interrupts nötig sein, um eine konsistente Ausführung der Programme zu erlauben.
- Durch das Sperren werden Interrupts in der Regel nur verzögert, nicht jedoch gelöscht.

Interrupt Prioritäten

- Unterbrechungen besitzen unterschiedliche Prioritäten. Beim Auftreten einer Unterbrechung werden Unterbrechungen gleicher oder niedrigerer Priorität gesperrt.
- Tritt dagegen während der Ausführung der Behandlungsroutine eine erneute Unterbrechung mit höherer Priorität auf, so wird die Unterbrechungsbehandlung gestoppt und die Behandlungsroutine für die Unterbrechung mit höherer Priorität durchgeführt.

Ablauf einer Unterbrechung

1. Sperren von Unterbrechungen mit gleicher oder niedrigerer Priorität
2. Retten des Prozessorstatus
3. Bestimmen der Interruptquelle
4. Laden des Interruptvektors (Herstellung des Anfangszustandes für Behandlungsroutine)
5. Abarbeiten der Routine
6. Rückkehr zur Programmausführung (nicht unbedingt der unterbrochene Prozess)



Hardware Interrupts

- Nachfolgend wird eine typische Belegung (Quelle von 2002) der Interrupts angegeben:

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 (Soundblaster-Emulation) 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 DIE-Kanal

Programmieren von Interrupts

- Implementierung der Unterbrechungsbehandlungsroutine

```
void interrupt yourisr() /* Interrupt Service
   Routine (ISR) */
{
   disable();
   /* Body of ISR goes here */
   oldhandler();
   outportb(0x20,0x20); /* Send EOI to PIC1 */
   enable();
}
```

Erläuterung

- `void interrupt yourisr`: Deklaration einer Interrupt Service Routine
- `disable()`: Ist eine weitere Unterbrechung von höher priorisierten Interrupts nicht gewünscht, so können auch diese gesperrt werden (Vorsicht bei der Verwendung).
- `oldhandler()`: Oftmals benutzen mehrere Programme einen Interrupt (z.B. die Uhr), in diesem Fall sollte man die bisherige ISR sichern (siehe nächste Folie) und an den neuen ISR anhängen
- `outportb()`: Dem PIC (Programmable Interrupt Controler) muss signalisiert werden, dass die Behandlung des Interrupts beendet ist.
- `enable`: Die Interrupt-Sperre muss aufgehoben werden.

Einfügen der Routine in Interrupt Vector Table

```
#include <dos.h>
#define INTNO 0x0B /* Interupt Number 3*/
void main(void)
{
   oldhandler = getvect(INTNO); /* Save Old Interrupt Vector */
   setvect(INTNO, yourisr); /* Set New Interrupt Vector Entry */
   outportb(0x21,(inportb(0x21) & 0xF7)); /*Un-Mask(Enable)IRQ3 */
   /* Set Card - Port to Generate Interrupts */
   /* Body of Program Goes Here */
   /* Reset Card - Port as to Stop Generating Interrupts */
   outportb(0x21,(inportb(0x21) | 0x08)); /*Mask (Disable) IRQ3 */
   setvect(INTNO, oldhandler); /*Restore old Vector Before Exit*/
}
```

Erläuterung

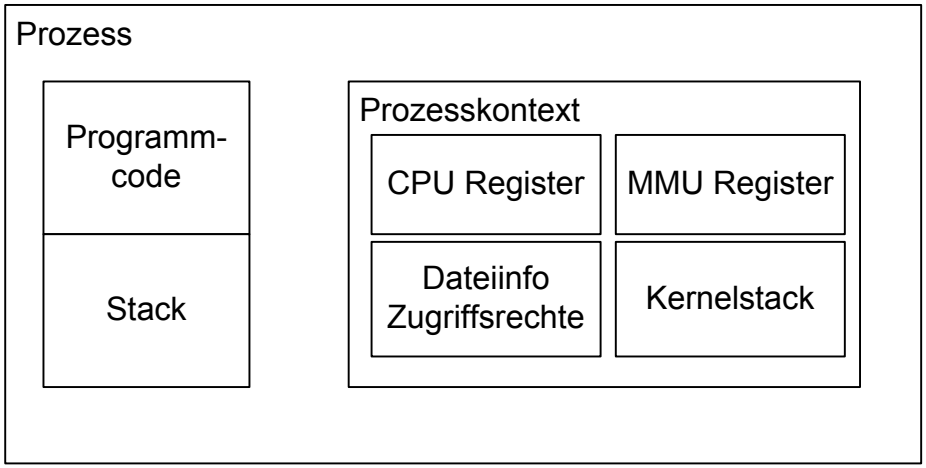
- Die Unterbrechungsvektortabelle enthält einen Verweis auf die entsprechende Unterbrechungsbehandlung für die einzelnen Unterbrechungen
- `INTNO`: Es soll der Hardwareinterrupt IRQ 3 (serielle Schnittstelle) verwendet werden, dieser Interrupt entspricht der Nummer 11 (insgesamt 255 Interrupts (vor allem Softwareinterrupts) vorhanden).
- `oldhandler=getvect(INTNO)`: Durch die Funktion `getvect()` kann die Adresse der Behandlungsfunktion zurückgelesen werden. Diese wird in der vorher angelegte
- `setvect`: setzen der neuen Routine
- `outportb`: setzen einer neuen Maskierung

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