



# Embedded Software Engineering 1

HS 2024 – Prof. Reto Bonderer

Autoren: Laurin Heitzer, Simone Stitz

<https://github.com/P4ntomime/EmbSW1>

V1.0.20241205

## Inhaltsverzeichnis

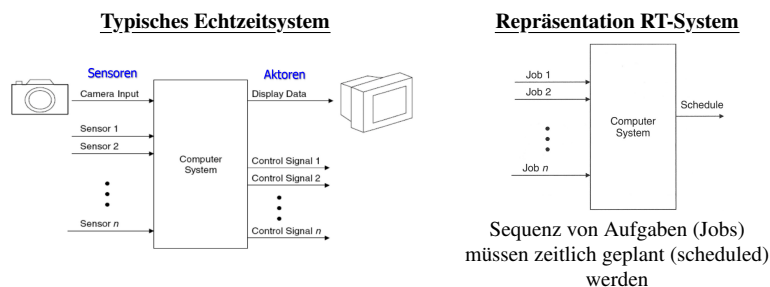
<b>1 Real-Time System (Echtzeitsystem)</b>	<b>2</b>		
1.1 Definitionen	2		
1.2 Fehlverhalten eines Systems (failed system)	2		
1.3 Echtzeitdefinition – Verschiedene Echtzeitsysteme	2		
1.4 Determinismus (determinacy)	2		
1.5 Auslastung (utilization)	2		
1.6 Real-time Scheduling	2		
<b>2 Modellierung eines Embedded Systems</b>	<b>2</b>		
2.1 V-Modell für Software-Entwicklungszyklus	2		
2.2 Model Driven Development (MDD)	2		
2.3 Vorgehen bei der Modellierung	2		
2.4 Systemgrenze definieren & Systemprozesse finden	2		
2.5 Verteilungen festlegen	3		
2.6 Systemprozesse detaillieren	3		
<b>3 Hardware-Software-Codesign</b>	<b>3</b>		
3.1 Ziele	3		
3.2 Anforderungen für praktische Anwendungen	4		
3.3 Spezifikationssprachen	4		
3.4 Virtuelle Prototypen	4		
3.5 X-in-the-loop	4		
3.6 Entwicklungsplattformen	4		
<b>4 Zustandsbasierte Systeme</b>	<b>4</b>		
4.1 Asynchrone vs. synchrone FSM	4		
4.2 Finite State Machines (FSM)	4		
4.3 State-Event-Diagramm (Zustandsdiagramm)	4		
4.4 Zustandstabelle	4		
<b>5 Statecharts (nach Marwedel)</b>	<b>5</b>		
5.1 Nachteile von State-Event-Diagrammen	5		
5.2 Definitionen	5		
5.3 Hierarchie (OR-super-states)	5		
5.4 Default-State	5		
5.5 History	5		
5.6 Kombination: History- und Default-Mechanismus	5		
5.7 Parallelität (AND-super-state, Teilautomaten)	5		
5.8 Timers	6		
5.9 Beispiel – Armbanduhr als Statechart	6		
<b>6 Realisierung flache FSM</b>	<b>6</b>		
6.1 Mögliche Realisierungen von flachen FSMs	6		
6.2 Realisierung mit Steuerkonstrukt (prozedural in C)	6		
6.3 Realisierung mit Steuerkonstrukt (objektorientiert in C++)	7		
6.4 Realisierung mit Tabelle	8		
6.5 Erweiterung der Realisierung mittels Tabellen	9		
6.6 Realisierung mit StatePattern	9		
<b>7 Modularisierung</b>	<b>11</b>		
7.1 Grundprinzip Modularisierung	11		
7.2 Bewertung einer Zerlegung	11		
7.3 Kopplung	11		
7.4 Kohäsion	11		
7.5 Guidelines – gute Modularisierung	11		
7.6 Package-Diagramm	11		
<b>8 Patterns (Lösungsmuster)</b>	<b>12</b>		
8.1 Arten von Patterns	12		
8.2 Wichtige Patterns für Embedded Systems	12		
<b>9 Event-based Systems</b>	<b>12</b>		
9.1 Ereignisse (Events)	12		
9.2 Synchrone Umsetzung von Ereignissen	12		
9.3 Asynchrone Umsetzung von Ereignissen	12		
9.4 Interrupt-Verarbeitung	12		
9.5 Interruptvektortabelle (IVT)	12		
9.6 Model View Controller (MVC) aka Observer Pattern	12		
9.7 Callback-Funktionen	12		
9.8 Umsetzung der Callback-Funktionen in C (clientseitig)	12		
		9.9 Umsetzung der Callback-Funktionen in C (serverseitig)	13
		9.10 Observer Pattern	14
<b>10 Scheduling</b>	<b>15</b>		
10.1 Multitasking	15		
10.2 Scheduability	15		
10.3 Scheduling-Strategien	15		
10.4 Cooperative Multitasking	15		
10.5 Preemptive Multitasking / Scheduling	15		
10.6 Rate Monotonic Scheduling (RMS)	15		
10.7 Rate Monotonic Scheduling Theorem	15		
10.8 Vorgehen – Rate Monotonic Scheduling	15		
10.9 RMA Bound (RMA = Rate Monotonic Approach)	16		
10.10 Anleitung für Zuweisung der Prioritäten bei RMS	16		
<b>11 Concurrency (Gleichzeitigkeit)</b>	<b>16</b>		
11.1 Parallel Computing vs. Concurrent Computing	16		
11.2 Warum man Concurrency nicht verwenden sollte	16		
11.3 Synchronisation	16		
<b>12 POSIX Threads Programming</b>	<b>16</b>		
12.1 UNIX Process vs. UNIX Thread	16		
12.2 pthreads API	16		
12.3 Beispiel: thread API	17		
12.4 Thread-safeness	17		
12.5 Quasi-Parallelität / 'Prozess'-Zustände	17		
12.6 Synchronisation	17		
12.7 Mutex (mutual exclusion)	17		
12.8 Thread Synchronisierung in C mit pthreads API	17		
12.9 Monitorprinzip (Monitor Pattern)	18		
12.10 'Stolperfallen' bei Synchronisation	18		
12.11 Informationen zwischen Threads austauschen	18		
12.12 Condition Variables mit pthreads	18		
12.13 Bounded Buffer Problem / Producer-consumer problem	19		
12.14 POSIX Interprocess Communication (IPC)	19		
<b>13 Resource Acquisition Is Initialisation (RAII)</b>	<b>19</b>		
13.1 Grundkonzept von RAII	19		
13.2 RAII bei Heapobjekten	19		
13.3 RAII bei Mutex	19		
<b>14 Interfacig in C</b>	<b>19</b>		
14.1 Plain Old Data Types (POD Types)	19		
14.2 Language Linkage	19		
14.3 Festlegen der Language Linkage	19		
14.4 Festlegen der Language Linkage – Umsetungen C++ seitig	19		
14.5 C++ Code aus C aufrufen	19		
<b>15 Programming Style Guide</b>	<b>20</b>		
15.1 Grundsätzliche Konventionen	20		
15.2 Namenskonventionen	20		
15.3 Typen mit genauer Breite	20		
<b>16 Multicore Systeme</b>	<b>20</b>		
16.1 Geschwindigkeit auf Prozessor steigern	20		
16.2 Speicherorganisation auf Multicore Prozessor	20		
16.3 Amdahl's Law	20		
16.4 Memory Hierarchy	20		
16.5 Cache-Speicher	20		
16.6 Cache Ersetzungsstrategien	21		
16.7 Caches in Echtzeitsystemen	21		
16.8 Cache Coherence	21		
16.9 Programmierung von Shared Memory in C und C++	21		
16.10 Speicherzugriff zur Laufzeit	21		
16.11 Datenkonsistenz	21		
<b>17 Real-Time Operating Systems (RTOS)</b>	<b>21</b>		
17.1 Operating System (OS) / Betriebssystem	21		
17.2 RTOS / Echtzeitbetriebssystem	21		
17.3 FreeRTOS vs. Zephyr	21		

# 1 Real-Time System (Echtzeitsystem)

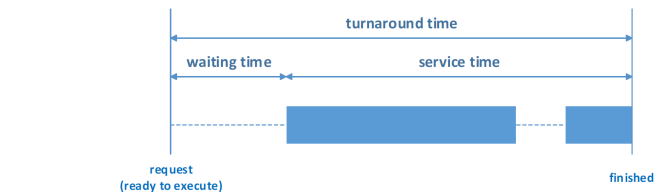
## 1.1 Definitionen

### 1.1.1 Real-Time System (Echtzeitsystem)

- Ein Echtzeitsystem ist ein System, das Informationen **innerhalb einer definierten Zeit (deadline) bearbeiten** muss.  
⇒ Explizite Anforderungen an **turnaround-time** (Antwortzeit) müssen erfüllt sein
- Wenn diese Zeit nicht eingehalten werden kann, ist mit einer **Fehlfunktion** zu rechnen.



### 1.1.2 Zeitdefinitionen (Task)



- turnaround time:** (response time, Antwortzeit)
  - Startet, wenn der Task bereit zur Ausführung ist und endet, wenn der Task fertig abgearbeitet ist
  - Zeit zwischen dem Vorhandensein von Eingangswerten an das System (Stimulus) bis zum Erscheinen der gewünschten Ausgangswerte.
- waiting time:** (Wartezeit)
  - Zeit zwischen Anliegen der Eingangswert und Beginn der Abarbeitung des Tasks
- service time:** (Bearbeitungszeit)
  - Zeit für Abarbeitung des Tasks ⇒ Unterbrechungen bzw. (preemptions) möglich

## 1.2 Fehlverhalten eines Systems (failed system)

- Ein fehlerhaftes System (failed system = missglücktes System) ist ein System, das nicht alle formal definierten Systemspezifikationen erfüllt.
- Die Korrektheit eines RT Systems bedingt sowohl die Korrektheit der Outputs als auch die Einhaltung der zeitlichen Anforderungen.**

## 1.3 Echtzeitdefinition – Verschiedene Echtzeitsysteme

- soft real-time system** (weiches Echtzeitsystem)
  - Durch Verletzung der Antwortzeiten wird das System **nicht** ernsthaft beeinflusst
  - Es kommt zu Komforteinbußen
- hard real-time system** (hartes Echtzeitsystem)
  - Durch Verletzung der Antwortzeiten wird das **System ernsthaft beeinflusst**
  - Es kann zu einem kompletten Ausfall oder katastrophalem Fehlverhalten kommen
- firm real-time system** (festes Echtzeitsystem)
  - Kombination aus soft real-time system und hard real-time system
  - Durch Verletzung einiger weniger Antwortzeiten wird das System nicht ernsthaft beeinflusst
  - Bei vielen Verletzungen der Antwortzeiten kann es zu einem kompletten Ausfall oder katastrophalem Fehlverhalten kommen

### 1.3.1 Beispiele verschiedener Echtzeitsysteme

System	Klassifizierung	Erläuterung
Geldautomat	soft	Auch wenn mehrere Deadlines nicht eingehalten werden können, entsteht dadurch keine Katastrophe. Im schlimmsten Fall erhält ein Kunde sein Geld nicht.
GPS-gesteuerter Rasenmäher	firm	Wenn die Positionsbestimmung versagt, könnte das Blumenbeet der Nachbarn platt gemäht werden.
Regelung eines Quadrocopters	hard	Das Versagen der Regelung kann dazu führen, dass der Quadrocopter ausser Kontrolle gerät und abstürzt.

## 1.4 Determinismus (determinacy)

Ein System ist deterministisch, wenn für jeden möglichen Zustand und für alle möglichen Eingabewerte **jederzeit der nächste Zustand und die Ausgabewerte** definiert sind.

Insbesondere race conditions können dazu führen, dass der nächste Zustand davon abhängt, 'wer das Rennen gewonnen hat und wie gross die Bestzeit ist', d.h. der nächste Zustand ist nicht klar bestimmt.

⇒ Nicht mehr deterministisch und nicht mehr echtzeittauglich

## 1.5 Auslastung (utilization)

Die (CPU-) Auslastung (utilization) ist der Prozentsatz der Zeit, zu der die XPU **nützliche (non-idle) Aufgaben** ausführt.

### 1.5.1 Berechnungen zur Auslastung (utilization)

Annahmen:

- System mit  $n \geq 1$  periodischen Tasks  $T_i$  und Periode  $p_i$
- Jeder Task  $T_i$  hat bekannte / geschätzte maximale (worst case) execution time  $e_i$

#### Auslastungsfaktor eines Tasks

$$u_i = \frac{e_i}{p_i}$$

⇒ utilization factor

#### Gesamtauslastung des Systems

$$U = \sum_{i=1}^n u_i = \sum_{i=1}^n \frac{e_i}{p_i}$$

⇒ utilization

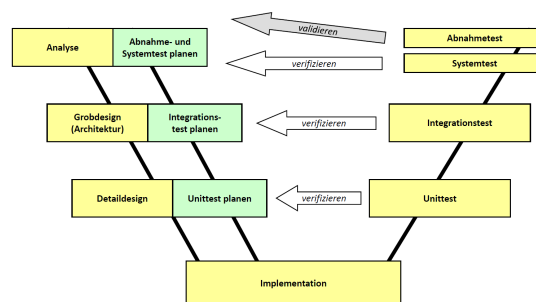
⇒ Bei 69 % Auslastung ist das 'theoretical limit'

## 1.6 Real-time Scheduling

- Alle kritischen Zeiteinschränkungen (deadlines, response time) sollen eingehalten werden
- Im Notfall muss der Scheduling Algorithmus entscheiden, um die kritischsten Tasks einhalten zu können.
  - Unter Umständen müssen dabei Deadlines von weniger kritischen Tasks verletzt werden.

# 2 Modellierung eines Embedded Systems

## 2.1 V-Modell für Software-Entwicklungszyklus



⇒ Nur Anforderungen (requirements) definieren, welche man auch testen kann!

## 2.2 Model Driven Development (MDD)

- Bei **modellbasierter Entwicklung** kommen in **allen Entwicklungsphasen** durchgängig Modelle zur Anwendung
- MDD geht davon aus, dass aus formalen Modellen lauffähige Software erzeugt wird  
⇒ Codegeneratoren
- Modelle werden traditionell als Werkzeug der Dokumentation angesehen
  - Unter Umständen wird zweimal dasselbe beschrieben (Code und Diagramm)  
⇒ **unbedingt zu vermeiden!**

## 2.3 Vorgehen bei der Modellierung

- Systemgrenze definieren**
  - Kontextdiagramm: Use-Case-Diagramm
  - Kontextdiagramm: Sequenzdiagramm
- Systemprozess finden**
  - Kontextdiagramm: Use-Case-Diagramm
  - Kontextdiagramm: Sequenzdiagramm
- Verteilungen festlegen**
  - Verteilungsdiagramm (deployment diagram)
- Systemprozesse detaillieren**
  - Umgangssprachlicher Text
  - Sequenzdiagramm
  - Aktivitätsdiagramm
  - Statecharts
  - Code (C, C++, ...)

Strukturmodellierung (Statische Aspekte)

Modellierung der dynamischen Aspekte

## 2.4 Systemgrenze definieren & Systemprozesse finden

### 2.4.1 Systemgrenze definieren

Die Festlegung der Systemgrenze ist das Wichtigste und Allererste bei **sämtlichen Systemen!**

Man sollte sich die folgenden Fragen stellen und diese beantworten:

- Was macht das System, d.h. was liegt innerhalb der Systemgrenze?
  - Was macht das System **nicht**?
- Mit welchen Teilen ausserhalb des Systems kommuniziert das System?
- Welches sind die Schnittstellen zu den Nachbarsystemen (Umsystemen, peripheren Systemen)?

### 2.4.2 Systemprozesse finden (use-cases)

Da man sich noch immer in der **Analyse** befindet, sollen nur die **Anforderungen** definiert werden. Die Umsetzung ist Teil des Designs!

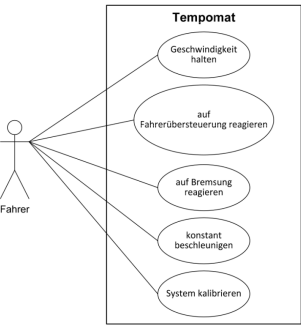
Um die Use-Cases zu identifizieren, sollte folgendes beachtet werden:

- Aussenbetrachtung des Systems (**oberflächlich!**)

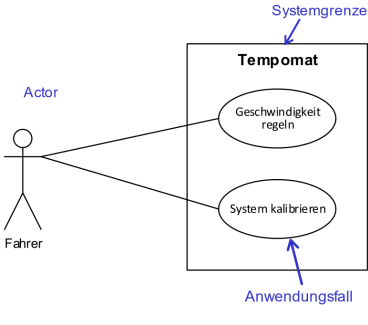
- Nicht komplizierter als nötig
- System als Blackbox betrachten
  - Was soll System können; (nicht: wie soll das System etwas machen)
- RTE-Systeme bestehen häufig aus nur einem einzigen Systemprozess
  - speziell wenn System 'nur' ein Regler ist

## 2.4.3 Kontextdiagramm: Use-Case Diagramm

Tempomat: zu detailliert



Tempomat: verbesserte Version



## 2.4.4 Kontextdiagramm: Sequenzdiagramm

- Speziell bei Systemen, deren Grenzen durch **Nachrichtenflüsse** charakterisiert werden können
- Details zu Sequenzdiagrammen siehe Abschnitt 2.6.1

## 2.5 Verteilungen festlegen

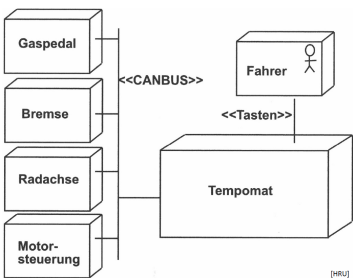
- Bei Embedded Systems werden häufig **mehrere Rechnersysteme** verwendet, um die verschiedenen Aufgaben zu erledigen
- Rechner sind örtlich verteilt und mittels Kommunikationskanal verbunden
  - ⇒ **Verteilte Systeme (distributed systems)**

### 2.5.1 Verteilungsdiagramm

**Knoten:** Darstellung der örtlichen Verteilung der Systeme  
Knoten können auch hierarchisch aufgebaut sein

**Linien:** Physikalische Verbindungen der Knoten (Netzwerke, Kabel, Wireless, etc.)

Beispiel: Tempomat

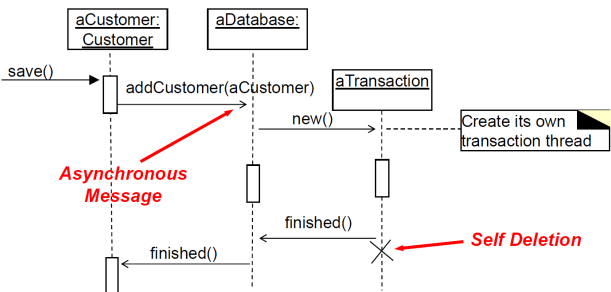
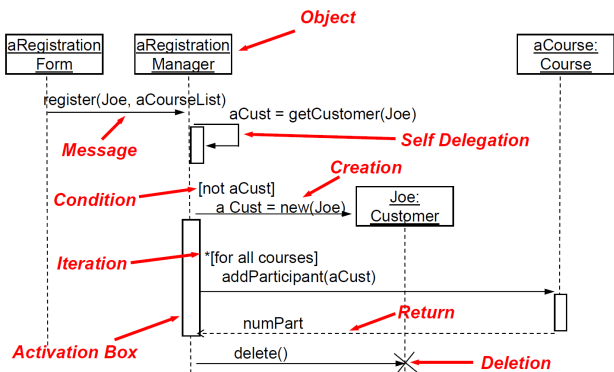


## 2.6 Systemprozesse detaillieren

- Die gefundenen Systemprozesse (use-cases) müssen genauer spezifiziert werden
  - **Nicht detaillierter spezifizieren als sinnvoll / gefordert!**
  - Jede weitere Spezifizierung soll einen 'added value' liefern
- Verschiedene Detaillierungsstufen für verschiedene Zielgruppen
  - Auftraggeber: Überblick (z.B. in Form von Umgangssprachlichem Text)
  - Systementwickler: 'Normale Sicht' enthält mehr Details

### 2.6.1 Sequenzdiagramm

- Gute Darstellung für **Austausch von Meldungen** zwischen Objekten innerhalb einer **beschränkten Zeitdauer**
  - Nachrichtenflüsse
  - Kommunikationsprotokolle
- Ideal für...
  - kurze Zeitdauer
  - wenige Objekte
  - wenige Verschachtelungen
  - wenige Verzweigungen

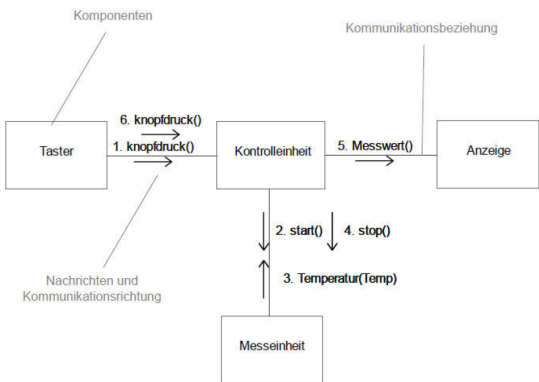


Pfeiltyp	Semantik
	Synchrone Aufrufe
	Asynchrone Nachrichten
	Datenfluss

- Beim Zeichnen von Hand unbedingt die **Pfeilkonventionen** beachten!
- Diagramme generell nicht 'überladen'

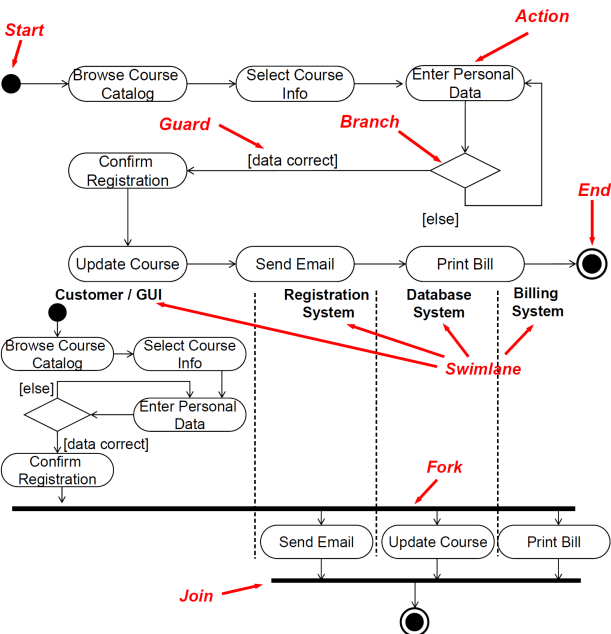
### 2.6.2 Kommunikationsdiagramm (Kollaborationsdiagramm)

- Kommunikationsdiagramm zeigt **dieselbe Information** wie Sequenzdiagramm
- **Schwerpunkt: Informationsfluss** zwischen den Objekten
  - ⇒ Beim Sequenzdiagramm liegt der Schwerpunkt auf dem zeitlichen Ablauf



### 2.6.3 Aktivitätsdiagramm

- Gut geeignet für ...
  - **workflow modelling**
  - Sequenzielle Abläufe
  - Prozess- und Steuerfluss
  - Gleichzeitige Prozesse (fork, join)
- Weniger geeignet für ...
  - komplexe logische Bedingungen



## 3 Hardware-Software-Codesign

### 3.1 Ziele

- Entwurf (Design) **so lange wie sinnvoll** (nicht so lange wie möglich) **lösungsneutral**
- **Systemdesign fördern**, statt separate Designs für Mechanik, Elektronik, Firmware, Software, etc., die sich unter Umständen auch widersprechen können
- Systemspezifikation erfolgt idealerweise mit Hilfe einer **eindeutigen Spezifikationsprache**, nicht in Prosa
- Die Spezifikation sollte simuliert (ausgeführt) werden können
- Implementationen können einfach geändert werden: HW ↔ SW
- Zielplattformen: diskrete Elektronik, ASIC, µC, DSP, **FPGA**, Software

## 3.2 Anforderungen für praktische Anwendungen

- Methoden / Tools sollten beim Systemdesign nicht zu fachlastig sein
  - Methoden sollten für Elektronik-, Firmware- und wenn möglich auch Mechanik-entwickler anwendbar sein
- Wenn möglich gute Toolunterstützung
- (Automatische Synthese aus dem Modell)

## 3.3 Spezifikationssprachen

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• <b>Formale Sprachen sind eindeutig</b> (Prosa immer mehrdeutig)</li><li>• Spezifikation kann kompiliert und ausgeführt werden <math>\Rightarrow</math> Simulationen</li><li>• Die ausführbare Spezifikation dient als <b>Golden Reference</b> für die künftigen Entwicklungsschritte</li></ul> | <b>Beispiele für Spezifikationssprachen</b> <ul style="list-style-type: none"><li>• SystemC (eine C++-Template Library)</li><li>• SysML</li><li>• SpecC</li><li>• SystemVerilog</li><li>• Esterel</li><li>• Matlab/Simulink</li><li>• Statecharts</li></ul> |
|--|---|

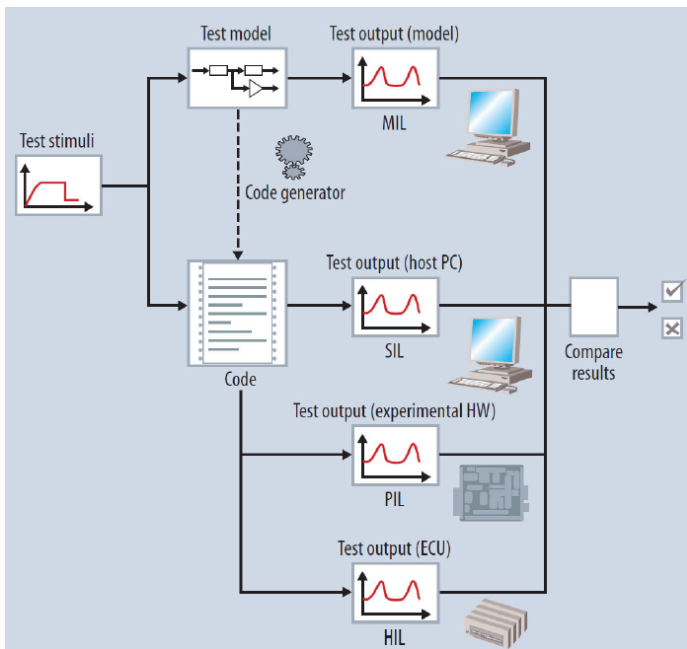
## 3.4 Virtuelle Prototypen

- Die Simulation des Systems kann unterschiedlich stark detailliert werden
- **Die simulierten Systeme sind Virtuelle Prototypen**
- Während der Entwicklung können einzelne (virtuelle) Teile des Prototyps laufend durch physische Teile ersetzt werden

## 3.5 X-in-the-loop

- **Model-in-the-Loop (MIL):** vollständig als Modell vorliegender virtueller Prototyp
- Je mehr der Prototyp durch konkretere Implementationen ersetzt wird, spricht man von
  - Software-in-the loop (SIL)
  - Processor-in-the loop (PIL)
  - Hardwarein-the loop (HIL)

$\Rightarrow$  Test outputs werden jeweils mit **Golden Reference** verglichen



## 3.6 Entwicklungsplattformen

Als Entwicklungsplattformen eignen sich häufig **FPGA basierte Systeme**.

- Hardware mit VHDL
- Software/Firmware in C/C++
  - auf integriertem  $\mu$ C (z.B. Zynq von AMD/Xilinx) (Hard core)
  - auf Soft Core innerhalb FPGA (z.B. Nios II von Intel/Altera)

## 4 Zustandsbasierte Systeme

### 4.1 Asynchrone vs. synchrone FSM

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>• <b>Asynchron</b><ul style="list-style-type: none"><li>– geänderte Inputsignale führen <b>direkt</b> zur Zustandsänderung</li><li>– schneller, aber enorm anfällig auf Glitches</li></ul></li><li>• Softwareimplementationen sind eigentlich immer <b>synchron</b>, da Rechner getaktet sind</li><li>• Rein softwareseitig besteht die Problematik der Asynchronizität nicht</li></ul> | <ul style="list-style-type: none"><li>• <b>Synchron</b><ul style="list-style-type: none"><li>– Inputsignale werden nur zu diskreten Zeitpunkten betrachtet <math>\Rightarrow</math> getaktete Systeme</li></ul></li></ul> |
|---|---|

### 4.2 Finite State Machines (FSM)



Eine FSM besitzt die folgenden Eigenschaften:

- Eine FSM befindet sich immer in einem **definierten Zustand**
- Die **Inputs X** bezeichnen üblicherweise **Ereignisse (Events)**
- Die **Outputs Y** werden oft auch **Actions** genannt
- Eine FSM benötigt immer **Speicherelemente** zur Speicherung des internen Zustands
  - Eine FSM ist ein sequenzielles und kein kombinatorisches System

Eine FSM kann auf zwei Arten dargestellt werden:

- State-Event-Diagramm
- Zustandstabelle

#### 4.2.1 Mealy-Automat

- Nächster Zustand  $Z_{n+1}$  abhängig vom Input  $X$  und vom internen Zustand  $Z_n$ 
  - $Z_{n+1} = f(Z_n, X)$
- Output  $Y$  ist abhängig vom internen Zustand  $Z_n$  **und vom Input X**
  - $Y = g(Z_n, X)$
- Actions liegen bei den Transitionen

#### 4.2.2 Moore-Automat

- Nächster Zustand  $Z_{n+1}$  abhängig vom Input  $X$  und vom internen Zustand  $Z_n$ 
  - $Z_{n+1} = f(Z_n, X)$
- Output  $Y$  ist **nur** abhängig vom internen Zustand  $Z_n$ 
  - $Y = g(Z_n)$
- Actions liegen bei den Zuständen

$\Rightarrow$  Wenn immer möglich sollten Moore-Automaten verwendet werden

#### 4.2.3 Medvedev-Automat

- Nächster Zustand  $Z_{n+1}$  abhängig vom Input  $X$  und vom internen Zustand  $Z_n$ 
  - $Z_{n+1} = f(Z_n, X)$
- Output  $Y$  entspricht **entspricht direkt** internem Zustand  $Z_n$ 
  - $Y = Z_n$
- Actions liegen bei den Zuständen

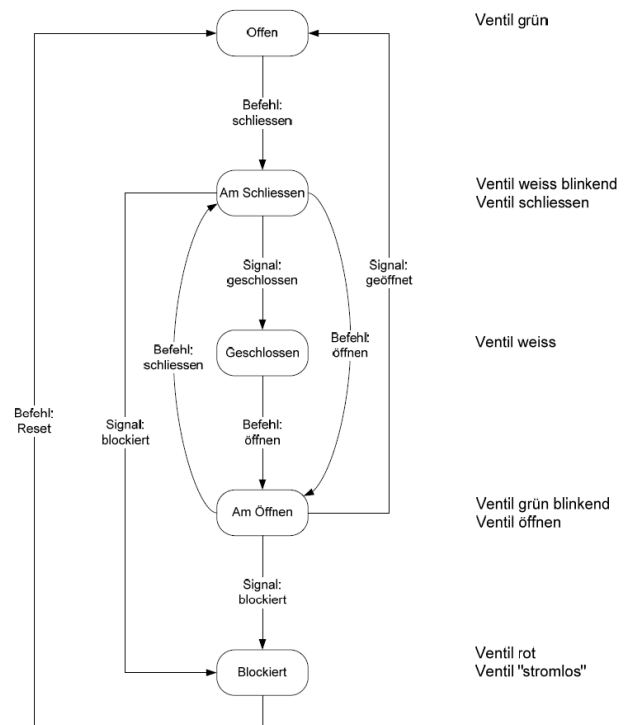
$\Rightarrow$  Wird hier nicht weiter behandelt...

## 4.3 State-Event-Diagramm (Zustandsdiagramm)

Ein State-Event-Diagramm ist eine grafische Möglichkeit, um eine FSM zu beschreiben. In einem State-Event-Diagramm gelten folgende Darstellungsformen

- **Zustände** werden mit einem **Kreis** gezeichnet
- **Ereignisse** werden mit **Pfeilen** zwischen Zuständen dargestellt (**Transitionen**)
- **Aktionen** werden entweder bei Zuständen oder bei Transitionen geschrieben (je nach Automatentyp)
- Ausführung einer Transition ist unendlich schneller
  - $\Rightarrow$  Bei Modellierung sind Zwischenzustände vorgehoben, z.B. 'closing', 'starting up'

### Beispiel: State-Event-Diagramm – Moore Automat



### 4.4 Zustandstabelle

Nebst der grafischen Darstellung einer FSM mittels State-Event-Diagramm kann die FSM auch tabellarisch mittels Zustandstabelle beschrieben werden.



### Beispiel: Zustandstabelle für Elektromotor

Momentaner Zustand	Ereignis	Nächster Zustand	Aktionen
AUS	EIN-Taste	Hochlaufen	Motor ausschalten Kühlung ausschalten Grüne Lampe aus Rote Lampe aus
Hochlaufen	Drehzahl_erreicht Signal	Drehzahl_ok	Motor einschalten Kühlung einschalten
	Aus-Taste	AUS	
	Wasserkühlung Störung	Störung	
Drehzahl_ok	Wasserkühlung Störung	Störung	Grüne Lampe anzeigen
	AUS-Taste	AUS	
Störung	RESET-Taste	AUS	Motor ausschalten Kühlung ausschalten Rote Lampe anzeigen

## 5 Statecharts (nach Marwedel)

## 5.1 Nachteile von State-Event-Diagrammen

- Zustandsdiagramme sind flach (es gibt keine Hierarchie)  $\Rightarrow$  schnell unübersichtlich
- Es kann keine zeitliche Parallelität modelliert werden

## 5.2 Definitionen

**active state:** Aktueller Zustand der FSM

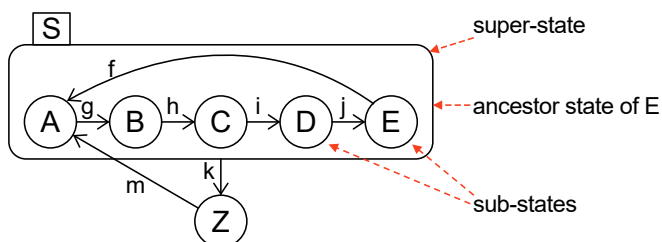
**basic states:** Zustände, die **nicht** aus anderen Zuständen bestehen

**super states:** Zustände, die andere Zustände enthalten

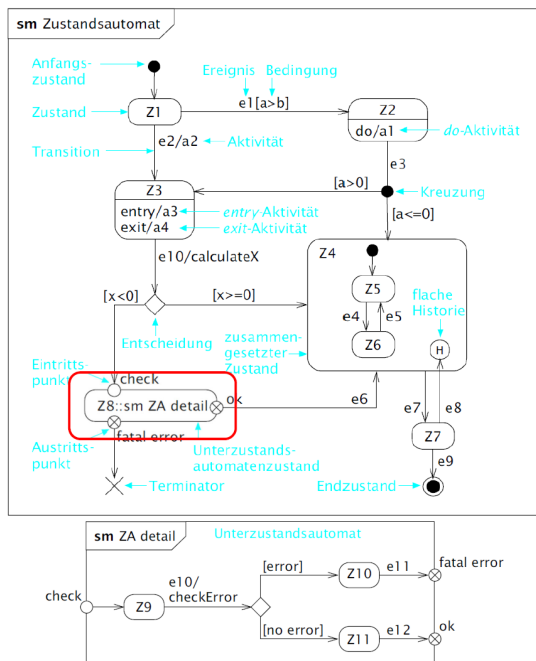
**ancestor states:** Für jeden basic state  $s$  werden die super states, die  $s$  enthalten, als **ancestor states** bezeichnet

**OR-super-states:** Super-states  $S$  werden OR-super-states genannt, wenn **genau einer** der sub-states von  $S$  aktiv ist, wenn  $S$  aktiv ist  $\Rightarrow$  Hierarchie

**AND-super-states:** Super-states  $S$  werden AND-super-states genannt, wenn **mehrere** der sub-states von  $S$  **gleichzeitig** aktiv sind, wenn  $S$  aktiv ist  $\Rightarrow$  Parallelität  
 $\Rightarrow$  Werden auch **Teilautomaten** genannt



### 5.2.1 Elemente der Statecharts



### 5.2.2 Allgemeine Syntax für Transitions-Pfeile

event [guard] / reaction

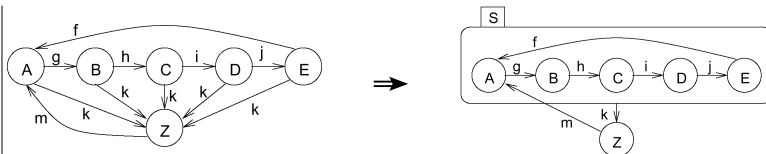
**event** auftretendes Event

**guard** Bedingung, welche zutreffen **muss**, damit Zustand überhaupt gewechselt wird

**reaction** Zuweisung einer Variablen / Erzeugung eines Events beim Zustandswechsel

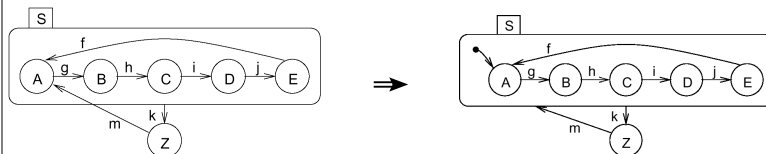
### 5.3 Hierarchie (OR-super-states)

- Die FSM befindet sich in **genau einem** sub-state von  $S$ , wenn  $S$  aktiv ist.  
( $\Rightarrow$  either in  $A$  **OR** in  $B$  **OR** ...)



## 5.4 Default-State

- Ziel: Interne Struktur des states vor der Aussenwelt verstecken  $\Rightarrow$  default state
- Ausgefüllter Kreis beschreibt den sub-state, welcher 'betreten' wird, wenn der super-state  $S$  'betreten' wird



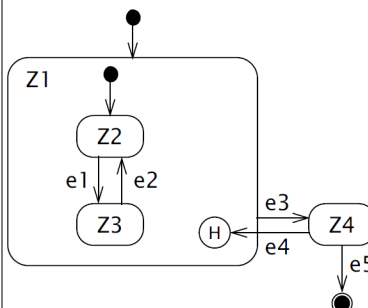
## 5.5 History

- Wenn Input  $m$  auftritt, wird in  $S$  derjenige sub-state betreten, in welchem man war, **bevor  $S$  verlassen wurde**
  - Wenn  $S$  zum ersten Mal betreten wird, ist der **default-Mechanismus** aktiv
- History und Default-Mechanismus können hierarchisch verwendet werden

### 5.5.1 Shallow History

- Der Histroy-Mechanismus merkt sich den entsprechenden sub-state
- Kennzeichnung:  $H$

## Beispiel: Shallow-History

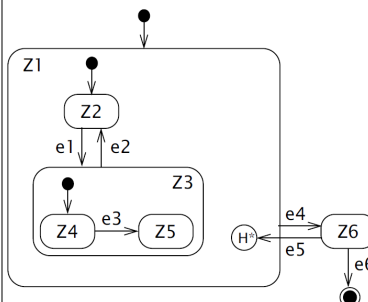


- In Zustand Z3 bewirkt der Event  $e3$  einen Übergang zu Z4
- Der Event  $e4$  führt nun zu einem Übergang zum früheren Zustand Z3

### 5.5.2 Deep History

- Der Histroy-Mechanismus merkt sich frühere Zustände **bis in die unterste Hierarchie**
- Kennezeichnung:  $H^*$

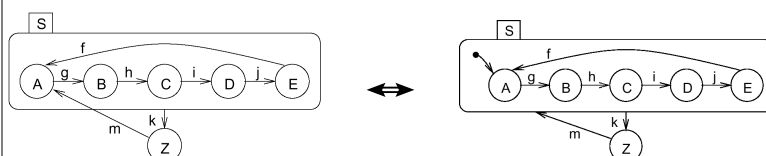
### Beispiel: Deep-History



- Im Zustand Z5 bewirkt der Event  $e4$  einen Übergang zu Z6
- Der Event  $e5$  führt nun zu einem Übergang zum früheren Zustand Z5
  - Eine Shallow History würde bei  $e5$  nur in den Zustand Z3, und damit in Z4, wechseln

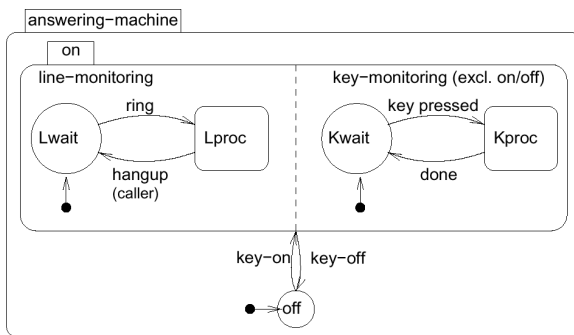
## 5.6 Kombination: History- und Default-Mechanismus

Folgende statecharts bilden genau das Gleiche ab



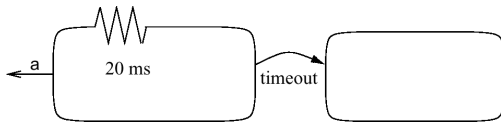
## 5.7 Parallelität (AND-super-state, Teilautomaten)

- Die FSM befindet sich in **allen** sub-states von einem super-state  $S$ , wenn  $S$  aktiv ist.  
( $\Rightarrow$  in  $A$  **AND** in  $B$  **AND** ...)

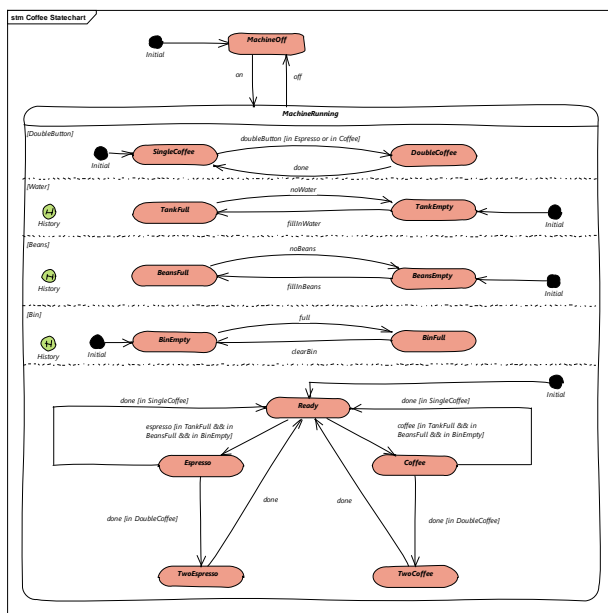


## 5.8 Timers

- Wenn Event  $a$  nicht eintritt, während das System für 20 ms im linken state ist, wird ein timeout passieren
- Eigentlich sind Timers nötig, da die Wartezeit auch als Übergangsbedingung (Ereignis) zwischen zwei states formuliert werden könnte



## 5.9 Beispiel – Armbanduhr als Statechart



## 6 Realisierung flache FSM

### 6.1 Mögliche Realisierungen von flachen FSMs

- Steuerkonstrukt (typischerweise mit **switch-case**)
  - prozedural oder objektorientiert
- Definition und Abarbeitung einer **Tabelle**
  - prozedural oder objektorientiert
- **State Pattern** (Gang of Four, GoF)
  - nur objektorientiert
- Generisch mit Templates
  - nur mit einer Sprache, die Templates unterstützt (z.B. C++)

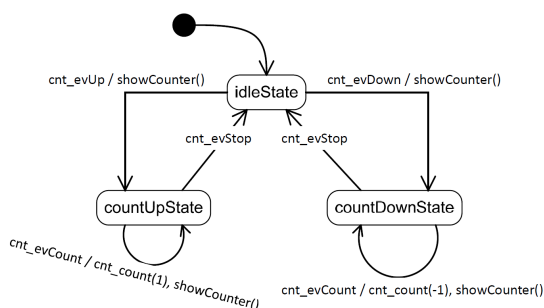
Jede hierarchische FSM kann in eine flache FSM umgewandelt werden.

⇒ Alle Varianten haben wie immer sowohl Vor- als auch Nachteile

⇒ Bei allen Varianten sind auch Variationen vorhanden

### 6.2 Realisierung mit Steuerkonstrukt (prozedural in C)

#### 6.2.1 State-Event-Diagramm – Up/Down-Counter



### 6.2.2 Implementation der Prozeduralen Realisierung in C

- **Ereignisse (events)**
  - Schnittstelle nach aussen ⇒ ändern Zustand der FSM
  - In enum definiert (**public**) ⇒ header-file
  - Einzelne Events und enum Bezeichnung enthalten **Unitkürzel** (hier: `cnt_`)
- **Zustände (states)**
  - In enum definiert (**nicht public**) ⇒ sourcecode-file
- **Aktueller Zustand** wird in einer **statischen Variablen** gehalten
- Die FSM wird in **zwei Funktionen** implementiert
  - Initialisierungs-Funktion (hier: `void cnt_ctrlInit(int initValue)`)
  - Prozess-Funktion (hier: `void cnt_ctrlProcess(cnt_Event e)`)
    - ⇒ Zustände prüfen, Zustandsübergänge veranlassen
- Anstoßen einer FSM
  - Initialisierung in `main`-Funktion
  - Überprüfung, welches Event aufgetreten ist meist in `do-while`-Schleife

### 6.2.3 Eigenschaften der Prozeduralen Realisierung in C

- Da aktueller Zustand eine statische Variable ist, kann es nur **eine einzige Instanz** der FSM geben
- Bei mehreren Instanzen in C...
  - darf `currentState` nicht `static` sein und muss als Parameter mitgegeben werden, bzw. ein Pointer auf die jeweilige Variable
  - Zustands-enum muss in die Schnittstelle (header-file) oder es muss z.B. mit `void*` gearbeitet werden
- In C ist **keine schöne Kapselung** der Attribute möglich (`currentState`)
- Funktion `cnt_ctrlProcess()` kann beliebig aufgerufen werden (periodischer Task, laufend, etc.)
- Bei exponierten Funktionen / Definitionen muss in C ein Unitkürzel vorangestellt werden (hier: `cnt_`)

### Beispiel: Up/Down-Counter (prozedural in C)

#### Schnittstelle Counter:

```
1 // counter.h
2 // implements an up/down-Counter
3
4 #ifndef COUNTER_H_
5 #define COUNTER_H_
6
7 void cnt_init(int val);
8 // initializes counter to val
9
10 void cnt_count(int step);
11 // counts the counter up (step>0)
12 // or down (step<0) by step
13
14 int cnt_getCounter();
15 // returns the counter value
16
17 void cnt_setCounter(int val);
18 // sets the counter to val
19
20 #endif
```

#### Implementation Counter:

```
1 // counter.c
2 // implements an up/down-Counter
3
4 #include "counter.h"
5
6 static int countValue;
7
8 void cnt_init(int val)
9 {
10     countValue = val;
11 }
12
13 void cnt_count(int step)
14 {
15     countValue += step;
16 }
17
18 int cnt_getCounter()
19 {
20     return countValue;
21 }
22
23 void cnt_setCounter(int val)
24 {
25     countValue = val;
26 }
```

Schnittstelle FSM:

```
1 // counterCtrl.h
2 // implements the Finite State Machine (FSM) of an up/down-Counter
3
4 #ifndef COUNTERCTRL_H_
5 #define COUNTERCTRL_H_
6
7 typedef enum {cnt_evUp,      // count upwards
8               cnt_evDown,    // count downwards
9               cnt_evCount,    // count (up or down)
10              cnt_evStop}     // stop counting
11               cnt_Event;
12
13 void cnt_ctrlInit(int initValue);
14 // initializes counter FSM
15
16 void cnt_ctrlProcess(cnt_Event e);
17 // changes the state of the FSM based on the event 'e'
18 // starts the actions
19
20 #endif
```

Implementation FSM:

```
1 // counterCtrl.c
2 // implements the Finite State Machine (FSM) of an up/down-Counter
3
4 #include <stdio.h>
5 #include "counterCtrl.h"
6 #include "counter.h"
7
8 typedef enum {idleState,      // idle state
9               countUpState,   // counting up at each count event
10              countDownState}  // counting down at each count event
11              State;
12
13 static State currentState = idleState; // holds the current state of the FSM
14
15 void cnt_ctrlInit(int initValue)
16 {
17     currentState = idleState; // set init-state
18     cnt_init(initValue);      // set initValue
19 }
20
21 void cnt_ctrlProcess(cnt_Event e)
22 {
23     switch (currentState)
24     {
25     case idleState:
26         printf("State: idleState\n");
27         if (cnt_evUp == e)
28         {
29             // actions (and exit-actions from idleState)
30             printf("State: idleState, counter = %d\n", cnt_getCounter());
31             // state transition (and entry-actions from countUpState)
32             printf("Changing to State: countUpState\n");
33             currentState = countUpState;
34         }
35         else if (cnt_evDown == e)
36         {
37             // actions (and exit-actions from idleState)
38             printf("State: idleState, counter = %d\n", cnt_getCounter());
39             // state transition (and entry-actions from countDownState)
40             printf("Changing to State: countDownState\n");
41             currentState = countDownState;
42         }
43         break;
44
45     case countUpState:
46         printf("State: countUpState\n");
47         if (cnt_evCount == e)
48         {
49             // actions
50             cnt_count(1);
51             printf("State: countUpState, counter = %d\n", cnt_getCounter());
52             // state transition
53         }
54         else if (cnt_evStop == e)
55         {
56             // actions
57             // state transition
58             printf("Changing to State: idleState\n");
59             currentState = idleState;
60         }
61         break;
62
63     case countDownState:
64         // ...
65         break;
66
67     default:
68         break;
69     }
70 }
```

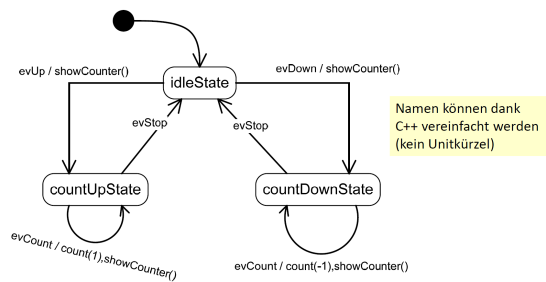
Anstossen der FSM:

```
1 // counterTest.c
2 // Test program for the Finite State Machine (FSM) of an up/down-Counter
3
4 #include <stdio.h>
5 #include "counterCtrl.h"
6
7 int main(void)
8 {
9     char answer;
10    cnt_ctrlInit(0);
11
12    do
13    {
14        printf("\n-----\n");
15        printf("  u  Count up\n");
16        printf("  d  Count down\n");
17        printf("  c  Count\n");
18        printf("  s  Stop counting\n");
19        printf("  q  Quit\n");
```

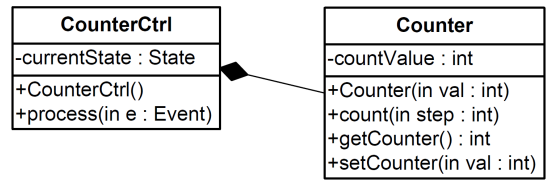
```
20    printf("\nPlease press key: ");
21    scanf("%c", &answer);
22    getchar(); // nach scanf() ist noch ein '\n' im Inputbuffer: auslesen und wegwerfen
23    printf("\n");
24
25    switch (answer)
26    {
27    case 'u':
28        cnt_ctrlProcess(cnt_evUp);
29        break;
30    case 'd':
31        cnt_ctrlProcess(cnt_evDown);
32        break;
33    case 'c':
34        cnt_ctrlProcess(cnt_evCount);
35        break;
36    case 's':
37        cnt_ctrlProcess(cnt_evStop);
38        break;
39    default:
40        break;
41    }
42    while (answer != 'q');
43
44    return 0;
45 }
```

6.3 Realisierung mit Steuerkonstrukt (objektorientiert in C++)

6.3.1 State-Event-Diagramm – Up/Down-Counter



6.3.2 Zusammenhang der Klassen Counter und CounterCtrl



- Klasse Counter führt eigentliche Rechenaufgaben durch
    - ist bei **allen** (objektorientierten) Realisierungsarten **identisch**
  - Klasse CounterCtrl ist FSM, welche Zugriff auf den Counter steuert
- ⇒ Generell sollten Steuerung und Element, das gesteuert wird, getrennt werden!

6.3.3 Implementation der Prozeduralen Realisierung in C++

- Ereignisse (events)
  - Schnittstelle nach aussen ⇒ ändern Zustand der FSM
  - Im **public** Teil der Klasse als enum definiert
  - Keine Unitkürzel nötig
- Zustände (states)
  - Im **private** Teil der Klasse als enum definiert ⇒ header-file
- Aktueller Zustand currentState wird in **privatem Attribut** der Schnittstelle gehalten
- Die FSM wird in **zwei Funktionen** implementiert
  - Kontruktor (hier: CounterCtrl::CounterCtrl(int initValue=0))
  - Prozess-Funktion (hier: void CounterCtrl::process(CounterCtrl::Event e))  
⇒ Zustände prüfen, Zustandsübergänge veranlassen
- Anstossen einer FSM
  - Initialisierung in main-Funktion
  - Überprüfung, welches Event aufgetreten ist meist in **do-while**-Schleife

Beispiel: Up/Down-Counter (prozedural in C++)

Schnittstelle Counter:

```
1 // Counter.h
2 // implements an up/down-Counter
3
4 #ifndef COUNTER_H_
5 #define COUNTER_H_
6
7 class Counter
8 {
9
10 public:
11     Counter(int val = 0);
12
13     void count(int step);
14     // counts the counter up (step>0)
15     // or down (step<0) by step
16
17     int getCounter() const;
18     // returns the counter value
19
20     void setCounter(int val);
21     // sets the counter to val
22 private:
23     int countValue;
24 };
25 #endif
```

Implementation Counter:

```
1 // Counter.cpp
2 // implements an up/down-Counter
3
4 #include "Counter.h"
5
6 Counter::Counter(int val): countValue(val)
7 {
8 }
9
10 void Counter::count(int step)
11 {
12     countValue += step;
13 }
14
15 int Counter::getCounter() const
16 {
17     return countValue;
18 }
19
20 void Counter::setCounter(int val)
21 {
22     countValue = val;
23 }
```

Schnittstelle FSM:

```
1 // CounterCtrl.h
2 // implements the Finite State Machine (FSM) of an up/down-Counter
3
4 #ifndef COUNTERCTRL_H_
5 #define COUNTERCTRL_H_
6 #include "Counter.h"
7
8 class CounterCtrl
9 {
10 public:
11     enum Event{evUp,      // count upwards
12                evDown,    // count downwards
13                evCount,    // count (up or down)
14                evStop};    // stop counting
15
16     CounterCtrl(int initValue = 0); // C-tor
17
18     void process(Event e);
19     // changes the state of the FSM based on the event 'e'
20     // starts the actions
21
22 private:
23     enum State{idleState,      // idle state
24                countUpState,   // counting up at each count event
25                countDownState}; // counting down at each count event
26
27     State currentState;        // holds the current state of the FSM
28     Counter myCounter;         // holds the counter for calculations
29 };
30 #endif
```

Implementation FSM:

```
1 // CounterCtrl.cpp
2 // implements the Finite State Machine (FSM) of an up/down-Counter
3
4 #include <iostream>
5 #include "CounterCtrl.h"
6 #include "Counter.h"
7 using namespace std;
8
9 CounterCtrl::CounterCtrl(int initValue) :
10     currentState(idleState),
11     myCounter(initValue)
12 {
13 }
14
15 void CounterCtrl::process(Event e)
16 {
17     switch (currentState)
18     {
19     case idleState:
20         cout << "State: idleState" << endl;
21         if (evUp == e)
22         {
23             // actions (and exit-actions from idleState)
24             cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
25             // state transition (and entry-actions from countUpState)
26             cout << "Changing to State: countUpState" << endl;
27             currentState = countUpState;
28         }
29         else if (evDown == e)
30         {
31             // actions (and exit-actions from idleState)
32             cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
33             // state transition (and entry-actions from countDownState)
34             cout << "Changing to State: countDownState" << endl;
35             currentState = countDownState;
36         }
37         break;
38
39     case countUpState:
40         cout << "State: countUpState" << endl;
41         if (evCount == e)
42         {
43             // actions
44             myCounter.count(1);
45             cout << "State: countUpState, counter = " << myCounter.getCounter() << endl;
46             // state transition
47         }
48         else if (evStop == e)
49         {
50             // actions
51             // state transition
52             cout << "Changing to State: idleState" << endl;
53             currentState = idleState;
54         }
55         break;
56
57     case countDownState:
58         // ...
59         break;
60
61     default:
62         break;
63     }
64 }
```

Anstossen der FSM:

```
1 // counterTest.cpp
2 // Test program for the Finite State Machine (FSM) of an up/down-Counter
3
4 #include <iostream>
5 #include "CounterCtrl.h"
6 using namespace std;
7
8 int main(void)
9 {
10     char answer;
11     CounterCtrl myFsm(0); // initValue of counter == 0
```

```
13 do
14 {
15     cout << endl << "-----" << endl;
16     cout << "    u    Count up" << endl;
17     cout << "    d    Count down" << endl;
18     cout << "    c    Count" << endl;
19     cout << "    s    Stop counting" << endl;
20     cout << "    q    Quit" << endl;
21
22     cout << endl << "Please press key: ";
23     cin >> answer;
24     cout << endl;
25
26     switch (answer)
27     {
28     case 'u':
29         myFsm.process(CounterCtrl::evUp);
30         break;
31     case 'd':
32         myFsm.process(CounterCtrl::evDown);
33         break;
34     case 'c':
35         myFsm.process(CounterCtrl::evCount);
36         break;
37     case 's':
38         myFsm.process(CounterCtrl::evStop);
39         break;
40     default:
41         break;
42     }
43     while (answer != 'q');
44
45     return 0;
46 }
```

6.4 Realisierung mit Tabelle

6.4.1 State-Event-Diagram – Up/Down-Counter

Siehe Abschnitt 6.3.1

6.4.2 FSM in Tabellenform

Das State-Event-Diagramm wird in eine Tabelle 'übersetzt'. Jede Zeile der Tabelle entspricht einer Transition (Pfeil) im State-Event-Diagramm

Current State	Event	Action	Next State
idleState	evUp	showCounter()	countUpState
idleState	evDown	showCounter()	countDownState
countUpState	evCount	count(1); showCounter();	countUpState
countUpState	evStop	-	idleState
countDownState	evCount	count(-1); showCounter();	countDownState
countDownState	evStop	-	idleState

6.4.3 Implementation der Realisierung mittels Tabelle in C++

- Die ganze FSM ist in einer Tabelle gespeichert
- **Aktionen** sind als **Funktion** implementiert, in der **Tabelle** steht der entsprechende **Funktionspointer**
- **Abarbeitung** der FSM erfolgt mittels **Execution Engine**, die in der Tabelle 'nachschaut', was zu tun ist
  - Execution Engine **ändert sich nicht**, wenn FSM geändert wird!
- **Transition** wird als klasseninterner **struct** deklariert
  - enthält aktuellen Zustand, Event, Funktionspointer auf Aktionsmethode und nächsten Zustand
- **FSM** wird als statischer, offener Array deklariert
  - Hier wird ganze FSM gespeichert
  - ein **struct** bildet konkret eine Zeile der Tabelle ab

6.4.4 Eigenschaften der Realisierung mittels Tabelle

- Die Tabelle kann prozedural oder **objektorientiert** implementiert werden
  - Objektorientierte Variante verwendet einzig die Datenkapselung (keine Vererbung, kein Polymorphismus)
  - Objektorientierte Variante ist klarer / schöner strukturiert
- **Aktions-Funktionen** können **nicht inlined** werden, da ein Pointer auf die Funktionen verwendet wird

6.4.5 Tabelle vs. prozedural

Gemeinsamkeiten	Unterschiede
<ul style="list-style-type: none"><li>• Testprogramm counterTest.cpp</li><li>• Schnittstelle (public-Teil) von Klasse CounterCtrl</li><li>• Gesamte Klasse Counter</li></ul>	<ul style="list-style-type: none"><li>• private-Teil von Klasse CounterCtrl und Implementation davon</li></ul>

Beispiel: Up/Down-Counter (mit Tabelle in C++)

Schnittstelle und Implementation von Counter:

Die Schnittstelle counter.h und die Implementation counter.cpp ändern sich nicht!  
⇒ Code-Beispiele siehe 6.3

Anstossen der FSM:

Die Implementation des Testprogramms counterTest.cpp ändern sich nicht!  
⇒ Code-Beispiele siehe 6.3



Schnittstelle FSM:

```
1 // CounterCtrl.h
2 // implements the Finite State Machine (FSM) of an up/down-Counter as a simple table
3
4 #ifndef COUNTERCTRL_H_
5 #define COUNTERCTRL_H_
6 #include "Counter.h"
7
8 class CounterCtrl
9 {
10 /* ----- NO CHANGES ----- */
11 public:
12     enum Event{evUp,      // count upwards
13                evDown,    // count downwards
14                evCount,    // count (up or down)
15                evStop};    // stop counting
16
17     CounterCtrl(int initValue = 0); // C-tor
18
19     void process(Event e); // execution engine
20     // changes the state of the FSM based on the event 'e'
21     // starts the actions
22
23 private:
24     enum State{idleState, // idle state
25                countUpState, // counting up at each count event
26                countDownState}; // counting down at each count event
27
28     State currentState; // holds the current state of the FSM
29     Counter myCounter; // holds the counter for calculation
30
31 /* ----- CHANGES ----- */
32
33     typedef void (CounterCtrl::*Action)(void); // function ptr for action function
34
35     // action functions (must match with function pointer!)
36     void actionIdleUp(void);
37     void actionIdleDown(void);
38     void actionDoNothing(void); // ensure that there is always a valid fkt-ptr
39     void actionUpUp(void);
40     void actionDownDown(void);
41
42     struct Transition
43     {
44         State currentState; // current state
45         Event ev; // event triggering the transition
46         Action pAction; // pointer to action function
47         State nextState; // next state
48     };
49
50     // static open array for transision structs
51     static const Transition fsm[];
52 };
53 #endif
```

Implementation FSM:

```
1 // CounterCtrl.cpp
2 // implements the Finite State Machine (FSM) of an up/down-Counter as a simple table
3
4 #include <iostream>
5 #include "CounterCtrl.h"
6 #include "Counter.h"
7 using namespace std;
8
9 const CounterCtrl::Transition CounterCtrl::fsm[] = // this table defines the fsm
10 {
11     {idleState, evUp,      &CounterCtrl::actionIdleUp, countUpState},
12     {idleState, evDown,    &CounterCtrl::actionIdleDown, countDownState},
13     {countUpState, evCount, &CounterCtrl::actionUpUp, countUpState},
14     {countUpState, evStop,  &CounterCtrl::actionDoNothing, idleState},
15     {countDownState, evCount, &CounterCtrl::actionDownDown, countDownState},
16     {countDownState, evStop, &CounterCtrl::actionDoNothing, idleState}
17 };
18
19 CounterCtrl::CounterCtrl(int initValue) : // initializations with initialization list
20     currentState(idleState),
21     myCounter(initValue)
22 {
23 }
24
25 void CounterCtrl::process(Event e) // execution engine, this function never changes!
26 {
27     // determine number of transitions automatically
28     for (size_t i = 0; i < sizeof(fsm) / sizeof(Transition); ++i)
29     {
30         // is there an entry in the table?
31         if (fsm[i].currentState == currentState && fsm[i].ev == e)
32         {
33             (this->*fsm[i].pAction)();
34             currentState = fsm[i].nextState
35             break;
36         }
37     }
38 }
39
40 // action functions
41 void CounterCtrl::actionIdleUp(void)
42 {
43     cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
44 }
45
46 void CounterCtrl::actionIdleDown(void)
47 {
48     cout << "State: idleState, counter = " << myCounter.getCounter() << endl;
49 }
50
51 void CounterCtrl::actionDoNothing(void)
52 {
53 }
54
55 // ...
```

6.5 Erweiterung der Realisierung mittels Tabellen

- Wenn der Zustandsübergang nicht durch einen Event, sondern eine **komplexere Prüfung (Event und Guard)** ausgelöst wird, dann könnte der **Event-Eintrag** in der Tabelle durch einen weiteren **Funktionspointer** auf eine **Checkfunktion** ersetzt werden.
- Ergänzung für die Behandlung von Entry- und Exit-Actions

Beispiel: Up/Down-Counter (mit Checker-Tabelle in C++)

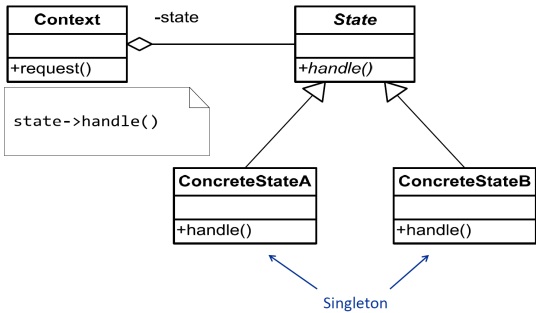
- Änderungen in CounterCtrl.h => siehe Beispiel-Code
- Änderungen in CounterCtrl.cpp
  - checker-Funktionen müssen implementiert werden
  - In Tabelle steht statt Event die Adresse der checker-Funktion (analog zu action-Funktionen)

```
1 // ...
2
3 typedef bool (CounterCtrl::*Checker)(Event); // function ptr for checker function
4 typedef void (CounterCtrl::*Action)(void); // no change here!
5
6 // check functions
7 bool checkIdleUp(Event e);
8 bool checkIdleDown(Event e);
9 // ...
10
11 struct Transition
12 {
13     State currentState; // current state
14     Checker pChecker; // pointer to checker function
15     Action pAction; // pointer to action function
16     State nextState; // next state
17 };
18
19 // ...
```

6.6 Realisierung mit StatePattern

6.6.1 Grundidee von StatePatterns

Das Grundkonzept von StatePatterns ist **Polymorphismus** (Vererbung)



- **Context-Klasse**
  - definiert **Schnittstelle** für Clients
  - unterhält eine **Instanz** einer konkreten Unterklasse von state, die den aktuellen Zustand repräsentiert
- **State-Klasse**
  - definiert die Schnittstelle zur FSM in Form einer **abstrakten Klasse**
- **ConcreteStateX** Unterklassen
  - Jede Unterklasse (Singleton) implementiert genau **einen Zustand**

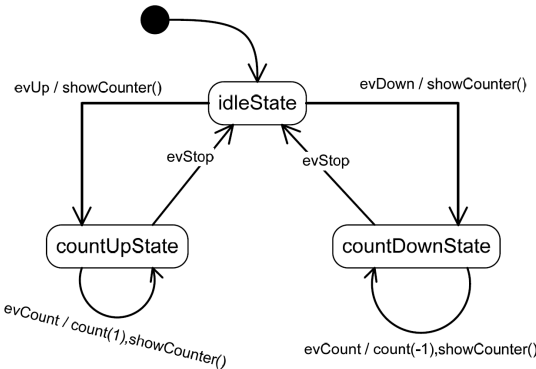
6.6.2 Transitions in StatePatterns

StatePattern definiert nicht, wo die Transitions umgesetzt werden sollen. Es gibt daher die zwei folgenden Varianten.

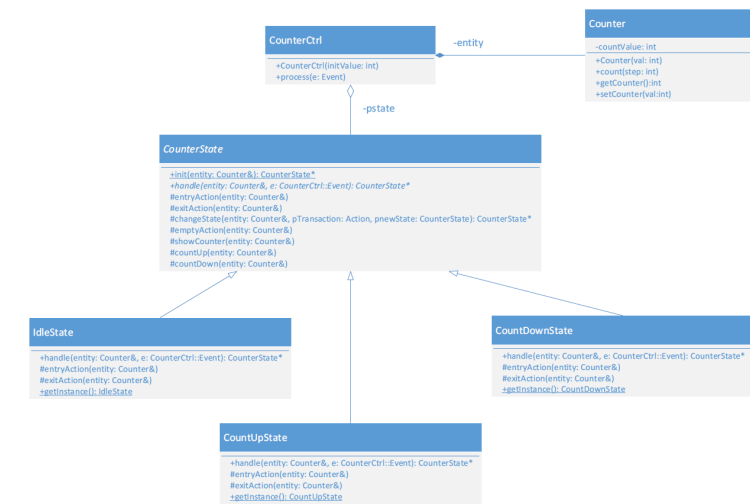
=> **Variante 2 ist klar zu bevorzugen!**

1. Transitionen könnten in der context -Klasse definiert werden.  
Nachteil: dort müsste zentral sehr viel Intelligenz vorhanden sein  
Da diese Klasse auch den Zugriff von der Aussenwelt darstellt, sollte sie möglichst schlank sein.
2. State-Klassen **realisieren ihre Transitionen selbst.**  
=> wir oft mittels **friend**-Deklaration realisiert, was jedoch nicht nötig ist

6.6.3 State-Event-Diagram – Up/Down-Counter



## 6.6.4 Klassendiagramm – Up/Down-Counter



## 6.6.5 Implementation der Realisierung mittels StatePattern

- **Ereignisse (events)**
  - Schnittstelle nach aussen ⇒ ändern Zustand der FSM
  - Im **public** Teil der abstrakten Basisklasse als enum definiert
- **Zustände (states)**
  - Jeder Zustand als eigene (Sub-)Klasse definiert
- **Aktueller Zustand pState** wird in **privatem Attribut (Pointer!)** der Schnittstelle gehalten
  - Es braucht daher in der **Context**-Klasse eine **forward declaration** der **State**-Klasse
- Die FSM wird in **zwei Funktionen** implementiert
  - Kontruktor (hier: `CounterCtrl::CounterCtrl(int initValue=0)`)
  - Prozess-Funktion (hier: `void CounterCtrl::process(CounterCtrl::Event e)`)
    - ⇒ Zustände prüfen, Zustandsübergänge veranlassen
- **Entry- und Exit-Actions**
  - Können in Basisklassen-Methode `changeState()` isoliert vorgenommen werden
  - Zwischen Exit- und Entry-Action müssen allfällige Transition-Actions ausgeführt werden. Diese wird in der Methode `changeState()` als **Funktionspointer** übergeben
  - In Basisklasse werden zwei **virtuelle Methoden** `entryAction()` und `exitAction()` deklariert
    - ⇒ Default-Implementation sinnvoll!

## Beispiel: Up/Down-Counter mit StatePattern

### Schnittstelle und Implementation von Counter:

Die Schnittstelle `counter.h` und die Implementation `counter.cpp` ändern sich nicht!

⇒ Code-Beispiele siehe 6.3

### Schnittstelle zur FSM:

```
1 // CounterCtrl.h
2 // implements the Finite State Machine (FSM) of an up/down-Counter
3
4 #ifndef COUNTERCTRL_H_
5 #define COUNTERCTRL_H_
6 #include "Counter.h"
7
8 class CounterState; // forward declaration
9
10 class CounterCtrl // this is the 'Context' class of the State pattern
11 {
12 public:
13     enum Event {evUp, // count upwards
14                 evDown, // count downwards
15                 evCount, // count (up or down)
16                 evStop}; // stop counting
17     CounterCtrl(int initValue = 0);
18     void process(Event e);
19     // changes the state of the FSM based on the event 'e'
20
21 private:
22     Counter entity;
23     CounterState* pState; // holds the current state
24 };
25 #endif
```

### Implementation der FSM:

```
1 // CounterCtrl.cpp
2 // implements the Finite State Machine (FSM) of an up/down-Counter
3 // CounterCtrl is the Context class in the State pattern
4
5 #include "Counter.h" // only needed if there is an entryAction in initState
6 #include "CounterCtrl.h"
7 #include "CounterState.h"
8
9 CounterCtrl::CounterCtrl(int initValue):
10     entity(initValue), // use either line 11 or 12
11     pState(CounterState::init(entity)) // initial state incl. entryAction
12 { pState(IdleState::getInstance()) // initial state without entryAction
13 {
14 }
15
16 void CounterCtrl::process(Event e)
17 { // delegates all requests to CounterState
18     pState = pState->handle(entity, e);
19 }
```

### Schnittstelle abstrakte State-Basisklasse:

```
1 // CounterState.h
2 // implements an up/down-Counter
3
4 #ifndef COUNTERSTATE_H_
5 #define COUNTERSTATE_H_
6 #include "CounterCtrl.h" // Events are defined here
7
8 class CounterState // abstract base class
9 {
10 public:
11     // should be called first, returns new state (if actions are used)
12     static CounterState* init(Counter& entity);
13
14     virtual CounterState* handle(Counter& entity, CounterCtrl::Event e) = 0;
15     // returns new state
16 protected: // only inherited classes may use these member functions
17
18     // if actions are used:
19     virtual void entryAction(Counter& entity) {};
20     virtual void exitAction(Counter& entity) {};
21     typedef void (CounterState::*Action)(Counter& entity); // ptr to action function
22
23     // if actions are used: transition actions
24     void emptyAction(Counter& entity) {};
25     void showCounter(Counter& entity);
26     void countUp(Counter& entity);
27     void countDown(Counter& entity);
28
29     // always (see extra comment)
30     CounterState* changeState(Counter& entity,
31                               Action ptransAction, // only if actions are used
32                               CounterState* pNewState);
33
34 };
```

### Implementation abstrakte State-Basisklasse:

```
1 // CounterState.cpp
2 // implements all states of an up/down-Counter
3
4 #include <iostream>
5 #include "CounterState.h"
6 using namespace std;
7
8 // only if actions are used:
9 CounterState* CounterState::init(Counter& entity) // it's static
10 {
11     CounterState* initState = IdleState::getInstance();
12     initState->entryAction(entity); // executes entry action into init state
13     return initState;
14 }
15
16 CounterState* CounterState::changeState(Counter& entity,
17                                         Action ptransAction, // only with actions
18                                         CounterState* pNewState)
19 {
20     exitAction(entity); // polymorphic call of exit action
21     (this->*ptransAction)(entity); // call of transition action
22     pNewState->entryAction(entity); // polymorphic call of entry action
23     return pNewState;
24 }
25
26 // default implementations of entryActions() and exitAction()
27 // ...
```

### Schnittstelle ConcreteStateX-Klassen (CountUpState):

```
1 // CountUpState.h
2 // interface of the CountUpState of an up/down-Counter
3
4 #ifndef COUNTUPSTATE_H_
5 #define COUNTUPSTATE_H_
6 #include "CounterCtrl.h" // Events are defined here
7
8 class CountUpState : public CounterState // it's a singleton
9 {
10 public:
11     static CountUpState* getInstance();
12     CounterState* handle(Counter& entity, CounterCtrl::Event e) override;
13
14 /* ----- if actions are used ----- */
15 protected:
16     void entryAction(Counter& entity) override; // only if default is not enough
17     void exitAction(Counter& entity) override; // only if default is not enough
18 /* ----- */
19 private:
20     CountUpState();
21 };
22 #endif
```

### Implementation ConcreteStateX-Klassen (CountUpState – no actions):

```
1 // CountUpState.cpp
2 // implements the CountUpState of an up/down-Counter without actions
3
4 #include <iostream>
5 #include "CountUpState.h"
6 #include "CounterCtrl.h" // Events are defined here
7 using namespace std;
8
9 CountUpState* CountUpState::getInstance()
10 {
11     static CountUpState instance;
12     return &instance;
13 }
```

```

15 CounterState* CountUpState::handle(Counter& entity, CounterCtrl::Event e)
16 {
17     cout << "State: countUpState" << endl;
18     if (CounterCtrl::evCount == e)
19     {
20         // transition actions
21         entity.count(1);
22         cout << "Counter = " << entity.getCounter() << endl;
23         // state transition
24         return changeState(entity, CountUpState::getInstance());
25     }
26     else if (CounterCtrl::evStop == e)
27     {
28         // transition actions
29         // state transition
30         return changeState(entity, IdleState::getInstance());

```

### Implementation ConcreteStateX-Klassen (CountUpState – with actions):

```

1 // CountUpState.cpp
2 // implements the CountUpState of an up/down-Counter with actions
3
4 #include <iostream>
5 #include "CountUpState.h"
6 #include "CounterCtrl.h" // Events are defined here
7 using namespace std;
8
9 // class CountUpState
10 CountUpState* CountUpState::getInstance()
11 {
12     static CountUpState instance;
13     return &instance;
14 }
15
16 CounterState* CountUpState::handle(Counter& entity, CounterCtrl::Event e)
17 {
18     cout << "State: countUpState" << endl;
19     if (CounterCtrl::evCount == e)
20     {
21         // state transition
22         return changeState(entity, &CountUpState::countUp, CountUpState::getInstance());
23     }
24     else if (CounterCtrl::evStop == e)
25     {
26         // state transition
27         return changeState(entity, &CountUpState::emptyAction, IdleState::getInstance());
28     }
29     return this;
30 }
31
32 void CountUpState::entryAction(Counter& entity)
33 {
34     cout << "Entering countUpState" << endl;
35 }
36
37 void CountUpState::exitAction(Counter& entity)
38 {
39     cout << "Exiting from countUpState" << endl;
40 }

```

### Anstossen der FSM:

Die Implementation des Testprogramms counterTest.cpp ändern sich nicht!  
 => Code-Beispiele siehe 6.3

## 7 Modularisierung

Ziel der Modularisierung ist eine **Reduktion der Komplexität**.

$$\sum_i \text{complexity}(\text{problem})_i < \text{complexity}\left(\sum_i \text{problem}_i\right)$$

### 7.1 Grundprinzip Modularisierung

- **Problem in (einfachere) Unterprobleme aufteilen** und diese Unterprobleme jeweils **einzeln angehen**
- Abstraktion

#### 7.1.1 Motivation für Modularisierung

- **Grosse Projekte – 'richtige' Softwaresysteme**
  - Systematischer Designansatz und strukturierter Aufbau ermöglichen effiziente **Arbeit im Team**
  - Schnittstellen müssen klar definiert werden
- **Informatin Hiding**
  - Für die Nutzung eines Moduls (Unit) muss es genügen, **nur die Schnittstellen** zu kennen

#### 7.1.2 Phasenunterteilung beim Entwurf

- **Grobentwurf, Architektur (architectural design)**
  - (Software-) System im Grossen
  - Schnittstellen zu anderen (Nicht-Software-) Systemen
  - Datenstruktur im Grossen
  - **Aufteilung in Subsysteme**
  - **Schnittstellen zwischen Subsystemen**
- **Feinentwurf**
  - Innenleben und Datenstruktur im Kleinen

### 7.2 Bewertung einer Zerlegung

- **Kopplung (coupling)**
  - Mass für Komplexität der Schnittstelle
- **Kohäsion (cohesion)**
  - Aussage, wie stark eine funktionale Einheit wirklich zusammengehört
  - Mass die die Stärke des inneren Zusammenhangs

=> Ziel ist eine schwache Kopplung mit starker Kohäsion!

### 7.3 Kopplung

schwach  
(gut)

stark  
(schlecht)

- **Keine direkte Kopplung**
- **Datenkopplung**
  - Kommunikation ausschliesslich über Parameter
- **Datenbereichskopplung**
  - Ein Modul hat Zugriff auf eine Datenstruktur eines anderen Moduls. Es werden allerdings nur einzelne Komponenten wirklich benötigt.
- **Steuerflusskopplung (control flow)**
  - Ein Modul beeinflusst Steuerfluss eines anderen Moduls
- **Globale Kopplung**
  - Kommunikation über globale Variablen, jedes Modul hat Zugriff
- **Inhaltskopplung (Todsünde!)**
  - Aus einem Modul heraus werden lokale Daten eines anderen Moduls modifiziert, obwohl dieses Modul gar nicht vom anderen Modul aufgerufen wird.

### 7.4 Kohäsion

schwach  
(gut)

stark  
(schlecht)

- **funktional**
  - Die Teile einer Einheit bilden zusammen eine Funktion, bzw. eine Funktionsgruppe
- **sequentiell**
  - Teilfunktionen einer Einheit werden nacheinander ausgeführt, wobei das Resultat einer Funktion als Eingabe für die nächste verwendet wird
- **kommunikativ**
  - Die Teilfunktionen einer Einheit werden auf den gleichen Daten ausgeführt, Reihenfolge spielt keine Rolle
- **prozedural**
  - Teilfunktionen werden nacheinander ausgeführt, verknüpft über Steuerfluss
- **zeitlich**
  - Die Teile einer Einheit sind alle zu einer bestimmten Zeit auszuführen
  - Typischer Fall: alle Initialisierungsfunktionen werden zusammengefasst
- **logisch**
  - (nicht zusammengehörende) Teilfunktionen einer Einheit gehören zu einer Einheit
- **zufällig**
  - Die Teilfunktionen einer Einheit haben keinen sinnvollen Zusammenhang

#### 7.4.1 Ziele bezüglich Kohäsion

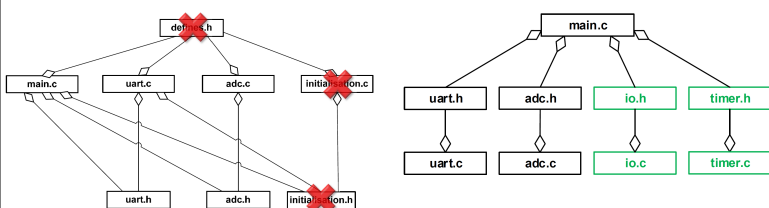
- Kohäsion soll maximiert werden  
 => **starke Kohäsion führt automatisch zu schwacher Kopplung!**
- Den genauen Wert der Kohäsion zu ermitteln ist **kein Ziel**  
 => **Zusammengehörendes zusammennehmen!**

### 7.5 Guidelines – gute Modularisierung

- **Zusammengehörendes zusammennehmen**
  - Definiert für spezifisches Modul in Header-File des Moduls
- Passende / aussagekräftige Namen für Variablen
- 'Interne' (private) Funktionen in .c-file deklarieren und definieren
- Schnittstellenbeschreibung in Header-Dateien
  - Falls möglich: Doxygen verwenden
- Lokale Funktionen (z.B. in main.c) bei Funktionsdeklarationen kommentieren
- Allenfalls 'globalen' Header für Typdefinitionen
  - besser: Typen aus stdint.h verwenden
- **uint8\_t** etc. verwenden, wenn gezielt ein 8 Bit Register angesprochen wird (und nur dann!)
- Keine initialization.h Dateien => zeitliche Kohäsion!
  - generell keine Dateien wie: global.h, defines.h, util.h, project.h

**Hinweis:** Für die Zurechtfindung in einem bestehenden Projekt müssen generell immer zuerst die **Header-Files studiert werden!**

#### Beispiel: Schlechte vs. gute Modularisierung

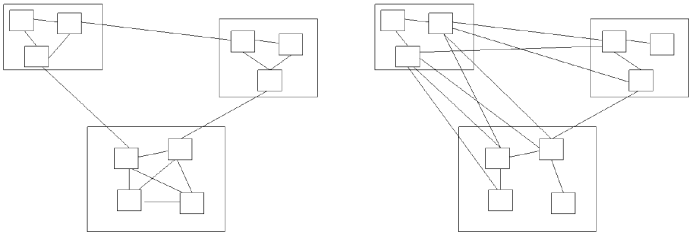


### 7.6 Package-Diagramm

- Ein Package besteht aus mindestens einer, üblich aus mehreren Klassen, die zusammengehören (Stichwort: **Kohäsion**)
- Im Package-Diagramm kann dargestellt werden, welche Packages mit welchen anderen Packages Verbindungen haben (dürfen)
  - **Abhängigkeiten zwischen Packages** können sichtbar gemacht werden

- Packagekonzept in C++: Namespaces umgesetzt
  - ein Namespace entspricht einem Package

### Beispiel: Schlechtes vs. gutes Packaging



links: hohe Kohäsion, tiefe Kopplung => gut  
rechts: tiefe Kohäsion, hohe Kopplung => schlecht

## 8 Patterns (Lösungsmuster)

Ein Software Pattern ist eine bekannte Lösung für eine Klasse von Problemen.

- + Rad muss nicht immer neu erfunden werden
- + Getestete / funktionierende Lösungen
- Wichtige Patterns müssen bekannt sein
- Problemstellungen müssen als solche erkannt werden

### 8.1 Arten von Patterns

- **Architekturmuster (Architectural Pattern)**
  - Legt die grundlegende Organisation einer Anwendung und die Interaktion zwischen den Komponenten fest
- **Entwurfsmuster (Design Pattern)**
  - Die ursprüngliche Form des Pattern-Ansatzes
- **Implementationsmuster (Implementation Pattern)**
  - Behandelt grundsätzliche Implementationen immer wiederkehrender Codefragmente

### 8.2 Wichtige Patterns für Embedded Systems

#### 8.2.1 Bereits bekannte Patterns

- **FSM Implementationen**
  - State Pattern
  - Singleton Pattern
  - (Steuerkonstrukt mit switch-case)
  - (Tabellenvariante)
- **'Mini-Patterns'**
  - Setzen / Löschen einzelner Bits
  - Behandlung asynchroner Ereignisse
    - \* Interrupts
    - \* Polling

#### 8.2.2 Creational Patterns

Creational Patterns behandeln die **Erzeugung (und Vernichtung) von Objekten**.

- **Factory (Dependency injection)**
  - Definition einer Schnittstelle zur Erzeugung eines Objekts, statt der direkten Erzeugung auf der Client-Seite
- **Singleton**
  - stellt sicher, dass eine Klasse nur **ein einziges Objekt** besitzt
- **RAII (Resource Acquisition Is Initialization)**
  - Die Belegung und Freigabe einer Ressource wird an die Lebensdauer eines Objektes gebunden. Dadurch wird eine Ressource z.B. 'automatisch' freigegeben.

#### 8.2.3 Structural Patterns

Structural Patterns **vereinfachen Beziehungen** zu anderen Teilen.

- **Adapter (Wrapper, Translator, glue code)**
  - Wandelt (adaptiert) eine Schnittstelle in eine für einen Client passendere Schnittstelle um
- **Facade**
  - Bietet eine **einfache Schnittstelle** für die Nutzung einer meist viel **grösseren Library**
- **Proxy**
  - 'A proxy, in its most general form, is a class functioning as an interface to something else.'
  - Oft ist es eine SW-Repräsentation eines HW-Teils, z.B. die Repräsentation einer Netzwerkverbindung

#### 8.2.4 Behavioral Patterns

Behavioral Patterns identifizieren **gemeinsame Kommunikationspatterns** zwischen Objekten und implementieren diese.

- **Mediator**
  - definiert ein Objekt, welches das Zusammenspiel einer Menge von Objekten regelt
  - ein Embedded System, das aus **mehreren Teilen** wie Sensoren und Aktoren besteht, wird **im Mediator softwaremässig zusammengebaut**
- **Observer (MVC)**
  - Nicht nur bei Embedded Systems wichtig
  - Wird als objektorientierte Variante präsentiert
  - MVC-Prinzip kann auch prozedural mit Callbackfunktionen implementiert werden

**Beispiel Mediator:** Bei einem Drucker mit mehreren Druckaufträgen von mehreren Personen teilt der Mediator die Aufträge jeweils korrekt

### 8.2.5 Concurrency Patterns

Concurrency Patterns kümmern sich um die **Ausführung in multi-threaded Umgebungen**.

- **Active Object**
  - entkoppelt den Methodenaufwurf von der Methodenausführung
  - Methode soll sich nicht kümmern, in welchem Kontext sie aufgerufen wird
- **Lock**
  - Synchronisationsprimitive, welche den unteilbaren Zugriff read-modify-write implementiert
- **Monitor**
  - Monitor versteckt Synchronisationsanforderungen vor Client

## 9 Event-based Systems

### 9.1 Ereignisse (Events)

Reaktive Systeme reagieren auf (oft externe) Ereignisse (z.B. Digitale Inputs, Timer, Buttonclicks, etc.). Solche **Ereignisse sind per Definition asynchron und treten somit zu einem beliebigen Zeitpunkt auf**. Die Ereignisse können jedoch **synchron oder asynchron** umgesetzt werden.

### 9.2 Synchrone Umsetzung von Ereignissen

Ein **'normales' Programm** ist immer **synchron**. (Programm gibt vor, was wann ausgeführt wird.)

#### 9.2.1 Polling

- Programm fragt periodisch oder dauernd ab, ob irgendein Ereignis eingetreten ist
- Maximale Reaktionszeit wird durch Abfrageperiode und Anzahl Abfragen definiert (Looptime bei SPS)
- + Sehr einfach zu implementieren
- Leerabfragen (Abfragen, bei welchen nichts eingetreten ist) können durch periodisches Abfragen (mittels Timer) reduziert, aber nicht vermieden werden

### 9.3 Asynchrone Umsetzung von Ereignissen

Ziel der asynchronen Verarbeitung von Events ist es, dass die Prozessorzeit **genau dann und nur dann** beansprucht wird, wenn ein Ereignis eingetreten ist. => Interrupts

### 9.4 Interrupt-Verarbeitung

1. I/O-Element generiert einen Interrupt Request
2. Die CPU unterbricht das laufende Programm
3. **Die Interrupts werden disabled (ausgeschaltet)**
4. Das I/O-Element wird informiert, dieses deaktiviert den Interrupt Request
5. Die Interrupt Service Routine (ISR) wird ausgeführt
6. **Die Interrupts werden wieder enabled (eingeschaltet)**
7. Die CPU führt das Programm an der unterbrochenen Stelle weiter

**Sprungadresse nach Interrupt-Auslösung (ISR):**

- **Non-vectored Interrupt (zentral)**
  - Alle Interrupts verzweigen zu einer **gemeinsamen Adresse**. Dort wird die Ursache bestimmt und zu einer spezifischen Behandlungsroutine verzweigt.
  - + Nur eine zentrale Routine für die Behandlung notwendig
  - Information über die Ursache ist beim Eintreten bereits bekannt. Dann verzweigt man in die zentrale Routine, d.h. diese Information ist dann verloren. In der Routine muss diese Information wieder ermittelt werden.
- **Vectored Interrupt (spezifisch)**
  - In einer Tabelle (**Interruptvektortabelle**, IVT) wird gespeichert, wohin bei welchem Interruptvektor verzweigt werden muss.
  - => zu bevorzugende Methode!

### 9.5 Interruptvektortabelle (IVT)

Für jeden Vektor muss eingetragen werden, welches die **Anfangsadresse** der Interrupt Service Routine (ISR) ist, d.h. die **IVT ist nichts anderes als eine Tabelle (Array) von Funktionspointern**.

=> Dieses Konzept kommt bei **allen asynchronen Mechanismen** zur Anwendung

### 9.6 Model View Controller (MVC) aka Observer Pattern

Ausgangslage: Daten (model) und verschiedene Darstellungsformen (views) der Daten (z.B. Balkendiagramm, Kuchendiagramm, Tabelle, etc.)

=> **Die views (clients) sollen unbedingt vom model (server) getrennt werden!**

Wie kann nun erreicht werden, dass bei **jeder Änderung** der Daten (model) alle Darstellungen aktualisiert werden? => Callback-Funktionen!

### 9.7 Callback-Funktionen

- + Views werden **asynchron** genau informiert, wenn sich etwas im **model geändert** hat
- + An und für sich sind alle registrierten Funktionen nichts anderes als **EventHandler eines bestimmten Events** => Darstellung (Definition der registrierten Funktionen) sauber von den Daten (model) **entkoppelt**

### 9.8 Umsetzung der Callback-Funktionen in C (clientseitig)

**Event registrieren (attach):**

- Jeder client meldet beim server an, welche Ereignisse ihn interessieren
  - Anmeldung erfolgt über eine Funktion, welche der server anbietet

```
1 int foo_registerCb(foo_Event e, foo_cbFunction f);
2 // registers function 'f' on event 'e' -> 'id' is returned
3 // sometimes called attach()
```



- Der server trägt diesen **Funktionspointer** `f` in eine Tabelle ein und ruft **beim Eintreten des Ereignisses alle registrierten Funktionen** der Reihe nach je über den einge-tragenen Funktionspointer auf

#### Event austragen (detach):

- Ein client kann sein Interesse an einem Ereignis beim Server auch wieder austragen
  - Abmeldung erfolgt über eine Funktion, welche ebenfalls der server anbietet
  - Der Server löscht dann den entsprechenden Eintrag (**Funktionspointer** `f`) wieder aus der Tabelle

```
1 int foo_unregisterCb(foo_Event e, int id);
2 // unregisters functionId 'id' on event 'e' -> 'id' is returned
3 // sometimes called detach()
```

## 9.9 Umsetzung der Callback-Funktionen in C (serverseitig)

- Funktionspointer `foo_cbFunction` zu Callback-Funktionen definieren

```
1 typedef void (*foo_cbFunction)(int);
2 // Schnittstelle: void f(int)
```

- Tabelle von Funktionspointern für jeden Event definieren und mit **Nullpointern initialisieren**

```
1 foo_cbFunction evClient[evNum] = {0};
2 // Note: NULL instead of 0 if stdio.h is included
```

- Aufruf der registrierten Clientfunktionen beim Eintreten des Events

```
1 void notify(foo_cbFunction client[], int evNum, int arg)
2 {
3     for(size_t i = 0; i < evNum; ++i)
4     {
5         if(evClient[i] != 0)    // entry found
6         {
7             evClient[i](arg);    // call client through function ptf
8         }
9     }
10 }
```

#### Mitteilung eines Events:

Sobald (**asynchron**) ein Event eingetreten ist, kann dieser dem Server mit der Funktion `void foo_signalEvent(foo_Event e);` mitgeteilt werden.

### Beispiel: Callback-Funktionen in C

#### Test-Applikation – Client:

```
1 // testApp.c
2 // this is the client
3
4 #include <stdio.h>
5 #include "fooServer.h"
6 #include "fooSigGen.h"
7
8 // functions to be registered (prototypes)
9 static void f1(int a);
10 // ...
11
12 int main(void)
13 {
14     enum {maxId = 8};
15     int fId[maxId] = {0};
16     // register functions on events
17     fId[0] = foo_registerCb(foo_ev1, f1);
18     fId[1] = foo_registerCb(foo_ev1, f2);
19     fId[2] = foo_registerCb(foo_ev1, f3);
20     fId[3] = foo_registerCb(foo_ev2, f4);
21     fId[4] = foo_registerCb(foo_ev2, f2); // same function registered on two events
22     fId[5] = foo_registerCb(foo_ev3, f5);
23
24     for (size_t i = 0; i < maxId; ++i)
25     {
26         if (foo_failed == fId[i])
27             printf("fId[%zu] failed to register\n", i);
28     }
29     foo_generateSignals();
30
31     // unregister some functions
32     if (foo_unregisterCb(foo_ev1, fId[0]) == fId[0])
33         printf("f1 successfully unregistered from foo_ev1\n");
34     else
35         printf("failed to unregister f1 from foo_ev1\n");
36
37     if (foo_unregisterCb(foo_ev1, 27) == 27) // should fail: unknown id
38         printf("xy successfully unregistered from qr\n");
39     else
40         printf("failed to unregister (unknown id)\n");
41
42     // register functions on events
43     printf("try to register f4 on foo_ev2 at fId[6]\n");
44     fId[6] = foo_registerCb(foo_ev2, f4); // should fail: too many registered functions
45     for (size_t i = 0; i < maxId; ++i)
46     {
47         if (foo_failed == fId[i])
48         {
49             printf("fId[%zu] failed to register\n", i);
50         }
51     }
52
53     foo_generateSignals();
54     return 0;
55 }
56
57 // local functions
58 void f1(int a)
59 {
60     printf("f1() called. Event# = %d.\n", a);
61 }
62
63 // ...
```

#### Server – Header-File:

```
1 // fooServer.h -> Callback server
2
3 #ifndef FOO_SERVER_H__
4 #define FOO_SERVER_H__
5
6 // typeless enum with integer for a failed result
7 enum {foo_failed = -1};
8
9 //function pointer to callback functions
10 typedef void (*foo_cbFunction)(int);
11
12 // enum with possible events
13 typedef enum {foo_ev1 = 1,        // foo example event 1
14              foo_ev2,            // foo example event 2
15              foo_ev3            // foo example event 3
16              }foo_Event;
17
18 // registers function 'f' on event 'e'
19 // returns id: success or foo_failed: no success
20 int foo_registerCb(foo_Event e, foo_cbFunction f);
21
22 // unregisters functionId 'id' on event 'e'
23 // returns id: success or foo_failed: no success
24 int foo_unregisterCb(foo_Event e, int id);
25
26 // signals that event 'e' occurred
27 void foo_signalEvent(foo_Event e);
28 #endif
```

#### Server – Implementation:

```
1 // fooServer.c --> Callback server
2
3 #include "fooServer.h"
4 #include <stddef.h>
5
6 enum {ev1Num = 3,        // max number of registered functions for event ev1
7       ev2Num = 2,        // dito ev2
8       ev3Num = 2};      // dito ev3
9
10 // definition of function tables
11 static foo_cbFunction ev1Client[ev1Num] = {0}; // clients for event ev1
12 static foo_cbFunction ev2Client[ev2Num] = {0}; // clients for event ev2
13 static foo_cbFunction ev3Client[ev3Num] = {0}; // clients for event ev3
14
15 // local function declarations
16 static int insertCb(foo_cbFunction f, foo_cbFunction client[], int evNum);
17 // inserts callback function 'f' in list 'client[]'
18
19 static int deleteCb(int id, foo_cbFunction client[], int evNum);
20 // deletes callback functionId 'id' in list 'client[]'
21
22 static void notify(foo_cbFunction client[], int evNum, int arg);
23 // notifies all registered clients
24
25 // interface functions' definitions
26 int foo_registerCb(foo_Event e, foo_cbFunction f)
27 {
28     switch (e)
29     {
30         case foo_ev1:
31             return insertCb(f, ev1Client, ev1Num);
32         case foo_ev2:
33             return insertCb(f, ev2Client, ev2Num);
34         case foo_ev3:
35             return insertCb(f, ev3Client, ev3Num);
36         default:
37             break;
38     }
39     return foo_failed; // no success if I get here
40 }
41
42 int foo_unregisterCb(foo_Event e, int id)
43 {
44     switch (e)
45     {
46         case foo_ev1:
47             return deleteCb(id, ev1Client, ev1Num);
48         case foo_ev2:
49             return deleteCb(id, ev2Client, ev2Num);
50         case foo_ev3:
51             return deleteCb(id, ev3Client, ev3Num);
52         default:
53             break;
54     }
55     return foo_failed; // no success if I get here
56 }
57
58 void foo_signalEvent(foo_Event e)
59 {
60     switch (e)
61     {
62         case foo_ev1:
63             // in this example, only the event # is passed as argument
64             notify(ev1Client, ev1Num, e);
65             break;
66         case foo_ev2:
67             notify(ev2Client, ev2Num, e);
68             break;
69         case foo_ev3:
70             notify(ev3Client, ev3Num, e);
71             break;
72         default:
73             break;
74     }
75 }
```

```

77 // local functions
78 int insertCb(foo_cbFunction f, foo_cbFunction client[], int evNum)
79 {
80     for (size_t i = 0; i < evNum; ++i)
81     {
82         if (0 == client[i]) // free entry found
83         {
84             client[i] = f;
85             return i; // success
86         }
87     }
88     return foo_failed; // number of registered functions exceeded
89 }
90
91 int deleteCb(int id, foo_cbFunction client[], int evNum)
92 {
93     if (id < evNum && id >= 0)
94     {
95         client[id] = 0;
96         return id; // success
97     }
98     else
99     {
100         return foo_failed; // illegal id
101     }
102 }
103
104 void notify(foo_cbFunction client[], int evNum, int arg)
105 {
106     for (size_t i = 0; i < evNum; ++i)
107     {
108         if (client[i] != 0) // entry found
109         {
110             client[i](arg); // call the registered client through function pointer
111         }
112     }
113 }

```

### Signal Generator – Header-File:

```

1 // fooSigGen.h -> signal generator
2 // generates (emulates) external signals
3
4 #ifndef FOO_SIGGEN_H__
5 #define FOO_SIGGEN_H__
6
7 void foo_generateSignals(void);
8
9 #endif

```

### Signal Generator – Implementation:

```

1 // fooSigGen.c
2 // Signal generator implementation
3
4 #include "fooServer.h"
5 #include <stdio.h>
6
7 // interface functions' definitions
8 void foo_generateSignals(void)
9 {
10     int answer;
11
12     do
13     {
14         printf("-----\n");
15         printf("\nChoose event to be signalled\n");
16         printf("    (1)    Event 1\n");
17         printf("    (2)    Event 2\n");
18         printf("    (3)    Event 3\n");
19         printf("\n    (0)    Exit\n");
20         printf("\nYour choice: ");
21         scanf("%d", &answer);
22         printf("\n-----\n");

```

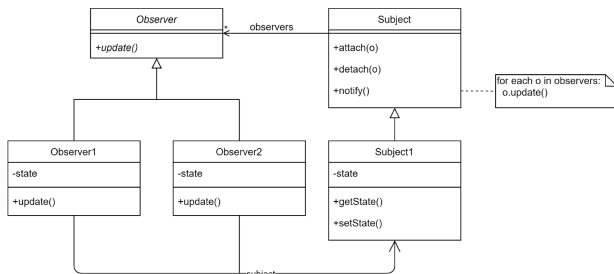
```

23         switch (answer)
24         {
25             case 1:
26                 foo_signalEvent(foo_ev1);
27                 break;
28             case 2:
29                 foo_signalEvent(foo_ev2);
30                 break;
31             case 3:
32                 foo_signalEvent(foo_ev3);
33                 break;
34             default:
35                 break;
36         } while (answer != 0);
37     }
38 }

```

## 9.10 Observer Pattern

### 9.10.1 Klassendiagramm (abstrakte Observer Basisklasse)



### 9.10.2 Implementation in C++

- **Observer-Klasse (abstrakte Basisklasse)**
  - Die Klasse muss **nicht** geändert werden
- **Observer-Subklassen (View)**
  - Enthalten jeweils eine private Referenz auf ein konkretes Subject
  - state entspricht z.B. einem counter-Wert, welcher jeweils updated wird
- **Subjekt Klasse (server, Model)**
  - liefert Administration für alle Subjects
  - Die Klasse muss **nicht** geändert werden
  - Enthält privates array mit Pointern auf Observer `const Observer* observers[size]`
  - `attach(o)` und `detach(o)` benutzen `const` Referenzen auf Observer als Parameter
- **Subjekt1 Subklasse**
  - Konkretes Subjekt (Server, Model)

## Beispiel: Observer Pattern in C++

### Test-Applikation:

```

1 // testApp.cpp
2 // client using observer pattern
3
4 #include <iostream>
5 #include "Subject1.h"
6 #include "Observer1.h"
7 using std::cout;
8 using std::endl;
9
10 int main(void)
11 {
12     Subject1 myS;
13     Observer1 myO(myS);
14     myS.setState(23);
15     myS.setState(87);
16
17     return 0;
18 }

```

### Observer – Abstrakte Basisklasse:

```

1 // Observer.h
2
3 #ifndef OBSERVER_H__
4 #define OBSERVER_H__
5
6 class Observer
7 {
8     public:
9         // method to update something
10        // (pure virtual)
11        virtual void update() const = 0;
12
13        // Dtor
14        virtual ~Observer() {}
15 };
16 #endif

```

### Observer1 – Konkreter Observer (View) – Header-file:

```

1 // Observer1.h -> implements an observer
2
3 #ifndef OBSERVER1_H__
4 #define OBSERVER1_H__
5
6 #include "Observer.h"
7
8 class Subject1; // forward declaration to subject
9
10 class Observer1 : public Observer
11 {
12     public:
13         Observer1(Subject1& s); // Ctor with reference to subject
14         void update() const override; // method to update something
15         virtual ~Observer1(); // Dtor
16
17     private:
18         Subject1& sub;
19 };
20 #endif

```

### Observer1 – Konkreter Observer (View) – Implementation:

```

1 // Observer1.cpp -> implements an observer
2
3 #include "Observer1.h"
4 #include <iostream>
5 #include "Subject1.h"
6 using namespace std;
7
8 Observer1::Observer1(Subject1& s) : sub(s)
9 {
10     sub.attach(*this);
11 }
12
13 void Observer1::update() const
14 {
15     cout << "Observer1 view: " << sub.getState() << endl;
16 }
17
18 Observer1::~~Observer1()
19 {
20     sub.detach(*this);
21 }

```

### Subject – Basisklasse (Server, Model) – Header-file:

```

1 // Subject.h -> Server, Model, Subject
2
3 #ifndef SUBJECT_H__
4 #define SUBJECT_H__
5
6 class Observer; // forward declaration
7
8 class Subject
9 {
10     public:
11         enum {ok = 0, // return value for good
12              failed = -1 // return value for error/failure
13             };
14
15         int attach(const Observer& ob); // attaches observer 'ob'
16         // return: ok: success or failed: no success
17
18         int detach(const Observer& ob); // detaches observer 'ob'
19         // return: ok: success or failed: no success
20
21         void notify() const; // notifies all attached observers (read-only)
22
23     private:
24         enum {size = 4}; // may use template parameter
25         const Observer* observers[size] = {nullptr}; // may use vector<> instead
26 };
27 #endif

```

Subject – Basisklasse (Server, Model) – Implementation:

```
1 // Subject.cpp -> Server, Model, Subject
2
3 #include <iostream>
4 #include "Subject.h"
5 #include "Observer.h"
6 using namespace std;
7
8 int Subject::attach(const Observer& ob)
9 {
10     for (size_t i = 0; i < size; ++i)
11     {
12         if (nullptr == observers[i]) // free entry found
13         {
14             observers[i] = &ob;
15             return ok; // success
16         }
17     }
18     return failed; // number of observers exceeded
19 }
20
21 int Subject::detach(const Observer& ob)
22 {
23     for (size_t i = 0; i < size; ++i)
24     {
25         if (&ob == observers[i])
26         {
27             observers[i] = nullptr;
28             return ok; // success
29         }
30     }
31     return failed; // illegal observer
32 }
33
34 void Subject::notify() const
35 {
36     for (size_t i = 0; i < size; ++i)
37     {
38         if (observers[i] != nullptr) // entry found
39             observers[i]->update();
40     }
41 }
```

Subject1 – Header-file:

```
1 // Subject1.h
2 // observed entity, Model, Subject
3
4 #ifndef SUBJECT1_H_
5 #define SUBJECT1_H_
6
7 #include "Subject.h"
8
9 class Subject1 : public Subject
10 {
11 public:
12     // sets the state of the subject
13     void setState(unsigned int newState);
14
15     // returns ok: success
16     // or failed: no success
17     unsigned int getState() const;
18 private:
19     unsigned int state = 0;
20 };
21 #endif
```

Subject1 – Implementation:

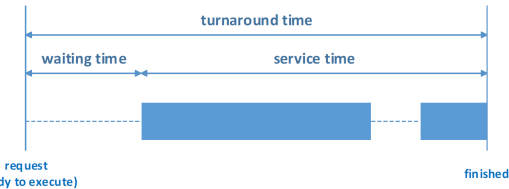
```
1 // Subject1.cpp
2 // The observed entity, Model, Subject
3
4 #include "Subject1.h"
5 #include <iostream>
6 using namespace std;
7
8 void Subject1::setState(unsigned int
9                           newState)
10 {
11     state = newState;
12     notify(); // inform observers
13 }
14
15 unsigned int Subject1::getState() const
16 {
17     return state;
18 }
```

10 Scheduling

10.1 Multitasking

Mehrere (gleiche oder unterschiedliche) Tasks müssen erledigt werden. Dazu werden Ressourcen benötigt (z.B. CPU, Speicher, ...). Wenn mehrere Tasks **dieselben Ressourcen** benötigen, nimmt der **Scheduler** die Zuteilung der Ressourcen an die einzelnen Tasks vor. Bei der Zuteilung der Ressourcen wird darauf geachtet, dass alle **kritischen deadlines eingehalten** werden.   
=> Der Scheduler **priorisiert** also die **kritischen Tasks**.  
Unter Umständen werden somit Deadlines von weniger kritischen Tasks verletzt.

10.1.1 Zeitdefinitionen (Task)



- **turnaround time:** (response time, Antwortzeit)
  - Startet, wenn der Task bereit zur Ausführung ist und endet, wenn der Task fertig abgearbeitet ist
  - Zeit zwischen dem Vorhandensein von Eingangswerten an das System (Stimulus) bis zum Erscheinen der gewünschten Ausgangswerte.
- **waiting time:** (Wartezeit)
  - Zeit zwischen Anliegen der Eingangswert und Beginn der Abarbeitung des Tasks
- **service time:** (Bearbeitungszeit)
  - Zeit für Abarbeitung des Tasks => Unterbrechungen bzw. (preemptions) möglich

10.1.2 Leistungsmerkmale

- Durchsatz (throughput)
  - Anzahl erledigte Tasks pro Zeiteinheit
- Mittlere Wartezeit (average waiting time)
- Auslastung (utilization)
  - Prozentuale Auslastung einer Ressource
- Weitere

10.2 Scheduability

Eine Menge von Tasks ist dann **scheduable**, wenn **alle Tasks zu allen Zeiten ihre deadlines einhalten** können. => Das ist immer das Ziel!

10.2.1 Deadline – Definition

- Spätestmöglicher Abschlusszeitpunkt (eines Tasks)
  - Bei periodischen Tasks ist dies meist gleichzeitig mit Beginn der nächsten Periode

10.3 Scheduling-Strategien

Folgende Algorithmen können für die Zuteilung der Ressourcen (Scheduling) verwendet werden:

- **FCFS (First Come First Served)**
  - Einfachste Variante
- **Round Robin**
  - Rund herum in fixer Reihenfolge
- **Random**
- **SJF (Shortest Job First)**
  - + Mittlere Wartezeit minimal
  - längere Tasks können 'verhungern'
- **Priority Scheduling**
  - unterbrechbar (preemptive) oder nicht unterbrechbar (non-preemptive)
  - tief priorisierte Tasks können 'verhungern'

**Hinweis:** 'verhungern' heisst, dass ein Task gar keine Ressourcen erhält

10.4 Cooperative Multitasking

Kooperative Task-Zuteilung ist bei **fairen** Tasks möglich.

- Aktiver Task entscheidet selbst, wann er CPU wieder für andere Tasks freigibt
  - Unfaire und abgestürzte Tasks blockiert andere Tasks
- Nächster Task kann mit beliebigem Algorithmus ermittelt werden
  - => siehe Abschnitt 10.3
- Sehr einfach zu implementieren

10.5 Preemptive Multitasking / Scheduling

Preemptive Multitasking wird meistens in RTOS verwendet. **Der Task mit höchster Priorität wird immer ausgeführt.** Unter Umständen muss dabei ein Task mit niedrigerer Priorität **unterbrochen** werden.

Es gibt zwei Arten von Preemptive Multitasking Algorithmen:

- **dynamic-priority Algorithmen**
  - Prioritäten werden zur Laufzeit laufen angepasst (z.B. aufgrund von vorhandenen deadlines)
- **static-priority Algorithmen**
  - Prioritäten werden zur Entwicklungszeit festgelegt und nicht geändert.
  - Einfacher als dynamic-priority Algorithmen!

10.6 Rate Monotonic Scheduling (RMS)

RMS beschreibt Regel, bei deren Einhaltung eine Konfiguration **immer scheduable** ist.

10.7 Rate Monotonic Scheduling Theorem

10.7.1 Zwingende Voraussetzungen

- Periodische Tasks
- static priority preemptive scheduling => siehe Abschnitt 10.5

10.7.2 Regeln für optimales Scheduling

Für jeden Task  $T_i$  wird die Periode  $p_i$  und die (worst case) execution time  $e_i$  ermittelt, bzw. geschätzt. Die Prioritäten müssen den Tasks zwingend folgendermassen zugewiesen werden:

**Tasks mit kürzerer Periode (d.h. mit hoher Rate) erhalten höhere Priorität (rate-monotonic)**

10.7.3 Berechnung der Auslastung einer Ressource

Jeder Task  $T_i$  trägt mit der Teilauslastung  $u_i = \frac{e_i}{p_i}$  zur Gesamtauslastung  $U$  bei. 
$$U = \sum_i \frac{e_i}{p_i}$$

10.7.4 Auslastung einer Ressource

Utilization (%)	Zone Type	Typical Application
0-25	significant excess processing power – CPU may be more powerful than necessary	various
26-50	very safe	various
51-68	safe	various
69	theoretical limit	embedded systems
70-82	questionable	embedded systems
83-99	dangerous	embedded systems
100+	overload	stressed systems

10.8 Vorgehen – Rate Monotonic Scheduling

1. Tasks priorisieren (Task mit kleinster Periode hat höchste Priorität!)
2. Task mit höchster Priorität aufzeichnen
3. Task mit zweithöchster Priorität 'regulär' zeichnen mit folgenden Sonderregelungen
  - Bei Bedarf warten (W), bis höher priorisierter Task abgeschlossen ist
  - Höher priorisierte Tasks (bereits gezeichnet) unterbrechen (P) aktuellen Task
4. Punkt 3 wiederholen, bis alle Tasks aufgezeichnet sind und sich das Muster wiederholt

Beispiel: Rate Monotonic Scheduling

Gemäss gegebener Tabelle sind die Tasks folgendermassen priorisiert:

T1 > T3 > T2

In dieser Reihenfolge werden die Tasks aufgezichnet!

↑	T <sub>2</sub>	W	W	W		P	P	P		P		P	P	P							
	T <sub>3</sub>	W															P				
	T <sub>1</sub>																				
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

10.9 RMA Bound (RMA = Rate Monotonic Approach)

Jede Konfiguration mit n periodischen Tasks ist immer RM scheduable, wenn die Gesamtauslastung U unterhalb oder gleich der RMA Bound U(n) liegt

RMA-Bound U(n) = n \* (2^(1/n) - 1)

n	2	3	4	5	10	∞
U(n) in %	82.4	78.0	75.7	74.4	71.7	ln(2) ≈ 69.3

⇒ Liegt die Gesamtauslastung unter 69.3 %, ist die Konfiguration immer RM schedulable

10.10 Anleitung für Zuweisung der Prioritäten bei RMS

- Prioritäten immer gemäss RMS zuweisen. (manuelle Zuweisung gibt keine bessere Lösung)
- Falls Auslastung nicht grösser als RMA Grenze, so ist Konfiguration RM schedulable
- Falls Auslastung grösser ist, muss manuell analysiert werden, ob Konfiguration schedulable ist
- 100 % Auslastung könnte erreicht werden, wenn alle Perioden harmonisch sind, d.h. jede längere Periode ist ein exaktes Vielfaches aller Perioden kürzerer Dauer, z.B. (10, 20, 40, 80)
- Harmonische Perioden verringern die Unterbrechung (preemptions) von niedriger priorisierten Tasks  
⇒ (10, 20, 40) ist gegenüber (10, 20, 50) zu bevorzugen, falls möglich

11 Concurrency (Gleichzeitigkeit)

Programme von praktischem Nutzen führen meist mehrere Arbeiten 'gleichzeitig' durch. Beispielsweise soll bei einem Embedded System ein Roboterarm bewegt werden, während 'gleichzeitig' mit einem übergeordneten System kommuniziert wird.

11.1 Parallel Computing vs. Concurrent Computing

- Parallel Computing
  - Ausführung verschiedener Tasks tatsächlich gleichzeitig
  - Nicht möglich auf single-core System
- Concurrent Computing
  - Ausführung verschiedener Tasks wirkt nur gleichzeitig
  - Verschiedene Tasks erhalten verschiedene 'time slices' ⇒ Ein Task pro time slice
  - Auf single- und multi-core Systemen möglich

11.2 Warum man Concurrency nicht verwenden sollte

- Concurrency (mit Prozessen, Tasks, Threads) kostet immer
    - Stack
    - Braucht context switch (Umschalten von einen zum anderen Prozess, Task, Thread)  
⇒ Alter context (Registerwerte, Steck, etc.) muss gespeichert, neuer geladen werden
    - Zugriff auf gemeinsame Ressourcen muss synchronisiert werden  
⇒ fehleranfällig (wird vergessen / falsch gemacht)
  - Komplexität steigt
    - Sequenzielle Programme sind einfacher zu verstehen als parallele Programme
- ⇒ Concurrency nur dann einsetzen, wenn wirklich ein Nutzen vorhanden ist!

11.3 Synchronisation

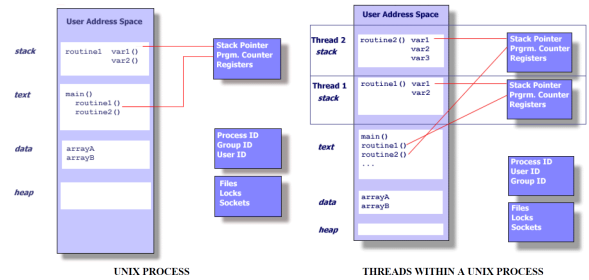
Wenn parallele Einheiten gemeinsame Ressourcen benötigen, muss der Zugriff auf die Ressourcen geregelt (synchronisiert) werden. Wenn dies nicht gemacht wird kann es sein, dass zwei Tasks dieselbe Ressource 'falsch' verwenden. ⇒ 'Deadlock'

Achtung: 'Ein Bisschen warten' ist keine Synchronisation!

12 POSIX Threads Programming

Für UNIX Systeme steht ein standardisiertes threads programming interface in C zur Verfügung (POSIX threads / pthreads).

12.1 UNIX Process vs. UNIX Thread



12.1.1 UNIX Process

- heavyweight process (generiert von Betriebssystem)
- Prozess erfordert erheblichen overhead, da Informationen über Programmressourcen und den Ausführungsstatus des Programms, beispielsweise:
  - Prozess-ID, Prozessgruppen-ID, Benutzer-ID und Gruppen-ID
  - Environment, Programmanweisungen
  - Register, Stack, Heap
  - Datei-Deskriptoren, Signal-Aktionen
  - Gemeinsame Bibliotheken
  - Werkzeuge für die prozessübergreifende Kommunikation

12.1.2 UNIX Thread

- lightweight 'process' (weniger overhead)
- Unabhängiger 'stream of instructions', welcher simultan mit anderen 'streams of instructions' ablaufen kann
- Prozedur, welche unabhängig von ihrem (aufrufenden) main-Programm abläuft
- Threadexistieren in einem Prozess und nutzen dessen Ressourcen
  - Sobald ein Prozess ended, enden auch die darin existierenden Threads!
- Ein Thread benutzt den gleichen Adressraum wie andere Threads im gleichen Prozess
  - Daten können einfach mit anderen Threads im gleichen Prozess geteilt werden
- Threads werden vom Betriebssystem 'gescheduled'
- Ein Thread dupliziert nur die essenziellen Ressourcen die er braucht, um unabhängig 'schedulable' zu sein:
  - Stack pointer, Register
  - Scheduling properties (policy / priority)
  - Set of pending and blocked signals
  - Thread-spezifische Daten

⇒ Gleichzeitigkeit wird in der Programmierung mit Threads umgesetzt!

12.2 pthreads API

12.2.1 Includes / Compile & Link

- #include <pthread.h> wird benötigt
- Methoden der pthreads API starten mit pthread\_
- Source files, welche pthreads verwenden, sollen mit -pthread kompiliert werden
- Für das file-linking muss der command -lpthread verwenet werden

Beispiel: Compiling / Linking file printer.c

```
Compiling: clang -c -Wall -pthread printer.c
Linking: clang -o printer printer.o -Wall -lpthread
```

12.2.2 Thread starten / beenden

- Jede Funktion mit der folgenden interface kann eine Thread-Methode werden
  - Als Parameter / Return-Wert sind alle Pointer-Datentypen möglich
- Ein Thread wird mit der folgenden Funktion gestartet:

```
void* threadRoutine(void* arg);
```
- Thread creation code:

```
1 int pthread_create(pthread_t* thread, // ptr to pthread_t instance
2     const pthread_attr_t* attr, // ptr to pthread_attr_t
3     // structure, often 0
4     // (default attributes)
5     void* (*startRoutine) (void*), // function ptr to thread routine
6     void* arg); // single argument that may be
7 // passed to startRoutine
8 // returns 0 if thread is started successfully
```

- Ein Thread kann mit einer der folgenden drei Arten beendet werden
  - Thread ruft Funktion pthread\_exit() auf
  - Thread springt aus Thread Routine startRoutine zurück
  - Thread wird mit Funktion pthread\_cancel() abgebrochen

12.2.3 Warten, bis ein Thread beendet ist

- Nach dem Starten des Threads bzw. am Ende des main-Programms kann eine Endlos-Schleife eingefügt werden
  - Dies sollte nie gemacht werden, da der Prozess so die gesamten CPU-Ressourcen braucht
- Entsprechende Funktion aus pthreads API verwenden

```
1 int pthread_join(pthread_t thread, // pthread_t instance
2     void** status) // ptr to status argum. passed at end of thread
3 // returns 0 if thread terminated successfully
```



## 12.3 Beispiel: thread API

```
1 #include <pthread.h> // for threads API
2 #include <stdio.h>
3 #include <unistd.h> // for usleep()
4
5 // function prototype
6 void* printDashes(void* arg);
7
8 int main(void)
9 {
10     int ret;
11     pthread_t dasher; // pthread_t instance
12
13     printf("start");
14
15     // starts thread -> immediately returns
16     // (thread maybe not fully started yet)
17     ret = pthread_create(&dasher, 0,
18                         printDashes, 0);
19     if (ret)
20     {
21         printf("ERROR CODE: %d\n", ret);
22         return -1;
23     }
24
25     // main thread shall wait until
26     // dasher is finished
27     ret = pthread_join(dasher, 0);
28     if (ret)
29     {
30         printf("ERROR CODE: %d\n", ret);
31         return -1;
32     }
33
34     printf("end\n");
35     return 0;
36 }
37
38 void* printDashes(void* arg)
39 {
40     for (size_t i = 0; i < 20; ++i)
41     {
42         usleep(40000);
43         putchar('-');
44         fflush(stdout); // write character-
45                        // wise and
46                        // don't buffer
47     }
48     return 0;
49 }
```

## 12.4 Thread-safeness

Thread-safeness bezieht sich auf die Fähigkeit einer Anwendung, mehrere Threads gleichzeitig auszuführen, **ohne 'clubbering' und 'race conditions'** zu verursachen. Damit Thread-safeness gewährleistet werden kann, ist **Synchronisation** erforderlich.

**clubbering:** Speicher durcheinander bringen, wenn mehrere Threads den gleichen Speicher benötigen und 'falsch' darauf zugreifen

**race conditions:** Programmablauf und Endergebnis hängen davon ab, in welcher Reihenfolge 'gleichzeitig' ablaufende Threads auf z.B. eine globale Variable im Speicher zugreifen und das Verhalten somit unvorhersehbar wird

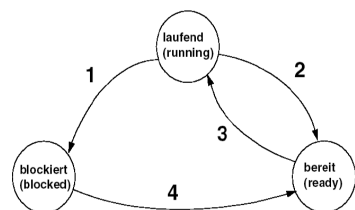
### 12.4.1 Empfehlung: Thread-Safeness

Wenn Thread-safeness nicht explizit garantiert ist (z.B. von einer Library, welche verwendet wird), muss angenommen werden, dass sie **nicht thread-safe** ist!

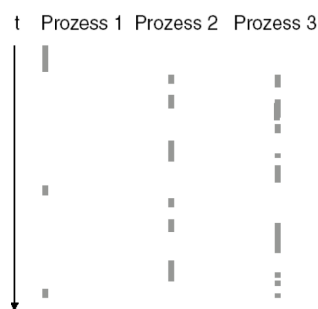
Um in einem solchen Fall Thread-safeness zu gewährleisten, können die Aufrufe einer 'unsicheren' Funktion **serialisiert** werden.

## 12.5 Quasi-Parallelität / 'Prozess'-Zustände

### 12.5.1 Prozess-Zustände



1. I/O Operation, Warten auf Bedingung
2. Scheduler entzieht CPU
3. Scheduler weist CPU zu
4. I/O beendet, Bedingung erfüllt



- Prozesse / Threads warten die 'meiste Zeit' ⇒ blocked (z.B. join blockiert andere Threads)
- Scheduler ordnet CPU denjenigen Prozess / Thread zu, die im Zustand 'ready' sind und 'etwas zu tun haben'
- Die Zuordnung hängt vom verwendeten Scheduling-Algorithmus ab:
  - First come First serve Scheduling: Eine Queue mit allen Prozessen, wobei nächster Prozess jeweils hinten angehängt wird und erster Eintrag der Queue aktuell ausgeführt wird
  - Priority Scheduling: Pro Priorität gibt es eine Queue. Abarbeitung je nach Algorithmus anders

## 12.6 Synchronisation

Synchronisation wird benötigt, um den **Zugriff auf gemeinsame Ressourcen** in Critical Sections (CS) zu 'kontrollieren'.

### 12.6.1 Definition: Critical Section (CS)

- Codebereich, in dem nebenläufige oder parallele Prozesse auf gemeinsame Ressourcen zugreifen
  - Zu jeder Zeit darf sich **höchstens ein Prozess** im kritischen Abschnitt befinden
- Der Exklusive Zugriff durch höchstens einen Prozess wird mittels **gegenseitigem Ausschluss (Mutex)** sichergestellt ⇒ Siehe Abschnitt 12.7

### 12.6.2 Forderungen an die Synchronisation

1. Maximal ein Prozess in einem kritischen Abschnitt (CS)
2. Über Abarbeitungsgeschwindigkeit, bzw. Anzahl Prozesse dürfen keine Annahmen getroffen werden
3. Kein Prozess darf **ausserhalb** eines kritischen Abschnitts einen anderen blockieren
4. Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss irgendwann den Abschnitt betreten dürfen (**fairness condition**) ⇒ Verhinderung von 'starvation'

## 12.7 Mutex (mutual exclusion)

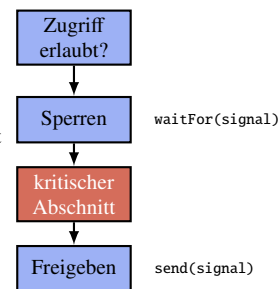
Die Lösungsstruktur 'Mutex' (gegenseitiger Ausschluss) stellt sicher, dass höchstens ein Prozess auf eine Critical Section (CS) zugreift.

### 12.7.1 Mutex – Ablauf

**Zugriffsprüfung:** Warten bis der Zugang frei wird

**Sperren:** Signal wird für andere auf Rot gesetzt, damit nur ein Prozess im kritischen Abschnitt sein kann

**Freigeben:** Rotes Signal wird wieder gelöscht



### 12.7.2 Verwendung von Signalen und Semaphoren

- Jeder Prozess wartet vor dem Betreten der CS auf ein gemeinsames Signal
  - Wenn das Signal gesetzt ist, ist CS frei
  - Mehrere Prozesse können gleichzeitig warten ⇒ Schedulingalgorithmus bestimmt 'nächsten' Thread
- waitFor(signal) blockiert **aufzufindenden** Prozess, falls Signal nicht gesetzt
- Jeder Prozess, der fertig ist, setzt das Signal mit send(signal)

**Semaphoren:**

- 'Semaphor' ist ein spezieller Name für ein Signal für den **Zutritt zu einer CS**
- Es gibt zwei atomare (nicht unterbrechbare) Operationen auf einer Semaphoren s
  - Passieren P(s): Beim Eintritt in CS ⇒ wait(s)
  - Verlassen V(s): Beim Austritt aus CS ⇒ send(s)

Bei der Verwendung von Semaphoren treten folgende Probleme auf

- Ressourcen können besetzt bleiben, wenn V(s) vergessen wird
  - **Für jedes P(s) braucht es auch ein V(s)**
- Größere Programme: Es können subtile Probleme entstehen, falls z.B. das V(s) in einer **if-Bedingung** gemacht wird
- Beim Auftreten von Exceptions kann das Freigeben schwierig werden

⇒ Lösung für das Freigabe-Problem: RAII (siehe Abschnitt 13.1)

### 12.7.3 Busy Waiting

- Prozesse warten **aktiv** in einer Schleife (**spin lock**)
  - Wartende Prozesse **belasten** unnötigerweise den Prozessor

Die Lösung für Busy Waiting ist, die wartenden Prozesse in eine **Warteschlange** einzutragen (**sleep and wakeup**)

## 12.8 Thread Synchronisierung in C mit pthreads API

Code Synchronisation wird mittels Mutex (**lock pattern**) sichergestellt. Das Konzept von Mutex ist, dass eine Mutex Variable **nur einem Thread gleichzeitig gehören kann**.

### 12.8.1 Ablauf einer Mutex-Sequenz in C

1. Mutex Variable erstellen / instanzieren
  - 'Schloss', welches Zugang zu CS schützt
2. Mehrere Threads versuchen, die Mutex Variable zu blockieren
  - ⇒ **Nur ein Thread** ist erfolgreich ⇒ diesem Thread ('owner') gehört die Mutex Variable
3. Dieser 'owner thread' führt Aktionen in der Critical Section (CS) aus
  - Häufig Update einer globalen (shared) Variable
4. 'owner' entblockt (unlock) die Mutex Variable
5. Dem nächsten Thread gehört die Mutex Variable ⇒ zurück zu Schritt 2
6. Wenn alle Threads abgearbeitet sind, wird die Mutex Variable zerstört

⇒ Dies ist ein sicherer Weg, um sicherzustellen, dass, wenn **mehrere Threads** dieselbe Variable aktualisieren, der **Endwert derselbe** ist, wie wenn nur **ein Thread** die **Aktualisierung durchführen** würde.

### Beispiel: Mutex in C

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h> /* for usleep */
5
6 static volatile int val = 0; // shared resource
7 static pthread_mutex_t valMtx; // create mutex_t variable
8
9 void* threadRoutine(void* arg); // prototype
10
11 int main(void)
12 {
13     pthread_t t1; // create pthread_t variable
14     pthread_t t2; // create pthread_t variable
15
16     pthread_mutex_init(&valMtx, 0); // init mutex
17
18     pthread_create(&t1, 0, threadRoutine, 0);
19     pthread_create(&t2, 0, threadRoutine, 0);
20
21     pthread_join(t1, 0); // wait for thread to finish
22     pthread_join(t2, 0); // wait for thread to finish
23
24     pthread_mutex_destroy(&valMtx); // destroy mutex
25
26     return 0;
27 }
28
```

```
29 // main routine of each counter thread
30 void* threadRoutine(void* arg)
31 {
32     unsigned int rState = 17;
33
34     while(1)
35     {
36         /* non critical section; simulate with usleep() */
37         usleep(rand_r(&rState) % 2000000);
38
39         /* start of critical section */
40         pthread_mutex_lock(&valMtx); // lock mutex
41
42         if (val < 20)
43         {
44             /* wait random time between 0s up to 0.3s */
45             usleep(rand_r(&rState) % 300000);
46             val = val + 1; // change shared resource
47             printf("val = %2d\n", val);
48         }
49         else
50         {
51             /* end of critical section */
52             pthread_mutex_unlock(&valMtx); // unlock mutex
53             break; // exit while(1)
54         }
55         /* end of critical section */
56         pthread_mutex_unlock(&valMtx); // unlock mutex
57     }
58     pthread_exit(0); // optional, good programming style!
59 }
```

12.9 Monitorprinzip (Monitor Pattern)

Das Monitorprinzip beschreibt eine Art Abstraktion des Mutex / Lock Patterns. Dabei muss sich der **Aufrufer nicht mehr um die Synchronisation der Threads kümmern**. Das Problem wird einmal im Monitor gelöst.

- Es wird ein Abstrakter Datentyp (ADT) definiert, der genau die Funktionen in der Schnittstelle anbietet, die notwendig sind
- Der Aufrufer ruft diese Funktion auf, muss sich aber **nicht um Synchronisation kümmern**
  - Synchronisation (z.B. mit Semaphoren) ist Implementation des Monitors lokal gelöst

12.10 'Stolperfallen' bei Synchronisation

12.10.1 Starvation (Verhungern)

- Zustand, bei dem ein Prozess nie dran kommt => er verhungert
- Kann auftreten bei:
  - prioritätsgetriebenen Systemen bei Prozessen mit niederer Priorität passieren
  - SJF (shortest job first) Systeme => kurze Jobs bremsen längere Jobs aus
- Fairness condition besagt, dass Starvation verhindert werden muss

12.10.2 Deadlock

- Situation, bei der sich **zwei Prozesse gegenseitig blockieren**
  - Zwei Prozesse benötigen gemeinsame Ressourcen A und B. Wenn Prozess 1 die Ressource A bereits besitzt und Prozess 2 die Ressource B, dann warten beide unendlich lange auf die jeweils andere Ressource

**Deadlock kann vermieden werden, indem alle Prozesse die gemeinsamen Ressourcen immer in derselben Reihenfolge anfordern (z.B. zuerst A, dann B)**

12.11 Informationen zwischen Threads austauschen

Der Austauschen von Daten zwischen verschiedenen Threads (z.B. anderen Thread benachrichtigen, wenn in eigenem Thread etwas passiert ist / warten, bis in anderem Thread etwas passiert ist ) ist mittels **shared resources** möglich.

Der **Nachteil** davon ist aber, dass diese shared resource mit **polling** abgefragt werden muss => nicht effizient!

Der korrekte Weg für den Informations-Austausch zwischen Threads sind **condition variables**.

12.11.1 Condition Variables

- Mit Hilfe von condition variables können Threads auf der Grundlage des aktuellen Datenwerts synchronisiert werden
  - **Kein polling nötig!**
- Condition Variables werden immer **zusammen mit einem 'Mutex lock'** verwendet

Beispiel: Anwendungsbeispiel für Condition Variables

Main Thread		
<ul style="list-style-type: none"><li>• Declare and initialize global data/variables which require synchronization</li><li>• Declare and initialize a condition variable object</li><li>• Declare and initialize an associated mutex</li><li>• Create threads A and B to do work</li></ul>		
Thread A		Thread B
<ul style="list-style-type: none"><li>• Do work up to the point where a certain condition must occur (such as count must reach a specified value)</li><li>• Lock associated mutex and check value of a global variable</li><li>• Call pthread_cond_wait() to perform a blocking wait for signal from Thread B</li></ul> <p><i>Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread B</i></p> <ul style="list-style-type: none"><li>• When signalled, wake up</li></ul> <p>Mutex is automatically and atomically locked</p> <ul style="list-style-type: none"><li>• Explicitly unlock mutex</li><li>• Continue</li></ul>		<ul style="list-style-type: none"><li>• Do work</li><li>• Lock associated mutex</li><li>• Change the value of the global variable that Thread A is waiting upon</li><li>• Check value of the global Thread A wait variable</li></ul> <ul style="list-style-type: none"><li>• If it fulfills the desired condition, signal Thread A</li><li>• Unlock mutex</li><li>• Continue</li></ul>
Main Thread		
<ul style="list-style-type: none"><li>• Join / Continue</li></ul>		

12.12 Condition Variables mit pthreads

12.12.1 Erstellen / initialisieren von Conditon Variables

```
1 int pthread_cond_init(pthread_cond_t* condVar, // ptr to condition variable
2                     const pthread_condattr_t* attr) // ptr to pthread_condattr_t
3 // structure, often 0
4 // (default attributes)
5 // returns 0 if condition variable is initiated successfully
```

12.12.2 Zerstören von Condition Variables

```
1 int pthread_cond_destroy(pthread_cond_t* condVar) // ptr to condition variable
2 // returns 0 if condition variable is destroyed successfully
```

12.12.3 Auf Conditon Variables warten

- Blockiert aufrufenden Thread bis Bedingung signalisiert wird
- Muss aufgerufen werden, wenn Mutex bockiert ist
  - Gibt Mutex automatisch frei, während gewartet wird
- **Programmierer ist verantwortlich für Freigabe der Mutex Variable, wenn Thread fertig**

```
1 int pthread_cond_wait(pthread_cond_t* condVar, // ptr to condition variable
2                     pthread_mutex_t* mutex) // ptr to pthread_mutex_t instance
3 // returns 0 if waiting (blocking) is successful
```

12.12.4 Signalisierung mit Conditon Variables

- Gibt Thread frei, welcher von Condition Variable blockiert wird
- Signalisierung für anderen Thread, welcher auf Condition Variable wartet
- Sollte aufgerufen werden, nachdem Mutex blockiert ist

```
1 int pthread_mutex_signal(pthread_cond_t* condVar) // ptr to condition variable
2 // returns 0 if signaling (unblocking) is successful
```

Beispiel: Condition Variables

Der zählende Thread, welcher den spezifizierten Wert der Variable count erreicht, signalisiert dem überwachenden Thread, dass die condition eingetreten ist.

```
1 #include <pthread.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 enum {maxCount = 10, numThreads = 3, countLimit = 12};
7
8 volatile static int count = 0; // shared resource
9 static pthread_mutex_t countMutex; // the mutex
10 static pthread_cond_t countThresholdCv; // the condition variable
11
12 void* incCount(void* t); // count thread function
13 void* watchCount(void* t); // watch thread function
14
15 int main(void)
16 {
17     int t[numThreads] = {1, 2, 3};
18     pthread_t threads[numThreads];
19
20     pthread_mutex_init(&countMutex, 0); // init mutex
21     pthread_cond_init (&countThresholdCv, 0); // init condition variable
22
23     pthread_create(&threads[0], 0, watchCount, (void*)&t[0]);
24     pthread_create(&threads[1], 0, incCount, (void*)&t[1]);
25     pthread_create(&threads[2], 0, incCount, (void*)&t[2]);
26
27     for (int i = 0; i < numThreads; ++i)
28         pthread_join(threads[i], 0);
29
30     pthread_mutex_destroy(&countMutex);
31     pthread_cond_destroy(&countThresholdCv);
32     pthread_exit(0);
33 }
34
35 void* incCount(void* t)
36 {
37     int myId = *(int*)t;
38     printf("Starting incCount(): thread %ld\n", myId);
39     for (int i = 0; i < maxCount; ++i)
40     {
41         pthread_mutex_lock(&countMutex); // start of critical section
42         ++count;
43         // Check the value of count and signal waiting thread when condition is
44         // reached. Note that this occurs while mutex is locked.
45         if (count == countLimit)
46             pthread_cond_signal(&countThresholdCv); // signal waiting thread
47
48         pthread_mutex_unlock(&countMutex); // end of critical section
49
50         sleep(1); // Do some work so threads can alternate on mutex lock
51     }
52     pthread_exit(0);
53 }
54
55 void* watchCount(void* t)
56 {
57     int myId = *(int*)t;
58     printf("Starting watchCount(): thread %ld\n", myId);
59
60     // Note that pthread_cond_wait() will automatically and atomically unlock
61     // mutex while it waits. Also, note that if countLimit is reached before this
62     // routine is run by the waiting thread, the loop will be skipped to prevent
63     // pthread_cond_wait() from never returning.
64
65     pthread_mutex_lock(&countMutex); // start of critical section
66     while (count < countLimit)
67     {
68         pthread_cond_wait(&countThresholdCv, &countMutex); // wait on condition
69         printf("watchCount(): condition signal received.\n");
70         count += 125; // updating the value of count
71     }
72     pthread_mutex_unlock(&countMutex); // end of critical section
73     pthread_exit(0);
74 }
```

## 12.13 Bounded Buffer Problem / Producer-consumer problem

Das Bounded Buffer Problem ist ein klassisches **multi-process Synchronisations-Problem**. Die Lösung für das Problem ist eine Anwendung von **Mutex-lock**.

Das Problem beschreibt folgende Situation:

- (mindestens) zwei Prozesse (producer, consumer) teilen sich einen gemeinsamen Buffer fixer Grösse (als queue benutzt)
  - producer: Daten generieren und in buffer schreiben
  - consumer: Daten aus Buffer lesen
- **Problem:** Es muss sichergestellt werden, dass
  - producer nur in Buffer schreibt, wenn dieser nicht voll ist
  - consumer nur Daten aus Buffer liest, wenn dieser nicht leer ist

### 12.13.1 Lösung für Bounded Buffer Problem: Mutex

- **Producer:** 'go to sleep' (oder Daten werfen), wenn Buffer voll
  - Erhält notification, wenn consumer Daten aus Buffer entfernt
- **Consumer:** 'go to sleep', wenn Buffer leer
  - Erhält notification, wenn producer Daten in Buffer speichert

Die **Umsetzung** erfolgt mittels 'inter-thread communication' ⇒ **Mutex** (Semaphoren)  
⇒ Schlechte Umsetzung resultiert in **deadlock** (beide Threads warten auf wake-up)

## 12.14 POSIX Interprocess Communication (IPC)

POSIX bietet folgende IPC Mechanismen in der `POSIX:XSI` extension. Diese Mechanismen erlauben unabhängigen Prozessen, Informationen effizient auszutauschen.

- Message Queues in `sys/msg.h`
- Shared Memory in `sys/shm.h`
- Semaphores in `sys/sem.h`

## 13 Resource Acquisition Is Initialisation (RAII)

RAII fordert sauber Speicher an und gibt diesen auch zuverlässig wieder frei.  
⇒ **Geht nur in C++**, nicht in Java ⇒ Java hat einen garbage collector, der 'aufräumt', wann er will und nicht zwingend am Ende des scopes

### 13.1 Grundkonzept von RAII

- Anforderung / Freigabe einer Ressource wird mit Hilfe einer **Klasse** implementiert
  - Konstruktor: Fordert Ressource an
  - Destruktor: Gibt Ressource wieder frei
- Ressource kann wie ein **Objekt** behandelt werden
  - Sobald Objekt seine Gültigkeit verliert (z.B. out-of-scope), gibt Destruktor die Ressource 'automatisch' frei

### 13.2 RAII bei Heapobjekten

**Problem:**

```
1 void f()
2 {
3     Person* p = new Person("irgendwer");
4     // mach etwas mit p
5     // Problem: hier tritt Exception auf
6     delete p;
7 }
```

**Lösung:**

```
1 #include <memory>
2 void f()
3 {
4     std::shared_ptr<Person>
5     p(new Person("irgendwer"))
6     // mach etwas mit p
7     // Beim Verlassen des Blocks räumt Dtor von
8     // shared_ptr automatisch auf und löscht
9     // Person
10 }
```

### 13.3 RAII bei Mutex

**Problem:**

Es muss sichergestellt werden, dass Mutex in jedem Fall wieder freigegeben wird

- Exceptions können dazwischenkommen
- Freigabe muss auch bei vorzeitigem Verlassen mit **return** erfolgen

```
1 static pthread_mutex_t m;
2 //...
3 void f()
4 {
5     pthread_mutex_lock(&m);
6     // do sth in crit. section
7     // problem: exception occurs
8     pthread_mutex_unlock(&m);
9 }
```

**Lösung:**

```
1 // file: ResourceLock.h
2 class ResourceLock
3 {
4 public:
5     ResourceLock(pthread_mutex_t& mx) : mutex(mx) {pthread_mutex_lock(&mutex); }
6     ~ResourceLock() {pthread_mutex_unlock(&mutex); }
7 private:
8     pthread_mutex_t& mutex; // reference to mutex of shared resource
9 };
```

```
1 // file: main.cpp (or other .cpp file)
2 void f()
3 {
4     { // place critical section inside a block
5         ResourceLock lock(myMutex); // calls Ctor
6         // do sth in critical section -> no problem if exception occurs
7     } // Dtor called when block is left (out of scope) -> mutex always unlocked
8     //...
9 }
```

**Hinweis:** Die Klasse ResourceLock ist '0 Byte gross, da die Funktionen 'inline' implementiert sind und eine Referenz verwendet wurde

## 14 Interfacig in C

### 14.1 Plain Old Data Types (POD Types)

POD Types sind Datentypen, welche bereits in C vorhanden sind. Sie funktionieren in C++ identisch wie in C.

- **char, short, int, long**  
⇒ jeweils **signed** und **unsigned**
- **float, double**

## 14.2 Language Linkage

- **Linker benötigt einen eindeutigen Namen für jede Funktion**
- In Assembler muss ein eindeutiger Label entstehen, der mit BSR (Branch to subroutine) angesprungen wird
- **In C: einfach**, da Funktionsname projektweit eindeutig sein muss
- **In C++: komplizierter**, da Funktionen in einem Namespace liegen, zu einer Klasse gehören und auch noch überladen werden können

### 14.2.1 C Language Linkage

- C Linker hängt für Interne Darstellung häufig ein **Underscore vorne an Funktionsnamen**, um ein Label in Assembler zu erhalten
  - `foo()` wird zu `_foo`

### 14.2.2 C++ Language Linkage

- C++ Linker hängt für Interne Darstellung **Name Mangling**  
⇒ `<namespace>_<functionName>_<parameterTypesShortened>`
  - `foo(int)` wird zu `_foo_i`
  - `myClass::foo(double, int)` wird zu `_MyClass_foo_d_i`

## 14.3 Festlegen der Language Linkage

### 14.3.1 Motivation

Aus C++ wird eine Funktion `foo(int)` aufgerufen, die in einer **kompilierten C-Bibliothek** vorliegt.

- Der verwendete C++-Linker sucht nach einer Name Mangling Darstellung der Funktion: `_foo_i`
- Da es sich aber um eine C-Bibliothek handelt, heisst die Funktion aber `foo`

⇒ **Dem C++-Linker muss mitgeteilt werden, dass diese Funktion C Linkage hat!**

### 14.4 Festlegen der Language Linkage – Umsetungen C++ seitig

Im **Funktionsprototypen** (C++-seitig) kann die Language Linkage festgelegt werden. Somit kann **C-Code aus C++ aufgerufen** werden.

```
1 extern "C" void foo(int); // use C language linkage -> preferred variant
2 void hoo(int); // use C++ language linkage -> preferred variant
3 extern void goo(int); // use C++ language linkage -> other variant
4 extern "C++" void koo(int); // use C++ language linkage -> other variant
5
6 extern "C"
7 {
8     // list multiple prototypes with C linkage or
9     // #include C header file(s) -> preferred variant
10 }
```

Optimalerweise wird die Language Linkage im **Header-File** vorgenommen.  
**Auf folgende Weise versehene Header-Files können sowohl in C- als auch in C++-Dateien included werden. Beide Compiler können diese Header übersetzen.**

```
1 #ifndef __cplusplus
2 extern "C"
3 {
4 #endif
5 // list multiple prototypes with C linkage or
6 // #include C header file(s)
7 #ifndef __cplusplus
8 }
9 #endif
```

⇒ `__cplusplus` ist dann definiert, wenn mit einem C++ Compiler kompiliert wird!

### 14.5 C++ Code aus C aufrufen

C++ Elementfunktionen können **nicht direkt** aus C aufgerufen werden, da C weder Klassen, noch den **this** Pointer kennt. Es werden **C Wrapper-Funktionen** benötigt, um C++ Code aufrufen zu können.

### Beispiel: Dining Philosopher

**Ausgangssituation:**

- 3 Personen an einem Tisch; 3 Stäbchen zum essen vorhanden
- Jede Person braucht 2 Stäbchen zum
  - Stäbchen sollen mittels Threading weitergereicht werden
- Ein **Deadlock** entsteht, wenn alle je ein Stäbchen haben und auf den nächsten warten

**Philosopher Class:**

```
1 class Philosopher
2 {
3 public:
4     // ...
5     void live(); // the philosopher's life
6
7 private:
8     pthread_attr_t attr;
9     pthread_t tid; // thread id
10    // ...
11    void lifeThread(); // the (C++) - tread function
12    static void* staticWrapper(void* p) // C Wrapper for thread_create()
13    { // p must be the this pointer
14        static_cast<Philosopher*>(p)->lifeThread();
15        return 0;
16    }
17 };
```

Die C Wrapper-Funktion `staticWrapper` muss **zwingend static** sein! Die Funktion muss auch ohne Zusammenhang mit einer Objekt-Instanz aufrufbar sein.  
Damit in C der **this** Pointer zur Verfügung steht, muss dieser der Wrapper-Funktion als Parameter übergeben werden.



Philosopher::live() Funktion:

```
1 void Philosopher::live()
2 {
3     pthread_attr_t_init(&attr);
4     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
5     pthread_create(&tid, &attr, staticWrapper, this);
6 }
```

Philosopher wird erstellt ('spawned'), indem ein Thread erstellt wird. Als Argument der Thread Funktion staticWrapper wird der this Pointer übergeben.

15 Programming Style Guide

Programme werden für Programmierer geschrieben, nicht für Compiler. Daher erleichtern Programmierkonventionen einen konsistenten Stil, der auch von anderen (Programmierern) verstanden wird.

In der Praxis wird jedoch häufig Wert auf Unwichtiges (z.B. Namensgebung von Variablen) gelegt. Wichtiges wird jedoch oft übersehen.

15.1 Grundsätzliche Konventionen

- Welche Regeln auch definiert werden: **Regeln konsistent und konsequent** anwenden!
- Mit höchstem Warning level kompilieren (-Wall)

15.1.1 'Small stuff'

- Eine Anweisung pro Zeile, nur eine Variablendefinition pro Zeile
  - (nicht notwendig aber sehr sinnvoll)
- Anweisungen in Blöcken einrücken (Empfehlung: 2 Leerzeichen)
  - Wo geschweifte Klammern stehen ist nicht wichtig

15.2 Namenskonventionen

- Wenig underscores \_ verwenden
- Keine Namen mit underscores \_ beginnen
- Zusammengesetzte Namen mit camelCase betonen
- Selbstdefinierte Typen / Klassen mit Grossbuchstaben beginnen
- Funktionen, Methoden und Namespaces, Variablen und Objekte mit Kleinbuchstaben beginnen
- Makros (#define) ausschliesslich mit Grossbuchstaben definieren, Trennung mit underscores \_

15.2.1 Namenskonventionen für Variablen

- Präfixe sehr zurückhaltend einsetzen (lieber vermeiden)
- Schleifenvariablen mit Namen i, j vom Typ size\_t
- c, ch für char Variablen
- s, str für string bzw. char\* Variablen

15.3 Typen mit genauer Breite

- Ab C99 werden im Headerfile <stdint.h> verschiedene Typen mit genauer Breite definiert.
  - signed: int8\_t, int16\_t, int32\_t
  - unsigned: uint8\_t, uint16\_t, uint32\_t

16 Multicore Systeme

16.1 Geschwindigkeit auf Prozessor steigern

16.1.1 Clockfrequenz erhöhen

- PCB-Design wird sehr anspruchsvoll (Leistungslängen, Reflexionen, etc.)
- Elektrische Verlustleistung steigt linear mit Clockfrequenz:  $P = f_{cl} \cdot C_L \cdot V_{dd}^2$
- Lichtgeschwindigkeit ist Grenze (Licht legt in 1 ns 30 cm zurück)

=> Erhöhung der Clockfrequenz hat eigentlich nur Nachteile

16.1.2 Instruction-level parallelism (ILP)

- Parallelismus wird auf Instruktionslevel angestrebt
  - Pipelines
  - Verschachteln der Instruktionen, um Pipeline möglichst optimal auszulasten
  - Vermeiden von Pipeline flush durch Vorhersage der Verzweigung (branch prediction)
- Branch prediction kann sehr aufwendig werden
- Compiler werden komplex

16.1.3 Thread-level parallelism (TLP)

- Parallelisierte Einheit ist in der Größenordnung einer Funktion
  - Umfangreichere Stufe der Parallelisierung
- Zugriffe auf gemeinsame Ressourcen (z.B. shared memory) müssen geregelt sein
- Design Issues
  - Jeder Thread erzeugt Overhead (Stack, context switch bei single-core Prozessor)
  - Nicht beliebig viele Threads möglich
  - Bei zu vielen Threads (wenn zusammengehöriges auseinandergenommen wird), dann werden gewisse Daten unnötigerweise shared

Umsetzung von TLP auf Uniprozessor (single-core):

- Threads werden time-sliced
- Pseudo-TLP
- Context switch notwendig (Umschalten von einem zum anderen Thread)
- Ergibt keinen Geschwindigkeitsgewinn

Umsetzung von TLP auf Multiprozessor:

- Mehrere parallele Prozesse können je einen Thread bearbeiten
- Echter-TLP
- Clockfrequenz kann tiefer gehalten werden

- Einfache Hardware wird multipliziert
- Datenaustausch zwischen Prozessen muss geregelt werden, z.B. mit Message Passing / Shared Memory

16.2 Speicherorganisation auf Multicore Prozessor

Shared Memory:

- Alle Prozessoren nutzen einen gemeinsamen Speicher
    - Kann zum Falschenhals werden
- Distributed Memory:
- Jeder Prozessor hat eigenen lokalen Speicher
    - Braucht Mechanismus für Datenaustausch, z.B. Message Parsing

16.2.1 Multicore Prozessor

- Spezieller Multiprozessor => alle Cores (Prozessoren) auf demselben Chip
- MIMD (Multiple Instructions Multiple Data)
- Komplexe Speicherorganisation
  - Caches, per core local memory, Shared Memory
- Homogene vs. heterogene Multicore Prozessoren

16.2.2 Embedded Computing vs. PC/Enterprise Computing

PC / Enterprise Computing:

- Meist homogene Multicores
  - Gesamtaufgabe aufteilen für optimale Ausführungszeit => Amdahl's Law
- Parallelisierung möglichst automatisiert (z.B. durch Compiler)

Embedded Computing:

- Oft heterogene Multicores
  - ARM Core für administrative Aufgaben
  - ein bis mehrere DSPs für effiziente mathematische Berechnungen
  - Aufteilung ergibt sich von selbst

16.3 Amdahl's Law

Amdahl's Law beschreibt den theoretisch möglichen Speedup S eines parallelen Tasks, wenn dieser auf mehrere Cores (Prozessoren) aufgeteilt wird.

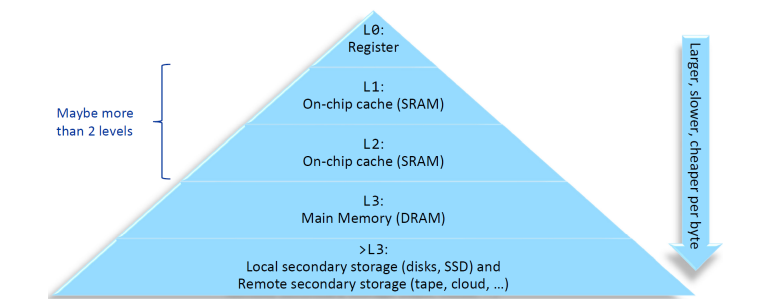
Hinweis: Typischerweise bleibt ein Teil des Tasks sequenziell. Dieser kann nicht aufgeteilt werden und limitiert somit den möglichen Speedup.

$$S(n) = \frac{1}{1 - p + \frac{p}{n}}$$
$$S \leq \frac{1}{1 - p}$$

$$S(n) = \frac{T}{t_s + \frac{t_p}{n}}$$
$$S \leq \frac{T}{T - t_p} = \frac{T}{t_s}$$

S	Speedup (Faktor, z.B. 4.0)
p	Parallel Portion (z.B. 0.6 = 60 %)
n	Anzahl Cores
T	Totale Ausführungszeit
t <sub>p</sub>	Parallele Ausführungszeit
t <sub>s</sub>	Serielle Ausführungszeit

16.4 Memory Hierarchy



16.5 Cache-Speicher

- Für Prozessor versteckte, schnelle Speicher => liegen zw. Prozessor und Hauptspeicher
- Führen Kopie von häufig benötigten Hauptspeicherdaten
  - Nutzen zeitliche und örtliche Lokalität von Programmen aus
- Cache Hit: Vom Prozessor benötigte Hauptspeicherdaten sind im Cache
  - Schnellerer Zugriff möglich
- Cache Miss: Daten müssen aus Hauptspeicher geholt werden

Mittlere Zugriffszeit = (Hit Time) · (Hit Rate) + (Miss Penalty) · (Miss Rate)

(Hit Time)

=

(Zeit zur Bestimmung von Hit oder Miss) +  
(Speicherzugriffszeit auf Cache)

(Miss Penalty)

=

(Zeit zur Bestimmung von Hit oder Miss) +  
(Zeit zum Ansprechen der nächsten Ebene) +  
(Zeit zum Übertragen von der nächsten Ebene)

16.5.1 Zeitliche / örtliche Lokalität

Zeitliche Lokalität (Temporal locality):

- Soeben verwendete Daten / Instruktionen werden mit hoher Wahrscheinlichkeit bald wieder verwendet
  - Caches nutzen zeitliche Lokalität aus

Örtliche Lokalität (Spatial locality):

- Bei soeben verwendete Daten / Instruktionen werden mit hoher Wahrscheinlichkeit auch benachbarte Daten / Instruktionen verwendet
  - Caches: Wenn auf eine bestimmte Adresse das erste Mal zugegriffen wird, werden nicht nur die Daten von dieser Adresse ins Cache geladen, sondern ein Speicherblock bestimmter Grösse um diese Adresse herum

=> Cache ist reine Wahrscheinlichkeit! => Hoffen, dass Daten dort sind...



16.6 Cache Ersetzungsstrategien

Es gibt mehrere Strategien, um zu ermitteln, welche Daten aus dem Cache entfernt werden sollen, um Platz für neue Daten zu schaffen. Dabei ist zu beachten, dass bei jeder Eretzung allenfalls Daten in die **nächste Stufe zurückgeschrieben** werden müssen. Dies ist immer dann der Fall, wenn das **Dirty Bit** (Signal, dass Cache-Daten geändert haben) gesetzt ist.

- **LRU (Least Recently Used)** => häufige Strategie
  - Bei Zugriff muss zusätzlich ein Timestamp gespeichert werden
- **LFU (Least Frequently Used)**
  - Bei Zugriff muss Counter gespeichert werden (wie oft Daten gebraucht wurden)
- **FIFO (First In First Out)**
  - doof, wenn man Daten oft braucht, diese aber schon lange im Cache liegen
- **Random**

16.7 Caches in Echtzeitsystemen

- In (harten) Echtzeitsystemen ist **Worst Case Execution Time (WCET)** wichtig
    - **WCET mit Cache immer schlechter** als ohne Cache!
      - => wegen Bestimmung ob Hit oder Miss
  - Wegen steigender WCET sind Caches in Echtzeitsystemen **allenfalls ungeeignet**
- => WCET tritt so selten ein, dass Caches trotzdem die schnellste Option sein sollten (wegen gesenkter **mittlerer Zugriffszeit**)

16.8 Cache Coherence

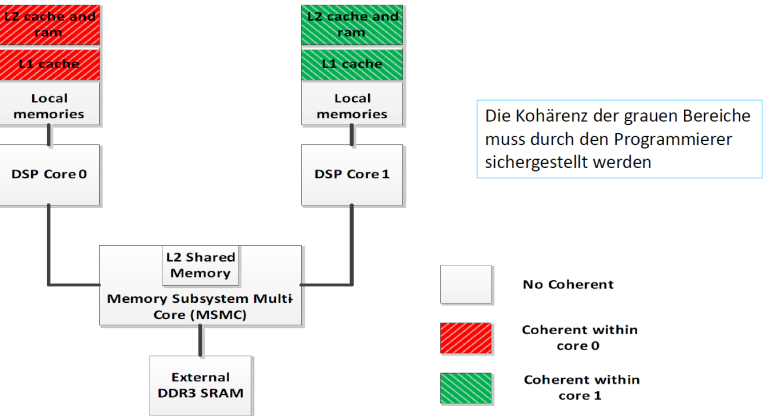
Bei Multtcore-Systemen können **Kopie derselben Daten in mehreren Cashes** liegen. Cache Coherence beschreibt, wie ein Cache eines Cores **erfährt**, dass ein anderer Core diese **Daten geändert** hat. Cache Coherence kann mit verschiedenen Protokollen oder auch mit Hardware-Unterstützung umgesetzt werden.

16.8.1 MESI Cache Coherence Protocol

Jede 'cache line' hat einen der folgenden Zustände:

- **Modified (M)**
    - Lokale Kopie modifiziert, keine Kopien in anderen Caches
    - Memory is 'stale' (dirty)
  - **Exclusive (E)**
    - Keine Kopien in anderen Caches
    - Memory up to date
- **Shared (S)**
    - Unmodifizierte Kopien in anderen Cashes möglich
    - Memory up to date
  - **Invalid (I)**
    - Cache line nicht benutzt

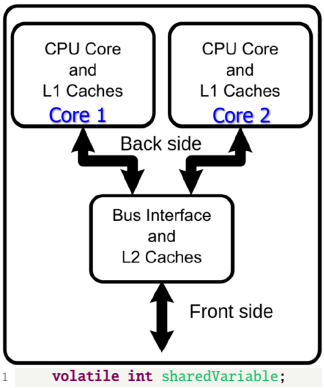
16.8.2 Cache Coherence mittels HW-Unterstützung



16.9 Programmierung von Shared Memory in C und C++

- Beide Cores haben **eigenen** Adressbereich
- **sharedVariable** liegt auf **fester Adresse in Shared Memory** und wird von beiden Cores genutzt
- Zugriff auf **sharedVariable** muss **synchronisiert** werden
- Zwischen Cores und **sharedVariable** können mehrere Caches liegen
- **Jeder Core muss sharedVariable für sich definieren => volatile**

**volatile** hat Einfluss zur Compile-time und 'verbietet' dem Compiler das Anlegen einer Kopie in einem Arbeitsregister



16.9.1 volatile Variablen

- **Alle Variablen, die ausserhalb des Programmkontextes des Prozessors/Threads geändert werden können müssen volatile sein!**
  - Variablen, die (speziell bei Embedded Systems) ein Hardwareregister darstellen
  - Globale Variablen, auf die in mehreren Threads zugegriffen wird (concurrency)
  - Globale Variablen, die in einer ISR geändert werden
  - Shared Variablen, die von einem anderen Prozessor/Core geändert werden können

16.10 Speicherzugriff zur Laufzeit

- Dank **volatile** besteht generierter Maschinencode aus **explizitem Speicherzugriff**
  - soll so schnell wie möglich sein

- Hoffnung: Daten sind in prozessornahem Cache
- Geschwindigkeitsgewinn erst ab zweitem Zugriff (erst dann sind Daten im Cache)
  - => **Daten, auf die nur einmal zugegriffen wird, sollen nicht gecached werden!**
- **volatile** löst Cache Coherence Problem nicht
  - **volatile ist notwendig, aber nicht hinreichend!**

16.11 Datenkonsistenz

Oft tritt folgendes Szenario ein:

- System empfängt Daten (z.B. ein Bild) und schreibt sie in Buffer
- Daten werden gefiltert (z.B. Image Processing) und wieder ausgegeben

Es stellt sich die Frage: Wie kann **verhindert werden**, dass der Filterschritt **inkonsistente Daten im Buffer** hat, weil der Empfangsteil laufend Daten in den Buffer schreibt?

**Schlechte Lösung:**

- Buffer wird synchronisiert (z.B. mit Semaphoren)
- Overhead, strenge Serialisierung

**Triviale Lösung:**

- Empfangsbuffer wird in zweiten Buffer kopiert sobald Daten fertig empfangen sind
- Kopieren ist teuer

**Vernünftige Lösung für Datenkonsistenz – Ping Pong Buffer:**

- Zwei identische Buffer definieren
- Ein Core (Master) bestimmt, in welchen Buffer die Daten geschrieben werden
- Nach Empfangen der Daten übergibt Master dem Filterschritt einen **Pointer** auf 'Daten-Buffer'
- Master übergibt Sender der Daten **Pointer auf zweiten Buffer**, damit Daten nun dort geschrieben werden
  - => Ping Pong zwischen den zwei Buffern

Noch besser ist es, wenn der Empfang der Daten über DMA (Direct Memory Access) erfolgt.

=> DMA-Einheit braucht nur Ziel-Pointer und Byte Counter

17 Real-Time Operating Systems (RTOS)

17.1 Operating System (OS) / Betriebssystem

Ein Betriebssystem bildet die **Schnittstelle** zwischen den HW-Komponenten und der Anwendungssoftware des Benutzers. Seine Aufgaben sind insbesondere:

- Benutzerkommunikation
  - Laden, Ausführen, Unterbrechen und Beenden von Programmen
  - **Verwaltung und Zuteilung der Prozessorzeit (Scheduling)**
- Speicherverwaltung
  - Verwaltung und Betrieb der angeschlossenen Geräte
  - Schutzfunktionen

17.2 RTOS / Echtzeitbetriebssystem

Ein RTOS ist ein Betriebssystem, das **Echtzeitanforderungen** erfüllen kann.

- Konzepte der POSIX-Programmierung finden hier Anwendung
- Das **Prinzip** aller RTOS ist gleich
  - API und Umfang der RTOS sind unterschiedlich

**Beispiele für RTOS:**

- FreeRTOS von Amazon
  - Zephyr von Linux Foundation
  - VxWorks von Wind River Systems
  - TI-RTOS von Texas Instruments
- µC/OS-III von Micrium
  - QNX von Blackberry
  - Windows 10 IoT Enterprise von Microsoft (nur soft real-time!)

17.3 FreeRTOS vs. Zephyr

Kriterium	FreeRTOS	Zephyr
<b>Leichtgewichtigkeit</b>	Sehr leichtgewichtig	Ressourceninversiver
<b>Feature-Set</b>	Grundlegende RTOS-Funktionalitäten	Umfangreicher (Treiber, Netzwerk, FS)
<b>Hardware-Support</b>	Sehr breit	Begrenzter, aber wachsender Support
<b>Lernkurve</b>	Einfach und schnell	Anspruchsvoller, aber leistungsfähiger
<b>Einsatzgebiete</b>	Kleinere, spezifische Projekte	Skalierbare & komplexere Anwendungen
<b>Lizenz</b>	MIT	Apache 2.0

17.3.1 Fazit der Gegenüberstellung

=> Welches RTOS geeigneter ist, **kommt auf die Anwendung an!**

Grundsätzlich gilt aber Folgendes:

- **FreeRTOS** für Systeme mit begrenzten Ressourcen (**Flash < 32 kB, RAM < 4 kB** ideal)
  - Bleibt performant für CPUs mit Taktfrequenz **< 100MHz**
    - Leichtgewichtiges RTOS für einfache / ressourcenbeschränkte Systeme
- **Zephyr** wenn zusätzliche Funktionalität für komplexere Anwendungen unverzichtbar Bedarf an leistungsfähiger MCU (**Flash ≥ 64 kB, RAM ≥ 8 kB**)
  - Modernes, skalierbares, funktionsreiches RTOS für komplexere Anwendungen und Projekte mit Netzwerk und Sicherheitsanforderungen

17.3.2 Beispiele für typische Anwendungen

System	FreeRTOS	Zephyr
<b>Sensorsteuerung</b>	gut umsetzbar	gut umsetzbar
<b>IoT-Gerät mit WLAN</b>	gut umsetzbar	gut umsetzbar
<b>Komplexeres IoT-Gerät (Bluetooth + FS)</b>	schwierig umzusetzen	gut umsetzbar
<b>Embedded Linux-Alternative (RTOS)</b>	nicht geeignet	gut umsetzbar