Embedded Software Engineering 1

HS 2024 – Prof. Reto Bonderer

Autoren: Laurin Heitzer, Simone Stitz

https://github.com/P4ntomime/EmbSW1

V1.0.20241012

Inhaltsverzeichnis

1 Embedded Systems – Allgemein

1.1 Definition

Ein Embedded System...
- ist ein System, das einen Computer beinhaltet, selbst aber kein Computer ist
- besteht üblicherweise aus Hardware (Mechanik, Elektronik) und Software
- ist sehr häufig ein Control System (Steuerung, Regelung)

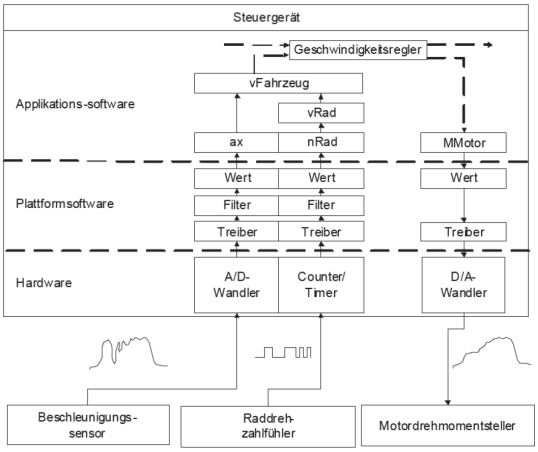
Ein Embedded System beinhaltet typischerweise folgende Komponenten:
- Sensoren
- Aktoren
- Mikrocomputer
- Software (Firmware)
- Hardware (Mechanik, Elektronik)

1.1.1 Charakterisierung von Embedded Systems

Embedded Systems können **(müssen aber nicht)** folgende Eigenschaften haben:
- **reactive systems:** Reaktive Systeme interagieren mit ihrer Umgebung
- **real-time systems:** Echtzeitsysteme haben nebst funktionale Anforderungen auch definierbaren zeilichen Anforderungen zu genügen
- **dependable systems:** Verlässliche Systeme sind Systeme, welche (sehr) hohe Zuverlässigkeitsanderungen erfüllen müssen
- **Weitere (häufige) Anforderungen:**
  - kleiner Energieverbrauch
  - kleine physikalische Abmessungen
  - Lärm, Vibration, etc.

1.1.2 Typischer Aufbau

Ein gutes Design beinhaltet unterschiedliche Abstraktionsschichten ➔ Layer
➔ Siehe Abschnitt 1.8



1.2 Beispiele

Fahrrad-Computer
- GPS-Navigation
- Geschwindigkeits- und Trittfrequenzmessung
- Pulsmesser
- Drahtlosübertragung (ANT+)
- Interface zu elektronischer Gangschaltung
- Barometer, Thermometer
- Trainingsassistent
- Display

Weitere Beispiele
- Smartphone
- Mobile Base Station
- CNC-Bearbeitungszentrum
- Hörgerät

Auto
- Sicherheitsrelevante Aufgaben
  - ABS, ASR
  - Motorenregelung
  - Drive-by-wire
  - Autonom fahrende Autos
- Unterhaltung / Komfort
  - Radio / CD / etc.
  - Navigation
  - Klima
- Mehrere Netzwerke
  - CAN, LIN, Ethernet
- Echtzeitteile und andere
- Von einfachsten μCs bis DSPs und GPUs

➔ Auto ist ein riesiges Embedded System

1.3 Deeply Embedded System

- 'Einfaches' Embedded System, mit **minimaler Benutzerschnittstelle**, üblicherweise mit **keinerlei GUI** und **ohne Betriebssystem**
- Beschränkt auf **eine** Aufgabe (z.B. Regelung eines physikalischen Prozesses)
- Muss oft zeitliche Bedingungen erfüllen ➔ Echtzeitsystem

1.3.1 Beispiele – Deeply Embedded System

- Hörgerät
- Motorenregelung
- ABS-Controller
- 'Sensor' im IoT
- etc...

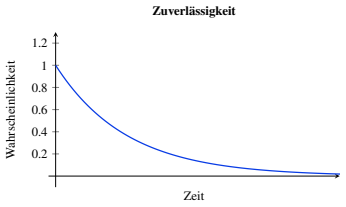
1.4 Betriebssysteme bei Embedded Systems

- Es kommen Betriebssysteme wie (Embedded) Linux oder Android zum Einsatz
  ➔ **Achtung: Linux und Android sind nicht echtzeitfähig!**
- Wenn Echtzeit verlangt wird: real-time operating systems (RTOS)
  - Beispiele: Zephyr, Free RTOS (Amazon), TI-RTOS (Texas Instuments), etc.

1.5 Bare Metal Embedded System

- Es kommt **keinerlei Betriebssystem** zum Einsatz
- Bare Metal Embedded Systems sind recht **häufig**, insbesondere bei **Deeply Embedded Systems**
- Bare Metal Embedded Systems stellen besondere Ansprüche an Programmierung

1.6 Zuverlässigkeit



- Je länger das System läuft, desto weniger zuverlässig ist es
- Die Wahrscheinlichkeit für einen Ausfall steigt stetig

Achtung: Hier ist nur die Alterung der Hardware berücksichtigt

1.7 Verfügbarkeit

Die Verfügbarkeit A (Availability) ist der Anteil der Betriebsdauer innerhalb dessen das System seine Funktion erfüllt.

$$\text{Verfügbarkeit} = \frac{\text{Gesamtzeit} - \text{Ausfallzeit}}{\text{Gesamtzeit}}$$

1.8 Abstraktionsschichten

- Bei µC-Programmierung (Firmware) müssen oft Bitmuster in Register geschrieben werden
- Solche Register-Zugriffe dürfen **nicht** 'willkürlich' überall im Code erfolgen
  ➔ schlecht lesbar, schlecht portiertbar, fehleranfällig
- **Damit Code lesbarer und besser auf andere Platform portierbar wird, beinhaltet jeder professionelle Code einen Hardware Abstraction Layer (HAL)**
- HAL führt **nicht** zum Verlust bei Laufzeit, wenn korrekt implementiert

1.8.1 Hardware-abstraction-layer (HAL)

- Trennt HW-Implementierung von SW-Logik
- Gleiche SW kann auf verschiedene HW verwendet werden ➔ Portabilität
- HW-Komponenten können einfach ausgetauscht werden ➔ Flexibilität

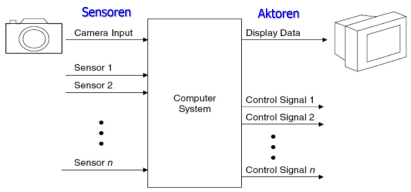
2 Real-Time System (Echtzeitsystem)

2.1 Definitionen

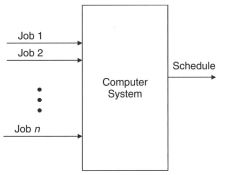
2.1.1 Real-Time System (Echtzeitsystem)

- 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.

Typisches Echtzeitsystem

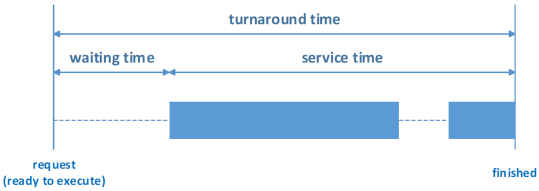


Repräsentation RT-System



Sequenz von Aufgaben (Jobs) müssen zeitlich geplant (scheduled) werden

2.1.2 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

2.2 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.**

2.3 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

2.3.1 Beispiele verschiedener Echtzeitsysteme

System	Klassifizierung	Erlärung
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.

2.4 Determinsismus (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

2.5 Auslastung (utilization)

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

2.5.1 Berechungen zur Auslastung (utilzation)

Annahmen:
- System mit $n \geq 1$ periodischen Tasks T_i und Periode p_1
- Jeder Task T_i het 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

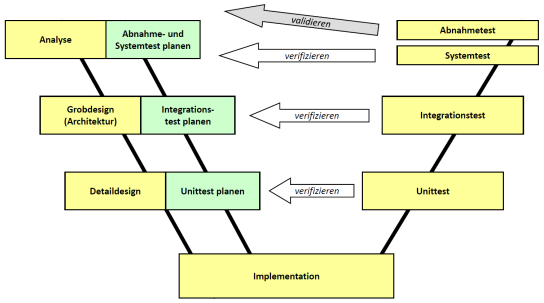
➔ Bei 69 % Auslastung ist das **'theoretical limit'**

2.6 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.

3 Modellierung eines Embedded Systems

3.1 V-Modell für Software-Entwicklungszyklus



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

3.2 Model Driven Development (MDD)

- Bei **modellbasierter Entwicklung** kommen in **allen Entwicklungsphasen** durchgängig Modelle zum 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 dasslbe beschrieben (Code und Diagramm) ➔ **unbedingt zu vermeiden!**

3.3 Vorgehen bei der Modellierung

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

Stukturmodellierung (Statische Aspekte)

Modellierung der dynamischen Aspekte

3.4 Systemgrenze definieren & Systemprozesse finden

3.4.1 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, periheral system)?

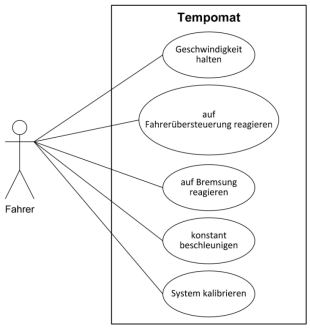
3.4.2 Systemprozesse finden (use-cases)

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:

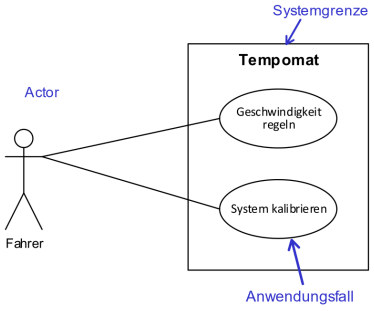
- Aussenbetrachtung des Systems (**oberflächlich!**)
  - Nicht komplizierter als nötig
- System als Blackbox betrachten
  - **Was** soll System können; (nicht: wie soll das System etwas machen)
- RTE-Systeme bestehen häufig aus nur einem einzigen Systemprozess
  - speziell wenn System 'nur' ein Regler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

3.4.3 Kontextdiagramm: Use-Case Diagramm

Tempomat: zu detailliert



Tempomat: verbesserte Version



3.4.4 Kontextdiagramm: Sequenzdiagramm

- Speziell bei Syetemen, deren Grenzen durch **Nachrichtenflüsse** charakterisiert werden können
- Details zu Sequenzdiagrammen siehe Abschnitt 3.6.1

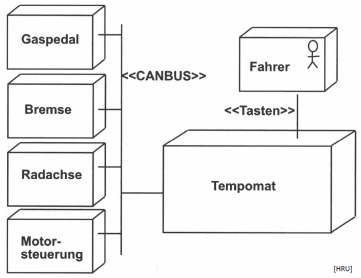
3.5 Verteilungen festlegen

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

3.5.1 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

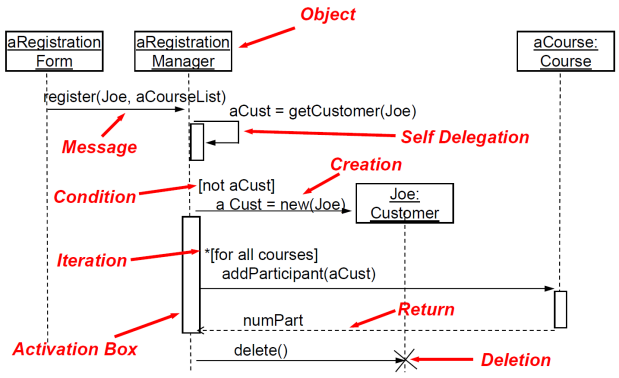


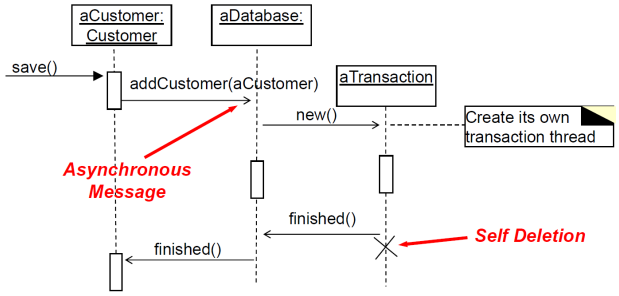
3.6 Systemprozesse detaillieren

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

3.6.1 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



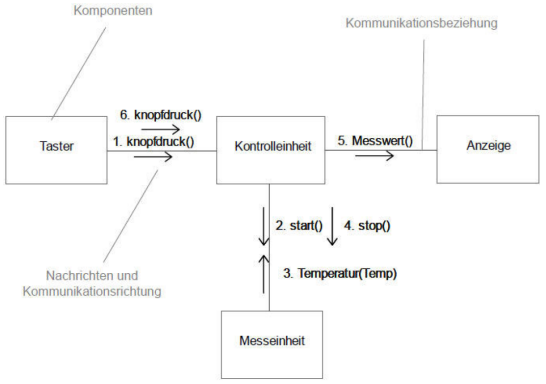


Pfeiltyp	Semantik
→	Synchrone Aufrufe
⇢	Asynchrone Nachrichten
- - - →	Datenfluss

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

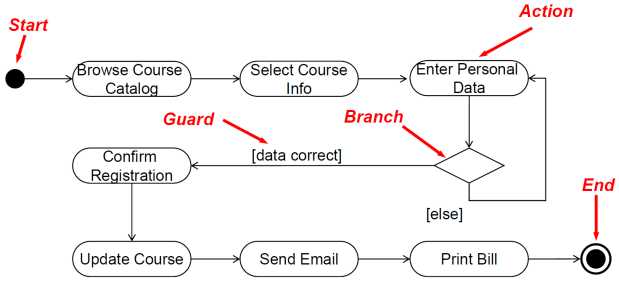
3.6.2 Kommunikationsdiagramm (Kollaborationsdiagramm)

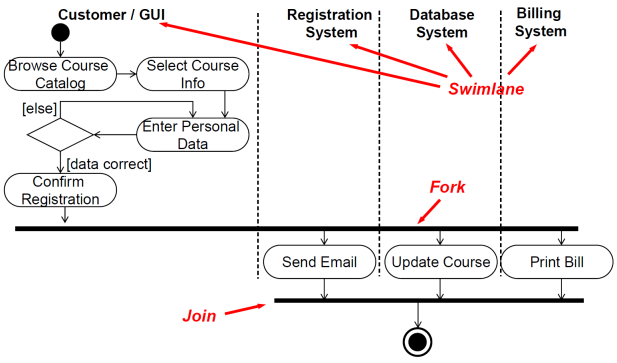
- Kommunikationsdiagramm zeigt **dieselbe Information** wie Sequenzdiagramm
- **Schwerpunkt: Informationsfluss** zwischen den Objekten
  ➡ Beim Sequenzdiagramm liegt der Schwerpunkt auf dem zeitlichen Ablauf



3.6.3 Aktivitätsdigramm

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





4 Hardware-Software-Codesign

4.1 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 Spezifikationssprache**, 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

4.2 Anforderungen für praktische Anwendungen

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

4.3 Spezifikationssprachen

- **Formale Sprachen sind eindeutig** (Prosa immer mehrdeutig)
- Spezifikation kann compiliert und ausgeführt werden ➡ Simulationen
- Die ausführbare Spezifikation dient als **Golden Reference** für die künftigen Entwicklungsschritte

Beispiele für Spezifikationssprachen

- SystemC (eine C++-Template Library)
- SysML
- SpecC
- SystemVerilog
- Esterel
- Matlab/Simulink
- Statecharts

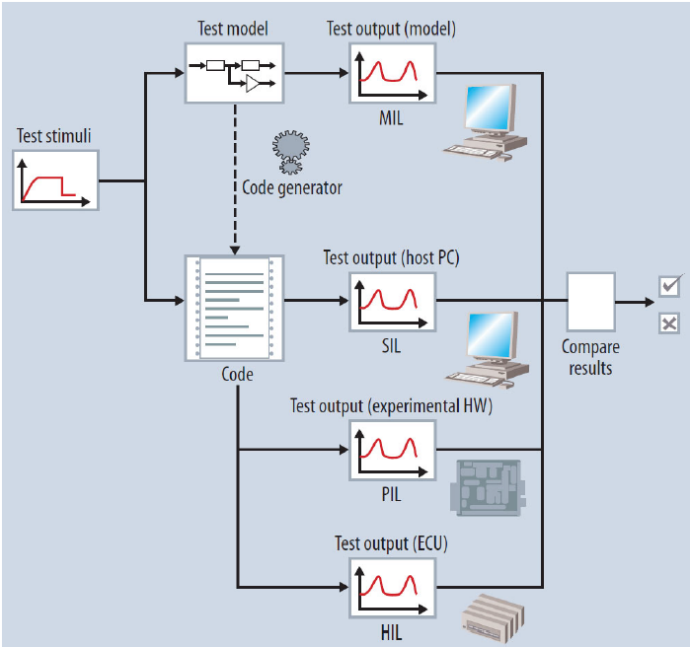
4.4 Virtuelle Prototypen

- 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

4.5 X-in-the-loop

- **Model-in-the-Loop (MIL):** vollständig als Modell vorliegender virtuellen 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)

➡ Test outputs werden jeweils mit **Golden Reference** verglichen



4.6 Entwicklungsplattformen

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

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

5 Zustandsbasierte Systeme

5.1 Asynchrone vs. synchrone FSM

- **Asynchron**
  - geänderte Inputsignale führen **direkt** zur Zustandsänderung
  - schneller, aber enorm anfällig auf Glitches
- **Synchron**
  - Inputsignale werden nur zu diskreten Zeitpunkten betrachtet
    ➔ getaktete Systeme
- Softwareimplementationen sind eigentlich immer **synchron**, da Rechner getaktet sind
- Rein softwareseitig besteht die Problematik der Asynchronizität nicht

5.2 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

5.2.1 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

5.2.2 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

➔ **Wenn immer möglich sollten Moore-Automaten verwendet werden**

5.2.3 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

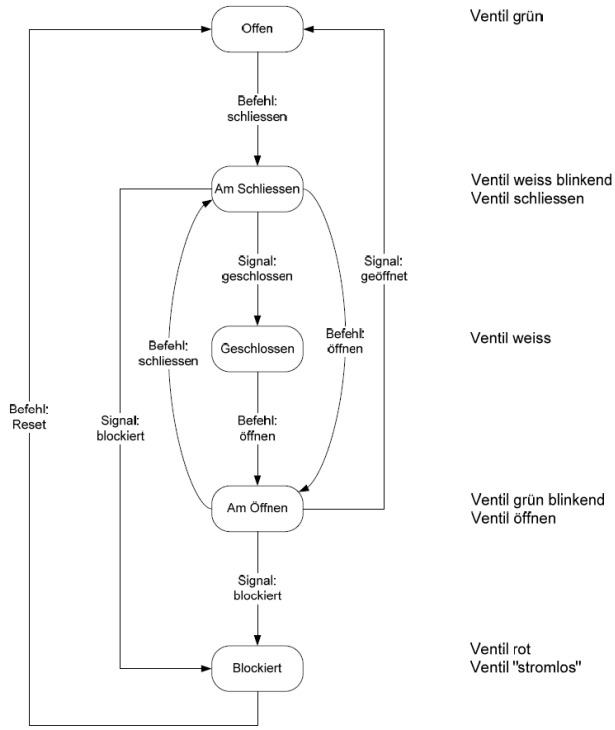
➔ Wird hier nicht weiter behandelt...

5.3 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
  ➔ Bei Modellierung sind Zwischenzustäde vorgehesen, z.B. 'closing', starting up'

Beispiel: State-Event-Diagramm – Moore Automat



5.4 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

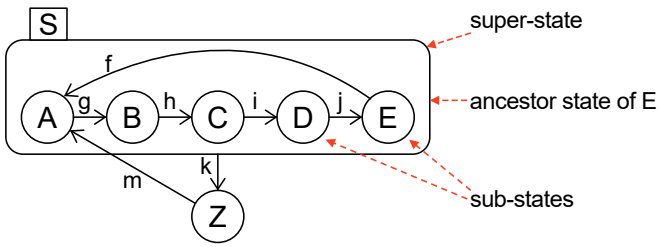
6 Statecharts (nach Marwedel)

6.1 Nachteile von State-Event-Diagrammen

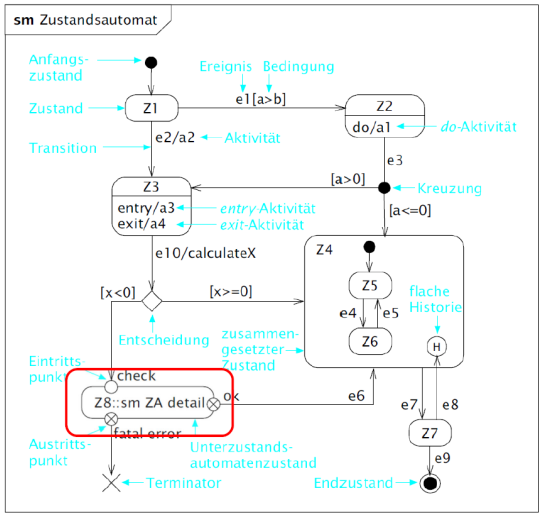
- Zustandsdiagramme sind flach (es gibt keine Hierarchie) ➔ schnell unübersichtlicht
- Es kann keine zeitliche Parallelität modelliert werden

6.2 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 ➔ 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 ➔ Parallelität
➔ Werden auch **Teilautomaten** genannt



6.2.1 Elemente der Statecharts



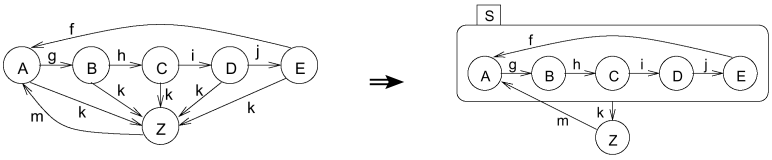
6.2.2 Allgemeine Syntax für Transitions-Pfeile

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

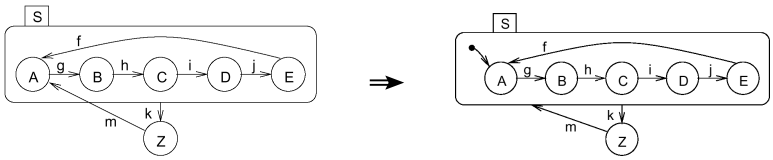
6.3 Hierarchie (OR-super-states)

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



6.4 Default-State

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



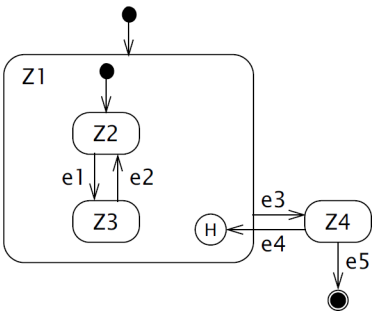
6.5 History

- 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

6.5.1 Shallow History

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

Beispiel: Shallow-History

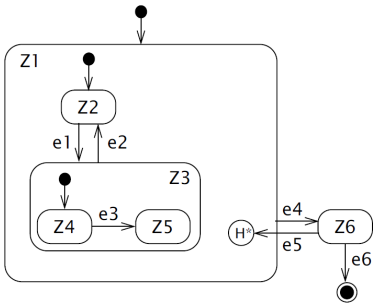


- In Zustand $Z3$ bewirkt der Event $e3$ einen Übergang zu $Z4$
- Der Event $e4$ führt nu zu einem Übergang zum früheren Zustand $Z3$

6.5.2 Deep History

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

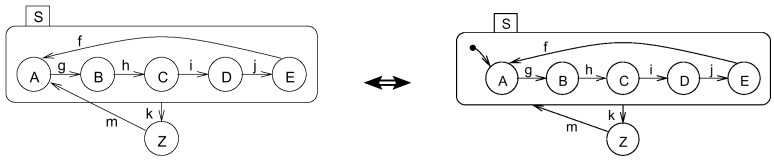
Beispiel: Deep-History



- 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

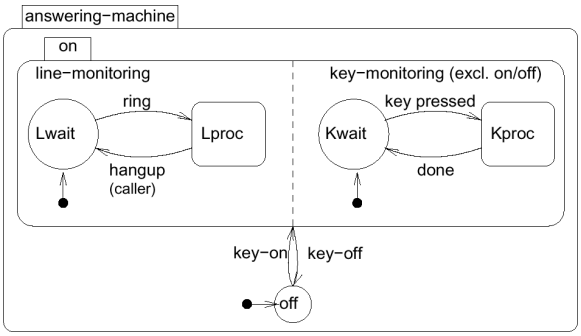
6.6 Kombination: History- und Default-Mechanismus

Folgende statecharts bilden genau das Gleiche ab



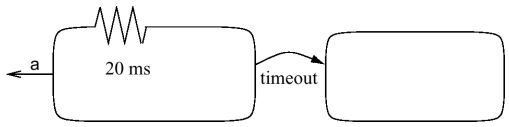
6.7 Parallelität (AND-super-state, Teilautomaten)

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

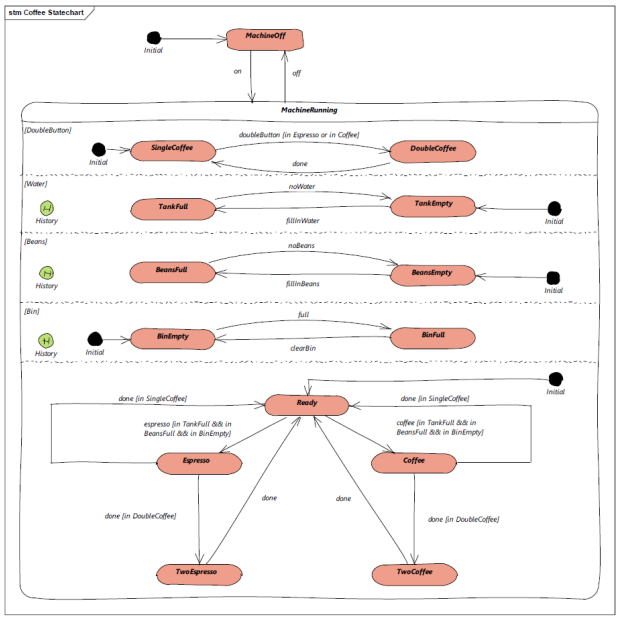


6.8 Timers

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



6.9 Beispiel – Armbanduhr als Statechart



7 Realisierung flache FSM

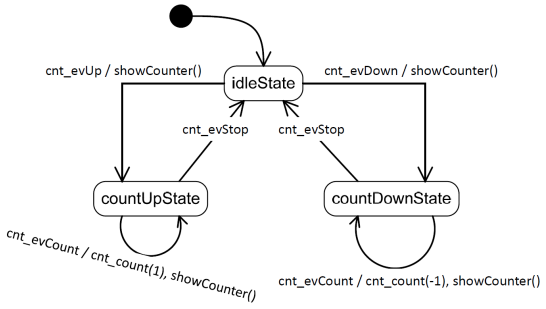
7.1 Mögliche Realisierungen von flachen FSMs

- 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++)

⇒ Alle Varianten haben wie immer sowohl Vor- als auch Nachteile
⇒ Bei allen Varianten sind auch Variationen vorhanden

7.2 Realisieurng mit Steuerkonstrukt (prozedural in C)

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



7.2.2 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 Varianlen** gehalten
- Die FSM wird in **zwei Funktionen** implementiert
  - Initialiserungs-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

7.2.3 Implementation 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 mitgegenen 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: UpD/Down-Counter (prozedural in C)

```c
// counterCtrl.h
// implements the Finite State Machine (FSM) of an up/down-Counter

#ifndef COUNTERCTRL_H__
#define COUNTERCTRL_H__

typedef enum {cnt_evUp,       // count upwards
              cnt_evDown,     // count downwards
              cnt_evCount,    // count (up or down)
              cnt_evStop}     // stop counting
             cnt_Event;

void cnt_ctrlInit(int initValue);
// initializes counter FSM

void cnt_ctrlProcess(cnt_Event e);
// changes the state of the FSM based on the event 'e'
// starts the actions

#endif
```

```c
// counterCtrl.c
// implements the Finite State Machine (FSM) of an up/down-Counter

#include <stdio.h>
#include "counterCtrl.h"
#include "counter.h"

typedef enum {idleState,        // idle state
              countUpState,     // counting up at each count event
              countDownState}   // counting down at each count event
             State;

static State currentState = idleState; // holds the current state of the FSM

void cnt_ctrlInit(int initValue)
{
  currentState = idleState;   // set init-state
  cnt_init(initValue);        // set initValue
}

void cnt_ctrlProcess(cnt_Event e)
{
  switch (currentState)
  {
    case idleState:
      printf("State: idleState\n");
      if (cnt_evUp == e)
      {
        // actions (and exit-actions von idleState)
        printf("State: idleState, counter = %d\n", cnt_getCounter());
        // state transition (andentry-actions von countUpState)
        printf("Changing to State: countUpState\n");
        currentState = countUpState;
      }
      else if (cnt_evDown == e)
      {
        // actions (and exit-actions von idleState)
        printf("State: idleState, counter = %d\n", cnt_getCounter());
        // state transition (and entry-actions von countDownState)
        printf("Changing to State: countDownState\n");
        currentState = countDownState;
      }
      break;

    case countUpState:
      printf("State: countUpState\n");
      if (cnt_evCount == e)
      {
        // actions
        cnt_count(1);
        printf("State: countUpState, counter = %d\n", cnt_getCounter());
        // state transition
      }
      else if (cnt_evStop == e)
      {
        // actions
        // state transition
        printf("Changing to State: idleState\n");
        currentState = idleState;
      }
      break;

    case countUpState:
      // ...
      break;

    default:
      break;
  }
}
```

```c
// counterTest.c
// Test program for the Finite State Machine (FSM) of an up/down-Counter

#include <stdio.h>
#include "counterCtrl.h"

int main(void)
{
  char answer;
  cnt_ctrlInit(0);

  do
  {
    printf("\n-----------------------------------------\n");
    printf("    u   Count up\n");
    printf("    d   Count down\n");
    printf("    c   Count\n");
    printf("    s   Stop counting\n");
    printf("    q   Quit\n");

    printf("\nPlease press key: ");
    scanf("%c", &answer);
    getchar();  // nach scanf() ist noch ein '\n' im Inputbuffer: auslesen und wegwerfen
    printf("\n");

    switch (answer)
    {
      case 'u':
        cnt_ctrlProcess(cnt_evUp);
        break;
      case 'd':
        cnt_ctrlProcess(cnt_evDown);
        break;
      case 'c':
        cnt_ctrlProcess(cnt_evCount);
        break;
      case 's':
        cnt_ctrlProcess(cnt_evStop);
        break;
      default:
        break;
    }
  } while (answer != 'q');

  return 0;
}
```

7.3 Realisieurng mit Steuerkonstrukt (objektorientiert in C++)

Anstossen der FSm: